eDisGo Documentation

Release 0.2.1

open$_e Go - Team$

Feb 21, 2024

CONTENTS

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
  - SimBEV and TracBEV for charging demand data of electric vehicles, respectively potential charging point locations
- 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
- Implementation of different charging strategies of electric vehicles
- Multiperiod optimal power flow based on julia package PowerModels.jl optimizing storage positioning and/or operation (Currently not maintained) as well as generator dispatch with regard to minimizing grid expansion costs
- Temporal complexity reduction
- Heuristic for grid-supportive generator curtailment (Currently not maintained)
- Heuristic grid-supportive battery storage integration (Currently not maintained)

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

ONE

CONTENTS

1.1 Getting started

1.1.1 Installation using Linux

Warning: Make sure to use python 3.8 or higher!

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

```
python -m pip install edisgo
```

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

1.1.2 Installation using Windows

Warning: Make sure to use python 3.8 or higher!

For Windows users we recommend using Anaconda and to install the geo stack 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
```

1.1.3 Installation using MacOS

We don't have any experience with our package on MacOS yet! If you try eDisGo on MacOS we would be happy if you let us know about your experience!

1.1.4 Requirements for edisgoOPF package

Warning: The non-linear optimal power flow is currently not maintained and might not work out of the box!

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 https://licences.stfc.ac.uk/product/coin-hsl 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"))
```

1.1.5 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.

1.1.6 A minimum working example

Following you find short examples on how to use eDisGo to set up a network and time series information for loads and generators in the network and afterwards conduct a power flow analysis and determine possible grid expansion needs and costs. 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. If you do not have an example grid, you can download one here.

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 - the EDisGo object provides the top-level API for
```

(continues on next page)

(continued from previous page)

```
# invocation of data import, power flow analysis, network reinforcement,
# flexibility measures, etc..
edisgo_obj = EDisGo(ding0_grid="ding0_example_grid")

# Import scenario for future generator park from the oedb
edisgo_obj.import_generators(generator_scenario="nep2035")

# Set up feed-in and load time series (here for a worst case analysis)
edisgo_obj.set_time_series_worst_case_analysis()

# Conduct power flow analysis (non-linear power flow using PyPSA)
edisgo_obj.analyze()

# Do grid reinforcement
edisgo_obj.reinforce()

# Determine costs for each line/transformer that was reinforced
costs = edisgo_obj.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 generator park scenario NEP2035
edisgo_obj = EDisGo(
    ding0_grid="ding0_example_grid",
    generator_scenario="nep2035"
)

# Set up your own time series by load sector and generator type (these are dummy
# time series!)
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),
     "cts": [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)
```

(continues on next page)

(continued from previous page)

```
     },
    index=timeindex)

# Before you can set the time series to the edisgo_obj you need to set the time
# index (this could also be done upon initialisation of the edisgo_obj) - the time
# index specifies which time steps to consider in power flow analysis
edisgo_obj.set_timeindex(timeindex)

# Now you can set the active power time series of loads and generators in the grid
edisgo_obj.set_time_series_active_power_predefined(
    conventional_loads_ts=timeseries_load,
    fluctuating_generators_ts=timeseries_generation_fluctuating,
    dispatchable_generators_ts=timeseries_generation_dispatchable
)

# Before you can now run a power flow analysis and determine grid expansion needs,
# reactive power time series of the loads and generators also need to be set. If you
# simply want to use default configurations, you can do the following.
edisgo_obj.set_time_series_reactive_power_control()

# Now you are ready to determine grid expansion needs
edisgo_obj.reinforce()

# Determine cost for each line/transformer that was reinforced
costs = edisgo_obj.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 with generator park scenario NEP2035 and time index
timeindex = pd.date_range("1/1/2011", periods=4, freq="H")
edisgo_obj = EDisGo(
    ding0_grid="ding0_example_grid",
    generator_scenario="nep2035",
    timeindex=timeindex
)

# Set up your own time series by load sector and generator type (these are dummy
# time series!)
# Set up active power time series of loads and generators in the grid using prede-
# fined profiles per load sector and technology type
# (There are currently no predefined profiles for dispatchable generators, wherefore
# their feed-in profiles need to be provided)
timeseries_generation_dispatchable = pd.DataFrame(
    {"biomass": [1] * len(timeindex),
     "coal": [1] * len(timeindex),
     "other": [1] * len(timeindex)
     },
    index=timeindex
```

(continues on next page)

(continued from previous page)

```
)
edisgo_obj.set_time_series_active_power_predefined(
    conventional_loads_ts="demandlib",
    fluctuating_generators_ts="oedb",
    dispatchable_generators_ts=timeseries_generation_dispatchable
)

# Before you can now run a power flow analysis and determine grid expansion needs,
# reactive power time series of the loads and generators also need to be set. Here,
# default configurations are again used.
edisgo_obj.set_time_series_reactive_power_control()

# Do grid reinforcement
edisgo_obj.reinforce()

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

1.1.7 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 understanding 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. The class `Electromobility` holds data on charging processes (how long cars are parking at a charging station, how much they need to charge, etc.) necessary to apply different charging strategies, as well as information on potential charging sites and integrated charging parks. The class `HeatPump` holds data on heat pump COP, heat demand to be served by the heat pumps and thermal storage units, which is necessary to determine flexibility potential of heat pumps. 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 Electromobility data container object
edisgo.electromobility

# Access HeatPump data container object
edisgo.heat_pump

# 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)
# the following yields the same
list(edisgo.topology.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 by providing its ID (e.g. 1) or name (e.g. "LVGrid_1")
lv_grid = edisgo.topology.get_lv_grid("LVGrid_402945")

# 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 Component time series

There are various options how to set active and reactive power time series. First, options for setting active power time series are explained, followed by options for setting reactive power time series. You can also check out the *A minimum working example* section to get a quick start.

Active power time series

There are various options how to set active time series:

- "manual": providing your own time series
- "worst-case": using simultaneity factors from config files
- "predefined": using predefined profiles, e.g. standard load profiles
- "optimised": using the LOPF to optimise e.g. vehicle charging
- "heuristic": using heuristics

Manual

Use this mode to provide your own time series for specific components. It can be invoked as follows:

```
edisgo.set_time_series_manual()
```

See *set_time_series_manual* for more information.

When using this mode make sure to previously set the time index. This can either be done upon initialisation of the EDisGo object by providing the input parameter 'timeindex' or by using the function *set_timeindex*.

Worst-case

Use this mode to set feed-in and load in heavy load flow case (here called "load_case") and/or reverse power flow case (here called "feed-in_case") using simultaneity factors used in conventional grid planning. It can be invoked as follows:

```
edisgo.set_time_series_worst_case_analysis()
```

See *set_time_series_worst_case_analysis* for more information.

When using this mode a fictitious time index starting 1/1/1970 00:00 is automatically set. This is done because pypsa needs time indeces. To find out which time index corresponds to which case check out:

```
edisgo.timeseries.timeindex_worst_cases
```

Predefined

Use this mode if you want to set time series by component type. You may either provide your own time series or use ones provided through the OpenEnergy DataBase or other python tools. This mode can be invoked as follows:

```
edisgo.set_time_series_active_power_predefined()
```

For the following components you can use existing time series:

- Fluctuating generators: Feed-in time series for solar and wind power plants can be retrieved from the OpenEnergy DataBase.
- Conventional loads: Standard load profiles for the different sectors residential, commercial, agricultural and industrial are generated using the oemof demandlib.

For all other components you need to provide your own time series. Time series for heat pumps cannot be set using this mode. See `set_time_series_active_power_predefined` for more information.

When using this mode make sure to previously set the time index. This can either be done upon initialisation of the EDisGo object by providing the input parameter 'timeindex' or by using the function `set_timeindex`.

Optimised

Use this mode to optimise flexibilities, e.g. charging of electric vehicles or dispatch of heat pumps with thermal storage units.

Todo: Add more details once the optimisation is merged.

Heuristic

Use this mode to use heuristics to set time series. So far, only heuristics for electric vehicle charging are implemented. The charging strategies can be invoked as follows:

```
edisgo.apply_charging_strategy()
```

See function docstring of `apply_charging_strategy` or documentation section *Charging strategies* for more information.

Further, there is currently one operating strategy for heat pumps implemented where the heat demand is directly served by the heat pump without buffering heat using a thermal storage. The operating strategy can be invoked as follows:

```
edisgo.apply_heat_pump_operating_strategy()
```

See function docstring of `apply_heat_pump_operating_strategy` for more information.

Reactive power time series

There are so far two options how to set reactive power time series:

- "manual": providing your own time series
- "fixed $cos\varphi$": using a fixed power factor

It is perspectively planned to also provide reactive power controls Q(U) and $cos\varphi(P)$.

Manual

See active power *Manual* mode documentation.

Fixed $cos\varphi$

Use this mode to set reactive power time series using fixed power factors. It can be invoked as follows:

```
edisgo.set_time_series_reactive_power_control()
```

See `set_time_series_reactive_power_control` for more information.

When using this mode make sure to previously set active power time series.

1.2.3 Identifying 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()`.

1.2.4 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*.

1.2.5 Electromobility

Electromobility data including charging processes as well as information on potential charging sites and integrated charging parks are stored in the `Electromobility` object.

You can access these data as follows:

```
# Access DataFrame with all SimBEV charging processes
edisgo.electromobility.charging_processes_df

# Access GeoDataFrame with all TracBEV potential charging parks
edisgo.electromobility.potential_charging_parks_gdf

# Access DataFrame with all charging parks that got integrated
edisgo.electromobility.integrated_charging_parks_df
```

The integrated charging points are also stored in the `Topology` object and can be accessed as follows:

```
# Access DataFrame with all integrated charging points.
edisgo.topology.charging_points_df
```

So far, adding electromobility data to an eDisGo object requires electromobility data from SimBEV (required version: 3083c5a) and TracBEV (required version: 14d864c) to be stored in the directories specified through the parameters simbev_directory and tracbev_directory. SimBEV provides data on standing times, charging demand, etc. per vehicle, whereas TracBEV provides potential charging point locations.

Todo: Add information on how to retrieve SimBEV and TracBEV data

Here is a small example on how to import electromobility data and apply a charging strategy. A more extensive example can be found in the example jupyter notebook electromobility_example.

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object
timeindex = pd.date_range("1/1/2011", periods=24*7, freq="H")
edisgo = EDisGo(
    ding0_grid=dingo_grid_path,
    timeindex=timeindex
)
edisgo.set_time_series_active_power_predefined(
    fluctuating_generators_ts="oedb",
    dispatchable_generators_ts=pd.DataFrame(
        data=1, columns=["other"], index=timeindex),
    conventional_loads_ts="demandlib",
)

edisgo.set_time_series_reactive_power_control()

# Resample edisgo timeseries to 15-minute resolution to match with SimBEV and
```

(continues on next page)

(continued from previous page)

```
# TracBEV data
edisgo.resample_timeseries()

# Import electromobility data
edisgo.import_electromobility(
    simbev_directory=simbev_path,
    tracbev_directory=tracbev_path,
)

# Apply charging strategy
edisgo.apply_charging_strategy(strategy="dumb")
```

Further information on the electromobility integration methodology and the charging strategies can be found in section *Electromobility integration*.

1.2.6 Heat pumps

Heat pump data including the heat pump's time variant COP, heat demand to be served as well as thermal storage capacities are stored in the *HeatPump* object.

You can access these data as follows:

```
# Access DataFrame with COP time series
edisgo.heat_pump.cop_df

# Access DataFrame with heat demand time series
edisgo.heat_pump.heat_demand_df

# Access DataFrame with information on thermal storage capacities
edisgo.heat_pump.thermal_storage_units_df
```

The heat pumps themselves are also stored in the *Topology* object and can be accessed as follows:

```
# Access DataFrame with all integrated heat pumps
edisgo.topology.loads_df[edisgo.topology.loads_df.type == "heat_pump"]
```

Here is a small example on how to integrate a heat pump and apply an operating strategy.

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object
timeindex = pd.date_range("1/1/2011", periods=4, freq="H")
edisgo = EDisGo(
    ding0_grid=dingo_grid_path,
    timeindex=timeindex
)

# Set up dummy heat pump data
bus = edisgo.topology.loads_df[
    edisgo.topology.loads_df.sector == "residential"].bus[0]
heat_pump_params = {"bus": bus, "p_set": 0.015, "type": "heat_pump"}
```

(continues on next page)

(continued from previous page)

```
cop = pd.Series([1.0, 2.0, 1.5, 3.4], index=timeindex)
heat_demand = pd.Series([0.01, 0.03, 0.015, 0.0], index=timeindex)

# Add heat pump to grid topology
hp_name = edisgo.add_component("load", **heat_pump_params)

# Add heat pump COP and heat demand to be served
edisgo.heat_pump.set_cop(edisgo, cop.to_frame(name=hp_name))
edisgo.heat_pump.set_heat_demand(edisgo, heat_demand.to_frame(name=hp_name))

# Apply operating strategy - this sets the heat pump's dispatch time series
# in timeseries.loads_active_power
edisgo.apply_heat_pump_operating_strategy()

hp_dispatch = edisgo.timeseries.loads_active_power.loc[:, hp_name]
hp_dispatch.plot()
```

1.2.7 Battery storage systems

Battery storage systems can be integrated into the grid as an alternative to classical grid expansion. 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)

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

# Set up worst case time series for loads, generators and storage unit
edisgo.set_time_series_worst_case_analysis()
```

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object
timeindex = pd.date_range("1/1/2011", periods=4, freq="H")
edisgo = EDisGo(
    ding0_grid=dingo_grid_path,
    generator_scenario="ego100",
```

(continues on next page)

(continued from previous page)

```
    timeindex=timeindex
)

# Add time series for loads and generators
timeseries_generation_dispatchable = pd.DataFrame(
    {"biomass": [1] * len(timeindex),
     "coal": [1] * len(timeindex),
     "other": [1] * len(timeindex)
     },
    index=timeindex
)
edisgo.set_time_series_active_power_predefined(
    conventional_loads_ts="demandlib",
    fluctuating_generators_ts="oedb",
    dispatchable_generators_ts=timeseries_generation_dispatchable
)
edisgo.set_time_series_reactive_power_control()

# Add storage unit to random bus with time series
edisgo.add_component(
    comp_type="storage_unit",
    bus=edisgo.topology.buses_df.index[3],
    p_nom=4,
    ts_active_power=pd.Series(
        [-3.4, 2.5, -3.4, 2.5],
        index=edisgo.timeseries.timeindex),
    ts_reactive_power=pd.Series(
        [0., 0., 0., 0.],
        index=edisgo.timeseries.timeindex)
)
```

To optimise storage positioning and operation eDisGo provides the options to use a heuristic (described in section *Storage integration*) or an optimal power flow approach. However, the storage integration heuristic is not yet adapted to the refactored code and therefore not available, and the OPF is not maintained and may therefore not work out of the box. Following you find 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)
```

1.2.8 Curtailment

The curtailment function is used to spatially distribute the power that is to be curtailed. To optimise which generators should be curtailed eDisGo provides the options to use a heuristics (heuristics *feedin-proportional* and *voltage-based*, in detail explained in section *Curtailment*) or an optimal power flow approach. However, the heuristics are not yet adapted to the refactored code and therefore not available, and the OPF is not maintained and may therefore not work out of the box.

In the following example the optimal power flow is used to find the optimal generator curtailment with regard to minimizing grid expansion costs for given curtailment requirements. It uses the EDisGo object from above.

```
edisgo.perform_mp_opf(
    timesteps=period,
    scenario='curtailment',
    results_path=results_path,
    curtailment_requirement=True,
    curtailment_requirement_series=[10, 20, 15, 0])
```

1.2.9 Plots

Todo: Add plotly plot option

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.

1.2.10 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 positioned at the substation's secondary side.

1.3.2 Multi period optimal power flow

Warning: The non-linear optimal power flow is currently not maintained and might not work out of the box!

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.

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.

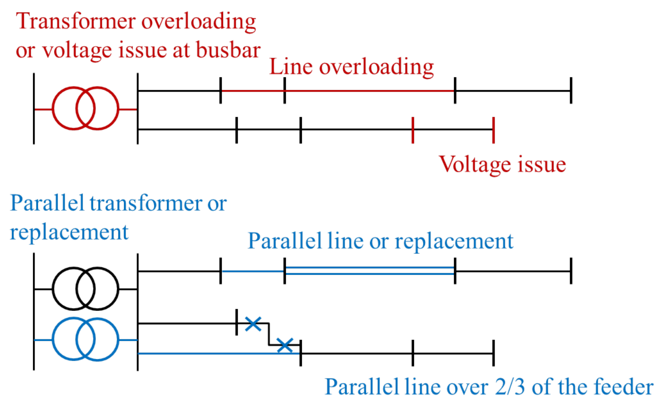
Transformer overloading or voltage issue at busbar
Line overloading
Voltage issue
Parallel transformer or replacement
Parallel line or replacement
Parallel line over 2/3 of the feeder

Fig. 1.1: Grid expansion measures

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 *split_voltage_band* of `reinforce_grid()`. By default *split_voltage_band* is set to True, resulting in separate voltage limits for MV and LV, as a combined limits 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_overload()` and `lv_line_oveload()`, 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_overload()` and `mv_lv_station_overload()`, 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_issue()` and `lv_voltage_issue()`, 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.

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

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

'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 Electromobility integration

The import and integration of electromobility data is implemented in `electromobility_import()`.

Allocation of charging demand

The allocation of charging processes to charging stations is implemented in `distribute_charging_demand()`. After electromobility data is loaded, the charging demand from SimBEV is allocated to potential charging parks from TracBEV. The allocation of the charging processes to the charging infrastructure is carried out with the help of the weighting factor of the potential charging parks determined by TracBEV. This involves a random and weighted selection of one charging park per charging process. In the case of private charging infrastructure, a separate charging point is set up for each electric vehicle (EV). All charging processes of the respective EV and charging use case are assigned to this charging point. The allocation of private charging processes to charging stations is implemented in `distribute_private_charging_demand()`.

For public charging infrastructure, the allocation is made explicitly per charging process. For each charging process it is determined whether a suitable charging point is already available. For this purpose it is checked whether the charging point is occupied by another EV in the corresponding period and whether it can provide the corresponding charging capacity. If no suitable charging point is available, a charging point is determined randomly and weighted in the same way as for private charging. The allocation of public charging processes to charging stations is implemented in `distribute_public_charging_demand()`.

Charging strategies

eDisGo right now provides three charging strategy methodologies called 'dumb', 'reduced' and 'residual', that are implemented in `charging_strategies`. The aim of the charging strategies 'reduced' and 'residual' is to generate the most grid-friendly charging behavior possible without restricting the convenience for end users. Therefore, the boundary condition of all charging strategies is that the charging requirement of each charging process must be fully covered. This means that charging processes can only be flexibilised if the EV can be fully charged while it is stationary. Furthermore, only private charging processes can be used as a flexibility, since the fulfillment of the service is the priority for public charging processes.

'dumb'

In this charging strategy the cars are charged directly after arrival with the maximum possible charging capacity.

'reduced'

This is a preventive charging strategy. The cars are charged directly after arrival with the minimum possible charging power. The minimum possible charging power is determined by the parking time and the parameter minimum_charging_capacity_factor.

'residual'

This is an active charging strategy. The cars are charged when the residual load in the MV grid is lowest (high generation and low consumption). Charging processes with a low flexibility are given priority.

1.3.6 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 storage units 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.

1.3.7 Spatial complexity reduction

eDisGo offers several methods for spatial complexity reduction. The methods reduce the number of nodes in the grid through a spatial clustering. For all methods it is a two-step procedure. At first, a busmap is determined, mapping each bus in the original grid to a new bus in the clustered grid. Then, according to the busmap, the eDisGo object is reduced. Some parts of the methods are based on the spatial clustering of [PyPSA].

You can apply the complexity reduction by calling the function `spatial_complexity_reduction`.

Important is that all grid buses have coordinates and the line length can be calculated through the Euclidean distance and a detour factor. If the grids do not match these conditions, the complexity reduction might not work as expected. If your grid does not have coordinates, you can calculate new coordinates by setting the parameter 'apply_pseudo_coordinates' to True (which is the default). Then coordinates for the grids are calculated based on the tree/radial topology of the grid graph.

The methods are selected through the 'mode' parameter. There are four methods implemented: 'kmeans', 'kmeans-dijkstra', 'aggregate_to_main_feeder' and 'equidistant_nodes'. Every node of the grid is assigned to a new node with new coordinates.

- 'kmeans' - Assignment to cluster centers of the K-Means clustering.
- 'kmeansdijkstra' - Assignment to the nearest nodes of the cluster center through the distance in the graph. For distance calculation, the Dijkstra algorithm is used.
- 'aggregate_to_main_feeder' - Assignment to the nearest node of the main feeder.
- 'equidistant_nodes' - Distribute nodes equidistant on the main feeder and then assign main feeder nodes to the new nodes.

Different cluster areas can be selected through the parameter 'cluster_area'. The cluster area is the area on which the clustering method is applied. You can choose between: 'grid', 'feeder' and 'main_feeder'. The main feeder is defined as the longest path in the feeder and is calculated with the method 'aggregate_to_main_feeder'.

The reduction factor describes how great the reduction of nodes is and can be set through the parameter 'reduction_factor'. A small reduction factor leads to a big reduction of the number of nodes and vice versa.

$$n_{\text{buses}} = k_{\text{reduction}} \cdot n_{\text{busesclusterarea}}$$

Also, there is the possibility to reduce the number of nodes to a larger degree in areas with no predicted reinforcement through the parameter 'reduction_factor_not_focused'. The areas which are not focused, are the areas that do not have components with voltage and overloading problems in the worst case power flow.

For the reduction of the grid graph, the function `apply_busmap()` is used. With this method, every line and all their parameters are recalculated and sometimes lines are combined to a new line. This is the part where the magic of reducing the grid object happens. For more information, read: [HoerschBrown] and [SCR].

If you want more flexibility in using the complexity reduction, you can also run the functions manually:

```
from edisgo.tools.spatial_complexity_reduction import make_busmap, apply_busmap
from edisgo.tools.pseudo_coordinates import make_pseudo_coordinates

# create pseudo coordinates
edisgo_obj = make_pseudo_coordinates(edisgo_obj)

# determine busmap
busmap_df = make_busmap(
    edisgo_obj,
    mode=mode,  # 'kmeans', 'kmeansdijkstra', 'aggregate_to_main_feeder' or 'equidistant_nodes
↪'
    cluster_area=cluster_area,  # 'grid' or 'feeder' or 'main_feeder'
    reduction_factor=k_rf,  # 0 < k_rf < 1
    reduction_factor_not_focused=k_rf_not_focused',  # 0 <= k_rf_not_focused < 1
)
# reduce EDisGo object
edisgo_reduced, linemap_df = apply_busmap(edisgo_obj, busmap_df)
```

For more details see the API documentation or the thesis where the methods were implemented and tested [SCR].

1.3.8 References

1.4 Notes to developers

1.4.1 Installation

Clone the repository from GitHub and change into the eDisGo directory:

```
cd eDisGo
```

Installation using Linux

To set up a source installation using linux simply use a virtual environment and install the source code with pip. Make sure to use python3.7 or higher (recommended python3.8). **After** setting up your virtual environment and activating it run the following commands within your eDisGo directory:

```
python -m pip install -e .[full]  # install eDisGo from source
pre-commit install  # install pre-commit hooks
```

Installation using Windows

For Windows users we recommend using Anaconda and to install the geo stack using the conda-forge channel prior to installing eDisGo. You may use the provided eDisGo_env_dev.yml file to do so. Create the virtual environment with:

```
conda env create -f path/to/eDisGo_env_dev.yml  # install eDisGo from source
```

Activate the newly created environment and install the pre-commit hooks with:

```
conda activate eDisGo_env_dev
pre-commit install  # install pre-commit hooks
```

This will install eDisGo with all its dependencies.

Installation using MacOS

We don't have any experience with our package on MacOS yet! If you try eDisGo on MacOS we would be happy if you let us know about your experience!

1.4.2 Code standards

- **pre-commit hooks**: Make sure to use the provided pre-commit hooks
- **pytest**: Make sure that all pytest tests are passing and add tests for every new code base
- **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

You can build the docs locally as follows (executed from top-level eDisGo directory):

```
sphinx-build -E -a -b html ./doc/ <outputdir>
```

To manually check if external links in the documentation work, you can run the following command (internal links are not checked by this):

```
sphinx-build ./doc/ -b linkcheck -d _build/doctrees _build/html
```

Internal links can be checked adding -n option when building the documentation. This will also raise warnings for type hinting, so it is a bit confusing, but can still be helpful.

```
sphinx-build -n -E -a -b html ./doc/ <outputdir>
```

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.

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.

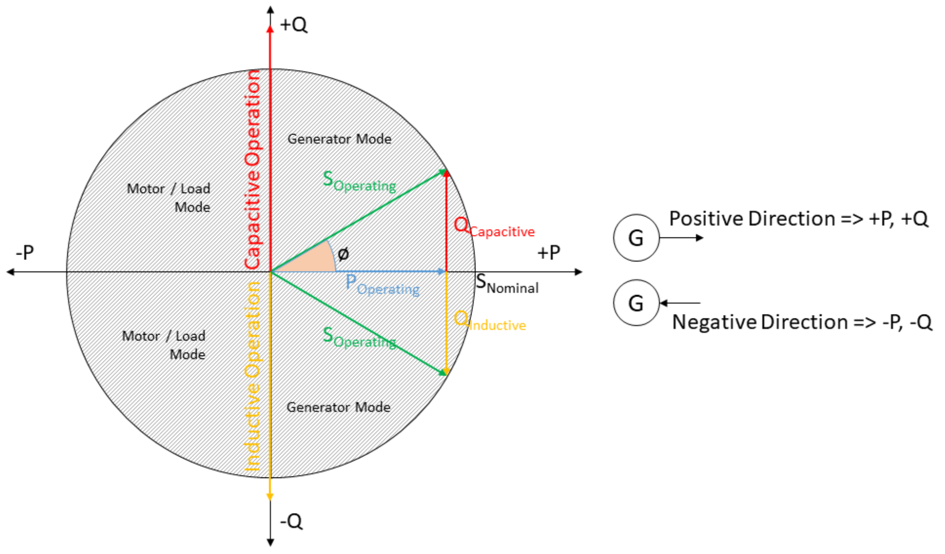

Fig. 1.2: Generator sign convention in detail

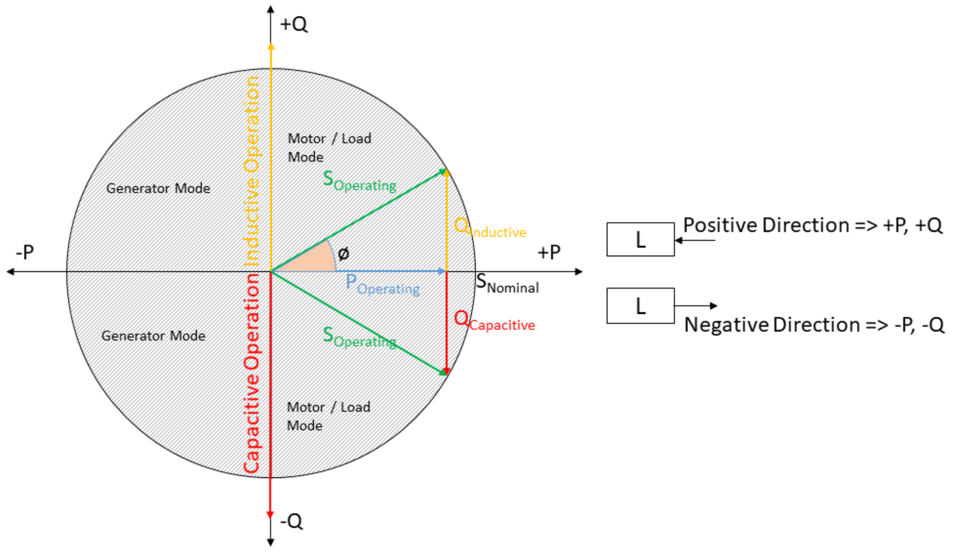

Fig. 1.3: Load sign convention in detail

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: https://edisgo.readthedocs.io/en/dev/

[data_source]

oedb_data_source = versioned

[model_draft]

conv_generators_prefix = t_ego_supply_conv_powerplant_
conv_generators_suffix = _mview
re_generators_prefix = t_ego_supply_res_powerplant_
```

(continues on next page)

(continued from previous page)

```
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: https://edisgo.readthedocs.io/en/dev/

[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_10kv = NA2XS2Y 3x1x185 RM/25
mv_line_20kv = NA2XS2Y 3x1x240
lv_line = NAYY 4x1x150

[grid_expansion_allowed_voltage_deviations]

# allowed voltage deviations
```

(continues on next page)

(continued from previous page)

```
# ==========================

# voltage at HV/MV station's secondary side
# ------------------------------------------
# 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

# COMBINED MV+LV
# --------------
# max. allowed voltage rise and drop in case voltage band is not allocated to different
#     voltage levels
#     (values according to DIN EN 50160)
#     caution: offset and control deviation at HV-MV station must be considered in␣
↪calculations!
mv_lv_max_v_rise = 0.1
mv_lv_max_v_drop = 0.1

# MV ONLY
# -------
# max. allowed voltage rise in MV grids
mv_max_v_rise = 0.05

# max. allowed voltage drop in MV grids
mv_max_v_drop = 0.015

# LV ONLY
# -------
# max. allowed voltage rise in LV grids
lv_max_v_rise = 0.035

# max. allowed voltage rise over MV/LV stations
mv_lv_station_max_v_rise = 0.015

# max. allowed voltage drop in LV grids
#     according to VDE-AR-N 4105
lv_max_v_drop = 0.065

# max. allowed voltage drop over MV/LV stations
mv_lv_station_max_v_drop = 0.02

[grid_expansion_load_factors]

# These are the load factors to use when grid issues in normal grid operation are␣
↪checked.
# Load factors for n-1 security are set in section grid_expansion_load_factors_n_minus_
```

(continues on next page)

(continued from previous page)

```
↪one.
mv_load_case_transformer = 1.0
mv_load_case_line = 1.0
mv_feed-in_case_transformer = 1.0
mv_feed-in_case_line = 1.0

lv_load_case_transformer = 1.0
lv_load_case_line = 1.0
lv_feed-in_case_transformer = 1.0
lv_feed-in_case_line = 1.0

[grid_expansion_load_factors_n_minus_one]

# These are the load factors to use when n-1 security is checked. Usually, only the
# MV grid components need to be n-1 secure.
# Source: Rehtanz et. al.: "Verteilnetzstudie für das Land Baden-Württemberg", 2017.

mv_load_case_transformer = 0.5
mv_load_case_line = 0.5
mv_feed-in_case_transformer = 1.0
mv_feed-in_case_line = 1.0

lv_load_case_transformer = 1.0
lv_load_case_line = 1.0
lv_feed-in_case_transformer = 1.0
lv_feed-in_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]
```

(continues on next page)

(continued from previous page)

```
# 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: https://edisgo.readthedocs.io/en/dev/

# 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]

# 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

# conventional load
# factors taken from "dena-Verteilnetzstudie. Ausbau- und
# Innovationsbedarf der Stromverteilnetze in Deutschland bis 2030", p. 98
mv_feed-in_case_load = 0.15
lv_feed-in_case_load = 0.1
mv_load_case_load = 1.0
lv_load_case_load = 1.0

# generators
# factors taken from "dena-Verteilnetzstudie. Ausbau- und
# Innovationsbedarf der Stromverteilnetze in Deutschland bis 2030", p. 98
feed-in_case_feed-in_pv = 0.85
feed-in_case_feed-in_wind = 1.0
feed-in_case_feed-in_other = 1.0
load_case_feed-in_pv = 0.0
load_case_feed-in_wind = 0.0
load_case_feed-in_other = 0.0
```

(continues on next page)

(continued from previous page)

```
# storage units (own values)
feed-in_case_storage = 1.0
load_case_storage = -1.0

# charging points (temporary own values)

# simultaneity in feed-in case is in dena study "Integrierte Energiewende" (p. 90) as␣
↪well assumed to be zero
mv_feed-in_case_cp_home = 0.0
mv_feed-in_case_cp_work = 0.0
mv_feed-in_case_cp_public = 0.0
mv_feed-in_case_cp_hpc = 0.0

lv_feed-in_case_cp_home = 0.0
lv_feed-in_case_cp_work = 0.0
lv_feed-in_case_cp_public = 0.0
lv_feed-in_case_cp_hpc = 0.0

# simultaneity in load case should be dependent on number of charging points in the grid
# as well as charging power
# assumed factors for home and work charging higher for LV, as simultaneity of charging
# decreases with the number of charging points

# simultaneity of 0.2 follows assumptions from dena study "Integrierte Energiewende" (p.␣
↪90) where
# simultaneity for 70-500 charging points lies around 20%
mv_load_case_cp_home = 0.2
mv_load_case_cp_work = 0.2
mv_load_case_cp_public = 1.0
mv_load_case_cp_hpc = 1.0

lv_load_case_cp_home = 1.0
lv_load_case_cp_work = 1.0
lv_load_case_cp_public = 1.0
lv_load_case_cp_hpc = 1.0

# heat pumps (temporary own values)

# simultaneity in feed-in case is in dena study "Integrierte Energiewende" (p. 90) as␣
↪well assumed to be zero
mv_feed-in_case_hp = 0.0
lv_feed-in_case_hp = 0.0

# simultaneity in load case should be dependent on number of heat pumps in the grid
# simultaneity of 0.8 follows assumptions from dena study "Integrierte Energiewende" (p.␣
↪90) where
# simultaneity for 70-500 heat pumps lies around 80%
mv_load_case_hp = 0.8
lv_load_case_hp = 1.0

[reactive_power_factor]
```

(continues on next page)

(continued from previous page)

```
# 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
mv_cp = 1.0
mv_hp = 1.0
lv_gen = 0.95
lv_load = 0.95
lv_storage = 0.95
lv_cp = 1.0
lv_hp = 1.0

[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
mv_cp = inductive
mv_hp = inductive
lv_gen = inductive
lv_load = inductive
lv_storage = inductive
lv_cp = inductive
lv_hp = inductive

[demandlib]

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

# scaling factors for night and day of weekdays and weekend days
week_day = 0.8
week_night = 0.6
weekend_day = 0.6
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: https://edisgo.readthedocs.io/en/dev/

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

[geo]

# WGS84: 4326
srid = 4326

[grid_connection]

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

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

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

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

# Upper limits for nominal capacity in MW at which generators, loads and storage units␣
↪are
# integrated into a certain voltage level. The voltage levels have the following meaning:
# voltage level 7 = LV grid
# voltage level 6 = LV side of MV-LV station
# voltage level 5 =  MV grid
# voltage level 4 = MV side of HV-MS station
# The upper limit of one voltage level at the same time signifies the lower limit of
```

(continues on next page)

