oemof.tabular

Release 0.0.3

Jan 26, 2023

Contents

1 Overview **1**
- 1.1 Installation . . . 1
- 1.2 Documentation . . . 1
- 1.3 Development . . . 1

2 Installation **3**

3 Usage **5**
- 3.1 Background . . . 5
- 3.2 Datapackage . . . 6
- 3.3 Foreign Keys . . . 10
- 3.4 Scripting . . . 10
- 3.5 Reproducible Workflows . . . 11
- 3.6 Debugging . . . 12

4 API Reference **15**
- 4.1 oemof.tabular package . . . 15

5 Tutorials **35**
- 5.1 Using the datapackage-reader . . . 35
- 5.2 Using oemof.tabular.facades . . . 38
- 5.3 Creating energy systems from spreadsheet . . . 47

6 Contributing **57**
- 6.1 Bug reports . . . 57
- 6.2 Documentation improvements . . . 57
- 6.3 Feature requests and feedback . . . 57
- 6.4 Development . . . 58

7 Authors **59**

8 Changelog **61**
- 8.1 0.0.3 (2022-01-26) . . . 61
- 8.2 0.0.2 (2019-07-08) . . . 62
- 8.3 0.0.1 (2018-12-12) . . . 62
- 8.4 0.0.0 (2018-11-23) . . . 62

9 Indices and tables **63**

Python Module Index **65**

Index **67**

CHAPTER 1

Overview

Load oemof energy systems from tabular data sources.

- Free software: BSD 3-Clause License

1.1 Installation

Simpy run:

```
pip install oemof.tabular
```

1.2 Documentation

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

1.3 Development

To run the all tests run:

```
pytest
```

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

The underlying concept of **oemof-tabular** is the oemof solph package. The Open Energy Modelling Framework (oemof) is based on a graph structure at its core. In addition it provides an optimization model generator to construct individual dispatch and investment 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.solph functionality. More importantly theses classes provide an interface to tabular data sources from that models can be created easily.

Note: To see the implemented facades check out the *facades* module.

3.1.1 Facades

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. The documentation for the facades

can be found *facades*. In addition you can check out the jupyter notebook from the tutorials and the examples directory.

Currently we provide the following facades:

- *Dispatchable*
- *Volatile*
- *Storage*
- *Reservoir*
- *BackpressureTurbine*
- *ExtractionTurbine*
- *Commodity*
- *Conversion*
- *Load*.
- *Link*
- *Excess*

These can be mixed with all oemof solph classes if your are scripting.

3.1.2 Datamodel and Naming Conventions

Facades require specific attributes. For all facades the attribute *carrier*, 'tech' and 'type' need to be set. The type of the attribute is string, therefore you can choose string for these. However, if you want to leverage full postprocessing functionality we recommend using one of the types listed below

Carriers

- solar, wind, biomass, coal, lignite, uranium, oil, gas, hydro, waste, electricity, heat, other

Tech types

- st, ocgt, ccgt, ce, pv, onshore, offshore, ror, rsv, phs, ext, bp, battery

We recommend use the following naming convention for your facade names *bus-carrier-tech-number*. For example: *DE-gas-ocgt-1*. This allows you to also take advantage of the color map from *facades* module.

```
from oemof.facades import TECH_COLOR_MAP, CARRIER_COLER_MAP

biomass_color = CARRIER_COLER_MAP["biomass"]
pv_color = TECH_COLOR_MAP["pv"]
```

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** to csv-files in the datapackage format.
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 `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",
```

(continues on next page)

(continued from previous page)

```
                                    "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              |
|------------|---------| ------|-------|------------------|------------------|
| 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",
```

(continues on next page)

(continued from previous page)

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

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",
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "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 get 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. This data can also be downloaded from an additional repository which adheres to FAIR principles, like zenodo. If you provide raw data, make sure the license is compatiple with other data in your repository. The *scenarios* directory allows you to specify different scenarios and describe them in a basic way via config files. The *toml* standard is

used by oemof-tabular, howerver you may also use *yaml*, *json*, etc.. The scripts inside the *scripts* directory will build input data for your scenarios from the *.toml* files and the raw-data. This data will be in the format that oemof-tabular datapackage reader can understand. In addition the script to compute the models and postprocess results are 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 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)

If you encounter this error message when reading a datapackage, you most likely provided *output_parameters* that are of type object for a tabular resource. However, there will be emtpy entries in the field of your *output_parameters*.

```
...
TypeError: type object argument after ** must be a mapping, not NoneType
```

Note: If your column / field in a tabular resource is of a specific type, make sure every entry in thies column has this type! For example numeric and empty entries in combination will yield string as a type and not numeric!

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 Solver and 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.

Also the following error might occure:

```
...
File "/home/admin/projects/oemof-tabular/venv/lib/python3.6/site-packages/
↪pyomo/repn/plugins/cpxlp.py", line 849, in _print_model_LP
% (_no_negative_zero(vardata_ub)))
TypeError: must be real number, not str
```

This message may indicate that fields in your datapackage that should be numeric are actually of type string. While pyomo seems sometimes still to be fine with this, solvers are not. Here also check your meta data types and the data. Most likely this happens if meta data is inferred from the data and fields with numeric values are left empty which will yield a string type for this field.

CHAPTER 4

API 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.**temporal_clustering**(*datapackage*, *n*, *path='/tmp'*, *how='daily'*)

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

Parameters

- **datapackage** (*string*) – String of meta data file datapackage.json
- **n** (*integer*) – Number of clusters
- **path** (*string*) – Path to directory where the aggregated datapackage is stored
- **how** (*string*) – How to cluster 'daily' or 'hourly'

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.**infer_metadata**(*package_name='default-name'*, *keep_resources=False*, *foreign_keys=None*, *path=None*)

Add basic meta data for a datapackage

Parameters

- **package_name** (*string*) – Name of the data package
- **keep_resources** (*boolean*) – Flag indicating of the resources meta data json-files should be kept after main datapackage.json is created. The resource 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*) – Absolute 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**(*config*, *directory='.'*)

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_build_config**(*file='build.toml'*)
Read config build file in toml format

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

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_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*, *periods=8760*, *freq='H'*)
Create pandas datetimeindex.

Parameters

- **year** (*string*) – Year of the index
- **periods** (*string*) – Number of periods, default: 8760
- **freq** (*string*) – Freq of the datetimeindex, default: 'H'

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

oemof.tabular.datapackage.building.**write_elements**(*filename*, *elements*, *directory='data/elements'*, *replace=False*, *overwrite=False*, *create_dir=True*)
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
- **overwrite** (*boolean*) – If set, existing elements will be overwritten
- **create_dir** (*boolean*) – Create the directory if not exists

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*, *create_dir=True*)
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
- **create_dir** (*boolean*) – Create the directory if not exists

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

oemof.tabular.datapackage.processing module

oemof.tabular.datapackage.processing.**clean**(*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
- **name (optional)** (*str*) – Name of datapackage
- **only_data** (*str*) – Name of directory to only copy subset of datapackage (for example only the 'data' directory)

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

The code in this module is partly based on third party code which has been licensed under GNU-GPL3. The following functions are copied and adapted from:

https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

- _shape2poly()
- simplify_poly()
- nuts()

oemof.tabular.tools.geometry.**Shapes2Shapes**(*orig*, *dest*, *normed=True*, *equalarea=False*, *prep_first=True*, ***kwargs*)

Notes

Copied from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**intersects**(*geom*, *labels*, *geometries*)

oemof.tabular.tools.geometry.**nuts**(*filepath=None*, *nuts=0*, *subset=None*, *tolerance=0.03*, *minarea=1.0*)

Reads shapefile with nuts regions and converts to polygons

Returns *OrderedDict* – Country keys as keys of dict and shapely polygons as corresponding values

Notes

Copied from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**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.tools.geometry.**reproject**(*geom*, *fr=<sphinx.ext.autodoc.importer._MockObject object>*, *to=<sphinx.ext.autodoc.importer._MockObject object>*)

Notes

Copied and adapted from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**simplify_poly**(*poly*, *tolerance*)

Notes

Copied from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**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.

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*.

Note The mathematical notation is as follows:

- Optimization variables (endogenous variables) are denoted by x
- Optimization parameters (exogenous variables) are denoted by c
- The set of timesteps T describes all timesteps of the optimization problem

SPDX-License-Identifier: BSD-3-Clause

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

Bases: oemof.solph.network.transformer.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
- **expandable** (*boolean*) – True, if capacity can be expanded within optimization. Default: False.
- **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.

Backpressure turbine power plants are modelled with a constant relation between heat and electrical output (power to heat coefficient).

$$x^{flow,carrier}(t) = \frac{x^{flow,electricity}(t) + x^{flow,heat}(t)}{c^{thermal\ efficiency}(t) + c^{electrical\ efficiency}(t)} \qquad \forall t \in T$$

$$\frac{x^{flow,electricity}(t)}{x_{flow,thermal}(t)} = \frac{c^{electrical\ efficiency}(t)}{c^{thermal\ efficiency}(t)} \qquad \forall t \in T$$

Ojective expression for operation includes marginal cost and/or carrier costs:

$$x^{opex} = \sum_t (x^{flow,out}(t) \cdot c^{marginal\_cost}(t) + x^{flow,carrier}(t) \cdot c^{carrier\_cost}(t))$$

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_elec_bus = solph.Bus('my_elec_bus')
>>> my_fuel_bus = solph.Bus('my_fuel_bus')
>>> my_heat_bus = solph.Bus('my_heat_bus')
>>> my_backpressure = BackpressureTurbine(
...     label='backpressure',
...     carrier='gas',
...     tech='bp',
...     fuel_bus=my_fuel_bus,
...     heat_bus=my_heat_bus,
...     electricity_bus=my_elec_bus,
...     capacity_cost=50,
...     carrier_cost=0.6,
...     electric_efficiency=0.4,
...     thermal_efficiency=0.35)
```

build_solph_components()

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

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

Commodity element with one output for example a biomass commodity

Parameters

- **bus** (*oemof.solph.Bus*) – An oemof bus instance where the unit is connected to with its output
- **amount** (*numeric*) – Total available amount to be used within the complete timehorzion of the problem
- **marginal_cost** (*numeric*) – Marginal cost for one unit used commodity

- **output_parameters** (*dict (optional)*) – Parameters to set on the output edge of the component (see. oemof.solph Edge/Flow class for possible arguments)

$$\sum_t x^{flow}(t) \leq c^{amount}$$

For constraints set through *output_parameters* see oemof.solph.Flow class.

