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

Albert Ludwig University of Freiburg

Daniil Aktanka

September 13, 2022

Abstract

The global trend towards electric vehicle (EV) adoption poses numerous challenges for the traditional electric grid operations and infrastructure. This paper examines the stability of low-voltage European grids in the context of EV integration. For this purpose, a power flow analysis is performed based on historical data of residential load profiles. A critical case scenario is used for computing the hosting capacity's upper and lower limits. The results show $\approx 13\,\%$ and $\approx 21\,\%$ grid hosting capacities for worst and best case scenarios respectively. The coordination of individual households within the grid is explored via a newly proposed spacio-temporal charging algorithm for a basic energy community. A comparison between a random charging pattern and a semi-coordinated energy community charging pattern is made based on a set of timeseries simulations. The results suggest that the energy community algorithm is $\approx 30\,\%$ more effective than the random case; the proposed energy community model is suggested as an intermediary solution for a transition towards future smart grids.

List of Tables

List of Figures

Contents

1 Introduction

Electrification of transportation represents a major role in a move towards a sustainable future of energy systems. It brings about a number of interdisciplinary challenges, requiring simultaneous coordination from multiple fields of study. This paper aims to address one of these components from the perspective of the power distribution network—namely, the hosting capacity limitations of the electric grid.

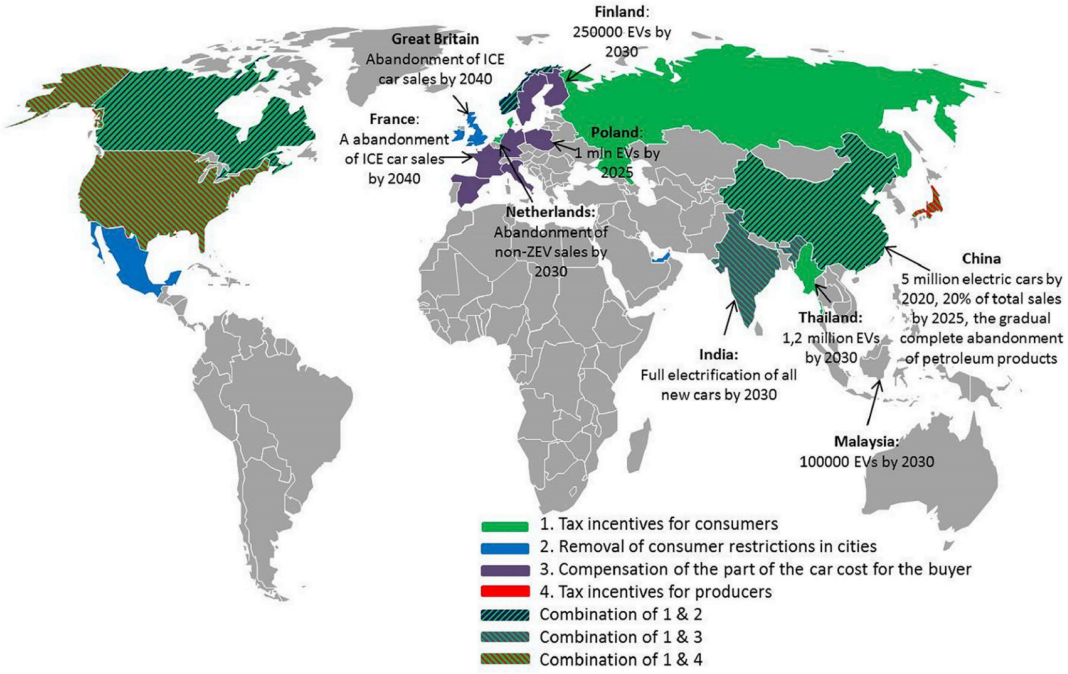


Figure 1.1: Electric vehicles state support mechanisms and national goals across the world. [1].

The transition from internal combustion engine (ICE) vehicles to electric vehicles (EVs) has entered world-wide adoption. A large portion of the western world has already set hard targets in order to facilitate this transition, ranging from tax incentives to outright abandonment of ICE car sales within the next 20 years (see fig. 1.1). While the EV technology has made tremendous advancements over the past several decades—especially in terms of mobility range—this alone may not be enough for complete integration of EVs. Full-scale transition towards EV adoption would require the underlying infrastructure to be able to support charging, discharging (Vehicle-to-Grid) and coordinated management of EVs. However, contrary to the rapid development

of EV technology, the electric grid remained relatively unchanged since the years of its early inception. The coveted transition towards the smart grid is mostly not feasible as a single-step on a large scale, which begs the question of how well does the existing system accommodate the transition towards EV integration.

This paper aims to investigate this question from multiple angles. The main premise is a simple substitution of ICE vehicles for EVs in the context of the current grid under a multitude of assumptions. The scope of this study is limited to the low-voltage grid, to which all of the EV charging is hypothetically constrained. Furthermore, no public charging stations were considered in this study. Despite their growing popularity, the goal of this paper was to investigate the minimum investment conditions for near-future EV charging scenarios; as such, only residential home chargers were considered.

The main method employed in this paper's study is a power flow analysis using randomized historical load profiles. Hereby the biggest block of information processing was arguably the data analysis, the details of which will be described first. Next, the concepts of various charging strategies will be introduced in a form of three scenarios: critical case, random time and energy community. Finally, the results will be presented in 2 parts: hosting capacity evaluation and the comparison between random and energy community scenarios. Additionally, suggested improvements will be provided for multiple areas of this paper.

2 Related Work

2.1 Literature review

The inspirational starting point for this thesis was the paper "Electric vehicles standards, charging infrastructure, and impact on grid integration: A technological review" by H.S. Das et al. [2]. It provides a comprehensive overview of the current EV status and the challenges of EV integration across multiple organizational fields. Notably, it classifies EV charging control strategies across different attributes, as seen in fig. 2.1. Hereby the concept of coordinated vs uncoordinated charging strategies became one of the main focal points for this thesis paper, leading to the introduction of the semi-coordinated charging method. Further points of interest in H.S. Das's paper include the distinction between centralized and decentralized control architectures, as seen in fig. 2.2. The researchers claim that decentralized/distributed mechanisms are superior in terms of computational complexity, real-time implementation and offer more user convenience. In a similar fashion, the paper "Integration of electric vehicles and management in the internet of energy" by K. Mahmud et al. discusses the future grid and the essentials of adaptive charging infrastructure for EV integration, noting the importance of coordinated EV management [3].

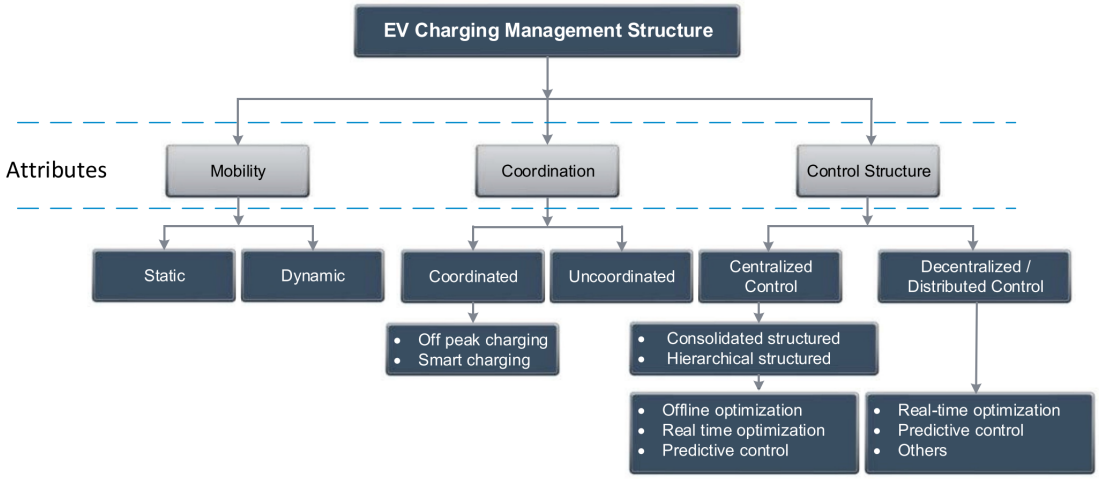


Figure 2.1: Classification of control strategies used in EV charging system [2].

