BSc Thesis

Control & Software

Aart Rozendaal
Pieter Van Santvliet

Group O
Distribution of the electricity grid of a tiny house community

TU Delft

BSc Thesis

Control & Software

by

Aart Rozendaal
Pieter Van Santvliet

Student number:	A.M. Rozendaal	4870255
	P.E.M. Van Santvliet	4864883
Project duration:	April 19, 2021 – July 2, 2021	
Thesis committee:	Prof. Dr. P. Palensky,	TU Delft, Chair
	Dr. P.P. Vergara Barrios,	TU Delft, Supervisor
	Dr. S. Du,	TU Delft
	Dr. M. Taouil,	TU Delft

TU Delft

Abstract

This thesis shows the detailed design of the control and software of a DC microgrid of a tiny house community on the roof of a high-rise building in the city of Rotterdam in the Netherlands, consisting of twelve tiny houses powered by solar and wind energy. This thesis is part of a project with two other subgroups, focusing on the microgrid design and the powerline communication.

First, an introduction to the problem is given together with a description of the tiny house community. After that, the general program of requirements is presented, as well as the requirements of this subgroup. Next, an artificial neural network design is presented, which is used to forecast solar and wind generation and energy demand. The designed dense neural network resulted in predictions with mean errors of 10.11%, 12.56%, and 6.95% as a fraction of the maximum value for solar generation, wind generation, and energy demand, respectively. The predictions functioned as an input for the model predictive controller, which used them to place restrictions on appliances in the community when necessary, to reduce dependency on the main power grid of Rotterdam. Using a mathematical optimization algorithm, a simulation of one year showed that the controller could reduce the grid dependency up to 25%, compared to simulating without the controller. The conclusion summarises the achieved results, discusses whether the requirements are met, and considers possible future works.

Preface

This thesis was written in the context of the Bachelor Graduation Project at the Delft University of Technology. It is a continuation of the final project of the minor of one of the writers, Pieter Van Santvliet. Together with four other strongly motivated students, passionate about sustainable energy, we decided to propose this challenge. While writing the thesis, we learned a lot about the sustainable energy market, neural network algorithms, control systems, and even ethics.

We want to express our gratitude to our supervisor Dr. Pedro Vergara Barrios, who was willing to take on the challenge of a self-proposed project, pushed us to keep on improving our work, and assisted us with his knowledge, guidance, and experience during the entirety of the project.

Further, we would like to thank the thesis committee, consisting of Prof. Dr. P. Palensky (chair), Dr. P. P. Vergara Barrios (supervisor), Dr. S. Du, and Dr. M. Taouil, for taking the time to read through the thesis and attend the Bachelor Graduation Exam.

We would also like to thank Prof. Dr. P. Bauer and Ir. J. Koeners for providing data on solar generation at the TU Delft.

Lastly, we would like to give special thanks to the Bachelor Graduation Coordinator, Dr. I. E. Lager, for the excellent organization of the Bachelor Graduation Project.

Aart Rozendaal and Pieter Van Santvliet
Delft, June 2021

Contents

1 Introduction

With climate change being a more pressing matter than ever, many people are looking for ways to minimize their ecological footprint. One of the most effective ways for individuals to reduce their impact on the environment is sustainable living. In the last two decades, sustainable housing has been gaining popularity with at the forefront of the movement a concept called 'tiny houses' [1]. According to the US-based International Residential Code (IRC) a tiny house is as a dwelling that has a floor area of less than 37 m^2 [2]. This description, however, is insufficient as tiny houses cannot be described merely by their size. Tiny houses are commonly designed to achieve efficient use of internal space, greater environmental sustainability, and the ability to live off-grid while minimizing possessions [3]. This is mainly achieved by multi-functional interior design, minimizing energy demand, and operating on green energy by using Renewable Energy Sources (RESs). As space is scarce, many tiny house users put or build their tiny houses in communities to share resources. According to [4], one of the key motivators for people deciding to live in a tiny house is being part of a community. And while these communities are connected socially, many of them consist of separately built homes that each have their own energy generation and storage. Designing a tiny house community that can collaborate on a technical level was the goal of the tunus project [5].

1.1. Tunus

In the tunus project, first, the design of a single tiny house was made. This tiny house was called tunus. Then, the electricity, heat, and water grids of 12 tuni - the plural of tunus - were connected. Besides the tiny houses, a common area (or central hub) was added to the tiny house community. It consists of batteries, a controller, and washing machines.

The tunus community is designed to be on top of larger buildings in the city of Rotterdam. Power is provided by Photo-Voltaic (PV) panels and Vertical Axis Wind Turbines (VAWTs) and distributed by an intelligent DC Grid (DCG). Batteries are used to store or provide extra power.

This thesis elaborates on the tunus project [5]. Whereas the tunus project mainly discussed the concepts of all technical aspects, for this thesis, code was written, and simulations were made to implement the concepts of the electricity grid. Both will be presented in this report.

It is likely no coincidence that the project coincides with an increasing interest in microgrid technology. Microgrids are relatively small to medium-sized energy supplying systems operating within the framework of clearly defined boundaries for the generation, storage, transmission, and distribution of energy [6]. This clearly includes both the tunus project and the scope of this thesis. Now, a situation assessment and state-of-the-art analysis of microgrids are made.

1.2. Microgrid Analysis

For the past decades, the electricity grid has functioned on centralized generation and transmission using Alternating Current (AC) [7]. Generation mainly was done using fossil fuels. This meant large generators and long transmission lines. The first renewable energy sources connected to the grid were

hydroelectric, which is often also generated far from cities and thus requires transmission. Although these big interconnected AC networks were and still are the global standard, it is vital to remember that ”AC electrical energy is a transportation medium and not a commodity in itself” [8, p. 1].

According to the International Energy Agency (IEA), the world's energy consumption is expected to grow by 4.6% in 2021. 70% of this growth is projected to be in emerging economies [9]. To accommodate this kind of growth, all losses should be minimized. In Europe, losses in transmission and distribution stages can be up to 11% of the total energy consumption [10]. To combat these losses, new methods are introduced. One relatively new method that is increasingly used, is microgrids. They can decrease the losses in the overall system by spatially reducing the distance between generation and consumption. Perhaps the most crucial advantage of microgrids is their ability to integrate residential RESs without drastically interfering with the AC main grid. The current main grid is not explicitly designed to accommodate for smaller amounts of energy supplied by many residential nodes even though the use of residential RESs is quickly increasing. Due to the reduction of price and improvement of quality of PV panels and more developments in smaller VAWTs, these renewable and decentralized generation methods become more affordable and accessible. These renewable sources do present issues, as discrepancies between generation and demand can occur. New developments in energy storage technology and control strategies are increasing the feasibility of these microgrids, and their use cases are growing. This can help the transition from the old centralized fossil fuel-based system to a new, more sustainable decentralized system. Besides improving the integration of RESs and reducing losses, microgrids can also improve reliability by operating in islanded mode when the grid goes down, reduce emissions when making use of renewable sources, and, in larger cities, reduce costs by relieving the grid at critical areas [11].

Optimizing control strategies in order to make microgrids 'smart' will further improve usability and thus popularity. Another important upcoming trend is the use of Direct Current (DC) grids instead of AC as this technology is more compatible with PVs and battery systems which both operate on DC. As DC is gaining in popularity, the market for DC appliances is growing as well [12]. These trends make for a promising future as a DC microgrid with DC appliances is estimated to be more than 30% more energy efficient compared to the current grid and appliances [13]. Especially in more remote areas, microgrids can be a more efficient way to provide reliable electricity to consumers. According to the IEA [14], microgrids are the most economical way to expand access to energy in remote regions and regions which lack electricity infrastructure. They predict that by 2030, 30–40% of people living in developing countries will be supplied by microgrids supplied by renewable energy sources.

1.3. Problem Definition

It is clear that microgrids will play an important role in the future. This project will contribute to this future by building on the work of the tunus project, though only the electricity grid will be considered. This means that several of the designs and design choices that were made for the tunus project will remain unchanged for the purpose of this study. These include the location (on a Rotterdam rooftop), the heat grid, and the design of a single tiny house. Unsolved or unfinished characteristics of these topics will not be treated here as they are of little relevance. For example, how to get on the roof is not part of this project's scope. The topology and control system are subject to change, as well as the choices on electricity generation and demand and the grid connection.

The result of this project should be a design of the DC smart grid of a tunus tiny house community. It should serve more as a Proof of Concept (PoC) than a finished design or instruction manual. Further, it should contribute to the development of microgrids, for tiny house communities and elsewhere.

1.4. Subdivision

The design of the microgrid is split into three parts. A different subgroup treats each part. The first part considers the hardware of the microgrid. It treats the design choices for all the components used in the grid, responsible for energy demand, generation, storage, and distribution. It considers the topology and layout of the microgrid, including its safety and stability. This part is done by the DC Grid (DCG) subgroup.

The second part considers the control and software of the microgrid. It discusses the forecasting of energy generation and demand using Artificial Neural Networks (ANNs). This forecast is used in combination with the measurements performed by the DCG subgroup to control the microgrid. This

part also treats the design of this control algorithm. This part is done by the Control & Software (CNS) subgroup.

The third part considers the communication within the microgrid. It treats the communication over the DC power line. It discusses the modulation technique, error detection, bit rate, and transmission and receiver module. Lastly, it considers the communication protocol. This part is done by the Power Line Communication (PLC) subgroup.

1.5. Thesis Outline

This thesis treats the Bachelor Graduation Project of the Control & Software (CNS) subgroup. Chapter 2 will provide the Program of Requirements (PoR). First the project-wide PoR is given in Section 2.1, followed by a listing of the subgroup-specific PoR in Section 2.2.

In Chapter 3, an algorithm will be designed that is able to make predictions on the energy generation and demand in the tiny house community, with the predictions' results discussed in Section 3.4. These predictions are used by the controller to anticipate the future. This controller is described in Chapter 4, with the results in Section 4.6. Lastly, in Chapter 5, all is concluded, and recommendations for future work are made.

In Appendix A.2, some of the derivations for the text are detailed. In Appendix B, supplementary tables are shown. A survey that was conducted is described in Appendix C. Lastly, details about the used software and all relevant code are given in Appendix D and Appendix E, respectively.

The full overview of the system is shown in Figure 1.1. The arrows show the information flow between the different subsystems within the control system. All blocks will be designed, discussed, or treated in this thesis to create a PoC.

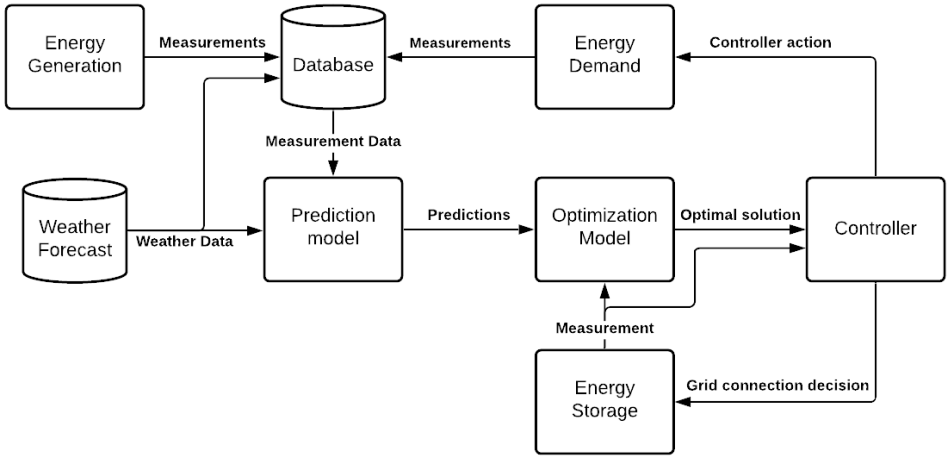

Figure 1.1: An overview of the overall control system. Measurements of the energy generation and demand and weather forecasts converge in the database. The database is used to make predictions which the optimization model uses. Based on the Optimal Control Strategy Solution, the controller makes control actions and decides on whether a grid connection should be present.

2

Program of Requirements

2.1. Overall Requirements

The following section will deal with the top-level system requirements of the Bachelor Graduation Project 'Distribution of the electricity grid of a tiny house community'.

The MoSCoW method [15] will be used to prioritize requirements. The method involves dividing requirements into 'Must have', 'Should have', 'Could have', and 'Won't have'.

Must haves are essential requirements (primary). Should haves are secondary and have less priority. Could haves are nice to have (tertiary/bonus requirements). Won't haves will not be implemented.

2.1.1. Functional Requirements

Must have

- **RQ-M.SYS.1:** The system must use the designed DCG, where information will be sent to the CNS subsystem using the designed PLC.
- **RQ-M.SYS.2:** The system must be able to supply the 12 tiny houses and the common usage of the community.
- **RQ-M.SYS.3:** The system must use renewable energy sources as supply units only.
- **RQ-M.SYS.4:** The system should have an availability of at least 90%.
- **RQ-M.SYS.5:** The system must be able to do forecasting of factors influencing the system based on the users' behaviour and the weather.
- **RQ-M.SYS.6:** The system must be able to communicate grid information over the power lines using a power line communication system.

Should have

- **RQ-S.SYS.1:** The system should be designed such that the costs are minimized for the given functional requirements.
- **RQ-S.SYS.2:** The system should be designed such that the efficiency is optimized.

Could have

- **RQ-C.SYS.1:** The system could be designed to operate in islanded mode all of the time.
- **RQ-C.SYS.2:** The system could be designed to be scalable, i.e., both the number of tiny houses per community and the number of communities can be easily increased.

2.1.2. Non-Functional Requirements

Below, all non-functional requirements of the top-level system will be given. They describe how the system should operate.

- **RQ-NF.SYS.1:** The minimum speed of the power line communication system should be such that it can sustain a transfer that is fast enough such that all data can be delivered to the devices within a timely manner.

2.2. Specific Requirements

The following section will deal with the specific requirements for the CNS subsystem.

2.2.1. Functional Requirements

Below, all functional requirements of the CNS subsystem will be given. They set boundaries on what the system should do.

Must have

- **RQ-M.CNS.1:** The system must retrieve weather data from the internet.
- **RQ-M.CNS.2:** The system must receive the hourly electricity usage, provided by the DCG, through the PLC subsystem.
- **RQ-M.CNS.3:** The system must forecast energy generation.
- **RQ-M.CNS.4:** The system must forecast energy demand.
- **RQ-M.CNS.5:** The system must estimate the state of charge of the battery using the forecasts.
- **RQ-M.CNS.6:** The system must reduce the time connected to the grid by altering power use.
- **RQ-M.CNS.7:** The system must ensure that the State of Charge (SoC) of the battery remains between its critical thresholds by deciding on whether a connection with the grid is necessary.

Should have

- **RQ-S.CNS.1:** The system should be able to update the database.
- **RQ-S.CNS.2:** The system should be able to update the forecasting algorithm.

Could have

- **RQ-C.CNS.1:** The system could give warnings based on discrepancies between predictions and measurements.
- **RQ-C.CNS.2:** The system could give warnings based on defects that are detected by the DCG subsystem.

Won’t have

- **RQ-W.CNS.1:** The system won’t have instantaneous, on a scale smaller than one hour, voltage/current control.
- **RQ-W.CNS.2:** The system won’t have instantaneous, on a scale smaller than one hour, power control.

2.2.2. Non-Functional Requirements

Below, all non-functional requirements of the CNS subsystem will be given. They describe how the system should operate.

- **RQ-NF.CNS.1:** The system should forecast with an accuracy of at least 80%, evaluated using a mean error as a fraction of the maximum value.
- **RQ-NF.CNS.2:** The system should forecast 7 days ahead.

3

Forecasting

The system requirement RQ-M.SYS.5 states that forecasts must be made of factors influencing the microgrid. This is to improve the performance of the controller by enabling it to anticipate the (forecasted) future [16]. In the subgroup-specific requirements, RQ-M.CNS.3 and RQ-M.CNS.4 specify that the energy generation and demand should be forecasted. These predictions will be made using weather forecasts (RQ-M.CNS.1) and measurements made in the microgrid (RQ-M.CNS.2).

First, it will be argued why Machine Learning is used for making these predictions. Further, it is detailed why an Artificial Neural Network is chosen as an appropriate Machine Learning (ML) algorithm. Then, it will be described how these ANNs are used to create a prediction model. A schematic showing the implementation of this prediction model is shown in Figure 3.1.

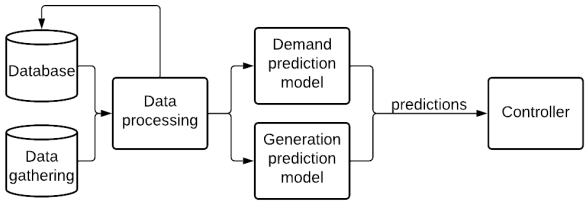

Figure 3.1: Schematic of the prediction algorithm. Data, either from the database or newly gathered, is processed and then used for the prediction models. These make predictions that are fed into the controller. The database is constantly being updated with the processed data. The arrows indicate the steps of the process.

3.1. Prediction Algorithm

In order to make a prediction, an algorithm is needed that uses information on factors that influence what is forecasted.

3.1.1. Machine Learning

Several studies have, with great success, used ML to predict both the generation of renewables and the demand of certain communities [17, 18, 19, 20]. Such precedents strongly support ML's suitability for this project. Their success may be due to an essential similarity between the forecasting of renewable energy generation and household energy demand: both depend on the weather - PV panels depend on the sun, wind turbines on the wind, and electricity usage depends (among others) on the outside temperature.

Another valuable advantage of using a ML algorithm is its flexibility. This flexibility is manifested as having a tailored model for a particular configuration for the energy generation forecasts. This is because a ML algorithm 'learns' from past data: patterns in the past will give rise to predictions in the future. Consider, for example, the influence of a building next to a solar panel configuration. A simple model that uses the expected solar irradiance and efficiency of the PV panels as input is not capable of taking the building into account. More complex models can be made that do this, but a different model

must be made for every single solar panel of the configuration. This leaves little flexibility for moving the solar panel configuration to a different location. Now, consider a ML algorithm. It will notice a power dip from past data - at the moment that the building is in between the sun and the solar panels. The algorithm will then incorporate the building's influence in the final model. As the solar panel configuration is moved around, the ML algorithm will notice different influences and include them in the prediction model that is created.

Besides accounting for the influence of buildings surrounding a solar panel configuration, the ML algorithm can also detect the influence of other external factors of a solar panel configuration, the efficiencies of the solar panels, the specific aerodynamics of a wind turbine's environment, and even the aging of solar panels or wind turbines.

The flexibility also enormously benefits the forecasting of energy demand. Two aspects are identified: capturing human behavior (that is changing through time) and enabling scalability.

Human behavior has a significant influence on energy demand [21]. The algorithm that predicts the energy demand must therefore be able to capture this human behavior. With information on the inhabitants - such as their sleep pattern or working hours - a tailored model can be created that takes the habits of the residents into account when predicting the energy demand.

For different sets of inhabitants, however, new models will need to be created. The challenge of being able to easily make several tiny house communities corresponds to the second 'could have' functional requirement of the overall project: RQ-C.SYS.2. Since ML algorithms use past demand data as opposed to information on the inhabitants' lives and schedules, they dynamically tailor their prediction models to the residents. This will improve the scalability of the technology, making the successful completion of requirement RQ-C.SYS.2 more likely.

Further, every time a resident would move out, a new prediction model must be devised based on the new resident's behavior. Moreover, the model should be updated even when the current residents change their behaviors because of a job change, the arrival of a baby, or simply because of aging. This updating of the model is also described in the second 'should have' functional requirement of the CNS subgroup: RQ-S.CNS.2. Because of their flexibility, some ML algorithms, such as ANN, are well suited to be updated regularly with minimal human input [22]. The ML algorithms that are designed to be updated are called 'online machine learning' [23].

Lastly, it can be noted that the documentation of applications of prediction algorithms for tiny houses is scarce. This makes it difficult to develop models that have good predictive performance. Once again, the wide range of applications that ML can be used for, i.e., ML's flexibility, is a crucial advantage.

3.1.2. Artificial Neural Network

Having opted for the use of ML, there remains the trade-off of which ML algorithms should be used. The three reasons that influenced the decision to choose for an ANN are discussed here, although this will be elaborated on later.

The main advantage of ANNs is that their use does not require knowledge of what it is applied on. In this case, that means that there is no need to know who will be living in the tiny house community or even how many people will be living there. Training the ANN will be similar in all cases.

Further, ANNs will simplify the completion of requirement RQ-S.CNS.2. They can be updated relatively quickly, by running the training algorithm, without the need for much external input [24].

Lastly, it must be remarked that ANNs often outperform alternative ML algorithms, which makes them an appealing option [17, 25].

In the following sections, ANNs will be designed. The first and arguably most crucial step of developing an accurate ANN is the collection of data [26]. Gathering much data is advantageous to any ML algorithm. However, ANNs differ from most ML algorithms in that they require little feature selection, i.e., the selection of features that are relevant for a prediction [24]. The next step in making an ANN is thus training the actual model [26]. When such a model has been designed, it must be tested with an independent data set. As more data is gathered throughout the operation of the tiny house community, a new ANN can be trained and tested, thus updating the prediction model.

This process of data gathering, training using an ANN, and updating the model as the database is updated is illustrated in Figure 3.2.

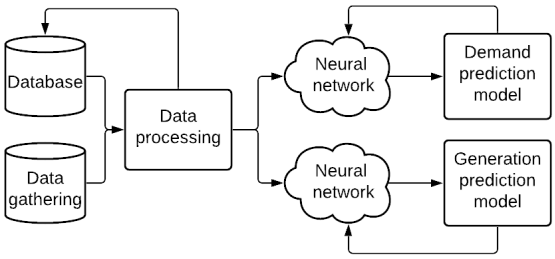

Figure 3.2: Schematic of training a prediction algorithm composed of several stages: data gathering, data processing, and creating a prediction model using ANNs. Two prediction models are developed: one for demand and another for generation. The arrows indicate the steps of the process.

3.2. Data

Data collection is essential for ANNs. The acquired data will be used for training, validating, testing, and improving the prediction model. Multiple attributes of the data are important to be able to train an accurate model. These consist of the following:

- **Time-step size:** The controller needs hourly forecasts. The forecasts must thus be made for intervals of one hour. The time-step size of the data should then be equal to (or smaller than) one hour. Data sets using smaller time-steps can be converted to hourly data by averaging the data of every hour.
- **Location:** The data sets should resemble the end-user as closely as possible. The location of the measurements of the data should thus be as close as possible to the Rotterdam area, as this is the location of the initial design of the tiny house community.
- **Aggregation level:** The aggregation level defines to what extend data has been summed up: data of a community can be on a per-household basis (low aggregation) or can be given as a single data point for the whole community (high aggregation). For ANNs, the latter is often beneficial and will be used here because data on individual households is more prone to noise and random behavior, increasing the difficulty of making predictions.
- **Overlap:** Different data sets will be acquired of different phenomena. These data sets will contain past years of data points. When the different data sets are combined, which will happen for simulating the controller, typically only the overlap of these data sets can be used. So, ideally, all data will have overlapping data points for the same year(s).
- **Amount of data:** Besides the quality of the data, the amount of available data has a large influence on the ANN [17]. Better performance can be achieved if more data can be used.