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_bus = solph.Bus('my_bus')
>>> my_commodity = Commodity(
...     label='biomass-commodity',
...     bus=my_bus,
...     carrier='biomass',
...     amount=1000,
...     marginal_cost=10,
...     output_parameters={
...         'max': [0.9, 0.5, 0.4]})
```

build_solph_components()

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

Bases: oemof.solph.network.transformer.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
- **carrier_cost** (*numeric*) – Carrier cost for one unit of used input. 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.
- **expandable** (*boolean or numeric (binary)*) – True, if capacity can be expanded within optimization. Default: False.
- **capacity_potential** (*numeric*) – Maximum invest capacity in unit of output capacity.
- **capacity_minimum** (*numeric*) – Minimum invest capacity in unit of output capacity.
- **input_parameters** (*dict (optional)*) – Set parameters on the input edge of the conversion unit (see oemof.solph for more information on possible parameters)
- **ouput_parameters** (*dict (optional)*) –

  Set parameters on the output edge of the conversion unit (see oemof.solph for more information on possible parameters)

$$x^{flow,from}(t) \cdot c^{efficiency}(t) = x^{flow,to}(t) \qquad \forall t \in T$$

Ojective expression for operation includes marginal cost and/or carrier costs:

$$x^{opex} = \sum_t (x^{flow,out}(t) \cdot c^{marginal\_cost}(t) + x^{flow,carrier}(t) \cdot c^{carrier\_cost}(t))$$

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_biomass_bus = solph.Bus('my_biomass_bus')
>>> my_heat_bus = solph.Bus('my_heat_bus')
>>> my_conversion = Conversion(
...     label='biomass_plant',
...     carrier='biomass',
...     tech='st',
...     from_bus=my_biomass_bus,
...     to_bus=my_heat_bus,
...     capacity=100,
...     efficiency=0.4)
```

build_solph_components()

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

Bases: oemof.solph.network.source.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 capacity yields the upper bound 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.
- **expandable** (*boolean*) – True, if capacity can be expanded within optimization. Default: False.
- **output_parameters** (*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 expanded
- **capacity_minimum** (*numeric*) – Minimum install capacity if capacity is to be expanded
- **The mathematical representations for this components are dependent on the**
- **user defined attributes. If the capacity is fixed before**

- **(**dispatch mode**) the following equation holds**
- **.. math::** – x^{flow}(t) leq c^{capacity} cdot c^{profile}(t) qquad forall t in T
- **Where :math:‘x^{flow}‘ denotes the production (endogenous variable)**
- **of the dispatchable object to the bus.**
- **If ‘expandable‘ is set to ‘True‘ (**investment mode**), the equation**
- **changes slightly**
- **.. math::** – x^{flow}(t) leq (x^{capacity} + c^{capacity}) cdot c^{profile}(t) qquad forall t in T
- **Where the bounded endogenous variable of the volatile component is added**
- **.. math::** – x^{capacity} leq c^{capacity_potential}
- ****Ojective expression** for operation**
- **.. math::** – x^{opex} = sum_t x^{flow}(t) cdot c^{marginal_cost}(t)
- **For constraints set through ‘output_parameters‘ see oemof.solph.Flow class.**

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_bus = solph.Bus('my_bus')
>>> my_dispatchable = Dispatchable(
...     label='ccgt',
...     bus=my_bus,
...     carrier='gas',
...     tech='ccgt',
...     capacity=1000,
...     marginal_cost=10,
...     output_parameters={
...         'min': 0.2})
```

build_solph_components()

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

Bases: oemof.solph.network.sink.Sink, *oemof.tabular.facades.Facade*

build_solph_components()

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

Bases: oemof.solph.components.extraction_turbine_chp.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.
- **expandable** (*boolean*) – True, if capacity can be expanded within optimization. Default: False.

The mathematical description is derived from the oemof base class ExtractionTurbineCHP :

$$x^{flow,carrier}(t) = \frac{x^{flow,electricity}(t) + x^{flow,heat}(t) \cdot c^{beta}(t)}{c^{condensing\_efficiency}(t)} \qquad \forall t \in T$$

$$x^{flow,electricity}(t) \geq x^{flow,thermal}(t) \cdot \frac{c^{electrical\_efficiency}(t)}{c^{thermal\_efficiency}(t)} \qquad \forall t \in T$$

where c^{beta} is defined as:

$$c^{beta}(t) = \frac{c^{condensing\_efficiency}(t) - c^{electrical\_efficiency(t)}}{c^{thermal\_efficiency}(t)} \qquad \forall t \in T$$

Ojective expression for operation includes marginal cost and/or carrier costs:

$$x^{opex} = \sum_t (x^{flow,out}(t) \cdot c^{marginal\_cost}(t) + x^{flow,carrier}(t) \cdot c^{carrier\_cost}(t))$$

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_elec_bus = solph.Bus('my_elec_bus')
>>> my_fuel_bus = solph.Bus('my_fuel_bus')
>>> my_heat_bus = solph.Bus('my_heat_bus')
>>> my_extraction = ExtractionTurbine(
...     label='extraction',
...     carrier='gas',
...     tech='ext',
...     electricity_bus=my_elec_bus,
...     heat_bus=my_heat_bus,
...     fuel_bus=my_fuel_bus,
...     capacity=1000,
...     condensing_efficiency=[0.5, 0.51, 0.55],
...     electric_efficiency=0.4,
...     thermal_efficiency=0.35)
```

build_solph_components()

class oemof.tabular.facades.**Facade**(**args*, ***kwargs*)
Bases: oemof.network.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.**HeatPump**(**args*, ***kwargs*)
Bases: oemof.solph.network.transformer.Transformer, *oemof.tabular.facades.Facade*

HeatPump unit with two inputs and one output.

Parameters

- **low_temperature_bus** (*oemof.solph.Bus*) – An oemof bus instance where unit is connected to with its low temperature input.
- **high_temperature_bus** (*oemof.solph.Bus*) – An oemof bus instance where the unit is connected to with its high temperature input.
- **capacity** (*numeric*) – The thermal capacity (high temperature output side) of the unit.
- **cop** (*numeric*) – Coefficienct of performance
- **carrier_cost** (*numeric*) – Carrier cost for one unit of used input. 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.
- **expandable** (*boolean or numeric (binary)*) – True, if capacity can be expanded within optimization. Default: False.
- **capacity_potential** (*numeric*) – Maximum invest capacity in unit of output capacity. Default: +inf.
- **low_temperature_parameters** (*dict (optional)*) – Set parameters on the input edge of the heat pump unit (see oemof.solph for more information on possible parameters)
- **high_temperature_parameters** (*dict (optional)*) – Set parameters on the output edge of the heat pump unit (see oemof.solph for more information on possible parameters)
- **input_parameters** (*dict (optional)*) –

  Set parameters on the input edge of the conversion unit (see oemof.solph for more information on possible parameters)

$$x^{flow}_{electricity\_bus,hp} = \frac{1}{c^{COP}} \cdot x^{flow}_{hp,high\_temperature\_bus}$$

$$x^{flow}_{low\_temperature\_source,low\_temperature\_bus} = x^{flow}_{hp,high\_temperature\_bus} \frac{c^{COP} - 1}{c^{COP}}$$

Ojective expression for operation includes marginal cost and/or carrier costs:

$$x^{opex} = \sum_t (x^{flow,out}(t) \cdot c^{marginal\_cost}(t) + x^{flow,carrier}(t) \cdot c^{carrier\_cost}(t))$$