A more advanced variation of this paper's method can be found in an IEEE Access article "Uncertainty-Aware Computational Tools for Power Distribution Networks Including Electrical Vehicle Charging and Load Profiles" by Giambattista Gruosso et al. [4]. It describes an efficient, uncertainty-aware load flow analysis reliant on generalized Polynomial Chaos and Stochastic

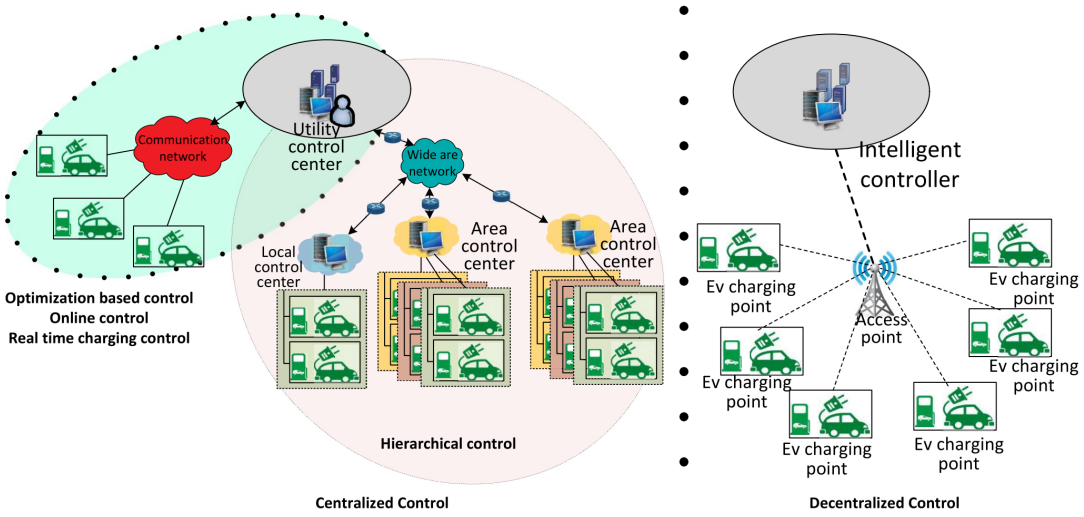


Figure 2.2: Centralized and decentralized control architecture of EV charging systems [2].

testing methods. The authors use historical load profiles to simulate variation in both residential loads and EV charging profiles, which was additionally compared against a Monte Carlo simulation. Similarly to this thesis, an example network of choice was the European Low-Voltage Test Feeder, whereby all of the loads were assumed to be single phase. The result was a technique able to account for real data-based load profiles for network quality assessment.

For the computation of distribution grid hosting capacity, consider the paper "Optimization-based distribution grid hosting capacity calculations" by Mansoor Alturki et al. [5]. The authors propose an optimization-based method for determining the hosting capacity using linear power flow equations. Unlike traditional nonlinear models, the proposed method does not rely on iteration in order to find a near-optimal hosting capacity solution. The obtained solution at operating point provides access to both active and reactive power flow information, additionally accounting for seasonal load variations. A more common, iterative approach—such as the one used in this paper—can be found in "EV Hosting Capacity Analysis on Distribution Grids" by Priti Paudyal et al. [6].

The paper "Applying responsible algorithm design to neighbourhood-scale batteries in Australia" by H. Ransan-Cooper et al. is an interdisciplinary study that examines the systemic concerns to design and operation of energy algorithms. The study focused on comparing 6 different metrics used for algorithm design, whereby the algorithm of interest is the timer algorithm. "The timer algorithm is designed to challenge assumptions in techno-economic optimization by explicitly forgoing economic or environmental optimization to instead optimize for explainability as a response to citizen concerns about a lack of transparency in the energy system" [7]. Interestingly, the timer algorithm employed the same night window as the one used in this paper. The researchers claim that "it is possible that non-'optimal' timer algorithms may meet public needs because they relieve concerns about complexity and a lack of transparency" [7]. This statement directly supports the motivation behind this paper's energy community algorithm.

The next section will summarize the basics of the power flow analysis and nonlinear models.

2.2 Power flow analysis

Power flow (or load flow) study is a numerical analysis of the flow of electric power in a network. Its objective is to calculate the bus voltages—consisting of magnitudes and angles—given the network conditions defined by loads, generation and other variables. The busses are classified into 3 types: load, generation and slack. Load (or PQ) busses have no generators connected to them. Conversely, all busses with at least one generator connected to them are called generator (or PV) buses, with one arbitrary exception—the slack bus. Also known as the swing or reference bus, the slack bus is used to compensate for system losses by either emitting or absorbing active and reactive power within the system. For a system consisting of n buses and g generators, there are $2(n-1)-(g-1)$ unknowns. In order to solve for these unknowns, a special admittance matrix is used to construct real and reactive power balance equations. This results in a system of nonlinear equations which can be solved in a variety of methods. The most common method is the Newton-Raphson method.

The Newton-Raphson technique is a method of successive approximation based on the Taylor's expansion approximation used to solve $f(x) = c$. In the context of power flow problems, the unknown variables x are: voltage magnitude at load busses ($|V_i|$), angles at load busses ($\angle\delta_i$) and voltage angles at regulated buses (δ_i). The specified quantities c are both the net scheduled (known) injected real power (P_i^{sch}) and the net scheduled injected reactive power (jQ_i^{sch}).

The iterative values of real power are calculated using:

$$P_i^{[k+1]} = \sum_{j=1}^{n} |V_i|^{[k]} |V_j|^{[k \text{ or } k+1]} |Y_{ij}| \cos\Big(\Theta_{ij} - \delta_i^{[k]} + \delta_j^{[k]}\Big) \tag{2.1}$$

The iterative values of reactive power are calculated as:

$$Q_i^{[k+1]} = -\sum_{j=1}^{n} |V_i|^{[k]} |V_j|^{[k \text{ or } k+1]} |Y_{ij}| \sin\Big(\Theta_{ij} - \delta_i^{[k]} + \delta_j^{[k]}\Big) \tag{2.2}$$

whereby Y is a symmetrical admittance matrix, consisting of admittance values of both lines and busses. The Newton-Raphson power flow formulation can be solved using the following eq. (2.3). See "Computational Models in Engineering" for more details [8].

$$\begin{bmatrix} \Delta P_i^{[k]} \\ \Delta Q_i^{[k]} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial P_i}{\partial \delta_i}^{[k]} & \frac{\partial P_i}{\partial |V_i|}^{[k]} \\ \frac{\partial Q^i}{\partial \delta_i}^{[k]} & \frac{\partial Q^i}{\partial |V_i|}^{[k]} \end{bmatrix}}_{\text{Jacobian matrix } f' = \begin{bmatrix} J_{P\delta} & J_{P|V|} \\ J_{Q\delta} & J_{Q|V|} \end{bmatrix}} \begin{bmatrix} \Delta \delta_i^{[k]} \\ \Delta |V_i|^{[k]} \end{bmatrix} \tag{2.3}$$

In this paper, we will be using the standard Newton-Raphson method. Some of the other power flow analysis methods include the Gauss-Siedel method, the fast-decoupled-load-flow method, the backward-forward sweep (BFS) method as well as the DC power flow technique.

