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

Albert Ludwig University of Freiburg

Daniil Aktanka

September 1, 2022

Abstract

Contents

1 Introduction

2 Literature Review

Good intro into EV situation and market: [1]
Really good method, but I can't math: [2]

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.
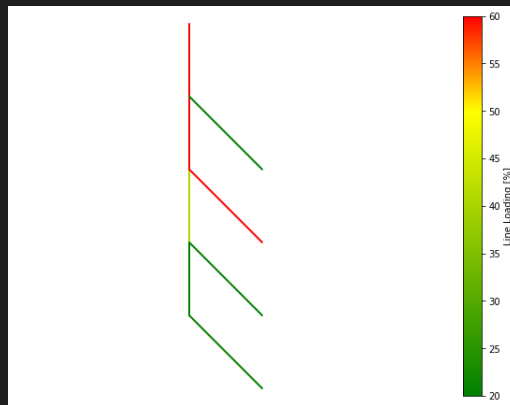

Figure 3.1: An example test power flow calculation plot with line loading results. Here, one static generator is active out of four.

Early on, it was decided to incorporate real-world data as a basis of the simulation. Arguably the best (at the time of writing) data platform for this application is Open Power System Data—more specifically, the contributed household data package. It contains measured time series data for various households and businesses relevant for low-voltage power system modeling. In terms of geographical scope, it incorporates data from 11 households in Konstanz, southern Germany, as part of the CoSSMic project for the optimization of energy systems in smart cities. The gathered data is quite expansive, spanning a time range of over 4 years with a resolution of up to 1 minute measurements, even tracking single-device consumption. Cumulative energy measurements accounted for data gaps in communication problems and "all data gaps are either interpolated linearly, or filled with data of prior days" [ ], though no further detail is provided in this regard. Overall, this was the perfect fit for my simulation aimed at a standard European low-voltage grid.

Using and incorporating the Open Power System Data household dataset proved to be quite a challenge on its own. Details regarding final data analysis are described in section 3.2. Once the data was processed and loaded into the network for further computations, the question became how to formulate and subsequently evaluate simulation scenarios. Hereby, the number of variables to take into account can practically be expanded far beyond the scope of the paper. For instance, an alternative tool considered for the simulation data was the LoadProfileGenerator (LPG), which is a highly-customizable tool for modeling residential energy consumption. LPG performs complete behavior simulation of a household residents in order to generate load curves. To that extent, it would have been possible to take said curves and implement them into our network case, while investigating individual households' behaviors across a community of like-minded, cooperative individuals. However, this was deemed far too comprehensive given the time constraints, but also not as potentially insightful.

Additionally, in order to make best use of the historical data, it was decided to perform time series iteration power flows. In a sense, this provides a similar advantage to LPG functionality by mimicking random human behavior over time, only with no unnecessary explanations behind it. The main idea became to sample a random historical load profile per household and to consider when/how an EV charger could optimally function in time. While this may be a relatively simple question to answer for a single household—given the residents' preferences and habits, it's merely a load optimization problem—the main issue lies with multiple parties involved. In turn, those parties are connected on a network in different ways, bringing topology into account. Thus, time and space both dictate the conditions for a given network.

Eventually, the rather abstract test network got superseded by the benchmark European Low Voltage Test Feeder (from hereon: test feeder) (see fig. 3.2). Three scenarios were chosen for evaluation: critical case, random charging, semi-coordinated charging. The critical case is used for hosting capacity evaluation of our simulation for a single time point, whereas the two other scenarios demonstrate time-dependent charging behavior models. Detailed explanation for this approach is given in section 3.3. From hereon, EV home charging stations will be referred to as static generators in the context of power modeling.
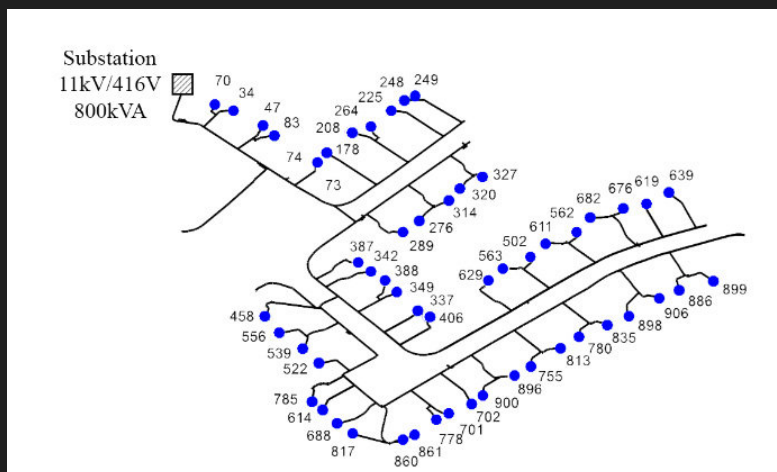

Figure 3.2: European Low Voltage Test Feeder network illustration. The blue dots are the active busses, representing households [4].