(continued from previous page)

```
# the next higher voltage level, i.e. upper limit of voltage limit 7 is lower limit for
# voltage level 6.
upper_limit_voltage_level_7 = 0.1
upper_limit_voltage_level_6 = 0.2
upper_limit_voltage_level_5 = 5.5
upper_limit_voltage_level_4 = 20.0

[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 kV	I_max_th kA	R_per_km ohm/km	L_per_km mH/km	C_per_km uF/km
NAYY 4x1x300	0.4	0.419	0.1	0.279	0
NAYY 4x1x240	0.4	0.364	0.125	0.254	0
NAYY 4x1x185	0.4	0.313	0.164	0.256	0
NAYY 4x1x150	0.4	0.275	0.206	0.256	0
NAYY 4x1x120	0.4	0.245	0.253	0.256	0
NAYY 4x1x95	0.4	0.215	0.320	0.261	0
NAYY 4x1x50	0.4	0.144	0.449	0.270	0
NAYY 4x1x35	0.4	0.123	0.868	0.271	0

Table 1.3: MV cables

name #-	U_n kV	I_max_th kA	R_per_km ohm/km	L_per_km mH/km	C_per_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 kV	I_max_th kA	R_per_km ohm/km	L_per_km mH/km	C_per_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 MVA	u_kr %	P_k 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 What's New

Changelog for each release.

1.8.1 Release v0.2.1

Release date: April 2, 2023

Changes

- Added automatic resampling when applying a charging strategy #332
- Added MV grid ID to logging output #309
- Added automatic link checking #297
- Bug fix #346

1.8.2 Release v0.2.0

Release date: November 10, 2022

Changes

- added pre-commit hooks (flake8, black, isort, pyupgrade) #229
- added issue and pull request templates #220
- added Windows installation yml and documentation
- added automatic testing for Windows #317
- dropped support for python 3.7
- added functionality to set up different loggers with individual logging levels and where to write output #295
- added integrity checks of eDisGo object #231
- added functionality to save to and load from zip archive #216
- added option to not raise error in case power flow did not converge #207
- added pyplot #214
- added functionality to create geopandas dataframes #224
- added functionality to resample time series #269
- added tests
- major refactoring of loads and time series
  - restructured time series module to allow more options on how to set component time series #236
  - added charging points to Topology.loads_df to make eDisGo more flexible when further new load types are added

- peak_load in Topology.loads_df is renamed to p_nom #242
- renamed all occurences of feedin in config files to feed-in
- added simultaneity factors for heat pumps and electric vehicles
- grid reinforcement now considers separate simultaneity factors for dimensioning of LV and MV #252

- added interface to electromobility data from tools SimBEV and TracBEV (SimBEV provides data on standing times, charging demand, etc. per vehicle, whereas TracBEV provides potential charging point locations) #174 and #191

1.8.3 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

1.8.4 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

1.8.5 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).

1.8.6 Release v0.0.9

Release date: December 3, 2018

Changes

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

1.8.7 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

1.8.8 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

1.8.9 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

1.8.10 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

1.8.11 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

1.8.12 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.

1.9 Index

Index

1.10 API Reference

This page contains auto-generated API reference documentation[1].

1.10.1 `edisgo`

Subpackages

`edisgo.flex_opt`

Submodules

`edisgo.flex_opt.battery_storage_operation`

Module Contents

Functions

`apply_reference_operation`(edisgo_obj[, ...])	Applies reference storage operation to specified home storage units.

`edisgo.flex_opt.battery_storage_operation.`**`apply_reference_operation`**(*edisgo_obj*, *storage_units_names=None*, *soe_init=0.0*, *freq=1*)

[1] Created with sphinx-autoapi

Applies reference storage operation to specified home storage units.

In the reference storage operation, the home storage system is directly charged when PV feed-in is higher than electricity demand of the building until the storage is fully charged. The storage is directly discharged, in case electricity demand of the building is higher than the PV feed-in, until it is fully discharged. The battery model handles generation positive and demand negative.

To determine the PV feed-in and electricity demand of the building that the home storage is located in (including demand from heat pumps and electric vehicles), this function matches the storage units to PV plants and building electricity demand using the building ID. In case there is no electricity load or no PV system, the storage operation is set to zero.

The resulting storage units' active power time series are written to `loads_active_power`. Further, reactive power time series are set up using function `set_time_series_reactive_power_control` with default values. The state of energy time series that are calculated within this function are not written anywhere, but are returned by this function.

Parameters

- **edisgo_obj** (*EDisGo*) – EDisGo object to obtain storage units and PV feed-in and electricity demand in same building from.
- **storage_units_names** (*list(str) or None*) – Names of storage units as in `storage_units_df` to set time for. If None, time series are set for all storage units in `storage_units_df`.
- **soe_init** (*float*) – Initial state of energy of storage device in MWh. Default: 0 MWh.
- **freq** (*float*) – Frequency of provided time series. Set to one, in case of hourly time series or 0.5 in case of half-hourly time series. Default: 1.

Returns

Dataframe with time index and state of energy in MWh of each storage in columns. Column names correspond to storage name as in `storage_units_df`.

Return type

pandas.DataFrame

Notes

This function requires that the storage parameters *building_id*, *efficiency_store*, *efficiency_dispatch* and *max_hours* are set in `storage_units_df` for all storage units specified in parameter *storage_units_names*.

`edisgo.flex_opt.charging_strategies`

Module Contents

Functions

`charging_strategy`(edisgo_obj[, strategy, ...])	Applies charging strategy to set EV charging time series at charging parks.
`harmonize_charging_processes_df`(df, edisgo_obj, ...[, ...])	Harmonizes the charging processes to prevent differences in the energy

edisgo.flex_opt.charging_strategies.**charging_strategy**(*edisgo_obj: edisgo.EDisGo*, *strategy: str = 'dumb'*, *timestamp_share_threshold: numbers.Number = 0.2*, *minimum_charging_capacity_factor: numbers.Number = 0.1*)

Applies charging strategy to set EV charging time series at charging parks.

See *apply_charging_strategy* for more information.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **strategy** (*str*) – Defines the charging strategy to apply. See *strategy* parameter *apply_charging_strategy* for more information. Default: 'dumb'.
- **timestamp_share_threshold** (*float*) – Percental threshold of the time required at a time step for charging the vehicle. See *timestamp_share_threshold* parameter *apply_charging_strategy* for more information. Default: 0.2.
- **minimum_charging_capacity_factor** (*float*) – Technical minimum charging power of charging points in p.u. used in case of charging strategy 'reduced'. See *minimum_charging_capacity_factor* parameter *apply_charging_strategy* for more information. Default: 0.1.

edisgo.flex_opt.charging_strategies.**harmonize_charging_processes_df**(*df*, *edisgo_obj*, *len_ts*, *timestamp_share_threshold*, *strategy=None*, *minimum_charging_capacity_factor=0.1*, *eta_cp=1.0*)

Harmonizes the charging processes to prevent differences in the energy demand per charging strategy.

Parameters

- **df** (pandas.DataFrame) – Charging processes DataFrame.
- **len_ts** (*int*) – Length of the timeseries.
- **timestamp_share_threshold** (*float*) – See description in *charging_strategy()*.
- **strategy** (*str*) – See description in *charging_strategy()*.
- **minimum_charging_capacity_factor** (*float*) – See description in *charging_strategy()*. Default: 0.1.
- **eta_cp** (*float*) – Charging point efficiency. Default: 1.0.

edisgo.flex_opt.check_tech_constraints

Module Contents

Functions

mv_line_max_relative_overload(edisgo_obj[, n_minus_one])	Returns time step and value of most severe overloading of lines in MV network.
lv_line_max_relative_overload(edisgo_obj[, ...])	Returns time step and value of most severe overloading of lines in LV networks.
lines_allowed_load(edisgo_obj[, lines, n_minus_one])	Returns allowed loading of specified lines per time step in MVA.
lines_relative_load(edisgo_obj[, lines, n_minus_one])	Returns relative line load.
hv_mv_station_max_overload(edisgo_obj)	Checks for over-loading of HV/MV station.
mv_lv_station_max_overload(edisgo_obj[, lv_grid_id])	Checks for over-loading of MV/LV stations.
stations_allowed_load(edisgo_obj[, grids])	Returns allowed loading of specified grids stations to the overlying voltage level
stations_relative_load(edisgo_obj[, grids])	Returns relative loading of specified grids stations to the overlying voltage level
components_relative_load(edisgo_obj[, n_minus_one])	Returns relative loading of all lines and stations included in power flow analysis.
voltage_issues(edisgo_obj, voltage_level[, ...])	Gives buses with voltage issues and their maximum voltage deviation in p.u..
allowed_voltage_limits(edisgo_obj[, buses, ...])	Calculates allowed upper and lower voltage limits.
voltage_deviation_from_allowed_voltage_limi	Function to detect under- and overvoltage at buses.

edisgo.flex_opt.check_tech_constraints.**mv_line_max_relative_overload**(*edisgo_obj*, *n_minus_one=False*)

Returns time step and value of most severe overloading of lines in MV network.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **n_minus_one** (*bool*) – Determines which allowed load factors to use (see *lines_allowed_load* for more information). Currently, n-1 security cannot be handled correctly, wherefore the case where this parameter is set to True will lead to an error being raised.

Returns

Dataframe containing over-loaded MV lines, their maximum relative over-loading in p.u. (maximum calculated apparent power over allowed apparent power) 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 occurred 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_max_relative_overload`**(*edisgo_obj*, *n_minus_one=False*, *lv_grid_id=None*)

Returns time step and value of most severe overloading of lines in LV networks.

Parameters

- **edisgo_obj** (`EDisGo`) –
- **n_minus_one** (`bool`) – Determines which allowed load factors to use (see `lines_allowed_load` for more information). Currently, n-1 security cannot be handled correctly, wherefore the case where this parameter is set to True will lead to an error being raised.
- **lv_grid_id** (`str or int or None`) – If None, checks overloading for all LV lines. Otherwise, only lines in given LV grid are checked. Default: None.

Returns

Dataframe containing over-loaded LV lines, their maximum relative over-loading in p.u. (maximum calculated apparent power over allowed apparent power) 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 occurred 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*, *lines=None*, *n_minus_one=False*)

Returns allowed loading of specified lines per time step in MVA.

Allowed loading 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'.

Parameters

- **edisgo_obj** (`EDisGo`) –
- **lines** (`list(str)`) – List of line names to get allowed loading for. Per default allowed loading is returned for all lines in the network. Default: None.
- **n_minus_one** (`bool`) – Determines which allowed load factors to use. In case it is set to False, allowed load factors defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors' are used. This is the default. In case it is set to True, allowed load factors defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors_n_minus_one' are used. This case is currently not implemented.

Returns
Dataframe containing the maximum allowed apparent power per line and time step in MVA. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Columns are line names as in index of `loads_df`.

Return type
pandas.DataFrame

`edisgo.flex_opt.check_tech_constraints.`**`lines_relative_load`**(*edisgo_obj*, *lines=None*, *n_minus_one=False*)

Returns relative line load.

The relative line load is here defined as the apparent power over a line, obtained from power flow analysis, divided by the allowed load of a line, which is the nominal apparent power times a security factor (see `lines_allowed_load` for more information).

Parameters

- **`edisgo_obj`** (`EDisGo`) –
- **`lines`** (`list(str) or None`) – List of line names to get relative loading for. Per default relative loading is returned for all lines included in the power flow analysis. Default: None.
- **`n_minus_one`** (`bool`) – Determines which allowed load factors to use. See `lines_allowed_load` for more information.

Returns
Dataframe containing the relative loading per line and time step in p.u.. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Columns are line names as in index of `loads_df`.

Return type
pandas.DataFrame

`edisgo.flex_opt.check_tech_constraints.`**`hv_mv_station_max_overload`**(*edisgo_obj*)

Checks for over-loading of HV/MV station.

Parameters
`edisgo_obj` (`EDisGo`) –

Returns
In case there are no over-loading problems returns an empty dataframe. In case of over-loading problems the dataframe contains the name of the over-loaded station (grid's name with the extension '_station') in the index. Columns are 's_missing' containing the missing apparent power at maximal over-loading in MVA as float, 'time_index' containing the corresponding time step the over-loading occurred in as pandas.Timestamp, and 'grid' containing the grid object as `MVGrid`.

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_max_overload`**`(`*edisgo_obj*, *lv_grid_id=None*`)`

Checks for over-loading of MV/LV stations.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **lv_grid_id** (*str or int or None*) – If None, checks overloading for all MV/LV stations. Otherwise, only station in given LV grid is checked. Default: None.

Returns

In case there are no over-loading problems returns an empty dataframe. In case of over-loading problems the dataframe contains the name of the over-loaded station (grid's name with the extension '_station') in the index. Columns are 's_missing' containing the missing apparent power at maximal over-loading in MVA as float, 'time_index' containing the corresponding time step the over-loading occurred in as pandas.Timestamp, and 'grid' containing the grid object as *LVGrid*.

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.`**`stations_allowed_load`**`(`*edisgo_obj*, *grids=None*`)`

Returns allowed loading of specified grids stations to the overlying voltage level per time step in MVA.

Allowed loading considers allowed load factors in heavy load flow case ('load case') and reverse power flow case ('feed-in case') that are defined in the config file 'config_grid_expansion' in section 'grid_expansion_load_factors'.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **grids** (list(*Grid*)) – List of MV and LV grids to get allowed station loading for. Per default allowed loading is returned for all stations in the network. Default: None.

Returns

Dataframe containing the maximum allowed apparent power over the grid's transformers to the overlying voltage level per time step in MVA. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Column names are the respective grid's name with the extension '_station'.

Return type

pandas.DataFrame

`edisgo.flex_opt.check_tech_constraints.`**`stations_relative_load`**`(`*edisgo_obj*, *grids=None*`)`

Returns relative loading of specified grids stations to the overlying voltage level per time step in p.u..

Stations relative loading is determined by dividing the stations loading (from power flow analysis) by the allowed loading (considering allowed load factors in heavy load flow case ('load case') and reverse power flow case ('feed-in case') from config files).

Parameters

- **edisgo_obj** (*EDisGo*) –
- **grids** (list(*Grid*)) – List of MV and LV grids to get relative station loading for. Per default relative loading is returned for all stations in the network that were included in the power flow analysis. Default: None.

Returns

Dataframe containing the relative loading of the grid's transformers to the overlying voltage level per time step in p.u.. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Column names are the respective grid's name with the extension '_station'.

Return type

pandas.DataFrame

`edisgo.flex_opt.check_tech_constraints.`**`components_relative_load`**`(`*edisgo_obj*, *n_minus_one=False*`)`

Returns relative loading of all lines and stations included in power flow analysis.

The component's relative loading is determined by dividing the stations loading (from power flow analysis) by the allowed loading (considering allowed load factors in heavy load flow case ('load case') and reverse power flow case ('feed-in case') from config files).

Parameters

- **edisgo_obj** (*EDisGo*) –
- **n_minus_one** (*bool*) – Determines which allowed load factors to use. See `lines_allowed_load` for more information.

Returns

Dataframe containing the relative loading of lines and stations power flow results are available for per time step in p.u.. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Columns are line names (as in index of `loads_df`) and station names (respective grid's name with the extension '_station', see `station_name`).

Return type

pandas.DataFrame

`edisgo.flex_opt.check_tech_constraints.`**`voltage_issues`**`(`*edisgo_obj*, *voltage_level*, *split_voltage_band=True*, *lv_grid_id=None*`)`

Gives buses with voltage issues and their maximum voltage deviation in p.u..

Parameters

- **edisgo_obj** (*EDisGo*) –
- **voltage_level** (*None or str*) – Specifies voltage level for which to determine voltage issues. Possible options are 'mv' to check voltage deviations at MV buses, 'mv_lv' to check voltage deviations at MV-LV stations, and 'lv' to check voltage deviations at LV buses. If None voltage deviations in all voltage levels are checked.
- **split_voltage_band** (*bool*) – If True the allowed voltage band of +/-10 percent is allocated to the different voltage levels MV, MV/LV and LV according to config values set in section *grid_expansion_allowed_voltage_deviations*. If False, the same voltage limits are used for all voltage levels. Default: True.
- **lv_grid_id** (*str or int or None*) – This parameter is only used in case *voltage_level* is "mv_lv" or "lv". If None, checks voltage issues for all LV buses. Otherwise, only buses in given LV grid are checked. Default: None.

Returns

Dataframe with maximum deviations from allowed lower or upper voltage limits in p.u. sorted descending from highest to lowest voltage deviation (it is not distinguished between over- or undervoltage). Columns of the dataframe are 'abs_max_voltage_dev' containing the maximum absolute voltage deviation as float, 'time_index' containing the corresponding time step the maximum voltage issue occured in as pandas.Timestamp, and 'lv_grid_id' giving the LV grid ID the bus is in as integer. Index of the dataframe are the names of all buses with voltage issues as in index of `buses_df`.

Return type

pandas.DataFrame

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.`**`allowed_voltage_limits`**`(`*edisgo_obj*, *buses=None*, *split_voltage_band=True*`)`

Calculates allowed upper and lower voltage limits.

Parameters

- **`edisgo_obj`** (`EDisGo`) –
- **`buses`** (`list(str)`) – List of bus names to get allowed voltage limits for. Per default allowed voltage limits are returned for all buses in the network. Default: None.
- **`split_voltage_band`** (`bool`) – If True the allowed voltage band of +/-10 percent is allocated to the different voltage levels MV, MV/LV and LV according to config values set in section *grid_expansion_allowed_voltage_deviations*. If False, the same voltage limits are used for all voltage levels. Default: True.

Returns

Dataframe containing the maximum allowed apparent power per line and time step in MVA. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Columns are bus names as in index of `buses_df`.

Return type

pandas.DataFrame

`edisgo.flex_opt.check_tech_constraints.`**`voltage_deviation_from_allowed_voltage_limits`**`(`*edisgo_obj*, *buses=None*, *split_voltage_band=Tru*

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) or None`) – List of buses to check voltage deviation for. Per default voltage deviation is returned for all buses included in the power flow analysis. Default: None.
- **`split_voltage_band`** (`bool`) – If True the allowed voltage band of +/-10 percent is allocated to the different voltage levels MV, MV/LV and LV according to config values set in

section *grid_expansion_allowed_voltage_deviations*. If False, the same voltage limits are used for all voltage levels. Default: True.

Returns

Dataframe with deviations from allowed lower voltage level in p.u.. Positive values signify an overvoltage whereas negative values signify an undervoltage. Zero values signify that the voltage is within the allowed limits. Index of the dataframe are all time steps power flow analysis was conducted for of type pandas.Timestamp. Columns are bus names as in index of *buses_df*.

Return type

pandas.DataFrame

edisgo.flex_opt.costs

Module Contents

Functions

grid_expansion_costs(edisgo_obj[, ...])	Calculates topology expansion costs for each reinforced transformer and line
line_expansion_costs(edisgo_obj[, lines_names])	Returns costs for earthwork and per added cable in kEUR as well as voltage level

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

bool

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
[str] Transformer size or cable name

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

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

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

voltage_level
[str {'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=None*)

Returns costs for earthwork and per added cable in kEUR as well as voltage level for chosen lines.

Parameters

- **edisgo_obj** (*EDisGo*) – eDisGo object
- **lines_names** (*None or list(str)*) – List of names of lines to return cost information for. If None, it is returned for all lines in `lines_df`.

Returns
costs – Dataframe with names of lines in index and columns 'costs_earthworks' with earthwork costs in kEUR, 'costs_cable' with costs per cable/line in kEUR, and 'voltage_level' with information on voltage level the line is in.

Return type
pandas.DataFrame

edisgo.flex_opt.exceptions

Module Contents

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

Bases: `Exception`

Base class for exceptions in this module.

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

Bases: *Error*

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

message

Explanation of the error

Type
str

exception `edisgo.flex_opt.exceptions.`**`ImpossibleVoltageReduction`**(*message*)

Bases: *`Error`*

Exception raised when voltage issue cannot be solved.

`message`

Explanation of the error

Type

str

exception `edisgo.flex_opt.exceptions.`**`InfeasibleModelError`**(*message*)

Bases: *`Error`*

Exception raised when OPF can not be solved.

`message`

Explanation of the error

Type

str

`edisgo.flex_opt.heat_pump_operation`

Module Contents

Functions

`operating_strategy`(edisgo_obj[, strategy, heat_pump_names])	Applies operating strategy to set electrical load time series of heat pumps.

`edisgo.flex_opt.heat_pump_operation.`**`operating_strategy`**(*edisgo_obj*, *strategy='uncontrolled'*, *heat_pump_names=None*)

Applies operating strategy to set electrical load time series of heat pumps.

See *`apply_heat_pump_operating_strategy`* for more information.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`strategy`** (*`str`*) – Defines the operating strategy to apply. See *strategy* parameter in *`apply_heat_pump_operating_strategy`* for more information. Default: 'uncontrolled'.
- **`heat_pump_names`** (*`list(str) or None`*) – Defines for which heat pumps to apply operating strategy. See *heat_pump_names* parameter in *`apply_heat_pump_operating_strategy`* for more information. Default: None.

`edisgo.flex_opt.q_control`

Module Contents

Functions

get_q_sign_generator(reactive_power_mode)	Get the sign of reactive power in generator sign convention.
get_q_sign_load(reactive_power_mode)	Get the sign of reactive power in load sign convention.
fixed_cosphi(active_power, q_sign, power_factor)	Calculates reactive power for a fixed cosphi operation.

`edisgo.flex_opt.q_control.`**`get_q_sign_generator`**(*reactive_power_mode*)

Get the sign of reactive power in generator sign convention.

In the generator sign convention the reactive power is negative in inductive operation (*reactive_power_mode* is ‘inductive’) and positive in capacitive operation (*reactive_power_mode* is ‘capacitive’).

Parameters
`reactive_power_mode` (*str*) – Possible options are ‘inductive’ and ‘capacitive’.

Returns
Sign of reactive power in generator sign convention.

Return type
int

`edisgo.flex_opt.q_control.`**`get_q_sign_load`**(*reactive_power_mode*)

Get the sign of reactive power in load sign convention.

In the load sign convention the reactive power is positive in inductive operation (*reactive_power_mode* is ‘inductive’) and negative in capacitive operation (*reactive_power_mode* is ‘capacitive’).

Parameters
`reactive_power_mode` (*str*) – Possible options are ‘inductive’ and ‘capacitive’.

Returns
Sign of reactive power in load sign convention.

Return type
int

`edisgo.flex_opt.q_control.`**`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.flex_opt.reinforce_grid

Module Contents

Functions

reinforce_grid(→ edisgo.network.results.Results)	Evaluates network reinforcement needs and performs measures.
catch_convergence_reinforce_grid(...)	Reinforcement strategy to reinforce grids with non-converging time steps.
enhanced_reinforce_grid(→ edisgo.EDisGo)	Reinforcement strategy to reinforce grids voltage level by voltage level in case
run_separate_lv_grids(→ None)	Separate all highly overloaded LV grids within the MV grid.

edisgo.flex_opt.reinforce_grid.**reinforce_grid**(*edisgo: edisgo.EDisGo, timesteps_pfa: str | pandas.DatetimeIndex | pandas.Timestamp | None = None, reduced_analysis: bool = False, max_while_iterations: int = 20, split_voltage_band: bool = True, mode: str | None = None, without_generator_import: bool = False, n_minus_one: bool = False, **kwargs*) → *edisgo.network.results.Results*

Evaluates network reinforcement needs and performs measures.

This function is the parent function for all network reinforcements.

Parameters

- **edisgo** (*EDisGo*) – The eDisGo object grid reinforcement is conducted on.
- **timesteps_pfa** (str or pandas.DatetimeIndex or pandas.Timestamp) – timesteps_pfa specifies for which time steps power flow analysis is conducted. See parameter *timesteps_pfa* in function `reinforce` for more information.
- **reduced_analysis** (*bool*) – Specifies, whether to run reinforcement on a subset of time steps that are most critical. See parameter *reduced_analysis* in function `reinforce` for more information.
- **max_while_iterations** (*int*) – Maximum number of times each while loop is conducted. Default: 20.
- **split_voltage_band** (*bool*) – If True the allowed voltage band of +/-10 percent is allocated to the different voltage levels MV, MV/LV and LV according to config values set in section *grid_expansion_allowed_voltage_deviations*. If False, the same voltage limits are used for all voltage levels. Be aware that this does currently not work correctly. Default: True.

- **mode** (*str*) – Determines network levels reinforcement is conducted for. See parameter *mode* in function `reinforce` for more information.
- **without_generator_import** (*bool*) – If True, excludes lines that were added in the generator import to connect new generators from calculation of network expansion costs. Default: False.
- **n_minus_one** (*bool*) – Determines whether n-1 security should be checked. Currently, n-1 security cannot be handled correctly, wherefore the case where this parameter is set to True will lead to an error being raised. Default: False.
- **lv_grid_id** (*str or int or None*) – LV grid id to specify the grid to check, if mode is "lv". See parameter *lv_grid_id* in function `reinforce` for more information.
- **scale_timeseries** (*float or None*) – If a value is given, the timeseries used in the power flow analysis are scaled with this factor (values between 0 and 1 will scale down the time series and values above 1 will scale the timeseries up). Downscaling of time series can be used to gradually reinforce the grid. If None, timeseries are not scaled. Default: None.
- **skip_mv_reinforcement** (*bool*) – If True, MV is not reinforced, even if *mode* is "mv", "mvlv" or None. This is used in case worst-case grid reinforcement is conducted in order to reinforce MV/LV stations for LV worst-cases. Default: False.
- **num_steps_loading** (*int*) – In case *reduced_analysis* is set to True, this parameter can be used to specify the number of most critical overloading events to consider. If None, *percentage* is used. Default: None.
- **num_steps_voltage** (*int*) – In case *reduced_analysis* is set to True, this parameter can be used to specify the number of most critical voltage issues to select. If None, *percentage* is used. Default: None.
- **percentage** (*float*) – In case *reduced_analysis* is set to True, this parameter can be used to specify the percentage of most critical time steps to select. The default is 1.0, in which case all most critical time steps are selected. Default: 1.0.
- **use_troubleshooting_mode** (*bool*) – In case *reduced_analysis* is set to True, this parameter can be used to specify how to handle non-convergence issues in the power flow analysis. See parameter *use_troubleshooting_mode* in function `reinforce` for more information. Default: False.
- **run_initial_analyze** (*bool*) – In case *reduced_analysis* is set to True, this parameter can be used to specify whether to run an initial analyze to determine most critical time steps or to use existing results. If set to False, *use_troubleshooting_mode* is ignored. Default: True.
- **weight_by_costs** (*bool*) – In case *reduced_analysis* is set to True, this parameter can be used to specify whether to weight time steps by estimated grid expansion costs. See parameter *weight_by_costs* in `get_most_critical_time_steps()` for more information. Default: False.

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_grid.`**`catch_convergence_reinforce_grid`**(*edisgo: edisgo.EDisGo, **kwargs*) → *edisgo.network.results.Results*

Reinforcement strategy to reinforce grids with non-converging time steps.

First, conducts a grid reinforcement with only converging time steps. Afterward, tries to run reinforcement with all time steps that did not converge in the beginning. At last, if there are still time steps that do not converge, the feed-in and load time series are iteratively scaled and the grid reinforced, starting with a low grid load and scaling-up the time series until the original values are reached.

Parameters

- **`edisgo`** (*`EDisGo`*) –
- **`kwargs`** (*`dict`*) – See parameters of function *`edisgo.flex_opt.reinforce_grid.reinforce_grid()`*.

Returns

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

Return type

`Results`

`edisgo.flex_opt.reinforce_grid.`**`enhanced_reinforce_grid`**(*edisgo_object: edisgo.EDisGo, activate_cost_results_disturbing_mode: bool = False, separate_lv_grids: bool = True, separation_threshold: int | float = 2, **kwargs*) → *edisgo.EDisGo*

Reinforcement strategy to reinforce grids voltage level by voltage level in case grid reinforcement method *`edisgo.flex_opt.reinforce_grid.catch_convergence_reinforce_grid()`* is not sufficient.

In a first step, if *separate_lv_grids* is set to True, LV grids with a large load, specified through parameter *separation_threshold*, are split, so that part of the load is served by a separate MV/LV station. See *`run_separate_lv_grids()`* for more information. In a second step, all LV grids are reinforced independently. Afterwards it is tried to run the grid reinforcement for all voltage levels at once. If this fails, reinforcement is first conducted for the MV level only, afterwards for the MV level including MV/LV stations and at last for each LV grid separately. For each LV grid is it checked, if all time steps converge in the power flow analysis. If this is not the case, the grid is split. Afterwards it is tried to be reinforced. If this fails and *activate_cost_results_disturbing_mode* parameter is set to True, further measures are taken. See parameter documentation for more information.

Parameters

- **`edisgo_object`** (*`EDisGo`*) –
- **`activate_cost_results_disturbing_mode`** (*`bool`*) – If True, LV grids where normal grid reinforcement does not solve all issues, two additional approaches are used to obtain a grid where power flow can be conducted without non-convergence. These two approaches are currently not included in the calculation of grid reinforcement costs, wherefore grid reinforcement costs will be underestimated. In the first approach, all lines in the LV grid are replaced by the standard line type. Should this not be sufficient to solve non-convergence issues, all components in the LV grid are aggregated to the MV/LV station. Default: False.
- **`separate_lv_grids`** (*`bool`*) – If True, all highly overloaded LV grids are separated in a first step.

- **`separation_threshold`** (*`int or float`*) – Overloading threshold for LV grid separation. If the overloading is higher than the threshold times the total nominal apparent power of the MV/LV transformer(s) the grid is separated.
- **`kwargs`** (*`dict`*) – Keyword arguments can be all parameters of function `edisgo.flex_opt.reinforce_grid.reinforce_grid()`, except *catch_convergence_problems* which will always be set to True, *mode* which is set to None, and *skip_mv_reinforcement* which will be ignored.

Returns

The reinforced eDisGo object.

Return type

`EDisGo`

`edisgo.flex_opt.reinforce_grid.`**`run_separate_lv_grids`**(*edisgo_obj:* edisgo.EDisGo, *threshold: int | float = 2*) → None

Separate all highly overloaded LV grids within the MV grid.

The loading is approximated by aggregation of all load and generator time series and comparison with the total nominal apparent power of the MV/LV transformer(s). This approach is chosen because this method aims at resolving highly overloaded grid situations in which cases the power flow often does not converge. This method ignores grid losses and voltage deviations. Original and new LV grids can be separated multiple times if the overloading is very high.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`threshold`** (*`int or float`*) – Overloading threshold. If the overloading is higher than the threshold times the total nominal apparent power of the MV/LV transformer(s), the grid is separated.

Returns

The reinforced eDisGo object.

Return type

`EDisGo`

edisgo.flex_opt.reinforce_measures

Module Contents

Functions

reinforce_mv_lv_station_overloading(edisgo_ob ...)	Reinforce MV/LV substations due to overloading issues.
reinforce_hv_mv_station_overloading(edisgo_ob ...)	Reinforce HV/MV station due to overloading issues.
reinforce_mv_lv_station_voltage_issues(edisgo ...)	Reinforce MV/LV substations due to voltage issues.
reinforce_lines_voltage_issues(edisgo_obj, grid, ...)	Reinforce lines in MV and LV topology due to voltage issues.
reinforce_lines_overloading(edisgo_obj, crit_lines)	Reinforce lines in MV and LV topology due to overloading.
separate_lv_grid(→ tuple[dict[Any, Any], dict[str, int]])	Separate LV grid by adding a new substation and connect half of each feeder.

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_station': ['transformer_reinforced_1',
                              ...,
                              'transformer_reinforced_x'],
           'Grid_10_station': ['transformer_reinforced_10']
           },
 'removed': {'Grid_1_station': ['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_station': ['transformer_reinforced_1',
                               ...,
                               'transformer_reinforced_x'],
           'Grid_10_station': ['transformer_reinforced_10']
           },
 'removed': {'Grid_1_station': ['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** (pandas.DataFrame) – Dataframe with maximum deviations from allowed lower or upper voltage limits in p.u. for all MV-LV stations with voltage issues. For more information on dataframe see `voltage_issues`.

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 maximum deviations from allowed lower or upper voltage limits in p.u. for all buses in specified grid. For more information on dataframe see `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.

`edisgo.flex_opt.reinforce_measures.`**`separate_lv_grid`**(*edisgo_obj:* edisgo.EDisGo, *grid:* edisgo.network.grids.LVGrid) → tuple[dict[Any, Any], dict[str, int]]

Separate LV grid by adding a new substation and connect half of each feeder.

If a feeder cannot be split because it has too few nodes or too few nodes outside a building, each second inept feeder is connected to the new LV grid. The new LV grid is equipped with standard transformers until the nominal apparent power is at least the same as in the original LV grid. The new substation is at the same location as the originating substation. The workflow is as follows:

- The point at half the length of the feeders is determined.
- The first node following this point is chosen as the point where the new connection will be made.
- New MV/LV station is connected to the existing MV/LV station.
- The determined nodes are disconnected from the previous nodes and connected to the new MV/LV station.

Notes:

- The name of the new LV grid will be a combination of the originating existing grid ID. E.g. 40000 + X = 40000X
- The name of the lines in the new LV grid are the same as in the grid where the nodes were removed
- Except line names, all the data frames are named based on the new grid name

Parameters

- **edisgo_obj** (*EDisGo*) –
- **grid** (*LVGrid*) –

Returns

- *dict* – Dictionary with name of lines as keys and the corresponding number of lines added as values.
- *dict* –

  Dictionary with added transformers in the form:

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

edisgo.io

Submodules

edisgo.io.db

Module Contents

Functions

config_settings(→ dict[str, dict[str, ...)	Return a nested dictionary containing the configuration settings.
credentials(→ dict[str, str | int | pathlib.Path])	Return local database connection parameters.
ssh_tunnel(→ str)	Initialize an SSH tunnel to a remote host according to the input arguments.
engine(→ sqlalchemy.engine.base.Engine)	Engine for local or remote database.
session_scope_egon_data(engine)	Provide a transactional scope around a series of operations.
sql_grid_geom(→ geoalchemy2.types.Geometry)	
get_srid_of_db_table(→ int)	
sql_within(geom_a, geom_b, srid)	Checks if geometry a is completely within geometry b.
sql_intersects(geom_col, geom_shape, srid)	

edisgo.io.db.**config_settings**(*path: pathlib.Path | str*) → dict[str, dict[str, str | int | pathlib.Path]]

Return a nested dictionary containing the configuration settings.

It's a nested dictionary because the top level has command names as keys and dictionaries as values where the second level dictionary has command line switches applicable to the command as keys and the supplied values as values.

So you would obtain the `--database-name` configuration setting used by the current invocation of `egon-data` via