3.2.1. Data Usage Strategy

Energy generation and demand are to be forecasted. As designed by the DCG subgroup, the former consists of generation by PV panels and multiple VAWTs. The latter describes the energy used by all tiny houses in the community combined with any shared appliances. Two different data usage stages are identified: making the models (that make the predictions) and making the predictions (using the models).

1. The models that make these forecasts will be trained or created with:
    - past weather data
    - past data of solar energy generation
    - past data of wind energy generation
    - past data of demand
2. When the prediction models are in operation, they will need:
    - weather forecasts of the upcoming hours, days, or weeks

- measurements of generation and demand, these include:
  - energy generated by solar panels
  - energy generated by wind turbines
  - energy used by the tiny houses
  - energy used by the shared appliances

The second stage can be implemented using the internet and work of the DCG and PLC subgroups of the project. The weather forecasts can be provided by services on the internet. All measurements are made by the DCG subgroup and are then transmitted to the controller using the PLC network of the PLC subgroup.

Implementation of the first stage needs a more elaborate discussion. It entirely depends on the availability of past data. As time passes by, more data will become available. Because the prediction models are updated at regular intervals, e.g., every week, this will enable the training of increasingly accurate models. There are four phases in the accumulation of past data.

1. In the first phase, there is no actual past data on energy generation or demand. This is simply because there is no tiny house community as designed by tunus and the DCG subgroup yet [5]. This means that other data must be gathered and processed in order to approximate the eventual data. The prediction models based on this data will serve the first tiny house community in the first period after it becoming operational.

2. Next follows a transition phase. In this phase, the prediction models will be trained using a combination of the data of Phase 1 and new data gathered in the community. The new data can be given more weight to make the prediction model more tailored to the community.

3. As more data of the actual community is added to the database, and the prediction model is regularly updated using the database, there may come a time at which the accuracy becomes better if the data from Phase 1 is neglected. At this moment, the database will consist solely of data from the tiny house community, making the models even more customized to the residents.

4. When a second tiny house community is built, the same problem as in Phase 1 is encountered: there is no past data for this specific community. However, this time, the data from the first community can be called upon besides that of Phase 1. As such, the second community will have access to a more extensive database in its first period. Later, it will repeat Phase 2 and Phase 3. This exact procedure is also followed for the third, fourth, and any community that is built afterward.

The process of estimation in Phase 1 will be done in Section 3.2.2. This data will also be used for creating the prediction models and simulating the controller in this thesis.

3.2.2. Gathering Data

The first phase of data collection, as described in Section 3.2.1, is detailed here for weather, sun, wind, and demand data.

For each one, it will first be established what is desired. Then, the actual data sets that were found are listed. These data sets then need to be extracted from the data files into Python. They must be put into the same format. Missing or erroneous data must be dealt with. Lastly, they must be edited to acquire a database that realistically approximates what can be expected of an actual tiny house community.

Weather

Since weather data will be used for all three prediction models, its overlap with the other data sets will dictate the amount of available data for training the models. This underlines the need for many data points. Further, it is desired that the weather data set has many features for every data point - to maximize the performance of the ANN - and that these features are predictable to a certain degree - as weather predictions will be used in the second stage, i.e., when the prediction model is in operation. Lastly, the weather data should, ideally, be for the exact location of the tiny house community.

A weather data set was extracted from the Koninklijk Nederlands Meteorologisch Instituut (KNMI). This data set consists of hourly weather data of Rotterdam of the past 30 years - which should satisfy the

needs. It comprises 22 data features (listed in Appendix B.3) documented every hour at a Rotterdam weather station [27]. The KNMI also provides accurate weather predictions for these data features that extend a week into the future, thus enabling the satisfaction of requirement RQ-NF.CNS.2. They are accessible at [28] using their Application Programming Interface (API).

The weather data was extracted from the csv data file and then put into an array. The function that performs this is given in Appendix E.1.1 as: retrieveWeatherData().

Sun

According to the DCG subgroup, the tiny house community consists of $1.7m^2/panel * 36panels = 61.2m^2$ of LG Neon R solar panels. The desired data set thus would comprise multiple years of data on energy generation of $61.2m^2$ of LG solar panels. Moreover, the solar panels used for this data set would ideally be configured in the same way and on the same location as the DCG-designed tiny house community.

The EEMCS faculty building at the Delft University of Technology has solar panels on top of the building and solar panels installed next to it. The latter is part of an e-bike charging station. EEMCS faculty members were contacted to get access to this data - it is available at [29, 30]. The data starts in 2016 and goes until the time of writing this report, Summer 2021, with the number of data points per hour increasing throughout the years. The raw data consists of: a timestamp, the PV voltage, and the PV current, among other less relevant data.

Only data of 2019 has been considered. It is the most recent, non-leap year. There are an inconsistent amount of data points per hour, ranging from none to over 60 for one hour. There are also multiple data gaps, with the amount of missing data ranging from an hour to several days. Because of the inconsistencies and missing samples, the data of solar panel generation needs several edits for it to approximate the PV installation of the tiny house community. Appendix A.1 describes the steps that were taken to obtain a consistent data set without missing values. The acquired data must still be scaled: it is for an installation of a few square meters instead of the $61.2m^2$ of LG solar panels. This scaling is done using a Scaling Factor (SF). In order to obtain a total yearly generation equal to that of the tiny house community, this SF comprises the mean generation of the data set and the designed installation.

$$P_{\text{new}} = SF * P_{\text{old}} = \frac{P_{\text{mean, installation}}}{P_{\text{mean, data set}}} * P_{\text{old}} \tag{3.1}$$

The resulting data set should be a good approximation of what can be expected from the tiny house community. The data was interpreted as power (in W), but the same number can represent the energy generated by this power in one hour (in Wh). The Python code that performs the steps described above is given in Appendix E.1.2.

Wind

From the DCG subgroup, it is inferred that the tiny house community will consist of 6 VAWTs of the model Aeolus V 2kW. So the desired data set would consist of several years of data of the energy (in Wh) generated every hour by six operational Aeolus V wind turbines. As with the solar panels, ideally, these wind turbines would be configured in the same way and on the exact location as the DCG-designed tiny house community.

On [31], a data set was found that gives the hourly Capacity Factor (CF) of Horizontal Axis Wind Turbines (HAWTs) in different regions of the European Union (EU), including Rotterdam, for a period of 30 years, 1986 to 2015. The CF is the fraction of the nominal power that is produced by a wind turbine. This makes it a relevant metric that can be used to approximate the generation of 6 Aeolus V wind turbines.

The CF data is consistent and contains no missing hours. A selection of data was made to ensure overlap between the weather data and the wind data: the period of 2001 to 2015 was taken. Further, the data has to be scaled. This scaling is done with Equation (3.1) using the same method as for the solar data. The result should be a reasonable approximation of the desired data set. The data was first interpreted as power (in W) but, from now on, it will be treated as the energy that is generated by this power in one hour (in Wh). The Python function that performs this is given in Appendix E.1.1 as: retrieveWindData().

Demand
The ideal data of demand would stem from a tiny house community as designed by tunus and the DCG subgroup [5].

For the demand data, there is a more significant gap between what is desired and what is available at the moment. Usage data of tiny houses is scarce, if not non-existent. Much of the concept of tunus, and tiny houses in general, is to rethink a household and (among others) its energy demand [5]. This makes data on regular households relatively unfit for approximating the tiny house community of this project. It will be used nonetheless because of the aforementioned lack of reliable data on tiny houses. because this project serves as a PoC rather than a detailed instruction manual, using household data for the predictions should suffice to demonstrate the correct working of both the forecasting algorithm and the controller. A data set of 82 households, provided by Liander [32], was found. Liander is a Dutch utility company connecting millions to the Dutch electricity grid [33]. Only a single year of data on the 82 households, being 2013, is available. This is due to privacy concerns, which regularly complicate the acquisition of household demand data. The data is on in Wh on a 15 minute basis.

The demand data from Liander consistently has 4 data points per hour representing the amount of energy used in that quarter of an hour (in Wh). For every hour, these 4 data points have to be summed. There are also missing values that have to be dealt with: not every hour has a value for all 82 households. Some are missing. These missing values are replaced by the mean of all households with data at that hour. Then, the data of all households is summed for every hour to get to a higher aggregation level. Now, the data must still be scaled. In this case, it will be scaled according to the means of the Liander and tiny house community demand. This ensures that the final data set will have the same yearly usage as the tiny house community. The SF is calculated by dividing the former by the latter. All values of the data set are then multiplied by this SF. The Python function that performs this is given in Appendix E.1.1 as: retrieveDemandData().

3.2.3. Preparing Data to Train the First Model

The data can now be retrieved from the data files. Although it is partly processed - missing values replaced and in the correct format - it still needs to be divided into an input and output that the ANN can operate on. An overview of this division is given in Table 3.1.

The input of the solar model is the weather data. As mentioned, the ideal output of the solar model would be data on the PV installation of the actual tiny house community. For both training and the controller, however, data on the PV installation next to the EEMCS building is used as the output of the model. Notice that for the controller, data of 2019 is used, as opposed to 2013 that is used for wind and demand. This imperfection is inevitable due to a lack of overlapping data.

For the wind model, the input is the weather data as well. The ideal output would be data on the VAWT generation in the tiny house community. For training, 15 years of recorded CFs can be used. For the controller, only 2013 is considered.

To save time during training and testing, the demand model uses only a selection of the weather data features (as from Appendix B.3): temperature (T), sunshine duration (SQ), rain (R), thunder (O). This selection was made to identify features that influence household electricity demand. However, because of the shallowness of this selection, in later stages, more features should be added, and their influence should be evaluated. Besides this, information on the time was added as a feature to the input data. This includes the month number (1-12), week number (1-7), and hour (1-24). This was done based on [17], to enable the ANN to better learn about the - extremely time-dependent - behaviour of humans. The perfect output would be data on the usage of the tiny houses in the community. For both training and the controller, data on 82 Dutch households will be used, however. Although it is not ideal, it is partially countered by the more aggregated - and thus more predictable - household data (82 houses instead of 12).

The nature of the data and the code both allow for the database to be updated as new data becomes available, therefore enabling requirement RQ-S.CNS.1.

Table 3.1: An overview of the inputs and the ideal, training, and controller outputs of the ANNs that will be trained in Section 3.3.

data	model		
	solar	wind	demand
input	weather	weather	weather and date
ideal output	PV installation of tiny house community	VAWTs of tiny house community	Usage by the tiny houses
training output	PV installation next to EEMCS building (2019)	CF of HAWTs surrounding Rotterdam (2001-2015)	82 households (2013)
controller output	PV installation next to EEMCS building (2019)	CF of HAWTs surrounding Rotterdam (2013)	82 households (2013)

3.3. Training

All data is now preprocessed and formatted as such that it can easily be used by the Python ML Libraries. First, the procedure for training a model will be treated. Methods of evaluating a model are then discussed. Using these, some configurations of the ANN are tried, and their result is given. In Appendix E.1.4, a script is given that will, among other operations, train a prediction model based on the data that is inputted. Using the most accurate model, a new feature can be considered: the prediction of the previous hour. The results of adding this new feature are then analyzed.

3.3.1. Procedure

Before using the data for training, it is good practice to split it into a training and test set. Here, this split - whether a data point ends up in the training or test set - is done randomly to ensure that all months are represented in both the training and test set. Standard allocation distributions are 70/30 and 80/20, representing training/test. The former distribution is chosen here because there is sufficient data.

Then, the base model of the ANN is defined. It consists of a certain amount of layers, with a certain amount of neurons per layer. Both these variables are hyperparameters, parameters that are used to configure the ANN. The base model will determine the structure of the ANN. The concept of a base model is visualized in Figure 3.3. As can be seen, the ANN starts with an input layer, where the features are given as input, and ends with an output layer, out of which come the results. In between the input and output layers are the so-called hidden layers. When counting the number of layers in a configuration, only the amount of hidden layers will be reported: a one-layered ANN has one hidden layer. Different configurations of the base model will be tried for each prediction model - sun, wind, and demand - later.

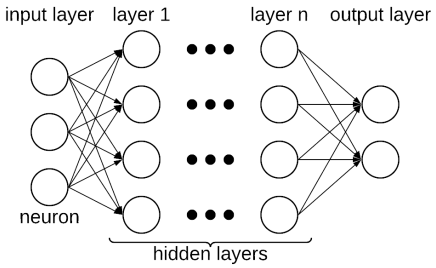

Figure 3.3: A denseANN with n layers of which the first and nth layer have 4 neurons. Features from the input layer are fed into the first hidden layer and propagate to the output layer, where finally a prediction is presented.

With the base model established, all that is still needed to train the ANN using the data is to define the batch size and amount of epochs. The former is ”a hyperparameter that defines the number of samples to work through before updating the internal model parameters” [34, Ch. 3]. The latter is ”a hyperparameter that defines the number of times that the learning algorithm will work through the entire training dataset” [34, Ch. 4]. Four hyperparameters can now be identified:

- The number of layers

- The number of neurons (for every layer)
- The batch size
- The number of epochs

In the following sections, the objective will be to find the ideal values for each of these parameters that maximize the accuracy. If the database is updated, as described in Section 3.2.3, the same procedure described above and code, described below, can be used to update, i.e., make more accurate, the ANN. Therefore requirement RQ-S.CNS.2 is satisfied.

Notice that other hyperparameters exist, though they will not be considered here. They could be tuned later to further improve the performance of the ANN. Nonetheless, some of them must be chosen in order to make a model. The choices for those relevant for creating the models of this project were based on [35]. These hyperparameters with corresponding choices are taken to be:

- Kernel Initializer: 'Normal'
- Activation: 'ReLU'
- Loss: 'Mean Squared Error'
- Optimizer: 'adam'
- Top-Level Architecture: 'Dense'

3.3.2. Evaluating

Two methods of evaluating a specific model will be reviewed. They have in common the fundamental method of training ANNs: For the batch size b and the number of epochs e, the base model is run through e times, with the internal model parameters updated every b samples. By updating the internal model parameters, it is attempted to minimize the loss.

The first method will make a model by training the ANN on the training set. The trained model is then used to predict all values of the test set. The discrepancy between the expected and predicted test set is then reported as a result. It is often given as Mean Squared Error (MSE).

A second method of evaluating is cross-validation. It is a more elaborate alternative to the first method. For n split cross-validation, the ANN is trained n times. Each iteration, the complete training data set is divided into a smaller training set and a validation set. The ANN is then trained with the training set and evaluated on the validation set using the method described for the first method. After the n iterations, all scores are combined into a single score. This final score is comparable to the result of the first method, though more accurate. The increased accuracy stems from the repetition and changing training/validation splits. Further, the model can be evaluated on a test set independent and separate from the training data set. This method is well suited for evaluating the structure (layers and neurons) of the prediction model. A Python function that performs cross-validation is given in Appendix E.1.1 as: performCrossValidation(X_train, y_train, n_splits, model). A scheme visualizing the concept of cross-validation is given in Figure 3.4.

3.3.3. Hyperparameter Tuning

Cross-validation can now be used to find values for the hyperparameters that yield reasonable accuracies.

First, a reference configuration will be presented, based either on a precedent or, if few are available, on intuition. For this reference, different combinations of specific batch sizes and certain amounts of epochs will be tried. Using those that yield the best result, three different layer configurations will be considered with more or fewer layers than the reference. For each of these configurations, a different amount of neurons per layer will be tried. The Python code that was used to do hyperparameter tuning is given in Appendix E.1.3.

A figure that visualizes this scheme is given in Figure 3.5. Notice that all training done in this scheme will be performed using cross-validation to minimize the influence of randomness during training.

As can be deduced from the figure, actually finding the best hyperparameters for an ANN configuration is an iterative process. In order to fully maximize the capabilities of the ANN, the scheme will have to be run through many times until a satisfactory result is acquired, or the result is not improving

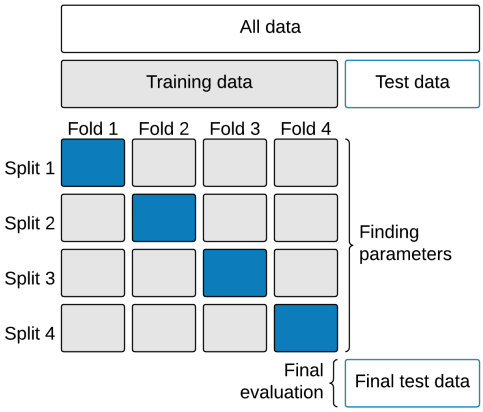

Figure 3.4: Visualization of 4-fold cross validation with a training set (grey), dynamic validation set (blue), and test data (white). In the training phase, different splits into training and validation sets are made. They serve to better evaluate the model. Finally, the model is evaluated using the final test data set.

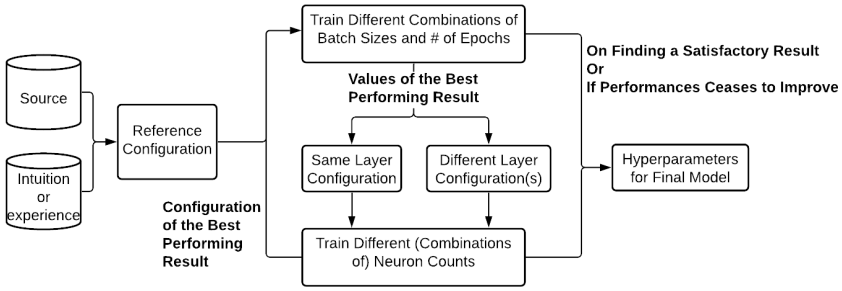

Figure 3.5: Possible scheme for finding the best hyperparameters for an ANN configuration. A reference configuration is made using intuition or from a precedent in a source. From there starts an iterative process of trying different batch sizes and amounts of epochs, trying layer configurations, and trying different neuron counts for the layers. The process ends with the parameters for a final model. The arrows indicate the steps of the process, with contained information or conditions.

anymore. However, only one iteration is performed here, as it is not the project's scope to find the best possible values.

Moreover, since the data sets used are not ideal for the actual tiny house community, there is little point in tuning the model to ideal circumstances. Nonetheless, one could use the scheme with its accompanying code to systematically find the best hyperparameters for a prediction model. That process will bear fruit when data from an actual implementation of the tiny house community becomes available.

Sun

From precedents of predicting the generation of a PV installation, it can be inferred that one hidden layer should suffice to create a well-performing ANN [19, 36, 37, 38]. Jeff Heaton further mentions that, in general, one hidden layer suffices to the needs of most problems [39]. This also supports starting with a reference of one hidden layer.

The number of neurons in this one hidden layer can initially be set on the input dimension, a method supported by [39, 35]. Given that the weather data set contains 22 features (listed in Appendix B.3), this makes for an initial neuron count of 22.

Using this initial ANN, appropriate values for the batch size and amount of epochs are looked for. The following options are tried for the batch size: $[200, 500, 1000]$. Regarding the epochs, the following amounts are used for training: $[10, 25, 50, 100, 200, 500]$. Of these two sets of numbers, all possible combinations are tried and evaluated using 3-fold cross-validation. Table 3.2 gives the losses of the 18 different combinations that were tried: the lower the loss, the better the prediction approximates the actual value.

A batch size of 200 with 500 epochs gives the best result. Further possibilities for the number of epochs are tried, using a batch size of 200: $[1000, 2000, 5000]$.

Table 3.2: The losses of different combinations of batch sizes and amounts of epochs for solar generation. The loss is given as an unscaled MSE in W^2h^2.

epochs	batch size		
	200	500	1000
10	204492	228035	235501
25	106797	203709	229059
50	97573	111524	215740
100	95007	97076	107474
200	95280	96200	95481
500	91735	92287	94310
1000	91943	-	-
2000	100853	-	-
5000	94815	-	-

The loss starts to increase again as the amount of epochs surpasses 500. More batch sizes could be tried as well, and more simulations can be done to better approximate the ideal amount of epochs. However, it suffices to use a batch size of 200 with 500 epochs from now on for this project.

Now, the number of neurons can be tuned to increase performance. For a one-layered configuration, the following neuron counts were tried: $[10, 15, 20, 25, 30]$. As 20 was found to be the best neuron count of these options, further neuron counts were tried to better approximate the ideal amount of neurons: $[16, 18, 20, 22, 24]$. The result of these simulations is given in Table 3.3.

Table 3.3: The losses for different neuron counts in the first layer of a one-layered ANN configuration for solar generation. The losses are given as an unscaled MSE in W^2h^2.

neurons	loss $[W^2h^2]$
10	94796
15	92197
16	91073
18	92154
20	88385
22	91881
24	91997
25	94459
30	92115

Of these options, 20 remained the best-performing neuron count. Moreover, the model with a loss of 88385 W^2h^2 should already make for a reasonable prediction model. With the MSE of the unscaled data at 88385 W^2h^2, the Mean Error becomes $\sqrt{88385W^2h^2} = 297Wh$, or $297Wh/2383Wh = 12.48\%$ of the maximal value.

Of a two-layered ANN, different amounts of neurons in the first layer, $[15, 20, 25]$, and second layer, $[10, 15, 20, 25, 30]$, were tried. Once again, all combinations were trained with the result of these simulations given in Table 3.4.

Finally, a three-layered ANN is considered. Contrary to what one may think, a well-performing neuron count for a one-layered configuration does not guarantee that the first layer best has the same amount of neurons in a three-layered configuration. Nonetheless, the best neuron count for a one-layered configuration (20) is chosen for the first layer here due to computational limits. Different neuron counts were tried for the second and third layers: $[5, 10, 15]$ and $[5, 10, 15, 20, 25, 30]$ respectively. The results of which are given in Table 3.5.

Out of all these 42 possibilities with different layer and neuron counts, The best performing ANN was the three-layered configuration with 20 neurons in the first layer, 5 in the second, and 30 in the third.

If the prediction model were to be implemented, for example, these steps would have only been the

Table 3.4: The losses for different neuron counts in the first and second layer of a two-layered ANN configuration for solar generation. The losses are given as an unscaled MSE in W^2h^2.

Neurons	Neurons layer 1		
layer 2	15	20	25
10	90102	89541	96710
15	93303	90031	91321
20	97351	92529	90138
25	93342	92191	89852
30	89704	91279	93313

Table 3.5: The losses for different neuron counts in the second and third layer of a three-layered ANN configuration for solar generation. The first layer is taken to have 20 neurons. The losses are given as an unscaled MSE in W^2h^2.

Neurons	Neurons layer 2		
layer 3	15	20	25
5	243615	93430	145280
10	90619	89113	90190
15	89622	94431	92954
20	91579	91289	91118
25	90350	94020	92372
30	87350	91623	89211

start of a long iterative process to find the best possible hyperparameters for the tiny house community. For the best performing configuration so far, the batch size and number of epochs can be optimized again. Using those, different configurations with more, or less, layers can be tried once more. Afterward, a new iteration can follow until the result ceases to improve or is satisfactory.

The mean error as a fraction of the maximum value is calculated by taking the square root of the MSE and dividing it by the maximum value, in this case, 2383 Wh. The mean error is then found to be $\sqrt{87350W^2h^2}/2383Wh = 12.40\%$ of the maximum value, which suffices for establishing a PoC and satisfies requirement RQ-NF.CNS.1. Hence, the following prediction model for wind can be considered now.

Wind

The ANN for wind generation will be initialized as that of the solar generation: one hidden layer with 22 neurons. Such an initial configuration is supported by [37, 39].