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> electricity_bus = solph.Bus("elec-bus")
>>> heat_bus= solph.Bus('heat_bus')
>>> heat_bus_low = solph.Bus('heat_bus_low')
>>> fc.HeatPump(
...     label="hp-storage",
...     carrier="electricity",
...     tech="hp",
...     cop=3,
...     carrier_cost=15,
...     electricity_bus=elec_bus,
...     high_temperature_bus=heat_bus,
...     low_temperature_bus=heat_bus_low)
```

build_solph_components()

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

Bases: oemof.solph.custom.link.Link, *oemof.tabular.facades.Facade*

Bidirectional 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.
- **from_to_capacity** (*numeric*) – The maximal capacity (output side to bus) of the unit. If not set, attr *capacity_cost* needs to be set.
- **to_from_capacity** (*numeric*) – The maximal capacity (output side from bus) 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.
- **marginal_cost** (*numeric*) – Cost per unit Transport in each timestep. Default: 0
- **expandable** (*boolean*) – True, if capacity can be expanded within optimization. Default: False.

Note: Assigning a small value like 0.00001 to *marginal_cost* may force unique solution of optimization problem.

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_elec_bus_1 = solph.Bus('my_elec_bus_1')
>>> my_elec_bus_2 = solph.Bus('my_elec_bus_2')
```

(continues on next page)

(continued from previous page)

```
>>> my_loadink = Link(
...     label='link',
...     carrier='electricity',
...     from_bus=my_elec_bus_1,
...     to_bus=my_elec_bus_2,
...     from_to_capacity=100,
...     to_from_capacity=80,
...     loss=0.04)
```

build_solph_components()

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

Bases: oemof.solph.network.sink.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)
- **marginal_utility** (*numeric*) – Marginal utility in for example Euro / MWh
- **input_parameters** (*dict (optional)*)

$$x^{flow}(t) = c^{amount}(t) \cdot x^{flow}(t) \qquad \forall t \in T$$

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_bus = solph.Bus('my_bus')
>>> my_load = Load(
...     label='load',
...     carrier='electricity',
...     bus=my_bus,
...     amount=100,
...     profile=[0.3, 0.2, 0.5])
```

build_solph_components()

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

Bases: oemof.solph.components.generic_storage.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 inflow 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

The reservoir is modelled as a storage with a constant inflow:

$$x^{level}(t) = x^{level}(t-1) \cdot (1 - c^{loss\_rate}(t)) + x^{profile}(t) - \frac{x^{flow,out}(t)}{c^{efficiency}(t)} \qquad \forall t \in T$$

$$x^{level}(0) = 0.5 \cdot c^{capacity}$$

The inflow is bounded by the exogenous inflow profile. Thus if the inflow exceeds the maximum capacity of the storage, spillage is possible by setting $x^{profile}(t)$ to lower values.

$$0 \leq x^{profile}(t) \leq c^{profile}(t) \qquad \forall t \in T$$

The spillage of the reservoir is therefore defined by: $c^{profile}(t) - x^{profile}(t)$.

Note: As the Reservoir is a sub-class of *oemof.solph.GenericStorage* you also pass all arguments of this class.

Examples

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

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_bus = solph.Bus('my_bus')
>>> my_reservoir = Reservoir(
...     label='my_reservoir',
...     bus=my_bus,
...     carrier='water',
...     tech='reservoir',
...     storage_capacity=1000,
...     capacity=50,
...     profile=[1, 2, 6],
...     loss_rate=0.01,
...     initial_storage_level=0,
...     max_storage_level = 0.9,
...     efficiency=0.93)
```

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.generic_storage.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)
- **efficiency** (*numeric*) – Efficiency of charging and discharging process: Default: 1
- **storage_capacity_cost** (*numeric*) – Investment costs for the storage unit e.g in €/MWh-capacity
- **capacity_cost** (*numeric*) – Investment costs for the storage unit e.g in €/MW-capacity
- **expandable** (*boolean*) – True, if capacity can be expanded within optimization. Default: False.
- **storage_capacity_potential** (*numeric*) – Potential of the investment for storage capacity in MWh. Default: +inf.
- **capacity_potential** (*numeric*) – Potential of the investment for capacity in MW. Default: +inf.
- **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)

Intertemporal energy balance of the storage:

$$x^{level}(t) = x^{level}(t-1) \cdot (1 - c^{loss\_rate}) + \sqrt{c^{efficiency}(t)} x^{flow,in}(t) - \frac{x^{flow,out}(t)}{\sqrt{c^{efficiency}(t)}} \qquad \forall t \in T$$

$$x^{level}(0) = 0.5 \cdot c^{capacity}$$

The **expression** added to the cost minimizing objective funtion for the operation is given as:

$$x^{opex} = \sum_t (x^{flow,out}(t) \cdot c^{marginal\_cost}(t))$$

Examples

```
>>> import pandas as pd
>>> from oemof import solph
>>> from oemof.tabular import facades as fc
>>> my_bus = solph.Bus('my_bus')
>>> es = solph.EnergySystem(
...     timeindex=pd.date_range('2019', periods=3, freq='H'))
>>> es.add(my_bus)
>>> es.add(
...     fc.Storage(
...         label="storage",
...         bus=my_bus,
...         carrier="lithium",
...         tech="battery",
...         storage_capacity_cost=10,
...         invest_relation_output_capacity=1/6, # oemof.solph
...         marginal_cost=5,
```

(continues on next page)

(continued from previous page)

```
...         balanced=True, # oemof.solph argument
...         initial_storage_level=1, # oemof.solph argument
...         max_storage_level=[0.9, 0.95, 0.8])) # oemof.solph argument
```

build_solph_components()

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

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

Volatile element with one output. This class can be used to model PV oder Wind power plants.

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_parameters** (*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
- **capacity_minimum** (*numeric*) – Minimum install capacity if investment
- **expandable** (*boolean*) – True, if capacity can be expanded within optimization. Default: False.

The mathematical representations for this components are dependent on the user defined attributes. If the capacity is fixed before (**dispatch mode**) the following equation holds:

$$x^{flow}(t) = c^{capacity} \cdot c^{profile}(t) \qquad \forall t \in T$$

Where $x^{flow}_{volatile}$ denotes the production (endogenous variable) of the volatile object to the bus.

If *expandable* is set to *True* (**investment mode**), the equation changes slightly:

$$x^{flow}(t) = (x^{capacity} + c^{capacity}) \cdot c^{profile}(t) \qquad \forall t \in T$$

Where the bounded endogenous variable of the volatile component is added:

$$x^{capacity}_{volatile} \leq c^{capacity\_potential}_{volatile}$$

Ojective expression for operation:

$$x^{opex} = \sum_t (x^{flow}(t) \cdot c^{marginal\_cost}(t))$$

Examples

```
>>> from oemof import solph
>>> from oemof.tabular import facades
>>> my_bus = solph.Bus('my_bus')
>>> my_volatile = Volatile(
...     label='wind',
...     bus=my_bus,
...     carrier='wind',
...     tech='onshore',
...     capacity_cost=150,
...     profile=[0.25, 0.1, 0.3])
```

build_solph_components()

oemof.tabular.facades.**add_subnodes**(*n*, ***kwargs*)

CHAPTER 5

Tutorials

5.1 Using the datapackage-reader

[1]:
```
import os
import pandas as pd
import pkg_resources as pkg
import pprint

from pyomo.opt import SolverFactory
from oemof.solph import EnergySystem, Model
from oemof.tabular.facades import TYPEMAP
import oemof.tabular.tools.postprocessing as pp

from oemof.tabular import datapackage
```

5.1.1 Setting Path to Datapackage

The package comes with some example datapackage that you can use for testing. You can adapt this path to point to your datapackage. This scripts should work without any necessary additional changes.

[2]:
```
name = "investment" # choose from ['dispatch', 'investment']

# path to directory with datapackage to load
datapackage_dir = pkg.resource_filename(
    "oemof.tabular", "examples/datapackages/{}".format(name)
)
```

The results path points to your home directory, a subdirectory oemof-results and the name of the datapackage specified above.

[3]:
```
# create  path for results (we use the datapackage_dir to store results)
results_path = os.path.join(os.path.expanduser("~"), "oemof-results", name, "output")
if not os.path.exists(results_path):
    os.makedirs(results_path)
```

5.1.2 Setting `attributmap` and `typemap`

The two arguments allow for adjusting the datapackage reader to your needs. The `attribute` map lets you specifiy to map column names of your datapackage resource (field names) to the facades. Take the following example:

The `Load` facade requires the argument `amount`. If you like (for whatever reason) to use a different naming in your csv-file like `total_energy` you can do this be specifying the following:

```
from oemof.tabular import facades as fc

...

attributemap = {
    fc.Load: {'amount', 'total_energy'}
}
```

So you can set the attribute map individually for all facades. However, we will use no mapping here for `attributemap` argument.

