# Simulation of the Dutch electricity system

E.I. de Wolff and M. van Zonneveld

**A software expansion for the Illuminator**


TU Delft

# Simulation of the Dutch electricity system

by

E.I. de Wolff and M. van Zonneveld

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Tuesday June 26, 2023 at 15:30.

## Abstract

The aim of this report is to discuss the design of the software that creates a simulation for the national electricity grid level of the Netherlands. This is done by further developing the open-source energy system integration development kit called the Illuminator. Where the goal of this software is to create an extra case, add a Graphical User Interface (GUI), and add a way to evaluate created configurations.

Student numbers: 5126967
5115523
Project duration: April 24, 2023 – June 30, 2023
Thesis committee: Dr. ir. M. Cvetkovic, TU Delft, supervisor
Dr. ir. P. Bauer, TU Delft
Dr. ir. P. Manganiello, TU Delft

TU Delft

# Preface

In advance, we would like to thank our supervisor Milos, for guiding us through the final stage of the Bachelor, being a sparring partner about some issues and giving us the freedom to form the project in a way we seemed fit. Next to that we would like to thank Milos and Patrizio for noticing the insurmountable workload ahead of time and helping us to find a way to still present an adequate product within the given timeframe.

*E.I. de Wolff and M. van Zonneveld*
*Delft, January 2013*

# Contents

# Chapter 1

# Introduction

The energy transition is one of the problems that need to be solved in the coming years. However, this comes with many obstacles, some of which are quite of technical nature. For people people that are non-experts in this area, it can be very difficult to see what the challenges are and why they can't be solved in a certain way. Part of this seems to be due to the abstractness of certain issues, which might be difficult to explain without the right visualisation. To educate en demonstrate this to people [18] have made an open-source energy system integration development kit called the Illuminator.
The Illuminator contains a certain set of models that are interconnected, such that a simulation on this network can be run. Such a simulation would consist of the models sending, receiving and processing varying data over time, representing the energy flows from one system component to the other. The energy flows are thus represented by signals, and not actual current flows .[18]
Currently, five models are included such that a residential case can be simulated. These models are a wind turbine, a PV system, battery storage, hydrogen storage, and a load. In Figure 1.1 the interaction of the models can be seen. These models are all made in the programming language Python and can run either on different Raspberry Pi's each simulating a model or via a centralized Python compiler. The goal of this kit was that it would be modular and flexible to reconfigure even for non-experts and that it could handle and depict a variety of the challenges the energy transition brings.[18] Taking this all in mind it would be useful to add a new case to the kit. For now, this residential case only shows certain obstacles that could arise on a small-scale. As there are a lot more to educate on, such a new case could prove very educational. Hence, within this project, the range of cases is expanded.

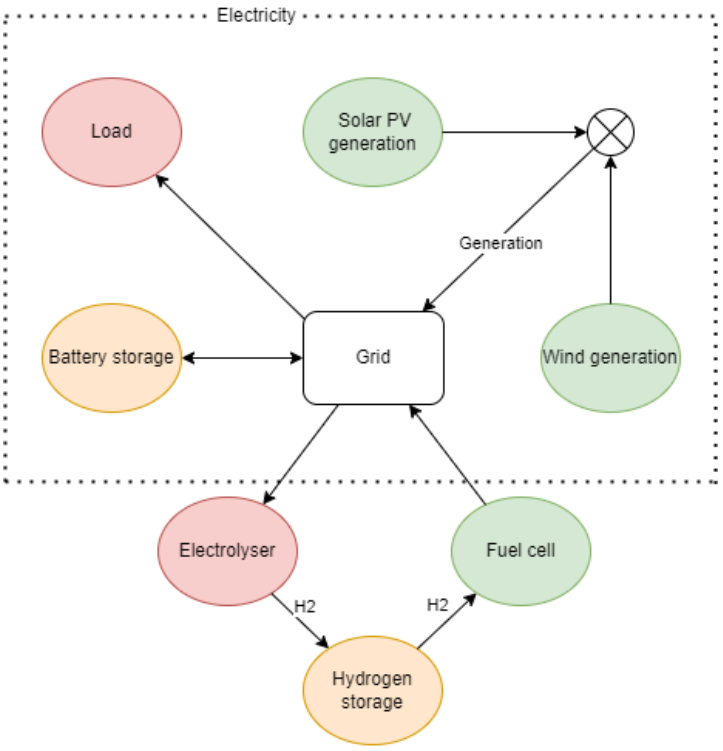


Figure 1.1: The interaction of different models within the system and the direction of the energy flows between them. Green icons depict delivering power to the grid, whereas red depicts the consumption of energy. Orange icons depict both consumption and delivery of energy.

The case that was decided to implement is a national electricity grid-level case of the Netherlands. In combination with 2 other groups, this project is done. One group worked on and created some hardware components, representing certain models.[50] This would add to the visualisation and plug-and-play factor of the Illuminator. The other group performed a case study about what the energy usage in 2030 would look like. [5] The group presenting this thesis made software representations of the components and models that were needed to scale the system to depict the national grid.
In Figure 1.2 the overview of the chosen representation of the national grid is shown. The generation sources and especially the loads have been taken under the loop more extensively, and different sectors are added, as opposed to a single load or wind turbine. Another very important part of the electricity grid that appeared to be missing, was the influence of fossil fuels. The dream is of course to have a net-zero electricity grid, but this is not yet the goal for 2030.[5] Hence this is also taken into the case.

All these components needed to be created and/or scaled to the national grid level of 2021 and 2030. The choice to have both 2021 and 2030, is that it could be very educative to see whether certain problems or obstacles could increase or decrease as the energy transition would set further course. That is why a baseline year was added as well.
In combination with [5] the scaling to 2030 is done as most of the inputs come from their report, as this group has focused more on making the Python code and 15 min scale load models.

Once started making the models in Python, it was noticed that it is quite unclear where to find and modify the parameters to start the simulation in any way possible. Apart from that, after having successfully run a simulation, all the data is sent to a database called Weight and Biases. As the data available here, was mainly coding-technical and hence a lot of unnecessary data, it did not present a good overview of the effect of the system. To solve these issues, it was decided to make and implement a graphical user interface (GUI). This way, the user would be guided more easily through the process of starting a simulation. Afterwards, only critical data would be shown. Overall, the GUI is supposed to make the Illuminator more user-friendly.

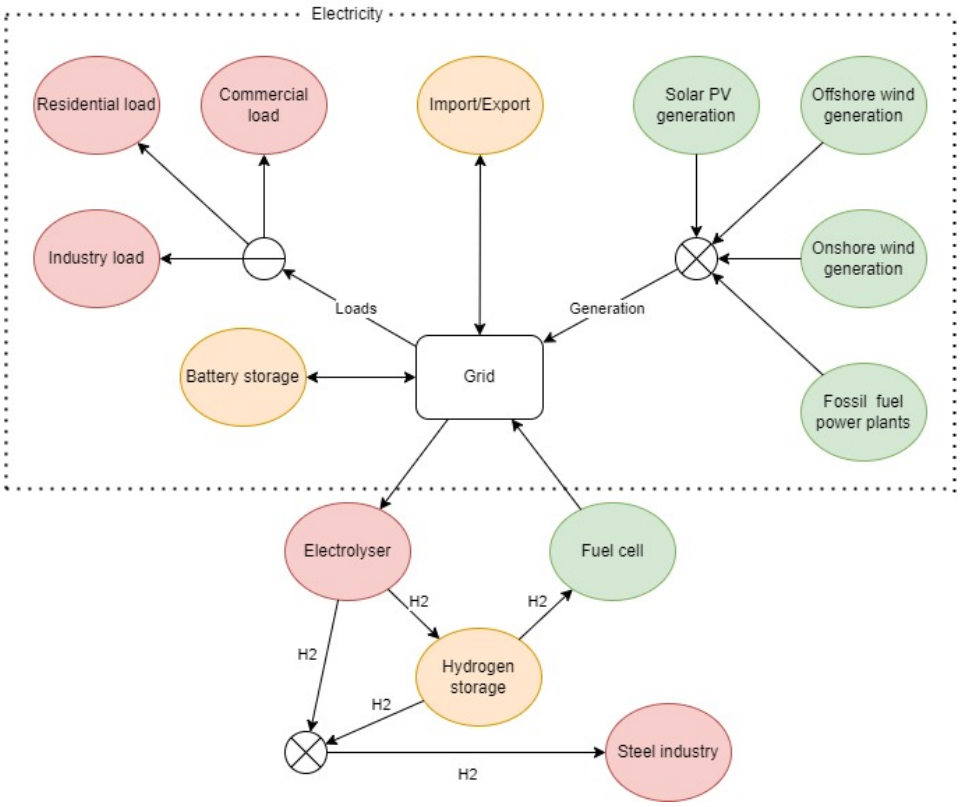


Figure 1.2: The interaction of different models within the system and the direction of their energy flows between them. Green icons depict delivering power to the grid, whereas red depicts the consumption of energy. Orange icons depict both consumption and delivery of energy.

Another thing that was noticed is that the user did not have a way to know the quality of the configuration they had made in comparison with another. So this was also decided to add.

## 1.1 Thesis overview

First of all the requirements for the project are defined and this can be found in Chapter 2. After this, the implementation of the project is described in Chapters 3, 4 and 5. The validation of the results is done and described in Chapter 6. Where the thesis is concluded with a discussion of the results in 7 and the conclusion in Chapter 8. Some further possible elaborations of the project are also discussed in Chapter 8.

As a large part of the collaboration and integration of this thesis project and the Case Study subgroup, is mainly information transfer, it is hard to depict. Therefore, it is advised to specifically keep an eye out for the citation of the thesis of this subgroup: ([5]). This probably indicates the model has been adjusted to fit the case study performed by this subgroup. Next to that, the exact scope of this case study has been coordinated in fine collaboration between the subgroups, in terms of what is feasible and interesting.

# Chapter 2

# Program of requirements

For this project, the main task is to develop the software to run the case study for the Dutch electricity grid for the years 2021 and 2030. The requirements that are formulated to fulfill this task are:

- The software must run in Python as the Illuminator already has been developed in this language.
- The software has to have two different operation modes in which either yearly recorded data can be used or inputs from created hardware can be used.
- The software has to be able to run the created models of the load for the industry, residential, and commercial sectors of the Netherlands.
- The simulation has to be able to run the created models for the different main generation sources of electricity in the Netherlands.
- The simulation must be controlled via a controller model that controls the interaction between different generation sources, loads, and energy storage.
- The software has to become user-friendly by adding a GUI.
- There must be a way inside the simulation to measure the quality of the setup that the end user can create.

For the GUI and the controller model, some extra requirements are specified. These are described in the following sections.

## 2.1 GUI

For the GUI the goal is that the software becomes more user-friendly. To make sure that this will happen there are some requirements specified, these are:

- The GUI needs to let the user choose which operation mode the simulation must run in.
- The GUI needs to display the electricity generation sources, state of charge of the storage, loads of commercial, residential, and industrial sectors, total load, and the measure of quality.
- As this might be a lot of information at once, the GUI must give the user the possibility to select which information is shown. To prevent an information-overload, a maximum of four graphs at a time is set.
- The GUI needs to give the user the possibility to choose the start date of the simulation and what time span in hours (not in real-time) must be simulated.
- The user must be able to choose which devices are attached to the simulation if running in the no-hardware mode.

The implementation of the GUI model can be found in Chapter 5.

## 2.2 Controller model

The controller model is the center of the simulation. This model needs to deal with the complexity of the electricity system and works in the way the current electricity systems deals with generation. The requirements are:

- Send out control signals about what energy goes where; e.g., directly to load, store in battery or H2 storage, if shortcomings, gain energy from storage or fossil fuels. A possible controller model has already been delivered by the Illuminator group, but this might be adjusted and has to be modified to the addition of fossil fuels.
- Calculate the measure of quality for the chosen setup.
- Using storage components to make sure that it is as sustainable or cheap as possible. Due to the addition of the fossil industry, this is a consideration to be made when implementing.
- Take every generation source and load into account if and only if selected to be part of the simulation.

The implementation of the controller can be found in Section 4.4.

# Chapter 3

# CO2

As can be seen in the programme of requirements, it was decided to add another feature to the demonstrator: a measure of quality. This would describe how good a system is, as opposed to possible other configurations. It was decided to use the total greenhouse gas (GHG) emissions related to the power generation in the built setup for this measure of quality. This will be expressed in a CO2-equivalent mass. Emissions other than CO2 can be converted to CO2-equivalent using their global warming potential (GWP). This is a measure of how much energy will be absorbed by this GHG, as opposed to the same mass of CO2, over a given period of time.[29] As global warming is a rising subject, the number of greenhouse gasses emitted can be a very crucial detail about such a setup. It has also been investigated that the acceptance of, in this case, PV panels under citizens is ”high when consumers believe that solar PV could mitigate carbon emission”. [44] Hence, being able to prove such mitigation, can indeed have a large influence, thus was considered one of the most important pieces of information output.

To be able to state the number of emissions emitted at one point, one must first have data available about emissions due to the main power generation options within the Netherlands. This project entailed solar PV generation, wind generation, and fossil fuel generation.

## 3.1 PV emissions

As the reaction that leads to electricity for photovoltaic cells does not depend on burning carbohydrates, no direct emissions are expected for this energy generation type. It is also assumed that the possible degradation or increase in maintenance due to much usage is insignificant to the rest of the emissions. This leads to the assumption that no emissions are related to the amount of power produced and all GHG (greenhouse gas) emissions could be considered as constant over time. There are different types of photovoltaic cells in use, and each of them has a different amount of CO2-eq emissions per kWh. Ribbon-silicon, multi-crystalline silicon, and mono-crystalline silicon PV cells 30, 35 and $45\, g\, CO_2\text{-}eq/kWh$, respectively. [6] This is mainly due to the production and planned demolition of such cells, which happens at the beginning and end of their lifetime. However, these emissions are said to be spread out over their entire lifetime. Here, an average of $35\, g\, CO_2\text{-}eq/kWh$ is assumed, especially since multi-crystalline cells are the most common nowadays.[38]

These GHG emission values are determined for South-Europian irradiation, at $1700 kWh/m^2/yr$.[6] More away from the equator, however, the irradiation is a lot less. In the Netherlands, this is around $1000 kWh/m^2/yr$. [1]. This would mean a factor of 1.7 less energy could be recovered from the sun, hence a factor of 1.7 more emissions per kWh. This leads to 59.5 g/kWh.

Yet, this still is dependent on the amount of power generated, not on the installed capacity. There is a difference between theoretical and actual output, this ratio is called the performance ratio. [2] This can be assumed in the range of 0.75 = $\frac{actual\, output}{theorethical\, output}$. [51] This would result in a 44.625 $g\, CO_2\text{-}eq/kWp_{\text{installed}}/h$ = 11.156 $g\, CO_2\text{-}eq/kWp_{\text{installed}}/15\, min$.

## 3.2 Wind energy

For wind energy, there is looked at onshore and offshore wind turbines. Since these have different specifications of what is most commonly used.

Offshore wind energy is estimated to consist of on average 6$MW$ turbines that are mounted to the ground, instead of floating. [5] A 5$MW$ turbine, which is mounted to the seabed using a jacket, has over its lifetime 18.9 $gCO2\text{-}eq/kWh$.[32] This takes into account material costs, installation and destruction as well as maintenance. The same assumption as with PV panels holds for Wind energy: all GHGs are assumed to be independent of electricity produced. Again, this constant then has to be yet changed into a number of emissions per installed capacity. For this conversion, the capacity factor is of importance. This is similar to the performance ratio of PV panels, namely the ratio between actual output and theoretical output. This difference appears due to the fact that the wind is not always blowing at speeds that reach maximum wind generation capacity. For offshore wind turbines in the Netherlands, this capacity factor is 0.39. [45] This leads to 7.371 $gCO2\text{-}eq/kWp_{installed}/h$ = 1.843 $g\,CO2\text{-}eq/kWp_{installed}/15min.$

For a 1.5$MW$ onshore wind turbine, their emission is 16.6 $g\,CO2\text{-}eq/kWh$.[31]. Using a capacity factor of 0.23, this comes down to 3.818 $gCO2\text{-}eq/kWp_{installed}/h$ = 0.955 $gCO2\text{-}eq/kWp_{installed}/15min.$

## 3.3 Fossil fuels power plants

In the year 2021, still, a lot of electricity generated is due to fossil fuels, as can be seen in Table 3.1. This total was still 59.4% of the annual electricity generation.

| Plant type | Energy [$GWh$] | Fraction |
|---|---|---|
| Coal | 16346.0 | 0.2203 |
| Natural gas | 56505.0 | 0.7617 |
| Oil | 1331.0 | 0.01794 |
| Total | 72182.0 | 1 |

Table 3.1: Total electricity generation per fossil source in the Netherlands in 2021 [46]

Unfortunately, for fossil fuel plants, emissions are not as straightforward as with solar and wind power plants. This is because fossil fuels have more than only construction/demolition that contributes to GHGs. To start with the fact that the energy is created by burning carbohydrates, which inherently emits CO2, leading to direct emissions. Next to that, mining and transport of these fossil fuels also are responsible for a lot of GHGs and are accounted for as indirect emissions.[30] This holds as well for construction and demolition. In table 3.2, the direct and total life cycle emissions are shown for each fossil fuel plant type.

| Plant type | Direct [$g\,CO_2\text{-}eq/kWh$] | Lifecycle [$g\,CO_2\text{-}eq/kWh$] |
|---|---|---|
| Coal | 881.6 | 985.1 |
| Natural gas | 420 | 460 |
| Oil | 809 | 964 |

Table 3.2: GHG emissions per fossil source [30] [36]

For the simulations, however, the emission constants would desirably be split into emissions that are dependent on power production and those that are solely dependent on installed capacity. Direct emissions fall under the first category, whilst indirect emissions have to be split.

For coal power plants in the UK, 72.9 g of the 108.1/kWh of indirect emissions is due to mining. As with more electricity production, the more coal is needed, the more mining is expected. Hence, this is fully accounted for generation-dependent, or variable, emissions. This is also assumed to be linear, even though this probably is not the case. 31.5% of indirect emissions, resulting in 34.05 g/kWh, is

due to material production and transport. This leaves 1.15g/kWh for actual construction. [30] An estimation is made that 10% of the material production and transport is used for the construction as well. Hence, 4.45 $g\ CO_2\text{-}eq/kWh$ (4.21% of indirect emissions, 0.465% of total emissions) is assumed to be invariant of how much energy is produced by the plant. This 0.465% is in the same range as the 0.4% and 0.5% found in other life cycle assessments for coal plants. [40] [20] A coal plant with the following specifications: 660$MW$ rated power, a load factor of 80% [30] and a lifespan of 40 years, results in $1.851 \cdot 10^{11} kWh$ created electricity and thus resulting in $8.239 \cdot 10^{11} g\ CO_2\text{-}eq$ in total. This accounts for 1248278 $g/kW_{\text{installed}}$ over 40 years, per 15 minutes this is $0.89 g/kW_{\text{installed}}/15min$. With the current 4012$MW$ capacity of fossil hard coal power plants installed in the Netherlands, this sums to $3570 kgCO2\text{-}eq/15minutes$ for production-independent emissions. [11].

Assuming the same percentage of indirect emissions is due to construction and demolition for oil and gas as for coal, and assuming the same load factor, one can find the emissions per installed capacity for oil and gas as well. Currently, 13372$MW$ of natural gas plants are installed in the Netherlands. [11]. As no oil plants are mentioned, it is assumed that the ratio between capacity installed and the amount of $GWh$ produced is the same as with coal plants. Hence, $1331.0 GWh \cdot \frac{4012MW}{16346GWh} = 321.5MW$ is assumed to be installed. As can be seen from Table 3.2, indirect emissions for oil are 946-809 g CO2-eq./kWh and for gas 468-420 g CO2-eq/kWh, which is 137g and 48g respectfully. Of that, 4.21% top is assumed to be due to construction and demolition. Using the load factor of 80%, this results in 1.154 for oil and 0.4042 $g/kW_{\text{installed}}/15min$ for gas. With an estimated 321.5$MW$ installed, non-variable emissions per 15 minutes seem to be 371.0 kg for the oil sector and with 13372$MW$ gas plants installed, non-variable emissions in this sector add up to $5405 kgCO2 - eq/15minutes$. In total, the construction and planned demolition of all power plants installed, on average, contributes to the emission of 9.346 tonnes $CO_2\text{-}eq$ per 15 minutes.

For variable emissions, this leaves the sum of the direct emissions and 95.79% of the indirect emissions per sector. For gas, for example, this would be $420 g/kWh + 0.9579 \cdot (468 - 420) g/kWh = 466.0 g\ CO_2\text{-}eq$. For oil and coal, this is 957.5 and 985.1 $g\ CO_2\text{-}eq/kWh$ respectively.

Taking into account the share of each type of plant within the fossil fuel electricity generation, the total variable emissions can be said to be $0.7617 \cdot 466.0 + 0.2203 \cdot 985.1 + 0.01794 \cdot 957.5 = 589.1 g\ CO_2\text{-}eq/kWh$.

## 3.4 Import and export

A lot of energy, under which electricity, is imported and exported from and to other countries. In the year 2021 in the Netherlands, this was composed of 20632 $GWh$ imports and 20885 $GWh$ export.[41] As this is such a large amount of electricity, it seems fair to account for emissions for this as well. The hard part here is to determine what type of electricity is imported, whether this is e.g. wind power or additionally created fossil electricity.

It seems logical that green, renewable energy in the Netherlands, instead of storing it, can be sold to neighboring countries, whenever there is a surplus. The same could hold for the energy imported if neighboring countries would have a surplus. This would, however still raise questions about the composition of this renewable energy. It raises the question of in what sense the surplus would be stored in the Netherlands when it can be directly sold as well.

Another option that seems quite evident, is pure influence by the energy market. It could be that at some point the prices at which energy can be sold are a lot higher in a neighboring country. A fossil electricity supplier might step in at that moment and decide to generate more than is necessary for the Netherlands, with the intention to sell this across borders.

There are multiple options about which way the electricity can be composed and they seem to be dependent on the economic energy market as well. As within the confines of this project, expanding the model to the market would seem too broad, it was decided to stick to the annual national electricity

composition. Based on the share of each supply, within the confines of wind, solar, and fossils, their emission contributions are taken into account. The averaged wind emissions are scaled to the ratio of energy collected for both offshore and onshore in 2021. This can be determined using the installed capacities and emission constants found in the section further above. With 5.3$GW$ of onshore and 2.460$GW$ of offshore, the total installed capacity is 7.76$GW$.[5] This would result in an weighted average of $\frac{5.3 \cdot 16.6 + 2.460 \cdot 18.9}{7.76} = 17.33 g\ CO_2\text{-}eq/kWh$

| Generation type | Power generation $[GWh]$ | Fraction $[-]$ | Emissions $[g\ CO_2\text{-}eq/kWh]$ | Emission fraction $[g\ CO_2\text{-}eq/kWh]$ |
|---|---|---|---|---|
| Coal | 16346.0 | 0.1550 | 985.1 | 152.69 |
| Natural gas | 56505.0 | 0.5357 | 460 | 246.42 |
| Oil | 1331.0 | 0.01262 | 964 | 12.166 |
| Wind | 17980.0 | 0.1705 | 17.33 | 2.955 |
| Solar | 13320.0 | 0.1263 | 35 | 4.421 |
| Total | 105482 | 1 | - | 418.65 |

Table 3.3: The emissions attributed to the import and export of electricity. [46][30][36][6]

## 3.5 Hydrogen use

Hydrogen can be used as a way of storing electrical energy, as can be read in section 4.3. This created hydrogen can then be used in a lot of applications, however. It can for example be fed back to a fuel cell to regenerate electricity. Next to that, hydrogen can be utilized in a lot of sectors, such as transport, industrial heat, and industrial processes such as steel or ammonia refinery.[7][3]

The amount of $CO_2$ that can be reduced by feeding hydrogen into that sector, differs per process. For example, if industrial heat is produced with hydrogen, instead of natural gas or coal, this would reduce emissions by 7 and 12 $kg_{CO_2}/kg_{H_2}$, respectively. Transportation in buses or shipping saves slightly more, namely 13 and 14 $kg_{CO_2}/kg_{H_2}$. A large standout is the steel industry. The blast furnace used is usually fired with coking coal or oil. By replacing this with hydrogen, 32 $kg_{CO_2}/kg_{H_2}$ can be saved. However, the usage of hydrogen in the blast furnace leads to the necessity of another, additional process, which needs more hydrogen. Hence, effectively 24 $kg_{CO_2}/kg_{H_2}$ can be saved, which is still a lot more than any other industry. [7]

# Chapter 4

# Models

For this project, certain models are made. In Figure 1.2, the connections between the models present, as implemented in this project, are shown. This shows only the energy flows and their directions. Here, the controller model is missing, as this does not directly consume or generate any energy. The diagram showing all models, including the communication between them is shown in Figure A.1.

In the sections below, the implementation and design of each model is explained.

## 4.1 Loads

It was chosen to use three different loads to map the electricity demand of the Netherlands. These are commercial, residential, and industrial loads. The decisions about the load profiles are given in this section. For every load, a load profile has been made for the years 2021 and 2030 and the load model from [35] is copied only some names are different such that the model can work with the created load profiles. These profiles supply data to the simulation about the varying load of that sector every 15 minutes.

### 4.1.1 Residential load

There are two load profiles made for the residential load. First, the 2021 profile is explained and then the 2030 profile is explained.

#### Load profile for 2021

The residential load is based on the load profiles that are made by MFFBAS. These load profiles are used by the grid operators of the Netherlands to estimate the profiles for all the different electricity connections [49]. According to [37] all electricity connections lower than $3x50\,A$ are for households. So this checks the boxes of five of the categories MFFBAS gives for the year 2021: lower than or equal to $3x25\,A$ with 1 tariff, lower than or equal to $3x25\,A$ with a different day and night tariff, lower than or equal to $3x25\,A$ with a different day and evening tariff, between $3x25$ and $3x80\,A$ with 1 tariff, and between $3x25$ and $3x80\,A$ with 2 tariffs either evening or night. There is assumed that most people have only 1 tariff for their electricity costs. So that profile is used for both lower than and equal to $3x25\,A$ and between $3x25$ and $3x80\,A$.

The input for these profiles is the total use of electricity in a year in kWh. So to find out how much electricity is used by the residential sector inside the different categories for the year 2021 there was first looked at all grid operator's data of small consumers' electricity usage. [26], [43], [27], [15], [24], and [33] are used to determine how much electricity is used in the two different categories. From this data, it was found that $18.8756854\,TWh$ was used for the category lower than or equal to $3x25\,A$. The connections $1x10$, $1x20$, $1x25$, and $3x25\,A$ are put under this category. For the category between $3x25$ and $3x80A$, there is found that $14513358369\,kWh$ was used. Looking at this total and comparing it with the data of [42] where $81.9\,PJ$ is for houses in the year 2021, which is $2.275 * 10^{10}\,kWh$ there can be

seen that there is a difference of $1.06 * 10^{10}\ kWh$, which is a lot but can be explained. This is due to the fact that the data from the grid operators only give the connection that is the most present in a certain number of electricity connections. So the total electricity between $3x25$ and $3x80\ A$ is determined by subtracting the electricity for lower than and equal to $3x25A$ from the total electricity for houses in the Netherlands. Which gives: $2.275 * 10^{10} - 18875685400 = 3.87 * 10^{9}\ kWh$.

Combing these two different categories together gives the load profile of the residential sector. In Figure 4.1 the load profile is given for a week in the winter and in Figure 4.2 the load profile is given for a week in the summer. As one can see is the load in the summer a lot less than the load in the winter, which is expected since people use more electricity in the winter than in the summer in the Netherlands [48]. As well as that on the weekend on average there is also a higher load in the afternoon than during the working days. This makes sense since people are more at home on those days than when they have to work.

What [42] doesn't take into account is the amount of electricity that is produced and used via solar panels on rooftops of houses behind the grid electricity connection. To approximate this, there is looked at how much GWp there lies on rooftops in 2021. This was 6 GWp according to [39] and [5] stated that the efficiency of an average solar panel in 2021 was 22%. Putting this in the PV model that was already created by [35] the generation for a whole year is determined. This is added to the created load profile, such that this extra load is taken into account. In Figure 4.4 and Figure 4.3 these load profiles can be seen for the same weeks as used before. One can see that the PV generation is visible in the model and that is it maybe not the best way to do this, but it will give some information about how much extra load there is.

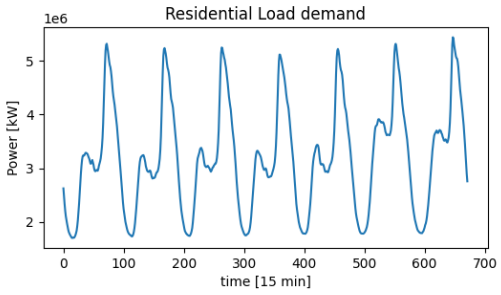


Figure 4.1: Residential Load of one week 11-17 January 2021

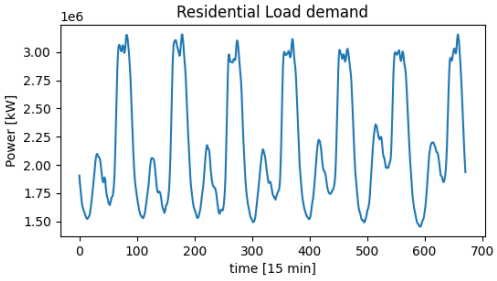


Figure 4.2: Residential load of one week 9-15 August 2021

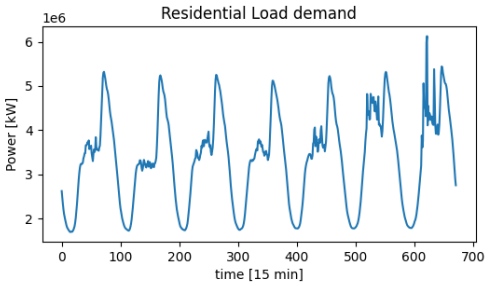


Figure 4.3: Residential Load of one week 11-17 January 2021 + PV

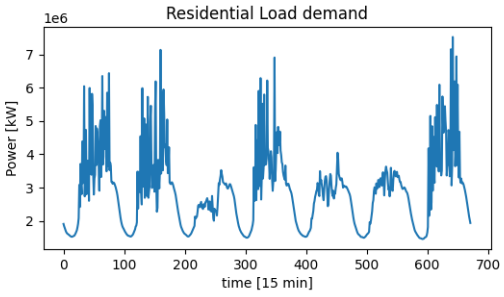


Figure 4.4: Residential load of one week 9-15 August 2021 + PV

### Load profile for 2030

For the profile of 2030, the profile for 2021 without PV is used as the base. This profile is scaled with the factor of the total energy demand of 2030 that is given by [5] (Unfortunately there wasn't time to

look only at the electricity demand) divided by the total electricity of 2021. Every time step of 2021 is then scaled with this factor, which gives the same load shape but with higher values.

### 4.1.2 Commercial load

There are again two load profiles made for the commercial load. One for 2021 and one for 2030.

#### Load profile for 2021

For the load profile of the commercial sector, there is first looked at how much final electricity this sector uses. According to [42] this was $129\,PJ$ or $35.89\,TWh$ for the commercial sector. Then the consumers profile from [49] was used to determine the profile. The profile that was chosen was the profile with the electrical connection between $3x25$ and $3x80\,A$. There is assumed that most companies in the commercial sector are not so big that they need a connection for $3x80\,A$. This is also verified with the data of all the grid operators and it was found that this was only $0.26\,TWh$ which is small enough (only 0,72 %) to neglect. In Figures 4.5 and 4.6 a typical week of the summer and winter of this load can be seen. As one can see, during the summer the loads are again lower than in the winter. Also on one average day, there are two peaks; one around 8:30 in the morning and one at 18:30 in the evening. These to can be explained as the morning around 8:30 most people start working at the companies and around 18:30 all restaurants and other entertainment are open and people start going to them.

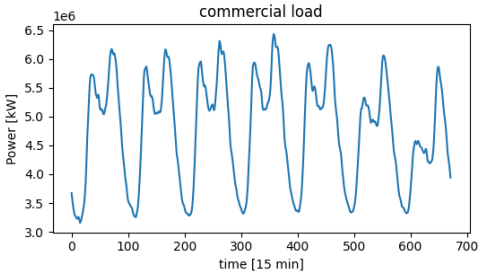


Figure 4.5: Commercial Load of one week 11-17 January 2021

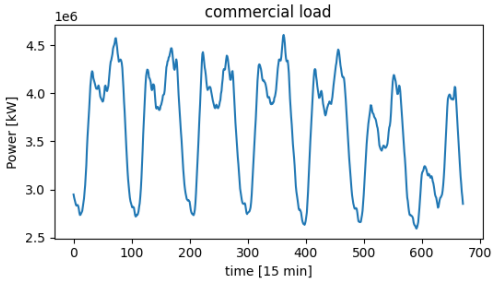


Figure 4.6: Commercial load of one week 9-15 August 2021

#### Load profile for 2030

For the profile of 2030, the profile of 2021 is used as the base and this profile is scaled with the factor of the total energy demand of 2030 that is given by [5] (Unfortunately there wasn't time to look only at the electricity demand) divided by the total electricity of 2021. Every time step of 2021 is then scaled with this factor, which gives the same load shape but with higher values.

### 4.1.3 Industrial load

There are again two load profiles made for the commercial load. One for 2021 and one for 2030.

#### Load profile for 2021

For the load profile, there is looked in different ways to how this would match reality the most. As most industries in the Netherlands are energy-intensive industries (according to [42], it was $91.7\,PJ$ of the total of $127.4\,PJ$). The first profile that was made is constant during the whole year as [19] says should have been done. But then when then all profiles combined were evaluated with the actual load that there was in [10], it did not match. The difference was $-7.68\,TWh$ which is quite a lot especially since at some time points there is more electricity than there should have been. This amount that was not taken into account comes from the agriculture sector. This difference is added to the total industrial

load to first of all give it a more realistic profile since it would make sense that there is some fluctuation in the load profile of the industry. In Figure 4.7 the first profile of the constant load can be seen and in Figures 4.8 and 4.9 the profile with the added load can be seen. As one can see is there a big peak at night. This can be explained as at night the electricity is cheaper and greenhouse horticulture needs to put on lights to let their plants grow[34]. As well as that the load is again higher in the winter than in the summer.

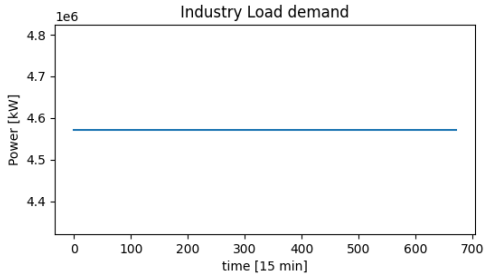


Figure 4.7: Industry constant load of one week for 2021

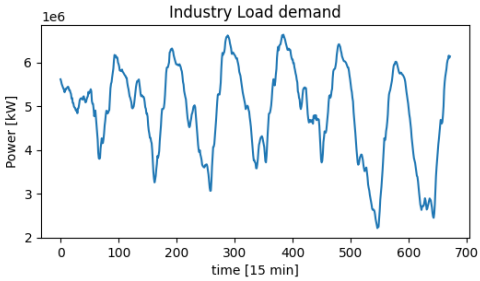


Figure 4.8: The industry load of one week 9-15 August 2021

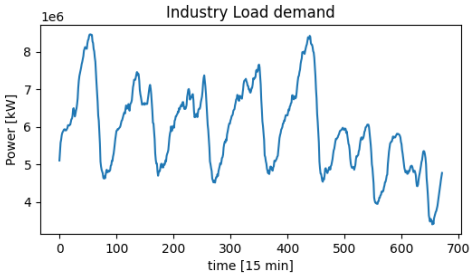


Figure 4.9: The industry load of one week 11-17 January 2021

Load profile for 2030

For the profile of 2030, the profile of 2021 is used as the base and this profile is scaled with the factor of the total electricity of 2030 that is given by [5] divided by the total electricity of 2021. The total electricity of 2030 for the industry sector is 194.2,. Unfortunately, there was no time to look at the change in total electricity for the agriculture sector so the value from 2021 is just added to the total electricity of 2030 for correct scaling. Every time step of 2021 is then scaled with this factor, which gives the same load shape but with higher values.

## 4.2 Generation models

### 4.2.1 Wind

The power generated by a wind turbine is related to wind speed as shown in Equation 4.1.[52] When the wind speed increases, so does the generation, with a power of 3. As shown in Figure 4.10, this does have its limitations. The power generated is limited to the rated power of a wind turbine, which is reached at the so-called rated wind speed. This maximum is kept until the cut-out speed, where the wind turbine is completely shut off, to shield it from unnecessary stress on the rotor. The cut-in speed is simply the wind speed where the blades start rotating and the turbine starts generating power.[21] The model to simulate such a wind turbine is based on this principle. Below cut-in wind speed no power is generated and between cut-in speed and rated speed, Equation 4.1 is followed. Above rated speed up until cut-out speed, the generation is equal to rated power, after which no power is generated anymore.

Hence, using wind speed data at one point in time, one can determine the power generation of a wind turbine. This data is extracted from the National Solar Radiance Database (NSRDB). [28]

$$P(V) = \frac{1}{2}V^3 \cdot \rho \cdot \pi(\frac{d}{2})^2 \cdot \eta \tag{4.1}$$

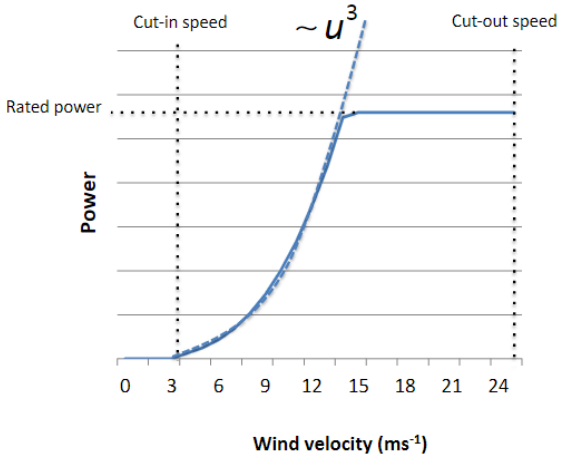


Figure 4.10: The relation between wind speed and power generation, showing the effect of the cut-in and cut-out speed, as well as the rated power and speed. [52]

It is shown that production is dependent on wind velocity. However, wind velocity is, besides intermittency, dependent on height above the ground or sea. This is shown in Figure 4.11. Using Equations 4.2 and 4.3, the relation between height en wind speed can be shown.[53] Lower to the ground another relation holds better than far above the ground, however. Around a so-called blending height, these relations create a well-fitting overflow, which is often set at 60m. [53] This blending height is pictured as the green dot and lines in Figure 4.11.
As the wind data used in the simulations is at 10m height, this can be set as the first reference height for Equation 4.2. This is illustrated as the orange dot in Figure 4.11 Using the $z_0$ value, the wind speed at a height up to 60m can be estimated. This $z_0$ value correlates to the ”roughness” of the surface below the wind. Large cities for example still hold back quite some wind. Hence with a high $z_0$, the wind speed does not rise as fast with height, as can be seen in Figure 4.11.
Above 60m, one can invoke Equation 4.3 to estimate the wind speed. Here, the reference height and speed are 60m and the wind speed at 60m is found using the other equation. The $\alpha$ is here also dependent on the surrounding. Using these equations, one can quite accurately estimate the wind speed at the rotor hub height. Even though the wind passing through the area covered by the rotors also differs per height, the wind speed at the rotor hub is assumed to be a good average.

$$V(h) = V(h_{ref})\frac{ln(\frac{h}{z_0})}{ln(\frac{h_{ref}}{z_0})} \tag{4.2}$$

$$V(h) = V(h_{ref})(\frac{h}{h_{ref}})^{\alpha} \tag{4.3}$$

For both offshore and onshore wind power, wind turbines are modeled, according to the phenomena mentioned above. It is important to state that the phenomena mentioned first was already applied. Something similar to the second phenomena seemed to have been applied, however it was changed to what is described above. Appendices C.1 and C.2 contain parameters used to model these turbines. This consists of both parameters on the average turbine installed in that sector, as well as parameters that are inherent in that region. To match the (estimated) installed wind power capacity in the years 2021 and 2030, a multiple of these turbines is selected and used as a multiplication factor.