Different combinations of batch sizes $[200, 500, 1000]$ and amount of epochs $[10, 25, 50, 100, 200, 500]$ are used for training. The results that this yields are given in Table B.1 in Appendix B.1. The best combination is found to be that with a batch size of 200 and 200 epochs. This combination will be used below.

In tuning the number of neurons, three different layer constructions are considered: one-, two-, and three-layered. For each, different combinations of neurons are tried, with a total of 34 different structures.

For the one-layered construction, the following neuron counts were used: $[5, 10, 15, 20, 25, 30, 35]$. The losses of the ANNs with these neuron counts are given in Table B.2 in Appendix B.1. This yielded a best performance of $\sqrt{0.020836} = 0.1443/1 = 14.43\%$ of the maximum value, for 10 neurons.

The results of the two-layered construction are given in Table B.3 in Appendix B.1. Multiple combinations of neuron counts in both layers were used for training. In the first layer, $[5, 10, 15]$ were used as amounts of neurons. In the second layer, the applied neuron counts were: $[5, 10, 15, 20]$.

Finally, a three-layered ANN was considered. Once again, with different combinations of neurons in the second layer $[5, 10, 15]$ and in the third layer $[10, 15, 20, 25, 30]$. For the first layer, the neuron count was 10: the best performing neuron count for a one-layered configuration. The results of these simulations are given in Table B.4 in Appendix B.1.

Out of all configurations with different layer and neuron counts, the best performing ANN was found to be the three-layered ANN with ten neurons in the first layer, 15 in the second, and 25 in the third.

The mean error is $\sqrt{0.1334}/1 = 13.34\%$ of the maximum value, which satisfies RQ-NF.CNS.1.

Demand

The initial model of the demand prediction model is based on [17]. Del Real [17] uses this ANN to predict the demand of the French grid. Their model's structure is given in Table 3.6.

Table 3.6: The ANN layer configuration used by [17] to predict the demand of the French grid.

Layer	Neurons	Activation function
1	256	ReLU
2	128	ReLU
3	64	ReLU
4	32	ReLU
5	16	ReLU

With the ANN of Table 3.6, different batch sizes $[200, 500, 1000]$ and amounts of epochs $[10, 25, 50, 100, 200, 500, 1000, 2000]$ were used for training. As can be seen from Table B.5 in Appendix B.2, the best performing combination uses a batch size of 500 and 1000 epochs.

For this five-layered configuration, 32 combinations of neuron counts were tried. For each layer, two options were given: $[150, 300]$ for Layer 1, $[100, 200]$ for Layer 2, $[50, 100]$ for Layer 3, $[25, 50]$ for Layer 4, and $[10, 20]$ for Layer 5.

The result of this simulation is given in Appendix B.2. The best performing option is that with 150, 200, 50, 25, and 20 neurons for the first, second, third, fourth, and fifth layers.

The mean error of this configuration is $\sqrt{266685W^2h^2}/6222Wh = 0.082998 = 8.30\%$ of the maximum value. No other layer structures are tried because of the computational difficulty of evaluating five-layered structures and the satisfactory result that has already been achieved - a mean error or inaccuracy of 8.3% corresponds to an accuracy of 91.7%, far exceeding the required 80% of RQ-NF.CNS.1.

3.3.4. Adding the Previous Hour

In the previous section, already good results were obtained. Further features could be added to improve the performance of the models further. However, in this case, the amount of available data limits the number of features that can be added. Nonetheless, one feature can be added without extra data, which might improve performance significantly: the (prediction of) the previous hour.

In many cases, the generation or demand of a particular hour is correlated to that of the hour preceding it. This makes it interesting to use the generation or demand of the previous hour as a feature for the next one. Two options can be identified: using the actual value of the previous hour (method 1) or using the prediction after hour preceding that to be predicted (method 2). Both options will be evaluated in this section.

Using the previous hour's generation or demands (method 1) is more straightforward to simulate than using the predictions. However, it is more challenging to implement method 1 in an actual tiny house community. This is due to the simple fact that when training the algorithm, the actual usage or generation is already known. When implementing in a tiny house village, on the other hand, data of the actual previous hour can only be used to predict the hour immediately following the current time.

Assuming that method 2 becomes more advantageous when the prediction of the previous hour is better, it can be beneficial to combine both methods. The hour succeeding the current time can use the actual value of the current time, therefore becoming more accurate. This, in turn, makes it a more helpful feature for the hour succeeding it, and so on. This spillover effect will be especially interesting for short time horizons.

The described spillover effect could be simulated. However, this would need thorough hyperparameter tuning for the models used to notice subtle differences that might surface.

Because of the limited amount of time and computational equipment available, only the simpler methods 1 and 2 are evaluated. For this evaluation, the best performing ANN structures of the previous section will be used.

The Python code that was used to evaluate the two methods is given in Appendix E.1.5. Notice that the code in Appendix E.1.1 is imported and is thus necessary for running the script.

Method 1
A model was trained using the actual past hour as a feature with 70% of the available data. This model was then evaluated on a test set containing 30% of the available data. The acquired mean error as a fraction of the maximum was:

- Solar: 9.80% as opposed to 12.40%
- Wind: 4.20% as opposed to 13.34%
- Demand: 6.40% as opposed to 8.30%

All three models significantly improve. However, the prediction for wind generation improves the most. Its inaccuracy more than halves. This discrepancy might be due to the difference in available data: there is 15 times more data for wind than for the two other models.

Nonetheless, these improvements make method 1 very suitable for predicting the hour immediately after the current time.

Method 2
As for method 1, a model was trained, this time, however, using the predicted past hour as a feature. For the training, 70% of the available data was used. The remaining 30% was used for the evaluation. The acquired mean error as a fraction of the maximum was:

- Solar: 9.86% as opposed to 12.40%
- Wind: 12.53% as opposed to 13.34%
- Demand: 7.41% as opposed to 8.30%

Although the improvement is not as significant as for method 1, method 2 yields improvements for all three models compared to the models that do not use the predicted previous hour as a feature. This, as well, supports the use of previous hours when making predictions for the next one.

3.4. Results

For each prediction model, an ANN configuration has been established that yields good results. However, in previous sections, cross-validation was used, meaning that no final model is required because multiple models are created to evaluate a configuration.

A final model is now created with a training set. For this model, forecasts by an earlier model (created using the Python code in Appendix E.1.4) are used as a feature. A final evaluation of the model on a test set, comprising 30% of the data, is given in Table 3.7. The model's prediction of a separate week of data - not used for training or testing - is shown in Figure 3.6.

After evaluation, the acquired model is used to make predictions of the whole year. This is necessary for simulations of the controller. Notice, however, that the results of this are not entirely representative since the training set is used for both training and prediction.

The predicted generation and demand of a whole year are given in Table 3.8. They are compared with the actual yearly generation and demand, as designed by the DCG.

Table 3.7: The final results of the three prediction models. For each model, the best performing batch size, amount of epochs, and structure are given. The corresponding mean error, as a fraction of the maximum value, is presented as well, for evaluation.

	Solar	Wind	Demand
Batch size	200	200	500
Epochs	500	200	1000
Layer 1	20	10	150
Layer 2	5	15	200
Layer 3	30	25	50
Layer 4	-	-	25
Layer 5	-	-	20
Mean Error	10.11%	12.56%	6.95%

Table 3.8: The predicted and actual total of a whole year, for all three models. The deviation between the predicted and actual total is given as well.

Model	Predicted [MWh]	Actual [MWh]	Deviation
Solar	11.81	12.27	3.9%
Wind	10.56	10.86	2.8%
Demand	15.13	14.97	1.1%

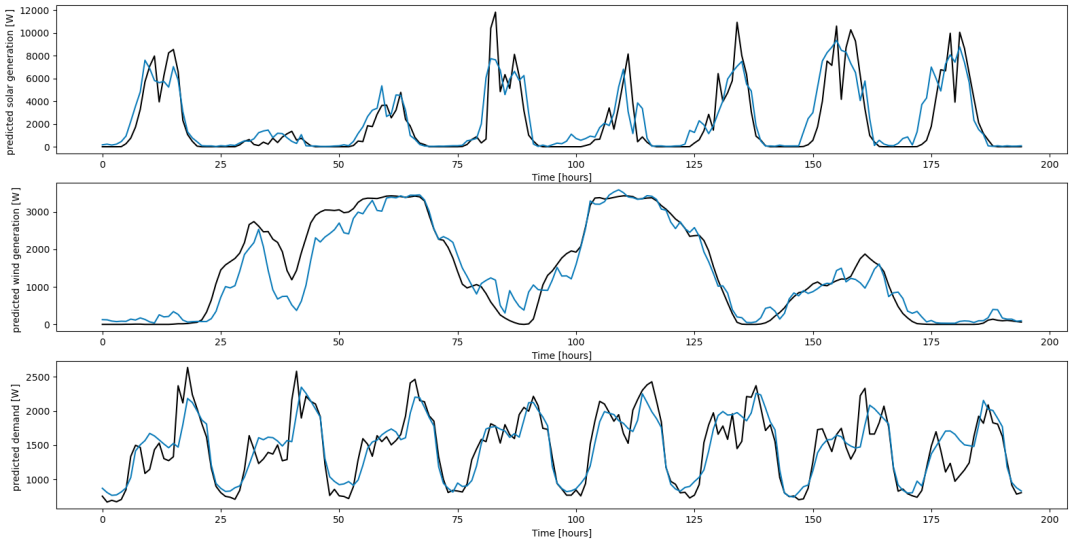

Figure 3.6: The solar and wind generation and energy demand of an 8 day period in May. The actual value (black) and prediction (blue) are both given. Notice that for solar generation, the model correctly anticipates on the generation having a dip every day at around noon (the middle of the day).

4

Controller

To operate the microgrid in the tiny house community, a controller is required. This is implemented as an algorithm governing the system's behavior, trying to drive it to its desired state, guided by its inputs. To obtain this desired state, a controller needs to be able to correct the behavior. Different types of controllers perform these tasks in different ways. All these different types of controllers have their benefits and drawbacks, and the controller choice is highly dependent on application and circumstances.

In this chapter, different controller options are presented as well as the controller choice. This choice should meet RQ-M.CNS.5, which states that the forecasts should be used in the control strategy. Further, the controller should reduce the time spent connected to the grid as mentioned in RQ-M.CNS.6. Lastly, it should operate according to a controller scheme such that the battery stays between its critical threshold, as in RQ-M.CNS.7. This controller is then implemented using the data of the tiny house community. After implementation, the controller will be tested using a simulation, and the results will be presented to check whether the requirements are met.

4.1. Controller

To make use of the ANNs and meet RQ-M.CNS.5, the controller should use the predictions in its decision-making process. Several well-established and widely used controllers are not able to utilize predictions in their decision-making process. For example, the Proportional Integral Derivative controller, better known as the PID controller, known for its simplicity and many use cases, lacks this ability. Controllers that do have this ability are called predictive controllers. Only a handful of these controllers have been successfully employed in industrial control applications [40]. Of those, two predictive controllers are most prevalent, namely Model Predictive Control (MPC) and the Linear Quadratic Regulator (LQR).

4.2. Predictive Controllers

Before discussing the advantages and disadvantages of both MPC and LQR, it is essential to establish the framework of the control strategy they both employ. This control strategy is called optimal control. As this name implies, both MPC and LQR determine the control strategy through optimization. Although they use a similar approach, they differ in one key element. To explain this and other differences, it is important to explain what optimal control is.

4.2.1. Optimal Control

Optimal control theory is a branch of mathematical optimization that handles controllers of dynamic systems over a period of time. In optimal control, the controller action is determined through minimizing or maximizing the objective or cost function [41]. The objective is a mathematical equation representing the phenomenon that needs to be optimized. Examples are the profit of a company, the hours spent on a project, or the cruising speed of a car.

Often when optimizing such an objective, the system is subject to constraints. The advantage of optimal control is that it can minimize the objective while accounting for these constraints. Examples

of constraints related to the previous examples are the maximum store capacity, the maximum number of hours that some team members can spend on the project, and the maximum acceleration of the car.

In optimal control, the cost function is minimized over a time horizon. This time horizon has a finite length N and a finite time step resolution k. The difference between MPC and LQR lies in the way they define this time horizon.

A further benefit of optimal control is that it translates well from verbal constraints to mathematical constraints to code implementation. An example of such a mathematical formulation is shown in Equation (4.1), where cost function J is minimized over time horizon N with time steps k, while subject to the constraint that $f(k) \geq 0$.

$$\begin{aligned} \min_J \quad & J = \sum_{k=1}^{N} f(k) \\ \text{s.t.} \quad & f(k) \geq 0 \end{aligned} \tag{4.1}$$

4.2.2. Controller Selection

As mentioned above, MPC and LQR differ in how they define the time horizon. LQR uses one-time optimization for the time horizon. This optimization is done offline and, thus, does not require recalculation as all has been done before implementation. The optimal controller actions are determined in advance and cannot respond to dynamic behavior. MPC, however, optimizes across a smaller and receding time horizon [42]. This means that it requires the optimization to be recalculated every time step, which is computationally heavier than LQR. Although calculated for the whole time horizon, only the control strategy for the next time step is implemented, resulting in a dynamic controller as shown in Figure 4.1. The figure shows the time horizon during three time steps, where it recalculates the optimal control strategy and only implements the first control strategy. Because MPC is run over shorter time horizons, this can sometimes lead to sub-optimal solutions, but this is compensated by the frequency of recalculations.

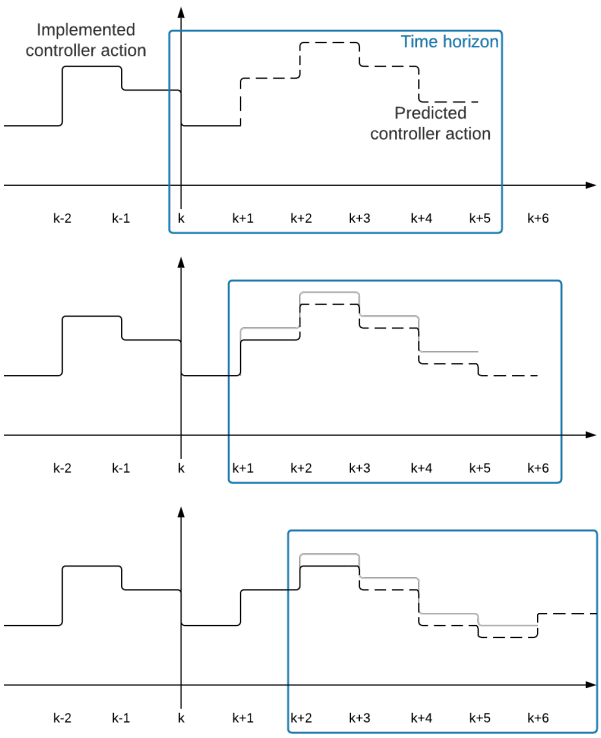

Figure 4.1: Functionality of an MPC controller showing a receding time horizon for three time steps, where it recalculates the optimal control strategy and only implements the first control strategy.

Furthermore, MPC can handle non-linear models and hard constraints, something LQR lacks. A hard constraint sets a condition for a variable that is required to be satisfied. The non-linearity of the ANN and hard constraints on the control strategy and battery size render the choice between MPC and

LQR obvious, namely MPC. Furthermore, as the predictions will always be off, recalculating the appropriate response is preferable over one-time optimization. Lastly, as long as the time steps between recalculations are larger than the calculation time, there is no drawback in the MPC being computationally more expensive.

4.3. Model Predictive Control

As mentioned above, MPC is an iterative process over a receding time horizon. When looking at an iterative process, it can prove helpful to first look at one cycle of the process. A cycle is defined here, as all actions performed during one iteration. A schematic of one cycle of the MPC is shown in Figure 4.2. The MPC uses the the measurements and predictions as inputs to find an optimal solution to minimize the objective. The objective function will be formulated so that it reduces the microgrid's dependency on the main grid. Minimizing this objective function leads to the optimal control strategy at the output. This control strategy is presented as a control level that influences the usage of the tiny house community by posing restrictions on certain facilities.

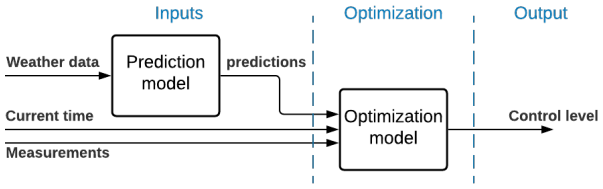

Figure 4.2: Overview of the MPC showing the inputs, being the current time, predictions, and measurements and the output, being the control level.

4.3.1. Time Step

Before defining the details of the inputs and outputs of the system, it is vital to establish the time step resolution for the inputs and output values to be formatted accordingly. The time step used for the optimization was chosen to be one hour. This is primarily because of the resolution of the weather data available at the input of the prediction model. The predictions that are available at the output of this prediction model also have this one-hour resolution. This limits the time step of the optimizer model. Further, the time horizon was chosen to be seven days to meet RQ-NF.CNS.2.

4.3.2. Inputs

The optimization model of the MPC has three inputs. These are the measurement of the SoC of the battery, the predictions of the generation and demand, and the current time. These values are required for the optimization of every time step.

Measurements

The measurement of the SoC is communicated to the controller via the PLC every time step and presented at the input of the optimizer. The measurement is used to set the initial value for the battery at every time step. This will get rid of any errors remaining from calculations in the previous time step.

Predictions

The prediction model uses the weather data for the coming week to make predictions on generation and demand. The predictions are presented in Wh to the optimizer. To keep the optimization as simple as possible, Wh is converted to SoC percentages. The battery chosen by the DCG group has a total capacity of 72200 Wh, corresponding to 100% SoC. Thus 1% SoC is $72200Wh/100\% = 722Wh/\%$. The actual input to the optimization model is the change in the SoC of the battery due to the predicted generation and demand. This is calculated for every time t in the horizon as shown in Equation (4.2).

$$\Delta SoC[t] = \frac{Generation[t] - Demand[t]}{722Wh/\%} \tag{4.2}$$

Current Time

The current time is presented to the optimizer every hour. This is used to initiate the set for the optimization model. Besides the functionality in initiating the model, it is also used to estimate the influence

the control level at the output has. For example, if the output control level dictates that the street lights can only operate at 50% to save power, then this restriction can only save power when the lights would generally have been on. This means that this control level can only save power during the night. This way, the actual savings are highly time-dependent.

4.3.3. Output

The output of the MPC is the control level. The control level can place restrictions on certain appliances to limit power use if needed. The control level is an integer that ranges from 0 to 4. Here, level zero is the base level without any restrictions, and level four has maximum restrictions. Every level above zero places restrictions on the appliances of the previous level and one extra. Because the optimizer cannot predict when and how often these devices are used and how much power this saves, it uses an estimation. The four restrictable devices are the boiler, streetlights, washing machine, and induction cooker. All of the devices have two operational states. They either are in regular operation or restricted operation. The power savings are calculated using the differences of these states on average per hour. In Appendix A.2, the average hourly energy use per appliance is calculated. The power consumption and usage per day of all devices are obtained from the DCG subgroup. The resulting average hourly energy usages are 253Wh, 30Wh, 18.9Wh, and 140.6Wh for the boiler, streetlights, washing machine and, induction cooker, respectively.

For some of the devices, this average might not always be realistic and for other devices, assumptions are made to simplify calculations. Actual savings vary drastically per hour, per day, per person, and per season. Because the MPC recalculates the optimal solution every hour based on the current measurements, a rough estimation on the average suffices. Furthermore, because of the type of optimizer utilized, the control levels will be linearly approximated, and thus accurate calculations might lose their accuracy during this process.

Control Levels

Some restrictions are more intrusive on the behavior of the residents. How intrusive the restrictions are will determine the order in which they are implemented at the control level. This is to minimize inconvenience for the residents. To determine the order, a survey was conducted amongst 31 people. The survey was sent to a group of green-minded people with ages ranging from 15 to 60. The full survey with responses is shown in Appendix C. The results show that the participants rated the induction cooker as the most intrusive restriction. All other three restrictions were deemed almost equally intrusive. Therefore it is beneficial to arrange them from most effective to least effective to reduce the total time restrictions are required. The order as follows from the survey and the previously mentioned argument is:

1. Boiler (B)
2. Street Lights (SL)
3. Washing Machine (WM)
4. Induction Cooker (IC)

Using this order, the control levels can be constructed as shown in Table 4.1. The estimated energy saving per hour for the control levels is calculated by summing the estimated savings of the restricted appliances in that control level. The result is shown in the second to last column. The last column contains the estimated saving on the SoC.

As will be mentioned in Section 4.4.1, the control level is required to be linearized, as a linear solver will be used. The continuous linearization is shown in Figure 4.3. The linearization function can be written as $y_{savings} = 116.49 * x_{CL}$ where $y_{savings}$ is the estimated savings in Wh and x_{CL} the control level. The linearization was chosen to have the lowest MSE while intersecting zero. The intersection in zero is required as the controller would otherwise assume it affects the system when no controller action is taken.

Table 4.1: The restrictions on the boiler (B), street lights (SL), washing machine (WM) and induction cooker (IC) for every control level with their respective estimated energy savings in Wh and SoC percentage per hour.

Control level	Restricted				Estimated energy	Estimated savings
	B	SL	WM	IC	savings per hour [Wh]	on SoC per hour [%]
0	No	No	No	No	0	0
1	Yes	No	No	No	253	0.350
2	Yes	Yes	No	No	283	0.367
3	Yes	Yes	Yes	No	301.9	0.391
4	Yes	Yes	Yes	Yes	442.5	0.573

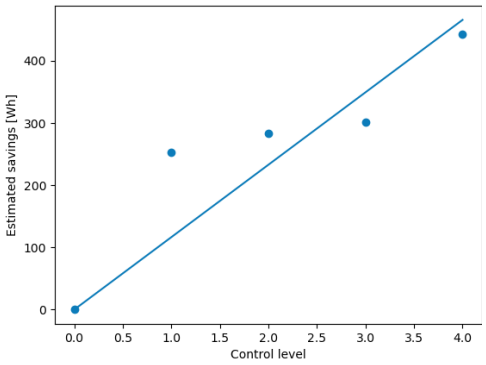

Figure 4.3: The estimated savings in Wh per control level (blue dots) and the linearization of the control level (blue line).

4.4. Optimization Model

After clearly defining the inputs and the output, the optimization model can be constructed. In the following sections, the optimizer selection, the mathematical structure of the model, and the relevant code will be discussed. First, the optimizer selection is discussed, followed by elaboration on sets, parameters and variables, the objective function, and the constraints.

4.4.1. Optimizer Selection

To create the optimization model, the Pyomo optimization framework is used [43, 44]. Pyomo is a collection of Python packages for formulating optimization problems. Pyomo itself is not a solver but is compatible with many respected Python solvers. Pyomo also has a handful of solvers that it is optimized for. From these solvers, the GNU Linear Programming Kit (GLPK) was chosen [45]. GLPK is a linear optimizer that is specialized in large-scale linear programming. Furthermore, it allows for discretized optimization, which some non-linear counterparts lack. The model itself is also easily linearized, which makes it run faster. Lastly, both Pyomo and GLPK are free to use for commercial purposes, making them suitable for implementation in the tiny house community.

