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

Albert Ludwig University of Freiburg

Daniil Aktanka

August 28, 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. Over the span of roughly two months, the method presented in this paper went through many iterations in terms of functionality as well as network application. The original testing models were performed on a simple test network with generic coordinates and arbitrary values (see fig. 3.1). These tests were done in order to get a feel for the chosen software tools and to better understand the potential real world application and its scope of work.

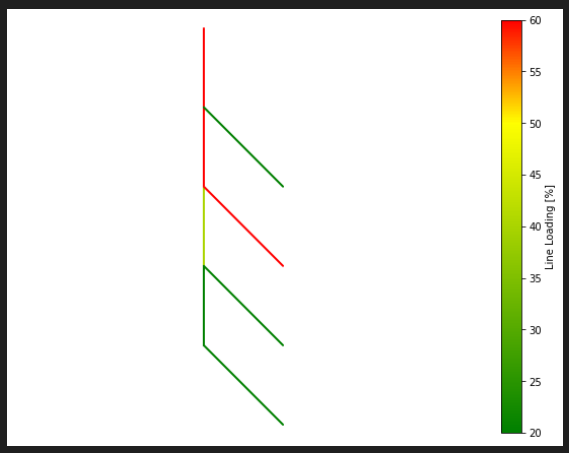

Figure 3.1: An example test power flow calculation plot with line loading results. Here, one static generator is active out of four.

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 timeseries 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 and controllers: In order to perform the timeseries 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.

The controller is a function whose purpose is to update the network values iteratively from a look-up table. The methodology will depend on your software of choice, but in our case we will be using the `pandapower` controllers made specifically for timeseries module. Figure 3.2 shows a general overview of the built-in module.

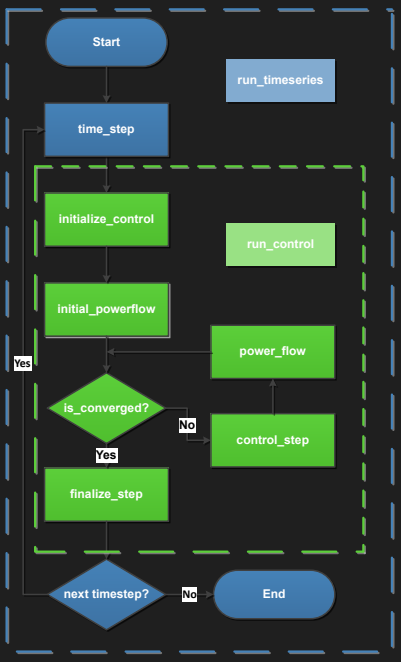

Figure 3.2: The diagram shows the time series loop of the time series module. The basic functionality is compute the power flow for each time step using the given values—such as sgens or loads. Step-wise convergence will depend on several factors, such as the network parameters (i.e. load magnitudes) as well as the algorithm used for solving the power flow problem. For more module details, see the `pandapower` wiki.

For our simulation, we will need 2 constant controllers for loads and sgens. Hereby, the correspondent data sources are the night dataframes that were defined in the DataAction class. Lastly, the timeseries iteration needs to be given a defined number of time steps, which in our case will be given by the index length of (either) night dataframe. With this information, the power flow can be computed iteratively based on tabular variable inputs. However, this will yield no useful results on its own, which is why we need to additionally define an output writer.

Time iteration and output: An output writer is a function with a simple job of recording computed information, in an desired format, for each timeseries step. In our case, we will again be using the output writer function supplied with the `pandapower` timeseries module and outputting into an Excel file for the sake of convenience and troubleshooting. However, we aim to continue data processing in `pandas`, so we define another import function for said Excel results file. The imported dataframe can now provide relevant statistics and metrics. For our evaluation, the one of the key computations is the all-time minimum load value across all busses (see section 3.3 for more details).

Hosting capacity evaluation: Hosting capacity was computed iteratively for a network in a single point in time. All pre-loaded network settings (even from the timeseries iteration) stay the same, except for the sgen columns—those need to be set to zero at first. Do not confuse the night sgen dataframe with the network sgen column; for a single point in time, only the network-defined sgens are taken into the calculation.

