eDisGo Documentation

Release 0.1.1

open$_{e}Go - Team$

Jul 22, 2022

Contents

1 Contents **3**
- 1.1 Getting started . . . 3
- 1.2 Usage details . . . 7
- 1.3 Features in detail . . . 12
- 1.4 Notes to developers . . . 17
- 1.5 Definition and units . . . 18
- 1.6 Default configuration data . . . 20
- 1.7 Equipment data . . . 26
- 1.8 API . . . 27
- 1.9 What's New . . . 95
- 1.10 Index . . . 99

Bibliography **101**

Python Module Index **103**

Index **105**

The python package eDisGo serves as a toolbox to evaluate flexibility measures as an economic alternative to conventional grid expansion in medium and low voltage grids.

The toolbox currently includes:

- Data import from external data sources
  - ding0 tool for synthetic medium and low voltage grid topologies for the whole of Germany
  - OpenEnergy DataBase (oedb) for feed-in time series of fluctuating renewables and scenarios for future power plant park of Germany
  - demandlib for electrical load time series
- Static, non-linear power flow analysis using PyPSA for grid issue identification
- Automatic grid reinforcement methodology solving overloading and voltage issues to determine grid expansion needs and costs based on measures most commonly taken by German distribution grid operators
- Multiperiod optimal power flow based on julia package PowerModels.jl optimizing storage positioning and/or operation as well as generator dispatch with regard to minimizing grid expansion costs
- Temporal complexity reduction
- Heuristic for grid-supportive generator curtailment
- Heuristic grid-supportive battery storage integration

Currently, a method to optimize the flexibility that can be provided by electric vehicles through controlled charging is being implemented. Prospectively, demand side management and reactive power management will be included.

See *Getting started* 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.

eDisGo was initially developed in the open_eGo research project as part of a grid planning tool that can be used to determine the optimal grid and storage expansion of the German power grid over all voltage levels and has been used in two publications of the project:

- Integrated Techno-Economic Power System Planning of Transmission and Distribution Grids
- Final report of the open_eGo project (in German)

CHAPTER 1

Contents

1.1 Getting started

1.1.1 Installation

Warning: Make sure to use python 3.7 or higher!

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

```
pip3 install edisgo
```

The above will install all packages for the basic usage of eDisGo. To install additional packages e.g. needed to create plots with background maps or to run the jupyter notebook examples, we provide installation with extra packages:

```
pip3 install edisgo[geoplot]  # for plotting with background maps
pip3 install edisgo[examples]  # to run examples
pip3 install edisgo[dev]  # developer packages
pip3 install edisgo[full]  # combines all of the extras above
```

You may also consider installing a developer version as detailed in *Notes to developers*.

Installation under Windows

For Windows users we recommend using Anaconda and to install the python package shapely using the conda-forge channel prior to installing eDisGo. You may use the provided eDisGo_env.yml file to do so. Download the file and create a virtual environment with:

```
conda env create -f path/to/eDisGo_env.yml
```

Activate the newly created environment with:

```
conda activate eDisGo_env
```

You can now install eDisGo using pip as described above, with or without extra packages.

Requirements for edisgoOPF package

To use the multiperiod optimal power flow that is provided in the julia package edisgoOPF in eDisGo you additionally need to install julia version 1.1.1. Download julia from julia download page and add it to your path (see platform specific instructions for more information).

Before using the edisgoOPF julia package for the first time you need to instantiate it. Therefore, in a terminal change directory to the edisgoOPF package located in eDisGo/edisgo/opf/edisgoOPF and call julia from there. Change to package mode by typing

```
]
```

Then activate the package:

```
(v1.0) pkg> activate .
```

And finally instantiate it:

```
(SomeProject) pkg> instantiate
```

Additional linear solver

As with the default linear solver in Ipopt (local solver used in the OPF) the limit for prolem sizes is reached quite quickly, you may want to instead use the solver HSL_MA97. The steps required to set up HSL are also described in the Ipopt Documentation. Here is a short version for reference:

First, you need to obtain an academic license for HSL Solvers. Under http://www.hsl.rl.ac.uk/ipopt/ download the sources for Coin-HSL Full (Stable). You will need to provide an institutional e-mail to gain access.

Unpack the tar.gz:

```
tar -xvzf coinhsl-2014.01.10.tar.gz
```

To install the solver, clone the Ipopt Third Party HSL tools:

```
git clone https://github.com/coin-or-tools/ThirdParty-HSL.git
cd ThirdParty-HSL
```

Under *ThirdParty-HSL*, create a folder for the HSL sources named *coinhsl* and copy the contents of the HSL archive into it. Under Ubuntu, you'll need BLAS, LAPACK and GCC for Fortran. If you don't have them, install them via:

```
sudo apt-get install libblas-dev liblapack-dev gfortran
```

You can then configure and install your HSL Solvers:

```
./configure
make
sudo make install
```

To make Ipopt pick up the solver, you need to add it to your path. During install, there will be an output that tells you where the libraries have been put. Usually like this:

```
Libraries have been installed in:
    /usr/local/lib
```

Add this path to the variable *LD_LIBRARY_PATH*:

```
export LD_LIBRARY="/usr/local/bin":$LD_LIBRARY_PATH
```

You might also want to add this to your .bashrc to make it persistent.

For some reason, Ipopt looks for a library named *libhsl.so*, which is not what the file is named, so we'll also need to provide a symlink:

```
cd /usr/local/lib
ln -s libcoinhsl.so libhsl.so
```

MA97 should now work and can be called from Julia with:

```
JuMP.setsolver(pm.model,IpoptSolver(linear_solver="ma97"))
```

1.1.2 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.

1.1.3 A minimum working example

Following you find short examples on how to use eDisGo. Further details are provided in *Usage details*. Further examples can be found in the examples directory.

All following examples assume you have a ding0 grid topology (directory containing csv files, defining the grid topology) in a directory "ding0_example_grid" in the directory from where you run your example.

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:

```
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_example_grid',
                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 using PyPSA)
edisgo.analyze()
```

(continues on next page)

(continued from previous page)

```
# Do grid reinforcement
edisgo.reinforce()

# Determine costs for each line/transformer that was reinforced
costs = edisgo.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),
     'retail': [0.0002] * len(timeindex),
     'industrial': [0.00015] * 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(
    ding0_grid='ding0_example_grid',
    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.results.grid_expansion_costs
```

Time series for loads 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(
    {'biomass': [1] * len(timeindex),
     'coal': [1] * len(timeindex),
     'other': [1] * len(timeindex)
     },
    index=timeindex)

edisgo = EDisGo(
    ding0_grid='ding0_example_grid',
    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.results.grid_expansion_costs
```

1.1.4 Parallelization

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

1.1.5 LICENSE

1.2 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. Below, we give a detailed introduction to the data structure and to how different features can be used.

1.2.1 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. It also provides access to all relevant data. Grid data is stored in the *Topology* class. Time series data can be found in the *TimeSeries* class. Results data holding results e.g. from the power flow analysis and grid expansion is stored in the *Results* class. Configuration data from the config files (see *Default configuration data*) is stored in the *Config* class. All these can be accessed through the *EDisGo* object. In the following code examples *edisgo* constitues an *EDisGo* object.

```
# Access Topology grid data container object
edisgo.topology

# Access TimeSeries data container object
edisgo.timeseries

# Access Results data container object
edisgo.results

# Access configuration data container object
edisgo.config
```

Grid data is stored in pandas.DataFrames in the *Topology* object. There are extra data frames for all grid elements (buses, lines, switches, transformers), as well as generators, loads and storage units. You can access those dataframes as follows:

```
# Access all buses in MV grid and underlying LV grids
edisgo.topology.buses_df

# Access all lines in MV grid and underlying LV grids
edisgo.topology.lines_df

# Access all MV/LV transformers
edisgo.topology.transformers_df

# Access all HV/MV transformers
edisgo.topology.transformers_hvmv_df

# Access all switches in MV grid and underlying LV grids
edisgo.topology.switches_df

# Access all generators in MV grid and underlying LV grids
edisgo.topology.generators_df

# Access all loads in MV grid and underlying LV grids
edisgo.topology.loads_df

# Access all storage units in MV grid and underlying LV grids
edisgo.topology.storage_units_df
```

The grids can also be accessed individually. The MV grid is stored in an *MVGrid* object and each LV grid in an *LVGrid* object. The MV grid topology can be accessed through

```
# Access MV grid
edisgo.topology.mv_grid
```

Its components can be accessed analog to those of the whole grid topology as shown above.

```
# Access all buses in MV grid
edisgo.topology.mv_grid.buses_df

# Access all generators in MV grid
edisgo.topology.mv_grid.generators_df
```

A list of all LV grids can be retrieved through:

```
# Get list of all underlying LV grids
# (Note that MVGrid.lv_grids returns a generator object that must first be
#  converted to a list in order to view the LVGrid objects)
list(edisgo.topology.mv_grid.lv_grids)
```

Access to a single LV grid's components can be obtained analog to shown above for the whole topology and the MV grid:

```
# Get single LV grid
lv_grid = list(edisgo.topology.mv_grid.lv_grids)[0]

# Access all buses in that LV grid
lv_grid.buses_df

# Access all loads in that LV grid
lv_grid.loads_df
```

A single grid's generators, loads, storage units and switches can also be retrieved as *Generator*, *Load*, *Storage*, and *Switch* objects, respecitvely:

```
# Get all switch disconnectors in MV grid as Switch objects
# (Note that objects are returned as a python generator object that must
#  first be converted to a list in order to view the Switch objects)
list(edisgo.topology.mv_grid.switch_disconnectors)

# Get all generators in LV grid as Generator objects
list(lv_grid.generators)
```

For some applications it is helpful to get a graph representation of the grid, e.g. to find the path from the station to a generator. The graph representation of the whole topology or each single grid can be retrieved as follows:

```
# Get graph representation of whole topology
edisgo.to_graph()

# Get graph representation for MV grid
edisgo.topology.mv_grid.graph

# Get graph representation for LV grid
lv_grid.graph
```

The returned graph is a networkx.Graph, where lines are represented by edges in the graph, and buses and transformers are represented by nodes.

1.2.2 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 a grid analysis and identifying grid issues (line overloading and voltage issues) using *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()`, that can be given as an input to the EDisGo object through the parameter *timeindex* or is otherwise set automatically. If you want to change the time steps that are analyzed, you can specify those through the parameter *timesteps* of the *analyze* function. Make sure that the specified time steps are a subset of `timeindex()`.

1.2.3 Grid expansion

Grid expansion 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_grid()` for more information.

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

```
# Get costs of grid expansion
costs = edisgo.results.grid_expansion_costs
```

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

1.2.4 Battery storage systems

Battery storage systems can be integrated into the grid as an alternative to classical grid expansion. The storage integration heuristic described in section *Storage integration* is not available at the moment. Instead, you may either integrate a storage unit at a specified bus manually or use the optimal power flow to optimally distribute a given storage capacity in the grid.

Here are two small examples on how to integrate a storage unit manually. In the first one, the EDisGo object is set up for a worst-case analysis, wherefore no time series needs to be provided for the storage unit, as worst-case definition is used. In the second example, a time series analysis is conducted, wherefore a time series for the storage unit needs to be provided.

```
from edisgo import EDisGo

# Set up EDisGo object
edisgo = EDisGo(ding0_grid=dingo_grid_path,
                worst_case_analysis='worst-case')

# Get random bus to connect storage to
random_bus = edisgo.topology.buses_df.index[3]
# Add storage instance
edisgo.add_component(
    "StorageUnit",
    bus=random_bus,
    p_nom=4)
```

```
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(
    {'biomass': [1] * len(timeindex),
     'coal': [1] * len(timeindex),
     'other': [1] * len(timeindex)
     },
    index=timeindex)
edisgo = EDisGo(
    ding0_grid='ding0_example_grid',
    generator_scenario='ego100',
    timeseries_load='demandlib',
    timeseries_generation_fluctuating='oedb',
    timeseries_generation_dispatchable=timeseries_generation_dispatchable,
    timeindex=timeindex)

# Get random bus to connect storage to
random_bus = edisgo.topology.buses_df.index[3]
# Add storage instance
edisgo.add_component(
    "StorageUnit",
    bus=random_bus,
    p_nom=4,
    ts_active_power=pd.Series(
        [-3.4, 2.5, -3.4, 2.5],
        index=edisgo.timeseries.timeindex))
```

Following is an example on how to use the OPF to find the optimal storage positions in the grid with regard to grid expansion costs. Storage operation is optimized at the same time. The example uses the same EDisGo instance as above. A total storage capacity of 10 MW is distributed in the grid. *storage_buses* can be used to specify certain buses storage units may be connected to. This does not need to be provided but will speed up the optimization.

```
random_bus = edisgo.topology.buses_df.index[3:13]
edisgo.perform_mp_opf(
    timesteps=period,
    scenario="storage",
    storage_units=True,
    storage_buses=busnames,
    total_storage_capacity=10.0,
    results_path=results_path)
```

1.2.5 Curtailment

The curtailment function is used to spatially distribute the power that is to be curtailed. The two heuristics *feedin-proportional* and *voltage-based*, in detail explained in section *Curtailment*, are currently not available. Instead you may use the optimal power flow to find the optimal generator curtailment with regard to minimizing grid expansion costs for given curtailment requirements. The following example again uses the EDisGo object from above.

```
edisgo.perform_mp_opf(
    timesteps=period,
```

(continues on next page)

(continued from previous page)

```
    scenario='curtailment',
    results_path=results_path,
    curtailment_requirement=True,
    curtailment_requirement_series=[10, 20, 15, 0])
```

1.2.6 Plots

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

```
# 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 *EDisGo* class for more plots and plotting options.

1.2.7 Results

Results such as voltages at nodes and line loading from the power flow analysis as well as grid expansion costs are provided through the *Results* class and can be accessed the following way:

```
edisgo.results
```

Get voltages at nodes from `v_res()` and line loading from `s_res()` or *i_res*. *equipment_changes* holds details about measures performed during grid expansion. Associated costs can be obtained through *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.results.save('path/to/results/directory/')
```

See `save()` for more information.

1.3 Features in detail

1.3.1 Power flow analysis

In order to analyse voltages and line loadings a non-linear power flow analysis (PF) using pypsa 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.

1.3.2 Multi period optimal power flow

Todo: Add

1.3.3 Grid expansion

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.

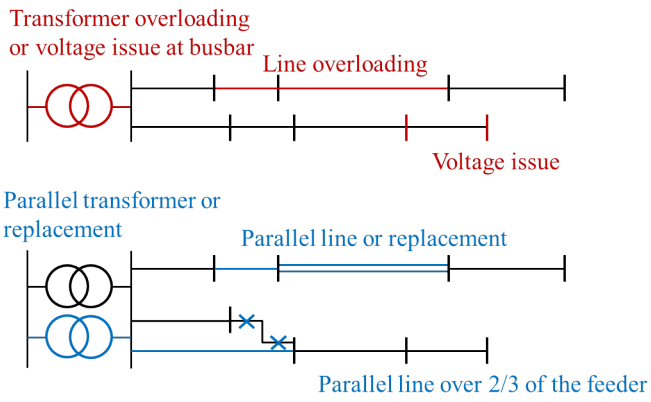

Fig. 1.1: Grid expansion measures

`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.

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 `timesteps_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*).

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_lines_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 *reinforce_hv_mv_station_overloading()* and *reinforce_mv_lv_station_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 *reinforce_mv_lv_station_voltage_issues()*. 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_lines_voltage_issues()*. 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.

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.

1.3.4 Curtailment

Warning: The curtailment methods are not yet adapted to the refactored code and therefore currently do not work.

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.

'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.

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

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.

'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.

1.3.5 Storage integration

Warning: The storage integration methods described below are not yet adapted to the refactored code and therefore currently do not work.

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.

1.3.6 References

1.4 Notes to developers

1.4.1 Installation

Clone repository from GitHub and install in developer mode:

```
pip3 install -e <path-to-repo>[full]
```

1.4.2 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

1.4.3 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
```

1.5 Definition and units

1.5.1 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.

Generator Sign Convention

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

Load Sign Convention

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

1.5.2 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.

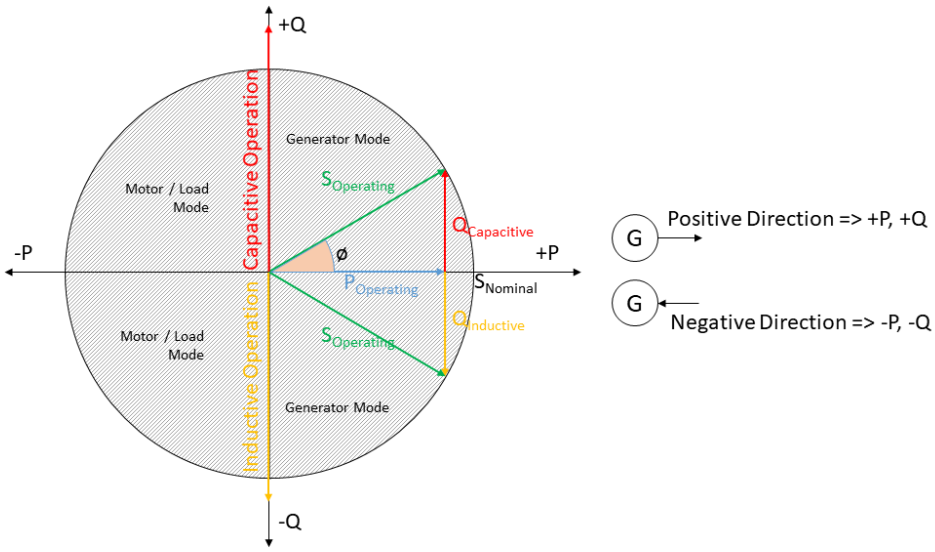

Fig. 1.2: Generator sign convention in detail

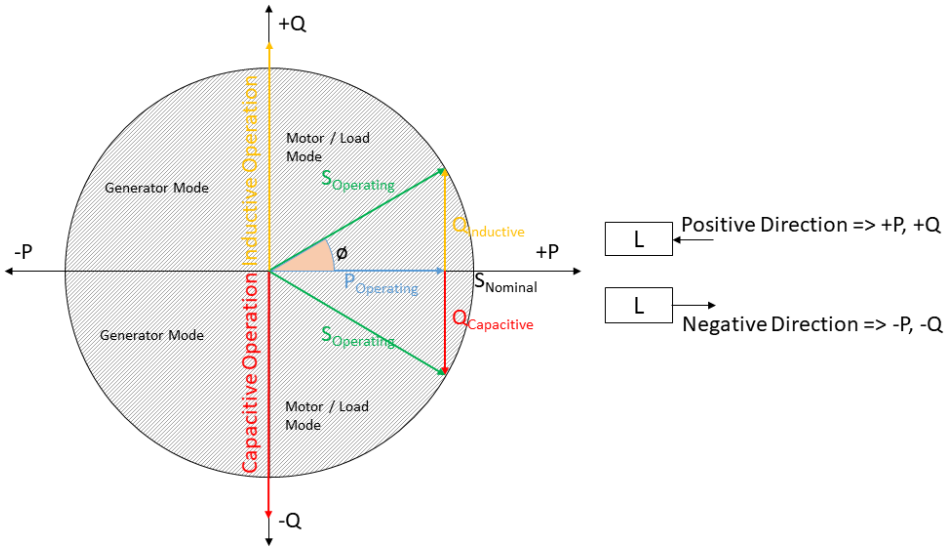

Fig. 1.3: Load sign convention in detail

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.

