eDisGo Documentation

Release 0.0.10

open$_e Go - Team$

Aug 11, 2020

Contents

eDisGo – Optimization of flexibility options and grid expansion for distribution grids based on PyPSA

CHAPTER 1

eDisGo

The python package eDisGo provides a toolbox to analyze distribution grids for grid issues and to evaluate measures responding these. This software lives in the context of the research project open_eGo. It is closely related to the python project Ding0 as this project is currently the single data source for eDisGo providing synthetic grid data for whole Germany.

The toolbox currently includes

- Data import from data sources of the open_eGo project
- Power flow analysis for grid issue identification (enabled by PyPSA)
- Grid reinforcement solving overloading and voltage issues
- Curtailment methodologies
- Battery storage integration

See *Quickstart* for the first steps. A deeper guide is provided in *Usage details*. Methodologies are explained in detail in *Features in detail*. For those of you who want to contribute see *Notes to developers* and the *API* reference.

1.1 LICENSE

CHAPTER 2

Quickstart

2.1 Installation

Install latest eDisGo version through pip. Therefore, we highly recommend using a virtual environment and use its pip.

```
pip3 install edisgo
```

In order to create plots with background maps you additionally need to install the python package contextily.

Consider to install a developer version as detailed in *Notes to developers*.

2.2 Prerequisites

Beyond a running and up-to-date installation of eDisGo you need **grid topology data**. Currently synthetic grid data generated with the python project Ding0 is the only supported data source. You can retrieve data from Zenodo (make sure you choose latest data) or check out the ding0 documentation on how to generate grids yourself.

2.3 A minimum working example

Following you find short examples on how to use eDisGo. Further examples and details are provided in *Usage details*.

All following examples assume you have a ding0 grid topology file named "ding0_grids__42.pkl" in current working directory.

Aside from grid topology data you may eventually need a dataset on future installation of power plants. You may therefore use the scenarios developed in the open_eGo project that are available in the OpenEnergy DataBase (oedb) hosted on the OpenEnergy Platform (OEP). eDisGo provides an interface to the oedb using the package ego.io. ego.io gives you a python SQL-Alchemy representations of the oedb and access to it by using the oedialect, an SQL-Alchemy dialect used by the OEP.

You can run a worst-case scenario as follows:

Using package included command-line script

```
edisgo_run -f ding0_grids__42.pkl -wc
```

Or coding the script yourself with finer control of details

```
from edisgo import EDisGo

# Set up the EDisGo object that will import the grid topology, set up
# feed-in and load time series (here for a worst case analysis)
# and other relevant data
edisgo = EDisGo(ding0_grid="ding0_grids__42.pkl",
                worst_case_analysis='worst-case')

# Import scenario for future generators from the oedb
edisgo.import_generators(generator_scenario='nep2035')

# Conduct grid analysis (non-linear power flow)
edisgo.analyze()

# Do grid reinforcement
edisgo.reinforce()

# Determine costs for each line/transformer that was reinforced
costs = edisgo.network.results.grid_expansion_costs
```

Instead of conducting a worst-case analysis you can also provide specific time series:

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object with your own time series
# (these are dummy time series!)
# timeindex specifies which time steps to consider in power flow
timeindex = pd.date_range('1/1/2011', periods=4, freq='H')
# load time series (scaled by annual demand)
timeseries_load = pd.DataFrame({'residential': [0.0001] * len(timeindex),
                                'commercial': [0.0002] * len(timeindex),
                                'industrial': [0.0015] * len(timeindex),
                                'agricultural': [0.00005] * len(timeindex)},
                               index=timeindex)
# feed-in time series of fluctuating generators (scaled by nominal power)
timeseries_generation_fluctuating = \
    pd.DataFrame({'solar': [0.2] * len(timeindex),
                  'wind': [0.3] * len(timeindex)},
                 index=timeindex)
# feed-in time series of dispatchable generators (scaled by nominal power)
timeseries_generation_dispatchable = \
    pd.DataFrame({'biomass': [1] * len(timeindex),
                  'coal': [1] * len(timeindex),
                  'other': [1] * len(timeindex)},
                 index=timeindex)

# Set up the EDisGo object with your own time series and generator scenario
# NEP2035
edisgo = EDisGo(
```

(continues on next page)

(continued from previous page)

```
    ding0_grid="ding0_grids__42.pkl",
    generator_scenario='nep2035',
    timeseries_load=timeseries_load,
    timeseries_generation_fluctuating=timeseries_generation_fluctuating,
    timeseries_generation_dispatchable=timeseries_generation_dispatchable,
    timeindex=timeindex)

# Do grid reinforcement
edisgo.reinforce()

# Determine cost for each line/transformer that was reinforced
costs = edisgo.network.results.grid_expansion_costs
```

Time series for load and fluctuating generators can also be automatically generated using the provided API for the oemof demandlib and the OpenEnergy DataBase:

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object using the OpenEnergy DataBase and the oemof
# demandlib to set up time series for loads and fluctuating generators
# (time series for dispatchable generators need to be provided)
timeindex = pd.date_range('1/1/2011', periods=4, freq='H')
timeseries_generation_dispatchable = \
    pd.DataFrame({'other': [1] * len(timeindex)},
                 index=timeindex)
edisgo = EDisGo(
    ding0_grid="ding0_grids__42.pkl",
    generator_scenario='ego100',
    timeseries_load='demandlib',
    timeseries_generation_fluctuating='oedb',
    timeseries_generation_dispatchable=timeseries_generation_dispatchable,
    timeindex=timeindex)

# Do grid reinforcement
edisgo.reinforce()

# Determine cost for each line/transformer that was reinforced
costs = edisgo.network.results.grid_expansion_costs
```

2.4 Parallelization

Try `run_edisgo_pool_flexible()` for parallelization of your custom function.

CHAPTER 3

Usage details

As eDisGo is designed to serve as a toolbox, it provides several methods to analyze distribution grids for grid issues and to evaluate measures responding these. We provide two examples, an `example script` and `jupyter notebook`.

Further, we discuss how different features can be used in detail below.

3.1 The fundamental data structure

It's worth to understand how the fundamental data structure of eDisGo is designed in order to make use of its entire features.

The class *EDisGo* serves as the top-level API for setting up your scenario, invocation of data import, analysis of hosting capacity, grid reinforcement and flexibility measures.

If you want to set up a scenario to do a worst-case analysis of a ding0 grid (see *Prerequisites*) you simply have to provide a grid and set the `worst_case_analysis` parameter. The following example assumes you have a file of a ding0 grid named "ding0_grids__42.pkl" in current working directory.

```
from edisgo import EDisGo

edisgo = EDisGo(ding0_grid="ding0_grids__42.pkl",
                worst_case_analysis='worst-case-feedin')
```

You can also provide your own time series for load and feed-in for the analysis.

```
import pandas as pd

# set up load and feed-in time series
timeindex = pd.date_range('1/1/1970', periods=3, freq='H')
feedin_renewables = pd.DataFrame(data={'solar': [0.1, 0.2, 0.3],
                                       'wind': [0.3, 0.15, 0.15]},
                                 index=timeindex)
feedin_dispatchable = pd.DataFrame(data={'coal': [0.5, 0.1, 0.5],
```

(continues on next page)

(continued from previous page)

```
                                        'other': [0.3, 0.1, 0.7]},
                                    index=timeindex)
load = pd.DataFrame(data={'residential': [0.00001, 0.00002, 0.00002],
                          'retail': [0.00005, 0.00005, 0.00005],
                          'industrial': [0.00002, 0.00003, 0.00002],
                          'agricultural': [0.00001, 0.000015, 0.00001]},
                    index=timeindex)

edisgo = EDisGo(ding0_grid="ding0_grids__42.pkl",
                timeseries_generation_fluctuating=feedin_renewables,
                timeseries_generation_dispatchable=feedin_dispatchable,
                timeseries_load=load)
```

EDisGo also offers methods to generate load time series and feed-in time series for fluctuating generators (see last *A minimum working example*). See *EDisGo* for more information on which options to choose from and what other data can be provided.

All data is stored in the class *Network*. The network class serves as an overall data container in eDisGo holding the grid data for the MVGrid and LVGrid s, *Config* data, *Results*, Timeseries, etc. It is linked from multiple locations and provides hierarchical access to all data. Network itself can be accessed via the EDisGo object.

```
# Access to Network data container object
edisgo.network
```

The grid data and results can e.g. be accessed via

```
# MV grid instance
edisgo.network.mv_grid

# List of LV grid instances
edisgo.network.mv_grid.lv_grids

# Results of network analysis
edisgo.network.results

# MV grid generators
edisgo.network.mv_grid.generators
```

The grid topology is represented by separate undirected graphs for the MV grid and each of the LV grids. The Graph is subclassed from networkx.Graph and extended by extra-functionality. Lines represent edges in the graph. Other equipment is represented by a node.

3.2 Identify grid issues

As detailed in *A minimum working example*, once you set up your scenario by instantiating an *EDisGo* object, you are ready for an analysis of grid issues (line overloading or voltage band violations) respectively the hosting capacity of the grid by *analyze()*:

```
# Do non-linear power flow analysis for MV and LV grid
edisgo.analyze()
```

The analyze function conducts a non-linear power flow using PyPSA.

The range of time analyzed by the power flow analysis is by default defined by the timeindex given to the EDisGo API but can also be specified by providing the parameter *timesteps* to analyze.

3.3 Grid extension

Grid extension can be invoked by *reinforce()*:

```
# Reinforce grid due to overloading and overvoltage issues
edisgo.reinforce()
```

You can further specify e.g. if to conduct a combined analysis for MV and LV (regarding allowed voltage deviations) or if to only calculate grid expansion needs without changing the topology of the graph. See *reinforce()* for more information.

Costs for the grid extension measures can be obtained as follows:

```
# Get costs of grid extension
costs = edisgo.network.results.grid_expansion_costs
```

Further information on the grid reinforcement methodology can be found in section *Grid expansion*.

3.4 Battery storages

Battery storages can be integrated into the grid as alternative to classical grid extension. A battery in eDisGo is represented by the class *Storage*. Using the method *integrate_storage()* provides a high-level interface to define the position, size and storage operation, based on user input and predefined rules. A limited set of storage integration rules are implemented. See *StorageControl* for available storage integration strategies.

Here is a small example on how to integrate a storage:

```
# define storage parameters
storage_parameters = {'nominal_power': 200}

# add storage instance to the grid
edisgo.integrate_storage(position='hvmv_substation_busbar',
                         timeseries='fifty-fifty',
                         parameters=storage_parameters)
```

Further information on the storage integration methodology 'distribute_storages_mv' can be found in section *Storage integration*.

3.5 Curtailment

The curtailment function is used to spatially distribute the power that is to be curtailed. There are currently two options for doing this distribution:

- ***feedin-proportional*** Distributes the curtailed power to all the fluctuating generators depending on their weather-dependent availability.
- ***voltage-based*** Distributes the curtailed power depending on the exceedance of the allowed voltage deviation at the nodes of the fluctuating generators.

The input to the curtailment function can be modified to curtail certain technologies or technologies by the weather cell they are in. Opposite to the load and feed-in time series curtailment time series need to be given in kW. Following are examples of the different options of how to specify curtailment requirements:

```
timeindex = pd.date_range('1/1/1970', periods=3, freq='H')

# curtailment is allocated to all solar and wind generators
curtailment = pd.Series(data=[0.0, 5000.0, 3000.0],
                        index=timeindex)

# curtailment is allocated by generator type
curtailment = pd.DataFrame(data={'wind': [0.0, 5000.0, 3000.0],
                                 'solar': [5500.0, 5400.0, 3200.0]},
                           index=timeindex)

# curtailment is allocated by generator type and weather cell
curtailment = pd.DataFrame(data={('wind', 1): [0.0, 5000.0, 3000.0],
                                 ('wind', 2): [100.0, 2000.0, 300.0],
                                 ('solar', 1): [500.0, 5000.0, 300.0]},
                           index=timeindex)
```

Set curtailment by calling the method *curtail()*:

```
edisgo.curtail(curtailment_methodology='feedin-proportional',
               timeseries_curtailment=curtailment)
```

or with

```
edisgo.curtail(curtailment_methodology='voltage-based',
               timeseries_curtailment=curtailment)
```

3.6 Plots

EDisGo provides a bunch of predefined plots to e.g. plot the MV grid topology, and line loading and node voltages in the MV grid or as a histogram.

```
# plot MV grid topology on a map
edisgo.plot_mv_grid_topology()

# plot grid expansion costs for lines in the MV grid and stations on a map
edisgo.plot_mv_grid_expansion_costs()

# plot voltage histogram
edisgo.histogram_voltage()
```

See `EDisGoRemiport` class for more plots and plotting options.

3.7 Results

Results such as voltage levels and line loading from the power flow analysis and grid extension costs are provided through the *Results* class and can be accessed the following way:

```
edisgo.network.results
```

Get voltage levels at nodes from *v_res()* and line loading from *s_res()* or *i_res*. *equipment_changes* holds details about measures performed during grid extension. Associated costs are determined by

grid_expansion_costs. Flexibility measures may not entirely resolve all issues. These unresolved issues are listed in *unresolved_issues*.

Results can be saved to csv files with:

```
edisgo.network.results.save('path/to/results/directory/')
```

To reimport saved results you can use the `EDisGoRemiport` class. After instantiating the class you can access results and plots the same way as you would with the EDisGo class.

```
# import EDisGoReimport class
from edisgo import EDisGoReimport

# instantiate EDisGoReimport class
edisgo = EDisGoReimport('path/to/results/directory/')

# access results
edisgo.network.results.grid_expansion_costs

# plot MV grid topology on a map
edisgo.plot_mv_grid_topology()
```

CHAPTER 4

Features in detail

4.1 Power flow analysis

In order to analyse voltages and line loadings a non-linear power flow analysis (PF) is conducted. All loads and generators are modelled as PQ nodes; the slack is modelled as a PV node with a set voltage of 1,p.u. and positioned at the substation's secondary side.

4.2 Grid expansion

4.2.1 General methodology

The grid expansion methodology is conducted in *reinforce_grid()*.

The order grid expansion measures are conducted is as follows:

- Reinforce stations and lines due to overloading issues
- Reinforce lines in MV grid due to voltage issues
- Reinforce distribution substations due to voltage issues
- Reinforce lines in LV grid due to voltage issues
- Reinforce stations and lines due to overloading issues

Reinforcement of stations and lines due to overloading issues is performed twice, once in the beginning and again after fixing voltage issues, as the changed power flows after reinforcing the grid may lead to new overloading issues. How voltage and overloading issues are identified and solved is shown in figure *Grid expansion measures* and further explained in the following sections.

reinforce_grid() offers a few additional options. It is e.g. possible to conduct grid reinforcement measures on a copy of the graph so that the original grid topology is not changed. It is also possible to only identify necessary reinforcement measures for two worst-case snapshots in order to save computing time and to set combined or separate allowed voltage deviation limits for MV and LV. See documentation of *reinforce_grid()* for more information.

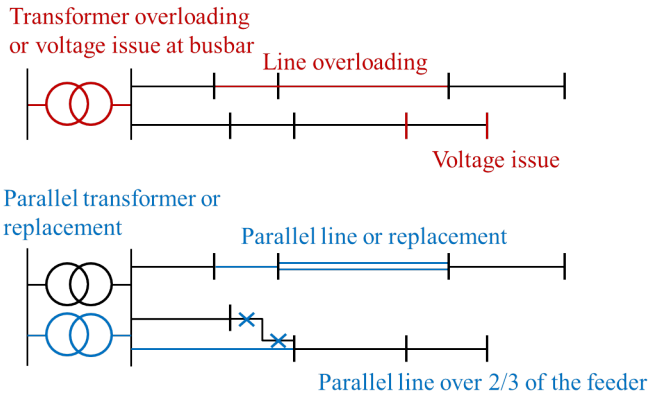

Fig. 4.1: Grid expansion measures

4.2.2 Identification of overloading and voltage issues

Identification of overloading and voltage issues is conducted in `check_tech_constraints`.

Voltage issues are determined based on allowed voltage deviations set in the config file *config_grid_expansion* in section *grid_expansion_allowed_voltage_deviations*. It is possible to set one allowed voltage deviation that is used for MV and LV or define separate allowed voltage deviations. Which allowed voltage deviation is used is defined through the parameter *combined_analysis* of `reinforce_grid()`. By default *combined_analysis* is set to false, resulting in separate voltage limits for MV and LV, as a combined limit may currently lead to problems if voltage deviation in MV grid is already close to the allowed limit, in which case the remaining allowed voltage deviation in the LV grids is close to zero.

Overloading is determined based on allowed load factors that are also defined in the config file *config_grid_expansion* in section *grid_expansion_load_factors*.

Allowed voltage deviations as well as load factors are in most cases different for load and feed-in case. Load and feed-in case are commonly used worst-cases for grid expansion analyses. Load case defines a situation where all loads in the grid have a high demand while feed-in by generators is low or zero. In this case power is flowing from the high-voltage grid to the distribution grid. In the feed-in case there is a high generator feed-in and a small energy demand leading to a reversed power flow. Load and generation assumptions for the two worst-cases are definded in the config file *config_timeseries* in section *worst_case_scale_factor* (scale factors describe actual power to nominal power ratio of generators and loads).

When conducting grid reinforcement based on given time series instead of worst-case assumptions, load and feed-in case also need to be definded to determine allowed voltage deviations and load factors. Therefore, the two cases are identified based on the generation and load time series of all loads and generators in the grid and defined as follows:

- Load case: positive ($\sum load$ - $\sum generation$)
- Feed-in case: negative ($\sum load$ - $\sum generation$) -> reverse power flow at HV/MV substation

Grid losses are not taken into account. See `assign_load_feedin_case()` for more details and implementation.

Check line load

Exceedance of allowed line load of MV and LV lines is checked in `mv_line_load()` and `lv_line_load()`, respectively. The functions use the given load factor and the maximum allowed current given by the manufacturer (see *I_max_th* in tables *LV cables*, *MV cables* and *MV overhead lines*) to calculate the allowed line load of each LV and MV line. If the line load calculated in the power

flow analysis exceeds the allowed line load, the line is reinforced (see *Reinforce lines due to overloading issues*).

Check station load

Exceedance of allowed station load of HV/MV and MV/LV stations is checked in `hv_mv_station_load()` and `mv_lv_station_load()`, respectively. The functions use the given load factor and the maximum allowed apparent power given by the manufacturer (see *S_nom* in tables *LV transformers*, and *MV transformers*) to calculate the allowed apparent power of the stations. If the apparent power calculated in the power flow analysis exceeds the allowed apparent power the station is reinforced (see *Reinforce stations due to overloading issues*).

Check line and station voltage deviation

Compliance with allowed voltage deviation limits in MV and LV grids is checked in `mv_voltage_deviation()` and `lv_voltage_deviation()`, respectively. The functions check if the voltage deviation at a node calculated in the power flow analysis exceeds the allowed voltage deviation. If it does, the line is reinforced (see *Reinforce MV/LV stations due to voltage issues* or *Reinforce lines due to voltage*).

4.2.3 Grid expansion measures

Reinforcement measures are conducted in `reinforce_measures`. Whereas overloading issues can usually be solved in one step, except for some cases where the lowered grid impedance through reinforcement measures leads to new issues, voltage issues can only be solved iteratively. This means that after each reinforcement step a power flow analysis is conducted and the voltage rechecked. An upper limit for how many iteration steps should be performed is set in order to avoid endless iteration. By default it is set to 10 but can be changed using the parameter *max_while_iterations* of `reinforce_grid()`.

Reinforce lines due to overloading issues

Line reinforcement due to overloading is conducted in `reinforce_branches_overloading()`. In a first step a parallel line of the same line type is installed. If this does not solve the overloading issue as many parallel standard lines as needed are installed.

Reinforce stations due to overloading issues

Reinforcement of HV/MV and MV/LV stations due to overloading is conducted in `extend_substation_overloading()` and `extend_distribution_substation_overloading()`, respectively. In a first step a parallel transformer of the same type as the existing transformer is installed. If there is more than one transformer in the station the smallest transformer that will solve the overloading issue is used. If this does not solve the overloading issue as many parallel standard transformers as needed are installed.

Reinforce MV/LV stations due to voltage issues

Reinforcement of MV/LV stations due to voltage issues is conducted in `extend_distribution_substation_overvoltage()`. To solve voltage issues, a parallel standard transformer is installed.

After each station with voltage issues is reinforced, a power flow analysis is conducted and the voltage rechecked. If there are still voltage issues the process of installing a parallel standard transformer and conducting a power flow analysis is repeated until voltage issues are solved or until the maximum number of allowed iterations is reached.

Reinforce lines due to voltage

Reinforcement of lines due to voltage issues is conducted in `reinforce_branches_overvoltage()`. In the case of several voltage issues the path to the node with the highest voltage deviation is reinforced first. Therefore, the line between the secondary side of the station and the node with the highest voltage deviation is disconnected at a distribution substation after 2/3 of the path length. If there is no distribution substation where the line can be disconnected, the node is directly connected to the busbar. If the node is already directly connected to the busbar a parallel standard line is installed.

