eDisGo Documentation

Release 0.1.0

open$_e Go - Team$

Jul 27, 2021

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
- Static, non-linear power flow analysis using PyPSA for grid issue identification
- Automatic grid reinforcement methodology solving overloading and voltage issues to determine grid expansion needs and costs based on measures most commonly taken by German distribution grid operators
- Multiperiod optimal power flow based on julia package PowerModels.jl optimizing storage positioning and/or operation as well as generator dispatch with regard to minimizing grid expansion costs
- Temporal complexity reduction
- Heuristic for grid-supportive generator curtailment
- Heuristic grid-supportive battery storage integration

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

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

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

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

CHAPTER 1

Contents

1.1 Getting started

1.1.1 Installation

Warning: Make sure to use python 3.7 or higher!

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

```
pip3 install edisgo
```

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

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

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

Installation under Windows

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

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

Activate the newly created environment with:

```
conda activate eDisGo_env
```

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

Requirements for edisgoOPF package

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

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

```
]
```

Then activate the package:

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

And finally instantiate it:

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

Additional linear solver

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

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

Unpack the tar.gz:

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

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

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

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

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

You can then configure and install your HSL Solvers:

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

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

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

Add this path to the variable *LD_LIBRARY_PATH*:

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

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

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

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

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

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

1.1.2 Prerequisites

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

1.1.3 A minimum working example

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

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

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

You can run a worst-case scenario as follows:

```
from edisgo import EDisGo

# Set up the EDisGo object that will import the grid topology, set up
# feed-in and load time series (here for a worst case analysis)
# and other relevant data
edisgo = EDisGo(ding0_grid='ding0_example_grid',
                worst_case_analysis='worst-case')

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

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

(continues on next page)

(continued from previous page)

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

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

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

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object with your own time series
# (these are dummy time series!)
# timeindex specifies which time steps to consider in power flow
timeindex = pd.date_range('1/1/2011', periods=4, freq='H')
# load time series (scaled by annual demand)
timeseries_load = pd.DataFrame(
    {'residential': [0.0001] * len(timeindex),
     'retail': [0.0002] * len(timeindex),
     'industrial': [0.00015] * len(timeindex),
     'agricultural': [0.00005] * len(timeindex)
     },
    index=timeindex)
# feed-in time series of fluctuating generators (scaled by nominal power)
timeseries_generation_fluctuating = pd.DataFrame(
    {'solar': [0.2] * len(timeindex),
     'wind': [0.3] * len(timeindex)
     },
    index=timeindex)
# feed-in time series of dispatchable generators (scaled by nominal power)
timeseries_generation_dispatchable = pd.DataFrame(
    {'biomass': [1] * len(timeindex),
     'coal': [1] * len(timeindex),
     'other': [1] * len(timeindex)
     },
    index=timeindex)

# Set up the EDisGo object with your own time series and generator scenario
# NEP2035
edisgo = EDisGo(
    ding0_grid='ding0_example_grid',
    generator_scenario='nep2035',
    timeseries_load=timeseries_load,
    timeseries_generation_fluctuating=timeseries_generation_fluctuating,
    timeseries_generation_dispatchable=timeseries_generation_dispatchable,
    timeindex=timeindex)

# Do grid reinforcement
edisgo.reinforce()

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

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

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object using the OpenEnergy DataBase and the oemof
# demandlib to set up time series for loads and fluctuating generators
# (time series for dispatchable generators need to be provided)
timeindex = pd.date_range('1/1/2011', periods=4, freq='H')
timeseries_generation_dispatchable = pd.DataFrame(
    {'biomass': [1] * len(timeindex),
     'coal': [1] * len(timeindex),
     'other': [1] * len(timeindex)
     },
    index=timeindex)

edisgo = EDisGo(
    ding0_grid='ding0_example_grid',
    generator_scenario='ego100',
    timeseries_load='demandlib',
    timeseries_generation_fluctuating='oedb',
    timeseries_generation_dispatchable=timeseries_generation_dispatchable,
    timeindex=timeindex)

# Do grid reinforcement
edisgo.reinforce()

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

1.1.4 Parallelization

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

1.1.5 LICENSE

1.2 Usage details

As eDisGo is designed to serve as a toolbox, it provides several methods to analyze distribution grids for grid issues and to evaluate measures responding these. Below, we give a detailed introduction to the data structure and to how different features can be used.

1.2.1 The fundamental data structure

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

The class `EDisGo` serves as the top-level API for setting up your scenario, invocation of data import, analysis of hosting capacity, grid reinforcement and flexibility measures. It also provides access to all relevant data. Grid data is stored in the `Topology` class. Time series data can be found in the `TimeSeries` class. Results data holding results e.g. from the power flow analysis and grid expansion is stored in the `Results` class. Configuration data from the config files (see *Default configuration data*) is stored in the `Config` class. All these can be accessed through the `EDisGo` object. In the following code examples *edisgo* constitues an `EDisGo` object.

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

# Access TimeSeries data container object
edisgo.timeseries

# Access Results data container object
edisgo.results

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A list of all LV grids can be retrieved through:

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

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

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

# Access all buses in that LV grid
lv_grid.buses_df

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

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

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

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

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

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

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

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

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

1.2.2 Identify grid issues

As detailed in *A minimum working example*, once you set up your scenario by instantiating an `EDisGo` object, you are ready for a grid analysis and identifying grid issues (line overloading and voltage issues) using `analyze()`:

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

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

The range of time analyzed by the power flow analysis is by default defined by the `timeindex()`, that can be given as an input to the EDisGo object through the parameter *timeindex* or is otherwise set automatically. If you want to change the time steps that are analyzed, you can specify those through the parameter *timesteps* of the *analyze* function. Make sure that the specified time steps are a subset of `timeindex()`.

1.2.3 Grid expansion

Grid expansion can be invoked by `reinforce()`:

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

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

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

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

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

1.2.4 Battery storage systems

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

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