```
settings()["egon-data"]["--database-name"]
```

Parameters
path (*pathlib.Path or str*) – Path to configuration YAML file of egon-data database.

Returns
Nested dictionary containing the egon-data and optional ssh tunnel configuration settings.

Return type
dict

edisgo.io.db.**credentials**(*path: pathlib.Path | str*) → dict[str, str | int | pathlib.Path]

Return local database connection parameters.

Parameters
path (*pathlib.Path or str*) – Path to configuration YAML file of egon-data database.

Returns
Complete DB connection information.

Return type
dict

`edisgo.io.db.`**`ssh_tunnel`**(*cred: dict*) → str

Initialize an SSH tunnel to a remote host according to the input arguments. See https://sshtunnel.readthedocs.io/en/latest/ for more information.

Parameters
`cred` (*`dict`*) – Complete DB connection information.

Returns
Name of local port.

Return type
str

`edisgo.io.db.`**`engine`**(*path: pathlib.Path* | *str*, *ssh: bool = False*) → sqlalchemy.engine.base.Engine

Engine for local or remote database.

Parameters

- **`path`** (*`str`*) – Path to configuration YAML file of egon-data database.
- **`ssh`** (*`bool`*) – If True try to establish ssh tunnel from given information within the configuration YAML. If False try to connect to local database.

Returns
Database engine

Return type
sqlalchemy.Engine

`edisgo.io.db.`**`session_scope_egon_data`**(*engine: sqlalchemy.engine.base.Engine*)

Provide a transactional scope around a series of operations.

`edisgo.io.db.`**`sql_grid_geom`**(*edisgo_obj: edisgo.EDisGo*) → geoalchemy2.types.Geometry

`edisgo.io.db.`**`get_srid_of_db_table`**(*session: sqlalchemy.orm.session.Session*, *geom_col: sqlalchemy.orm.attributes.InstrumentedAttribute*) → int

`edisgo.io.db.`**`sql_within`**(*geom_a: geoalchemy2.types.Geometry*, *geom_b: geoalchemy2.types.Geometry*, *srid: int*)

Checks if geometry a is completely within geometry b.

Parameters

- **`geom_a`** (*`Geometry`*) – Geometry within *geom_b*.
- **`geom_b`** (*`Geometry`*) – Geometry containing *geom_a*.
- **`srid`** (*`int`*) – SRID geometries are transformed to in order to use the same SRID for both geometries.

`edisgo.io.db.`**`sql_intersects`**(*geom_col: sqlalchemy.orm.attributes.InstrumentedAttribute*, *geom_shape: geoalchemy2.types.Geometry*, *srid: int*)

edisgo.io.ding0_import

Module Contents

Functions

import_ding0_grid(path, edisgo_obj[, legacy_ding0_grids])	Import an eDisGo network topology from

edisgo.io.ding0_import.**import_ding0_grid**(*path*, *edisgo_obj*, *legacy_ding0_grids=True*)

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.
- **legacy_ding0_grids** (*bool*) – Allow import of old ding0 grids. Default: True.

edisgo.io.dsm_import

Module Contents

Functions

oedb(edisgo_obj, scenario, engine[, timeindex])	Gets industrial and CTS DSM profiles from the
get_profiles_per_industrial_load(load_ids, scenario, ...)	Gets industrial DSM profiles per site and OSM area.
get_profile_cts(edisgo_obj, scenario, engine)	Gets CTS DSM profiles for all CTS loads in the MV grid.

edisgo.io.dsm_import.**oedb**(*edisgo_obj: edisgo.EDisGo*, *scenario: str*, *engine: sqlalchemy.engine.base.Engine*, *timeindex=None*)

Gets industrial and CTS DSM profiles from the OpenEnergy DataBase.

Profiles comprise minimum and maximum load increase in MW as well as maximum energy pre- and postponing in MWh.

Parameters

- **edisgo_object** (*EDisGo*) –
- **scenario** (*str*) – Scenario for which to retrieve DSM data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to return data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year (2035 in case of the 'eGon2035' and to 2045

in case of the 'eGon100RE' scenario) and returned for the whole year. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using the default year and returned for the whole year.

Returns
Dictionary with DSM data with keys *p_min*, *p_max*, *e_min* and *e_max* (see `DSM` for more information). Values contain dataframes with DSM profiles per load for one year in an hourly resolution in MW. Index of the dataframes are time indices. Columns contain the load name the DSM profile is associated with as in index of `loads_df`.

Return type
dict(str, pandas.DataFrame)

`edisgo.io.dsm_import.`**`get_profiles_per_industrial_load`**(*load_ids*, *scenario: str*, *engine: sqlalchemy.engine.base.Engine*)

Gets industrial DSM profiles per site and OSM area.

Parameters

- **`load_ids`** (*`list(int)`*) – List of industrial site and OSM IDs to retrieve DSM profiles for.
- **`scenario`** (*`str`*) – Scenario for which to retrieve DSM data. Possible options are 'eGon2035' and 'eGon100RE'.
- **`engine`** (sqlalchemy.Engine) – Database engine.

Returns
Dictionary with DSM data with keys *p_min*, *p_max*, *e_min* and *e_max*. Values contain dataframes with DSM profiles per site and OSM area for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are site ID as integer.

Return type
dict(str, pandas.DataFrame)

`edisgo.io.dsm_import.`**`get_profile_cts`**(*edisgo_obj: edisgo.EDisGo*, *scenario: str*, *engine: sqlalchemy.engine.base.Engine*)

Gets CTS DSM profiles for all CTS loads in the MV grid.

Parameters

- **`edisgo_object`** (*`EDisGo`*) –
- **`scenario`** (*`str`*) – Scenario for which to retrieve DSM data. Possible options are 'eGon2035' and 'eGon100RE'.
- **`engine`** (sqlalchemy.Engine) – Database engine.

Returns
Dictionary with DSM data with keys *p_min*, *p_max*, *e_min* and *e_max*. Values contain dataframes with DSM profiles per CTS load for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are site ID as integer.

Return type
dict(str, pandas.DataFrame)

Notes

Be aware, that in this function the DSM time series are disaggregated to all CTS loads in the grid. In some cases, this can lead to an over- or underestimation of the DSM potential, as in egon_data buildings are mapped to a grid based on the zensus cell they are in whereas in ding0 buildings are mapped to a grid based on the geolocation. As it can happen that buildings lie outside an MV grid but within a zensus cell that is assigned to that MV grid, they are mapped differently in egon_data and ding0.

`edisgo.io.electromobility_import`

Module Contents

Functions

import_electromobility_from_dir(edisgo_obj, ...)	Import electromobility data from
read_csvs_charging_processes(csv_path[, mode, csv_dir])	Reads all CSVs in a given path and returns a DataFrame with all
read_simbev_config_df(path, edisgo_obj[, ...])	Get SimBEV config data.
read_gpkg_potential_charging_parks(path, edisgo_obj)	Get GeoDataFrame with all
assure_minimum_potential_charging_parks(edisg ...)	
distribute_charging_demand(edisgo_obj, **kwargs)	Distribute charging demand from SimBEV onto potential charging parks from TracBEV.
get_weights_df(edisgo_obj, ...)	Get weights per potential charging point for a given set of grid connection indices.
normalize(weights_df)	Normalize a given DataFrame so that its sum equals 1 and return a
combine_weights(potential_charging_park_indices, ...)	Add designated charging capacity weights into the initial weights and
weighted_random_choice(edisgo_obj, ...[, rng])	Weighted random choice of a potential charging park. Setting the chosen
distribute_private_charging_demand(edisgo_obj)	Distributes all private charging processes. Each car gets its own
distribute_public_charging_demand(edisgo_obj, **kwargs)	Distributes all public charging processes. For each process it is
determine_grid_connection_capacity(...[, lower_limit, ...])	
integrate_charging_parks(edisgo_obj)	Integrates all designated charging parks into the grid.
import_electromobility_from_oedb(edisgo_obj, scenario, ...)	Gets electromobility data for specified scenario from oedb.
simbev_config_from_oedb(scenario, engine)	Gets *simbev_config_df*
potential_charging_parks_from_oedb(edisgo_obj, engine)	Gets *potential_charging_parks_gdf*
charging_processes_from_oedb(edisgo_obj, engine, ...)	Gets *charging_processes_df* data

`edisgo.io.electromobility_import.`**`import_electromobility_from_dir`**(*edisgo_obj:* edisgo.EDisGo, *simbev_directory:* *pathlib.PurePath* | *str*, *tracbev_directory:* *pathlib.PurePath* | *str*, ***kwargs*)

Import electromobility data from SimBEV and TracBEV from directory.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`simbev_directory`** (*`str` or `pathlib.PurePath`*) – SimBEV directory holding SimBEV data.
- **`tracbev_directory`** (*`str` or `pathlib.PurePath`*) – TracBEV directory holding TracBEV data.
- **`kwargs`** – Kwargs may contain any further attributes you want to specify.

  gc_to_car_rate_home
  [float] Specifies the minimum rate between potential charging parks points for the use case "home" and the total number of cars. Default 0.5 .

  gc_to_car_rate_work
  [float] Specifies the minimum rate between potential charging parks points for the use case "work" and the total number of cars. Default 0.25 .

  gc_to_car_rate_public
  [float] Specifies the minimum rate between potential charging parks points for the use case "public" and the total number of cars. Default 0.1 .

  gc_to_car_rate_hpc
  [float] Specifies the minimum rate between potential charging parks points for the use case "hpc" and the total number of cars. Default 0.005 .

  mode_parking_times
  [str] If the mode_parking_times is set to "frugal" only parking times with any charging demand are imported. Default "frugal".

  charging_processes_dir
  [str] Charging processes sub-directory. Default None.

  simbev_config_file
  [str] Name of the simbev config file. Default "metadata_simbev_run.json".

`edisgo.io.electromobility_import.`**`read_csvs_charging_processes`**(*csv_path*, *mode='frugal'*, *csv_dir=None*)

Reads all CSVs in a given path and returns a DataFrame with all SimBEV charging processes.

Parameters

- **`csv_path`** (*`str`*) – Main path holding SimBEV output data
- **`mode`** (*`str`*) – Returns all information if None. Returns only rows with charging demand greater than 0 if "frugal". Default: "frugal".
- **`csv_dir`** (*`str`*) – Optional sub-directory holding charging processes CSVs under path. Default: None.

Returns
DataFrame with AGS, car ID, trip destination, charging use case (private or public), netto charging capacity, charging demand, charge start, charge end, potential charging park ID and charging point ID.

Return type
pandas.DataFrame

`edisgo.io.electromobility_import.`**`read_simbev_config_df`**(*path*, *edisgo_obj*, *simbev_config_file='metadata_simbev_run.json'*)

Get SimBEV config data.

Parameters

- **`path`** (*`str`*) – Main path holding SimBEV output data.
- **`edisgo_obj`** (*`EDisGo`*) –
- **`simbev_config_file`** (*`str`*) – SimBEV config file name. Default: "metadata_simbev_run.json".

Returns
DataFrame with used random seed, used threads, stepsize in minutes, year, scenarette, simulated days, maximum number of cars per AGS, completed standing times and time series per AGS and used ramp up data CSV.

Return type
pandas.DataFrame

`edisgo.io.electromobility_import.`**`read_gpkg_potential_charging_parks`**(*path*, *edisgo_obj*)

Get GeoDataFrame with all TracBEV potential charging parks.

Parameters

- **`path`** (*`str`*) – Main path holding TracBEV data.
- **`edisgo_obj`** (*`EDisGo`*) –

Returns
GeoDataFrame with AGS, charging use case (home, work, public or hpc), user-centric weight and geometry.

Return type
geopandas.GeoDataFrame

`edisgo.io.electromobility_import.`**`assure_minimum_potential_charging_parks`**(*edisgo_obj:* edisgo.EDisGo, *potential_charging_parks_gdf: geopandas.GeoDataFrame*, ***kwargs*)

`edisgo.io.electromobility_import.`**`distribute_charging_demand`**(*edisgo_obj*, ***kwargs*)

Distribute charging demand from SimBEV onto potential charging parks from TracBEV.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`kwargs`** – Kwargs may contain any further attributes you want to specify.

mode
[str] Distribution mode. If the mode is set to "user_friendly" only the simbev weights are used for the distribution. If the mode is "grid_friendly" also grid conditions are respected. Default "user_friendly".

generators_weight_factor
[float] Weighting factor of the generators weight within an LV grid in comparison to the loads weight. Default 0.5.

distance_weight
[float] Weighting factor for the distance between a potential charging park and its nearest substation in comparison to the combination of the generators and load factors of the LV grids. Default 1 / 3.

user_friendly_weight
[float] Weighting factor of the user friendly weight in comparison to the grid friendly weight. Default 0.5.

`edisgo.io.electromobility_import.`**`get_weights_df`**`(`*edisgo_obj*, *potential_charging_park_indices*, ***kwargs*`)`

Get weights per potential charging point for a given set of grid connection indices.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`potential_charging_park_indices`** (*`list`*) – List of potential charging parks indices
- **`mode`** (*`str`*) – Only use user friendly weights ("user_friendly") or combine with grid friendly weights ("grid_friendly"). Default: "user_friendly".
- **`user_friendly_weight`** (*`float`*) – Weight of user friendly weight if mode "grid_friendly". Default: 0.5.
- **`distance_weight`** (*`float`*) – Grid friendly weight is a combination of the installed capacity of generators and loads within a LV grid and the distance towards the nearest substation. This parameter sets the weight for the distance parameter. Default: 1/3.

Returns
DataFrame with numeric weights

Return type
pandas.DataFrame

`edisgo.io.electromobility_import.`**`normalize`**`(`*weights_df*`)`

Normalize a given DataFrame so that its sum equals 1 and return a flattened Array.

Parameters
`weights_df` (pandas.DataFrame) – DataFrame with single numeric column

Returns
Array with normalized weights

Return type
Numpy 1-D array

`edisgo.io.electromobility_import.`**`combine_weights`**`(`*potential_charging_park_indices*, *designated_charging_point_capacity_df*, *weights_df*`)`

Add designated charging capacity weights into the initial weights and normalize weights

Parameters

- **potential_charging_park_indices** (*list*) – List of potential charging parks indices
- **designated_charging_point_capacity_df** – pandas.DataFrame DataFrame with designated charging point capacity per potential charging park
- **weights_df** (pandas.DataFrame) – DataFrame with initial user or combined weights

Returns
Array with normalized weights

Return type
Numpy 1-D array

`edisgo.io.electromobility_import.`**`weighted_random_choice`**(*edisgo_obj*, *potential_charging_park_indices*, *car_id*, *destination*, *charging_point_id*, *normalized_weights*, *rng=None*)

Weighted random choice of a potential charging park. Setting the chosen values into `charging_processes_df`

Parameters

- **edisgo_obj** (*EDisGo*) –
- **potential_charging_park_indices** (*list*) – List of potential charging parks indices
- **car_id** (*int*) – Car ID
- **destination** (*str*) – Trip destination
- **charging_point_id** (*int*) – Charging Point ID
- **normalized_weights** (*Numpy 1-D array*) – Array with normalized weights
- **rng** (*Numpy random generator*) – If None a random generator with seed=charging_point_id is initialized

Returns
Chosen Charging Park ID

Return type
int

`edisgo.io.electromobility_import.`**`distribute_private_charging_demand`**(*edisgo_obj*)

Distributes all private charging processes. Each car gets its own private charging point if a charging process takes place.

Parameters
edisgo_obj (*EDisGo*) –

`edisgo.io.electromobility_import.`**`distribute_public_charging_demand`**(*edisgo_obj*, ***kwargs*)

Distributes all public charging processes. For each process it is checked if a matching charging point exists to minimize the number of charging points.

Parameters
edisgo_obj (*EDisGo*) –

`edisgo.io.electromobility_import.`**`determine_grid_connection_capacity`**(*total_charging_point_capacity*, *lower_limit=0.3*, *upper_limit=1.0*, *minimum_factor=0.45*) `edisgo.io.electromobility_import.`**`integrate_charging_parks`**(*edisgo_obj*)

Integrates all designated charging parks into the grid.

The charging time series at each charging park are not set in this function.

Parameters

edisgo_obj (*EDisGo*) –

`edisgo.io.electromobility_import.`**`import_electromobility_from_oedb`**(*edisgo_obj: edisgo.EDisGo*, *scenario: str*, *engine: sqlalchemy.engine.base.Engine*, ***kwargs*)

Gets electromobility data for specified scenario from oedb.

Electromobility data includes data on standing times, charging demand, etc. per vehicle, as well as information on potential charging point locations.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **scenario** (*str*) – Scenario for which to retrieve electromobility data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.
- **kwargs** – Possible options are *gc_to_car_rate_home*, *gc_to_car_rate_work*, *gc_to_car_rate_public*, *gc_to_car_rate_hpc*, and *mode_parking_times*. See parameter documentation of *import_electromobility_data_kwds* parameter in `import_electromobility` for more information.

`edisgo.io.electromobility_import.`**`simbev_config_from_oedb`**(*scenario: str*, *engine: sqlalchemy.engine.base.Engine*)

Gets `simbev_config_df` for specified scenario from oedb.

Parameters

- **scenario** (*str*) – Scenario for which to retrieve electromobility data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.

Returns

See `simbev_config_df` for more information.

Return type

pandas.DataFrame

`edisgo.io.electromobility_import.`**`potential_charging_parks_from_oedb`**(*edisgo_obj: edisgo.EDisGo*, *engine: sqlalchemy.engine.base.Engine*)

Gets `potential_charging_parks_gdf` data from oedb.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **engine** (sqlalchemy.Engine) – Database engine.

Returns

See `potential_charging_parks_gdf` for more information.

Return type
geopandas.GeoDataFrame

edisgo.io.electromobility_import.**charging_processes_from_oedb**(*edisgo_obj: edisgo.EDisGo*, *engine: sqlalchemy.engine.base.Engine*, *scenario: str*, ***kwargs*)

Gets *charging_processes_df* data for specified scenario from oedb.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **engine** (sqlalchemy.Engine) – Database engine.
- **scenario** (*str*) – Scenario for which to retrieve data. Possible options are 'eGon2035' and 'eGon100RE'.
- **kwargs** – Possible option is *mode_parking_times*. See parameter documentation of *import_electromobility_data_kwds* parameter in *import_electromobility* for more information.

Returns
See *charging_processes_df* for more information.

Return type
pandas.DataFrame

edisgo.io.generators_import

Module Contents

Functions

oedb_legacy(edisgo_object, generator_scenario, **kwargs)	Gets generator park for specified scenario from oedb and integrates generators into
oedb(edisgo_object, scenario, engine[, max_capacity])	Gets generator park for specified scenario from oedb and integrates generators into

edisgo.io.generators_import.**oedb_legacy**(*edisgo_object*, *generator_scenario*, ***kwargs*)

Gets generator park for specified scenario from oedb and integrates generators into the grid.

The importer uses SQLAlchemy ORM objects. These are defined in ego.io. The data is imported from the tables 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*.

Parameters

- **edisgo_object** (*EDisGo*) –
- **generator_scenario** (*str*) – Scenario for which to retrieve generator data. Possible options are 'nep2035' and 'ego100'.
- **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 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.generators_import.`**`oedb`**(*edisgo_object: edisgo.EDisGo*, *scenario: str*, *engine: sqlalchemy.engine.base.Engine*, *max_capacity=20*)

Gets generator park for specified scenario from oedb and integrates generators into the grid.

The data is imported from the tables supply.egon_chp_plants, supply.egon_power_plants and supply.egon_power_plants_pv_roof_building.

For the grid integration it is distinguished between PV rooftop plants and all other power plants. For PV rooftop the following steps are conducted:

- Removes decommissioned PV rooftop plants (plants whose source ID cannot be matched to a source ID of an existing plant).
- Updates existing PV rooftop plants. The following two cases are distinguished:
  - Nominal power increases: It is checked, if plant needs to be connected to a higher voltage level and if that is the case, the existing plant is removed from the grid and the new one integrated based on the geolocation.
  - Nominal power decreases: Nominal power of existing plant is overwritten.
- Integrates new PV rooftop plants at corresponding building ID. If the plant needs to be connected to a higher voltage level than the building, it is integrated based on the geolocation.

For all other power plants the following steps are conducted:

- Removes decommissioned power and CHP plants (all plants that do not have a source ID or whose source ID can not be matched to a new plant and are not of subtype pv_rooftop, as these are handled in a separate function)
- Updates existing power plants (plants whose source ID can be matched; solar, wind and CHP plants never have a source ID in the future scenarios and are therefore never updated). The following two cases are distinguished:

  - Nominal power increases: It is checked, if plant needs to be connected to a higher voltage level and if that is the case, the existing plant is removed from the grid and the new one integrated based on the geolocation.
  - Nominal power decreases: Nominal power of existing plant is overwritten.
- Integrates new power and CHP plants based on the geolocation.

Parameters

- **`edisgo_object`** (*`EDisGo`*) –
- **`scenario`** (*`str`*) – Scenario for which to retrieve generator data. Possible options are "eGon2035" and "eGon100RE".
- **`engine`** (sqlalchemy.Engine) – Database engine.
- **`max_capacity`** (*`float`*) – Maximum capacity in MW of power plants to retrieve from database. In general, the generators that are retrieved from the database are selected based on the voltage level they are in. In some cases, the voltage level is not correct as it was wrongly set in the MaStR dataset. To avoid having unrealistically large generators in the grids, an upper limit is also set. Per default this is 20 MW.

Notes

Note, that PV rooftop plants are queried using the building IDs not the MV grid ID as in egon_data buildings are mapped to a grid based on the zensus cell they are in whereas in ding0 buildings are mapped to a grid based on the geolocation. As it can happen that buildings lie outside an MV grid but within a zensus cell that is assigned to that MV grid, they are mapped differently in egon_data and ding0, and it is therefore better to query using the building IDs.

`edisgo.io.heat_pump_import`

Module Contents

Functions

`oedb`(edisgo_object, scenario, engine[, import_types])	Gets heat pumps for specified scenario from oedb and integrates them into the grid.
`efficiency_resistive_heaters_oedb`(scenario, engine)	Get efficiency of resistive heaters from the

`edisgo.io.heat_pump_import.`**`oedb`**(*edisgo_object*, *scenario*, *engine*, *import_types=None*)

Gets heat pumps for specified scenario from oedb and integrates them into the grid.

See *`import_heat_pumps`* for more information.

Parameters

- **`edisgo_object`** (*`EDisGo`*) –
- **`scenario`** (*`str`*) – Scenario for which to retrieve heat pump data. Possible options are "eGon2035" and "eGon100RE".
- **`engine`** (sqlalchemy.Engine) – Database engine.

- **import_types** (*list(str) or None*) – Specifies which technologies to import. Possible options are "individual_heat_pumps", "central_heat_pumps" and "central_resistive_heaters". If None, all are imported.

Returns
List with names (as in index of `loads_df`) of integrated heat pumps.

Return type
list(str)

`edisgo.io.heat_pump_import.`**`efficiency_resistive_heaters_oedb`**(*scenario*, *engine*)

Get efficiency of resistive heaters from the OpenEnergy DataBase.

Parameters

- **scenario** (*str*) – Scenario for which to retrieve efficiency data. Possible options are "eGon2035" and "eGon100RE".
- **engine** (sqlalchemy.Engine) – Database engine.

Returns
Dictionary with efficiency of resistive heaters in district and individual heating. Keys of the dictionary are "central_resistive_heater" giving the efficiency of resistive heaters in district heating and "rural_resistive_heater" giving the efficiency of resistive heaters in individual heating systems. Values are of type float and given in p.u.

Return type
dict

edisgo.io.powermodels_io

This module provides tools to convert eDisGo representation of the network topology and timeseries to PowerModels network data format and to retrieve results from PowerModels OPF in PowerModels network data format to eDisGo representation. Call `to_powermodels()` to retrieve the PowerModels network container and `from_powermodels()` to write OPF results to edisgo object.

Module Contents

Functions

`to_powermodels`(edisgo_object[, s_base, flexible_cps, ...])	Convert eDisGo representation of the network topology and timeseries to
`from_powermodels`(edisgo_object, pm_results, hv_flex_dict)	Convert results from optimization in PowerModels network data format to eDisGo data
`aggregate_parallel_transformers`(psa_net)	Calculate impedance for parallel transformers and aggregate them. Replace

`edisgo.io.powermodels_io.`**`to_powermodels`**(*edisgo_object*, *s_base=1*, *flexible_cps=None*, *flexible_hps=None*, *flexible_loads=None*, *flexible_storage_units=None*, *opf_version=1*)

Convert eDisGo representation of the network topology and timeseries to PowerModels network data format.

Parameters

- **edisgo_object** (*EDisGo*) –

- **s_base** (*int*) – Base value of apparent power for per unit system. Default: 1 MVA.
- **flexible_cps** (numpy.ndarray or None) – Array containing all charging points that allow for flexible charging.
- **flexible_hps** (numpy.ndarray or None) – Array containing all heat pumps that allow for flexible operation due to an attached heat storage.
- **flexible_loads** (numpy.ndarray or None) – Array containing all flexible loads that allow for application of demand side management strategy.
- **flexible_storage_units** (numpy.ndarray or None) – Array containing all flexible storages. Non-flexible storage units operate to optimize self consumption. Default: None.
- **opf_version** (*int*) – Version of optimization models to choose from. Must be one of [1, 2, 3, 4]. For more information see `edisgo.opf.powermodels_opf.pm_optimize()`. Default: 1.

Returns

First dictionary contains all network data in PowerModels network data format. Second dictionary contains time series of HV requirement for each flexibility retrieved from overlying_grid component of edisgo object and reduced by non-flexible components.

Return type

(dict, dict)

edisgo.io.powermodels_io.**from_powermodels**(*edisgo_object*, *pm_results*, *hv_flex_dict*, *s_base=1*)

Convert results from optimization in PowerModels network data format to eDisGo data format and updates timeseries values of flexibilities on eDisGo object.

Parameters

- **edisgo_object** (*EDisGo*) –
- **pm_results** (*dict or str*) – Dictionary or path to json file that contains all optimization results in PowerModels network data format.
- **hv_flex_dict** (*dict*) – Dictionary containing time series of HV requirement for each flexibility retrieved from overlying grid component of edisgo object.
- **s_base** (*int*) – Base value of apparent power for per unit system. Default: 1 MVA.

edisgo.io.powermodels_io.**aggregate_parallel_transformers**(*psa_net*)

Calculate impedance for parallel transformers and aggregate them. Replace psa_net.transformers dataframe by aggregated transformer dataframe.

Parameters

psa_net (PyPSA.Network) – PyPSA.Network representation of network.

edisgo.io.pypsa_io

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

Module Contents

Functions

`to_pypsa`(edisgo_object[, mode, timesteps])	Convert grid to PyPSA.Network representation.
`set_seed`(edisgo_obj, pypsa_network)	Set initial guess for the Newton-Raphson algorithm.
`process_pfa_results`(edisgo, pypsa, timesteps[, dtype])	Passing power flow results from PyPSA to `Results`.

`edisgo.io.pypsa_io.`**`to_pypsa`**(*edisgo_object*, *mode=None*, *timesteps=None*, ***kwargs*)

Convert grid to PyPSA.Network representation.

You can choose between translation of the MV and all underlying LV grids (mode=None (default)), the MV network only (mode='mv' or mode='mvlv') or a single LV network (mode='lv').

Parameters

- **edisgo_object** (`EDisGo`) – EDisGo object containing grid topology and time series information.
- **mode** (`str`) – Determines network levels that are translated to PyPSA.Network. See *mode* parameter in `to_pypsa` for more information.
- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – See *timesteps* parameter in `to_pypsa` for more information.

:param See other parameters in `to_pypsa` for more: :param information.:

Returns
PyPSA.Network representation.

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*, *dtype='float'*)

Passing power flow results from PyPSA to `Results`.

Parameters

- **edisgo** (`EDisGo`) –

- **pypsa** (pypsa.Network) – The PyPSA network to retrieve results from.
- **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, analysis

edisgo.io.storage_import

Module Contents

Functions

home_batteries_oedb(edisgo_obj, scenario, engine)	Gets home battery data from oedb and integrates them into the grid.

edisgo.io.storage_import.**home_batteries_oedb**(*edisgo_obj:* edisgo.EDisGo, *scenario:* *str*, *engine:* *sqlalchemy.engine.base.Engine*)

Gets home battery data from oedb and integrates them into the grid.

See *import_home_batteries* for more information.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **scenario** (*str*) – Scenario for which to retrieve home battery data. Possible options are "eGon2035" and "eGon100RE".
- **engine** (sqlalchemy.Engine) – Database engine.

Returns
List with names (as in index of *storage_units_df*) of integrated storage units.

Return type
list(str)

edisgo.io.timeseries_import

Module Contents

Functions

feedin_oedb_legacy(edisgo_object[, timeindex])	Import feed-in time series data for wind and solar power plants from the
feedin_oedb(edisgo_object, engine[, timeindex])	Import feed-in time series data for wind and solar power plants from the
load_time_series_demandlib(edisgo_obj[, timeindex])	Get normalized sectoral electricity load time series using the
cop_oedb(edisgo_object, engine, weather_cell_ids[, ...])	Get COP (coefficient of performance) time series data from the
heat_demand_oedb(edisgo_obj, scenario, engine[, timeindex])	Get heat demand profiles for heat pumps from the
electricity_demand_oedb(edisgo_obj, scenario, engine)	Get electricity demand profiles for all conventional loads from the
get_residential_heat_profiles_per_building(..	Gets residential heat demand profiles per building.
get_district_heating_heat_demand_profiles(...)	Gets heat demand profiles of district heating networks from oedb.
get_cts_profiles_per_building(edisgo_obj, scenario, ...)	Gets CTS heat demand profiles per CTS building for all CTS buildings in MV grid.
get_cts_profiles_per_grid(bus_id, scenario, sector, engine)	Gets CTS heat or electricity demand profiles per building for all buildings in the
get_residential_electricity_profiles_per_bui	Gets residential electricity demand profiles per building.
get_industrial_electricity_profiles_per_site ...)	Gets industrial electricity demand profiles per site and OSM area.

`edisgo.io.timeseries_import.`**`feedin_oedb_legacy`**(*edisgo_object*, *timeindex=None*)

Import feed-in time series data for wind and solar power plants from the OpenEnergy DataBase.

Parameters

- **`edisgo_obj`** (*EDisGo*) –
- **`timeindex`** (pandas.DatetimeIndex or None) – Specifies time steps for which to return feed-in data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year 2011 and returned for the whole year. If no timeindex is provided, the timeindex set in *`timeindex`* is used. If *`timeindex`* is not set, the data is indexed using the default year 2011 and returned for the whole year.

Returns

DataFrame with hourly time series for active power feed-in per generator type (wind or solar, in column level 0) and weather cell (in column level 1), normalized to a capacity of 1 MW.

Return type

pandas.DataFrame

`edisgo.io.timeseries_import.`**`feedin_oedb`**(*edisgo_object*, *engine: sqlalchemy.engine.base.Engine*, *timeindex=None*)

Import feed-in time series data for wind and solar power plants from the OpenEnergy DataBase.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **engine** (sqlalchemy.Engine) – Database engine.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to return feed-in data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year 2011 and returned for the whole year. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using the default year 2011 and returned for the whole year.

Returns
DataFrame with hourly feed-in time series per generator type (wind or solar, in column level 0) and weather cell (in column level 1), normalized to a capacity of 1 MW. Index of the dataframe depends on parameter *timeindex*.

Return type
pandas.DataFrame

edisgo.io.timeseries_import.**load_time_series_demandlib**(*edisgo_obj*, *timeindex=None*)

Get normalized sectoral electricity load time series using the demandlib.

Resulting electricity load profiles hold time series of hourly conventional electricity demand for the sectors residential, cts, agricultural and industrial. Time series are normalized to a consumption of 1 MWh per year.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to return data. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using the default year 2011 and returned for the whole year.

Returns
DataFrame with conventional electricity load time series for sectors residential, cts, agricultural and industrial. Index is a pandas.DatetimeIndex. Columns hold the sector type.

Return type
pandas.DataFrame

edisgo.io.timeseries_import.**cop_oedb**(*edisgo_object*, *engine*, *weather_cell_ids*, *timeindex=None*)

Get COP (coefficient of performance) time series data from the OpenEnergy DataBase.

Parameters

- **edisgo_object** (*EDisGo*) –
- **engine** (sqlalchemy.Engine) – Database engine.
- **weather_cell_ids** (*list(int) or list(float)*) – List (or array) of weather cell IDs to obtain COP data for.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to return data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year 2011 and returned for the whole year. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using the default year 2011 and returned for the whole year.

Returns
DataFrame with hourly COP time series in p.u. per weather cell. Index of the dataframe is a time index. Columns contain the weather cell ID as integer.

Return type
pandas.DataFrame

`edisgo.io.timeseries_import.`**`heat_demand_oedb`**(*edisgo_obj*, *scenario*, *engine*, *timeindex=None*)

Get heat demand profiles for heat pumps from the OpenEnergy DataBase.

Heat demand data is returned for all heat pumps in the grid. For more information on how individual heat demand profiles are obtained see functions *`get_residential_heat_profiles_per_building()`* and *`get_cts_profiles_per_building()`*. For more information on how district heating heat demand profiles are obtained see function *`get_district_heating_heat_demand_profiles()`*.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`scenario`** (*`str`*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **`engine`** (sqlalchemy.Engine) – Database engine.
- **`timeindex`** (pandas.DatetimeIndex or None) – Specifies time steps for which to return data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year (2035 in case of the 'eGon2035' and to 2045 in case of the 'eGon100RE' scenario) and returned for the whole year. If no timeindex is provided, the timeindex set in *`timeindex`* is used. If *`timeindex`* is not set, the data is indexed using the default year and returned for the whole year.

Returns

DataFrame with hourly heat demand for one year in MW per heat pump. Index of the dataframe is a time index. Columns contain the heat pump name as in index of *`loads_df`*.

Return type

pandas.DataFrame

`edisgo.io.timeseries_import.`**`electricity_demand_oedb`**(*edisgo_obj*, *scenario*, *engine*, *timeindex=None*, *load_names=None*)

Get electricity demand profiles for all conventional loads from the OpenEnergy DataBase.

Conventional loads comprise conventional electricity applications in the residential, CTS and industrial sector. For more information on how the demand profiles are obtained see functions *`get_residential_electricity_profiles_per_building()`*, *`get_cts_profiles_per_building()`* and *`get_industrial_electricity_profiles_per_site()`*.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –
- **`scenario`** (*`str`*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **`engine`** (sqlalchemy.Engine) – Database engine.
- **`timeindex`** (pandas.DatetimeIndex or None) – Specifies time steps for which to return data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year (2035 in case of the 'eGon2035' and to 2045 in case of the 'eGon100RE' scenario) and returned for the whole year. If no timeindex is provided, the timeindex set in *`timeindex`* is used. If *`timeindex`* is not set, the data is indexed using the default year and returned for the whole year.
- **`load_names`** (*`list(str) or None`*) – Conventional loads (as in index of *`loads_df`*) for which to retrieve electricity demand time series. If none are provided, profiles for all conventional loads are returned.

Returns
DataFrame with hourly electricity demand for one year in MW per conventional load. Index of the dataframe is a time index. Columns contain the load name as in index of `loads_df`.

Return type
pandas.DataFrame

`edisgo.io.timeseries_import.`**`get_residential_heat_profiles_per_building`**(*building_ids*, *scenario*, *engine*)

Gets residential heat demand profiles per building.

Parameters

- **`building_ids`** (*`list(int)`*) – List of building IDs to retrieve heat demand profiles for.
- **`scenario`** (*`str`*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **`engine`** (sqlalchemy.Engine) – Database engine.

Returns
Dataframe with residential heat demand profiles per building for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are building ID as integer.

Return type
pandas.DataFrame

`edisgo.io.timeseries_import.`**`get_district_heating_heat_demand_profiles`**(*district_heating_ids*, *scenario*, *engine*)

Gets heat demand profiles of district heating networks from oedb.

Parameters

- **`district_heating_ids`** (*`list(int)`*) – List of district heating area IDs to get heat demand profiles for.
- **`scenario`** (*`str`*) – Scenario for which to retrieve data. Possible options are 'eGon2035' and 'eGon100RE'.
- **`engine`** (sqlalchemy.Engine) – Database engine.

Returns
Dataframe with heat demand profiles per district heating network for one year in an hourly resolution in MW. Index contains hour of the year (from 1 to 8760) and column names are district heating network ID as integer.

Return type
pandas.DataFrame

`edisgo.io.timeseries_import.`**`get_cts_profiles_per_building`**(*edisgo_obj*, *scenario*, *sector*, *engine*)

Gets CTS heat demand profiles per CTS building for all CTS buildings in MV grid.

This function is a helper function that should not be but is necessary, as in egon_data buildings are mapped to a grid based on the zensus cell they are in whereas in ding0 buildings are mapped to a grid based on the geolocation. As it can happen that buildings lie outside an MV grid but within a zensus cell that is assigned to that MV grid, they are mapped differently in egon_data and ding0. This function therefore checks, if there are CTS loads with other grid IDs and if so, gets profiles for other grid IDs (by calling `get_cts_profiles_per_grid()` with different grid IDs) in order to obtain a demand profile for all CTS loads.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) –

- **scenario** (*str*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **sector** (*str*) – Demand sector for which profile is calculated: "electricity" or "heat"
- **engine** (sqlalchemy.Engine) – Database engine.

Returns
Dataframe with CTS demand profiles per building for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are building ID as integer.

Return type
pandas.DataFrame

edisgo.io.timeseries_import.**get_cts_profiles_per_grid**(*bus_id*, *scenario*, *sector*, *engine*)

Gets CTS heat or electricity demand profiles per building for all buildings in the given MV grid.

Parameters

- **bus_id** (*int*) – MV grid ID.
- **scenario** (*str*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **sector** (*str*) – Demand sector for which profile is calculated: "electricity" or "heat"
- **engine** (sqlalchemy.Engine) – Database engine.

Returns
Dataframe with CTS demand profiles per building for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are building ID as integer.

Return type
pandas.DataFrame

edisgo.io.timeseries_import.**get_residential_electricity_profiles_per_building**(*building_ids*, *scenario*, *engine*)

Gets residential electricity demand profiles per building.

Parameters

- **building_ids** (*list(int)*) – List of building IDs to retrieve electricity demand profiles for.
- **scenario** (*str*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.

Returns
Dataframe with residential electricity demand profiles per building for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are building ID as integer.

Return type
pandas.DataFrame

edisgo.io.timeseries_import.**get_industrial_electricity_profiles_per_site**(*site_ids*, *scenario*, *engine*)

Gets industrial electricity demand profiles per site and OSM area.

Parameters

- **site_ids** (*list(int)*) – List of industrial site and OSM IDs to retrieve electricity demand profiles for.
- **scenario** (*str*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.

Returns
Dataframe with industrial electricity demand profiles per site and OSM area for one year in an hourly resolution in MW. Index contains hour of the year (from 0 to 8759) and column names are site ID as integer.

Return type
pandas.DataFrame

edisgo.network

Submodules

edisgo.network.components

Module Contents

Classes

BasicComponent	Generic component
Component	Generic component for all components that can be considered nodes,
Load	Load object
Generator	Generator object
Storage	Storage object
Switch	Switch object
PotentialChargingParks	Generic component

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.

property id

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

Returns
Unique identifier of component.

Return type
`str`

property edisgo_obj

EDisGo container

Return type
EDisGo

property topology

Network topology container

Return type
Topology

property 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

abstract property grid

Grid component is in.

Returns
Grid component is in.

Return type
Grid

__repr__()

Return repr(self).

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

Bases: *BasicComponent*

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

property 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

property grid

Grid the component is in.

Returns
Grid object the component is in.

Return type
Grid

property geom

Geolocation of component.

Return type
shapely.Point

__repr__()

Return repr(self).

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

Bases: *Component*

Load object

property p_set

Peak load in MW.

Parameters
p_set (float) – Peak load in MW.

Returns
Peak load in MW.

Return type
float

property 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

property 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’, ‘cts’, ‘residential’, ‘industrial’.

Parameters
sector (str) –

Returns

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

property active_power_timeseries

Active power time series of load in MW.

Returns
Active power time series of load in MW.

Return type
pandas.Series

property 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: `Component`

Generator object

property 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

property 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?*)

property 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?*)

property active_power_timeseries

Active power time series of generator in MW.

Returns
Active power time series of generator in MW.

Return type
pandas.Series

property reactive_power_timeseries

Reactive power time series of generator in Mvar.

Returns
Reactive power time series of generator in Mvar.

Return type
pandas.Series

property 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: *Component*

Storage object

property nominal_power

Nominal power of storage unit in MW.

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

Returns
Nominal power of storage unit in MW.

Return type
float

property active_power_timeseries

Active power time series of storage unit in MW.

Returns
Active power time series of storage unit in MW.

Return type
pandas.Series

property reactive_power_timeseries

Reactive power time series of storage unit in Mvar.

Returns
Reactive power time series of storage unit in Mvar.

Return type
pandas.Series

property state_of_charge_timeseries

State of charge time series of storage unit in MWh.

Returns
State of charge time series of storage unit in MWh.

Return type
pandas.Series

__repr__()

Return repr(self).

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

Bases: `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 substitued by the bus specified in *bus_closed*.

property type

Type of switch.

So far edisgo only considers switch disconnectors.

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

Returns
Type of switch.

Return type
`str`

property 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`

property 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`

property state

State of switch (open or closed).

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

Return type
str

property branch

Branch the switch is represented by.

Returns
Branch the switch is represented by.

Return type
str

property grid

Grid switch is in.

Returns
Grid switch is in.

Return type
`Grid`

open()

Open switch.

close()

Close switch.

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

Bases: *BasicComponent*

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.

property 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`

property grid

Grid component is in.

Returns
Grid component is in.

Return type
Grid

property ags

8-digit AGS (Amtlicher Gemeindeschlüssel, eng. Community Identification Number) number the potential charging park is in. Number is given as `int` and leading zeros are therefore missing.

Returns
AGS number

Return type
int

property use_case

Charging use case (home, work, public or hpc) of the potential charging park.

Returns
Charging use case

Return type
str

property designated_charging_point_capacity

Total gross designated charging park capacity in kW.

This is not necessarily equal to the connection rating.

Returns
Total gross designated charging park capacity

Return type
float

property user_centric_weight

User centric weight of the potential charging park determined by SimBEV.

Returns
User centric weight

Return type
float

property geometry

Location of the potential charging park as Shapely Point object.

Returns
Location of the potential charging park.

Return type
Shapely Point object.

property nearest_substation

Determines the nearest LV Grid, substation and distance.

Returns

int
LV Grid ID

str
ID of the nearest substation

float
Distance to nearest substation

Return type
`dict`

property edisgo_id

property charging_processes_df

Determines designated charging processes for the potential charging park.

Returns
DataFrame with AGS, car ID, trip destination, charging use case (private or public), netto charging capacity, charging demand, charge start, charge end, potential charging park ID and charging point ID.

Return type
pandas.DataFrame

property grid_connection_capacity

property within_grid

Determines if the potential charging park is located within the grid district.

edisgo.network.dsm

Module Contents

Classes

DSM	Data container for demand side management potential data.

class edisgo.network.dsm.**DSM**(***kwargs*)

Data container for demand side management potential data.

property p_min

Maximum load decrease in MW.

Parameters
df (pandas.DataFrame) – Maximum load decrease in MW. Index of the dataframe is a time index and column names are names of DSM loads as in `loads_df`.

Returns
Maximum load decrease in MW. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property p_max

Maximum load increase in MW.

Parameters
df (pandas.DataFrame) – Maximum load increase in MW. Index of the dataframe is a time index and column names are names of DSM loads as in `loads_df`.

Returns
Maximum load decrease in MW. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property e_min

Maximum energy preponing in MWh.

Parameters
df (pandas.DataFrame) – Maximum energy preponing in MWh. Index of the dataframe is a time index and column names are names of DSM loads as in `loads_df`.

Returns
Maximum energy preponing in MWh. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property e_max

Maximum energy postponing in MWh.

Parameters
df (pandas.DataFrame) – Maximum energy postponing in MWh. Index of the dataframe is a time index and column names are names of DSM loads as in `loads_df`.

Returns
Maximum energy postponing in MWh. 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 to save memory.

See `reduce_memory` for more information.

Parameters

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

to_csv(*directory: str | pathlib.Path*, *reduce_memory=False*, ***kwargs*)

Exports DSM data to csv files.

The following attributes are exported:

- 'p_min' : Attribute `p_min` is saved to *p_min.csv*.
- 'p_max' : Attribute `p_max` is saved to *p_max.csv*.
- 'e_min' : Attribute `e_min` is saved to *e_min.csv*.
- 'e_max' : Attribute `e_max` is saved to *e_max.csv*.

Parameters

- **directory** (*str*) – Path to save DSM data to.
- **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.
- **kwargs** – Kwargs may contain arguments of *reduce_memory*.

from_csv(*data_path: str | pathlib.Path, from_zip_archive: bool = False*)

Restores DSM data from csv files.

Parameters

- **data_path** (*str*) – Path to DSM csv files or zip archive.
- **from_zip_archive** (*bool*) – Set to True if data is archived in a zip archive. Default: False.

check_integrity()

Check data integrity.

Checks for duplicated and missing labels as well as implausible values.

edisgo.network.electromobility

Module Contents

Classes

Electromobility	Data container for all electromobility data.

class edisgo.network.electromobility.**Electromobility**(***kwargs*)

Data container for all electromobility data.

This class holds data on charging processes (how long cars are parking at a charging station, how much they need to charge, etc.) necessary to apply different charging strategies, as well as information on potential charging sites and integrated charging parks.

property charging_processes_df

DataFrame with all charging processes.

Returns

DataFrame with AGS, car ID, trip destination, charging use case, netto charging capacity, charging demand, charge start, charge end, grid connection point and charging point ID. The columns are:

ags
[int] 8-digit AGS (Amtlicher Gemeindeschlüssel, eng. Community Identification Number). Leading zeros are missing.

car_id
[int] Car ID to differentiate charging processes from different cars.

destination
[str] SimBEV driving destination.

use_case
[str] SimBEV use case. Can be “hpc”, “home”, “public” or “work”.

nominal_charging_capacity_kW
[float] Vehicle charging capacity in kW.

grid_charging_capacity_kW
[float] Grid-sided charging capacity including charging infrastructure losses (nominal_charging_capacity_kW / eta_cp) in kW.

chargingdemand_kWh
[float] Charging demand in kWh.

park_time_timesteps
[int] Number of parking time steps.

park_start_timesteps
[int] Time step the parking event starts.

park_end_timesteps
[int] Time step the parking event ends.

charging_park_id
[int] Designated charging park ID from potential_charging_parks_gdf. Is NaN if the charging demand is not yet distributed.

charging_point_id
[int] Designated charging point ID. Is used to differentiate between multiple charging points at one charging park.

Return type
pandas.DataFrame

`property potential_charging_parks_gdf`

GeoDataFrame with all potential charging parks.

Returns

GeoDataFrame with ID as index, AGS, charging use case (home, work, public or hpc), user-centric weight and geometry. Columns are:

index
[int] Charging park ID.

use_case
[str] TracBEV use case. Can be “hpc”, “home”, “public” or “work”.

user_centric_weight
[flaot] User centric weight used in distribution of charging demand. Weight is determined by TracBEV but normalized from 0 .. 1.

geometry
[GeoSeries] Geolocation of charging parks.

Return type
geopandas.GeoDataFrame

`property potential_charging_parks`

Potential charging parks within the AGS.

Returns
List of potential charging parks within the AGS.

Return type
list(*PotentialChargingParks*)

property simbev_config_df

Dictionary containing configuration data.

Returns

DataFrame with used regio type, charging point efficiency, stepsize in minutes, start date, end date, minimum SoC for hpc, grid timeseries setting, grid timeseries by use case setting and the number of simulated days. Columns are:

regio_type
[str] RegioStaR 7 ID used in SimBEV.

eta_cp
[float or int] Charging point efficiency used in SimBEV.

stepsize
[int] Stepsize in minutes the driving profile is simulated for in SimBEV.

start_date
[datetime64] Start date of the SimBEV simulation.

end_date
[datetime64] End date of the SimBEV simulation.

soc_min
[float] Minimum SoC when an HPC event is initialized in SimBEV.

grid_timeseries
[bool] Setting whether a grid timeseries is generated within the SimBEV simulation.

grid_timeseries_by_usecase
[bool] Setting whether a grid timeseries by use case is generated within the SimBEV simulation.

days
[int] Timedelta between the end_date and start_date in days.

Return type
pandas.DataFrame

property integrated_charging_parks_df

Mapping DataFrame to map the charging park ID to the internal eDisGo ID.

The eDisGo ID is determined when integrating components using `add_component()` or `integrate_component_based_on_geolocation()` method.

Returns
Mapping DataFrame to map the charging park ID to the internal eDisGo ID.

Return type
pandas.DataFrame

property stepsize

Stepsize in minutes used in SimBEV.

Returns
Stepsize in minutes

Return type
int

property simulated_days

Number of simulated days in SimBEV.

Returns
Number of simulated days

Return type
int

property eta_charging_points

Charging point efficiency.

Returns
Charging point efficiency in p.u..

Return type
float

property flexibility_bands

Dictionary with flexibility bands (lower and upper energy band as well as upper power band).

Parameters
flex_dict (dict(str, pandas.DataFrame)) – Keys are 'upper_power', 'lower_energy' and 'upper_energy'. Values are dataframes containing the corresponding band per each charging point. Columns of the dataframe are the charging point names as in `loads_df`. Index is a time index.

Returns
See input parameter *flex_dict* for more information on the dictionary.

Return type
dict(str, pandas.DataFrame)

get_flexibility_bands(*edisgo_obj*, *use_case*, *resample=True*, *tol=1e-06*)

Method to determine flexibility bands (lower and upper energy band as well as upper power band).

Besides being returned by this function, flexibility bands are written to `flexibility_bands`.

Parameters

- **edisgo_obj** (`EDisGo`) –
- **use_case** (`str or list(str)`) – Charging point use case(s) to determine flexibility bands for.
- **resample** (`bool (optional)`) – If True, flexibility bands are resampled to the same frequency as time series data in `TimeSeries` object. If False, original frequency is kept. Default: True.
- **tol** (`float`) – Tolerance to reduce or increase flexibility band values by to fix possible rounding errors that may lead to failing integrity checks and infeasibility when used to optimise charging. See `fix_flexibility_bands_rounding_errors` for more information. To avoid this behaviour, set *tol* to 0.0. Default: 1e-6.

Returns
Keys are 'upper_power', 'lower_energy' and 'upper_energy'. Values are dataframes containing the corresponding band for each charging point of the specified use case. Columns of the dataframe are the charging point names as in `loads_df`. Index is a time index.

Return type
dict(str, pandas.DataFrame)

fix_flexibility_bands_rounding_errors(*tol=1e-06*)

Fixes possible rounding errors that may lead to failing integrity checks.

Due to rounding errors it may occur, that e.g. the upper energy band is lower than the lower energy band. This does in some cases lead to infeasibilities when used to optimise charging processes.

This function increases or reduces a flexibility band by the specified tolerance in case an integrity check fails as follows:

- If there are cases where the upper power band is not sufficient to meet the charged upper energy, the upper power band is increased for all charging points and all time steps.
- If there are cases where the lower energy band is larger than the upper energy band, the lower energy band is reduced for all charging points and all time steps.
- If there are cases where upper power band is not sufficient to meet charged lower energy, the upper power band is increased for all charging points and all time steps.

Parameters

tol (*float*) – Tolerance to reduce or increase values by to fix rounding errors. Default: 1e-6.

resample(*freq: str = '15min'*)

Resamples flexibility bands.

Parameters

freq (str or pandas.Timedelta, optional) – Frequency that time series is resampled to. Offset aliases can be found here: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases. Default: '15min'.

check_integrity()

Method to check the integrity of the Electromobility object.

Raises an error in case any of the checks fails.

Currently only checks integrity of flexibility bands.

to_csv(*directory*, *attributes=None*)

Exports electromobility data to csv files.

The following attributes can be exported:

- 'charging_processes_df' : Attribute *charging_processes_df* is saved to *charging_processes.csv*.
- 'potential_charging_parks_gdf' : Attribute *potential_charging_parks_gdf* is saved to *potential_charging_parks.csv*.
- 'integrated_charging_parks_df' : Attribute *integrated_charging_parks_df* is saved to *integrated_charging_parks.csv*.
- 'simbev_config_df' : Attribute *simbev_config_df* is saved to *simbev_config.csv*.
- 'flexibility_bands' : The three flexibility bands in attribute *flexibility_bands* are saved to *flexibility_band_upper_power.csv*, *flexibility_band_lower_energy.csv*, and *flexibility_band_upper_energy.csv*.

Parameters

- **directory** (*str*) – Path to save electromobility data to.
- **attributes** (*list(str) or None*) – List of attributes to export. See above for attributes that can be exported. If None, all specified attributes are exported. Default: None.

from_csv(*data_path*, *edisgo_obj*, *from_zip_archive=False*)

Restores electromobility from csv files.

Parameters

- **data_path** (*str*) – Path to electromobility csv files.
- **edisgo_obj** (*EDisGo*) –
- **from_zip_archive** (*bool, optional*) – Set True if data is archived in a zip archive. Default: False

edisgo.network.grids

Module Contents

Classes

Grid	Defines a basic grid in eDisGo.
MVGrid	Defines a medium voltage network in eDisGo.
LVGrid	Defines a low voltage network in eDisGo.

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

property id

ID of the grid.

property edisgo_obj

EDisGo object the grid is stored in.

property nominal_voltage

Nominal voltage of network in kV.

Parameters
nominal_voltage (*float*) –

Returns
Nominal voltage of network in kV.

Return type
float

property 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

property geopandas

Returns components as geopandas.GeoDataFrames

Returns container with geopandas.GeoDataFrames containing all georeferenced components within the grid.

Returns
Data container with GeoDataFrames containing all georeferenced components within the grid(s).

Return type
GeoPandasGridContainer or list(*GeoPandasGridContainer*)

property station

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

property station_name

Name of station to the overlying voltage level.

Name of station is composed of grid name with the extension '_station'.

property 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

property generators

Connected generators within the network.

Returns
List of generators within the network.

Return type
list(*Generator*)

property 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

property loads

Connected loads within the network.

Returns
List of loads within the network.

Return type
list(*Load*)

property 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

property charging_points_df

Connected charging points within the network.

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

Return type
pandas.DataFrame

property 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

property switch_disconnectors

Switch disconnectors within the network.

Returns
List of switch disconnectory within the network.

Return type
list(*Switch*)

property 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

abstract property 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

property weather_cells

Weather cells in network.

Returns
List of weather cell IDs in network.

Return type
list(int)

property 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

property 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

property p_set

Cumulative peak load of loads in the network in MW.

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

Return type
float

property p_set_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

assign_length_to_grid_station()

Assign length in km from each bus in the grid to the grid's station.

The length is written to column 'length_to_grid_station' in `buses_df`.

assign_grid_feeder(*mode: str = 'grid_feeder'*)

Assigns MV or LV feeder to each bus and line, depending on the *mode*.

See `assign_feeders` for more information.

Parameters
mode (`str`) – Specifies whether to assign MV or grid feeder. If mode is "mv_feeder" the MV feeder the buses and lines are in are determined. If mode is "grid_feeder" LV buses and lines are assigned the LV feeder they are in and MV buses and lines are assigned the MV feeder they are in. Default: "grid_feeder".

get_feeder_stats() → pandas.DataFrame

Generate statistics of the grid's feeders.

So far, only the feeder length is determined.

Returns
Dataframe with feeder name in index and column 'length' containing the respective feeder length in km.

Return type
pandas.DataFrame

__repr__()

Return repr(self).

class edisgo.network.grids.**MVGrid**(***kwargs*)

Bases: *Grid*

Defines a medium voltage network in eDisGo.

property **lv_grids**

Yields generator object with all underlying low voltage grids.

Returns
Yields generator object with *LVGrid* object.

Return type
LVGrid

property **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

property **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

abstract **draw**()

Draw MV network.

class edisgo.network.grids.**LVGrid**(***kwargs*)

Bases: *Grid*

Defines a low voltage network in eDisGo.

property 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

property 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

abstract property geopandas

Remove this as soon as LVGrids are georeferenced

Type
TODO

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.heat

Module Contents

Classes

HeatPump	Data container for all heat pump data.