3 Goal and Method

3.1 Approach

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 countless iterations in terms of functionality as well as network application (down to the last two weeks before deadline). The original testing models were performed on a simple test network with 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 their potential real world application and the required scope of work.

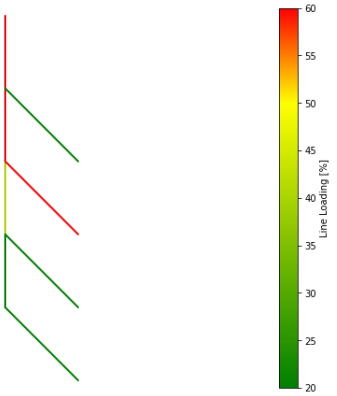


Figure 3.1: An example power flow calculation plot with line loading results. Here, one static generator is active out of four. The example network has generic coordinates instead of geographical, as seen in the rest of the plots in this paper.

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 timeseries 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 from communication problems and "all data gaps are either interpolated linearly, or filled with data of prior days" [9], though no further detail is provided in this regard. Overall, this was the perfect fit for this paper's simulation aimed at a

standard European low-voltage grid of a residential area.

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 of load 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 of power flows. In a sense, this provides a similar advantage to LPG's 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 abstract example 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 singular time points, 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 (negative) static generators in the context of power modeling.

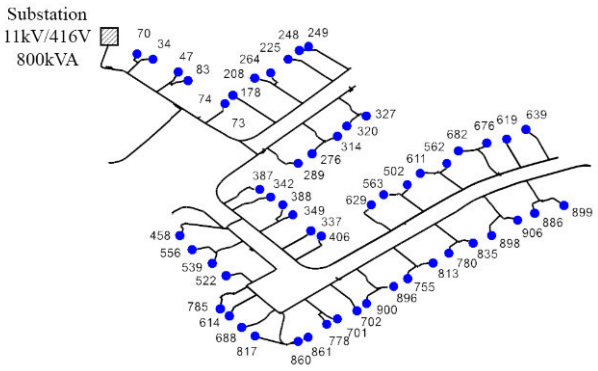


Figure 3.2: European Low Voltage Test Feeder network illustration. The blue dots are the active busses, representing households [10].

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 libraries. Data handling was done using the `pandas` library, whereas the network simulation was performed using the `pandapower` toolbox library.

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` library. 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 (`household_data_1min_singleindex_filtered.csv`) 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 following 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). If you are using class variables for storing the dataframe list, don't forget to clear it before each sampling run when performing bulk simulations (see section 3.2.3).

Datetime parsing function: This function converts a segment of the imported data into a specific, time-indexed dataframe. First, we parse the 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 author's 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. Such datetime manipulations are very commonplace for this simulation.

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 $\times$ 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 units 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 iteratively 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 (see chapter 5 for caveats), 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 minute, 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 timeseries 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 its class name.

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 section 3.3). 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.

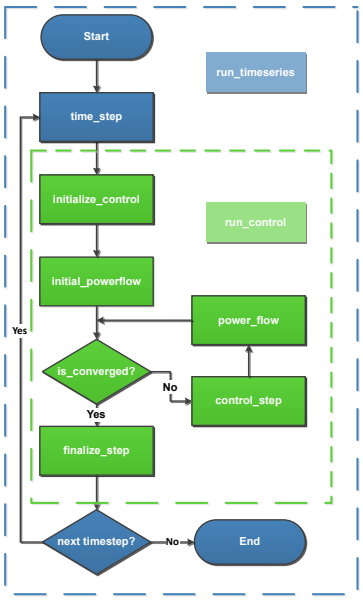


Figure 3.3: The diagram shows the timeseries loop of the timeseries module. The basic functionality is compute the power flow for each time step using the given values—such as sgens or loads. Step-wise convergence will depend on several factors, such as the network parameters (i.e. load magnitudes) as well as the algorithm used for solving the power flow problem. For more module details, see the `pandapower` wiki.

The controller is a function whose purpose is to update the network values iteratively from a lookup 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 3.3 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.

3.2.3 Additional files

In addition to the formally described python class file and jupyter data processing notebooks, several other files were used for the simulation project. They serve as an extension to the already discussed functionality and will therefore not be covered in similar detail but deserve mentioning nonetheless.

Bulk simulation: Due to the random nature of the simulated load profiles selected from historical data, the results are prone to some degree of deviation. In particular, the network bus voltage minimum and maximum values in time are affected. While such deviations may not have a direct impact on the analysis of the hosting capacity evaluation, the two other scenarios—random charging time and energy community charging pattern—are more severely affected. Given the fact that our aim is to make a comparison between the random and the community scenarios, the simulation needed to be performed multiple times in order to obtain a reliable average and a convincing statement. For this purpose, a python file was created to perform bulk simulations. It follows through the same general steps described above with the inclusion of saving the vm p.u. minima (our limiting factor) to an external file per simulation per scenario. The loops are performed for 3 scenarios: random and community, as well as control, which does not include any active sgens. The results are loaded in and processed in jupyter and can be found in section 4.2.

Average minima and maxima times: As supplementary part of the analysis, the time extrema for the entire dataset were calculated. The goal was to compute the time point of highest and lowest network bus loads. The computation was performed in jupyter and follows the same data parsing principles described earlier with the addition of a running average. This is required in order to analyze the general trend and not the outlying points; for this purpose, a rolling average of 10 minutes was found to give the best results. While this computation does not directly affect the core simulation, it does justify the chosen time window as well as the starting point of the energy community algorithm. See chapter 5 for how these two points can be used to improve the proposed model.

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. First, consider the following assumptions.

The network used in the simulation is an altered version 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 don't 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. The network was furthermore scaled down from 3-phase by dividing all the bus voltages by $\sqrt{3}$ to arrive at its single-phase equivalent.

The night-time bracket was chosen as `18:00:00-06:00:00` and serves as the allowed time frame for EV charging during the night. This night time 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. Hereby the EVs are assumed to be used for transportation to and from work, where charging is assumed to not be available. The chosen hours are commonly referred to as night time hours in this paper and code, even if the phrasing is applied somewhat generously.

One of the key factors of the simulation was the chosen sgen value. This corresponds to the charging rate of EVs, measured in kW. In terms of voltage levels—and by extension, the EV charging speed—AC charging is commonly separated into three levels. "The Levels 1 and 2 charging facilities can be installed in a private location while setting up of Level 3 charging facilities, involving separate wiring and transformer, requires permission from utility providers and are usually built in public charging stations" [2]. For this reason, level 2 charging was chosen. Most home charging stations that support level 2 charging usually provide an output of around 7 kW. The most frequent figure on the market as of this writing appears to be 7.7 kW, which was is why it was chosen as the sgen value.

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 and maximum load time points
7. Load the network configuration for those time points

8. Calculate hosting capacity at both critical time points

Hereby, at step 8, the sgens are placed in a random network locations for two points (minutes) in time. Timeseries iteration in step 4 is used to locate those points in time, but it does not compute over any sgen values. The hosting capacity is expected to be under 100 % for this scenario.