4.4.2. Pyomo

A Pyomo model is defined using sets, variables, parameters, an objective, and constraints. The order of these is critical, as there exist dependencies between the five concepts. The parameters and variables depend on the set. Further, both the objective and the constraints depend on the set, parameters, and variables. These dependencies also determine the order in which they are treated in the following sections. The code for the initialization of the model is shown in Appendix E.2.1 and will be discussed in more detail in the following sections.

4.4.3. Set, Parameters, and Variables

First, the set is defined as both the parameters and variables depend on it. After the set is defined, the parameters and variables are defined. If all three are defined, this concludes the input, output, and internal variables.

Set
The set defines the index of the parameters and variables. This means that the parameters and variables indexed in the set can obtain a different value at every index. For the case of this MPC, the set is defined in time or more specifically in hours as this is the time step of the MPC. It ranges the whole time horizon, which is one week. This results in a set containing integers representing hours ranging from 0, the current hour, to $24hours/day * 7days = 168hours$, precisely one week later.

Parameters
Parameters are values that are immutable during optimization. They are the input for the optimizer and can be changed every time the optimizer recalculates. Two types of parameters exist. These are indexed parameters that change over time and constant parameters, that remain constant over time. The names used for the parameters are the same as used in the code in Appendix E.2.1. The indexed parameters are:

- SoCDiff: The SoCDiff represents the SoC of the battery due to the predicted difference between generation and demand at every hour. This is the output of the prediction model that uses weather forecast data as an input.
- setPoint: The set point in this case is the reference SoC. The controller will try to get the real SoC as close as possible to this reference SoC. The set point is kept constant for this project but could be altered during different seasons. For example, during the winter, when there is less solar power generation, the set point might be set higher such that the controller takes controller actions sooner.
- weight: The weight is the cost attached to the difference between the predicted SoC and the set point. It influences the minimization of the cost function in the objective. As weather forecasts for hours further in the week are less accurate, a lower weight can be given to the deviation far ahead.
- dCost: The dCost is the cost attached to changing the control level.
- cCost: The cCost is the cost attached to the control level itself. A higher control level is more intrusive, and thus has a higher cost.

The non-indexed parameters are:

- SoCIni: The SoCIni represents the measured SoC of the battery at hour zero.
- controlLevelIni: The controlLevelIni represents the currently implemented control level.
- dMax: dMax is the maximum allowed change in the control level per hour. This prevents the controller from jumping from the lowest to the highest restriction-level in one hour.

Variables
Variables are values that are mutable during optimization. They are mutable according to the constraints. Variables do not have to be provided outside of the initialization of the optimizer. They function purely within the optimizer itself. Variables have one more important property, namely their domain that can bound them. The names used for the variables are the same as used in the code in Appendix E.2.1. The first two variables are:

- controlLevel: The control level is used as the output of the optimizer. It is an integer ranging from 0 to 4. This control level then results in restricted appliances as mentioned in Table 4.1.
- SoC: The SoC represents the measured SoC at $t = 0$ and the calculated SoC due to the predictions on generation and demand for $t > 0$.

Two more variables are required to represent the deviation of the SoC from the set point and the deviation from zero of the control level. The former is represented in the code by the variables deltaSetPointPos and deltaSetPointNeg and the latter by controlLevelPos and controlLevelNeg. The reason for utilizing a total of four variables instead of two is discussed in Section 4.4.5

4.4.4. Objective Function

After defining the set and all the parameters and variables, the objective function can be defined. For the MPC, this is done using a cost function, J. In an objective function, a cost or weight is assigned to undesirable behavior. The objective is to minimize the total cost. The objective function, or cost function, of the MPC consists of three behaviors that have a cost associated with them, one of which is the main objective. Within MPC, two options for formulating the main objective are often considered. The formulation can be done using a reference trajectory or a set point. A reference trajectory is the ideal trajectory the controller tries to follow. This is used in cruise controllers, as a smooth transition from one speed to the next is required. The trajectory shows the ideal path towards a desired state. A set point is the ideal state of the system. This is used when the path towards the end goal is not required to follow a trajectory. As the trajectory of the SoC itself is not relevant, the objective function was chosen to be formulated using a set point, rather than a reference trajectory.

Deviation from Set Point Cost

The first - and main - part of the cost function assigns a cost, $w_{\Delta sp}[t]$, to the difference between the SoC, $x_{SoC}[t]$, and the set point, $x_{sp}[t]$, at every time, t, in the horizon, N. The set point is initialized as 60% SoC. Intuitively, the set point would be at the middle of the range of an SoC: 50%. Here, however, the SoC is only used for a range of 10%-100%. The middle then becomes 55%. Because overproducing is preferred over underproducing, this is rounded to 60%. In Section 4.6.2, the set point is tuned to obtain the best results. This is the part of the cost function that will reduce the actual grid dependency. The goal is to minimize the deviation from the set point over the whole time horizon. This is done by summing the deviation for every time over the horizon. Because the cost function is minimized, the difference needs to be squared as the deviation should be non-negative. Squaring a function, however, is not linear. Thus another approach is required. Using an absolute function is also not linear, but within optimization modeling, methods exist for handling these absolute functions. This will be discussed in Section 4.4.5. The first cost function, $J_{\Delta sp}$, can be mathematically represented as in Equation (4.3).

$$J_{\Delta sp} = \sum_{t=0}^{N} w_{\Delta sp}[t] * abs(x_{sp}[t] - x_{SoC}[t]) \tag{4.3}$$

Change in Control Level Cost

Changing the control level and thus changing the restrictions every hour is undesirable as the restrictions can feel random to the residents. Minimizing this discomfort is important for creating a livable community. The cost, $w_{DCost}[t]$, is assigned to the change in control level, $x_{CL}[t] - x_{CL}[t-1]$, at every time over the horizon. The change in control level is always zero for $t = 0$. This way, the summation can be performed over the whole time horizon. The second cost function, J_{DCost}, is shown in Equation (4.4).

$$J_{DCost} = \sum_{t=0}^{N} w_{DCost}[t] * abs(x_{CL}[t] - x_{CL}[t-1]) \tag{4.4}$$

Control Level Cost

The last cost function assigns a cost to the control level itself. A higher control level is less desirable as it places restrictions on the residents of the community. The cost, $w_{CCost}[t]$, is assigned to the control level, $x_{CL}[t]$. Because the control level is always positive, it does not require an absolute function. If, however, a negative control level would be added, the absolute value could be added in the same way as for the other cost functions. Just as for the other cost functions, for the last cost function, J_{CCost}, the summation is performed over the whole time horizon. The last cost function is shown in Equation (4.5)

$$J_{CCost} = \sum_{t=0}^{N} w_{CCost}[t] * x_{CL}[t] \tag{4.5}$$

Cost Function

By summing the cost functions mentioned in the previous section $J_{\Delta sp}$, J_{DCost}, and J_{CCost}, the total cost function, J, can be calculated. The total cost function will be minimized by the model. Minimization is

then called the sense of the objective function. The total cost function, as it is obtained from summing Equation (4.3), Equation (4.4), and Equation (4.5), is shown in Equation (4.6).

$$\begin{aligned} J =& J_{\Delta sp} + J_{DCost} + J_{CCost} \\ =& \sum_{t=0}^{N} \{w_{\Delta sp}[t] * abs(x_{sp}[t] - x_{SoC}[t]) \\ &+ w_{DCost}[t] * abs(x_{CL}[t] - x_{CL}[t-1]) \\ &+ w_{CCost}[t] * x_{CL}[t]\} \end{aligned} \tag{4.6}$$

4.4.5. Constraints

Constraints in Pyomo perform the function of both constraints - assigning an upper and/or lower bound - and equations - assigning equalities. Although one could argue that an equation can be considered a constraint, within mathematical modelling, they will be treated separately. As mentioned before, constraints can also be used to handle the non-linear absolute function in linear programming. The latter is discussed first, after which the equations and constraints will be treated.

Handling the Absolute function

To show the way an absolute function can be used within linear programming, an example is used [46]. Consider the following:

$$\min_{J} \quad J = \sum_{i \in I} c_i |x_i| \quad c_i > 0 \tag{4.7}$$

To avoid using the absolute function $|x_i|$, x_i can be replaced by two other values. Instead of using $|x_i|$ in the cost function, it gets replaced as follows:

$$\begin{aligned} |x_i| &= x_i^+ + x_i^- \\ x_i &= x_i^+ - x_i^- \\ x_i^+, x_i^- &\geq 0 \end{aligned} \tag{4.8}$$

As long as either x_i^+ or x_i^- is zero for all values of i, these statements hold and $x_i = x_i^+$ for $x_i \geq 0$ and $x_i = -x_i^-$ for $x_i < 0$. Now assume neither is zero and thus both are a positive value. Let $\delta = min\{x_i^+, x_i^-\}$. Subtracting $\delta > 0$ from both x_i^+ or x_i^- leaves the value of x_i unchanged but reduces the value of $|x_i|$. This contradicts the optimal solution as the cost function could be further minimized by a factor of $2\delta c_i$. The absolute function can now be altered in the cost function in the same way as in the example. This gives Equation (4.9).

$$\begin{aligned} abs(x_{sp}[t] - x_{SoC}[t]) &= x_{\Delta sp}^+[t] + x_{\Delta sp}^-[t], \qquad x_{\Delta sp}^+[t], x_{\Delta sp}^-[t] \geq 0 \\ abs(x_{CL}[t] - x_{CL}[t-1]) &= x_{CL}^+[t] + x_{CL}^-[t], \qquad x_{CL}^+[t], x_{CL}^-[t] \geq 0 \end{aligned} \tag{4.9}$$

Equations

The optimizer is governed by one main equation that defines the SoC of the battery and two initialization equations that initialize the current control level and SoC. In Equation (4.10) the initialization of the control level at $t = 0$ is done by setting it equal to the controlLevelInit parameter. The SoC is initialized in similar to the control level, which is shown in Equation (4.11).

$$x_{CL}[0] = x_{CLinit} \tag{4.10}$$

$$x_{SoC}[0] = x_{SoCinit} \tag{4.11}$$

Lastly, the equation that describes the SoC over time remains. The SoC, x_{SoC}, at time t is calculated by summing the SoC at time $t-1$, the SoC due to the predicted generation and demand, $x_{\Delta SoC}$, at time $t-1$, and the estimated effect of the controller at that control level, x_{CL}, at time $t-1$. This equation holds for $t \geq 1 \vee t \leq N$ with N being the horizon length, as shown in Equation (4.12). Because of the linearization of the control level and the conversion from Wh to SoC%, x_{CL} is multiplied with 116.49 and divided by 722, which results in a multiplication by $116.49/722 = 0.1613$.

$$x_{SoC}[t] = x_{SoC}[t-1] + x_{\Delta SoC}[t-1] + 0.1613 * x_{CL}[t-1] \quad \text{for} \quad t \geq 1 \vee t \leq N \tag{4.12}$$

Constraints
To limit the change in control level, a constraint needs to be formulated. It constrains the control level to change more than Dmax every hour. The constraint is split in two, where one constrains the control level from going down more than Dmax and the other constrains the control level from going up more than Dmax. The constraints are shown in Equation (4.13) where x_{CL} is the control level, and D_{max} is the maximum allowed change in the control level.

$$\begin{aligned} x_{CL}[t] - x_{CL}[t-1] &\leq D_{max} \\ x_{CL}[t] - x_{CL}[t-1] &\geq -D_{max} \end{aligned} \tag{4.13}$$

4.4.6. Finishing the Optimizer
The set, parameters, variables, the objective, and the constraints are all defined. Now that everything is defined, a mathematical notation for the optimization problem can be created by adding everything together. The complete optimization problem is shown in Equation (4.14).

$$\begin{aligned} \min_{J} \quad & J = \sum_{t=0}^{N} \{ w_{\Delta sp}[t] x^{+}_{\Delta sp}[t] + x^{-}_{\Delta sp}[t] \\ & + w_{DCost}[t] * x^{+}_{CL}[t] + x^{-}_{CL}[t] \\ & + w_{CCost}[t] * x_{CL}[t] \} \end{aligned}$$

$$\begin{aligned} \text{s.t.} \quad & x_{sp}[t] - x_{SoC}[t] = x^{+}_{\Delta sp}[t] + x^{-}_{\Delta sp}[t] \\ & x^{+}_{\Delta sp}[t], x^{-}_{\Delta sp}[t] \geq 0 \\ & x_{CL}[t] - x_{CL}[t-1] = x^{+}_{CL}[t] + x^{-}_{CL}[t] \\ & x^{+}_{CL}[t], x^{-}_{CL}[t] \geq 0 \\ & x_{CL}[0] = x_{CLinit} \\ & x_{SoC}[0] = x_{SoCinit} \\ & x_{SoC}[t] = x_{SoC}[t-1] + x_{\Delta SoC}[t-1] \\ & \qquad + 0.1613 * x_{CL}[t-1], \qquad \text{for } t \neq 0 \\ & x_{CL}[t] - x_{CL}[t-1] \leq D_{max} \\ & x_{CL}[t] - x_{CL}[t-1] \geq -D_{max} \end{aligned} \tag{4.14}$$

4.5. Simulation and Testing
Now that both the prediction model and optimization model, as shown in Figure 4.2, are completed, a simulation is conducted and testing can be performed. Before simulating, the three different costs are initialized and later improved. After that, one cycle of the MPC is simulated to verify its proper function. After this is verified, the control scheme will be briefly explained as well as the simulated controller effect. When all is finished, simulations for multiple cycles will be performed.

4.5.1. Cost Factors
The three different cost factors $w_{\Delta sp}$, w_{DCost}, and w_{CCost} are all dependent on the hour. They decay linearly to zero. This way, the weight of an hour always has more impact on the cost function than the next. This is done because weather predictions get more inaccurate the further in the future they are, and therefore the predictions should influence the controller less. The weight $w_{\Delta sp}$ is the weight associated with the deviation from the set point. The other two weights are calculated relative to the first. Using this relative term allows for setting the importance of both w_{DCost} and w_{CCost}. For example, if $w_{DCost} = 5 * w_{\Delta sp}$, this means that within 5% of the set point, a change in the control level is deemed as a worse option regardless of the deviation. Alternatively, if $w_{CCost} = 5 * w_{\Delta sp}$ then a control level of 4 is deemed less beneficial than level 3 when within 20% from the set point. The initial values of w_{DCost} and w_{CCost} are 5 and 2 respectively. These will later be changed to look for possible improvements.

4.5.2. One Cycle Optimization

After initializing the weights, one cycle of the optimization can be run, the code of which can be found in Appendix E.2.2. Here the controller optimizes one week or 168 hours of data according to the cost factors. Plotting the controller action and predicted SoC provides relevant information about the controller.

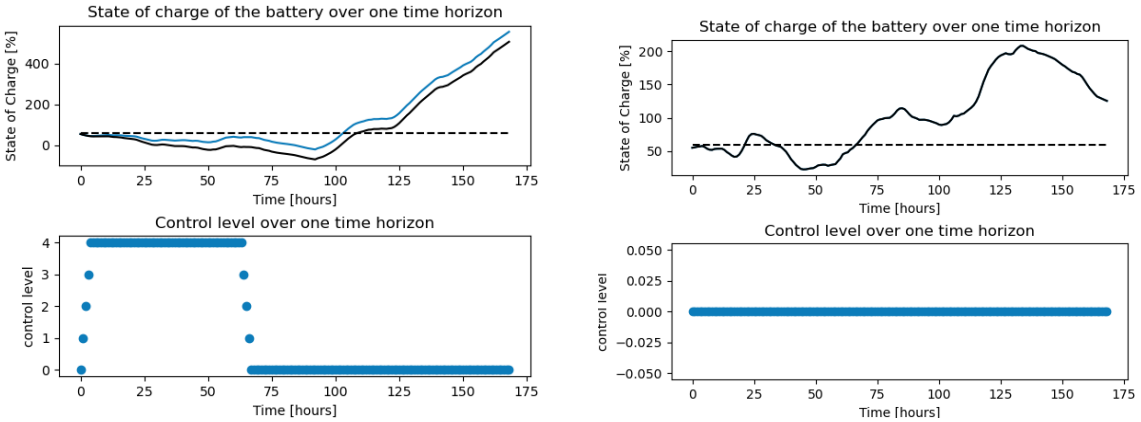

Figure 4.4: Plot of the SoC of the battery over one week (upper) with the SoC with controller action (blue) and the SoC without controller action (black) and the controller action over one week (lower) starting at hour 600 (a) and at hour 220 (b) of the year.

Figure 4.4a gives an example of a plot of the SoC starting at hour 600 and initialized at 55% is given. Visible is that it surpasses 100%, which is, of course, impossible. Limiting this would enforce non-linearity on the system, which is not solvable for a linear solver. For the multi-cycle simulation, this issue is tackled as described in the next chapter. Furthermore, the plot shows that the constraint on raising and lowering the control level performs as expected. Figure 4.4b shows the influence both w_{DCost} and w_{CCost} have on the system. As of hour 220, it is visible that it falls below the set point, but the controller does not respond because a change in control level and higher control level are deemed to be worse than remaining as is. The controller can conclude this from future predictions that show high generations: thus responding is unnecessary.

4.5.3. Control Scheme

To not exceed the 100% and 10% SoC boundaries of the battery, a control scheme is needed. This involves when to connect to the grid and how to use the batteries, thus meeting RQ-M.CNS.7. Here a simple control scheme is implemented as shown in Figure 4.5. It shows a Finite State Machine (FSM) that determines the power flow. In short, when the generation is higher than demand, the battery is charged, and when demand is higher than generation, the battery is used. This holds until the battery is either at 100%, when it supplies to the grid until generation goes below demand, or at 10%, when it uses the grid until generation rises above demand.

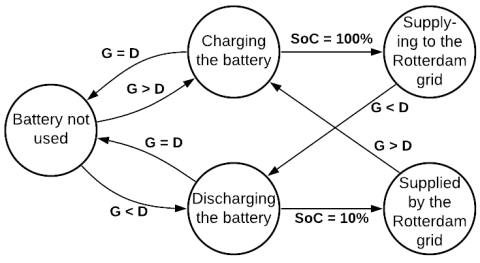

Figure 4.5: Sketch of the FSM of the control scheme which depends on the Generation (G), Demand (D) and the State of Charge (SoC).

4.5.4. Controller Effect

Using the estimated savings of the controller, as mentioned in Table 4.1, for the simulation would provide an unrealistic result, as all the savings are highly time-dependent. In the code provided in Appendix E.2.3, the effect is calculated using the current hour of the day as an input. The effect of the streetlights is only during the night, and the other restrictions are only effective during the day. The latter's effect uses the built-in 'random' library from Python to randomize the actual effect to create a more realistic simulation, as the time the residents are home and use the appliances varies from day to day.

4.5.5. Full Simulation

The entire simulation uses the predictions to optimize for one week in the same way as the one-cycle optimization. After the optimization is performed, it implements the controller action and calculates how much energy is saved. Then the actual SoC is calculated according to the actual generation and demand and the controller scheme and updated. This process is iterated over one month or year and can then be plotted. Relevant parameters are printed and can be used for improving the controller. The code for the simulation is presented in Appendix E.2.4.

4.6. Results

After finishing the simulations, the results can be analyzed. In the following sections, the initial results are discussed, and simulations with different parameters are run to explore the controller's potential.

4.6.1. Initial Results

The first simulation was performed using the initial values 5, 2, and 60 for w_{DCost}, w_{CCost} and the set point, respectively. To make the predictions, the weather, wind generation, and demand data of 2013 were used. Because no solar data was available for this period, the predictions and generation of the year 2019 were used. Table 4.2 shows the energy delivered to and taken from the grid with and without the controller. It also shows the relative improvement due to the controller. The improvement, I, is calculated using Equation (4.15) with $E_{without}$ the energy taken from the grid without the controller, and E_{with} the energy taken from the grid with the controller.

$$I = \frac{E_{without} - E_{with}}{E_{without}} * 100\% \tag{4.15}$$

Table 4.2: Results of the simulation running for one year, with initial values 5, 2, and 60 for w_{DCost}, w_{CCost}, and the set point, respectively, showing energy taken from the grid, percentage the controller is active and the relative improvement with the controller.

Energy used from the grid without controller [kWh]	Energy used from the grid with controller [kWh]	controller active [%]	improvement [%]
775.55	629.76	9.19	18.79

Over one year, 629.76 kWh is required from the grid when using the controller instead of 775.55 kWh. The usage of the tiny house community is 41 kWh, as calculated by the DCG sub-team. The improvement of 18.79% saves $(775.55kWh - 629.76kWh)/(41kWh/day) = 3.56days$ worth of energy usage of the village. During the summer and spring, the controller is inactive, as is shown in Figure 4.6a, which shows the months April, May, and June. Interestingly, the controller does not raise the control level during the dip below the set point as it anticipates on the good weather that is predicted. Likely, a simpler controller would have reacted to this dip.

Because the control level remains at zero during the summer and spring, all the controller actions are in the winter. This means that the controller activity is effectively twice the average during the winter and thus becomes $2 * 9.19\% = 18.38\%$. This can also be seen in Figure 4.6b, showing months January through March.

To show the effect of generation and demand on the battery's SoC, a sunny week was selected. Figure 4.7 shows the generation, demand, and SoC from hour 1020 through hour 1200. It shows that,

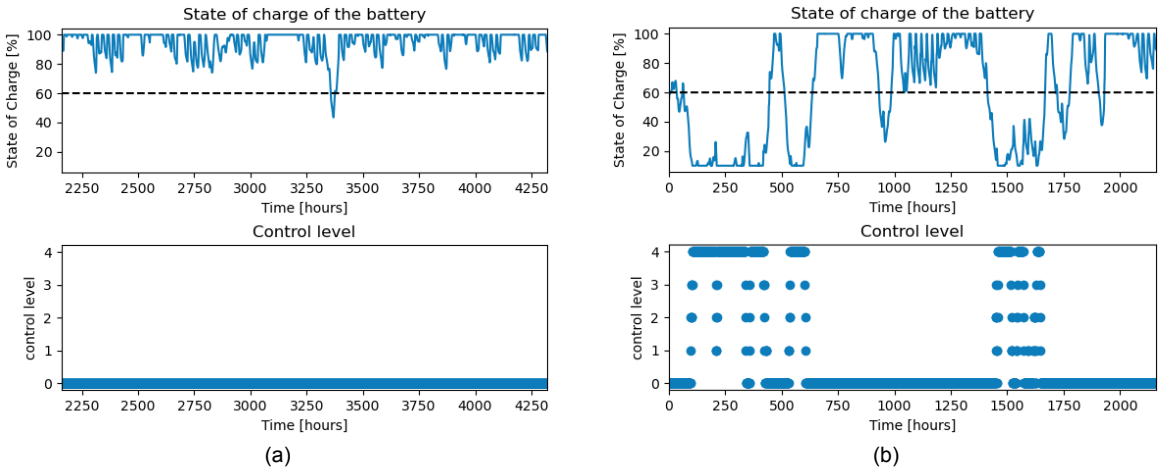

Figure 4.6: Plot of the SoC of the battery during the simulation (upper) and the corresponding controller action (lower) from April 1st till June 30th (a) and from January 1st till March 31th (b).

during the day, the battery charges when the solar generation peaks and that the battery discharges during the night, when there is close to no generation.