[4]:
```
attributemap= {}
```

The `typemap` argument can be specified for your field (column name) `type` that specifcies which facade class should be used for instantiating the objects. We provide a default typemap. If you add your own facade classes, you **must** add these to typemap

```
typemap.update(
    {'my_new_class': my_class}
)
```

[5]:
```
typemap = TYPEMAP

# Look at current typemap
pprint.pprint(typemap)
```

```
{'backpressure': <class 'oemof.tabular.facades.BackpressureTurbine'>,
 'bus': <class 'oemof.solph.network.bus.Bus'>,
 'commodity': <class 'oemof.tabular.facades.Commodity'>,
 'conversion': <class 'oemof.tabular.facades.Conversion'>,
 'dispatchable': <class 'oemof.tabular.facades.Dispatchable'>,
 'electrical bus': <class 'oemof.solph.custom.electrical_line.ElectricalBus'>,
 'electrical line': <class 'oemof.solph.custom.electrical_line.ElectricalLine'>,
 'excess': <class 'oemof.tabular.facades.Excess'>,
 'extraction': <class 'oemof.tabular.facades.ExtractionTurbine'>,
 'generator': <class 'oemof.tabular.facades.Generator'>,
 'heatpump': <class 'oemof.tabular.facades.HeatPump'>,
 'link': <class 'oemof.tabular.facades.Link'>,
 'load': <class 'oemof.tabular.facades.Load'>,
 'reservoir': <class 'oemof.tabular.facades.Reservoir'>,
 'shortage': <class 'oemof.tabular.facades.Shortage'>,
 'storage': <class 'oemof.tabular.facades.Storage'>,
 'volatile': <class 'oemof.tabular.facades.Volatile'>}
```

5.1.3 Create EnergySystem from Datapackage

Using the `.from_datapackge` method, creating your EnergySystem is straight forward

[6]:
```
# create energy system object
es = EnergySystem.from_datapackage(
    os.path.join(datapackage_dir, "datapackage.json"),
    attributemap=attributemap,
    typemap=typemap,
)
pprint.pprint(
    {n.label: n for n in es.nodes}
)
```

```
{'bp': "<oemof.tabular.facades.BackpressureTurbine: 'bp'>",
 'bus0': "<oemof.solph.network.bus.Bus: 'bus0'>",
 'bus1': "<oemof.solph.network.bus.Bus: 'bus1'>",
 'coal-st': "<oemof.tabular.facades.Dispatchable: 'coal-st'>",
 'conn1': "<oemof.tabular.facades.Link: 'conn1'>",
 'conn2': "<oemof.tabular.facades.Link: 'conn2'>",
 'demand0': "<oemof.tabular.facades.Load: 'demand0'>",
 'demand1': "<oemof.tabular.facades.Load: 'demand1'>",
 'el-storage': "<oemof.tabular.facades.Storage: 'el-storage'>",
 'ext': "<oemof.tabular.facades.ExtractionTurbine: 'ext'>",
 'gas-bus': "<oemof.solph.network.bus.Bus: 'gas-bus'>",
 'gas-gt': "<oemof.tabular.facades.Dispatchable: 'gas-gt'>",
 'heat-bus': "<oemof.solph.network.bus.Bus: 'heat-bus'>",
 'lignite-st': "<oemof.tabular.facades.Dispatchable: 'lignite-st'>",
 'power2heat': "<oemof.tabular.facades.Conversion: 'power2heat'>",
 'pv': "<oemof.tabular.facades.Volatile: 'pv'>",
 'wind': "<oemof.tabular.facades.Volatile: 'wind'>"}
```

5.1.4 Create Model

This again is straight forward and just the way you would use the `oemof.solph` package.

[7]:
```
# check if the coin-branch-and-cut (cbc) solver library is available
cbc = SolverFactory('cbc').available()

if cbc:
    # create model from energy system (this is just oemof.solph)
    m = Model(es)

    # if you want dual variables / shadow prices uncomment line below
    # m.receive_duals()

    # select solver 'gurobi', 'cplex', 'glpk' etc
    m.solve("cbc")

    # get the results from the the solved model(still oemof.solph)
    results = m.results()
```

```
WARNING: Could not locate the 'cbc' executable, which is required for solver
    cbc
```

5.1.5 Writing Results

The nice thing about `oemof.tabular.facades` classes is their data model that allows for writing results in a standardized way. If you want more indidual result postprocessing, you can use all functionalities of the `oemof.outputlib` as the results object is a standard object.

[8]:
```
if cbc:
    # now we use the write results method to write the results in oemof-tabular
    # format
    pp.write_results(m, results, results_path)
```

5.1.6 Postprocessing Results

[9]:
```
if cbc:
    print(os.listdir(results_path))
    bus = 'bus0'
    path = os.path.join(results_path, ("").join([bus, '.csv']))
    df = pd.read_csv(path, index_col=0, parse_dates=True)
```

5.2 Using oemof.tabular.facades

This ipython-notebook is designed to describe the usage and functionality of the facades that are based on the oemof.solph package. If you are scripting and writing your own model you can easily use the classes provided by solph. The main potential of the facades is to provide an easy interface when defining the energy system in a input datapackage or any other tabular source. To see how you can do this have a look at the `model-from-tabular-data.ipynb` example.

NOTE: Numeric values in this example are not representative for any energy system and just randomly selected. Also this model is not necessarily a feasible optimization problem. It is **only** designed for illustration of class usage.

Author:

Simon.Hilpert (@uni-flensburg.de), Europa Universitaet Flensburg, March 2019

5.2.1 Python Imports

[1]:
```
import pandas as pd

from oemof.solph import EnergySystem, Model, Bus
import oemof.tabular.facades as fc

# for simplicity we just use 3 timesteps
es = EnergySystem(timeindex=pd.date_range('2018', periods=3, freq='H'))
```

5.2.2 Bus

First we will create the required buses for this example. As these objects will be used when instantiating the components, we assign them to python variables and add these to the energy system object `es` using ist `.add()` method.

The `balanced` (default: True) argument can be used to relax the energy balance for a bus, i.e. all inputs do not have to sum with all output flows to zero any longer.

[2]:
```
elec_bus = Bus(label="elec_bus")
elec_bus_1 = Bus(label="elec_bus_1")
heat_bus = Bus(label="heat_bus")
fuel_bus = Bus(label="fuel_bus", balanced=True)

es.add(elec_bus, elec_bus_1, heat_bus, fuel_bus)
```

5.2.3 Volatile

This class can be used to model PV oder Wind power plants. The equations for this component are:

$$x_{wind}^{flow} = c_{wind}^{capacity} \cdot c_{wind}^{profile}(t) \qquad \forall t \in T$$

Where x_{wind}^{flow} denotes the production (endogenous variable) of the volatile object to the bus.

[3]:
```
es.add(
    fc.Volatile(
        label="wind",
        carrier="wind",
        tech="onshore",
        capacity=150,
        bus=elec_bus,
        profile=[0.2, 0.3, 0.25],
    )
)
```

Volatile component investment

If the investment mode is used, i.e. `capacity_cost` attribute not None and `capacity` attribute set to None the right hand side of the equation changes as the exogenous variable $c^{capacity}$ is now replaced by an endognous variable.

$$x_{wind}^{f,ow} \leq x_{wind}^{capacity} \cdot c_{wind}^{profile}(t) \qquad \forall t \in T$$

[4]:
```
es.add(
    fc.Volatile(
        label="wind_invest",
        carrier="wind",
        tech="onshore",
        capacity_cost=200,
        exapandable=True,
        bus=elec_bus,
        profile=[0.2, 0.3, 0.25],
    )
)
```

5.2.4 Dispatchable

The `Dispatchble` component works very similar to the volatile component. The only difference here is the $\leq$ sign in the constraint, which allows to dispatch the power within the limit of the lower and upper bounds.

$$x_{flow}^{elec\_bus} \leq c_{ccgt}^{capacity} \cdot c_{ccgt}^{profile}(t) \qquad \forall t \in T$$

NOTE:

You can also set the parameters for the output of this component by using the argument `output_parameters`. To see all options see: oemof Flow

[5]:
```
es.add(
    fc.Dispatchable(
        bus=elec_bus,
        label="ccgt",
        carrier="gas",
        tech="ccgt",
        capacity=100,
        marginal_cost=25,
        output_parameters={
            'summed_min': 1000,
            'summed_max': 2000}
    )
)
```

5.2.5 Reservoir

The reservoir component inherit from the GenericStorage. However the input of the reservoir is not taken from the bus but defined as an absolute inflow to the reservoir from a source. This source object is created under the hood interanally when a `Reservoir` object is instantiated.

[6]:
```
es.add(
    fc.Reservoir(
        bus=elec_bus,
        label="rsv",
        carrier="hydro",
        tech="reservoir",
        capacity=150,
        storage_capacity=1500,
        efficiency=0.9,
        profile=[10, 5, 3],
        initial_storage_level=1, # oemof.solph arguments
        balanced=False # oemof.solph argument
    )
)
```

5.2.6 Storage

The Storage component is based on the GenericStorage class of `oemof.solph`. Therefore you may use all arguments that exist for the parent class in addition to the ones defined for the component itself.

[7]:
```
es.add(
    fc.Storage(
        label="storage",
        bus=elec_bus,
        carrier="lithium",
        tech="battery",
        capacity_cost=10,
        expandable=True,
        invest_relation_output_capacity=1/6, # oemof.solph
        marginal_cost=5,
        balanced=True, # oemof.solph argument
        initial_storage_level=1, # oemof.solph argument
        max_storage_level=[0.9, 0.95, 0.8], # oemof.solph argument
    )
)
```

5.2.7 Extraction and Backpressure Turbine

The extraction turbine facade directly inherit from the ExtractionTurbineCHP class where as the backpressure facade directly inherit from the Transformer class.

Both components may also be used in the investment mode by setting the capacity costs. The capacity cost are related to the electrical output of the component, i.e. Euro/MWhel.