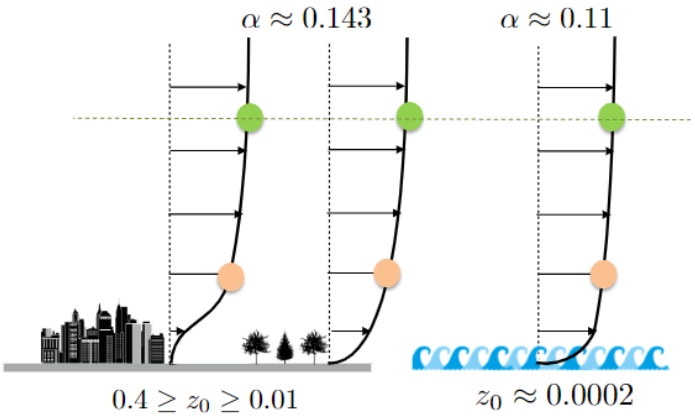


Figure 4.11: The relation between wind speed and height, showing different patterns for different surroundings.[53]

### 4.2.2 Solar PV

The entire solar PV panel model had already been created and no changes to the functionality of the model have been made. Below is a short description of how the model works as a black box, if one is more interested in the functionality of this model, it is advised to read the report of Raghav Saini[35].

Again, the model uses data to determine its power generation per 15 minutes. This data consists of solar data every 15 minutes, and model instance data. The solar data had also already been given, from the year 2012. It was estimated that, over a span of 9 and 18 years, the stance and effect of the sun would not change radically, besides the inherent randomness of the weather. Hence it was decided to also keep the solar data given. The data consists of solar intensity, the ambient temperature on the ground, the elevation, and the angle under which the sun stands. The higher the temperature of PV cells, the less efficient they become, hence the ambient temperature is of importance. [17] Also the angle of incidence plays a large role in how much energy is absorbed and what part is reflected back for example. Here an angle of incidence of zero degrees is optimal. The higher the solar intensity, the more energy can be won, but the cell gets heated up more, complicating the system even more. Raghav Saini has created a detailed model on how to handle these inputs. [35]

However, its case study was mainly focused on a few solar panels that would be located on the roofs of a few residents. This had to be scaled to the case study of the entire energy grid in 2030 instead of 2012, containing all roof PV panels, as well as solar PV parks. Here, the capacity installed rises, the average tilt of a panel is changed, as well as the power density and efficiency rise. [5] This model instance data is changed to fit the case study and with that, the model is considered complete.

In 2021, the installed capacity is said to be 14.911 GWp and the efficiency 18%. In 2030, this is estimated to be 26.76GWp and 20%, respectively. The average tilt estimated in both these years is 24.55 and the power density 151.52 $Wp/m^2$.[5]

## 4.3 Storage models

If over-generation, renewable sources generate more than is demanded, happens, it is energy-efficient to store this surplus, for later use. For this, two major energy storage systems are taken into account within the system: electrochemical storage in batteries and hydrogen. In order to create hydrogen electricity, an electrolyser plant has to be present in the system as well. To reverse the process, a fuel cell plant is a necessity. Therefore, these two components are also described in this section.

### 4.3.1 Battery storage

The battery storage is a model that already was present within the Illuminator project, for which the functionality was not changed. This model has model instance data, but the rest of the model only reacts to the system. This means no external data is needed to run the simulation. Here only some of the model instance data has been amended. The battery model works as follows; It has a state of charge (SoC). Whenever this SoC is above its minimum, the battery can be discharged. As long as it is below its maximum SoC the battery can be charged. This SoC, as well as the maximum and minimum, is continuously known by the controller and is scaled to the total amount of stored energy capacity installed. Whenever the controller sees a surplus or shortage, as long as the SoC permits, the controller can send or draw power to or from the battery. This however is also limited to the total power capacity. During charge and discharge, unfortunately, energy is lost, as both processes have an efficiency of 90%.[35]

Only the power capacity and storage capacity are changed within this model, as the model seemed to suffice for the way the system was implemented. For the year 2030, an estimation of 10GW for power capacity and 20GWh for storage capacity was estimated.[5] If no battery is attached during a simulation run, simply no power is sent or drawn from the battery. This power flow is indicated with 'flow2b'.

### 4.3.2 Hydrogen storage

The hydrogen storage works quite exactly the same as the battery model. The storage has a SoC, and a maximum and minimum for it to operate. Whenever the SoC meets the requirements, hydrogen can be stored or released from the hydrogen tank. This again comes at a price, as during both storage and release 10% hydrogen is lost.[35]

By 2028 one salt cavern is transformed into a hydrogen storage facility. By the end of 2030, three additional cavern storages will be realized. In total, these caverns can store up to 20 ktonnes of hydrogen. [23] [22] Originally, the storage was depicted in kWh, however, due to the implementation of hydrogen being sent to factories, it seemed more logical to depict it in kg stored. More about the hydrogen to factories possibility can be read in Section 4.4.

The flow of hydrogen into the tank is indicated with 'h2_gen' or 'h2_in'. The flow out of the tank is indicated as 'h2_out'. An important difference between these two is that the signal for the flow inward comes from the electrolyser, while the one for the flow outward comes from the controller itself.

### 4.3.3 Electrolyser

The electrolyser is a component creating hydrogen from electricity. This can be used when there is a surplus in electricity, to store the energy in the form of hydrogen. The controller determines the amount of electric flow that is sent to the electrolyser, for which the electrolyser calculates the amount of hydrogen formed. This takes into account the efficiency of 62.5%.[5] This is sent via the signal 'flow2e'. This formed hydrogen can then be sent to the hydrogen storage or the industry, as described in Section 3.5. The amount of hydrogen that is sent to the factories is given by the controller, in the signal 'h2tofa_e', in kg. The electrolyser subtracts this value from the amount of hydrogen formed due to the current 'flow2e', and sends this amount to the hydrogen storage, if present and not full. It is up to the controller, however, to make sure that if that is not the case, no surplus in hydrogen is produced.

The fact that the electrolyser calculates the amount of hydrogen that is to be stored is the reason this signal does not come from the controller. It could be possible that the controller would know that amount of hydrogen. However, this would undermine the entire functionality of creating different models for different components and functionalities. To avoid cyclic dependencies, however, the controller already knows the hydrogen mass corresponding to the share of electricity sent to the electrolysers that are meant for the industry. This is needed for the emission calculations.

If no fuel cell is attached, 'flow2e' is set to zero, and the power will most probably be dumped.

The architecture and electricity-to-hydrogen conversion were already present, however, the possibility of splitting the hydrogen production outflow to factories and storage is added within this project. Next to that, the estimated capacity of 3.5GW is implemented. [5]

### 4.3.4 Fuel cell

The fuel cell can be used to regenerate electricity using the hydrogen stored in the hydrogen storage. It gets the signal 'h2_given' from the hydrogen storage, indicating the flow of hydrogen. The amount of electricity is then calculated, taking the efficiency of 0.49% into account.[16] This electricity is then added to the total electricity grid flow.
Only the efficiency is updated as opposed to what was already present in the Illuminator project.

## 4.4 Controller model

The controller model is responsible for determining the energy flows between all models. It must take into account the total generation and total loads, as well as import and decide what to do with the energy. It also has to calculate the entire CO2 emission at each point in time.

There are multiple scenarios that could occur for which the controller has to handle its outputs. To start with the situation where there is more electricity needed from the loads than it is generated by renewable energy supplies. According to one of the requirements of the controller model, it is supposed to "use storage components to make sure that it is as sustainable or cheap as possible". As previously discussed, the measure of quality is GHG emissions. To better match this, it was decided to implement the sustainable way, instead of cost-effective. This means that using stored energy is preferred over generating additional energy using fossil plants. Hence, whenever there is a shortage, first it is filled with power from the battery storage. Whenever the battery is at its minimum SoC, or is limited to the power it can supply, the remainder is filled with fossil plant power. Hydrogen storage is not used in this scenario, but the reason why is described in the paragraphs below.

Whenever there is more generation than demand, including import and export, multiple scenarios arise. What to do with the residual power is the main question. Battery storage has a charge and discharge efficiency of 90%.[35] This way, when electricity is stored and reused in batteries, only 19% of energy is lost. Another way of storing energy available in this system is hydrogen. However, an electrolyser has an efficiency of 62.5%, which means 37.5% is lost.[5] Hence, to use the storage components as sustainable as possible, first the battery storage is maximally applied. Whenever the battery storage is fully charged, or the surplus of power is more than the capacity of the battery storage, the residual power is sent to the electrolyser. If then still the residual power is more than the electrolyser capacity, unfortunately, the electricity has to be dumped.

Then the question remains, what to do with the created hydrogen? A fuel cell is present within the entire system. If the created hydrogen gets stored, this stored hydrogen could be fed to the fuel cell to regenerate electricity whenever there is a renewable generation shortage. By doing this, one can reduce the amount of fossil fuel generation that is switched on, thus reducing the amount of emissions. Both while charging and discharging, just as with battery storage, energy is lost, however. The hydrogen has a charge and discharge efficiency of 90%, meaning 19% of the hydrogen energy is already lost when it reaches the fuel cell. [35] A typical fuel cell itself has an efficiency range of 45-49%. [16] This means that one kg hydrogen creates, if optimistically, $1\,kg \cdot 0.9 \cdot 0.9 \cdot 0.49 \cdot 39.4\,kWh/kg = 14.07\,kWh$. [35] Using the variable emission constant found in Section 3.3, the saved emissions can be determined for this implementation: $0.58915\,kgCO_2\text{-}eq/kWh \cdot 14.07\,kWh = 8.290 kgCO_2\text{-}eq$.

This is only 57.8% of what could be saved by sending 1kg of hydrogen directly to steel refineries, as shown in Section 3.5. In both cases, transport losses are neglected. As the steel industry also seems to outweigh every other sector, based on $CO_2$ emissions saved, it is decided to make this the primary

usage of the created hydrogen. TATA Steel plans to change a part of their blast furnaces to work fully on hydrogen. Their plans are to scale up to use $10PJ$ of hydrogen energy, which scales to 70.5 Mtonne, per year in the year 2030.[4]. Per 15 minutes, this is on average 2.012 tonnes of hydrogen that can be fed to TATA Steel. Per every kg of hydrogen delivered to the industry, 24kg of CO2 is subtracted from the total emissions. This seems a little incorrect, as the usage of hydrogen gas in this process does not use CO2 as well. However, since the GHG emissions in this project are scoped around the electricity grid, and hydrogen is a non-electric product that is delivered outside this scope, it is accounted for as negative emissions.

The expected planned electrolyser capacity in 2030 is $3.5GW$, leading to $875MWh$ per 15min. Using an electrolyser efficiency of 62.5%, and the hydrogen energy density of $39.4kWh/kg$, this comes down to 13.88 tonnes of hydrogen.[5] This means, that at full electrolyser capacity, there is a hydrogen overcapacity of 11.86 tonne. This overcapacity can then be stored in the hydrogen storage. Whenever not enough power can be delivered to the electrolysers, the stored hydrogen can step in to make sure as much hydrogen is fed to the steel industry as possible. With storing hydrogen, again 19% of energy is lost, but this is still not enough to make another hydrogen sector more expedient as an overcapacity target. In case, the hydrogen storage is full (or not attached), for now, hydrogen production is scaled down, to match only the demand from the industry. Sending it back directly to the fuelcell would make no sense, as there already is a surplus in electric energy for hydrogen to be formed. If residual energy is left, this will be ”dumped”.

A detailed version of the entire controller model and its decisions is schematically graphed in Appendix B.1. As previously mentioned, the decisions are all based on creating the most sustainable system, and costs are herein neglected.

As the controller knows the power created by the renewable energy generation sources, im-/export, and the loads, it was able to determine the necessity of the fossil electricity, along with the possibility to send hydrogen to the steel industry. Using the constants in Table 4.1, the controller can calculate the GHGs emitted at each time instance.

| Generation type | Constant emissions $[g\ CO_2\text{-}eq/kW/15min]$ | Variable emissions $[g\ CO_2\text{-}eq/kWh/15min]$ |
|---|---|---|
| Fossil plants | 0.305 | 985.1 |
| Offshore wind | 1.843 | - |
| Onshore wind | 0.955 | - |
| Solar PV | 11.156 | - |
| Import/Export | - | 418.65 |
| Steel industry* | - | -24000 |

Table 4.1: The amount of GHG emissions per energy sector that the controller uses to calculate the current emissions. *The steel industry is accounted for per kg hydrogen instead of kWh.

## 4.5 Import and export of electricity

As already mentioned in Section 3.4, in the year 2021, $20632GWh$ was imported and $20885GWh$ was exported. It was already discussed that the nature of why import and export happens is hard to pinpoint. This is due to the fact that it is partly influenced by the economic electricity market of both the Netherlands, as well as its neighboring countries. Therefore, creating a model that determines at each point what the import and export could be would be very difficult. Export could be said to be the surplus of green electricity generated. However, sometimes more electricity is generated via fossil plants for the sole purpose of export. It also asks to make a decision as to whether to store energy or sell it to neighboring countries.

As implementing this would be too far out of the scope of this project, it was determined to use previous, actual data. This way, import and export are implemented, within the right scale. Data is available about the import from and export to the grid-connected countries: Austria, Belgium, Denmark, Germany, Luxembourg, Norway, and the UK.[12] This data consists of import and export power per

hour per country connection. These are all added to form one net flow through the borders of the Netherlands. It was assumed that the power over the hour remained constant, and hence the 15-minute timestamps were filled with the hourly power statistics as well.

It is estimated that the import and export will rise by 1.5% in the year 2030 opposed to 2021.[5] Hence, for the im-/export model in 2030, the same model is used as per 2021, but only scaled with the before mentioned factor.

# Chapter 5

# GUI

The GUI is responsible for making the software user-friendly. As stated in the Introduction it was first very difficult to find out how to start the simulation and where the most important information stands that needs to be changed. So there was looked at what is necessary to create a new simulation. Based on those findings, the requirements were made for the GUI that are specified in Section 2.1. In Appendix F, figures of the interface can be seen.

First of all the GUI needs to let the user choose in which operation mode the simulation must run. This can be seen in Figure F.1. A choice can be made between mode one when the user has hardware attached and mode two when there is no hardware attached. The GUI is different for both modes so first the GUI of mode one will be explained.

After this there is chosen for mode 1, the second step pops on the screen. Now the user can choose which devices he wants to use in the simulation. For mode one the options that show up will be: a Wind model, PV model, Battery model, and Load model. This can be seen in Figure F.13. For those models, the hardware group has created components.[50] Both modes are based on the Dutch national electricity grid. This is done by making sure that the load is the sum of the three sectors, and for the hardware wind representation, only the offshore wind model is used. This was done with the idea that an additional hardware piece could eventually represent onshore wind as well. For both the battery model and PV model nothing has to change since those are already scaled at the national level.
After the user has chosen its devices the start date can be filled in (Figure F.4). There is a choice for either the year 2021 or the year 2030. When this is finished the number of hours the simulation needs to run can be filled in (Figure F.5). Due to the next part of the GUI and the slow computing power of one Raspberry PI, it takes a lot of time to run the program so it is advised to not run it for more than 24 hours otherwise it will take a lot of time to finish.
As the user has given all the inputs to start, the simulation will begin. During the simulation, it must be possible to see changes in input in the attached hardware. This is done with graphs that are created at every time step showing the old values plus the new values of the models. These graphs can be saved at the end of the simulation and stay empty if there is no device attached.
Then after the whole simulation has run the end results are again shown in graphs. There are two categories for the graphs. One is the generation, here the generation of the PV panels and the wind turbines is shown, as well as the electricity that is over/to less after subtracting the load and the electricity that is over after filling the battery. The second category is the load, battery SoC, and CO2 emissions of the system. This is all of the GUI for mode 1.

For mode two, there are some more models to choose from than with mode one. These models are an offshore wind model, onshore wind model, PV model, battery model, hydrogen storage model, electrolyzer model, fuel cell model, industry load, residential load model, and commercial load model. In Figure F.12 this is shown.
After this the same options for start date and amount of hours can be given and the simulation will start. When the simulation has finished, to prevent the user from being overloaded with information, the graphs of the simulation show up and are divided into four categories: generation, storage, loads, and

GHG emissions. For generation, the same graphs are shown as for mode one. For the storage, the state of charge for both hydrogen storage and battery storage is shown. For the loads, all the loads are shown and the total of the attached loads is shown in a graph. The GHG emissions at every time step are shown as well. All these graphs can be exported by pressing a button to prevent the results from going to waste.

To make sure the GUI can run properly the file 'Run everything.py' is created which will start the whole simulation.

# Chapter 6

# Prototype implementation and validation

## 6.1 Validation and debugging

In order to realize the models described in Chapter 4, Python code was written. These codes can be seen in Appendix E. To validate and debug these codes, input parameters were continuously changed. With these different input parameters, different cases arise. Within these cases, some functionalities can be expected to arise and it will immediately be shown if there were bugs present.
In Appendix D, multiple GUI outputs are shown. These outputs are from a few different cases that are simulated. These cases are taken to the extreme for demonstration purposes. The main debugging, however, has happened using the data sent to the Weight and Bias database. Here, a lot more detailed information was sent each run. To prevent information overload, not all of this is shown in the GUI. This is also because some data is only interesting for debugging, and not per se for the user.

In Appendix D.1, a simulation is shown that ran the predicted case as described by the Case Study group.[5] Figure D.21 shows the sum of all loads. Using simple addition of timestamps, one can easily determine this sum is correct. Figure D.22 shows, to start with, the generation sources. Here, one can see that the first of January was a cloudy, windy day. This results in an overall high generation, which suffers less from spontaneous high peaks, originating from PV. Besides the generation sources, the 'Electricity that is needed/over after Generation-loads' is shown. This indicates whether there is a surplus in generation or a shortage of renewable electricity. In the Python code, this is represented by the variable 'flow'. The figure would respectively be positive and negative in those cases. Again, simple addition at certain timestamps shows that the flow is the sum of both generation types and all the loads. This proves the fourth controller requirement is met.
The figure also shows the 'Electricity that cannot be used'. This is electricity that simply cannot be sent to the loads or possible energy storage, in the code 'dump'. This could happen if for example storage is full, or the surplus is above power capacity.
In Figure D.23, one can see that the latter is the case, as the hydrogen storage is not yet full when the electricity is 'dumped'. Lastly, the emissions are shown in Figure D.24. One can see that the first emission peak is slightly delayed due to the fact that the battery still gives power.

### 6.1.1 Testcase simulations

As previously mentioned, the entire functionality is hard to test and validate using one simulation. Therefore multiple simulation cases are used. In Section D.2, a simulation is shown where the power and storage capacity of the battery is severely increased. This has been done, in order to test the hierarchy between the two storage types.
In the case depicted above, both storages are nearly charging at the same time. Here it is uncertain whether this is due to the possibility that the maximum battery charging capacity is reached, or whether there is a bug in the system. Hence, increasing the charging capacity, such that all flow can be collected by the battery, this theory can be tested. Figure D.6 indeed shows that the hydrogen storage is filled only after the battery storage reaches its maximum SoC. Another thing that can be learned from

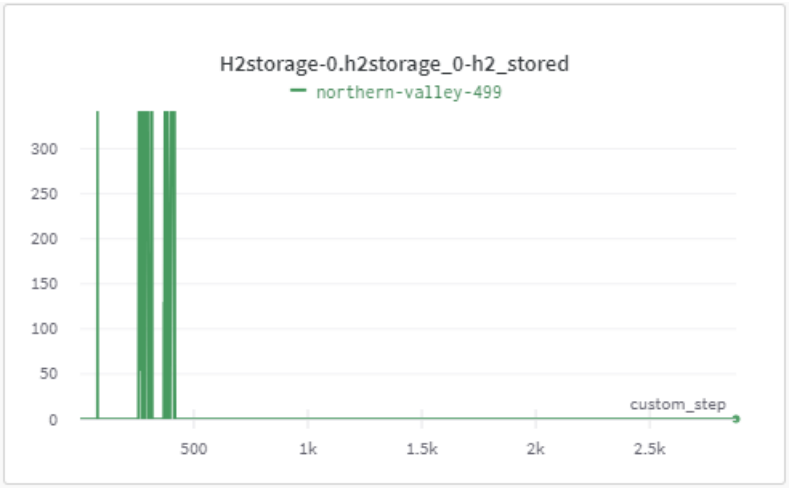


Figure 6.1: Graph from the Weight and Biases database. Here, the variable 'h2_stored' is shown in kg, as opposed to time, with timestamps of 15 minutes

this graph, is that the functionality that the hydrogen tank empties to send hydrogen to the factories works. This can be known for sure, as due to the fact that all the flow is collected by the battery, no electricity is sent to the electrolyser.

Appendix sections D.3 until D.7 show other interesting cases, that in their extreme show the effect of certain components or functionalities. Removing hydrogen facilities for example shows a small rise in the bottom line of emissions as opposed to having those facilities in the system. This shows the impact of sending hydrogen to the steel industry. The case where only the residential sector is shown, confirms the fact that, dependent on what generation sources or loads are connected, the flow changes. This proves the "and only if" part of the fourth controller requirement. Further extraction of information from these test cases are left as an exercise to the reader.

### 6.1.2 Long testruns

As can be seen, these shown test cases all are run for only five days. The main functionalities can be retrieved from these simulations. However, some extremities, that might lead to bugs, will most probably not show up in such a short time span. On the advice of P. Manganiello, some long-term simulations had been executed as well. Indeed this way some bugs appeared. Ironically enough, the example given in Figure 6.1, showed up, shortly after five days. The bug, which unfortunately can not be understood from this one sole image, happened the moment the hydrogen storage for the first time was fully filled. Even though this would be emptied, no hydrogen could be stored anymore.
Another example of bugs that were found by running longer moments, is the instance of winter time and summer time. By running the simulation for three months, it was noticed that in some time instances, timestamps would be skipped and repeated. This was due to the Dutch implementation of winter and summer time, where the clock is shifted an hour, twice a year. This crashed the simulation, hence it was a good thing long runs were executed, otherwise, it must have come across by pure chance, for it to be found.

## 6.2 Integration

To test the different modes of the software some runs are done with both the scaling to 2030 and the hardware components.

For the 2030 model, the code works as mentioned in Appendix D.1. It is only unfortunate that the loads for the residential and commercial sectors aren't scaled to the values of only electricity use but to total energy usage. But the rest of the system works appropriately.

For the integration with the hardware, the hardware group has made some functions that could be called, such that a value between zero and one is given for both the PV model and wind model which is scaled in the corresponding models to values of either the year 2021 or 2030. For the load and battery, the values given by the corresponding Python models the SoC value and the load value are passed through when the devices are connected.[50]. In the Python file hardwareattached these functions are stored. Testing this set up there was found that with some small changes in the code, the simulation would run in combination with the hardware which was the desired result.
Showing that this mode where live data is plotted works, is quite difficult to do via words and figures. A demonstration would be best suited for this.

# Chapter 7

# Discussion

The discussion section aims to interpret the results and critically analyze the way they are established.

First of all, as stated in Section 4.2, the input data used for the wind and solar models date from 2019 and 2012, respectively. Even though this in theory does not differ much in total energy output, it might have brought some unwanted discrepancies. The import and export, for which 2021 data is used, are assumed to partly depend on the generation by renewable energy sources. It might for example happen that in 2021 the first of September was a very sunny day and a lot of energy was exported due to a large surplus of solar PV power. If in 2012 the first of September was a very cloudy day, this surplus would not be present in the simulation. To match the electricity demand, a lot of fossil electricity would have to be generated. In reality, this did not happen, which leads to possibly false or misleading information. Hence, it is recommended to collect the same data for 2021 as well.//

Secondly, the extra load of the residential sector that comes from solar panels that lie on the rooftops of the houses can be modeled in a better way for the 15-minute scale stored data. This could be done by looking into how much electricity a household that has a PV system on its roof uses or gives back to the grid. This will probably give a smoother load profile, especially for the days when there is a lot of solar electricity available.

On the topic of the PV power data, this itself can be improved as well. The input data for the model now originates from one single measurement location. If one cloud would pass over this measurement location, the solar irradiance would drop immediately. This would result in very large drops and spikes for the entire national PV generation. Whilst in reality, not the entire solar PV network is covered by that single cloud. These large spikes and oscillations can be seen in most of the PV power outputs as well. By taking the data of multiple areas throughout the Netherlands, a more averaged and smoothed data input would be given to the model, representing the PV power generation probably a little bit better. The 15 min input data for the PV model is still based on only one location in the Netherlands. So it would be recommended to use more data points to even out the local variations.

The 15 min load profiles of 2030 can also be improved. This can be done by using old data from different years and comparing them to get a better insight into what the demand does in certain periods of time. Here the year 2020 could be used as an outlier case since the electricity load changed a lot due to the pandemic. As well as determining the residential and commercial electricity demand in 2030 would be quite interesting to see. Unfortunately, only the total energy demand was given by [5] or these sectors so this is now used in the model instead of only the electricity demand.

Regarding the GUI, the data sets shown are in no way fixed. For now, an assumption has been made about what would be educative and insightful to show the average user. Due to the GUI requirement not to create an information overload, decisions had to be made whether to show one thing or the other. In the requirements certain data sets were obligated, leaving little room for additional choices. This could mean that one thing that could be considered insightful by the user, did not make the cut. As these choices made were purely subjective opinions, no real foundation backs this choice up. This could be further developed by doing a customer evaluation survey, hopefully giving a real foundation

on what to picture and what not.

# Chapter 8

# Conclusions and Future Work

The main goal of this project was to create a simulation in Python that can run a national case of the Dutch electricity network.

## 8.1 Conclusions

An expansion to already existing simulation software has been produced for a table-top electricity demonstrator, the Illuminator. This integrates the case study performed on the Dutch national electricity grid and implements the option to connect physical hardware representations of certain models, as well as a GUI and measure of quality of the by the user proposed setup.
The integration of the case study entails under more the addition of models for industrial, residential and commercial sector loads. The main renewable generation sources are modeled as well, solar PV power, offshore and onshore wind.
The measure of quality for the configuration is set to be GHGs emitted. The controller model, managing the interconnection between all models, is thus set to function as sustainably as possible. The functionality of this controller model is extensively tested and debugged. Another feature, to reach a more sustainable system is added as well. That is the option to use formed or stored hydrogen, which originally was used as feed for a fuel cell, in industrial sectors. This appears to save more emissions than the fuel cell setup. Also import and export of electricity is taken into account.
The GUI provides an easier, more user-friendly way to start the simulation, than it was before. First mode selection is asked, then input for simulation configurations, start date and duration is asked and the simulation can be run. Afterwards, all necessary data is shown, divided in selectable subplots.
The created models appeared not as robust as wanted at first, but they function and give a good representation. Next to that, the requirements for each sub-component of the project is met. Hence it can be stated this project has been executed well. Besides that, the different cases mentioned in Chapter 6, were mainly used for debugging purposes, as explained. However, these extreme cases already show not only the correctness of controller functionalities, but also the effect a component can have by removing or adding it. Providing such information to the user is the eventual goal of this project, so it can be concluded that the project has reached its final goal.

## 8.2 Future work

As only the basis is created for the national grid level case study there are still a couple of things that could be improved and done to make it closer to reality.

First of all, it could be interesting to measure the quality of the setup with more than one factor. A cost analysis executed in a similar way as the co2 analysis would give a lot of insight. It could answer for example the following questions: what does this very sustainable configuration cost, would it be worth the small increase in sustainability? At what point has this PV installation paid itself back? Additionally, the controller is now set to be optimized for sustainability. Adding a cost-effective controller to

match the cost analysis measure of quality as well, would be very much suited.

Another thing that can be added is a load profile for the transport sector. In the year 2021, there are over 5.5 million cars that are at least partly electric driven in Europe, and this number is expected to rise in the year 2030. [47] Having to charge all these cars can have a large impact on the electricity grid. Using information such as driven distance per trip, and arrival time of those trips, as shown in [8], one can make an assumption on the load profile of the BEV model. Taking this a step further, to prevent grid congestion due to the large number of BEVs expected to charge at the same time, charging scheduling algorithms can be implemented, making use of grid-to-vehicle and vehicle-to-grid charging. This would essentially mean BEVs can also be used as battery storage for the grid. Such algorithms can be found in [25], [14], [13] and [9].

To make the Illuminator kit even more insightful cases between the one-house scenario or full national level could be added. For example, one county, one village, or one neighborhood all have their own challenges that need to be figured out in the coming years.

# Bibliography


[1] [Online; accessed 12. Jun. 2023]. Feb. 2016. URL: https://ocw.tudelft.nl/wp-content/uploads/Solar-Cells-R2-CH2_Solar_radiation.pdf.

[2] [Online; accessed 11. Jun. 2023]. Sept. 2022. URL: https://files.sma.de/downloads/Perfratio-TI-en-11.pdf.

[3] [Online; accessed 13. Jun. 2023]. Mar. 2022. URL: https://cedelft.eu/wp-content/uploads/sites/2/2022/03/CE_Delft_210426_50_percent_green_hydrogen_for_Dutch_industry_FINAL.pdf.

[4] [Online; accessed 13. Jun. 2023]. Mar. 2022. URL: https://cedelft.eu/wp-content/uploads/sites/2/2022/03/CE_Delft_210426_50_percent_green_hydrogen_for_Dutch_industry_FINAL.pdf.

[5] A. Akaouche and M. Hameed. *Navigating the Energy Transition: A Comprehensive Modeling Approach for the Netherlands*. June 2023.

[6] E.A. Alsema and Mariska de Wild-Scholten. "Environmental Impact of Crystalline Silicon Photovoltaic Module Production". In: *Materials Research Society Symposium Proceedings* 895 (Jan. 2011). DOI: 10.1557/PROC-0895-G03-05.

[7] Thomas Koch Blank and Partrick Molly. "Hydrogen's Decarbonization Impact for Industry: Near-term challenges and long-term potential". In: (Jan. 2020). URL: https://rmi.org/wp-content/uploads/2020/01/hydrogen_insight_brief.pdf.

[8] Sourav Das, Parimal Acharjee, and Aniruddha Bhattacharya. "Charging Scheduling of Electric Vehicle incorporating Grid-to-Vehicle (G2V) and Vehicle-to-Grid (V2G) technology in Smart-Grid". In: *2020 IEEE International Conference on Power Electronics, Smart Grid and Renewable Energy (PESGRE2020)*. IEEE, Jan. 2020, pp. 1–6. DOI: 10.1109/PESGRE45664.2020.9070489.

[9] Sourav Das, Parimal Acharjee, and Aniruddha Bhattacharya. "Charging Scheduling of Electric Vehicle Incorporating Grid-to-Vehicle and Vehicle-to-Grid Technology Considering in Smart Grid". In: *IEEE Transactions on Industry Applications* 57.2 (2021), pp. 1688–1702. DOI: 10.1109/TIA.2020.3041808.

[10] *Data View*. [Online; accessed 13. Jun. 2023]. June 2023. URL: https://transparency.entsoe.eu/load-domain/r2/totalLoadR2/show?name=&defaultValue=true&viewType=TABLE&areaType=BZN&atch=false&dateTime.dateTime=01.01.2021+00:00%5C%7CCET%5C%7CDAY&biddingZone.values=CTY%5C%7C10YNL----------L!BZN%5C%7C10YNL----------L&dateTime.timezone=CET_CEST&dateTime.timezone_input=CET+(UTC+1)+/+CEST+(UTC+2).

[11] *Data view*. [Online; accessed 11. Jun. 2023]. June 2023. URL: https://transparency.entsoe.eu/generation/r2/installedCapacityPerProductionUnit/show?name=&defaultValue=true&viewType=TABLE&areaType=BZN&atch=false&dateTime.dateTime=01.01.2021+00:00%5C%7CUTC%5C%7CYEAR&area.values=CTY%5C%7C10YNL----------L!BZN%5C%7C10YNL----------L&productionType.values=B01&productionType.values=B02&productionType.values=B03&productionType.values=B04&productionType.values=B05&productionType.values=B06&productionType.values=B07&productionType.values=B08&productionType.values=B09&productionType.values=B10&productionType.values=B11&productionType.values=B12&productionType.values=B13&productionType.values=B14&productionType.values=B20&productionType.values=B15&productionType.

values=B16&productionType.values=B17&productionType.values=B18&productionType.values=B19&DataTables_Table_0_length=50.

[12] *Data view*. [Online; accessed 13. Jun. 2023]. June 2023. URL: https://transparency.entsoe.eu/transmission-domain/physicalFlow/show.

[13] *Electric vehicle smart charging and vehicle-to-grid operation*. [Online; accessed 16. Jun. 2023]. June 2023. DOI: 10.1080/17445760.2012.663757.

[14] *Electric Vehicle–Smart Grid Integration: Load Modeling, Scheduling, and Cyber Security - ProQuest*. [Online; accessed 16. Jun. 2023]. June 2023. URL: https://www.proquest.com/openview/caeb0a3cea42f614a6d0c644f2f30a99/1?cbl=18750&diss=y&pq-origsite=gscholar&parentSessionId=nL9C9EYmtHq4UryrQf7eSp9RcDW9%2FNOx2a%2FCWBK7WN0%3D.

[15] Enexis. *Kleinverbruiksgegevens 2021*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://www.enexis.nl/over-ons/open-data#datasets.

[16] M. Farooque and H. Maru. “FUEL CELLS – MOLTEN CARBONATE FUEL CELLS | Full-scale Prototypes”. In: *Encyclopedia of Electrochemical Power Sources*. Walthm, MA, USA: Elsevier, Jan. 2009, pp. 508–518. ISBN: 978-0-444-52745-5. DOI: 10.1016/B978-044452745-5.00269-0.

[17] Stuart Fox. *How Does Heat Affect Solar Panel Efficiencies?* [Online; accessed 14. Jun. 2023]. May 2022. URL: https://www.greentechrenewables.com/article/how-does-heat-affect-solar-panel-efficiencies.

[18] Aihui Fu et al. “The Illuminator: An Open Source Energy System Integration Development Kit”. In: *PowerTech 2023*. IEEE Power and Energy Society. Belgrade, Serbia, June 2023, p. 5. DOI: DOI. URL: URL.

[19] Hans Christian Gils. “Balancing of intermittent renewable power generation by demand response and thermal energy storage”. PhD thesis. 2015. URL: https://elib.uni-stuttgart.de/handle/11682/6905.

[20] Hiroki Hondo. “Life cycle GHG emission analysis of power generation systems: Japanese case”. In: *Energy* 30.11 (Aug. 2005), pp. 2042–2056. ISSN: 0360-5442. DOI: 10.1016/j.energy.2004.07.020.

[21] *How Do Wind Turbines Survive Severe Storms?* [Online; accessed 12. Jun. 2023]. June 2023. URL: https://www.energy.gov/eere/articles/how-do-wind-turbines-survive-severe-storms.

[22] *HyStock*. [Online; accessed 14. Jun. 2023]. June 2023. URL: https://www.hystock.nl/en.

[23] *HyStock hydrogen storage*. [Online; accessed 14. Jun. 2023]. June 2023. URL: https://www.gasunie.nl/en/projects/hystock-hydrogen-storage.

[24] Westland infra. *Open data 2021*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://www.westlandinfra.nl/over-westland-infra/duurzaamheid-innovaties/open-data.

[25] Linni Jian et al. “Optimal scheduling for vehicle-to-grid operation with stochastic connection of plug-in electric vehicles to smart grid”. In: *Appl. Energy* 146 (May 2015), pp. 150–161. ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2015.02.030.

[26] Liander. *1 januari 2021*. [Online; accessed 11. Jun. 2023]. June 2023. URL: https://www.liander.nl/partners/datadiensten/open-data/data.

[27] Coteq Netbeheer. *kleinverbruiksgegevens 2021*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://coteqnetbeheer.nl/open-data.

[28] *NSRDB*. [Online; accessed 13. Jun. 2023]. May 2023. URL: https://nsrdb.nrel.gov/data-viewer.

[29] Oar. “Understanding Global Warming Potentials”. In: *US EPA* (Apr. 2023). URL: https://www.epa.gov/ghgemissions/understanding-global-warming-potentials.

[30] Naser A. Odeh and Timothy T. Cockerill. “Life cycle analysis of UK coal fired power plants”. In: *Energy Conversion and Management* 49.2 (2008), pp. 212–220. ISSN: 0196-8904. DOI: https://doi.org/10.1016/j.enconman.2007.06.014. URL: https://www.sciencedirect.com/science/article/pii/S0196890407001756.

[31] Matthew Ozoemena, Wai M. Cheung, and Reaz Hasan. “Comparative LCA of technology improvement opportunities for a 1.5-MW wind turbine in the context of an onshore wind farm”. In: *Clean Technol. Environ. Policy* 20.1 (Jan. 2018), pp. 173–190. ISSN: 1618-9558. DOI: 10.1007/s10098-017-1466-2.

[32] Hanne Lerche Raadal et al. “GHG emissions and energy performance of offshore wind power”. In: *Renewable Energy* 66 (2014), pp. 314–324. ISSN: 0960-1481. DOI: https://doi.org/10.1016/j.renene.2013.11.075. URL: https://www.sciencedirect.com/science/article/pii/S0960148113006654.

[33] N.V. RENDO. *NV-RENDO$_k leinverbruiksgegevens_0 1-01-2021$*. [Online; accessed 12. Jun. 2023]. Jan. 2023. URL: https://www.rendonetwerken.nl/disclaimer-open-data.

[34] Frans Rooijers et al. *Elektrificatie en vraagprofiel 2030 - CE Delft*. [Online; accessed 15. Jun. 2023]. Feb. 2023. URL: https://ce.nl/publicaties/elektrificatie-en-vraagprofiel-2030.

[35] Raghav Saini. *The Illuminator Energy System Development Kit*. Aug. 2022.

[36] E. Santoyo-Castelazo, H. Gujba, and A. Azapagic. “Life cycle assessment of electricity generation in Mexico”. In: *Energy* 36.3 (2011), pp. 1488–1499. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2011.01.018. URL: https://www.sciencedirect.com/science/article/pii/S0360544211000193.

[37] *Soorten aansluitingen | Liander*. [Online; accessed 9. Jun. 2023]. June 2023. URL: https://www.liander.nl/consument/aansluitingen/soorten-aansluitingen.

[38] Bent Sørensen. “Chapter 11 - Environmental Issues Associated with Solar Electric and Thermal Systems with Storage”. In: *Solar Energy Storage*. Ed. by Bent Sørensen. Boston: Academic Press, 2015, pp. 247–271. ISBN: 978-0-12-409540-3. DOI: https://doi.org/10.1016/B978-0-12-409540-3.00011-6. URL: https://www.sciencedirect.com/science/article/pii/B9780124095403000116.

[39] Christian Sparborth et al. *Nationaal solar trendrapport 2023*. n.a. 2023.

[40] Kerr D.R. Spath P.L. Mann M.K. “Life cycle assessment of coal-fired power production.” In: (1999).

[41] *StatLine - Elektriciteitsbalans; aanbod en verbruik*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://opendata.cbs.nl/statline/#/CBS/nl/dataset/84575NED/table?ts=1686218045505.

[42] *StatLine - Energiebalans; aanbod en verbruik, sector*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://opendata.cbs.nl/statline/#/CBS/nl/dataset/83989NED/table?ts=1686561298626.

[43] Stedin. *Verbruiksdata 2021*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://www.stedin.net/zakelijk/open-data/verbruiksgegevens.

[44] Arifa Tanveer et al. “Do Perceived Risk, Perception of Self-Efficacy, and Openness to Technology Matter for Solar PV Adoption? An Application of the Extended Theory of Planned Behavior”. In: *Energies* 14.16 (Aug. 2021), p. 5008. ISSN: 1996-1073. DOI: 10.3390/en14165008.