1.5.3 Units

Table 1.1: List of variables and units

Variable	Symbol	Unit	Comment
Current	I	kA	
Length	l	km	
Active Power	P	MW	
Reactive Power	Q	MVar	
Apparent Power	S	MVA	
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	

1.6 Default configuration data

Following you find the default configuration files.

1.6.1 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_
```

(continues on next page)

(continued from previous page)

```
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
#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
```

1.6.2 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
# ==========================
# relevant for all cases
feedin_case_lower = 0.9
```

(continues on next page)

(continued from previous page)

```
load_case_upper = 1.1

# 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

# 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
```

(continues on next page)

(continued from previous page)

```
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
# 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
```

1.6.3 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.
# Scale factors are used in worst-case scenarios.
# Power factors are used to generate reactive power time series.

[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_wind = 1
feedin_case_feedin_other = 1
load_case_feedin_pv = 0
load_case_feedin_wind = 0
load_case_feedin_other = 0

# temporary own values
feedin_case_storage = 1
load_case_storage = -1

[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
```

(continues on next page)

(continued from previous page)

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

1.6.4 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
```

(continues on next page)

(continued from previous page)

```
#     unit: m
conn_buffer_radius_inc = 1000

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


[disconnecting_point]

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

1.7 Equipment data

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

Table 1.2: LV cables

name	U_n	I_max_th	R_per_km	L_per_km
#-	kV	kA	ohm/km	mH/km
NAYY 4x1x300	0.4	0.419	0.1	0.279
NAYY 4x1x240	0.4	0.364	0.125	0.254
NAYY 4x1x185	0.4	0.313	0.164	0.256
NAYY 4x1x150	0.4	0.275	0.206	0.256
NAYY 4x1x120	0.4	0.245	0.253	0.256
NAYY 4x1x95	0.4	0.215	0.320	0.261
NAYY 4x1x50	0.4	0.144	0.449	0.270
NAYY 4x1x35	0.4	0.123	0.868	0.271

Table 1.3: MV cables

name	U_n	I_max_th	R_per_km	L_per_km	C_per_km
#-	kV	kA	ohm/km	mH/km	uF/km
NA2XS2Y 3x1x185 RM/25	10	0.357	0.164	0.38	0.41
NA2XS2Y 3x1x240 RM/25	10	0.417	0.125	0.36	0.47
NA2XS2Y 3x1x300 RM/25	10	0.466	0.1	0.35	0.495
NA2XS2Y 3x1x400 RM/35	10	0.535	0.078	0.34	0.57
NA2XS2Y 3x1x500 RM/35	10	0.609	0.061	0.32	0.63
NA2XS2Y 3x1x150 RE/25	20	0.319	0.206	0.4011	0.24
NA2XS2Y 3x1x240	20	0.417	0.13	0.3597	0.304
NA2XS(FL)2Y 3x1x300 RM/25	20	0.476	0.1	0.37	0.25
NA2XS(FL)2Y 3x1x400 RM/35	20	0.525	0.078	0.36	0.27
NA2XS(FL)2Y 3x1x500 RM/35	20	0.598	0.06	0.34	0.3

Table 1.4: MV overhead lines

name	U_n	I_max_th	R_per_km	L_per_km	C_per_km
#-	kV	kA	ohm/km	mH/km	uF/km
48-AL1/8-ST1A	10	0.21	0.35	1.11	0.0104
94-AL1/15-ST1A	10	0.35	0.33	1.05	0.0112
122-AL1/20-ST1A	10	0.41	0.31	0.99	0.0115
48-AL1/8-ST1A	20	0.21	0.37	1.18	0.0098
94-AL1/15-ST1A	20	0.35	0.35	1.11	0.0104
122-AL1/20-ST1A	20	0.41	0.34	1.08	0.0106

Table 1.5: LV transformers

name	S_nom	u_kr	P_k
#	MVA	%	MW
100 kVA	0.1	4	0.00175
160 kVA	0.16	4	0.00235
250 kVA	0.25	4	0.00325
400 kVA	0.4	4	0.0046
630 kVA	0.63	4	0.0065
800 kVA	0.8	6	0.0084
1000 kVA	1.0	6	0.00105

Table 1.6: MV transformers

name	S_nom
#	MVA
20 MVA	20
32 MVA	32
40 MVA	40
63 MVA	63

1.8 API

1.8.1 EDisGo class

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

Provides the top-level API for invocation of data import, power flow analysis, network reinforcement, flexibility measures, etc..

Parameters

- **ding0_grid** (str) – Path to directory containing csv files of network to be loaded.
- **generator_scenario** (None or str, optional) – If None, the generator park of the imported grid is kept as is. Otherwise defines which scenario of future generator park to use and invokes grid integration of these generators. Possible options are 'nep2035' and 'ego100'. These are scenarios from the research project open_eGo (see final report for more information on the scenarios). See import_generators for further information on how generators are integrated and what further options there are. Default: None.

- **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 and load for the two worst-case scenarios feed-in case and load case are generated.

  – 'worst-case-feedin'

    Feed-in and load for the worst-case scenario feed-in case is generated.

  – 'worst-case-load'

    Feed-in and load for the worst-case scenario load case is generated.

  Worst case scaling factors for loads and generators are specified in the config section *worst_case_scale_factor*.

  Be aware that if you choose to conduct a worst-case analysis your input for all other time series parameters (e.g. *timeseries_generation_fluctuating*, *timeseries_generation_dispatchable*, *timeseries_load*) will not be used. As eDisGo is designed to work with time series but worst cases are not time specific, a random time index 1/1/1970 is used.

  Default: None.

- **timeseries_generation_fluctuating** (str or pandas.DataFrame or None, optional) – Parameter used to obtain time series for active power feed-in of fluctuating renewables wind and solar. Possible options are:

  – 'oedb'

    Hourly time series for the year 2011 are obtained from the OpenEnergy DataBase. See *edisgo.io.timeseries_import.import_feedin_timeseries()* for more information.

  – pandas.DataFrame

    DataFrame with time series for active power feed-in, normalized to a capacity of 1 MW.

    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.

    When importing a ding0 grid and/or using predefined scenarios of the future generator park (see parameter *generator_scenario*), each generator has an assigned weather cell ID that identifies the weather data cell from the weather data set used in the research project open_eGo to determine feed-in profiles. The weather cell ID can be retrieved from column *weather_cell_id* in *generators_df* and could be overwritten to use own weather cells.

  Default: None.

- **timeseries_generation_dispatchable** (pandas.DataFrame or None, optional) – DataFrame with time series for active power of each type of dispatchable generator, normalized to a capacity of 1 MW.

  Index needs to be a pandas.DatetimeIndex.

Columns represent generator type (e.g. 'gas', 'coal', 'biomass'). All in the current grid existing generator types can be retrieved from column *type* in `generators_df`. Use 'other' if you don't want to explicitly provide every possible type.

Default: None.

- **timeseries_generation_reactive_power** (pandas.DataFrame or None, 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 containing the weather cell ID in the second level. If the technology doesn't contain weather cell information, i.e. if it is not a solar or wind generator, this second level can be left as a numpy Nan or a 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.

  Default: None.

- **timeseries_load** (`str` or pandas.DataFrame or None, optional) – Parameter used to obtain time series of active power of loads. Possible options are:

  – 'demandlib'

  Time series for the year specified in input parameter *timeindex* are generated using standard electric load profiles from the oemof demandlib.

  – pandas.DataFrame

  DataFrame with load time series of each type of load normalized with corresponding annual energy demand. Index needs to be a pandas.DatetimeIndex. Columns represent load type. The in the current grid existing load types can be retrieved from column *sector* in `loads_df`. In ding0 grids the differentiated sectors are 'residential', 'retail', 'industrial', and 'agricultural'.

  Default: None.

- **timeseries_load_reactive_power** (pandas.DataFrame or None, 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. The in the current grid existing load types can be retrieved from column *sector* in `loads_df`. In ding0 grids the differentiated sectors are 'residential', 'retail', 'industrial', and 'agricultural'.

  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.

  Default: None.

- **timeindex** (None or pandas.DatetimeIndex, optional) – 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.

- **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.

topology
The topology is a container object holding the topology of the grids.

Type *Topology*

timeseries
Container for component time series.

Type *TimeSeries*

results
This is a container holding all calculation results from power flow analyses, curtailment, storage integration, etc.

Type *Results*

config
eDisGo configuration data.

Returns Config object with configuration data from config files.

Return type *Config*

import_ding0_grid(*path*)
Import ding0 topology data from csv files in the format as Ding0 provides it.

Parameters **path** (*str*) – Path to directory containing csv files of network to be loaded.

to_pypsa(***kwargs*)
Convert to PyPSA network representation.

A network topology representation based on pandas.DataFrame. The overall container object of this data model, the pypsa.Network, is set up.

Parameters **kwargs** – See *to_pypsa()* for further information.

Returns PyPSA network representation.

Return type pypsa.Network

to_graph()
Returns graph representation of the grid.

Returns Graph representation of the grid as networkx Ordered Graph, where lines are represented by edges in the graph, and buses and transformers are represented by nodes.

Return type networkx.Graph

import_generators(*generator_scenario=None*, ***kwargs*)
Gets generator park for specified scenario and integrates them into the grid.

Currently, the only supported data source is scenario data generated in the research project open_eGo. You can choose between two scenarios: 'nep2035' and 'ego100'. You can get more information on the scenarios in the final report.

The generator data is retrieved from the open energy platform from tables for conventional power plants and renewable power plants.

When the generator data is retrieved, the following steps are conducted:

- Step 1: Update capacity of existing generators if ' update_existing' is True, which it is by default.
- Step 2: Remove decommissioned generators if *remove_decommissioned* is True, which it is by default.
- Step 3: Integrate new MV generators.
- Step 4: Integrate new LV generators.

For more information on how generators are integrated, see `connect_to_mv` and `connect_to_lv`.

After the generator park is changed there may be grid issues due to the additional in-feed. These are not solved automatically. If you want to have a stable grid without grid issues you can invoke the automatic grid expansion through the function `reinforce`.

Parameters **generator_scenario** (*str*) – Scenario for which to retrieve generator data. Possible options are 'nep2035' and 'ego100'.

Other Parameters **kwargs** – See *edisgo.io.generators_import.oedb()*.

analyze(*mode=None*, *timesteps=None*, ***kwargs*)
Conducts a static, non-linear power flow analysis

Conducts a static, non-linear power flow analysis using PyPSA and writes results (active, reactive and apparent power as well as current on lines and voltages at buses) to *Results* (e.g. *v_res* for voltages). See *to_pypsa()* for more information.

Parameters

- **mode** (*str*) – Allows to toggle between power flow analysis (PFA) on the whole network topology (default: None), only MV ('mv' or 'mvlv') or only LV ('lv'). Defaults to None which equals power flow analysis for MV + LV.
- **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 *timeindex* are used.

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

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

perform_mp_opf(*timesteps*, *storage_series=[]*, ***kwargs*)
Run optimal power flow with julia.

Parameters

- **timesteps** (*list*) – List of timesteps to perform OPF for.
- **kwargs** – See *run_mp_opf()* for further information.

Returns Status of optimization.

Return type str

aggregate_components(*mode='by_component_type', aggregate_generators_by_cols=['bus'], aggregate_loads_by_cols=['bus'], aggregate_charging_points_by_cols=['bus']*)

Aggregates generators, loads and charging points at the same bus.

There are several options how to aggregate. By default all components of the same type are aggregated separately. You can specify further columns to consider in the aggregation, such as the generator type or the load sector.

Be aware that by aggregating components you lose some information e.g. on load sector or charging point use case.

Parameters

- **mode** (*str*) – Valid options are 'by_component_type' and 'by_load_and_generation'. In case of aggregation 'by_component_type' generators, loads and charging points are aggregated separately, by the respectively specified columns, given in *aggregate_generators_by_cols*, *aggregate_loads_by_cols*, and *aggregate_charging_points_by_cols*. In case of aggregation 'by_load_and_generation', all loads and charging points at the same bus are aggregated. Input in *aggregate_loads_by_cols* and *aggregate_charging_points_by_cols* is ignored. Generators are aggregated by the columns specified in *aggregate_generators_by_cols*.
- **aggregate_generators_by_cols** (*list(str)*) – List of columns to aggregate generators at the same bus by. Valid columns are all columns in *generators_df*.
- **aggregate_loads_by_cols** (*list(str)*) – List of columns to aggregate loads at the same bus by. Valid columns are all columns in *loads_df*.
- **aggregate_charging_points_by_cols** (*list(str)*) – List of columns to aggregate charging points at the same bus by. Valid columns are all columns in *charging_points_df*.

plot_mv_grid_topology(*technologies=False*, ***kwargs*)

Plots plain MV network topology and optionally nodes by technology type (e.g. station or generator).

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

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

plot_mv_voltages(***kwargs*)

Plots voltages in MV network on network 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 grid expansion costs per MV line.

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

plot_mv_storage_integration(***kwargs*)
Plots storage position in MV topology of integrated storage units.

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

plot_mv_grid(***kwargs*)
General plotting function giving all options of function *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'

save(*directory*, *save_results=True*, *save_topology=True*, *save_timeseries=True*, ***kwargs*)
Saves EDisGo object to csv.

It can be chosen if results, topology and timeseries should be saved. For each one, a separate directory is created.

Parameters

- **directory** (*str*) – Main directory to save EDisGo object to.
- **save_results** (*bool, optional*) – Indicates whether to save *Results* object. Per default it is saved. See `to_csv` for more information.
- **save_topology** (*bool, optional*) – Indicates whether to save *Topology*. Per default it is saved. See `to_csv` for more information.
- **save_timeseries** (*bool, optional*) – Indicates whether to save `Timeseries`. Per default it is saved. See `to_csv` for more information.

Other Parameters

- **reduce_memory** (*bool, optional*) – If True, size of dataframes containing time series in *Results* and *TimeSeries* is reduced. See reduce_memory and reduce_memory for more information. Type to convert to can be specified by providing *to_type* as keyword argument. Further parameters of reduce_memory functions cannot be passed here. Call these functions directly to make use of further options. Default: False.
- **to_type** (*str, optional*) – Data type to convert time series data to. This is a tradeoff between precision and memory. Default: "float32".

add_component(*comp_type*, *add_ts=True*, *ts_active_power=None*, *ts_reactive_power=None*, ***kwargs*)

Adds single component to network topology.

Components can be lines or buses as well as generators, loads, charging points or storage units.

Parameters

- **comp_type** (*str*) – Type of added component. Can be 'Bus', 'Line', 'Load', 'Generator', 'StorageUnit', 'Transformer' or 'ChargingPoint'.
- **add_ts** (*bool*) – Indicator if time series for component are added as well.
- **ts_active_power** (pandas.Series) – Active power time series of added component. Index of the series must contain all time steps in *timeindex*. Values are active power per time step in MW.
- **ts_reactive_power** (pandas.Series) – Reactive power time series of added component. Index of the series must contain all time steps in *timeindex*. Values are reactive power per time step in MVA.
- ****kwargs** (*dict*) – Attributes of added component. See respective functions for required entries. For 'Load', 'Generator' and 'StorageUnit' the boolean add_ts determines whether a time series is created for the new component or not.
- **Todo** (*change into add_components to allow adding of several components*) – at a time, change topology.add_load etc. to add_loads, where lists of parameters can be inserted

integrate_component(*comp_type*, *geolocation*, *voltage_level=None*, *add_ts=True*, *ts_active_power=None*, *ts_reactive_power=None*, ***kwargs*)

Adds single component to topology based on geolocation.

Currently components can be generators or charging points.

Parameters

- **comp_type** (*str*) – Type of added component. Can be 'Generator' or 'ChargingPoint'.
- **geolocation** (shapely.Point or tuple) – Geolocation of the new component. In case of tuple, the geolocation must be given in the form (longitude, latitude).
- **voltage_level** (*int, optional*) – Specifies the voltage level the new component is integrated in. Possible options are 4 (MV busbar), 5 (MV grid), 6 (LV busbar) or 7 (LV grid). If no voltage level is provided the voltage level is determined based on the nominal power *p_nom* (given as kwarg) as follows:
  - voltage level 4 (MV busbar): nominal power between 4.5 MW and 17.5 MW
  - voltage level 5 (MV grid) : nominal power between 0.3 MW and 4.5 MW
  - voltage level 6 (LV busbar): nominal power between 0.1 MW and 0.3 MW
  - voltage level 7 (LV grid): nominal power below 0.1 MW

- **add_ts** (*bool, optional*) – Indicator if time series for component are added as well. Default: True.
- **ts_active_power** (pandas.Series, optional) – Active power time series of added component. Index of the series must contain all time steps in `timeindex`. Values are active power per time step in MW. Currently, if you want to add time series (if *add_ts* is True), you must provide a time series. It is not automatically retrieved.
- **ts_reactive_power** (pandas.Series, optional) – Reactive power time series of added component. Index of the series must contain all time steps in `timeindex`. Values are reactive power per time step in MVA. Currently, if you want to add time series (if *add_ts* is True), you must provide a time series. It is not automatically retrieved.

Other Parameters **kwargs** – Attributes of added component. See `add_generator` respectively `add_charging_point` methods for more information on required and optional parameters of generators and charging points.

remove_component(*comp_type*, *comp_name*, *drop_ts=True*)

Removes single component from respective DataFrame. If drop_ts is set to True, timeseries of elements are deleted as well.

Parameters

- **comp_type** (*str*) – Type of removed component. Can be 'Bus', 'Line', 'Load', 'Generator', 'StorageUnit', 'Transformer'.
- **comp_name** (*str*) – Name of component to be removed.
- **drop_ts** (*Boolean*) – Indicator if timeseries for component are removed as well. Defaults to True.
- **Todo** (*change into remove_components, when add_component is changed into*) – add_components, to allow removal of several components at a time

reduce_memory(***kwargs*)

Reduces size of dataframes containing time series to save memory.

Per default, float data is stored as float64. As this precision is barely needed, this function can be used to convert time series data to a data subtype with less memory usage, such as float32.

Other Parameters

- **to_type** (*str, optional*) – Data type to convert time series data to. This is a tradeoff between precision and memory. Default: "float32".
- **results_attr_to_reduce** (*list(str), optional*) – See *attr_to_reduce* parameter in `reduce_memory` for more information.
- **timeseries_attr_to_reduce** (*list(str), optional*) – See *attr_to_reduce* parameter in `reduce_memory` for more information.

1.8.2 edisgo.network package

edisgo.network.components module

class edisgo.network.components.**BasicComponent**(***kwargs*)

Bases: abc.ABC

Generic component

Can be initialized with EDisGo object or Topology object. In case of Topology object component time series attributes currently will raise an error.

id
Unique identifier of component as used in component dataframes in *Topology*.

Returns Unique identifier of component.

Return type str

edisgo_obj
EDisGo container

Returns

Return type *EDisGo*

topology
Network topology container

Returns

Return type *Topology*

voltage_level
Voltage level the component is connected to ('mv' or 'lv').

Returns Voltage level. Returns 'lv' if component connected to the low voltage and 'mv' if component is connected to the medium voltage.

Return type str

grid
Grid component is in.

Returns Grid component is in.

Return type Grid

class edisgo.network.components.**Component**(***kwargs*)
Bases: *edisgo.network.components.BasicComponent*

Generic component for all components that can be considered nodes, e.g. generators and loads.

bus
Bus component is connected to.

Parameters **bus** (str) – ID of bus to connect component to.

Returns Bus component is connected to.

Return type str

grid
Grid component is in.

Returns Grid component is in.

Return type Grid

geom
Geo location of component.

Returns

Return type shapely.Point

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

Load object

peak_load
Peak load in MW.

Parameters **peak_load** (float) – Peak load in MW.

Returns Peak load in MW.

Return type float

annual_consumption
Annual consumption of load in MWh.

Parameters **annual_consumption** (float) – Annual consumption in MWh.

Returns Annual consumption of load in MWh.

Return type float

sector
Sector load is associated with.

The sector is e.g. used to assign load time series to a load using the demandlib. The following four sectors are considered: 'agricultural', 'retail', 'residential', 'industrial'.

Parameters **sector** (str) –

Returns

- str – Load sector
- **#ToDo** (*Maybe return 'not specified' in case sector is None?*)