```
from edisgo import EDisGo

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

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

```
import pandas as pd
from edisgo import EDisGo

# Set up the EDisGo object using the OpenEnergy DataBase and the oemof
# demandlib to set up time series for loads and fluctuating generators
# (time series for dispatchable generators need to be provided)
timeindex = pd.date_range('1/1/2011', periods=4, freq='H')
timeseries_generation_dispatchable = pd.DataFrame(
    {'biomass': [1] * len(timeindex),
     'coal': [1] * len(timeindex),
     'other': [1] * len(timeindex)
     },
    index=timeindex)
edisgo = EDisGo(
    ding0_grid='ding0_example_grid',
    generator_scenario='ego100',
    timeseries_load='demandlib',
    timeseries_generation_fluctuating='oedb',
    timeseries_generation_dispatchable=timeseries_generation_dispatchable,
    timeindex=timeindex)

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

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

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

1.2.5 Curtailment

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

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

(continues on next page)

(continued from previous page)

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

1.2.6 Plots

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

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

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

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

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

1.2.7 Results

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

```
edisgo.results
```

Get voltages at nodes from `v_res()` and line loading from `s_res()` or `i_res`. `equipment_changes` holds details about measures performed during grid expansion. Associated costs can be obtained through `grid_expansion_costs`. Flexibility measures may not entirely resolve all issues. These unresolved issues are listed in `unresolved_issues`.

Results can be saved to csv files with:

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

See `save()` for more information.

1.3 Features in detail

1.3.1 Power flow analysis

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

1.3.2 Multi period optimal power flow

Todo: Add

1.3.3 Grid expansion

General methodology

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

The order grid expansion measures are conducted is as follows:

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

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

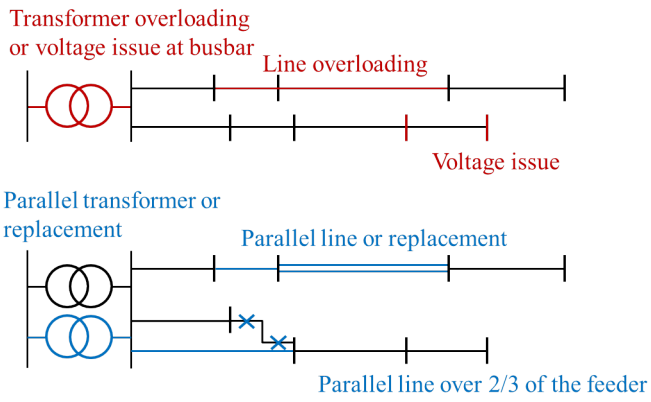

Fig. 1.1: Grid expansion measures

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

Identification of overloading and voltage issues

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

Voltage issues are determined based on allowed voltage deviations set in the config file *config_grid_expansion* in section *grid_expansion_allowed_voltage_deviations*. It is possible to set one allowed voltage deviation that is used for MV and LV or define separate allowed voltage deviations. Which allowed voltage deviation is used is defined through the parameter *combined_analysis* of `reinforce_grid()`. By default *combined_analysis* is set to false, resulting

in separate voltage limits for MV and LV, as a combined limit may currently lead to problems if voltage deviation in MV grid is already close to the allowed limit, in which case the remaining allowed voltage deviation in the LV grids is close to zero.

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

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

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

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

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

Check line load

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

Check station load

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