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 test feeder 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

5

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 test feeder network, the resulting night dataframes are 721 rows × 55 columns (minutes × 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. (You may need to include a conditional statement for the beginning charging time, otherwise the loop will subtract one minute and cause a set of

sgen columns to remain unfilled. Similarly, you may need to include a conditional statement for the last minute in the time series index, since that too will remain unfilled otherwise.) 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—test feeder 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 5.1 in chapter 5 shows a general overview of the built-in module. 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 (vm p.u.) 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 test feeder.

The network used in the simulation is an alteration of the standard test feeder. The original network itself is 3-phase, which would require a different power flow model—in case of `pandapower`, that would be handled by `pandapower.pf.runpp_3ph.runpp_3ph()`. However, given the right circumstances, it is possible to treat a 3-phase network as single-phase; namely, if we assume that all network loads are balanced. This assumption is plausible and corresponds to most real-world households, to the extent that some power modeling software (such as BLANK) don't even include a separate 3-phase computation option for this reason. With this assumption, all loads and sgens were defined and handled as single-phase in our simulation.

Furthermore, the standard test feeder was (in most cases) not able to support direct application of the household load profiles. During timeseries iteration, the console encounters a exactly-singular matrix error, indicating that the Jacobian cannot be solved for this power flow. In practical terms, this meant that the network was overloading at some point in time. Indeed this point in time was almost certainly converging to the average evening peak load time for this dataset. From past applications of data analysis, with data spanning over 500 days, the average maximum load time (across both households 1 and 2) was calculated to be `19:19:00` in the night-time bracket. The solution was to scale the historical load curves. The resultant network busses, with no sgens active, fall in an acceptable voltage magnitude per unit range between 0.95 and 1.05 for any random simulation.

The night-time bracket was chosen as `18:00:00-06:00:00`. This is based around the common Western world business day hours, such as the standard office hours of 9 a.m. to 5 p.m. in USA. The time bracket is adjusted to account for often earlier starting hours in Europe as well as travel time. The chosen hours are commonly referred to as night time hours in this paper and code, whereby the phrasing is applied somewhat generously.

Another assumption that was made concerns the charging times—these were meant to stay constant in length. MOST CHARGERS ON THE MARKET... MOST AVERAGE EVS... BLAH BLAH, 1 HOUR CHARGING TIME GOTTA EXPLAIN

With these assumptions and adjustments to the test feeder, the following 3 case scenarios were identified and analyzed.

3.3.1   Critical case scenario

This case scenario aims to address the hosting capacity of our test feeder given the most unfavorable circumstances. The evaluation process outline was as follows:

1. Load and prepare dataset

2. Create loads and sgens night dataframes

3. Load and prepare test feeder network

4. Define output writer and run the timeseries

5. Import the voltage magnitude timeseries results

6. Evaluate the all-time minimum load time point

7. Load the network configuration for that time point

8. Calculate hosting capacity at the critical time point

Hereby, at step 8, the sgens are placed in a random network locations for a single point (minute) in time. Timeseries iteration in step 4 is used to locate that point in time, but it does not compute over any sgen values. Based on test simulations, the evaluated hosting capacity never reaches 100% for peak load time, unlike the other two scenarios.

3.3.2   Random time scenario

This case scenario aims to represent completely uncoordinated charging patterns over the night time window period. It's used as a control comparison for the energy community scenario. The evaluation process is similar to the critical case scenario:

1. Load and prepare dataset

2. Create loads and sgens night dataframes

3. Place sgens at random locations in the network

4. Load and prepare test feeder network

5. Define output writer and run the timeseries

6. Import the voltage magnitude timeseries results

7. Evaluate the all-time minimum load time point

Opt: Load network configuration for all-time minimum vm p.u. time point

The main difference from the critical case scenario is that we load sgens in our network before we run the timeseries iteration. Since the sgens charging times are spread across the night window, all sgens can be active at one point or another. This only becomes problematic should multiple sgens (i.e. more than 10–15) happen to be charging at once. At the same time, the household load profiles may offset a potential network overload if the charging convergence happens late a night when the loads are lowest—since most people are asleep and little to no household appliances are running.