class edisgo.network.heat.**HeatPump**(***kwargs*)

Data container for all heat pump data.

This class holds data on heat pump COP, heat demand time series, and thermal storage data.

property cop_df

DataFrame with COP time series of heat pumps.

Parameters

df (pandas.DataFrame) – DataFrame with COP time series of heat pumps in p.u.. Index of the dataframe is a time index and should contain all time steps given in *timeindex*. Column names are names of heat pumps as in *loads_df*.

Returns

DataFrame with COP time series of heat pumps in p.u.. For more information on the dataframe see input parameter *df*.

Return type

pandas.DataFrame

property heat_demand_df

DataFrame with heat demand time series of heat pumps.

Parameters

df (pandas.DataFrame) – DataFrame with heat demand time series of heat pumps in MW. Index of the dataframe is a time index and should contain all time steps given in *timeindex*. Column names are names of heat pumps as in *loads_df*.

Returns

DataFrame with heat demand time series of heat pumps in MW. For more information on the dataframe see input parameter *df*.

Return type

pandas.DataFrame

property thermal_storage_units_df

DataFrame with heat pump's thermal storage information.

Parameters

df (pandas.DataFrame) – DataFrame with thermal storage information. Index of the dataframe are names of heat pumps as in *loads_df*. Columns of the dataframe are:

capacity
[float] Thermal storage capacity in MWh.

efficiency
[float] Charging and discharging efficiency in p.u..

state_of_charge_initial
[float] Initial state of charge in p.u..

Returns
DataFrame with thermal storage information. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

`set_cop`(*edisgo_object*, *ts_cop*, ***kwargs*)

Write COP time series for heat pumps to py:attr:~*cop_df*.

COP time series can either be given to this function or be obtained from the OpenEnergy DataBase. In case they are obtained from the OpenEnergy DataBase the heat pumps need to already be integrated into the grid, i.e. given in `loads_df`.

In case COP time series are set for heat pumps that were already assigned a COP time series, their existing COP time series is overwritten by this function.

Parameters

- **edisgo_object** (`EDisGo`) –
- **ts_cop** (str or pandas.DataFrame) – Defines option used to set COP time series. Possible options are:
  - 'oedb'

    COP / efficiency data are obtained from the OpenEnergy DataBase. In case of heat pumps weather cell specific hourly COP time series are obtained (see `edisgo.io.timeseries_import.cop_oedb()` for more information). Using information on which weather cell each heat pump is in, the weather cell specific time series are mapped to each heat pump. In case of resistive heaters a constant efficiency is set (see `edisgo.io.heat_pump_import.efficiency_resistive_heaters_oedb()`).

    Weather cell information of heat pumps is obtained from column 'weather_cell_id' in `loads_df`. In case no heat pump has weather cell information, this function will throw an error. In case only some heat pumps are missing weather cell information, a random existing weather cell is used to fill missing information.

    This option requires that the parameter *engine* is provided as keyword argument. For further settings, the parameters *timeindex* and *heat_pump_names* can also be provided as keyword arguments.
  - pandas.DataFrame

    DataFrame with self-provided COP time series per heat pump. See `cop_df` on information on the required dataframe format.
- **engine** (sqlalchemy.Engine) – Database engine. This parameter is required in case *ts_cop* is 'oedb'.
- **heat_pump_names** (`list(str) or None`) – Defines for which heat pumps to set COP time series in case *ts_cop* is 'oedb'. If None, all heat pumps in `loads_df` (type is 'heat_pump') are used. Default: None.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to set data in case *ts_cop* is 'oedb'. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year 2011 and returned for the whole year. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using the default year 2011 and returned for the whole year.

set_heat_demand(*edisgo_object*, *ts_heat_demand*, ***kwargs*)

Write heat demand time series of heat pumps to py:attr:~*heat_demand_df*.

Heat demand time series can either be given to this function or be obtained from the OpenEnergy DataBase. In case they are obtained from the OpenEnergy DataBase the heat pumps need to already be integrated into the grid, i.e. given in `loads_df`.

In case heat demand time series are set for heat pumps that were already assigned a heat demand time series, their existing heat demand time series is overwritten by this function.

Parameters

- **edisgo_object** (`EDisGo`) –
- **ts_heat_demand** (str or pandas.DataFrame) – Defines option used to set heat demand time series. Possible options are:
  - ’oedb’

    Heat demand time series are obtained from the OpenEnergy DataBase (see `edisgo.io.timeseries_import.heat_demand_oedb()` for more information). Time series are only obtained for heat pumps that are already integrated into the grid. This option requires that the parameters *engine* and *scenario* are provided as keyword arguments. For further settings, the parameters *timeindex* and *heat_pump_names* can also be provided as keyword arguments.
  - pandas.DataFrame

    DataFrame with self-provided heat demand time series per heat pump. See `heat_demand_df` for information on the required dataframe format.
- **scenario** (`str`) – Scenario for which to retrieve heat demand data. This parameter is required in case *ts_heat_demand* is ‘oedb’. Possible options are ‘eGon2035’ and ‘eGon100RE’.
- **engine** (sqlalchemy.Engine) – Database engine. This parameter is required in case *ts_heat_demand* is ‘oedb’.
- **heat_pump_names** (`list(str) or None`) – Defines for which heat pumps to get heat demand time series for in case *ts_heat_demand* is ‘oedb’. If None, all heat pumps in `loads_df` (type is ‘heat_pump’) are used. Default: None.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to set data in case *ts_heat_demand* is ‘oedb’. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year (2035 in case of the ‘eGon2035’ and to 2045 in case of the ‘eGon100RE’ scenario) and returned for the whole year. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using the default year and returned for the whole year.

reduce_memory(*attr_to_reduce=None*, *to_type='float32'*)

Reduces size of dataframes to save memory.

See `reduce_memory` for more information.

Parameters

- **attr_to_reduce** (`list(str), optional`) – List of attributes to reduce size for. Per default, the following attributes are reduced if they exist: cop_df, heat_demand_df.
- **to_type** (`str, optional`) – Data type to convert time series data to. This is a tradeoff between precision and memory. Default: “float32”.

to_csv(*directory*, *reduce_memory=False*, ***kwargs*)

Exports heat pump data to csv files.

The following attributes are exported:

- 'cop_df'

  Attribute `cop_df` is saved to *cop.csv*.

- 'heat_demand_df'

  Attribute `heat_demand_df` is saved to *heat_demand.csv*.

- 'thermal_storage_units_df'

  Attribute `thermal_storage_units_df` is saved to *thermal_storage_units.csv*.

Parameters

- **directory** (`str`) – Path to save data to.
- **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.
- **kwargs** – Kwargs may contain arguments of `reduce_memory`.

from_csv(*data_path*, *from_zip_archive=False*)

Restores heat pump data from csv files.

Parameters

- **data_path** (`str`) – Path to heat pump csv files.
- **from_zip_archive** (`bool, optional`) – Set True if data is archived in a zip archive. Default: False

resample_timeseries(*method: str = 'ffill'*, *freq: str | pandas.Timedelta = '15min'*)

Resamples COP and heat demand time series to a desired resolution.

Both up- and down-sampling methods are possible.

Parameters

- **method** (`str, optional`) – See `resample_timeseries` for more information.
- **freq** (`str, optional`) – See `resample_timeseries` for more information.

check_integrity()

Check data integrity.

Checks for duplicated and missing labels as well as implausible values.

`edisgo.network.overlying_grid`

Module Contents

Classes

`OverlyingGrid`	Data container for requirements from the overlying grid.

Functions

`distribute_overlying_grid_requirements`(edisgo	Distributes overlying grid requirements to components in grid.

class `edisgo.network.overlying_grid.`**`OverlyingGrid`**(***kwargs*)

Data container for requirements from the overlying grid.

The requirements from the overlying grid are used as constraints for flexibilities.

`renewables_curtailment`

Curtailment of fluctuating renewables per time step in MW.

Type

pandas.Series

`storage_units_active_power`

Aggregated dispatch of storage units per time step in MW.

Type

pandas.Series

`storage_units_soc`

State of charge of storage units per time step in p.u.. The state of charge at time step t here constitutes the state of charge at the beginning of time step t.

Type

pandas.Series

`dsm_active_power`

Aggregated demand side management utilisation per time step in MW.

Type

pandas.Series

`electromobility_active_power`

Aggregated charging demand at all charging sites in grid per time step in MW.

Type

pandas.Series

`heat_pump_decentral_active_power`

Aggregated demand of flexible decentral heat pumps per time step in MW.

Type

pandas.Series

thermal_storage_units_decentral_soc

State of charge of decentral thermal storage units in p.u..

Type

pandas.Series

heat_pump_central_active_power

Aggregated demand of flexible central heat pumps per time step in MW.

Type

pandas.Series

thermal_storage_units_central_soc

State of charge of central thermal storage units per district heating area (in columns as string of integer, i.e. "130" instead of "130.0") and time step (in index) in p.u.. The state of charge at time step t here constitutes the state of charge at the beginning of time step t.

Type

pandas.DataFrame

feedin_district_heating

Other thermal feed-in into district heating per district heating area (in columns as string of integer, i.e. "130" instead of "130.0") and time step (in index) in MW.

Type

pandas.DataFrame

reduce_memory(*attr_to_reduce=None*, *to_type='float32'*)

Reduces size of time series data to save memory.

Parameters

- **attr_to_reduce** (`list(str), optional`) – List of attributes to reduce size for. Per default, all time series data 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 data in object to csv.

Parameters

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

from_csv(*data_path*, *dtype=None*, *from_zip_archive=False*, ***kwargs*)

Restores data in object from csv files.

Parameters

- **data_path** (`str`) – Path to directory to obtain data from. Must be a directory or zip archive.
- **dtype** (`str, optional`) – Numerical data type for data to be loaded from csv. E.g. "float32". Default: None.

- **from_zip_archive** (*bool, optional*) – Set True if data is archived in a zip archive. Default: False.

resample(*method: str = 'ffill', freq: str | pandas.Timedelta = '15min'*)

Resamples all time series to a desired resolution.

See *resample_timeseries* for more information.

Parameters

- **method** (*str, optional*) – See *resample_timeseries* for more information.
- **freq** (*str, optional*) – See *resample_timeseries* for more information.

edisgo.network.overlying_grid.**distribute_overlying_grid_requirements**(*edisgo_obj*)

Distributes overlying grid requirements to components in grid.

Overlying grid requirements for e.g. electromobility charging are distributed to all charging points where cars are parked, and for DSM to all DSM loads based on their available load increase and decrease at each time step.

Parameters
edisgo_obj (*EDisGo*) – The eDisGo API object

Returns
New EDisGo object with only the topology data and adjusted time series data.

Return type
EDisGo

edisgo.network.results

Module Contents

Classes

Results	Power flow analysis results management

class edisgo.network.results.**Results**(*edisgo_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

property 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

property 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

property 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

property 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

property 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

property 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

property 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

`property 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().

property 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.

property 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

property 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

property 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

property 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_overload()` for format of remaining overloading issues of lines, `hv_mv_station_overload()` for format of remaining overloading issues of transformers, and *voltage_issues()* 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 tradeoff 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.

equality_check(*results_obj*)

Checks the equality of two results objects.

Parameters
results_obj (*:class:~.network.results.Results*) – Contains the results of analyze function with default settings.

Returns
True if equality check is successful, False otherwise.

Return type
bool

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

from_csv(*data_path*, *parameters=None*, *dtype=None*, *from_zip_archive=False*)

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

- **data_path** (*str*) – Main data path results are saved in. Must be directory or zip archive.
- **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.

- **dtype** (*str, optional*) – Numerical data type for data to be loaded from csv, e.g. "float32". Per default this is None in which case data type is inferred.
- **from_zip_archive** (*bool, optional*) – Set True if data is archived in a zip archive. Default: False.

`edisgo.network.timeseries`

Module Contents

Classes

TimeSeries	Holds component-specific active and reactive power time series.
TimeSeriesRaw	Holds raw time series data, e.g. sector-specific demand and standing times of EV.

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

Holds component-specific active and reactive power time series.

All time series are fixed time series that in case of flexibilities result after application of a heuristic or optimisation. They can be used for power flow calculations.

Also holds any raw time series data that was used to generate component-specific time series in attribute *time_series_raw*. See *TimeSeriesRaw* for more information.

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.

time_series_raw

Raw time series. See *TimeSeriesRaw* for more information.

Type

TimeSeriesRaw

property is_worst_case: bool

Time series mode.

Is used to distinguish between normal time series analysis and worst-case analysis. Is determined by checking if the timindex starts before 1971 as the default for worst-case is 1970. Be mindful when creating your own worst-cases.

Returns

Indicates if current time series is worst-case time series with different assumptions for mv and lv simultaneities.

Return type

bool

property timeindex

Time index all time-dependent attributes are indexed by.

Is used as default time steps in e.g. power flow analysis.

Parameters

ind (pandas.DatetimeIndex) – Time index all time-dependent attributes are indexed by.

Returns
Time index all time-dependent attributes are indexed by.

Return type
pandas.DatetimeIndex

property generators_active_power

Active power time series of generators in MW.

Parameters
df (pandas.DataFrame) – Active power time series of all generators in topology in MW. Index of the dataframe is a time index and column names are names of generators.

Returns
Active power time series of all generators in topology in MW for time steps given in `timeindex`. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property generators_reactive_power

Reactive power time series of generators in MVA.

Parameters
df (pandas.DataFrame) – Reactive power time series of all generators in topology in MVA. Index of the dataframe is a time index and column names are names of generators.

Returns
Reactive power time series of all generators in topology in MVA for time steps given in `timeindex`. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property loads_active_power

Active power time series of loads in MW.

Parameters
df (pandas.DataFrame) – Active power time series of all loads in topology in MW. Index of the dataframe is a time index and column names are names of loads.

Returns
Active power time series of all loads in topology in MW for time steps given in `timeindex`. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property loads_reactive_power

Reactive power time series of loads in MVA.

Parameters
df (pandas.DataFrame) – Reactive power time series of all loads in topology in MVA. Index of the dataframe is a time index and column names are names of loads.

Returns
Reactive power time series of all loads in topology in MVA for time steps given in `timeindex`. For more information on the dataframe see input parameter *df*.

Return type
pandas.DataFrame

property storage_units_active_power

Active power time series of storage units in MW.

Parameters

`df` (pandas.DataFrame) – Active power time series of all storage units in topology in MW. Index of the dataframe is a time index and column names are names of storage units.

Returns

Active power time series of all storage units in topology in MW for time steps given in `timeindex`. For more information on the dataframe see input parameter *df*.

Return type

pandas.DataFrame

property storage_units_reactive_power

Reactive power time series of storage units in MVA.

Parameters

`df` (pandas.DataFrame) – Reactive power time series of all storage units in topology in MVA. Index of the dataframe is a time index and column names are names of storage units.

Returns

Reactive power time series of all storage units in topology in MVA for time steps given in `timeindex`. For more information on the dataframe see input parameter *df*.

Return type

pandas.DataFrame

property residual_load

Returns residual load in network.

Residual load for each time step is calculated from total load 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

property 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 ('feed-in_case') for each time step in `timeindex`.

Return type

pandas.Series

charging_points_active_power(*edisgo_object:* edisgo.EDisGo)

Returns a subset of *loads_active_power* containing only the time series of charging points.

Parameters

edisgo_object (*EDisGo*) –

Returns

Pandas DataFrame with active power time series of charging points.

Return type

pandas.DataFrame

charging_points_reactive_power(*edisgo_object:* edisgo.EDisGo)

Returns a subset of *loads_reactive_power* containing only the time series of charging points.

Parameters

edisgo_object (*EDisGo*) –

Returns

Pandas DataFrame with reactive power time series of charging points.

Return type

pandas.DataFrame

reset()

Resets all time series.

Active and reactive power time series of all loads, generators and storage units are deleted, as well as timeindex and everything stored in *time_series_raw*.

set_active_power_manual(*edisgo_object*, *ts_generators=None*, *ts_loads=None*, *ts_storage_units=None*)

Sets given component active power time series.

If time series for a component were already set before, they are overwritten.

Parameters

- **edisgo_object** (*EDisGo*) –
- **ts_generators** (pandas.DataFrame) – Active power time series in MW of generators. Index of the data frame is a datetime index. Columns contain generators names of generators to set time series for.
- **ts_loads** (pandas.DataFrame) – Active power time series in MW of loads. Index of the data frame is a datetime index. Columns contain load names of loads to set time series for.
- **ts_storage_units** (pandas.DataFrame) – Active power time series in MW of storage units. Index of the data frame is a datetime index. Columns contain storage unit names of storage units to set time series for.

set_reactive_power_manual(*edisgo_object*, *ts_generators=None*, *ts_loads=None*, *ts_storage_units=None*)

Sets given component reactive power time series.

If time series for a component were already set before, they are overwritten.

Parameters

- **edisgo_object** (*EDisGo*) –

- **ts_generators** (pandas.DataFrame) – Reactive power time series in MVA of generators. Index of the data frame is a datetime index. Columns contain generators names of generators to set time series for.
- **ts_loads** (pandas.DataFrame) – Reactive power time series in MVA of loads. Index of the data frame is a datetime index. Columns contain load names of loads to set time series for.
- **ts_storage_units** (pandas.DataFrame) – Reactive power time series in MVA of storage units. Index of the data frame is a datetime index. Columns contain storage unit names of storage units to set time series for.

`set_worst_case`(*edisgo_object*, *cases*, *generators_names=None*, *loads_names=None*, *storage_units_names=None*)

Sets demand and feed-in of loads, generators and storage units for the specified worst cases.

Per default time series are set for all loads, generators and storage units in the network.

Possible worst cases are 'load_case' (heavy load flow case) and 'feed-in_case' (reverse power flow case). Each case is set up once for dimensioning of the MV grid ('load_case_mv'/'feed-in_case_mv') and once for the dimensioning of the LV grid ('load_case_lv'/'feed-in_case_lv'), as different simultaneity factors are assumed for the different voltage levels.

Assumed simultaneity factors specified in the config section *worst_case_scale_factor* are used to generate active power demand or feed-in. For the reactive power behavior fixed cosphi is assumed. The power factors set in the config section *reactive_power_factor* and the power factor mode, defining whether components behave inductive or capacitive, given in the config section *reactive_power_mode*, are used.

Component specific information is given below:

- Generators

  Worst case feed-in time series are distinguished by technology (PV, wind and all other) and whether it is a load or feed-in case. In case of generator worst case time series it is not distinguished by whether it is used to analyse the MV or LV. However, both options are generated as it is distinguished in the case of loads. Worst case scaling factors for generators are specified in the config section *worst_case_scale_factor* through the parameters: 'feed-in_case_feed-in_pv', 'feed-in_case_feed-in_wind', 'feed-in_case_feed-in_other', 'load_case_feed-in_pv', load_case_feed-in_wind', and 'load_case_feed-in_other'.

  For reactive power a fixed cosphi is assumed. A different reactive power factor is used for generators in the MV and generators in the LV. The reactive power factors for generators are specified in the config section *reactive_power_factor* through the parameters: 'mv_gen' and 'lv_gen'.

- Conventional loads

  Worst case load time series are distinguished by whether it is a load or feed-in case and whether it used to analyse the MV or LV. Worst case scaling factors for conventional loads are specified in the config section *worst_case_scale_factor* through the parameters: 'mv_feed-in_case_load', 'lv_feed-in_case_load', 'mv_load_case_load', and 'lv_load_case_load'.

  For reactive power a fixed cosphi is assumed. A different reactive power factor is used for loads in the MV and loads in the LV. The reactive power factors for conventional loads are specified in the config section *reactive_power_factor* through the parameters: 'mv_load' and 'lv_load'.

- Charging points

Worst case demand time series are distinguished by use case (home charging, work charging, public (slow) charging and HPC), by whether it is a load or feed-in case and by whether it used to analyse the MV or LV. Worst case scaling factors for charging points are specified in the config section *worst_case_scale_factor* through the parameters: 'mv_feed-in_case_cp_home', 'mv_feed-in_case_cp_work', 'mv_feed-in_case_cp_public', and 'mv_feed-in_case_cp_hpc', 'lv_feed-in_case_cp_home', 'lv_feed-in_case_cp_work', 'lv_feed-in_case_cp_public', and 'lv_feed-in_case_cp_hpc', 'mv_load-in_case_cp_home', 'mv_load-in_case_cp_work', 'mv_load-in_case_cp_public', and 'mv_load-in_case_cp_hpc', 'lv_load-in_case_cp_home', 'lv_load-in_case_cp_work', 'lv_load-in_case_cp_public', and 'lv_load-in_case_cp_hpc'.

For reactive power a fixed cosphi is assumed. A different reactive power factor is used for charging points in the MV and charging points in the LV. The reactive power factors for charging points are specified in the config section *reactive_power_factor* through the parameters: 'mv_cp' and 'lv_cp'.

- Heat pumps

  Worst case demand time series are distinguished by whether it is a load or feed-in case and by whether it used to analyse the MV or LV. Worst case scaling factors for heat pumps are specified in the config section *worst_case_scale_factor* through the parameters: 'mv_feed-in_case_hp', 'lv_feed-in_case_hp', 'mv_load_case_hp', and 'lv_load_case_hp'.

  For reactive power a fixed cosphi is assumed. A different reactive power factor is used for heat pumps in the MV and heat pumps in the LV. The reactive power factors for heat pumps are specified in the config section *reactive_power_factor* through the parameters: 'mv_hp' and 'lv_hp'.

- Storage units

  Worst case feed-in time series are distinguished by whether it is a load or feed-in case. In case of storage units worst case time series it is not distinguished by whether it is used to analyse the MV or LV. However, both options are generated as it is distinguished in the case of loads. Worst case scaling factors for storage units are specified in the config section *worst_case_scale_factor* through the parameters: 'feed-in_case_storage' and 'load_case_storage'.

  For reactive power a fixed cosphi is assumed. A different reactive power factor is used for storage units in the MV and storage units in the LV. The reactive power factors for storage units are specified in the config section *reactive_power_factor* through the parameters: 'mv_storage' and 'lv_storage'.

Parameters

- **edisgo_object** (*EDisGo*) –
- **cases** (*list(str)*) – List with worst-cases to generate time series for. Can be 'feed-in_case', 'load_case' or both.
- **generators_names** (*list(str)*) – Defines for which generators to set worst case time series. If None, time series are set for all generators. Default: None.
- **loads_names** (*list(str)*) – Defines for which loads to set worst case time series. If None, time series are set for all loads. Default: None.
- **storage_units_names** (*list(str)*) – Defines for which storage units to set worst case time series. If None, time series are set for all storage units. Default: None.

Notes

Be careful, this function overwrites all previously set time series in the case that these are not worst case time series. If previously set time series are worst case time series is checked using `is_worst_case`.

Further, if this function is called for a component whose worst-case time series are already set, they are overwritten, even if previously set time series were set for a different worst-case.

Also be aware that loads for which type information is not set are handled as conventional loads.

`predefined_fluctuating_generators_by_technology`(*edisgo_object*, *ts_generators*, *generator_names=None*, *timeindex=None*, *engine=None*)

Set active power feed-in time series for fluctuating generators by technology.

In case time series are provided per technology and weather cell ID, active power feed-in time series are also set by technology and weather cell ID.

Parameters

- **`edisgo_object`** (`EDisGo`) –
- **`ts_generators`** (str or pandas.DataFrame) – Defines which technology-specific or technology and weather cell specific active power time series to use. Possible options are:
  - 'oedb'

    Technology and weather cell specific hourly feed-in time series are obtained from the OpenEnergy DataBase. See `edisgo.io.timeseries_import.feedin_oedb()` for more information.

    This option requires that the parameter *engine* is provided in case new ding0 grids with geo-referenced LV grids are used. For further settings, the parameter *timeindex* can also be provided.
  - pandas.DataFrame

    DataFrame with self-provided feed-in time series per technology or per technology and weather cell ID normalized to a nominal capacity of 1. In case time series are provided only by technology, columns of the DataFrame contain the technology type as string. In case time series are provided by technology and weather cell ID columns need to be a pandas.MultiIndex with the first level containing the technology as string and the second level the weather cell ID as integer. Index needs to be a pandas.DatetimeIndex.

    When importing a ding0 grid and/or using predefined scenarios of the future generator park, 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.
- **`generator_names`** (`list(str)`) – Defines for which fluctuating generators to use technology-specific time series. If None, all generators for which technology- (and weather cell-) specific time series are provided are used. In case the time series are retrieved from the oedb, all solar and wind generators are used. Default: None.
- **`timeindex`** (pandas.DatetimeIndex or None) – Specifies time steps for which to set feed-in time series. This parameter is only used in case *ts_generators* is 'oedb'. See pa-

rameter *timeindex* in `edisgo.io.timeseries_import.feedin_oedb()` for more information.

- **engine** (sqlalchemy.Engine) – Database engine. This parameter is only required in case *ts_generators* is 'oedb' and new ding0 grids with geo-referenced LV grids are used.

predefined_dispatchable_generators_by_technology(*edisgo_object*, *ts_generators*, *generator_names=None*)

Set active power feed-in time series for dispatchable generators by technology.

Parameters

- **edisgo_object** (*EDisGo*) –
- **ts_generators** (pandas.DataFrame) – DataFrame with self-provided active power time series of each type of dispatchable generator normalized to a nominal capacity of 1. Columns contain the technology type as string, e.g. 'gas', 'coal'. Use 'other' if you don't want to explicitly provide a time series for every possible technology. In the current grid existing generator technologies can be retrieved from column *type* in `generators_df`. Index needs to be a pandas.DatetimeIndex.
- **generator_names** (*list(str)*) – Defines for which dispatchable generators to use technology-specific time series. If None, all dispatchable generators technology-specific time series are provided for are used. In case *ts_generators* contains a column 'other', all dispatchable generators in the network (i.e. all but solar and wind generators) are used.

predefined_conventional_loads_by_sector(*edisgo_object*, *ts_loads*, *load_names=None*, *timeindex=None*)

Set active power demand time series for conventional loads by sector.

Parameters

- **edisgo_object** (*EDisGo*) –
- **ts_loads** (str or pandas.DataFrame) – Defines which sector-specific active power time series to use. Possible options are:
  - 'demandlib'

    See parameter *conventional_loads_ts* in `set_time_series_active_power_predefined()` for more information.
  - pandas.DataFrame

    See parameter *conventional_loads_ts* in `set_time_series_active_power_predefined()` for more information.
- **load_names** (*list(str)*) – Defines for which conventional loads to use sector-specific time series. If None, all loads of sectors for which sector-specific time series are provided are used. In case the demandlib is used, all loads of sectors 'residential', 'cts', 'industrial', and 'agricultural' are used.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to set time series. This parameter is only used in case *ts_loads* is 'demandlib'. See parameter *timeindex* in `edisgo.io.timeseries_import.load_time_series_demandlib()` for more information.

predefined_charging_points_by_use_case(*edisgo_object*, *ts_loads*, *load_names=None*)

Set active power demand time series for charging points by their use case.

Parameters

- **edisgo_object** (*EDisGo*) –
- **ts_loads** (pandas.DataFrame) – DataFrame with self-provided load time series per use case normalized to a nominal power of the charging point of 1. Index needs to be a pandas.DatetimeIndex. Columns contain the use case as string. In the current grid existing use case types can be retrieved from column *sector* in `loads_df` (make sure to select *type* 'charging_point'). When using charging point input from SimBEV the differentiated use cases are 'home', 'work', 'public' and 'hpc'.
- **load_names** (*list(str)*) – Defines for which charging points to use use-case-specific time series. If None, all charging points of use cases for which use-case-specific time series are provided are used.

fixed_cosphi(*edisgo_object*, *generators_parametrisation=None*, *loads_parametrisation=None*, *storage_units_parametrisation=None*)

Sets reactive power of specified components assuming a fixed power factor.

Overwrites reactive power time series in case they already exist.

Parameters

- **generators_parametrisation** (str or pandas.DataFrame or None) – Sets fixed cosphi parameters for generators. Possible options are:
  - 'default'

    Default configuration is used for all generators in the grid. To this end, the power factors set in the config section *reactive_power_factor* and the power factor mode, defining whether components behave inductive or capacitive, given in the config section *reactive_power_mode*, are used.
  - pandas.DataFrame

    DataFrame with fix cosphi parametrisation for specified generators. Columns are:

    * **'components'**
      [list(str)] List with generators to apply parametrisation for.
    * **'mode'**
      [str] Defines whether generators behave inductive or capacitive. Possible options are 'inductive', 'capacitive' or 'default'. In case of 'default', configuration from config section *reactive_power_mode* is used.
    * **'power_factor'**
      [float or str] Defines the fixed cosphi power factor. The power factor can either be directly provided as float or it can be set to 'default', in which case configuration from config section *reactive_power_factor* is used.

    Index of the dataframe is ignored.
  - None

    No reactive power time series are set.

  Default: None.
- **loads_parametrisation** (str or pandas.DataFrame or None) – Sets fixed cosphi parameters for loads. The same options as for parameter *generators_parametrisation* apply.

- **storage_units_parametrisation** (str or pandas.DataFrame or None) – Sets fixed cosphi parameters for storage units. The same options as for parameter *generators_parametrisation* apply.

Notes

This function requires active power time series to be previously set.

reduce_memory(*attr_to_reduce=None*, *to_type='float32'*, *time_series_raw=True*, ***kwargs*)

Reduces size of dataframes to save memory.

See `reduce_memory` for more information.

Parameters

- **attr_to_reduce** (`list(str), optional`) – List of attributes to reduce size for. Per default, all active and reactive power time series of generators, loads, and storage units are reduced.
- **to_type** (`str, optional`) – Data type to convert time series data to. This is a tradeoff between precision and memory. Default: "float32".
- **time_series_raw** (`bool, optional`) – If True raw time series data in `time_series_raw` is reduced as well. Default: True.
- **attr_to_reduce_raw** (`list(str), optional`) – List of attributes in `TimeSeriesRaw` to reduce size for. See `reduce_memory` for default.

to_csv(*directory*, *reduce_memory=False*, *time_series_raw=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
- storage_units_active_power and storage_units_reactive_power

If parameter *time_series_raw* is set to True, raw time series data is saved to csv as well. See `to_csv` for more information.

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.
- **time_series_raw** (`bool, optional`) – If True raw time series data in `time_series_raw` is saved to csv as well. Per default all raw time series data is then stored in a subdirectory of the specified *directory* called "time_series_raw". Further, if *reduce_memory* is set to True, raw time series data is reduced as well. To change this default behavior please call `to_csv` separately. Default: False.
- **kwargs** – Kwargs may contain arguments of `reduce_memory`.

from_csv(*data_path*, *dtype=None*, *time_series_raw=False*, *from_zip_archive=False*, ***kwargs*)

Restores time series from csv files.

See `to_csv()` for more information on which time series can be saved and thus restored.

Parameters

- **data_path** (*str*) – Data path time series are saved in. Must be a directory or zip archive.
- **dtype** (*str, optional*) – Numerical data type for data to be loaded from csv. E.g. "float32". Default: None.
- **time_series_raw** (*bool, optional*) – If True raw time series data is as well read in (see `from_csv` for further information). Directory data is restored from can be specified through kwargs. Default: False.
- **from_zip_archive** (*bool, optional*) – Set True if data is archived in a zip archive. Default: False.
- **directory_raw** (*str, optional*) – Directory to read raw time series data from. Per default this is a subdirectory of the specified *directory* called "time_series_raw".

check_integrity()

Check for NaN, duplicated indices or columns and if time series is empty.

drop_component_time_series(*df_name*, *comp_names*)

Drops component time series if they exist.

Parameters

- **df_name** (*str*) – Name of attribute of given object holding the dataframe to remove columns from. Can e.g. be "generators_active_power" if time series should be removed from `generators_active_power`.
- **comp_names** (*str or list(str)*) – Names of components to drop.

add_component_time_series(*df_name*, *ts_new*)

Add component time series by concatenating existing and provided dataframe.

Possibly already component time series are dropped before appending newly provided time series using `drop_component_time_series`.

Parameters

- **df_name** (*str*) – Name of attribute of given object holding the dataframe to add columns to. Can e.g. be "generators_active_power" if time series should be added to `generators_active_power`.
- **ts_new** (pandas.DataFrame) – Dataframe with new time series to add to existing time series dataframe.

resample(*method: str = 'ffill'*, *freq: str | pandas.Timedelta = '15min'*)

Resamples all generator, load and storage time series to a desired resolution.

See `resample_timeseries` for more information.

Parameters

- **method** (*str, optional*) – See `resample_timeseries` for more information.
- **freq** (*str, optional*) – See `resample_timeseries` for more information.

scale_timeseries(*p_scaling_factor: float = 1.0*, *q_scaling_factor: float = 1.0*)

Scales component time series by given factors.

The changes are directly applied to the TimeSeries object.

Parameters

- **p_scaling_factor** (*float*) – Scaling factor to use for active power time series. Values between 0 and 1 will scale down the time series and values above 1 will scale the timeseries up. Default: 1.

- **q_scaling_factor** (*float*) – Scaling factor to use for reactive power time series. Values between 0 and 1 will scale down the time series and values above 1 will scale the timeseries up. Default: 1.

class edisgo.network.timeseries.**TimeSeriesRaw**

Holds raw time series data, e.g. sector-specific demand and standing times of EV.

Normalised time series are e.g. sector-specific demand time series or technology-specific feed-in time series. Time series needed for flexibilities are e.g. heat time series or curtailment time series.

q_control

Dataframe with information on applied reactive power control or in case of conventional loads assumed reactive power behavior. Index of the dataframe are the component names as in index of *generators_df*, *loads_df*, and *storage_units_df*. Columns are "type" with the type of Q-control applied (can be "fixed_cosphi", "cosphi(P)", or "Q(V)"), "power_factor" with the (maximum) power factor, "q_sign" giving the sign of the reactive power (only applicable to "fixed_cosphi"), "parametrisation" with the parametrisation of the respective Q-control (only applicable to "cosphi(P)" and "Q(V)").

Type

pandas.DataFrame

fluctuating_generators_active_power_by_technology

DataFrame with feed-in time series per technology or technology and weather cell ID normalized to a nominal capacity of 1. Columns can either just contain the technology type as string or be a pandas.MultiIndex with the first level containing the technology as string and the second level the weather cell ID as integer. Index is a pandas.DatetimeIndex.

Type

pandas.DataFrame

dispatchable_generators_active_power_by_technology

DataFrame with feed-in time series per technology normalized to a nominal capacity of 1. Columns contain the technology type as string. Index is a pandas.DatetimeIndex.

Type

pandas.DataFrame

conventional_loads_active_power_by_sector

DataFrame with load time series of each type of conventional load normalized to an annual consumption of 1. Index needs to be a pandas.DatetimeIndex. Columns represent load type. In ding0 grids the differentiated sectors are 'residential', 'cts', 'industrial', and 'agricultural'.

Type

pandas.DataFrame

charging_points_active_power_by_use_case

DataFrame with charging demand time series per use case normalized to a nominal capacity of 1. Columns contain the use case as string. Index is a pandas.DatetimeIndex.

Type

pandas.DataFrame

reduce_memory(*attr_to_reduce=None*, *to_type='float32'*)

Reduces size of dataframes 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. Per default the following attributes are reduced if they exist: q_control, fluctuating_generators_active_power_by_technology, dispatchable_generators_active_power_by_technology, conventional_loads_active_power_by_sector, charging_points_active_power_by_use_case.

- **to_type** (*str, optional*) – Data type to convert time series data to. This is a tradeoff between precision and memory. Default: "float32".

to_csv(*directory*, *reduce_memory=False*, ***kwargs*)

Saves time series to csv.

Saves all attributes that are set to csv files with the same file name. See class definition for possible attributes.

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.
- **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.topology`

Module Contents

Classes

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

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

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

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

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.

property loads_df

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

Parameters

df (pandas.DataFrame) – Dataframe with all loads (incl. charging points, heat pumps, etc.) 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.

p_set
[float] Peak load or nominal capacity in MW.

type
[str] Type of load, e.g. 'conventional_load', 'charging_point' or 'heat_pump' (resistive heaters are as well treated as heat pumps with a COP smaller than 1). This information is for example currently necessary when setting up a worst case analysis, as different types of loads are treated differently.

sector
[str] Further specifies type of load.

In case of conventional loads this attribute is used if demandlib is used to generate sector-specific time series (see function `predefined_conventional_loads_by_sector`). It is further used when new generators are integrated into the grid in case the LV is not geo-referenced, as e.g. smaller PV rooftop generators are most likely to be located in a household (see function `connect_to_lv`). The sector needs to either be 'industrial', 'residential' or 'cts'.

In case of charging points this attribute is used to define the charging point use case ('home', 'work', 'public' or 'hpc') to determine whether a charging process can be flexibilised, as it is assumed that only charging processes at private charging points ('home' and 'work') can be flexibilised (see function `charging_strategy`). It is further used when charging points are integrated into the grid, as e.g. 'home' charging points are allocated to a household (see function `connect_to_lv`).

In case of heat pumps it is used when heat pumps are integrated into the grid, as e.g. heat pumps for individual heating are allocated to an existing load (see function `connect_to_lv`). It is further used to specify, if component is a resistive heater, as resistive heaters are treated as heat pumps. The sector needs to either be 'individual_heating', 'district_heating', 'individual_heating_resistive_heater' or 'district_heating_resistive_heater'.

building_id
[int] ID of the building the load is associated with. This is e.g. used to get electricity and heat demand time series as well as information on existing heat pumps and PV rooftop plants for scenarios developed in the eGo^n research project.

annual_consumption
[float] Annual consumption in MWh.

number_households
[int] Number of households in the building. This information is currently not used in eDisGo.

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

`property 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

`property 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.

max_hours
[float] Maximum state of charge capacity in terms of hours at full output capacity p_nom.

efficiency_store
[float] Efficiency of storage system in case of charging. So far only used in `()`

efficiency_dispatch
[float] Efficiency of storage system in case of discharging. So far only used in `()`

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

property 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

property 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

property 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

`property 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

`property 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

`property charging_points_df`

Returns a subset of *`loads_df`* containing only charging points.

Parameters

`type` (*`str`*) – Load type. Default: "charging_point"

Returns

Pandas DataFrame with all loads of the given type.

Return type

pandas.DataFrame

`property id`

MV network ID.

Returns

MV network ID.

Return type

int

`property grids`

Gives a list with *`MVGrid`* object and all *`LVGrid`* objects.

`property 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

property lv_grids

Yields generator object with all low voltage grids in network.

Returns
Yields generator object with *LVGrid* object.

Return type
LVGrid

property 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

property 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)

property 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_lv_grid(*name*)

Returns *LVGrid* object for given LV grid ID or name.

Parameters
name (*int or str*) – LV grid ID as integer or LV grid name (string representation) as string of the LV grid object that should be returned.

Returns
LV grid object with the given LV grid ID or LV grid name (string representation).

Return type
LVGrid

`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

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

`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*, *p_set*, *type='conventional_load'*, ***kwargs*)

Adds load to topology.

Load name is generated automatically.

Parameters

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

- **type** (*str*) – See `loads_df` for more information. Default: "conventional_load"
- **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'.
- **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_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'.
- **kwargs** – Kwargs may contain any further attributes you want to specify, e.g. *max_hours*.

add_line(*bus0*, *bus1*, *length*, ***kwargs*)

Adds line to topology.

Line name is generated automatically. If *type_info* is provided, *x*, *r*, *b* 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.
- **kwargs** – Kwargs may contain any further attributes in *lines_df*. It is necessary to either provide *type_info* to determine *x*, *r*, *b* and *s_nom* of the line, or to provide *x*, *r*, *b* 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.
- **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.

If no other elements are connected, line and bus are removed as well.

Parameters

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

remove_generator(*name*)

Removes generator with given name from topology.

If no other elements are connected, line and bus are removed as well.

Parameters

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

remove_storage_unit(*name*)

Removes storage with given name from topology.

If no other elements are connected, line and bus are removed as well.

Parameters

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

remove_line(*name*)

Removes line with given name from topology.

Line is only removed, if it does not result in isolated buses. A warning is raised in that case.

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, b, 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.

sort_buses()

Sorts buses in *lines_df* such that bus0 is always the upstream bus.

The changes are directly written to *lines_df* dataframe.

connect_to_mv(*edisgo_object*, *comp_data*, *comp_type='generator'*)

Add and connect new component.

Currently, components can be generators, charging points, heat pumps and storage units.

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*, *add_storage_unit* respectively *add_load*, 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 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', 'charging_point', 'heat_pump' or 'storage_unit'. 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 component to LV grid topology.

This function is used in case the LV grids are not geo-referenced. In case LV grids are geo-referenced function `connect_to_lv_based_on_geolocation` is used.

Currently, components can be generators, charging points, heat pumps and storage units.

This function connects the new component depending on the voltage level, and information on the MV/LV substation ID, geometry and sector, 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 and storage units with specified voltage level 7**
  - with a nominal capacity of <=30 kW to LV loads of sector residential, if available
  - with a nominal capacity of >30 kW to LV loads of sector cts, 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 sector 'home' to LV loads of sector residential, if available
  - with sector 'work' to LV loads of sector cts, industrial or agricultural, if available, otherwise
  - with sector 'public' or 'hpc' 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)
- **Heat pumps with specified voltage level 7**
  - with sector 'individual_heating' or 'individual_heating_resistive_heater' to LV loads
  - with sector 'district_heating' or 'district_heating_resistive_heater' to some bus in the grid that is not a house connection
  - to random bus in the LV grid that 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 components of the same type 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_load`, 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 key 'voltage_level' and may contain the geolocation in key 'geom' and the LV grid ID to connect the component 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', 'charging_point' or 'heat_pump'. Default: 'generator'.
- **allowed_number_of_comp_per_bus** (*int*) – Specifies, how many components of the same type 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.

connect_to_lv_based_on_geolocation(*edisgo_object*, *comp_data*, *comp_type*, *max_distance_from_target_bus=0.02*)

Add and connect new component to LV grid topology based on its geolocation.

This function is used in case the LV grids are geo-referenced. In case LV grids are not geo-referenced function `connect_to_lv` is used.

Currently, components can be generators, charging points, heat pumps and storage units.

In case the component is integrated in voltage level 6 it is connected to the closest MV/LV substation; in case it is integrated in voltage level 7 it is connected to the closest LV bus. In contrast to the connection of components to the MV level splitting of a line to connect a new component is not conducted.

A new bus for the new component is only created in case the closest existing bus is farther away than what is specified through parameter *max_distance_from_target_bus*. Otherwise, the new component is directly connected to the nearest bus.

Parameters

- **edisgo_object** (*EDisGo*) –
- **comp_data** (*dict*) – Dictionary with all information on component. The dictionary must contain all required arguments of method `add_generator`, `add_storage_unit` respectively `add_load`, 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 to in key 'voltage_level' and the geolocation in key 'geom'. 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.

- **comp_type** (*str*) – Type of new component. Can be 'generator', 'charging_point', 'heat_pump' or 'storage_unit'.
- **max_distance_from_target_bus** (*int*) – Specifies the maximum distance of the component to the target bus in km before a new bus is created. If the new component is closer to the target bus than the maximum specified distance, it is directly connected to that target bus. Default: 0.1.

Returns
The identifier of the newly connected component as in index of `generators_df`, `loads_df` or `storage_units_df`, depending on component type.

Return type
str

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_geopandas(*mode: str = 'mv'*)

Returns components as geopandas.GeoDataFrames.

Returns container with geopandas.GeoDataFrames containing all georeferenced components within the grid.

Parameters
mode (*str*) – Return mode. If mode is "mv" the mv components are returned. If mode is "lv" a generator with a container per lv grid is returned. Default: "mv"

Returns
Data container with GeoDataFrames containing all georeferenced components within the grid(s).

Return type
`GeoPandasGridContainer` or list(`GeoPandasGridContainer`)

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*.
- '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(*data_path*, *edisgo_obj*, *from_zip_archive=False*)

Restores topology from csv files.

Parameters

- **data_path** (*str*) – Path to topology csv files or zip archive.
- **edisgo_obj** (*EDisGo*) –
- **from_zip_archive** (*bool*) – Set to True if data is archived in a zip archive. Default: False.

check_integrity()

Check data integrity.

Checks for duplicated labels and isolated components. Further checks for very small impedances that can cause stability problems in the power flow calculation and large line lengths that might be implausible.

assign_feeders(*mode: str = 'grid_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 *grid_feeder*, depending on the *mode*, in *Topology*'s `buses_df` and `lines_df`.

The MV feeder name corresponds to the name of the neighboring node of the HV/MV station. The grid feeder name corresponds to the name of the neighboring node of the respective grid's station. The feeder name of the source node, i.e. the station, is set to "station_node".

Parameters

mode (*str*) – Specifies whether to assign MV or grid feeder. If mode is "mv_feeder" the MV feeder the buses and lines are in are determined. If mode is "grid_feeder" LV buses and lines are assigned the LV feeder they are in and MV buses and lines are assigned the MV feeder they are in. Default: "grid_feeder".

aggregate_lv_grid_at_station(*lv_grid_id: int | str*) → None

Aggregates all LV grid components to secondary side of the grid's station.

All lines of the LV grid are dropped, as well as all buses except the station's secondary side bus. Buses, the loads, generators and storage units are connected to are changed to the station's secondary side bus. The changes are directly applied to the Topology object.

Parameters

lv_grid_id (*int or str*) – ID of the LV grid to aggregate.

__repr__()

Return repr(self).

edisgo.opf

Subpackages

edisgo.opf.results

Submodules

edisgo.opf.results.opf_result_class

Module Contents

Classes

LineVariables	
HeatStorage	
BatteryStorage	
GridSlacks	
OPFResults	

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

class edisgo.opf.results.opf_result_class.**HeatStorage**

class edisgo.opf.results.opf_result_class.**BatteryStorage**

class edisgo.opf.results.opf_result_class.**GridSlacks**

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

to_csv(*directory*, *attributes=None*)

Exports OPF results data to csv files.

The following attributes can be exported:

- 'lines_t' : The results of the three variables in attribute `lines_t` are saved to *lines_t_p.csv*, *lines_t_p.csv*, and *lines_t_ccm.csv*.
- 'slack_generator_t' : Attribute `slack_generator_t` is saved to *slack_generator_t.csv*.
- 'heat_storage_t' : The results of the two variables in attribute `heat_storage_t` are saved to *heat_storage_t_p.csv* and *heat_storage_t_e.csv*.
- 'hv_requirement_slacks_t' : Attribute `hv_requirement_slacks_t` is saved to *hv_requirement_slacks_t.csv*.
- 'grid_slacks_t' : The results of the five variables in attribute `grid_slacks_t` are saved to *dispatchable_gen_crt.csv*, *non_dispatchable_gen_crt.csv*, *load_shedding.csv*, *cp_load_shedding.csv* and *hp_load_shedding.csv*.
- 'overlying_grid' : Attribute `overlying_grid` is saved to *overlying_grid.csv*.

Parameters

- **directory** (*str*) – Path to save OPF results data to.
- **attributes** (*list(str) or None*) – List of attributes to export. See above for attributes that can be exported. If None, all specified attributes are exported. Default: None.

from_csv(*data_path*, *from_zip_archive=False*)

Restores OPF results from csv files.

Parameters

- **data_path** (*str*) – Path to OPF results csv files.
- **from_zip_archive** (*bool, optional*) – Set True if data is archived in a zip archive. Default: False.

Submodules

`edisgo.opf.powermodels_opf`

Module Contents

Functions

pm_optimize(edisgo_obj[, s_base, flexible_cps, ...])	Run OPF for edisgo object in julia subprocess and write results of OPF to edisgo

`edisgo.opf.powermodels_opf.`**`pm_optimize`**(*edisgo_obj*, *s_base=1*, *flexible_cps=None*, *flexible_hps=None*, *flexible_loads=None*, *flexible_storage_units=None*, *opf_version=1*, *method='soc'*, *warm_start=False*, *silence_moi=False*)

Run OPF for edisgo object in julia subprocess and write results of OPF to edisgo object. Results of OPF are time series of operation schedules of flexibilities.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **s_base** (*int*) – Base value of apparent power for per unit system. Default: 1 MVA.
- **flexible_cps** (numpy.ndarray or None) – Array containing all charging points that allow for flexible charging. Default: None.
- **flexible_hps** (numpy.ndarray or None) – Array containing all heat pumps that allow for flexible operation due to an attached heat storage. Default: None.
- **flexible_loads** (numpy.ndarray or None) – Array containing all flexible loads that allow for application of demand side management strategy. Default: None.
- **flexible_storage_units** (numpy.ndarray or None) – Array containing all flexible storage units. Non-flexible storage units operate to optimize self consumption. Default: None
- **opf_version** (*int*) – Version of optimization models to choose from. The grid model is a radial branch flow model (BFM). Optimization versions differ in lifted or additional constraints and the objective function. Implemented versions are:
  - **1**
    - Lifted constraints: grid restrictions
    - Objective: minimize line losses and maximal line loading
  - **2**
    - Objective: minimize line losses and grid related slacks
  - **3**
    - Additional constraints: high voltage requirements
    - Lifted constraints: grid restrictions
    - Objective: minimize line losses, maximal line loading and HV slacks
  - **4**

    * Additional constraints: high voltage requirements
    * Objective: minimize line losses, HV slacks and grid related slacks

  Must be one of [1, 2, 3, 4]. Default: 1.

- **method** (*str*) – Optimization method to use. Must be either "soc" (Second Order Cone) or "nc" (Non Convex). If method is "soc", OPF is run in PowerModels with Gurobi solver with SOC relaxation of equality constraint P^2+Q^2 = V^2*I^2. If method is "nc", OPF is run with Ipopt solver as a non-convex problem due to quadratic equality constraint P^2+Q^2 = V^2*I^2. Default: "soc".
- **warm_start** (*bool*) – If set to True and if method is set to "soc", non-convex IPOPT OPF will be run additionally and will be warm started with Gurobi SOC solution. Warm-start will only be run if results for Gurobi's SOC relaxation is exact. Default: False.
- **silence_moi** (*bool*) – If set to True, MathOptInterface's optimizer attribute "MOI.Silent" is set to True in julia subprocess. This attribute is for silencing the output of an optimizer. When set to True, it requires the solver to produce no output, hence there will be no logging coming from julia subprocess in python process. Default: False.
- **save_heat_storage** (*bool*) – Indicates whether to save results of heat storage variables from the optimization to eDisGo object. Default: True.
- **save_slack_gen** (*bool*) – Indicates whether to save results of slack generator variables from the optimization to eDisGo object. Default: True.
- **save_slacks** (*bool*) – Indicates whether to save results of slack variables of OPF. Depending on chosen opf_version, different slacks are used. For more information see `edisgo.io.powermodels_io.from_powermodels()`. Default: True.

edisgo.tools

Submodules

edisgo.tools.config

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: https://edisgo.readthedocs.io/en/dev/

Based on code by oemof developing group

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

Module Contents

Classes

Config	Container for all configurations.

Functions

load_config(filename[, config_dir, copy_default_config])	Loads the specified config file.
get(section, key)	Returns the value of a given key of a given section of the main
get_default_config_path()	Returns the basic edisgo config path. If it does not yet exist it creates
make_directory(directory)	Makes directory if it does not exist.

`class edisgo.tools.config.Config(**kwargs)`

Container for all configurations.

Parameters

- **config_path** (*None or str or :dict*) – Path to the config directory. Options are:
  - **'default' (default)**
    If *config_path* is set to 'default', the provided default config files are used directly.
  - **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 options, the directories and config files must exist and are not automatically created.
  - **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.

  Default: "default".

- **from_json** (*bool*) – Set to True to load config data from json file. In that case the json file is assumed to be located in path specified through *config_path*. Per default this is set to False in which case config data is loaded from cfg files. Default: False.
- **json_filename** (*str*) – Filename of the json file. If None, it is loaded from file with name "configs.json". Default: None.
- **from_zip_archive** (*bool*) – Set to True to load json config file from zip archive. Default: False.

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']
```