[45] *The Netherlands*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://iea-wind.org/about-iea-wind-tcp/members/the-netherlands.

[46] *The Netherlands - Countries & Regions - IEA*. [Online; accessed 11. Jun. 2023]. June 2023. URL: https://www.iea.org/countries/the-netherlands.

[47] *Trends in electric light-duty vehicles – Global EV Outlook 2023 – Analysis - IEA*. [Online; accessed 16. Jun. 2023]. June 2023. URL: https://www.iea.org/reports/global-ev-outlook-2023/trends-in-electric-light-duty-vehicles.

[48] R.A. van den Wijngaart, K. Blok, and A.J.M. van Wijk. “Sector analysis of the Dutch electricity load shape”. In: *Energy* 19.6 (1994), pp. 693–706. ISSN: 0360-5442. DOI: https://doi.org/10.1016/0360-5442(94)90008-6. URL: https://www.sciencedirect.com/science/article/pii/0360544294900086.

[49] *Verbruiksprofielen elektriciteit en aardgas 2023 gepubliceerd - MFFBAS*. [Online; accessed 8. Jun. 2023]. June 2023. URL: https://www.mffbas.nl/nieuws/verbruiksprofielen-elektriciteit-en-aardgas-2023-gepubliceerd.

[50] J. de Waal and N. Rauf. *Illuminator Hardware*. June 2023.

[51] Mariska de Wild-Scholten and E.A. Alsema. “Environmental Life Cycle Inventory of Crystalline Silicon Photovoltaic Module Production”. In: *MRS Proceedings* 895 (Dec. 2005). DOI: 10.1557/PROC-0895-G03-04.

[52] *WindTurbines - AE3516A Fundamentals of Wind Energy I (2022/23 Q1)*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://brightspace.tudelft.nl/d2l/le/content/498723/viewContent/3004981/View.

[53] *WindWaveResource - AE3516A Fundamentals of Wind Energy I (2022/23 Q1)*. [Online; accessed 12. Jun. 2023]. June 2023. URL: https://brightspace.tudelft.nl/d2l/le/content/498723/viewContent/2997220/View.

# Appendices

# Appendix A

# System

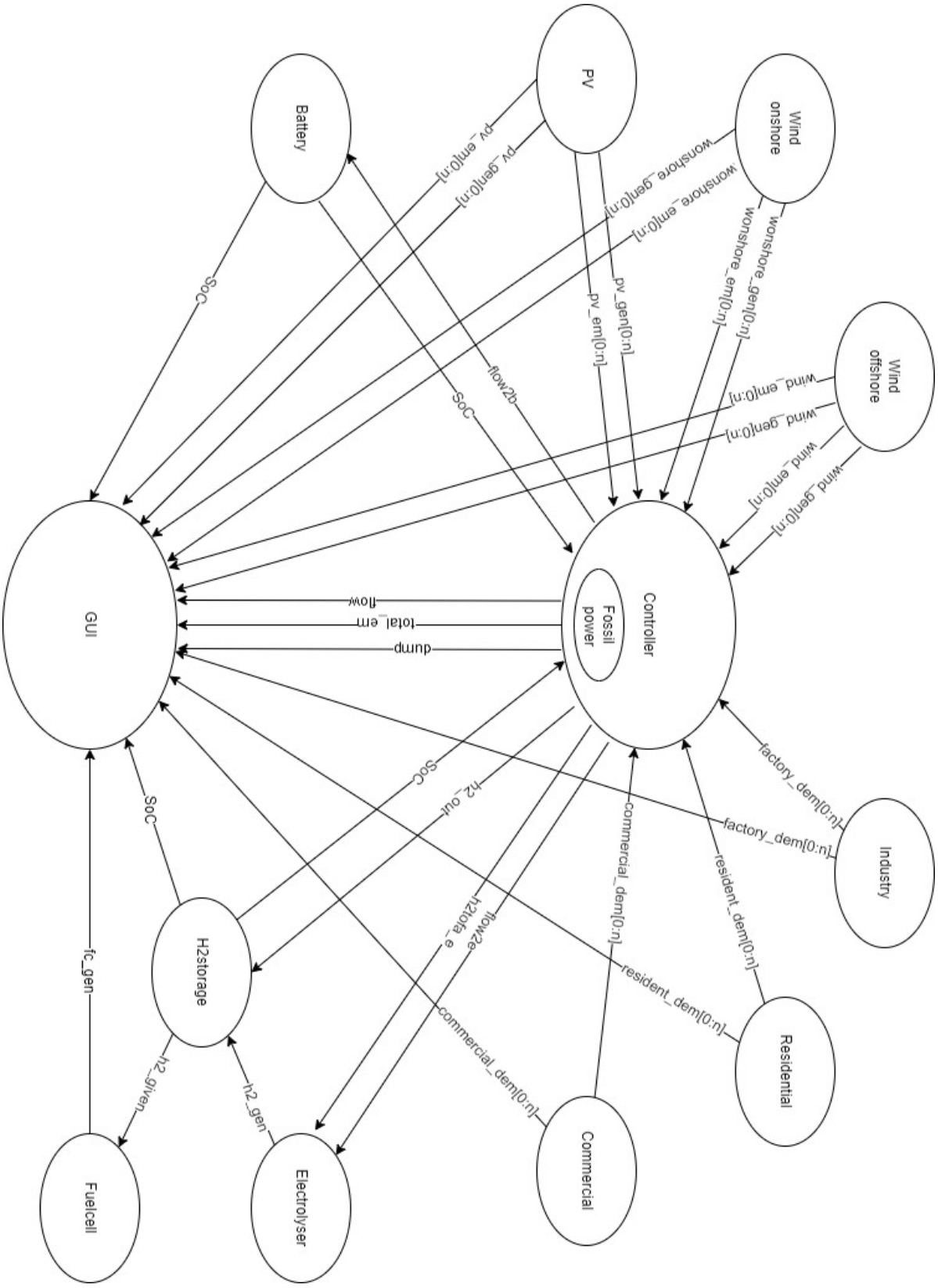


Figure A.1: Schematic representation of the interaction between the models on signal level. Fossil power is not a model in its own, but has an important presence within the controller, hence a pseudomodel is represented. '_dem' stands for demand, '_em' for emissions and '_gen' for generation.

Appendix B

# Controller

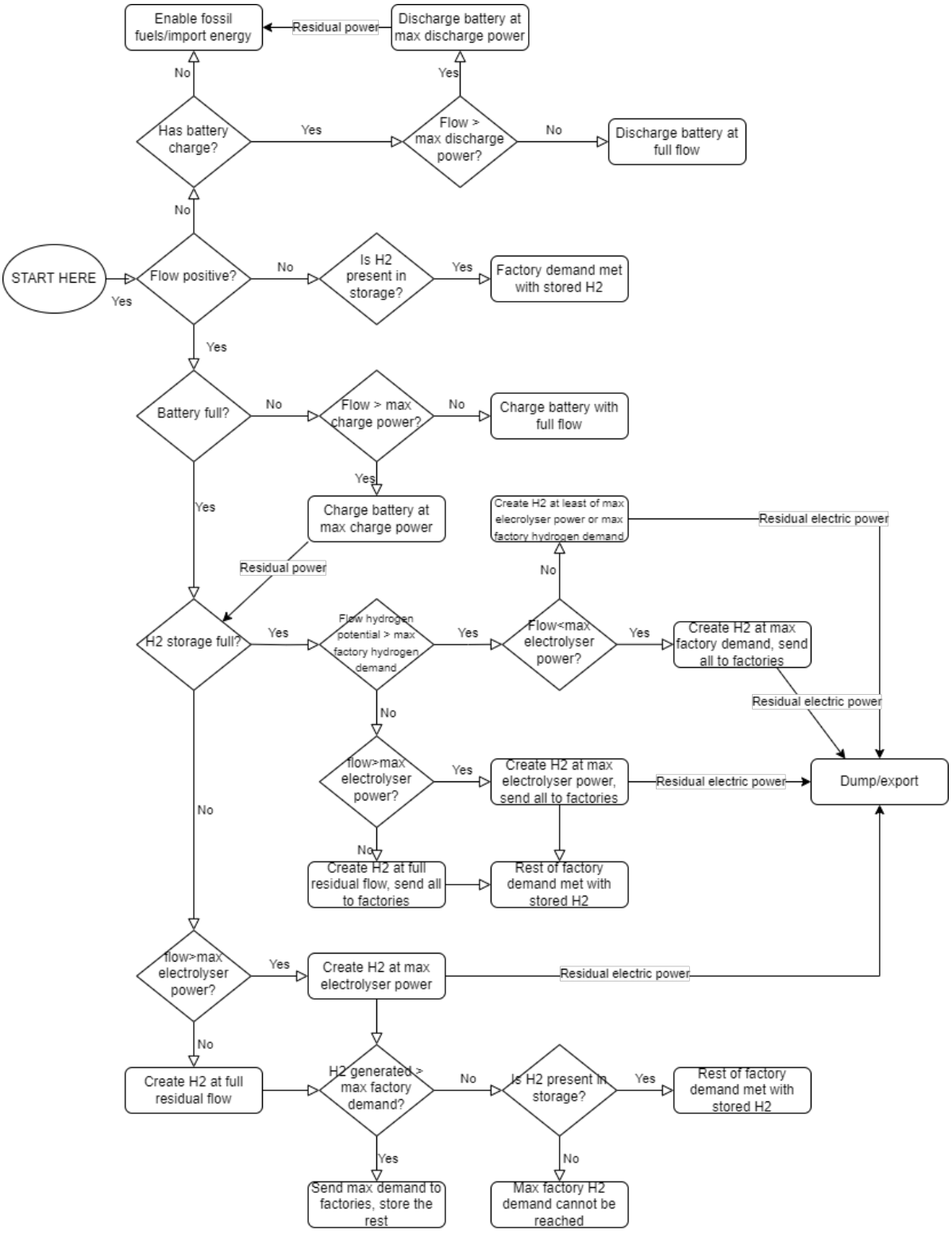


Figure B.1: Decision flow diagram for the controller.

# Appendix C

# Models

| Turbine parameters | | Situation parameters | |
|---|---|---|---|
| Rated power [kW] | 6000 | z0 | 0.0002 |
| Diameter (d) [m] | 154 | Air density ($\rho$) | 1.225 |
| Cut-in windspeed [m/s] | 4 | Alpha | 0.11 |
| Cut-out windspeed [m/s] | 25 | | |
| Efficiency ($\eta$) | 0.4 | | |
| Hub height [m] | 136 | | |
| Rated windspeed [m/s] | 10.955 | | |

Table C.1: Input parameters for offshore wind energy model [53][5]

| Turbine parameters | | Situation parameters | |
|---|---|---|---|
| Rated power [kW] | 2200 | z0 | 0.2 |
| Diameter (d) [m] | 154 | Air density ($\rho$) | 1.225 |
| Cut-in windspeed [m/s] | 3 | Alpha | 0.143 |
| Cut-out windspeed [m/s] | 25 | | |
| Efficiency ($\eta$) | 0.4 | | |
| Hub height [m] | 136 | | |
| Rated windspeed [m/s] | 11.536 | | |

Table C.2: Input parameters for onshore wind energy model [53][5]

# Appendix D

# Simulations

For these simulations, unless specified otherwise, the first 5 days of the year 2030 are shown. Multiple cases are depicted below.

## D.1 Predicted 2030 case

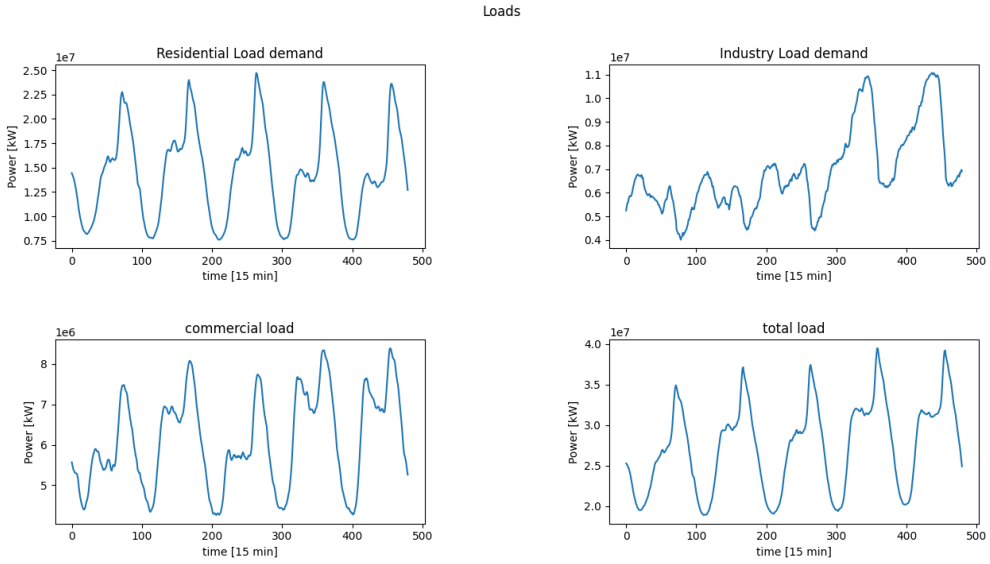


Figure D.1: The load profiles of the different sectors and their sum

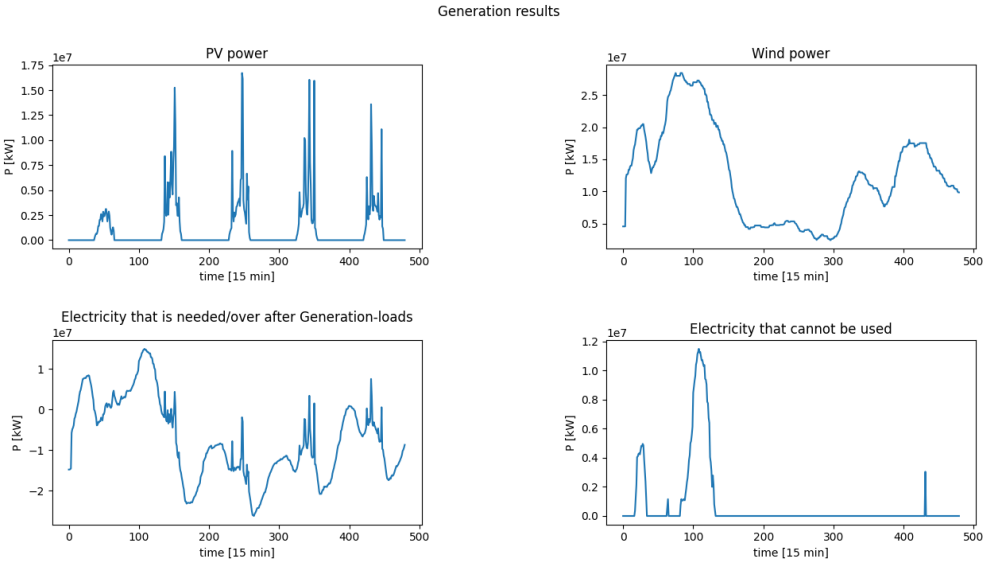


Figure D.2: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

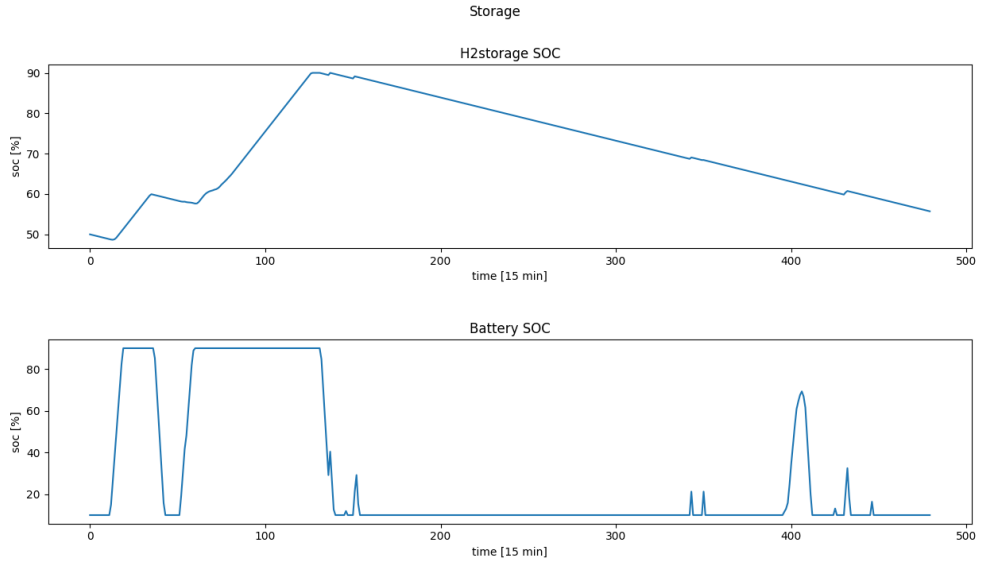


Figure D.3: The state of charge of both the battery and hydrogen storage

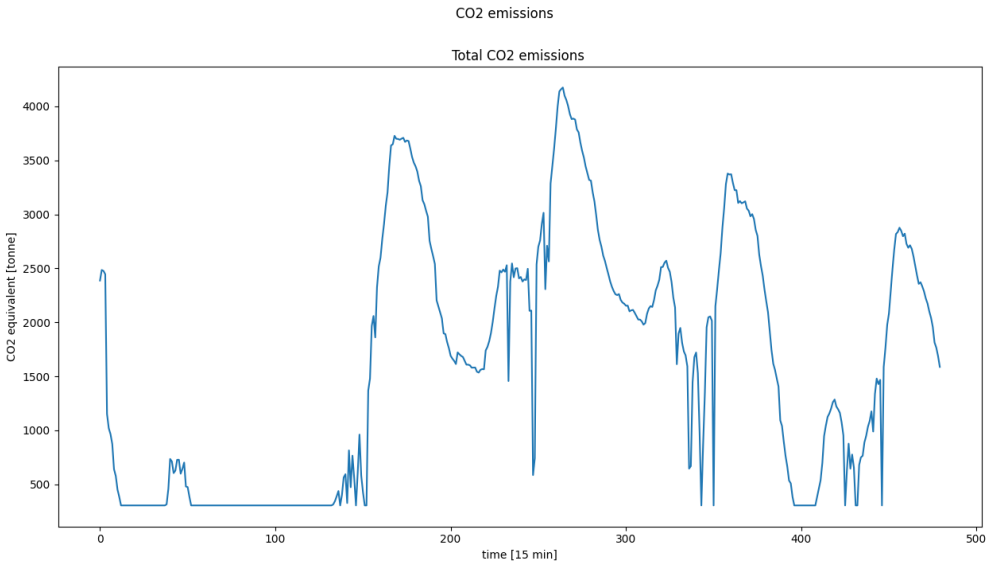


Figure D.4: The GHGs emitted at that moment

## D.2 Larger battery

This simulation, the battery power capacity has been increased by a factor 100, and the storage capacity has increase by 25. The effects are shown below. The loads remained the same.

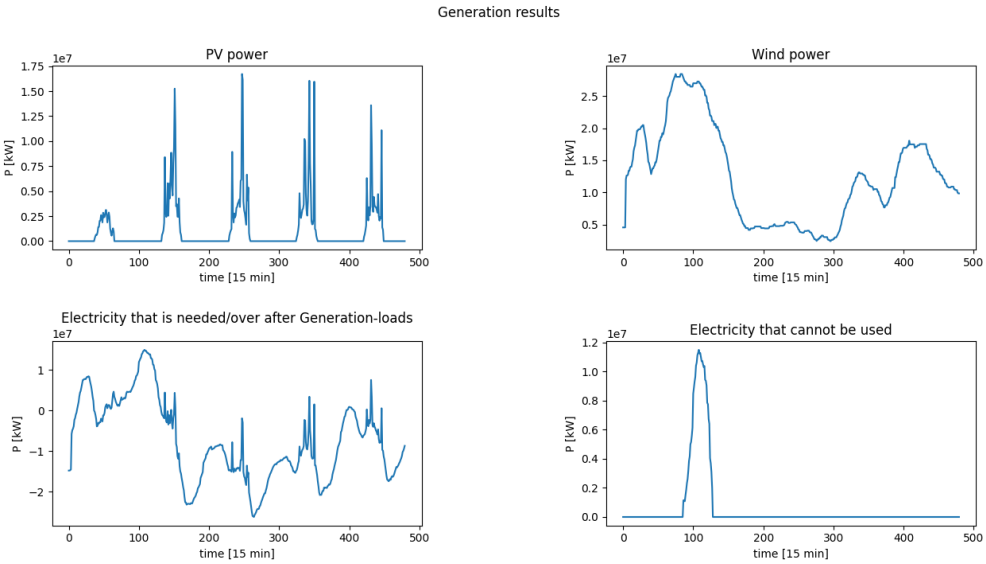


Figure D.5: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

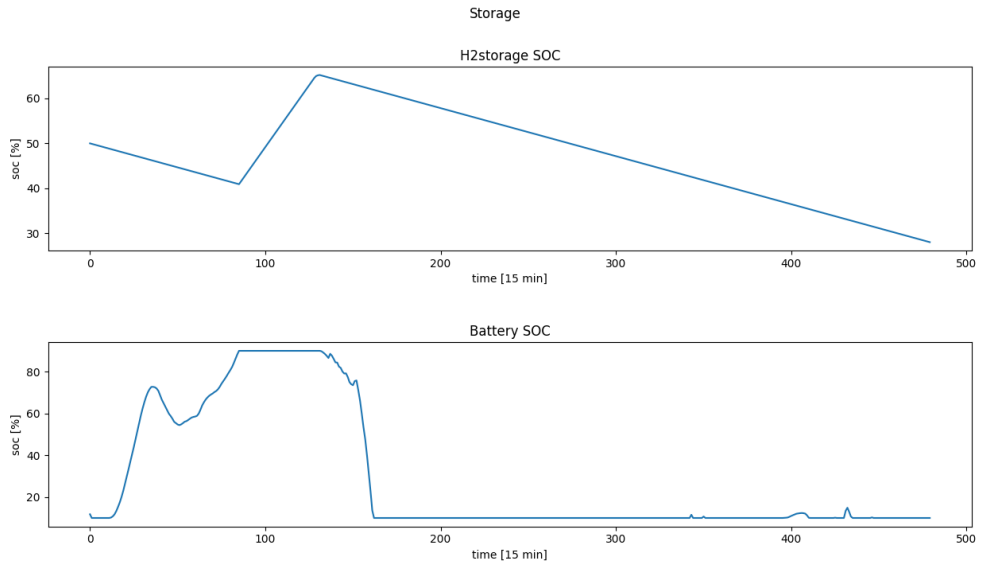


Figure D.6: The state of charge of both the battery and hydrogen storage

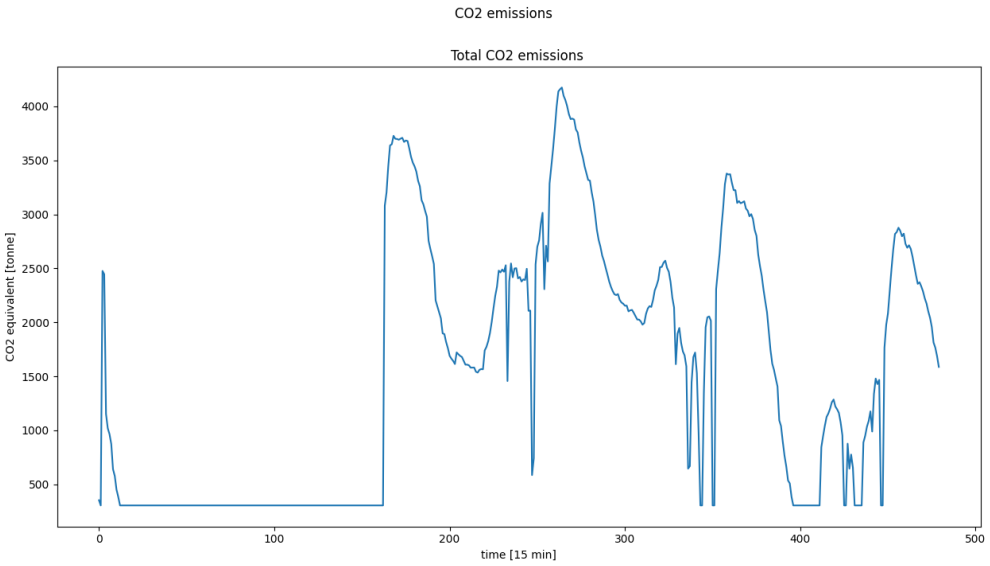


Figure D.7: The GHGs emitted at that moment

## D.3 No battery

In this simulation, the battery is not attached.

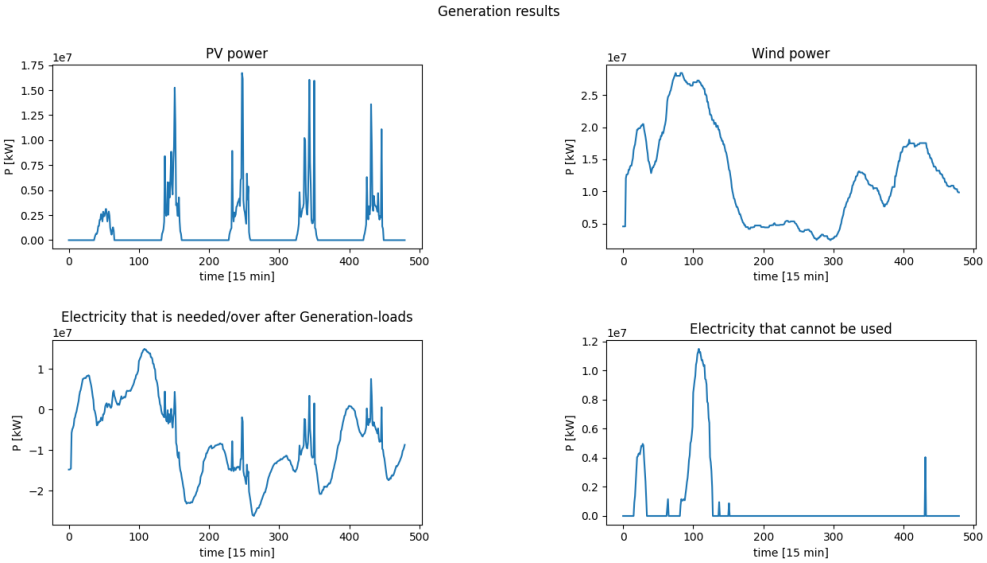


Figure D.8: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

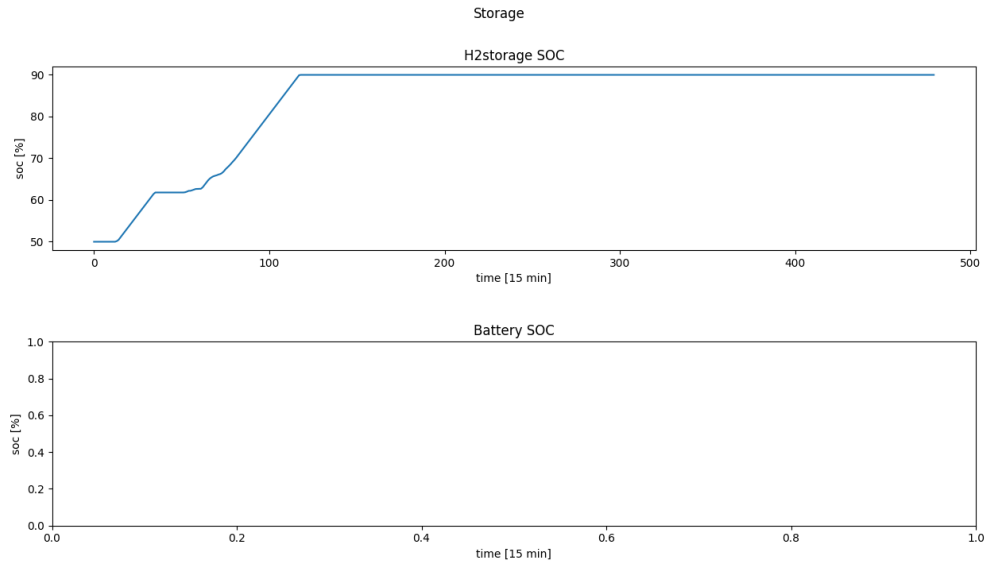


Figure D.9: The state of charge of both the battery and hydrogen storage

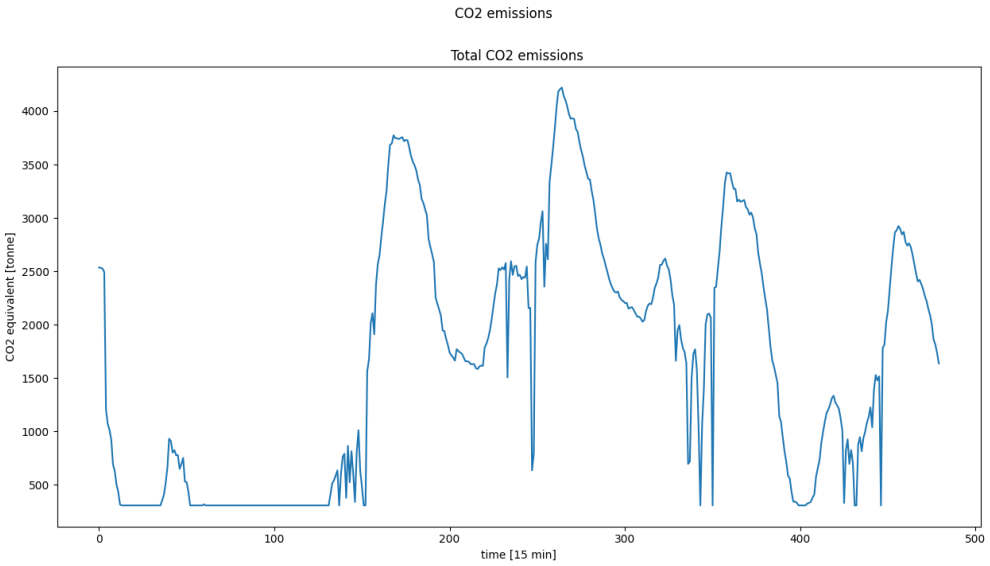


Figure D.10: The GHGs emitted at that moment

## D.4 No hydrogen

In this simulation, the entire hydrogen structure is not attached.

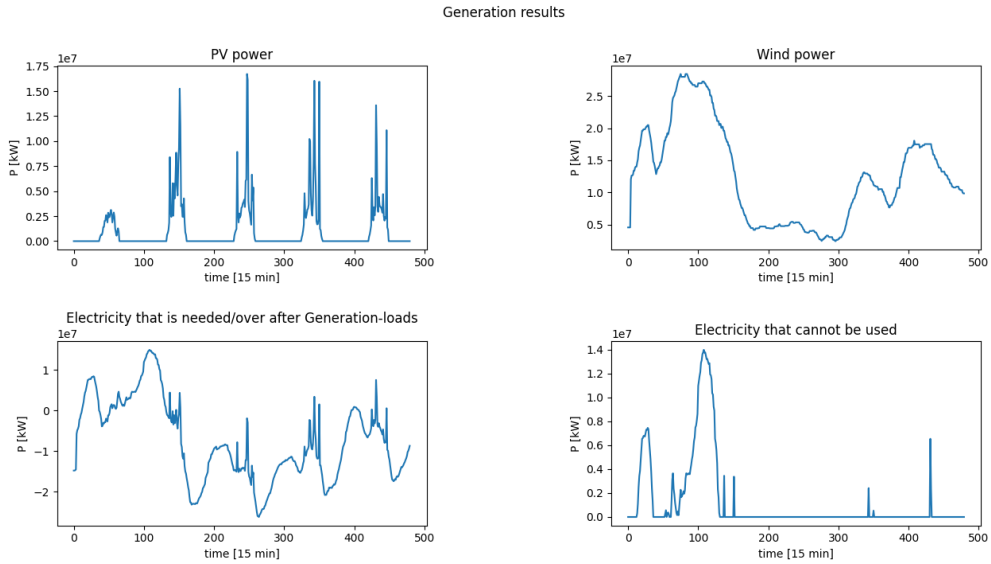


Figure D.11: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

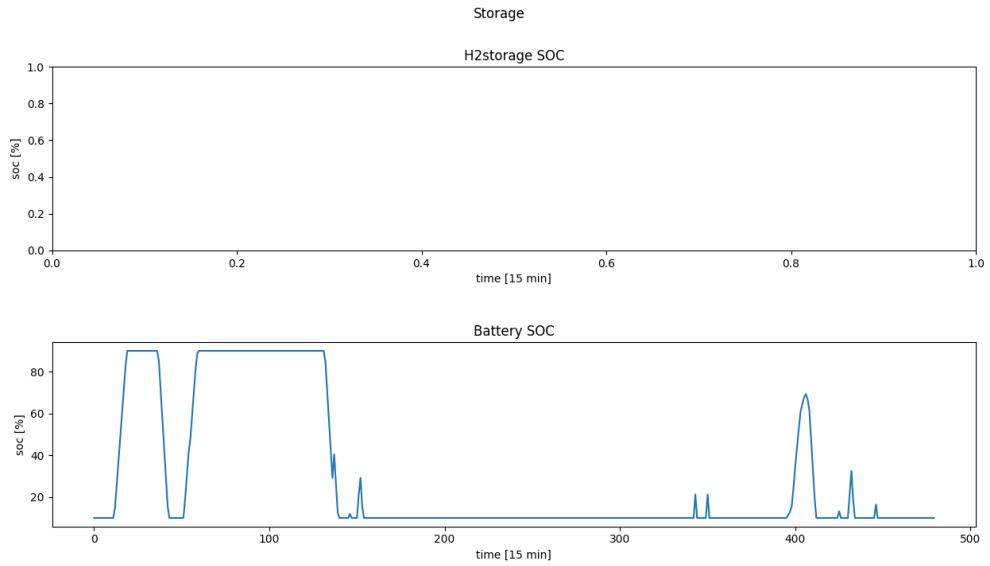


Figure D.12: The state of charge of both the battery and hydrogen storage

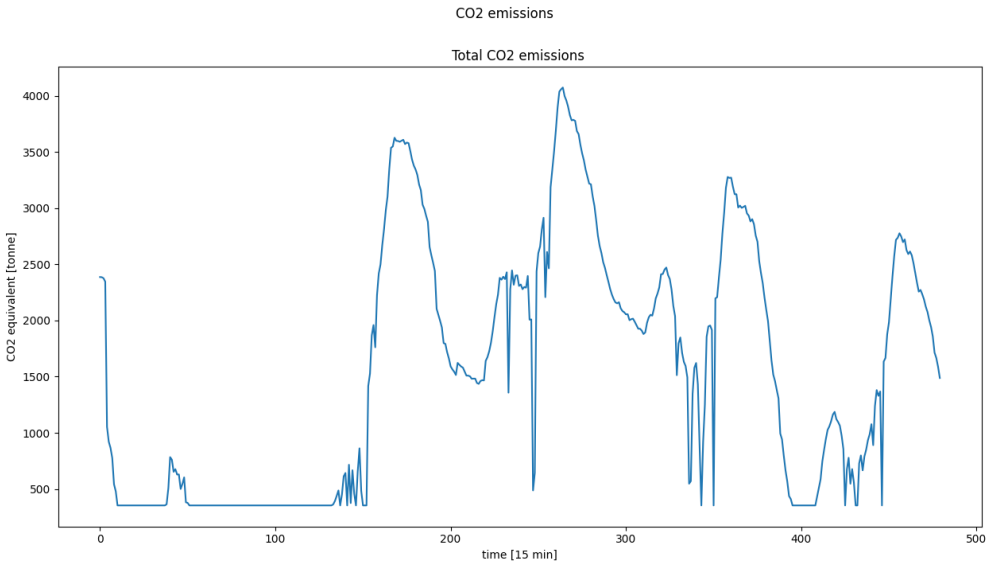


Figure D.13: The GHGs emitted at that moment

## D.5 No import/export

In this simulation, the import and export of energy is neglected.

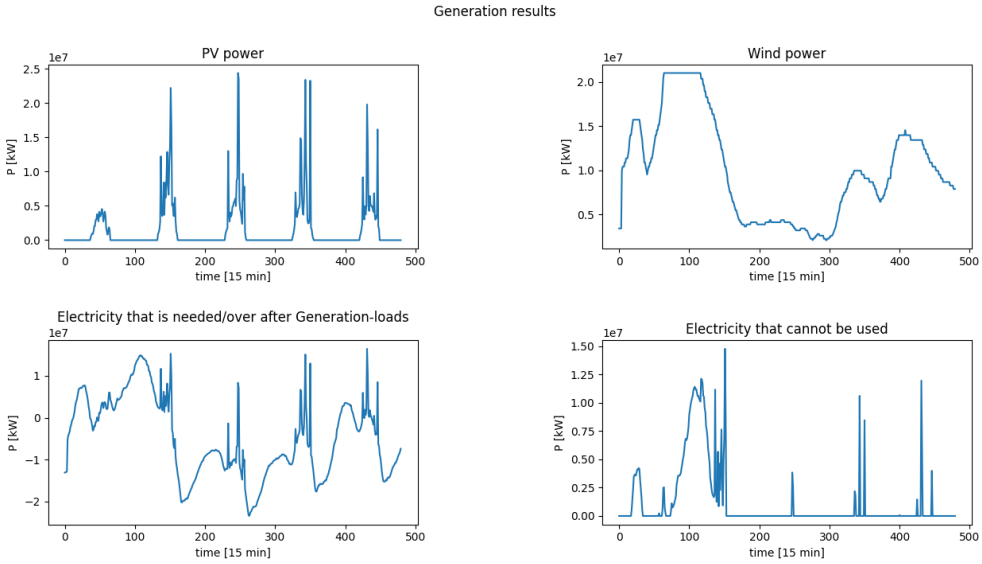


Figure D.14: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

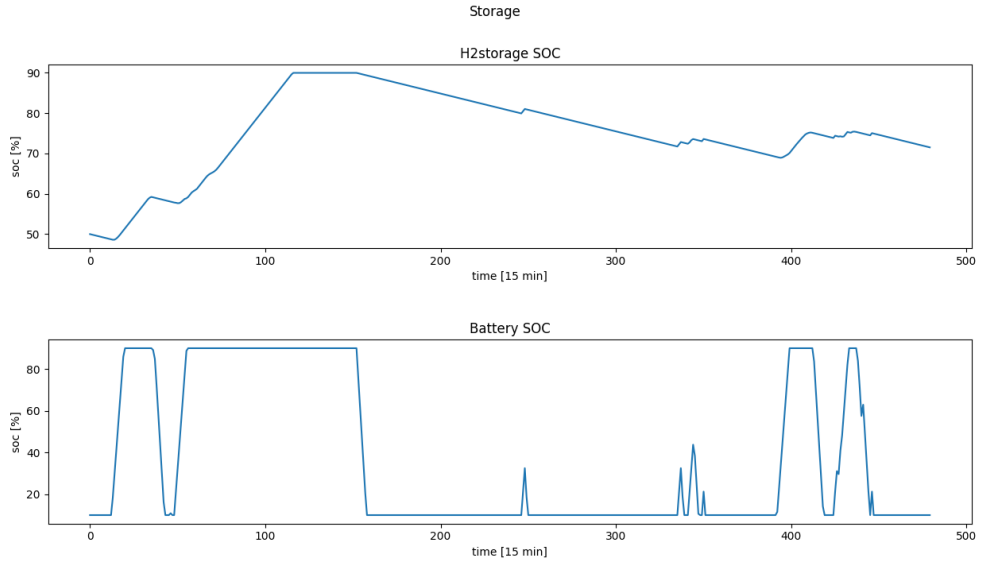


Figure D.15: The state of charge of both the battery and hydrogen storage

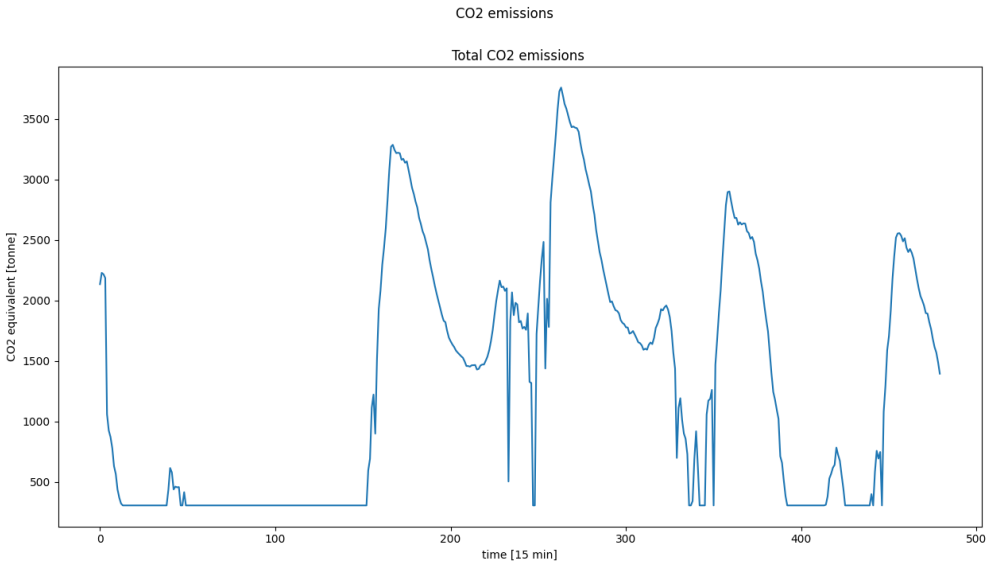


Figure D.16: The GHGs emitted at that moment

## D.6 Only residential load

In this simulation, only the residential load is selected.