active_power_timeseries
Active power time series of load in MW.

Returns Active power time series of load in MW.

Return type pandas.Series

reactive_power_timeseries
Reactive power time series of load in Mvar.

Returns Reactive power time series of load in Mvar.

Return type pandas.Series

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

Generator object

nominal_power
Nominal power of generator in MW.

Parameters **nominal_power** (float) – Nominal power of generator in MW.

Returns Nominal power of generator in MW.

Return type float

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

Parameters **type** (str) –

Returns

- str – Technology type
- **#ToDo** (*Maybe return 'not specified' in case type is None?*)

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

Parameters **subtype** (str) –

Returns

- str – Technology subtype
- **#ToDo** (*Maybe return 'not specified' in case subtype is None?*)

active_power_timeseries
Active power time series of generator in MW.

Returns Active power time series of generator in MW.

Return type pandas.Series

reactive_power_timeseries
Reactive power time series of generator in Mvar.

Returns Reactive power time series of generator in Mvar.

Return type pandas.Series

weather_cell_id
Weather cell ID of generator.

The weather cell ID is only used to obtain generator feed-in time series for solar and wind generators.

Parameters **weather_cell_id** (*int*) – Weather cell ID of generator.

Returns Weather cell ID of generator.

Return type int

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

Storage object

ToDo: adapt to refactored code!

Describes a single storage instance in the eDisGo network. 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 topology 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

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

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

Returns

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

class edisgo.network.components.**Switch**(***kwargs*)
Bases: *edisgo.network.components.BasicComponent*

Switch object

Switches are for example medium voltage disconnecting points (points where MV rings are split under normal operation conditions). They are represented as branches and can have two states: 'open' or 'closed'. When the switch is open the branch it is represented by connects some bus and the bus specified in *bus_open*. When it is closed bus *bus_open* is substituted by the bus specified in *bus_closed*.

type
Type of switch.

So far edisgo only considers switch disconnectors.

Parameters **type** (str) – Type of switch.

Returns Type of switch.

Return type str

bus_open
Bus ID of bus the switch is 'connected' to when state is 'open'.

As switches are represented as branches they connect two buses. *bus_open* specifies the bus the branch is connected to in the open state.

Returns Bus in 'open' state.

Return type str

bus_closed
Bus ID of bus the switch is 'connected' to when state is 'closed'.

As switches are represented as branches they connect two buses. *bus_closed* specifies the bus the branch is connected to in the closed state.

Returns Bus in 'closed' state.

Return type str

state
State of switch (open or closed).

Returns State of switch: 'open' or 'closed'.

Return type str

branch
Branch the switch is represented by.

Returns Branch the switch is represented by.

Return type str

grid
Grid switch is in.

Returns Grid switch is in.

Return type Grid

open()
Open switch.

close()
Close switch.

edisgo.network.grids module

class edisgo.network.grids.**Grid**(***kwargs*)
Bases: abc.ABC

Defines a basic grid in eDisGo.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **id** (*str or int, optional*) – Identifier

id

edisgo_obj

nominal_voltage
Nominal voltage of network in kV.

Parameters **nominal_voltage** (*float*) –

Returns Nominal voltage of network in kV.

Return type float

graph
Graph representation of the grid.

Returns Graph representation of the grid as networkx Ordered Graph, where lines are represented by edges in the graph, and buses and transformers are represented by nodes.

Return type networkx.Graph

station
DataFrame with form of buses_df with only grid's station's secondary side bus information.

generators_df
Connected generators within the network.

Returns Dataframe with all generators in topology. For more information on the dataframe see *generators_df*.

Return type pandas.DataFrame

generators
Connected generators within the network.

Returns List of generators within the network.

Return type list(*Generator*)

loads_df
Connected loads within the network.

Returns Dataframe with all loads in topology. For more information on the dataframe see *loads_df*.

Return type pandas.DataFrame

loads
Connected loads within the network.

Returns List of loads within the network.

Return type list(*Load*)

storage_units_df
Connected storage units within the network.

Returns Dataframe with all storage units in topology. For more information on the dataframe see *storage_units_df*.

Return type pandas.DataFrame

charging_points_df
Connected charging points within the network.

Returns Dataframe with all charging points in topology. For more information on the dataframe see *charging_points_df*.

Return type pandas.DataFrame

switch_disconnectors_df
Switch disconnectors in network.

Switch disconnectors are points where rings are split under normal operating conditions.

Returns Dataframe with all switch disconnectors in network. For more information on the dataframe see *switches_df*.

Return type pandas.DataFrame

switch_disconnectors
Switch disconnectors within the network.

Returns List of switch disconnectory within the network.

Return type list(*Switch*)

lines_df
Lines within the network.

Returns Dataframe with all buses in topology. For more information on the dataframe see *lines_df*.

Return type pandas.DataFrame

buses_df
Buses within the network.

Returns Dataframe with all buses in topology. For more information on the dataframe see *buses_df*.

Return type pandas.DataFrame

weather_cells
Weather cells in network.

Returns List of weather cell IDs in network.

Return type list(int)

peak_generation_capacity
Cumulative peak generation capacity of generators in the network in MW.

Returns Cumulative peak generation capacity of generators in the network in MW.

Return type float

peak_generation_capacity_per_technology
Cumulative peak generation capacity of generators in the network per technology type in MW.

Returns Cumulative peak generation capacity of generators in the network per technology type in MW.

Return type pandas.DataFrame

peak_load
Cumulative peak load of loads in the network in MW.

Returns Cumulative peak load of loads in the network in MW.

Return type float

peak_load_per_sector
Cumulative peak load of loads in the network per sector in MW.

Returns Cumulative peak load of loads in the network per sector in MW.

Return type pandas.DataFrame

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

Defines a medium voltage network in eDisGo.

lv_grids
Underlying LV grids.

Parameters **lv_grids** (list(*LVGrid*)) –

Returns Generator object of underlying LV grids of type *LVGrid*.

Return type list generator

buses_df
Buses within the network.

Returns Dataframe with all buses in topology. For more information on the dataframe see *buses_df*.

Return type pandas.DataFrame

transformers_df
Transformers to overlaying network.

Returns Dataframe with all transformers to overlaying network. For more information on the dataframe see *transformers_df*.

Return type pandas.DataFrame

draw()
Draw MV network.

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

Defines a low voltage network in eDisGo.

buses_df
Buses within the network.

Returns Dataframe with all buses in topology. For more information on the dataframe see *buses_df*.

Return type pandas.DataFrame

transformers_df
Transformers to overlaying network.

Returns Dataframe with all transformers to overlaying network. For more information on the dataframe see *transformers_df*.

Return type pandas.DataFrame

draw(*node_color='black'*, *edge_color='black'*, *colorbar=False*, *labels=False*, *filename=None*)
Draw LV network.

Currently, edge width is proportional to nominal apparent power of the line and node size is proportional to peak load of connected loads.

Parameters

- **node_color** (str or pandas.Series) – Color of the nodes (buses) of the grid. If provided as string all nodes will have that color. If provided as series, the index of the series must contain all buses in the LV grid and the corresponding values must be float values, that will be translated to the node color using a colormap, currently set to "Blues". Default: "black".
- **edge_color** (str or pandas.Series) – Color of the edges (lines) of the grid. If provided as string all edges will have that color. If provided as series, the index of the series must contain all lines in the LV grid and the corresponding values must be float values, that will be translated to the edge color using a colormap, currently set to "inferno_r". Default: "black".
- **colorbar** (*bool*) – If True, a colorbar is added to the plot for node and edge colors, in case these are sequences. Default: False.
- **labels** (*bool*) – If True, displays bus names. As bus names are quite long, this is currently not very pretty. Default: False.
- **filename** (*str or None*) – If a filename is provided, the plot is saved under that name but not displayed. If no filename is provided, the plot is only displayed. Default: None.

edisgo.network.results module

class edisgo.network.results.**Results**(*edisgo_object*)
Bases: object

Power flow analysis results management

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

edisgo_object

Type *EDisGo*

measures
List with measures conducted to increase network's hosting capacity.

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

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

Return type list

pfa_p
Active power over components in MW from last power flow analysis.

The given active power for each line / transformer is the active power at the line ending / transformer side with the higher apparent power determined from active powers p_0 and p_1 and reactive powers q_0 and q_0 at the line endings / transformer sides:

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

Parameters **df** (pandas.DataFrame) – Results for active power over lines and transformers in MW from last power flow analysis. Index of the dataframe is a pandas.DatetimeIndex indicating the time period the power flow analysis was conducted for; columns of the dataframe are the representatives of the lines and stations included in the power flow analysis.

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

Returns Results for active power over lines and transformers in MW from last power flow analysis. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

pfa_q
Active power over components in Mvar from last power flow analysis.

The given reactive power over each line / transformer is the reactive power at the line ending / transformer side with the higher apparent power determined from active powers p_0 and p_1 and reactive powers q_0 and q_1 at the line endings / transformer sides:

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

Parameters **df** (pandas.DataFrame) – Results for reactive power over lines and transformers in Mvar from last power flow analysis. Index of the dataframe is a pandas.DatetimeIndex indicating the time period the power flow analysis was conducted for; columns of the dataframe are the representatives of the lines and stations included in the power flow analysis.

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

Returns Results for reactive power over lines and transformers in Mvar from last power flow analysis. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

v_res
Voltages at buses in p.u. from last power flow analysis.

Parameters **df** (pandas.DataFrame) – Dataframe with voltages at buses in p.u. from last power flow analysis. Index of the dataframe is a pandas.DatetimeIndex indicating the time steps the power flow analysis was conducted for; columns of the dataframe are the bus names of all buses in the analyzed grids.

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

Returns Dataframe with voltages at buses in p.u. from last power flow analysis. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

i_res
Current over components in kA from last power flow analysis.

Parameters **df** (pandas.DataFrame) – Results for currents over lines and transformers in kA from last power flow analysis. Index of the dataframe is a pandas.DatetimeIndex indicating

the time steps the power flow analysis was conducted for; columns of the dataframe are the representatives of the lines and stations included in the power flow analysis.

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

Returns Results for current over lines and transformers in kA from last power flow analysis. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

s_res

Apparent power over components in MVA from last power flow analysis.

The given apparent power over each line / transformer is the apparent power at the line ending / transformer side with the higher apparent power determined from active powers p_0 and p_1 and reactive powers q_0 and q_1 at the line endings / transformer sides:

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

Returns Apparent power in MVA over lines and transformers. Index of the dataframe is a pandas.DatetimeIndex indicating the time steps the power flow analysis was conducted for; columns of the dataframe are the representatives of the lines and stations included in the power flow analysis.

Return type pandas.DataFrame

equipment_changes

Tracks changes to the grid topology.

When the grid is reinforced using `reinforce` or new generators added using `import_generators`, new lines and/or transformers are added, lines split, etc. This is tracked in this attribute.

Parameters **df** (pandas.DataFrame) – Dataframe holding information on added, changed and removed lines and transformers. Index of the dataframe is in case of lines the name of the line, and in case of transformers the name of the grid the station is in (in case of MV/LV transformers the name of the LV grid and in case of HV/MV transformers the name of the MV grid). Columns are the following:

equipment [str] Type of new line or transformer as in *equipment_data*.

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

iteration_step [int] Grid reinforcement iteration step the change was conducted in. For changes conducted during grid integration of new generators the iteration step is set to 0.

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

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

Returns Dataframe holding information on added, changed and removed lines and transformers. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

grid_expansion_costs

Costs per expanded component in kEUR.

Parameters **df** (pandas.DataFrame) – Costs per expanded line and transformer in kEUR. Index of the dataframe is the name of the expanded component as string. Columns are the following:

type [str] Type of new line or transformer as in *equipment_data*.

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.

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').

Provide this if you want to set grid expansion costs. For retrieval of costs do not pass an argument.

Returns Costs per expanded line and transformer in kEUR. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

Notes

Network expansion measures are tracked in `equipment_changes`. Resulting costs are calculated using `grid_expansion_costs()`. Total network expansion costs can be obtained through grid_expansion_costs.total_costs.sum().

grid_losses

Active and reactive network losses in MW and Mvar, respectively.

Parameters **df** (pandas.DataFrame) – Results for active and reactive network losses in columns 'p' and 'q' and in MW and Mvar, respectively. Index is a pandas.DatetimeIndex.

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

Returns Results for active and reactive network losses MW and Mvar, respectively. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

Notes

Grid losses are calculated as follows:

$$P_{loss} = |\sum infeed - \sum load + P_{slack}|$$

$$Q_{loss} = |\sum infeed - \sum load + Q_{slack}|$$

As the slack is placed at the station's secondary side (if MV is included, it's positioned at the HV/MV station's secondary side and if a single LV grid is analysed it's positioned at the LV station's secondary side) losses do not include losses over the respective station's transformers.

pfa_slack

Active and reactive power from slack in MW and Mvar, respectively.

In case the MV level is included in the power flow analysis, the slack is placed at the secondary side of the HV/MV station and gives the energy transferred to and taken from the HV network. In case a single LV network is analysed, the slack is positioned at the respective station's secondary, in which case this gives the energy transferred to and taken from the overlying MV network.

Parameters **df** (pandas.DataFrame) – Results for active and reactive power from the slack in MW and Mvar, respectively. Dataframe has the columns 'p', holding the active power results, and 'q', holding the reactive power results. Index is a pandas.DatetimeIndex.

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

Returns Results for active and reactive power from the slack in MW and Mvar, respectively. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

pfa_v_mag_pu_seed

Voltages in p.u. from previous power flow analyses to be used as seed.

See *set_seed()* for more information.

Parameters **df** (pandas.DataFrame) – Voltages at buses in p.u. from previous power flow analyses including the MV level. Index of the dataframe is a pandas.DatetimeIndex indicating the time steps previous power flow analyses were conducted for; columns of the dataframe are the representatives of the buses included in the power flow analyses.

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

Returns Voltages at buses in p.u. from previous power flow analyses to be opionally used as seed in following power flow analyses. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

pfa_v_ang_seed

Voltages in p.u. from previous power flow analyses to be used as seed.

See *set_seed()* for more information.

Parameters **df** (pandas.DataFrame) – Voltage angles at buses in radians from previous power flow analyses including the MV level. Index of the dataframe is a pandas.DatetimeIndex indicating the time steps previous power flow analyses were conducted for; columns of the dataframe are the representatives of the buses included in the power flow analyses.

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

Returns Voltage angles at buses in radians from previous power flow analyses to be opionally used as seed in following power flow analyses. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

unresolved_issues

Lines and buses with remaining grid issues after network reinforcement.

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

Parameters **df** (pandas.DataFrame) – Dataframe containing remaining grid issues. Names of remaining critical lines, stations and buses are in the index of the dataframe. Columns depend on the equipment type. See *mv_line_load()* for format of remaining overloading issues of lines, *hv_mv_station_load()* for format of remaining overloading issues of transformers, and *mv_voltage_deviation()* for format of remaining voltage issues.

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

Returns Dataframe with remaining grid issues. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

reduce_memory(*attr_to_reduce=None*, *to_type='float32'*)
Reduces size of dataframes containing time series to save memory.

See `reduce_memory` for more information.

Parameters

- **attr_to_reduce** (*list(str), optional*) – List of attributes to reduce size for. Attributes need to be dataframes containing only time series. Possible options are: 'pfa_p', 'pfa_q', 'v_res', 'i_res', and 'grid_losses'. Per default, all these attributes are reduced.
- **to_type** (*str, optional*) – Data type to convert time series data to. This is a trade-off between precision and memory. Default: "float32".

Notes

Reducing the data type of the seeds for the power flow analysis, *pfa_v_mag_pu_seed* and *pfa_v_ang_seed*, can lead to non-convergence of the power flow analysis, wherefore memory reduction is not provided for those attributes.

to_csv(*directory*, *parameters=None*, *reduce_memory=False*, *save_seed=False*, ***kwargs*)
Saves results to csv.

Saves power flow results and grid expansion results to separate directories. Which results are saved depends on what is specified in *parameters*. Per default, all attributes are saved.

Power flow results are saved to directory 'powerflow_results' and comprise the following, if not otherwise specified:

- 'v_res' : Attribute *v_res* is saved to *voltages_pu.csv*.
- 'i_res' : Attribute *i_res* is saved to *currents.csv*.
- 'pfa_p' : Attribute *pfa_p* is saved to *active_powers.csv*.
- 'pfa_q' : Attribute *pfa_q* is saved to *reactive_powers.csv*.
- 's_res' : Attribute *s_res* is saved to *apparent_powers.csv*.
- 'grid_losses' : Attribute *grid_losses* is saved to *grid_losses.csv*.
- 'pfa_slack' : Attribute *pfa_slack* is saved to *pfa_slack.csv*.
- 'pfa_v_mag_pu_seed' : Attribute *pfa_v_mag_pu_seed* is saved to *pfa_v_mag_pu_seed.csv*, if *save_seed* is set to True.
- 'pfa_v_ang_seed' : Attribute *pfa_v_ang_seed* is saved to *pfa_v_ang_seed.csv*, if *save_seed* is set to True.

Grid expansion results are saved to directory 'grid_expansion_results' and comprise the following, if not otherwise specified:

- grid_expansion_costs : Attribute *grid_expansion_costs* is saved to *grid_expansion_costs.csv*.
- equipment_changes : Attribute *equipment_changes* is saved to *equipment_changes.csv*.
- unresolved_issues : Attribute *unresolved_issues* is saved to *unresolved_issues.csv*.

Parameters

- **directory** (*str*) – Main directory to save the results in.

- **parameters** (*None or dict, optional*) – Specifies which results to save. By default this is set to None, in which case all results are saved. To only save certain results provide a dictionary. Possible keys are 'powerflow_results' and 'grid_expansion_results'. Corresponding values must be lists with attributes to save or None to save all attributes. For example, with the first input only the power flow results *i_res* and *v_res* are saved, and with the second input all power flow results are saved.