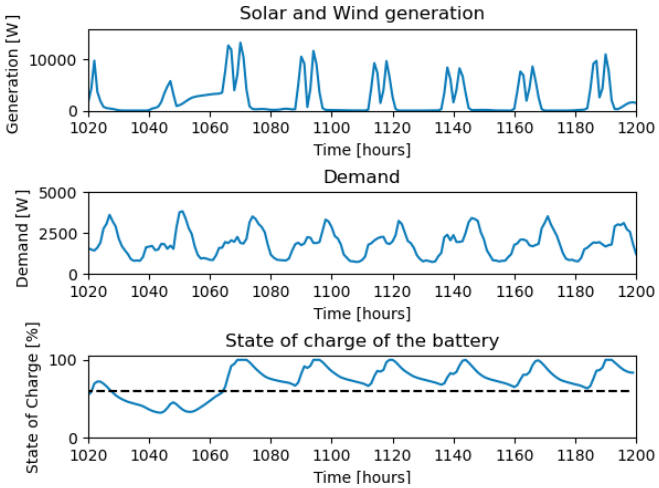

Figure 4.7: Plot of the solar and wind generation, demand, and SoC of the battery during hours 1020 through 1200 of the year. this shows the effect generation and demand have on the SoC.

4.6.2. Improving the Parameters

All previous results were obtained using the initial values for the weights and set point. Changing these values results in different behavior of the controller. Table 4.3 shows the effect of varying the w_{DCost} parameter. The table shows that the controller improvement and activity decrease between $0 * w_{\delta sp}$ and $7.5 * w_{\delta sp}$ as changing the control level becomes costlier. It drops down to 5.3% for $10 * w_{\delta sp}$, which is undesirable. Lowering the value of w_{DCost} will cause more controller activity and more changes of the control level, which is not necessarily the desired behavior, but it does show that the controller could improve 2% more. Table 4.4 shows the effect of varying the w_{CCost} parameter. It also shows that the improvement and activity decrease as w_{CCost} increases, but does not show a drop-off in improvement within the simulated range. When set to zero, the w_{CCost} parameter improves the system by 1%. This might result in unnecessary controller actions that could otherwise be avoided. Lastly, Table 4.5 shows the effect of varying the set point. This has drastic consequences as, for too low values, the controller has no effect at all, but for higher values, the controller is active for long consecutive periods.

In short, improving the controller is achievable, but only at the cost of the residents' comfort. When

the simulation is run with 0, 0, and 80 for w_{DCost}, w_{CCost}, and the set point, respectively, this results in an improvement of 25.61%: $25.61 - 18.78 = 6.83\%$ better than the first case considered. This extreme case has a controller activity of 14.89%. This shows that improving the controller is possible but at the cost of restricted appliances.

Table 4.3: Results of the simulation running for one year, with initial values 2 and 60 for w_{CCost} and, the set point respectively and a varying w_{DCost}, showing energy taken from the grid, percentage the controller is active, and the relative improvement with the controller.

w_{DCost}	Energy used from the grid without controller [kWh]	Energy used from the grid with controller [kWh]	controller active [%]	improvement[%]
$0 * w_{\delta sp}$	775.55	617.54	9.88	20.36
$2.5 * w_{\delta sp}$	775.55	625.08	9.51	19.39
$5 * w_{\delta sp}$	775.55	629.76	9.19	18.78
$7.5 * w_{\delta sp}$	775.55	633.30	8.92	18.33
$10 * w_{\delta sp}$	775.55	734.29	3.92	5.3

Table 4.4: Results of the simulation running for one year, with initial values 5 and 60 for w_{DCost} and the set point, respectively and a varying w_{CCost}, showing energy taken from the grid, percentage the controller is active and, the relative improvement with the controller.

w_{CCost}	Energy used from the grid without controller [kWh]	Energy used from the grid with controller [kWh]	controller active [%]	improvement[%]
$0 * w_{\delta sp}$	775.55	625.99	9.46	19.27
$1 * w_{\delta sp}$	775.55	632.38	9.37	18.45
$2 * w_{\delta sp}$	775.55	629.76	9.19	18.78
$3 * w_{\delta sp}$	775.55	632.00	9.09	18.50
$4 * w_{\delta sp}$	775.55	638.05	8.71	17.71

Table 4.5: Results of the simulation running for one year, with initial values 5 and 2 for w_{DCost} and w_{CCost}, respectively and a varying set point, showing energy taken from the grid, percentage the controller is active, and the relative improvement with the controller.

Set point	Energy used from the grid without controller [kWh]	Energy used from the grid with controller [kWh]	controller active [%]	improvement[%]
40	775.55	775.55	0	0
50	775.55	775.55	0	0
60	775.55	629.75	9.19	18.78
70	775.55	607.45	11.71	21.66
80	775.55	583.73	14.07	24.72

5

Conclusion

The thesis demonstrated the design and testing of a control system. The system consisted of two parts, namely forecasting and control. Forecasting was performed using a prediction model created using ANNs. Three different models were created for predicting solar generation, wind generation, and energy demand. The controller was implemented using an MPC to make use of the prediction model. The MPC utilizes an optimization model that tries to minimize the time spent connected to the grid.

The system successfully predicts both energy generation and demand, as mentioned in RQ-M.CNS.3 and RQ-M.CNS.4. The prediction model uses the weather data at the input. The thesis discusses the possibility of retrieving this data from the internet as in RQ-M.CNS.1. The system can receive and use the hourly measurements for the predictions and the controller mentioned in RQ-M.CNS.2. Although not implemented, the report discusses the possibility of updating the database and the prediction model, partially satisfying RQ-S.CNS.1 and RQ-S.CNS.2. The forecasting was done with a mean error of 10.11%, 12.56%, and 6.95% for solar, wind, and demand, respectively. This shows an accuracy higher than 80%. Thus the system meets RQ-NF.CNS.1.

The controller uses the predictions to estimate the SoC of the battery for the upcoming week. This satisfies RQ-M.CNS.5. After simulating the controller system for a year, a reduction of 18% to 25% of energy needed from the grid could be achieved. This reduced the time spent connected to the grid by altering the power use, complying with RQ-M.CNS.6. The control scheme allowed the battery to remain between its critical boundaries during the simulation. This meets RQ-M.CNS.7. Lastly, all controller actions and the forecasts could be performed seven days ahead, satisfying RQ-NF.CNS.2.

All 'must have' requirements of the CNS subsystem were met as well as all relevant system requirements (RQ-C.SYS.2 and RQ-M.SYS.5). All 'should have' requirements were partially met but described in the thesis as well. Due to time constraints, none of the 'could have' requirements were met. Lastly, all non-functional requirements were met as described above.

Future Work

While the results comply with the requirements, there is still room for improvement and future work:

- Data from actual tiny houses can be used to perform hyperparameter tuning. This would be a better fit than the scaled values used in this thesis.
- More control levels can be utilized than the four mentioned in this thesis. These could reduce power consumption more and be less intrusive, further increasing the effect of the MPC.
- A more optimal control scheme could be researched. Although simple to implement, the current control scheme reduces the battery life by deep cycling the batteries. Smarter control schemes could consider this.
- If a prototype of a tiny house community, using this DC grid, could be built, testing can be performed to validate the simulation and allow for further development of the controller.
- The software can be implemented on a microprocessor. This could allow for testing the run time of the prediction and optimization. If deemed possible, the time step, currently one hour, could be reduced to increase the controller effect.

A

Derivations

A.1. Processing Sun Data

The following steps were taken to extract the information of the solar data, to remove inconsistencies, and to replace the missing values.

1. First, the time and date, with their corresponding voltage and current, are extracted from the data set.

2. The time and date are used to calculate what hour of the year the moment corresponds to. It will range from 1 - January 1st, 00h00 - to the last, or $24 * 365 =$8,760th hour of the year - December 31st, 23h00. This supporting variable will facilitate the creation of a single data point per hour. It is needed because of the inconsistent many-to-one correspondence of the data set - an irregular amount of data points point to the same hour.

3. Before the different data points that correspond to the same hour can be combined, a conversion of voltage and current to power must occur using the formula for electrical power: $P = V * I$.

4. Combining data points of the same hour is done by taking the average of all of them. There is, however, the problem that the amount of data points per hour is inconsistent and thus unknown. There are different ways to solve this. Here, the average is calculated using a weight factor that changes as more samples corresponding to the same hour are encountered. The weight factor is unique for every hour, is initialized for every hour as 1, and increases by 1 for a particular hour, every time a calculation for that hour is made. It is implemented using the following equation, where P is the power during a certain hour, w is the weight factor of this hour, and p is a new data point corresponding to the same hour.

$$P_{\text{new}} = P_{\text{old}} * w + \frac{p}{w+1} \tag{A.1}$$

5. Now, there is at most a single data point per hour. However, there are still many missing data points: over 4% of hours in the year do not have a corresponding power yet. There are several options to fill these gaps. The missing data points could be left out entirely. This would not pose issues for training the prediction model. However, it would gravely complicate making simulations. Another option is to fill the data points with the overall average. That would make simulations possible. However, this could have an immense impact on the training of prediction models: data points during the night that would previously never contain power generation now suddenly do. This would reduce the capabilities of the ANN's learning patterns. An option that combines the advantages of the previous two, but has none of their disadvantages, is replacing the missing data with the value of the same hour, the day before. This should have a minimal influence on the forecasting - since it is only for 4% of the data points - and enables the simulation of long periods.

A.2. Control Level Calculations

Street Lights
When turned on, the street lights use a total of 120 Wh per hour. These lights are on for an average of eight hours per day when in normal operation. This results in $120 * 8 = 960Wh$ per day on average. When restricted, the street lights will be on at 50% power during the first two hours and last two hours of the night and turned off during the rest of the night. Though they are still allowed to turn on when movement is detected, as this is only a fraction of the night, it is not considered in the estimation. This results in $(2hours * 0.5 + 4hours * 0 + 2hours * 0.5) * 120W = 240Wh$ per day when in restricted mode. Thus on average, the street lights save $(960 - 240)/24 = 30Wh$ per hour. In reality, the street lights either save nothing, 60 Wh or 120 Wh per hour. This is, however, difficult to implement in a mathematical optimizer.

Boiler
When in regular operation, the boiler uses high amounts of power for short durations spread over an entire day. One boiler uses on average 1700 Wh per day. Twelve boilers, one for every tunus, use a total of $12 * 1700 = 20400Wh$ per day or $20400/24 = 850Wh$ per hour. In restricted mode, the boiler is turned off if no one is home. Let us assume that every tunus resident works a total of 40 hours a week and spends an additional 10 hours per week on other activities like doing groceries, meeting friends, or exercising. During these 50 hours every week, a resident is not at home, and the boiler is turned off when in restricted mode. Then the boiler is turned off $50/(7 * 24) * 100\% = 29.8\%$ of the time. This results in an average saving of $0.298 * 850 = 253Wh$ per hour.

Washing Machine
The community has two washing machines in the central hub that use an average of 1512 Wh per day. In restricted mode, the washing machine can only operate on eco-mode. All modern washing machines have this capability which can reduce energy consumption by 30% on average [47]. This results in an average saving of $1512/24 * 0.3 = 18.9Wh$ per hour.

Induction Cooker
One induction cooker uses on average 1125 Wh per day during regular operation. All induction cookers together use on average $1125 * 12/24 = 562.5Wh$ per hour. This is roughly equivalent to every household using two induction zones on boost, which is the highest setting. When restricted, the induction cooker is limited to one induction zone in boost and one to half the capacity. This way, proper cooking is still possible while, on average, this allows for a power-saving of 25%. This results in an average saving of $562.5 * 0.25 = 140.6Wh$ per hour.

B

Tables

B.1. Hyperparameter Tuning of Wind Generation

Table B.1: The losses of different combinations of batch sizes and amounts of epochs for wind generation. The loss is given as an unscaled MSE in Wh^2.

epochs	batch size		
	200	500	1000
10	0.03933	0.051497	0.609744
25	0.032044	0.03141	0.051694
50	0.043442	0.027233	0.055289
100	0.03963	0.025429	0.050772
200	0.01912	0.027899	0.033007
500	0.01934	0.025442	0.043416

Table B.2: The losses for different neuron counts in the first layer of a one-layered ANN configuration for wind generation. The losses are given as an unscaled MSE in Wh^2.

neurons	loss $[Wh^2]$
5	0.022607
10	0.020836
15	0.029482
20	0.032803
25	0.035767
30	0.067064
35	0.033442

Table B.3: The losses for different neuron counts in the first and second layer of a two-layered ANN configuration for the wind generation. The losses are given as an unscaled MSE in Wh^2.

Neurons	Neurons layer 1		
layer 2	5	10	15
5	0.02036	0.050966	0.050322
10	0.051871	0.019937	0.019523
15	0.019571	0.018521	0.018366
20	0.020018	0.018288	0.018784

Table B.4: The losses for different neuron counts in the second and third layer of a three-layered ANN configuration for the wind generation. The first layer is taken to have 10 neurons. The losses are given as an unscaled MSE in Wh^2.

Neurons	Neurons layer 2		
layer 3	5	10	15
10	0.018705	0.050833	0.018422
15	0.018505	0.018123	0.018031
20	0.018272	0.0183	0.018073
25	0.017787	0.018022	0.018132
30	0.01891	0.018797	0.018012

B.2. Hyperparameter Tuning of Demand

Table B.5: The losses of different combinations of batch sizes and amounts of epochs for energy demand. The loss is given as an unscaled MSE in Wh^2.

epochs	batch size		
	200	500	1000
10	681748	2117969	4139132
25	691954	630963	848661
50	830852	653520	635523
100	769615	723732	675698
200	737150	924448	700238
500	408707	636226	697733
1000	493284	342509	991998
2000	814383	484059	1702872

Table B.6: The losses of different combinations of neurons in the first (l1), second (l2), third (l3), fourth (l4), and fifth layer (l5) of a five-layered ANN for energy demand. The loss is given as an unscaled MSE in Wh^2.

		l1=150				l1=300			
l4	l5	l2=100		l2=200		l2=100		l2=200	
		l3=50	l3=100	l3=50	l3=100	l3=50	l3=100	l3=50	l3=100
25	10	507554	756537	1050996	2821384	363330	509004	474422	591607
25	20	879559	434989	266685	1193883	721197	446616	576589	514398
50	10	640687	538554	454299	980643	508894	975895	1220269	707768
50	20	353251	544433	291881	303912	499885	407672	833902	727576

B.3. Features of the Weather Data

Table B.7: The 22 features of the weather data set from [27]. Their code and description is given.

Code	Description
YYYYMMDD	date (YYYY=year,MM=month,DD=day)
HH	time (HH uur/hour, UT. 12 UT=13 MET, 14 MEZT. Hourly division 05 runs from 04.00 UT to 5.00 UT
DD	Mean wind direction (in degrees) during the 10-minute period preceding the time of observation (360=north, 90=east, 180=south, 270=west, 0=calm 990=variable)
FH	Hourly mean wind speed (in 0.1 m/s)
FF	Mean wind speed (in 0.1 m/s) during the 10-minute period preceding the time of observation
FX	Maximum wind gust (in 0.1 m/s) during the hourly division
T	Temperature (in 0.1 degrees Celsius) at 1.50 m at the time of observation
T10N	Minimum temperature (in 0.1 degrees Celsius) at 0.1 m in the preceding 6-hour period
TD	Dew point temperature (in 0.1 degrees Celsius) at 1.50 m at the time of observation
SQ	Sunshine duration (in 0.1 hour) during the hourly division, calculated from global radiation (-1 for <0.05 hour)
Q	Global radiation (in J/cm2) during the hourly division
DR	Precipitation duration (in 0.1 hour) during the hourly division
RH	Hourly precipitation amount (in 0.1 mm) (-1 for <0.05 mm)
P	Air pressure (in 0.1 hPa) reduced to mean sea level, at the time of observation
VV	Horizontal visibility at the time of observation (0=less than 100m, 1=100-200m, 2=200-300m,..., 49=4900-5000m, 50=5-6km, 56=6-7km, 57=7-8km, ..., 79=29-30km, 80=30-35km, 81=35-40km,..., 89=more than 70km)
N	Cloud cover (in octants), at the time of observation (9=sky invisible)
U	Relative atmospheric humidity (in percents) at 1.50 m at the time of observation
WW	Present weather code (00-99), description for the hourly division.
IX	Indicator present weather code (1=manned and recorded (using code from visual observations), 2,3=manned and omitted (no significant weather phenomenon to report, not available), 4=automatically recorded (using code from visual observations), 5,6=automatically omitted (no significant weather phenomenon to report, not available), 7=automatically set (using code from automated observations)
M	Fog 0=no occurrence, 1=occurred during the preceding hour and/or at the time of observation
R	Rainfall 0=no occurrence, 1=occurred during the preceding hour and/or at the time of observation
S	Snow 0=no occurrence, 1=occurred during the preceding hour and/or at the time of observation
O	Thunder 0=no occurrence, 1=occurred during the preceding hour and/or at the time of observation
Y	Ice formation 0=no occurrence, 1=occurred during the preceding hour and/or at the time of observation

C

Survey on Control Level

C.1. Questions

Before answering the questions on intrusiveness of the control levels, the participants were asked to read the text as shown in Figure C.1. After reading the text the participants answered in order questions 1 through 4 as shown in Figure C.2, Figure C.3, Figure C.4 and Figure C.5. The answers to these questions can be found in Appendix C in Table C.1

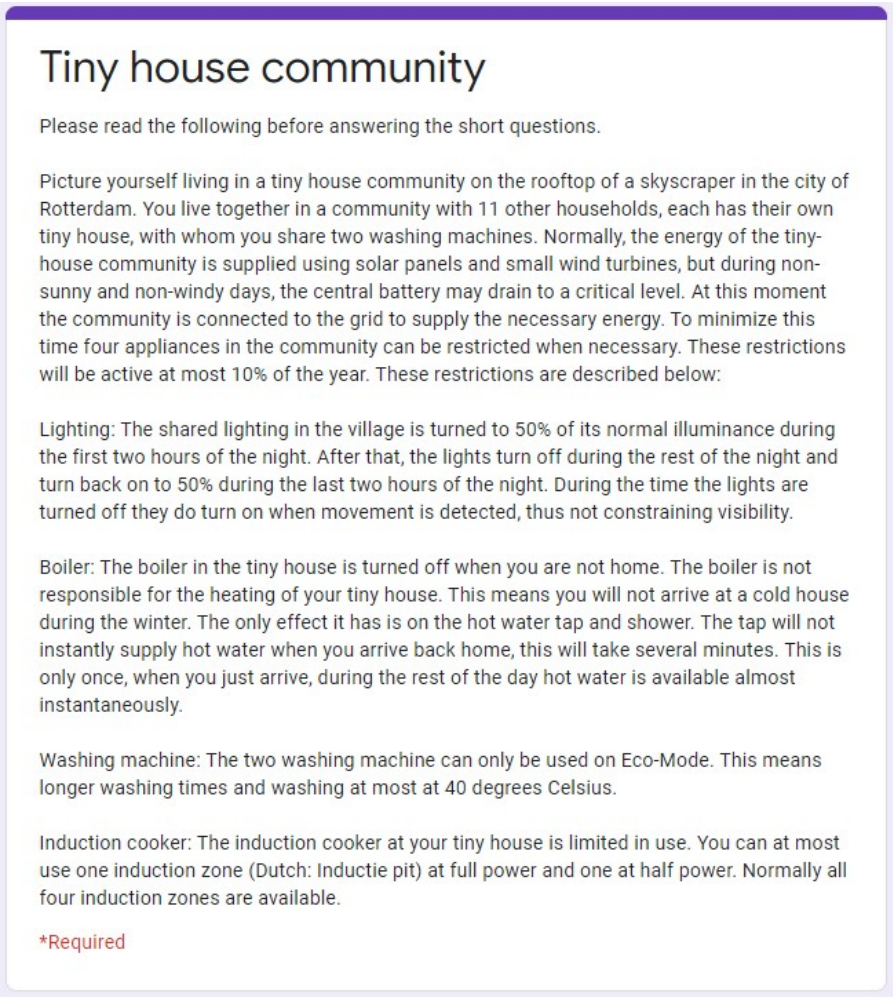

Figure C.1: The reading performed by the participants during the survey, before answering the questions.

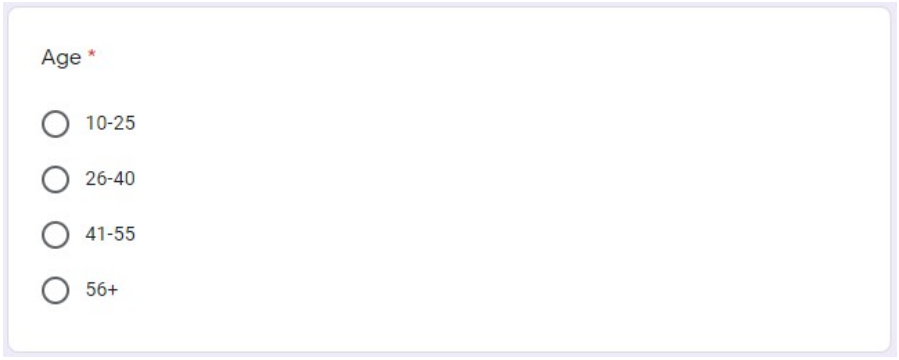

Figure C.2: The first question answered by the participants.

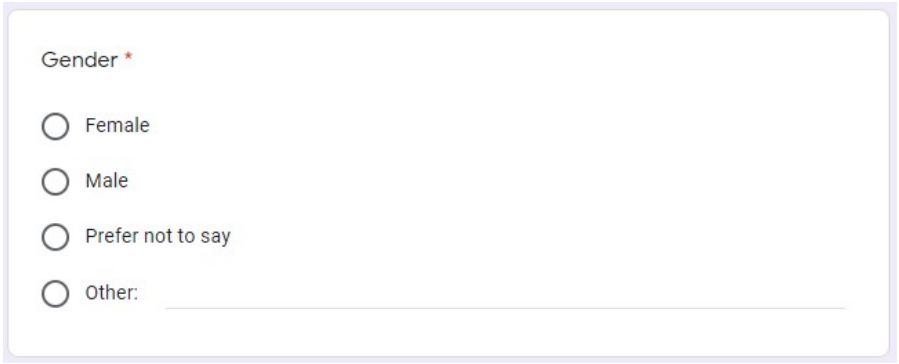

Figure C.3: The second question answered by the participants.

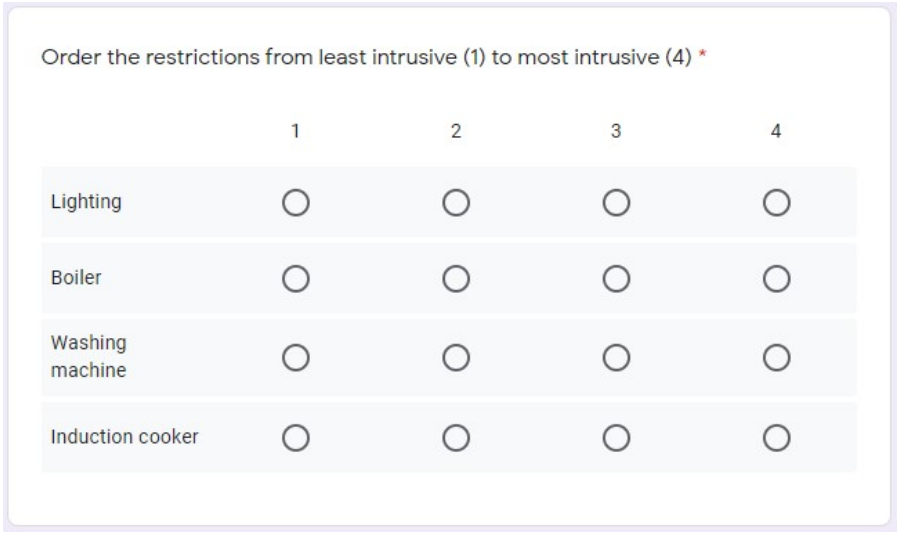

Figure C.4: The third question answered by the participants.