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 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 a relatively large number of sgens 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 section 2.1, 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.

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 simplistic way to improve grid stability through altered charging patterns. This method 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. As such, the method can be viewed as the first stepping-stone in transition towards smart grid EV charging.

Theoretical considerations

Consider the following theoretical network depicted in fig. 3.4, not dissimilar from the original test case network.

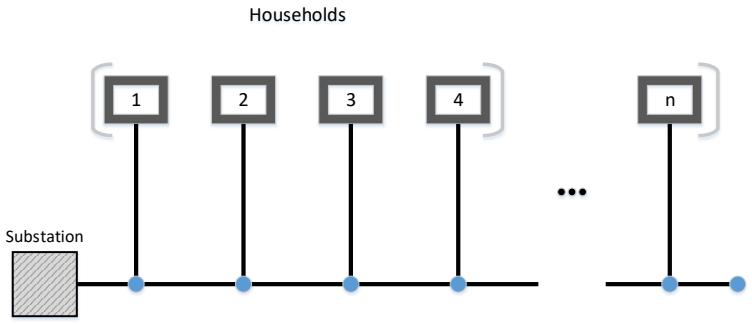


Figure 3.4: A theoretical network scenario for linearly placed single-line households. The energy community aims to include every new household member.

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 medium-voltage (MV) and low-voltage (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. Given these spacial considerations, we must also 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 (generally) is an even distribution of sgens in time. Second, should there be a crossover

in 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, begins with communication between multiple parties. 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. 3.4, 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 available charging time slots per night have been filled, the time slots will cycle times (the same way they were initially filled) for new sgens.

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 $55\,\text{sgens}/12\,\text{hours} \approx 4.583$ sgens per given hour in the night frame, which correlates with our results.

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 temporal distribution of sgens.

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.

4 Results and Discussion

The simulation results can be divided into two groups: hosting capacity and random vs energy community scenarios comparison. The hosting capacity results feature single simulation examples chosen at random, as well as a group average. The random vs energy community scenarios comparison results are based on a set of simulations and consequent group averages.

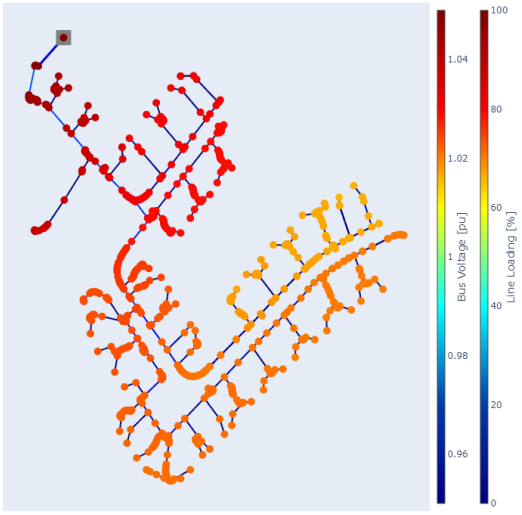
4.1 Hosting capacity

Hosting capacity calculations began by computing the timeseries with no sgens active. The network bus load example results can be seen in table 4.1. As expected, the the voltage magnitude per unit drops with each bus load—hereby the extrema are sometimes shared across multiple busses (even at higher significant figure counts). The time points for voltage magnitude extremes are unique however, which leads to two cases: minimum bus vm p.u and maximum bus vm p.u.

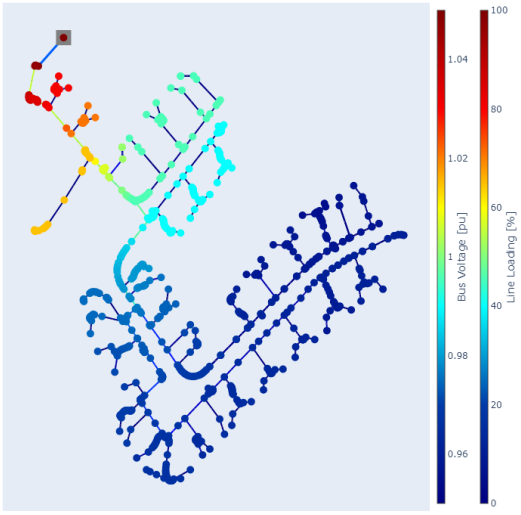
	bus_0	bus_1	bus_2	...	bus_904	bus_905	bus_906
count	721.000	721.000	721.000	...	721.000	721.000	721.000
mean	1.050	1.050	1.049	...	1.031	1.031	1.031
std	0.000	0.000	0.000	...	0.006	0.006	0.006
min	1.050	1.049	1.049	...	1.014	1.014	1.014
25%	1.050	1.050	1.049	...	1.027	1.027	1.027
50%	1.050	1.050	1.050	...	1.033	1.033	1.033
75%	1.050	1.050	1.050	...	1.036	1.036	1.036
max	1.050	1.050	1.050	...	1.041	1.041	1.041

Table 4.1: Example simulation timeseries results. The first bus's vm p.u. is at the upper acceptable bracket limit (both minima and maxima), whereby the last bus's vm p.u. minima is close to the middle of the bracket, which is optimal. This indicates that historical network loads are well suited for this network.

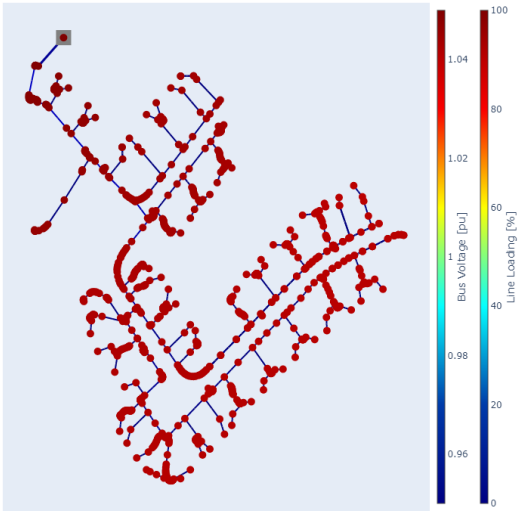
The example network graphs for hosting capacity time extremes calculation are shown in fig. 4.1. Visually, we can see a big difference in color profiles for cases with different starting time points. Figure 4.1a shows significant line stress in the bottom two branches of the network (seen in yellow), as opposed to fig. 4.1c where the entire network is stabilized, implying less deviation between bus loads. Hereby we expect a greater hosting capacity for the maximum bus voltage time point case due to its color profile homogeneity (shown in red) and a bigger maximum range of vm p.u., which would allow to accommodate more sgens at the lower branches. As far as filled sgen hosting capacity graphs are concerned, there's little visual difference between fig. 4.1b and fig. 4.1d at this scale. For more insight, consider numerical data in table 4.2 and table 4.3.