[8]:
```
es.add(
    fc.ExtractionTurbine(
        label="ext",
        electricity_bus=elec_bus,
        heat_bus=heat_bus,
        fuel_bus=fuel_bus,
        carrier='gas',
        tech='ext',
        capacity=10,
        condensing_efficiency=0.5,
        electric_efficiency=0.4,
        thermal_efficiency=0.3
    )
)

es.add(
    fc.BackpressureTurbine(
        label="bp",
        electricity_bus=elec_bus,
        heat_bus=heat_bus,
        fuel_bus=fuel_bus,
        carrier='gas',
        tech='bp',
        capacity=10,
        electric_efficiency=0.4,
        thermal_efficiency=0.3
    )
)
```

5.2.8 Conversion

The conversion component is a simplified interface to the Transformer class with the **restriction of 1 input and 1 output**.

[9]:
```
es.add(
    fc.Conversion(
        label='pth',
        from_bus=elec_bus,
        to_bus=heat_bus,
        carrier='electricity',
        tech='hp',
        capacity=10,
        capacity_cost=54,
        expandable=True,
        capacity_potential=20,
        efficiency=0.9,
        thermal_efficiency=0.3
    )
)
```

5.2.9 Commodity

A commodity can be used to model a limited source for the complete time horizon of the problem.

$$\sum_t x_{fuel}^{flow}(t) \leq c_{fuel}^{amount}$$

[10]:
```
es.add(
    fc.Commodity(
        label='fuel',
        bus=fuel_bus,
        amount=1000000,
        carrier='fuel',
        tech='commodity',
        marginal_cost=0.5
    )
)
```

5.2.10 Link

This component also just provides an simpliefied interface for the Link of the oemof solph package.

[11]:
```
es.add(
    fc.Link(
        label='link',
        from_bus=elec_bus,
        to_bus=elec_bus_1,
        loss=0.05,
        capacity=100
    )
)
```

5.2.11 Load

The load is similar to the `Volatile` component, except that it acts as a sink (1-input). Therefore you may also use the argument `input_parameters` to adapt its behaviour.

[12]:
```
es.add(
    fc.Load(
        label="elec_load",
        bus=elec_bus,
        amount=500e3,
        profile=[0.4, 0.1, 0.5]))

es.add(
    fc.Load(
        label="heat_load",
        bus=heat_bus,
        amount=200e3,
        profile=[0.1, 0.23, 0.7]))
```

5.2.12 Adding Other oemof.solph Components

As the energy system and all components are solph based objects you may add any other oemof.solph object to your energy system. This is straight forward when you are scripting. The full levearge of the facades is however gained when using the datapackage reader. For this datapackage reader **only** facades are working in a proper way.

5.2.13 Create Model and Inspect

[13]:
```
m = Model(es)

# uncommet to get lp-file (path will be of this file)
# m.write(io_options={'symbolic_solver_labels': True})
```

[14]:
```
m.InvestmentFlow.pprint()
```

```
InvestmentFlow : Size=1, Index=None, Active=True
    11 Set Declarations
        CONVEX_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     2 :    Any :    3 : {("<oemof.tabular.facades.Storage: 'storage
↪'>", "<oemof.solph.network.bus.Bus: 'elec_bus'>"), ("<oemof.solph.network.bus.Bus:
↪'elec_bus'>", "<oemof.tabular.facades.Storage: 'storage'>"), ("<oemof.tabular.
↪facades.Conversion: 'pth'>", "<oemof.solph.network.bus.Bus: 'heat_bus'>")}
        FIXED_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     2 :    Any :    3 : {("<oemof.tabular.facades.Storage: 'storage
↪'>", "<oemof.solph.network.bus.Bus: 'elec_bus'>"), ("<oemof.solph.network.bus.Bus:
↪'elec_bus'>", "<oemof.tabular.facades.Storage: 'storage'>"), ("<oemof.tabular.
↪facades.Conversion: 'pth'>", "<oemof.solph.network.bus.Bus: 'heat_bus'>")}
        MIN_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
```

(continues on next page)

(continued from previous page)

```
            None :    -- :    Any :    0 :      {}
        NON_CONVEX_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        NON_FIXED_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     2 :    Any :    3 : {("<oemof.tabular.facades.Storage: 'storage
↪'>", "<oemof.solph.network.bus.Bus: 'elec_bus'>"), ("<oemof.solph.network.bus.Bus:
↪'elec_bus'>", "<oemof.tabular.facades.Storage: 'storage'>"), ("<oemof.tabular.
↪facades.Conversion: 'pth'>", "<oemof.solph.network.bus.Bus: 'heat_bus'>")}
        SUMMED_MAX_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        SUMMED_MIN_INVESTFLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        fixed_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                      : Size : Members
            None :    -- : InvestmentFlow.FIXED_INVESTFLOWS*TIMESTEPS :    0 :      {}
        max_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                        : Size :␣
↪Members
            None :     3 : InvestmentFlow.NON_FIXED_INVESTFLOWS*TIMESTEPS :    9 : {("
↪<oemof.tabular.facades.Storage: 'storage'>", "<oemof.solph.network.bus.Bus: 'elec_
↪bus'>", 0), ("<oemof.tabular.facades.Storage: 'storage'>", "<oemof.solph.network.
↪bus.Bus: 'elec_bus'>", 1), ("<oemof.tabular.facades.Storage: 'storage'>", "<oemof.
↪solph.network.bus.Bus: 'elec_bus'>", 2), ("<oemof.solph.network.bus.Bus: 'elec_bus'>
↪", "<oemof.tabular.facades.Storage: 'storage'>", 0), ("<oemof.solph.network.bus.Bus:
↪ 'elec_bus'>", "<oemof.tabular.facades.Storage: 'storage'>", 1), ("<oemof.solph.
↪network.bus.Bus: 'elec_bus'>", "<oemof.tabular.facades.Storage: 'storage'>", 2), ("
↪<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.network.bus.Bus: 'heat_bus
↪'>", 0), ("<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.network.bus.
↪Bus: 'heat_bus'>", 1), ("<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.
↪network.bus.Bus: 'heat_bus'>", 2)}
        min_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                      : Size : Members
            None :    -- : InvestmentFlow.MIN_INVESTFLOWS*TIMESTEPS :    0 :      {}

    2 Var Declarations
        invest : Size=3, Index=InvestmentFlow.INVESTFLOWS
            Key
↪                  : Lower : Value : Upper : Fixed : Stale : Domain                 ␣
            ("<oemof.solph.network.bus.Bus: 'elec_bus'>", "<oemof.tabular.facades.
↪Storage: 'storage'>") :     0 :  None :   inf : False :  True : NonNegativeReals
             ("<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.network.bus.
↪Bus: 'heat_bus'>") :     0 :  None :    10 : False :  True : NonNegativeReals
            ("<oemof.tabular.facades.Storage: 'storage'>", "<oemof.solph.network.bus.
↪Bus: 'elec_bus'>") :     0 :  None :   inf : False :  True : NonNegativeReals
        invest_status : Size=0, Index=InvestmentFlow.NON_CONVEX_INVESTFLOWS
            Key : Lower : Value : Upper : Fixed : Stale : Domain

    1 Expression Declarations
        investment_costs : Size=1, Index=None
            Key  : Expression
            None : 10*InvestmentFlow.invest[elec_bus,storage] + 54*InvestmentFlow.
↪invest[pth,heat_bus]
```

(continues on next page)

(continued from previous page)

```
    7 Constraint Declarations
        fixed : Size=0, Index=InvestmentFlow.fixed_index, Active=True
            Key : Lower : Body : Upper : Active
        max : Size=9, Index=InvestmentFlow.max_index, Active=True
            Key
↪                    : Lower : Body
↪                : Upper : Active
            ("<oemof.solph.network.bus.Bus: 'elec_bus'>", "<oemof.tabular.facades.
↪Storage: 'storage'>", 0) :  -Inf : flow[elec_bus,storage,0] - InvestmentFlow.
↪invest[elec_bus,storage] :   0.0 :   True
            ("<oemof.solph.network.bus.Bus: 'elec_bus'>", "<oemof.tabular.facades.
↪Storage: 'storage'>", 1) :  -Inf : flow[elec_bus,storage,1] - InvestmentFlow.
↪invest[elec_bus,storage] :   0.0 :   True
            ("<oemof.solph.network.bus.Bus: 'elec_bus'>", "<oemof.tabular.facades.
↪Storage: 'storage'>", 2) :  -Inf : flow[elec_bus,storage,2] - InvestmentFlow.
↪invest[elec_bus,storage] :   0.0 :   True
             ("<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.network.bus.
↪Bus: 'heat_bus'>", 0) :  -Inf :  flow[pth,heat_bus,0] - (10 + InvestmentFlow.
↪invest[pth,heat_bus]) :   0.0 :   True
             ("<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.network.bus.
↪Bus: 'heat_bus'>", 1) :  -Inf :  flow[pth,heat_bus,1] - (10 + InvestmentFlow.
↪invest[pth,heat_bus]) :   0.0 :   True
             ("<oemof.tabular.facades.Conversion: 'pth'>", "<oemof.solph.network.bus.
↪Bus: 'heat_bus'>", 2) :  -Inf :  flow[pth,heat_bus,2] - (10 + InvestmentFlow.
↪invest[pth,heat_bus]) :   0.0 :   True
            ("<oemof.tabular.facades.Storage: 'storage'>", "<oemof.solph.network.bus.
↪Bus: 'elec_bus'>", 0) :  -Inf : flow[storage,elec_bus,0] - InvestmentFlow.
↪invest[storage,elec_bus] :   0.0 :   True
            ("<oemof.tabular.facades.Storage: 'storage'>", "<oemof.solph.network.bus.
↪Bus: 'elec_bus'>", 1) :  -Inf : flow[storage,elec_bus,1] - InvestmentFlow.
↪invest[storage,elec_bus] :   0.0 :   True
            ("<oemof.tabular.facades.Storage: 'storage'>", "<oemof.solph.network.bus.
↪Bus: 'elec_bus'>", 2) :  -Inf : flow[storage,elec_bus,2] - InvestmentFlow.
↪invest[storage,elec_bus] :   0.0 :   True
        maximum_rule : Size=0, Index=InvestmentFlow.NON_CONVEX_INVESTFLOWS,␣
↪Active=True
            Key : Lower : Body : Upper : Active
        min : Size=0, Index=InvestmentFlow.min_index, Active=True
            Key : Lower : Body : Upper : Active
        minimum_rule : Size=0, Index=InvestmentFlow.NON_CONVEX_INVESTFLOWS,␣
↪Active=True
            Key : Lower : Body : Upper : Active
        summed_max : Size=0, Index=InvestmentFlow.SUMMED_MAX_INVESTFLOWS, Active=True
            Key : Lower : Body : Upper : Active
        summed_min : Size=0, Index=InvestmentFlow.SUMMED_MIN_INVESTFLOWS, Active=True
            Key : Lower : Body : Upper : Active

    21 Declarations: INVESTFLOWS CONVEX_INVESTFLOWS NON_CONVEX_INVESTFLOWS FIXED_
↪INVESTFLOWS NON_FIXED_INVESTFLOWS SUMMED_MAX_INVESTFLOWS SUMMED_MIN_INVESTFLOWS MIN_
↪INVESTFLOWS invest invest_status minimum_rule maximum_rule fixed_index fixed max_
↪index max min_index min summed_max summed_min investment_costs
```