Iteration was performed over the list of sgen indices using a `for` loop. The list is scrambled prior iteration in order to mimic uncoordinated charging patterns. An additional variable for storing the hosting capacity count is created. For each cycle of the loop, the following metric variables are computed by running a power flow calculation: minimum bus voltage magnitude (per unit) and maximum line loading percent. The next step is conditional. If the metric variables don't exceed preset limits, add an sgen value and increase the hosting capacity count by one. The loop proceeds onto the next cycle and computes the new power flow state of the network. Checking the metric variables again, if (at some point) the preset limits are exceeded, then we need to undo the effects of the last cycle. As such, the hosting capacity count is reduced by one and the last filled sgen is reset to zero. This breaks the loop. The function then computes one final power flow computation to obtain for an output plot and prints the maximum hosting capacity result.

3.3 Network application

This section builds upon the fundamental operating principles of the two classes introduced in section 3.2. It aims to explain the usage of the python class file in order to obtain the three scenarios for the European Low Voltage Test Feeder.

Assumptions:
Treating three-phase network as single phase
Network asymmetric loads set to zero
Adjusting the historical load magnitude
1 hr window because...

4 Results and Discussion

4.1 Results

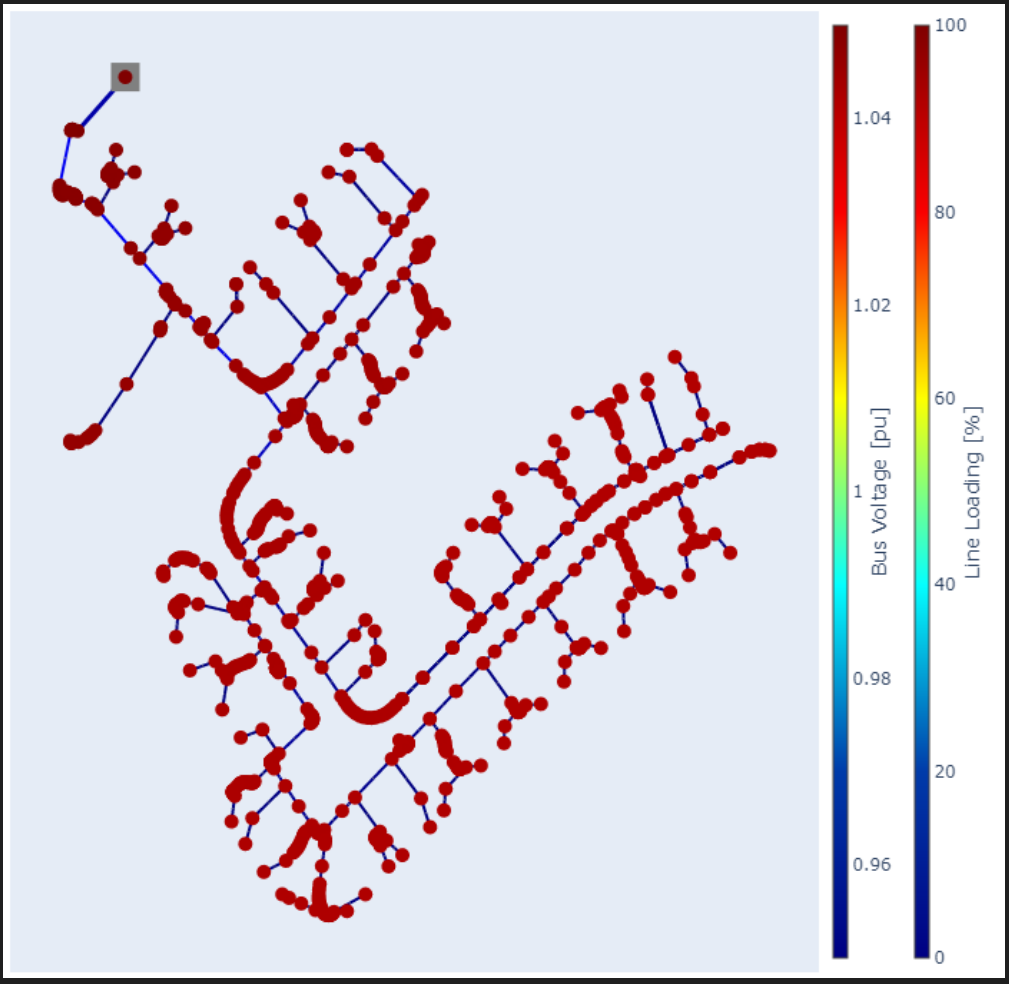

Figure 4.1: blahblah

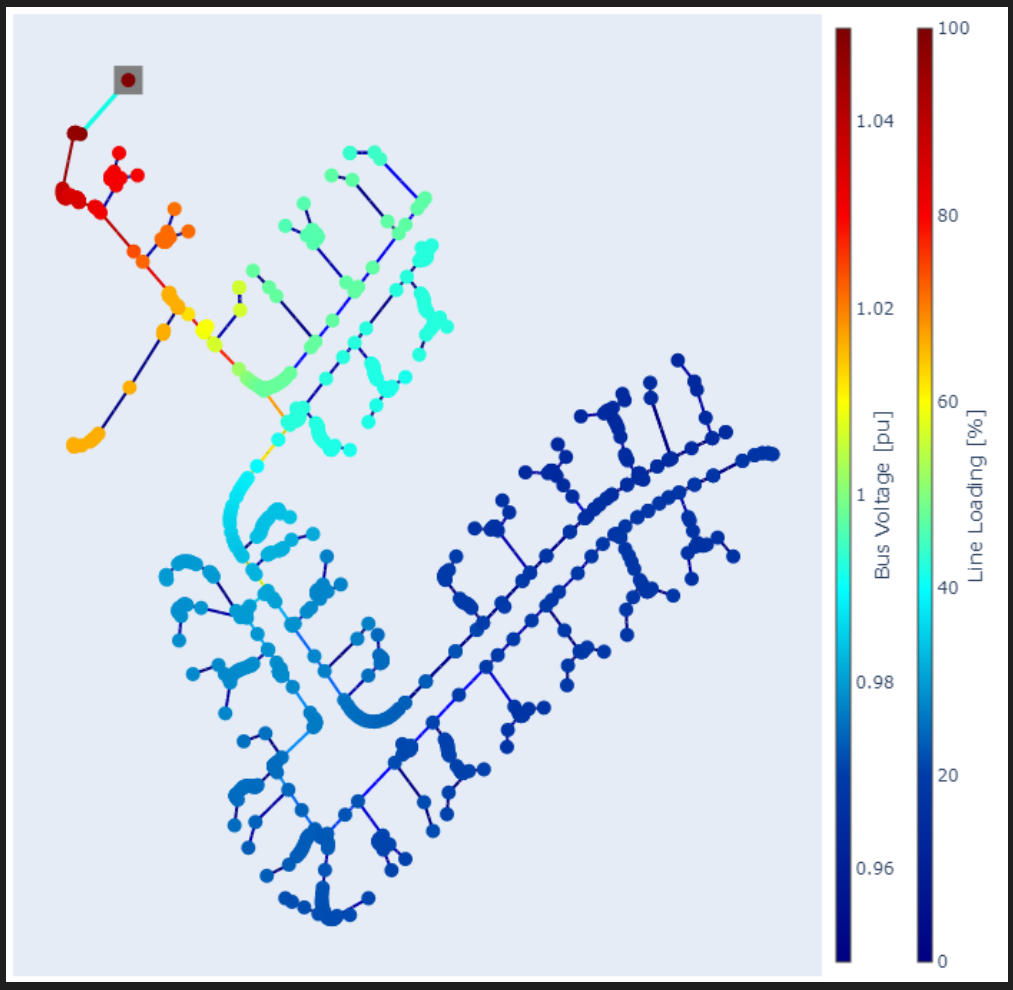

Figure 4.2: blahblah

4.2 Suggested improvements

5 Conclusion

Appendix