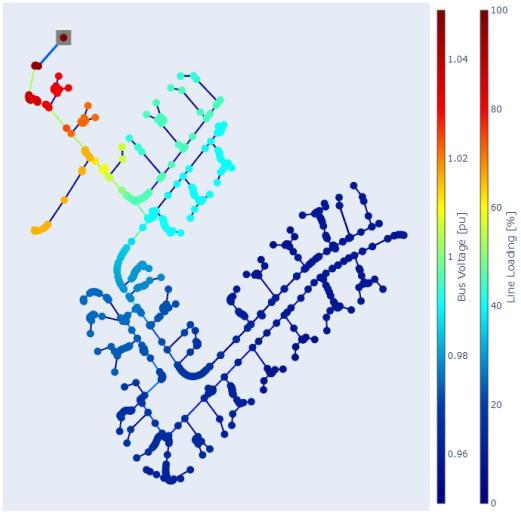
(a) No sgens at min bus voltage



(b) Filled sgens at min bus voltage



(c) No sgens at max bus voltage



(d) Filled sgens at max bus voltage

Figure 4.1: Comparison between hosting capacity during minimum (subfigures (a) and (b)) and maximum (subfigures (c) and (d)) load time cases with and without sgens. These are singular simulation examples.

The details of time point cases for each example network are shown in table 4.2. We can see a significant difference in minimum vm p.u. values between the "Min bus empty" and "Max bus empty" cases. This is a numerical indication of a greater hosting capacity potential for the maximum time point case. For the columns "Min bus full" and "Max bus full", we observe no significant difference between the two min values—this indicates that the maximum hosting capacity has been reached in both cases at a vm p.u. lower bracket.

	Min bus empty	Min bus full	Max bus empty	Max bus full
count	907.000	907.000	907.000	907.000
mean	1.020	0.980	1.042	0.976
std	0.009	0.022	0.002	0.021
min	1.005	0.953	1.039	0.950
25%	1.014	0.963	1.040	0.963
50%	1.018	0.975	1.041	0.970
75%	1.025	0.994	1.043	0.987
max	1.050	1.050	1.050	1.050

Table 4.2: Example data of hosting capacity voltage magnitudes in detail.

The key results of the hosting capacity simulations are summarized in table 4.3. This table contains average values based on 10 simulation runs.

	Minimum bus voltage	Maximum bus voltage
number of sgens	7.65	11.36
total capacity [%]	13.14	20.75

Table 4.3: Hosting capacity averages for 10 runs.

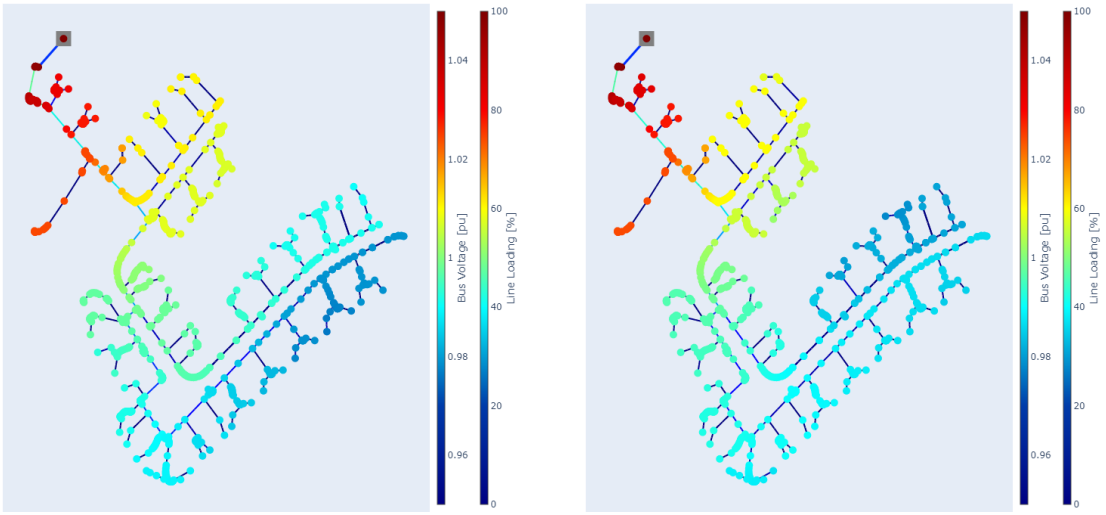
The scenario results show a rather low hosting capacity for both time point cases. As expected, the best case of maximum bus voltage has a better hosting capacity—about 8 % more total capacity than the worst case of minimum bus voltage. In a way, this is alarming because the probability of a network overload is definitely not negligible: it only takes $\approx 13\,\%$ of all households to charge their EVs during peak hours to cause a potential brownout/blackout. Furthermore, this time point is the most probable charging time point for any given individual under our assumptions of European work hours; without the presence of smart meters, a person is most likely to park their EV and set it to charge immediately. Considering the habitual nature of human actions, a network overload is only a matter of time in this case.

That being said, the issue can be easily avoided with relatively little intervention. Undoubtedly, all EV chargers do (and if not, will soon) include a time setting option, which is very easy to implement. Hereby, an EV owner only needs to enable and setup the charging hour parameter once, which is a matter of seconds. At the same time, one cannot deny that not everyone will bother to do so, nor that any potential software/firmware updates will reset the setting. As such, one is able to compensate for technological shortcomings (i.e. lack of smart EV charging) with conscious human intervention (i.e. manually setting the EV to charge outside of peak hours) and vice versa.

With this discussion we arrive at the transitional phase of semi-smart charging and the comparison between uncoordinated and semi-coordinated scenarios.

4.2 Random vs energy community

This section puts the energy community algorithm to the test against a random charging pattern. A representative graphical result is shown in fig. 4.2.



(a) Random scenario

(b) Energy community scenario

Figure 4.2: Graphical comparison between random and energy community scenarios. These are two random example simulations.

The random scenario represents an uncoordinated charging pattern with all 55 households charging at some point during the night window. While the spacio-temporal allocation of sgens is random, we can expect results generally similar to those shown in fig. 4.2a. A defining characteristic of this scenario is the local maximum line load on the lowest branch (seen in darker blue). This corresponds with our theoretical network model characteristic in section 3.3.3 and outlines one of the biggest weaknesses of radial operation topology. It is reassuring to then see a visual improvement in this area in fig. 4.2b. The lower branch load has a considerably more even color gradient indicating a more even distribution of loads at a lower voltage magnitude. Interestingly, the local minimum for the energy community scenario is located on a different branch to that of the random community scenario, slightly higher up the sgen list (closer towards the substation). Even then, that upper branch is still showing a lower bus voltage throughout in comparison to the random scenario's lower branch.

Scenario	Random	Community	Control
simulations count	100	100	100
avg min voltage mag [p.u.]	0.941	0.971	1.010
bracket deviation [%]	108.6	79.2	40.3

Table 4.4: Comparison between random and energy community scenarios.

To obtain a more concrete set of results, bulk simulation was performed. The results are summarized in table 4.4. The raw data gives us the average minimum vm p.u. for both random

and energy community scenarios; an additional control scenario represents a test feeder with no sgens active for comparison. Notably, the random charging pattern on average leads to a network overload (0.941 vm p.u.), whereas the energy community scenario doesn't (0.971 vm p.u.). The control value falls within 1% of the optimal network bus load value, indicating a good choice of parameters. Total simulation runtime for 3 scenarios, 100 runs each, was 3 hours and 40 minutes (with no hardware bottleneck).

The results are additionally interpreted via the so-called bracket deviation metric in this paper. Recall that the load bracket is defined as the acceptable range of bus voltage magnitude per unit. Following common network guidelines, it's taken as the open interval $(0.95\,, 1.05)$. Load bracket deviation percent hereby is defined as:

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

Taking the difference of two brackets, we can show that the energy community scenario is approximately 29.4% more effective than random charging scenario on average. This was a pleasantly surprising result, given how rudimentary and overly-simplified the energy community algorithm is. The next section will further explain the simulations' shortcomings and suggest various improvements.