from_cfg(*config_path=None*)

Load config files.

Parameters

config_path (*None or str or dict*) – See class definition for more information.

Returns

eDisGo configuration data from config files.

Return type

collections.OrderedDict

to_json(*directory*, *filename=None*)

Saves config data to json file.

Parameters

- **directory** (*str*) – Directory, the json file is saved to.
- **filename** (*str or None*) – Filename the json file is saved under. If None, it is saved under the filename "configs.json". Default: None.

from_json(*directory*, *filename=None*, *from_zip_archive=False*)

Imports config data from json file as dictionary.

Parameters

- **directory** (*str*) – Directory, the json file is loaded from.
- **filename** (*str or None*) – Filename of the json file. If None, it is loaded from file with name "configs.json". Default: None.
- **from_zip_archive** (*bool*) – Set to True if data is archived in a zip archive. Default: False.

Returns
Dictionary with config data loaded from json file.

Return type
dict

`__getitem__`(*key1*, *key2=None*)

`__setitem__`(*key*, *value*)

`__delitem__`(*key*)

`__iter__`()

`__len__`()

`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` (*bool*) – 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 bool 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.geo

Module Contents

Functions

proj2equidistant(srid)	Transforms to equidistant projection (epsg:3035).
proj2equidistant_reverse(srid)	Transforms back from equidistant projection to given projection.
proj_by_srids(srid1, srid2)	Transforms from specified projection to other specified projection.
calc_geo_lines_in_buffer(grid_topology, bus, grid[, ...])	Determines lines that are at least partly within buffer around given bus.
calc_geo_dist_vincenty(grid_topology, bus_source, ...)	Calculates the geodesic distance between two buses in km.
find_nearest_bus(point, bus_target)	Finds the nearest bus in *bus_target* to a given point.
find_nearest_conn_objects(grid_topology, bus, lines[, ...])	Searches all lines for the nearest possible connection object per line.
mv_grid_gdf(edisgo_obj)	

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

Transforms to equidistant projection (epsg:3035).

Parameters

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

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.

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.

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 the 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 in km.

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.geo.`**`mv_grid_gdf`**(*edisgo_obj:* edisgo.EDisGo)

edisgo.tools.geopandas_helper

Module Contents

Classes

GeoPandasGridContainer	Grids geo data for all components with information about their geolocation.

Functions

to_geopandas(grid_obj)	Translates all DataFrames with geolocations within a Grid class to GeoDataFrames.

class `edisgo.tools.geopandas_helper.`**`GeoPandasGridContainer`**(*crs:* str, *grid_id:* str | int, *grid:* edisgo.network.grids.Grid, *buses_gdf: geopandas.GeoDataFrame*, *generators_gdf: geopandas.GeoDataFrame*, *loads_gdf: geopandas.GeoDataFrame*, *storage_units_gdf: geopandas.GeoDataFrame*, *transformers_gdf: geopandas.GeoDataFrame*, *lines_gdf: geopandas.GeoDataFrame*)

Grids geo data for all components with information about their geolocation.

Parameters

- **crs** (*str*) – Coordinate Reference System of the geometry objects.
- **id** (*str or int*) – Grid identifier
- **grid** (*Grid*) – Matching grid object
- **buses_gdf** (geopandas.GeoDataFrame) – GeoDataframe with all buses in the Grid. See `buses_df` for more information.
- **generators_gdf** (geopandas.GeoDataFrame) – GeoDataframe with all generators in the Grid. See `generators_df` for more information.
- **loads_gdf** (geopandas.GeoDataFrame) – GeoDataframe with all loads in the Grid. See `loads_df` for more information.
- **storage_units_gdf** (geopandas.GeoDataFrame) – GeoDataframe with all storage units in the Grid. See `storage_units_df` for more information.
- **transformers_gdf** (geopandas.GeoDataFrame) – GeoDataframe with all transformers in the Grid. See `transformers_df` for more information.
- **lines_gdf** (geopandas.GeoDataFrame) – GeoDataframe with all lines in the Grid. See `loads_df` for more information.

`edisgo.tools.geopandas_helper.`**`to_geopandas`**(*grid_obj:* edisgo.network.grids.Grid)

Translates all DataFrames with geolocations within a Grid class to GeoDataFrames.

Parameters

grid_obj (*Grid*) – Grid object to transform.

Returns

Data container with the grids geo data for all components with information about their geolocation.

Return type

GeoPandasGridContainer

edisgo.tools.logger

Module Contents

Functions

setup_logger([file_name, log_dir, loggers, ...])	Setup different loggers with individual logging levels and where to write output.

`edisgo.tools.logger.`**`setup_logger`**(*file_name=None*, *log_dir=None*, *loggers=None*, *stream_output=sys.stdout*, *debug_message=False*, *reset_loggers=False*)

Setup different loggers with individual logging levels and where to write output.

The following table from python ‘Logging Howto’ shows you when which logging level is used.

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

Parameters

- **file_name** (*str or None*) – Specifies file name of file logging information is written to. Possible options are:
  - **None (default)**
    Saves log file with standard name *%Y_%m_%d-%H:%M:%S_edisgo.log*.
  - **str**
    Saves log file with the specified file name.
- **log_dir** (*str or None*) – Specifies directory log file is saved to. Possible options are:
  - **None (default)**
    Saves log file in current working directory.
  - **"default"**
    Saves log file into directory configured in the configs.
  - **str**
    Saves log file into the specified directory.
- **loggers** (*None or list(dict)*) –
  - **None**
    Configuration as shown in the example below is used. Configures root logger with file and stream level warning and the edisgo logger with file and stream level debug.
  - **list(dict)**
    List of dicts with the logger configuration. Each dictionary must contain the following keys and corresponding values:
    - **'name'**
      Specifies name of the logger as string, e.g. 'root' or 'edisgo'.
    - **'file_level'**
      Specifies file logging level. Possible options are:
      - **"debug"**
        Logs logging messages with logging level logging.DEBUG and above.
      - **"info"**
        Logs logging messages with logging level logging.INFO and above.
      - **"warning"**
        Logs logging messages with logging level logging.WARNING and above.