Figure C.5: The fourth question answered by the participants.

C.2. Answers

Appendix C shows the answers of 31 participants to the survey questions as shown in Appendix C.1.

Table C.1: Answers to a survey on intrusiveness of restrictions on appliances

Age	Gender	Lighting	Boiler	Washing machine	Induction cooker	green-minded
10-25	Male	1	2	3	4	Yes
10-25	Female	3	2	1	4	Yes
10-25	Female	3	2	1	4	Yes
10-25	Male	3	4	1	2	Yes
10-25	Male	4	2	1	3	Maybe
10-25	Male	3	2	1	4	Yes
10-25	Other	1	2	3	4	Yes
26-40	Male	1	2	3	4	Yes
41-55	Female	3	2	1	4	Yes
10-25	Male	1	2	3	4	Maybe
26-40	Female	1	2	3	4	Yes
10-25	Male	1	4	2	3	Maybe
10-25	Male	3	1	4	2	Maybe
10-25	Male	1	3	2	4	No
10-25	Male	2	3	1	4	Yes
10-25	Male	2	1	4	3	Yes
26-40	Female	2	1	4	3	Yes
10-25	Female	2	3	1	4	Yes
10-25	Male	4	2	1	3	Yes
10-25	Male	2	1	3	4	No
10-25	Female	1	2	3	4	Maybe
10-25	Male	1	3	2	4	Maybe
56+	Female	3	1	2	4	Yes
56+	Female	2	3	1	4	Yes
10-25	Female	3	2	1	4	Yes
10-25	Male	2	1	4	3	Maybe
10-25	Female	2	1	3	4	Yes
10-25	Female	4	3	1	2	Maybe
10-25	Female	1	4	3	2	Yes
10-25	Female	4	3	1	2	Yes
41-55	Female	1	2	3	4	Maybe
	Total	67	68	67	108	

D

Software

In this chapter, an overview of the software that was used throughout the thesis project, is presented. First, it is argued why Python will be the main programming language that is used. Then, the Python libraries that are needed for the code are summarized.

D.1. Programming Language

Python is used for both the forecasting and the control. This is because of the extensive availability of Python libraries, that offer high-quality algorithms and functions. Further, Python is open-source, which makes it more suitable for implementation in a tiny house village than licensed software, such as MATLAB.

D.2. Python Libraries

When imported into a script, Python Libraries enable the use of its included functions. To handle the Python Libraries, the popular Python Distribution, Anaconda [48], is used.

Libraries for Forecasting

The Python Libraries used for forecasting are shown in Table D.1.

Table D.1: List of Python libraries used for Forecasting

Package
keras [49]
matplotlib [50]
scikit-learn [51]
tensorflow [52]

Libraries for Control

The Python Libraries used for control are shown in Table D.2.

Table D.2: List of Python libraries used for Control.

Package
glpk [45]
matplotlib [50]
pyomo [43, 44]
scipy [53]

E

Code

E.1. Forecasting

This section contains the code that was needed to perform the calculations and simulations done in Chapter 3.

E.1.1. Forecasting Functions

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: Different functions are defined that are used to
 ↪  retrieve/process/evaluate data.
'''


import datetime as dt
import matplotlib.pyplot as plt
import numpy as np
import numpy.ma as ma
import sklearn as sk
import sklearn.impute
from keras.layers import Dense
from keras.models import Sequential
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, cross_val_score,
 ↪  train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow import keras


# print the shapes of the train and test sets
def printSets(X_train, X_test, y_train, y_test):
    print("shape of X_train: {}".format(X_train.shape))
    print("shape of X_test: {}".format(X_test.shape))
    print("shape of y_train: {}".format(y_train.shape))
    print("shape of y_test: {}".format(y_test.shape))


# retrieve the weather data from the data file
def retrieveWeatherData():
```

```
32      # extracting input data from txt file
33      try:
34          data = np.genfromtxt('dataFile.txt',
35          dtype=float, delimiter=',', skip_header=33)
36      except:
37          print('Error while retrieving weather data'); exit()
38
39      return data
40
41
42  # retrieve the wind data from the data file
43  def retrieveWindData():
44      # extracting wind data from csv file
45      try:
46          data = np.genfromtxt('dataFile.csv',
47          dtype=float, delimiter=',', skip_header=1, skip_footer=1)
48          y = data[:,1] # only relevant stuff; all rows of column 1
49      except:
50          print('Error while retrieving y'); exit()
51
52      # extracting weather data from txt file
53      try:
54          data = np.genfromtxt('dataFile.txt',
55          dtype=float, delimiter=',', skip_header=33)
56          X = data[:,3:] # only relevant stuff; all rows of column 3 till end
57      except:
58          print('Error while retrieving X'); exit()
59
60      return X, y
61
62
63  # retrieve the solar data from the data file
64  def retrieveSolarData():
65      y = np.load('savedData.npy')
66
67      try:
68          data = np.genfromtxt('dataFile.txt',
69          dtype=float, delimiter=',', skip_header=33)
70          X = data[:,3:] # only relevant stuff; all rows of column 3 till end
71      except:
72          print('Error while retrieving X'); exit()
73
74      return X, y
75
76
77  # retrieve the demand data from the data file; it is also already processed
78  def retrieveDemandData():
79      # extracting input data from txt file
80      try:
81          data = np.genfromtxt('dataFile.txt',
82          dtype=float, delimiter=',', skip_header=33)
83          X = data[:,[1,2,7,10,21,23]] # only relevant stuff:
84          # select YYYYMMDD (col 1; datum), HH (col 2; hour), T (col 7;
              ↪ temperature),
85          # SQ (col 10; sunshine duration), R (col 21; rain), O (col 23;
              ↪ storm)
```

```
86      except:
87          print('Error while retrieving input data'); exit()
88
89      # we want the weeknumber and daynumber instead of the date
90      timeInfo = np.empty((0,2), int)
91      for i in X[:,0]:
92          # get the date info from the data file
93          year = int(str(i)[0:4])
94          month = int(str(i)[4:6])
95          day = int(str(i)[6:8])
96
97          # make a date and season from the date info
98          time = dt.datetime(year, month, day)
99          season = time.month%12//3+1 # month2season: from
              ↪ https://stackoverflow.com/a/44124490
100         month = time.month
101
102         # timeInfo will contain the month and the daynumber (%u)
103         timeInfo = np.append(timeInfo, np.array([[month,time.weekday()]]),
              ↪ axis=0)
104
105     # the date-column is replaced by a season-number and daynumber column
106     X = np.append(timeInfo, np.delete(X,0,1), 1)
107
108     # extracting output data from csv file
109     try:
110         data = np.genfromtxt('dataFile.csv',
111         dtype=float, delimiter=',', skip_header=1, skip_footer=34992)
112         y = data[:,2:-2] # only relevant stuff:
113         # select YYYYMMDD (col 1; datum), HH (col 2; hour), T (col 7;
              ↪ temperature),
114         # SQ (col 10; sunshine duration), R (col 21; rain), O (col 23;
              ↪ storm)
115     except:
116         print('Error while retrieving output data'); exit()
117
118     # conversion of /15min to /1hr data
119     y = y.reshape(-1,4,y.shape[-1]).sum(1) # summing every 4 columns
120
121     # dealing with nans: from https://stackoverflow.com/q/18689235
122     y = np.where(np.isnan(y), ma.array(y,
          ↪ mask=np.isnan(y)).mean(axis=1)[:, np.newaxis], y)
123     y = y.sum(1) # summing all 'households'
124
125     # scaling the output data
126     y_days = y.reshape(-1,24).sum(1) # create /day data
127     mean_of_one_day_liander = np.mean(y_days) # calc mean of one day
128     mean_of_one_day_tunect = 41000 # mean of 1 day at tunect (from DCG)
129     scalingFactor = mean_of_one_day_tunect/mean_of_one_day_liander
130     y = scalingFactor*y # scaling the data
131
132     return X, y
133
134
135 # make a single model and evaluate it with the test set
```

```
def trainWithoutCurve(X_train, y_train, X_test, y_test, model):
    model.fit(X_train,y_train)
    y_pred = model.predict(X_test)
    MSE = mean_squared_error(y_test, y_pred)
    return MSE


# make a single model and show the learning curve
def trainWithCurve(X_train, y_train, model):
    history = model.fit(X_train,y_train)
    print(history.history.keys())

    # summarize history for loss
    plt.plot(history.history['loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.show()


# make multiple models using cross_val_score and evaluate it using validation sets from the training set
def performCrossValidation(X_train, y_train, n_splits, model):
    kfold = KFold(n_splits=n_splits)
    results = cross_val_score(model, X_train, y_train, cv=kfold)

    MSE = results.mean().item()
    STD = results.std().item()
    rootMSE = abs(results.mean().item())**0.5

    return MSE,STD


# print the results
def printTrainingResults(X_train, epochs, batch_size, n_splits, baseline_model, MSE):
    # print('\n\n')
    baseline_model().summary() # enable to print a summary of the NN model

    print('\nParameters:')
    print('\tepochs:\t\t', epochs)
    print('\tbatch_size:\t', batch_size)
    # print('\tn_splits:\t', n_splits)
    print('\tinput shape:\t', X_train.shape)

    print('\nMSE becomes: {:.4f}'.format(abs(MSE)))
    print('Root MSE becomes: {:.4f}'.format(abs(MSE)**0.5))
```

E.1.2. Processing Solar Data

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: In this script, the data of solar panels next to the EEMCS building is processed.
```

```
'''


import numpy as np
import datetime as dt
import math


# retrieve the data
data = np.genfromtxt('forecasting/generationData/2019Bikechargerdata_
voltageandcurrentWithoutQuotes.csv', dtype=None, encoding=None,
 ↪  delimiter=',', skip_header=0, comments='#')
# note on the data: date, id, charger on/off, charge state, pv voltage, pv
 ↪  current, ...


# return the hour of the year, given a certain date
def hourOfYear(date):
    beginningOfYear = dt.datetime(date.year, 1, 1, tzinfo=date.tzinfo)
    return int((date - beginningOfYear).total_seconds() // 3600)


# extract the date from the data
time = []
for t in data:
    time.append(dt.datetime(year=2019, month=int(t['f0'][3:5]),
     ↪  day=int(t['f0'][0:2]), hour=int(t['f0'][9:11])))
time = np.array(time)

# convert the date to the hour of the year
timeAsHour = []
for t in time: timeAsHour.append(hourOfYear(t))
timeAsHour = np.array(timeAsHour)

# create a numpy array from the hour, volt, and current data
data =
 ↪  np.concatenate((data['f4'][:,np.newaxis],data['f5'][:,np.newaxis]),
 ↪  axis=1, dtype=None)

# calculate power from the voltage and current
irregularPowerData = []
for t in data: irregularPowerData.append(t[0]*t[1])
irregularPowerData = np.array(irregularPowerData)

# initialize
hourlyPowerData = np.empty(np.max(timeAsHour)+1)
hourlyPowerData[:] = np.NaN
i = 0
w = np.ones(np.max(timeAsHour)+1)

# mean for every hour
for t in timeAsHour:

    if math.isnan(hourlyPowerData[t]):
        hourlyPowerData[t] = irregularPowerData[i]
```

```
    else:
        hourlyPowerData[t] =
         ↪  (hourlyPowerData[t]*w[t]+irregularPowerData[i])/(w[t]+1) #
         ↪  weight is equal for every variable
        w[t] += 1 # this requires the weight factor to be increased
         ↪  throughout iterations

    i += 1

# replace all nans with the value of 24 hours earlier
replacementValue = np.nanmean(hourlyPowerData)      # calculate the mean
nanCount = 0
for i in range(len(hourlyPowerData)):   # loop through all elements
    if math.isnan(hourlyPowerData[i]):  # check for nans
        hourlyPowerData[i] = hourlyPowerData[i-24]
        # hourlyPowerData[i] = replacementValue       # replace nans with
         ↪  the mean
        nanCount += 1


# save the acquired array
np.save('hourlyPowerData', hourlyPowerData)
```

E.1.3. Hyperparameter Tuning

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: With this script, the hyperparameters of the ANN for a
 ↪  certain data set can tuned.
'''


import datetime as dt
import matplotlib.pyplot as plt
import numpy as np
import functions as fs
import sklearn as sk
import sklearn.impute
import math
from keras.layers import Dense
from keras.models import Sequential
from keras.wrappers.scikit_learn import KerasRegressor
from tensorflow import keras
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, cross_val_score,
 ↪  train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from playsound import playsound


# suppress depreciation warnings
```

```
import tensorflow.python.util.deprecation as deprecation
deprecation._PRINT_DEPRECATION_WARNINGS = False


# retrieve data of one of these three
# X, y = fs.retrieveSolarData()
# X, y = fs.retrieveWindData()
# X, y = fs.retrieveDemandData()


# splitting data in test and training sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3,
    random_state=42)
fs.printSets(X_train, X_test, y_train, y_test) # enable to print set
    shapes

# checking and handling missing values
imp = sk.impute.SimpleImputer(missing_values=np.nan, strategy='median')
X = imp.fit_transform(X)
X_train = imp.fit_transform(X_train)
X_test = imp.fit_transform(X_test)


# defining certain variables
epochs = 1000
batch_size = 500
verbose = 2          # 0 to show nothing; 1 or 2 to show the progress
n_splits = 2

# from https://machinelearningmastery.com/regression-tutorial-keras-deep-
    learning-library-python/

# define solar base model
def solarBaselineModel(neurons1=1, neurons2=1, neurons3=1):
    # create model
    model = Sequential()

    model.add(Dense(neurons1, input_dim=22, kernel_initializer='normal',
        activation='relu'))
    model.add(Dense(neurons2, kernel_initializer='normal',
        activation='relu'))
    model.add(Dense(neurons3, kernel_initializer='normal',
        activation='relu'))

    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# define generation base model
def windBaselineModel(neurons1=1, neurons2=1, neurons3=1):
    # create model
    model = Sequential()
    model.add(Dense(neurons1, input_dim=22, kernel_initializer='normal',
        activation='relu'))
```

```
75      model.add(Dense(neurons2, kernel_initializer='normal',
          ↪ activation='relu'))
76      model.add(Dense(neurons3, kernel_initializer='normal',
          ↪ activation='relu'))
77      model.add(Dense(1, kernel_initializer='normal'))
78      # Compile model
79      model.compile(loss='mean_squared_error', optimizer='adam')
80      return model
81
82  # define demand base model
83  def demandBaselineModel(neurons1=1, neurons2= 1, neurons3=1, neurons4=1,
      ↪ neurons5=1):
84      # create model
85      model = Sequential()
86      model.add(Dense(neurons1, input_dim=7, kernel_initializer='normal',
          ↪ activation='relu'))
87      model.add(Dense(neurons2, kernel_initializer='normal',
          ↪ activation='relu'))
88      model.add(Dense(neurons3, kernel_initializer='normal',
          ↪ activation='relu'))
89      model.add(Dense(neurons4, kernel_initializer='normal',
          ↪ activation='relu'))
90      model.add(Dense(neurons5, kernel_initializer='normal',
          ↪ activation='relu'))
91      model.add(Dense(1, kernel_initializer='normal'))
92      # Compile model
93      model.compile(loss='mean_squared_error', optimizer='adam')
94      return model
95
96  # create the model of one of these three
97  # model = KerasRegressor(build_fn=solarBaselineModel, epochs=epochs,
      ↪ batch_size=batch_size, verbose=verbose)
98  # model = KerasRegressor(build_fn=windBaselineModel, epochs=epochs,
      ↪ batch_size=batch_size, verbose=verbose)
99  # model = KerasRegressor(build_fn=demandBaselineModel, epochs=epochs,
      ↪ batch_size=batch_size, verbose=verbose)
100
101
102 # define the grid search parameters
103 batch_size = [200, 500, 1000]
104 epochs = [10, 25, 50, 100, 200, 500, 1000, 2000]
105
106 neurons1 = [150, 300]
107 neurons2 = [100, 200]
108 neurons3 = [50, 100]
109 neurons4 = [25, 50]
110 neurons5 = [10, 20]
111
112 param_grid = dict(batch_size=batch_size, epochs=epochs,
      ↪ neurons1=neurons1, neurons2=neurons2, neurons3=neurons3,
      ↪ neurons4=neurons4, neurons5=neurons5)
113
114
115 # evaluate all combinations using 3-fold cross validation
116 # from https://machinelearningmastery.com/grid-search-hyperparameters-
      ↪ deep-learning-models-python-keras/
```

```
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

E.1.4. Creating Predictions

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: In this script, ANN models are trained with a training set and evaluated on a test set.
'''


import datetime as dt
import matplotlib.pyplot as plt
import numpy as np
import functions as fs
import sklearn as sk
import sklearn.impute
import math
from keras.layers import Dense
from keras.models import Sequential
from keras.wrappers.scikit_learn import KerasRegressor
from tensorflow import keras
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV


# suppress depreciation warnings
import tensorflow.python.util.deprecation as deprecation
deprecation._PRINT_DEPRECATION_WARNINGS = False


# retrieve all input and output data
X_s, y_s = fs.retrieveSolarData()
X_w, y_w = fs.retrieveWindData()
X_d, y_d = fs.retrieveDemandData()

# check for nans in the output data
if sum(np.isnan(y_s))+sum(np.isnan(y_w))+sum(np.isnan(y_d)) != 0:
    print('nans found')
```

```
37
38  # scaling
39  meanSunPower = 12.27*10**6/365/24 # average sun generation per hour
40  y_s = meanSunPower/np.mean(y_s)*y_s # [Wh/hour] # scale the data
41
42  meanWindPower = 10.87*10**6/365/24 # [W] # average wind generation per
     ↪ hour
43  y_w = meanWindPower/np.mean(y_w)*y_w # [Wh/hour] # scale the data
44
45
46  # splitting data in test and training sets
47  X_train_s, X_test_s, y_train_s, y_test_s = train_test_split(X_s, y_s,
     ↪ test_size=.3, random_state=42)
48  X_train_w, X_test_w, y_train_w, y_test_w = train_test_split(X_w, y_w,
     ↪ test_size=.3, random_state=42)
49  X_train_d, X_test_d, y_train_d, y_test_d = train_test_split(X_d, y_d,
     ↪ test_size=.3, random_state=42)
50
51  # checking and handling missing values
52  imp = sk.impute.SimpleImputer(missing_values=np.nan, strategy='median')
53
54  X_s = imp.fit_transform(X_s)
55  X_train_s = imp.fit_transform(X_train_s)
56  X_test_s = imp.fit_transform(X_test_s)
57
58  X_w = imp.fit_transform(X_w)
59  X_train_w = imp.fit_transform(X_train_w)
60  X_test_w = imp.fit_transform(X_test_w)
61
62  X_d = imp.fit_transform(X_d)
63  X_train_d = imp.fit_transform(X_train_d)
64  X_test_d = imp.fit_transform(X_test_d)
65
66  # defining certain variables
67  verbose = 0         # 0 to show nothing; 1 (much) or 2 (little) to show
     ↪ the progress
68  n_splits = 5
69
70  # from https://machinelearningmastery.com/regression-tutorial-keras-deep-
     ↪ learning-library-python/
71
72  # define solar base model
73  def solarBaselineModel():
74      # create model
75      model = Sequential()
76      model.add(Dense(20, input_dim=22, kernel_initializer='normal',
         ↪ activation='relu'))
77      model.add(Dense(5, kernel_initializer='normal', activation='relu'))
78      model.add(Dense(30, kernel_initializer='normal', activation='relu'))
79      model.add(Dense(1, kernel_initializer='normal'))
80      # Compile model
81      model.compile(loss='mean_squared_error', optimizer='adam')
82      return model
83
84  # define generation base model
```

```
85  def windBaselineModel():
86      # create model
87      model = Sequential()
88      model.add(Dense(10, input_dim=22, kernel_initializer='normal',
          ↪ activation='relu'))
89      model.add(Dense(15, kernel_initializer='normal', activation='relu'))
90      model.add(Dense(25, kernel_initializer='normal', activation='relu'))
91      model.add(Dense(1, kernel_initializer='normal'))
92      # Compile model
93      model.compile(loss='mean_squared_error', optimizer='adam')
94      return model
95
96  # define demand base model
97  def demandBaselineModel():
98      # create model
99      model = Sequential()
100     model.add(Dense(150, input_dim=7, kernel_initializer='normal',
          ↪ activation='relu'))
101     model.add(Dense(200, kernel_initializer='normal', activation='relu'))
102     model.add(Dense(50, kernel_initializer='normal', activation='relu'))
103     model.add(Dense(25, kernel_initializer='normal', activation='relu'))
104     model.add(Dense(20, kernel_initializer='normal', activation='relu'))
105     model.add(Dense(1, kernel_initializer='normal'))
106     # Compile model
107     model.compile(loss='mean_squared_error', optimizer='adam')
108     return model
109
110 # solar
111 batch_size_s = 200
112 epochs_s = 500
113
114 # wind
115 batch_size_w = 200
116 epochs_w = 200
117
118 # demand
119 batch_size_d = 500
120 epochs_d = 1000
121
122
123 # combine base models, batch sizes, and epochs
124 solarModel = KerasRegressor(build_fn=solarBaselineModel, epochs=epochs_s,
      ↪ batch_size=batch_size_s, verbose=verbose)
125 windModel = KerasRegressor(build_fn=windBaselineModel, epochs=epochs_w,
      ↪ batch_size=batch_size_w, verbose=verbose)
126 demandModel = KerasRegressor(build_fn=demandBaselineModel,
      ↪ epochs=epochs_d, batch_size=batch_size_d, verbose=verbose)
127
128
129 # make predictions on the test set
130 MSE_s = fs.trainWithoutCurve(X_train_s, y_train_s, X_test_s, y_test_s,
      ↪ solarModel)
131 y_pred_s = solarModel.predict(X_s)
132
133 MSE_w = fs.trainWithoutCurve(X_train_w, y_train_w, X_test_w, y_test_w,
      ↪ windModel)
```

```
y_pred_w = windModel.predict(X_w)

MSE_d = fs.trainWithoutCurve(X_train_d, y_train_d, X_test_d, y_test_d,
    demandModel)
y_pred_d = demandModel.predict(X_d)


