# BAP_A1: Enhancing Public Understanding of Energy Transition through Physical Modelling and Real-Time Visualization

## Bachelor Thesis Report

## EE3L11 Bachelor Graduation Project Electrical Engineering

J.I.Libier and R. Bonouvrie




TU Delft

# BAP_A1: Enhancing Public Understanding of Energy Transition through Physical Modelling and Real-Time Visualization

## Bachelor Thesis Report

by

J.I.Libier and R. Bonouvrie

Students Number: 5771226, 5594006
Supervisor: Dr. M. (Milos) Cvetkovic
Daily Supervisors: D. (Despoina) Georgiadi
Jort Groen

Project Duration: April 2025–June 2025
Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

TU Delft

# Abstract

The Energy System Integration Demonstrator, developed by the Illuminator team at TU Delft, is a modular, tabletop tool designed to educate and engage citizens in the benefits and challenges of the energy transition. It simulates important aspects of national electricity grids through a combination of Raspberry Pi control systems, 3D models, and LED visualizations.

The B.Sc. graduation project focused on enhancing the demonstrator's visualization and interaction capabilities. The project subgroup developed dynamic 3D models such as houses, windmills, cars, solar panels, carbon emissions, and batteries. On-screen elements are also created to represent key simulation agents like the battery state-of-charge and green energy percentage. Multiple dashboard versions were designed to tailor the experience to different stakeholder audiences. The report details the design, implementation, and validation of the visualized components.

# Preface

In the context of the Bachelor Graduation Project, this thesis is written to complete our Electrical Engineering studies at Delft University of Technology. It describes our contribution to the "Illuminator" project: a demonstrator for energy systems that is developed to raise awareness of the challenges of the energy transition among the public. Within this project, we developed a hardware interface and integrate the software with the peripherals, with the aim to create a scalable and interactive demonstrator.

We would like to express our gratitude to our supervisor Dr. Milos Cvetkovic, and daily supervisors Jort Groen and Despoina Georgiadi for their guidance and support for this project. We would also like to thank the other group members Bram Dorland, Alje Vermeer Kyrian Rahimatulla and Andy Zhang for this pleasant collaboration. Last but not least, we would like to thank our family and friends for their unconditional support during our studies.

# Contents

# 1 Introduction

Achieving climate neutrality by 2050 requires a fundamental transformation in our energy production and consumption. However, the increasing complexity that comes with this transition poses a major challenge. Engaging citizens is essential for meeting climate targets. The Illuminator project, developed by the Illuminator team at TU Delft, aims to bridge the gap between complex energy system and citizens with little prior understanding by providing an intuitive and interactive way to explore the benefits and challenges of this transition.

## 1.1. State of the art

Several tools exist for modeling, analyzing, or teaching energy systems, including Digsilent PowerFactory and ETAP. These tools also offer accurate results and are real-time but are not designed for educational use and lack intuitive visualizations, plug-and-play and interactivity. The Illuminator project, developed by the Illuminator team at TU Delft, intents on connecting the two by providing an intuitive and interactive way to explore the challenges and benefits of energy system integration.

One of the most effective ways to better understanding is through visualization. Research has shown that interactive and graphical representations of data can significantly improve the learning process, particularly for complex scientific concepts [1]. In science education, visualizations have been proven to support better understanding, critical thinking, and support, especially when learners are exposed to dynamic content [2].

## 1.2. The Illuminator project

The Illuminator project, developed at TU Delft by the Intelligent Electrical Power Grids (IEPG) group, addresses this gap through the development of a modular, Raspberry Pi–based demonstrator. The demonstrator represents a scaled-down energy system. It is designed to be modular, interactive, and educational. Users can arrange the setup to include models such as wind turbines, solar panels, electric vehicles, fossil generators, and batteries, making the system a versatile platform for demonstration and engagement [3].

## 1.3. Problem definition

The success of the energy transition relies not only on technological innovation but also on public understanding and acceptance of the fundamental changes in how energy is produced, distributed, and consumed. However, most tools currently available to explain these systems are either too abstract, too technical, or insufficiently interactive. This creates a barrier to engagement and slows down the potential for behavioral change and everyone's participation. Therefore, there is an urgent need for accessible, intuitive, and visually driven educational tools that can help close this knowledge gap between a wide range of audiences.

## 1.4. Contribution of the thesis

This thesis contributes to the Illuminator project in two main areas:

- A set of custom 3D models, enhanced with visuals such as LEDs or motion, that physically represent the behavior of models such as batteries, heat pumps, and fossil generators.
- A dashboard system for four user groups (educators, researchers, policymakers, and utility companies), integrating data from the simulation.

Together, these outputs aim to make energy systems more understandable and support the broader educational and outreach goals of the Illuminator platform.

## 1.5. Thesis structure

This thesis is organized into six chapters, each addressing an aspect of the project:

- Chapter chapter 2: Program of Requirements. Describes the functional and technical requirements of the system.
- Chapter chapter 3: Design and Justification. Describes the design choices for the 3D models and the interactive dashboard.
- Chapter chapter 4: Integration with the Illuminator. Details the implementation and integration of the design into the existing Illuminator framework.
- Chapter chapter 5: Results. Presents the outcomes of the project, evaluating the performance, functionality, and educational impact of the developed modules and dashboards.
- Chapter chapter 6: Conclusion and Discussion. Summarizes the findings of the project, evaluates whether the goals were met, and discusses potential improvements and future work.