Only one voltage problem for each feeder is considered at a time since each measure effects the voltage of each node in that feeder.

After each feeder with voltage problems has been considered, a power flow analysis is conducted and the voltage rechecked. The process of solving voltage issues is repeated until voltage issues are solved or until the maximum number of allowed iterations is reached.

4.2.4 Grid expansion costs

Total grid expansion costs are the sum of costs for each added transformer and line. Costs for lines and transformers are only distinguished by the voltage level they are installed in and not by the different types. In the case of lines it is further taken into account wether the line is installed in a rural or an urban area, whereas rural areas are areas with a population density smaller or equal to 500 people per km^2 and urban areas are defined as areas with a population density higher than 500 people per km^2 [DENA]. The population density is calculated by the population and area of the grid district the line is in (See `Grid`).

Costs for lines of aggregated loads and generators are not considered in the costs calculation since grids of aggregated areas are not modeled but aggregated loads and generators are directly connected to the MV busbar.

4.3 Curtailment

eDisGo right now provides two curtailment methodologies called 'feedin-proportional' and 'voltage-based', that are implemented in `curtailment`. Both methods are intended to take a given curtailment target obtained from an optimization of the EHV and HV grids using eTraGo and allocate it to the generation units in the grids. Curtailment targets can be specified for all wind and solar generators, by generator type (solar or wind) or by generator type in a given weather cell. It is also possible to curtail specific generators internally, though a user friendly implementation is still in the works.

4.3.1 'feedin-proportional'

The 'feedin-proportional' curtailment is implemented in `feedin_proportional()`. The curtailment that has to be met in each time step is allocated equally to all generators depending on their share of total feed-in in that time step.

$$c_{g,t} = \frac{a_{g,t}}{\sum\limits_{g \in gens} a_{g,t}} \times c_{target,t} \quad \forall t \in timesteps$$

where $c_{g,t}$ is the curtailed power of generator g in timestep t, $a_{g,t}$ is the weather-dependent availability of generator g in timestep t and $c_{target,t}$ is the given curtailment target (power) for timestep t to be allocated to the generators.

4.3.2 'voltage-based'

The 'voltage-based' curtailment is implemented in `voltage_based()`. The curtailment that has to be met in each time step is allocated to all generators depending on the exceedance of the allowed voltage deviation at the nodes of the generators. The higher the exceedance, the higher the curtailment.

The optional parameter *voltage_threshold* specifies the threshold for the exceedance of the allowed voltage deviation above which a generator is curtailed. By default it is set to zero, meaning that all generators at nodes with voltage deviations that exceed the allowed voltage deviation are curtailed. Generators at nodes where the allowed voltage deviation is not exceeded are not curtailed. In the case that the required curtailment exceeds the weather-dependent availability of all generators with voltage deviations above the specified threshold, the voltage threshold is lowered in steps of 0.01 p.u. until the curtailment target can be met.

Above the threshold, the curtailment is proportional to the exceedance of the allowed voltage deviation.

$$\frac{c_{g,t}}{a_{g,t}} = n \cdot (V_{g,t} - V_{threshold,g,t}) + offset$$

where $c_{g,t}$ is the curtailed power of generator g in timestep t, $a_{g,t}$ is the weather-dependent availability of generator g in timestep t, $V_{g,t}$ is the voltage at generator g in timestep t and $V_{threshold,g,t}$ is the voltage threshold for generator g in timestep t. $V_{threshold,g,t}$ is calculated as follows:

$$V_{threshold,g,t} = V_{g_{station},t} + \Delta V_{g_{allowed}} + \Delta V_{offset,t}$$

where $V_{g_{station},t}$ is the voltage at the station's secondary side, $\Delta V_{g_{allowed}}$ is the allowed voltage deviation in the reverse power flow and $\Delta V_{offset,t}$ is the exceedance of the allowed voltage deviation above which generators are curtailed.

n and $offset$ in the equation above are slope and y-intercept of a linear relation between the curtailment and the exceedance of the allowed voltage deviation. They are calculated by solving the following linear problem that penalizes the offset using the python package pyomo:

$$min\left(\sum_t offset_t\right)$$

$$s.t. \sum_g c_{g,t} = c_{target,t} \; \forall g \in (solar, wind)$$

$$c_{g,t} \leq a_{g,t} \forall g \in (solar, wind), t$$

where $c_{target,t}$ is the given curtailment target (power) for timestep t to be allocated to the generators.

4.4 Storage integration

Besides the possibility to connect a storage with a given operation to any node in the grid, eDisGo provides a methodology that takes a given storage capacity and allocates it to multiple smaller storages such that it reduces line overloading and voltage deviations. The methodology is implemented in `one_storage_per_feeder()`. As the above described curtailment allocation methodologies it is intended to be used in combination with eTraGo where storage capacity and operation is optimized.

For each feeder with load or voltage issues it is checked if integrating a storage will reduce peaks in the feeder, starting with the feeder with the highest theoretical grid expansion costs. A heuristic approach is used to estimate storage sizing and siting while storage operation is carried over from the given storage operation.

A more thorough documentation will follow soon.

4.5 References

CHAPTER 5

Notes to developers

5.1 Installation

Clone repository from GitHub and install in developer mode:

```
pip3 install -e <path-to-repo>
```

5.2 Code style

- **Documentation of '@property' functions: Put documentation of getter and** setter both in Docstring of getter, see on Stackoverflow
- **Order of public/private/protected methods, property decorators, etc. in a** class: TBD

5.3 Documentation

Build the docs locally by first setting up the sphinx environment with (executed from top-level folder)

```
sphinx-apidoc -f -o doc/api edisgo
```

And then you build the html docs on your computer with

```
sphinx-build -E -a doc/ doc/_html
```

CHAPTER 6

Definition and units

6.1 Sign Convention

Generators and Loads in an AC power system can behave either like an inductor or a capacitor. Mathematically, this has two different sign conventions, either from the generator perspective or from the load perspective. This is defined by the direction of power flow from the component.

Both sign conventions are used in eDisGo depending upon the components being defined, similar to pypsa.

6.1.1 Generator Sign Convention

While defining time series for *Generator*, *GeneratorFluctuating*, and *Storage*, the generator sign convention is used.

6.1.2 Load Sign Convention

The time series for *Load* is defined using the load sign convention.

6.2 Reactive Power Sign Convention

Generators and Loads in an AC power system can behave either like an inductor or a capacitor. Mathematically, this has two different sign conventions, either from the generator perspective or from the load perspective.

Both sign conventions are used in eDisGo, similar to pypsa. While defining time series for *Generator*, *GeneratorFluctuating*, and *Storage*, the generator sign convention is used. This means that when the reactive power (Q) is positive, the component shows capacitive behaviour and when the reactive power (Q) is negative, the component shows inductive behaviour.

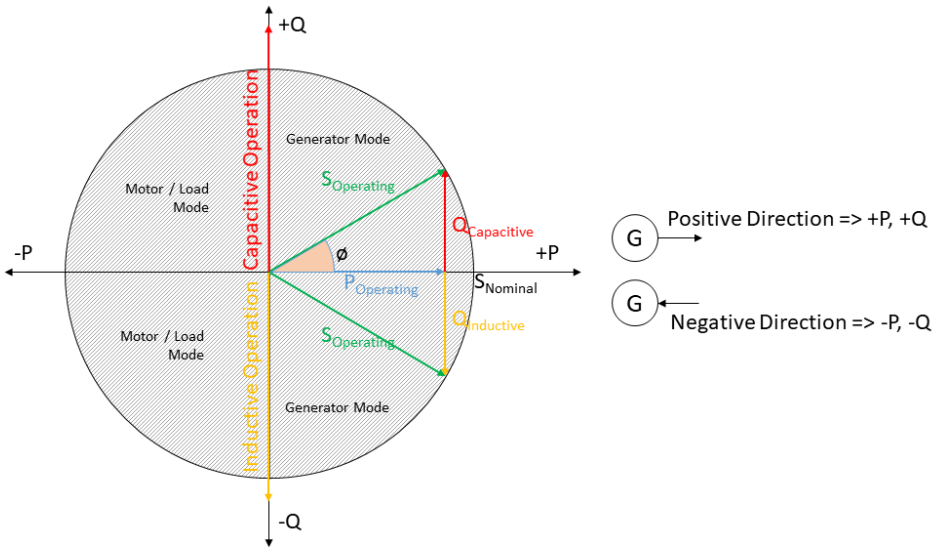

Fig. 6.1: Generator sign convention in detail

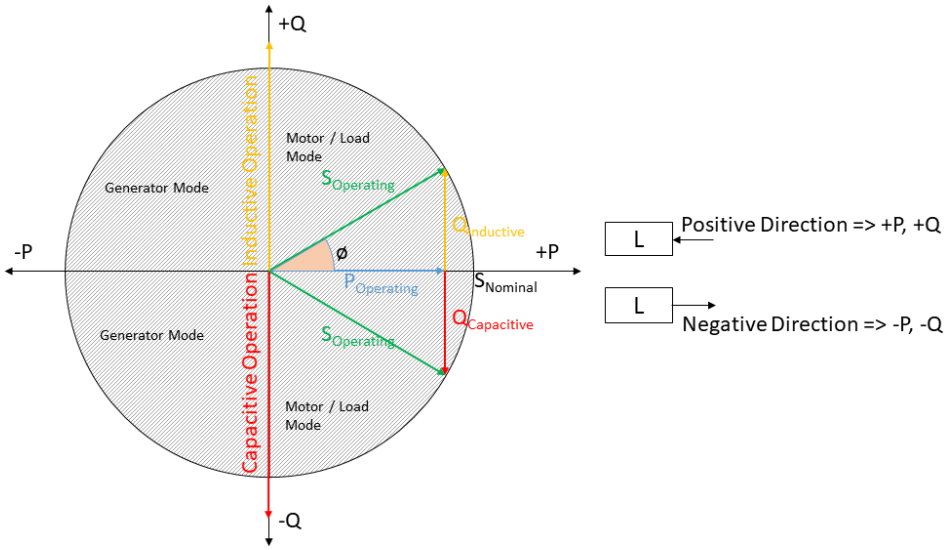

Fig. 6.2: Load sign convention in detail

The time series for *Load* is defined using the load sign convention. This means that when the reactive power (Q) is positive, the component shows inductive behaviour and when the reactive power (Q) is negative, the component shows capacitive behaviour. This is the direct opposite of the generator sign convention.

6.3 Units

Table 6.1: List of variables and units

Variable	Symbol	Unit	Comment
Current	I	A	
Length	l	km	
Active Power	P	kW	In PyPSA representation (*pypsa*) MW are used
Reactive Power	Q	kvar	In PyPSA representation (*pypsa*) MVar are used
Apparent Power	S	kVA	In PyPSA representation (*pypsa*) MVA are used
Resistance	R	Ohm or Ohm/km	Ohm/km applies to lines
Reactance	X	Ohm or Ohm/km	Ohm/km applies to lines
Voltage	V	kV	
Inductance	L	mH/km	
Capacitance	C	µF/km	
Costs	•	kEUR	

CHAPTER 7

Default configuration data

Following you find the default configuration files.

7.1 config_db_tables

The config file `config_db_tables.cfg` holds data about which database connection to use from your saved database connections and which dataprocessing version.

```
# This file is part of eDisGo, a python package for distribution grid
# analysis and optimization.
#
# It is developed in the project open_eGo: https://openegoproject.wordpress.com
#
# eDisGo lives on github: https://github.com/openego/edisgo/
# The documentation is available on RTD: http://edisgo.readthedocs.io

[data_source]

oedb_data_source = versioned

[model_draft]

conv_generators_prefix = t_ego_supply_conv_powerplant_
conv_generators_suffix = _mview
re_generators_prefix = t_ego_supply_res_powerplant_
re_generators_suffix = _mview
res_feedin_data = EgoRenewableFeedin
load_data = EgoDemandHvmvDemand
load_areas = EgoDemandLoadarea

#conv_generators_nep2035 = t_ego_supply_conv_powerplant_nep2035_mview
#conv_generators_ego100 = ego_supply_conv_powerplant_ego100_mview
#re_generators_nep2035 = t_ego_supply_res_powerplant_nep2035_mview
```

(continues on next page)

(continued from previous page)

```
#re_generators_ego100 = t_ego_supply_res_powerplant_ego100_mview

[versioned]

conv_generators_prefix = t_ego_dp_conv_powerplant_
conv_generators_suffix = _mview
re_generators_prefix = t_ego_dp_res_powerplant_
re_generators_suffix = _mview
res_feedin_data = EgoRenewableFeedin
load_data = EgoDemandHvmvDemand
load_areas = EgoDemandLoadarea

version = v0.4.5
```

7.2 config_grid_expansion

The config file `config_grid_expansion.cfg` holds data mainly needed to determine grid expansion needs and costs - these are standard equipment to use in grid expansion and its costs, as well as allowed voltage deviations and line load factors.

```
# This file is part of eDisGo, a python package for distribution grid
# analysis and optimization.
#
# It is developed in the project open_eGo: https://openegoproject.wordpress.com
#
# eDisGo lives on github: https://github.com/openego/edisgo/
# The documentation is available on RTD: http://edisgo.readthedocs.io

[grid_expansion_standard_equipment]

# standard equipment
# ==================
# Standard equipment for grid expansion measures. Source: Rehtanz et. al.:
↪"Verteilnetzstudie für das Land Baden-Württemberg", 2017.
hv_mv_transformer = 40 MVA
mv_lv_transformer = 630 kVA
mv_line = NA2XS2Y 3x1x185 RM/25
lv_line = NAYY 4x1x150

[grid_expansion_allowed_voltage_deviations]

# allowed voltage deviations
# ==========================
# COMBINED MV+LV
# --------------
# hv_mv_trafo_offset:
#     offset which is set at HV-MV station
#     (pos. if op. voltage is increased, neg. if decreased)
hv_mv_trafo_offset = 0.0

# hv_mv_trafo_control_deviation:
#     control deviation of HV-MV station
#     (always pos. in config; pos. or neg. usage depending on case in edisgo)
hv_mv_trafo_control_deviation = 0.0
```

(continues on next page)

(continued from previous page)

```
# mv_lv_max_v_deviation:
#     max. allowed voltage deviation according to DIN EN 50160
#     caution: offset and control deviation at HV-MV station must be considered in␣
↪calculations!
mv_lv_feedin_case_max_v_deviation = 0.1
mv_lv_load_case_max_v_deviation = 0.1

# MV ONLY
# -------
# mv_load_case_max_v_deviation:
#     max. allowed voltage deviation in MV grids (load case)
mv_load_case_max_v_deviation = 0.015

# mv_feedin_case_max_v_deviation:
#     max. allowed voltage deviation in MV grids (feedin case)
#     according to BDEW
mv_feedin_case_max_v_deviation = 0.05

# LV ONLY
# -------
# max. allowed voltage deviation in LV grids (load case)
lv_load_case_max_v_deviation = 0.065

# max. allowed voltage deviation in LV grids (feedin case)
#     according to VDE-AR-N 4105
lv_feedin_case_max_v_deviation = 0.035

# max. allowed voltage deviation in MV/LV stations (load case)
mv_lv_station_load_case_max_v_deviation = 0.02

# max. allowed voltage deviation in MV/LV stations (feedin case)
mv_lv_station_feedin_case_max_v_deviation = 0.015

[grid_expansion_load_factors]

# load factors
# ============
# Source: Rehtanz et. al.: "Verteilnetzstudie für das Land Baden-Württemberg", 2017.
mv_load_case_transformer = 0.5
mv_load_case_line = 0.5
mv_feedin_case_transformer = 1.0
mv_feedin_case_line = 1.0

lv_load_case_transformer = 1.0
lv_load_case_line = 1.0
lv_feedin_case_transformer = 1.0
lv_feedin_case_line = 1.0

# costs
# ============

[costs_cables]

# costs in kEUR/km
# costs for cables without earthwork are taken from [1] (costs for standard
# cables are used here as representative since they have average costs), costs
```

(continues on next page)

(continued from previous page)

```
# including earthwork are taken from [2]
# [1] https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Energie/
↪Unternehmen_Institutionen/Netzentgelte/Anreizregulierung/GA_
↪AnalytischeKostenmodelle.pdf?__blob=publicationFile&v=1
# [2] https://shop.dena.de/fileadmin/denashop/media/Downloads_Dateien/esd/9100_dena-
↪Verteilnetzstudie_Abschlussbericht.pdf
# costs including earthwork costs depend on population density according to [2]
# here "rural" corresponds to a population density of <= 500 people/km²
# and "urban" corresponds to a population density of > 500 people/km²
lv_cable = 9
lv_cable_incl_earthwork_rural = 60
lv_cable_incl_earthwork_urban = 100
mv_cable = 20
mv_cable_incl_earthwork_rural = 80
mv_cable_incl_earthwork_urban = 140

[costs_transformers]

# costs in kEUR, source: DENA Verteilnetzstudie
lv = 10
mv = 1000
```

7.3 config_timeseries

The config file `config_timeseries.cfg` holds data to define the two worst-case scenarions heavy load flow ('load case') and reverse power flow ('feed-in case') used in conventional grid expansion planning, power factors and modes (inductive or capacitative) to generate reactive power time series, as well as configurations of the demandlib in case load time series are generated using the oemof demandlib.

```
# This file is part of eDisGo, a python package for distribution grid
# analysis and optimization.
#
# It is developed in the project open_eGo: https://openegoproject.wordpress.com
#
# eDisGo lives on github: https://github.com/openego/edisgo/
# The documentation is available on RTD: http://edisgo.readthedocs.io

# This file contains relevant data to generate load and feed-in time series.
# Peakload consumption ratios and scale factors are used in worst-case scenarios.
# Power factors are used to generate reactive power time series.

[peakload_consumption_ratio]

# peakload consumption ratios
# ===========================
# ratios of peak load to annual consumption per sector based on BDEW synthetic
# load profiles; used only in worst case analyses
residential = 0.00021372
retail = 0.0002404
industrial = 0.000132
agricultural = 0.00024036

[worst_case_scale_factor]
```

(continues on next page)

(continued from previous page)

```
# scale factors
# ============================
# scale factors describe actual power to nominal power ratio of generators and loads␣
↪in worst-case scenarios
# following values provided by "dena-Verteilnetzstudie. Ausbau- und
# Innovationsbedarf der Stromverteilnetze in Deutschland bis 2030", .p. 98

mv_feedin_case_load = 0.15
lv_feedin_case_load = 0.1
mv_load_case_load = 1.0
lv_load_case_load = 1.0

feedin_case_feedin_pv = 0.85
feedin_case_feedin_other = 1
load_case_feedin_pv = 0
load_case_feedin_other = 0

[reactive_power_factor]

# power factors
# ============================
# power factors used to generate reactive power time series for loads and generators

mv_gen = 0.9
mv_load = 0.9
mv_storage = 0.9
lv_gen = 0.95
lv_load = 0.95
lv_storage = 0.95

[reactive_power_mode]

# power factor modes
# ============================
# power factor modes used to generate reactive power time series for loads and␣
↪generators

mv_gen = inductive
mv_load = inductive
mv_storage = inductive
lv_gen = inductive
lv_load = inductive
lv_storage = inductive

[demandlib]

# demandlib data
# ============================
# data used in the demandlib to generate industrial load profile
# see IndustrialProfile in https://github.com/oemof/demandlib/blob/master/demandlib/
↪particular_profiles.py
# for further information

# scaling factors for night and day of weekdays and weekend days
week_day = 0.8
week_night = 0.6
weekend_day = 0.6
```

(continues on next page)

(continued from previous page)

```
weekend_night = 0.6
# tuple specifying the beginning/end of a workday (e.g. 18:00)
day_start = 6:00
day_end = 22:00
```

7.4 config_grid

The config file `config_grid.cfg` holds data to specify parameters used when connecting new generators to the grid and where to position disconnecting points.

```
# This file is part of eDisGo, a python package for distribution grid
# analysis and optimization.
#
# It is developed in the project open_eGo: https://openegoproject.wordpress.com
#
# eDisGo lives on github: https://github.com/openego/edisgo/
# The documentation is available on RTD: http://edisgo.readthedocs.io

# Config file to specify parameters used when connecting new generators to the grid␣
↪and
# where to position disconnecting points.

[geo]

# WGS84: 4326
srid = 4326

[grid_connection]

# branch_detour_factor:
#     normally, lines do not go straight from A to B due to obstacles etc. Therefore,␣
↪a detour factor is used.
#     unit: -
branch_detour_factor = 1.3

# conn_buffer_radius:
#     radius used to find connection targets
#     unit: m
conn_buffer_radius = 2000

# conn_buffer_radius_inc:
#     radius which is incrementally added to connect_buffer_radius as long as no␣
↪target is found
#     unit: m
conn_buffer_radius_inc = 1000

# conn_diff_tolerance:
```

(continues on next page)

(continued from previous page)

```
#     threshold which is used to determine if 2 objects are on the same position
#     unit: -
conn_diff_tolerance = 0.0001

random_seed = 111344501344111


[disconnecting_point]

# Positioning of disconnecting points: Can be position at location of most
# balanced load or generation. Choose load, generation, loadgen
position = load
```

CHAPTER 8

Equipment data

The following tables hold all data of cables, lines and transformers used.