3.3.3   Energy community scenario

This case scenario aims to propose the concept of semi-coordinated charging, whose main idea is that of simplicity with minimum requirements. As mentioned in chapter 2, there is no shortage of highly optimized models proposed. Both centralized charging management algorithms and decentralized/distributed charging mechanisms should undoubtedly serve as the goal of EV technology and its future implementation. That being said, these models remain mostly conceptual in nature, owing to their intricacy of operation as well as high upfront resource demands.

Indeed, while there have been tremendous advancements made in making individual grid elements and devices smarter, there's yet to exist a truly unified system at scale. The internet of things (IoT) offers a glimpse into a fully synchronized smart-device architecture but this concept does not transfer to vehicles as readily, due to their mobile nature and dependence on existing infrastructure. Arguably the minimum precondition for large-scale smart EV charging mechanisms is the existence of installed smart meters in residential homes. The spread of smart meter technology has proven to be highly incentivised by progressive energy tariffs and some success has been found in countries across Europe, such as Spain adopting a Real-Time-Pricing (RTP) tariff in 2015. Unfortunately, many countries are still lagging behind the practically-feasible due to their complex governmental policies—Germany in particular has been struggling with residential smart meter roll out, despite its ambitious push towards embracing renewables. Needless to say, there still remain plenty of obstacles in the way of widespread, advanced EV charging systems.

With these considerations in mind, this paper proposes the most basic, simplistic way to improve grid stability through altered charging patterns. This method in fact does not rely on the existence of smart meters or intelligent controllers and can therefore be implemented in any local network (section) in a form of a rudimentary energy community. In the scope of this paper, the concept of energy community is also simplified to its most basic premise—coordinated energy use.

Theoretical considerations

Consider the following theoretical network depicted in fig. 3.3, not dissimilar from the original test case network.
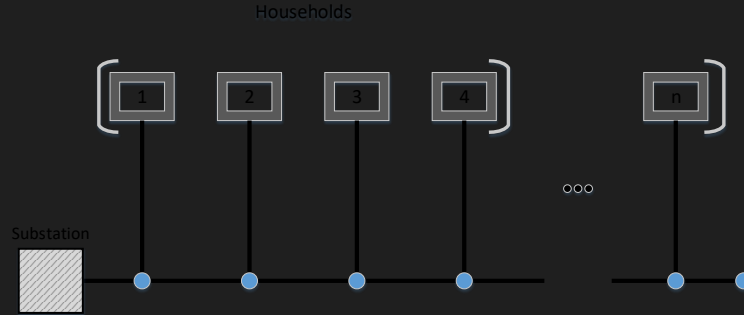

Figure 3.3: Pictured is a theoretical network scenario for linearly placed single-line households. The energy community aims to include every new household member.

The linear topology of the theoretical network is reminiscent of the test feeder. Known as radial operation, this is the most widespread design in both MV and LV networks. Hereby, the

North American system differs from the European in transformer placement and is not discussed further, as the application of the developed model is aimed at the European LV grid, although the results can in some extent be transferable. Furthermore, spot networks (commonly found in business and industry districts) as well as grid networks (typical for downtowns of large cities) are omitted in this theoretical scenario.

The theoretical radial network demonstrates a relatively straightforward load pattern. Every consecutively added household down the line from the substation increases the overall network load; the last household n acquires the network minimum in voltage magnitude in case of constant household load profiles. The goal is then to avoid the critical case scenario of network overload, which is most likely to happen at, or close to, the n'th household bus. Under these spacial considerations, we must distribute the sgen loads temporally so as to minimize the network load. This amounts to two conditions. First, avoid multiple sgens charging at the same time as much as possible—this is the deciding factor between the two conditions. The best case scenario is then an even distribution of sgens in time. Second, should there be a in crossover charging times, the beginning of the network line should be prioritized, since its local voltage magnitudes will be (albeit slightly) lower than those at the end of the transmission line. The crossover of charging times is guaranteed to occur for a sufficiently large n and depends on charging time, which we assume is constant across all households. Next, consider the concept of coordination.

Coordination of energy use, in the context of our energy communities, is begins with coordinating pairs. In the most primitive sense, this can be as simple as an agreement between two neighbors to not charge their EVs at the same time of day. Assuming they both come home from work around the same time and want to immediately plug in their EVs to charge, one neighbor can agree to start charging after the other one is done. This alone will have an obvious beneficial effect on grid stability but would still imply most households charging around peak grid hours in the evening. The more coordinating pairs, the better. Eventually, we can extend this idea to the entire neighborhood, thus forming an energy community. The community decides to agree on dedicated charging times for individual households based on the philosophy described above.