# print all results
print('\n############################## SOLAR
    ##############################\n')
fs.printTrainingResults(X_s, epochs_s, batch_size_s, n_splits,
    solarBaselineModel, MSE_s)
print('Mean Error as fraction of Maximum:', abs(MSE_s)**0.5/np.max(y_s))

print('\n\n############################## WIND
    ##############################\n')
fs.printTrainingResults(X_w, epochs_w, batch_size_w, n_splits,
    windBaselineModel, MSE_w)
print('Mean Error as fraction of Maximum:', abs(MSE_w)**0.5/np.max(y_w))

print('\n\n############################## DEMAND
    ##############################\n')
fs.printTrainingResults(X_d, epochs_d, batch_size_d, n_splits,
    demandBaselineModel, MSE_d)
print('Mean Error as fraction of Maximum:', abs(MSE_d)**0.5/np.max(y_d))
```

E.1.5. Creating Predictions Using the Previous Hour

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: In this script, ANN models are trained with a training set
    and evaluated on a test set. The training set consists of the data of
    the previous hour: either the actual value or a prediction.
'''


import datetime as dt
import matplotlib.pyplot as plt
import numpy as np
import functions as fs
import sklearn as sk
import sklearn.impute
import math
from keras.layers import Dense
from keras.models import Sequential
from keras.wrappers.scikit_learn import KerasRegressor
from tensorflow import keras
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, cross_val_score,
    train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
```

```
23
24
25  # suppress depreciation warnings
26  import tensorflow.python.util.deprecation as deprecation
27  deprecation._PRINT_DEPRECATION_WARNINGS = False
28
29
30  # retrieve all input and output data
31  X_s, y_s = fs.retrieveSolarData()
32  X_w, y_w = fs.retrieveWindData()
33  X_d, y_d = fs.retrieveDemandData()
34
35  # check for nans in the output data
36  if sum(np.isnan(y_s))+sum(np.isnan(y_w))+sum(np.isnan(y_d)) != 0:
     ↪  print('nans found')
37
38  # scaling
39  meanSunPower = 12.27*10**6/365/24 # average sun generation per hour
40  y_s = meanSunPower/np.mean(y_s)*y_s # [Wh/hour] # scale the data
41
42  meanWindPower = 10.87*10**6/365/24 # [W] # average wind generation per
     ↪  hour
43  y_w = meanWindPower/np.mean(y_w)*y_w # [Wh/hour] # scale the data
44
45
46  # adding data of the previous hour
47  dataForControl = np.load('savedData.npz')
48  predSolar =      dataForControl['predSolar']
49  predWind =       np.load('predWind.npy')
50  predDemand =     dataForControl['predDemand']
51
52  # # using the actual previous hour
53  # newFeature = np.delete(np.insert(y_s,0,y_s[0]),-1)[:,np.newaxis]
54  # X_s = np.append(X_s, newFeature, axis=1)
55  # newFeature = np.delete(np.insert(y_w,0,y_w[0]),-1)[:,np.newaxis]
56  # X_w = np.append(X_w, newFeature, axis=1)
57  # newFeature = np.delete(np.insert(y_d,0,y_d[0]),-1)[:,np.newaxis]
58  # X_d = np.append(X_d, newFeature, axis=1)
59
60  # # using the prediction of the previous hour
61  # newFeature =
     ↪  np.delete(np.insert(predSolar,0,predSolar[0]),-1)[:,np.newaxis]
62  # X_s = np.append(X_s, newFeature, axis=1)
63  # newFeature =
     ↪  np.delete(np.insert(predWind,0,predWind[0]),-1)[:,np.newaxis]
64  # X_w = np.append(X_w, newFeature, axis=1)
65  # newFeature =
     ↪  np.delete(np.insert(predDemand,0,predDemand[0]),-1)[:,np.newaxis]
66  # X_d = np.append(X_d, newFeature, axis=1)
67
68
69  # splitting data in test and training sets
70  X_train_s, X_test_s, y_train_s, y_test_s = train_test_split(X_s, y_s,
     ↪  test_size=.3, random_state=42)
71  X_train_w, X_test_w, y_train_w, y_test_w = train_test_split(X_w, y_w,
     ↪  test_size=.3, random_state=42)
```

```
X_train_d, X_test_d, y_train_d, y_test_d = train_test_split(X_d, y_d,
    test_size=.3, random_state=42)

# checking and handling missing values
imp = sk.impute.SimpleImputer(missing_values=np.nan, strategy='median')

X_s = imp.fit_transform(X_s)
X_train_s = imp.fit_transform(X_train_s)
X_test_s = imp.fit_transform(X_test_s)

X_w = imp.fit_transform(X_w)
X_train_w = imp.fit_transform(X_train_w)
X_test_w = imp.fit_transform(X_test_w)

X_d = imp.fit_transform(X_d)
X_train_d = imp.fit_transform(X_train_d)
X_test_d = imp.fit_transform(X_test_d)


# defining certain variables
verbose = 0          # 0 to show nothing; 1 (much) or 2 (little) to show
    the progress
n_splits = 5

# from https://machinelearningmastery.com/regression-tutorial-keras-deep-
    learning-library-python/

# define solar base model
def solarBaselineModel():
    # create model
    model = Sequential()
    model.add(Dense(20, input_dim=23, kernel_initializer='normal',
        activation='relu'))
    model.add(Dense(5, kernel_initializer='normal', activation='relu'))
    model.add(Dense(30, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# define generation base model
def windBaselineModel():
    # create model
    model = Sequential()
    model.add(Dense(10, input_dim=23, kernel_initializer='normal',
        activation='relu'))
    model.add(Dense(15, kernel_initializer='normal', activation='relu'))
    model.add(Dense(25, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# define demand base model
def demandBaselineModel():
```

```
122     # create model
123     model = Sequential()
124     model.add(Dense(150, input_dim=8, kernel_initializer='normal',
          ↪ activation='relu'))
125     model.add(Dense(200, kernel_initializer='normal', activation='relu'))
126     model.add(Dense(50, kernel_initializer='normal', activation='relu'))
127     model.add(Dense(25, kernel_initializer='normal', activation='relu'))
128     model.add(Dense(20, kernel_initializer='normal', activation='relu'))
129     model.add(Dense(1, kernel_initializer='normal'))
130     # Compile model
131     model.compile(loss='mean_squared_error', optimizer='adam')
132     return model
133 
134 # solar
135 batch_size_s = 200
136 epochs_s = 500
137 
138 # wind
139 batch_size_w = 200
140 epochs_w = 200
141 
142 # demand
143 batch_size_d = 500
144 epochs_d = 1000
145 
146 
147 # combine base models, batch sizes, and epochs
148 solarModel = KerasRegressor(build_fn=solarBaselineModel, epochs=epochs_s,
      ↪ batch_size=batch_size_s, verbose=verbose)
149 windModel = KerasRegressor(build_fn=windBaselineModel, epochs=epochs_w,
      ↪ batch_size=batch_size_w, verbose=verbose)
150 demandModel = KerasRegressor(build_fn=demandBaselineModel,
      ↪ epochs=epochs_d, batch_size=batch_size_d, verbose=verbose)
151 
152 
153 # make predictions on the test set
154 MSE_s = fs.trainWithoutCurve(X_train_s, y_train_s, X_test_s, y_test_s,
      ↪ solarModel)
155 y_pred_s = solarModel.predict(X_s)
156 
157 MSE_w = fs.trainWithoutCurve(X_train_w, y_train_w, X_test_w, y_test_w,
      ↪ windModel)
158 y_pred_w = windModel.predict(X_w)
159 
160 MSE_d = fs.trainWithoutCurve(X_train_d, y_train_d, X_test_d, y_test_d,
      ↪ demandModel)
161 y_pred_d = demandModel.predict(X_d)
162 
163 
164 # print all results
165 print('############################ WITH PREV HOUR
      ↪ ############################')
166 print('\n')
167 
168 print('\n############################## SOLAR
      ↪ ##############################\n')
```

```
169 fs.printTrainingResults(X_s, epochs_s, batch_size_s, n_splits,
      ↪ solarBaselineModel, MSE_s)
170 print('Mean Error as fraction of Maximum:', abs(MSE_s)**0.5/np.max(y_s))
171
172 print('\n\n############################## WIND
      ↪ ##############################\n')
173 fs.printTrainingResults(X_w, epochs_w, batch_size_w, n_splits,
      ↪ windBaselineModel, MSE_w)
174 print('Mean Error as fraction of Maximum:', abs(MSE_w)**0.5/np.max(y_w))
175
176 print('\n\n############################## DEMAND
      ↪ ##############################\n')
177 fs.printTrainingResults(X_d, epochs_d, batch_size_d, n_splits,
      ↪ demandBaselineModel, MSE_d)
178 print('Mean Error as fraction of Maximum:', abs(MSE_d)**0.5/np.max(y_d))
```

E.2. Control

E.2.1. Optimizer Initialization

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: In this script, the optimization for the mpc is initialized.
Parameters and variables are defined and a model is created.
'''
import os
# Importing libraries
from pyomo.environ import *
from pyomo.dae import *

'''
function: create and initialize the optimization for the mpc
Inputs:
time                    - list of length 169 representing every hour in the coming week
SoCIni                   - integer respresenting the SoC at t = 0
SoCDiff                  - List of length 169 with the predicted difference in time
setPoint                - Set of length 169 with the Setpoint for every hour
weight                   - Set of length 169 with the associated weight for deviation from the setpoint for every hour
dCost                    - Set of length 169 with the associated weight penalty for changing the control level
cCost                    - Set of length 169 with the associated weight for deviation of the control level from level zero
dMax                     - integer respresenting the maximum change in the control signal per hour
controlLevelIni          - integer respresenting current control level
output:
Mpc                      - the model for the optimizer of the model predictive controller
'''

def modelPredictiveControl(time, SoCIni, SoCDiff, setPoint, weight, dCost, cCost, dMax, controlLevelIni):

    #initializing the mpc model using pyomo
    mpc = ConcreteModel()

    # Define the Set
    mpc.time = Set(initialize= time) # Define the time steps from 0 to 168 hours

    # Define Parameters
    mpc.SoCDiff = Param(mpc.time, initialize=SoCDiff, mutable=True)
    mpc.setPoint = Param(mpc.time, initialize=setPoint, mutable=True)
    mpc.weight = Param(mpc.time, initialize=weight, mutable=True)
    mpc.dCost = Param(mpc.time, initialize=dCost, mutable=True )
    mpc.cCost = Param(mpc.time, initialize=cCost, mutable=True )
    mpc.SoCIni = Param(initialize=SoCIni, mutable = True)
    mpc.controlLevelIni = Param(initialize=controlLevelIni, mutable = True)
```

```
43      mpc.dMax = Param(initialize=dMax, mutable = True)
44
45      # Define Variables
46      mpc.controlLevel = Var(mpc.time, within = Integers, bounds = (0,4))
47      mpc.SoC = Var(mpc.time, within = Reals)
48      mpc.deltaSetPointPos = Var(mpc.time , within = NonNegativeReals)
49      mpc.deltaSetPointNeg = Var(mpc.time , within = NonNegativeReals)
50      mpc.controlLevelPos = Var(mpc.time , within = NonNegativeIntegers)
51      mpc.controlLevelNeg = Var(mpc.time , within = NonNegativeReals)
52
53      # Define Objective functions
54      mpc.obj = Objective(expr = sum(mpc.weight[t]*(mpc.deltaSetPointPos[t]
         ↪  + mpc.deltaSetPointNeg[t]) + mpc.cCost[t] * mpc.controlLevel[t] +
         ↪  mpc.dCost[t] * (mpc.controlLevelPos[t] + mpc.controlLevelNeg[t])
         ↪  for t in mpc.time), sense = minimize )
55
56      # Define Constraints
57      # constraint calculating absolute value for change in Control level
58      def controlLevelNegcnstr(mpc, t):
59          if t == 0:
60              return Constraint.Skip
61          else:
62              constr = mpc.controlLevel[t]-mpc.controlLevel[t-1]
63              return constr == mpc.controlLevelPos[t] -
                 ↪  mpc.controlLevelNeg[t]
64      mpc.controlLevelNegcnstr = Constraint( mpc.time, rule=
         ↪  controlLevelNegcnstr)
65
66      # constraint calculating absolute value for change in SoC
67      def deltaSetPointcnstr(mpc, t):
68          constr = mpc.SoC[t]-mpc.setPoint[t]
69          return constr == mpc.deltaSetPointPos[t] - mpc.deltaSetPointNeg[t]
70      mpc.deltaSetPointcnstr = Constraint( mpc.time, rule=
         ↪  deltaSetPointcnstr)
71
72      # constraint constraints the controller level from changing more than
         ↪  one level up in an hour
73      def constrDMax(mpc, t):
74          if t == 0:
75              return Constraint.Skip
76          else:
77              Constrnt = (mpc.controlLevel[t]-mpc.controlLevel[t-1] <=
                 ↪  mpc.dMax)
78              return Constrnt
79      mpc.constrDMax = Constraint( mpc.time, rule= constrDMax )
80
81      # constraint constraints the controller level from changing more than
         ↪  one level down in an hour
82      def constrDMin(mpc, t):
83          if t == 0:
84              return Constraint.Skip
85          else:
86              Constrnt = (mpc.controlLevel[t]-mpc.controlLevel[t-1] >=
                 ↪  -1*mpc.dMax)
87              return Constrnt
```

```
    mpc.constrDMin = Constraint( mpc.time, rule= constrDMin )

    # constraint initializes the first value of the Control level
    mpc.controlSoCInicnstr = Constraint(expr =  mpc.controlLevel[0] ==
     ↪  mpc.controlLevelIni)

    # constraint initializes the first value of the SoC
    def constrSoCini(mpc,t):
        if t == 0:
            return mpc.SoC[t] == mpc.SoCIni
        else:
            return Constraint.Skip
    mpc.constrSoCini = Constraint( mpc.time, rule= constrSoCini)

    # constraint calculates the next value of the SoC
    def constrSoC(mpc,t):
        if t == 0:
            return Constraint.Skip
        else:
            Constrnt = (mpc.SoC[t] == mpc.SoC[t-1] +
             ↪  139.85*mpc.controlLevel[t-1]/722 + mpc.SoCDiff[t-1])
            return Constrnt
    mpc.constrSoC = Constraint( mpc.time, rule= constrSoC)

    # Return the model
    return mpc
```

E.2.2. One Cycle Optimization

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: In this script, the functionality of the controller for one
 ↪  week is programmed
It used different functions and a loop to control the system.
'''

import os
from numpy.core.function_base import linspace
from numpy.lib.function_base import append, diff
os.system('cls') # clears the command window
import datetime as dt; start_time = dt.datetime.now()
# display a "Run started" message
print('Run started at ', start_time.strftime("%X"), '\n')

# import libraries and python functions
import numpy as np
import matplotlib.pyplot as plt
from pyomo.environ import *
from pyomo.dae import *
from measurements import readMeasurement
from measurements import readPredictions
from measurements import determineControlSoC
from optimizationSetup import modelPredictiveControl
```

```
24
25  # load predictions
26  dataForControl = np.load('dataForControl.npz')
27  predSun = dataForControl['predSolar']
28  predWind = dataForControl['predWind']
29  predDemand = dataForControl['predDemand']
30
31  # load actual data
32  Sun = dataForControl['realSolar']
33  Wind = dataForControl['realWind']
34  Demand = dataForControl['realDemand']
35
36  # Setup input data for the initialization of the model
37  # Initialize the time array, this represents all hours in the upcoming
     ↪ week.
38  # This will initialize the Set for the Pyomo model
39  time = []
40  for i in range(169):
41      time.append(i)
42
43  # Initialize all other values and create key parameter pairs
44  # This will initialize the indexed Parameters for the Pyomo model
45
46  # Initialize the difference in state of charge due to the predicted demand
     ↪ and generation
47  SoCDiff_ini = readPredictions(0,len(time),predSun,predWind,predDemand) #
     ↪ one week
48  SoCDiff = {time[i]: SoCDiff_ini[i] for i in range(len(time))} # Make it a
     ↪ Dictionary
49
50  # Initialize the setpoint the controller tries to reach, this will
     ↪ determine the objective in the Pyomo model
51  setPoint_ini = []
52  for i in range(len(time)):
53      setPoint_ini.append(60.0) # Is constant, but can be variable
54  setPoint = {time[i]: setPoint_ini[i] for i in range(len(time))} # Make it
     ↪ a Dictionary
55
56  # Initialize the weight for the deviation of the SoC from the setpoint, is
     ↪ used by the objective
57  weight_ini = []
58  for i in range(len(time)):
59      weight_ini.append(float(len(time)-i)) # The weight decreases linearly
         ↪ over time
60  weight = {time[i]: weight_ini[i] for i in range(len(time))}# Make it a
     ↪ Dictionary
61
62  # Initialize the weight for the change in control signal, is used by the
     ↪ objective
63  dCost_ini = []
64  for i in range(len(time)):
65      dCost_ini.append(5*weight_ini[i])# The weight decreases linearly over
         ↪ time
66  dCost = {time[i]: dCost_ini[i] for i in range(len(time))}# Make it a
     ↪ Dictionary
```

```
67
68  # Initialize the weight for the deviation of the control signal from zero,
     ↪ is used by the objective
69  cCost_ini = []
70  for i in range(len(time)):
71      cCost_ini.append(2*weight_ini[i])# The weight decreases linearly over
         ↪ time
72  cCost = {time[i]: cCost_ini[i] for i in range(len(time))}# Make it a
     ↪ Dictionary
73
74
75  # Initialize all other non indexed values
76  # This will initialize the non indexed Parameters for the Pyomo model
77  dMax = 1 # The maximum chance allowed in the control signal per hour
78  SoCIni = 55.0 # The measured SoC at t=0
79  controlLevelIni = 0 # The currently employed control signal
80
81  # Create controller model, parameters are set to mutable such that the
     ↪ model can be resolved with different parameters
82  mpc = modelPredictiveControl(time, SoCIni, SoCDiff, setPoint, weight,
     ↪ dCost, cCost, dMax, controlLevelIni)
83
84  # Optional line usefull for debugging the controller
85  #mpc.pprint()
86
87  # Select a solver for solving the model
88  solver = SolverFactory('glpk') # glpk is a linear solver that can handle
     ↪ discrete values
89
90  # Solve the model
91  solver.solve(mpc, tee = False) # solving the model, tee = true provides
     ↪ extra information on the solution of the solver
92
93  # Read values from the model
94  tempSoC = [mpc.SoC[i].value for i in mpc.time]
95
96  # Calculate what happens without the controller
97  SoCRaw = [SoCIni]
98  for i in range(len(time)-1):
99      SoCRaw.append(SoCRaw[i] + SoCDiff_ini[i])
100
101 # plot the SoC over time
102 plt.subplot(2,1,1)
103 plt.plot(time,tempSoC, c = '#0C7CBA', ls = '-') # plot the SOC
104 plt.plot(time,setPoint_ini, c = 'black', ls = '--') # plot the set point
105 plt.plot(time,SoCRaw, c = 'black', ls = '-') # plot the SoC without the
     ↪ controller
106 plt.xlabel("Time [hours]")
107 plt.ylabel("State of Charge [%]")
108 plt.title("State of charge of the battery over one time horizon")
109 plt.subplots_adjust(wspace=0.05, hspace=.5)
110 # plot the optimized control level over the time horizon
111 tempControlLevel = (mpc.controlLevel[i].value for i in time)
112 tempControlLevel = list(tempControlLevel)
113 plt.subplot(2,1,2)
```

```
plt.plot(time,tempControlLevel, marker ='o', c = '#0C7CBA', ls ='')
plt.xlabel("Time [hours]")
plt.ylabel("control level")
plt.title("Control level over one time horizon")
# plot the difference when using the controller or not using the controller
#SoCDiffRaw = []
#for i in range(len(time)):
#    SoCDiffRaw.append(tempSoC[i] - SoCRaw[i])
#plt.subplot(3,1,3)
#plt.plot(time,SoCDiffRaw, c = '#0C7CBA', ls ='-')
#plt.xlabel("Time [hours]")
#plt.title("Difference in SoC due to the controller action")

# print the runtime
print('\nRuntime was', (dt.datetime.now() - start_time).total_seconds(),
 ↪  'seconds')

#show plot
plt.show()
```

E.2.3. Measurements and Controller Effect

```
'''
Authors: Aart Rozendaal and Pieter Van Santvliet
Description: In this script, three functions are writen regarding reading
 ↪  prediction data, reading the current measurement and calculating the
 ↪  influence of the controller using the random library and the current
 ↪  hour.
'''
# add libraries
import numpy as np
import random as rnd

'''
Function: calculates the predicted change in SoC due to the predictions
inputs: index - current hour
length - length of the time horizon
sun - predictions of solar generation over the time horizon
wind - predictions of wind generation over the time horizon
demand - predictions of demand over the time horizon
output: SoCChange - predicted change in SoC of the battery due to the
 ↪  predicted generation and demand
'''
def readPredictions(index,length,sun,wind,demand):
    SoCChange = [] # initiate list
    length = length +index
    for i in range(index,length):
        SoCChange.append((sun[i] + wind[i] - demand[i])/772.0) # read the
         ↪  predictions for every hour and scale Wh to SoC
    return SoCChange # return Change in SoC

'''
Function: Reads the current value of the SoC
```

```
inputs: index - current hour
sun - measured solar generation at the current hour
wind - measured wind generation at the current hour
demand - measured demand at the current hour
output: SoCChange - actual change in SoC of the battery due to the measured generation and demand
'''
def readMeasurement(index,sun,wind,demand):
    SoCChange = (sun[index] + wind[index] - demand[index])/772.0 # read the measured generation and demand and scale Wh to SoC
    return SoCChange # return Change in SoC

'''
Function: calculates the effect of the controller at the current hour using the control level
inputs: index - current hour
controlLevel - current control level
output: SoCChange - actual change in SoC of the battery due to the control level
'''
def determineControlSoC(index, controlLevel): # TODO switch the first two
    ControlSoC = 0 # intialize the change in SoC as zero
    hour = index%24 # calculate which hour of the day it is
    if controlLevel > 0: # control level 1 or higher
        if hour > 5 and hour < 21:
            for i in range(12): # calculate at random who is not home
                temp = rnd.randint(1,1000)
            if temp < 408:
                ControlSoC += 70 # when not home boiler is turned off
            else:
                ControlSoC += 0
        if controlLevel > 1: # control level 2 or higher
            if hour == 20 or hour == 21 or hour == 4 or hour == 5:
                ControlSoC += 60 # use lighting at 50% during these hours
            elif hour > 21 or hour < 4:
                ControlSoC += 120 # dont use lighting during these hours
            else:
                ControlSoC += 0 # no savings during the day
            if controlLevel > 2:# control level 3 or higher
                if hour > 5 and hour < 22:
                    for i in range(12): # calculate at random who is using the wachingmachine
                        temp = rnd.randint(1,1000)
                        if temp < 400:
                            ControlSoC += 10.5 # when using it it can only use eco mode
                else:
                    ControlSoC += 0
                if controlLevel >3:# control level 4
                    if hour > 17 and hour < 22:
                        for i in range(12):# calculate at random who is using the stove
                            temp = rnd.randint(1,1000)
                            if temp < 250:
                                ControlSoC += 281.25# when using it it can only use one on high and one on medium
```

```
75
76      ControlSoC = ControlSoC/722 # convert Wh to SoC
77
78      return ControlSoC # return the change in SoC
```