Table 8.1: LV cables

name	U_n	I_max_th	R_per_km	L_per_km
#-	V	A	ohm/km	mH/km
NAYY 4x1x300	400	419	0.1	0.279
NAYY 4x1x240	400	364	0.125	0.254
NAYY 4x1x185	400	313	0.164	0.256
NAYY 4x1x150	400	275	0.206	0.256
NAYY 4x1x120	400	245	0.253	0.256
NAYY 4x1x95	400	215	0.320	0.261
NAYY 4x1x50	400	144	0.449	0.270
NAYY 4x1x35	400	123	0.868	0.271

Table 8.2: MV cables

name	U_n	I_max_th	R_per_km	L_per_km	C_per_km
#-	kV	A	ohm/km	mH/km	uF/km
NA2XS2Y 3x1x185 RM/25	10	357	0.164	0.38	0.41
NA2XS2Y 3x1x240 RM/25	10	417	0.125	0.36	0.47
NA2XS2Y 3x1x300 RM/25	10	466	0.1	0.35	0.495
NA2XS2Y 3x1x400 RM/35	10	535	0.078	0.34	0.57
NA2XS2Y 3x1x500 RM/35	10	609	0.061	0.32	0.63
NA2XS2Y 3x1x150 RE/25	20	319	0.206	0.4011	0.24
NA2XS2Y 3x1x240	20	417	0.13	0.3597	0.304
NA2XS(FL)2Y 3x1x300 RM/25	20	476	0.1	0.37	0.25
NA2XS(FL)2Y 3x1x400 RM/35	20	525	0.078	0.36	0.27
NA2XS(FL)2Y 3x1x500 RM/35	20	598	0.06	0.34	0.3

Table 8.3: MV overhead lines

name	U_n	I_max_th	R_per_km	L_per_km	C_per_km
#-	kV	A	ohm/km	mH/km	uF/km
48-AL1/8-ST1A	10	210	0.35	1.11	0.0104
94-AL1/15-ST1A	10	350	0.33	1.05	0.0112
122-AL1/20-ST1A	10	410	0.31	0.99	0.0115
48-AL1/8-ST1A	20	210	0.37	1.18	0.0098
94-AL1/15-ST1A	20	350	0.35	1.11	0.0104
122-AL1/20-ST1A	20	410	0.34	1.08	0.0106

Table 8.4: LV transformers

name	S_nom	u_kr	P_k
#	kVA	%	W
100 kVA	100	4	1750
160 kVA	160	4	2350
250 kVA	250	4	3250
400 kVA	400	4	4600
630 kVA	630	4	6500
800 kVA	800	6	8400
1000 kVA	1000	6	10500

Table 8.5: MV transformers

name	S_nom
#	kVA
20 MVA	20000
32 MVA	32000
40 MVA	40000
63 MVA	63000

CHAPTER 9

API

9.1 edisgo package

9.1.1 Subpackages

edisgo.data package

Submodules

edisgo.data.export_data module

edisgo.data.import_data module

edisgo.data.import_data.**import_from_ding0**(*file*, *network*)

Import an eDisGo grid topology from Ding0 data.

This import method is specifically designed to load grid topology data in the format as Ding0 provides it via pickles.

The import of the grid topology includes

- the topology itself
- equipment parameter
- generators incl. location, type, subtype and capacity
- loads incl. location and sectoral consumption

Parameters

- **file** (str or ding0.core.NetworkDing0) – If a str is provided it is assumed it points to a pickle with Ding0 grid data. This file will be read. If an object of the type ding0.core.NetworkDing0 data will be used directly from this object.

- **network** (*Network*) – The eDisGo data container object

Notes

Assumes `ding0.core.NetworkDing0` provided by *file* contains only data of one mv_grid_district.

`edisgo.data.import_data.`**import_generators**(*network*, *data_source=None*, *file=None*)

Import generator data from source.

The generator data include

- nom. capacity
- type ToDo: specify!
- timeseries

Additional data which can be processed (e.g. used in OEDB data) are

- location
- type
- subtype
- capacity

Parameters

- **network** (*Network*) – The eDisGo container object
- **data_source** (`str`) – Data source. Supported sources:
  - 'oedb'
- **file** (`str`) – File to import data from, required when using file-based sources.

Returns List of generators

Return type pandas.DataFrame

`edisgo.data.import_data.`**import_feedin_timeseries**(*config_data*, *weather_cell_ids*)

Import RES feed-in time series data and process

Parameters

- **config_data** (*dict*) – Dictionary containing config data from config files.
- **weather_cell_ids** (`list`) – List of weather cell id's (integers) to obtain feed-in data for.

Returns Feedin time series

Return type pandas.DataFrame

`edisgo.data.import_data.`**import_load_timeseries**(*config_data*, *data_source*, *mv_grid_id=None*, *year=None*)

Import load time series

Parameters

- **config_data** (*dict*) – Dictionary containing config data from config files.
- **data_source** (*str*) – Specify type of data source. Available data sources are

- **'demandlib'** Determine a load time series with the use of the demandlib. This calculates standard load profiles for 4 different sectors.

- **mv_grid_id** (*str*) – MV grid ID as used in oedb. Provide this if *data_source* is 'oedb'. Default: None.
- **year** (*int*) – Year for which to generate load time series. Provide this if *data_source* is 'demandlib'. Default: None.

Returns Load time series

Return type pandas.DataFrame

Module contents

edisgo.flex_opt package

Submodules

edisgo.flex_opt.check_tech_constraints module

edisgo.flex_opt.check_tech_constraints.**mv_line_load**(*network*)
Checks for over-loading issues in MV grid.

Parameters **network** (*Network*) –

Returns Dataframe containing over-loaded MV lines, their maximum relative over-loading and the corresponding time step. Index of the dataframe are the over-loaded lines of type *Line*. Columns are 'max_rel_overload' containing the maximum relative over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Return type pandas.DataFrame

Notes

Line over-load is determined based on allowed load factors for feed-in and load cases that are defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors'.

edisgo.flex_opt.check_tech_constraints.**lv_line_load**(*network*)
Checks for over-loading issues in LV grids.

Parameters **network** (*Network*) –

Returns Dataframe containing over-loaded LV lines, their maximum relative over-loading and the corresponding time step. Index of the dataframe are the over-loaded lines of type *Line*. Columns are 'max_rel_overload' containing the maximum relative over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Return type pandas.DataFrame

Notes

Line over-load is determined based on allowed load factors for feed-in and load cases that are defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors'.

edisgo.flex_opt.check_tech_constraints.**hv_mv_station_load**(*network*)
Checks for over-loading of HV/MV station.

Parameters **network** (*Network*) –

Returns Dataframe containing over-loaded HV/MV stations, their apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the over-loaded stations of type *MVStation*. Columns are 's_pfa' containing the apparent power at maximal over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Return type pandas.DataFrame

Notes

Over-load is determined based on allowed load factors for feed-in and load cases that are defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors'.

edisgo.flex_opt.check_tech_constraints.**mv_lv_station_load**(*network*)
Checks for over-loading of MV/LV stations.

Parameters **network** (*Network*) –

Returns Dataframe containing over-loaded MV/LV stations, their apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the over-loaded stations of type *LVStation*. Columns are 's_pfa' containing the apparent power at maximal over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Return type pandas.DataFrame

Notes

Over-load is determined based on allowed load factors for feed-in and load cases that are defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors'.

edisgo.flex_opt.check_tech_constraints.**mv_voltage_deviation**(*network*, *voltage_levels='mv_lv'*)
Checks for voltage stability issues in MV grid.

Parameters

- **network** (*Network*) –
- **voltage_levels** (str) – Specifies which allowed voltage deviations to use. Possible options are:
  - 'mv_lv' This is the default. The allowed voltage deviation for nodes in the MV grid is the same as for nodes in the LV grid. Further load and feed-in case are not distinguished.
  - 'mv' Use this to handle allowed voltage deviations in the MV and LV grid differently. Here, load and feed-in case are differentiated as well.

Returns Dictionary with *MVGrid* as key and a pandas.DataFrame with its critical nodes, sorted descending by voltage deviation, as value. Index of the dataframe are all nodes (of type *Generator*, *Load*, etc.) with over-voltage issues. Columns are 'v_mag_pu' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the over-voltage occured in as pandas.Timestamp.

Return type dict

Notes

Voltage issues are determined based on allowed voltage deviations defined in the config file 'config_grid_expansion' in section 'grid_expansion_allowed_voltage_deviations'.

edisgo.flex_opt.check_tech_constraints.**lv_voltage_deviation**(*network*, *mode=None*, *voltage_levels='mv_lv'*)

Checks for voltage stability issues in LV grids.

Parameters

- **network** (*Network*) –
- **mode** (*None or String*) – If None voltage at all nodes in LV grid is checked. If mode is set to 'stations' only voltage at busbar is checked.
- **voltage_levels** (str) – Specifies which allowed voltage deviations to use. Possible options are:
  - 'mv_lv' This is the default. The allowed voltage deviation for nodes in the MV grid is the same as for nodes in the LV grid. Further load and feed-in case are not distinguished.
  - 'lv' Use this to handle allowed voltage deviations in the MV and LV grid differently. Here, load and feed-in case are differentiated as well.

Returns Dictionary with *LVGrid* as key and a pandas.DataFrame with its critical nodes, sorted descending by voltage deviation, as value. Index of the dataframe are all nodes (of type *Generator*, *Load*, etc.) with over-voltage issues. Columns are 'v_mag_pu' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the over-voltage occured in as pandas.Timestamp.

Return type dict

Notes

Voltage issues are determined based on allowed voltage deviations defined in the config file 'config_grid_expansion' in section 'grid_expansion_allowed_voltage_deviations'.

edisgo.flex_opt.check_tech_constraints.**check_ten_percent_voltage_deviation**(*network*)

Checks if 10% criteria is exceeded.

Parameters **network** (*Network*) –

edisgo.flex_opt.costs module

edisgo.flex_opt.costs.**grid_expansion_costs**(*network*, *without_generator_import=False*)

Calculates grid expansion costs for each reinforced transformer and line in kEUR.

edisgo.flex_opt.costs.**network**

Type *Network*

edisgo.flex_opt.costs.**without_generator_import**

If True excludes lines that were added in the generator import to connect new generators to the grid from calculation of grid expansion costs. Default: False.

Type Boolean

Returns

DataFrame containing type and costs plus in the case of lines the line length and number of parallel lines of each reinforced transformer and line. Index of the DataFrame is the respective object that can either be a *Line* or a *Transformer*. Columns are the following:

type: String Transformer size or cable name

total_costs: float Costs of equipment in kEUR. For lines the line length and number of parallel lines is already included in the total costs.

quantity: int For transformers quantity is always one, for lines it specifies the number of parallel lines.

line_length: float Length of line or in case of parallel lines all lines in km.

voltage_level [str {'lv' | 'mv' | 'mv/lv'}] Specifies voltage level the equipment is in.

mv_feeder [*Line*] First line segment of half-ring used to identify in which feeder the grid expansion was conducted in.

Return type *pandas.DataFrame<dataframe>*

Notes

Total grid expansion costs can be obtained through self.grid_expansion_costs.total_costs.sum().

edisgo.flex_opt.curtailment module

edisgo.flex_opt.curtailment.**voltage_based**(*feedin*, *generators*, *curtailment_timeseries*, *edisgo*, *curtailment_key*, ***kwargs*)

Implements curtailment methodology 'voltage-based'.

The curtailment that has to be met in each time step is allocated depending on the exceedance of the allowed voltage deviation at the nodes of the generators. The higher the exceedance, the higher the curtailment.

The optional parameter *voltage_threshold* specifies the threshold for the exceedance of the allowed voltage deviation above which a generator is curtailed. By default it is set to zero, meaning that all generators at nodes with voltage deviations that exceed the allowed voltage deviation are curtailed. Generators at nodes where the allowed voltage deviation is not exceeded are not curtailed. In the case that the required curtailment exceeds the weather-dependent availability of all generators with voltage deviations above the specified threshold, the voltage threshold is lowered in steps of 0.01 p.u. until the curtailment target can be met.

Above the threshold, the curtailment is proportional to the exceedance of the allowed voltage deviation. In order to find the linear relation between the curtailment and the voltage difference a linear problem is formulated and solved using the python package pyomo. See documentation for further information.

Parameters

- **feedin** (pandas.DataFrame) – Dataframe holding the feed-in of each generator in kW for the technology (and weather cell) specified in *curtailment_key* parameter. Index of the dataframe is a pandas.DatetimeIndex. Columns are the representatives of the fluctuating generators.
- **generators** (pandas.DataFrame) – Dataframe with all generators of the type (and in weather cell) specified in *curtailment_key* parameter. See return value of *edisgo.grid.tools.get_gen_info()* for more information.

- **curtailment_timeseries** (pandas.Series) – The curtailment in kW to be distributed amongst the generators in *generators* parameter. Index of the series is a pandas.DatetimeIndex.
- **edisgo** (*edisgo.grid.network.EDisGo*) –
- **curtailment_key** (str or tuple with str) – The technology and weather cell ID if tuple or only the technology if str the curtailment is specified for.
- **voltage_threshold** (float) – The node voltage below which no curtailment is assigned to the respective generator if not necessary. Default: 0.0.
- **solver** (str) – The solver used to optimize the curtailment assigned to the generator. Possible options are:
  - 'cbc' coin-or branch and cut solver
  - 'glpk' gnu linear programming kit solver
  - any other available compatible with 'pyomo' like 'gurobi' or 'cplex'

  Default: 'cbc'

edisgo.flex_opt.curtailment.**feedin_proportional**(*feedin*, *generators*, *curtailment_timeseries*, *edisgo*, *curtailment_key*, ***kwargs*)

Implements curtailment methodology 'feedin-proportional'.

The curtailment that has to be met in each time step is allocated equally to all generators depending on their share of total feed-in in that time step.

Parameters

- **feedin** (pandas.DataFrame) – Dataframe holding the feed-in of each generator in kW for the technology (and weather cell) specified in *curtailment_key* parameter. Index of the dataframe is a pandas.DatetimeIndex. Columns are the representatives of the fluctuating generators.
- **generators** (pandas.DataFrame) – Dataframe with all generators of the type (and in weather cell) specified in *curtailment_key* parameter. See return value of *edisgo.grid.tools.get_gen_info()* for more information.
- **curtailment_timeseries** (pandas.Series) – The curtailment in kW to be distributed amongst the generators in *generators* parameter. Index of the series is a pandas.DatetimeIndex.
- **edisgo** (*edisgo.grid.network.EDisGo*) –
- **curtailment_key** (str or tuple with str) – The technology and weather cell ID if tuple or only the technology if str the curtailment is specified for.

edisgo.flex_opt.exceptions module

exception edisgo.flex_opt.exceptions.**Error**

Bases: Exception

Base class for exceptions in this module.

exception edisgo.flex_opt.exceptions.**MaximumIterationError**(*message*)

Bases: *edisgo.flex_opt.exceptions.Error*

Exception raised when maximum number of iterations in grid reinforcement is exceeded.

message -- explanation of the error

exception edisgo.flex_opt.exceptions.**ImpossibleVoltageReduction**(*message*)
Bases: *edisgo.flex_opt.exceptions.Error*

Exception raised when voltage issue cannot be solved.

message -- explanation of the error

edisgo.flex_opt.reinforce_grid module

edisgo.flex_opt.reinforce_grid.**reinforce_grid**(*edisgo*, *timesteps_pfa=None*, *copy_graph=False*, *max_while_iterations=10*, *combined_analysis=False*)

Evaluates grid reinforcement needs and performs measures.

This function is the parent function for all grid reinforcements.

Parameters

- **edisgo** (*EDisGo*) – The eDisGo API object
- **timesteps_pfa** (str or pandas.DatetimeIndex or pandas.Timestamp) – timesteps_pfa specifies for which time steps power flow analysis is conducted and therefore which time steps to consider when checking for over-loading and over-voltage issues. It defaults to None in which case all timesteps in timeseries.timeindex (see *TimeSeries*) are used. Possible options are:
  - None Time steps in timeseries.timeindex (see *TimeSeries*) are used.
  - 'snapshot_analysis' Reinforcement is conducted for two worst-case snapshots. See *edisgo.tools.tools.select_worstcase_snapshots()* for further explanation on how worst-case snapshots are chosen. Note: If you have large time series choosing this option will save calculation time since power flow analysis is only conducted for two time steps. If your time series already represents the worst-case keep the default value of None because finding the worst-case snapshots takes some time.
  - pandas.DatetimeIndex or pandas.Timestamp Use this option to explicitly choose which time steps to consider.
- **copy_graph** (Boolean) – If True reinforcement is conducted on a copied graph and discarded. Default: False.
- **max_while_iterations** (int) – Maximum number of times each while loop is conducted.
- **combined_analysis** (Boolean) – If True allowed voltage deviations for combined analysis of MV and LV grid are used. If False different allowed voltage deviations for MV and LV are used. See also config section *grid_expansion_allowed_voltage_deviations*. Default: False.

Returns Returns the Results object holding grid expansion costs, equipment changes, etc.

Return type *Results*

Notes

See *Features in detail* for more information on how grid reinforcement is conducted.

edisgo.flex_opt.reinforce_measures module

edisgo.flex_opt.reinforce_measures.**extend_distribution_substation_overloading**(*network*, *critical_stations*)

Reinforce MV/LV substations due to overloading issues.

In a first step a parallel transformer of the same kind is installed. If this is not sufficient as many standard transformers as needed are installed.

Parameters

- **network** (*Network*) –
- **critical_stations** (pandas.DataFrame) – Dataframe containing over-loaded MV/LV stations, their apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the over-loaded stations of type *LVStation*. Columns are 's_pfa' containing the apparent power at maximal over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp. See *mv_lv_station_load()* for more information.

Returns Dictionary with lists of added and removed transformers.

Return type dict

edisgo.flex_opt.reinforce_measures.**extend_distribution_substation_overvoltage**(*network*, *critical_stations*)

Reinforce MV/LV substations due to voltage issues.

A parallel standard transformer is installed.

Parameters

- **network** (*Network*) –
- **critical_stations** (dict) – Dictionary with *LVGrid* as key and a pandas.DataFrame with its critical station and maximum voltage deviation as value. Index of the dataframe is the *LVStation* with over-voltage issues. Columns are 'v_mag_pu' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the over-voltage occured in as pandas.Timestamp.

Returns

Return type Dictionary with lists of added transformers.

edisgo.flex_opt.reinforce_measures.**extend_substation_overloading**(*network*, *critical_stations*)

Reinforce HV/MV station due to overloading issues.

In a first step a parallel transformer of the same kind is installed. If this is not sufficient as many standard transformers as needed are installed.

Parameters

- **network** (*Network*) –
- **critical_stations** (pandas:*pandas.DataFrame<dataframe>*) – Dataframe containing over-loaded HV/MV stations, their apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the over-loaded stations of type

MVStation. Columns are 's_pfa' containing the apparent power at maximal over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp. See *hv_mv_station_load()* for more information.

Returns

Return type Dictionary with lists of added and removed transformers.

edisgo.flex_opt.reinforce_measures.**reinforce_branches_overvoltage**(*network*, *grid*, *crit_nodes*)

Reinforce MV and LV grid due to voltage issues.

Parameters

- **network** (*Network*) –
- **grid** (*MVGrid* or *LVGrid*) –
- **crit_nodes** (pandas.DataFrame) – Dataframe with critical nodes, sorted descending by voltage deviation. Index of the dataframe are nodes (of type *Generator*, *Load*, etc.) with over-voltage issues. Columns are 'v_mag_pu' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the over-voltage occured in as pandas.Timestamp.

Returns

- Dictionary with *Line* and the number of lines
- *added.*

Notes

Reinforce measures:

1. Disconnect line at 2/3 of the length between station and critical node farthest away from the station and install new standard line 2. Install parallel standard line

In LV grids only lines outside buildings are reinforced; loads and generators in buildings cannot be directly connected to the MV/LV station.

In MV grids lines can only be disconnected at LVStations because they have switch disconnectors needed to operate the lines as half rings (loads in MV would be suitable as well because they have a switch bay (Schaltfeld) but loads in dingo are only connected to MV busbar). If there is no suitable LV station the generator is directly connected to the MV busbar. There is no need for a switch disconnector in that case because generators don't need to be n-1 safe.

edisgo.flex_opt.reinforce_measures.**reinforce_branches_overloading**(*network*, *crit_lines*)

Reinforce MV or LV grid due to overloading.