5 Conclusion

This paper presents several key findings on the topic of EV integration. Firstly, the hosting capacity of a typical European low-voltage grid is surprisingly low ($\approx 13\,\%$) for a worst-case scenario. This serves as a strong motivation for further research and investment required in the topic of grid stability. Secondly, as part of a solution to this problem, this paper introduces a simple but relatively effective spacio-temporal algorithm for EV charging. The semi-coordinated charging algorithm was proven to be more effective ($\approx 30\,\%$) than the uncoordinated charging scenario, which is surprisingly good given its extreme simplicity. One can argue in favor of unoptimized algorithms when taking into account the public's perception of energy transparency. An algorithm such as this one could prove to be a valuable stepping-stone for the purposes of transition towards the future of smart grids and EV integration. Additionally, consider the following limitations of this paper's work.

Perhaps the biggest uncertainty in the simulation parameters was the charging window. Ideally, the charging window should represent the average charging time for a single household under the most common circumstances. If one were to calculate said average from real-world statistics, one would have to consider (at least) the following average values: daily driving distance, daily time spent in traffic, typical EV energy consumption/efficiency and typical battery discharge rate. This paper assumes that the end user will continuously top-up the EV battery on a daily basis. Hereby, the EV battery technology is the key factor to widespread EV adoption. The field is rapidly developing, which makes it hard to predict precise technical details, such as maximum charging rates. At the same time, since it is highly unlikely to expect Level 3 DC charging reach widespread household integration, our assumption for 7.7 kW charging rate remains plausible. Workplace or public charging stations are most likely to see installation in the near future, which could reduce the total number of household EV charging stations drastically. Lastly, a completely thorough analysis could additionally factor in energy tariffs and their effects on person's charging habits.

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 for when there is none, and to serve as a demonstration for what minimum effort can achieve in the context of grid stability. Consider the shortcomings of both dimensions separately.

Spacial: The current algorithm relies on correct ordering of busses for iteration and does not account for significant deviation or branching from linear topology. This implies that the results depend significantly on manual ordering of busses, which can become unintuitive with enough complexity. A possible solution would be to implement a tree path length comparison algorithm, though certain cases such as branches of equal lengths may be difficult to account for. (This is even possible to do in `pandapower` using `calc_distance_to_bus()` function.) Alternatively, one could limit the current algorithm to smaller sections of the network and run it locally, since it would still optimize for distance from substation. As last resort, constrained brute-forcing

different bus ordering cases should not take long, given the relative simplicity of the simulation.

Temporal: There are two time-based shortcomings of the algorithm. Firstly, the programmed value for sgen allocation starting time (`18:00:00`) does not match the computed historical dataset peak-time (`19:23:00`) as dictated in section 3.3.3. A function for proper sgen charging time was actually written, which involved nested time-conditional `for`-loops. However the updated function was eventually discarded due to the complexity of new code, as the energy community function would also need to be rewritten and there was simply not enough time.

Secondly, average sgen distribution in time is not optimal. Ideally, most sgens would be shifted towards the time of vm p.u. minimum, which was computed to be `02:26:00` based on historical data. Referring back to table 3.1, we can see that there are 5 sgens active during the `18:00:00`–`00:00:00` interval, followed by the remaining time slots at night to be filled with 4 sgens—this ordering should be reversed. Unfortunately there is no easy fix that comes to mind, since it would require one to separate the spacial from the temporal, thus writing a new algorithm altogether. What this simulation ultimately shows however, is that the spacial component has a far greater impact on the results than the temporal and that this shortcoming could therefore be ignored.

A considerable amount of effort was put into making the data analysis code proper (to the best of author's knowledge and ability at the time). Nevertheless, several problems remained unfixed: these should not have had a negative impact on the simulation results, only on the runtime. For instance, the dataframe list splitting function ultimately discards 2 days of data per list (beginning and end). In scope of the total size of the household dataset, this is a negligible loss of information but it could still use improvement. Alternatively, one could make use of additional residential household datasets from the Open Power System Data.

A more problematic issue was the following terminal dual warning:

```
pandapower create_jacobian.py:
           "RuntimeWarning:  invalid value encountered in true_divide"

     scipy linsolve.py:
           "MatrixRankWarning:  Matrix is exactly singular"
```

This would only appear occasionally, seemingly independent of most simulation parameters (such as load magnitudes or sgen values). It did not prevent the power flow from returning a solution and the results always appeared within the expected range. As such, this warning was mostly avoided through re-runs or, in case of the bulk simulation, through averaging of results. Lastly, occasionally there appeared to be mismatch in index and table lengths when parsing the dataframe tables, which would be solved via a simple rerun. It's possible that this bug is kernel-related to the local python installation; like the others above-mentioned, it does not impact the validity of results.

In conclusion, the results of this study show potential despite the circumstantial limitations. Given more time, a sensitivity analysis would have further aided this paper to better establish factor dependence and robustness of results. Overall, the optimal integration of EVs will require a lot more research for a smooth transition towards smart energy systems.

Bibliography

[1] N. O. Kapustin and D. A. Grushevenko, "Long-term electric vehicles outlook and their potential impact on electric grid," *Energy Policy*, vol. 137, p. 111103, feb 2020.

[2] 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.

[3] K. Mahmud, G. E. Town, S. Morsalin, and M. Hossain, "Integration of electric vehicles and management in the internet of energy," *Renewable and Sustainable Energy Reviews*, vol. 82, pp. 4179–4203, feb 2018.

[4] 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.

[5] M. Alturki, A. Khodaei, A. Paaso, and S. Bahramirad, "Optimization-based distribution grid hosting capacity calculations," *Applied Energy*, vol. 219, pp. 350–360, jun 2018.

[6] S. V. D. T. Priti Paudyal, Shibani Ghosh and J. Desai, "Ev hosting capacity analysis on distribution grids," in *2021 IEEE Power and Energy Society General Meeting*. National Renewable Energy Laboratory, 2021. [Online]. Available: https://www.nrel.gov/docs/fy21osti/75639.pdf

[7] H. Ransan-Cooper, B. C. P. Sturmberg, M. E. Shaw, and L. Blackhall, "Applying responsible algorithm design to neighbourhood-scale batteries in australia," *Nature Energy*, vol. 6, no. 8, pp. 815–823, jul 2021.

[8] M. Albadi, "Power flow analysis," in *Computational Models in Engineering*. IntechOpen, mar 2020.