Referring back to fig. 3.3, the energy community decides to allocate charging times iteratively, starting from the household closest to the substation. The first household will take the charging time closest to peak grid hours in the evening, the second will take the next available time slot after the first and so on. Once all the available charging time slots per night have been filled, the time slots will cycle.

Test feeder application

In the case of the test feeder, the allocated sgen charging times and cycles can be seen in table 3.1. On average we can expect 55 sgens/12 hours ≈ 4.583 sgens per given hour in the night frame, which correlates with our results.

For spacial allocation, the sequence of predefined sgen locations in `pandapower` test feeder network was used. Additionally, line loading is generally expected to be around 1% per household line and was shown not to be the limiting factor for hosting capacity calculation.

start time	end time	sgens active
18:00:00	18:59:00	5
19:00:00	19:59:00	5
20:00:00	20:59:00	5
21:00:00	21:59:00	5
22:00:00	22:59:00	5
23:00:00	23:59:00	5
00:00:00	00:59:00	5
01:00:00	01:59:00	4
02:00:00	02:59:00	4
03:00:00	03:59:00	4
04:00:00	04:59:00	4
05:00:00	05:59:00	4

Table 3.1: Energy community charging times for test feeder network.

4 Results and Discussion

4.1 Results

4.1.1 Hosting capacity

	bus_0	bus_1	bus_2	...	bus_904	bus_905	bus_906
count	721.000	721.000	721.000	...	721.000	721.000	721.000
mean	1.050	1.050	1.050	...	1.046	1.046	1.046
std	0.000	0.000	0.000	...	0.001	0.001	0.001
min	1.050	1.050	1.050	...	1.040	1.040	1.040
25%	1.050	1.050	1.050	...	1.045	1.045	1.045
50%	1.050	1.050	1.050	...	1.047	1.047	1.046
75%	1.050	1.050	1.050	...	1.047	1.047	1.047
max	1.050	1.050	1.050	...	1.048	1.048	1.048

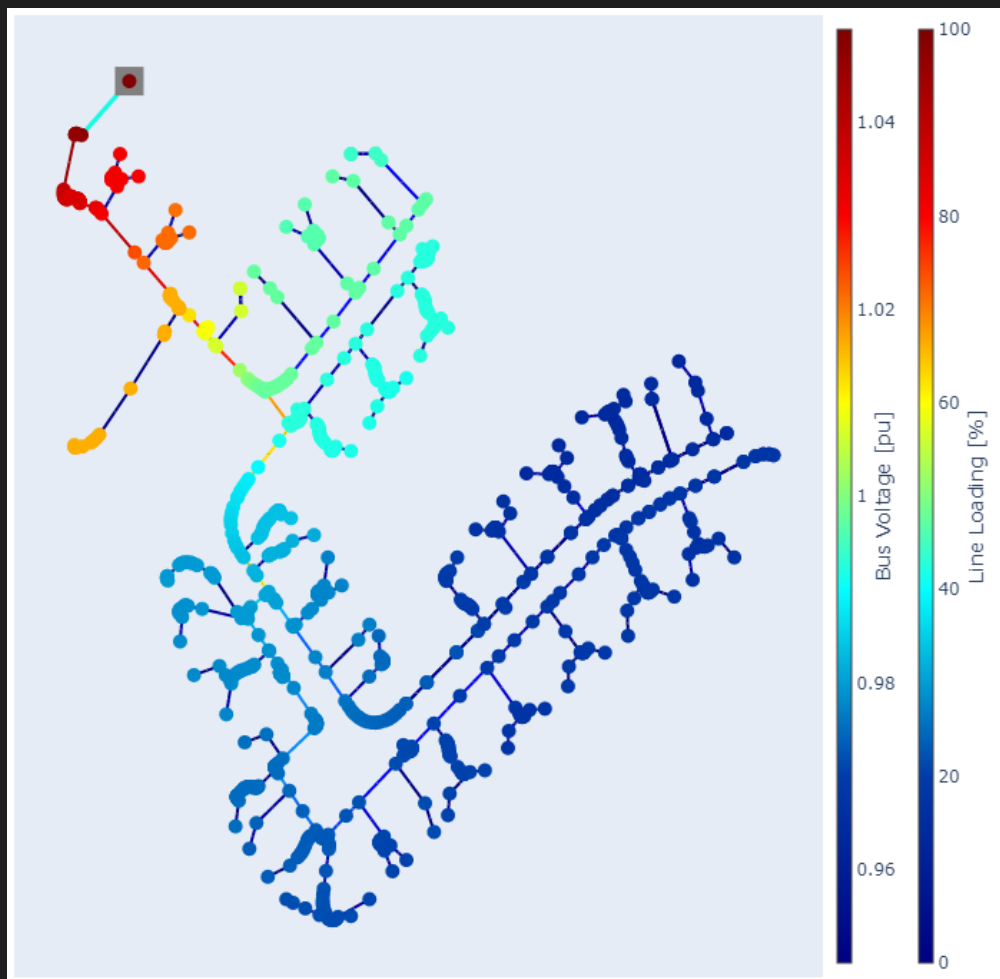Table 4.1: Example simulation timeseries results.