Parameters

- **network** (*Network*) –
- **crit_lines** (pandas.DataFrame) – Dataframe containing over-loaded lines, their maximum relative over-loading and the corresponding time step. Index of the dataframe are the over-loaded lines of type *Line*. Columns are 'max_rel_overload' containing the maximum relative over-loading as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Returns

- Dictionary with *Line* and the number of Lines
- *added.*

Notes

Reinforce measures:

1. Install parallel line of the same type as the existing line (Only if line is a cable, not an overhead line. Otherwise a standard equipment cable is installed right away.)
2. Remove old line and install as many parallel standard lines as needed.

edisgo.flex_opt.storage_integration module

edisgo.flex_opt.storage_integration.**storage_at_hvmv_substation**(*mv_grid*, *parameters*, *mode=None*)

Place storage at HV/MV substation bus bar.

Parameters

- **mv_grid** (*MVGrid*) – MV grid instance
- **parameters** (dict) – Dictionary with storage parameters. Must at least contain 'nominal_power'. See *StorageControl* for more information.
- **mode** (str, optional) – Operational mode. See *StorageControl* for possible options and more information. Default: None.

Returns Created storage instance and newly added line to connect storage.

Return type *Storage*, *Line*

edisgo.flex_opt.storage_integration.**set_up_storage**(*node*, *parameters*, *voltage_level=None*, *operational_mode=None*)

Sets up a storage instance.

Parameters

- **node** (*Station* or *BranchTee*) – Node the storage will be connected to.
- **parameters** (dict, optional) – Dictionary with storage parameters. Must at least contain 'nominal_power'. See *StorageControl* for more information.
- **voltage_level** (str, optional) – This parameter only needs to be provided if *node* is of type *LVStation*. In that case *voltage_level* defines which side of the LV station the storage is connected to. Valid options are 'lv' and 'mv'. Default: None.
- **operational_mode** (str, optional) – Operational mode. See *StorageControl* for possible options and more information. Default: None.

edisgo.flex_opt.storage_integration.**connect_storage**(*storage*, *node*)

Connects storage to the given node.

The storage is connected by a cable The cable the storage is connected with is selected to be able to carry the storages nominal power and equal amount of reactive power. No load factor is considered.

Parameters

- **storage** (*Storage*) – Storage instance to be integrated into the grid.

- **node** (*Station* or *BranchTee*) – Node the storage will be connected to.

Returns Newly added line to connect storage.

Return type *Line*

edisgo.flex_opt.storage_operation module

edisgo.flex_opt.storage_operation.**fifty_fifty**(*network*, *storage*, *feedin_threshold=0.5*)

Operational mode where the storage operation depends on actual power by generators. If cumulative generation exceeds 50% of nominal power, the storage is charged. Otherwise, the storage is discharged. The time series for active power is written into the storage.

Parameters

- **network** (*Network*) –
- **storage** (*Storage*) – Storage instance for which to generate time series.
- **feedin_threshold** (float) – Ratio of generation to installed power specifying when to charge or discharge the storage. If feed-in threshold is e.g. 0.5 the storage will be charged when the total generation is 50% of the installed generator capacity and discharged when it is below.

edisgo.flex_opt.storage_positioning module

edisgo.flex_opt.storage_positioning.**one_storage_per_feeder**(*edisgo*, *storage_timeseries*, *storage_nominal_power=None*, ***kwargs*)

Allocates the given storage capacity to multiple smaller storages.

For each feeder with load or voltage issues it is checked if integrating a storage will reduce peaks in the feeder, starting with the feeder with the highest theoretical grid expansion costs. A heuristic approach is used to estimate storage sizing and siting while storage operation is carried over from the given storage operation.

Parameters

- **edisgo** (*EDisGo*) –
- **storage_timeseries** (pandas.DataFrame) – Total active and reactive power time series that will be allocated to the smaller storages in feeders with load or voltage issues. Columns of the dataframe are 'p' containing active power time series in kW and 'q' containing the reactive power time series in kvar. Index is a pandas.DatetimeIndex.
- **storage_nominal_power** (float or None) – Nominal power in kW that will be allocated to the smaller storages in feeders with load or voltage issues. If no nominal power is provided the maximum active power given in *storage_timeseries* is used. Default: None.
- **debug** (Boolean, optional) – If dedug is True a dataframe with storage size and path to storage of all installed and possibly discarded storages is saved to a csv file and a plot with all storage positions is created and saved, both to the current working directory with filename *storage_results_{MVgrid_id}*. Default: False.
- **check_costs_reduction** (Boolean or str, optional) – This parameter specifies when and whether it should be checked if a storage reduced grid expansion costs or not. It can be used as a safety check but can be quite time consuming. Possible options are:

- 'each_feeder' Costs reduction is checked for each feeder. If the storage did not reduce grid expansion costs it is discarded.
- 'once' Costs reduction is checked after the total storage capacity is allocated to the feeders. If the storages did not reduce grid expansion costs they are all discarded.
- False Costs reduction is never checked.

Default: False.

Module contents

edisgo.grid package

Submodules

edisgo.grid.components module

class edisgo.grid.components.**Component**(***kwargs*)

Bases: object

Generic component

Notes

In case of a MV-LV voltage station, *grid* refers to the LV grid.

id

Unique ID of component

Returns Unique ID of component

Return type int

geom

Location of component

Returns Location of the *Component* as Shapely Point or LineString

Return type Shapely Point object or Shapely LineString object

grid

Grid the component belongs to

Returns The MV or LV grid the component belongs to

Return type *MVGrid* or *LVGrid*

class edisgo.grid.components.**Station**(***kwargs*)

Bases: *edisgo.grid.components.Component*

Station object (medium or low voltage)

Represents a station, contains transformers.

transformers

Transformers located in station

Type list of *Transformer*

add_transformer(*transformer*)

class edisgo.grid.components.**Transformer**(**kwargs)
Bases: *edisgo.grid.components.Component*

Transformer object

_voltage_op
Operational voltage

Type float

_type
Specification of type, refers to ToDo: ADD CORRECT REF TO (STATIC) DATA

Type pandas.DataFrame

mv_grid

voltage_op

type

class edisgo.grid.components.**Load**(**kwargs)
Bases: *edisgo.grid.components.Component*

Load object

_timeseries
See *timeseries* getter for more information.

Type pandas.Series, optional

_consumption
See *consumption* getter for more information.

Type dict, optional

_timeseries_reactive
See *timeseries_reactive* getter for more information.

Type pandas.Series, optional

_power_factor
See *power_factor* getter for more information.

Type float, optional

_reactive_power_mode
See *reactive_power_mode* getter for more information.

Type str, optional

_q_sign
See *q_sign* getter for more information.

Type int, optional

timeseries
Load time series

It returns the actual time series used in power flow analysis. If *_timeseries* is not None, it is returned. Otherwise, *timeseries()* looks for time series of the according sector in *TimeSeries* object.

Returns DataFrame containing active power in kW in column 'p' and reactive power in kVA in column 'q'.

Return type pandas.DataFrame

timeseries_reactive
Reactive power time series in kvar.

Parameters **timeseries_reactive** (pandas.Seriese) – Series containing reactive power in kvar.

Returns Series containing reactive power time series in kvar. If it is not set it is tried to be retrieved from *load_reactive_power* attribute of global TimeSeries object. If that is not possible None is returned.

Return type pandas.Series or None

pypsa_timeseries(*attr*)
Return time series in PyPSA format

Parameters **attr** (*str*) – Attribute name (PyPSA conventions). Choose from {p_set, q_set}

consumption
Annual consumption per sector in kWh

Sectors

- retail/industrial
- agricultural
- residential

The format of the dict is as follows:

```
{
    'residential': 453.4
}
```

Type dict

peak_load
Get sectoral peak load

power_factor
Power factor of load

Parameters **power_factor** (float) – Ratio of real power to apparent power.

Returns Ratio of real power to apparent power. If power factor is not set it is retrieved from the network config object depending on the grid level the load is in.

Return type float

reactive_power_mode
Power factor mode of Load.

This information is necessary to make the load behave in an inductive or capacitive manner. Essentially this changes the sign of the reactive power.

The convention used here in a load is that: - when *reactive_power_mode* is 'inductive' then Q is positive - when *reactive_power_mode* is 'capacitive' then Q is negative

Parameters **reactive_power_mode** (str or None) – Possible options are 'inductive', 'capacitive' and 'not_applicable'. In the case of 'not_applicable' a reactive power time series must be given.

Returns In the case that this attribute is not set, it is retrieved from the network config object depending on the voltage level the load is in.

Return type str

q_sign
Get the sign of reactive power based on *_reactive_power_mode*.

Returns In case of inductive reactive power returns +1 and in case of capacitive reactive power returns -1. If reactive power time series is given, *q_sign* is set to None.

Return type int or None

class edisgo.grid.components.**Generator**(***kwargs*)
Bases: *edisgo.grid.components.Component*

Generator object

_timeseries
See *timeseries* getter for more information.

Type pandas.Series, optional

_nominal_capacity
See *nominal_capacity* getter for more information.

Type dict, optional

_type
See *type* getter for more information.

Type pandas.Series, optional

_subtype
See *subtype* getter for more information.

Type str, optional

_v_level
See *v_level* getter for more information.

Type str, optional

_q_sign
See *q_sign* getter for more information.

Type int, optional

_power_factor
See *power_factor* getter for more information.

Type float, optional

_reactive_power_mode
See *reactive_power_mode* getter for more information.

Type str, optional

_q_sign
See *q_sign* getter for more information.

Type int, optional

Notes

The attributes *_type* and *_subtype* have to match the corresponding types in Timeseries to allow allocation of time series to generators.

See also:

edisgo.network.TimeSeries Details of global *TimeSeries*

timeseries
Feed-in time series of generator

It returns the actual dispatch time series used in power flow analysis. If *_timeseries* is not None, it is returned. Otherwise, *timeseries()* looks for time series of the according type of technology in *TimeSeries*. If the reactive power time series is provided through _timeseries_reactive, this is added to *_timeseries*. When _timeseries_reactive is not set, the reactive power is also calculated in *_timeseries* using *power_factor* and *reactive_power_mode*. The *power_factor* determines the magnitude of the reactive power based on the power factor and active power provided and the *reactive_power_mode* determines if the reactive power is either consumed (inductive behaviour) or provided (capacitive behaviour).

Returns DataFrame containing active power in kW in column 'p' and reactive power in kvar in column 'q'.

Return type pandas.DataFrame

timeseries_reactive
Reactive power time series in kvar.

Parameters **timeseries_reactive** (pandas.Seriese) – Series containing reactive power in kvar.

Returns Series containing reactive power time series in kvar. If it is not set it is tried to be retrieved from *generation_reactive_power* attribute of global TimeSeries object. If that is not possible None is returned.

Return type pandas.Series or None

pypsa_timeseries(*attr*)
Return time series in PyPSA format

Convert from kW, kVA to MW, MVA

Parameters **attr** (str) – Attribute name (PyPSA conventions). Choose from {p_set, q_set}

type
Technology type (e.g. 'solar')

Type str

subtype
Technology subtype (e.g. 'solar_roof_mounted')

Type str

nominal_capacity
Nominal generation capacity in kW

Type float

v_level
Voltage level

Type int

power_factor
Power factor of generator

Parameters **power_factor** (float) – Ratio of real power to apparent power.

Returns Ratio of real power to apparent power. If power factor is not set it is retrieved from the network config object depending on the grid level the generator is in.

Return type float

reactive_power_mode
Power factor mode of generator.

This information is necessary to make the generator behave in an inductive or capacitive manner. Essentially this changes the sign of the reactive power.

The convention used here in a generator is that: - when *reactive_power_mode* is 'capacitive' then Q is positive - when *reactive_power_mode* is 'inductive' then Q is negative

In the case that this attribute is not set, it is retrieved from the network config object depending on the voltage level the generator is in.

Parameters **reactive_power_mode** (str or None) – Possible options are 'inductive', 'capacitive' and 'not_applicable'. In the case of 'not_applicable' a reactive power time series must be given.

Returns **:obj:'str'** – In the case that this attribute is not set, it is retrieved from the network config object depending on the voltage level the generator is in.

Return type Power factor mode

q_sign
Get the sign of reactive power based on *_reactive_power_mode*.

Returns In case of inductive reactive power returns -1 and in case of capacitive reactive power returns +1. If reactive power time series is given, *q_sign* is set to None.

Return type int or None

class edisgo.grid.components.**GeneratorFluctuating**(***kwargs*)
Bases: *edisgo.grid.components.Generator*

Generator object for fluctuating renewables.

_curtailment
See *curtailment* getter for more information.

Type pandas.Series

_weather_cell_id
See *weather_cell_id* getter for more information.

Type int

timeseries
Feed-in time series of generator

It returns the actual time series used in power flow analysis. If _timeseries is not None, it is returned. Otherwise, *timeseries()* looks for generation and curtailment time series of the according type of technology (and weather cell) in *TimeSeries*.

Returns DataFrame containing active power in kW in column 'p' and reactive power in kVA in column 'q'.

Return type pandas.DataFrame

timeseries_reactive
Reactive power time series in kvar.

:param pandas.Series: Series containing reactive power time series in kvar.

Returns Series containing reactive power time series in kvar. If it is not set it is tried to be retrieved from *generation_reactive_power* attribute of global TimeSeries object. If that is not possible None is returned.

Return type pandas.DataFrame or None

curtailment
param curtailment_ts: See class definition for details. :type curtailment_ts: pandas.Series

Returns If self._curtailment is set it returns that. Otherwise, if curtailment in *TimeSeries* for the corresponding technology type (and if given, weather cell ID) is set this is returned.

Return type pandas.Series

weather_cell_id
Get weather cell ID

Returns See class definition for details.

Return type str

class edisgo.grid.components.**Storage**(***kwargs*)
Bases: *edisgo.grid.components.Component*

Storage object

Describes a single storage instance in the eDisGo grid. Includes technical parameters such as *Storage.efficiency_in* or *Storage.standing_loss* as well as its time series of operation *Storage.timeseries()*.

timeseries
Time series of storage operation

Parameters **ts** (pandas.DataFrame) – DataFrame containing active power the storage is charged (negative) and discharged (positive) with (on the grid side) in kW in column 'p' and reactive power in kvar in column 'q'. When 'q' is positive, reactive power is supplied (behaving as a capacitor) and when 'q' is negative reactive power is consumed (behaving as an inductor).

Returns See parameter *timeseries*.

Return type pandas.DataFrame

pypsa_timeseries(*attr*)
Return time series in PyPSA format

Convert from kW, kVA to MW, MVA

Parameters **attr** (*str*) – Attribute name (PyPSA conventions). Choose from {p_set, q_set}

nominal_power
Nominal charging and discharging power of storage instance in kW.

Returns Storage nominal power

Return type float

max_hours
Maximum state of charge capacity in terms of hours at full discharging power *nominal_power*.

Returns Hours storage can be discharged for at nominal power

Return type float

nominal_capacity
Nominal storage capacity in kWh.

Returns Storage nominal capacity

Return type float

soc_initial
Initial state of charge in kWh.

Returns Initial state of charge

Return type float

efficiency_in
Storage charging efficiency in per unit.

Returns Charging efficiency in range of 0..1

Return type float

efficiency_out
Storage discharging efficiency in per unit.

Returns Discharging efficiency in range of 0..1

Return type float

standing_loss
Standing losses of storage in %/100 / h

Losses relative to SoC per hour. The unit is pu (%/100%). Hence, it ranges from 0..1.

Returns Standing losses in pu.

Return type float

operation
Storage operation definition

Returns

Return type `str`

power_factor
Power factor of storage

If power factor is not set it is retrieved from the network config object depending on the grid level the storage is in.

Returns **:obj:'float'** – Ratio of real power to apparent power.

Return type Power factor

reactive_power_mode
Power factor mode of storage.

If the power factor is set, then it is necessary to know whether it is leading or lagging. In other words this information is necessary to make the storage behave in an inductive or capacitive manner. Essentially this changes the sign of the reactive power Q.

The convention used here in a storage is that: - when *reactive_power_mode* is 'capacitive' then Q is positive - when *reactive_power_mode* is 'inductive' then Q is negative

In the case that this attribute is not set, it is retrieved from the network config object depending on the voltage level the storage is in.

Returns Either 'inductive' or 'capacitive'

Return type obj: *str* : Power factor mode

q_sign
Get the sign reactive power based on the :attr: *_reactive_power_mode*

Returns

Return type obj: *int* : +1 or -1

class edisgo.grid.components.**MVDisconnectingPoint**(***kwargs*)
Bases: *edisgo.grid.components.Component*

Disconnecting point object

Medium voltage disconnecting points = points where MV rings are split under normal operation conditions (= switch disconnectors in DINGO).

_nodes
Nodes of switch disconnector line segment

Type tuple

open()
Toggle state to open switch disconnector

close()
Toggle state to closed switch disconnector

state
Get state of switch disconnector

Returns

State of MV ring disconnector: 'open' or 'closed'.

Returns *None* if switch disconnector line segment is not set. This refers to an open ring, but it's unknown if the grid topology was built correctly.

Return type str or None

line
Get or set line segment that belongs to the switch disconnector

The setter allows only to set the respective line initially. Once the line segment representing the switch disconnector is set, it cannot be changed.

Returns Line segment that is part of the switch disconnector model

Return type *Line*

class edisgo.grid.components.**BranchTee**(***kwargs*)
Bases: *edisgo.grid.components.Component*

Branch tee object

A branch tee is used to branch off a line to connect another node (german: Abzweigmuffe)

class edisgo.grid.components.**MVStation**(***kwargs*)
Bases: *edisgo.grid.components.Station*

MV Station object

class edisgo.grid.components.**LVStation**(***kwargs*)
Bases: *edisgo.grid.components.Station*

LV Station object

mv_grid

class edisgo.grid.components.**Line**(***kwargs*)
Bases: *edisgo.grid.components.Component*

Line object

Parameters

- **_type** (pandas.Series) – Equipment specification including R and X for power flow analysis Columns:

Column	Description	Unit	Data type
name	Name (e.g. NAYY..)	-	str
U_n	Nominal voltage	kV	int
I_max_th	Max. th. current	A	float
R	Resistance	Ohm/km	float
L	Inductance	mH/km	float
C	Capacitance	uF/km	float
Source Data source - str			

- **_length** (*float*) – Length of the line calculated in linear distance. Unit: m
- **_quantity** (*float*) – Quantity of parallel installed lines.
- **_kind** (*String*) – Specifies whether the line is an underground cable ('cable') or an overhead line ('line').

geom
Provide Shapely LineString object geometry of *Line*

type

length

quantity

kind

edisgo.grid.connect module

edisgo.grid.connect.**connect_mv_generators**(*network*)
Connect MV generators to existing grids.

This function searches for unconnected generators in MV grids and connects them.

It connects

- **generators of voltage level 4**
  - to HV-MV station
- **generators of voltage level 5**
  - with a nom. capacity of <=30 kW to LV loads of type residential
  - **with a nom. capacity of >30 kW and <=100 kW to LV loads of type** retail, industrial or agricultural
  - to the MV-LV station if no appropriate load is available (fallback)

Parameters **network** (*Network*) – The eDisGo container object

Notes

Adapted from Ding0.

edisgo.grid.connect.**connect_lv_generators**(*network*, *allow_multiple_genos_per_load=True*)
Connect LV generators to existing grids.

This function searches for unconnected generators in all LV grids and connects them.

It connects

- **generators of voltage level 6**
  - to MV-LV station
- **generators of voltage level 7**
  - with a nom. capacity of <=30 kW to LV loads of type residential
  - **with a nom. capacity of >30 kW and <=100 kW to LV loads of type** retail, industrial or agricultural
  - to the MV-LV station if no appropriate load is available (fallback)

Parameters

- **network** (*Network*) – The eDisGo container object
- **allow_multiple_genos_per_load** (bool) – If True, more than one generator can be connected to one load

Notes

For the allocation, loads are selected randomly (sector-wise) using a predefined seed to ensure reproducibility.

edisgo.grid.grids module

class edisgo.grid.grids.**Grid**(***kwargs*)
Bases: object

Defines a basic grid in eDisGo

_id
Identifier

Type str

_network
Network which this grid is associated with

Type *Network*

_voltage_nom
Nominal voltage

Type int

_peak_load
Cumulative peak load of grid

Type float

_peak_generation
Cumulative peak generation of grid

Type float

_grid_district
Contains information about grid district (supplied region) of grid, format: ToDo: DEFINE FORMAT

Type dict

_station
The station the grid is fed by

Type *Station*

_weather_cells
Contains a list of weather_cells within the grid

Type list

_generators
Contains a list of the generators

Type :obj:'edisgo.components.Generator'

_loads
Contains a list of the loads

Type :obj:'edisgo.components.Load'

connect_generators (*generators*)
Connects generators to grid

Parameters **generators** (pandas.DataFrame) – Generators to be connected

graph
Provide access to the graph

station
Provide access to station

voltage_nom
Provide access to nominal voltage

id

network

grid_district
Provide access to the grid_district

weather_cells
Weather cells contained in grid

Returns list of weather cell ids contained in grid

Return type list

peak_generation
Cumulative peak generation capacity of generators of this grid

Returns Ad-hoc calculated or cached peak generation capacity

Return type float

peak_generation_per_technology
Peak generation of each technology in the grid

Returns Peak generation index by technology

Return type pandas.Series

peak_generation_per_technology_and_weather_cell
Peak generation of each technology and the corresponding weather cell in the grid

Returns Peak generation index by technology

Return type pandas.Series

peak_load
Cumulative peak load capacity of generators of this grid

Returns Ad-hoc calculated or cached peak load capacity

Return type float

consumption
Consumption in kWh per sector for whole grid

Returns Indexed by demand sector

Return type pandas.Series

generators
Connected Generators within the grid

Returns List of Generator Objects

Return type list

loads
Connected Generators within the grid

Returns List of Generator Objects

Return type list

class edisgo.grid.grids.**MVGrid**(***kwargs*)
Bases: *edisgo.grid.grids.Grid*

Defines a medium voltage grid in eDisGo

_mv_disconn_points
MVDisconnectingPoint

Medium voltage disconnecting points = points where MV rings are split under normal operation conditions (= switch disconnectors in DINGO).