```
{'powerflow_results': ['i_res', 'v_res']}
```

```
{'powerflow_results': None}
```

  See function docstring for possible power flow and grid expansion results to save and under which file name they are saved.

- **reduce_memory** (*bool, optional*) – If True, size of dataframes containing time series to save memory is reduced using `reduce_memory`. Optional parameters of `reduce_memory` can be passed as kwargs to this function. Default: False.
- **save_seed** (*bool, optional*) – If True, *pfa_v_mag_pu_seed* and *pfa_v_ang_seed* are as well saved as csv. As these are only relevant if calculations are not final, the default is False, in which case they are not saved.

Other Parameters **kwargs** – Kwargs may contain optional arguments of `reduce_memory`.

from_csv(*directory*, *parameters=None*)

Restores results from csv files.

See *to_csv()* for more information on which results can be saved and under which filename and directory they are stored.

Parameters

- **directory** (*str*) – Main directory results are saved in.
- **parameters** (*None or dict, optional*) – Specifies which results to restore. By default this is set to None, in which case all available results are restored. To only restore certain results provide a dictionary. Possible keys are 'powerflow_results' and 'grid_expansion_results'. Corresponding values must be lists with attributes to restore or None to restore all available attributes. See function docstring *parameters* parameter in *to_csv()* for more information.

edisgo.network.timeseries module

class edisgo.network.timeseries.**TimeSeries**(***kwargs*)

Bases: object

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

Can also contain time series for loads (sector-specific), generators (technology-specific), and curtailment (technology-specific).

Parameters

- **timeindex** (pandas.DatetimeIndex, optional) – Can be used to define a time range for which to obtain the provided time series and run power flow analysis. Default: None.
- **generators_active_power** (pandas.DataFrame, optional) – Active power timeseries of all generators in topology. Index of DataFrame has to contain timeindex and column names are names of generators.

- **generators_reactive_power** (pandas.DataFrame, optional) – Reactive power time-series of all generators in topology. Format is the same as for generators_active power.
- **loads_active_power** (pandas.DataFrame, optional) – Active power timeseries of all loads in topology. Index of DataFrame has to contain timeindex and column names are names of loads.
- **loads_reactive_power** (pandas.DataFrame, optional) – Reactive power timeseries of all loads in topology. Format is the same as for loads_active power.
- **storage_units_active_power** (pandas.DataFrame, optional) – Active power time-series of all storage units in topology. Index of DataFrame has to contain timeindex and column names are names of storage units.
- **storage_units_reactive_power** (pandas.DataFrame, optional) – Reactive power timeseries of all storage_units in topology. Format is the same as for storage_units_active power.
- **curtailment** (pandas.DataFrame or list, optional) – 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.

Notes

Can also hold the following attributes when specific mode of *get_component_timeseries()* is called: mode, generation_fluctuating, generation_dispatchable, generation_reactive_power, load, load_reactive_power. See description of meth:*get_component_timeseries* for format of these.

timeindex
Defines analysed time steps.

Can be used to define a time range for which to obtain the provided time series and run power flow analysis.

Parameters **ind** (*timestamp or list(timestamp)*) –

Returns See class definition for details.

Return type pandas.DatetimeIndex

generators_active_power
Active power time series of all generators in MW.

Returns See class definition for details.

Return type pandas.DataFrame

generators_reactive_power
Reactive power timeseries of generators in MVA.

Returns See class definition for details.

Return type pandas.DataFrame

loads_active_power
Active power timeseries of loads in MW.

Returns See class definition for details.

Return type dict or pandas.DataFrame

loads_reactive_power
Reactive power timeseries in MVA.

Returns See class definition for details.

Return type pandas.DataFrame

storage_units_active_power
Active power timeseries of storage units in MW.

Returns See class definition for details.

Return type dict or pandas.DataFrame

storage_units_reactive_power
Reactive power timeseries of storage units in MVA.

Returns See class definition for details.

Return type pandas.DataFrame

charging_points_active_power
Active power timeseries of charging points in MW.

Returns See class definition for details.

Return type dict or pandas.DataFrame

charging_points_reactive_power
Reactive power timeseries of charging points in MVA.

Returns See class definition for details.

Return type pandas.DataFrame

residual_load
Returns residual load.

Residual load for each time step is calculated from total load (including charging points) minus total generation minus storage active power (discharge is positive). A positive residual load represents a load case while a negative residual load here represents a feed-in case. Grid losses are not considered.

Returns Series with residual load in MW.

Return type pandas.Series

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 network. Grid losses are not considered.

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

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

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*.

Return type pandas.Series

reduce_memory(*attr_to_reduce=None*, *to_type='float32'*)
Reduces size of dataframes to save memory.

See `EDisGo.reduce_memory` for more information.

Parameters

- **attr_to_reduce** (*list(str), optional*) – List of attributes to reduce size for. Attributes need to be dataframes containing only time series. Per default, all active and reactive power time series of generators, loads, storage units and charging points are reduced.
- **to_type** (*str, optional*) – Data type to convert time series data to. This is a trade-off between precision and memory. Default: "float32".

to_csv(*directory*, *reduce_memory=False*, ***kwargs*)
Saves component time series to csv.

Saves the following time series to csv files with the same file name (if the time series dataframe is not empty):

- loads_active_power and loads_reactive_power
- generators_active_power and generators_reactive_power
- charging_points_active_power and charging_points_reactive_power
- storage_units_active_power and storage_units_reactive_power

Parameters

- **directory** (*str*) – Directory to save time series in.
- **reduce_memory** (*bool, optional*) – If True, size of dataframes is reduced using `reduce_memory`. Optional parameters of `reduce_memory` can be passed as kwargs to this function. Default: False.

Other Parameters **kwargs** – Kwargs may contain optional arguments of `reduce_memory`.

from_csv(*directory*)
Restores time series from csv files.

See *to_csv()* for more information on which time series are saved.

Parameters **directory** (*str*) – Directory time series are saved in.

`edisgo.network.timeseries.`**get_component_timeseries**(*edisgo_obj*, ***kwargs*)
Sets up TimeSeries Object.

Parameters

- **edisgo_obj** (*EDisGo*) – 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. Mode can also be set to manual in order to give standard timeseries, that are not obtained from oedb or demandlib.
- **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.

  - 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.

edisgo.network.timeseries.**add_loads_timeseries**(*edisgo_obj*, *load_names*, ***kwargs*)

Define load time series for active and reactive power. For more information on required and optional parameters see description of *get_component_timeseries()*. The mode initially set within get_component_timeseries is used here to set new timeseries. If a different mode is required, change edisgo_obj.timeseries.mode to the desired mode and provide respective parameters.

Parameters

- **edisgo_obj** (EDisGo) – The eDisGo model overall container
- **load_names** (*str or list of str*) – Names of loads to add timeseries for. Default None, timeseries for all loads of edisgo_obj are set then.

edisgo.network.timeseries.**add_generators_timeseries**(*edisgo_obj*, *generator_names*, ***kwargs*)

Define generator time series for active and reactive power. For more information on required and optional parameters see description of *get_component_timeseries()*.The mode initially set within get_component_timeseries is used here to set new timeseries. If a different mode is required, change edisgo_obj.timeseries.mode to the desired mode and provide respective parameters.

Parameters

- **edisgo_obj** (EDisGo) – The eDisGo model overall container
- **generator_names** (*str or list of str*) – Names of generators to add timeseries for.

Other Parameters

- **generators_active_power** (pandas.DataFrame) – Active power time series in MW.
- **generators_reactive_power** (pandas.DataFrame) – Reactive power time series in MW.

edisgo.network.timeseries.**add_charging_points_timeseries**(*edisgo_obj*, *charging_point_names*, ***kwargs*)

Define generator time series for active and reactive power.

Parameters

- **edisgo_obj** (EDisGo) – The eDisGo model overall container
- **charging_point_names** (*str or list of str*) – Names of charging points to add timeseries for.

Other Parameters

- **ts_active_power** (pandas.DataFrame) – Active power time series in MW.
- **ts_reactive_power** (pandas.DataFrame) – Reactive power time series in MW.

edisgo.network.timeseries.**add_storage_units_timeseries**(*edisgo_obj*, *storage_unit_names*, ***kwargs*)

Define storage unit time series for active and reactive power. For more information on required and optional parameters see description of *get_component_timeseries()*. The mode initially set within get_component_timeseries is used here to set new timeseries. If a different mode is required, change edisgo_obj.timeseries.mode to the desired mode and provide respective parameters.

Parameters

- **edisgo_obj** (EDisGo) – The eDisGo model overall container

- **storage_unit_names** (*str or list of str*) – Names of storage units to add timeseries for. Default None, timeseries for all storage units of edisgo_obj are set then.

edisgo.network.timeseries.**check_timeseries_for_index_and_cols**(*edisgo_obj*, *timeseries*, *component_names*)

Checks index and column names of inserted timeseries to make sure, they have the right format.

Parameters

- **timeseries** (pandas.DataFrame) – inserted timeseries
- **component_names** (*list of str*) – names of components of which timeseries are to be added

edisgo.network.timeseries.**import_load_timeseries**(*config_data*, *data_source*, *year=2018*)

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

edisgo.network.timeseries.**fixed_cosphi**(*active_power*, *q_sign*, *power_factor*)

Calculates reactive power for a fixed cosphi operation.

Parameters

- **active_power** (pandas.DataFrame) – Dataframe with active power time series. Columns of the dataframe are names of the components and index of the dataframe are the time steps reactive power is calculated for.
- **q_sign** (pandas.Series or int) – *q_sign* defines whether the reactive power is positive or negative and must either be -1 or +1. In case *q_sign* is given as a series, the index must contain the same component names as given in columns of parameter *active_power*.
- **power_factor** (pandas.Series or float) – Ratio of real to apparent power. In case *power_factor* is given as a series, the index must contain the same component names as given in columns of parameter *active_power*.

Returns Dataframe with the same format as the *active_power* dataframe, containing the reactive power.

Return type pandas.DataFrame

edisgo.network.topology module

class edisgo.network.topology.**Topology**(***kwargs*)

Bases: object

Container for all grid topology data of a single MV grid.

Data may as well include grid topology data of underlying LV grids.

Other Parameters **config** (None or *Config*) – Provide your configurations if you want to load self-provided equipment data. Path to csv files containing the technical data is set in *config_system.cfg* in sections *system_dirs* and *equipment*. The default is None in which case the equipment data provided by eDisGo is used.

_grids
Dictionary containing all grids (keys are grid representatives and values the grid objects)

Type dict

loads_df
Dataframe with all loads in MV network and underlying LV grids.

Parameters **df** (pandas.DataFrame) – Dataframe with all loads in MV network and underlying LV grids. Index of the dataframe are load names as string. Columns of the dataframe are:

bus [str] Identifier of bus load is connected to.

peak_load [float] Peak load in MW.

annual_consumption [float] Annual consumption in MWh.

sector [str] Specifies type of load. If demandlib is used to generate sector-specific time series, the sector needs to either be 'agricultural', 'industrial', 'residential' or 'retail'. Otherwise sector can be chosen freely.

Returns Dataframe with all loads in MV network and underlying LV grids. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

generators_df
Dataframe with all generators in MV network and underlying LV grids.

Parameters **df** (pandas.DataFrame) – Dataframe with all generators in MV network and underlying LV grids. Index of the dataframe are generator names as string. Columns of the dataframe are:

bus [str] Identifier of bus generator is connected to.

p_nom [float] Nominal power in MW.

type [str] Type of generator, e.g. 'solar', 'run_of_river', etc. Is used in case generator type specific time series are provided.

control [str] Control type of generator used for power flow analysis. In MV and LV grids usually 'PQ'.

weather_cell_id [int] ID of weather cell, that identifies the weather data cell from the weather data set used in the research project open_eGo to determine feed-in profiles of wind and solar generators. Only required when time series of wind and solar generators are assigned using precalculated time series from the OpenEnergy DataBase.

subtype [str] Further specification of type, e.g. 'solar_roof_mounted'. Currently not required for any functionality.

Returns Dataframe with all generators in MV network and underlying LV grids. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

charging_points_df
Dataframe with all charging points in MV grid and underlying LV grids.

Parameters **df** (pandas.DataFrame) – Dataframe with all charging points in MV grid and underlying LV grids. Index of the dataframe are charging point names as string. Columns of the dataframe are:

bus [str] Identifier of bus charging point is connected to.

p_nom [float] Maximum charging power in MW.

use_case [str] Specifies if charging point is e.g. for charging at home, at work, in public, or public fast charging. Used in charging point integration (`integrate_component`) to determine possible grid connection points, in which case use cases 'home', 'work', 'public', and 'fast' are distinguished.

Returns Dataframe with all charging points in MV network and underlying LV grids. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

storage_units_df
Dataframe with all storage units in MV grid and underlying LV grids.

Parameters **df** (pandas.DataFrame) – Dataframe with all storage units in MV grid and underlying LV grids. Index of the dataframe are storage names as string. Columns of the dataframe are:

bus [str] Identifier of bus storage unit is connected to.

control [str] Control type of storage unit used for power flow analysis, usually 'PQ'.

p_nom [float] Nominal power in MW.

Returns Dataframe with all storage units in MV network and underlying LV grids. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

transformers_df
Dataframe with all MV/LV transformers.

Parameters **df** (pandas.DataFrame) – Dataframe with all MV/LV transformers. Index of the dataframe are transformer names as string. Columns of the dataframe are:

bus0 [str] Identifier of bus at the transformer's primary (MV) side.

bus1 [str] Identifier of bus at the transformer's secondary (LV) side.

x_pu [float] Per unit series reactance.

r_pu [float] Per unit series resistance.

s_nom [float] Nominal apparent power in MW.

type_info [str] Type of transformer.

Returns Dataframe with all MV/LV transformers. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

transformers_hvmv_df
Dataframe with all HV/MV transformers.

Parameters **df** (pandas.DataFrame) – Dataframe with all HV/MV transformers, with the same format as *transformers_df*.

Returns Dataframe with all HV/MV transformers. For more information on format see *transformers_df*.

Return type pandas.DataFrame

lines_df

Dataframe with all lines in MV network and underlying LV grids.

Parameters **df** (pandas.DataFrame) – Dataframe with all lines in MV network and underlying LV grids. Index of the dataframe are line names as string. Columns of the dataframe are:

bus0 [str] Identifier of first bus to which line is attached.

bus1 [str] Identifier of second bus to which line is attached.

length [float] Line length in km.

x [float] Reactance of line (or in case of multiple parallel lines total reactance of lines) in Ohm.

r [float] Resistance of line (or in case of multiple parallel lines total resistance of lines) in Ohm.

s_nom [float] Apparent power which can pass through the line (or in case of multiple parallel lines total apparent power which can pass through the lines) in MVA.

num_parallel [int] Number of parallel lines.

type_info [str] Type of line as e.g. given in *equipment_data*.

kind [str] Specifies whether line is a cable ('cable') or overhead line ('line').

Returns Dataframe with all lines in MV network and underlying LV grids. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

buses_df

Dataframe with all buses in MV network and underlying LV grids.

Parameters **df** (pandas.DataFrame) – Dataframe with all buses in MV network and underlying LV grids. Index of the dataframe are bus names as strings. Columns of the dataframe are:

v_nom [float] Nominal voltage in kV.

x [float] x-coordinate (longitude) of geolocation.

y [float] y-coordinate (latitude) of geolocation.

mv_grid_id [int] ID of MV grid the bus is in.

lv_grid_id [int] ID of LV grid the bus is in. In case of MV buses this is NaN.

in_building [bool] Signifies whether a bus is inside a building, in which case only components belonging to this house connection can be connected to it.

Returns Dataframe with all buses in MV network and underlying LV grids.

Return type pandas.DataFrame

switches_df

Dataframe with all switches in MV network and underlying LV grids.

Switches are implemented as branches that, when they are closed, are connected to a bus (*bus_closed*) such that there is a closed ring, and when they are open, connected to a virtual bus (*bus_open*), such that there is no closed ring. Once the ring is closed, the virtual is a single bus that is not connected to the rest of the grid.

Parameters **df** (pandas.DataFrame) – Dataframe with all switches in MV network and underlying LV grids. Index of the dataframe are switch names as string. Columns of the dataframe are:

bus_open [str] Identifier of bus the switch branch is connected to when the switch is open.

bus_closed [str] Identifier of bus the switch branch is connected to when the switch is closed.

branch [str] Identifier of branch that represents the switch.

type [str] Type of switch, e.g. switch disconnector.

Returns Dataframe with all switches in MV network and underlying LV grids. For more information on the dataframe see input parameter *df*.

Return type pandas.DataFrame

id

MV network ID.

Returns MV network ID.

Return type int

mv_grid

Medium voltage network.

The medium voltage network object only contains components (lines, generators, etc.) that are in or connected to the MV grid and does not include any components of the underlying LV grids (also not MV/LV transformers).

Parameters **mv_grid** (*MVGrid*) – Medium voltage network.

Returns Medium voltage network.

Return type *MVGrid*

grid_district

Dictionary with MV grid district information.

Parameters **grid_district** (*dict*) – Dictionary with the following MV grid district information:

'population' [int] Number of inhabitants in grid district.

'geom' [shapely.MultiPolygon] Geometry of MV grid district as (Multi)Polygon.

'srid' [int] SRID (spatial reference ID) of grid district geometry.

Returns Dictionary with MV grid district information. For more information on the dictionary see input parameter *grid_district*.

Return type dict

rings

List of rings in the grid topology.

A ring is represented by the names of buses within that ring.

Returns List of rings, where each ring is again represented by a list of buses within that ring.

Return type list(list)

equipment_data
Technical data of electrical equipment such as lines and transformers.

Returns Dictionary with pandas.DataFrame containing equipment data. Keys of the dictionary are 'mv_transformers', 'mv_overhead_lines', 'mv_cables', 'lv_transformers', and 'lv_cables'.

Return type dict

get_connected_lines_from_bus (*bus_name*)
Returns all lines connected to specified bus.

Parameters **bus_name** (*str*) – Name of bus to get connected lines for.

Returns Dataframe with connected lines with the same format as *lines_df*.

Return type pandas.DataFrame

get_line_connecting_buses (*bus_1*, *bus_2*)
Returns information of line connecting bus_1 and bus_2.

Parameters

- **bus_1** (*str*) – Name of first bus.
- **bus_2** (*str*) – Name of second bus.

Returns Dataframe with information of line connecting bus_1 and bus_2 in the same format as *lines_df*.

Return type pandas.DataFrame

get_connected_components_from_bus (*bus_name*)
Returns dictionary of components connected to specified bus.

Parameters **bus_name** (*str*) – Identifier of bus to get connected components for.

Returns Dictionary of connected components with keys 'generators', 'loads', 'charging_points', 'storage_units', 'lines', 'transformers', 'transformers_hvmv', 'switches'. Corresponding values are component dataframes containing only components that are connected to the given bus.

Return type dict of pandas.DataFrame

get_neighbours (*bus_name*)
Returns a set of neighbour buses of specified bus.

Parameters **bus_name** (*str*) – Identifier of bus to get neighbouring buses for.

Returns Set of identifiers of neighbouring buses.

Return type set(str)

add_load (*bus*, *peak_load*, *annual_consumption*, ***kwargs*)
Adds load to topology.

Load name is generated automatically.

Parameters

- **bus** (*str*) – See *loads_df* for more information.
- **peak_load** (*float*) – See *loads_df* for more information.