· **"error"**
Logs logging messages with logging level logging.ERROR and above.

· **"critical"**
Logs logging messages with logging level logging.CRITICAL.

· **None**
No logging messages are logged.

∗ **'stream_level'**
Specifies stream logging level. Possible options are the same as for *file_level*.

- **stream_output** (*stream*) – Default sys.stdout is used. sys.stderr is also possible.
- **debug_message** (*bool*) – If True the handlers of every configured logger is printed.
- **reset_loggers** (*bool*) – If True the handlers of all loggers are cleared before configuring the loggers. Only use if you know what you do, it could be dangerous.

Examples

```
>>> setup_logger(
>>>     loggers=[
>>>         {"name": "root", "file_level": "warning", "stream_level": "warning"},
>>>         {"name": "edisgo", "file_level": "info", "stream_level": "info"}
>>>     ]
>>> )
```

edisgo.tools.networkx_helper

Module Contents

Functions

translate_df_to_graph(→ networkx.Graph)	Translate DataFrames to networkx Graph Object.

edisgo.tools.networkx_helper.**translate_df_to_graph**(*buses_df: pandas.DataFrame*, *lines_df: pandas.DataFrame*, *transformers_df: pandas.DataFrame | None = None*) → networkx.Graph

Translate DataFrames to networkx Graph Object.

Parameters

- **buses_df** (pandas.DataFrame) – Dataframe with all buses to use as Graph nodes. For more information about the Dataframe see *buses_df*.
- **lines_df** (pandas.DataFrame) – Dataframe with all lines to use as Graph branches. For more information about the Dataframe see *lines_df*
- **transformers_df** (pandas.DataFrame, optional) – Dataframe with all transformers to use as additional Graph nodes. For more information about the Dataframe see *transformers_df*

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

`edisgo.tools.plots`

Module Contents

Functions

histogram(data, **kwargs)	Function to create histogram, e.g. for voltages or currents.
add_basemap(ax[, zoom])	Adds map to a plot.
get_grid_district_polygon(config[, subst_id, projection])	Get MV network district polygon from oedb for plotting.
mv_grid_topology(edisgo_obj[, timestep, line_color, ...])	Plot line loading as color on lines.
color_map_color(→ str)	Get matching color for a value on a matplotlib color map.
plot_plotly(→ plotly.basedatatypes.BaseFigure)	Draws a plotly html figure.
chosen_graph(→ tuple[networkx.Graph, ...)	Get the matching networkx graph from a chosen grid.
plot_dash_app(→ jupyter_dash.JupyterDash)	Generates a jupyter dash app from given eDisGo object(s).
plot_dash(edisgo_objects[, mode, debug, port, height])	Shows the generated jupyter dash app from given eDisGo object(s).

`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*, *grid_expansion_costs=None*, *filename=None*, *arrows=False*, *grid_district_geom=True*, *background_map=True*, *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 voltage at each node. In case several time steps are selected the maximum voltage is shown.

  - ’voltage_deviation’ Node color is set according to voltage deviation from 1 p.u.. In case several time steps are selected the maximum absolute voltage deviation from 1 p.u. is shown.
  - ’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.
- **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.plots.`**`color_map_color`**(*value: numbers.Number*, *vmin: numbers.Number*, *vmax: numbers.Number*, *cmap_name: str | list = 'coolwarm'*) → str

Get matching color for a value on a matplotlib color map.

Parameters

- **value** (*float or int*) – Value to get color for
- **vmin** (*float or int*) – Minimum value on color map
- **vmax** (*float or int*) – Maximum value on color map
- **cmap_name** (*str or list*) – Name of color map to use, or the colormap

Returns

Color name in hex format

Return type

str

`edisgo.tools.plots.`**`plot_plotly`**(*edisgo_obj: edisgo.EDisGo, grid: edisgo.network.grids.Grid | None = None, line_color: None | str = 'relative_loading', node_color: None | str = 'voltage_deviation', line_result_selection: str = 'max', node_result_selection: str = 'max', selected_timesteps: pandas.Timestamp | list | None = None, plot_map: bool = False, pseudo_coordinates: bool = False, node_selection: list | bool = False, height: int = 500*) → plotly.basedatatypes.BaseFigure

Draws a plotly html figure.

Parameters

- **edisgo_obj** (*EDisGo*) – Selected edisgo_obj to get plotting information from.
- **grid** (*Grid*) – Grid to plot. If None, the MVGrid of the edisgo_obj is plotted. Default: None.
- **line_color** (*str or None*) – Defines whereby to choose line colors. Possible options are:
  - **'loading'**
    Line color is set according to loading of the line.
  - **'relative_loading' (default)**
    Line color is set according to relative loading of the line.
  - **'reinforce'**
    Line color is set according to investment costs of the line.
  - **None**
    Line color is black. This is also the fallback, in case other options fail.
- **node_color** (*str or None*) – Defines whereby to choose node colors. Possible options are:
  - **'adjacencies'**
    Node color as well as size is set according to the number of direct neighbors.
  - **'voltage_deviation' (default)**
    Node color is set according to voltage deviation from 1 p.u..
  - **None**
    Line color is black. This is also the fallback, in case other options fail.
- **line_result_selection** (*str*) – Defines which values are shown for the load of the lines:
  - **'min'**
    Minimal line load of all time steps.
  - **'max' (default)**
    Maximal line load of all time steps.
- **node_result_selection** (*str*) – Defines which values are shown for the voltage of the nodes:

  - **'min'**
    Minimal node voltage of all time steps.
  - **'max' (default)**
    Maximal node voltage of all time steps.
- **selected_timesteps** (pandas.Timestamp or list(pandas.Timestamp) or None) – Selected time steps to show results for.
  - **None (default)**
    All time steps are used.
  - list(pandas.Timestamp) or pandas.Timestamp Selected time steps are used.
- **plot_map** (*bool*) – Enable the plotting of a background map.
- **pseudo_coordinates** (*bool*) – Enable pseudo coordinates for the plotted grid. Default: False.
- **node_selection** (*bool or list(str)*) – Only plot selected nodes. Default: False.
- **height** (*int*) – Height of the plotly plot in pixels.

Returns
Plotly figure with branches and nodes.

Return type
plotly.plotly.graph_objects.Figure

`edisgo.tools.plots.`**`chosen_graph`**(*edisgo_obj: edisgo.EDisGo*, *selected_grid: str*) → tuple[networkx.Graph, bool | *edisgo.network.grids.Grid*]

Get the matching networkx graph from a chosen grid.

Parameters
- **edisgo_obj** (*EDisGo*) –
- **selected_grid** (*str*) – Grid name. Can be either 'Grid' to select the MV grid with all LV grids or the name of the MV grid to select only the MV grid or the name of one of the LV grids of the eDisGo object to select a specific LV grid.

Returns
Tuple with the first entry being the networkx graph of the selected grid and the second entry the grid to use as root node. See `draw_plotly()` for more information.

Return type
(networkx.Graph, *Grid* or bool)

`edisgo.tools.plots.`**`plot_dash_app`**(*edisgo_objects: edisgo.EDisGo | dict[str, edisgo.EDisGo]*, *debug: bool = False*, *height: int = 500*) → jupyter_dash.JupyterDash

Generates a jupyter dash app from given eDisGo object(s).

Parameters
- **edisgo_objects** (*EDisGo* or dict[str, *EDisGo*]) – eDisGo objects to show in plotly dash app. In the case of multiple edisgo objects pass a dictionary with the eDisGo objects as values and the respective eDisGo object names as keys.
- **height** (*int*) – Height of the plotly plot in pixels.
- **debug** (*bool*) – Debugging for the dash app:
  - **False (default)**
    Disable debugging for the dash app.
  - **True**
    Enable debugging for the dash app.

Returns
Jupyter dash app.

Return type
JupyterDash

`edisgo.tools.plots.`**`plot_dash`**(*edisgo_objects: edisgo.EDisGo | dict[str, edisgo.EDisGo]*, *mode: str = 'inline'*, *debug: bool = False*, *port: int = 8050*, *height: int = 820*)

Shows the generated jupyter dash app from given eDisGo object(s).

Parameters

- **edisgo_objects** (*EDisGo* or dict[str, *EDisGo*]) – eDisGo objects to show in plotly dash app. In the case of multiple edisgo objects pass a dictionary with the eDisGo objects as values and the respective eDisGo object names as keys.
- **mode** (*str*) – Display mode
  - **"inline" (default)**
    Jupyter lab inline plotting.
  - **"jupyterlab"**
    Plotting in own Jupyter lab tab.
  - **"external"**
    Plotting in own browser tab.
- **debug** (*bool*) – If True, enables debugging of the jupyter dash app.
- **port** (*int*) – Port which the app uses. Default: 8050.
- **height** (*int*) – Height of the jupyter dash cell.

edisgo.tools.powermodels_io

Module Contents

Functions

to_powermodels(pypsa_net)	Convert pypsa network to network dictionary format, using the pypower
convert_storage_series(timeseries)	
add_storage_from_edisgo(edisgo_obj, psa_net, pm_dict)	Read static storage data (position and capacity) from eDisGo and export to
pypsa2ppc(psa_net)	Converter from pypsa data structure to pypower data structure
ppc2pm(ppc, psa_net)	converter from pypower datastructure to powermodels dictionary,

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/pandamodels/to_pm.py

Parameters

`ppc` –

Returns

`edisgo.tools.preprocess_pypsa_opf_structure`

Module Contents

Functions

`preprocess_pypsa_opf_structure`(edisgo_grid, psa_network)	Prepares pypsa network for OPF problem.
`aggregate_fluct_generators`(psa_network)	Aggregates fluctuating generators of same type at the same node.

`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.pseudo_coordinates

Module Contents

Functions

make_pseudo_coordinates_graph(→ networkx.Graph)	Generates pseudo coordinates for one graph.
make_pseudo_coordinates(edisgo_obj[, mv_coordinates])	Generates pseudo coordinates for all LV grids and optionally MV grid.

edisgo.tools.pseudo_coordinates.**make_pseudo_coordinates_graph**(*G: networkx.Graph*, *branch_detour_factor: float*) → networkx.Graph

Generates pseudo coordinates for one graph.

Parameters

- **G** (networkx.Graph) – Graph object to generate pseudo coordinates for.
- **branch_detour_factor** (*float*) – Defines the quotient of the line length and the distance of the buses.

Returns

Graph with pseudo coordinates for all nodes.

Return type

networkx.Graph

edisgo.tools.pseudo_coordinates.**make_pseudo_coordinates**(*edisgo_obj: edisgo.EDisGo*, *mv_coordinates: bool = False*)

Generates pseudo coordinates for all LV grids and optionally MV grid.

Bus coordinates are changed in the Topology object directly. If you want to keep information on the original coordinates, hand a copy of the EDisGo object to this function.

Parameters

- **edisgo_obj** (*EDisGo*) – eDisGo object to create pseudo coordinates for.
- **mv_coordinates** (*bool, optional*) – If False, pseudo coordinates are only generated for LV buses. If True, pseudo coordinates are as well generated for MV buses. Default: False.

edisgo.tools.spatial_complexity_reduction

Module Contents

Functions

find_buses_of_interest(→ set)	Return buses with load and voltage issues, determined doing a worst-case powerflow
rename_virtual_buses(→ pandas.DataFrame)	Rename virtual buses so that no virtual transformer bus is created.
remove_short_end_lines(edisgo_obj)	Method to remove end lines under 1 meter to reduce size of edisgo object.
remove_lines_under_one_meter(→ edisgo.EDisGo)	Remove the lines under one meter. Sometimes these line are causing convergence
make_busmap_grid(→ pandas.DataFrame)	Making busmap for the cluster area 'grid'.
make_busmap_feeders(→ pandas.DataFrame)	Making busmap for the cluster area 'feeder'.
make_busmap_main_feeders(→ pandas.DataFrame)	Making busmap for the cluster area 'main_feeder'.
make_busmap(→ pandas.DataFrame)	Determines which busses are clustered.
apply_busmap(→ pandas.DataFrame)	Function to reduce the EDisGo object with a previously generated busmap.
spatial_complexity_reduction(→ tuple[pandas.DataFrame, ...)	Reduces the number of busses and lines by applying a spatial clustering.
compare_voltage(→ tuple[pandas.DataFrame, float])	Compares the voltages per node between the unreduced and the reduced EDisGo object.
compare_apparent_power(→ tuple[pandas.DataFrame, float])	Compares the apparent power over each line between the unreduced and the reduced

`edisgo.tools.spatial_complexity_reduction.`**`find_buses_of_interest`**(*edisgo_root: edisgo.EDisGo*) → set

Return buses with load and voltage issues, determined doing a worst-case powerflow analysis.

Parameters
`edisgo_root` (*EDisGo*) – The investigated EDisGo object.

Returns
Set with the names of the buses with load and voltage issues.

Return type
set(str)

`edisgo.tools.spatial_complexity_reduction.`**`rename_virtual_buses`**(*partial_busmap_df: pandas.DataFrame, transformer_node: str*) → pandas.DataFrame

Rename virtual buses so that no virtual transformer bus is created.

Parameters
- **`partial_busmap_df`** (pandas.DataFrame) – Busmap to work on.
- **`transformer_node`** (*str*) – Transformer node name.

Returns
Busmap with applied changes.

Return type
pandas.DataFrame

`edisgo.tools.spatial_complexity_reduction.`**`remove_short_end_lines`**(*edisgo_obj: edisgo.EDisGo*)

Method to remove end lines under 1 meter to reduce size of edisgo object.

Short lines inside at the end are removed in this function, including the end node. Components that were originally connected to the end node are reconnected to the upstream node.

This function does currently not remove short lines that are no end lines.

Parameters

edisgo (*EDisGo*) –

`edisgo.tools.spatial_complexity_reduction.`**`remove_lines_under_one_meter`**(*edisgo_obj:* edisgo.EDisGo) → *edisgo.EDisGo*

Remove the lines under one meter. Sometimes these line are causing convergence problems of the power flow calculation or making problems with the clustering methods.

Function might be a bit overengineered, so that the station bus is never dropped.

`edisgo.tools.spatial_complexity_reduction.`**`make_busmap_grid`**(*edisgo_obj:* edisgo.EDisGo, *grid: None* | *str = None*, *mode: str = 'kmeansdijkstra'*, *reduction_factor: float = 0.25*, *preserve_trafo_bus_coordinates: bool = True*) → pandas.DataFrame

Making busmap for the cluster area 'grid'.

Every grid is clustered individually.

Parameters

- **edisgo_obj** (*EDisGo*) – EDisGo object for which the busmap is created.
- **grid** (*str or None*) – If None, busmap is created for all grids, else only for the selected grid. Default: None.
- **mode** (*str*) – "kmeans" or "kmeansdijkstra" as clustering method. See parameter *mode* in function `spatial_complexity_reduction` for more information. Default: "kmeansdijkstra".
- **reduction_factor** (*float*) – Factor to reduce number of nodes by. Must be between 0 and 1. Default: 0.25.
- **preserve_trafo_bus_coordinates** (*True*) – If True, transformers have the same coordinates after the clustering, else the transformer coordinates are changed by the clustering. Default: True.

Returns

Busmap which maps the old bus names to the new bus names with new coordinates. See return value in function `make_busmap()` for more information.

Return type

pandas.DataFrame

References

In parts based on PyPSA spatial complexity reduction.

`edisgo.tools.spatial_complexity_reduction.`**`make_busmap_feeders`**(*edisgo_obj:* edisgo.EDisGo *= None*, *grid: None* | edisgo.network.grids.Grid *= None*, *mode: str = 'kmeansdijkstra'*, *reduction_factor: float = 0.25*, *reduction_factor_not_focused: bool* | *float = False*) → pandas.DataFrame

Making busmap for the cluster area 'feeder'.

Every feeder is clustered individually.

Parameters

- **edisgo_obj** (*EDisGo*) – EDisGo object for which the busmap is created.
- **grid** (*str or None*) – If None, busmap is created for all grids, else only for the selected grid. Default: None.
- **mode** (*str*) – "kmeans" or "kmeansdijkstra" as clustering method. See parameter *mode* in function `spatial_complexity_reduction` for more information. Default: "kmeansdijkstra".
- **reduction_factor** (*float*) – Factor to reduce number of nodes by. Must be between 0 and 1. Default: 0.25.
- **reduction_factor_not_focused** (*bool or float*) – If False, the focus method is not used. If between 0 and 1, this sets the reduction factor for buses not of interest. See parameter *reduction_factor_not_focused* in function `make_busmap()` for more information. Default: False.

Returns

Busmap which maps the old bus names to the new bus names with new coordinates. See return value in function `make_busmap()` for more information.

Return type

pandas.DataFrame

References

In parts based on PyPSA spatial complexity reduction.

`edisgo.tools.spatial_complexity_reduction.`**`make_busmap_main_feeders`**`(`*edisgo_obj: edisgo.EDisGo = None*, *grid: None | edisgo.network.grids.Grid = None*, *mode: str = 'kmeansdijkstra'*, *reduction_factor: float = 0.25*, *reduction_factor_not_focused: bool | float = False*`)` → pandas.DataFrame

Making busmap for the cluster area 'main_feeder'.

Every main feeder is clustered individually. The main feeder is selected as the longest path in the feeder. All nodes are aggregated to this main feeder and then the feeder is clustered.

Parameters

- **edisgo_obj** (*EDisGo*) – EDisGo object for which the busmap is created.
- **grid** (*str or None*) – If None, busmap is created for all grids, else only for the selected grid. Default: None.
- **mode** (*str*) – "kmeans", "kmeansdijkstra", "aggregate_to_main_feeder" or "equidistant_nodes" as clustering method. See parameter *mode* in function `spatial_complexity_reduction` for more information. Default: "kmeansdijkstra".
- **reduction_factor** (*float*) – Factor to reduce number of nodes by. Must be between 0 and 1. Default: 0.25.
- **reduction_factor_not_focused** (*bool or float*) – If False, the focus method is not used. If between 0 and 1, this sets the reduction factor for buses not of interest. See parameter *reduction_factor_not_focused* in function `make_busmap()` for more information. Default: False.

Returns

Busmap which maps the old bus names to the new bus names with new coordinates. See return value in function `make_busmap()` for more information.

Return type

pandas.DataFrame

References

In parts based on PyPSA spatial complexity reduction.

`edisgo.tools.spatial_complexity_reduction.`**`make_busmap`**(*edisgo_obj: edisgo.EDisGo*, *mode: str = 'kmeansdijkstra'*, *cluster_area: str = 'feeder'*, *reduction_factor: float = 0.25*, *reduction_factor_not_focused: bool | float = False*, *grid: None | edisgo.network.grids.Grid = None*) → pandas.DataFrame

Determines which busses are clustered.

The information on which original busses are clustered to which new busses is given in the so-called busmap dataframe. The busmap can be used with the function `apply_busmap()` to perform a spatial complexity reduction.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) – EDisGo object for which the busmap is created.
- **`mode`** (*`str`*) – Clustering method to use. See parameter *mode* in function `spatial_complexity_reduction` for more information.
- **`cluster_area`** (*`str`*) – The cluster area is the area the different clustering methods are applied to. Possible options are 'grid', 'feeder' or 'main_feeder'. Default: "feeder".
- **`reduction_factor`** (*`float`*) – Factor to reduce number of nodes by. Must be between 0 and 1. Default: 0.25.
- **`reduction_factor_not_focused`** (*`bool or float`*) – If False, uses the same reduction factor for all cluster areas. If between 0 and 1, this sets the reduction factor for buses not of interest (these are buses without voltage or overloading issues, that are determined through a worst case power flow analysis). When selecting 0, the nodes of the clustering area are aggregated to the transformer bus. This parameter is only used when parameter *cluster_area* is set to 'feeder' or 'main_feeder'. Default: False.
- **`grid`** (*`str or None`*) – If None, busmap is created for all grids, else only for the selected grid.

Returns

Busmap which maps the old bus names to the new bus names with new coordinates. Columns are "new_bus" with new bus name, "new_x" with new x-coordinate and "new_y" with new y-coordinate. Index of the dataframe holds bus names of original buses as in buses_df.

Return type

pandas.DataFrame

References

In parts based on PyPSA spatial complexity reduction.

`edisgo.tools.spatial_complexity_reduction.`**`apply_busmap`**(*edisgo_obj: edisgo.EDisGo*, *busmap_df: pandas.DataFrame*, *line_naming_convention: str = 'standard_lines'*, *aggregation_mode: bool = False*, *load_aggregation_mode: str = 'sector'*, *generator_aggregation_mode: str = 'type'*) → pandas.DataFrame

Function to reduce the EDisGo object with a previously generated busmap.

Warning: After reduction, 'in_building' of all buses is set to False. Also, the method only works if all buses have x and y coordinates. If this is not the case, you can use the function `make_pseudo_coordinates()` to set coordinates for all buses.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) – EDisGo object to reduce.
- **`busmap_df`** (pandas.DataFrame) – Busmap holding the information which nodes are merged together.
- **`line_naming_convention`** (*`str`*) – Determines how to set "type_info" and "kind" in case two or more lines are aggregated. Possible options are "standard_lines" or "combined_name". If "standard_lines" is selected, the values of the standard line of the respective voltage level are used to set "type_info" and "kind". If "combined_name" is selected, "type_info" and "kind" contain the concatenated values of the merged lines. x and r of the lines are not influenced by this as they are always determined from the x and r values of the aggregated lines. Default: "standard_lines".
- **`aggregation_mode`** (*`bool`*) – Specifies, whether to aggregate loads and generators at the same bus or not. If True, loads and generators at the same bus are aggregated according to their selected modes (see parameters *load_aggregation_mode* and *generator_aggregation_mode*). Default: False.
- **`load_aggregation_mode`** (*`str`*) – Specifies, how to aggregate loads at the same bus, in case parameter *aggregation_mode* is set to True. Possible options are "bus" or "sector". If "bus" is chosen, loads are aggregated per bus. When "sector" is chosen, loads are aggregated by bus, type and sector. Default: "sector".
- **`generator_aggregation_mode`** (*`str`*) – Specifies, how to aggregate generators at the same bus, in case parameter *aggregation_mode* is set to True. Possible options are "bus" or "type". If "bus" is chosen, generators are aggregated per bus. When "type" is chosen, generators are aggregated by bus and type.

Returns

Linemap which maps the old line names (in the index of the dataframe) to the new line names (in column "new_line_name").

Return type

pandas.DataFrame

References

In parts based on PyPSA spatial complexity reduction.

`edisgo.tools.spatial_complexity_reduction.`**`spatial_complexity_reduction`**(*edisgo_obj:* edisgo.EDisGo, *mode:* str *= 'kmeansdijkstra'*, *cluster_area:* str *= 'feeder'*, *reduction_factor:* float *= 0.25*, *reduction_factor_not_focused:* bool | float *= False*, *apply_pseudo_coordinates:* bool *= True*, ***kwargs*) → tuple[pandas.DataFrame, pandas.DataFrame]

Reduces the number of busses and lines by applying a spatial clustering.

See function `spatial_complexity_reduction` for more information.

Parameters

- **edisgo_obj** (*EDisGo*) – EDisGo object to apply spatial complexity reduction to.
- **mode** (*str*) – Clustering method to use. See parameter *mode* in function `spatial_complexity_reduction` for more information.
- **cluster_area** (*str*) – The cluster area is the area the different clustering methods are applied to. See parameter *cluster_area* in function `spatial_complexity_reduction` for more information.
- **reduction_factor** (*float*) – Factor to reduce number of nodes by. Must be between 0 and 1. Default: 0.25.
- **reduction_factor_not_focused** (*bool or float*) – If False, uses the same reduction factor for all cluster areas. If between 0 and 1, this sets the reduction factor for buses not of interest. See parameter *reduction_factor_not_focused* in function `spatial_complexity_reduction` for more information.
- **apply_pseudo_coordinates** (*bool*) – If True pseudo coordinates are applied. The spatial complexity reduction method is only tested with pseudo coordinates. Default: True.
- **line_naming_convention** (*str*) – Determines how to set "type_info" and "kind" in case two or more lines are aggregated. See parameter *line_naming_convention* in function `spatial_complexity_reduction` for more information.
- **aggregation_mode** (*bool*) – Specifies, whether to aggregate loads and generators at the same bus or not. See parameter *aggregation_mode* in function `spatial_complexity_reduction` for more information.
- **load_aggregation_mode** (*str*) – Specifies, how to aggregate loads at the same bus, in case parameter *aggregation_mode* is set to True. See parameter *load_aggregation_mode* in function `spatial_complexity_reduction` for more information.
- **generator_aggregation_mode** (*str*) – Specifies, how to aggregate generators at the same bus, in case parameter *aggregation_mode* is set to True. See parameter *generator_aggregation_mode* in function `spatial_complexity_reduction` for more information.
- **mv_pseudo_coordinates** (*bool, optional*) – If True pseudo coordinates are also generated for MV grid. Default: False.

Returns

Returns busmap and linemap dataframes. The busmap maps the original busses to the new busses with new coordinates. Columns are "new_bus" with new bus name, "new_x" with new x-coordinate and "new_y" with new y-coordinate. Index of the dataframe holds bus names of original buses as in buses_df. The linemap maps the original line names (in the index of the dataframe) to the new line names (in column "new_line_name").

Return type

tuple(pandas.DataFrame, pandas.DataFrame)

`edisgo.tools.spatial_complexity_reduction.`**`compare_voltage`**(*edisgo_unreduced: edisgo.EDisGo, edisgo_reduced: edisgo.EDisGo, busmap_df: pandas.DataFrame, timestep: str | pandas.Timestamp*) → tuple[pandas.DataFrame, float]

Compares the voltages per node between the unreduced and the reduced EDisGo object.

The voltage difference for each node in p.u. as well as the root-mean-square error is returned. For the mapping of nodes in the unreduced and reduced network the busmap is used. The calculation is performed for one timestep or the minimum or maximum values of the node voltages.

Parameters

- **edisgo_unreduced** (*EDisGo*) – Unreduced EDisGo object.
- **edisgo_reduced** (*EDisGo*) – Reduced EDisGo object.

- **busmap_df** (pandas.DataFrame) – Busmap for the mapping of nodes.
- **timestep** (str or pandas.Timestamp) – Timestep for which to compare the bus voltage. Can either be a certain time step or 'min' or 'max'.

Returns
Returns a tuple with the first entry being a DataFrame containing the node voltages as well as voltage differences and the second entry being the root-mean-square error. Columns of the DataFrame are "v_unreduced" with voltage in p.u. in unreduced EDisGo object, "v_reduced" with voltage in p.u. in reduced EDisGo object, and "v_diff" with voltage difference in p.u. between voltages in unreduced and reduced EDisGo object. Index of the DataFrame contains the bus names of buses in the unreduced EDisGo object.

Return type
(pandas.DataFrame, rms)

`edisgo.tools.spatial_complexity_reduction.`**`compare_apparent_power`**(*edisgo_unreduced:* edisgo.EDisGo, *edisgo_reduced:* edisgo.EDisGo, *linemap_df: pandas.DataFrame*, *timestep: str*) → tuple[pandas.DataFrame, float]

Compares the apparent power over each line between the unreduced and the reduced EDisGo object.

The difference of apparent power over each line in MVA as well as the root-mean-square error is returned. For the mapping of lines in the unreduced and reduced network the linemap is used. The calculation is performed for one timestep or the minimum or maximum values of the node voltages.

Parameters
- **edisgo_unreduced** (*EDisGo*) – Unreduced EDisGo object.
- **edisgo_reduced** (*EDisGo*) – Reduced EDisGo object.
- **linemap_df** (pandas.DataFrame) – Linemap for the mapping.
- **timestep** (str or pandas.Timestamp) – Timestep for which to compare the apparent power. Can either be a certain time step or 'min' or 'max'.

Returns
Returns a tuple with the first entry being a DataFrame containing the apparent power as well as difference of apparent power for each line and the second entry being the root-mean-square error. Columns of the DataFrame are "s_unreduced" with apparent power in MVA in unreduced EDisGo object, "s_reduced" with apparent power in MVA in reduced EDisGo object, and "s_diff" with difference in apparent power in MVA between apparent power over line in unreduced and reduced EDisGo object. Index of the DataFrame contains the line names of lines in the unreduced EDisGo object.

Return type
(pandas.DataFrame, rms)

`edisgo.tools.temporal_complexity_reduction`

Module Contents

Functions

get_most_critical_time_intervals(edisgo_obj[, ...])	Get time intervals sorted by severity of overloadings as well as voltage issues.
get_most_critical_time_steps(→ pandas.DatetimeIndex)	Get the time steps with the most critical overloading and voltage issues.

`edisgo.tools.temporal_complexity_reduction.`**`get_most_critical_time_intervals`**(*edisgo_obj*, *num_time_intervals=None*, *percentage=1.0*, *time_steps_per_time_interval=168*, *time_step_day_start=0*, *save_steps=False*, *path=''*, *use_troubleshooting_mode=True*, *overloading_factor=0.95*, *voltage_deviation_factor=0.95*, *weight_by_costs=True*)

Get time intervals sorted by severity of overloadings as well as voltage issues.

The overloading and voltage issues can be weighted by the estimated expansion costs solving the issue would require. The length of the time intervals and hour of day at which the time intervals should begin can be set through the parameters *time_steps_per_time_interval* and *time_step_day_start*.

This function currently only works for an hourly resolution!

Parameters

- **edisgo_obj** (*EDisGo*) – The eDisGo API object
- **num_time_intervals** (*int*) – The number of time intervals of most critical line loading and voltage issues to select. If None, *percentage* is used. Default: None.
- **percentage** (*float*) – The percentage of most critical time intervals to select. The default is 1.0, in which case all most critical time steps are selected. Default: 1.0.
- **time_steps_per_time_interval** (*int*) – Amount of continuous time steps in an interval that violation is determined for. Currently, these can only be multiples of 24. Default: 168.
- **time_step_day_start** (*int*) – Time step of the day at which each interval should start. If you want it to start at midnight, this should be set to 0. Default: 0.
- **save_steps** (*bool*) – If set to True, dataframe with time intervals is saved to csv file. The path can be specified through parameter *path*. Default: False.
- **path** (*str*) – Directory the csv file is saved to. Per default, it takes the current working directory.
- **use_troubleshooting_mode** (*bool*) – If set to True, non-convergence issues in power flow are tried to be handled by reducing load and feed-in in steps of 10% down to 20% of the original load and feed-in until the power flow converges. The most critical time intervals are then determined based on the power flow results with the reduced load and feed-in. If False, an error will be raised in case time steps do not converge. Default: True.
- **overloading_factor** (*float*) – Factor at which an overloading of a component is considered to be close enough to the highest overloading of that component. This is used to determine the number of components that reach their highest overloading in each time interval. Per default, it is set to 0.95, which means that if the highest overloading of a component is 2, it will be considered maximally overloaded at an overloading of higher or equal to 2*0.95. Default: 0.95.
- **voltage_deviation_factor** (*float*) – Factor at which a voltage deviation at a bus is considered to be close enough to the highest voltage deviation at that bus. This is used to determine the number of buses that reach their highest voltage deviation in each time interval. Per default, it is set to 0.95. This means that if the highest voltage deviation at a bus is 0.2, it will be included in the determination of number of buses that reach their maximum voltage deviation in a certain time interval at a voltage deviation of higher or equal to 0.2*0.95. Default: 0.95.
- **weight_by_costs** (*bool*) – Defines whether overloading and voltage issues should be

weighted by estimated grid expansion costs or not. This can be done in order to take into account that some grid issues are more relevant, as reinforcing a certain line or feeder will be more expensive than another one.

In case of voltage issues: If True, the costs for each MV and LV feeder, as well as MV/LV station are determined using the costs for earth work and new lines over the full length of the feeder respectively for a new MV/LV station. In each time interval, the estimated costs are only taken into account, in case there is a voltage issue somewhere in the feeder. The costs don't convey the actual costs but are an estimation, as the real number of parallel lines needed is not determined and the whole feeder length is used instead of the length over two-thirds of the feeder. If False, only the maximum voltage deviation in the feeder is used to determine the most relevant time intervals.

In case of overloading issues: If True, the overloading of each line is multiplied by the respective grid expansion costs of that line including costs for earth work and one new line. The costs don't convey the actual costs but are an estimation, as the discrete number of needed parallel lines is not considered. If False, only the relative overloading is used to determine the most relevant time intervals.

Default: True.

Returns

Contains time intervals in which grid expansion needs due to overloading and voltage issues are detected. The time intervals are determined independently for overloading and voltage issues and sorted descending by the expected cumulated grid expansion costs, so that the time intervals with the highest expected costs correspond to index 0. In case of overloading, the time steps in the respective time interval are given in column "time_steps_overloading" and the share of components for which the maximum overloading is reached during the time interval is given in column "percentage_max_overloaded_components". For voltage issues, the time steps in the respective time interval are given in column "time_steps_voltage_issues" and the share of buses for which the maximum voltage deviation is reached during the time interval is given in column "percentage_buses_max_voltage_deviation".

Return type

pandas.DataFrame

`edisgo.tools.temporal_complexity_reduction.`**`get_most_critical_time_steps`**(*edisgo_obj: edisgo.EDisGo, mode=None, timesteps=None, lv_grid_id=None, scale_timeseries=None, num_steps_loading=None, num_steps_voltage=None, percentage: float = 1.0, use_troubleshooting_mode=True, run_initial_analyze=True, weight_by_costs=True*) → pandas.DatetimeIndex

Get the time steps with the most critical overloading and voltage issues.

Parameters

- **`edisgo_obj`** (*`EDisGo`*) – The eDisGo API object
- **`mode`** (*`str or None`*) – Allows to toggle between power flow analysis for the whole network or just the MV or one LV grid. See parameter *mode* in function `analyze` for more information.
- **`timesteps`** (pandas.DatetimeIndex or pandas.Timestamp) – Timesteps specifies from

which time steps to select most critical ones. It defaults to None in which case all time steps in `timeindex` are used.

- **`lv_grid_id`** (*`int or str`*) – ID (e.g. 1) or name (string representation, e.g. "LV-Grid_1") of LV grid to analyze in case mode is 'lv'. Default: None.
- **`scale_timeseries`** (*`float or None`*) – See parameter *scale_timeseries* in function `analyze` for more information.
- **`num_steps_loading`** (*`int`*) – The number of most critical overloading events to select. If None, *percentage* is used. Default: None.
- **`num_steps_voltage`** (*`int`*) – The number of most critical voltage issues to select. If None, *percentage* is used. Default: None.
- **`percentage`** (*`float`*) – The percentage of most critical time steps to select. The default is 1.0, in which case all most critical time steps are selected. Default: 1.0.
- **`use_troubleshooting_mode`** (*`bool`*) – If set to True, non-convergence issues in power flow are tried to be handled by reducing load and feed-in in steps of 10% down to 20% of the original load and feed-in until the power flow converges. The most critical time steps are then determined based on the power flow results with the reduced load and feed-in. If False, an error will be raised in case time steps do not converge. Default: True.
- **`run_initial_analyze`** (*`bool`*) – This parameter can be used to specify whether to run an initial analyze to determine most critical time steps or to use existing results. If set to False, *use_troubleshooting_mode* is ignored. Default: True.
- **`weight_by_costs`** (*`bool`*) – Defines whether overloading and voltage issues should be weighted by estimated grid expansion costs or not. This can be done in order to take into account that some grid issues are more relevant, as reinforcing a certain line or feeder will be more expensive than another one.

  In case of voltage issues: If True, the voltage issues at each bus are weighted by the estimated grid expansion costs for the MV or LV feeder the bus is in or in case of MV/LV stations by the costs for a new transformer. Feeder costs are determined using the costs for earth work and new lines over the full length of the feeder. The costs don't convey the actual costs but are an estimation, as the real number of parallel lines needed is not determined and the whole feeder length is used instead of the length over two-thirds of the feeder. If False, the severity of each feeder's voltage issue is set to be the same.

  In case of overloading issues: If True, the overloading of each line is multiplied by the respective grid expansion costs of that line including costs for earth work and one new line. The costs don't convey the actual costs but are an estimation, as the discrete needed number of parallel lines is not considered. If False, only the relative overloading is used.

  Default: True.

Returns

Time index with unique time steps where maximum overloading or maximum voltage deviation is reached for at least one component respectively bus.