Type list of

_aggregates
This attribute is used for DINGO-imported data only. It contains data from DINGO's Aggregated Load Areas. Each list element represents one aggregated Load Area.

Type list of dict

lv_grids
list of *LVGrid*

Type LV grids associated to this MV grid

draw()
Draw MV grid's graph using the geo data of nodes

Notes

This method uses the coordinates stored in the nodes' geoms which are usually conformal, not equidistant. Therefore, the plot might be distorted and does not (fully) reflect the real positions or distances between nodes.

class edisgo.grid.grids.**LVGrid**(***kwargs*)
Bases: *edisgo.grid.grids.Grid*

Defines a low voltage grid in eDisGo

class edisgo.grid.grids.**Graph**(*incoming_graph_data=None*, ***attr*)
Bases: networkx.classes.graph.Graph

Graph object

This graph is an object subclassed from *networkX.Graph* extended by extra functionality and specific methods.

nodes_from_line(*line*)
Get nodes adjacent to line

Here, line refers to the object behind the key 'line' of the attribute dict attached to each edge.

Parameters **line** (*edisgo.grid.components.Line*) – A eDisGo line object

Returns Nodes adjacent to this edge

Return type tuple

line_from_nodes(*u*, *v*)
Get line between two nodes u and v.

Parameters

- **u** (*Component*) – One adjacent node
- **v** (*Component*) – The other adjacent node

Returns Line segment connecting u and v.

Return type *Line*

nodes_by_attribute(*attr_val*, *attr='type'*)
Select Graph's nodes by attribute value

Get all nodes that share the same attribute. By default, the attr 'type' is used to specify the nodes type (generator, load, etc.).

Examples

```
>>> import edisgo
>>> G = edisgo.grids.Graph()
>>> G.add_node(1, type='generator')
>>> G.add_node(2, type='load')
>>> G.add_node(3, type='generator')
>>> G.nodes_by_attribute('generator')
[1, 3]
```

Parameters

- **attr_val** (*str*) – Value of the *attr* nodes should be selected by
- **attr** (*str, default: 'type'*) – Attribute key which is 'type' by default

Returns A list containing nodes elements that match the given attribute value

Return type list

lines_by_attribute(*attr_val=None*, *attr='type'*)

Returns a generator for iterating over Graph's lines by attribute value.

Get all lines that share the same attribute. By default, the attr 'type' is used to specify the lines' type (line, agg_line, etc.).

The edge of a graph is described by the two adjacent nodes and the line object itself. Whereas the line object is used to hold all relevant power system parameters.

Examples

```
>>> import edisgo
>>> G = edisgo.grids.Graph()
>>> G.add_node(1, type='generator')
>>> G.add_node(2, type='load')
>>> G.add_edge(1, 2, type='line')
>>> lines = G.lines_by_attribute('line')
>>> list(lines)[0]
<class 'tuple'>: ((node1, node2), line)
```

Parameters

- **attr_val** (*str*) – Value of the *attr* lines should be selected by
- **attr** (*str, default: 'type'*) – Attribute key which is 'type' by default

Returns A list containing line elements that match the given attribute value

Return type Generator of dict

Notes

There are generator functions for nodes (*Graph.nodes()*) and edges (*Graph.edges()*) in NetworkX but unlike graph nodes, which can be represented by objects, branch objects can only be accessed by using an edge attribute ('line' is used here)

To make access to attributes of the line objects simpler and more intuitive for the user, this generator yields a dictionary for each edge that contains information about adjacent nodes and the line object.

Note, the construction of the dictionary highly depends on the structure of the in-going tuple (which is defined by the needs of networkX). If this changes, the code will break.

Adapted from Dingo.

lines()

Returns a generator for iterating over Graph's lines

Returns A list containing line elements

Return type Generator of dict

Notes

For a detailed description see lines_by_attribute()

edisgo.grid.network module

class edisgo.grid.network.**EDisGoReimport**(*results_path*, ***kwargs*)
Bases: object

EDisGo class created from saved results.

plot_mv_grid_topology(*technologies=False*, ***kwargs*)
Plots plain MV grid topology and optionally nodes by technology type (e.g. station or generator).

Parameters **technologies** (Boolean) – If True plots stations, generators, etc. in the grid in different colors. If False does not plot any nodes. Default: False.

:param For more information see *edisgo.tools.plots.mv_grid_topology()*.:

plot_mv_voltages(***kwargs*)
Plots voltages in MV grid on grid topology plot.

For more information see *edisgo.tools.plots.mv_grid_topology()*.

plot_mv_line_loading(***kwargs*)
Plots relative line loading (current from power flow analysis to allowed current) of MV lines.

For more information see *edisgo.tools.plots.mv_grid_topology()*.

plot_mv_grid_expansion_costs(***kwargs*)
Plots costs per MV line.

For more information see *edisgo.tools.plots.mv_grid_topology()*.

plot_mv_storage_integration(***kwargs*)
Plots storage position in MV grid of integrated storages.

For more information see *edisgo.tools.plots.mv_grid_topology()*.

histogram_voltage(*timestep=None*, *title=True*, ***kwargs*)
Plots histogram of voltages.

For more information on the histogram plot and possible configurations see *edisgo.tools.plots.histogram()*.

Parameters

- **timestep** (pandas.Timestamp or list(pandas.Timestamp) or None, optional) – Specifies time steps histogram is plotted for. If timestep is None all time steps voltages are calculated for are used. Default: None.
- **title** (str or bool, optional) – Title for plot. If True title is auto generated. If False plot has no title. If str, the provided title is used. Default: True.

histogram_relative_line_load(*timestep=None*, *title=True*, *voltage_level='mv_lv'*, ***kwargs*)
Plots histogram of relative line loads.

For more information on how the relative line load is calculated see edisgo.tools.tools.get_line_loading_from_network(). For more information on the histogram plot and possible configurations see *edisgo.tools.plots.histogram()*.

Parameters

- **timestep** (pandas.Timestamp or list(pandas.Timestamp) or None, optional) – Specifies time step(s) histogram is plotted for. If *timestep* is None all time steps currents are calculated for are used. Default: None.

- **title** (str or bool, optional) – Title for plot. If True title is auto generated. If False plot has no title. If str, the provided title is used. Default: True.
- **voltage_level** (str) – Specifies which voltage level to plot voltage histogram for. Possible options are 'mv', 'lv' and 'mv_lv'. 'mv_lv' is also the fallback option in case of wrong input. Default: 'mv_lv'

class edisgo.grid.network.**EDisGo**(***kwargs*)

Bases: *edisgo.grid.network.EDisGoReimport*

Provides the top-level API for invocation of data import, analysis of hosting capacity, grid reinforcement and flexibility measures.

Parameters

- **worst_case_analysis** (None or str, optional) – If not None time series for feed-in and load will be generated according to the chosen worst case analysis Possible options are:
  - 'worst-case' feed-in for the two worst-case scenarios feed-in case and load case are generated
  - 'worst-case-feedin' feed-in for the worst-case scenario feed-in case is generated
  - 'worst-case-load' feed-in for the worst-case scenario load case is generated

  Be aware that if you choose to conduct a worst-case analysis your input for the following parameters will not be used: * *timeseries_generation_fluctuating* * *timeseries_generation_dispatchable* * *timeseries_load*
- **mv_grid_id** (str) – MV grid ID used in import of ding0 grid.
- **ding0_grid** (file: str or ding0.core.NetworkDing0) – If a str is provided it is assumed it points to a pickle with Ding0 grid data. This file will be read. If an object of the type ding0.core.NetworkDing0 data will be used directly from this object. This will probably be removed when ding0 grids are in oedb.
- **config_path** (None or str or dict) – Path to the config directory. Options are:
  - None If *config_path* is None configs are loaded from the edisgo default config directory ($HOME$/.edisgo). If the directory does not exist it is created. If config files don't exist the default config files are copied into the directory.
  - str If *config_path* is a string configs will be loaded from the directory specified by *config_path*. If the directory does not exist it is created. If config files don't exist the default config files are copied into the directory.
  - dict A dictionary can be used to specify different paths to the different config files. The dictionary must have the following keys:
    - 'config_db_tables'
    - 'config_grid'
    - 'config_grid_expansion'
    - 'config_timeseries'

  Values of the dictionary are paths to the corresponding config file. In contrast to the other two options the directories and config files must exist and are not automatically created.

  Default: None.
- **scenario_description** (None or str) – Can be used to describe your scenario but is not used for anything else. Default: None.

- **timeseries_generation_fluctuating** (str or pandas.DataFrame) – Parameter used to obtain time series for active power feed-in of fluctuating renewables wind and solar. Possible options are:
  - 'oedb' Time series for the year 2011 are obtained from the OpenEnergy DataBase.
  - pandas.DataFrame DataFrame with time series, normalized with corresponding capacity. Time series can either be aggregated by technology type or by type and weather cell ID. In the first case columns of the DataFrame are 'solar' and 'wind'; in the second case columns need to be a pandas.MultiIndex with the first level containing the type and the second level the weather cell id. Index needs to be a pandas.DatetimeIndex.

- **timeseries_generation_dispatchable** (pandas.DataFrame) – DataFrame with time series for active power of each (aggregated) type of dispatchable generator normalized with corresponding capacity. Index needs to be a pandas.DatetimeIndex. Columns represent generator type:
  - 'gas'
  - 'coal'
  - 'biomass'
  - 'other'
  - ...

  Use 'other' if you don't want to explicitly provide every possible type.
- **timeseries_generation_reactive_power** (pandas.DataFrame, optional) – DataFrame with time series of normalized reactive power (normalized by the rated nominal active power) per technology and weather cell. Index needs to be a pandas.DatetimeIndex. Columns represent generator type and can be a MultiIndex column containing the weather cell ID in the second level. If the technology doesn't contain weather cell information i.e. if it is other than solar and wind generation, this second level can be left as a numpy Nan or a None. Default: None. If no time series for the technology or technology and weather cell ID is given, reactive power will be calculated from power factor and power factor mode in the config sections *reactive_power_factor* and *reactive_power_mode* and a warning will be raised. See *Generator* and *GeneratorFluctuating* for more information.
- **timeseries_load** (str or pandas.DataFrame) – Parameter used to obtain time series of active power of (cumulative) loads. Possible options are:
  - 'demandlib' Time series for the year specified in *timeindex* are generated using the oemof demandlib.
  - pandas.DataFrame DataFrame with load time series of each (cumulative) type of load normalized with corresponding annual energy demand. Index needs to be a pandas.DatetimeIndex. Columns represent load type: * 'residential' * 'retail' * 'industrial' * 'agricultural'
- **timeseries_load_reactive_power** (pandas.DataFrame, optional) – DataFrame with time series of normalized reactive power (normalized by annual energy demand) per load sector. Index needs to be a pandas.DatetimeIndex. Columns represent load type:
  - 'residential'
  - 'retail'
  - 'industrial'

– 'agricultural'

Default: None. If no time series for the load sector is given, reactive power will be calculated from power factor and power factor mode in the config sections *reactive_power_factor* and *reactive_power_mode* and a warning will be raised. See *Load* for more information.

- **generator_scenario** (None or str) – If provided defines which scenario of future generator park to use and invokes import of these generators. Possible options are 'nep2035' and 'ego100'.
- **timeindex** (None or pandas.DatetimeIndex) – Can be used to select time ranges of the feed-in and load time series that will be used in the power flow analysis. Also defines the year load time series are obtained for when choosing the 'demandlib' option to generate load time series.

network
The network is a container object holding all data.

Type *Network*

Examples

Assuming you have the Ding0 *ding0_data.pkl* in CWD

Create eDisGo Network object by loading Ding0 file

```
>>> from edisgo.grid.network import EDisGo
>>> edisgo = EDisGo(ding0_grid='ding0_data.pkl', mode='worst-case-feedin')
```

Analyze hosting capacity for MV and LV grid level

```
>>> edisgo.analyze()
```

Print LV station secondary side voltage levels returned by PFA

```
>>> lv_stations = edisgo.network.mv_grid.graph.nodes_by_attribute(
>>>     'lv_station')
>>> print(edisgo.network.results.v_res(lv_stations, 'lv'))
```

curtail(*methodology*, *curtailment_timeseries*, ***kwargs*)
Sets up curtailment time series.

Curtailment time series are written into *TimeSeries*. See *CurtailmentControl* for more information on parameters and methodologies.

import_from_ding0(*file*, ***kwargs*)
Import grid data from DINGO file

For details see *edisgo.data.import_data.import_from_ding0()*

import_generators(*generator_scenario=None*)
Import generators

For details see *edisgo.data.import_data.import_generators()*

analyze(*mode=None*, *timesteps=None*)
Analyzes the grid by power flow analysis

Analyze the grid for violations of hosting capacity. Means, perform a power flow analysis and obtain voltages at nodes (load, generator, stations/transformers and branch tees) and active/reactive power at lines.

The power flow analysis can currently only be performed for both grid levels MV and LV. See ToDos section for more information.

A static non-linear power flow analysis is performed using PyPSA. The high-voltage to medium-voltage transformer are not included in the analysis. The slack bus is defined at secondary side of these transformers assuming an ideal tap changer. Hence, potential overloading of the transformers is not studied here.

Parameters

- **mode** (*str*) – Allows to toggle between power flow analysis (PFA) on the whole grid topology (MV + LV), only MV or only LV. Defaults to None which equals power flow analysis for MV + LV which is the only implemented option at the moment. See ToDos section for more information.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies for which time steps to conduct the power flow analysis. It defaults to None in which case the time steps in timeseries.timeindex (see *TimeSeries*) are used.

Notes

The current implementation always translates the grid topology representation to the PyPSA format and stores it to `self.network.pypsa`.

The option to export only the edisgo MV grid (mode = 'mv') to conduct a power flow analysis is implemented in *to_pypsa()* but NotImplementedError is raised since the rest of edisgo does not handle this option yet. The analyze function will throw an error since *process_pfa_results()* does not handle aggregated loads and generators in the LV grids. Also, grid reinforcement, pypsa update of time series, and probably other functionalities do not work when only the MV grid is analysed.

Further ToDos are: * explain how power plants are modeled, if possible use a link * explain where to find and adjust power flow analysis defining parameters

See also:

to_pypsa() Translator to PyPSA data format

analyze_lopf(*mode=None*, *timesteps=None*, *etrago_max_storage_size=None*)

Analyzes the grid by power flow analysis

Analyze the grid for violations of hosting capacity. Means, perform a power flow analysis and obtain voltages at nodes (load, generator, stations/transformers and branch tees) and active/reactive power at lines.

The power flow analysis can currently only be performed for both grid levels MV and LV. See ToDos section for more information.

A static non-linear power flow analysis is performed using PyPSA. The high-voltage to medium-voltage transformer are not included in the analysis. The slack bus is defined at secondary side of these transformers assuming an ideal tap changer. Hence, potential overloading of the transformers is not studied here.

Parameters

- **mode** (*str*) – Allows to toggle between power flow analysis (PFA) on the whole grid topology (MV + LV), only MV or only LV. Defaults to None which equals power flow analysis for MV + LV which is the only implemented option at the moment. See ToDos section for more information.

- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies for which time steps to conduct the power flow analysis. It defaults to None in which case the time steps in timeseries.timeindex (see *TimeSeries*) are used.

Notes

The current implementation always translates the grid topology representation to the PyPSA format and stores it to `self.network.pypsa`.

The option to export only the edisgo MV grid (mode = 'mv') to conduct a power flow analysis is implemented in *to_pypsa()* but NotImplementedError is raised since the rest of edisgo does not handle this option yet. The analyze function will throw an error since *process_pfa_results()* does not handle aggregated loads and generators in the LV grids. Also, grid reinforcement, pypsa update of time series, and probably other functionalities do not work when only the MV grid is analysed.

Further ToDos are: * explain how power plants are modeled, if possible use a link * explain where to find and adjust power flow analysis defining parameters

See also:

to_pypsa() Translator to PypSA data format

reinforce(***kwargs*)
Reinforces the grid and calculates grid expansion costs.

See *edisgo.flex_opt.reinforce_grid()* for more information.

integrate_storage(*timeseries*, *position*, ***kwargs*)
Integrates storage into grid.

See *StorageControl* for more information.

class edisgo.grid.network.**Network**(***kwargs*)
Bases: object

Used as container for all data related to a single *MVGrid*.

Parameters

- **scenario_description** (str, optional) – Can be used to describe your scenario but is not used for anything else. Default: None.
- **config_path** (None or str or dict, optional) – See *Config* for further information. Default: None.
- **metadata** (dict) – Metadata of Network such as ?
- **data_sources** (dict of str) – Data Sources of grid, generators etc. Keys: 'grid', 'generators', ?
- **mv_grid** (*MVGrid*) – Medium voltage (MV) grid
- **generator_scenario** (str) – Defines which scenario of future generator park to use.

results
Object with results from power flow analyses

Type *Results*

id
MV grid ID

Returns MV grid ID

Return type str

config
eDisGo configuration data.

Returns Config object with configuration data from config files.

Return type *Config*

metadata
Metadata of Network

Returns Metadata of Network

Return type dict

generator_scenario
Defines which scenario of future generator park to use.

Parameters **generator_scenario_name** (str) – Name of scenario of future generator park

Returns Name of scenario of future generator park

Return type str

scenario_description
Used to describe your scenario but not used for anything else.

Parameters **scenario_description** (str) – Description of scenario

Returns Scenario name

Return type str

equipment_data
Technical data of electrical equipment such as lines and transformers

Returns Data of electrical equipment

Return type dict of pandas.DataFrame

mv_grid
Medium voltage (MV) grid

Parameters **mv_grid** (*MVGrid*) – Medium voltage (MV) grid

Returns Medium voltage (MV) grid

Return type *MVGrid*

timeseries
Object containing load and feed-in time series.

Parameters **timeseries** (*TimeSeries*) – Object containing load and feed-in time series.

Returns Object containing load and feed-in time series.

Return type *TimeSeries*

data_sources
Dictionary with data sources of grid, generators etc.

Returns Data Sources of grid, generators etc.

Return type dict of str

set_data_source(*key*, *data_source*)
Set data source for key (e.g. 'grid')

Parameters

- **key** (str) – Specifies data
- **data_source** (str) – Specifies data source

dingo_import_data
Temporary data from ding0 import needed for OEP generator update

pypsa
PyPSA grid representation

A grid topology representation based on pandas.DataFrame. The overall container object of this data model, the pypsa.Network, is assigned to this attribute.

Parameters **pypsa** (pypsa.Network) – The PyPSA network container.

Returns PyPSA grid representation. The attribute *edisgo_mode* is added to specify if pypsa representation of the edisgo network was created for the whole grid topology (MV + LV), only MV or only LV. See parameter *mode* in `analyze()` for more information.

Return type pypsa.Network

class edisgo.grid.network.**Config**(***kwargs*)
Bases: object

Container for all configurations.

Parameters **config_path** (None or str or dict) – Path to the config directory. Options are:

- None If *config_path* is None configs are loaded from the edisgo default config directory ($HOME$/.edisgo). If the directory does not exist it is created. If config files don't exist the default config files are copied into the directory.
- str If *config_path* is a string configs will be loaded from the directory specified by *config_path*. If the directory does not exist it is created. If config files don't exist the default config files are copied into the directory.
- dict A dictionary can be used to specify different paths to the different config files. The dictionary must have the following keys: * 'config_db_tables' * 'config_grid' * 'config_grid_expansion' * 'config_timeseries'

  Values of the dictionary are paths to the corresponding config file. In contrast to the other two options the directories and config files must exist and are not automatically created.

Default: None.

Notes

The Config object can be used like a dictionary. See example on how to use it.

Examples

Create Config object from default config files

```
>>> from edisgo.grid.network import Config
>>> config = Config()
```

Get reactive power factor for generators in the MV grid

```
>>> config['reactive_power_factor']['mv_gen']
```

class edisgo.grid.network.**TimeSeriesControl**(*network*, ***kwargs*)
Bases: object

Sets up TimeSeries Object.