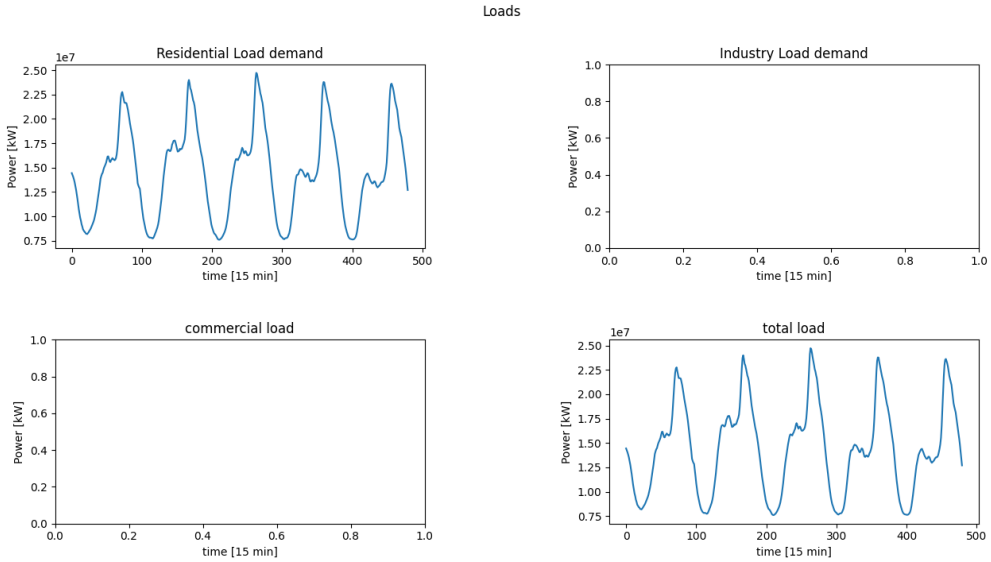


Figure D.17: The load profiles of the different sectors and their sum

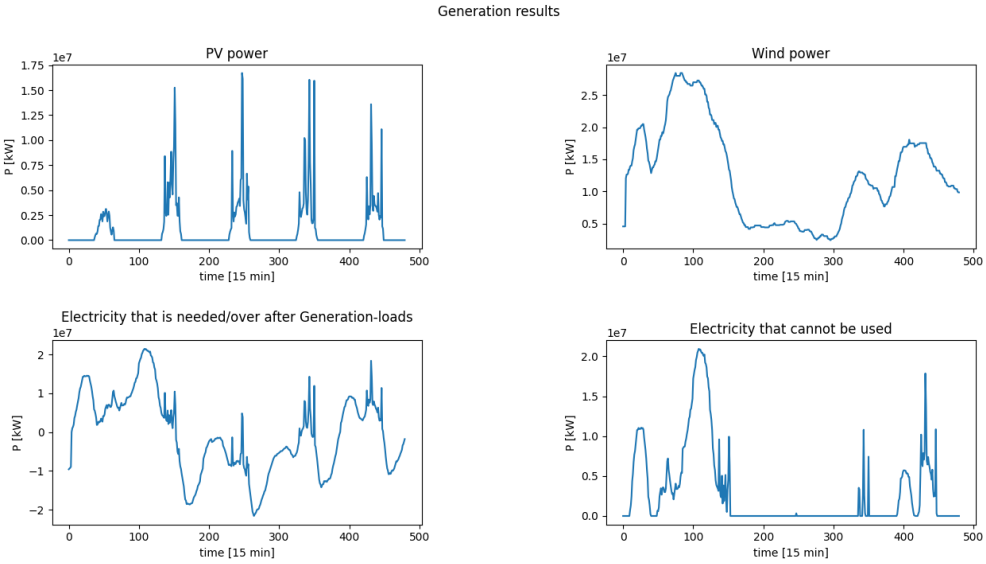


Figure D.18: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

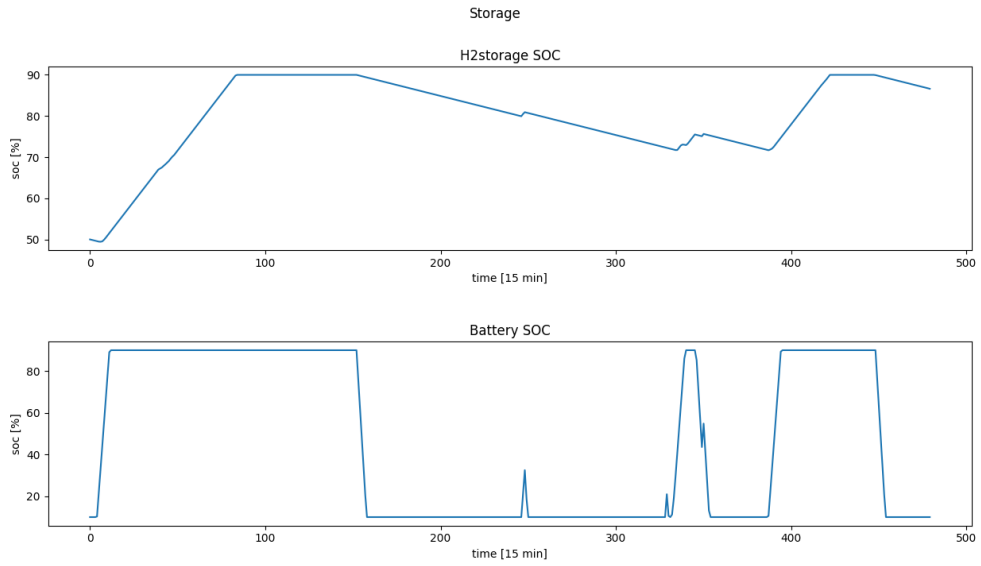


Figure D.19: The state of charge of both the battery and hydrogen storage

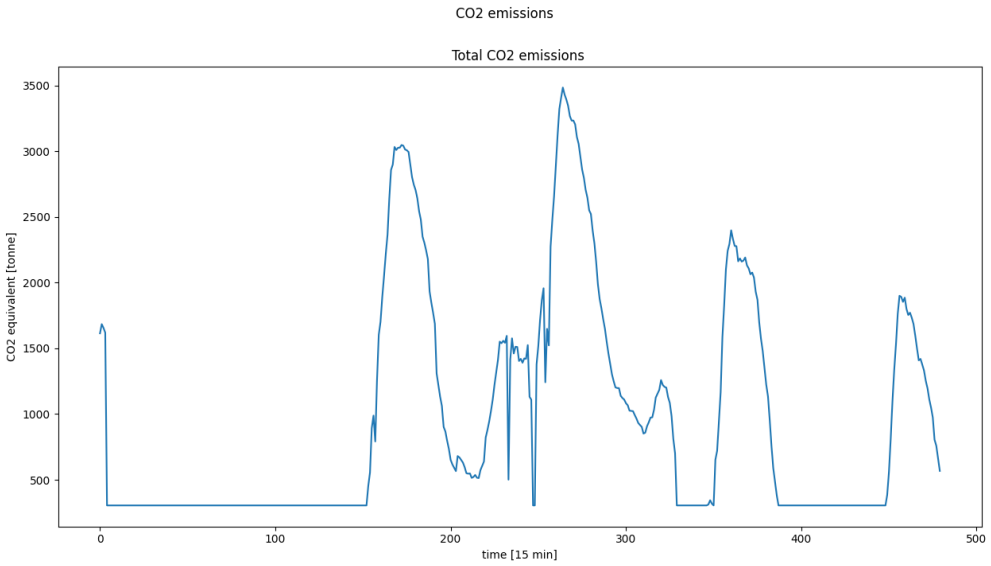


Figure D.20: The GHGs emitted at that moment

## D.7 Summer days, instead of winter

In this simulation, another set of days is run. Instead of the first of January as start, the first of July is chosen.

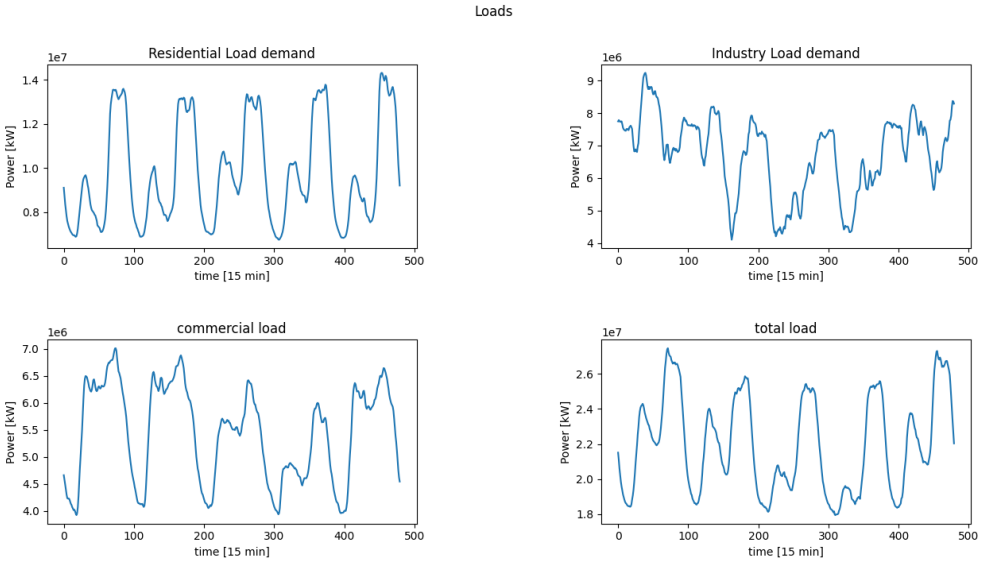


Figure D.21: The load profiles of the different sectors and their sum

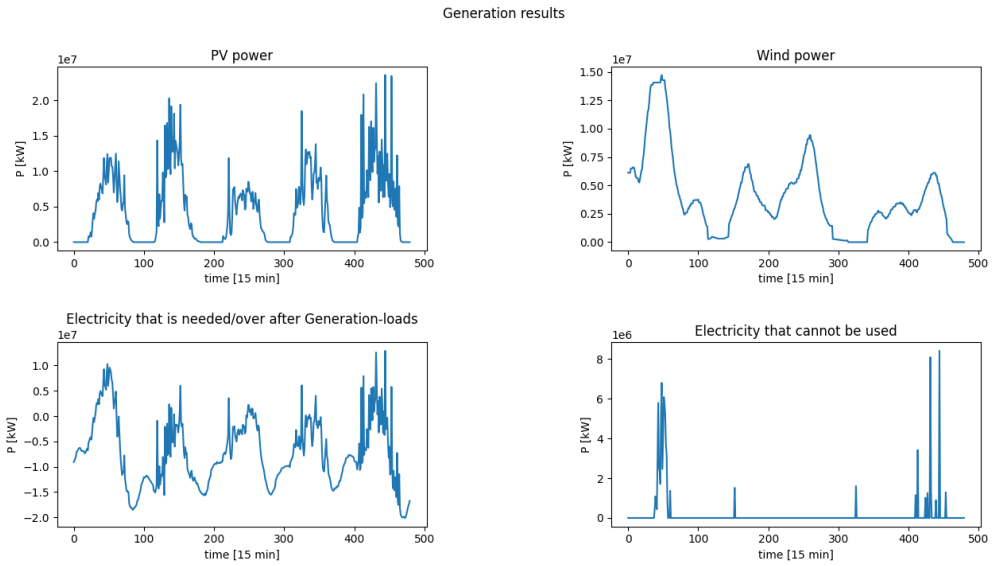


Figure D.22: The generation profiles of solar and wind energy (both onshore and offshore), the residual flow and the total energy dumped

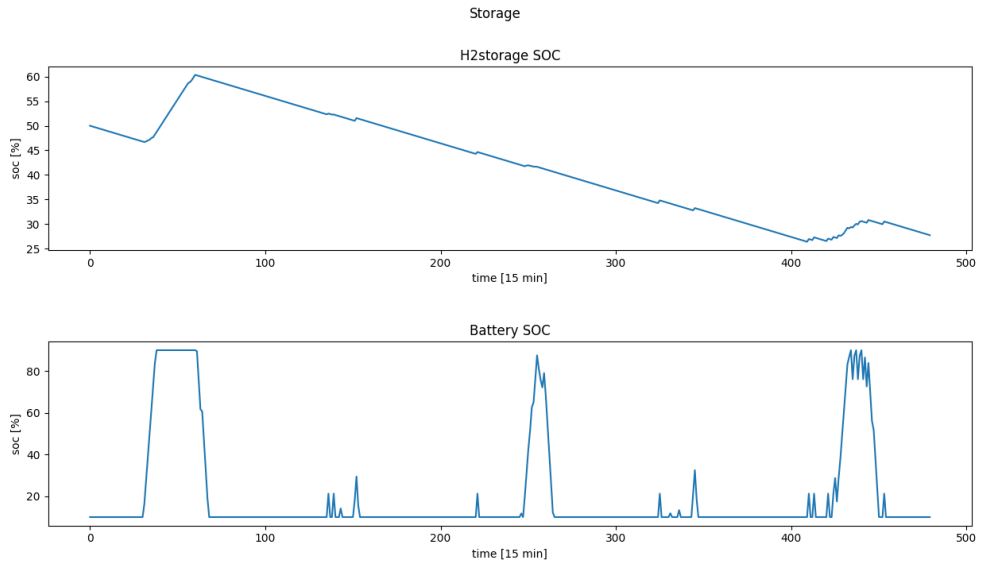


Figure D.23: The state of charge of both the battery and hydrogen storage

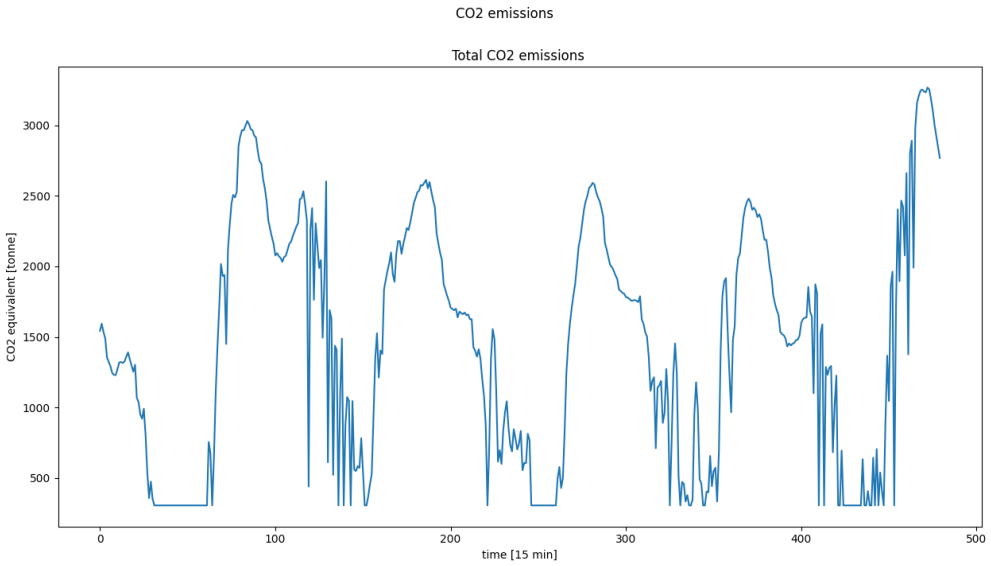


Figure D.24: The GHGs emitted at that moment

# Appendix E

# Python code

## E.1 Run everything.py

Listing E.1: Run everything.py

```
# run this file to start the simulation
import PySimpleGUI as sg
import configuration.build_configuration_xml
import pandas as pd

#2012-10-31 00:00:00
import simulation_creator
```

## E.2 hardwareattached.py

Listing E.2: hardwareattached.py

```
# code voor de hardware
import RPi.GPIO as GPIO
from time import sleep
import spidev
import random

GPIO.setmode(GPIO.BCM)
select_pins = [21, 26, 20, 13, 25, 23]
detect_pins = [16, 6, 12, 19, 24, 22]

ports = [
   {
   ”name” : ”j1”,
   ”select” : 21,
   ”detect” : 16,
   },
   {
   ”name” : ”j2”,
   ”select” : 26,
   ”detect” : 6,
   },
   {
   ”name” : ”j3”,
```

```
   ”select” : 20,
   ”detect” : 12,
   },
   {
   ”name” : ”j4”,
   ”select” : 13,
   ”detect” : 19,
   },
   {
   ”name” : ”j5”,
   ”select” : 25,
   ”detect” : 24,
   },
   {
   ”name” : ”j6”,
   ”select” : 23,
   ”detect” : 22,
   },
   ]


GPIO.setup(17, GPIO.OUT) # diode_EN
GPIO.setup(18, GPIO.OUT) # efuse_EN
GPIO.output(17, GPIO.HIGH)
GPIO.output(18, GPIO.HIGH)

for pin in select_pins:
   GPIO.setup(pin, GPIO.OUT)
   GPIO.output(pin, GPIO.LOW)
for pin in detect_pins:
   GPIO.setup(pin, GPIO.IN)



spi = spidev.SpiDev()
spi.open(0,0)
spi.max_speed_hz = 10000
spi.mode = 0b10

num = 0

def getSolarPower():
   total = 0
   for port in ports:
      if GPIO.input(port[”detect”]) == GPIO.LOW:
         x = [0x00,]
         GPIO.output(port[”select”], GPIO.HIGH)
         x = [255^i for i in x] # NOT4
         x = spi.xfer2(x)
         x = [255^i for i in x] # NOT4
         print(x)
         print(x[0])
         if (x[0] == 101):
            x = spi.xfer2([0,])
            x = [255^i for i in x] # NOT4
            total += x[0]
```

```
            GPIO.output(port["select"], GPIO.LOW)
    total /= 0xFE
    print("solar power = " + str(total))
    return (total)

def getWindPower():
    total = 0
    for port in ports:
        if GPIO.input(port["detect"]) == GPIO.LOW:
            x = [0x00,]
            GPIO.output(port["select"], GPIO.HIGH)
            x = [255^i for i in x] # NOT4
            x = spi.xfer2(x)
            x = [255^i for i in x] # NOT4
            print(x)
            print(x[0])
            if (x[0] == 102):
                x = spi.xfer2([0,])
                x = [255^i for i in x] # NOT4
                total += x[0]
            GPIO.output(port["select"], GPIO.LOW)
    total /= 0xFE
    print("wind power = " + str(total))
    return (total)


def setBatterySOC(newstatus): # er moet nog komen dat er niks vanuit de
    ↪ controller naar de battery gaat als hij tijdens de simulatie eruit is
    ↪  getrokken
    SOC = int(newstatus * 255)
    total = 0
    for port in ports:
        if GPIO.input(port["detect"]) == GPIO.LOW:
            x = [0x01, ]
            GPIO.output(port["select"], GPIO.HIGH)
            x = [255 ^ i for i in x] # NOT4
            x = spi.xfer2(x)
            x = [255 ^ i for i in x] # NOT4
            print(x)
            print(x[0])
            if (x[0] == 105):
                x = [SOC, ]
                x = [255 ^ i for i in x] # NOT4
                x = spi.xfer2(x)
                x = [255 ^ i for i in x] # NOT4
                total += x[0]
                print("battery found")
            GPIO.output(port["select"], GPIO.LOW)
    total /= 0xFE
    print("new SOC = " + str(SOC))


def setLoad(newstatus):
    load = int(newstatus * 100)
    total = 0
    for port in ports:
```

```
      if GPIO.input(port["detect"]) == GPIO.LOW:
         x = [0x01, ]
         GPIO.output(port["select"], GPIO.HIGH)
         x = [255 ^ i for i in x] # NOT4
         x = spi.xfer2(x)
         x = [255 ^ i for i in x] # NOT4
         print(x)
         print(x[0])
         if (x[0] == 106):
            x = [load, ]
            x = [255 ^ i for i in x] # NOT4
            x = spi.xfer2(x)
            x = [255 ^ i for i in x] # NOT4
            total += x[0]
            print("load found")
         GPIO.output(port["select"], GPIO.LOW)
   total /= 0xFE
   print("new load = " + str(load))
```

## E.3 simulation_creator.py

Listing E.3: simulation_creator.py

```
import subprocess
import pandas as pd
import mosaik
import mosaik.util
import numpy as np
from mosaik.util import connect_many_to_one
import time
import matplotlib
import datetime
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import PySimpleGUI as sg
from configuration.buildmodelset import *
from configuration.build_configuration_xml import *
matplotlib.use('TkAgg')

# This part of code is part of the GUI and let you give the input of the
    ↪ startdate
# and time you want to run the simulation

layout11 = [[sg.Text('Give a startdate in yyyy-mm-dd hh:mm:ss, where year
    ↪ can be 2021 or 2030,')],
          [sg.Text('minutes can be 00, 15, 30 or 45 and seconds can be
              ↪ only 00 and press submit.')],
         [sg.InputText(key='start_date')], [sg.Button("Submit1")]]
layout22 = [[sg.Text('Give amount of hours you want to run the simulation
    ↪ and press submit.'), sg.InputText(key='hours')], [sg.Button("Submit2"
    ↪ )]]

layout = [[sg.Column(layout11, key='-COL1-'), sg.Column(layout22, visible=
    ↪ False, key='-COL2-')],[sg.Button('Exit')]]
window2 = sg.Window(
```

```
      "Matplotlib Single Graph",
      layout,
      location=(0, 0),
      finalize=True,
      element_justification="center",
      font="Helvetica 18",
   )
layout = 11
while True:
   event, values = window2.read()

   if event == "Submit1":
      START_DATE = values['start_date'] # Saves the input start date in
         ↪ the variable Start_Date
      print(START_DATE)
      window2['-COL1-'].update(visible=False)
      layout = 22
      window2['-COL2-'].update(visible=True)
   if event == "Submit2":
      print(values['hours'])
      end = float(values['hours']) * 3600 # Saves the input amount of
         ↪ hours in the variable end
      break
   if event == "Exit" or event == sg.WIN_CLOSED:
      break
window2.close()

# If in mode 1 when you have attached devices in hardware you see every
   ↪ step of the simulation a updated graph
# if in mode 2 you see only at the end the graphs
if mode1:
   RESULTS_SHOW_TYPE['Update_graphs_during_simulation'] = True
   RESULTS_SHOW_TYPE['dashboard_show'] = False
else:
   RESULTS_SHOW_TYPE['Update_graphs_during_simulation'] = False

# read the config file that is created in build_configuration
sim_config_ddf = pd.read_xml('./configuration/config.xml')
sim_config_d = pd.read_xml('./config.xml')
if sim_config_ddf.equals(sim_config_d):
   sim_config = {row[1]: {row[2]: row[3]} for row in sim_config_ddf.values
      ↪ }
else:
   sim_config = {row[1]: {row[2]: row[3]} for row in sim_config_d.values}


tosh = sim_config_ddf[sim_config_ddf['method'] == 'connect']
if not tosh.empty:
   with open('run.sh', 'w') as rsh:
      rsh.write("#! /bin/bash")
      for row in tosh.values:
         rsh.write("\n"+"lxterminal -e ssh illuminator@"+row[3].replace('
            ↪ :5123', ' ')+"'./Desktop/Final_illuminator/configuration/
            ↪ runshfile/run"+row[1]+".sh'&")
```

```
    subprocess.run(['/bin/bash', '/home/illuminator/Desktop/
        ↪ Final_illuminator/configuration/run.sh'])


# read the right connection file that is created in
    ↪ build_configuration_xml
con1 = pd.read_xml('./connection.xml')
connection = pd.read_xml('./configuration/connection.xml')

if con1.equals(connection):
    print('connction.xml was equal to configuration/connection.xml')
else:
    connection = con1


if RESULTS_SHOW_TYPE['dashboard_show']:
    import wandb
    wandb.init(project="illuminator-project")
    wandb.define_metric("custom_step")
# makes sure the correct data is loaded for the chosen year
START_DATE_obj = datetime.datetime.strptime(START_DATE, '%Y-%m-%d %H:%M:%S
    ↪ ')

if str(START_DATE_obj.year) == '2021':
    WIND_DATA = 'Scenarios/Offshore_wind_2021.txt'
    Pv_DATA = 'Scenarios/pv_data_2021.txt' # solar data file
        ↪ ########################
 # load_DATA = 'Scenarios/Load_data_2021.txt' ######################
    if mode1:
        factory_DATA = 'Scenarios/Load_data_2021.txt'
    else:
        factory_DATA = 'Scenarios/industry_load_2021_witharg.txt'
    resident_DATA = 'Scenarios/Residential_load_2021+PV.txt'
    commercial_DATA = 'Scenarios/Commercial_load_2021_withoutarg.txt'
    imexport_DATA = 'Scenarios/imexport_2021.txt'
    WONSHORE_DATA = 'Scenarios/Onshore_wind_2021.txt'
    pv_panel_set['Module_Efficiency'] = 0.18
    pv_set['cap'] = 14911000
    Wind_set['cap'] = 2460000
    Wonshore_set['cap'] = 5300000
elif str(START_DATE_obj.year) == '2030':
    WIND_DATA = 'Scenarios/Offshore_wind_2030.txt'
    Pv_DATA = 'Scenarios/pv_data_2030.txt' # solar data file
        ↪ ########################
# load_DATA = 'Scenarios/Load_data_2030.txt'
    if mode1:
        factory_DATA = 'Scenarios/Load_data_2030.txt'
    else:
        factory_DATA = 'Scenarios/industry_load_2030.txt'
    resident_DATA = 'Scenarios/Residential_load_2030.txt'
    commercial_DATA = 'Scenarios/Commercial_Load_2030.txt'
    WONSHORE_DATA = 'Scenarios/Onshore_wind_2030.txt'

    imexport_DATA = 'Scenarios/imexport_2030.txt'
    pv_panel_set['Module_Efficiency'] = 0.20
    pv_set['cap'] = 26758432
```

```
    Wind_set['cap'] = 21000000
    Wonshore_set['cap'] = 7000000
else:
    print('You did not fill in the correct year format')

#set up the "world" of the scenario

world = mosaik.World(sim_config, debug=True)

models=pd.concat([connection["send"], connection["receive"]])
models=models.drop_duplicates(keep='first', inplace=False)
models.reset_index(drop=True, inplace=True)

#
Defined_models=pd.DataFrame()
if model1:
    Defined_models['model'] = pd.Series(['Wind', 'PV', 'Battery','Factory'
        ↪ ])
else:
    Defined_models['model']= pd.Series(['Wind', 'PV', 'Electrolyser', '
        ↪ H2storage', 'Battery', 'Fuelcell',
                                'Factory', 'Resident', 'Commercial', '
                                    ↪ Wonshore', 'Imexport'])
number=[]
for model in Defined_models['model']:
    number.append(
        int((models.str.startswith(model.lower())==True).sum())
    )
Defined_models['number']=number

ctrlsim = world.start('Controller')
ctrl = ctrlsim.Ctrl(sim_start=START_DATE, soc_min=Battery_set['soc_min'],
    ↪ soc_max=Battery_set['soc_max'],
                b_p_max = Battery_set['max_p'], b_p_min = Battery_set['
                    ↪ min_p'], el_p_max = electrolyser_set['max_p'],
                h2_soc_min=h2_set['h2storage_soc_min'], h2_soc_max=h2_set['
                    ↪ h2storage_soc_max'],
                fc_eff=fuelcell_set['eff'], fossil_em_const =
                    ↪ fossil_emissions_set['fossil_emissions_const'],
                fossil_em_var = fossil_emissions_set['fossil_emissions_var'
                    ↪ ],
                h2tofa_em_saved = factory_set['h2tofa_em_saved'],
                    ↪ max_fah2_dem = factory_set['max_h2_dem'])


    #### the controller gives us just a basic value of power flow which
        ↪ needs to be there. so -ve or +ve
collector = world.start('Collector', start_date=START_DATE,results_show=
    ↪ RESULTS_SHOW_TYPE)
monitor = collector.Monitor()

for model_i in Defined_models.iterrows():
    if model_i[1]['model'] == 'PV':
        solardata = world.start('CSVB', sim_start=START_DATE, datafile=
            ↪ Pv_DATA) # loading the data file to mosaik
```

```
        pvsim = world.start('PV') # the name given to mosaik file while
            ↪ importing in scenario sim_config
        # p_data = {'Module_area': 1.26, 'NOCT': 44, 'Module_Efficiency':
            ↪ 0.198, 'Irradiance_at_NOCT': 800,
        # 'Power_output_at_STC': 250}
        ## instantiating the pv model by giving the parameter values.
        print(pv_set['cap'])
        pv = pvsim.PVset.create(model_i[1]['number'], sim_start=START_DATE,
            ↪ panel_data=pv_panel_set,
                        m_tilt=pv_set['m_tilt'], m_az=pv_set['m_az'], cap=
                            ↪ pv_set['cap'],
                            output_type=pv_set['output_type']) # cap in W
        solarprofile_data = solardata.Solar_data.create(model_i[1]['number'
            ↪ ]) # instantiating an entity of the solar data file
        for i in range(model_i[1]['number']):
            world.connect(solarprofile_data[i], pv[i], 'G_Gh', 'G_Dh', 'G_Bn'
                ↪ , 'Ta', 'hs', 'FF', 'Az')

    elif model_i[1]['model'] == 'Wind':
        WSdata = world.start('CSVB', sim_start=START_DATE, datafile=
            ↪ WIND_DATA) # loading the data file to mosaik
        windsim = world.start('Wind') # the name you gave to the in
            ↪ sim_config above
        wind = windsim.windmodel.create(model_i[1]['number'], sim_start=
            ↪ START_DATE,
                                    cap=Wind_set['cap'],
                                    p_rated=Wind_set['p_rated'], u_rated=
                                        ↪ Wind_set['u_rated'],
                                    u_cutin=Wind_set['u_cutin'],u_cutout=
                                        ↪ Wind_set['u_cutout'],
                                    cp=Wind_set['cp'],diameter=Wind_set['
                                        ↪ diameter'], output_type=Wind_set['
                                        ↪ output_type']) # p_rated in kW #
                                        ↪ Resolution here is in minutes
        ## print(wind.full_id)
        windspeed_data = WSdata.WS_datafile.create(model_i[1]['number']) #
            ↪ instantiating an entity of the wind data file
        for i in range(model_i[1]['number']):
            world.connect(windspeed_data[i], wind[i], 'u')

    elif model_i[1]['model'] == 'Wonshore':
        WonSdata = world.start('CSVB', sim_start=START_DATE, datafile=
            ↪ WONSHORE_DATA) # loading the data file to mosaik
        wonshoresim = world.start('Wonshore') # the name you gave to the in
            ↪ sim_config above
        wonshore = wonshoresim.wonshoremodel.create(model_i[1]['number'],
            ↪ sim_start=START_DATE,
                                    cap=Wonshore_set['cap'],
                                    p_rated=Wonshore_set['p_rated'], u_rated=
                                        ↪ Wonshore_set['u_rated'],
                                    u_cutin=Wonshore_set['u_cutin'],u_cutout=
                                        ↪ Wonshore_set['u_cutout'],
                                    cp=Wonshore_set['cp'],diameter=Wonshore_set[
                                        ↪ 'diameter'], output_type=Wonshore_set['
                                        ↪ output_type']) # p_rated in kW #
                                        ↪ Resolution here is in minutes
```

```
    ## print(wind.full_id)
    windonshorespeed_data = WonSdata.wonshore_datafile.create(model_i
        ↪ [1]['number']) # instantiating an entity of the wind data file
    for i in range(model_i[1]['number']):
        world.connect(windonshorespeed_data[i], wonshore[i], 'u')

elif model_i[1]['model'] == 'Load':
    loaddata = world.start('CSVB', sim_start=START_DATE, datafile=
        ↪ load_DATA) # loading the data file to mosaik
    loadsim = world.start('Load') # the name you gave to the in
        ↪ sim_config above

    load = loadsim.loadmodel.create(model_i[1]['number'], sim_start=
        ↪ START_DATE,
                                houses=load_set['houses'], output_type=
                                    ↪ load_set['output_type']) # loadmodel is
                                    ↪  the name we gave in the mosaik API
                                    ↪ file while writing META

    load_data = loaddata.Load_data.create(model_i[1]['number']) #
        ↪ Load_data is the header in the txt file containing the load
        ↪ values.
    for i in range(model_i[1]['number']):
        world.connect(load_data[i], load[i], 'load') #third attribute
            ↪ here must match the second attribute in the second line of
            ↪ the txt file containing data

elif model_i[1]['model'] == 'Imexport':
    imexportdata = world.start('CSVB', sim_start=START_DATE, datafile=
        ↪ imexport_DATA) # loading the data file to mosaik
    imexportsim = world.start('Imexport') # the name you gave to the in
        ↪ sim_config above

    imexport = imexportsim.imexportmodel.create(model_i[1]['number'],
        ↪ sim_start=START_DATE,
                                imexports=imexport_set['imexports'],
                                    ↪ output_type=imexport_set['output_type'
                                    ↪ ]) # imexportmodel is the name we gave
                                    ↪ in the mosaik API file while writing
                                    ↪ META

    imexport_data = imexportdata.imexport_data.create(model_i[1]['number
        ↪ ']) # Load_data is the header in the txt file containing the
        ↪ load values.
    for i in range(model_i[1]['number']):
        world.connect(imexport_data[i], imexport[i], 'imexport') #third
            ↪ attribute here must match the second attribute in the second
            ↪  line of the txt file containing data


elif model_i[1]['model'] == 'Factory':
    factorydata = world.start('CSVB', sim_start=START_DATE,
                          datafile=factory_DATA) # loading the data file to
                              ↪  mosaik
    factorysim = world.start('Factory') # the name you gave to the model
        ↪  in sim_config in build_configuration_xml.py
```

```
        factory = factorysim.factorymodel.create(model_i[1]['number'],
            ↪ sim_start=START_DATE,
                                        factories=factory_set['factories'],
                                            ↪ output_type=factory_set['
                                            ↪ output_type']) # factorymodel is
                                            ↪  the name we gave in the mosaik
                                            ↪ API file while writing META
        factory_data = factorydata.Factory_load.create(model_i[1]['number'])
            ↪  # Factory_load is the header in the txt file containing the
            ↪ industry load values.
        for i in range(model_i[1]['number']):
            world.connect(factory_data[i], factory[i], 'factory_load') #third
                ↪  attribute here must match the second attribute in the
                ↪ second line of the txt file containing data

    elif model_i[1]['model'] == 'Commercial':
        commercialdata = world.start('CSVB', sim_start=START_DATE, datafile=
            ↪ commercial_DATA) # loading the data file to mosaik
        commercialsim = world.start('Commercial') # the name you gave to the
            ↪  in sim_config above

        commercial = commercialsim.commercialmodel.create(model_i[1]['number
            ↪ '], sim_start=START_DATE,
                                    companies=commercial_set['companies'],
                                        ↪ output_type=commercial_set['output_type
                                        ↪ ']) # loadmodel is the name we gave in
                                        ↪ the mosaik API file while writing META

        commercial_data = commercialdata.Commercial_Load.create(model_i[1]['
            ↪ number']) # Load_data is the header in the txt file containing
            ↪ the load values.
        for i in range(model_i[1]['number']):
            world.connect(commercial_data[i], commercial[i], 'commercial_load
                ↪ ') #third attribute here must match the second attribute in
                ↪ the second line of the txt file containing data

    elif model_i[1]['model'] == 'Resident':
        residentdata = world.start('CSVB', sim_start=START_DATE,
                            datafile=resident_DATA) # loading the data file
                                ↪ to mosaik
        residentsim = world.start(
            'Resident') # the name you gave to the model in sim_config in
                ↪ build_configuration_xml.py
        resident = residentsim.residentmodel.create(model_i[1]['number'],
            ↪ sim_start=START_DATE,
                                        residents=resident_set['residents'],
                                            ↪ output_type=resident_set[
                'output_type']) # factorymodel is the name we gave in the
                    ↪ mosaik API file while writing META
        resident_data = residentdata.Residential_load.create(
            model_i[1]['number']) # Residential_load is the header in the txt
                ↪  file containing the residential
        # load values.
        for i in range(model_i[1]['number']):
            world.connect(resident_data[i], resident[i],
```

```
                    'residents_load') # third attribute here must match the
                        ↪  second attribute in the second line of the txt
                        ↪ file containing data

    elif model_i[1]['model'] == 'Battery':
        #### no battery input data file as the input 'p_ask' comes from the
            ↪ controller. We connect the controller and the battery ahead.

        batterysim = world.start('Battery') # the name you gave to the in
            ↪ sim_config above
        #Battery_initialset = {'initial_soc': 20}
        # Battery_set = {'max_p': 500, 'min_p': -500, 'max_energy': 500,
        # 'charge_efficiency': 0.9, 'discharge_efficiency': 0.9,
        # 'soc_min': 10, 'soc_max': 90, 'flag': -1, 'resolution': 15} # p in
            ↪  kW
        battery = batterysim.Batteryset(sim_start=START_DATE, initial_set=
            ↪ Battery_initialset,
                                        battery_set=Battery_set)
        ## print(battery.full_id)
    elif model_i[1]['model'] == 'Electrolyser':
        electrosim = world.start('Electrolyser')
        h2storagesim = world.start('H2storage')
        fcsim = world.start('Fuelcell')
        electrolyser = electrosim.electrolysermodel(sim_start=START_DATE,
            ↪ eff=electrolyser_set['eff'],
                                                    resolution=electrolyser_set['
                                                        ↪ resolution']) # here
                                                        ↪ resolution should be the same
                                                        ↪ we have for the data sets for
        ## wind and pv and load. Make sure to keep the same resolution all
            ↪ across

        #h2storage_initial = {'initial_soc': 20}
        #h2_set = {'h2storage_soc_min': 10, 'h2storage_soc_max': 90, 'max_h2
            ↪ ': 200, 'min_h2': -200, 'flag': -1}
        h2storage = h2storagesim.compressed_hydrogen(sim_start=START_DATE,
            ↪ initial_set=h2storage_initial,
                                                     h2_set=h2_set)
        ##
        fuelcell = fcsim.fuelcellmodel(sim_start=START_DATE, eff=
            ↪ fuelcell_set['eff'])
for i, row in connection.iterrows():
    try:
        row['more']
    except KeyError:
        world.connect(eval(row['send']), eval(row['receive']), row['message'
            ↪ ])
    else:
        if row['more']=='async_requests=True':
            world.connect(eval(row['send']),eval(row['receive']),row['message
                ↪ '],async_requests=True)
        elif row['more']==None:
            #print(eval(row['receive']), 'pep')
            #print(eval(row['send']))
            world.connect(eval(row['send']), eval(row['receive']), row['
                ↪ message'])
```

```
      else:
         world.connect(eval(row['send']),eval(row['receive']),(row['
            ↪ message'],row['more']))
if realtimefactor==0:
   world.run(until=end)
else:
   world.run(until=end,rt_factor=realtimefactor)
##################################final results show
   ↪ ##########################################
if RESULTS_SHOW_TYPE['Finalresults_show']==True:
   import matplotlib.pyplot as plt
   import matplotlib
   from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
   import PySimpleGUI as sg
   matplotlib.use('TkAgg')
   import pandas as pd

   data=pd.read_csv('./Result/results.csv')

      # Get all axes of figure
   if RESULTS_SHOW_TYPE['Update_graphs_during_simulation'] == True:
      fig1 = matplotlib.figure.Figure(figsize=(15,7.5),dpi=100) # just
         ↪ make sure the figsize is big enough that it is on whole page
      fig2 = matplotlib.figure.Figure(figsize=(15,7.5),dpi=100)
      fig3 = matplotlib.figure.Figure(figsize=(15, 7.5), dpi=100)
      fig4 = matplotlib.figure.Figure(figsize=(15, 7.5), dpi=100)
      fig5 = matplotlib.figure.Figure(figsize=(15, 7.5), dpi=100)
      fig1.subplots_adjust(hspace=0.5, wspace=0.5)
      fig2.subplots_adjust(hspace=0.5,wspace = 0.5)
      fig3.subplots_adjust(hspace=0.5, wspace=0.5)
      fig4.subplots_adjust(hspace=0.5, wspace=0.5)
      fig5.subplots_adjust(hspace=0.5, wspace=0.5)
      fig1.suptitle('Generation results')
      fig2.suptitle('Storage')
      fig3.suptitle('Loads')
      fig4.suptitle('CO2 emissions')
      fig5.suptitle('Battery, Load and Emissions')
      # Plot new data
   else:
      fig1 = plt.figure(figsize=(15,7.5))
      fig1.subplots_adjust(hspace=0.5, wspace=0.5)
      fig1.suptitle('Generation results')
      fig2 = plt.figure(figsize=(15, 7.5))
      fig2.subplots_adjust(hspace=0.5, wspace=0.5)
      fig2.suptitle('Storage')
      fig3 = plt.figure(figsize=(15, 7.5))
      fig3.subplots_adjust(hspace=0.5, wspace=0.5)
      fig3.suptitle('Loads')
      fig4 = plt.figure(figsize=(15, 7.5))
      fig4.subplots_adjust(hspace=0.5, wspace=0.5)
      fig4.suptitle('CO2 emissions')

   # Plot new data

   ax1=fig1.add_subplot(221)
```

```
pvgen = 0
try:
    while True:
        data[f'PV-0.pv_{pvgen}-pv_gen']
        pvgen += 1
except KeyError:
    pv_gen_sum = []

try:
    pv_gen_sum = []
    for j in range(pvgen):
        pv_gen_sum.append(data[f'PV-0.pv_{j}-pv_gen'])
    ax1.plot(sum(pv_gen_sum))
except KeyError:
    print('There is no PV model')
ax1.set_title('PV power')
ax1.set_xlabel('time [15 min]')
ax1.set_ylabel('P [kW]')


ax2=fig1.add_subplot(222)
wgen = 0
wongen = 0
try:
    while True:
        data[f'Wind-0.wind_{wgen}-wind_gen']
        wgen += 1
except KeyError:
    pass
try:
    while True:
        data[f'Wonshore-0.wonshore_{wongen}-wonshore_gen']
        wongen +=1
except KeyError:
    wind_gen_sum = []
try:
    wind_gen_sum = []
    for j in range(wgen):
        wind_gen_sum.append(data[f'Wind-0.wind_{j}-wind_gen'])
except KeyError:
    pass
try:
    for i in range(wongen):
        wind_gen_sum.append(data[f'Wonshore-0.wonshore_{i}-wonshore_gen'
            ↪ ])
except KeyError:
    print('There is no Wind model')
ax2.plot(sum(wind_gen_sum))
ax2.set_title('Wind power')
ax2.set_xlabel('time [15 min]')
ax2.set_ylabel('P [kW]')

ax3=fig1.add_subplot(223)
try:
    ax3.plot(data['Controller-0.ctrl_0-flow'])
except KeyError:
```

```
    print('There is no controller model')
ax3.set_title('Electricity that is needed/over after Generation-loads')
ax3.set_xlabel('time [15 min]')
ax3.set_ylabel('P [kW]')


ax4=fig1.add_subplot(224)
try:
    ax4.plot(data['Controller-0.ctrl_0-dump'])
except KeyError:
    print('There is no controller model')
ax4.set_title('Electricity that cannot be used')
ax4.set_xlabel('time [15 min]')
ax4.set_ylabel('P [kW]')
if mode1:
    ax5 =fig5.add_subplot(221)
    try:
        ax5.plot(data['Battery-0.Battery_0-soc'])
    except KeyError:
        print('There is no battery model')
    ax5.set_title('Battery SOC')
    ax5.set_xlabel('time [15 min]')
    ax5.set_ylabel('soc [%]')
else:
    ax5 = fig2.add_subplot(211)
    try:
        ax5.plot(data['H2storage-0.h2storage_0-h2_soc'])
    except KeyError:
        print('There is no H2 model')
    ax5.set_title('H2storage SOC')
    ax5.set_xlabel('time [15 min]')
    ax5.set_ylabel('soc [%]')

if mode1:
    ax6 = fig5.add_subplot(222)
else:
    ax6 = fig2.add_subplot(212)
    try:
        ax6.plot(data['Battery-0.Battery_0-soc'])
    except KeyError:
        print('There is no battery model')
    ax6.set_title('Battery SOC')
    ax6.set_xlabel('time [15 min]')
    ax6.set_ylabel('soc [%]')

allloads = []
ax7=fig3.add_subplot(221)
try:
    ax7.plot(data['Resident-0.resident_0-resident_dem'])
    allloads.append(data['Resident-0.resident_0-resident_dem'])
except KeyError:
    print('There is no Residential Load model')
ax7.set_title('Residential Load demand')
ax7.set_xlabel('time [15 min]')
ax7.set_ylabel('Power [kW]')
```

```
ax8= fig3.add_subplot(222)
try:
   ax8.plot(data['Factory-0.factory_0-factory_dem'])
   allloads.append(data['Factory-0.factory_0-factory_dem'])
except KeyError:
   print('There is no Industry load model')
ax8.set_title('Industry Load demand')
ax8.set_xlabel('time [15 min]')
ax8.set_ylabel('Power [kW]')


ax9= fig3.add_subplot(223)
try:
   ax9.plot(data['Commercial-0.commercial_0-commercial_dem'])
   allloads.append(data['Commercial-0.commercial_0-commercial_dem'])
except KeyError:
   print('There is no commercial load model')
ax9.set_title('commercial load')
ax9.set_xlabel('time [15 min]')
ax9.set_ylabel('Power [kW]')

ax10 = fig3.add_subplot(224)

try:
   if mode1:
      ax6.plot(sum(allloads))
   else:
      ax10.plot(sum(allloads))
except KeyError:
   print('there is no load attached')
if mode1:
   ax6.set_title('total load')
   ax6.set_xlabel('time [15 min]')
   ax6.set_ylabel('Power [kW]')
else:
   ax10.set_title('total load')
   ax10.set_xlabel('time [15 min]')
   ax10.set_ylabel('Power [kW]')

if mode1:
   ax11 = fig5.add_subplot(223)
else:
   ax11 = fig4.add_subplot(211)
try:
   ax11.plot(data['Controller-0.ctrl_0-emissions'])
except KeyError:
   print('There is no controller model')
ax11.set_title('Total CO2 emissions ')
ax11.set_xlabel('time [15 min]')
ax11.set_ylabel('CO2 equivalent [tonne]')

ax12 = fig4.add_subplot(212)
try:
   ax12.plot(data['Imexport-0.imexport_0-imexport_dem'])
except KeyError:
   print('There is no Imexport model')
```

```
    ax12.set_title('Total Imexport load ')
    ax12.set_xlabel('time [15 min]')
    ax12.set_ylabel('Power [kW]')


    #start of GUI
    layout1 = [
        [sg.Text("Click a button to see the graphs. If you want to save the
            ↪ graphs do that before closing the window.")]]
    layout2 = [[sg.Text("Generation")],
              [sg.Canvas(key="-CANVAS-")]]
    layout3 = [[sg.Text("Storage")],
             [sg.Canvas(key="-CANVAS2-")]]
    layout4 = [[sg.Text("Loads")],
             [sg.Canvas(key="-CANVAS3-")]]
    layout5 = [[sg.Text("Emissions and im- and exported electricity")],
             [sg.Canvas(key="-CANVAS4-")]]
    layout6 = [[sg.Text("The graphs are already made, so you can just save
        ↪ those")]]
    layout7 = [[sg.Text("Battery, Loads and Emission")],[sg.Canvas(key="-
        ↪ CANVAS5-")]]
    if mode1:
        layout_graphs = [[sg.Column(layout1, key='-COL1-'), sg.Column(
            ↪ layout2, visible=False, key='-COL2-'),
                      sg.Column(layout6, visible=False, key='-COL6-'), sg.
                          ↪ Column(layout7, visible=False, key='-COL7-')],
                     [sg.Button('Generation'), sg.Button('Battery, Load and
                         ↪ Emission'), sg.Button('Export Results'), sg.Button
                         ↪ ('Exit')]]
    else:
        layout_graphs = [[sg.Column(layout1, key='-COL1-'), sg.Column(
            ↪ layout2, visible=False, key='-COL2-'),sg.Column(layout3,
            ↪ visible=False, key='-COL3-'),sg.Column(layout4, visible=False,
            ↪ key='-COL4-'),sg.Column(layout5, visible=False, key='-COL5-'),
                       sg.Column(layout6, visible=False, key='-COL6-')],[sg.
                           ↪ Button('Generation'), sg.Button('Storage'),
                       sg.Button('Loads'), sg.Button('Emissions'), sg.Button
                           ↪ ('Export Results'), sg.Button('Exit')]]
    window = sg.Window(
        "Matplotlib Single Graph",
        layout_graphs,
        location=(0, 0),
        finalize=True,
        element_justification="center",
        font="Helvetica 18",
    )

    layout_graphs = 1
#flow dump all emissions

    def draw_figure(canvas, figure):
        figure_canvas_agg = FigureCanvasTkAgg(figure, canvas)
        figure_canvas_agg.draw()
        figure_canvas_agg.get_tk_widget().pack(side="top", fill="both",
            ↪ expand=5)
        return figure_canvas_agg
```

```
    # Add the plot to the window
    if mode1:
        draw_figure(window["-CANVAS-"].TKCanvas, fig1)
        draw_figure(window["-CANVAS5-"].TKCanvas, fig5)
    else:
        draw_figure(window["-CANVAS-"].TKCanvas, fig1)
        draw_figure(window["-CANVAS2-"].TKCanvas, fig2)
        draw_figure(window["-CANVAS3-"].TKCanvas, fig3)
        draw_figure(window["-CANVAS4-"].TKCanvas, fig4)
    while True:
        event, values = window.read()
        if event in (None, 'Exit'):
            break
        if event in 'Generation':
            window.maximize()
            window[f'-COL{layout_graphs}-'].update(visible=False)
            layout_graphs = 2
            window[f'-COL{layout_graphs}-'].update(visible=True)
        if event in 'Storage':
            window.maximize()
            window[f'-COL{layout_graphs}-'].update(visible=False)
            layout_graphs = 3
            window[f'-COL{layout_graphs}-'].update(visible=True)
        if event in 'Loads':
            window.maximize()
            window[f'-COL{layout_graphs}-'].update(visible=False)
            layout_graphs = 4
            window[f'-COL{layout_graphs}-'].update(visible=True)
        if event in 'Emissions':
            window.maximize()
            window[f'-COL{layout_graphs}-'].update(visible=False)
            layout_graphs = 5
            window[f'-COL{layout_graphs}-'].update(visible=True)
        if event in 'Export Results':
            if RESULTS_SHOW_TYPE['Update_graphs_during_simulation'] ==
                ↪ False:
                plt.show()
            else:
                window[f'-COL{layout_graphs}-'].update(visible=False)
                layout_graphs = 6
                window[f'-COL{layout_graphs}-'].update(visible=True)
        if event in 'Battery, Load and Emission':
            window.maximize()
            window[f'-COL{layout_graphs}-'].update(visible=False)
            layout_graphs = 7
            window[f'-COL{layout_graphs}-'].update(visible=True)

    window.close()
```