Return type

pandas.DatetimeIndex

edisgo.tools.tools

Module Contents

Functions

select_worstcase_snapshots(edisgo_obj)	Select two worst-case snapshots from time series
calculate_line_reactance(line_inductance_per_km ...)	Calculates line reactance in Ohm.
calculate_line_resistance(line_resistance_per_km ...)	Calculates line resistance in Ohm.
calculate_line_susceptance(line_capacitance_per_ ...)	Calculates line shunt susceptance in Siemens.
calculate_apparent_power(nominal_voltage, current, ...)	Calculates apparent power in MVA from given voltage and current.
drop_duplicated_indices(dataframe[, keep])	Drop rows of duplicate indices in dataframe.
drop_duplicated_columns(df[, keep])	Drop columns of dataframe that appear more than once.
select_cable(edisgo_obj, level, apparent_power)	Selects suitable cable type and quantity using given apparent power.
get_path_length_to_station(edisgo_obj)	Determines path length from each bus to HV-MV station.
get_downstream_buses(edisgo_obj, comp_name[, comp_type])	Returns all buses downstream (farther away from station) of the given bus or line.
assign_voltage_level_to_component(df, buses_df)	Adds column with specification of voltage level component is in.
determine_grid_integration_voltage_level(...)	Gives voltage level component should be integrated into based on its nominal power.
determine_bus_voltage_level(edisgo_object, bus_name)	Gives voltage level as integer from 4 to 7 of given bus.
get_weather_cells_intersecting_with_grid_dis set)	Get all weather cells that intersect with the grid district.
get_directory_size(start_dir)	Calculates the size of all files within the start path.
get_files_recursive(path[, files])	Recursive function to get all files in a given path and its sub directories.
calculate_impedance_for_parallel_components(pu])	Method to calculate parallel impedance and power of parallel elements.
add_line_susceptance(edisgo_obj[, mode])	Adds line susceptance information in Siemens to lines in existing grids.
aggregate_district_heating_components(edisgo_ ...])	Aggregate PtH components that feed into the same district heating network.
reduce_timeseries_data_to_given_timeindex(ed ...)	Reduces timeseries data in EDisGo object to given time index.
resample(object, freq_orig[, method, freq, ...])	Resamples time series data to a desired resolution.
reduce_memory_usage(→ pandas.DataFrame)	Function to automatically check if columns of a pandas DataFrame can
get_year_based_on_timeindex(edisgo_obj)	Checks if *timeindex* is already set and
get_year_based_on_scenario(scenario)	Returns the year the given scenario was set up for.
hash_dataframe(→ str)	Get hash of dataframe.

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_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_line_susceptance`**(*line_capacitance_per_km*, *line_length*, *num_parallel*)

Calculates line shunt susceptance in Siemens.

Parameters
- **`line_capacitance_per_km`** (*float*) – Line capacitance in uF/km.
- **`line_length`** (*float*) – Length of line in km.
- **`num_parallel`** (*int*) – Number of parallel lines.

Returns
Shunt susceptance in Siemens.

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='last'*)

Drop rows of duplicate indices in dataframe.

Be aware that this function changes the dataframe inplace. To avoid this behavior provide a copy of the dataframe to this function.

Parameters

- **`dataframe`** (pandas.DataFrame) – Dataframe to drop indices from.
- **`keep`** (*`str`*) – Indicator of whether to keep first ("first"), last ("last") or none (False) of the duplicated indices. See pandas.DataFrame.duplicated for more information. Default: "last".

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

Drop columns of dataframe that appear more than once.

Be aware that this function changes the dataframe inplace. To avoid this behavior provide a copy of the dataframe to this function.

Parameters

- **`df`** (pandas.DataFrame) – Dataframe to drop columns from.
- **`keep`** (*`str`*) – Indicator of whether to keep first ("first"), last ("last") or none (False) of the duplicated columns. See pandas.DataFrame.duplicated for more information. Default: "last".

`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.`**`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.`**`get_downstream_buses`**(*edisgo_obj*, *comp_name*, *comp_type='bus'*)

Returns all buses downstream (farther away from station) of the given bus or line.

In case a bus is given, returns all buses downstream of the given bus plus the given bus itself. In case a line is given, returns all buses downstream of the bus that is closer to the station (thus only one bus of the line is included in the returned buses).

Parameters

- **edisgo_obj** (*EDisGo object*) –
- **comp_name** (*str*) – Name of bus or line (as in index of `buses_df` or `lines_df`) to get downstream buses for.
- **comp_type** (*str*) – Can be either 'bus' or 'line'. Default: 'bus'.

Returns

List of buses (as in index of `buses_df`) downstream of the given component incl. the initial bus.

Return type

list(str)

`edisgo.tools.tools.`**`assign_voltage_level_to_component`**`(`*df*, *buses_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

- **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.
- **buses_df** (pandas.DataFrame) – Dataframe with bus information. Bus names are in the index. Only required column is column 'v_nom', giving the nominal voltage of the voltage level the bus is in.

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.`**`determine_grid_integration_voltage_level`**`(`*edisgo_object*, *power*`)`

Gives voltage level component should be integrated into based on its nominal power.

The voltage level is specified through an integer value from 4 to 7 with 4 = MV busbar, 5 = MV grid, 6 = LV busbar and 7 = LV grid.

The voltage level is determined using upper limits up to which capacity a component is integrated into a certain voltage level. These upper limits are set in the config section *grid_connection* through the parameters 'upper_limit_voltage_level_{4:7}'.

Parameters

- **edisgo_object** (*EDisGo*) –
- **power** (*float*) – Nominal power of component in MW.

Returns

Voltage level component should be integrated into. Possible options are 4 (MV busbar), 5 (MV grid), 6 (LV busbar) or 7 (LV grid).

Return type

int

`edisgo.tools.tools.`**`determine_bus_voltage_level`**`(`*edisgo_object*, *bus_name*`)`

Gives voltage level as integer from 4 to 7 of given bus.

The voltage level is specified through an integer value from 4 to 7 with 4 = MV busbar, 5 = MV grid, 6 = LV busbar and 7 = LV grid.

Buses that are directly connected to a station and not part of a longer feeder or half-ring, i.e. they are only part of one line, are as well considered as voltage level 4 or 6, depending on if they are connected to an HV/MV station or MV/LV station.

Parameters

- **edisgo_object** (*EDisGo*) –

- **bus_name** (*str*) – Name of bus as in index of *buses_df*.

Returns
Voltage level of bus. Possible options are 4 (MV busbar), 5 (MV grid), 6 (LV busbar) or 7 (LV grid).

Return type
int

`edisgo.tools.tools.`**`get_weather_cells_intersecting_with_grid_district`**(*edisgo_obj: edisgo.EDisGo*, *engine: sqlalchemy.engine.base.Engine | None = None*) → set

Get all weather cells that intersect with the grid district.

Parameters
- **edisgo_obj** (*EDisGo*) –
- **engine** (sqlalchemy.Engine) – Database engine. Only needed when using new egon_data data.

Returns
Set with weather cell IDs.

Return type
set(int)

`edisgo.tools.tools.`**`get_directory_size`**(*start_dir*)

Calculates the size of all files within the start path.

Walks through all files and sub-directories within a given directory and calculate the sum of size of all files in the directory. See also stackoverflow.

Parameters
start_dir (*str*) – Start path.

Returns
Size of the directory.

Return type
int

`edisgo.tools.tools.`**`get_files_recursive`**(*path*, *files=None*)

Recursive function to get all files in a given path and its sub directories.

Parameters
- **path** (*str*) – Directory to start from.
- **files** (*list, optional*) – List of files to start with. Default: None.

`edisgo.tools.tools.`**`calculate_impedance_for_parallel_components`**(*parallel_components*, *pu=False*)

Method to calculate parallel impedance and power of parallel elements.

`edisgo.tools.tools.`**`add_line_susceptance`**(*edisgo_obj*, *mode='mv_b'*)

Adds line susceptance information in Siemens to lines in existing grids.

Parameters
- **edisgo_obj** (*EDisGo*) – EDisGo object to which line susceptance information is added.
- **mode** (*str*) – Defines how the susceptance is added:
  - **'no_b'**
    Susceptance is set to 0 for all lines.
  - **'mv_b' (Default)**
    Susceptance is for the MV lines set according to the equipment parameters and for the LV lines it is set to zero.

- **'all_b'**
  Susceptance is for the MV lines set according to the equipment parameters and for the LV lines 0.25 uF/km is chosen.

Return type
EDisGo

`edisgo.tools.tools.`**`aggregate_district_heating_components`**(*edisgo_obj*, *feedin_district_heating=None*)

Aggregate PtH components that feed into the same district heating network.

Besides aggregating PtH components, feed-in from other heat supply sources can be specified, which is subtracted from the heat demand in the district heating network in order to determine the heat demand that needs to be covered by the PtH units.

Concerning the aggregated components, rated power of the single components is added up and COP for combined component is calculated from COP of all components weighed with their rated power. If active and reactive power time series were previously set for the PtH units they are overwritten.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **feedin_district_heating** (pandas.DataFrame) – Other thermal feed-in into district heating per district heating area (in columns) and time step (in index) in MW.

`edisgo.tools.tools.`**`reduce_timeseries_data_to_given_timeindex`**(*edisgo_obj*, *timeindex*, *freq='1H'*, *timeseries=True*, *electromobility=True*, *save_ev_soc_initial=True*, *heat_pump=True*, *dsm=True*, *overlying_grid=True*)

Reduces timeseries data in EDisGo object to given time index.

Parameters

- **edisgo_obj** (*EDisGo*) –
- **timeindex** (pandas.DatetimeIndex) – Time index to set.
- **freq** (str or pandas.Timedelta, optional) – Frequency of time series data. This is only needed if it cannot be inferred from the given *timeindex* and if electromobility data and/or overlying grid data is reduced, as the initial SoC is tried to be set using the time step before the first time step in the given *timeindex*. Offset aliases can be found here: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases. Default: '1H'.
- **timeseries** (*bool*) – Indicates whether timeseries in `TimeSeries` are reduced to given time index. Default: True.
- **electromobility** (*bool*) – Indicates whether timeseries in `Electromobility` are reduced to given time index. Default: True.
- **save_ev_soc_initial** (*bool*) – Indicates whether to save initial EV SOC from timestep before first timestep of given time index. Default: True.
- **heat_pump** (*bool*) – Indicates whether timeseries in `HeatPump` are reduced to given time index. Default: True.
- **dsm** (*bool*) – Indicates whether timeseries in `DSM` are reduced to given time index. Default: True.
- **overlying_grid** (*bool*) – Indicates whether timeseries in `OverlyingGrid` are reduced to given time index. Default: True.

`edisgo.tools.tools.`**`resample`**(*object*, *freq_orig*, *method: str = 'ffill'*, *freq: str | pandas.Timedelta = '15min'*, *attr_to_resample=None*)

Resamples time series data to a desired resolution.

Both up- and down-sampling methods are possible.

Parameters

- **object** (*TimeSeries* or *HeatPump*) – Object of which to resample time series data.
- **freq_orig** (pandas.Timedelta) – Frequency of original time series data.
- **method** (*str, optional*) – See *method* parameter in `resample_timeseries` for more information.
- **freq** (*str, optional*) – See *freq* parameter in `resample_timeseries` for more information.
- **attr_to_resample** (*list(str), optional*) – List of attributes to resample. Per default, all attributes specified in respective object's *_attributes* are resampled.

`edisgo.tools.tools.`**`reduce_memory_usage`**(*df: pandas.DataFrame*, *show_reduction: bool = False*) → pandas.DataFrame

Function to automatically check if columns of a pandas DataFrame can be reduced to a smaller data type.

Source: https://mikulskibartosz.name/how-to-reduce-memory-usage-in-pandas

Parameters

- **df** (pandas.DataFrame) – DataFrame to reduce memory usage for.
- **show_reduction** (*bool*) – If True, print amount of memory reduced.

Returns

DataFrame with decreased memory usage.

Return type

pandas.DataFrame

`edisgo.tools.tools.`**`get_year_based_on_timeindex`**(*edisgo_obj*)

Checks if `timeindex` is already set and if so, returns the year of the time index.

Parameters

edisgo_object (*EDisGo*) –

Returns

If a time index is available returns the year of the time index, otherwise it returns None.

Return type

int or None

`edisgo.tools.tools.`**`get_year_based_on_scenario`**(*scenario*)

Returns the year the given scenario was set up for.

Parameters

scenario (*str*) – Scenario for which to set year. Possible options are 'eGon2035' and 'eGon100RE'.

Returns

Returns the year of the scenario (2035 in case of the 'eGon2035' scenario and 2045 in case of the 'eGon100RE' scenario). If another scenario name is provided it returns None.

Return type

int or None

`edisgo.tools.tools.`**`hash_dataframe`**(*df: pandas.DataFrame*) → str

Get hash of dataframe.

Can be used to check if dataframes have the same content.

Parameters

df (pandas.DataFrame) – DataFrame to hash.

Returns

Hash of dataframe as string.

Return type

str

Package Contents

Functions

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

edisgo.tools.**session_scope**()

Function to ensure that sessions are closed properly.

Submodules

edisgo.edisgo

Module Contents

Classes

EDisGo	Provides the top-level API for invocation of data import, power flow

Functions

import_edisgo_from_pickle(filename[, path])	Restores EDisGo object from pickle file.
import_edisgo_from_files(edisgo_path[, ...])	Sets up EDisGo object from csv files.

class edisgo.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.
- **timeindex** (None or pandas.DatetimeIndex, optional) – Defines the time steps feed-in and demand time series of all generators, loads and storage units need to be set. The time index is for example used as default for time steps considered in the power flow analysis and when checking the integrity of the network. Providing a time index is only optional in case a worst case analysis is set up using *set_time_series_worst_case_analysis()*. In all other cases a time index needs to be set manually.
- **config_path** (*None or str or dict*) – Path to the config directory. Options are:

- **'default' (default)**

  If *config_path* is set to 'default', the provided default config files are used directly.

- **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 options, the directories and config files must exist and are not automatically created.

- **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.

Default: "default".

• `legacy_ding0_grids` (*bool*) – Allow import of old ding0 grids. Default: True.

`topology`

The topology is a container object holding the topology of the grids including buses, lines, transformers, switches and components connected to the grid including generators, loads and storage units.

Type

Topology

`timeseries`

Container for active and reactive power time series of generators, loads and storage units.

Type

TimeSeries

`results`

This is a container holding all calculation results from power flow analyses and grid reinforcement.

Type

Results

`electromobility`

This class holds data on charging processes (how long cars are parking at a charging station, how much they need to charge, etc.) necessary to apply different charging strategies, as well as information on potential charging sites and integrated charging parks.

Type

Electromobility

`heat_pump`

This is a container holding heat pump data such as COP, heat demand to be served and heat storage information.

Type
HeatPump

overlying_grid

This is a container holding data from the overlying grid such as curtailment requirements or power plant dispatch.

Type
OverlyingGrid

dsm

This is a container holding data on demand side management potential.

Type
DSM

property config

eDisGo configuration data.

Parameters
kwargs (*dict*) – Dictionary with keyword arguments to set up Config object. See parameters of *Config* class for more information on possible input parameters.

Returns
Config object with configuration data from config files.

Return type
Config

import_ding0_grid(*path*, *legacy_ding0_grids=True*)

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.
- **legacy_ding0_grids** (*bool*) – Allow import of old ding0 grids. Default: True.

set_timeindex(*timeindex*)

Sets *timeindex* all time-dependent attributes are indexed by.

The time index is for example used as default for time steps considered in the power flow analysis and when checking the integrity of the network.

Parameters
timeindex (pandas.DatetimeIndex) – Time index to set.

set_time_series_manual(*generators_p=None*, *loads_p=None*, *storage_units_p=None*, *generators_q=None*, *loads_q=None*, *storage_units_q=None*)

Sets given component time series.

If time series for a component were already set before, they are overwritten.

Parameters

- **generators_p** (pandas.DataFrame) – Active power time series in MW of generators. Index of the data frame is a datetime index. Columns contain generators names of generators to set time series for. Default: None.
- **loads_p** (pandas.DataFrame) – Active power time series in MW of loads. Index of the data frame is a datetime index. Columns contain load names of loads to set time series for. Default: None.
- **storage_units_p** (pandas.DataFrame) – Active power time series in MW of storage units. Index of the data frame is a datetime index. Columns contain storage unit names of storage units to set time series for. Default: None.

- **generators_q** (pandas.DataFrame) – Reactive power time series in MVA of generators. Index of the data frame is a datetime index. Columns contain generators names of generators to set time series for. Default: None.
- **loads_q** (pandas.DataFrame) – Reactive power time series in MVA of loads. Index of the data frame is a datetime index. Columns contain load names of loads to set time series for. Default: None.
- **storage_units_q** (pandas.DataFrame) – Reactive power time series in MVA of storage units. Index of the data frame is a datetime index. Columns contain storage unit names of storage units to set time series for. Default: None.

Notes

This function raises a warning in case a time index was not previously set. You can set the time index upon initialisation of the EDisGo object by providing the input parameter 'timeindex' or using the function `set_timeindex`. Also make sure that the time steps for which time series are provided include the set time index.

set_time_series_worst_case_analysis(*cases=None*, *generators_names=None*, *loads_names=None*, *storage_units_names=None*)

Sets demand and feed-in of all loads, generators and storage units for the specified worst cases.

See `set_worst_case` for more information.

Parameters

- **cases** (*str or list(str)*) – List with worst-cases to generate time series for. Can be 'feed-in_case', 'load_case' or both. Defaults to None in which case both 'feed-in_case' and 'load_case' are set up.
- **generators_names** (*list(str)*) – Defines for which generators to set worst case time series. If None, time series are set for all generators. Default: None.
- **loads_names** (*list(str)*) – Defines for which loads to set worst case time series. If None, time series are set for all loads. Default: None.
- **storage_units_names** (*list(str)*) – Defines for which storage units to set worst case time series. If None, time series are set for all storage units. Default: None.

set_time_series_active_power_predefined(*fluctuating_generators_ts=None*, *fluctuating_generators_names=None*, *dispatchable_generators_ts=None*, *dispatchable_generators_names=None*, *conventional_loads_ts=None*, *conventional_loads_names=None*, *charging_points_ts=None*, *charging_points_names=None*, ***kwargs*)

Uses predefined feed-in or demand profiles to set active power time series.

Predefined profiles comprise i.e. standard electric conventional load profiles for different sectors generated using the oemof demandlib or feed-in time series of fluctuating solar and wind generators provided on the OpenEnergy DataBase. This function can also be used to provide your own profiles per technology or load sector.

The active power time series are written to `generators_active_power` or `loads_active_power`. As data in `TimeSeries` is indexed by `timeindex` it is better to set `timeindex` before calling this function. You can set the time index upon initialisation of the EDisGo object by providing the input parameter 'timeindex' or using the function *set_timeindex*. Also make sure that the time steps of self-provided time series include the set time index.

Parameters

- **fluctuating_generators_ts** (str or pandas.DataFrame or None) – Defines option to set technology-specific or technology- and weather cell specific active power time series of wind and solar generators. Possible options are:
  - 'oedb'

    Technology- and weather cell-specific hourly feed-in time series are obtained from the OpenEnergy DataBase. See *edisgo.io.timeseries_import.feedin_oedb()* for more information.

    This option requires that the parameter *engine* is provided in case new ding0 grids with geo-referenced LV grids are used. For further settings, the parameter *timeindex* can also be provided.
  - pandas.DataFrame

    DataFrame with self-provided feed-in time series per technology or per technology and weather cell ID normalized to a nominal capacity of 1. In case time series are provided only by technology, columns of the DataFrame contain the technology type as string. In case time series are provided by technology and weather cell ID columns need to be a pandas.MultiIndex with the first level containing the technology as string and the second level the weather cell ID as integer. Index needs to be a pandas.DatetimeIndex.

    When importing a ding0 grid and/or using predefined scenarios of the future generator park, 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.
  - None

    If None, time series are not set.

  Default: None.
- **fluctuating_generators_names** (*list(str) or None*) – Defines for which fluctuating generators to apply technology-specific time series. See parameter *generator_names* in *predefined_fluctuating_generators_by_technology* for more information. Default: None.
- **dispatchable_generators_ts** (pandas.DataFrame or None) – Defines which technology-specific time series to use to set active power time series of dispatchable generators. See parameter *ts_generators* in *predefined_dispatchable_generators_by_technology* for more information. If None, no time series of dispatchable generators are set. Default: None.
- **dispatchable_generators_names** (*list(str) or None*) – Defines for which dispatchable generators to apply technology-specific time series. See parameter *generator_names* in

predefined_dispatchable_generators_by_technology for more information. Default: None.

- **conventional_loads_ts** (str or pandas.DataFrame or None) – Defines option to set active power time series of conventional loads. Possible options are:
  - 'oedb'

    Sets active power demand time series using individual hourly electricity load time series for one year obtained from the OpenEnergy DataBase.

    This option requires that the parameters *engine* and *scenario* are provided. For further settings, the parameter *timeindex* can also be provided.
  - 'demandlib'

    Sets active power demand time series using hourly electricity load time series obtained using standard electric load profiles from the oemof demandlib. The demandlib provides sector-specific time series for the sectors 'residential', 'cts', 'industrial', and 'agricultural'.

    For further settings, the parameter *timeindex* can also be provided.
  - pandas.DataFrame

    Sets active power demand time series using sector-specific demand time series provided in this DataFrame. The load time series per sector need to be normalized to an annual consumption of 1. Index needs to be a pandas.DatetimeIndex. Columns need to contain the sector as string. In the current grid existing load types can be retrieved from column *sector* in *loads_df* (make sure to select *type* 'conventional_load'). In ding0 grids the differentiated sectors are 'residential', 'cts', and 'industrial'.
  - None

    If None, conventional load time series are not set.

  Default: None.
- **conventional_loads_names** (*list(str) or None*) – Defines for which conventional loads to set time series. In case *conventional_loads_ts* is 'oedb' see parameter *load_names* in *edisgo.io.timeseries_import.electricity_demand_oedb()* for more information. For other cases see parameter *load_names* in *predefined_conventional_loads_by_sector* for more information. Default: None.
- **charging_points_ts** (pandas.DataFrame or None) – Defines which use-case-specific time series to use to set active power time series of charging points. See parameter *ts_loads* in *predefined_charging_points_by_use_case* for more information. If None, no time series of charging points are set. Default: None.
- **charging_points_names** (*list(str) or None*) – Defines for which charging points to apply use-case-specific time series. See parameter *load_names* in *predefined_charging_points_by_use_case* for more information. Default: None.

- **engine** (sqlalchemy.Engine) – Database engine. This parameter is only required in case *conventional_loads_ts* or *fluctuating_generators_ts* is 'oedb'.
- **scenario** (*str*) – Scenario for which to retrieve demand data. Possible options are 'eGon2035' and 'eGon100RE'. This parameter is only required in case *conventional_loads_ts* is 'oedb'.
- **timeindex** (pandas.DatetimeIndex or None) – This parameter can optionally be provided in case *conventional_loads_ts* is 'oedb' or 'demandlib' and in case *fluctuating_generators_ts* is 'oedb'. It is used to specify time steps for which to set active power data. Leap years can currently not be handled when data is retrieved from the oedb. In case the given timeindex contains a leap year, the data will be indexed using a default year and set for the whole year. If no timeindex is provided, the timeindex set in `timeindex` is used. If `timeindex` is not set, the data is indexed using a default year and set for the whole year.

set_time_series_reactive_power_control(*control='fixed_cosphi'*, *generators_parametrisation='default'*, *loads_parametrisation='default'*, *storage_units_parametrisation='default'*)

Set reactive power time series of components.

Parameters

- **control** (*str*) – Type of reactive power control to apply. Currently, the only option is 'fixed_coshpi'. See `fixed_cosphi` for further information.
- **generators_parametrisation** (str or pandas.DataFrame) – See parameter *generators_parametrisation* in `fixed_cosphi` for further information. Here, per default, the option 'default' is used.
- **loads_parametrisation** (str or pandas.DataFrame) – See parameter *loads_parametrisation* in `fixed_cosphi` for further information. Here, per default, the option 'default' is used.
- **storage_units_parametrisation** (str or pandas.DataFrame) – See parameter *storage_units_parametrisation* in `fixed_cosphi` for further information. Here, per default, the option 'default' is used.

Notes

Be careful to set parametrisation of other component types to None if you only want to set reactive power of certain components. See example below for further information.

Examples

To only set reactive power time series of one generator using default configurations you can do the following:

```
>>> self.set_time_series_reactive_power_control(
>>>     generators_parametrisation=pd.DataFrame(
>>>         {
>>>             "components": [["Generator_1"]],
>>>             "mode": ["default"],
>>>             "power_factor": ["default"],
>>>         },
```

(continues on next page)

(continued from previous page)

```
>>>         index=[1],
>>>     ),
>>>     loads_parametrisation=None,
>>>     storage_units_parametrisation=None
>>> )
```

In the example above, *loads_parametrisation* and *storage_units_parametrisation* need to be set to None, otherwise already existing time series would be overwritten.

To only change configuration of one load and for all other components use default configurations you can do the following:

```
>>> self.set_time_series_reactive_power_control(
>>>     loads_parametrisation=pd.DataFrame(
>>>         {
>>>             "components": [["Load_1"],
>>>                            self.topology.loads_df.index.drop(["Load_1"])],
>>>             "mode": ["capacitive", "default"],
>>>             "power_factor": [0.98, "default"],
>>>         },
>>>         index=[1, 2],
>>>     )
>>> )
```

In the example above, *generators_parametrisation* and *storage_units_parametrisation* do not need to be set as default configurations are per default used for all generators and storage units anyway.

to_pypsa(*mode=None*, *timesteps=None*, *check_edisgo_integrity=False*, ***kwargs*)

Convert grid to PyPSA.Network representation.

You can choose between translation of the MV and all underlying LV grids (mode=None (default)), the MV network only (mode='mv' or mode='mvlv') or a single LV network (mode='lv').

Parameters

- **mode** (*str*) – Determines network levels that are translated to PyPSA.Network. Possible options are:
  - None

    MV and underlying LV networks are exported. This is the default.
  - 'mv'

    Only MV network is exported. MV/LV transformers are not exported in this mode. Loads, generators and storage units in underlying LV grids are connected to the respective MV/LV station's primary side. Per default, they are all connected separately, but you can also choose to aggregate them. See parameters *aggregate_loads*, *aggregate_generators* and *aggregate_storages* for more information.
  - 'mvlv'

    This mode works similar as mode 'mv', with the difference that MV/LV transformers are as well exported and LV components connected to the respective MV/LV station's secondary side. Per default, all components are connected separately, but you can also choose to aggregate them. See parameters *aggregate_loads*, *aggregate_generators* and *aggregate_storages* for more information.

  – 'lv'

    Single LV network topology including the MV/LV transformer is exported. The LV grid to export is specified through the parameter *lv_grid_id*. The slack is positioned at the secondary side of the MV/LV station.

- **timesteps** (pandas.DatetimeIndex or pandas.Timestamp) – Specifies which time steps to export to pypsa representation to e.g. later on use in power flow analysis. It defaults to None in which case all time steps in `timeindex` are used. Default: None.
- **check_edisgo_integrity** (*bool*) – Check integrity of edisgo object before translating to pypsa. This option is meant to help the identification of possible sources of errors if the power flow calculations fail. See `check_integrity` for more information. Default: False.
- **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.
- **lv_grid_id** (*int or str*) – ID (e.g. 1) or name (string representation, e.g. "LVGrid_1") of LV grid to export in case mode is 'lv'. Default: None.
- **aggregate_loads** (*str*) – Mode for load aggregation in LV grids in case mode is 'mv' or 'mvlv'. Can be 'sectoral' aggregating the loads sector-wise, 'all' aggregating all loads into one or None, not aggregating loads but appending them to the station one by one. Default: None.
- **aggregate_generators** (*str*) – Mode for generator aggregation in LV grids in case mode is 'mv' or 'mvlv'. Can be 'type' aggregating generators per generator type, 'curtailable' aggregating 'solar' and 'wind' generators into one and all other generators into another one, or None, where no aggregation is undertaken and generators are added to the station one by one. Default: None.
- **aggregate_storages** (*str*) – Mode for storage unit aggregation in LV grids in case mode is 'mv' or 'mvlv'. Can be 'all' where all storage units in an LV grid are aggregated to one storage unit or None, in which case no aggregation is conducted and storage units are added to the station. Default: None.

Returns
PyPSA.Network representation.

Return type
PyPSA.Network

to_powermodels(*s_base=1*, *flexible_cps=None*, *flexible_hps=None*, *flexible_loads=None*, *flexible_storage_units=None*, *opf_version=1*)

Convert eDisGo representation of the network topology and timeseries to PowerModels network data format.

Parameters

- **s_base** (*int*) – Base value of apparent power for per unit system. Default: 1 MVA
- **flexible_cps** (numpy.ndarray or None) – Array containing all charging points that allow for flexible charging.

- **flexible_hps** (numpy.ndarray or None) – Array containing all heat pumps that allow for flexible operation due to an attached heat storage.
- **flexible_loads** (numpy.ndarray or None) – Array containing all flexible loads that allow for application of demand side management strategy.
- **flexible_storage_units** (numpy.ndarray or None) – Array containing all flexible storages. Non-flexible storages operate to optimize self consumption. Default: None.
- **opf_version** (*int*) – Version of optimization models to choose from. Must be one of [1, 2, 3, 4]. For more information see `edisgo.opf.powermodels_opf.pm_optimize()`. Default: 1.

Returns
Dictionary that contains all network data in PowerModels network data format.

Return type
dict

pm_optimize(*s_base=1*, *flexible_cps=None*, *flexible_hps=None*, *flexible_loads=None*, *flexible_storage_units=None*, *opf_version=1*, *method='soc'*, *warm_start=False*, *silence_moi=False*, *save_heat_storage=True*, *save_slack_gen=True*, *save_slacks=True*)

Run OPF in julia subprocess and write results of OPF back to edisgo object.

Results of OPF are time series of operation schedules of flexibilities.

Parameters

- **s_base** (*int*) – Base value of apparent power for per unit system. Default: 1 MVA.
- **flexible_cps** (numpy.ndarray or None) – Array containing all charging points that allow for flexible charging. Default: None.
- **flexible_hps** (numpy.ndarray or None) – Array containing all heat pumps that allow for flexible operation due to an attached heat storage. Default: None.
- **flexible_loads** (numpy.ndarray or None) – Array containing all flexible loads that allow for application of demand side management strategy. Default: None.
- **flexible_storage_units** (numpy.ndarray or None) – Array containing all flexible storages. Non-flexible storages operate to optimize self consumption. Default: None.
- **opf_version** (*int*) – Version of optimization models to choose from. Must be one of [1, 2, 3, 4]. For more information see `edisgo.opf.powermodels_opf.pm_optimize()`. Default: 1.
- **method** (*str*) – Optimization method to use. Must be either “soc” (Second Order Cone) or “nc” (Non Convex). For more information see `edisgo.opf.powermodels_opf.pm_optimize()`. Default: “soc”.
- **warm_start** (*bool*) – If set to True and if method is set to “soc”, non-convex IPOPT OPF will be run additionally and will be warm started with Gurobi SOC solution. Warm-start will only be run if results for Gurobi’s SOC relaxation is exact. Default: False.
- **silence_moi** (*bool*) – If set to True, MathOptInterface’s optimizer attribute “MOI.Silent” is set to True in julia subprocess. This attribute is for silencing the output of an optimizer. When set to True, it requires the solver to produce no output, hence there will be no logging coming from julia subprocess in python process. Default: False.

`to_graph()`

Returns networkx 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 generators into grid.

The generator data is retrieved from the open energy platform. Decommissioned generators are removed from the grid, generators with changed capacity updated and new generators newly integrated into the grid.

In case you are using new ding0 grids, where the LV is geo-referenced, the supported data source is scenario data generated in the research project eGo^n. You can choose between two scenarios: ‘eGon2035’ and ‘eGon100RE’. For more information on database tables used and how generator park is adapted see `oedb()`.

In case you are using old ding0 grids, where the LV is not geo-referenced, the 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. For more information on database tables used and how generator park is adapted see `oedb_legacy()`.

After the generator park is adapted there may be grid issues due to the additional feed-in. 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. In case you are using new ding0 grids, where the LV is geo-referenced, possible options are ‘eGon2035’ and ‘eGon100RE’. In case you are using old ding0 grids, where the LV is not geo-referenced, possible options are ‘nep2035’ and ‘ego100’.
- **kwargs** – In case you are using new ding0 grids, where the LV is geo-referenced, a database engine needs to be provided through keyword argument *engine*. In case you are using old ding0 grids, where the LV is not geo-referenced, you can check `edisgo.io.generators_import.oedb_legacy()` for possible keyword arguments.

`analyze`(*mode: str | None = None*, *timesteps: pandas.Timestamp | pandas.DatetimeIndex | None = None*, *raise_not_converged: bool = True*, *troubleshooting_mode: str | None = None*, *range_start: numbers.Number = 0.1*, *range_num: int = 10*, *scale_timeseries: float | None = None*, ***kwargs*) → tuple[pandas.DatetimeIndex, pandas.DatetimeIndex]

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).

Parameters

- **mode** (`str or None`) – Allows to toggle between power flow analysis for the whole network or just the MV or one LV grid. Possible options are:
  - None (default)

    Power flow analysis is conducted for the whole network including MV grid and underlying LV grids.
  - ’mv’

Power flow analysis is conducted for the MV level only. LV loads, generators and storage units are aggregated at the respective MV/LV stations' primary side. Per default, they are all connected separately, but you can also choose to aggregate them. See parameters *aggregate_loads*, *aggregate_generators* and *aggregate_storages* in `to_pypsa` for more information.

  - 'mvlv'

    Power flow analysis is conducted for the MV level only. In contrast to mode 'mv' LV loads, generators and storage units are in this case aggregated at the respective MV/LV stations' secondary side. Per default, they are all connected separately, but you can also choose to aggregate them. See parameters *aggregate_loads*, *aggregate_generators* and *aggregate_storages* in `to_pypsa` for more information.

  - 'lv'

    Power flow analysis is conducted for one LV grid only. ID or name of the LV grid to conduct power flow analysis for needs to be provided through keyword argument 'lv_grid_id' as integer or string. See parameter *lv_grid_id* in `to_pypsa` for more information. The slack is positioned at the secondary side of the MV/LV station.

- **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 all time steps in `timeindex` are used.
- **raise_not_converged** (*bool*) – If True, an error is raised in case power flow analysis did not converge for all time steps. Default: True.
- **troubleshooting_mode** (*str or None*) – Two optional troubleshooting methods in case of non-convergence of nonlinear power flow (cf. [1])
  - **None (default)**
    Power flow analysis is conducted using nonlinear power flow method.
  - **'lpf'**
    Non-linear power flow initial guess is seeded with the voltage angles from the linear power flow.
  - **'iteration'**
    Power flow analysis is conducted by reducing all power values of generators and loads to a fraction, e.g. 10%, solving the load flow and using it as a seed for the power at 20%, iteratively up to 100%. Using parameters *range_start* and *range_num* you can define at what scaling factor the iteration should start and how many iterations should be conducted.
- **range_start** (*float, optional*) – Specifies the minimum fraction that power values are set to when using *troubleshooting_mode* 'iteration'. Must be between 0 and 1. Default: 0.1.
- **range_num** (*int, optional*) – Specifies the number of fraction samples to generate when using *troubleshooting_mode* 'iteration'. Must be non-negative. Default: 10.
- **scale_timeseries** (*float or None, optional*) – If a value is given, the timeseries in the pypsa object are scaled with this factor (values between 0 and

1 will scale down the time series and values above 1 will scale the timeseries up). Downscaling of time series can be used to check if power flow converges for smaller grid loads. If None, timeseries are not scaled. In case of *troubleshooting_mode* 'iteration' this parameter is ignored. Default: None.

- **kwargs** (*dict*) – Possible other parameters comprise all other parameters that can be set in `edisgo.io.pypsa_io.to_pypsa()`.

Returns
First index contains time steps for which power flow analysis did converge. Second index contains time steps for which power flow analysis did not converge.

Return type
tuple(pandas.DatetimeIndex, pandas.DatetimeIndex)

References

[1] https://pypsa.readthedocs.io/en/latest/troubleshooting.html

reinforce(*timesteps_pfa: str | pandas.DatetimeIndex | pandas.Timestamp | None = None*, *reduced_analysis: bool = False*, *copy_grid: bool = False*, *max_while_iterations: int = 20*, *split_voltage_band: bool = True*, *mode: str | None = None*, *without_generator_import: bool = False*, *n_minus_one: bool = False*, *catch_convergence_problems: bool = False*, ***kwargs*) → *edisgo.network.results.Results*

Reinforces the network and calculates network expansion costs.