Parameters

- **network** (*Network*) – The eDisGo data container
- **mode** (str, optional) – Mode must be set in case of worst-case analyses and can either be 'worst-case' (both feed-in and load case), 'worst-case-feedin' (only feed-in case) or 'worst-case-load' (only load case). All other parameters except of *config-data* will be ignored. Default: None.
- **timeseries_generation_fluctuating** (str or pandas.DataFrame, optional) – Parameter used to obtain time series for active power feed-in of fluctuating renewables wind and solar. Possible options are:
  - 'oedb' Time series for 2011 are obtained from the OpenEnergy DataBase. *mv_grid_id* and *scenario_description* have to be provided when choosing this option.
  - pandas.DataFrame DataFrame with time series, normalized with corresponding capacity. Time series can either be aggregated by technology type or by type and weather cell ID. In the first case columns of the DataFrame are 'solar' and 'wind'; in the second case columns need to be a pandas.MultiIndex with the first level containing the type and the second level the weather cell ID.

  Default: None.
- **timeseries_generation_dispatchable** (pandas.DataFrame, optional) – DataFrame with time series for active power of each (aggregated) type of dispatchable generator normalized with corresponding capacity. Columns represent generator type:
  - 'gas'
  - 'coal'
  - 'biomass'
  - 'other'
  - ...

  Use 'other' if you don't want to explicitly provide every possible type. Default: None.
- **timeseries_generation_reactive_power** (pandas.DataFrame, optional) – DataFrame with time series of normalized reactive power (normalized by the rated nominal active power) per technology and weather cell. Index needs to be a pandas.DatetimeIndex. Columns represent generator type and can be a MultiIndex column containing the weather cell ID in the second level. If the technology doesn't contain weather cell information i.e. if it is other than solar and wind generation, this second level can be left as an empty string ''.

  Default: None.
- **timeseries_load** (str or pandas.DataFrame, optional) – Parameter used to obtain time series of active power of (cumulative) loads. Possible options are:
  - 'demandlib' Time series are generated using the oemof demandlib.

– pandas.DataFrame DataFrame with load time series of each (cumulative) type of load normalized with corresponding annual energy demand. Columns represent load type:

* 'residential'
* 'retail'
* 'industrial'
* 'agricultural'

Default: None.

- **timeseries_load_reactive_power** (pandas.DataFrame, optional) – Parameter to get the time series of the reactive power of loads. It should be a DataFrame with time series of normalized reactive power (normalized by annual energy demand) per load sector. Index needs to be a pandas.DatetimeIndex. Columns represent load type:
  – 'residential'
  – 'retail'
  – 'industrial'
  – 'agricultural'

  Default: None.
- **timeindex** (pandas.DatetimeIndex) – Can be used to define a time range for which to obtain load time series and feed-in time series of fluctuating renewables or to define time ranges of the given time series that will be used in the analysis.

class edisgo.grid.network.**CurtailmentControl**(*edisgo*, *methodology*, *curtailment_timeseries*, ***kwargs*)

Bases: object

Allocates given curtailment targets to solar and wind generators.

Parameters

- **edisgo** (edisgo.EDisGo) – The parent EDisGo object that this instance is a part of.
- **methodology** (str) – Defines the curtailment strategy. Possible options are:
  – 'feedin-proportional' The curtailment that has to be met in each time step is allocated equally to all generators depending on their share of total feed-in in that time step. For more information see *edisgo.flex_opt.curtailment.feedin_proportional()*.
  – 'voltage-based' The curtailment that has to be met in each time step is allocated based on the voltages at the generator connection points and a defined voltage threshold. Generators at higher voltages are curtailed more. The default voltage threshold is 1.0 but can be changed by providing the argument 'voltage_threshold'. This method formulates the allocation of curtailment as a linear optimization problem using Pyomo and requires a linear programming solver like coin-or cbc (cbc) or gnu linear programming kit (glpk). The solver can be specified through the parameter 'solver'. For more information see *edisgo.flex_opt.curtailment.voltage_based()*.
- **curtailment_timeseries** (pandas.Series or pandas.DataFrame, optional) – Series or DataFrame containing the curtailment time series in kW. Index needs to be a pandas.DatetimeIndex. Provide a Series if the curtailment time series applies to wind and solar generators. Provide a DataFrame if the curtailment time series applies to a specific technology and optionally weather cell. In the first case columns of the DataFrame are e.g. 'solar'

and 'wind'; in the second case columns need to be a pandas.MultiIndex with the first level containing the type and the second level the weather cell ID. Default: None.

- **solver** (str) – The solver used to optimize the curtailment assigned to the generators when 'voltage-based' curtailment methodology is chosen. Possible options are:
  - 'cbc'
  - 'glpk'
  - any other available solver compatible with 'pyomo' such as 'gurobi' or 'cplex'

  Default: 'cbc'.
- **voltage_threshold** (float) – Voltage below which no curtailment is assigned to the respective generator if not necessary when 'voltage-based' curtailment methodology is chosen. See *edisgo.flex_opt.curtailment.voltage_based()* for more information. Default: 1.0.

class edisgo.grid.network.**StorageControl**(*edisgo*, *timeseries*, *position*, ***kwargs*)

Bases: object

Integrates storages into the grid.

Parameters

- **edisgo** (*EDisGo*) –
- **timeseries** (str or pandas.Series or dict) – Parameter used to obtain time series of active power the storage(s) is/are charged (negative) or discharged (positive) with. Can either be a given time series or an operation strategy. Possible options are:
  - pandas.Series Time series the storage will be charged and discharged with can be set directly by providing a pandas.Series with time series of active charge (negative) and discharge (positive) power in kW. Index needs to be a pandas.DatetimeIndex. If no nominal power for the storage is provided in *parameters* parameter, the maximum of the time series is used as nominal power. In case of more than one storage provide a dict where each entry represents a storage. Keys of the dictionary have to match the keys of the *parameters dictionary*, values must contain the corresponding time series as pandas.Series.
  - 'fifty-fifty' Storage operation depends on actual power of generators. If cumulative generation exceeds 50% of the nominal power, the storage will charge. Otherwise, the storage will discharge. If you choose this option you have to provide a nominal power for the storage. See *parameters* for more information.

  Default: None.
- **position** (None or str or *Station* or *BranchTee* or *Generator* or *Load* or dict) – To position the storage a positioning strategy can be used or a node in the grid can be directly specified. Possible options are:
  - 'hvmv_substation_busbar' Places a storage unit directly at the HV/MV station's bus bar.
  - *Station* or *BranchTee* or *Generator* or *Load* Specifies a node the storage should be connected to. In the case this parameter is of type *LVStation* an additional parameter, *voltage_level*, has to be provided to define which side of the LV station the storage is connected to.
  - 'distribute_storages_mv' Places one storage in each MV feeder if it reduces grid expansion costs. This method needs a given time series of active power. ToDo: Elaborate

  In case of more than one storage provide a dict where each entry represents a storage. Keys of the dictionary have to match the keys of the *timeseries* and *parameters* dictionaries,

values must contain the corresponding positioning strategy or node to connect the storage to.

- **parameters** (dict, optional) – Dictionary with the following optional storage parameters:

```
{
    'nominal_power': <float>, # in kW
    'max_hours': <float>, # in h
    'soc_initial': <float>, # in kWh
    'efficiency_in': <float>, # in per unit 0..1
    'efficiency_out': <float>, # in per unit 0..1
    'standing_loss': <float> # in per unit 0..1
}
```

  See *Storage* for more information on storage parameters. In case of more than one storage provide a dict where each entry represents a storage. Keys of the dictionary have to match the keys of the *timeseries* dictionary, values must contain the corresponding parameters dictionary specified above. Note: As edisgo currently only provides a power flow analysis storage parameters don't have any effect on the calculations, except of the nominal power of the storage. Default: {}.

- **voltage_level** (str or dict, optional) – This parameter only needs to be provided if any entry in *position* is of type *LVStation*. In that case *voltage_level* defines which side of the LV station the storage is connected to. Valid options are 'lv' and 'mv'. In case of more than one storage provide a dict specifying the voltage level for each storage that is to be connected to an LV station. Keys of the dictionary have to match the keys of the *timeseries* dictionary, values must contain the corresponding voltage level. Default: None.

- **timeseries_reactive_power** (pandas.Series or dict) – By default reactive power is set through the config file *config_timeseries* in sections *reactive_power_factor* specifying the power factor and *reactive_power_mode* specifying if inductive or capacitive reactive power is provided. If you want to over-write this behavior you can provide a reactive power time series in kvar here. Be aware that eDisGo uses the generator sign convention for storages (see *Definitions and units* section of the documentation for more information). Index of the series needs to be a pandas.DatetimeIndex. In case of more than one storage provide a dict where each entry represents a storage. Keys of the dictionary have to match the keys of the *timeseries* dictionary, values must contain the corresponding time series as pandas.Series.

class edisgo.grid.network.**TimeSeries**(*network*, ***kwargs*)

Bases: object

Defines time series for all loads and generators in network (if set).

Contains time series for loads (sector-specific), generators (technology-specific), and curtailment (technology-specific).

generation_fluctuating

DataFrame with active power feed-in time series for fluctuating renewables solar and wind, normalized with corresponding capacity. Time series can either be aggregated by technology type or by type and weather cell ID. In the first case columns of the DataFrame are 'solar' and 'wind'; in the second case columns need to be a pandas.MultiIndex with the first level containing the type and the second level the weather cell ID. Default: None.

Type pandas.DataFrame, optional

generation_dispatchable

DataFrame with time series for active power of each (aggregated) type of dispatchable generator normalized with corresponding capacity. Columns represent generator type:

- 'gas'
- 'coal'
- 'biomass'
- 'other'
- ...

Use 'other' if you don't want to explicitly provide every possible type. Default: None.

Type pandas.DataFrame, optional

generation_reactive_power

DataFrame with reactive power per technology and weather cell ID, normalized with the nominal active power. Time series can either be aggregated by technology type or by type and weather cell ID. In the first case columns of the DataFrame are 'solar' and 'wind'; in the second case columns need to be a pandas.MultiIndex with the first level containing the type and the second level the weather cell ID. If the technology doesn't contain weather cell information, i.e. if it is other than solar or wind generation, this second level can be left as a numpy Nan or a None. Default: None.

Type

pandas *pandasDataFrame<dataframe>*, optional

load

DataFrame with active power of load time series of each (cumulative) type of load, normalized with corresponding annual energy demand. Columns represent load type:

- 'residential'
- 'retail'
- 'industrial'
- 'agricultural'

Default: None.

Type pandas.DataFrame, optional

load_reactive_power

DataFrame with time series of normalized reactive power (normalized by annual energy demand) per load sector. Index needs to be a pandas.DatetimeIndex. Columns represent load type:

- 'residential'
- 'retail'
- 'industrial'
- 'agricultural'

Default: None.

Type pandas.DataFrame, optional

curtailment

In the case curtailment is applied to all fluctuating renewables this needs to be a DataFrame with active power curtailment time series. Time series can either be aggregated by technology type or by type and weather cell ID. In the first case columns of the DataFrame are 'solar' and 'wind'; in the second case

columns need to be a pandas.MultiIndex with the first level containing the type and the second level the weather cell ID. In the case curtailment is only applied to specific generators, this parameter needs to be a list of all generators that are curtailed. Default: None.

Type pandas.DataFrame or List, optional

timeindex
Can be used to define a time range for which to obtain the provided time series and run power flow analysis. Default: None.

Type pandas.DatetimeIndex, optional

See also:

timeseries getter in *Generator*, *GeneratorFluctuating* and *Load*.

generation_dispatchable
Get generation time series of dispatchable generators (only active power)

Returns See class definition for details.

Return type pandas.DataFrame

generation_fluctuating
Get generation time series of fluctuating renewables (only active power)

Returns See class definition for details.

Return type pandas.DataFrame

generation_reactive_power
Get reactive power time series for generators normalized by nominal active power.

Returns See class definition for details.

Return type pandas: *pandas.DataFrame<dataframe>*

load
Get load time series (only active power)

Returns See class definition for details.

Return type dict or pandas.DataFrame

load_reactive_power
Get reactive power time series for load normalized by annual consumption.

Returns See class definition for details.

Return type pandas: *pandas.DataFrame<dataframe>*

timeindex
param time_range: Time range of power flow analysis :type time_range: pandas.DatetimeIndex

Returns See class definition for details.

Return type pandas.DatetimeIndex

curtailment
Get curtailment time series of dispatchable generators (only active power)

Parameters **curtailment** (list or pandas.DataFrame) – See class definition for details.

Returns In the case curtailment is applied to all solar and wind generators curtailment time series either aggregated by technology type or by type and weather cell ID are returnded. In the first case columns of the DataFrame are 'solar' and 'wind'; in the second case columns need to be a pandas.MultiIndex with the first level containing the type and the second level

the weather cell ID. In the case curtailment is only applied to specific generators, curtailment time series of all curtailed generators, specified in by the column name are returned.

Return type pandas.DataFrame

timesteps_load_feedin_case
Contains residual load and information on feed-in and load case.

Residual load is calculated from total (load - generation) in the grid. Grid losses are not considered.

Feed-in and load case are identified based on the generation and load time series and defined as follows:

1. Load case: positive (load - generation) at HV/MV substation
2. Feed-in case: negative (load - generation) at HV/MV substation

See also *assign_load_feedin_case()*.

Parameters **timeseries_load_feedin_case** (pandas.DataFrame) – Dataframe with information on whether time step is handled as load case ('load_case') or feed-in case ('feedin_case') for each time step in *timeindex*. Index of the series is the *timeindex*.

Returns Series with information on whether time step is handled as load case ('load_case') or feed-in case ('feedin_case') for each time step in *timeindex*. Index of the dataframe is *timeindex*. Columns of the dataframe are 'residual_load' with (load - generation) in kW at HV/MV substation and 'case' with 'load_case' for positive residual load and 'feedin_case' for negative residual load.

Return type pandas.Series

class edisgo.grid.network.**Results**(*network*)
Bases: object

Power flow analysis results management

Includes raw power flow analysis results, history of measures to increase the grid's hosting capacity and information about changes of equipment.

network
The network is a container object holding all data.

Type *Network*

measures
List with the history of measures to increase grid's hosting capacity.

Parameters **measure** (str) – Measure to increase grid's hosting capacity. Possible options are 'grid_expansion', 'storage_integration', 'curtailment'.

Returns **measures** – A stack that details the history of measures to increase grid's hosting capacity. The last item refers to the latest measure. The key *original* refers to the state of the grid topology as it was initially imported.

Return type list

pfa_p
Active power results from power flow analysis in kW.

Holds power flow analysis results for active power for the last iteration step. Index of the DataFrame is a DatetimeIndex indicating the time period the power flow analysis was conducted for; columns of the DataFrame are the edges as well as stations of the grid topology.

Parameters **pypsa** (pandas.DataFrame) – Results time series of active power P in kW from the PyPSA network

Provide this if you want to set values. For retrieval of data do not pass an argument

Returns Active power results from power flow analysis

Return type pandas.DataFrame

pfa_q

Reactive power results from power flow analysis in kvar.

Holds power flow analysis results for reactive power for the last iteration step. Index of the DataFrame is a DatetimeIndex indicating the time period the power flow analysis was conducted for; columns of the DataFrame are the edges as well as stations of the grid topology.

Parameters **pypsa** (pandas.DataFrame) – Results time series of reactive power Q in kvar from the PyPSA network

Provide this if you want to set values. For retrieval of data do not pass an argument

Returns Reactive power results from power flow analysis

Return type pandas.DataFrame

pfa_v_mag_pu

Voltage deviation at node in p.u.

Holds power flow analysis results for relative voltage deviation for the last iteration step. Index of the DataFrame is a DatetimeIndex indicating the time period the power flow analysis was conducted for; columns of the DataFrame are the nodes as well as stations of the grid topology.

Parameters **pypsa** (pandas.DataFrame) – Results time series of voltage deviation in p.u. from the PyPSA network

Provide this if you want to set values. For retrieval of data do not pass an argument

Returns Voltage level nodes of grid

Return type pandas.DataFrame

i_res

Current results from power flow analysis in A.

Holds power flow analysis results for current for the last iteration step. Index of the DataFrame is a DatetimeIndex indicating the time period the power flow analysis was conducted for; columns of the DataFrame are the edges as well as stations of the grid topology.

Parameters **pypsa** (pandas.DataFrame) – Results time series of current in A from the PyPSA network

Provide this if you want to set values. For retrieval of data do not pass an argument

Returns Current results from power flow analysis

Return type pandas.DataFrame

equipment_changes

Tracks changes in the equipment (e.g. replaced or added cable, etc.)

The DataFrame is indexed by the component(*Line*, *Station*, etc.) and has the following columns:

equipment : detailing what was changed (line, station, storage, curtailment). For ease of referencing we take the component itself. For lines we take the line-dict, for stations the transformers, for storages the storage-object itself and for curtailment either a dict providing the details of curtailment or a curtailment object if this makes more sense (has to be defined).

change [str] Specifies if something was added or removed.

iteration_step [int] Used for the update of the pypsa network to only consider changes since the last power flow analysis.

quantity [int] Number of components added or removed. Only relevant for calculation of grid expansion costs to keep track of how many new standard lines were added.

Parameters **changes** (pandas.DataFrame) – Provide this if you want to set values. For retrieval of data do not pass an argument.

Returns Equipment changes

Return type pandas.DataFrame

grid_expansion_costs

Holds grid expansion costs in kEUR due to grid expansion measures tracked in self.equipment_changes and calculated in edisgo.flex_opt.costs.grid_expansion_costs()

Parameters **total_costs** (pandas.DataFrame) – DataFrame containing type and costs plus in the case of lines the line length and number of parallel lines of each reinforced transformer and line. Provide this if you want to set grid_expansion_costs. For retrieval of costs do not pass an argument.

Index of the DataFrame is the respective object that can either be a *Line* or a *Transformer*. Columns are the following:

type [str] Transformer size or cable name

total_costs [float] Costs of equipment in kEUR. For lines the line length and number of parallel lines is already included in the total costs.

quantity [int] For transformers quantity is always one, for lines it specifies the number of parallel lines.

line_length [float] Length of line or in case of parallel lines all lines in km.

voltage_level [str] Specifies voltage level the equipment is in ('lv', 'mv' or 'mv/lv').

mv_feeder [*Line*] First line segment of half-ring used to identify in which feeder the grid expansion was conducted in.

Returns Costs of each reinforced equipment in kEUR.

Return type pandas.DataFrame

Notes

Total grid expansion costs can be obtained through costs.total_costs.sum().

grid_losses

Holds active and reactive grid losses in kW and kvar, respectively.

Parameters **pypsa_grid_losses** (pandas.DataFrame) – Dataframe holding active and reactive grid losses in columns 'p' and 'q' and in kW and kvar, respectively. Index is a pandas.DatetimeIndex.

Returns Dataframe holding active and reactive grid losses in columns 'p' and 'q' and in kW and kvar, respectively. Index is a pandas.DatetimeIndex.

Return type pandas.DataFrame

Notes

Grid losses are calculated as follows:

$$P_{loss} = \sum feed-in - \sum load + P_{slack} Q_{loss} = \sum feed-in - \sum load + Q_{slack}$$

As the slack is placed at the secondary side of the HV/MV station losses do not include losses of the HV/MV transformers.

hv_mv_exchanges
Holds active and reactive power exchanged with the HV grid.

The exchanges are essentially the slack results. As the slack is placed at the secondary side of the HV/MV station, this gives the energy transferred to and taken from the HV grid at the secondary side of the HV/MV station.

Parameters **hv_mv_exchanges** (pandas.DataFrame) – Dataframe holding active and reactive power exchanged with the HV grid in columns 'p' and 'q' and in kW and kvar, respectively. Index is a pandas.DatetimeIndex.

Returns Dataframe holding active and reactive power exchanged with the HV grid in columns 'p' and 'q' and in kW and kvar, respectively. Index is a pandas.DatetimeIndex.

Return type pandas:'pandas.DataFrame<dataframe>

curtailment
Holds curtailment assigned to each generator per curtailment target.

Returns Keys of the dictionary are generator types (and weather cell ID) curtailment targets were given for. E.g. if curtailment is provided as a pandas.DataFrame with :pandas.'pandas.MultiIndex' columns with levels 'type' and 'weather cell ID' the dictionary key is a tuple of ('type','weather_cell_id'). Values of the dictionary are dataframes with the curtailed power in kW per generator and time step. Index of the dataframe is a pandas.DatetimeIndex. Columns are the generators of type *edisgo.grid.components.GeneratorFluctuating*.

Return type `dict` with pandas.DataFrame

storages
Gathers relevant storage results.

Returns

Dataframe containing all storages installed in the MV grid and LV grids. Index of the dataframe are the storage representatives, columns are the following:

nominal_power [`float`] Nominal power of the storage in kW.

voltage_level [`str`] Voltage level the storage is connected to. Can either be 'mv' or 'lv'.

Return type pandas.DataFrame

storages_timeseries()
Returns a dataframe with storage time series.

Returns Dataframe containing time series of all storages installed in the MV grid and LV grids. Index of the dataframe is a pandas.DatetimeIndex. Columns are the storage representatives.