## E.4 configuration

### E.4.1 buildclientremoterun.py

Listing E.4: configuration/buildclientremoterun.py

```
import socket, subprocess,re
p=subprocess.Popen(["ifconfig"],stdout=subprocess.PIPE)
ifc_resp=p.communicate()
patt=re.compile(r'inet \d{3}\.\d{3}.\d{1,3}.\d{1,3}')
resp=patt.findall(str(ifc_resp[0]))
print(resp[0].replace('inet ',''))
IPAddr=resp[0].replace('inet ','')


# import socket
# hostname=socket.gethostname()
# IPAddr=socket.gethostbyname(hostname)
# print("Your Computer Name is:"+hostname)
# print("Your Computer IP Address is:"+IPAddr)
models=['Battery','Controller','Electrolyser','H2storage','Load','PV','
    ↪ Wind','Fuelcell','Factory', 'Resident', 'Commercial']
#models=['Battery']
for model in models:
   with open('./configuration/runshfile/run'+model+'.sh', 'w') as rsh:
      rsh.write("#! /bin/bash")
      rsh.write("\n" + "cd /home/illuminator/Desktop/Final_illuminator/
         ↪ Models/"+model)
      rsh.write("\npython "+model.lower()+'_mosaik.py '+IPAddr+':5123 --
         ↪ remote')

# import socket
# import fcntl
# import struct
# def get_ip_address(ifname):
# s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
# return socket.inet_ntoa(fcntl.ioctl(s.fileno(),0x8915,
# struct.pack('256s',ifname[:15]))[20:24])
# get_ip_address('eth0')
```

## E.4.2 buildmodelset.py

Listing E.5: configuration/buildmodelset.py

```
Battery_initialset = {'initial_soc': 20}
Battery_set = {'max_p': 1000000, 'min_p': -1000000, 'max_energy': 2000000,
        'charge_efficiency': 0.9, 'discharge_efficiency': 0.9,
        'soc_min': 10, 'soc_max': 90, 'flag': 0, 'resolution': 15} #p in
           ↪ kW; max_energy in kWh;
#Set flag as 1 to show fully discharged state; -1 show fully charged,0
   ↪ show ready to charge and discharge

h2storage_initial = {'initial_soc': 50}
h2_set = {'h2storage_soc_min': 10, 'h2storage_soc_max': 90, 'max_h2':
   ↪ 5000000, 'min_h2': -5000000, 'flag': -1, 'capacity':2000000}
#max_h2 min_h2 in kg/15min;
#flag as 1 to show fully discharged state; -1 show fully charged,0 show
   ↪ ready to charge and discharge;
```

```
#capacity in kg


pv_panel_set ={'panels':1,'Module_area': 1.65
            ,'NOCT': 44, 'Module_Efficiency': 0.22, 'Irradiance_at_NOCT':
                ↪ 800,
        'Power_output_at_STC': 250} #module area in m^2; irradiance in W/m
            ↪ ^2; power in W
pv_set={'m_tilt':24.55,'m_az':180,'cap':26758432,'output_type':'power'} #
    ↪ degree, degree, kWp
# 'NOCT':degree celsius; 'Irradiance_at_NOCT':W/m2 This is the irradiance
    ↪ that falls on the panel under NOCT conditions
# KW. Available in spec sheet of a module
load_set={'houses':1, 'output_type':'power'}
#3.329218e+08
factory_set={'factories':1, 'output_type':'power', 'max_h2_dem':2012, '
    ↪ h2tofa_em_saved':24/1000} #kg, tonCo2/kgh2

#Annual energy usage : Coal = 16346.0 GWh (76.16%), Natural gas = 56505.0
    ↪ GWh(22.03%), Oil = 1331.0 GWh(1.794%), Total = 74182 GWh
#Variable emission : Coal = 985.1 g/kWh, Gas = 466.0 g/kWh, Oil = 957.5 g/
    ↪ kWh, Total = 0.7617*466.0 + 0.2203*985.1 + 0.01794 * 957.5 = 589.1g/
    ↪ kWh

fossil_emissions_set={'fossil_emissions_const': 9.346, '
    ↪ fossil_emissions_var':589.1/1000000/4} #tonnes CO2-eq/15minutes

resident_set = {'residents': 1, 'output_type':'power'}
commercial_set={'companies':1, 'output_type':'power'}

imexport_set = {'imexports':1, 'output_type':'power'}

Wind_set={'cap':2100000,'p_rated':6000, 'u_rated':10.955, 'u_cutin':4, '
    ↪ u_cutout':25, 'cp':0.40, 'diameter':154, 'output_type':'power'}


Wonshore_set = {'cap':7000000,'p_rated':2200, 'u_rated':11.563, 'u_cutin'
    ↪ :3.0, 'u_cutout':25, 'cp':0.40, 'diameter':86, 'output_type':'power'}


# p_rated # kW power it generates at rated wind speed and above
# u_rated # m/s #windspeed it generates most power at
# u_cutin # m/s #below this wind speed no power generation
# u_cutout # m/s #above this wind speed no power generation. Blades are
    ↪ pitched
# cp # coefficient of performance of a turbine. Usually around0.40. Never
    ↪ more than 0.59
# powerout = 0 # output power at wind speed u

fuelcell_set={'eff':0.45}

electrolyser_set={'max_p': 3500000, 'eff':0.62,'resolution':15} #kw

RESULTS_SHOW_TYPE={'write2csv':True, 'dashboard_show':True, '
    ↪ Finalresults_show':True, 'Update_graphs_during_simulation':True}
#'write2csv':True/Flause Write the results to csv file
```

```
# #'Realtime_show':True/Flause, show the results in dashboard
# 'Finalresults_show':True/Flause, show the results after finish the
    ↪ simulation

realtimefactor=0
livedatamode = False

#0 as soon as possible. 1/60 using 1 second simulate 1 mintes
```

### E.4.3 build_configuration_xml.py

Listing E.6: configuration/build_configuration_xml.py

```
import pandas as pd
import PySimpleGUI as sg
# part of the GUI
layoutbuild1 = [[sg.Text("Choose if you have hardware attached mode 1, if
    ↪ not choose mode 2.")], [sg.Button("Mode 1"),
        sg.Button("Mode 2")]]
layoutbuild2 = [[sg.Text('Choose which devices you want to use. For wind
    ↪ and pv you can select more then one.')],
            [sg.Text('Generation sources:'),sg.Button("Wind Offshore") ,sg
                ↪ .Button("pv"), sg.Button("Wind Onshore")],
            [sg.Text('Storage:'), sg.Button("battery"), sg.Button("
                ↪ h2storage"),sg.Button("electrolyser"),sg.Button("fuelcell
                ↪ ")],
            [sg.Text('Loads:'), sg.Button("industry_load"), sg.Button("
                ↪ residential_load"), sg.Button("commercial_load"), sg.
                ↪ Button("Im- and Export electricity")],[sg.Button("submit"
                ↪ )]]
layoutbuild3 = [[sg.Text('Which devices are attached?')],
            [sg.Text('Generation sources:'),sg.Button("Wind"), sg.Button("
                ↪ Pv")],
            [sg.Text('Storage:'), sg.Button("Battery")],
            [sg.Text('Loads:'), sg.Button("Load")],[sg.Button("Submit")]]
layoutbuild = [[sg.Column(layoutbuild1, key='-COL1-'), sg.Column(
    ↪ layoutbuild2, visible=False, key='-COL2-'),
            sg.Column(layoutbuild3, visible=False, key='-COL3-')],[sg.
                ↪ Button('Exit')]]
window2 = sg.Window(
      "Matplotlib Single Graph",
      layoutbuild,
      location=(0, 0),
      finalize=True,
      element_justification="center",
      font="Helvetica 18",
   )
layoutbuild =1
attachedwind=0
attachedwindon = 0
attachedpv = 0
attachedbat =0
attachedh2stor = 0
attachedelectrolyser =0
attachedfuelcell =0
```

```
attachedcommercial =0
attachedresident =0
attachedindustry=0
attachedload = 0
attachedinex =0
while True:
   event, values = window2.read()

   if event == "Wind Offshore" or event == "Wind": # if you press the wind
      ↪ button it will attach a wind model
      attachedwind = attachedwind+1
      if attachedwind == 1:
         print('there is', attachedwind, 'offshore windfarm attached')
      else:
         print('there are', attachedwind, 'offshore windfarms attached')
   if event == "Wind Onshore": # if you press the wind button it will
      ↪ attach a wind model
      attachedwindon = attachedwindon+1
      if attachedwindon == 1:
         print('there is', attachedwindon, 'onshore windfarm attached')
      else:
         print('there are', attachedwindon, 'onshore windfarms attached')
   if event == "pv" or event == "Pv": # if you press the pv button it will
      ↪ attach a pv model
      attachedpv = 1 + attachedpv
      if attachedpv ==1:
         print('there is', attachedpv, 'pvfarm attached')
      else:
         print('there are', attachedpv, 'pvfarms attached')
   if event == "h2storage" or event == "H2storage": # If you press the
      ↪ h2storage button it will attach a h2storage model
      attachedh2stor = 1
      print('there is 1 hydrogen storage attached')
   if event == "electrolyser" or event == "Electrolyser": # if you press
      ↪ the electrolyser button it will attach a electrolyser model
      attachedelectrolyser = 1
      print('there is 1 electrolyser attached')
   if event == "fuelcell" or event == "Fuelcell": # if you press the
      ↪ fuelcell button it will attach a fuelcell model
      attachedfuelcell = 1
      print('there is 1 fuelcell attached')
   if event == "industry_load": # if you press the industry_load button it
      ↪ will attach a factory_load model
      attachedindustry = 1
      if event == "industry_load":
         print('there is 1 load from the industry attached')
   if event == "residential_load": # if you press the residential_load
      ↪ button it will attach a residential_load model
      attachedresident = 1
      print('there is 1 load from the residential sector attached')
   if event == "commercial_load": # if you press the commercial_load
      ↪ button it will attach a commercial_load model
      attachedcommercial = 1
      print('there is 1 load from the commercial sector attached') # if
         ↪ you press the battery button it will attached a battery model
   if event == "battery" or event == "Battery":
```

```
        attachedbat = 1
        print('there is 1 battery attached')
    if event == "In- and Export electricity":
        attachedinex = 1
        print('there is 1 import and export model attached')
    if event == "Mode 1":
        mode1 = True
        window2['-COL1-'].update(visible=False)
        layoutbuild = 3
        window2['-COL3-'].update(visible=True)
    if event == "Mode 2":
        mode1 = False
        window2['-COL1-'].update(visible=False)
        layoutbuild = 2
        window2['-COL2-'].update(visible=True)
    if event == "Load":
        attachedload = 1
        print( 'there is 1 load model attached')
    if event == "Submit" or event == "submit" or event =="Exit" or event ==
        ↪  sg.WIN_CLOSED:
        break
window2.close()
#sim_config gives all the models you can use in your simulation
if mode1 == False:
    sim_config=[['Wind' ,'python','Models.Wind.wind_mosaik:WindSim'],
                ['PV' ,'python', 'Models.PV.pv_mosaik:PvAdapter'],
                ['Collector','python', 'Models.collector:Collector'],
                ['CSVB','python', 'mosaik_csv:CSV'],
                ['Controller','python', 'Models.Controller.controller_mosaik:
                    ↪ controlSim'],
                ['Battery','python', 'Models.Battery.battery_mosaik:
                    ↪ BatteryholdSim'],
                ['Electrolyser','python', 'Models.Electrolyser.
                    ↪ electrolyser_mosaik:ElectrolyserSim'],
                ['H2storage','python', 'Models.H2storage.h2storage_mosaik:
                    ↪ compressedhydrogen'],
                ['Fuelcell','python', 'Models.Fuelcell.fuelcell_mosaik:
                    ↪ FuelCellSim'],
                ['Factory', 'python', 'Models.Factory.factory_mosaik:
                    ↪ FactorySim'],
                ['Resident', 'python', 'Models.Resident.Resident_mosaik:
                    ↪ ResidentSim'],
                ['Commercial', 'python', 'Models.Commercial.commercial_mosaik:
                    ↪ CommercialSim'],
                ['Wonshore', 'python', 'Models.Wonshore.Wonshore_mosaik:
                    ↪ WonshoreSim'],
                ['Imexport', 'python', 'Models.imexport.imexport_mosaik:
                    ↪ imexportSim'],
                ]
elif mode1:
    sim_config = [['Wind' ,'python','Models.Wind.wind_mosaik:WindSim'],
                ['PV' ,'python', 'Models.PV.pv_mosaik:PvAdapter'],
                ['Collector','python', 'Models.collector:Collector'],
                ['CSVB','python', 'mosaik_csv:CSV'],
                ['Controller','python', 'Models.Controller.controller_mosaik:
                    ↪ controlSim'],
```

```
            ['Battery','python', 'Models.Battery.battery_mosaik:
               ↪ BatteryholdSim'],
            ['Factory', 'python', 'Models.Factory.factory_mosaik:
               ↪ FactorySim']
             ]

sim_config_df=pd.DataFrame(sim_config, columns=['model','method','location
   ↪ '])
sim_config_data=sim_config_df.to_xml()
# config stores the created sim_config in a separated file
with open('config.xml','w') as file:
   file.write(sim_config_data)
# This section of code is part of the GUI that let you choose which mode
   ↪ you operate in and which models you want to use

#creats the attached list such that the models are made in the correct
   ↪ order
attached = []
for i in range(attachedwind):
   attached.append('wind[' + str(i) + ']')
for j in range(attachedpv):
   attached.append('pv[' + str(j) + ']')
if attachedbat == 1:
   attached.append('battery')
if attachedh2stor == 1:
   attached.append('h2storage')
if attachedelectrolyser == 1:
   attached.append('electrolyser')
if attachedfuelcell == 1:
   attached.append('fuelcell')
if attachedindustry == 1:
   attached.append('factory[0]')
if attachedresident == 1:
   attached.append('resident[0]')
if attachedcommercial == 1:
   attached.append('commercial[0]')
if attachedload ==1:
   attached.append('factory[0]')

for i in range(attachedwindon):
   attached.append('wonshore[' + str(i) + ']')

if attachedinex:
   attached.append('imexport[0]')

print(attached)
   #create connections list

connections= [None]*100

connections[0] = ['ctrl', 'monitor', 'dump'] #independent of what is
   ↪ connected, ctrl and monitor are present
connections[1] = ['ctrl', 'monitor', 'emissions']
connections[2] = ['ctrl', 'monitor', 'flow']
con_nr = 3 #index of connection list
e_h_conn = 0
```

```
for n in attached:
   if n == 'battery':
      connections[con_nr] = ['ctrl', 'battery', 'flow2b', 'async_requests=
         ↪ True']
      connections[con_nr + 1] = ['battery', 'monitor', 'p_out']
      connections[con_nr + 2] = ['battery', 'monitor', 'p_in']
      connections[con_nr + 3] = ['battery', 'monitor', 'mod']
      connections[con_nr + 4] = ['battery', 'monitor', 'flag']
      connections[con_nr + 5] = ['battery', 'monitor', 'soc']
      connections[con_nr + 6] = ['ctrl', 'monitor', 'flow2b']
      con_nr += 7
   elif n[:4] == 'load':
      connections[con_nr] = [n, 'ctrl', 'load_dem']
      connections[con_nr + 1] = [n, 'monitor', 'load_dem']
   elif n[:8] == 'imexport':
      connections[con_nr] = [n, 'ctrl', 'imexport_dem']
      connections[con_nr + 1] = [n, 'monitor', 'imexport_dem']
      con_nr += 2
   elif n[:2] == 'pv':
      connections[con_nr] = [n, 'ctrl', 'pv_gen']
      connections[con_nr + 1] = [n, 'monitor', 'pv_gen']
      connections[con_nr + 2] = [n, 'ctrl', 'pv_em']
      connections[con_nr + 3] = [n, 'monitor', 'pv_em']
      con_nr += 4
   elif n[:4] == 'wind' :
      connections[con_nr] = [n, 'ctrl', 'wind_gen']
      connections[con_nr + 1] = [n, 'monitor', 'wind_gen']
      connections[con_nr + 2] = [n, 'ctrl', 'wind_em']
      connections[con_nr + 3] = [n, 'monitor', 'wind_em']
      con_nr += 4
   elif n[:8] == 'wonshore' :
      connections[con_nr] = [n, 'ctrl', 'wonshore_gen']
      connections[con_nr + 1] = [n, 'monitor', 'wonshore_gen']
      connections[con_nr + 2] = [n, 'ctrl', 'wonshore_em']
      connections[con_nr + 3] = [n, 'monitor', 'wonshore_em']
      con_nr += 4
   elif n == 'electrolyser':
      connections[con_nr] = ['ctrl', 'electrolyser', 'flow2e']
      connections[con_nr + 1] = ['electrolyser', 'monitor', 'h2_gen']
      connections[con_nr + 2] = ['ctrl', 'monitor', 'flow2e']
      connections[con_nr + 3] = ['ctrl', 'electrolyser', 'h2tofa_e']
      con_nr += 4
   elif n == 'h2storage':
      connections[con_nr] = ['h2storage', 'monitor', 'h2_stored']
      connections[con_nr + 1] = ['h2storage', 'monitor', 'h2_given']
      connections[con_nr + 2] = ['h2storage', 'monitor', 'h2_stored']
      connections[con_nr + 3] = ['h2storage', 'monitor', 'h2_soc']
      connections[con_nr + 4] = ['ctrl', 'h2storage', 'h2_out', '
         ↪ async_requests=True']
      connections[con_nr + 5] = ['ctrl', 'monitor', 'h2_out']
      con_nr +=6
   elif n == 'fuelcell' :
      connections[con_nr] = ['fuelcell', 'monitor', 'fc_gen']
      connections[con_nr + 1] = ['fuelcell', 'ctrl', 'fc_gen']
   elif n[:7] == 'factory':
      connections[con_nr] = [n, 'ctrl', 'factory_dem']
```

```
        connections[con_nr + 1] = [n, 'monitor', 'factory_dem']
        con_nr += 2
    elif n[:8] == 'resident':
        connections[con_nr] = [n, 'ctrl', 'resident_dem']
        connections[con_nr + 1] = [n, 'monitor', 'resident_dem']
        con_nr += 2
    elif n[:10] == 'commercial':
        connections[con_nr] = [n, 'ctrl', 'commercial_dem']
        connections[con_nr + 1] = [n, 'monitor', 'commercial_dem']
        con_nr += 2


if ('electrolyser' in attached) and ('h2storage' in attached):
    connections[con_nr] = ['electrolyser', 'h2storage', 'h2_gen', 'h2_in']
    con_nr += 1
if ('h2storage' in attached) and ('fuelcell' in attached):
    connections[con_nr] = ['h2storage', 'fuelcell', 'h2_given', 'h2_consume
        ↪ ']
    con_nr += 1

connection = connections[:(con_nr)]


# connection stores the created connection list such that it can be used
    ↪ in different files
try:
    df=pd.DataFrame(connection, columns=['send','receive','message','more'
        ↪ ])

except ValueError:
    df=pd.DataFrame(connection, columns=['send','receive','message'])
data=df.to_xml()

with open('connection.xml','w') as file:
    file.write(data)
```

### E.4.4 interpreter.py

Listing E.7: configuration/interpreter.py

```
import shutil

src_path_1 = r"Cases/Residential Case/connection.xml"
dst_path_1 = r"configuration/connection.xml"
shutil.copy(src_path_1, dst_path_1)
print('Copied',dst_path_1)

src_path_2 = r"Cases/Residential Case/config.xml"
dst_path_2 = r"configuration/config.xml"
shutil.copy(src_path_2, dst_path_2)
print('Copied',dst_path_2)
```

## E.5 Models

### E.5.1 collector.py

Listing E.8: Models/collector.py

```
import collections
import pandas as pd
import mosaik_api
import matplotlib.pyplot as plt

META = {
   'type': 'event-based',
   'models': {
      'Monitor': {
         'public': True,
         'any_inputs': True,
         'params': [],
         'attrs': [],
      },
   },
}
import wandb
class Collector(mosaik_api.Simulator):
   def __init__(self):
      super().__init__(META)
      self.eid = None
      self.data = collections.defaultdict(lambda: collections.defaultdict(
         ↪ dict))
      # creates the graphs that update during the running of the
         ↪ simulation
      self.fig1 = None
      self.fig2 = None
      self.ax1 = None
      self.ax2 = None
      self.ax3 = None
      self.ax4 = None
      self.ax5 = None
      self.ax6 = None
      self.ax7 = None

   def init(self, sid, time_resolution, start_date, results_show,
      ↪ date_format='%Y-%m-%d %H:%M:%S',
          output_file='Result/results.csv', print_results=False,): # this
             ↪ function initializes the collector inputs
      self.time_resolution = time_resolution
      self.start_date = pd.to_datetime(start_date, format=date_format)
      self.output_file = output_file
      self.print_results = print_results
      self.results_show=results_show
      return self.meta

   def create(self, num, model): # This function creates the monitor model
      print('Collector create: hi')
      if num > 1 or self.eid is not None:
         raise RuntimeError('Can only create one instance of Monitor.')
```

```
        self.eid = 'Monitor'
        print('Collector create: bye')
        return [{'eid': self.eid, 'type': model}]

    def step(self, time, inputs, max_advance): # this function makes sure
        ↪ the data gets stored in results.csv, is send
        # to wandb and creates the graphs for every time step
        current_date = (self.start_date
                        + pd.Timedelta(time * self.time_resolution, unit='
                            ↪ seconds'))

        df_dict = {'date': current_date}

        data = inputs.get(self.eid, {})
        for attr, values in data.items():
            for src, value in values.items():
                self.data[src][attr][time] = value
                df_dict[f'{src}-{attr}'] = [value]

        df = pd.DataFrame.from_dict(df_dict)
        df = df.set_index('date')
        # wandb part
        if self.results_show['dashboard_show']==True:
            for key, value in df.items():
                wandb.log({key: value[0],
                           "custom_step":time/900})
        # csv file part
        if self.results_show['write2csv']==True:
            if time == 0:
                df.to_csv(self.output_file, mode='w', header=True)
                data = pd.read_csv('Result/results.csv')
            else:
                df.to_csv(self.output_file, mode='a', header=False)
                data = pd.read_csv('Result/results.csv')

        # graphs part
        if self.results_show['Update_graphs_during_simulation'] == True:

            if self.fig1 is None:
                plt.ion()
                self.fig1 = plt.figure()
                self.fig1.subplots_adjust(hspace=0.5, wspace=0.5)
                self.ax1 = self.fig1.add_subplot(221)
                self.ax2 = self.fig1.add_subplot(222)
                self.ax3 = self.fig1.add_subplot(223)
                self.ax4 = self.fig1.add_subplot(224)
                self.ax1.set_title('PV power')
                self.ax1.set_xlabel('time [15 min]')
                self.ax1.set_ylabel('P [kW]')
                self.ax2.set_title('Wind power')
                self.ax2.set_xlabel('time [15 min]')
                self.ax2.set_ylabel('P [kW]')
                self.ax3.set_title('electricity that is needed/over after the
                    ↪ loads are substracted from the generation')
                self.ax3.set_xlabel('time [15 min]')
```

```
            self.ax3.set_ylabel('P [kW]')
            self.ax4.set_title('electricity that is not used even after
                ↪ hydrogen storage and battery storage are charged')
            self.ax4.set_xlabel('time [15 min]')
            self.ax4.set_ylabel('P [kW]')
        if self.fig2 is None:
            plt.ion()
            self.fig2 = plt.figure()
            self.fig2.subplots_adjust(hspace=0.5, wspace=0.5)
            self.ax5 = self.fig2.add_subplot(221)
            self.ax6 = self.fig2.add_subplot(222)
            self.ax7 = self.fig2.add_subplot(223)
            self.ax5.set_title('Battery SOC')
            self.ax5.set_xlabel('time [15 min]')
            self.ax5.set_ylabel('soc [%]')
            self.ax6.set_title('Load')
            self.ax6.set_xlabel('time [15 min]')
            self.ax6.set_ylabel('Power [kW]')
            self.ax7.set_title('CO2 emissions per time frame')
            self.ax7.set_xlabel('time')
            self.ax7.set_ylabel('CO2 equivalent [tonne]')


        self.ax1.cla()
        self.ax2.cla()
        self.ax3.cla()
        self.ax4.cla()
        self.ax5.cla()
        self.ax6.cla()
        self.ax7.cla()

        pvgen = 0
        try:
            while True:
                data[f'PV-0.pv_{pvgen}-pv_gen']
                pvgen += 1
        except KeyError:
            pv_gen_sum = []

        try:
            pv_gen_sum = []
            for j in range(pvgen):
                pv_gen_sum.append(data[f'PV-0.pv_{j}-pv_gen'])
            self.ax1.plot(sum(pv_gen_sum))
        except KeyError:
            print('There is no PV model')
        wgen = 0
        try:
            while True:
                data[f'Wind-0.wind_{wgen}-wind_gen']
                wgen += 1
        except KeyError:
            wind_gen_sum = []
        try:
            wind_gen_sum = []
            for j in range(wgen):
```

```
            wind_gen_sum.append(data[f'Wind-0.wind_{j}-wind_gen'])
        self.ax2.plot(sum(wind_gen_sum))
    except KeyError:
        print('There is no Wind model')
    try:
        self.ax3.plot(data['Controller-0.ctrl_0-flow'])
    except KeyError:
        print('There is no controller model')
    try:
        self.ax4.plot(data['Controller-0.ctrl_0-dump'])
    except KeyError:
        print('There is no controller model')

    try:
        self.ax5.plot(data['Battery-0.Battery_0-soc'])
    except KeyError:
        print('There is no battery model')


    allloads = []
    try:
        allloads.append(data['Resident-0.resident_0-resident_dem'])

    except KeyError:
        pass
    try:
        allloads.append(data['Factory-0.factory_0-factory_dem'])
    except KeyError:
        pass
    try:
        allloads.append(data['Commercial-0.commercial_0-commercial_dem
            ↪ '])
    except KeyError:
        pass
    try:
        self.ax6.plot(sum(allloads))
    except KeyError:
        print('there is no load attached')

    try:
        self.ax7.plot(data['Controller-0.ctrl_0-emissions'])
    except KeyError:
        print('There is no controller model')

    self.ax1.set_title('PV power')
    self.ax1.set_xlabel('time')
    self.ax1.set_ylabel('P [kW]')
    self.ax2.set_title('Wind power')
    self.ax2.set_xlabel('time')
    self.ax2.set_ylabel('P [kW]')
    self.ax3.set_title('Electricity that is needed/over after
        ↪ Generation-loads')
    self.ax3.set_xlabel('time [15 min]')
    self.ax3.set_ylabel('P [kW]')
    self.ax4.set_title('Electricity that cannot be used')
    self.ax4.set_xlabel('time [15 min]')
```

```
            self.ax4.set_ylabel('P [kW]')
            self.ax5.set_title('Battery SOC')
            self.ax5.set_xlabel('time [15 min]')
            self.ax5.set_ylabel('soc [%]')
            self.ax6.set_title('Total load')
            self.ax6.set_xlabel('time [15 min]')
            self.ax6.set_ylabel('Power [kW]')
            self.ax7.set_title('CO2 emissions per time frame')
            self.ax7.set_xlabel('time')
            self.ax7.set_ylabel('CO2 equivalent [tonne]')

            self.fig1.canvas.draw()
            self.fig1.canvas.flush_events()
            self.fig2.canvas.draw()
            self.fig2.canvas.flush_events()
        return None

    def finalize(self): # This function lets the user know that one step is
        ↪ finished
        if self.print_results:
            print('Collected data:')
            for sim, sim_data in sorted(self.data.items()):
                print('- %s:' % sim)
                for attr, values in sorted(sim_data.items()):
                    print('  - %s: %s' % (attr, values))


if __name__ == '__main__':
    mosaik_api.start_simulation(Collector())
```