Check line and station voltage deviation

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

Grid expansion measures

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

Reinforce lines due to overloading issues

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

Reinforce stations due to overloading issues

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

Reinforce MV/LV stations due to voltage issues

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

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

Reinforce lines due to voltage

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

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

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

Grid expansion costs

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

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

1.3.4 Curtailment

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

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

'feedin-proportional'

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

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

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

'voltage-based'

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

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

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

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

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

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

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

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

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

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

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

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

1.3.5 Storage integration

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

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

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

A more thorough documentation will follow soon.

1.3.6 References

1.4 Notes to developers

1.4.1 Installation

Clone repository from GitHub and install in developer mode:

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

1.4.2 Code style

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

1.4.3 Documentation

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

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

And then you build the html docs on your computer with

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

1.5 Definition and units

1.5.1 Sign Convention

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

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

Generator Sign Convention

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

Load Sign Convention

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

1.5.2 Reactive Power Sign Convention

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

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

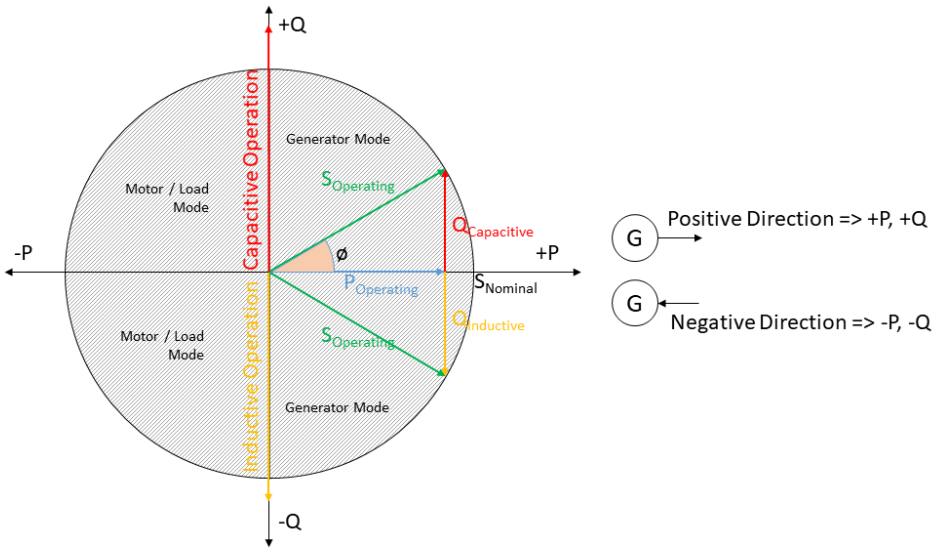

Fig. 1.2: Generator sign convention in detail

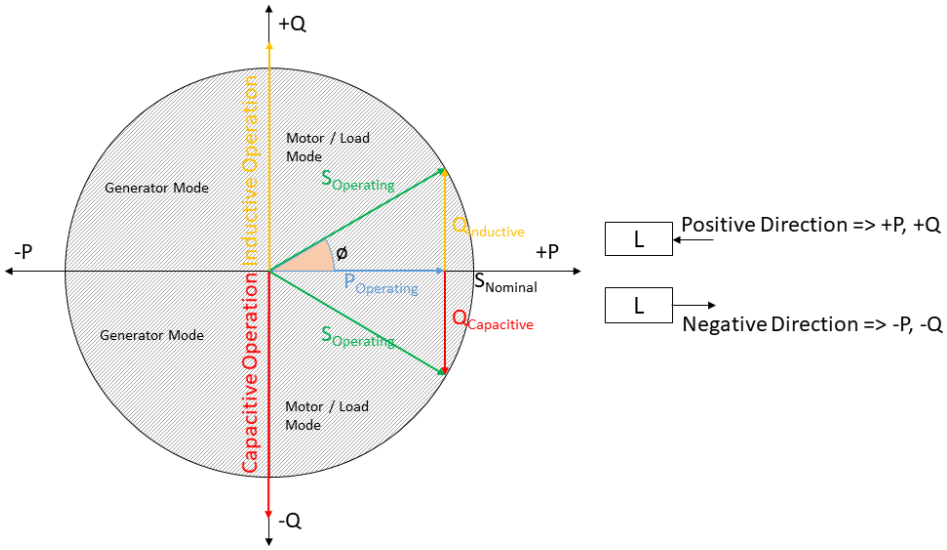

Fig. 1.3: Load sign convention in detail

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

1.5.3 Units

Table 1.1: List of variables and units

Variable	Symbol	Unit	Comment
Current	I	kA	
Length	l	km	
Active Power	P	MW	
Reactive Power	Q	MVar	
Apparent Power	S	MVA	
Resistance	R	Ohm or Ohm/km	Ohm/km applies to lines
Reactance	X	Ohm or Ohm/km	Ohm/km applies to lines
Voltage	V	kV	
Inductance	L	mH/km	
Capacitance	C	µF/km	
Costs	•	kEUR	

1.6 Default configuration data

Following you find the default configuration files.

1.6.1 config_db_tables

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

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

[data_source]

oedb_data_source = versioned

[model_draft]

conv_generators_prefix = t_ego_supply_conv_powerplant_
```