- **annual_consumption** (*float*) – See *loads_df* for more information.

Other Parameters **kwargs** – Kwargs may contain any further attributes you want to specify. See *loads_df* for more information on additional attributes used for some functionalities in edisgo. Kwargs may also contain a load ID (provided through keyword argument *load_id* as string) used to generate a unique identifier for the newly added load.

Returns Unique identifier of added load.

Return type str

add_generator(*bus*, *p_nom*, *generator_type*, *control='PQ'*, ***kwargs*)
Adds generator to topology.

Generator name is generated automatically.

Parameters

- **bus** (*str*) – See *generators_df* for more information.
- **p_nom** (*float*) – See *generators_df* for more information.
- **generator_type** (*str*) – Type of generator, e.g. 'solar' or 'gas'. See 'type' in *generators_df* for more information.
- **control** (*str*) – See *generators_df* for more information. Defaults to 'PQ'.

Other Parameters **kwargs** – Kwargs may contain any further attributes you want to specify. See *generators_df* for more information on additional attributes used for some functionalities in edisgo. Kwargs may also contain a generator ID (provided through keyword argument *generator_id* as string) used to generate a unique identifier for the newly added generator.

Returns Unique identifier of added generator.

Return type str

add_charging_point(*bus*, *p_nom*, *use_case*, ***kwargs*)
Adds charging point to topology.

Charging point identifier is generated automatically.

Parameters

- **bus** (*str*) – See *charging_points_df* for more information.
- **p_nom** (*float*) – See *charging_points_df* for more information.
- **use_case** (*str*) – See *charging_points_df* for more information.

Other Parameters **kwargs** – Kwargs may contain any further attributes you want to specify.

add_storage_unit(*bus*, *p_nom*, *control='PQ'*, ***kwargs*)
Adds storage unit to topology.

Storage unit name is generated automatically.

Parameters

- **bus** (*str*) – See *storage_units_df* for more information.
- **p_nom** (*float*) – See *storage_units_df* for more information.
- **control** (*str, optional*) – See *storage_units_df* for more information. Defaults to 'PQ'.

Other Parameters **kwargs** – Kwargs may contain any further attributes you want to specify.

add_line(*bus0*, *bus1*, *length*, ***kwargs*)
Adds line to topology.

Line name is generated automatically. If *type_info* is provided, *x*, *r* and *s_nom* are calculated.

Parameters

- **bus0** (*str*) – Identifier of connected bus.
- **bus1** (*str*) – Identifier of connected bus.
- **length** (*float*) – See *lines_df* for more information.

Other Parameters **kwargs** – Kwargs may contain any further attributes in *lines_df*. It is necessary to either provide *type_info* to determine *x*, *r* and *s_nom* of the line, or to provide *x*, *r* and *s_nom* directly.

add_bus(*bus_name*, *v_nom*, ***kwargs*)
Adds bus to topology.

If provided bus name already exists, a unique name is created.

Parameters

- **bus_name** (*str*) – Name of new bus.
- **v_nom** (*float*) – See *buses_df* for more information.

Other Parameters

- **x** (*float*) – See *buses_df* for more information.
- **y** (*float*) – See *buses_df* for more information.
- **lv_grid_id** (*int*) – See *buses_df* for more information.
- **in_building** (*bool*) – See *buses_df* for more information.

Returns Name of bus. If provided bus name already exists, a unique name is created.

Return type str

remove_load(*name*)
Removes load with given name from topology.

Parameters **name** (*str*) – Identifier of load as specified in index of *loads_df*.

remove_generator(*name*)
Removes generator with given name from topology.

Parameters **name** (*str*) – Identifier of generator as specified in index of *generators_df*.

remove_charging_point(*name*)
Removes charging point from topology.

Parameters **name** (*str*) – Identifier of charging point as specified in index of *charging_points_df*.

remove_storage_unit(*name*)
Removes storage with given name from topology.

Parameters **name** (*str*) – Identifier of storage as specified in index of *storage_units_df*.

remove_line(*name*)
Removes line with given name from topology.

Parameters **name** (*str*) – Identifier of line as specified in index of *lines_df*.

remove_bus(*name*)
Removes bus with given name from topology.

Parameters **name** (*str*) – Identifier of bus as specified in index of *buses_df*.

Notes

Only isolated buses can be deleted from topology. Use respective functions first to delete all connected components (e.g. lines, transformers, loads, etc.). Use function `get_connected_components_from_bus()` to get all connected components.

update_number_of_parallel_lines(*lines_num_parallel*)
Changes number of parallel lines and updates line attributes.

When number of parallel lines changes, attributes x, r, and s_nom have to be adapted, which is done in this function.

Parameters **lines_num_parallel** (pandas.Series) – Index contains identifiers of lines to update as in index of *lines_df* and values of series contain corresponding new number of parallel lines.

change_line_type(*lines*, *new_line_type*)
Changes line type of specified lines to given new line type.

Be aware that this function replaces the lines by one line of the given line type. Lines must all be in the same voltage level and the new line type must be a cable with technical parameters given in equipment parameters.

Parameters

- **lines** (*list(str)*) – List of line names of lines to be changed to new line type.
- **new_line_type** (*str*) – Specifies new line type of lines. Line type must be a cable with technical parameters given in "mv_cables" or "lv_cables" of equipment data.

connect_to_mv(*edisgo_object*, *comp_data*, *comp_type='Generator'*)
Add and connect new generator or charging point to MV grid topology.

This function creates a new bus the new component is connected to. The new bus is then connected to the grid depending on the specified voltage level (given in *comp_data* parameter). Components of voltage level 4 are connected to the HV/MV station. Components of voltage level 5 are connected to the nearest MV bus or line. In case the component is connected to a line, the line is split at the point closest to the new component (using perpendicular projection) and a new branch tee is added to connect the new component to.

Parameters

- **edisgo_object** (*EDisGo*) –
- **comp_data** (*dict*) – Dictionary with all information on component. The dictionary must contain all required arguments of method `add_generator` respectively `add_charging_point`, except the *bus* that is assigned in this function, and may contain all other parameters of those methods. Additionally, the dictionary must contain the voltage level to connect in in key 'voltage_level' and the geolocation in key 'geom'. The voltage level must be provided as integer, with possible options being 4 (component is connected directly to the HV/MV station) or 5 (component is connected somewhere in the MV grid). The geolocation must be provided as Shapely Point object.
- **comp_type** (*str*) – Type of added component. Can be 'Generator' or 'ChargingPoint'. Default: 'Generator'.

Returns The identifier of the newly connected component.

Return type str

connect_to_lv(*edisgo_object*, *comp_data*, *comp_type='Generator'*, *allowed_number_of_comp_per_bus=2*)

Add and connect new generator or charging point to LV grid topology.

This function connects the new component depending on the voltage level, and information on the MV/LV substation ID and geometry, all provided in the *comp_data* parameter. It connects

- **Components with specified voltage level 6**
  - to MV/LV substation (a new bus is created for the new component, unless no geometry data is available, in which case the new component is connected directly to the substation)
- **Generators with specified voltage level 7**
  - with a nominal capacity of <=30 kW to LV loads of type residential, if available
  - with a nominal capacity of >30 kW to LV loads of type retail, industrial or agricultural, if available
  - to random bus in the LV grid as fallback if no appropriate load is available
- **Charging Points with specified voltage level 7**
  - with use case 'home' to LV loads of type residential, if available
  - with use case 'work' to LV loads of type retail, industrial or agricultural, if available, otherwise
  - with use case 'public' or 'fast' to some bus in the grid that is not a house connection
  - to random bus in the LV grid that is not a house connection if no appropriate load is available (fallback)

In case no MV/LV substation ID is provided a random LV grid is chosen. In case the provided MV/LV substation ID does not exist (i.e. in case of components in an aggregated load area), the new component is directly connected to the HV/MV station (will be changed once generators in aggregated areas are treated differently in ding0).

The number of generators or charging points connected at one load is restricted by the parameter *allowed_number_of_comp_per_bus*. If every possible load already has more than the allowed number then the new component is directly connected to the MV/LV substation.

Parameters

- **edisgo_object** (*EDisGo*) –
- **comp_data** (*dict*) – Dictionary with all information on component. The dictionary must contain all required arguments of method `add_generator` respectively `add_charging_point`, except the *bus* that is assigned in this function, and may contain all other parameters of those methods. Additionally, the dictionary must contain the voltage level to connect in in key 'voltage_level' and may contain the geolocation in key 'geom' and the LV grid ID to connect the component in in key 'mvlv_subst_id'. The voltage level must be provided as integer, with possible options being 6 (component is connected directly to the MV/LV substation) or 7 (component is connected somewhere in the LV grid). The geolocation must be provided as Shapely Point object and the LV grid ID as integer.
- **comp_type** (*str*) – Type of added component. Can be 'Generator' or 'ChargingPoint'. Default: 'Generator'.
- **allowed_number_of_comp_per_bus** (*int*) – Specifies, how many generators respectively charging points are at most allowed to be placed at the same bus. Default: 2.

Returns The identifier of the newly connected component.

Return type str

Notes

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

to_graph()
Returns graph representation of the grid.

Returns Graph representation of the grid as networkx Ordered Graph, where lines are represented by edges in the graph, and buses and transformers are represented by nodes.

Return type networkx.Graph

to_csv(*directory*)
Exports topology to csv files.

The following attributes are exported:

- 'loads_df' : Attribute *loads_df* is saved to *loads.csv*.
- 'generators_df' : Attribute *generators_df* is saved to *generators.csv*.
- 'charging_points_df' : Attribute *charging_points_df* is saved to *charging_points.csv*.
- 'storage_units_df' : Attribute *storage_units_df* is saved to *storage_units.csv*.
- 'transformers_df' : Attribute *transformers_df* is saved to *transformers.csv*.
- 'transformers_hvmv_df' : Attribute *transformers_df* is saved to *transformers.csv*.
- 'lines_df' : Attribute *lines_df* is saved to *lines.csv*.
- 'buses_df' : Attribute *buses_df* is saved to *buses.csv*.
- 'switches_df' : Attribute *switches_df* is saved to *switches.csv*.
- 'grid_district' : Attribute *grid_district* is saved to *network.csv*.

Attributes are exported in a way that they can be directly imported to pypsa.

Parameters **directory** (*str*) – Path to save topology to.

from_csv(*directory*, *edisgo_obj*)
Restores topology from csv files.

Parameters **directory** (*str*) – Path to topology csv files.

1.8.3 edisgo.flex_opt package

edisgo.flex_opt.check_tech_constraints module

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

Parameters **edisgo_obj** (*EDisGo*) –

Returns Dataframe containing over-loaded MV lines, their maximum relative over-loading (maximum calculated current over allowed current) and the corresponding time step. Index of the dataframe are the names of the over-loaded lines. Columns are 'max_rel_overload' containing the maximum relative over-loading as float, 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp, and 'voltage_level' specifying the voltage level the line is in (either 'mv' or 'lv').

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**(*edisgo_obj*)

Checks for over-loading issues in LV network.

Parameters **edisgo_obj** (*EDisGo*) –

Returns Dataframe containing over-loaded LV lines, their maximum relative over-loading (maximum calculated current over allowed current) and the corresponding time step. Index of the dataframe are the names of the over-loaded lines. Columns are 'max_rel_overload' containing the maximum relative over-loading as float, 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp, and 'voltage_level' specifying the voltage level the line is in (either 'mv' or 'lv').

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.**lines_allowed_load**(*edisgo_obj*, *voltage_level*)

Get allowed maximum current per line per time step

Parameters

- **edisgo_obj** (*EDisGo*) –
- **voltage_level** (*str*) – Grid level, allowed line load is returned for. Possible options are "mv" or "lv".

Returns Dataframe containing the maximum allowed current per line and time step in kA. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Columns are line names of all lines in the specified voltage level.

Return type pandas.DataFrame

edisgo.flex_opt.check_tech_constraints.**lines_relative_load**(*edisgo_obj*, *lines_allowed_load*)

Calculates relative line load based on specified allowed line load.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **lines_allowed_load** (pandas.DataFrame) – Dataframe containing the maximum allowed current per line and time step in kA. Index of the dataframe are time steps of type pandas.Timestamp and columns are line names.

Returns Dataframe containing the relative line load per line and time step. Index and columns of the dataframe are the same as those of parameter *lines_allowed_load*.

Return type pandas.DataFrame

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

Parameters **edisgo_obj** (*EDisGo*) –

Returns Dataframe containing over-loaded HV/MV station, their apparent power at maximal over-loading and the corresponding time step. Index of the dataframe is the representative of the MVGrid. Columns are 's_missing' containing the missing apparent power at maximal over-loading in MVA 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**(*edisgo_obj*)
Checks for over-loading of MV/LV stations.

Parameters **edisgo_obj** (*EDisGo*) –

Returns Dataframe containing over-loaded MV/LV stations, their missing apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the representatives of the grids with over-loaded stations. Columns are 's_missing' containing the missing apparent power at maximal over-loading in MVA 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**(*edisgo_obj*, *voltage_levels='mv_lv'*)
Checks for voltage stability issues in MV network.

Returns buses with voltage issues and their maximum voltage deviation.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **voltage_levels** (str) – Specifies which allowed voltage deviations to use. Possible options are:
  - 'mv_lv' This is the default. The allowed voltage deviations for buses in the MV is the same as for buses in the LV. Further, load and feed-in case are not distinguished.
  - 'mv' Use this to handle allowed voltage limits in the MV and LV topology differently. In that case, load and feed-in case are differentiated.

Returns Dictionary with representative of *MVGrid* as key and a pandas.DataFrame with voltage deviations from allowed lower or upper voltage limits, sorted descending from highest to lowest voltage deviation, as value. Index of the dataframe are all buses with voltage issues. Columns are 'v_diff_max' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the voltage issue 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**(*edisgo_obj*, *mode=None*, *voltage_levels='mv_lv'*)

Checks for voltage stability issues in LV networks.

Returns buses with voltage issues and their maximum voltage deviation.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **mode** (None or str) – If None voltage at all buses in LV networks is checked. If mode is set to 'stations' only voltage at bus bar is checked. Default: None.
- **voltage_levels** (str) – Specifies which allowed voltage deviations to use. Possible options are:
  - 'mv_lv' This is the default. The allowed voltage deviations for buses in the LV is the same as for buses in the MV. Further, load and feed-in case are not distinguished.
  - 'lv' Use this to handle allowed voltage limits in the MV and LV topology differently. In that case, load and feed-in case are differentiated.

Returns Dictionary with representative of *LVGrid* as key and a pandas.DataFrame with voltage deviations from allowed lower or upper voltage limits, sorted descending from highest to lowest voltage deviation, as value. Index of the dataframe are all buses with voltage issues. Columns are 'v_diff_max' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the voltage issue 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.**voltage_diff**(*edisgo_obj*, *buses*, *v_dev_allowed_upper*, *v_dev_allowed_lower*)

Function to detect under- and overvoltage at buses.

The function returns both under- and overvoltage deviations in p.u. from the allowed lower and upper voltage limit, respectively, in separate dataframes. In case of both under- and overvoltage issues at one bus, only the highest voltage deviation is returned.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **buses** (*list (str)*) – List of buses to check voltage deviation for.
- **v_dev_allowed_upper** (pandas.Series) – Series with time steps (of type pandas.Timestamp) power flow analysis was conducted for and the allowed upper limit of voltage deviation for each time step as float in p.u..
- **v_dev_allowed_lower** (pandas.Series) – Series with time steps (of type pandas.Timestamp) power flow analysis was conducted for and the allowed lower limit of voltage deviation for each time step as float in p.u..

Returns

- pandas.DataFrame – Dataframe with deviations from allowed lower voltage level. Columns of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp; in the index are all buses for which undervoltage was detected. In case of a higher over- than undervoltage deviation for a bus, the bus does not appear in this dataframe, but in the dataframe with overvoltage deviations.
- pandas.DataFrame – Dataframe with deviations from allowed upper voltage level. Columns of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp; in the index are all buses for which overvoltage was detected. In case of a higher under- than overvoltage deviation for a bus, the bus does not appear in this dataframe, but in the dataframe with undervoltage deviations.

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

Checks if 10% criteria is exceeded.

Through the 10% criteria it is ensured that voltage is kept between 0.9 and 1.1 p.u.. In case of higher or lower voltages a ValueError is raised.

Parameters **edisgo_obj** (*EDisGo*) –

edisgo.flex_opt.costs module

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

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

edisgo.flex_opt.costs.**edisgo_obj**

Type EDisGo

edisgo.flex_opt.costs.**without_generator_import**

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

Type Boolean

edisgo.flex_opt.costs.**mode**

Specifies topology levels reinforcement was conducted for to only return costs in the considered topology level. Specify

- None to return costs in MV and LV topology levels. None is the default.
- 'mv' to return costs of MV topology level only, including MV/LV stations. Costs to connect LV generators are excluded as well.

Type str

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 name of either line or 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 network expansion was conducted in.

Return type *pandas.DataFrame<DataFrame>*

Notes

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

edisgo.flex_opt.costs.**line_expansion_costs**(*edisgo_obj*, *lines_names*)

Returns costs for earthworks and per added cable as well as voltage level for chosen lines in edisgo_obj.

Parameters

- **edisgo_obj** (*EDisGo*) – eDisGo object of which lines of lines_df are part
- **lines_names** (*list of str*) – List of names of evaluated lines

Returns costs – Dataframe with names of lines as index and entries for 'costs_earthworks', 'costs_cable', 'voltage_level' for each line

Return type pandas.DataFrame

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 network 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_grid=False*, *max_while_iterations=10*, *combined_analysis=False*, *mode=None*)

Evaluates network reinforcement needs and performs measures.

This function is the parent function for all network 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_grid** (Boolean) – If True reinforcement is conducted on a copied grid 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 topology are used. If False different allowed voltage deviations for MV and LV are used. See also config section *grid_expansion_allowed_voltage_deviations*. If *mode* is set to 'mv' *combined_analysis* should be False. Default: False.
- **mode** (str) – Determines network levels reinforcement is conducted for. Specify
  - None to reinforce MV and LV network levels. None is the default.
  - 'mv' to reinforce MV network level only, neglecting MV/LV stations, and LV network topology. LV load and generation is aggregated per LV network and directly connected to the primary side of the respective MV/LV station.
  - 'mvlv' to reinforce MV network level only, including MV/LV stations, and neglecting LV network topology. LV load and generation is aggregated per LV network and directly connected to the secondary side of the respective MV/LV station.

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

Return type Results

Notes

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

edisgo.flex_opt.reinforce_measures module

edisgo.flex_opt.reinforce_measures.**reinforce_mv_lv_station_overloading**(*edisgo_obj*, *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

- **edisgo_obj** (*EDisGo*) –
- **critical_stations** (pandas.DataFrame) – Dataframe containing over-loaded MV/LV stations, their missing apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the representatives of the grids with over-loaded stations. Columns are 's_missing' containing the missing apparent power at maximal over-loading in MVA as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Returns

Dictionary with added and removed transformers in the form:

```
{'added': {'Grid_1': ['transformer_reinforced_1',
                      ...,
                      'transformer_reinforced_x'],
           'Grid_10': ['transformer_reinforced_10']
           },
 'removed': {'Grid_1': ['transformer_1']}
}
```

Return type dict

edisgo.flex_opt.reinforce_measures.**reinforce_hv_mv_station_overloading**(*edisgo_obj*, *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

- **edisgo_obj** (*EDisGo*) –
- **critical_stations** (pandas:*pandas.DataFrame<DataFrame>*) – Dataframe containing over-loaded HV/MV stations, their missing apparent power at maximal over-loading and the corresponding time step. Index of the dataframe are the representatives of the grids with over-loaded stations. Columns are 's_missing' containing the missing apparent power at maximal over-loading in MVA as float and 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp.

Returns

Dictionary with added and removed transformers in the form:

```
{'added': {'Grid_1': ['transformer_reinforced_1',
                      ...,
                      'transformer_reinforced_x'],
           'Grid_10': ['transformer_reinforced_10']
           },
 'removed': {'Grid_1': ['transformer_1']}
}
```

Return type dict

edisgo.flex_opt.reinforce_measures.**reinforce_mv_lv_station_voltage_issues**(*edisgo_obj*, *critical_stations*)

Reinforce MV/LV substations due to voltage issues.

A parallel standard transformer is installed.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **critical_stations** (dict) – Dictionary with representative of *LVGrid* as key and a pandas.DataFrame with station's voltage deviation from allowed lower or upper voltage limit as value. Index of the dataframe is the station with voltage issues. Columns are 'v_diff_max' containing the maximum voltage deviation as float and 'time_index' containing the corresponding time step the voltage issue occured in as pandas.Timestamp.

Returns

Dictionary with added transformers in the form:

```
{'added': {'Grid_1': ['transformer_reinforced_1',
                      ...,
                      'transformer_reinforced_x'],
           'Grid_10': ['transformer_reinforced_10']
           }
}
```

Return type dict

edisgo.flex_opt.reinforce_measures.**reinforce_lines_voltage_issues**(*edisgo_obj*, *grid*, *crit_nodes*)

Reinforce lines in MV and LV topology due to voltage issues.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **grid** (*MVGrid* or *LVGrid*) –
- **crit_nodes** (pandas.DataFrame) – Dataframe with all nodes with voltage issues in the grid and their maximal deviations from allowed lower or upper voltage limits sorted descending from highest to lowest voltage deviation (it is not distinguished between over- or undervoltage). Columns of the dataframe are 'v_diff_max' containing the maximum absolute voltage deviation as float and 'time_index' containing the corresponding time step the

voltage issue occured in as pandas.Timestamp. Index of the dataframe are the names of all buses with voltage issues.

Returns Dictionary with name of lines as keys and the corresponding number of lines added as values.

Return type dict

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 LV stations 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_lines_overloading**(*edisgo_obj*, *crit_lines*)

Reinforce lines in MV and LV topology due to overloading.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **crit_lines** (pandas.DataFrame) – Dataframe containing over-loaded lines, their maximum relative over-loading (maximum calculated current over allowed current) and the corresponding time step. Index of the dataframe are the names of the over-loaded lines. Columns are 'max_rel_overload' containing the maximum relative over-loading as float, 'time_index' containing the corresponding time step the over-loading occured in as pandas.Timestamp, and 'voltage_level' specifying the voltage level the line is in (either 'mv' or 'lv').

Returns Dictionary with name of lines as keys and the corresponding number of lines added as values.

Return type dict

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.

1.8.4 edisgo.io package

edisgo.io.ding0_import module

edisgo.io.ding0_import.**import_ding0_grid**(*path*, *edisgo_obj*)
Import an eDisGo network topology from Ding0 data.

This import method is specifically designed to load network topology data in the format as Ding0 provides it via csv files.

Parameters

- **path** (*str*) – Path to ding0 network csv files.
- **edisgo_obj** (*EDisGo*) – The eDisGo data container object.

edisgo.io.generators_import module

edisgo.io.generators_import.**oedb**(*edisgo_object*, *generator_scenario*, ***kwargs*)
Gets generator park for specified scenario from oedb and integrates them into the grid.

The importer uses SQLAlchemy ORM objects. These are defined in ego.io.

For further information see also `import_generators`.

Parameters

- **edisgo_object** (*EDisGo*) –
- **generator_scenario** (*str*) – Scenario for which to retrieve generator data. Possible options are 'nep2035' and 'ego100'.

Other Parameters

- **remove_decommissioned** (*bool*) – If True, removes generators from network that are not included in the imported dataset (=decommissioned). Default: True.
- **update_existing** (*bool*) – If True, updates capacity of already existing generators to capacity specified in the imported dataset. Default: True.
- **p_target** (*dict or None*) – Per default, no target capacity is specified and generators are expanded as specified in the respective scenario. However, you may want to use one of the scenarios but have slightly more or less generation capacity than given in the respective scenario. In that case you can specify the desired target capacity per technology type using this input parameter. The target capacity dictionary must have technology types (e.g. 'wind' or 'solar') as keys and corresponding target capacities in MW as values. If a target capacity is given that is smaller than the total capacity of all generators of that type in the future scenario, only some of the generators in the future scenario generator park are installed, until the target capacity is reached. If the given target capacity is greater than that of all generators of that type in the future scenario, then each generator capacity is scaled up to reach the target capacity. Be careful to not have much greater target capacities as this will lead to unplausible generation capacities being connected to the different voltage levels. Also be aware that only technologies specified in the dictionary are expanded. Other technologies are kept the same. Default: None.
- **allowed_number_of_comp_per_lv_bus** (*int*) – Specifies, how many generators are at most allowed to be placed at the same LV bus. Default: 2.

edisgo.io.pypsa_io module

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

edisgo.io.pypsa_io.**to_pypsa**(*grid_object*, *timesteps*, ***kwargs*)
Export edisgo object to PyPSA Network

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

Translating eDisGo's network topology to PyPSA representation is structured into translating the topology and adding time series for components of the network. In both cases translation of MV network only (*mode='mv'*, *mode='mvlv'*), LV network 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 network components are exported including the medium voltage side of LV station. Transformers are not exported in this mode. LV network load and generation is considered using `append_lv_components()`. Time series are collected and imported to PyPSA network.
- Medium-voltage including transformers (*mode='mvlv'*). Works similar as the first mode, only attaching LV components to the LV side of the LVStation and therefore also adding the transformers to the PyPSA network.
- Low-voltage only (*mode='lv'*): LV network 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 network topology is translated to PyPSA in order to perform a complete power flow analysis in both levels together. First, both network levels are translated seperately and then merged. Time series are obtained at once for both network levels.

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

- Sub-graphs/ Sub-networks: It is ensured the network 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

- **grid_object** (*EDisGo* or *Grid*) – EDisGo or grid object
- **mode** (*str*) – Determines network levels that are translated to PyPSA network representation. Specify
  - None to export MV and LV network levels. None is the default.
  - 'mv' to export MV network level only. This includes cumulative load and generation from underlying LV network aggregated at respective LV station's primary side.
  - 'mvlv' to export MV network level only. This includes cumulative load and generation from underlying LV network aggregated at respective LV station's secondary side. #ToDo change name of this mode or use kwarg to define where to aggregate lv loads and generation
  - 'lv' to export specified LV network only.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies which time steps to export to pypsa representation and use in power flow analysis.

Other Parameters **use_seed** (*bool*) – Use a seed for the initial guess for the Newton-Raphson algorithm. Only available when MV level is included in the power flow analysis. If True, uses

voltage magnitude results of previous power flow analyses as initial guess in case of PQ buses. PV buses currently do not occur and are therefore currently not supported. Default: False.

Returns The PyPSA network container.

Return type pypsa.Network

edisgo.io.pypsa_io.**set_seed**(*edisgo_obj*, *pypsa_network*)

Set initial guess for the Newton-Raphson algorithm.

In PyPSA an initial guess for the Newton-Raphson algorithm used in the power flow analysis can be provided to speed up calculations. For PQ buses, which besides the slack bus, is the only bus type in edisgo, voltage magnitude and angle need to be guessed. If the power flow was already conducted for the required time steps and buses, the voltage magnitude and angle results from previously conducted power flows stored in *pfa_v_mag_pu_seed* and *pfa_v_ang_seed* are used as the initial guess. Always the latest power flow calculation is used and only results from power flow analyses including the MV level are considered, as analysing single LV grids is currently not in the focus of edisgo and does not require as much speeding up, as analysing single LV grids is usually already quite quick. If for some buses or time steps no power flow results are available, default values are used. For the voltage magnitude the default value is 1 and for the voltage angle 0.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **pypsa_network** (pypsa.Network) – Pypsa network in which seed is set.

edisgo.io.pypsa_io.**process_pfa_results**(*edisgo*, *pypsa*, *timesteps*)

Passing power flow results from PyPSA to *Results*.

Parameters

- **edisgo** (*EDisGo*) –
- **pypsa** (pypsa.Network) – The PyPSA Network container
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Time steps for which latest power flow analysis was conducted and for which to retrieve pypsa results.

Notes

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

$$S_{max} = max(\sqrt{P_0^2 + Q_0^2}, \sqrt{P_1^2 + Q_1^2})\ P = P_0 P_1(S_{max})\ Q = Q_0 Q_1(S_{max})$$

See also:

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

edisgo.io.timeseries_import module

edisgo.io.timeseries_import.**import_feedin_timeseries**(*config_data*, *weather_cell_ids*, *timeindex*)

Import RES feed-in time series data and process

ToDo: Update docstring.

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 DataFrame with time series for active power feed-in, normalized to a capacity of 1 MW.

Return type pandas.DataFrame

1.8.5 edisgo.opf package

edisgo.opf.run_mp_opf module

edisgo.opf.run_mp_opf.**convert**(*o*)
Helper function for json dump, as int64 cannot be dumped.

edisgo.opf.run_mp_opf.**bus_names_to_ints**(*pypsa_network*, *bus_names*)
This remaps a list of eDisGo bus names from Strings to Integers.

Integer indices are needed for the optimization. The result uses one-based indexing, as it gets passed on to Julia directly.

Parameters

- **pypsa_network** –
- **bus_names** (*list (str)*) – List of bus names to be remapped to indices.

Returns List of one-based bus indices.

Return type list(int)

edisgo.opf.run_mp_opf.**run_mp_opf**(*edisgo_network*, *timesteps=None*, *storage_series=[]*, ***kwargs*)

Parameters

- **edisgo_network** –
- **timesteps** (*pandas.DatetimeIndex<DatetimeIndex>* or *pandas.Timestamp<Timestamp>*) –
- ****kwargs** – "scenario" : "nep" # objective function "objective": "nep", # chosen relaxation "relaxation": "none", # upper bound on network expansion "max_exp": 10, # number of time steps considered in optimization "time_horizon": 2, # length of time step in hours "time_elapsed": 1.0, # storage units are considered "storage_units": False, # positioning of storage units, if empty list, all buses are potential positions of storage units and # capacity is optimized "storage_buses": [], # total storage capacity in the network "total_storage_capacity": 0.0, # Requirements for curtailment in every time step is considered "storage_series": [], # Time series for storage operation required by upper grid layer "curtailment_requirement": False, # List of total curtailment for each time step, len(list)== "time_horizon" "curtailment_requirement_series": [], # An overall allowance of curtailment is considered "curtailment_allowance": False, # Maximal allowed curtailment over entire time horizon, # DEFAULT: "3percent"=> 3% of total RES generation in time horizon may be curtailed, else: Float "curtailment_total": "3percent", "results_path": "opf_solutions" # path to where OPF results are stored

edisgo.opf.timeseries_reduction module

edisgo.opf.timeseries_reduction.**get_steps_curtailment**(*edisgo_obj*, *percentage=0.5*)
Get the time steps with the most critical violations for curtailment optimization.

Parameters

- **edisgo_obj** (*EDisGo*) – The eDisGo API object
- **percentage** (*float*) – The percentage of most critical time steps to select

Returns the reduced time index for modeling curtailment

Return type *pandas.DatetimeIndex*

edisgo.opf.timeseries_reduction.**get_steps_storage**(*edisgo_obj*, *window=5*)

Get the most critical time steps from series for storage problems.

Parameters

- **edisgo_obj** (*EDisGo*) – The eDisGo API object
- **window** (*int*) – The additional hours to include before and after each critical time step.

Returns the reduced time index for modeling storage

Return type *pandas.DatetimeIndex*

edisgo.opf.timeseries_reduction.**get_linked_steps**(*cluster_params*, *num_steps=24*, *keep_steps=[]*)

Use provided data to identify representative time steps and create mapping Dict that can be passed to optimization

Parameters

- **cluster_params** (pandas.DataFrame) – Time series containing the parameters to be considered for distance between points.
- **num_steps** (*int*) – The number of representative time steps to be selected.
- **keep_steps** (*Iterable of the same type as cluster_params.index*) – Time steps to retain with full resolution, regardless of clustering result.

Returns Dictionary where each represented time step is a key and its representative time step is a value.

Return type dict

edisgo.opf.results package

edisgo.opf.results.opf_expand_network.**expand_network**(*edisgo*, *tolerance=1e-06*)

Apply network expansion factors that were obtained by optimization to eDisGo MVGrid

Parameters

- **edisgo** (*EDisGo*) –
- **tolerance** (*float*) – The acceptable margin with which an expansion factor can deviate from the nearest Integer before it gets rounded up

edisgo.opf.results.opf_expand_network.**grid_expansion_costs**(*opf_results*, *tolerance=1e-06*)

Calculates grid expansion costs from OPF.

As grid expansion is conducted continuously number of expanded lines is determined by simply rounding up (including some tolerance).

Parameters

- **opf_results** (*OPFResults class*) –

- **tolerance** (*float*) –

Returns Grid expansion costs determined by OPF

Return type float

edisgo.opf.results.opf_expand_network.**integrate_storage_units**(*edisgo*, *min_storage_size=0.3*, *timeseries=True*, *as_load=False*)

Integrates storage units from OPF into edisgo grid topology.

Storage units that are too small to be connected to the MV grid or that are not used (time series contains only zeros) are discarded.

Parameters

- **edisgo** (*EDisGo object*) –
- **min_storage_size** (*float*) – Minimal storage size in MW needed to connect storage unit to MV grid. Smaller storage units are ignored.
- **timeseries** (*bool*) – If True time series is added to component.
- **as_load** (*bool*) – If True, storage is added as load to the edisgo topology. This is temporarily needed as the OPF cannot handle storage units from edisgo yet. This way, storage units with fixed position and time series can be considered in OPF.

Returns First return value contains the names of the added storage units and the second return value the capacity of storage units that were too small to connect to the MV grid or not used.

Return type list(str), float

edisgo.opf.results.opf_expand_network.**get_curtailment_per_node**(*edisgo*, *curtailment_ts=None*, *tolerance=0.001*)

Gets curtailed power per node.

As LV generators are aggregated at the corresponding LV station curtailment is not determined per generator but per node.

This function also checks if curtailment requirements were met by OPF in case the curtailment requirement time series is provided.

Parameters

- **edisgo** (*EDisGo object*) –
- **curtailment_ts** (*pd.Series*) – Series with curtailment requirement per time step. Only needs to be provided if you want to check if requirement was met.
- **tolerance** (*float*) – Tolerance for checking if curtailment requirement and curtailed power are equal.

Returns DataFrame with curtailed power in MW per node. Column names correspond to nodes and index to time steps calculated.

Return type pd.DataFrame

edisgo.opf.results.opf_expand_network.**get_load_curtailment_per_node**(*edisgo*, *tolerance=0.001*)

Gets curtailed load per node.

Parameters

- **edisgo** (*EDisGo object*) –
- **tolerance** (*float*) – Tolerance for checking if curtailment requirement and curtailed power are equal.

Returns DataFrame with curtailed power in MW per node. Column names correspond to nodes and index to time steps calculated.

Return type pd.DataFrame

edisgo.opf.results.opf_expand_network.**integrate_curtailment_as_load**(*edisgo*, *curtailment_per_node*)

Adds load curtailed power per node as load

This is done because curtailment results from OPF are not given per generator but per node (as LV generators are aggregated per LV grid).

Parameters

- **edisgo** –
- **curtailment_per_node** –

Returns

edisgo.opf.results.opf_result_class.**read_from_json**(*edisgo_obj*, *path*, *mode='mv'*)

Read optimization results from json file.

This reads the optimization results directly from a JSON result file without carrying out the optimization process.

Parameters

- **edisgo_obj** (*EDisGo*) – An edisgo object with the same topology that the optimization was run on.
- **path** (*str*) – Path to the optimization result JSON file.
- **mode** (*str*) – Voltage level, currently only supports "mv"

class edisgo.opf.results.opf_result_class.**LineVariables**

Bases: object

class edisgo.opf.results.opf_result_class.**BusVariables**

Bases: object

class edisgo.opf.results.opf_result_class.**GeneratorVariables**

Bases: object

class edisgo.opf.results.opf_result_class.**LoadVariables**

Bases: object

class edisgo.opf.results.opf_result_class.**StorageVariables**

Bases: object

class edisgo.opf.results.opf_result_class.**OPFResults**

Bases: object

set_solution(*solution_name*, *pypsa_net*)

read_solution_file(*solution_name*)

dump_solution_file(*solution_name=[]*)

set_solution_to_results(*pypsa_net*)

set_line_variables(*pypsa_net*)

set_bus_variables(*pypsa_net*)

set_gen_variables(*pypsa_net*)

set_load_variables(*pypsa_net*)

set_strg_variables(*pypsa_net*)

edisgo.opf.util package

edisgo.opf.util.plot_solutions.**plot_line_expansion**(*edisgo_obj*, *timesteps*)

edisgo.opf.util.scenario_settings.**opf_settings**()

1.8.6 edisgo.tools package

edisgo.tools.config module

This file is part of eDisGo, a python package for distribution network 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.

class edisgo.tools.config.**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.tools.config import Config
>>> config = Config()
```

Get reactive power factor for generators in the MV network

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

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.edisgo_run.**setup_logging**(*logfilename=None*, *logfile_loglevel='debug'*, *console_loglevel='info'*, ***logging_kwargs*)

edisgo.tools.edisgo_run.**run_edisgo_basic**(*ding0_filepath*, *generator_scenario=None*, *analysis='worst-case'*, **edisgo_grid*)
Analyze edisgo network extension cost as reference scenario

ToDo: adapt to refactored code!

Parameters

- **ding0_filepath** (*str*) – Path to ding0 data ending typically */path/to/ding0_data/"ding0_grids__" + str(`grid_district`) + ".xxx"`*
- **analysis** (*str*) – Either 'worst-case' or 'timeseries'
- **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'.

Returns

- **edisgo_grid** (EDisGo) – eDisGo network container
- **costs** (pandas.Dataframe) – Cost of network extension
- **grid_issues** (*dict*) – Grids resulting in an error including error message

edisgo.tools.edisgo_run.**run_edisgo_twice**(*run_args*)

Run network analysis twice on same network: once w/ and once w/o new generators

ToDo: adapt to refactored code!

First run without connection of new generators approves sufficient network hosting capacity. Otherwise, network is reinforced. Second run assessment network extension needs in terms of RES integration

Parameters **run_args** (*list*) – Optional parameters for *run_edisgo_basic()*.

Returns

- **all_costs_before_geno_import** (pandas.Dataframe) – Grid extension cost before network connection of new generators
- **all_grid_issues_before_geno_import** (*dict*) – Remaining overloading or over-voltage issues in network
- **all_costs** (pandas.Dataframe) – Grid extension cost due to network connection of new generators
- **all_grid_issues** (*dict*) – Remaining overloading or over-voltage issues in network

edisgo.tools.edisgo_run.**run_edisgo_pool**(*ding0_file_list, run_args_opt=[None, 'worst-case'], workers=2, worker_lifetime=1*)

Use python multiprocessing toolbox for parallelization

Several grids are analyzed in parallel.

Parameters

- **ding0_file_list** (*list*) – Ding0 network data file names
- **run_args_opt** (*list*) – eDisGo options, see *run_edisgo_basic()* and *run_edisgo_twice()*, has to contain generator_scenario and analysis as entries
- **workers** (*int*) – Number of parallel process
- **worker_lifetime** (*int*) – Bunch of grids sequentially analyzed by a worker

Returns

- **all_costs_before_geno_import** (*list*) – Grid extension cost before network connection of new generators
- **all_grid_issues_before_geno_import** (*list*) – Remaining overloading or over-voltage issues in network
- **all_costs** (*list*) – Grid extension cost due to network connection of new generators

- **all_grid_issues** (*list*) – Remaining overloading or over-voltage issues in network

edisgo.tools.edisgo_run.**run_edisgo_pool_flexible**(*ding0_id_list*, *func*, *func_arguments*, *workers=2*, *worker_lifetime=1*)

Use python multiprocessing toolbox for parallelization

Several grids are analyzed in parallel based on your custom function that defines the specific application of eDisGo.