## E.6 Models/Battery

### E.6.1 battery_model.py

Listing E.9: Models/Battery/battery_model.py

```


import datetime
import pandas as pd
import matplotlib.pyplot as plt

# sign convension: -ve means discharge, +ve means Charge

class BatteryModel:
        #'battery_set',#max_p,min_p,max_energy,charge_efficiency,
            ↪ discharge_efficiency
                        #soc_min,soc_max,flag, resolution
    #'initial_set', # initial_soc

    def __init__(self, initial_set, battery_set): # this is called a method
        ↪ (since it is basically a function)
  # by using '__int__' we are making our own constructor. This helps us to
      ↪ pass all the attributes we want to the
  # object that is instantiated.
```

```
  # 'self' refers to the object calling it. Eg: If we want an object
      ↪ BatteryModel-1,
  # self helps in connecting this object with the attributes under __init__
      ↪ ()
  # everything in the bracket is a parameter. We provide aruguments (or
      ↪ values) for them.
       self.soc = initial_set['initial_soc']
       # self.i_soc = initial_set['initial-soc']
          ↪ ##########################################################
# for every object we create, soc is the attribute it gets with a value we
    ↪  provide.
       self.powerout = 0 # an attribute
       self.soc_min = battery_set['soc_min'] # an attribute
       self.soc_max = battery_set['soc_max'] # an attribute
       self.max_p = battery_set['max_p'] # kW # an attribute
       self.min_p = battery_set['min_p'] # kW # an attribute
       self.max_energy = battery_set['max_energy'] # kWh # an attribute
       self.charge_efficiency = battery_set['charge_efficiency'] # an
          ↪ attribute
       self.discharge_efficiency = battery_set['discharge_efficiency'] # an
          ↪  attribute
       self.flag = battery_set['flag'] # an attribute
       self.resolution = battery_set['resolution'] # minitues #an attribute



    # this method is called from the output_power method when conditions
       ↪ are met.
    def discharge_battery(self, flow2b): #flow2b is in kw

       hours = self.resolution / 60
       flow = max(self.min_p, flow2b)
       if (flow < 0) and (self.flag != -1):
          energy2discharge = flow * hours / self.discharge_efficiency # (-
             ↪ ve)
          energy_capacity = ((self.soc_min - self.soc) / 100) * self.
             ↪ max_energy
          if self.soc <= self.soc_min:
             self.flag = -1
             self.powerout = 0
          else:
             if energy2discharge > energy_capacity:
                # more than enough energy to discharge
                # Check if minimum energy of the battery is reached ->
                   ↪ Adjust power if necessary
                self.soc = self.soc + (energy2discharge / self.max_energy *
                   ↪  100)
                self.powerout = flow
                self.flag = 0 # Set flag as ready to discharge or charge

             else: # Fully-discharge Case
                self.powerout = energy_capacity / self.discharge_efficiency
                   ↪  / hours
                # warn('\n Home Battery is fully discharged!! Cannot
                   ↪ deliver more energy!')
                self.soc = self.soc_min
```

```
                self.flag = -1 # Set flag as 1 to show fully discharged
                    ↪ state
        self.soc = round(self.soc, 3)
        re_params = {'p_out': self.powerout,
                     # 'energy_drain': output_show,
                     # 'energy_consumed': 0,
                     'p_in': flow,
                     'soc': self.soc,
                     'mod': -1,
                     'flag': self.flag}
                     #'i_soc': self.i_soc}

        # here we are returning the values of these parameters which will be
            ↪ needed by another python model
        return re_params

    def charge_battery(self, flow2b):
        hours = self.resolution / 60
        flow = min(self.max_p, flow2b)
        if (flow > 0) and (self.flag != 1):
            energy2charge = flow * hours * self.charge_efficiency # (-ve)
            energy_capacity = ((self.soc_max - self.soc) / 100) * self.
                ↪ max_energy
            if self.soc >= self.soc_max:
                self.flag = 1
                self.powerout = 0
            else:
                if energy2charge <= energy_capacity:
                    self.soc = self.soc + (energy2charge / self.max_energy *
                        ↪ 100)
                    self.powerout = flow
                    self.flag = 0 # Set flag as ready to discharge or charge

                else: # Fully-charge Case
                    self.powerout = energy_capacity / self.charge_efficiency /
                        ↪ hours
                    # warn('\n Home Battery is fully discharged!! Cannot
                        ↪ deliver more energy!')
                    self.soc = self.soc_max
                    self.flag = 1 # Set flag as 1 to show fully discharged
                        ↪ state
        self.soc = round(self.soc, 3)
        re_params = {'p_out': self.powerout,
                     # 'energy_consumed': output_show,
                     # 'energy_drain': 0,
                     'p_in': flow,
                     'soc': self.soc,
                     'mod': 1,
                     'flag': self.flag}

        return re_params


# this method is like a controller which calls a method depending on the
    ↪ condition.
# first, this is checked. As per the p_ask and soc, everything happens.
```

```
# p_ask and soc are the parameters whos values we have to provide when we
    ↪ want to create an object of this class. i.e,
   # when we want to make a battery model.
   def output_power(self, flow2b, soc):#charging power: positive;
       ↪ discharging power:negative
      self.soc = soc # here we assign the value of soc we provide to the
          ↪ attribute self.soc
      data_ret = {}
      # {'p_out',
      # 'soc',
      # 'mod', # 0 = noaction,1 = charge,-1=discharge
      # 'flag',} # 1 means full charge, -1 means full discharge, 0 means
          ↪ available for control
   # conditions start:
      if flow2b == 0: # i.e when there isn't a demand of power at all,

         # soc can never exceed the limit so when it is equal to the max,
             ↪ we tell it is completely charged
         if self.soc >= self.soc_max:
            self.flag = 1 # meaning battery object we created is fully
                ↪ charged

         # soc can never exceed the limit so when it is equal to the min,
             ↪ we tell it is completely discharged
         elif self.soc <= self.soc_min:
            self.flag = -1

         # if the soc is between the min and max values, it is ready to be
             ↪  discharged or charged as per the situation
         else:
            self.flag = 0 # meaning it is available to operate.

         # here we are sending the current state of the battery
         re_params={'p_out': 0,
                  # 'energy_drain': 0,
                  # 'energy_consumed': 0,
                  'p_in': 0,
                   'soc': self.soc,
                   'mod': 0,
                   'flag': self.flag}
      else:
         # if the p_ask is a -ve value, it means battery needs to
             ↪ discharge.
         if flow2b < 0: #discharge

            # Can the battery discharge or not depends on the current
                ↪ state of the battery for which we call the
            # method discharge_battery. If the p_ask < 0 condition is met,
                ↪  the program directly goes the method.
            re_params = self.discharge_battery(flow2b)

         # other option is for p_ask to be > 0 which means we need to
             ↪ charge.
         else:
```

```
            # Can the battery charge depends on the current state of the
                ↪ battery for which we call the
            # method charge_battery. If the p_ask > 0 condition is met,
                ↪ the program directly goes the method.
            re_params = self.charge_battery(flow2b)


      return re_params
   # conditions end

# if __name__=='__main__':
# initial_set={'initial_soc':50}
# Battery_set = {'max_p': 2, 'min_p': -2, 'max_energy': 10,
# 'charge_efficiency': 0.9, 'discharge_efficiency': 0.9,
# 'soc_min': 10, 'soc_max': 90, 'flag': 0, 'resolution': 15}
#
#
# #max_p,min_p,max_energy,charge_efficiency,discharge_efficiency
# #soc_min,soc_max,flag, resolution
# Model=BatteryModel(initial_set,Battery_set)
# Model.charge_battery(0.1)
# Model.discharge_battery(-0.1)
# Model.output_power(0.1,95)
# Model.output_power(-0.1,95)
```

### E.6.2 battery_mosaik.py

Listing E.10: Models/Battery/battery_mosaik.py

```
# only can build one battery model
import mosaik.scheduler
import mosaik_api
from configuration.build_configuration_xml import mode1
if mode1:
      import hardwareattached
try:
   import Models.Battery.battery_model as batterymodelset
except ModuleNotFoundError:
   import battery_model as batterymodelset
else:
   import Models.Battery.battery_model as batterymodelset
import pandas as pd

#todo: convert this battery model simAPI to a controller api. This becomes
   ↪  a mosaik API to start the battery and the electrolyser.
# A condition checks the battery SOC and then initiates the electrolyser.

meta = {
   'type': 'event-based',
   'models': {
      'Batteryset': {
         'public': True,
         'params': [
            # these are the parameters which we defined in the __init__()
                ↪ of the python file
```

```
            'initial_set', # initial_soc
            'battery_set', # max_p,min_p,max_energy,charge_efficiency,
                ↪ discharge_efficiency, soc_min,soc_max
            'sim_start', # this is an additional parameter we are passing.
            'b_p_max',
            'b_p_min',
            'el_p_max'

        ],
        'attrs': [ # anything followed by self. in the python file is an
            ↪ attribute. We can have new ones too.
            'battery_id', # new attribute we provide here for the first
                ↪ time.
            # 'p_ask', # present in python file.
            'flow2b',
            'p_out', # in the python file this existed in the re_params.
                    # re_params returns values from the python file, so we
                        ↪ need to have it here so that mosaik
                    # can connect them and allow data flow.
            'p_in',
            'soc', # present in python file.
            'mod', # 0:no action, 1:charge, -1:discharge # in the python
                ↪ file this existed in the re_params.
                    # re_params returns values from the python file, so we
                        ↪ need to have it here so that mosaik
                    # can connect them and allow data flow.
            'flag', # present in the python file.
            'time',
            'energy_drain',
            'energy_consumed',
        ],
        'trigger': ['flow2b'],
    },
  },
}


class BatteryholdSim(mosaik_api.Simulator): # this is the main class that
    ↪ is running in Mosaik.
  def __init__(self):
    super().__init__(meta) # through this command we are passing more
        ↪ information about the model to the subclass we have created
        ↪ under the main
    # class - simulator

    # all these attributes are being stored in the common data flow
        ↪ reference model of Mosaik
    self.entities = {} # we store the model entity of our technology
        ↪ model
    self.eid_prefix = 'Battery_' # every entity that we create will
        ↪ start with 'Battery_
    self._cache = {}
    self._data_next = {}
    self.soc = {}
    self.flag = {}
    self.test = []
```

```
        self.pflag = []



        # this command runs only once when the simulation starts from the
            ↪ scenario file
    def init(self, sid, time_resolution, step_size=1 ): # sid and
        ↪ time_resolution are the positional arguments. Rest all we want to
        ↪ put will be keyword argument
        self.sid = sid
        self.time_resolution = time_resolution
        self.step_size = step_size
        return self.meta

    def create(self, num, model, initial_set, battery_set, sim_start):
        self.start = pd.to_datetime(sim_start)
        # next_eid=len(self.model)
        self._entities = []
        # for i in range(next_eid,next_eid+num):

        # num is the number of models of battery we want.
        for i in range(num):
            # we provide an ID to each entity we create. %s%d will be
                ↪ replaced by the values of eid_prefix and i
            self.eid = '%s%d' % (self.eid_prefix, i)

            # new instance of the battery is created
            # batterymodelset is the name we gave to the model (the battery
                ↪ python model) while importing it
            # BatteryModel is the class present in the model (the battery
                ↪ python model)
            # initial_set and battery_set are the parameters we want our
                ↪ battery_instance to have
            battery_instance = batterymodelset.BatteryModel(initial_set,
                ↪ battery_set)
            # self.model is an empty dictionary which will hold the entities
                ↪ we create. So sor every eid_b, the
            # self.model dictionary will hold a corresponding
                ↪ battery_instance !
            self.entities[self.eid] = battery_instance
            # self.battery_set[eid_b]=battery_set
            # self.battery_initial[eid_b]=initial_set

            # for every eid_b, we want to communicate the initial soc, and
                ↪ hence for every eid_b, we store the soc value.
            self.soc[self.eid] = initial_set['initial_soc']
            self.flag[self.eid] = battery_set['flag']

            # self.battery_flag={}

            # the empty entities list will hold the following information.
            self._entities.append({'eid': self.eid, 'type': model, 'rel': [],
                ↪ })
            # print(self._entities)

        return self._entities
```

```
# the step method tells the Mosaik when to initiate the next step and
    ↪ perform the calculations and repeat all the process again.
    # for the input, we need the values coming from another mosaik file.
        ↪ which means that file's output is our input.
    # the input has to be of a specific format.
    def step(self, time, inputs, max_advance):
        self.time = time
        current_time = (self.start + pd.Timedelta(time * self.
            ↪ time_resolution, unit='seconds'))
        print('from battery %%%%%%%%', current_time)
        for eid, attrs in inputs.items(): #raghav: In this model, the input
            ↪ should come from the controller p_ask
            # print(eid)
            # print(attrs)

            # In {'p_ask': {'CSVB-0.BATTEYP_0': 0.5}}, 'p_ask' is the attr,
                ↪ and 0.5 is vals
            for attr, vals in attrs.items():
                if attr == 'flow2b':
                    energy_ask = list(vals.values())[0]

                    # todo: Add conditions for checking SOC of battery and
                        ↪ making a decision
                    self._cache[eid] = self.entities[eid].output_power(
                        ↪ energy_ask, self.soc[eid])

                    # print(self._cache[eid])
                    # [p_out:,soc:,flag:]
                    self.soc[eid] = self._cache[eid]['soc']
                    self.flag = self._cache[eid]['flag']
                    check = list(self.soc.values())
                    check2 = check[0] # this is so that the value that battery
                        ↪ sends is dictionary and not a dictionary of a
                        ↪ dictionary.
                    out = yield self.mosaik.set_data({'Battery-0': {'Controller
                        ↪ -0.ctrl_0': {'soc': check2}}}) # this code is supposed
                        ↪  to hold the soc value and


        return None

# this method is used to get the specific values we want and write them in
    ↪  a new file.
    def get_data(self, outputs):
        data = {}
# # self.test.append(self.flag) # if we do this code, then we end up with
    ↪ a list which increases with each step. Duh!
# # try:
# # # the following code takes the vale at -2 position in the list. The -2
    ↪  vale of the list represents the value of the previous step
# # self.pflag = self.test[-2] # first python tries this line of code. If
    ↪ it doesnt work then it follows the code in except.
# # except:
# # self.pflag = self.flag
#
```

```
        for eid, attrs in outputs.items():
            model = self.entities[eid]
            # data['time'] = self.time
            data[eid] = {}
            for attr in attrs:
                # data[eid][attr] = getattr(model, attr) # this line of a code
                    ↪ is short form for the following code which is commented
                    ↪ out
                if attr == 'p_out':
                    data[eid][attr] = self._cache[eid]['p_out']
                elif attr == 'soc':
                    data[eid][attr] = self._cache[eid]['soc']
                    if mode1:
                        hardwareattached.setBatterySOC(self._cache[eid]['soc'
                            ↪ ]/100)
                elif attr == 'mod':
                    data[eid][attr] = self._cache[eid]['mod']
                elif attr == 'battery_id':
                    data[eid][attr] = eid
                elif attr == 'flag':
                    data[eid][attr] = self._cache[eid]['flag']
                elif attr == 'p_in':
                    data[eid][attr] = self._cache[eid]['p_in']
                # elif attr == 'energy_consumed':
                # data[eid][attr] = self._cache[eid]['energy_consumed']
                # elif attr == 'energy_drain':
                # data[eid][attr] = self._cache[eid]['energy_drain']
                # if eid in self._cache:


        return data


def main():
    mosaik_api.start_simulation(BatteryholdSim(), 'Battery-Simulator')

if __name__ == "__main__":
    main()
```

## E.7 Models/Commercial

### E.7.1 commercial_model.py

Listing E.11: Models/Commercial/commercial_model.py

```
class commercial_python:

    def __init__(self, companies, output_type):
        self.consumption = 0
        self.companies = companies #possible scaling factor
        self.output_type = output_type

    def demand(self, commercial_load):
        # incoming load is in kWh at every 15 min interval
        # incoming value of load is in kWh
```

```
        if self.output_type == 'energy':
            self.consumption = (self.companies * commercial_load) # kWh
        elif self.output_type == 'power':
            self.consumption = (self.companies * commercial_load)*4 # kW

        re_params = {'commercial_dem': self.consumption}
        return re_params
```

### E.7.2 commercial_mosaik.py

Listing E.12: Models/Commercial/commercial_mosaik.py

```
import mosaik_api
#import Load.load_model as load_model
try:
    import Models.Commercial.commercial_model as commercial_model
except ModuleNotFoundError:
    import commercial_model as commercial_model
else:
    import Models.Commercial.commercial_model as commercial_model

import pandas as pd

META = {
    'type': 'event-based',

    'models': {
        'commercialmodel': {
            'public': True,
            'params': ['sim_start', 'companies', 'output_type'],
            'attrs': ['commercial_id','commercial_dem','commercial_load',], #
                ↪ third attribute here must match the second attribute in the
                ↪ second line of the txt file containing data
            'trigger': [],
        },
    },
}


class CommercialSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'commercial_' # every entity that we create will
            ↪ start with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
    # we can use this to pass additional initialization tasks
```

```
    def init(self, sid, time_resolution):
        #print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        #print(sid) # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        #print('hi, you have entered create of SimAPI') # working (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = commercial_model.commercial_python(**
                ↪ model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):

        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution,
                                     unit='seconds')) # timedelta represents a
                                        ↪ duration of time
        print('from commercial %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            #print(eid)
            # print(attrs)
            for attr, vals in attrs.items():
                # if attr == 'u':
                commercial_loads = list(vals.values())[0]
                # print(u)
                self._cache[eid] = self.entities[eid].demand(commercial_loads)
                    ↪  # not necessary to have u in brackets. It is not
                    ↪ necessary to keep the same name as the one in python file
                # print(self._cache)
        return None

    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            data[eid] = {}
            for attr in attrs:
                if attr == 'commercial_dem':
                    data[eid][attr] = self._cache[eid]['commercial_dem']
        return data
def main():
    mosaik_api.start_simulation(CommercialSim(), 'Commercial Simulator')

if __name__ == "__main__":
```

```
main()
```

## E.8 Models/Controller

### E.8.1 controller_model.py

Listing E.13: Models/Controller/controller_model.py

```
from configuration.build_configuration_xml import *
class controller_python:
    def __init__(self, soc_min, soc_max, b_p_max, b_p_min, el_p_max,
        ↪ h2_soc_min, h2_soc_max, fc_eff, fossil_em_var, fossil_em_const,
        ↪ h2tofa_em_saved, max_fah2_dem):
        self.soc_max_b = soc_max
        self.soc_min_b = soc_min
        self.soc_max_h2 = h2_soc_max
        self.soc_min_h2 = h2_soc_min
        self.fc_eff = fc_eff
        self.foss_em_c = fossil_em_const
        self.foss_em_v = fossil_em_var
        self.b_p_max = b_p_max
        self.b_p_min = b_p_min
        self.el_p_max = el_p_max
        self.fah2_dem_max = max_fah2_dem #kg/15min
        self.h2tofa_em_saved = h2tofa_em_saved #kgco2/kgh2
        self.flow_e_fa_max = max_fah2_dem * 286.6*1000 / (60*15*2.02*0.6) #
            ↪ kW, maximal flow needed for generation of factory hydrogen



        self.dump = 0
        self.flow_b = 0
        self.flow_e = 0
        self.fc_out = 0
        self.current_em = 0
        self.total_em = 0


    def control(self, load_dem, wind_gen, wonshore_gen, pv_gen, factory_dem
        ↪ , res_dem, commercial_dem, imexport_dem, soc, h2_soc, fc_gen,
        ↪ wind_em, wonshore_em, pv_em):#, fc_gen):
    # def control(self,soc , pv_gen, load_dem, wind_gen):
        self.soc_b = soc
        self.soc_h = h2_soc
        self.current_em = 0
        flow = wind_gen + wonshore_gen + pv_gen - load_dem - factory_dem -
            ↪ res_dem - commercial_dem + imexport_dem# kW
        res_p = flow
        constant_em = wind_em + pv_em + wonshore_em + self.foss_em_c
        self.current_em += constant_em # g
        h2tofa_e = 0 # init
        self.h_out = 0 # init
        self.dump = 0 #init
```

```
if res_p < 0: # means that the demand is not completely met and we
    ↪ need support from battery and fuel cell
    self.flow_e = 0 # no hydrogen will be created

    if attachedbat == 1: # battery must be attached, for energy to be
        ↪ recovered from it
        if self.soc_b > self.soc_min_b: # checking if battery soc is
            ↪ above minimum. It can be == to soc_max as well.
            if res_p >= self.b_p_min: # enough power can be supplied by
                ↪ the battery
                self.flow_b = res_p # all power is delivered by battery

            else:
                self.flow_b = self.b_p_min # maximum power is supplied
                    ↪ by battery

        elif self.soc_b <= self.soc_min_b: # battery is empty
            self.flow_b = 0 # no power is drawn from battery
            print('Battery Discharged')
    else: # no battery, no power
        self.flow_b = 0

    fossil_flow = res_p - self.flow_b # the needed power that can not
        ↪ be met by the battery will be supplemented with fossil fuel
        ↪ plants

    self.current_em -= fossil_flow * self.foss_em_v # which has
        ↪ influence on emissions

    if attachedh2stor == 1: # h2storage must be present for it to
        ↪ send h2 to factories
        if h2_soc > self.soc_min_h2: #hydrogen present in storage
            self.h_out = -self.fah2_dem_max #kg



elif res_p > 0: # means we have over generation and we want to
    ↪ utilize it for charging battery and creating hydrogen

    if attachedbat == 1: # battery must be attached to store energy
        ↪ in it
        if self.soc_b < self.soc_max_b: # battery not already fully
            ↪ charged
            if res_p <= self.b_p_max: # power below maximum battery
                ↪ charging rate
                self.flow_b = res_p # battery gets charged at res_p
                self.flow_e = 0
                res_p =0 # no power left
            else: # power above max charge capacity
                self.flow_b = self.b_p_max # battery gets charged at max
                    ↪ charge capacity
                res_p = res_p - self.b_p_max # residual power flow
        else:
            self.flow_b = 0 # if charged, nothing happens with battery,
                ↪ power is passed on
```

```
        else:
            self.flow_b = 0 # no battery no charging

        if attachedelectrolyser == 1: # electrolyser must be present to
            ↪ create hydrogen
            if (self.soc_h < self.soc_max_h2) and attachedh2stor == 1: #
                ↪ storage present and not full
                self.flow_e = min(res_p,self.el_p_max) # hydrogen is formed
                    ↪  at either maximum generation or the residual power,
                    ↪ whichever is lowest
            else:
                self.flow_e = min(self.flow_e_fa_max, self.el_p_max, res_p)
                    ↪  # hydrogen is formed at either maximum generation or
                    ↪ residual power, but limited to the capacity of
                    ↪ hydrogen that can be send to factories.
        else:
            self.flow_e = 0 # no electrolyser, no hydrogen formed
        self.dump = res_p - self.flow_e # residual power gets dumped


        h2potential = (self.flow_e * 60 * 15 / 286.6 * 2.02 / 1000 * 0.6)
            ↪  # amount of kg hydrogen formed with flow to electrolyser
            ↪ per 15 minutes
        # kW*60s*15min /(kJ/mol) *(g/mol) to kg *eff
        h2tofa_e = min(h2potential, self.fah2_dem_max) # kg/15min, limit
            ↪ amount sent to factories to its max

        if attachedh2stor:
            if h2tofa_e < self.fah2_dem_max: # electrolyser does not
                ↪ deliver max to factories
                if h2_soc > self.soc_min_h2: #if h2 present
                    self.h_out = -self.fah2_dem_max+h2tofa_e #rest is
                        ↪ released from storage



    #print(self.flow_e, h2potential, self.fah2_dem_max, h2tofa_e, 'h2')
    self.current_em += (self.h_out - h2tofa_e) * self.h2tofa_em_saved #
        ↪ take away emissions saved by hydrogen to factories kg h2 *
        ↪ kgco2/kgh2
    self.total_em +=self.current_em
    re_params = {'flow2b': self.flow_b, 'flow2e': self.flow_e, 'dump':
        ↪ self.dump, 'h2_out':self.h_out,
            'emissions':self.current_em, 'flow': flow, 'h2tofa_e':
                ↪ h2tofa_e}

    return re_params

"""Old controller

def control(self, wind_gen, pv_gen, load_dem, factory_dem, res_dem, soc
    ↪ , h2_soc, wind_em, pv_em):#, fc_gen):
    self.soc_b = soc
    self.soc_h = h2_soc
    flow = wind_gen + pv_gen - load_dem - factory_dem - res_dem -
        ↪ commercial_dem# kW
```

```
        added_em = wind_em + pv_em + self.foss_em_c
        self.total_em += added_em #g
        if flow < 0:
            self.total_em -= flow*self.foss_em_v

        if flow < 0: # means that the demand is not completely met and we
            ↪ need support from battery and fuel cell
            if self.soc_b > self.soc_min_b: # checking if soc is above
                ↪ minimum. It can be == to soc_max as well.
                self.flow_b = flow
                self.flow_e = 0
                self.h_out = 0

            elif self.soc_b <= self.soc_min_b:
                self.flow_b = 0
                q = 39.4
                self.h_out = (flow / q) / self.fc_eff

                print('Battery Discharged')


        elif flow > 0: # means we have over generation and we want to
            ↪ utilize it for charging battery and storing hydrogen
            if self.soc_b < self.soc_max_b:
                self.flow_b = flow
                self.flow_e = 0
                self.h_out = 0
            elif self.soc_b == self.soc_max_b:
                self.flow_b = 0
                if self.soc_h < self.soc_max_h2:
                    self.flow_e = flow
                    self.dump = 0
                    self.h_out = 0
                elif self.soc_h == self.soc_max_h2:
                    self.flow_e = 0
                    self.dump = flow
                    self.h_out = 0

        re_params = {'flow2b': self.flow_b, 'flow2e': self.flow_e, 'dump':
            ↪ self.dump, 'h2_out':self.h_out, 'emissions':self.total_em, '
            ↪ flow': flow}
        return re_params
    """
```

### E.8.2 controller_mosaik.py

Listing E.14: Models/Controller/controller_mosaik.py

```
import mosaik_api
import pandas as pd
#import Controller.controller_model as controller_model
try:
    import Models.Controller.controller_model as controller_model
except ModuleNotFoundError:
    import controller_model as controller_model
```

```
else:
   import Models.Controller.controller_model as controller_model
#import Battery.model as batterymodel
import sys
sys.path.insert(1,'/home/illuminator/Desktop/Final_illuminator')

try:
   import Models.Battery.battery_model as batterymodel
except ModuleNotFoundError:
   import battery_model as batterymodel
else:
   import Models.Battery.battery_model as batterymodel

import itertools
META = {
   'type': 'event-based',
   # wind is an event based event because the event here is a wind speed.
      ↪ It doesnt purely run because of time interval, I think.
   # if I put it to time-based, there is type error:
   # File "C:\Users\ragha\AppData\Local\Programs\Python\Python310\lib\site
      ↪ -packages\mosaik\scheduler.py", line 405, in step
   # sim.progress_tmp = next_step - 1
   # TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'

   'models': {
      'Ctrl': {
         'public': True,
         'params': ['sim_start', 'b_p_max', 'b_p_min', 'el_p_max', '
            ↪ soc_min', 'soc_max', 'h2_soc_min', 'h2_soc_max', 'fc_eff', '
            ↪ fossil_em_const',
                 'fossil_em_var', 'h2tofa_em_saved', 'max_fah2_dem'],

         'attrs': ['controller_id', 'flow2e', 'flow2b', 'wind_gen', '
            ↪ wonshore_gen', 'load_dem', 'factory_dem', 'pv_gen', 'soc',
                'h2_soc', 'dump', 'h2_out', 'wind_em', 'wonshore_em', '
                   ↪ pv_em', 'emissions', 'resident_dem' , '
                   ↪ commercial_dem',
                'flow', 'fc_gen','h2tofa_e','imexport_dem'
                ],
         'trigger': [],
      },
   },
}


class controlSim(mosaik_api.Simulator):
   def __init__(self):
      super().__init__(META)
      self.eid_prefix = 'ctrl_' # every entity that we create will start
         ↪ with 'wind_'
      self.entities = {} # we store the model entity of our technology
         ↪ model
      self._cache = {} # used in the step function to store the values
         ↪ after running the python model of the technology
      self.soc_max = {}
      self.temp = 0
```

```
        self.h2fc = {}
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
    # we can use this to pass additional initialization tasks

    def init(self, sid, time_resolution, step_size=1):
        self.step_size = step_size
        self.sid = sid
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        self._entities = []
        # print(next_eid) # working (20220524)

        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = controller_model.controller_python(**
                ↪ model_params) #1
            self.entities[eid] = model_instance #2
            # print(self.entities)
            # self.soc_max[eid] = soc_max
            self._entities.append({'eid': eid, 'type': model})
        return self._entities

    def step(self, time, inputs, max_advance):
        # inputs is a dictionary, which contains another dictionary.
        # print(inputs)
        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution,
                                     unit='seconds')) # timedelta represents a
                                        ↪ duration of time
        print('from controller %%%%%%%%%', current_time)
        _cache = {}
        for eid, attrs in inputs.items():
            # print('#eid: ', eid)
            #print('#attrs: ', attrs)
            w = 0
            won = 0
            p = 0
            l = 0
            fa = 0
            res = 0
            comer = 0
            ie=0
            w_e = 0
```

```
        won_e = 0
        p_e = 0
        fc = 0
        for attr, vals in attrs.items():
            #print('#attr: ', attr)
            #print('#vals: ', vals)
            # s=0
            # h=0
            ###################################
            if attr == 'load_dem': #use sum(vals.values()) if multiple
                ↪ enitities possible
                l = sum(vals.values())
            elif attr == 'pv_gen':
                p = sum(vals.values())
            elif attr == 'wind_gen':
                w = sum(vals.values())
            elif attr == 'wonshore_gen':
                won = sum(vals.values())
            elif attr == 'factory_dem':
                fa = sum(vals.values())
            elif attr == 'resident_dem':
                res = sum(vals.values())
            elif attr == 'imexport_dem':
                ie = sum(vals.values())
            elif attr == 'wonshore_gen':
                won = sum(vals.values())
            elif attr == 'soc':
                s = list(vals.values())[0] #use list(vals.values())[0] if
                    ↪ only one entity possible
            elif attr == 'h2_soc':
                h = list(vals.values())[0]
            elif attr == 'fc_gen':
                fc = list(vals.values())[0]
            elif attr == 'wind_em':
                w_e = sum(vals.values())
            elif attr == 'wonshore_em':
                won_e = sum(vals.values())
            elif attr == 'pv_em':
                p_e = sum(vals.values())
        try:
            _cache[eid] = self.entities[eid].control(l, p, w, won, fa, res
                ↪ , comer, ie, s, h, fc, w_e, won_e, p_e)

        except:

            s = 50
            h = 0

            _cache[eid] = self.entities[eid].control(l, p, w, won, fa, res
                ↪ , comer, ie, s, h, fc, w_e, won_e, p_e)
        self._cache = _cache
    #print(l,p,w,won,fa,res,comer,ie,s,h,w_e,won_e,p_e)
    return None

def get_data(self, outputs):
    data = {}
```

```
        for eid, attrs in outputs.items():
            # model = self.entities[eid]
            # data['time'] = self.time
            data[eid] = {}
            for attr in attrs:
                # if we want more values to print in the output file, mimic
                    ↪ the below for new attributes and make sure
                # those parameters are present in the re_params in the python
                    ↪ file of the technology
                if attr == 'flow2b': # e2b = energy to battery
                    data[eid][attr] = self._cache[eid]['flow2b']
                elif attr == 'flow2e': # flow2e = energy to electrolyser
                    data[eid][attr] = self._cache[eid]['flow2e']
                elif attr == 'dump':
                    data[eid][attr] = self._cache[eid]['dump']
                elif attr == 'h2_out':
                    data[eid][attr] = self._cache[eid]['h2_out']
                elif attr == 'emissions':
                    data[eid][attr] = self._cache[eid]['emissions']
                elif attr == 'flow':
                    data[eid][attr] = self._cache[eid]['flow']
                elif attr == 'h2tofa_e':
                    data[eid][attr] = self._cache[eid]['h2tofa_e']
                # print(data)
        return data
def main():
    mosaik_api.start_simulation(controlSim(), 'Controller-Illuminator')
if __name__ == '__main__':
    main()
```

## E.9 Models/Electrolyser

### E.9.1 electrolyser_model.py

Listing E.15: Models/Electrolyser/electrolyser_model.py

```
class electrolyser_python:
    def __init__(self, eff, resolution):
        self.p_in = None
        self.h_out = None
        self.eff = eff
        self.resolution = resolution

    def electrolyser(self, flow2e, h2tofa_e):

        if flow2e > 0:
            self.p_in = flow2e # in kW
            conversion = self.p_in * self.resolution * 60 # kJ # since we are
                ↪  gentting power for 15mins, hence we convert kW to kJ by
                ↪ multiplying with 15mins * 60 seconds in a minute
            hhv = 286.6 # kJ/mol, higher heating value
            mole = conversion / hhv * self.eff # gives number of moles of
                ↪ hydrogen
            out = 2.02 * mole # weight of hydrogen is 2.02 grams/mole
            h_created = out / 1000 # kg of hydrogen produced
```

```
            self.h_out = h_created - h2tofa_e #amount of hydrogen stored
            if abs(self.h_out) < 0.0001: #get rid of small rounding errors
                self.h_out = 0 #kg
        # elif flow2e < 0:
        # q = 39.4 # kW/Kg # this is the amount of power that can be
            ↪ generated from 1 kg of Hydrogen or kWh of energy in 1 hour from
            ↪  1 kg hydrogen. The power remains constant for a period of 1hr
        #
        # self.h_out = (flow2e / q) / self.fc_eff # Kg
        #
        # re_params = {'h2_out': self.h_out, 'h2_gen': 0}
        else:
            self.h_out = 0

        re_params = {'h2_gen': self.h_out} #, 'h2_out': 0}

        return re_params
```

## E.9.2 electrolyser_mosaik.py

Listing E.16: Models/Electrolyser/electrolyser_mosaik.py

```
import mosaik_api
#import Electrolyser.electrolyser_model as electrolyser_model
try:
    import Models.Electrolyser.electrolyser_model as electrolyser_model
except ModuleNotFoundError:
    import electrolyser_model as electrolyser_model
else:
    import Models.Electrolyser.electrolyser_model as electrolyser_model

import pandas as pd

META = {
    'type': 'event-based',
    # wind is an event based event because the event here is a wind speed.
        ↪ It doesnt purely run because of time interval, I think.
    # if I put it to time-based, there is type error:
    # File "C:\Users\ragha\AppData\Local\Programs\Python\Python310\lib\site
        ↪ -packages\mosaik\scheduler.py", line 405, in step
    # sim.progress_tmp = next_step - 1
    # TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'

    'models': {
        'electrolysermodel': {
            'public': True,
            'params': ['sim_start', 'eff', 'fc_eff', 'resolution'],
            'attrs': ['electro_id','h2_gen', 'flow2e', 'fc_eff', 'h2_out','
                ↪ h2tofa_e'],
            'trigger': [],
        },
    },
}
```

```
class ElectrolyserSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'electro_' # every entity that we create will
            ↪ start with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = electrolyser_model.electrolyser_python(**
                ↪ model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})

        return entities

    def step(self, time, inputs, max_advance):

        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution,
                                     unit='seconds')) # timedelta represents a
                                        ↪ duration of time
        # print('from electrolyser %%%%%%%%%',current_time)

        for eid, attrs in inputs.items():
            # print(eid)
            # print(attrs)
            for attr, vals in attrs.items():
                if attr == 'flow2e':
                    flow2e = list(vals.values())[0]
                elif attr == 'h2tofa_e':
                    h2tofa_e = list(vals.values())[0]
            self._cache[eid] = self.entities[eid].electrolyser(flow2e,
                ↪ h2tofa_e) # not necessary to have u in brackets. It is not

        return None
```

```
   def get_data(self, outputs):
      data = {}
      for eid, attrs in outputs.items():
         data[eid] = {}
         for attr in attrs:
            if attr == 'h2_gen':
               data[eid][attr] = self._cache[eid]['h2_gen']
            # if attr == 'h2_out':
            # data[eid][attr] = self._cache[eid]['h2_out']
      return data


def main():
   mosaik_api.start_simulation(ElectrolyserSim(), 'Electrolyser-Simulator'
      ↪ )

if __name__ == "__main__":
   main()
```

## E.10 Models/Factory

### E.10.1 factory_model.py

Listing E.17: Models/Factory/factory_model.py

```
class factory_python:

   def __init__(self, factories, output_type):
      self.consumption = 0
      self.factories = factories #possible scaling factor
      self.output_type = output_type

   def demand(self, factory_load):
      # incoming load is in kWh at every 15 min interval
      # incoming value of load is in kWh

      if self.output_type == 'energy':
         self.consumption = (self.factories * factory_load) # kWh
      elif self.output_type == 'power':
         self.consumption = (self.factories * factory_load)*4 # kW

      re_params = {'factory_dem': self.consumption}
      return re_params
```

### E.10.2 factory_mosaik.py

Listing E.18: Models/Factory/factory_mosaik.py

```
import mosaik_api
from configuration.build_configuration_xml import mode1
if mode1:
   import hardwareattached
#import Factory.factory_model as factory_model
```

```
try:
   import Models.Factory.factory_model as factory_model
except ModuleNotFoundError:
   import factory_model as factory_model
else:
   import Models.Factory.factory_model as factory_model

import pandas as pd

META = {
   'type': 'event-based',

   'models': {
      'factorymodel': {
         'public': True,
         'params': ['sim_start', 'factories', 'output_type'],
         'attrs': ['factory_id','factory_dem','factory_load',], #third
            ↪ attribute here must match the second attribute in the second
            ↪  line of the txt file containing data
         'trigger': [],
      },
   },
}


class FactorySim(mosaik_api.Simulator):
   def __init__(self):
      super().__init__(META)
      self.eid_prefix = 'factory_' # every entity that we create will
         ↪ start with 'factory_'
      self.entities = {} # we store the model entity of our technology
         ↪ model
      self._cache = {} # used in the step function to store the values
         ↪ after running the python model of the technology
      # self.start_date = None

   # the following API call is will be called only once when we initiate
      ↪ the model in the scenario file.
   # we can use this to pass additional initialization tasks

   def init(self, sid, time_resolution):
      # print('hi, you have entered init') # working (20220524)
      self.time_resolution = time_resolution
      # print('Exited init os SimAPI') # working (20220524)
      return self.meta

   def create(self, num, model, sim_start, **model_params):
      # print('hi, you have entered create of SimAPI') # working
         ↪ (20220524)
      self.start = pd.to_datetime(sim_start)
      # print(type(self.entities))
      # next_eid = len(self.entities)
      # print('from create of SimAPI:', next_eid)
      entities = []
      # print(next_eid) # working (20220524)
```

```
        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = factory_model.factory_python(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):
        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution,
                                     unit='seconds')) # timedelta represents a
                                        ↪ duration of time
        print('from factory %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            # print(eid)
            for attr, vals in attrs.items():
                # if attr == 'u':
                factory_load = list(vals.values())[0]
                # print(u)
                self._cache[eid] = self.entities[eid].demand(factory_load) #
                    ↪ not necessary to have u in brackets. It is not necessary
                    ↪ to keep the same name as the one in python file
                # print(self._cache)
        return None

    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            data[eid] = {}
            for attr in attrs:
                if attr == 'factory_dem':
                    data[eid][attr] = self._cache[eid]['factory_dem']
                    if mode1:
                        hardwareattached.setLoad(self._cache[eid]['factory_dem'
                            ↪ ]/(7784959*4))
        return data


def main():
    mosaik_api.start_simulation(FactorySim(), 'factory Simulator')


if __name__ == "__main__":
    main()
```