[15]:
```
m.Flow.pprint()
```

```
Flow : Size=1, Index=None, Active=True
    11 Set Declarations
        INTEGER_FLOWS : Size=1, Index=None, Ordered=Insertion
```

(continues on next page)

(continued from previous page)

```
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        NEGATIVE_GRADIENT_FLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        POSITIVE_GRADIENT_FLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :    -- :    Any :    0 :      {}
        SUMMED_MAX_FLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     2 :    Any :    2 : {("<oemof.tabular.facades.Commodity: 'fuel
↪'>", "<oemof.solph.network.bus.Bus: 'fuel_bus'>"), ("<oemof.tabular.facades.
↪Dispatchable: 'ccgt'>", "<oemof.solph.network.bus.Bus: 'elec_bus'>")}
        SUMMED_MIN_FLOWS : Size=1, Index=None, Ordered=Insertion
            Key  : Dimen : Domain : Size : Members
            None :     2 :    Any :    1 : {("<oemof.tabular.facades.Dispatchable:
↪'ccgt'>", "<oemof.solph.network.bus.Bus: 'elec_bus'>"),}
        integer_flow_constr_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                       : Size : Members
            None :    -- : Flow.INTEGER_FLOWS*TIMESTEPS :    0 :      {}
        integer_flow_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                       : Size : Members
            None :    -- : Flow.INTEGER_FLOWS*TIMESTEPS :    0 :      {}
        negative_gradient_constr_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                : Size : Members
            None :    -- : Flow.NEGATIVE_GRADIENT_FLOWS*TIMESTEPS :    0 :      {}
        negative_gradient_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                : Size : Members
            None :    -- : Flow.NEGATIVE_GRADIENT_FLOWS*TIMESTEPS :    0 :      {}
        positive_gradient_constr_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                : Size : Members
            None :    -- : Flow.POSITIVE_GRADIENT_FLOWS*TIMESTEPS :    0 :      {}
        positive_gradient_index : Size=1, Index=None, Ordered=True
            Key  : Dimen : Domain                                : Size : Members
            None :    -- : Flow.POSITIVE_GRADIENT_FLOWS*TIMESTEPS :    0 :      {}

    3 Var Declarations
        integer_flow : Size=0, Index=Flow.integer_flow_index
            Key : Lower : Value : Upper : Fixed : Stale : Domain
        negative_gradient : Size=0, Index=Flow.negative_gradient_index
            Key : Lower : Value : Upper : Fixed : Stale : Domain
        positive_gradient : Size=0, Index=Flow.positive_gradient_index
            Key : Lower : Value : Upper : Fixed : Stale : Domain

    5 Constraint Declarations
        integer_flow_constr : Size=0, Index=Flow.integer_flow_constr_index,␣
↪Active=True
            Key : Lower : Body : Upper : Active
        negative_gradient_constr : Size=0, Index=Flow.negative_gradient_constr_index,␣
↪Active=True
            Key : Lower : Body : Upper : Active
        positive_gradient_constr : Size=0, Index=Flow.positive_gradient_constr_index,␣
↪Active=True
            Key : Lower : Body : Upper : Active
        summed_max : Size=2, Index=Flow.SUMMED_MAX_FLOWS, Active=True
            Key                                                                    ␣
↪                    : Lower : Body                                                ␣
↪                : Upper     : Active
```

(continues on next page)

(continued from previous page)

```
            ("<oemof.tabular.facades.Dispatchable: 'ccgt'>", "<oemof.solph.network.
↪bus.Bus: 'elec_bus'>") :  -Inf : flow[ccgt,elec_bus,0] + flow[ccgt,elec_bus,1] +␣
↪flow[ccgt,elec_bus,2] :  200000.0 :   True
                ("<oemof.tabular.facades.Commodity: 'fuel'>", "<oemof.solph.network.
↪bus.Bus: 'fuel_bus'>") :  -Inf : flow[fuel,fuel_bus,0] + flow[fuel,fuel_bus,1] +␣
↪flow[fuel,fuel_bus,2] : 1000000.0 :   True
        summed_min : Size=1, Index=Flow.SUMMED_MIN_FLOWS, Active=True
            Key
↪                    : Lower    : Body                                                ␣
↪                  : Upper : Active
            ("<oemof.tabular.facades.Dispatchable: 'ccgt'>", "<oemof.solph.network.
↪bus.Bus: 'elec_bus'>") : 100000.0 : flow[ccgt,elec_bus,0] + flow[ccgt,elec_bus,1] +␣
↪flow[ccgt,elec_bus,2] :  +Inf :   True

    4 BuildAction Declarations
        negative_gradient_build : Size=0, Index=None, Active=True
        positive_gradient_build : Size=0, Index=None, Active=True
        summed_max_build : Size=0, Index=None, Active=True
        summed_min_build : Size=0, Index=None, Active=True

    23 Declarations: SUMMED_MAX_FLOWS SUMMED_MIN_FLOWS NEGATIVE_GRADIENT_FLOWS␣
↪POSITIVE_GRADIENT_FLOWS INTEGER_FLOWS positive_gradient_index positive_gradient␣
↪negative_gradient_index negative_gradient integer_flow_index integer_flow summed_
↪max summed_max_build summed_min summed_min_build positive_gradient_constr_index␣
↪positive_gradient_constr positive_gradient_build negative_gradient_constr_index␣
↪negative_gradient_constr negative_gradient_build integer_flow_constr_index integer_
↪flow_constr
```

5.3 Creating energy systems from spreadsheet

[1]:
```
import os
import pkg_resources as pkg
import pandas as pd

from pyomo.opt import SolverFactory
from oemof.solph import EnergySystem, Model, Bus
from oemof.tools.economics import annuity as annuity
from oemof.solph import constraints
import oemof.tabular.tools.postprocessing as pp
import oemof.tabular.facades as fc
```

5.3.1 Creating and Setting the Datapaths

Setting the datapath for raw-data and results. Data handling looks more complex than it is. You can easily adapt this to a simple `pd.read_excel(filepath,...)` in the next block if your file is located somewhere else. Otherwise we will use data from the oemof tabular repository.

In addition a results directory will be created in `home/user/oemof-results/dispatch/output`.

[2]:
```
scenario_name = "base-scenario"

# datapath for input data from the oemof tabular pacakge
```

(continues on next page)

(continued from previous page)

```
datapath = os.path.join(
    pkg.resource_filename("oemof.tabular", ""),
    "data/data.xls",
)

# results path
results_path = os.path.join(
    os.path.expanduser("~"), "oemof-results"
)

scenario_path = os.path.join(
    results_path, scenario_name, "output"
)

if not os.path.exists(scenario_path):
    os.makedirs(scenario_path)
print(scenario_path)
```

```
/home/docs/oemof-results/base-scenario/output
```

Next we will read the required input data. The profiles index will be used for the `EnergySystem` object below. All generator data etc. will also be loaded.