Return type pandas.DataFrame

storages_costs_reduction
Contains costs reduction due to storage integration.

Parameters **costs_df** (pandas.DataFrame) – Dataframe containing grid expansion costs in kEUR before and after storage integration in columns 'grid_expansion_costs_initial' and 'grid_expansion_costs_with_storages', respectively. Index of the dataframe is the MV grid id.

Returns Dataframe containing grid expansion costs in kEUR before and after storage integration in columns 'grid_expansion_costs_initial' and 'grid_expansion_costs_with_storages', respectively. Index of the dataframe is the MV grid id.

Return type pandas.DataFrame

unresolved_issues

Holds lines and nodes where over-loading or over-voltage issues could not be solved in grid reinforcement.

In case over-loading or over-voltage issues could not be solved after maximum number of iterations, grid reinforcement is not aborted but grid expansion costs are still calculated and unresolved issues listed here.

Parameters **issues** (*dict*) – Dictionary of critical lines/stations with relative over-loading and critical nodes with voltage deviation in p.u.. Format:

```
{crit_line_1: rel_overloading_1, ...,
 crit_line_n: rel_overloading_n,
 crit_node_1: v_mag_pu_node_1, ...,
 crit_node_n: v_mag_pu_node_n}
```

Provide this if you want to set unresolved_issues. For retrieval of unresolved issues do not pass an argument.

Returns Dictionary of critical lines/stations with relative over-loading and critical nodes with voltage deviation in p.u.

Return type Dictionary

s_res(*components=None*)

Get resulting apparent power in kVA at line(s) and transformer(s).

The apparent power at a line (or transformer) is determined from the maximum values of active power P and reactive power Q.

$$S = max(\sqrt{p_0^2 + q_0^2}, \sqrt{p_1^2 + q_1^2})$$

Parameters **components** (list) – List with all components (of type *Line* or *Transformer*) to get apparent power for. If not provided defaults to return apparent power of all lines and transformers in the grid.

Returns Apparent power in kVA for lines and/or transformers.

Return type pandas.DataFrame

v_res(*nodes=None*, *level=None*)

Get voltage results (in p.u.) from power flow analysis.

Parameters

- **nodes** (*Load*, *Generator*, etc. or list) – Grid topology component or list of grid topology components. If not provided defaults to column names available in grid level *level*.
- **level** (*str*) – Either 'mv' or 'lv' or None (default). Depending on which grid level results you are interested in. It is required to provide this argument in order to distinguish voltage levels at primary and secondary side of the transformer/LV station. If not provided (respectively None) defaults to ['mv', 'lv'].

Returns Resulting voltage levels obtained from power flow analysis

Return type pandas.DataFrame

Notes

Limitation: When power flow analysis is performed for MV only (with aggregated LV loads and generators) this methods only returns voltage at secondary side busbar and not at load/generator.

save(*directory*, *parameters='all'*)

Saves results to disk.

Depending on which results are selected and if they exist, the following directories and files are created:

- *powerflow_results* directory
  - *voltages_pu.csv*

    See `pfa_v_mag_pu` for more information.
  - *currents.csv*

    See `i_res()` for more information.
  - *active_powers.csv*

    See `pfa_p` for more information.
  - *reactive_powers.csv*

    See `pfa_q` for more information.
  - *apparent_powers.csv*

    See `s_res()` for more information.
  - *grid_losses.csv*

    See `grid_losses` for more information.
  - *hv_mv_exchanges.csv*

    See `hv_mv_exchanges` for more information.
- *pypsa_network* directory

  See `pypsa.Network.export_to_csv_folder()`
- *grid_expansion_results* directory
  - *grid_expansion_costs.csv*

    See `grid_expansion_costs` for more information.
  - *equipment_changes.csv*

    See `equipment_changes` for more information.
  - *unresolved_issues.csv*

    See `unresolved_issues` for more information.
- *curtailment_results* directory

  Files depend on curtailment specifications. There will be one file for each curtailment specification, that is for every key in `curtailment` dictionary.
- *storage_integration_results* directory

– *storages.csv*

See `storages()` for more information.

Parameters

- **directory** (`str`) – Directory to save the results in.
- **parameters** (`str` or `list` of `str`) – Specifies which results will be saved. By default all results are saved. To only save certain results set *parameters* to one of the following options or choose several options by providing a list:
  – 'pypsa_network'
  – 'powerflow_results'
  – 'grid_expansion_results'
  – 'curtailment_results'
  – 'storage_integration_results'

class edisgo.grid.network.**NetworkReimport**(*results_path*, ***kwargs*)
Bases: `object`

Network class created from saved results.

class edisgo.grid.network.**ResultsReimport**(*results_path*, *parameters='all'*)
Bases: `object`

Results class created from saved results.

v_res(*nodes=None*, *level=None*)
Get resulting voltage level at node.

Parameters

- **nodes** (`list`) – List of string representatives of grid topology components, e.g. `Generator`. If not provided defaults to all nodes available in grid level *level*.
- **level** (`str`) – Either 'mv' or 'lv' or None (default). Depending on which grid level results you are interested in. It is required to provide this argument in order to distinguish voltage levels at primary and secondary side of the transformer/LV station. If not provided (respectively None) defaults to ['mv', 'lv'].

Returns Resulting voltage levels obtained from power flow analysis

Return type pandas.DataFrame

s_res(*components=None*)
Get apparent power in kVA at line(s) and transformer(s).

Parameters **components** (`list`) – List of string representatives of `Line` or `Transformer`. If not provided defaults to return apparent power of all lines and transformers in the grid.

Returns Apparent power in kVA for lines and/or transformers.

Return type pandas.DataFrame

storages_timeseries()
Returns a dataframe with storage time series.

Returns Dataframe containing time series of all storages installed in the MV grid and LV grids. Index of the dataframe is a pandas.DatetimeIndex. Columns are the storage representatives.

Return type pandas.DataFrame

edisgo.grid.tools module

edisgo.grid.tools.**position_switch_disconnectors**(*mv_grid*, *mode='load'*, *status='open'*)

Determine position of switch disconnector in MV grid rings

Determination of the switch disconnector location is motivated by placing it to minimized load flows in both parts of the ring (half-rings). The switch disconnecter will be installed to a LV station, unless none exists in a ring. In this case, a node of arbitrary type is chosen for the location of the switch disconnecter.

Parameters

- **mv_grid** (*MVGrid*) – MV grid instance
- **mode** (*str*) – Define modus switch disconnector positioning: can be performed based of 'load', 'generation' or both 'loadgen'. Defaults to 'load'
- **status** (*str*) – Either 'open' or 'closed'. Define which status is should be set initially. Defaults to 'open' (which refers to conditions of normal grid operation).

Returns A tuple of size 2 specifying their pair of nodes between which the switch disconnector is located. The first node specifies the node that actually includes the switch disconnector.

Return type tuple

Notes

This function uses *nx.algorithms.find_cycle()* to identify nodes that are part of the MV grid ring(s). Make sure grid topology data that is provided has closed rings. Otherwise, no location for a switch disconnector can be identified.

edisgo.grid.tools.**implement_switch_disconnector**(*mv_grid*, *node1*, *node2*)

Install switch disconnector in grid topology

The graph that represents the grid's topology is altered in such way that it explicitly includes a switch disconnector. The switch disconnector is always located at `node1`. Technically, it does not make any difference. This is just an convention ensuring consistency of multiple runs.

The ring is still closed after manipulations of this function.

Parameters

- **mv_grid** (*MVGrid*) – MV grid instance
- **node1** – A rings node
- **node2** – Another rings node

edisgo.grid.tools.**select_cable**(*network*, *level*, *apparent_power*)

Selects an appropriate cable type and quantity using given apparent power.

Considers load factor.

Parameters

- **network** (*Network*) – The eDisGo container object
- **level** (`str`) – Grid level ('mv' or 'lv')
- **apparent_power** (`float`) – Apparent power the cable must carry in kVA

Returns

- pandas.Series – Cable type

- `int` – Cable count

Notes

Cable is selected to be able to carry the given *apparent_power*, no load factor is considered.

`edisgo.grid.tools.`**`get_gen_info`**(*network*, *level='mvlv'*, *fluctuating=False*)
Gets all the installed generators with some additional information.

Parameters

- **`network`** (*`Network`*) – Network object holding the grid data.
- **`level`** (`str`) – Defines which generators are returned. Possible options are:
  - 'mv' Only generators connected to the MV grid are returned.
  - 'lv' Only generators connected to the LV grids are returned.
  - 'mvlv' All generators connected to the MV grid and LV grids are returned.

  Default: 'mvlv'.
- **`fluctuating`** (`bool`) – If True only returns fluctuating generators. Default: False.

Returns

Dataframe with all generators connected to the specified voltage level. Index of the dataframe are the generator objects of type *`Generator`*. Columns of the dataframe are:

- 'gen_repr' The representative of the generator as `str`.
- 'type' The generator type, e.g. 'solar' or 'wind' as `str`.
- 'voltage_level' The voltage level the generator is connected to as `str`. Can either be 'mv' or 'lv'.
- 'nominal_capacity' The nominal capacity of the generator as as `float`.
- 'weather_cell_id' The id of the weather cell the generator is located in as `int` (only applies to fluctuating generators).

Return type pandas.DataFrame

`edisgo.grid.tools.`**`assign_mv_feeder_to_nodes`**(*mv_grid*)
Assigns an MV feeder to every generator, LV station, load, and branch tee

Parameters **`mv_grid`** (*`MVGrid`*) –

`edisgo.grid.tools.`**`get_mv_feeder_from_line`**(*line*)
Determines MV feeder the given line is in.

MV feeders are identified by the first line segment of the half-ring.

Parameters **`line`** (*`Line`*) – Line to find the MV feeder for.

Returns MV feeder identifier (representative of the first line segment of the half-ring)

Return type *`Line`*

`edisgo.grid.tools.`**`disconnect_storage`**(*network*, *storage*)
Removes storage from network graph and pypsa representation.

Parameters

- **`network`** (*`Network`*) –

- **storage** (*Storage*) – Storage instance to be removed.

Module contents

edisgo.tools package

Submodules

edisgo.tools.config module

This file is part of eDisGo, a python package for distribution grid analysis and optimization.

It is developed in the project open_eGo: https://openegoproject.wordpress.com

eDisGo lives on github: https://github.com/openego/edisgo/ The documentation is available on RTD: http://edisgo.readthedocs.io

Based on code by oemof developing group

This module provides a highlevel layer for reading and writing config files.

edisgo.tools.config.**load_config**(*filename*, *config_dir=None*, *copy_default_config=True*)
Loads the specified config file.

Parameters

- **filename** (str) – Config file name, e.g. 'config_grid.cfg'.
- **config_dir** (str, optional) – Path to config file. If None uses default edisgo config directory specified in config file 'config_system.cfg' in section 'user_dirs' by subsections 'root_dir' and 'config_dir'. Default: None.
- **copy_default_config** (*Boolean*) – If True copies a default config file into *config_dir* if the specified config file does not exist. Default: True.

edisgo.tools.config.**get**(*section*, *key*)
Returns the value of a given key of a given section of the main config file.

Parameters

- **section** (str) –
- **key** (str) –

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

Return type float or int or Boolean or str

edisgo.tools.config.**get_default_config_path**()
Returns the basic edisgo config path. If it does not yet exist it creates it and copies all default config files into it.

Returns Path to default edisgo config directory specified in config file 'config_system.cfg' in section 'user_dirs' by subsections 'root_dir' and 'config_dir'.

Return type str

edisgo.tools.config.**make_directory**(*directory*)
Makes directory if it does not exist.

Parameters **directory** (str) – Directory path

edisgo.tools.edisgo_run module

edisgo.tools.geo module

edisgo.tools.geo.**proj2equidistant**(*network*)
Defines conformal (e.g. WGS84) to ETRS (equidistant) projection Source CRS is loaded from Network's config.

Parameters **network** (*Network*) – The eDisGo container object

Returns

Return type functools.partial()

edisgo.tools.geo.**proj2conformal**(*network*)
Defines ETRS (equidistant) to conformal (e.g. WGS84) projection. Target CRS is loaded from Network's config.

Parameters **network** (*Network*) – The eDisGo container object

Returns

Return type functools.partial()

edisgo.tools.geo.**calc_geo_lines_in_buffer**(*network*, *node*, *grid*, *radius*, *radius_inc*)
Determines lines in nodes' associated graph that are at least partly within buffer of radius from node. If there are no lines, the buffer is successively extended by radius_inc until lines are found.

Parameters

- **network** (*Network*) – The eDisGo container object
- **node** (*Component*) – Origin node the buffer is created around (e.g. *Generator*). Node must be a member of grid's graph (grid.graph)
- **grid** (*Grid*) – Grid whose lines are searched
- **radius** (float) – Buffer radius in m
- **radius_inc** (float) – Buffer radius increment in m

Returns Sorted (by repr()) list of lines

Return type list of *Line*

Notes

Adapted from Ding0.

edisgo.tools.geo.**calc_geo_dist_vincenty**(*network*, *node_source*, *node_target*)
Calculates the geodesic distance between node_source and node_target incorporating the detour factor in config.

Parameters

- **network** (*Network*) – The eDisGo container object
- **node_source** (*Component*) – Node to connect (e.g. *Generator*)
- **node_target** (*Component*) – Target node (e.g. *BranchTee*)

Returns Distance in m

Return type float

edisgo.tools.plots module

edisgo.tools.plots.**histogram**(*data*, ***kwargs*)
Function to create histogram, e.g. for voltages or currents.

Parameters

- **data** (pandas.DataFrame) – Data to be plotted, e.g. voltage or current (*v_res* or *i_res* from *edisgo.grid.network.Results*). Index of the dataframe must be a pandas.DatetimeIndex.
- **timeindex** (pandas.Timestamp or list(pandas.Timestamp) or None, optional) – Specifies time steps histogram is plotted for. If timeindex is None all time steps provided in *data* are used. Default: None.
- **directory** (str or None, optional) – Path to directory the plot is saved to. Is created if it does not exist. Default: None.
- **filename** (str or None, optional) – Filename the plot is saved as. File format is specified by ending. If filename is None, the plot is shown. Default: None.
- **color** (str or None, optional) – Color used in plot. If None it defaults to blue. Default: None.
- **alpha** (float, optional) – Transparency of the plot. Must be a number between 0 and 1, where 0 is see through and 1 is opaque. Default: 1.
- **title** (str or None, optional) – Plot title. Default: None.
- **x_label** (str, optional) – Label for x-axis. Default: "".
- **y_label** (str, optional) – Label for y-axis. Default: "".
- **normed** (bool, optional) – Defines if histogram is normed. Default: False.
- **x_limits** (tuple or None, optional) – Tuple with x-axis limits. First entry is the minimum and second entry the maximum value. Default: None.
- **y_limits** (tuple or None, optional) – Tuple with y-axis limits. First entry is the minimum and second entry the maximum value. Default: None.
- **fig_size** (str or tuple, optional) –

  Size of the figure in inches or a string with the following options:

  - 'a4portrait'
  - 'a4landscape'
  - 'a5portrait'
  - 'a5landscape'

  Default: 'a5landscape'.
- **binwidth** (float) – Width of bins. Default: None.

edisgo.tools.plots.**add_basemap**(*ax*, *zoom=12*)
Adds map to a plot.

edisgo.tools.plots.**get_grid_district_polygon**(*config*, *subst_id=None*, *projection=4326*)
Get MV grid district polygon from oedb for plotting.

edisgo.tools.plots.**mv_grid_topology**(*pypsa_network*, *configs*, *timestep=None*, *line_color=None*, *node_color=None*, *line_load=None*, *grid_expansion_costs=None*, *filename=None*, *arrows=False*, *grid_district_geom=True*, *background_map=True*, *voltage=None*, *limits_cb_lines=None*, *limits_cb_nodes=None*, *xlim=None*, *ylim=None*, *lines_cmap='inferno_r'*, *title=''*, *scaling_factor_line_width=None*)

Plot line loading as color on lines.

Displays line loading relative to nominal capacity.

Parameters

- **pypsa_network** (pypsa.Network) –
- **configs** (dict) – Dictionary with used configurations from config files. See *Config* for more information.
- **timestep** (pandas.Timestamp) – Time step to plot analysis results for. If *timestep* is None maximum line load and if given, maximum voltage deviation, is used. In that case arrows cannot be drawn. Default: None.
- **line_color** (str or None) – Defines whereby to choose line colors (and implicitly size). Possible options are:
  - 'loading' Line color is set according to loading of the line. Loading of MV lines must be provided by parameter *line_load*.
  - 'expansion_costs' Line color is set according to investment costs of the line. This option also effects node colors and sizes by plotting investment in stations and setting *node_color* to 'storage_integration' in order to plot storage size of integrated storages. Grid expansion costs must be provided by parameter *grid_expansion_costs*.
  - None (default) Lines are plotted in black. Is also the fallback option in case of wrong input.
- **node_color** (str or None) – Defines whereby to choose node colors (and implicitly size). Possible options are:
  - 'technology' Node color as well as size is set according to type of node (generator, MV station, etc.).
  - 'voltage' Node color is set according to voltage deviation from 1 p.u.. Voltages of nodes in MV grid must be provided by parameter *voltage*.
  - 'storage_integration' Only storages are plotted. Size of node corresponds to size of storage.
  - None (default) Nodes are not plotted. Is also the fallback option in case of wrong input.
- **line_load** (pandas.DataFrame or None) – Dataframe with current results from power flow analysis in A. Index of the dataframe is a pandas.DatetimeIndex, columns are the line representatives. Only needs to be provided when parameter *line_color* is set to 'loading'. Default: None.
- **grid_expansion_costs** (pandas.DataFrame or None) – Dataframe with grid expansion costs in kEUR. See *grid_expansion_costs* in *Results* for more information. Only needs to be provided when parameter *line_color* is set to 'expansion_costs'. Default: None.
- **filename** (str) – Filename to save plot under. If not provided, figure is shown directly. Default: None.

- **arrows** (Boolean) – If True draws arrows on lines in the direction of the power flow. Does only work when *line_color* option 'loading' is used and a time step is given. Default: False.
- **grid_district_geom** (Boolean) – If True grid district polygon is plotted in the background. This also requires the geopandas package to be installed. Default: True.
- **background_map** (Boolean) – If True map is drawn in the background. This also requires the contextily package to be installed. Default: True.
- **voltage** (pandas.DataFrame) – Dataframe with voltage results from power flow analysis in p.u.. Index of the dataframe is a pandas.DatetimeIndex, columns are the bus representatives. Only needs to be provided when parameter *node_color* is set to 'voltage'. Default: None.
- **limits_cb_lines** (tuple) – Tuple with limits for colorbar of line color. First entry is the minimum and second entry the maximum value. Only needs to be provided when parameter *line_color* is not None. Default: None.
- **limits_cb_nodes** (tuple) – Tuple with limits for colorbar of nodes. First entry is the minimum and second entry the maximum value. Only needs to be provided when parameter *node_color* is not None. Default: None.
- **xlim** (tuple) – Limits of x-axis. Default: None.
- **ylim** (tuple) – Limits of y-axis. Default: None.
- **lines_cmap** (str) – Colormap to use for lines in case *line_color* is 'loading' or 'expansion_costs'. Default: 'inferno_r'.
- **title** (str) – Title of the plot. Default: ''.
- **scaling_factor_line_width** (float or None) – If provided line width is set according to the nominal apparent power of the lines. If line width is None a default line width of 2 is used for each line. Default: None.

edisgo.tools.pypsa_io module

This module provides tools to convert graph based representation of the grid topology to PyPSA data model. Call *to_pypsa()* to retrieve the PyPSA grid container.

edisgo.tools.pypsa_io.**to_pypsa**(*network*, *mode*, *timesteps*)

Translate graph based grid representation to PyPSA Network

For details from a user perspective see API documentation of *analyze()* of the API class *EDisGo*.

Translating eDisGo's grid topology to PyPSA representation is structured into translating the topology and adding time series for components of the grid. In both cases translation of MV grid only (*mode='mv'*), LV grid only (*mode='lv'*), MV and LV (*mode=None*) share some code. The code is organized as follows:

- Medium-voltage only (*mode='mv'*): All medium-voltage grid components are exported by *mv_to_pypsa()* including the LV station. LV grid load and generation is considered using *add_aggregated_lv_components()*. Time series are collected by *_pypsa_load_timeseries* (as example for loads, generators and buses) specifying *mode='mv'*). Timeseries for aggregated load/generation at substations are determined individually.
- Low-voltage only (*mode='lv'*): LV grid topology including the MV-LV transformer is exported. The slack is defind at primary side of the MV-LV transformer.
- Both level MV+LV (*mode=None*): The entire grid topology is translated to PyPSA in order to perform a complete power flow analysis in both levels together. First, both grid levels are translated seperately

using *mv_to_pypsa()* and *lv_to_pypsa()*. Those are merge by *combine_mv_and_lv()*. Time series are obtained at once for both grid levels.