[9] 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/

[10] 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

Included is the python class file code. For a complete reference, see the project repository on GitHub: https://github.com/akthonka/EV_energy_comm

```python
import os
import pandas as pd
import numpy as np
from datetime import timedelta
import pandapower as pp
from pandapower.timeseries import DFData
from pandapower.timeseries import OutputWriter
from pandapower.timeseries.run_time_series import run_timeseries
from pandapower.control import ConstControl
from pandapower.plotting.plotly import pf_res_plotly
from itertools import cycle


class DataAction:
    """
    This DataAction class contains the most common data processing functions specifically for
    working with the Open Power System Household Data sets. The main goal is to process
    raw data using pandas module in preparation for computation with pandapower module.

    The cricital data processing steps include:
    - file import
    - initial raw data filtering and formatting
    - splitting of imported data for piece-wise processing
    - datetime index parsing

    This class includes several helper functions for common actions, such as unique date
    identification for data spanning several days.

    Lastly, composite functions for heavy data manipulation are written here. These concern
    themselves with generation of test data sets for pandapower simulations - in particular,
    the timeseries iterations.

    Written by Daniil Akthonka for his Bachelor thesis:
    'Electric Vehicles in Energy Communities: Investigating the Distribution Grid Hosting
     Capacity'
    """

    def __init__(self):
        """itinitialization of class variables"""

        self.folder_path = None
        self.imp = None
        self.df = None  # comment abbreviation from hereon: df = DataFrame
        self.chunk_size = 10000  # number of datapoints in each df
        self.dfList = []
        self.conv_fac = 1000 * 60  # convert from kW/min to W (1 min avg)
        self.night_evening_t = "18:00:00"
        self.night_morning_t = "06:00:00"
        self.night_max_t = "19:23:00"  # obtained from max_load_times.ipynb
        self.night_min_t = "02:26:00"  # obtained from max_load_times.ipynb
        self.wind_length = 60  # in minutes
        self.iter_time = None
        self.night_loads = None
        self.night_sgens = None
```

```
        self.sgen_val = 0.0077  # Level 2 7.7kW AC charger

    def data_imp(self, file_name):
        """basic data import from file string"""

        folder_path = self.folder_path
        file_path = folder_path + file_name
        self.imp = pd.read_csv(file_path, low_memory=False)

    def data_filter(self, df, keep_cols):
        """initial raw data filtering and formatting"""

        col_names = df.columns.tolist()
        for col in keep_cols:
            col_names.remove(col)
        df.drop(columns=col_names, inplace=True)
        df.dropna(inplace=True)
        df.rename(columns={"utc_timestamp": "date_time"}, inplace=True)
        self.df = df.set_index("date_time")

    def df_split(self, chunk_size):
        """fragmentation of import data into smaller dfs"""

        for i in range(0, self.df.shape[0], chunk_size):
            self.dfList.append(self.df[i : i + chunk_size])

    def imp_procc(self, file_name, keep_cols):
        """all-in-one import processing helper function"""
        self.dfList = []

        self.data_imp(file_name)
        self.data_filter(self.imp, keep_cols)
        self.df_split(self.chunk_size)
        print("dfList created successfully.")

    def parse_procc(self, df):
        """parsing of datetime for minute-wise energy differences"""

        df.index = pd.to_datetime(
            df.index,
            exact=True,
            cache=True,
            format="%Y-%m-%d %H:%M:%S",
            dayfirst=True,
            utc=True,
        )  # convert index to datetime
        df = df.tz_convert("Europe/Berlin")  # match region data
        ts = df.diff()  # obtain minute-wise energy changes
        ts.dropna(inplace=True)  # drops only the first row, from the diff()

        return ts

    def unique_date(self, df):  # helper function
        """find unique days in the time-series index df"""

        return df.index.map(lambda t: str(t.date())).unique().tolist()

    def get_night(self, ts, evening_date):
        """return overnight time-delimited df data slice for evening date"""

        # get evening and morning datetime limits
        start = evening_date + " " + self.night_evening_t
        foo = pd.to_datetime(evening_date)
        bar = foo + timedelta(days=1)
        end = bar.replace(
            hour=int(self.night_morning_t[0:2]),
            minute=int(self.night_morning_t[3:5]),
            second=int(self.night_morning_t[6:8]),
        ).strftime("%Y-%m-%d %H:%M:%S")

        # check for available date and return data
        dates = self.unique_date(ts)
        if evening_date in dates:
```

```
            return ts.loc[start:end]
        else:
            print("Error: Evening_date is not part of the selected dataset!")

    def time_wind(self, ts, wind_length):  # helper function for sgen_rand
        """select random night time window for a single party (household)"""

        length = len(ts.index) - wind_length + 1  # +1 due to zero based indexing
        time_0 = np.random.randint(0, length)
        foo = ts.index[time_0]
        start = foo.strftime("%Y-%m-%d %H:%M:%S")
        bar = foo + timedelta(minutes=wind_length)
        end = bar.strftime("%Y-%m-%d %H:%M:%S")

        return start, end

    def night_rand(self):
        """create a random night load profile"""

        # choose random df identifying list number from fragemented import set
        df_rand = np.random.choice(len(self.dfList[1:-1]))  # incomplete lists excluded

        # choose random load profile (between two) and parse selected data
        rand_col = np.random.randint(0, 2)
        ts1 = self.parse_procc(self.dfList[df_rand].iloc[:, rand_col])

        # limit df to a random night in data set
        date1 = np.random.choice(self.unique_date(ts1)[1:-2])
        night1 = self.get_night(ts1, date1).copy()

        return night1

    def load_sgen_make(self, load_number=55):
        """create pandapower time series simulation df's for any number of households"""

        # initialize targets
        night_loads = pd.DataFrame()  # output df
        list_loads = []  # helper list
        night_sgens = pd.DataFrame()  # output df
        list_sgens = []  # helper list

        # generate list of Series for concatenation
        for i in range(1, load_number + 1):
            night1 = self.night_rand()
            list_loads.append(pd.Series(night1.values, name="loadh_" + str(i)))
            list_sgens.append(pd.Series([0] * night1.shape[0], name="sgen_" + str(i)))

        # merge series into output df's
        night_loads = pd.concat(list_loads, axis=1) * self.conv_fac / 1000000  # to MW
        night_loads.index = night1.index
        night_sgens = pd.concat(list_sgens, axis=1)
        night_sgens.index = night1.index

        # write output to class variables for later use
        self.night_loads = night_loads
        self.night_sgens = night_sgens

    def sgen_write(self, ts, start, end, col_name, val):  # helper function
        """write sgen value to df on column across a given time window
        Note: can't overwrite filled time-slots! Empty those first."""

        ts.loc[start:end, col_name] = -val  # 'start' & 'end' variables are str

        return ts

    def sgen_rand(self, sgen_val):
        """fill self.night_sgens df at random times with select sgen value"""

        # reset existing values (else won't overwrite)
        self.night_sgens[:] = 0

        sgens = self.night_sgens.columns
        for name in sgens:
```

```
            # writes directly to night_mw
            start, end = self.time_wind(self.night_sgens, self.wind_length)
            self.sgen_write(self.night_sgens, start, end, name, sgen_val)

    def get_start_times(self, nmbr_sgens):  # helper function for sgen_comm()
        """make list of start times to cycle through based on number of sgens and time window
    """

        time_pt = self.night_sgens.index[0]  # starting time point

        # get stop time with date
        foo = time_pt + timedelta(days=1)
        bar = foo.replace(
            hour=int(self.night_morning_t[0:2]),
            minute=int(self.night_morning_t[3:5]),
            second=int(self.night_morning_t[6:8]),
        )
        stop_time = bar + timedelta(minutes=self.wind_length)  # because of loop -1 min

        start_times = [time_pt.strftime("%Y-%m-%d %H:%M:%S")]