## E.11 Models/Fuelcell

### E.11.1 fuelcell_model.py

Listing E.19: Models/Fuelcell/fuelcell_model.py

```
class fuelcell_python:
    def __init__(self, eff):
```

```
        self.p_in = None
        self.p_out = None
        self.eff = eff

    def output(self, h2_consume):
        # hydrogen input will be in Kg. Output will be electricity using the
            ↪ HHv value of hydrogen
        energy = 39.4 # kWh generated from 1 Kg hydrogen in 1 hour
        power = energy/60 # kW
        out = (-h2_consume*power*self.eff*15) #KWh generated from fuelcell
            ↪ taking into consideration its efficiency
        re_params = {'fc_gen': out}
        return re_params
```

### E.11.2 fuelcell_mosaik.py

Listing E.20: Models/Fuelcell/fuelcell_mosaik.py

```
import mosaik_api
import numpy as np
import pandas as pd

#import Fuelcell.fuelcell_model as fc
try:
    import Models.Fuelcell.fuelcell_model as fc
except ModuleNotFoundError:
    import fuelcell_model as fc
else:
    import Models.Fuelcell.fuelcell_model as fc


meta = {
    'type': 'event-based',
    #wind is an event based event because the event here is a wind speed.
        ↪ It doesnt purely run because of time interval, I think.
    # if I put it to time-based, there is type error:
    # File "C:\Users\ragha\AppData\Local\Programs\Python\Python310\lib\site
        ↪ -packages\mosaik\scheduler.py", line 405, in step
    # sim.progress_tmp = next_step - 1
    # TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'

    'models': {
        'fuelcellmodel': {
            'public': True,
            'params': ['sim_start', 'eff'],
            'attrs': ['h2_consume', 'fc_gen'],
            'trigger': [],
        },
    },
}

class FuelCellSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(meta)
```

```
        self.eid_prefix = 'fc_' # every entity that we create will start
            ↪ with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None
        self.fc_gen = {}

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range (num):
            eid = '%s%d' % (self.eid_prefix, i)
            # print('########### FC id', eid)
            model_instance = fc.fuelcell_python(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):
        self.time = time
        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution, unit='seconds'
                            ↪ )) # timedelta represents a duration of time
        print('from fc %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            # raghav: Inputs come from a controller. This happens when you
                ↪ define the connection in the scenario file
            # print(eid)
            # print(attrs)
            for attr, vals in attrs.items():
                if attr == 'h2_consume':
                    h2_consume = list(vals.values())[0]
                    # print(h2_consume)
                    self._cache[eid] = self.entities[eid].output(h2_consume)
                    # in the above line, we have called our entity of fuelcell
                        ↪ model we created in create followed by a
                    # definition 'output. Since self.entities = model_instance
                        ↪ and model instance calls out fuelcell
                    # python model, so in this step we are called the function
                        ↪ 'output' and give it a value for h2_in.
```

```
                # This step makes the python file run and do the
                    ↪ calculations for us of wind_gen.
                # print(self._cache[eid])
                # self.fc_gen[eid] = self._cache[eid]['fc_gen']
                # fc_gen_val = list(self.fc_gen.values())
                # fc_gen = fc_gen_val[0]
                # print('++++++++++++++++++++', fc_gen)
                # out = yield self.mosaik.set_data({'fc-0': {'ctrl-0.ctrl_0
                    ↪ ': {'fc_gen': fc_gen}}})
        return None

    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            data[eid] = {}
            for attr in attrs:
                # if we want more values to print in the output file, mimic
                    ↪ the below for new attributes and make sure
                # those parameters are present in the re_params in the python
                    ↪ file of the technology
                if attr == 'fc_gen':
                    data[eid][attr] = self._cache[eid]['fc_gen']

        return data

def main():
    mosaik_api.start_simulation(FuelCellSim(), 'FuelCell Simulator')

if __name__ == "__main__":
    main()
```

## E.12 Models/H2storage

### E.12.1 h2storage_model.py

Listing E.21: Models/H2storage/h2storage_model.py

```
# compressed hydrogen storage tank at 700bar and storing about 100kg of
    ↪ hydrogen

class hydrogenstorage_python:
    def __init__(self, initial_set, h2_set):
        self.h2storage_soc = initial_set['initial_soc']
        self.h2storage_soc_min = h2_set['h2storage_soc_min'] # an attribute
        self.h2storage_soc_max = h2_set['h2storage_soc_max'] # an attribute
        self.p_in = None
        self.p_out = None
        self.eff = 0.94 # approx efficiency of compressed hydrogen storage
            ↪ tanks. Roundtrip efficiency
        self.max_h2 = h2_set['max_h2']
        self.min_h2 = h2_set['min_h2']
        self.capacity=h2_set['capacity']
        self.flag=0

    def charge_h2(self, h2_in):
```

```
        h2_flow = min(self.max_h2, h2_in) # kg/15min
        if (h2_flow > 0) and (self.flag != 1):
            self.h2discharge = h2_flow * self.eff # (+ve) We multiply it with
                ↪  efficiency because while charging we are not 100% efficient
                ↪ , so we should end up with less energy than incoming
            h2_capacity = ((self.h2storage_soc_max - self.h2storage_soc) /
                ↪ 100) * self.capacity # gives amount of energy that can be
                ↪ stored in the battery
            if self.h2storage_soc >= self.h2storage_soc_max:
                self.flag = 1
                self.h2out = 0
                output_show = 0
            else:
                if self.h2discharge <= h2_capacity:
                    h2_consumed = self.h2discharge
                    self.h2storage_soc = self.h2storage_soc + (self.h2discharge
                        ↪  / self.capacity * 100)
                    # self.powerout = 0 # because we are consuming the incoming
                        ↪  energy and nothing is going out
                    self.flag = 0 # Set flag as ready to discharge or charge
                    output_show = h2_flow # just for showing purpose that what
                        ↪ ever is extra is being consumed while internally, a
                        ↪ bit less is used to charge battery because of losses

                else: # Fully-charge Case
                    h2_consumed = h2_capacity / self.eff # because to reach
                        ↪ full soc, it needs more energy considering the losses.
                        ↪  Since now we have the option to draw in more energy
                        ↪ because it is surplus, we can do this, I think
                    h2_excess = self.h2discharge - h2_consumed
                    output_show = h2_consumed
                    # self.powerout = 0
                    # warn('\n Home Battery is fully discharged!! Cannot
                        ↪ deliver more energy!')
                    self.h2storage_soc = self.h2storage_soc_max
                    self.flag = 1 # Set flag as 1 to show fully discharged
                        ↪ state
        else:#if (h2_flow == 0)# and (self.flag == 1):
            output_show = 0
        self.h2storage_soc = round(self.h2storage_soc, 3)
        re_params = {
                    # 'p_out': self.powerout,
                    'h2_stored': output_show,
                    'h2_given': 0,
                    'h2storage_soc_min': self.h2storage_soc_min,
                    'h2storagesoc_max': self.h2storage_soc_max,
                    # 'energy_drain': 0,
                    'h2_soc': self.h2storage_soc,
                    'mod': 1,
                    'flag': self.flag}

        return re_params

    def discharge_h2(self, h2_out):
        h2_flow = max(self.min_h2, h2_out) # kg
        if (h2_flow < 0) and (self.flag != -1):
```

```
            self.h2discharge = h2_flow / self.eff # (+ve) We multiply it with
                ↪  efficiency because while charging we are not 100% efficient
                ↪ , so we should end up with less energy than incoming
            h2_capacity = ((self.h2storage_soc_min - self.h2storage_soc) /
                ↪ 100) * self.capacity # gives amount of energy that can be
                ↪ stored in the battery
            if self.h2storage_soc <= self.h2storage_soc_min:
                self.flag = -1
                self.h2out = 0
                output_show = 0
            else:
                if self.h2discharge > h2_capacity:
                    h2_given = self.h2discharge
                    self.h2storage_soc = self.h2storage_soc + (self.h2discharge
                        ↪  / self.capacity * 100)
                    # self.powerout = 0 # because we are consuming the incoming
                        ↪  energy and nothing is going out
                    self.flag = 0 # Set flag as ready to discharge or charge
                    output_show = h2_flow # just for showing purpose that what
                        ↪ ever is extra is being consumed while internally, a
                        ↪ bit less is used to charge battery because of losses

                else: # Fully-discharge Case
                    h2_given = h2_capacity * self.eff
                    h2_excess = self.h2discharge - h2_given
                    output_show = h2_given
                    # self.powerout = 0
                    # warn('\n Home Battery is fully discharged!! Cannot
                        ↪ deliver more energy!')
                    self.h2storage_soc = self.h2storage_soc_min
                    self.flag = -1 # Set flag as 1 to show fully discharged
                        ↪ state
        else:
            output_show = 0
        self.h2storage_soc = round(self.h2storage_soc, 3)
        re_params = {
            # 'p_out': self.powerout,
            'h2_stored': 0,
            'h2_given': output_show,
            'h2storage_soc_min': self.h2storage_soc_min,
            'h2storage_soc_max': self.h2storage_soc_max,
            # 'energy_drain': 0,
            'h2_soc': self.h2storage_soc,
            'mod': 1,
            'flag': self.flag}

        return re_params

    def output_h2(self, h2_in, h2_out, soc): # charging power: positive;
        ↪ discharging power:negative
        self.h2storage_soc = soc # here we assign the value of soc we
            ↪ provide to the attribute self.soc
        data_ret = {}
        # {'p_out',
        # 'soc',
        # 'mod', # 0 = noaction,1 = charge,-1=discharge
```

```
        # 'flag',} # 1 means full charge, -1 means full discharge, 0 means
            ↪ available for control
        # conditions start:
        if h2_in == 0 and h2_out == 0: # i.e when there isn't a demand of
            ↪ power at all,

            # soc can never exceed the limit so when it is equal to the max,
                ↪ we tell it is completely charged
            if self.h2storage_soc >= self.h2storage_soc_max:
                self.flag = 1 # meaning battery object we created is fully
                    ↪ charged

            # soc can never exceed the limit so when it is equal to the min,
                ↪ we tell it is completely discharged
            elif self.h2storage_soc <= self.h2storage_soc_min:
                self.flag = -1

            # if the soc is between the min and max values, it is ready to be
                ↪  discharged or charged as per the situation
            else:
                self.flag = 0 # meaning it is available to operate.

            # here we are sending the current state of the battery
            re_params = {
                      # 'h2_out': 0,
                      'h2_soc': self.h2storage_soc,
                      'h2storage_soc_min': self.h2storage_soc_min,
                      'h2storage_soc_max': self.h2storage_soc_max,
                      'h2_stored': 0,
                      'h2_given': 0,
                      'mod': 0,
                      'flag': self.flag}
        elif h2_out < 0: # discharge
            re_params = self.discharge_h2(h2_out)

        else:
            re_params = self.charge_h2(h2_in)

        return re_params
```

## E.12.2 h2storage_mosaik.py

Listing E.22: Models/H2storage/h2storage_mosaik.py

```
import mosaik_api
#import H2storage.h2storage_model as hydrogen_storage
try:
    import Models.H2storage.h2storage_model as hydrogen_storage
except ModuleNotFoundError:
    import h2storage_model as hydrogen_storage
else:
    import Models.H2storage.h2storage_model as hydrogen_storage
import pandas as pd

META = {
```

```
    'type': 'event-based',
        # storage is an event based event because the event here is a wind
            ↪ speed. It doesnt purely run because of time interval, I think.
        # if I put it to time-based, there is type error:
        # File "C:\Users\ragha\AppData\Local\Programs\Python\Python310\lib\
            ↪ site-packages\mosaik\scheduler.py", line 405, in step
        # sim.progress_tmp = next_step - 1
        # TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'

    'models': {
        'compressed_hydrogen': {
            'public': True,
            'params': ['sim_start', 'initial_set', 'h2_set'],
            'attrs': ['h2storage_id',
                    'h2_in', # in the python file this existed in the
                        ↪ re_params.
                    'h2_stored', # re_params returns values from the python
                        ↪ file, so we need to have it here so that mosaik
                    # can connect them and enter the values.
                    'h2_soc', 'mod', 'flag', 'time', 'h2storage_soc_max', '
                        ↪ h2storage_soc_min', 'h2_given', 'h2_out'],
            'trigger': [],
        },
    },
}

class compressedhydrogen(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'h2storage_' # every entity that we create will
            ↪ start with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None
        self.soc = {}
        self.flag = {}
        self.h2_stored = {}

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, initial_set, h2_set, sim_start):
        self.start = pd.to_datetime(sim_start)
        # next_eid=len(self.model)
        self._entities = []
        # for i in range(next_eid,next_eid+num):

        # num is the number of models of battery we want.
        for i in range(num):
            # we provide an ID to each entity we create. %s%d will be
                ↪ replaced by the values of eid_prefix and i
```

```
            self.eid = '%s%d' % (self.eid_prefix, i)


            h2storage_instance = hydrogen_storage.hydrogenstorage_python(
                ↪ initial_set, h2_set)

            self.entities[self.eid] = h2storage_instance
            self.soc[self.eid] = initial_set['initial_soc']
            self.flag[self.eid] = h2_set['flag']
            self._entities.append({'eid': self.eid, 'type': model, 'rel': [],
                ↪  })
            # print(self._entities)

        return self._entities

    def step(self, time, inputs, max_advance):
        self.time = time
        current_time = (self.start + pd.Timedelta(time * self.
            ↪ time_resolution, unit='seconds'))
        # print('from battery %%%%%%%%', current_time)
        for eid, attrs in inputs.items():
            # print(eid)
            # print(attrs)

            # In {'p_ask': {'CSVB-0.BATTEYP_0': 0.5}}, 'p_ask' is the attr,
                ↪ and 0.5 is vals
            for attr, vals in attrs.items():
                if attr == 'h2_in':
                    # of all the values present in the dictionary, we make a
                        ↪ list out of it.
                    # the [0] means we are calling the 0th entity in that list
                    h2_in = list(vals.values())[0]
                    print('#hydrogen h2_in input:', h2_in)

                elif attr == 'h2_out':
                    h2_out = list(vals.values())[0]


            self._cache[eid] = self.entities[eid].output_h2(h2_in, h2_out,
                ↪ self.soc[eid])

            self.soc[eid] = self._cache[eid]['h2_soc']
            self.flag = self._cache[eid]['flag']
            self.h2_stored[eid] = self._cache[eid]['h2_stored']
            # print('h2 flag', self.flag)
            print('------------------------->', self.h2_stored)
            # yield self.mosaik.set_data(self._cache)

            # self.battery_flag[eid]=self._cache[eid]['flag']
            soc_val = list(self.soc.values())
            h2_soc = soc_val[0] # this is so that the value that battery
                ↪ sends is dictionary and not a dictionary of a dictionary.
            out = yield self.mosaik.set_data({'H2storage-0': {'Controller-0.
                ↪ ctrl_0': {'h2_soc': h2_soc}}}) # this code is supposed to
                ↪ hold the soc value and
```

```
        return None

    def get_data(self, outputs):
        data = {}
# # self.test.append(self.flag) # if we do this code, then we end up with
    ↪ a list which increases with each step. Duh!
# # try:
# # # the following code takes the vale at -2 position in the list. The -2
    ↪  vale of the list represents the value of the previous step
# # self.pflag = self.test[-2] # first python tries this line of code. If
    ↪ it doesnt work then it follows the code in except.
# # except:
# # self.pflag = self.flag
#
        for eid, attrs in outputs.items():
            model = self.entities[eid]
            # data['time'] = self.time
            data[eid] = {}
            for attr in attrs:
                # data[eid][attr] = getattr(model, attr) # this line of a code
                    ↪  is short form for the following code which is commented
                    ↪ out
                if attr == 'h2_soc':
                    data[eid][attr] = self._cache[eid]['h2_soc']
                elif attr == 'mod':
                    data[eid][attr] = self._cache[eid]['mod']
                elif attr == 'h2storage_id':
                    data[eid][attr] = eid
                elif attr == 'flag':
                    data[eid][attr] = self._cache[eid]['flag']
                elif attr == 'h2_stored':
                    data[eid][attr] = self._cache[eid]['h2_stored']
                elif attr == 'h2_given':
                    data[eid][attr] = self._cache[eid]['h2_given']
                elif attr == 'h2_consumed':
                    data[eid][attr] = self._cache[eid]['h2_consumed']
                # if eid in self._cache:
                # data['time'] = self.time
                # data.setdefault(eid, {})[attr] = self._cache[eid][attr]

        return data


def main():
    mosaik_api.start_simulation(compressedhydrogen(), 'H2storage-Simulator'
        ↪ )

if __name__ == "__main__":
    main()
```

## E.13 Models/imexport

### E.13.1 imexport_model.py

Listing E.23: Models/imexport/imexport_model.py

```
class imexport_python():

   def __init__(self, imexports, output_type):
      self.consumption = 0
      self.output_type = output_type
      self.imexports = imexports #possible scaling factor

   def demand(self, imexport):
      # incoming imexport is in kW at every 15 min interval

      if self.output_type == 'energy':
         self.consumption = self.imexports*imexport/4 # kWh
      elif self.output_type == 'power':
         self.consumption = self.imexports*imexport # kW

      re_params = {'imexport_dem': self.consumption}
      return re_params
```

## E.13.2 imexport_mosaik.py

Listing E.24: Models/imexport/imexport_mosaik.py

```
import mosaik_api

try:
   import Models.imexport.imexport_model as imexport_model
except ModuleNotFoundError:
   import imexport_model as imexport_model
else:
   import Models.imexport.imexport_model as imexport_model

import pandas as pd

META = {
   'type': 'event-based',

   'models': {
      'imexportmodel': {
         'public': True,
         'params': ['sim_start', 'imexports', 'output_type'],
         'attrs': ['imexport_id', 'imexport_dem', 'imexport', ],
         # third attribute here must match the second attribute in the
            ↪ second line of the txt file containing data
         'trigger': [],
      },
   },
}


class imexportSim(mosaik_api.Simulator):
   def __init__(self):
      super().__init__(META)
      self.eid_prefix = 'imexport_' # every entity that we create will
         ↪ start with 'wind_'
```

```
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
    # we can use this to pass additional initialization tasks

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = imexport_model.imexport_python(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):

        current_time = (self.start +
                    pd.Timedelta(time * self.time_resolution,
                                 unit='seconds')) # timedelta represents a
                                    ↪ duration of time
        print('from imexport %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            # print(eid)
            # print(attrs)
            for attr, vals in attrs.items():
                # if attr == 'u':
                imexport = list(vals.values())[0]
                # print(u)
                self._cache[eid] = self.entities[eid].demand(imexport) # not
                    ↪ necessary to have u in brackets. It is not necessary to
                    ↪ keep the same name as the one in python file
                # print(self._cache)
        return None

    def get_data(self, outputs):
        data = {}
```

```
      for eid, attrs in outputs.items():
         data[eid] = {}
         for attr in attrs:
            if attr == 'imexport_dem':
               data[eid][attr] = self._cache[eid]['imexport_dem']
      return data


def main():
   mosaik_api.start_simulation(imexportSim(), 'imexport Simulator')


if __name__ == "__main__":
   main()
```

## E.14 Models/Load

### E.14.1 load_model.py

Listing E.25: Models/Load/load_model.py

```
class load_python():

   def __init__(self, houses, output_type):
      self.consumption = 0
      self.houses = houses #possible scaling factor
      self.output_type = output_type

   def demand(self, load):
      # incoming load is in kWh at every 15 min interval
      # incoming value of load is in kWh

      if self.output_type == 'energy':
         self.consumption = (self.houses * load) # kWh
      elif self.output_type == 'power':
         self.consumption = (self.houses * load)*4 # kW

      re_params = {'load_dem': self.consumption}
      return re_params
```

### E.14.2 load_mosaik.py

Listing E.26: Models/Load/load_mosaik.py

```
import mosaik_api

#import Load.load_model as load_model
try:
   import Models.Load.load_model as load_model
except ModuleNotFoundError:
   import load_model as load_model
else:
   import Models.Load.load_model as load_model
```

```
import pandas as pd

META = {
    'type': 'event-based',

    'models': {
        'loadmodel': {
            'public': True,
            'params': ['sim_start', 'houses', 'output_type'],
            'attrs': ['load_id','load_dem','load',], #third attribute here
                ↪ must match the second attribute in the second line of the
                ↪ txt file containing data
            'trigger': [],
        },
    },
}


class loadSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'load_' # every entity that we create will start
            ↪ with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
    # we can use this to pass additional initialization tasks

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = load_model.load_python(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities
```

```
   def step(self, time, inputs, max_advance):

      current_time = (self.start +
                  pd.Timedelta(time * self.time_resolution,
                               unit='seconds')) # timedelta represents a
                                  ↪ duration of time
      print('from load %%%%%%%%%%', current_time)

      for eid, attrs in inputs.items():
         # print(eid)
         # print(attrs)
         for attr, vals in attrs.items():
            # if attr == 'u':
            u = list(vals.values())[0]
            # print(u)
            self._cache[eid] = self.entities[eid].demand(u) # not
               ↪ necessary to have u in brackets. It is not necessary to
               ↪ keep the same name as the one in python file
            # print(self._cache)
      return None

   def get_data(self, outputs):
      data = {}
      for eid, attrs in outputs.items():
         data[eid] = {}
         for attr in attrs:
            if attr == 'load_dem':
               data[eid][attr] = self._cache[eid]['load_dem']
      return data
def main():
   mosaik_api.start_simulation(loadSim(), 'load Simulator')

if __name__ == "__main__":
   main()
```

# E.15 Models/PV

## E.15.1 pv_model_new.py

Listing E.27: Models/PV/pv_model_new.py

```
import numpy as np
from numpy import sin, cos
from configuration.build_configuration_xml import mode1
class PV_py_model:

   def __init__(self, panel_data, m_tilt, m_az, cap, output_type):
      # with open ('L1234.csv') as data:
      # self.start_date=data.readline(1)
      # self.start_date = (genfromtxt('L1234.csv', delimiter=',', usecols
         ↪ =0, max_rows=1, skip_header=1))

      self.m_area = panel_data['Module_area'] #m^2
      # module area. available in the spec sheet of a pv module
      self.NOCT = panel_data['NOCT'] #degree celsius
```

```
        self.m_efficiency_stc = panel_data['Module_Efficiency']
        self.G_NOCT = panel_data['Irradiance_at_NOCT'] #W/m^2
        # W/m2 This is the irradiance that falls on the panel under NOCT
            ↪ conditions
        self.P_STC = panel_data['Power_output_at_STC']
        # Watts. Available in spec sheet of a module
        self.m_tilt = m_tilt #degrees
        self.m_az = m_az #degrees
        self.cap = cap #kW
        self.output_type = output_type
        self.emission = cap*11.156/1000000 #capacity installed * tonne/co2
            ↪ per kWp installed per 15 min

    def sun_azimuth(self): # need to load sun_az
        sun_azimuth = self.sun_az
        return sun_azimuth

    def sun_elevation(self):
        sun_elevation = self.sun_el
        return sun_elevation

    def aoi(self):
        cos_aoi = np.array(cos(np.radians(90 - self.m_tilt)) * cos(np.
            ↪ radians(self.sun_elevation())) * cos(
            np.radians(self.m_az - self.sun_azimuth())) + sin(np.radians(90 -
                ↪  self.m_tilt)) * sin(
            self.sun_elevation()))
        if cos_aoi < 0:
            cos_aoi = 0
        return cos_aoi

    def diffused_irr(self):
        self.svf = np.array((1 + cos(np.radians(self.m_tilt))) / 2)
        g_diff = self.svf * self.dhi # global diffused irradiance #W/m2
        return g_diff

    def reflected_irr(self):
        albedo = 0.2
        g_ref = albedo * (1 - self.svf) * self.ghi
        return g_ref

    def direct_irr(self):
        g_dir = self.dni * self.aoi()
        return g_dir

    def total_irr(self):
        self.g_aoi = self.diffused_irr() + self.reflected_irr() + self.
            ↪ direct_irr()
        return self.g_aoi


    # the effect of temperature and wind speed on the module efficiency.
    def Temp_effect(self):
        m_temp = self.temp + (np.divide(self.total_irr(), self.G_NOCT)) * (
            ↪ self.NOCT - 20) * (
```

```
            np.divide(9.5, (5.7 + 3.8 * self.ws))) * (1 - (self.
                ↪ m_efficiency_stc / 0.90))

        efficiency = self.m_efficiency_stc * (1 + (-0.0035 * (m_temp - 25)))
        print(efficiency)
        return efficiency

    def output(self):

        # constants
        # inverter efficiency. We can use sandia model to actually find an
            ↪ inverter that suits our needs
        inv_eff = 0.96
        mppt_eff = 0.99 # again, can calculate it accurately
        losses = 0.97 # other losses
        sf = 1.1

        # generation calculation
        num_of_modules = np.ceil(self.cap * sf / self.P_STC)


        # [W] again we get this for every time step
        # this is for the DC output from the number of panes we require (
            ↪ calculated above) at every hour
        p_dc = self.Temp_effect() * num_of_modules * self.m_area * self.
            ↪ total_irr()
        total_m_area = num_of_modules * self.m_area

        # AC output at every hour from all the panels (a solar farm)

        if self.output_type == 'energy':
            p_ac = (total_m_area * self.total_irr() *
                  self.Temp_effect() * inv_eff * mppt_eff * losses)/4 # kWh
        elif self.output_type == 'power':
            p_ac = ((total_m_area * self.total_irr() *
                  self.Temp_effect() * inv_eff * mppt_eff * losses) ) # kW
        if mode1:
            return {'pv_gen': self.cap, 'total_irr': self.g_aoi, 'pv_em':
                ↪ self.emission}
        else:
            return {'pv_gen': p_ac, 'total_irr': self.g_aoi, 'pv_em': self.
                ↪ emission}

    def connect(self, G_Gh, G_Dh, G_Bn, Ta, hs, FF, Az):
        self.ghi = G_Gh
        self.dhi = G_Dh
        self.dni = G_Bn
        self.temp = Ta
        self.sun_el = hs
        self.ws = FF
        self.sun_az = Az
        # print( self.sun_az, self.ws, self.dni)
        # print('1')
        # print(sun_az, ws, dni, dhi, ghi, sun_el, ambient_temp)
        return self.output()
```

### E.15.2 pv_mosaik.py

Listing E.28: Models/PV/pv_mosaik.py

```
import itertools
import mosaik_api

from configuration.build_configuration_xml import mode1
if mode1:
    import hardwareattached
#import PV.PV_model_new as PV_model_new
try:
    import Models.PV.pv_model_new as PV_model_new
except ModuleNotFoundError:
    import pv_model_new as PV_model_new
else:
    import Models.PV.pv_model_new as PV_model_new
import pandas as pd
import itertools

meta = {
    'type': 'event-based', #if reading from a csv file then it is time
        ↪ based
    'models': {
        'PVset': {
            'public': True,
            'params': ['panel_data',
                    'm_tilt','m_az', 'cap', 'sim_start', 'output_type'],
            # and are attrs the specific outputs we want from the code? to
                ↪ connect with other models
            'attrs': ['pv_id', 'G_Gh', 'G_Dh', 'G_Bn', 'Ta', 'hs', 'FF', 'Az'
                ↪ , 'pv_gen', 'total_irr', 'pv_em'],
        },
    },
}

class PvAdapter(mosaik_api.Simulator):
    def __init__(self):
        super(PvAdapter, self).__init__(meta)
        self.eid_prefix='pv_'
        self.entities = {} # every entity that we create of PV gets stored
            ↪ in this dictionary as a list
        self.mods = {}
        self._cache = {} #we store the final outputs after calling the
            ↪ python model (#PV1) here.

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
```

```
        entities = []
        for i in range (num):
            eid = '%s%d' % (self.eid_prefix, i)

            # we are creating an instance for PV and call the python file for
                ↪ that. **model_params refers to the
            # parameters we have mentioned above in the META. New instance
                ↪ will have those parameters.
            model_instance = PV_model_new.PV_py_model(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        # print(entities)
        return entities

    def step(self, time, inputs, max_advance):
        # in this method, we call the python file at every data interval and
            ↪ perform the calculations.
        current_time = (self.start + pd.Timedelta(time * self.
            ↪ time_resolution,
                                        unit='seconds')) # timedelta
                                            ↪ represents a duration of time
        print('from pv %%%%%%%%%', current_time)
        # print('#inouts: ', inputs)
        for eid, attrs in inputs.items():
            # print('#eid: ', eid)
            # print('#attrs: ', attrs)
            # and relate it with the information in mosaik document.
            v = [] # we create this empty list to hold all the input values
                ↪ we want to give since we have more than 2
            for attr, vals in attrs.items():

                # print('#attr: ', attr)
                # print('#vals: ', vals)
                # inputs is a dictionary, which contains another dictionary.
                # value of U is a list. we need to combine all the values into
                    ↪ a single list. But is we just simply
                # append them in v, we have a nested list, hence just 1 list.
                    ↪ that creates a problem as it just
                # gives all 7 values to only sun_az in the python model and we
                    ↪ get an error that other 6 values are missing.
                u = list(vals.values())
                # print('#u: ', u)
                v.append(u) # we append every value of u to v from this
                    ↪ command.
            # print('#v: ', v)

            # the following code helps us to convert the nested list into a
                ↪ simple plain list and we can use that simply
            v_merged = list(itertools.chain(*v))
            # print('#v_merged: ', v_merged)
            self._cache[eid] = self.entities[eid].connect(v_merged[0],
                ↪ v_merged[1], v_merged[2], v_merged[3],
                                                v_merged[4], v_merged[5],
                                                    ↪ v_merged[6]) # PV1
            # print(self._cache)
            # print('# cache[eid]: ', self._cache[eid])
```

```
    # the following code desnt work because it just put one value 7 times
        ↪ :/! Dumb move
                # self._cache[eid] = self.entities[eid].connect(u, u, u, u,
                    ↪ u, u, u)
        return None

    def get_data(self, outputs):
        data = {}

        # to write the data in an external file, we use this method. This
            ↪ API inturn calls a file within Mosaik
        # which handles the writing of the outputs provided the attrs are
            ↪ present in the base python model file you made

        for eid, attrs in outputs.items():
            # model = self.entities[eid]
            # data['time'] = self.time
            data[eid] = {}

            for attr in attrs:
                # if we want more values to print in the output file, mimic
                    ↪ the below for new attributes and make sure
                # those parameters are present in the re_params in the python
                    ↪ file of the model
                if attr == 'pv_gen':
                        if mode1:
                            data[eid][attr] = hardwareattached.getSolarPower() *
                                ↪ self._cache[eid]['pv_gen']
                        else:
                            data[eid][attr] = self._cache[eid]['pv_gen']
                elif attr == 'total_irr':
                    data[eid][attr] = self._cache[eid]['total_irr']
                elif attr == 'pv_em':
                    data[eid][attr] = self._cache[eid]['pv_em']
        return data

def main():
    mosaik_api.start_simulation(PvAdapter(), 'PV-Illuminator')
if __name__ == '__main__':
    main()
```

## E.16 Models/Resident

### E.16.1 Resident_model.py

Listing E.29: Models/Resident/Resident_model.py

```
class resident_python:

    def __init__(self, residents, output_type):
        self.consumption = 0
        self.residents = residents #possible scaling factor
        self.output_type = output_type

    def demand(self, resident_load):
```

```
        # incoming load is in kWh at every 15 min interval
        # incoming value of load is in kWh

        if self.output_type == 'energy':
            self.consumption = (self.residents * resident_load) # kWh
        elif self.output_type == 'power':
            self.consumption = (self.residents * resident_load)*4 # kW

        re_params = {'resident_dem': self.consumption}
        return re_params
```

## E.16.2 Resident_mosaik.py

Listing E.30: Models/Resident/Resident_mosaik.py

```
import mosaik_api
#import Factory.factory_model as factory_model
try:
    import Models.Resident.Resident_model as resident_model
except ModuleNotFoundError:
    import Resident_model as resident_model
else:
    import Models.Resident.Resident_model as resident_model

import pandas as pd

META = {
    'type': 'event-based',

    'models': {
        'residentmodel': {
            'public': True,
            'params': ['sim_start', 'residents', 'output_type'],
            'attrs': ['resident_id','resident_dem','residents_load',], #third
                ↪ attribute here must match the second attribute in the
                ↪ second line of the txt file containing data
            'trigger': [],
        },
    },
}


class ResidentSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'resident_' # every entity that we create will
            ↪ start with 'factory_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
```

```
    # we can use this to pass additional initialization tasks

    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range(num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = resident_model.resident_python(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):
        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution,
                                     unit='seconds')) # timedelta represents a
                                        ↪ duration of time
        print('from resident %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            # print(eid)
            for attr, vals in attrs.items():
                # if attr == 'u':
                resident_load = list(vals.values())[0]
                # print(u)
                self._cache[eid] = self.entities[eid].demand(resident_load) #
                    ↪ not necessary to have u in brackets. It is not necessary
                    ↪ to keep the same name as the one in python file
                # print(self._cache)
        return None

    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            data[eid] = {}
            for attr in attrs:
                if attr == 'resident_dem':
                    data[eid][attr] = self._cache[eid]['resident_dem']
        return data


def main():
    mosaik_api.start_simulation(ResidentSim(), 'Resident Simulator')
```

```
if __name__ == "__main__":
    main()
```

## E.17 Models/Wind

### E.17.1 Wind_model.py

Listing E.31: Models/Wind/Wind_model.py

```
import numpy as np
from numpy import genfromtxt
import math


from configuration.build_configuration_xml import mode1
if mode1:
    import hardwareattached

# this model will be called by the step method in the wind_SimAPI file and
    ↪ there it will get input for calculations.
# remove the input of turbine height and just make changing the wind speed
    ↪ to a height of 80m by default.

class wind_py_model:

    def __init__(self, cap, p_rated, u_rated, u_cutin, u_cutout, diameter,
        ↪ cp, output_type):
        self.turbines = np.ceil(cap/p_rated)
        print(self.turbines, 'turbines installed for offshore')
        self.p_rated = p_rated # kW power it generates at rated wind speed
            ↪ and above
        self.u_rated = u_rated # m/s #windspeed it generates most power at
        self.u_cutin = u_cutin # m/s #below this wind speed no power
            ↪ generation
        self.u_cutout = u_cutout # m/s #above this wind speed no power
            ↪ generation. Blades are pitched
        self.cp = cp # coefficient of performance of a turbine. Usually
            ↪ around0.40. Never more than 0.59
        self.powerout = 0 # output power at wind speed u
        self.output_type = output_type
        self.dia = diameter #m
        self.emission = self.turbines*p_rated * 1.843/1000000 #tonnes CO2eq
            ↪ per kW per quarter hour
        self.hub_h = 136

    # def windprofile(self):
    # """ Assuming that the wind speed data is measured at 10m height, it
        ↪ will be changed to a height of 80m
    # which is a general hub height of a wind turbine"""

    # def noproduction(self, u):
    # re_params = {'p_out': 0}
    #
```

```
# return re_params


def production(self, u):
    radius = self.dia/2
    air_density = 1.225
    z0 = 0.0002
    self.resolution = 15
    self.time_interval = self.resolution/60 #hours (15 minutes time
        ↪ interval/ number of minutes in an hour)


    if self.hub_h >60: #if hub height is above blending height
        u60 = u * (np.log(60 / z0) / np.log(10 / z0)) #determine
            ↪ windspeed at blending height
        uh = u60*((self.hub_h/60)**0.11) #determine windspeed at
            ↪ hubheight
    else: #hub height is below blending height
        uh = u * (np.log(self.hub_h / z0) / np.log(10 / z0)) #use loglaw
            ↪ to determine windspeed at hubheight

    if self.output_type == 'energy':
        p = ((0.5 * (uh ** 3) * (math.pi * (radius ** 2.0)) * air_density
            ↪  * self.cp) / 1000) * self.time_interval # kWh
    elif self.output_type == 'power':
        p = ((0.5 * (uh ** 3) * (math.pi * (radius ** 2.0)) * air_density
            ↪  * self.cp) / 1000) # kW
    re_params = {'wind_gen': p*self.turbines, 'u': uh, 'wind_em': self.
        ↪ emission}
    return re_params

def generation(self, u):

    z0 = 0.0002
    if self.hub_h > 60: # if hub height is above blending height
        u60 = u * (np.log(60 / z0) / np.log(10 / z0)) # determine
            ↪ windspeed at blending height
        uh = u60 * ((self.hub_h / 60) ** 0.11) # determine windspeed at
            ↪ hubheight
    else: # hub height is below blending height
        uh = u * (np.log(self.hub_h / z0) / np.log(10 / z0)) # use loglaw
            ↪  to determine windspeed at hubheight

    if not mode1: #hardware mode is not selected
        if uh >= self.u_rated:
            if uh == self.u_rated: #speed at rated windspeed->rated power
                if self.output_type == 'power':
                    re_params = {'wind_gen': (self.p_rated)*self.turbines, '
                        ↪ u': uh, 'wind_em': self.emission}
                elif self.output_type == 'energy':
                    re_params = {'wind_gen': (self.p_rated * self.
                        ↪ time_interval)*self.turbines, 'u': uh, 'wind_em':
                        ↪ self.emission}
            elif uh <= self.u_cutout: #speed between rated windspeed and
                ↪ cutout -> capped at rated power
                if self.output_type == 'power':
```

```
                re_params = {'wind_gen': (self.p_rated)*self.turbines, '
                    ↪ u': uh, 'wind_em': self.emission}
             elif self.output_type == 'energy':
                re_params = {'wind_gen': (self.p_rated * self.
                    ↪ time_interval)*self.turbines, 'u': uh, 'wind_em':
                    ↪ self.emission}
             # return re_params
          else: #speed above coutout->generation is stopped
             re_params = {'wind_gen': 0, 'u': uh, 'wind_em': self.
                 ↪ emission}
             # return re_params
       elif uh < self.u_cutin: #speed belowe cutin->no generation can
           ↪ happen
          re_params = {'wind_gen': 0, 'u': uh, 'wind_em': self.emission}
       else:
          re_params = self.production(u) #between cutin and rated
              ↪ windspeed-> generation is windspeed dependent
    else: #wind generation is capacity scaled by input from hardware
       re_params = {'wind_gen': self.p_rated * self.turbines *
           ↪ hardwareattached.getWindPower(), 'wind_em': self.emission}

    return re_params
```