This PyPSA interface is aware of translation errors and performs so checks on integrity of data converted to PyPSA grid representation

- Sub-graphs/ Sub-networks: It is ensured the grid has no islanded parts
- Completeness of time series: It is ensured each component has a time series
- Buses available: Each component (load, generator, line, transformer) is connected to a bus. The PyPSA representation is check for completeness of buses.
- Duplicate labels in components DataFrames and components' time series DataFrames

Parameters

- **network** (*Network*) – eDisGo grid container
- **mode** (*str*) – Determines grid levels that are translated to PyPSA grid representation. Specify
  - None to export MV and LV grid levels. None is the default.
  - ('mv' to export MV grid level only. This includes cumulative load and generation from underlying LV grid aggregated at respective LV station. This option is implemented, though the rest of edisgo does not handle it yet.)
  - **('lv' to export LV grid level only. This option is not yet** implemented)
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps to export to pypsa representation and use in power flow analysis.

Returns The PyPSA network container.

Return type pypsa.Network

edisgo.tools.pypsa_io.**mv_to_pypsa**(*network*)

Translate MV grid topology representation to PyPSA format

MV grid topology translated here includes

- MV station (no transformer, see *analyze()*)
- Loads, Generators, Lines, Storages, Branch Tees of MV grid level as well as LV stations. LV stations do not have load and generation of LV level.

Parameters **network** (Network) – eDisGo grid container

Returns

- dict of pandas.DataFrame – A DataFrame for each type of PyPSA components constituting the grid topology. Keys included
  - 'Generator'
  - 'Load'
  - 'Line'
  - 'BranchTee'
  - 'Transformer'
  - 'StorageUnit'

- .. *warning::* – PyPSA takes resistance R and reactance X in p.u. The conversion from values in ohm to pu notation is performed by following equations

$$r_{p.u.} = R_\Omega / Z_B$$
$$x_{p.u.} = X_\Omega / Z_B$$
$$with$$
$$Z_B = V_B / S_B$$

  I'm quite sure, but its not 100 % clear if the base voltage V_B is chosen correctly. We take the primary side voltage of transformer as the transformers base voltage. See #54 for discussion.

edisgo.tools.pypsa_io.**lv_to_pypsa**(*network*)
Convert LV grid topology to PyPSA representation

Includes grid topology of all LV grids of `lv_grids`

Parameters **network** (Network) – eDisGo grid container

Returns

A DataFrame for each type of PyPSA components constituting the grid topology. Keys included

- 'Generator'
- 'Load'
- 'Line'
- 'BranchTee'
- 'StorageUnit'

Return type dict of pandas.DataFrame

edisgo.tools.pypsa_io.**combine_mv_and_lv**(*mv*, *lv*)
Combine MV and LV grid topology in PyPSA format

edisgo.tools.pypsa_io.**add_aggregated_lv_components**(*network*, *components*)
Aggregates LV load and generation at LV stations

Use this function if you aim for MV calculation only. The according DataFrames of *components* are extended by load and generators representing these aggregated respecting the technology type.

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **components** (dict of pandas.DataFrame) – PyPSA components in tabular format

Returns The dictionary components passed to the function is returned altered.

Return type `dict` of pandas.DataFrame

edisgo.tools.pypsa_io.**process_pfa_results**(*network*, *pypsa*, *timesteps*)
Assing values from PyPSA to *results*

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **pypsa** (pypsa.Network) – The PyPSA Network container
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Time steps for which latest power flow analysis was conducted for and for which to retrieve pypsa results.

Notes

P and Q (and respectively later S) are returned from the line ending/ transformer side with highest apparent power S, exemplary written as

$$S_{max} = max(\sqrt{P0^2 + Q0^2}, \sqrt{P1^2 + Q1^2}) P = P0P1(S_{max}) Q = Q0Q1(S_{max})$$

See also:

Results Understand how results of power flow analysis are structured in eDisGo.

edisgo.tools.pypsa_io.**update_pypsa_generator_import**(*network*)

Translate graph based grid representation to PyPSA Network

For details from a user perspective see API documentation of *analyze()* of the API class *EDisGo*.

Translating eDisGo's grid topology to PyPSA representation is structured into translating the topology and adding time series for components of the grid. In both cases translation of MV grid only (*mode='mv'*), LV grid only (*mode='lv'*), MV and LV (*mode=None*) share some code. The code is organized as follows:

- Medium-voltage only (*mode='mv'*): All medium-voltage grid components are exported by *mv_to_pypsa()* including the LV station. LV grid load and generation is considered using *add_aggregated_lv_components()*. Time series are collected by *_pypsa_load_timeseries* (as example for loads, generators and buses) specifying *mode='mv'*). Timeseries for aggregated load/generation at substations are determined individually.
- Low-voltage only (*mode='lv'*): LV grid topology including the MV-LV transformer is exported. The slack is defind at primary side of the MV-LV transformer.
- Both level MV+LV (*mode=None*): The entire grid topology is translated to PyPSA in order to perform a complete power flow analysis in both levels together. First, both grid levels are translated seperately using *mv_to_pypsa()* and *lv_to_pypsa()*. Those are merge by *combine_mv_and_lv()*. Time series are obtained at once for both grid levels.

This PyPSA interface is aware of translation errors and performs so checks on integrity of data converted to PyPSA grid representation

- Sub-graphs/ Sub-networks: It is ensured the grid has no islanded parts
- Completeness of time series: It is ensured each component has a time series
- Buses available: Each component (load, generator, line, transformer) is connected to a bus. The PyPSA representation is check for completeness of buses.
- Duplicate labels in components DataFrames and components' time series DataFrames

Parameters

- **network** (*Network*) – eDisGo grid container
- **mode** (*str*) – Determines grid levels that are translated to PyPSA grid representation. Specify
  - None to export MV and LV grid levels. None is the default.
  - ('mv' to export MV grid level only. This includes cumulative load and generation from underlying LV grid aggregated at respective LV station. This option is implemented, though the rest of edisgo does not handle it yet.)
  - **('lv' to export LV grid level only. This option is not yet** implemented)

- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps to export to pypsa representation and use in power flow analysis.

Returns

The PyPSA network container.

Return type pypsa.Network

edisgo.tools.pypsa_io.**update_pypsa_grid_reinforcement**(*network*, *equipment_changes*)

Update equipment data of lines and transformers after grid reinforcement.

During grid reinforcement (cf. *edisgo.flex_opt.reinforce_grid.reinforce_grid()*) grid topology and equipment of lines and transformers are changed. In order to save time and not do a full translation of eDisGo's grid topology to the PyPSA format, this function provides an updater for data that may change during grid reinforcement.

The PyPSA grid topology *edisgo.grid.network.Network.pypsa()* is update by changed equipment stored in `edisgo.grid.network.Network.equipment_changes`.

Parameters

- **network** (Network) – eDisGo grid container
- **equipment_changes** (*pandas.DataFrame<dataframe>*) – Dataframe with latest equipment changes (of latest iteration step) from grid reinforcement. See *equipment_changes* property of *Results* for more information on the Dataframe.

edisgo.tools.pypsa_io.**update_pypsa_storage**(*pypsa*, *storages*, *storages_lines*)

Adds storages and their lines to pypsa representation of the edisgo graph.

This function effects the following attributes of the pypsa network: components ('StorageUnit'), storage_units, storage_units_t (p_set, q_set), buses, lines

Parameters

- **pypsa** (pypsa.Network) –
- **storages** (list) – List with storages of type *Storage* to add to pypsa network.
- **storages_lines** (list) – List with lines of type *Line* that connect storages to the grid.

edisgo.tools.pypsa_io.**update_pypsa_timeseries**(*network*, *loads_to_update=None*, *generators_to_update=None*, *storages_to_update=None*, *timesteps=None*)

Updates load, generator, storage and bus time series in pypsa network.

See functions *update_pypsa_load_timeseries()*, *update_pypsa_generator_timeseries()*, *update_pypsa_storage_timeseries()*, and *update_pypsa_bus_timeseries()* for more information.

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **loads_to_update** (list, optional) – List with all loads (of type *Load*) that need to be updated. If None all loads are updated depending on mode. See `to_pypsa()` for more information.
- **generators_to_update** (list, optional) – List with all generators (of type *Generator*) that need to be updated. If None all generators are updated depending on mode. See `to_pypsa()` for more information.

- **storages_to_update** (list, optional) – List with all storages (of type *Storage*) that need to be updated. If None all storages are updated depending on mode. See to_pypsa() for more information.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps of the load time series to export to pypsa representation and use in power flow analysis. If None all time steps currently existing in pypsa representation are updated. If not None current time steps are overwritten by given time steps. Default: None.

edisgo.tools.pypsa_io.**update_pypsa_load_timeseries**(*network*, *loads_to_update=None*, *timesteps=None*)

Updates load time series in pypsa representation.

This function overwrites p_set and q_set of loads_t attribute of pypsa network. Be aware that if you call this function with *timesteps* and thus overwrite current time steps it may lead to inconsistencies in the pypsa network since only load time series are updated but none of the other time series or the snapshots attribute of the pypsa network. Use the function *update_pypsa_timeseries()* to change the time steps you want to analyse in the power flow analysis. This function will also raise an error when a load that is currently not in the pypsa representation is added.

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **loads_to_update** (list, optional) – List with all loads (of type *Load*) that need to be updated. If None all loads are updated depending on mode. See to_pypsa() for more information.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps of the load time series to export to pypsa representation. If None all time steps currently existing in pypsa representation are updated. If not None current time steps are overwritten by given time steps. Default: None.

edisgo.tools.pypsa_io.**update_pypsa_generator_timeseries**(*network*, *generators_to_update=None*, *timesteps=None*)

Updates generator time series in pypsa representation.

This function overwrites p_set and q_set of generators_t attribute of pypsa network. Be aware that if you call this function with *timesteps* and thus overwrite current time steps it may lead to inconsistencies in the pypsa network since only generator time series are updated but none of the other time series or the snapshots attribute of the pypsa network. Use the function *update_pypsa_timeseries()* to change the time steps you want to analyse in the power flow analysis. This function will also raise an error when a generator that is currently not in the pypsa representation is added.

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **generators_to_update** (list, optional) – List with all generators (of type *Generator*) that need to be updated. If None all generators are updated depending on mode. See to_pypsa() for more information.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps of the generator time series to export to pypsa representation. If None all time steps currently existing in pypsa representation are updated. If not None current time steps are overwritten by given time steps. Default: None.

edisgo.tools.pypsa_io.**update_pypsa_storage_timeseries**(*network*, *storages_to_update=None*, *timesteps=None*)

Updates storage time series in pypsa representation.

This function overwrites p_set and q_set of storage_unit_t attribute of pypsa network. Be aware that if you call this function with *timesteps* and thus overwrite current time steps it may lead to inconsistencies in the pypsa network since only storage time series are updated but none of the other time series or the snapshots attribute of the pypsa network. Use the function *update_pypsa_timeseries()* to change the time steps you want to analyse in the power flow analysis. This function will also raise an error when a storage that is currently not in the pypsa representation is added.

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **storages_to_update** (list, optional) – List with all storages (of type *Storage*) that need to be updated. If None all storages are updated depending on mode. See to_pypsa() for more information.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps of the storage time series to export to pypsa representation. If None all time steps currently existing in pypsa representation are updated. If not None current time steps are overwritten by given time steps. Default: None.

edisgo.tools.pypsa_io.**update_pypsa_bus_timeseries**(*network*, *timesteps=None*)

Updates buses voltage time series in pypsa representation.

This function overwrites v_mag_pu_set of buses_t attribute of pypsa network. Be aware that if you call this function with *timesteps* and thus overwrite current time steps it may lead to inconsistencies in the pypsa network since only bus time series are updated but none of the other time series or the snapshots attribute of the pypsa network. Use the function *update_pypsa_timeseries()* to change the time steps you want to analyse in the power flow analysis.

Parameters

- **network** (Network) – The eDisGo grid topology model overall container
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps of the time series to export to pypsa representation. If None all time steps currently existing in pypsa representation are updated. If not None current time steps are overwritten by given time steps. Default: None.

edisgo.tools.tools module

edisgo.tools.tools.**select_worstcase_snapshots**(*network*)

Select two worst-case snapshots from time series

Two time steps in a time series represent worst-case snapshots. These are

1. **Load case: refers to the point in the time series where the** (load - generation) achieves its maximum and is greater than 0.
2. **Feed-in case: refers to the point in the time series where the** (load - generation) achieves its minimum and is smaller than 0.

These two points are identified based on the generation and load time series. In case load or feed-in case don't exist None is returned.

Parameters **network** (*Network*) – Network for which worst-case snapshots are identified.

Returns Dictionary with keys 'load_case' and 'feedin_case'. Values are corresponding worst-case snapshots of type pandas.Timestamp or None.

Return type dict

edisgo.tools.tools.**get_residual_load_from_pypsa_network**(*pypsa_network*)
Calculates residual load in MW in MV grid and underlying LV grids.

Parameters **pypsa_network** (pypsa.Network) – The PyPSA network container, containing load flow results.

Returns Series with residual load in MW for each time step. Positiv values indicate a higher demand than generation and vice versa. Index of the series is a pandas.DatetimeIndex

Return type pandas.Series

edisgo.tools.tools.**assign_load_feedin_case**(*network*)
For each time step evaluate whether it is a feed-in or a load case.

Feed-in and load case are identified based on the generation and load time series and defined as follows:

1. Load case: positive (load - generation) at HV/MV substation
2. Feed-in case: negative (load - generation) at HV/MV substation

Output of this function is written to *timesteps_load_feedin_case* attribute of the network.timeseries (see *TimeSeries*).

Parameters **network** (*Network*) – Network for which worst-case snapshots are identified.

Returns Dataframe with information on whether time step is handled as load case ('load_case') or feed-in case ('feedin_case') for each time step in *timeindex* attribute of network.timeseries. Index of the dataframe is network.timeseries.timeindex. Columns of the dataframe are 'residual_load' with (load - generation) in kW at HV/MV substation and 'case' with 'load_case' for positive residual load and 'feedin_case' for negative residual load.

Return type pandas.DataFrame

edisgo.tools.tools.**calculate_relative_line_load**(*network*, *configs*, *line_load*, *line_voltages*, *lines=None*, *timesteps=None*)
Calculates relative line loading.

Line loading is calculated by dividing the current at the given time step by the allowed current.

Parameters

- **network** (pypsa.Network) – Pypsa network with lines to calculate line loading for.
- **configs** (dict) – Dictionary with used configurations from config files. See *Config* for more information.
- **line_load** (pandas.DataFrame) – Dataframe with current results from power flow analysis in A. Index of the dataframe is a pandas.DatetimeIndex, columns are the line representatives.
- **line_voltages** (pandas.Series) – Series with nominal voltages of lines in kV. Index of the dataframe are the line representatives.
- **lines** (*list(str) or None, optional*) – Line names/representatives of lines to calculate line loading for. If None line loading of all lines in *line_load* dataframe are used. Default: None.
- **timesteps** (pandas.Timestamp or list(pandas.Timestamp) or None, optional) – Specifies time steps to calculate line loading for. If timesteps is None all time steps in *line_load* dataframe are used. Default: None.

Returns Dataframe with relative line loading (unitless). Index of the dataframe is a pandas.DatetimeIndex, columns are the line representatives.

Return type pandas.DataFrame

Module contents

edisgo.tools.**session_scope**()
Function to ensure that sessions are closed properly.

9.1.2 Module contents

9.2 edisgo

CHAPTER 10

What's New

Changelog for each release.

10.1 Release v0.0.10

Release date: October 18, 2019

10.1.1 Changes

- Updated to networkx 2.0
- Changed data of transformers #240
- Proper session handling and readonly usage (PR #160)

10.1.2 Bug fixes

- Corrected calculation of current from pypsa power flow results (PR #153).

10.2 Release v0.0.9

Release date: December 3, 2018

10.2.1 Changes

- bug fix in determining voltage deviation in LV stations and LV grid

10.3 Release v0.0.8

Release date: October 29, 2018

10.3.1 Changes

- added tolerance for curtailment targets slightly higher than generator availability to allow small rounding errors

10.4 Release v0.0.7

Release date: October 23, 2018

This release mainly focuses on new plotting functionalities and making reimporting saved results to further analyze and visualize them more comfortable.

10.4.1 Changes

- new plotting methods in the EDisGo API class (plottings of the MV grid topology showing line loadings, grid expansion costs, voltages and/or integrated storages and histograms for voltages and relative line loadings)
- new classes EDisGoReimport, NetworkReimport and ResultsReimport to reimport saved results and enable all analysis and plotting functionalities offered by the original classes
- bug fixes

10.5 Release v0.0.6

Release date: September 6, 2018

This release comes with a bunch of new features such as results output and visualization, speed-up options, a new storage integration methodology and an option to provide separate allowed voltage deviations for calculation of grid expansion needs. See list of changes below for more details.

10.5.1 Changes

- A methodolgy to integrate storages in the MV grid to reduce grid expansion costs was added that takes a given storage capacity and operation and allocates it to multiple smaller storages. This methodology is mainly to be used together with the eTraGo tool where an optimization of the HV and EHV levels is conducted to calculate optiomal storage size and operation at each HV/MV substation.
- The voltage-based curtailment methodolgy was adapted to take into account allowed voltage deviations and curtail generators with voltages that exceed the allowed voltage deviation more than generators with voltages that do not exceed the allowed voltage deviation.
- When conducting grid reinforcement it is now possible to apply separate allowed voltage deviations for different voltage levels (#108). Furthermore, an additional check was added at the end of the grid expansion methodology if the 10%-criterion was observed.
- To speed up calculations functions to update the pypsa representation of the edisgo graph after generator import, storage integration and time series update, e.g. after curtailment, were added.
- Also as a means to speed up calculations an option to calculate grid expansion costs for the two worst time steps, characterized by highest and lowest residual load at the HV/MV substation, was added.
- For the newly added storage integration methodology it was necessary to calculate grid expansion costs without changing the topology of the graph in order to identify feeders with high grid expansion needs. Therefore, the option to conduct grid reinforcement on a copy of the graph was added to the grid expansion function.
- So far loads and generators always provided or consumed inductive reactive power with the specified power factor. It is now possible to specify whether loads and generators should behave as inductors or capacitors and to provide a concrete reactive power time series(#131).
- The Results class was extended by outputs for storages, grid losses and active and reactive power at the HV/MV substation (#138) as well as by a function to save all results to csv files.
- A plotting function to plot line loading in the MV grid was added.
- Update ding0 version to v0.1.8 and include data processing v0.4.5 data
- Bug fix

10.6 Release v0.0.5

Release date: July 19, 2018

Most important changes in this release are some major bug fixes, a differentiation of line load factors and allowed voltage deviations for load and feed-in case in the grid reinforcement and a possibility to update time series in the pypsa representation.

10.6.1 Changes

- Switch disconnecters in MV rings will now be installed, even if no LV station exists in the ring #136
- Update to new version of ding0 v0.1.7
- Consider feed-in and load case in grid expansion methodology
- Enable grid expansion on snapshots
- Bug fixes

10.7 Release v0.0.3

Release date: July 6 2018

New features have been included in this release. Major changes being the use of the weather_cell_id and the inclusion of new methods for distributing the curtailment to be more suitable to network operations.

10.7.1 Changes

- As part of the solution to github issues #86, #98, Weather cell information was of importance due to the changes in the source of data. The table *ego_renewable_feedin_v031* is now used to provide this feedin time series indexed using the weather cell id's. Changes were made to ego.io and ding0 to correspondingly allow the use of this table by eDisGo.
- A new curtailment method have been included based on the voltages at the nodes with *GeneratorFluctuating* objects. The method is called *curtail_voltage* and its objective is to increase curtailment at locations where voltages are very high, thereby alleviating over-voltage issues and also reducing the need for network reinforcement.
- Add parallelization for custon functions #130
- Update ding0 version to v0.1.6 and include data processing v.4.2 data
- Bug Fixes

10.8 Release v0.0.2

Release date: March 15 2018

The code was heavily revised. Now, eDisGo provides the top-level API class `EDisGo` for user interaction. See below for details and other small changes.

10.8.1 Changes

- Switch disconnector/ disconnecting points are now relocated by eDisGo #99. Before, locations determined by Ding0 were used. Relocation is conducted according to minimal load differences in both parts of the ring.
- Switch disconnectors are always located in LV stations #23
- Made all round speed improvements as mentioned in the issues #43
- The structure of eDisGo and its input data has been extensively revised in order to make it more consistent and easier to use. We introduced a top-level API class called `EDisGo` through which all user input and measures are now handled. The EDisGo class thereby replaces the former Scenario class and parts of the Network class. See *A minimum working example* for a quick overview of how to use the EDisGo class or *Usage details* for a more comprehensive introduction to the edisgo structure and usage.
- We introduce a CLI script to use basic functionality of eDisGo including parallelization. CLI uses higher level functions to run eDisGo. Consult `edisgo_run` for further details. #93.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Bibliography

[DENA] A.C. Agricola et al.: *dena-Verteilnetzstudie: Ausbau- und Innovationsbedarf der Stromverteilnetze in Deutschland bis 2030*. 2012.

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W