        for hour in range(nmbr_sgens):
            # perform window time step
            time_pt = pd.to_datetime(time_pt) + timedelta(minutes=self.wind_length)
            if time_pt < stop_time:
                start_times.append(time_pt.strftime("%Y-%m-%d %H:%M:%S"))
            else:
                break

        return start_times

    def sgen_comm(self, start_times, val):
        """fill self.night_sgens df consequently/cyclically based on start_times"""

        # reset existing values (else won't overwrite)
        self.night_sgens[:] = 0

        # initiate cycle and reset starting point
        time_cycle = cycle(start_times)
        self.iter_time = next(time_cycle)

        for col_name in self.night_sgens.columns.tolist():

            # starting time val
            foo = next(time_cycle)
            foo_dt = pd.to_datetime(foo)

            if foo_dt.strftime("%H:%M:%S") == self.night_evening_t:
                # special case: first interval
                start = foo_dt.strftime("%Y-%m-%d %H:%M:%S")

                # ending time val - 1 minute
                bar = (
                    foo_dt + timedelta(minutes=self.wind_length) - timedelta(minutes=1)
                ).strftime("%Y-%m-%d %H:%M:%S")

                # write gen value
                self.sgen_write(self.night_sgens, start, bar, col_name, val)
                foo = next(time_cycle)
            else:
                if foo_dt.strftime("%H:%M:%S") == self.night_morning_t:
                    # last minute is filled
                    bar = foo_dt.strftime("%Y-%m-%d %H:%M:%S")
                else:
                    # ending time val - 1 minute
                    bar = (foo_dt - timedelta(minutes=1)).strftime("%Y-%m-%d %H:%M:%S")
                # write gen value
                self.sgen_write(self.night_sgens, self.iter_time, bar, col_name, val)

            # update first time for next iter
            self.iter_time = foo


class net_calc:
```

```
    """
    This NetworkCalculation class is used in conjunction with the DataAction class. It contains
    functions for time series simulations using the pandapower module. The main goal is to run the
    time series iteration using preset controllers and settings.

    Further data analysis and results can be found in external jupyter notebook files.

    Written by Daniil Akthonka for his Bachelor thesis:
    'Electric Vehicles in Energy Communities: Investigating the Distribution Grid Hosting Capacity'
    """

    def __init__(self):
        """ititialization of class variables"""

        self.net = None
        self.night_mw = None
        self.n_timesteps = None
        self.time_steps = None
        self.iter_time = None  # start time cycle helper var

    def net_asym_prep(self, net, night_loads, night_sgens):
        """prepare an asymmetric load network for time series iteration"""

        # passed network will be stored as a class variable
        self.net = net

        # create load and sgen columns at asymmetric load bus locations
        for i in range(0, len(net.asymmetric_load)):
            bus_nmbr = net.asymmetric_load.bus.at[i]
            load_name = "loadh_" + str(i)
            sgen_name = "sgen_" + str(i)
            pp.create_load(net, bus_nmbr, 0, name=load_name)
            pp.create_sgen(net, bus_nmbr, 0, name=sgen_name)

        # create dataset copy w/ index for timeseries
        night_loads_ts = night_loads.copy()
        night_loads_ts.index = range(0, night_loads.shape[0])
        night_sgens_ts = night_sgens.copy()
        night_sgens_ts.index = range(0, night_sgens.shape[0])

        # create load controller
        ds_loads = DFData(night_loads_ts)
        ConstControl(
            net,
            element="load",
            variable="p_mw",
            element_index=net.load.index,
            profile_name=night_loads_ts.columns.tolist(),
            data_source=ds_loads,
        )

        # create sgen controller
        ds_sgens = DFData(night_sgens_ts)
        ConstControl(
            net,
            element="sgen",
            variable="p_mw",
            element_index=net.sgen.index,
            profile_name=night_sgens_ts.columns.tolist(),
            data_source=ds_sgens,
        )

        # note the time series iteration step variables
        self.n_timesteps = night_loads_ts.shape[0]
        self.time_steps = range(0, self.n_timesteps)

    def output_writer(self, var, index):
        """create output writer to store results"""

        path = "..\\results\\"  # one folder up the file tree
```

```
342         ow = OutputWriter(
343             self.net,
344             time_steps=self.time_steps,
345             output_path=path,
346             output_file_type=".xlsx",
347         )
348         ow.log_variable(var, index)
349 
350     def ts_run(self):
351         """run the time series iteration"""
352 
353         run_timeseries(self.net, time_steps=self.time_steps)
354 
355     def read_output(self):
356         """load the saved output files as df for further processing"""
357 
358         # load excel file
359         path = "..\\results\\"
360         vm_file = os.path.join(path, "res_bus", "vm_pu.xlsx")
361         vm_pu = pd.read_excel(vm_file, index_col=0)
362 
363         # create renaming dictionary
364         line_dict = {}
365         keys = vm_pu.columns.tolist()
366         values = []
367         for i in range(vm_pu.shape[1]):
368             line_name = "bus_" + str(i)
369             values.append(line_name)
370 
371         for i in range(vm_pu.shape[1]):
372             line_dict[keys[i]] = values[i]
373 
374         # rename cols
375         vm_pu.rename(columns=line_dict, inplace=True)
376 
377         return vm_pu
378 
379     def load_graph(self, time_step):  # helper function
380         """update network with step value"""
381 
382         run_timeseries(self.net, time_steps=(0, time_step))
383 
384     def vm_stats(self, vm_pu):
385         """return key timeseries result values for minima"""
386 
387         min_min_ind = (
388             vm_pu.idxmin().unique()[1:].min()
389         )  # first is always zero, from grid
390         min_min_vm = round(vm_pu.min().min(), 5)  # min voltage across all busses
391         min_time = (
392             pd.to_datetime(DataAction().night_evening_t)
393             + timedelta(minutes=int(min_min_ind))
394         ).strftime(
395             "%H:%M:%S"
396         )  # time of min_min_ind
397 
398         max_max_ind = (
399             vm_pu.idxmax().unique()[1:].max()
400         )  # first is always zero, from grid
401         max_max_vm = round(vm_pu.max().max(), 5)  # min voltage across all busses
402         max_time = (
403             pd.to_datetime(DataAction().night_evening_t)
404             + timedelta(minutes=int(max_max_ind))
405         ).strftime(
406             "%H:%M:%S"
407         )  # time of max_max_ind
408         print("All-time min and max:", min_min_vm, ";", max_max_vm)
409         # print("All-time min-load time across all busses:", min_min_ind, ",", min_time)
410 
411         return (min_min_vm, min_min_ind, min_time), (max_max_vm, max_max_ind, max_time)
412 
413     def plotly_res(self):  # helper function
414         """return a network diagram of results"""
```

```
415
416         x = pf_res_plotly(self.net, climits_volt=(0.95, 1.05))  # x is arbitrary
417
418     def hosting_cap(self, sgen_val):
419         """compute hosting capacity"""
420
421         # reset network sgen values
422         for sgen in self.net.sgen.name.tolist():
423             self.net.sgen.p_mw = 0
424
425         # create random sgen name list to populate loads
426         rand_ind = np.arange(len(self.net.sgen.name.tolist()))
427         np.random.shuffle(rand_ind)
428
429         hosting_cap = 0
430         for i in rand_ind:
431             # compute power flow
432             pp.runpp(self.net)
433             min_vm_pm = self.net.res_bus.vm_pu.min()
434             max_line_load = self.net.res_line.loading_percent.max()
435
436             # conditional iteration step
437             if min_vm_pm > 0.95 and max_line_load < 100:
438                 # set single sgen value and run power flow
439                 self.net.sgen.p_mw.at[i] = -sgen_val
440                 hosting_cap = hosting_cap + 1
441             else:  # since last step exceeded limit, undo it
442                 hosting_cap = hosting_cap - 1
443                 self.net.sgen.p_mw.at[rand_ind[hosting_cap]] = 0
444                 print("Max hosting capacity:", hosting_cap, "out of", len(rand_ind))
445                 break
446
447         host_pct = round((hosting_cap / len(rand_ind) * 100), 1)
448
449         pp.runpp(self.net)
450         self.plotly_res()
451
452         return hosting_cap, host_pct
```