## E.17.2 wind_mosaik.py

Listing E.32: Models/Wind/wind_mosaik.py

```
from collections import namedtuple

import mosaik_api
import numpy as np
import pandas as pd

#import Wind.Wind_model as Wind_model


try:
   import Models.Wind.Wind_model as Wind_model
except ModuleNotFoundError:
   import Wind_model as Wind_model
else:
   import Models.Wind.Wind_model as Wind_model

META = {
   'type': 'event-based',
   #wind is an event based event because the event here is a wind speed.
       ↪ It doesnt purely run because of time interval, I think.
   # if I put it to time-based, there is type error:
   # File "C:\Users\ragha\AppData\Local\Programs\Python\Python310\lib\site
       ↪ -packages\mosaik\scheduler.py", line 405, in step
   # sim.progress_tmp = next_step - 1
   # TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'

   'models': {
      'windmodel': {
```

```
        'public': True,
        'params': ['cap','p_rated', 'u_rated', 'u_cutin', 'u_cutout', 'cp
            ↪ ', 'sim_start', 'output_type', 'diameter'],
        'attrs': ['wind_id',
                'wind_gen', # in the python file this existed in the
                    ↪ re_params.
                    # re_params returns values from the python file, so we
                        ↪ need to have it here so that mosaik
                    # can connect them and enter the values.
                'u',
                'wind_em'],
    },
  },
}


class WindSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'wind_' # every entity that we create will start
            ↪ with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
    # we can use this to pass additional initialization tasks
    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range (num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = Wind_model.wind_py_model(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):

        current_time = (self.start +
```

```
                pd.Timedelta(time * self.time_resolution, unit='seconds'
                    ↪ )) # timedelta represents a duration of time
        print('from wind %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            # raghav: Inputs come from a CSV file which needs to be read by a
                ↪  Mosaik # CSV reader -
            # and it gives the output in a manner we want. The output from
                ↪ the CSV file will be the Input here.
            # they are connected in the scenario file. ##W1 (see W1 in
                ↪ comments in scenario file to understand)
            # print(eid)
            # print(attrs)
            for attr, vals in attrs.items():
                if attr == 'u':
                    u = list(vals.values())[0]
                    # print(u)
                    self._cache[eid] = self.entities[eid].generation(u) #not
                        ↪ necessary to have u in brackets. It is not
                    # necessary to keep the same name as the one in python file
                    # in the above line, we have called our entity of wind
                        ↪ model we created in create followed by a
                    # definition 'generation'. Since self.entities =
                        ↪ model_instance and model instance calls out wind
                    # python model, so in this step we are called the function
                        ↪ 'generation' and give it a value for u.
                    # This step makes the python file run and do the
                        ↪ calculations for us of wind_gen.
                    # print(self._cache[eid])
                    # [wind_gen:,soc:,flag:]
                    # self.soc[eid] = self._cache[eid]['soc']
                    # self.battery_flag[eid]=self._cache[eid]['flag']
                    # print(self._cache)

        return None
        # # CURRENTLY THE INPUT IS THE WIND SPEED FILE BUT IT CAN BE CHANGED
            ↪  TO THE SUM OF pv AND wind AND THE grid load
        # print (' entered STEP of SimAPI')
        # self.time=time
        #
        # for eid, model_instance in self.entities.items():
        # for i in wind_data:
        # model_instance.u = i
        # print(model_instance[0])
        # print('Exited STEP of SimAPI')
        # return time + 300

    def get_data(self, outputs):
        data={}
        for eid, attrs in outputs.items():
            # model = self.entities[eid]
            # data['time'] = self.time
            data[eid]={}
            for attr in attrs:
                # if we want more values to print in the output file, mimic
                    ↪ the below for new attributes and make sure
```

```
            # those parameters are present in the re_params in the python
               ↪ file of the technology
            if attr == 'wind_gen':
               data[eid][attr] = self._cache[eid]['wind_gen']
            elif attr == 'u':
               data[eid][attr] = self._cache[eid]['u']
            elif attr == 'wind_em':
               data[eid][attr] = self._cache[eid]['wind_em']
      return data



def main():
   return mosaik_api.start_simulation(WindSim(), 'WindEnergy Simulator')

if __name__ == "__main__":
   main()
```

## E.18 Models/Wonshore

### E.18.1 Wonshore_model.py

Listing E.33: Models/Wonshore/Wonshore_model.py

```
import numpy as np
from numpy import genfromtxt
import math

from configuration.buildmodelset import livedatamode


# this model will be called by the step method in the wind_SimAPI file and
   ↪ there it will get input for calculations.
# remove the input of turbine height and just make changing the wind speed
   ↪ to a height of 80m by default.

class wonshore_py_model:

   def __init__(self, cap, p_rated, u_rated, u_cutin, u_cutout, diameter,
      ↪ cp, output_type):
      self.turbines = np.ceil(cap/p_rated)
      print(self.turbines, 'turbines installed for onshore')
      self.p_rated = p_rated # kW power it generates at rated wind speed
         ↪ and above
      self.u_rated = u_rated # m/s #windspeed it generates most power at
      self.u_cutin = u_cutin # m/s #below this wind speed no power
         ↪ generation
      self.u_cutout = u_cutout # m/s #above this wind speed no power
         ↪ generation. Blades are pitched
      self.cp = cp # coefficient of performance of a turbine. Usually
         ↪ around0.40. Never more than 0.59
      self.powerout = 0 # output power at wind speed u
      self.output_type = output_type
      self.dia = diameter
```

```
        self.emission = self.turbines*p_rated * 0.955/1000000 #tonnes CO2eq
            ↪ per kW per quarter hour
        self.hub_h = 136

    # def windprofile(self):
    # """ Assuming that the wind speed data is measured at 10m height, it
        ↪ will be changed to a height of 80m
    # which is a general hub height of a wind turbine"""

    # def noproduction(self, u):
    # re_params = {'p_out': 0}
    #
    # return re_params


    def production(self, u):
        radius = self.dia/2
        air_density = 1.225 #kg/m^3
        z0 = 0.2
        self.resolution = 15
        self.time_interval = self.resolution/60 #hours (15 minutes time
            ↪ interval/ number of minutes in an hour)

        if self.hub_h > 60:
            u60 = u * (np.log(60 / z0) / np.log(10 / z0))
            uh = u60 * ((self.hub_h / 60) ** 0.143)
        else:
            uh = u * (np.log(self.hub_h / z0) / np.log(10 / z0))

        if self.output_type == 'energy':
            p = ((0.5 * (uh ** 3) * (math.pi * (radius ** 2.0)) * air_density
                ↪  * self.cp) / 1000) * self.time_interval # kWh
        elif self.output_type == 'power':
            p = ((0.5 * (uh ** 3) * (math.pi * (radius ** 2.0)) * air_density
                ↪  * self.cp) / 1000) # kW
        re_params = {'wonshore_gen': p*self.turbines, 'u': uh, 'wonshore_em'
            ↪ : self.emission}
        return re_params

    def generation(self, u):
        z0 = 0.2
        if self.hub_h > 60: # if hub height is above blending height
            u60 = u * (np.log(60 / z0) / np.log(10 / z0)) # determine
                ↪ windspeed at blending height
            uh = u60 * ((self.hub_h / 60) ** 0.11) # determine windspeed at
                ↪ hubheight
        else: # hub height is below blending height
            uh = u * (np.log(self.hub_h / z0) / np.log(10 / z0)) # use loglaw
                ↪  to determine windspeed at hubheight

        if uh >= self.u_rated:
            if uh == self.u_rated: # speed at rated windspeed->rated power
                if self.output_type == 'power':
                    re_params = {'wonshore_gen': (self.p_rated) * self.turbines
                        ↪ , 'u': uh, 'wonshore_em': self.emission}
                elif self.output_type == 'energy':
```

```
                re_params = {'wonshore_gen': (self.p_rated * self.
                    ↪ time_interval) * self.turbines, 'u': uh,
                          'wonshore_em': self.emission}
            elif uh <= self.u_cutout: # speed between rated windspeed and
                ↪ cutout -> capped at rated power
                if self.output_type == 'power':
                    re_params = {'wonshore_gen': (self.p_rated * self.turbines)
                        ↪ , 'u': uh, 'wonshore_em': self.emission}
                elif self.output_type == 'energy':
                    re_params = {'wonshore_gen': (self.p_rated * self.
                        ↪ time_interval) * self.turbines, 'u': uh,
                              'wonshore_em': self.emission}
                # return re_params
            else: # speed above coutout->generation is stopped
                re_params = {'wonshore_gen': 0, 'u': uh, 'wonshore_em': self.
                    ↪ emission}
                # return re_params
        elif uh < self.u_cutin: # speed belowe cutin->no generation can
            ↪ happen
            re_params = {'wonshore_gen': 0, 'u': uh, 'wonshore_em': self.
                ↪ emission}
        else:
            re_params = self.production(u) # between cutin and rated
                ↪ windspeed-> generation is windspeed dependent


        # if u == self.u_rated:
        # p = (0.5 * (self.u_rated ** 3) * (math.pi * (radius ** 2.0)) *
            ↪ air_density * self.cp * 0.59)
        # re_params = {'p_out': p}
        # else:
        # if u < self.u_cutin:
        # re_params = self.noproduction(u)
        # elif u > self.u_cutout:
        # re_params = self.noproduction(u)
        # elif
        # else:
        # re_params = self.production(u)p
        return re_params
```

### E.18.2 Wonshore_mosaik.py

Listing E.34: Models/Wonshore/Wonshore_mosaik.py

```
from collections import namedtuple

import mosaik_api
import numpy as np
import pandas as pd

#import Wind.Wind_model as Wind_model


try:
    import Models.Wonshore.Wonshore_model as Wonshore_model
```

```
except ModuleNotFoundError:
    import Wonshore_model as Wonshore
else:
    import Models.Wonshore.Wonshore_model as Wonshore_model

META = {
    'type': 'event-based',
    #wind is an event based event because the event here is a wind speed.
        ↪ It doesnt purely run because of time interval, I think.
    # if I put it to time-based, there is type error:
    # File "C:\Users\ragha\AppData\Local\Programs\Python\Python310\lib\site
        ↪ -packages\mosaik\scheduler.py", line 405, in step
    # sim.progress_tmp = next_step - 1
    # TypeError: unsupported operand type(s) for -: 'NoneType' and 'int'

    'models': {
        'wonshoremodel': {
            'public': True,
            'params': ['cap','p_rated', 'u_rated', 'u_cutin', 'u_cutout', 'cp
                ↪ ', 'sim_start', 'output_type', 'diameter'],
            'attrs': ['wonshore_id',
                    'wonshore_gen', # in the python file this existed in the
                        ↪ re_params.
                        # re_params returns values from the python file, so we
                            ↪ need to have it here so that mosaik
                        # can connect them and enter the values.
                    'u',
                    'wonshore_em'],
        },
    },
}


class WonshoreSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'wonshore_' # every entity that we create will
            ↪ start with 'wind_'
        self.entities = {} # we store the model entity of our technology
            ↪ model
        self._cache = {} # used in the step function to store the values
            ↪ after running the python model of the technology
        # self.start_date = None

    # the following API call is will be called only once when we initiate
        ↪ the model in the scenario file.
    # we can use this to pass additional initialization tasks
    def init(self, sid, time_resolution):
        # print('hi, you have entered init') # working (20220524)
        self.time_resolution = time_resolution
        # print('Exited init os SimAPI') # working (20220524)
        return self.meta

    def create(self, num, model, sim_start, **model_params):
        # print('hi, you have entered create of SimAPI') # working
            ↪ (20220524)
```

```
        self.start = pd.to_datetime(sim_start)
        # print(type(self.entities))
        # next_eid = len(self.entities)
        # print('from create of SimAPI:', next_eid)
        entities = []
        # print(next_eid) # working (20220524)

        for i in range (num):
            eid = '%s%d' % (self.eid_prefix, i)
            model_instance = Wonshore_model.wonshore_py_model(**model_params)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs, max_advance):

        current_time = (self.start +
                        pd.Timedelta(time * self.time_resolution, unit='seconds'
                            ↪ )) # timedelta represents a duration of time
        print('from onshore wind %%%%%%%%%%', current_time)

        for eid, attrs in inputs.items():
            # raghav: Inputs come from a CSV file which needs to be read by a
                ↪  Mosaik # CSV reader -
            # and it gives the output in a manner we want. The output from
                ↪ the CSV file will be the Input here.
            # they are connected in the scenario file. ##W1 (see W1 in
                ↪ comments in scenario file to understand)
            # print(eid)
            # print(attrs)
            for attr, vals in attrs.items():
                if attr == 'u':
                    u = list(vals.values())[0]
                    # print(u)
                    self._cache[eid] = self.entities[eid].generation(u) #not
                        ↪ necessary to have u in brackets. It is not
                    # necessary to keep the same name as the one in python file
                    # in the above line, we have called our entity of wind
                        ↪ model we created in create followed by a
                    # definition 'generation'. Since self.entities =
                        ↪ model_instance and model instance calls out wind
                    # python model, so in this step we are called the function
                        ↪ 'generation' and give it a value for u.
                    # This step makes the python file run and do the
                        ↪ calculations for us of wind_gen.
                    # print(self._cache[eid])
                    # [wind_gen:,soc:,flag:]
                    # self.soc[eid] = self._cache[eid]['soc']
                    # self.battery_flag[eid]=self._cache[eid]['flag']
                    # print(self._cache)

        return None
        # # CURRENTLY THE INPUT IS THE WIND SPEED FILE BUT IT CAN BE CHANGED
            ↪  TO THE SUM OF pv AND wind AND THE grid load
        # print (' entered STEP of SimAPI')
        # self.time=time
```

```
        #
        # for eid, model_instance in self.entities.items():
        # for i in wind_data:
        # model_instance.u = i
        # print(model_instance[0])
        # print('Exited STEP of SimAPI')
        # return time + 300

    def get_data(self, outputs):
        data={}
        for eid, attrs in outputs.items():
            # model = self.entities[eid]
            # data['time'] = self.time
            data[eid]={}
            for attr in attrs:
                # if we want more values to print in the output file, mimic
                    ↪ the below for new attributes and make sure
                # those parameters are present in the re_params in the python
                    ↪ file of the technology
                if attr == 'wonshore_gen':
                    data[eid][attr] = self._cache[eid]['wonshore_gen']
                elif attr == 'u':
                    data[eid][attr] = self._cache[eid]['u']
                elif attr == 'wonshore_em':
                    data[eid][attr] = self._cache[eid]['wonshore_em']
        return data



def main():
    return mosaik_api.start_simulation(WonshoreSim(), 'WindEnergy Simulator
        ↪ ')

if __name__ == "__main__":
    main()
```

# Appendix F

# GUI Userguide

In this appendix, the figures of the different interfaces the GUI create are shown. There is started with mode 1 and after that mode 2 is shown. In Chapter 5 the explanation can be found.

## F.1 Mode 1

The first screen that pops up when the simulation is started is the same for mode 1 and 2. This is the following figure:

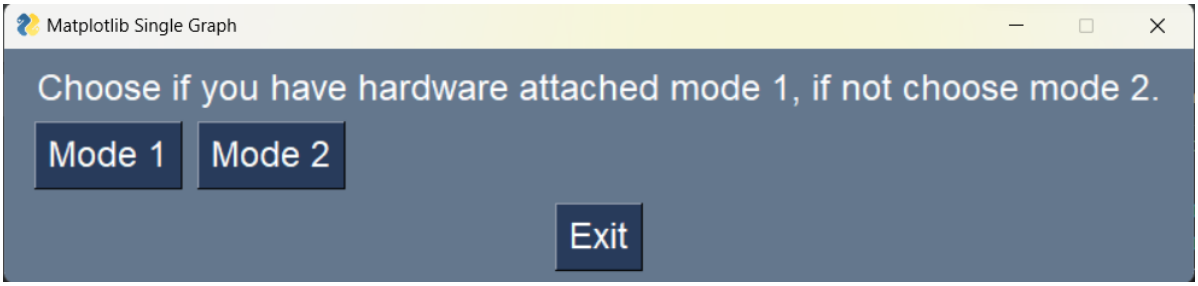


Figure F.1: Start screen GUI

Now mode 1 is chosen so the next interface that shows up is:

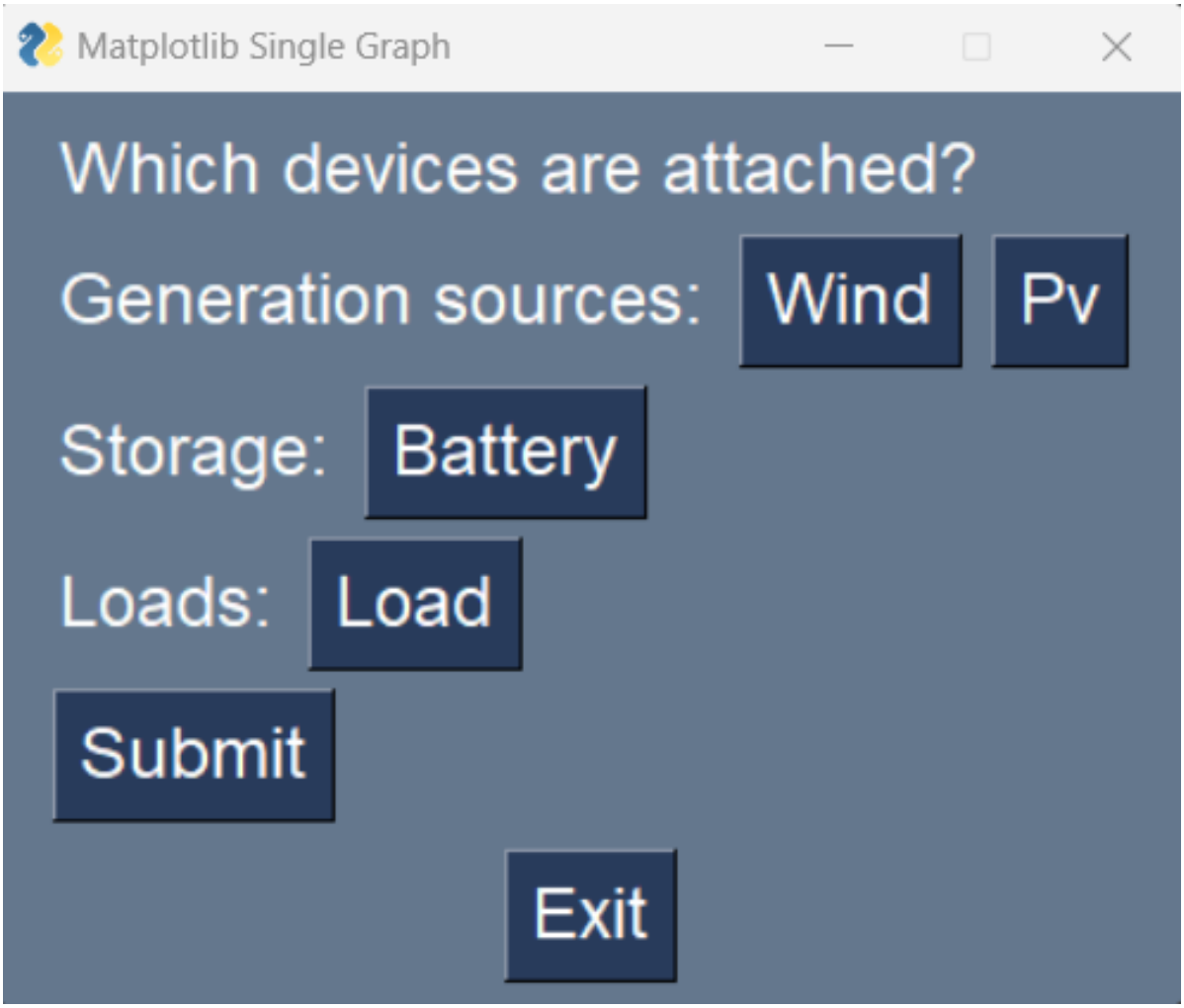


Figure F.2: Hardware one has attached that can be chosen to use.

Where the attached hardware can be chosen to use.If some hardware is not attached it still can be chosen but then the output will be zero the whole time. Every time there is pressed a button in the terminal this is printed in the following way:

```
C:\Users\eidew\Documents\GitHub\BAPIlluminator
there is 1 battery attached
there is 1 load attached
```

Figure F.3: Displays that a battery storage and a load is attached.

After this the Start date and amount of hours that the simulation need to run need to be filled in.The interfaces look like:

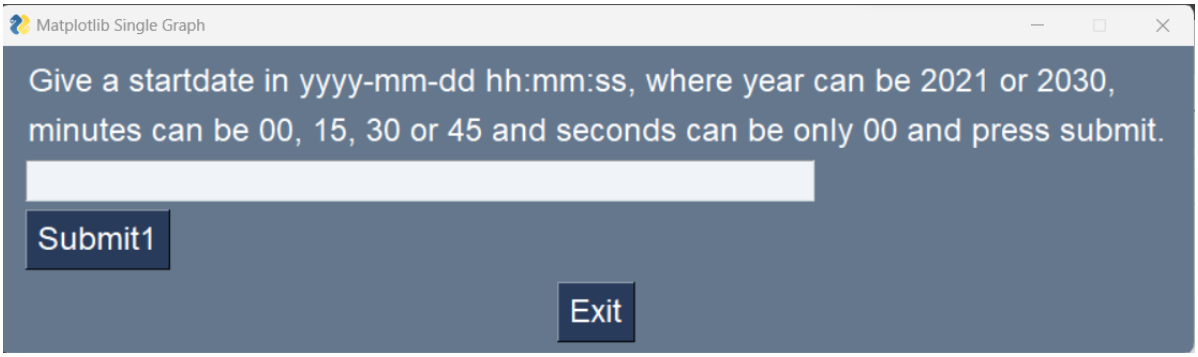


Figure F.4: Interface to put in the start date

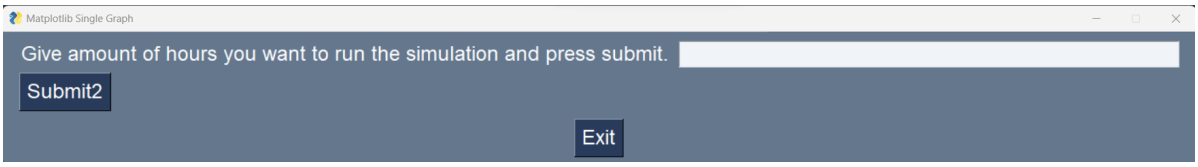


Figure F.5: Interface to put in the amount of hours

when they are submitted the terminal will print the start date and amount of hours in the following way:

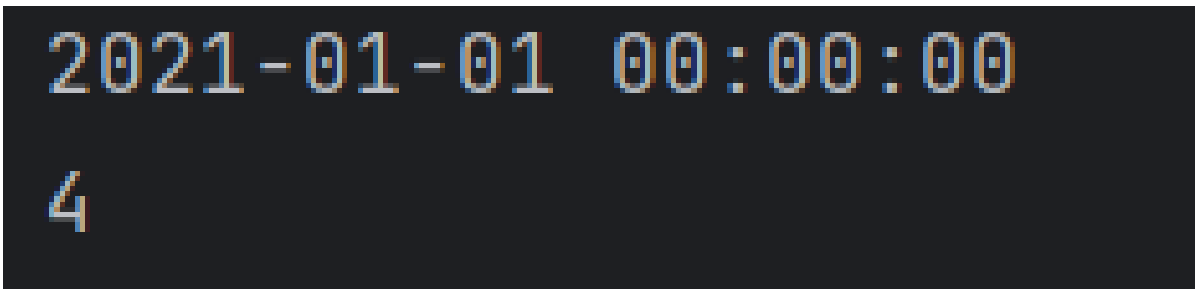


Figure F.6: Displays the start date and amount of hours the simulation will run

Now the simulation will start and during this graphs will be updated. This looks like the following figures:

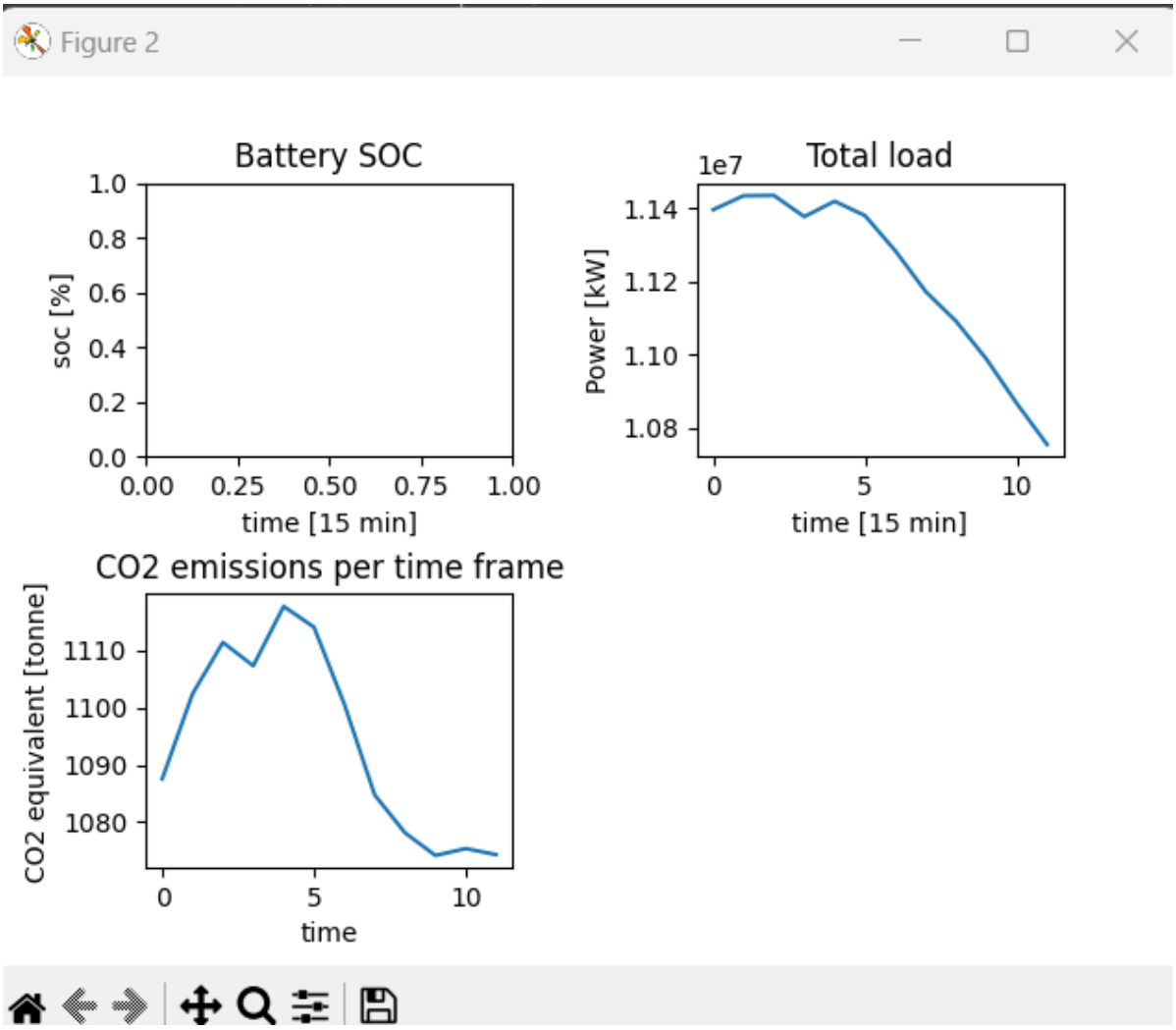


Figure F.7: Displays the battery storage state of charge, the total load, and the CO2 emissions of the current simulation

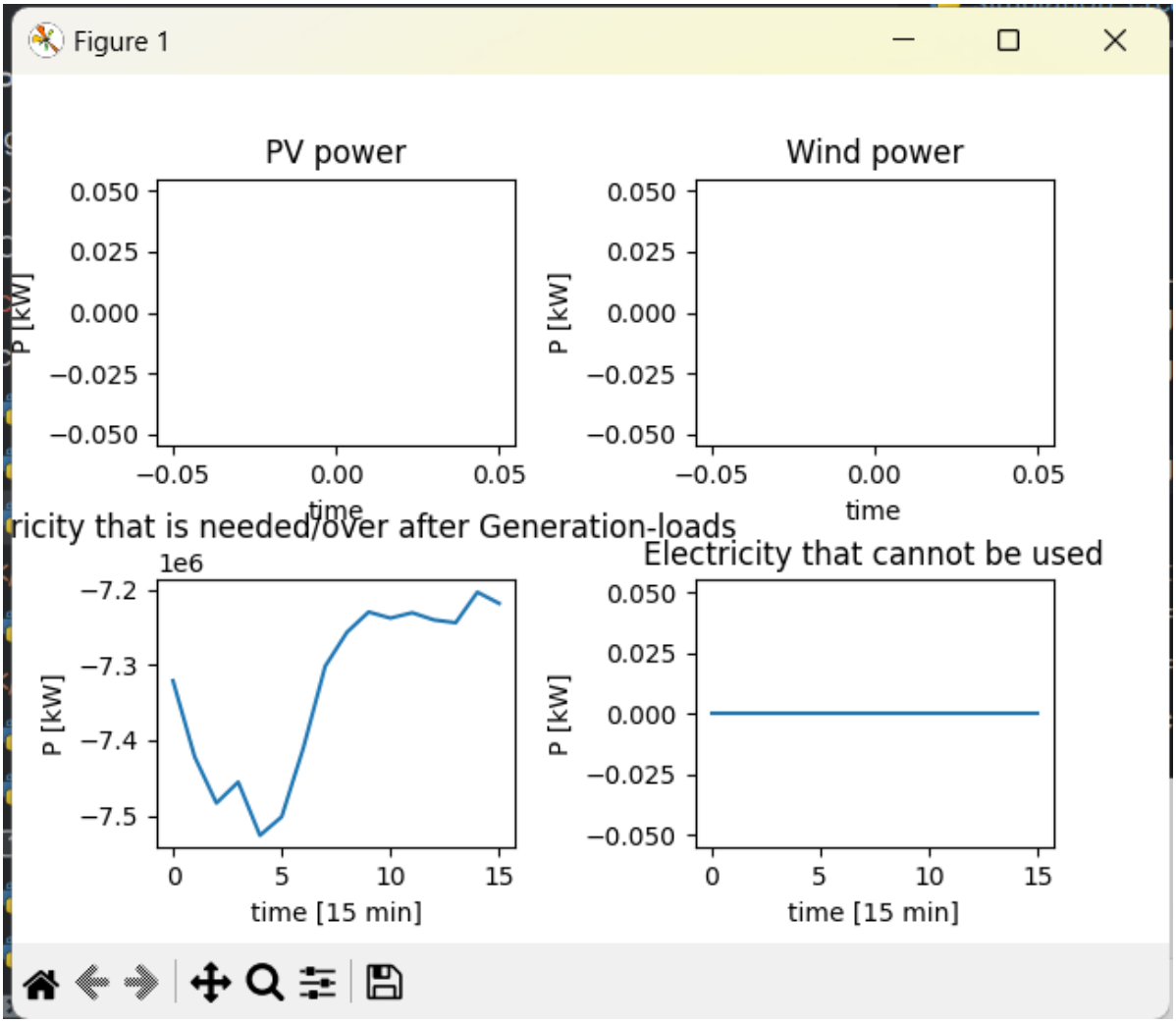


Figure F.8: Displays that there is no generation so all electricity that is needed for the load is in the graph bottom left

after the whole simulation has run the following interface pops up:

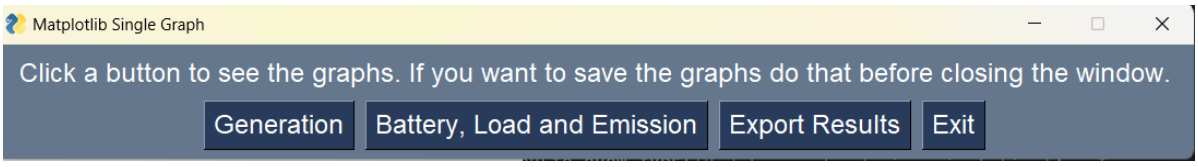


Figure F.9: Here there can be chosen which graphs one wants to see

If for example the generation button is chosen the following screen will show up:

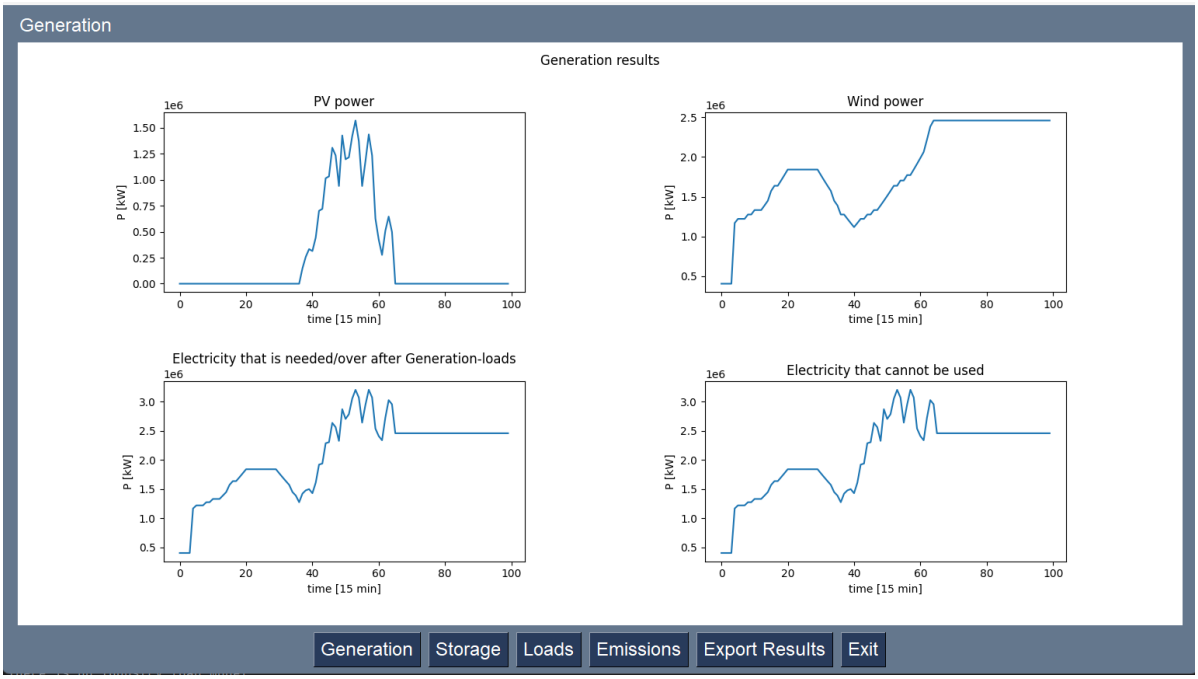


Figure F.10: Displays the electricity generation graphs from an attached PV and wind model.

If the button export results is chosen this screen will show up:

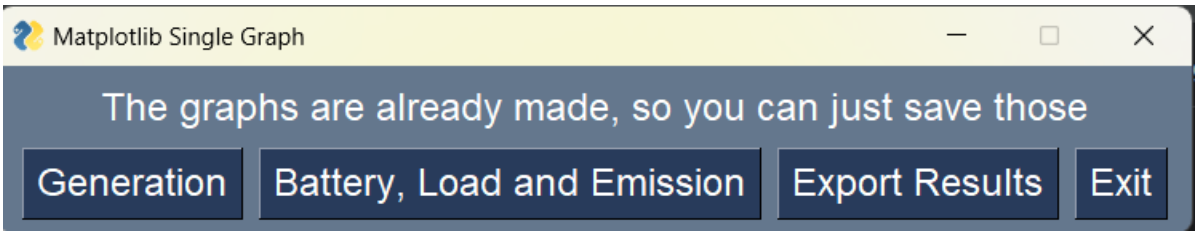


Figure F.11: Displays that these graphs are already made and can be saved

## F.2 Mode 2

For mode 2 after Figure F.1 is displayed, the following screen will show up:

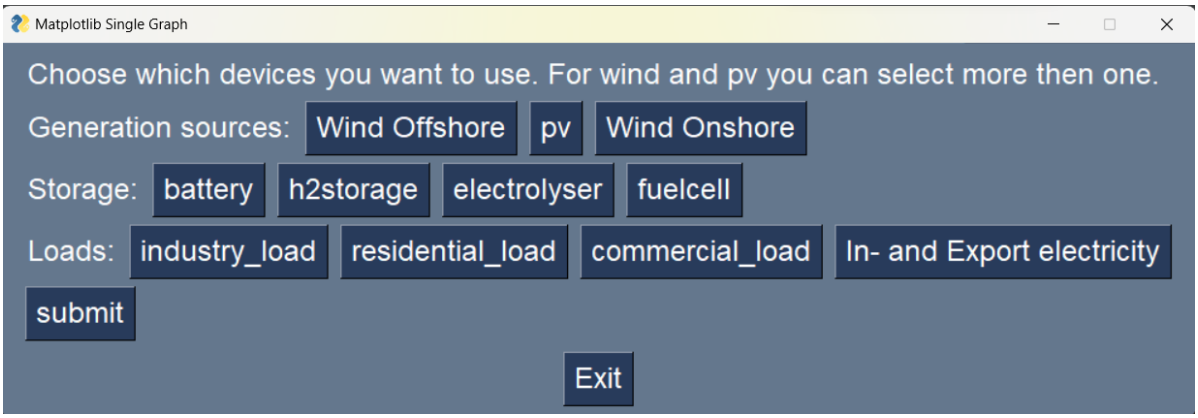


Figure F.12: Shows the models that can be used in mode 2

Then again which models are used are printed in the terminal in the same way as Figure F.11. After the models are chosen again Figures F.4, F.5 and F.6 are shown. After this all the simulation will start and when it has ended the following screen will be displayed:

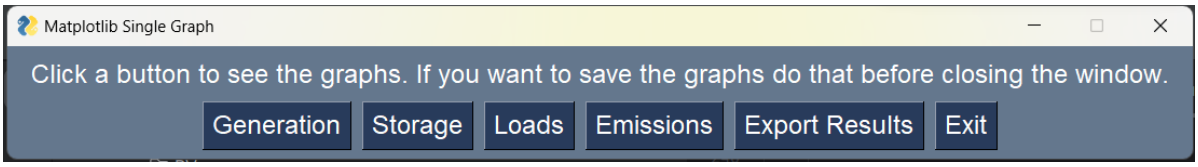


Figure F.13: Shows the models that can be used in mode 2

When now is pressed on the button export results the following figures will show up:

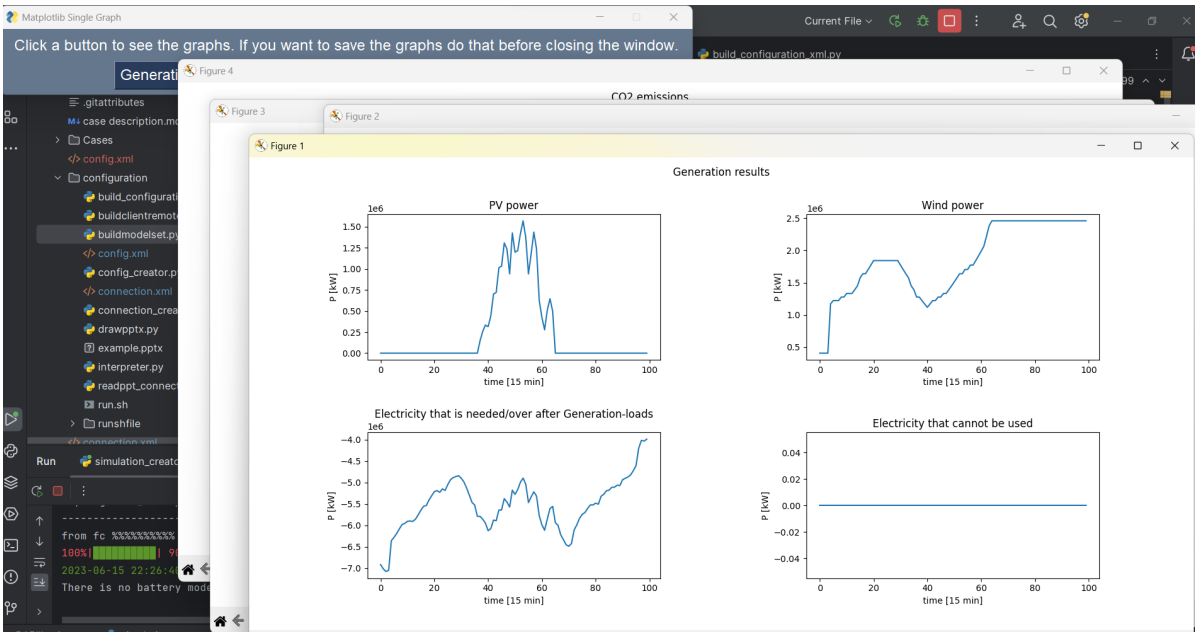


Figure F.14: All the made graphs are displayed in a format that can be saved