Parameters

- **ding0_id_list** (*list of int*) – List of ding0 network data IDs (also known as HV/MV substation IDs)
- **func** (*any function*) – Your custom function that shall be parallelized
- **func_arguments** (*tuple*) – Arguments to custom function `func`
- **workers** (*int*) – Number of parallel process
- **worker_lifetime** (*int*) – Bunch of grids sequentially analyzed by a worker

Notes

Please note, the following requirements for the custom function which is to be executed in parallel

1. It must return an instance of the type *EDisGo*.
2. The first positional argument is the MV network district id (as int). It is prepended to the tuple of arguments `func_arguments`

Returns **containers** – Dict of EDisGo instances keyed by its ID

Return type dict of *EDisGo*

edisgo.tools.edisgo_run.**edisgo_run**()

edisgo.tools.geo module

edisgo.tools.geo.**proj2equidistant**(*srid*)

Transforms to equidistant projection (epsg:3035).

Parameters **srid** (*int*) – Spatial reference identifier of geometry to transform.

Returns

Return type `functools.partial()`

edisgo.tools.geo.**proj2equidistant_reverse**(*srid*)

Transforms back from equidistant projection to given projection.

Parameters **srid** (*int*) – Spatial reference identifier of geometry to transform.

Returns

Return type `functools.partial()`

edisgo.tools.geo.**proj_by_srids**(*srid1*, *srid2*)

Transforms from specified projection to other specified projection.

Parameters

- **srid1** (*int*) – Spatial reference identifier of geometry to transform.

- **srid2** (*int*) – Spatial reference identifier of destination CRS.

Returns

Return type functools.partial()

Notes

Projections often used are conformal projection (epsg:4326), equidistant projection (epsg:3035) and spherical mercator projection (epsg:3857).

edisgo.tools.geo.**calc_geo_lines_in_buffer**(*grid_topology*, *bus*, *grid*, *buffer_radius=2000*, *buffer_radius_inc=1000*)

Determines lines that are at least partly within buffer around given bus.

If there are no lines, the buffer specified in *buffer_radius* is successively extended by *buffer_radius_inc* until lines are found.

Parameters

- **grid_topology** (*Topology*) –
- **bus** (pandas.Series) – Data of origin bus the buffer is created around. Series has same rows as columns of *buses_df*.
- **grid** (*Grid*) – Grid whose lines are searched.
- **buffer_radius** (*float, optional*) – Radius in m used to find connection targets. Default: 2000.
- **buffer_radius_inc** (*float, optional*) – Radius in m which is incrementally added to *buffer_radius* as long as no target is found. Default: 1000.

Returns List of lines in buffer (meaning close to the bus) sorted by the lines' representatives.

Return type list(str)

edisgo.tools.geo.**calc_geo_dist_vincenty**(*grid_topology*, *bus_source*, *bus_target*, *branch_detour_factor=1.3*)

Calculates the geodesic distance between two buses in km.

The detour factor in config_grid is incorporated in the geodesic distance.

Parameters

- **grid_topology** (*Topology*) –
- **bus_source** (*str*) – Name of source bus as in index of *buses_df*.
- **bus_target** (*str*) – Name of target bus as in index of *buses_df*.
- **branch_detour_factor** (*float*) – Detour factor to consider that two buses can usually not be connected directly. Default: 1.3.

Returns Distance in km.

Return type float

edisgo.tools.geo.**find_nearest_bus**(*point*, *bus_target*)

Finds the nearest bus in *bus_target* to a given point.

Parameters

- **point** (shapely.Point) – Point to find nearest bus for.

- **bus_target** (pandas.DataFrame) – Dataframe with candidate buses and their positions given in 'x' and 'y' columns. The dataframe has the same format as *buses_df*.

Returns Tuple that contains the name of the nearest bus and its distance.

Return type tuple(str, float)

edisgo.tools.geo.**find_nearest_conn_objects**(*grid_topology*, *bus*, *lines*, *conn_diff_tolerance=0.0001*)

Searches all lines for the nearest possible connection object per line.

It picks out 1 object out of 3 possible objects: 2 line-adjacent buses and 1 potentially created branch tee on the line (using perpendicular projection). The resulting stack (list) is sorted ascending by distance from bus.

Parameters

- **grid_topology** (*Topology*) –
- **bus** (pandas.Series) – Data of bus to connect. Series has same rows as columns of *buses_df*.
- **lines** (*list (str)*) – List of line representatives from index of *lines_df*.
- **conn_diff_tolerance** (*float, optional*) – Threshold which is used to determine if 2 objects are at the same position. Default: 0.0001.

Returns List of connection objects. Each object is represented by dict with representative, shapely object and distance to node.

Return type list(dict)

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 `network.results.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 network district polygon from oedb for plotting.

edisgo.tools.plots.**mv_grid_topology**(*edisgo_obj*, *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*, *curtailment_df=None*, ***kwargs*)
Plot line loading as color on lines.

Displays line loading relative to nominal capacity.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **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 storage units. 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 maximum voltage at each node. Voltages of nodes in MV network must be provided by parameter *voltage*.
  - 'voltage_deviation' Node color is set according to voltage deviation from 1 p.u.. Voltages of nodes in MV network must be provided by parameter *voltage*.
  - 'storage_integration' Only storage units 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.
  - 'curtailment' Plots curtailment per node. Size of node corresponds to share of curtailed power for the given time span. When this option is chosen a dataframe with curtailed power per time step and node needs to be provided in parameter *curtailment_df*.
  - 'charging_park' Plots nodes with charging stations in red.
- **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 network 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 network 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.
- **curtailment_df** (pandas.DataFrame) – Dataframe with curtailed power per time step and node. Columns of the dataframe correspond to buses and index to the time step. Only needs to be provided if *node_color* is set to 'curtailment'.
- **legend_loc** (*str*) – Location of legend. See matplotlib legend location options for more information. Default: 'upper left'.

edisgo.tools.powermodels_io module

edisgo.tools.powermodels_io.**to_powermodels**(*pypsa_net*)
Convert pypsa network to network dictionary format, using the pypower structure as an intermediate steps

powermodels network dictionary: https://lanl-ansi.github.io/PowerModels.jl/stable/network-data/

pypower caseformat: https://github.com/rwl/PYPOWER/blob/master/pypower/caseformat.py

Parameters **pypsa_net** –

Returns

edisgo.tools.powermodels_io.**convert_storage_series**(*timeseries*)

edisgo.tools.powermodels_io.**add_storage_from_edisgo**(*edisgo_obj*, *psa_net*, *pm_dict*)
Read static storage data (position and capacity) from eDisGo and export to Powermodels dict

edisgo.tools.powermodels_io.**pypsa2ppc**(*psa_net*)
Converter from pypsa data structure to pypower data structure

adapted from pandapower's pd2ppc converter

https://github.com/e2nIEE/pandapower/blob/911f300a96ee0ac062d82f7684083168ff052586/pandapower/pd2ppc.py

edisgo.tools.powermodels_io.**ppc2pm**(*ppc*, *psa_net*)
converter from pypower datastructure to powermodels dictionary,

adapted from pandapower to powermodels converter: https://github.com/e2nIEE/pandapower/blob/develop/pandapower/converter/powermodels/to_pm.py

Parameters **ppc** –

Returns

edisgo.tools.preprocess_pypsa_opf_structure module

edisgo.tools.preprocess_pypsa_opf_structure.**preprocess_pypsa_opf_structure**(*edisgo_grid*, *psa_network*, *hvmv_trafo=False*)
Prepares pypsa network for OPF problem.

- adds line costs
- adds HV side of HV/MV transformer to network

- moves slack to HV side of HV/MV transformer

Parameters

- **edisgo_grid** (*EDisGo*) –
- **psa_network** (pypsa.Network) –
- **hvmv_trafo** (Boolean) – If True, HV side of HV/MV transformer is added to buses and Slack generator is moved to HV side.

edisgo.tools.preprocess_pypsa_opf_structure.**aggregate_fluct_generators**(*psa_network*)

Aggregates fluctuating generators of same type at the same node.

Iterates over all generator buses. If multiple fluctuating generators are attached, they are aggregated by type.

Parameters **psa_network** (pypsa.Network) –

edisgo.tools.tools module

edisgo.tools.tools.**select_worstcase_snapshots**(*edisgo_obj*)

Select two worst-case snapshots from time series

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

1. **Maximum Residual Load: refers to the point in the time series where the** (load - generation) achieves its maximum.
2. **Minimum Residual Load: refers to the point in the time series where the** (load - generation) achieves its minimum.

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 **edisgo_obj** (*EDisGo*) –

Returns Dictionary with keys 'min_residual_load' and 'max_residual_load'. Values are corresponding worst-case snapshots of type pandas.Timestamp.

Return type dict

edisgo.tools.tools.**calculate_relative_line_load**(*edisgo_obj*, *lines=None*, *timesteps=None*)

Calculates relative line loading for specified lines and time steps.

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

Parameters

- **edisgo_obj** (*EDisGo*) –
- **lines** (*list(str) or None, optional*) – Line names/representatives of lines to calculate line loading for. If None, line loading is calculated for all lines in the network. 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 power flow analysis was conducted for 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

edisgo.tools.tools.**calculate_line_reactance**(*line_inductance_per_km*, *line_length*, *num_parallel*)

Calculates line reactance in Ohm.

Parameters

- **line_inductance_per_km** (*float or array-like*) – Line inductance in mH/km.
- **line_length** (*float*) – Length of line in km.
- **num_parallel** (*int*) – Number of parallel lines.

Returns Reactance in Ohm

Return type float

edisgo.tools.tools.**calculate_line_resistance**(*line_resistance_per_km*, *line_length*, *num_parallel*)

Calculates line resistance in Ohm.

Parameters

- **line_resistance_per_km** (*float or array-like*) – Line resistance in Ohm/km.
- **line_length** (*float*) – Length of line in km.
- **num_parallel** (*int*) – Number of parallel lines.

Returns Resistance in Ohm

Return type float

edisgo.tools.tools.**calculate_apparent_power**(*nominal_voltage*, *current*, *num_parallel*)

Calculates apparent power in MVA from given voltage and current.

Parameters

- **nominal_voltage** (*float or array-like*) – Nominal voltage in kV.
- **current** (*float or array-like*) – Current in kA.
- **num_parallel** (*int or array-like*) – Number of parallel lines.

Returns Apparent power in MVA.

Return type float

edisgo.tools.tools.**drop_duplicated_indices**(*dataframe*, *keep='first'*)

Drop rows of duplicate indices in dataframe.

Parameters

- **dataframe** (pandas.DataFrame) – handled dataframe
- **keep** (*str*) – indicator of row to be kept, 'first', 'last' or False, see pandas.DataFrame.drop_duplicates() method

edisgo.tools.tools.**drop_duplicated_columns**(*df*, *keep='first'*)

Drop columns of dataframe that appear more than once.

Parameters

- **df** (pandas.DataFrame) – Dataframe of which columns are dropped.
- **keep** (*str*) – Indicator of whether to keep first ('first'), last ('last') or none (False) of the duplicated columns. See *drop_duplicates()* method of pandas.DataFrame.

edisgo.tools.tools.**select_cable**(*edisgo_obj*, *level*, *apparent_power*)
Selects suitable cable type and quantity using given apparent power.

Cable is selected to be able to carry the given *apparent_power*, no load factor is considered. Overhead lines are not considered in choosing a suitable cable.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **level** (*str*) – Grid level to get suitable cable for. Possible options are 'mv' or 'lv'.
- **apparent_power** (*float*) – Apparent power the cable must carry in MVA.

Returns

- pandas.Series – Series with attributes of selected cable as in equipment data and cable type as series name.
- *int* – Number of necessary parallel cables.

edisgo.tools.tools.**assign_feeder**(*edisgo_obj*, *mode='mv_feeder'*)
Assigns MV or LV feeder to each bus and line, depending on the *mode*.

The feeder name is written to a new column *mv_feeder* or *lv_feeder* in *Topology*'s *buses_df* and *lines_df*. The MV respectively LV feeder name corresponds to the name of the first bus in the respective feeder.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **mode** (*str*) – Specifies whether to assign MV or LV feeder. Valid options are 'mv_feeder' or 'lv_feeder'. Default: 'mv_feeder'.

edisgo.tools.tools.**get_path_length_to_station**(*edisgo_obj*)
Determines path length from each bus to HV-MV station.

The path length is written to a new column *path_length_to_station* in *buses_df* dataframe of *Topology* class.

Parameters **edisgo_obj** (*EDisGo*) –

Returns Series with bus name in index and path length to station as value.

Return type pandas.Series

edisgo.tools.tools.**assign_voltage_level_to_component**(*edisgo_obj*, *df*)
Adds column with specification of voltage level component is in.

The voltage level ('mv' or 'lv') is determined based on the nominal voltage of the bus the component is connected to. If the nominal voltage is smaller than 1 kV, voltage level 'lv' is assigned, otherwise 'mv' is assigned.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **df** (pandas.DataFrame) – Dataframe with component names in the index. Only required column is column 'bus', giving the name of the bus the component is connected to.

Returns Same dataframe as given in parameter *df* with new column 'voltage_level' specifying the voltage level the component is in (either 'mv' or 'lv').

Return type pandas.DataFrame

edisgo.tools.tools.**get_weather_cells_intersecting_with_grid_district**(*edisgo_obj*)
Get all weather cells that intersect with the grid district.

Parameters **edisgo_obj** (*EDisGo*) –

Returns Set with weather cell IDs

Return type set

Module contents

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

1.9 What's New

Changelog for each release.

- *Release v0.1.1*
- *Release v0.1.0*
- *Release v0.0.10*
- *Release v0.0.9*
- *Release v0.0.8*
- *Release v0.0.7*
- *Release v0.0.6*
- *Release v0.0.5*
- *Release v0.0.3*
- *Release v0.0.2*

1.9.1 Release v0.1.1

Release date: July 22, 2022

This release comes with some minor additions and bug fixes.

Changes

- Added a pull request and issue template
- Fix readthecods API doc
- Consider parallel lines in calculation of x, r and s_nom
- Bug fix calculation of x and r of new lines
- Bug fix of getting a list of all weather cells within grid district

1.9.2 Release v0.1.0

Release date: July 26, 2021

This release comes with some major refactoring. The internal data structure of the network topologies was changed from a networkx graph structure to a pandas dataframe structure based on the PyPSA data structure. This comes along with major API changes. Not all functionality of the previous eDisGo release 0.0.10 is yet refactored (e.g. the heuristics for grid supportive storage integration and generator curtailment), but we are working on it and the upcoming releases will have the full functionality again.

Besides the refactoring we added extensive tests along with automatic testing with GitHub Actions and coveralls tool to track test coverage.

Further, from now on python 3.6 is not supported anymore. Supported python versions are 3.7, 3.8 and 3.9.

Changes

- Major refactoring #159
- Added support for Python 3.7, 3.8 and 3.9 #181
- Added GitHub Actions for testing and coverage #180
- Adapted to new ding0 release #184 - loads and generators in the same building are now connected to the same bus instead of separate buses and loads and generators in aggregated load areas are connected via a MV/LV station instead of directly to the HV/MV station)
- Added charging points as new components along with a methodology to integrate them into the grid
- Added multiperiod optimal power flow based on julia package PowerModels.jl optimizing storage positioning and/or operation as well as generator dispatch with regard to minimizing grid expansion costs

1.9.3 Release v0.0.10

Release date: October 18, 2019

Changes

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

Bug fixes

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

1.9.4 Release v0.0.9

Release date: December 3, 2018

Changes

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

1.9.5 Release v0.0.8

Release date: October 29, 2018

Changes

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

1.9.6 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.

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

1.9.7 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.

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

1.9.8 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.

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

1.9.9 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.

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

1.9.10 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.

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.

1.10 Index

Index

Bibliography

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

Python Module Index

e

edisgo.flex_opt.check_tech_constraints, 66
edisgo.flex_opt.costs, 70
edisgo.flex_opt.exceptions, 71
edisgo.flex_opt.reinforce_grid, 72
edisgo.flex_opt.reinforce_measures, 73
edisgo.io.ding0_import, 76
edisgo.io.generators_import, 76
edisgo.io.pypsa_io, 76
edisgo.io.timeseries_import, 78
edisgo.network.components, 35
edisgo.network.grids, 41
edisgo.network.results, 44
edisgo.network.timeseries, 50
edisgo.network.topology, 56
edisgo.opf.results.opf_expand_network, 80
edisgo.opf.results.opf_result_class, 82
edisgo.opf.run_mp_opf, 79
edisgo.opf.timeseries_reduction, 79
edisgo.opf.util.plot_solutions, 83
edisgo.opf.util.scenario_settings, 83
edisgo.tools, 95
edisgo.tools.config, 83
edisgo.tools.edisgo_run, 84
edisgo.tools.geo, 86
edisgo.tools.plots, 88
edisgo.tools.powermodels_io, 91
edisgo.tools.preprocess_pypsa_opf_structure, 91
edisgo.tools.tools, 92

Index

Symbols

_grids (*edisgo.network.topology.Topology attribute*), 57

A

active_power_timeseries (*edisgo.network.components.Generator attribute*), 38
active_power_timeseries (*edisgo.network.components.Load attribute*), 37
add_basemap() (*in module edisgo.tools.plots*), 89
add_bus() (*edisgo.network.topology.Topology method*), 63
add_charging_point() (*edisgo.network.topology.Topology method*), 62
add_charging_points_timeseries() (*in module edisgo.network.timeseries*), 55
add_component() (*edisgo.EDisGo method*), 34
add_generator() (*edisgo.network.topology.Topology method*), 62
add_generators_timeseries() (*in module edisgo.network.timeseries*), 55
add_line() (*edisgo.network.topology.Topology method*), 62
add_load() (*edisgo.network.topology.Topology method*), 61
add_loads_timeseries() (*in module edisgo.network.timeseries*), 55
add_storage_from_edisgo() (*in module edisgo.tools.powermodels_io*), 91
add_storage_unit() (*edisgo.network.topology.Topology method*), 62
add_storage_units_timeseries() (*in module edisgo.network.timeseries*), 55
aggregate_components() (*edisgo.EDisGo method*), 32
aggregate_fluct_generators() (*in module edisgo.tools.preprocess_pypsa_opf_structure*), 92
analyze() (*edisgo.EDisGo method*), 31
annual_consumption (*edisgo.network.components.Load attribute*), 37
assign_feeder() (*in module edisgo.tools.tools*), 94
assign_voltage_level_to_component() (*in module edisgo.tools.tools*), 94

B

BasicComponent (*class in edisgo.network.components*), 35
branch (*edisgo.network.components.Switch attribute*), 40
bus (*edisgo.network.components.Component attribute*), 36
bus_closed (*edisgo.network.components.Switch attribute*), 40
bus_names_to_ints() (*in module edisgo.opf.run_mp_opf*), 79
bus_open (*edisgo.network.components.Switch attribute*), 40
buses_df (*edisgo.network.grids.Grid attribute*), 42
buses_df (*edisgo.network.grids.LVGrid attribute*), 43
buses_df (*edisgo.network.grids.MVGrid attribute*), 43
buses_df (*edisgo.network.topology.Topology attribute*), 59
BusVariables (*class in edisgo.opf.results.opf_result_class*), 82