If the `edisgo.network.timeseries.TimeSeries.is_worst_case` is True input for *timesteps_pfa* is overwritten and therefore ignored.

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

Parameters

- **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 `timeindex` are used. Possible options are:
  - None Time steps in `timeindex` 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.
- **reduced_analysis** (*bool*) – If True, reinforcement is conducted for all time steps at which at least one branch shows its highest overloading or one bus shows its highest voltage violation. Time steps to consider are specified through parameter *timesteps_pfa*. If False, all time steps in parameter *timesteps_pfa* are used. Default: False.
- **copy_grid** (*bool*) – 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. Default: 20.
- **split_voltage_band** (*bool*) – If True the allowed voltage band of +/-10 percent is allocated to the different voltage levels MV, MV/LV and LV according to config values set in section *grid_expansion_allowed_voltage_deviations*. If False, the same voltage limits are used for all voltage levels. Be aware that this does currently not work correctly. Default: True.
- **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 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.
  - 'lv' to reinforce LV networks. In case an LV grid is specified through parameter *lv_grid_id*, the grid's MV/LV station is not included. In case no LV grid ID is given, all MV/LV stations are included.
- **without_generator_import** (*bool*) – If True, excludes lines that were added in the generator import to connect new generators from calculation of network expansion costs. Default: False.
- **n_minus_one** (*bool*) – Determines whether n-1 security should be checked. Currently, n-1 security cannot be handled correctly, wherefore the case where this parameter is set to True will lead to an error being raised. Default: False.
- **catch_convergence_problems** (*bool*) – Uses reinforcement strategy to reinforce not converging grid. Reinforces first with only converging timesteps. Reinforce again with at start not converging timesteps. If still not converging, scale timeseries. Default: False
- **is_worst_case** (*bool*) – Is used to overwrite the return value from *edisgo.network.timeseries.TimeSeries.is_worst_case*. If True, reinforcement is calculated for worst-case MV and LV cases separately.
- **lv_grid_id** (*str or int or None*) – LV grid id to specify the grid to check, if mode is "lv". If no grid is specified, all LV grids are checked. In that case, the power flow analysis is conducted including the MV grid, in order to check loading and voltage drop/rise of MV/LV stations.
- **skip_mv_reinforcement** (*bool*) – If True, MV is not reinforced, even if *mode* is "mv", "mvlv" or None. This is used in case worst-case grid reinforcement is conducted in order to reinforce MV/LV stations for LV worst-cases. Default: False.
- **num_steps_loading** (*int*) – In case *reduced_analysis* is set to True, this parameter can be used to specify the number of most critical overloading events to consider. If None, *percentage* is used. Default: None.
- **num_steps_voltage** (*int*) – In case *reduced_analysis* is set to True, this parameter can be used to specify the number of most critical voltage issues to select. If None, *percentage* is used. Default: None.

- **percentage** (*float*) – In case *reduced_analysis* is set to True, this parameter can be used to specify the percentage of most critical time steps to select. The default is 1.0, in which case all most critical time steps are selected. Default: 1.0.
- **use_troubleshooting_mode** (*bool*) – In case *reduced_analysis* is set to True, this parameter can be used to specify how to handle non-convergence issues when determining the most critical time steps. If set to True, non-convergence issues are tried to be circumvented by reducing load and feed-in until the power flow converges. The most critical time steps are then determined based on the power flow results with the reduced load and feed-in. If False, an error will be raised in case time steps do not converge. Setting this to True doesn't make sense for the grid reinforcement as the troubleshooting mode is only used when determining the most critical time steps not when running a power flow analysis to determine grid reinforcement needs. To handle non-convergence in the grid reinforcement set parameter *catch_convergence_problems* to True. Default: False.
- **run_initial_analyze** (*bool*) – In case *reduced_analysis* is set to True, this parameter can be used to specify whether to run an initial analyze to determine most critical time steps or to use existing results. If set to False, *use_troubleshooting_mode* is ignored. Default: True.
- **weight_by_costs** (*bool*) – In case *reduced_analysis* is set to True, this parameter can be used to specify whether to weight time steps by estimated grid expansion costs. See parameter *weight_by_costs* in `get_most_critical_time_steps()` for more information. Default: False.

Returns

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

Return type

`Results`

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

Adds single component to network.

Components can be lines or buses as well as generators, loads, or storage units. If add_ts is set to True, time series of elements are set as well. Currently, time series need to be provided.

Parameters

- **comp_type** (*str*) – Type of added component. Can be 'bus', 'line', 'load', 'generator', or 'storage_unit'.
- **ts_active_power** (pandas.Series or None) – 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. Defaults to None in which case no time series is set.
- **ts_reactive_power** (pandas.Series or str or None) – Possible options are:
  - 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.
  - "default"

    Reactive power time series is determined based on assumptions on fixed power factor of the component. To this end, the power factors set in the config section *reactive_power_factor* and the power factor mode, defining whether components behave inductive or capacitive, given in the config section *reactive_power_mode*, are used. This option requires you to provide an active power time series. In case it was not provided, reactive power cannot be set and a warning is raised.

  – None

    No reactive power time series is set.

  Default: None

- **`**kwargs`** (*`dict`*) – Attributes of added component. See respective functions for required entries.
  - 'bus' : *`add_bus`*
  - 'line' : *`add_line`*
  - 'load' : *`add_load`*
  - 'generator' : *`add_generator`*
  - 'storage_unit' : *`add_storage_unit`*

Returns

The identifier of the newly integrated component as in index of *`generators_df`*, *`loads_df`*, etc., depending on component type.

Return type

str

`integrate_component_based_on_geolocation`(*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, charging points, heat pumps and storage units.

See *`connect_to_mv`*, *`connect_to_lv`* and *`connect_to_lv_based_on_geolocation`* for more information.

Parameters

- **`comp_type`** (*`str`*) – Type of added component. Can be 'generator', 'charging_point', 'heat_pump' or 'storage_unit'.
- **`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* or *p_set* (given as kwarg). For this, upper limits up to which capacity a component is integrated into a certain voltage level (set in the config section *grid_connection* through the parameters 'upper_limit_voltage_level_{4:7}') are used.
- **`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. 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. If you want to add time series (if *add_ts* is True), you must provide a time series. It is not automatically retrieved.

- **kwargs** – Attributes of added component. See `add_generator`, `add_storage_unit` respectively `add_load` methods for more information on required and optional parameters.

Returns
The identifier of the newly integrated component as in index of `generators_df`, `loads_df` or `storage_units_df`, depending on component type.

Return type
str

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

Removes single component from network.

Components can be lines or buses as well as generators, loads, or storage units. If drop_ts is set to True, time series of elements are deleted as well.

Parameters

- **comp_type** (*str*) – Type of removed component. Can be 'bus', 'line', 'load', 'generator', or 'storage_unit'.
- **comp_name** (*str*) – Name of component to be removed.
- **drop_ts** (*bool*) – Indicator if time series for component are removed as well. Defaults to True.

aggregate_components(*aggregate_generators_by_cols=None*, *aggregate_loads_by_cols=None*)

Aggregates generators and loads at the same bus.

By default all generators respectively loads at the same bus are aggregated. You can specify further columns to consider in the aggregation, such as the generator type or the load sector. Make sure to always include the bus in the list of columns to aggregate by, as otherwise the topology would change.

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

Parameters

- **aggregate_generators_by_cols** (*list(str) or None*) – List of columns to aggregate generators at the same bus by. Valid columns are all columns in `generators_df`. If an empty list is given, generators are not aggregated. Defaults to None, in which case all generators at the same bus are aggregated.
- **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`. If an empty list is given, generators are not aggregated. Defaults to None, in which case all loads at the same bus are aggregated.

import_electromobility(*data_source: str*, *scenario: str = None*, *engine: sqlalchemy.engine.base.Engine = None*, *charging_processes_dir: pathlib.PurePath | str = None*, *potential_charging_points_dir: pathlib.PurePath | str = None*, *import_electromobility_data_kwds=None*, *allocate_charging_demand_kwds=None*)

Imports electromobility data and integrates charging points into grid.

Electromobility data can be obtained from the OpenEnergy DataBase or from self-provided data. In case you want to use self-provided data, it needs to be generated using the tools SimBEV (required version: 3083c5a) and TracBEV (required version: 14d864c). SimBEV provides data on standing times, charging demand, etc. per vehicle, whereas TracBEV provides potential charging point locations.

After electromobility data is loaded, the charging demand from is allocated to potential charging points. Afterwards, all potential charging points with charging demand allocated to them are integrated into the grid.

Be aware that this function does not yield charging time series per charging point but only charging processes (see `charging_processes_df` for more information). The actual charging time series are determined through applying a charging strategy using the function `charging_strategy`.

Parameters

- **`data_source`** (*str*) – Specifies source from where to obtain electromobility data. Possible options are:
  - ”oedb”

    Electromobility data is obtained from the OpenEnergy DataBase.

    This option requires that the parameters *scenario* and *engine* are provided.
  - ”directory”

    Electromobility data is obtained from directories specified through parameters *charging_processes_dir* and *potential_charging_points_dir*.
- **`scenario`** (*str*) – Scenario for which to retrieve electromobility data in case *data_source* is set to “oedb”. Possible options are “eGon2035” and “eGon100RE”.
- **`engine`** (sqlalchemy.Engine) – Database engine. Needs to be provided in case *data_source* is set to “oedb”.
- **`charging_processes_dir`** (*str or pathlib.PurePath*) – Directory holding data on charging processes (standing times, charging demand, etc. per vehicle), including metadata, from SimBEV.
- **`potential_charging_points_dir`** (*str or pathlib.PurePath*) – Directory holding data on potential charging point locations from TracBEV.
- **`import_electromobility_data_kwds`** (*dict*) – These may contain any further attributes you want to specify when importing electromobility data.

  gc_to_car_rate_home
  [float] Specifies the minimum rate between potential charging points for the use case “home” and the total number of cars. Default: 0.5.

  gc_to_car_rate_work
  [float] Specifies the minimum rate between potential charging points for the use case “work” and the total number of cars. Default: 0.25.

  gc_to_car_rate_public
  [float] Specifies the minimum rate between potential charging points for the use case “public” and the total number of cars. Default: 0.1.

  gc_to_car_rate_hpc
  [float] Specifies the minimum rate between potential charging points for the use case “hpc” and the total number of cars. Default: 0.005.

  mode_parking_times
  [str] If the mode_parking_times is set to “frugal” only parking times with any charging demand are imported. Any other input will lead to all parking and driving events being imported. Default: “frugal”.

charging_processes_dir
[str] Charging processes sub-directory. Only used when *data_source* is set to "directory". Default: None.

simbev_config_file
[str] Name of the simbev config file. Only used when *data_source* is set to "directory". Default: "metadata_simbev_run.json".

- **allocate_charging_demand_kwds** – These may contain any further attributes you want to specify when calling the function `distribute_charging_demand()` that allocates charging processes to potential charging points.

  mode
  [str] Distribution mode. If the mode is set to "user_friendly" only the simbev weights are used for the distribution. If the mode is "grid_friendly" also grid conditions are respected. Default: "user_friendly".

  generators_weight_factor
  [float] Weighting factor of the generators weight within an LV grid in comparison to the loads weight. Default: 0.5.

  distance_weight
  [float] Weighting factor for the distance between a potential charging park and its nearest substation in comparison to the combination of the generators and load factors of the LV grids. Default: 1 / 3.

  user_friendly_weight
  [float] Weighting factor of the user-friendly weight in comparison to the grid friendly weight. Default: 0.5.

apply_charging_strategy(*strategy='dumb'*, ***kwargs*)

Applies charging strategy to set EV charging time series at charging parks.

This function requires that standing times, charging demand, etc. at charging parks were previously set using `import_electromobility`.

It is assumed that only 'private' charging processes at 'home' or at 'work' can be flexibilized. 'public' charging processes will always be 'dumb'.

The charging time series at each charging parks are written to `loads_active_power`. Reactive power in `loads_reactive_power` is set to 0 Mvar.

Parameters

- **strategy** (`str`) – Defines the charging strategy to apply. The following charging strategies are valid:
  - 'dumb'

    The cars are charged directly after arrival with the maximum possible charging capacity.
  - 'reduced'

    The cars are charged directly after arrival with the minimum possible charging power. The minimum possible charging power is determined by the parking time and the parameter *minimum_charging_capacity_factor*.
  - 'residual'

The cars are charged when the residual load in the MV grid is lowest (high generation and low consumption). Charging processes with a low flexibility are given priority.

Default: 'dumb'.

- **timestamp_share_threshold** (*float*) – Percental threshold of the time required at a time step for charging the vehicle. If the time requirement is below this limit, then the charging process is not mapped into the time series. If, however, it is above this limit, the time step is mapped to 100% into the time series. This prevents differences between the charging strategies and creates a compromise between the simultaneity of charging processes and an artificial increase in the charging demand. Default: 0.2.
- **minimum_charging_capacity_factor** (*float*) – Technical minimum charging power of charging points in p.u. used in case of charging strategy 'reduced'. E.g. for a charging point with a nominal capacity of 22 kW and a minimum_charging_capacity_factor of 0.1 this would result in a minimum charging power of 2.2 kW. Default: 0.1.

Notes

If the frequency of time series data in `TimeSeries` (checked using `timeindex`) differs from the frequency of SimBEV data, then the time series in `TimeSeries` is first automatically resampled to match the SimBEV data frequency and after determining the charging demand time series resampled back to the original frequency.

import_heat_pumps(*scenario*, *engine*, *timeindex=None*, *import_types=None*)

Gets heat pump data for specified scenario from oedb and integrates the heat pumps into the grid.

Besides heat pump capacity the heat pump's COP and heat demand to be served are as well retrieved.

Currently, the only supported data source is scenario data generated in the research project eGo^n. You can choose between two scenarios: 'eGon2035' and 'eGon100RE'.

The data is retrieved from the open energy platform.

# ToDo Add information on scenarios and from which tables data is retrieved.

The following steps are conducted in this function:

- Heat pump capacities for individual and district heating per building respectively district heating area are obtained from the database for the specified scenario and integrated into the grid using the function `oedb()`.
- Heat pumps are integrated into the grid (added to `loads_df`) as follows.
  - Grid connection points of heat pumps for individual heating are determined based on the corresponding building ID. In case the heat pump is too large to use the same grid connection point, they are connected via their own grid connection point.
  - Grid connection points of heat pumps for district heating are determined based on their geolocation and installed capacity. See `connect_to_mv` and `connect_to_lv_based_on_geolocation` for more information.
- COP and heat demand for each heat pump are retrieved from the database, using the functions `cop_oedb()` respectively `heat_demand_oedb()`, and stored in the `HeatPump` class that can be accessed through `heat_pump`.

Be aware that this function does not yield electricity load time series for the heat pumps. The actual time series are determined through applying an operation strategy or optimising heat pump dispatch. Further, the heat pumps do not yet have a thermal storage and can therefore not yet be used as a flexibility. Thermal storage units need to be added manually to `thermal_storage_units_df`.

After the heat pumps are integrated there may be grid issues due to the additional load. 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

- **scenario** (*str*) – Scenario for which to retrieve heat pump data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to set COP and heat demand data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year (2035 in case of the 'eGon2035' and to 2045 in case of the 'eGon100RE' scenario) and returned for the whole year. If no timeindex is provided, the timeindex set in *timeindex* is used. If *timeindex* is not set, the data is indexed using the default year and returned for the whole year.
- **import_types** (*list(str) or None*) – Specifies which technologies to import. Possible options are "individual_heat_pumps", "central_heat_pumps" and "central_resistive_heaters". If None, all are imported.

apply_heat_pump_operating_strategy(*strategy='uncontrolled'*, *heat_pump_names=None*, ***kwargs*)

Applies operating strategy to set electrical load time series of heat pumps.

This function requires that COP and heat demand time series, and depending on the operating strategy also information on thermal storage units, were previously set in *heat_pump*. COP and heat demand information is automatically set when using *import_heat_pumps*. When not using this function it can be manually set using *set_cop* and *set_heat_demand*.

The electrical load time series of each heat pump are written to *loads_active_power*. Reactive power in *loads_reactive_power* is set to 0 Mvar.

Parameters

- **strategy** (*str*) – Defines the operating strategy to apply. The following strategies are valid:
  - 'uncontrolled'

    The heat demand is directly served by the heat pump without buffering heat using a thermal storage. The electrical load of the heat pump is determined as follows:

$$P_{el} = P_{th}/COP$$

  Default: 'uncontrolled'.
- **heat_pump_names** (*list(str) or None*) – Defines for which heat pumps to apply operating strategy. If None, all heat pumps for which COP information in *heat_pump* is given are used. Default: None.

import_dsm(*scenario: str*, *engine: sqlalchemy.engine.base.Engine*, *timeindex=None*)

Gets industrial and CTS DSM profiles from the OpenEnergy DataBase.

Profiles comprise minimum and maximum load increase in MW as well as maximum energy pre- and postponing in MWh. The data is written to the *DSM* object.

Currently, the only supported data source is scenario data generated in the research project eGo^n. You can choose between two scenarios: 'eGon2035' and 'eGon100RE'.

Parameters

- **edisgo_object** (*EDisGo*) –
- **scenario** (*str*) – Scenario for which to retrieve DSM data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.
- **timeindex** (pandas.DatetimeIndex or None) – Specifies time steps for which to get data. Leap years can currently not be handled. In case the given timeindex contains a leap year, the data will be indexed using the default year (2035 in case of the 'eGon2035' and to 2045 in case of the 'eGon100RE' scenario) and returned for the whole year. If no timeindex is provided, the timeindex set in *timeindex* is used. If *timeindex* is not set, the data is indexed using the default year and returned for the whole year.

import_home_batteries(*scenario: str*, *engine: sqlalchemy.engine.base.Engine*)

Gets home battery data for specified scenario and integrates the batteries into the grid.

Currently, the only supported data source is scenario data generated in the research project eGo^n. You can choose between two scenarios: 'eGon2035' and 'eGon100RE'.

The data is retrieved from the open energy platform.

The batteries are integrated into the grid (added to *storage_units_df*) based on their building ID. In case the battery is too large to use the same grid connection point as the generator or, if no generator is allocated at the same building ID, the load, they are connected via their own grid connection point, based on their geolocation and installed capacity.

Be aware that this function does not yield time series for the batteries. The actual time series can be determined through a dispatch optimisation.

Parameters

- **scenario** (*str*) – Scenario for which to retrieve home battery data. Possible options are 'eGon2035' and 'eGon100RE'.
- **engine** (sqlalchemy.Engine) – Database engine.

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 (*bool*) – 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.calculate_relative_line_load()`. 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_topology=True*, *save_timeseries=True*, *save_results=True*, *save_electromobility=False*, *save_opf_results=False*, *save_heatpump=False*, *save_overlying_grid=False*, *save_dsm=False*, ***kwargs*)

Saves EDisGo object to csv files.

It can be chosen what is included in the csv export (e.g. power flow results, electromobility flexibility, etc.). Further, in order to save disk storage space the data type of time series data can be reduced, e.g. to float32 and data can be archived, e.g. in a zip archive.

Parameters

- **directory** (`str`) – Main directory to save EDisGo object to.
- **save_topology** (`bool, optional`) – Indicates whether to save `Topology` object. Per default, it is saved to sub-directory 'topology'. See `to_csv` for more information. Default: True.
- **save_timeseries** (`bool, optional`) – Indicates whether to save `TimeSeries` object. Per default it is saved to subdirectory 'timeseries'. Through the keyword arguments *reduce_memory* and *to_type* it can be chosen if memory should be reduced. See `to_csv` for more information. Default: True.

- **save_results** (*bool, optional*) – Indicates whether to save `Results` object. Per default, it is saved to subdirectory 'results'. Through the keyword argument *parameters* the results that should be stored can be specified. Further, through the keyword parameters *reduce_memory* and *to_type* it can be chosen if memory should be reduced. See `to_csv` for more information. Default: True.
- **save_electromobility** (*bool, optional*) – Indicates whether to save `Electromobility` object. Per default, it is not saved. If set to True, it is saved to subdirectory 'electromobility'. See `to_csv` for more information.
- **save_opf_results** (*bool, optional*) – Indicates whether to save `OPFResults` object. Per default, it is not saved. If set to True, it is saved to subdirectory 'opf_results'. See `to_csv` for more information.
- **save_heatpump** (*bool, optional*) – Indicates whether to save `HeatPump` object. Per default, it is not saved. If set to True, it is saved to subdirectory 'heat_pump'. See `to_csv` for more information.
- **save_overlying_grid** (*bool, optional*) – Indicates whether to save `OverlyingGrid` object. Per default, it is not saved. If set to True, it is saved to subdirectory 'overlying_grid'. See `to_csv` for more information.
- **save_dsm** (*bool, optional*) – Indicates whether to save `DSM` object. Per default, it is not saved. If set to True, it is saved to subdirectory 'dsm'. See `to_csv` for more information.
- **reduce_memory** (*bool, optional*) – If True, size of dataframes containing time series in `Results`, `TimeSeries`, `HeatPump`, `OverlyingGrid` and `DSM` is reduced. See respective classes *reduce_memory* functions 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 trade-off between precision and memory. Default: "float32".
- **parameters** (*None or dict*) – Specifies which results to store. By default, this is set to None, in which case all available results are stored. To only store certain results provide a dictionary. See function docstring *parameters* parameter in `to_csv` for more information.
- **electromobility_attributes** (*None or list(str)*) – Specifies which electromobility attributes to store. By default, this is set to None, in which case all attributes are stored. See function docstring *attributes* parameter in `to_csv` for more information.
- **archive** (*bool, optional*) – Save disk storage capacity by archiving the csv files. The archiving takes place after the generation of the CSVs and therefore temporarily the storage needs are higher. Default: False.
- **archive_type** (*str, optional*) – Set archive type. Default: "zip".
- **drop_unarchived** (*bool, optional*) – Drop the unarchived data if parameter archive is set to True. Default: True.

save_edisgo_to_pickle(*path='', filename=None*)

Saves EDisGo object to pickle file.

Parameters

- **path** (*str*) – Directory the pickle file is saved to. Per default it takes the current working directory.

- **filename** (*str or None*) – Filename the pickle file is saved under. If None, filename is 'edisgo_object_{grid_id}.pkl'.

save_edisgo_to_json(*filename=None*, *path=''*, *s_base=1*, *flexible_cps=None*, *flexible_hps=None*, *flexible_loads=None*, *flexible_storage_units=None*, *opf_version=1*)

Saves EDisGo object in PowerModels network data format to json file.

Parameters

- **filename** (*str or None*) – Filename the json file is saved under. If None, filename is 'ding0_{grid_id}_t_{#timesteps}.json'.
- **path** (*str*) – Directory the json file is saved to. Per default, it takes the current working directory.
- **s_base** (*int*) – Base value of apparent power for per unit system. Default: 1 MVA
- **flexible_cps** (numpy.ndarray or None) – Array containing all charging points that allow for flexible charging.
- **flexible_hps** (numpy.ndarray or None) – Array containing all heat pumps that allow for flexible operation due to an attached heat storage.
- **flexible_loads** (numpy.ndarray or None) – Array containing all flexible loads that allow for application of demand side management strategy.
- **flexible_storage_units** (numpy.ndarray or None) – Array containing all flexible storages. Non-flexible storages operate to optimize self consumption. Default: None
- **opf_version** (*Int*) – Version of optimization models to choose from. Must be one of [1, 2, 3, 4]. For more information see `edisgo.opf.powermodels_opf.pm_optimize()`. Default: 1.

Returns

Dictionary that contains all network data in PowerModels network data format.

Return type

dict

reduce_memory(***kwargs*)

Reduces size of time series data 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.

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.
- **heat_pump_attr_to_reduce** (*list(str), optional*) – See *attr_to_reduce* parameter in `reduce_memory` for more information.
- **overlying_grid_attr_to_reduce** (*list(str), optional*) – See *attr_to_reduce* parameter in `reduce_memory` for more information.
- **dsm_attr_to_reduce** (*list(str), optional*) – See *attr_to_reduce* parameter in `reduce_memory` for more information.

spatial_complexity_reduction(*copy_edisgo: bool = False*, *mode: str = 'kmeansdijkstra'*, *cluster_area: str = 'feeder'*, *reduction_factor: float = 0.25*, *reduction_factor_not_focused: bool | float = False*, *apply_pseudo_coordinates: bool = True*, ***kwargs*) → tuple[*EDisGo*, pandas.DataFrame, pandas.DataFrame]

Reduces the number of busses and lines by applying a spatial clustering.

Per default, this function creates pseudo coordinates for all busses in the LV grids (see function `make_pseudo_coordinates()`). In case LV grids are not geo-referenced, this is a necessary step. If they are already geo-referenced it can still be useful to obtain better results.

Which busses are clustered is determined in function `make_busmap()`. The clustering method used can be specified through the parameter *mode*. Further, the clustering can be applied to different areas such as the whole grid or the separate feeders, which is specified through the parameter *cluster_area*, and to different degrees, specified through the parameter *reduction_factor*.

The actual spatial reduction of the EDisGo object is conducted in function `apply_busmap()`. The changes, such as dropping of lines connecting the same buses and adapting buses loads, generators and storage units are connected to, are applied directly in the Topology object. If you want to keep information on the original grid, hand a copy of the EDisGo object to this function. You can also set how loads and generators at clustered busses are aggregated through the keyword arguments *load_aggregation_mode* and *generator_aggregation_mode*.

Parameters

- **copy_edisgo** (*bool*) – Defines whether to apply the spatial complexity reduction directly on the EDisGo object or on a copy. Per default, the complexity reduction is directly applied.
- **mode** (*str*) – Clustering method to use. Possible options are "kmeans", "kmeans-dijkstra", "aggregate_to_main_feeder" or "equidistant_nodes". The clustering methods "aggregate_to_main_feeder" and "equidistant_nodes" only work for the cluster area "main_feeder".
  - **"kmeans":** Perform the k-means algorithm on the cluster area and then map the buses to the cluster centers.
  - **"kmeansdijkstra":** Perform the k-means algorithm and then map the nodes to the cluster centers through the shortest distance in the graph. The distances are calculated using the dijkstra algorithm.
  - **"aggregate_to_main_feeder":** Aggregate the nodes in the feeder to the longest path in the feeder, here called main feeder.
  - **"equidistant_nodes":** Uses the method "aggregate_to_main_feeder" and then reduces the nodes again through a reduction of the nodes by the specified reduction factor and distributing the remaining nodes on the graph equidistantly.

  Default: "kmeansdijkstra".
- **cluster_area** (*str*) – The cluster area is the area the different clustering methods are applied to. Possible options are 'grid', 'feeder' or 'main_feeder'. Default: "feeder".
- **reduction_factor** (*float*) – Factor to reduce number of nodes by. Must be between 0 and 1. Default: 0.25.

- **reduction_factor_not_focused** (*bool or float*) – If False, uses the same reduction factor for all cluster areas. If between 0 and 1, this sets the reduction factor for buses not of interest (these are buses without voltage or overloading issues, that are determined through a worst case power flow analysis). When selecting 0, the nodes of the clustering area are aggregated to the transformer bus. This parameter is only used when parameter *cluster_area* is set to 'feeder' or 'main_feeder'. Default: False.
- **apply_pseudo_coordinates** (*bool*) – If True pseudo coordinates are applied. The spatial complexity reduction method is only tested with pseudo coordinates. Default: True.
- **line_naming_convention** (*str*) – Determines how to set "type_info" and "kind" in case two or more lines are aggregated. Possible options are "standard_lines" or "combined_name". If "standard_lines" is selected, the values of the standard line of the respective voltage level are used to set "type_info" and "kind". If "combined_name" is selected, "type_info" and "kind" contain the concatenated values of the merged lines. x and r of the lines are not influenced by this as they are always determined from the x and r values of the aggregated lines. Default: "standard_lines".
- **aggregation_mode** (*bool*) – Specifies, whether to aggregate loads and generators at the same bus or not. If True, loads and generators at the same bus are aggregated according to their selected modes (see parameters *load_aggregation_mode* and *generator_aggregation_mode*). Default: False.
- **load_aggregation_mode** (*str*) – Specifies, how to aggregate loads at the same bus, in case parameter *aggregation_mode* is set to True. Possible options are "bus" or "sector". If "bus" is chosen, loads are aggregated per bus. When "sector" is chosen, loads are aggregated by bus, type and sector. Default: "sector".
- **generator_aggregation_mode** (*str*) – Specifies, how to aggregate generators at the same bus, in case parameter *aggregation_mode* is set to True. Possible options are "bus" or "type". If "bus" is chosen, generators are aggregated per bus. When "type" is chosen, generators are aggregated by bus and type.
- **mv_pseudo_coordinates** (*bool, optional*) – If True pseudo coordinates are also generated for MV grid. Default: False.

Returns

Returns the EDisGo object (which is only relevant in case the parameter *copy_edisgo* was set to True), as well as the busmap and linemap dataframes. The busmap maps the original busses to the new busses with new coordinates. Columns are "new_bus" with new bus name, "new_x" with new x-coordinate and "new_y" with new y-coordinate. Index of the dataframe holds bus names of original buses as in buses_df. The linemap maps the original line names (in the index of the dataframe) to the new line names (in column "new_line_name").

Return type

tuple(*EDisGo*, pandas.DataFrame, pandas.DataFrame)

check_integrity()

Method to check the integrity of the EDisGo object.

Checks for consistency of topology (see *edisgo.network.topology.Topology.check_integrity()*), timeseries (see *edisgo.network.timeseries.TimeSeries.check_integrity()*) and the interplay of both. Further, checks integrity of electromobility object (see *edisgo.network.electromobility.Electromobility.check_integrity()*), the heat pump object (see *edisgo.network.heat.HeatPump.check_integrity()*) and the DSM object (see *edisgo.network.dsm.DSM.check_integrity()*). Additionally, checks whether time series data in

`HeatPump`, `Electromobility`, `OverlyingGrid` and `DSM` contains all time steps in `edisgo.network.timeseries.TimeSeries.timeindex`.

resample_timeseries(*method: str = 'ffill'*, *freq: str | pandas.Timedelta = '15min'*)

Resamples time series data in `TimeSeries`, `HeatPump`, `Electromobility` and `OverlyingGrid`.

Both up- and down-sampling methods are possible.

The following time series are affected by this:

- `generators_active_power`
- `loads_active_power`
- `storage_units_active_power`
- `generators_reactive_power`
- `loads_reactive_power`
- `storage_units_reactive_power`
- `flexibility_bands`
- `cop_df`
- `heat_demand_df`
- All data in `OverlyingGrid`

Parameters

- **method** (`str, optional`) – Method to choose from to fill missing values when resampling time series data (only exception from this is for flexibility bands in `Electromobility` object where method cannot be chosen to assure consistency of flexibility band data). Possible options are:

  – **'ffill' (default)**
  Propagate last valid observation forward to next valid observation. See pandas.DataFrame.ffill.

  – **'bfill'**
  Use next valid observation to fill gap. See pandas.DataFrame.bfill.

  – **'interpolate'**
  Fill NaN values using an interpolation method. See pandas.DataFrame.interpolate.

  Default: 'ffill'.

- **freq** (`str, optional`) – Frequency that time series is resampled to. Offset aliases can be found here: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases. Default: '15min'.

edisgo.edisgo.**import_edisgo_from_pickle**(*filename*, *path=''*)

Restores EDisGo object from pickle file.

Parameters

- **filename** (`str`) – Filename the pickle file is saved under.
- **path** (`str`) – Directory the pickle file is restored from. Per default it takes the current working directory.

edisgo.edisgo.**import_edisgo_from_files**(*edisgo_path: str | pathlib.PurePath*, *import_topology: bool = True*, *import_timeseries: bool = False*, *import_results: bool = False*, *import_electromobility: bool = False*, *import_opf_results: bool = False*, *import_heat_pump: bool = False*, *import_dsm: bool = False*, *import_overlying_grid: bool = False*, *from_zip_archive: bool = False*, ***kwargs*)

Sets up EDisGo object from csv files.

This is the reverse function of `save` and if not specified differently assumes all data in the default sub-directories created in the `save` function.

Parameters

- **edisgo_path** (*str or pathlib.PurePath*) – Main directory to restore EDisGo object from. This directory must contain the config files. Further, if not specified differently, it is assumed to be the main directory containing sub-directories with e.g. topology data. In case *from_zip_archive* is set to True, *edisgo_path* is the name of the archive.
- **import_topology** (*bool*) – Indicates whether to import `Topology` object. Per default, it is set to True, in which case topology data is imported. The default directory topology data is imported from is the sub-directory 'topology'. A different directory can be specified through keyword argument *topology_directory*. Default: True.
- **import_timeseries** (*bool*) – Indicates whether to import `TimeSeries` object. Per default, it is set to False, in which case timeseries data is not imported. The default directory time series data is imported from is the sub-directory 'timeseries'. A different directory can be specified through keyword argument *timeseries_directory*. Default: False.
- **import_results** (*bool*) – Indicates whether to import `Results` object. Per default, it is set to False, in which case results data is not imported. The default directory results data is imported from is the sub-directory 'results'. A different directory can be specified through keyword argument *results_directory*. Default: False.
- **import_electromobility** (*bool*) – Indicates whether to import `Electromobility` object. Per default, it is set to False, in which case electromobility data is not imported. The default directory electromobility data is imported from is the sub-directory 'electromobility'. A different directory can be specified through keyword argument *electromobility_directory*. Default: False.
- **import_opf_results** (*bool*) – Indicates whether to import `OPFResults` object. Per default, it is set to False, in which case opf results data is not imported. The default directory results data is imported from is the sub-directory 'opf_results'. A different directory can be specified through keyword argument *opf_results_directory*. Default: False.
- **import_heat_pump** (*bool*) – Indicates whether to import `HeatPump` object. Per default, it is set to False, in which case heat pump data containing information on COP, heat demand time series, etc. is not imported. The default directory heat pump data is imported from is the sub-directory 'heat_pump'. A different directory can be specified through keyword argument *heat_pump_directory*. Default: False.
- **import_overlying_grid** (*bool*) – Indicates whether to import `OverlyingGrid` object. Per default, it is set to False, in which case overlying grid data containing information on renewables curtailment requirements, generator dispatch, etc. is not imported. The default directory overlying grid data is imported from is the sub-directory 'overlying_grid'. A different directory can be specified through keyword argument *overlying_grid_directory*. Default: False.
- **import_dsm** (*bool*) – Indicates whether to import `DSM` object. Per default, it is set to False, in which case DSM data is not imported. The default directory DSM data is imported from is the sub-directory 'dsm'. A different directory can be specified through keyword argument *dsm_directory*. Default: False.
- **from_zip_archive** (*bool*) – Set to True if data needs to be imported from an archive, e.g. a zip archive. Default: False.
- **topology_directory** (*str*) – Indicates directory `Topology` object is imported from. Per default, topology data is imported from *edisgo_path* sub-directory 'topology'.
- **timeseries_directory** (*str*) – Indicates directory `TimeSeries` object is imported from. Per default, time series data is imported from *edisgo_path* sub-directory 'timeseries'.
- **results_directory** (*str*) – Indicates directory `Results` object is imported from. Per default, results data is imported from *edisgo_path* sub-directory 'results'.

- **electromobility_directory** (*str*) – Indicates directory *Electromobility* object is imported from. Per default, electromobility data is imported from *edisgo_path* sub-directory 'electromobility'.
- **opf_results_directory** (*str*) – Indicates directory *OPFResults* object is imported from. Per default, results data is imported from *edisgo_path* sub-directory 'opf_results'.
- **heat_pump_directory** (*str*) – Indicates directory *HeatPump* object is imported from. Per default, heat pump data is imported from *edisgo_path* sub-directory 'heat_pump'.
- **overlying_grid_directory** (*str*) – Indicates directory *OverlyingGrid* object is imported from. Per default, overlying grid data is imported from *edisgo_path* sub-directory 'overlying_grid'.
- **dsm_directory** (*str*) – Indicates directory *DSM* object is imported from. Per default, DSM data is imported from *edisgo_path* sub-directory 'dsm'.
- **dtype** (*str*) – Numerical data type for time series and results data to be imported, e.g. "float32". Per default, this is None in which case data type is inferred.
- **parameters** (*None or dict*) – 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. See function docstring *parameters* parameter in `to_csv()` for more information.

Returns
Restored EDisGo object.

Return type
EDisGo

BIBLIOGRAPHY

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

[PyPSA] PyPSA - Spatial Clustering Documentation

[SCR] Master Thesis - Malte Jahn - Analysis of the effects of spatial complexity reduction on Distribution network expansion planning with flexibilities (written in German)

[HoerschBrown] Jonas Hörsch, Tom Brown: The role of spatial scale in joint optimisations of generation and transmission for European highly renewable scenarios

PYTHON MODULE INDEX

INDEX