(continues on next page)

(continued from previous page)

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

#conv_generators_nep2035 = t_ego_supply_conv_powerplant_nep2035_mview
#conv_generators_ego100 = ego_supply_conv_powerplant_ego100_mview
#re_generators_nep2035 = t_ego_supply_res_powerplant_nep2035_mview
#re_generators_ego100 = t_ego_supply_res_powerplant_ego100_mview

[versioned]

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

version = v0.4.5
```

1.6.2 config_grid_expansion

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

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

[grid_expansion_standard_equipment]

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

[grid_expansion_allowed_voltage_deviations]

# allowed voltage deviations
# ==========================
# relevant for all cases
feedin_case_lower = 0.9
```

(continues on next page)

(continued from previous page)

```
load_case_upper = 1.1

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

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

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

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

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

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

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

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

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

[grid_expansion_load_factors]

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

(continues on next page)

(continued from previous page)

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

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

[costs_cables]

# costs in kEUR/km
# costs for cables without earthwork are taken from [1] (costs for standard
# cables are used here as representative since they have average costs), costs
# including earthwork are taken from [2]
# [1] https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Energie/
↪Unternehmen_Institutionen/Netzentgelte/Anreizregulierung/GA_
↪AnalytischeKostenmodelle.pdf?__blob=publicationFile&v=1
# [2] https://shop.dena.de/fileadmin/denashop/media/Downloads_Dateien/esd/9100_dena-
↪Verteilnetzstudie_Abschlussbericht.pdf
# costs including earthwork costs depend on population density according to [2]
# here "rural" corresponds to a population density of <= 500 people/km²
# and "urban" corresponds to a population density of > 500 people/km²
lv_cable = 9
lv_cable_incl_earthwork_rural = 60
lv_cable_incl_earthwork_urban = 100
mv_cable = 20
mv_cable_incl_earthwork_rural = 80
mv_cable_incl_earthwork_urban = 140

[costs_transformers]

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

1.6.3 config_timeseries

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

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

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

[worst_case_scale_factor]
```

(continues on next page)

(continued from previous page)

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

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

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

# temporary own values
feedin_case_storage = 1
load_case_storage = -1

[reactive_power_factor]

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

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

[reactive_power_mode]

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

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

[demandlib]

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

(continues on next page)

(continued from previous page)

```
# for further information

# scaling factors for night and day of weekdays and weekend days
week_day = 0.8
week_night = 0.6
weekend_day = 0.6
weekend_night = 0.6
# tuple specifying the beginning/end of a workday (e.g. 18:00)
day_start = 6:00
day_end = 22:00
```

1.6.4 config_grid

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

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

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

[geo]

# WGS84: 4326
srid = 4326

[grid_connection]

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

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

# conn_buffer_radius_inc:
#     radius which is incrementally added to connect_buffer_radius as long as no␣
↪target is found
```

(continues on next page)

(continued from previous page)

```
#     unit: m
conn_buffer_radius_inc = 1000

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


[disconnecting_point]

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

1.7 Equipment data

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

Table 1.2: LV cables

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

Table 1.3: MV cables

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

Table 1.4: MV overhead lines

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

Table 1.5: LV transformers

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

Table 1.6: MV transformers

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

1.8 API

1.8.1 EDisGo class

1.8.2 edisgo.network package

edisgo.network.components module

edisgo.network.grids module

edisgo.network.results module

edisgo.network.timeseries module

edisgo.network.topology module

1.8.3 edisgo.flex_opt package

1.8.4 edisgo.io package

1.8.5 edisgo.opf package

1.8.6 edisgo.tools package

1.9 What's New

Changelog for each release.

1.9.1 Release v0.1.0

Release date: July 26, 2021

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

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

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

Changes

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

1.9.2 Release v0.0.10

Release date: October 18, 2019

Changes

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

Bug fixes

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

1.9.3 Release v0.0.9

Release date: December 3, 2018

Changes

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

1.9.4 Release v0.0.8

Release date: October 29, 2018

Changes

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

1.9.5 Release v0.0.7

Release date: October 23, 2018

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

Changes

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

1.9.6 Release v0.0.6

Release date: September 6, 2018

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

Changes

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

1.9.7 Release v0.0.5

Release date: July 19, 2018

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

Changes

- Switch disconnecters in MV rings will now be installed, even if no LV station exists in the ring #136

- Update to new version of ding0 v0.1.7
- Consider feed-in and load case in grid expansion methodology
- Enable grid expansion on snapshots
- Bug fixes

1.9.8 Release v0.0.3

Release date: July 6 2018

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

Changes

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

1.9.9 Release v0.0.2

Release date: March 15 2018

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

Changes

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

1.10 Index

Index

Bibliography

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