E.2.4. Simulation

```
1   '''
2   Authors: Aart Rozendaal and Pieter Van Santvliet
3   Description: In this script, the functionality of the controller is
     ↪ programmed
4   It used different functions and a loop to control the system.
5   '''
6
7   import os
8   from numpy.core.function_base import linspace
9   from numpy.lib.function_base import append, diff
10  os.system('cls') # clears the command window
11  import datetime as dt; start_time = dt.datetime.now()
12  # display a "Run started" message
13  print('Run started at ', start_time.strftime("%X"), '\n')
14
15  # import libraries and python functions
16  import numpy as np
17  import matplotlib.pyplot as plt
18  from pyomo.environ import *
19  from pyomo.dae import *
20  from measurements import readMeasurement
21  from measurements import readPredictions
22  from measurements import determineControlSoC
23  from optimizationSetup import modelPredictiveControl
24
25  # load predictions
26  dataForControl = np.load('dataForControl.npz')
27  predSun = dataForControl['predSolar']
28  predWind = dataForControl['predWind']
29  predDemand = dataForControl['predDemand']
30
31  # load actual data
32  sun = dataForControl['realSolar']
33  wind = dataForControl['realWind']
34  demand = dataForControl['realDemand']
35
36  # Setup input data for the initialization of the model
37  # Initialize the time array, this represents all hours in the upcoming
     ↪ week.
38  # This will initialize the Set for the Pyomo model
39  time = []
40  for i in range(169):
41      time.append(i)
42
43  # Initialize all other values and create key parameter pairs
44  # This will initialize the indexed Parameters for the Pyomo model
```

```
45
46  # Initialize the difference in state of charge due to the predicted demand
     ↪ and generation
47  SoCDiff_ini = readPredictions(0,len(time),predSun,predWind,predDemand) #
     ↪ one week
48  SoCDiff = {time[i]: SoCDiff_ini[i] for i in range(len(time))} # Make it a
     ↪ Dictionary
49
50  # Initialize the setpoint the controller tries to reach, this will
     ↪ determine the objective in the Pyomo model
51  setPoint_ini = []
52  for i in range(len(time)):
53      setPoint_ini.append(60.0) # Is constant, but can be variable
54  setPoint = {time[i]: setPoint_ini[i] for i in range(len(time))} # Make it
     ↪ a Dictionary
55
56  # Initialize the weight for the deviation of the SoC from the setpoint, is
     ↪ used by the objective
57  weight_ini = []
58  for i in range(len(time)):
59      weight_ini.append(1/(1+i)) # The weight decreases linearly over time
60  weight = {time[i]: weight_ini[i] for i in range(len(time))}# Make it a
     ↪ Dictionary
61
62  # Initialize the weight for the change in control signal, is used by the
     ↪ objective
63  dCost_ini = []
64  for i in range(len(time)):
65      dCost_ini.append(0.1*weight_ini[i])# The weight decreases linearly
         ↪ over time
66  dCost = {time[i]: dCost_ini[i] for i in range(len(time))}# Make it a
     ↪ Dictionary
67
68  # Initialize the weight for the deviation of the control signal from zero,
     ↪ is used by the objective
69  cCost_ini = []
70  for i in range(len(time)):
71      cCost_ini.append(0.1*weight_ini[i])# The weight decreases linearly
         ↪ over time
72  cCost = {time[i]: cCost_ini[i] for i in range(len(time))}# Make it a
     ↪ Dictionary
73
74
75  # Initialize all other non indexed values
76  # This will initialize the non indexed Parameters for the Pyomo model
77  dMax = 1 # The maximum chance allowed in the control signal per hour
78  SoCIni = 55.0 # The measured SoC at t=0
79  controlLevelIni = 0 # The currently employed control signal
80
81  # Create controller model, parameters are set to mutable such that the
     ↪ model can be resolved with different parameters
82  mpc = modelPredictiveControl(time, SoCIni, SoCDiff, setPoint, weight,
     ↪ dCost, cCost, dMax, controlLevelIni)
83
84  # Optional line usefull for debugging the controller
```

```
#mpc.pprint()

# Select a solver for solving the model
solver = SolverFactory('glpk') # glpk is a linear solver that can handle discrete values

# Solve the model
solver.solve(mpc, tee = False) # solving the model, tee = true provides extra information on the solution of the solver

# prepare lists for plotting
# general
timePlot = [0]
setPointPlot = [60.0]
# With controller
SoCPlot = [SoCIni]
controlLevelPlot = [controlLevelIni]
gridPowerGivePlot = [0]
gridPowerTakePlot = [0]
# without controller
SoCRawPlot = [SoCIni]
gridPowerGiveRawPlot = [0]
gridPowerTakeRawPlot = [0]

#loop
# optimize for the coming week every hour
for t in range(1,8760-(len(time)+1)):
    #update control level
    controlLevelIni = mpc.controlLevel[1].value

    # Retrieve the measurements when using the controller
    prevControl = determineControlSoC(t-1, mpc.controlLevel[0].value)
    if (SoCPlot[-1] + readMeasurement(t,sun,wind,demand) + prevControl > 100):
        mpc.SoCIni.value= SoCIni = 100
        gridPowerGive = abs(SoCPlot[-1] + readMeasurement(t,sun,wind,demand) + prevControl - 100)
        gridPowerTake = 0
    elif (SoCPlot[-1] + readMeasurement(t,sun,wind,demand) + prevControl < 10):
        mpc.SoCIni.value= SoCIni = 10
        gridPowerGive = 0
        gridPowerTake = abs(10 - (SoCPlot[-1] + readMeasurement(t,sun,wind,demand) + prevControl))
    else:
        mpc.SoCIni.value= SoCIni = SoCPlot[-1] + readMeasurement(t,sun,wind,demand) + prevControl
        gridPowerGive = 0
        gridPowerTake = 0

    # Retrieve the measurements without using the controller
    if SoCRawPlot[-1] + readMeasurement(t,sun,wind,demand) >100:
        SoCRaw = 100
        gridPowerGiveRaw = abs(SoCPlot[-1] + readMeasurement(t,sun,wind,demand) - 100)
```

```
132             gridPowerTakeRaw = 0
133         elif (SoCPlot[-1] + readMeasurement(t,sun,wind,demand)< 10):
134             SoCRaw = 10
135             gridPowerGiveRaw = 0
136             gridPowerTakeRaw = abs(10 - (SoCPlot[-1] +
                ↪  readMeasurement(t,sun,wind,demand) ))
137         else:
138             SoCRaw = SoCPlot[-1] + readMeasurement(t,sun,wind,demand)
139             gridPowerGiveRaw = 0
140             gridPowerTakeRaw = 0
141
142         # Make predictions using the ANN for demand and generation and prep
                ↪  for the model
143         SoCDiff_ini = readPredictions(t,len(time),predSun,predWind,predDemand)
144         mpc.SoCDiff[i].value = [SoCDiff_ini[i] for i in range(len(time))]
145         #SoCDiff = {time[i]: SoCDiff_ini[i] for i in range(len(time))}
146         mpc.controlLevelIni.value = controlLevelIni
147         # Solve the pyomo optimizer model
148         # mpc = modelPredictiveControl(time, SoCIni, SoCDiff, setPoint,
                ↪  weight, dCost, cCost, dMax, controlLevelIni)
149         solver.solve(mpc, tee = False)
150
151         # update plot values for both with and without controller
152         timePlot.append(t)
153         SoCPlot.append(SoCIni)
154         setPointPlot.append(60.0)
155         controlLevelPlot.append(controlLevelIni)
156         gridPowerGivePlot.append(gridPowerGive)
157         gridPowerTakePlot.append(gridPowerTake)
158         SoCRawPlot.append(SoCRaw)
159         gridPowerGiveRawPlot.append(gridPowerGiveRaw)
160         gridPowerTakeRawPlot.append(gridPowerTakeRaw)
161
162         # print confirmation every 25 cycles
163         if t%25 == 0:
164             print('cycle: ', t , ' is done')
165
166         # loop back and repeat for the next hours
167
168 #plot relevant data for both with and without controller to compare the
      ↪  effect of the controller
169 print('\n------ With controller ------\n')
170 print('power delivered to grid:{:.2f}Wh'.format(sum(gridPowerGivePlot)))
171 k=0
172 for i in gridPowerGivePlot:
173     if i>0:
174         k += 1
175 print('Hours of power delivered to grid: {} hours, which is {:.2f}
      ↪  %'.format(k,k/len(gridPowerGivePlot)*100))
176 print('power taken from grid:{:.2f}Wh'.format(sum(gridPowerTakePlot)))
177 k=0
178 for i in gridPowerTakePlot:
179     if i>0:
180         k += 1
181 print('Hours of power taken from grid: {} hours, which is {:.2f}
      ↪  %'.format(k,k/len(gridPowerGivePlot)*100))
```

```
k=0
for i in controlLevelPlot:
    if i>0:
        k += 1
print('Hours the controller is active: {} hours, which is {:.2f}
 ↪  %'.format(k,k/len(controlLevelPlot)*100))

print('\n------ Without controller ------\n')
print('power delivered to
 ↪  grid:{:.2f}Wh'.format(sum(gridPowerGiveRawPlot)))
k=0
for i in gridPowerGiveRawPlot:
    if i>0:
        k += 1
print('Hours of power delivered to grid: {} hours, which is {:.2f}
 ↪  %'.format(k,k/len(gridPowerGiveRawPlot)*100))
print('power taken from grid:{:.2f}Wh'.format(sum(gridPowerTakeRawPlot)))
k=0
for i in gridPowerTakeRawPlot:
    if i>0:
        k += 1
print('Hours of power taken from grid: {} hours, which is {:.2f}
 ↪  %'.format(k,k/len(gridPowerGiveRawPlot)*100))

# Plot SoC over time
plt.subplot(2,1,1)
plt.plot(timePlot,SoCPlot, c = '#0C7CBA', ls = '-') # plot the SOC
plt.plot(timePlot,setPointPlot, c = 'black', ls = '--') # plot the set
 ↪  point
plt.plot(timePlot,SoCRawPlot, c = 'black', ls = '-') # plot the SoC
 ↪  without the controller
plt.xlabel("Time [hours]")
plt.ylabel("State of Charge [%]")
plt.title("State of charge of the battery over one time horizon")
plt.subplots_adjust(wspace=0.05, hspace=.5)
# plot the optimized control level over the time
plt.subplot(2,1,2)
plt.plot(timePlot,controlLevelPlot, marker ='o', c = '#0C7CBA', ls ='')
plt.xlabel("Time [hours]")
plt.ylabel("control level")
plt.title("Control level over one time horizon")

# print the runtime
print('\nRuntime was', (dt.datetime.now() - start_time).total_seconds(),
 ↪  'seconds')

# show plot
plt.show()
```

Acronyms

AC Alternating Current.

ANN Artificial Neural Network.

API Application Programming Interface.

CF Capacity Factor.

CNS Control & Software.

DC Direct Current.

DCG DC Grid.

EU European Union.

FSM Finite State Machine.

GLPK GNU Linear Programming Kit.

HAWT Horizontal Axis Wind Turbine.

IEA International Energy Agency.

KNMI Koninklijk Nederlands Meteorologisch Instituut.

LQR Linear Quadratic Regulator.

ML Machine Learning.

MPC Model Predictive Control.

MSE Mean Squared Error.

PID Proportional Integral Derivative.

PLC Power Line Communication.

PoC Proof of Concept.

PoR Program of Requirements.

PV Photo-Voltaic.

RES Renewable Energy Source.

SF Scaling Factor.

SoC State of Charge.

VAWT Vertical Axis Wind Turbine.

Bibliography

[1] Timothy Carlin. “Tiny homes: Improving carbon footprint and the American lifestyle on a large scale”. In: *Celebrating Scholarship & Creativity Day* (Apr. 24, 2014). URL: https://digitalcommons.csbsju.edu/elce_cscday/35.

[2] *The International Residential Code*. ICC. Mar. 20, 2015. URL: https://www.iccsafe.org/products-and-services/i-codes/2018-i-codes/irc/ (visited on 06/09/2021).

[3] Heather Shearer and Paul Burton. “Towards a Typology of Tiny Houses”. In: *Housing, Theory and Society* 36.3 (June 24, 2018). Publisher: Routledge, pp. 298–318. ISSN: 1403-6096. URL: https://www-tandfonline-com.tudelft.idm.oclc.org/doi/abs/10.1080/14036096.2018.1487879 (visited on 06/09/2021).

[4] Maria Jebbink. “Life in a shoebox: About people and their motivation to go tiny”. In: *Twente University* (), p. 45. URL: http://essay.utwente.nl/78118/1/Jebbink_BA_BMS.pdf.

[5] Wim Landuyt et al. “tunus - tiny house project: an interdisciplinary approach to architecture”. In: (2021). URL: http://resolver.tudelft.nl/uuid:de3a7d8b-f558-4bd7-affb-27e7fedf3b8f (visited on 04/26/2021).

[6] B. Urishev. “Microgrid Control Based on the Use and Storage of Renewable Energy Sources”. en. In: *Applied Solar Energy* 54.5 (Nov. 2018), pp. 388–391. ISSN: 0003-701X, 1934-9424. DOI: 10.3103/S0003701X18050201. URL: http://link.springer.com/10.3103/S0003701X18050201 (visited on 04/19/2021).

[7] Zengxun Liu et al. “Development of the interconnected power grid in Europe and suggestions for the energy internet in China”. In: *Global Energy Interconnection* 3.2 (2020), pp. 111–119. ISSN: 2096-5117. DOI: https://doi.org/10.1016/j.gloei.2020.05.003. URL: https://www.sciencedirect.com/science/article/pii/S2096511720300451.

[8] M. Safiuddin. “HISTORY OF ELECTRIC GRID”. In: Jan. 2013, pp. 6–11. ISBN: 978-1-60263-070-3.

[9] IEA. “Global Energy Review 2021”. en. In: 5 (2021). URL: https://www.iea.org/reports/global-energy-review-2021 (visited on 05/10/2021).

[10] Council of European Energy Regulators. “CEER Report on Power Losses”. en. In: (Oct. 2017). URL: https://www.ceer.eu/documents/104400/-/-/09ecee88-e877-3305-6767-e75404637087 (visited on 05/10/2021).

[11] Greg Young Morris et al. “Evaluation of the costs and benefits of Microgrids with consideration of services beyond energy supply”. In: *2012 IEEE Power and Energy Society General Meeting*. 2012, pp. 1–9. DOI: 10.1109/PESGM.2012.6345380.

[12] Karina Garbesi, Vagelis Vossos, and Hongxia Shen. *Catalog of DC Appliances and Power Systems*. LBNL-5364E, 1076790. Oct. 13, 2010, LBNL–5364E, 1076790. DOI: 10.2172/1076790. URL: http://www.osti.gov/servlets/purl/1076790/ (visited on 06/10/2021).

[13] Vagelis Vossos, Karina Garbesi, and Hongxia Shen. “Energy savings from direct-DC in U.S. residential buildings”. In: *Energy and Buildings* 68 (Jan. 1, 2014), pp. 223–231. ISSN: 0378-7788. DOI: 10.1016/j.enbuild.2013.09.009. URL: https://www.sciencedirect.com/science/article/pii/S0378778813005720 (visited on 06/10/2021).

[14] IEA. “Energy Access Outlook 2017”. en. In: (2017). URL: https://www.iea.org/reports/energy-access-outlook-2017 (visited on 04/22/2021).

[15] Wikimedia Foundation. *Moscow Method*. Last accessed 30 September 2020. URL: https://en.wikipedia.org/wiki/MoSCoW_method.

[16] Agustín Agüera-Pérez et al. “Weather forecasts for microgrid energy management: Review, discussion and recommendations”. In: *Applied Energy* 228 (2018), pp. 265–278. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2018.06.087. URL: https://www.sciencedirect.com/science/article/pii/S0306261918309565.

[17] Alejandro J. del Real, Fernando Dorado, and Jaime Durán. “Energy Demand Forecasting Using Deep Learning: Applications for the French Grid”. In: *Energies* 13.9 (May 3, 2020), p. 2242. ISSN: 1996-1073. DOI: 10.3390/en13092242. URL: https://www.mdpi.com/1996-1073/13/9/2242 (visited on 05/01/2021).

[18] Adel Mellit and Alessandro Massi Pavan. “A 24-h forecast of solar irradiance using artificial neural network: Application for performance prediction of a grid-connected PV plant at Trieste, Italy”. In: *Solar Energy* 84.5 (2010), pp. 807–821. ISSN: 0038-092X. DOI: https://doi.org/10.1016/j.solener.2010.02.006. URL: https://www.sciencedirect.com/science/article/pii/S0038092X10000782.

[19] Fermín Rodríguez et al. “Predicting solar energy generation through artificial neural networks using weather forecasts for microgrid control”. In: *Renewable Energy* 126 (2018), pp. 855–864. ISSN: 0960-1481. DOI: https://doi.org/10.1016/j.renene.2018.03.070. URL: https://www.sciencedirect.com/science/article/pii/S0960148118303793.

[20] Sobrina Sobri, Sam Koohi-Kamali, and Nasrudin Abd. Rahim. “Solar photovoltaic generation forecasting methods: A review”. In: *Energy Conversion and Management* 156 (Jan. 15, 2018), pp. 459–497. ISSN: 0196-8904. DOI: 10.1016/j.enconman.2017.11.019. URL: https://www.sciencedirect.com/science/article/pii/S0196890417310622.

[21] Frédéric Haldi and Darren Robinson. “The impact of occupants’ behaviour on building energy demand”. In: *Journal of Building Performance Simulation* 4.4 (Dec. 1, 2011). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/19401493.2011.558213, pp. 323–338. ISSN: 1940-1493. DOI: 10.1080/19401493.2011.558213. URL: https://doi.org/10.1080/19401493.2011.558213 (visited on 06/03/2021).

[22] Alexander Rakhlin. *6.883: Online Methods in Machine Learning*. URL: http://www.mit.edu/~rakhlin/6.883/#notes (visited on 06/04/2021).

[23] Jason Brownlee. *14 Different Types of Learning in Machine Learning*. Machine Learning Mastery. Nov. 10, 2019. URL: https://machinelearningmastery.com/types-of-learning-in-machine-learning/ (visited on 06/04/2021).

[24] Jinjiang Wang et al. “Deep learning for smart manufacturing: Methods and applications”. In: *Journal of Manufacturing Systems*. Special Issue on Smart Manufacturing 48 (July 1, 2018), pp. 144–156. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2018.01.003. URL: https://www.sciencedirect.com/science/article/pii/S0278612518300037 (visited on 05/17/2021).

[25] K. Liu et al. “Comparison of very short-term load forecasting techniques”. In: *IEEE Transactions on Power Systems* 11.2 (May 1996). Conference Name: IEEE Transactions on Power Systems, pp. 877–882. ISSN: 1558-0679. DOI: 10.1109/59.496169.

[26] Jason Brownlee. *Your First Deep Learning Project in Python with Keras Step-By-Step*. Machine Learning Mastery. July 23, 2019. URL: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/ (visited on 05/17/2021).

[27] KNMI. *Uurgegevens van het weer in Nederland*. 2021. URL: https://www.knmi.nl/nederland-nu/klimatologie/uurgegevens (visited on 04/24/2021).

[28] KNMI. *Actuele waarnemingen*. 2021. URL: https://www.knmi.nl/nederland-nu/weer/waarnemingen (visited on 04/24/2021).

[29] Joris Koeners. *Solar Powered Bikes*. VictronData. URL: http://solarpoweredbikes.tudelft.nl/phpmyadmin/index.php (visited on 06/06/2021).

[30] *PVLEWI*. URL: http://pvlewi.ewi.tudelft.nl/ (visited on 06/06/2021).

[31] Carine Nieuweling. *European Meteorological derived high resolution renewable energy source generation time series*. EU Science Hub - European Commission. Nov. 17, 2016. URL: https://ec.europa.eu/jrc/en/scientific-tool/emhires (visited on 06/14/2021).

[32] Liander. *Beschikbare data | Liander*. URL: https://www.liander.nl/partners/datadiensten/open-data/data (visited on 05/17/2021).

[33] *Over Liander | Liander*. URL: https://www.liander.nl/over-liander (visited on 06/06/2021).

[34] Jason Brownlee. *Difference Between a Batch and an Epoch in a Neural Network*. Machine Learning Mastery. July 19, 2018. URL: https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/ (visited on 06/08/2021).

[35] Jason Brownlee. *Regression Tutorial with the Keras Deep Learning Library in Python*. Machine Learning Mastery. June 8, 2016. URL: https://machinelearningmastery.com/regression-tutorial-keras-deep-learning-library-python/ (visited on 06/11/2021).

[36] Ming Ding, Lei Wang, and Rui Bi. "An ANN-based Approach for Forecasting the Power Output of Photovoltaic System". In: *Procedia Environmental Sciences* 11 (2011), pp. 1308–1315. ISSN: 18780296. DOI: 10.1016/j.proenv.2011.12.196. URL: https://linkinghub.elsevier.com/retrieve/pii/S187802961101019X (visited on 06/11/2021).

[37] G Amarasinghe and S Abeygunawardane. "An artificial neural network for solar power generation forecasting using weather parameters". In: 112th Annual Sessions, Institution of Engineers Sri Lanka. Colombo, Sri Lanka, Oct. 19, 2018, pp. 431–438. URL: https://www.researchgate.net/publication/328530283_An_artificial_neural_network_for_solar_power_generation_forecasting_using_weather_parameters.

[38] Jose Manuel Barrera et al. "Solar Energy Prediction Model Based on Artificial Neural Networks and Open Data". In: *Sustainability* 12.17 (Jan. 2020). Number: 17 Publisher: Multidisciplinary Digital Publishing Institute, p. 6915. DOI: 10.3390/su12176915. URL: https://www.mdpi.com/2071-1050/12/17/6915 (visited on 06/11/2021).

[39] Jeff Heaton. *Introduction to Neural Networks with Java*. Google-Books-ID: Swlcw7M4uD8C. Heaton Research, Inc., 2008. 440 pp. ISBN: 978-1-60439-008-7.

[40] P Zhang. *Advanced Industrial Control Technology*. Elsevier, 2010. DOI: 10.1016/c2009-0-20337-0. URL: https://doi.org/10.1016/c2009-0-20337-0.

[41] I Ross. *A primer on Pontryagin's principle in optimal control*. Carmel, Calif: Collegiate Publishers, 2009. ISBN: 978-0-9843571-0-9.

[42] Liuping Wang. *Model predictive control system design and implementation using MATLAB*. London: Springer, 2009. ISBN: 978-1-84882-331-0.

[43] William E. Hart, Jean-Paul Watson, and David L. Woodruff. "Pyomo: modeling and solving mathematical programs in Python". In: *Mathematical Programming Computation* 3.3 (2011), pp. 219–260.

[44] Michael L. Bynum et al. *Pyomo–optimization modeling in python*. Third. Vol. 67. Springer Science & Business Media, 2021.

[45] GLPK. *GNU Linear Programming Kit, Version X.Y*. URL: http://www.gnu.org/software/glpk/glpk.html.

[46] Ferguson. T. S. *Linear Programming A Concise Introdcution*. Springer, 1998.

[47] Milieu Centraal. *Wasmachine*. 2020. URL: https://www.milieucentraal.nl/energie-besparen/apparaten-in-huis/wasmachine/.

[48] *Anaconda Software Distribution*. Version Vers. 2-2.4.0. 2020. URL: https://docs.anaconda.com/.

[49] François Chollet et al. *Keras*. https://keras.io. 2015.

[50] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[51] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[52] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[53] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.