Electric Vehicles in Energy Communities: Investigating the Distribution Grid Hosting Capacity

Albert Ludwig University of Freiburg

Daniil Aktanka

August 26, 2022

Abstract

Contents

1 Introduction

2 Literature Review

3 Goal and Method

3.1 Approach

To reiterate, the goal of this thesis was to investigate the distribution grid hosting capacity of electric vehicles and the effect the involvement of energy communities can have on it.

testing network then big boi network

The next section will describe the functionality of custom python classes. For more details, consult the source code on GitHub at: https://github.com/akthonka/EV_energy_comm .

3.2 Data processing

Given the scope of the project, the idea was to create a system flexible enough for iterative data manipulation and ideally one that would support multiple network types. The project was written entirely in python due to language simplicity as well as availability of advanced modules. Data handling was done using the `pandas` module, whereas the network simulation was performed using the `pandapower` module.

For the sake of clarity, most of the code was split between a python class file and jupyter notebooks. For early testing and code prototyping, jupyter notebooks are sufficient but they quickly lose readability as the code complexity grows. The general methodology was therefore to write core functions in jupyter to then integrate them into an external python class file. The resultant python class file consists of two classes: one for handling data processing and another for operations based on the `pandapower` module. Common input settings (for instance, the time window) are saved as class variables, such that only the most top-level functions are called in jupyter.

3.2.1 DataAction class

The following information describes the code of the python class responsible for core data processing. While the general methodology is applicable to any dataset, these functions are tailored specifically to the household dataset and will be described as such. Additionally, the functions mentioned here target the European LV network as our simulation case but can be adapted for any network, with several caveats mentioned later.

The paragraphs serve to summarize the operation of one or more functions based on their application.

Data import and segmenting: The raw dataset is imported as a dataframe without additional options. Dropping the rest, we keep only 3 columns: `DE_KN_residential1_grid_import`, `DE_KN_residential2_grid_import` and `utc_timestamp`. Since we will be performing conditional time-based operations, we set the timestamp column as index for ease of use. While it is possible to front-load a lot of data processing functions at this step—such as parsing datetimes—it is not recommended due to unnecessary computation time. A better approach is to segment the data for piece-wise processing and function testing, whereby it would be possible to iterate computations over the entire dataset in the future. Therefore, we split the imported dataframe (of a little over a million data points) into a list of smaller dataframes (each 10000 data points long, with residual last dataframe being a bit smaller).

Datetime parsing function: This function converts a segment of the imported data into a specific, time-indexed dataframe. First, we parse datetime index, specifying the format as `"Year-Month-Day Hour:Minute:Second"`, after which we convert from UTC to Berlin time—the local time of the recorded dataset. While it is possible to import data with local time column `cet_cest_timestamp` without the need of conversion, it is not recommended for `pandas 1.4.3` since that will cause errors in datetime operations based on my experience. Second, we take the difference between consecutive rows. This is done in order to obtain minute-wise energy changes, since the household dataset only tracks cumulative energy values. If we use the `pandas.diff()` function, we must also drop the resulting first row, since it's a NaN row.

Night profile functions: This set of functions is fundamental for the simulation, as they are used to create a load profile for a single household. First, we must select a random dataframe segment with either of the two historical load profiles and parse it for datetime. The segment contains data on several days/nights, the dates of which we must identify. Then, for a random date, we must get the starting and ending datetimes for dataframe slicing. Knowing the starting time of our window—in our case, 18:00:00—we need to create a datetime object for the chosen date, given the starting time. From said datetime (for example, `2015-08-16 18:00:00`) we add one day to the date and update the time with our morning value (getting `2015-08-17 06:00:00`). With these two datetimes, we can now slice the selected dataframe to obtain the required single house overnight load profile.

Create loads and static generators: This function creates the dataframes used for timeseries iteration—from hereon referred to as night dataframes. These hold the inputs for the network values, for both the loads as well as the static generators (sgens). Depending on the timeseries controller, it is important to create the night dataframes in a very specific way: the names and the indices of the network components will dictate the conventions. Unless you are writing your own controller, refer to your documentation for more details.

To create the night dataframes, first get a random night profile. This contains all the necessary time steps in the index, which is what we need; the load values can be set to zero. Then, generate a list of appropriate load names and form a dataframe using the time index. For the case of the European LV network, the resulting night dataframes are 721 rows $\times$ 55 columns (minutes $\times$ number of profiles). Repeat for both loads and sgens night dataframes. It is recommended to save these as class variables, since we will be referencing them throughout the entire simulation, as well as for troubleshooting purposes. Finally, we convert our night loads to MW via a simple conversion factor multiplication.

Random static generators: These functions fill the night sgen dataframe at random times with a select sgen value. It's recommended to first reset the night dataframe by setting all sgen values to zero, otherwise location slicing may not overwrite the value. Getting a random set of starting and ending datetimes follows the same principle as described for night profile functions. It's important to keep track of zero-based indexing when calling random values from range. Additionally, in case of `pandapower`, the sgen sign needs to be set as negative in order to act as a separate static load that is an EV charger.

Energy community static generators: These functions fill the night sgen dataframe consequently and cyclically (see section 3.3 for motivation). This method consists of two parts. In the first part, the starting times for sgens are determined. The allowed time values are bound by the time window, and are dependent on the time step (charging time). To compute the list of starting charging times, begin with the evening time value and add the time step iteratively for the number of sgens present. In our case, we have 55 sgens and a time window of 12 hours; for a charging time of 1 hour, that would result in 12 starting charging times. If we only had 7 sgens in the same time conditions, we would have 7 starting charging times, etc. The next part concerns the starting time sgen cycle.

For best practice, set all night sgen values to zero before writing new ones. The following step was done using `cycle` function from the `itertools` module in python. Beginning with the evening start time, write the sgen charging value for the first generator, for the entire charging duration minus the last minute. The next minute, when the first generator is no longer active, fill the charging time window for the second generator, minus the last minute. Repeat the process until the last unique starting charging time is reached, after which the next starting charging time will loop back to the beginning of the list. Repeat this process for all given sgens. The looping time values can be handled with an extra class placeholder-variable for starting times instead of an entire additional look-up table.

3.2.2 NetworkCalculation class

The following information describes the code of the python class generally responsible for functions related to the `pandapower` module. In particular, its main purpose is perform time series iteration on a specific network. An additional function to calculate the hosting capacity is also contained here. The NetworkCalculation class is meant to be used directly in conjunction with the DataAction class and contains several direct references to it.

Network preparation: In order to perform the time series iteration we first need to prepare our network. In the case of our chosen simulation—European LV network—this involves some additional steps (see next section). First, create the network, including the loads and sgens. It's recommended to stick to a consistent naming convention for loads and sgens across the network properties and the external night dataframes, in order to avoid unnecessary problems when configuring the controller in the next step.

Time iteration and output:

Hosting capacity evaluation:

3.3 Network application

Treating three-phase network as single phase
Network asymmetric loads set to zero
Adjusting the historical load magnitude

3.3.1 Critical case scenario

3.3.2 Random time scenario

3.3.3 Energy community scenario

4 Results and Discussion

4.1 Results

4.2 Suggested improvements

5 Conclusion

Appendix