[3]:
```
file = pd.ExcelFile(datapath)

sheet_names = [typ for typ in file.sheet_names if typ in fc.TYPEMAP.keys()]

data = {}

for sheet in sheet_names:
    data[sheet]= pd.read_excel(datapath, sheet_name=sheet, index_col=0)

# profiles and tech data not be part of TYPEMAP and need to be read seperately
profiles = pd.read_excel(
    datapath,
    sheet_name="profiles",
    index_col=[0],
    parse_dates=True,
)
profiles.index.freq = "1H"

technology = pd.read_excel(
    datapath, sheet_name="technology-data", index_col=[0, 1]
)
carrier = pd.read_excel(
    datapath, sheet_name="carrier", index_col=[0]
)
```

[4]:
```
all_components = pd.concat([v for k, v in data.items() if k != "bus"], sort=False)
# Only be used for Latex export of tables
#columns = ['profile', 'capacity_potential']
#print(all_components.to_latex(columns=columns, na_rep="-"))
```

5.3.2 Creating the EnergySystem and its Nodes

We are starting by creating a `EnergySystem` object which will hold all information (nodes, etc.) of hour energy system that we will add below. This is just the standard way of using the `oemof.solph` library for your modelling.

[5]:
```
es = EnergySystem(timeindex=profiles.index)
```

Add Bus

Before we add any components we will create all bus objects for our model and add it to the energy system object.

[6]:
```
buses = {
    name: Bus(label=name, balanced=bool(arg.balanced))
    for name, arg in data["bus"].iterrows()
}
es.add(*buses.values())
```

Bus Constraints

With the set of all Buses B all inputs $x^{flow}_{i(b),b}$ to a bus b must equal all its outputs $x^{flow}_{b,o(b)}$

$$\sum_i x^{flow}_{i(b),b}(t) - \sum_o x^{flow}_{b,o(b)}(t) = 0 \qquad \forall t \in T, \forall b \in B$$

This equation will be build once the complete energy system is setup with its component. Every time a `Component` is created, the connected bus inputs/outputs will be updated. By this update every bus has all required information of its inputs and outputs available to construct the constraints.

Add Load

[7]:
```
for name, l in data["load"].iterrows():
    es.add(
        fc.Load(
            label=name,
            bus=buses[
                l.bus
            ],  # reference the bus in the buses dictionary
            amount=l.amount,  # amount column
            profile=profiles[l.profile],
        )
    )
```

Load Constraint

For the set of all Load denoted with $l \in L$ the load x_l at timestep t equals the exogenously defined profile value $c^{profile}_l$ multiplied by the amount of this load c^{amount}_l

$$x^{flow}_l(t) = c^{profile}_l(t) \cdot c^{amount}_l \qquad \forall t \in T, \forall l \in L$$

Add Generators

[8]:
```
for name, g in data["dispatchable"].iterrows():
    es.add(
        fc.Dispatchable(
            label=name,
            bus=buses[g.bus],
            carrier=g.carrier,
            tech=g.tech,
            marginal_cost=(
                carrier.at[g.carrier, "cost"]
                / technology.at[
                    (g.carrier, g.tech), "efficiency"
                ]
            ),
            # efficiency=technology.at[(g.carrier, g.tech), 'efficiency'],
            expandable=g.expandable,
            capacity=g.capacity,
            capacity_potential=g.capacity_potential,
            capacity_cost=annuity(
                technology.at[
                    (g.carrier, g.tech), "capex"
                ],  # to $/MW
                technology.at[
                    (g.carrier, g.tech), "lifetime"
                ],
                0.07,
            )
            * 1000,
            output_parameters={
                "emission_factor": (
                    carrier.at[g.carrier, "emission_factor"]
                    / technology.at[
                        (g.carrier, g.tech), "efficiency"
                    ]
                )
            },
        )
    )
```

Dispatchable Generator Constraint

A `Generator` component can be used to model all types of dispatchble units in a energy system. This can include diesel generators oder coal fired power plants but also hot water boilers for heat. Every generator **must** be connected to an `Bus` object.

This basic mathematical model for the component with the set of all dispatchable generators being $d \in D$ looks as follows:

$$x_d^{flow}(t) \leq x_d^{capacity} \qquad \forall t \in T, \forall d \in D$$

Meaning, the production of the generator x^{flow} must be less than its maximum capacity c^{max} in every timestep.

[9]:
```
for name, v in data["volatile"].iterrows():
    es.add(
```

(continues on next page)

(continued from previous page)

```
        fc.Volatile(
            label=name,
            bus=buses[v.bus],
            carrier=v.carrier,
            tech=v.tech,
            expandable=v.expandable,
            capacity=v.capacity,
            capacity_potential=v.capacity_potential,
            capacity_cost=annuity(
                technology.at[(v.carrier, v.tech), "capex"],
                technology.at[
                    (v.carrier, v.tech), "lifetime"
                ],
                0.07,
            )
            * 1000,  # $/kW -> $/MW
            profile=profiles[v.profile],
        )
    )
```

Volatile Generator Constraint

Using the `Generator` component with `output_parameters={"fixed": True}` is very similar to the Dispatchable component. However, in this case the flow of the volatile components denoted with $v \in V$ will be fixed to a specific value.

$$x_v^{flow}(t) = c_v^{profile}(t) \cdot x_v^{capacity} \qquad \forall t \in T, \forall v \in V$$

Alternatively you can use the `Volatile` component which automatically enforced the fixed behaviour.

Add Storage

[10]:
```
for name, s in data["storage"].iterrows():
    es.add(
        fc.Storage(
            label=name,
            bus=buses[s.bus],
            carrier=s.carrier,
            tech=s.tech,
            marginal_cost=s.marginal_cost,
            capacity=s.capacity,
            storage_capacity=s.storage_capacity,
            expandable=s.expandable,
            efficiency=technology.at[
                (s.carrier, s.tech), "efficiency"
            ],
            loss_rate=s.loss_rate,
            storage_capacity_cost=annuity(
                technology.at[
                    (s.carrier, s.tech), "storage_capex"
                ],
                technology.at[
```

(continues on next page)

(continued from previous page)

```
                    (s.carrier, s.tech), "lifetime"
                ],
                0.07,
            )
            * 1000,  # $/kW -> $/MW
            capacity_cost=annuity(
                technology.at[(s.carrier, s.tech), "capex"],
                technology.at[
                    (s.carrier, s.tech), "lifetime"
                ],
                0.07,
            )
            * 1000,  # $/kW -> $/MW
        )
    )
```

Storage Constraints

The mathematical representation of the storage for all storages $s \in S$ will include the flow into the storage, out of the storage and a storage level. The defaul efficiency for input/output is 1. Note that this is is included during charge and discharge. If you want to set the round trip efficiency you need to do for example: $\eta = \sqrt{\eta^{roundtrip}}$

Intertemporal energy balance of the storage:

$$x_s^{level}(t) = \eta^{loss} x_s^{level}(t) + \eta^{in} x_{s,in}^{flow} - \frac{x_{s,out}^{flow}(t)}{\eta^{out}} \qquad \forall t \in T, \forall s \in S$$

Bounds of the storage level variable $x_s^{level}(t)$:

$$x_s^{level}(t) \leq c_s^{max,level} \qquad \forall t \in T, \forall s \in S$$

$$x_s^{level}(1) = x_s^{level}(t_e) = 0.5 \cdot c_s^{max,level} \qquad \forall t \in T, \forall s \in S$$

Of course, in addition the inflow/outflow of the storage also needs to be within the limit of the minimum and maximum power.

$$-c_s^{capacity} \leq x_s^{flow}(t) \leq c_s^{capacity} \qquad \forall t \in T, \forall s \in S$$

Add Conversion

A conversion unit will take from a bus and feed into another:

$$x_{c,to}^{flow}(t) = c_c^{efficiencty} \cdot x_{c,from}^{flow}(t), \qquad \forall c \in C, \forall t \in T$$

[11]:
```
for name, c in data["conversion"].iterrows():
    es.add(
        fc.Conversion(
            label=name,
            from_bus=buses[c.from_bus],
            to_bus=buses[c.to_bus],
```

(continues on next page)

(continued from previous page)

```
            carrier=c.carrier,
            tech=c.tech,
            efficiency=technology.at[
                (c.carrier, c.tech), "efficiency"
            ],
            marginal_cost=(
                carrier.at[c.carrier, "cost"]
                / technology.at[
                    (c.carrier, c.tech), "efficiency"
                ]
            ),
            expandable=c.expandable,
            capacity=c.capacity,
            capacity_potential=c.capacity_potential,
            capacity_cost=annuity(
                technology.at[(c.carrier, c.tech), "capex"],
                technology.at[
                    (c.carrier, c.tech), "lifetime"
                ],
                0.07,
            )
            * 1000,  # $/kW -> $/MW
            output_parameters={
                "emission_factor": (
                    carrier.at[c.carrier, "emission_factor"]
                    / technology.at[
                        (c.carrier, c.tech), "efficiency"
                    ]
                )
            },
        )
    )
```

Add Commodity

[12]:
```
for name, c in data["commodity"].iterrows():
    es.add(
        fc.Commodity(
            label=name,
            bus=buses[c.bus],
            carrier=c.carrier,
            tech=c.tech,
            amount=c.amount,
        )
    )
```

Add Link

[13]:
```
for name, c in data["link"].iterrows():
    es.add(
        fc.Link(
```

(continues on next page)

(continued from previous page)

```
            label=name,
            from_bus=buses[c.from_bus],
            to_bus=buses[c.to_bus],
            capacity=c.capacity,
            expandable=True,
            capacity_cost=99999,
            loss=c.loss
        )
    )
```