C

calc_geo_dist_vincenty() (*in module edisgo.tools.geo*), 87
calc_geo_lines_in_buffer() (*in module edisgo.tools.geo*), 87
calculate_apparent_power() (*in module edisgo.tools.tools*), 93

calculate_line_reactance() (*in module edisgo.tools.tools*), 92
calculate_line_resistance() (*in module edisgo.tools.tools*), 93
calculate_relative_line_load() (*in module edisgo.tools.tools*), 92
change_line_type() (*edisgo.network.topology.Topology method*), 64
charging_points_active_power (*edisgo.network.timeseries.TimeSeries attribute*), 52
charging_points_df (*edisgo.network.grids.Grid attribute*), 42
charging_points_df (*edisgo.network.topology.Topology attribute*), 57
charging_points_reactive_power (*edisgo.network.timeseries.TimeSeries attribute*), 52
check_ten_percent_voltage_deviation() (*in module edisgo.flex_opt.check_tech_constraints*), 70
check_timeseries_for_index_and_cols() (*in module edisgo.network.timeseries*), 56
close() (*edisgo.network.components.Switch method*), 40
Component (*class in edisgo.network.components*), 36
Config (*class in edisgo.tools.config*), 83
config (*edisgo.EDisGo attribute*), 30
connect_to_lv() (*edisgo.network.topology.Topology method*), 65
connect_to_mv() (*edisgo.network.topology.Topology method*), 64
convert() (*in module edisgo.opf.run_mp_opf*), 79
convert_storage_series() (*in module edisgo.tools.powermodels_io*), 91

D

draw() (*edisgo.network.grids.LVGrid method*), 44
draw() (*edisgo.network.grids.MVGrid method*), 43
drop_duplicated_columns() (*in module edisgo.tools.tools*), 93
drop_duplicated_indices() (*in module edisgo.tools.tools*), 93
dump_solution_file() (*edisgo.opf.results.opf_result_class.OPFResults method*), 82

E

EDisGo (*class in edisgo*), 27
edisgo.flex_opt.check_tech_constraints (*module*), 66
edisgo.flex_opt.costs (*module*), 70
edisgo.flex_opt.exceptions (*module*), 71
edisgo.flex_opt.reinforce_grid (*module*), 72
edisgo.flex_opt.reinforce_measures (*module*), 73
edisgo.io.ding0_import (*module*), 76
edisgo.io.generators_import (*module*), 76
edisgo.io.pypsa_io (*module*), 76
edisgo.io.timeseries_import (*module*), 78
edisgo.network.components (*module*), 35
edisgo.network.grids (*module*), 41
edisgo.network.results (*module*), 44
edisgo.network.timeseries (*module*), 50
edisgo.network.topology (*module*), 56
edisgo.opf.results.opf_expand_network (*module*), 80
edisgo.opf.results.opf_result_class (*module*), 82
edisgo.opf.run_mp_opf (*module*), 79
edisgo.opf.timeseries_reduction (*module*), 79
edisgo.opf.util.plot_solutions (*module*), 83
edisgo.opf.util.scenario_settings (*module*), 83
edisgo.tools (*module*), 95
edisgo.tools.config (*module*), 83
edisgo.tools.edisgo_run (*module*), 84
edisgo.tools.geo (*module*), 86
edisgo.tools.plots (*module*), 88
edisgo.tools.powermodels_io (*module*), 91
edisgo.tools.preprocess_pypsa_opf_structure (*module*), 91
edisgo.tools.tools (*module*), 92
edisgo_obj (*edisgo.network.components.BasicComponent attribute*), 36
edisgo_obj (*edisgo.network.grids.Grid attribute*), 41
edisgo_obj (*in module edisgo.flex_opt.costs*), 70
edisgo_object (*edisgo.network.results.Results attribute*), 44
edisgo_run() (*in module edisgo.tools.edisgo_run*), 86
efficiency_in (*edisgo.network.components.Storage attribute*), 39
efficiency_out (*edisgo.network.components.Storage attribute*), 39
equipment_changes (*edisgo.network.results.Results attribute*), 46
equipment_data (*edisgo.network.topology.Topology attribute*), 60
Error, 71
expand_network() (*in module edisgo.opf.results.opf_expand_network*), 80

F

find_nearest_bus() (*in module edisgo.tools.geo*), 87
find_nearest_conn_objects() (*in module edisgo.tools.geo*), 88
fixed_cosphi() (*in module edisgo.network.timeseries*), 56
from_csv() (*edisgo.network.results.Results method*), 50
from_csv() (*edisgo.network.timeseries.TimeSeries method*), 53
from_csv() (*edisgo.network.topology.Topology method*), 66

G

Generator (*class in edisgo.network.components*), 37
generators (*edisgo.network.grids.Grid attribute*), 41
generators_active_power (*edisgo.network.timeseries.TimeSeries attribute*), 51
generators_df (*edisgo.network.grids.Grid attribute*), 41
generators_df (*edisgo.network.topology.Topology attribute*), 57
generators_reactive_power (*edisgo.network.timeseries.TimeSeries attribute*), 51
GeneratorVariables (*class in edisgo.opf.results.opf_result_class*), 82
geom (*edisgo.network.components.Component attribute*), 36
get() (*in module edisgo.tools.config*), 84
get_component_timeseries() (*in module edisgo.network.timeseries*), 53
get_connected_components_from_bus() (*edisgo.network.topology.Topology method*), 61
get_connected_lines_from_bus() (*edisgo.network.topology.Topology method*), 61
get_curtailment_per_node() (*in module edisgo.opf.results.opf_expand_network*), 81
get_default_config_path() (*in module edisgo.tools.config*), 84
get_grid_district_polygon() (*in module edisgo.tools.plots*), 89
get_line_connecting_buses() (*edisgo.network.topology.Topology method*), 61
get_linked_steps() (*in module edisgo.opf.timeseries_reduction*), 80
get_load_curtailment_per_node() (*in module edisgo.opf.results.opf_expand_network*), 81
get_neighbours() (*edisgo.network.topology.Topology method*), 61
get_path_length_to_station() (*in module edisgo.tools.tools*), 94
get_steps_curtailment() (*in module edisgo.opf.timeseries_reduction*), 79
get_steps_storage() (*in module edisgo.opf.timeseries_reduction*), 80
get_weather_cells_intersecting_with_grid_district() (*in module edisgo.tools.tools*), 94
graph (*edisgo.network.grids.Grid attribute*), 41
Grid (*class in edisgo.network.grids*), 41
grid (*edisgo.network.components.BasicComponent attribute*), 36
grid (*edisgo.network.components.Component attribute*), 36
grid (*edisgo.network.components.Switch attribute*), 40
grid_district (*edisgo.network.topology.Topology attribute*), 60
grid_expansion_costs (*edisgo.network.results.Results attribute*), 46
grid_expansion_costs() (*in module edisgo.flex_opt.costs*), 70
grid_expansion_costs() (*in module edisgo.opf.results.opf_expand_network*), 80
grid_losses (*edisgo.network.results.Results attribute*), 47

H

histogram() (*in module edisgo.tools.plots*), 88
histogram_relative_line_load() (*edisgo.EDisGo method*), 33
histogram_voltage() (*edisgo.EDisGo method*), 33
hv_mv_station_load() (*in module edisgo.flex_opt.check_tech_constraints*), 68

I

i_res (*edisgo.network.results.Results attribute*), 45
id (*edisgo.network.components.BasicComponent attribute*), 36
id (*edisgo.network.grids.Grid attribute*), 41
id (*edisgo.network.topology.Topology attribute*), 60
import_ding0_grid() (*edisgo.EDisGo method*), 30
import_ding0_grid() (*in module edisgo.io.ding0_import*), 76
import_feedin_timeseries() (*in module edisgo.io.timeseries_import*), 78
import_generators() (*edisgo.EDisGo method*), 31
import_load_timeseries() (*in module edisgo.network.timeseries*), 56
ImpossibleVoltageReduction, 71

integrate_component() (*edisgo.EDisGo method*), 34
integrate_curtailment_as_load() (*in module edisgo.opf.results.opf_expand_network*), 82
integrate_storage_units() (*in module edisgo.opf.results.opf_expand_network*), 81

L

line_expansion_costs() (*in module edisgo.flex_opt.costs*), 71
lines_allowed_load() (*in module edisgo.flex_opt.check_tech_constraints*), 67
lines_df (*edisgo.network.grids.Grid attribute*), 42
lines_df (*edisgo.network.topology.Topology attribute*), 59
lines_relative_load() (*in module edisgo.flex_opt.check_tech_constraints*), 67
LineVariables (*class in edisgo.opf.results.opf_result_class*), 82
Load (*class in edisgo.network.components*), 36
load_config() (*in module edisgo.tools.config*), 84
loads (*edisgo.network.grids.Grid attribute*), 41
loads_active_power (*edisgo.network.timeseries.TimeSeries attribute*), 51
loads_df (*edisgo.network.grids.Grid attribute*), 41
loads_df (*edisgo.network.topology.Topology attribute*), 57
loads_reactive_power (*edisgo.network.timeseries.TimeSeries attribute*), 52
LoadVariables (*class in edisgo.opf.results.opf_result_class*), 82
lv_grids (*edisgo.network.grids.MVGrid attribute*), 43
lv_line_load() (*in module edisgo.flex_opt.check_tech_constraints*), 67
lv_voltage_deviation() (*in module edisgo.flex_opt.check_tech_constraints*), 69
LVGrid (*class in edisgo.network.grids*), 43

M

make_directory() (*in module edisgo.tools.config*), 84
max_hours (*edisgo.network.components.Storage attribute*), 39
MaximumIterationError, 71
measures (*edisgo.network.results.Results attribute*), 44
mode (*in module edisgo.flex_opt.costs*), 70
mv_grid (*edisgo.network.topology.Topology attribute*), 60
mv_grid_topology() (*in module edisgo.tools.plots*), 89
mv_line_load() (*in module edisgo.flex_opt.check_tech_constraints*), 66
mv_lv_station_load() (*in module edisgo.flex_opt.check_tech_constraints*), 68
mv_voltage_deviation() (*in module edisgo.flex_opt.check_tech_constraints*), 68
MVGrid (*class in edisgo.network.grids*), 43

N

nominal_capacity (*edisgo.network.components.Storage attribute*), 39
nominal_power (*edisgo.network.components.Generator attribute*), 37
nominal_power (*edisgo.network.components.Storage attribute*), 38
nominal_voltage (*edisgo.network.grids.Grid attribute*), 41

O

oedb() (*in module edisgo.io.generators_import*), 76
open() (*edisgo.network.components.Switch method*), 40
operation (*edisgo.network.components.Storage attribute*), 39
opf_settings() (*in module edisgo.opf.util.scenario_settings*), 83
OPFResults (*class in edisgo.opf.results.opf_result_class*), 82

P

peak_generation_capacity (*edisgo.network.grids.Grid attribute*), 42
peak_generation_capacity_per_technology (*edisgo.network.grids.Grid attribute*), 42
peak_load (*edisgo.network.components.Load attribute*), 37
peak_load (*edisgo.network.grids.Grid attribute*), 43
peak_load_per_sector (*edisgo.network.grids.Grid attribute*), 43
perform_mp_opf() (*edisgo.EDisGo method*), 31
pfa_p (*edisgo.network.results.Results attribute*), 44
pfa_q (*edisgo.network.results.Results attribute*), 45
pfa_slack (*edisgo.network.results.Results attribute*), 47
pfa_v_ang_seed (*edisgo.network.results.Results attribute*), 48

pfa_v_mag_pu_seed (*edisgo.network.results.Results attribute*), 48
plot_line_expansion() (*in module edisgo.opf.util.plot_solutions*), 83
plot_mv_grid() (*edisgo.EDisGo method*), 33
plot_mv_grid_expansion_costs() (*edisgo.EDisGo method*), 32
plot_mv_grid_topology() (*edisgo.EDisGo method*), 32
plot_mv_line_loading() (*edisgo.EDisGo method*), 32
plot_mv_storage_integration() (*edisgo.EDisGo method*), 33
plot_mv_voltages() (*edisgo.EDisGo method*), 32
ppc2pm() (*in module edisgo.tools.powermodels_io*), 91
preprocess_pypsa_opf_structure() (*in module edisgo.tools.preprocess_pypsa_opf_structure*), 91
process_pfa_results() (*in module edisgo.io.pypsa_io*), 78
proj2equidistant() (*in module edisgo.tools.geo*), 86
proj2equidistant_reverse() (*in module edisgo.tools.geo*), 86
proj_by_srids() (*in module edisgo.tools.geo*), 86
pypsa2ppc() (*in module edisgo.tools.powermodels_io*), 91

Q

q_sign (*edisgo.network.components.Storage attribute*), 39

R

reactive_power_timeseries (*edisgo.network.components.Generator attribute*), 38
reactive_power_timeseries (*edisgo.network.components.Load attribute*), 37
read_from_json() (*in module edisgo.opf.results.opf_result_class*), 82
read_solution_file() (*edisgo.opf.results.opf_result_class.OPFResults method*), 82
reduce_memory() (*edisgo.EDisGo method*), 35
reduce_memory() (*edisgo.network.results.Results method*), 48
reduce_memory() (*edisgo.network.timeseries.TimeSeries method*), 52
reinforce() (*edisgo.EDisGo method*), 31
reinforce_grid() (*in module edisgo.flex_opt.reinforce_grid*), 72
reinforce_hv_mv_station_overloading() (*in module edisgo.flex_opt.reinforce_measures*), 73
reinforce_lines_overloading() (*in module edisgo.flex_opt.reinforce_measures*), 75
reinforce_lines_voltage_issues() (*in module edisgo.flex_opt.reinforce_measures*), 74
reinforce_mv_lv_station_overloading() (*in module edisgo.flex_opt.reinforce_measures*), 73
reinforce_mv_lv_station_voltage_issues() (*in module edisgo.flex_opt.reinforce_measures*), 74
remove_bus() (*edisgo.network.topology.Topology method*), 63
remove_charging_point() (*edisgo.network.topology.Topology method*), 63
remove_component() (*edisgo.EDisGo method*), 35
remove_generator() (*edisgo.network.topology.Topology method*), 63
remove_line() (*edisgo.network.topology.Topology method*), 63
remove_load() (*edisgo.network.topology.Topology method*), 63
remove_storage_unit() (*edisgo.network.topology.Topology method*), 63
residual_load (*edisgo.network.timeseries.TimeSeries attribute*), 52
Results (*class in edisgo.network.results*), 44
results (*edisgo.EDisGo attribute*), 30
rings (*edisgo.network.topology.Topology attribute*), 60
run_edisgo_basic() (*in module edisgo.tools.edisgo_run*), 84
run_edisgo_pool() (*in module edisgo.tools.edisgo_run*), 85
run_edisgo_pool_flexible() (*in module edisgo.tools.edisgo_run*), 86
run_edisgo_twice() (*in module edisgo.tools.edisgo_run*), 85
run_mp_opf() (*in module edisgo.opf.run_mp_opf*), 79

S

s_res (*edisgo.network.results.Results attribute*), 46
save() (*edisgo.EDisGo method*), 33
sector (*edisgo.network.components.Load attribute*), 37
select_cable() (*in module edisgo.tools.tools*), 93
select_worstcase_snapshots() (*in module edisgo.tools.tools*), 92
session_scope() (*in module edisgo.tools*), 95
set_bus_variables() (*edisgo.opf.results.opf_result_class.OPFResults method*), 82

set_gen_variables() (*edisgo.opf.results.opf_result_class.OPFResults method*), 83
set_line_variables() (*edisgo.opf.results.opf_result_class.OPFResults method*), 82
set_load_variables() (*edisgo.opf.results.opf_result_class.OPFResults method*), 83
set_seed() (*in module edisgo.io.pypsa_io*), 78
set_solution() (*edisgo.opf.results.opf_result_class.OPFResults method*), 82
set_solution_to_results() (*edisgo.opf.results.opf_result_class.OPFResults method*), 82
set_strg_variables() (*edisgo.opf.results.opf_result_class.OPFResults method*), 83
setup_logging() (*in module edisgo.tools.edisgo_run*), 84
soc_initial (*edisgo.network.components.Storage attribute*), 39
standing_loss (*edisgo.network.components.Storage attribute*), 39
state (*edisgo.network.components.Switch attribute*), 40
station (*edisgo.network.grids.Grid attribute*), 41
Storage (*class in edisgo.network.components*), 38
storage_units_active_power (*edisgo.network.timeseries.TimeSeries attribute*), 52
storage_units_df (*edisgo.network.grids.Grid attribute*), 41
storage_units_df (*edisgo.network.topology.Topology attribute*), 58
storage_units_reactive_power (*edisgo.network.timeseries.TimeSeries attribute*), 52
StorageVariables (*class in edisgo.opf.results.opf_result_class*), 82
subtype (*edisgo.network.components.Generator attribute*), 38
Switch (*class in edisgo.network.components*), 39
switch_disconnectors (*edisgo.network.grids.Grid attribute*), 42
switch_disconnectors_df (*edisgo.network.grids.Grid attribute*), 42
switches_df (*edisgo.network.topology.Topology attribute*), 59

T

timeindex (*edisgo.network.timeseries.TimeSeries attribute*), 51
TimeSeries (*class in edisgo.network.timeseries*), 50
timeseries (*edisgo.EDisGo attribute*), 30
timeseries (*edisgo.network.components.Storage attribute*), 38
timesteps_load_feedin_case (*edisgo.network.timeseries.TimeSeries attribute*), 52
to_csv() (*edisgo.network.results.Results method*), 49
to_csv() (*edisgo.network.timeseries.TimeSeries method*), 53
to_csv() (*edisgo.network.topology.Topology method*), 66
to_graph() (*edisgo.EDisGo method*), 30
to_graph() (*edisgo.network.topology.Topology method*), 66
to_powermodels() (*in module edisgo.tools.powermodels_io*), 91
to_pypsa() (*edisgo.EDisGo method*), 30
to_pypsa() (*in module edisgo.io.pypsa_io*), 76
Topology (*class in edisgo.network.topology*), 56
topology (*edisgo.EDisGo attribute*), 30
topology (*edisgo.network.components.BasicComponent attribute*), 36
transformers_df (*edisgo.network.grids.LVGrid attribute*), 43
transformers_df (*edisgo.network.grids.MVGrid attribute*), 43
transformers_df (*edisgo.network.topology.Topology attribute*), 58
transformers_hvmv_df (*edisgo.network.topology.Topology attribute*), 58
type (*edisgo.network.components.Generator attribute*), 37
type (*edisgo.network.components.Switch attribute*), 40

U

unresolved_issues (*edisgo.network.results.Results attribute*), 48
update_number_of_parallel_lines() (*edisgo.network.topology.Topology method*), 64

V

v_res (*edisgo.network.results.Results attribute*), 45
voltage_diff() (*in module edisgo.flex_opt.check_tech_constraints*), 69
voltage_level (*edisgo.network.components.BasicComponent attribute*), 36

W

weather_cell_id (*edisgo.network.components.Generator attribute*), 38
weather_cells (*edisgo.network.grids.Grid attribute*), 42

without_generator_import (*in module edisgo.flex_opt.costs*), 70