Figure 4.1: blahblah

Limiting factor is the voltage (line loading never exceeds 100 pct before vm pu fails)

Scenario	Random	Community	Control
simulations count	10	10	10
avg voltage [vm p.u.]	0.949	0.974	1.010
bracket deviation [%]	100.8	75.9	39.5

Community is approximately 24.9% more effective than random case.
Load bracket deviation percent is measured:

$$\text{Deviation } [\%] = \frac{\text{upper bracket} - \text{average voltage}}{\text{upper bracket} - \text{lower bracket}} \times 100 \tag{4.1}$$

4.2   Suggested improvements

4.2.1   Method

The energy community spatio-temporal algorithm is in no way optimal and is an extreme over-simplification of the problem. Its main purpose is to serve as a starting point when there is none, and to serve as a demonstration for what said minimum effort can achieve in the context of grid stability.

Spacial:
relies on correct ordering of busses for iteration and does not account for significant deviation or branching from linear topology

Temporal:

4.2.2   Code

splitting function could waste less data (first day, last two days in list)
energy comm func could start the cycle at peak time (requires nested conditional for loop for both times and community function)
the matrix is exactly singular—jacobian could not be solved: no clue

5 Conclusion

Bibliography

[1] H. Das, M. Rahman, S. Li, and C. Tan, "Electric vehicles standards, charging infrastructure, and impact on grid integration: A technological review," *Renewable and Sustainable Energy Reviews*, vol. 120, p. 109618, mar 2020.

[2] G. Gruosso, G. S. Gajani, Z. Zhang, L. Daniel, and P. Maffezzoni, "Uncertainty-aware computational tools for power distribution networks including electrical vehicle charging and load profiles," *IEEE Access*, vol. 7, pp. 9357–9367, 2019.

[3] O. P. S. Data, open Power System Data is provided by Neon Neue Energieökonomik, Technical University of Berlin and ETH Zürich. [Online]. Available: https://data.open-power-system-data.org/household_data/

[4] A. I. Nousdilis, A. I. Chrysochos, G. K. Papagiannis, and G. C. Christoforidis, "The impact of photovoltaic self-consumption rate on voltage levels in LV distribution grids," in *2017 11th IEEE International Conference on Compatibility, Power Electronics and Power Engineering (CPE-POWERENG)*.   IEEE, 2017.
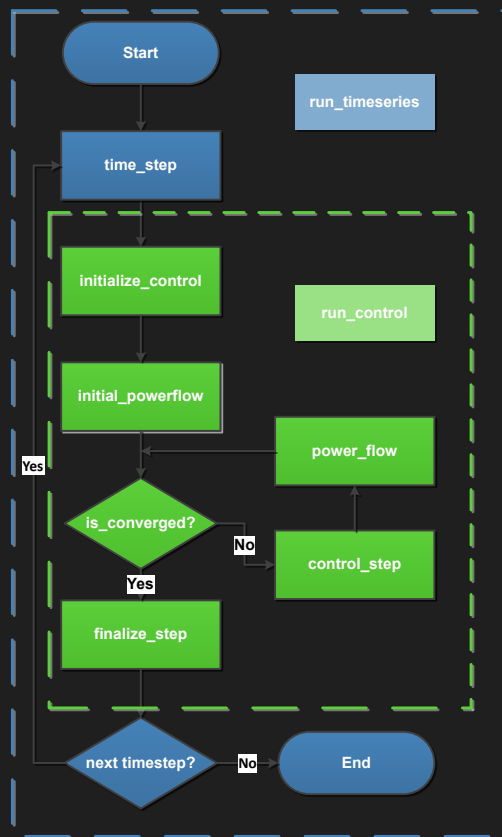
Appendix

Figure 5.1: 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.