Objective Function

The objective function is created from all instantiated objects. It will use all operating costs (i.e. `marginal_cost` argument) and if set all investment costs (i.e. `capacity_cost` argument)

$$\text{min:} \sum_g \sum_t \overbrace{c_g^{marginal\_cost} \cdot x_g^{flow}(t)}^{\text{operating cost}}$$

$$\sum_g \sum_t \overbrace{c_g^{capacity\_cost} \cdot x_g^{capacity}(t)}^{\text{investment cost}}$$

Add Shortage/Excess Slack Components

[14]:
```
for name, e in data["excess"].iterrows():
    es.add(fc.Excess(label=name, bus=buses[e.bus]))

for name, s in data["shortage"].iterrows():
    es.add(
        fc.Shortage(
            label=name,
            carrier="electricity",
            tech="shortage",
            bus=buses[s.bus],
            marginal_cost=s.marginal_cost,
        )
    )
```

5.3.3 Creating the Mathematical Model

[15]:
```
# create model based on energy system and its components
m = Model(es)

# inspect objective function
# m.objective.pprint()

m.receive_duals()
```

```
WARNING: Reassigning the non-component attribute dual on block (model).Model
    with a new Component with type <class 'pyomo.core.base.suffix.Suffix'>.
    This is usually indicative of a modelling error. To avoid this warning,
    explicitly delete the attribute:
        del Model.dual
WARNING: Reassigning the non-component attribute rc on block (model).Model
    with a new Component with type <class 'pyomo.core.base.suffix.Suffix'>.
    This is usually indicative of a modelling error. To avoid this warning,
    explicitly delete the attribute:
        del Model.rc
```

Add CO2 Constraint

To add a CO2-constraint we will use the `oemof.solph.constraints` module which allows to add such a constraint in a easy way.

$$\sum_t \sum_f x_f^{flow}(t) \cdot c_f^{emission\_factor} \leq \overline{L_{CO_2}}$$

The constraint will sum all flows for the complete time horzion that have an attribute `emission_factor` and multiple the flow value with this factor.

[16]:
```
#m = constraints.emission_limit(m, limit=0)
m.write(io_options={"symbolic_solver_labels":True})
```

[16]:
```
('Model.lp', 140676637999248)
```

5.3.4 Solving the Model and Writing Results

[17]:
```
# check if cbc solver library is available
cbc = SolverFactory('cbc').available()

if cbc:
    #  solve the model using cbc solver
    m.solve("cbc")

    # write results back to the model object
    m.results = m.results()

    # writing results with the standard oemof-tabular output formatt
    pp.write_results(m, m.results, scenario_path)

    print(
        "Optimization done. Results are in {}.".format(
            results_path
        )
    )

    # write the lp-file
    # m.write(io_options={'symbolic_solver_labels': True})
```

```
WARNING: Could not locate the 'cbc' executable, which is required for solver
    cbc
```

5.3.5 Result Analysis

5.3.6 Plotting the Results

[18]:
```
if cbc:
    import os
    from plotly import offline
    from oemof.tabular.tools.plots import (
        hourly_plot,
        stacked_plot,
    )

    offline.init_notebook_mode(connected=True)


    offline.iplot(
        hourly_plot(
            scenario_name,
            "north-bus",
            os.path.join(
                os.path.expanduser("~"), "oemof-results"
            ),
            plot_filling_levels=False,
        ),
        filename=os.path.join(
            scenario_path, "hourly-plot.html"
        ),
    )
```

CHAPTER 6

Contributing

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

6.1 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.

6.2 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.

6.3 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 :)

6.4 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.

6.4.1 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`.

6.4.2 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 7

Authors

- Stephan Günther - https://oemof.org
- Simon Hilpert
- Martin Söthe
- Cord Kaldemeyer
- Jann Launer
- Hendrik Huyskens
- Monika Orlowski
- Francesco Witte
- Sarah Berendes
- Marie-Claire Gering

CHAPTER 8

Changelog

8.1 0.0.3 (2022-01-26)

8.1.1 Fixes

- Fix link by not setting constraints that limit direction #38
- Fix storage investment #33
- Link investment #28
- Variable cost #24
- Marginal cost #23

8.1.2 Features

- Adjust to new oemof.solph structure #21
- Allow to define custom foreign keys #39
- Add constraint tests for most facades #35, #42
- Reduce number of imported packages #32, #49
- Cleaned up the badges in README #59
- Move most CI services to github actions #37

8.2 0.0.2 (2019-07-08)

8.3 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.

8.4 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 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

oemof.tabular, 15
oemof.tabular.datapackage, 15
oemof.tabular.datapackage.aggregation, 15
oemof.tabular.datapackage.building, 16
oemof.tabular.datapackage.processing, 18
oemof.tabular.datapackage.reading, 18
oemof.tabular.facades, 21
oemof.tabular.tools, 19
oemof.tabular.tools.geometry, 20

Index

A

add_subnodes() (*in module oemof.tabular.facades*), 33

B

BackpressureTurbine (*class in oemof.tabular.facades*), 21
build_solph_components() (*oemof.tabular.facades.BackpressureTurbine method*), 22
build_solph_components() (*oemof.tabular.facades.Commodity method*), 23
build_solph_components() (*oemof.tabular.facades.Conversion method*), 24
build_solph_components() (*oemof.tabular.facades.Dispatchable method*), 25
build_solph_components() (*oemof.tabular.facades.Excess method*), 25
build_solph_components() (*oemof.tabular.facades.ExtractionTurbine method*), 26
build_solph_components() (*oemof.tabular.facades.HeatPump method*), 28
build_solph_components() (*oemof.tabular.facades.Link method*), 29
build_solph_components() (*oemof.tabular.facades.Load method*), 29
build_solph_components() (*oemof.tabular.facades.Reservoir method*), 30
build_solph_components() (*oemof.tabular.facades.Storage method*), 32
build_solph_components() (*oemof.tabular.facades.Volatile method*), 33

C

clean() (*in module oemof.tabular.datapackage.processing*), 18
Commodity (*class in oemof.tabular.facades*), 22
Conversion (*class in oemof.tabular.facades*), 23
copy_datapackage() (*in module oemof.tabular.datapackage.processing*), 18

D

deserialize_energy_system() (*in module oemof.tabular.datapackage.reading*), 18
Dispatchable (*class in oemof.tabular.facades*), 24
download_data() (*in module oemof.tabular.datapackage.building*), 16

E

Excess (*class in oemof.tabular.facades*), 25
ExtractionTurbine (*class in oemof.tabular.facades*), 25

F

Facade (*class in oemof.tabular.facades*), 27

G

Generator (*class in oemof.tabular.facades*), 27

H

HeatPump (*class in oemof.tabular.facades*), 27
HSN (*class in oemof.tabular.tools*), 19

I

infer_metadata() (*in module oemof.tabular.datapackage.building*), 16
infer_resources() (*in module oemof.tabular.datapackage.building*), 16
initialize() (*in module oemof.tabular.datapackage.building*), 16
input_filepath() (*in module oemof.tabular.datapackage.building*), 16
intersects() (*in module oemof.tabular.tools.geometry*), 20

L

Link (*class in oemof.tabular.facades*), 28
Load (*class in oemof.tabular.facades*), 29

N

nuts() (*in module oemof.tabular.tools.geometry*), 20

O

oemof.tabular (*module*), 15
oemof.tabular.datapackage (*module*), 15
oemof.tabular.datapackage.aggregation (*module*), 15
oemof.tabular.datapackage.building (*module*), 16
oemof.tabular.datapackage.processing (*module*), 18
oemof.tabular.datapackage.reading (*module*), 18
oemof.tabular.facades (*module*), 21
oemof.tabular.tools (*module*), 19
oemof.tabular.tools.geometry (*module*), 20

P

package_from_resources() (*in module oemof.tabular.datapackage.building*), 16

R

raisestatement() (*in module oemof.tabular.tools*), 19
read_build_config() (*in module oemof.tabular.datapackage.building*), 17
read_elements() (*in module oemof.tabular.datapackage.building*), 17
read_facade() (*in module oemof.tabular.datapackage.reading*), 18
read_geometries() (*in module oemof.tabular.tools.geometry*), 20
read_sequences() (*in module oemof.tabular.datapackage.building*), 17
remap() (*in module oemof.tabular.tools*), 19
reproject() (*in module oemof.tabular.tools.geometry*), 20
Reservoir (*class in oemof.tabular.facades*), 29

S

sequences() (*in module oemof.tabular.datapackage.reading*), 19
Shapes2Shapes() (*in module oemof.tabular.tools.geometry*), 20
Shortage (*class in oemof.tabular.facades*), 30
simplify_poly() (*in module oemof.tabular.tools.geometry*), 20
Storage (*class in oemof.tabular.facades*), 30

T

temporal_clustering() (*in module oemof.tabular.datapackage.aggregation*), 15
temporal_skip() (*in module oemof.tabular.datapackage.aggregation*), 15
timeindex() (*in module oemof.tabular.datapackage.building*), 17
to_dict() (*in module oemof.tabular.datapackage.processing*), 18

U

update() (*oemof.tabular.facades.Facade method*), 27
update_package_descriptor() (*in module oemof.tabular.datapackage.building*), 17

V

Volatile (*class in oemof.tabular.facades*), 32

W

write_elements() (*in module oemof.tabular.datapackage.building*), 17
write_geometries() (*in module oemof.tabular.tools.geometry*), 21
write_sequences() (*in module oemof.tabular.datapackage.building*), 17