# 2

# Program of Requirements

From the project proposal of the Illuminator team at the (IEPG) department [3], the aim is to educate all citizens about the challenges and benefits of the energy transition by visualizing energy assets.

The design requirements from the proposal are:

- The Raspberry-Pis (model 3B+) should be integrated in physical models to visually represent energy assets.
- Multiple Raspberry-Pis should be connected to represent an energy system configuration.
- Connections should be "plug and play" to enable flexible experimentation with different configurations.
- The connections should also visualize energy flow and load.
- Raspberry-Pis are preferably connected to individual (touch)screens to visualize model states.
- All peripherals are off-the-shelf components.

This thesis focuses on developing physical models to visually represent energy assets, as well as connecting the models to a dashboard to represent their model states. The Illuminator team wants to target a broad audience, including the public, students, and energy professionals, each with varying levels of technical expertise.

Potential use cases after the completion of this project include monitoring of energy consumption and generation of a neighbourhood powered by wind turbines, solar panels, and battery systems. This may involve homeowners tracking the output of their solar installations, or Distribution System Operators (DSOs) and Transmission System Operators (TSOs) analysing grid stability as renewable energy penetration increases.

The following design considerations were emphasized by the project supervisors:

- The system should be user-friendly for non-experts.
- Each Raspberry-Pi does not need their own screen, the data can be displayed together on a main screen.
- The system must work within the existing Illuminator architecture and software (Mosaik based).
- The simulations should be real time.
- Interactive visualization is appreciated to enhance engagement of non-expert users.
- Grafana is the preferred tool among the presented visualization options.

# 3

# Design and Justification

The Illuminator tabletop demonstrator is based on the Raspberry Pi single board computers. According to the official documentation [4], each Raspberry Pi has 40-pins that supports alternative functions. Some of the functions are Pulse Width Modulation (PWM), I2C, SPI and serial communication.

The single board computers also have a variety of interface options: HDMI, USB ports, CSI (Camera Serial Interface)-port, DSI (Display Serial Interface)-port, analogue AV-jack, Ethernet, Wi-Fi, Bluetooth, microSD for storage and operating system and a micro-USB-port for power supply. Figure 3.1 displays the hardware layout.

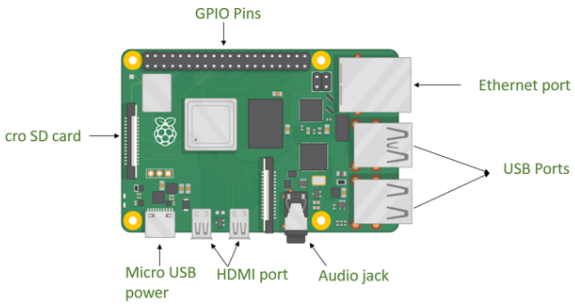



**Figure 3.1:** Raspberry Pi connectivity [5].

Following the Program of Requirements (PoR), different off-the-shelf components are considered based on compatibility and ease of integration with the Raspberry Pi. Chosen components assures a balance between modularity, performance, and simplicity in implementation.

The Illuminator software has different models to visualize, as can be seen in their documentation [6]. How they are visualized are divided in two categories:

- 3D models. This includes the 3D visualization of battery, load, heat pump, electric vehicle (EV), solar panel (PV), wind turbine and fossil fuel power plant. The fossil fuel power plant model is developed by another subgroup (A.3 [7]) and is not part of the Illuminator team's implementation.
- Dashboard Visualization. Model data to be visualized are specified in the YAML file of the study case. The data will be visualized on Grafana-dashboard, where real time monitoring and interaction is possible. This dashboard complements the visualization by displaying absolute values.

Additionally, the PV and wind turbines models have been designed with an interactive touch functionality. Together with Grafana's interactive functionality, the whole system would have an interactive and demonstrative value.

## 3.1. The design of the 3D models

To meet the PoR criteria—modularity, flexibility, and visual relevance — most energy assets are implemented using off-the-shelf components. Some components are symbolic (non-functional), selected primarily for their intuitive aspect in 3D visualization, to ensure fast prototyping within the project's timeframe.

This project implements a real time physical simulation where each physical model represent a physical energy assets. Each model is coupled to Raspberry Pi in a compact 'model box' that also has the necessary circuits and hardware. These model boxes together form a visually interactive table-top model for public interaction and education purposes.

The design follows a modular structure where each model operates individually. With this modular approach, it is easy to scale, debug and is resilience to single-point failures.

Each Raspberry Pi is synchronised in a cluster setup done by subgroup A.2 [8] and controls a single energy asset with the General Purpose Input/Output (GPIO)-interfaces.

All model up-to-date prototype STL file can be found here.

### 3.1.1. Physical and hardware design choices

#### Use of jumper cables

Jumper cables are chosen to connect the Raspberry Pi with the sensors and actuators. This is due to their flexibility, easy to rearrange during prototyping, and well-suited for GPIO experimentation. Although custom PCB's or hook-up wire offer more reliability, jumper cables provides fast iteration and debugging possibilities, which is essential considering the period available for the project.

#### Raspberry Pi 'model box'

The electrical components of each model are encapsulated in an independent 'Raspberry Pi box'. Each box has a cooling fan[9] on top of the Raspberry Pi chip (the BCM2837b0) for ventilation. The cooling fan has similar specification as the one recommended in Raspberry Pi documentation[4]. This design simplifies the modularity by making sure that:

- it is easier to test and debug each model.
- shortage of GPIO pins are avoided.
- it is less likely that the supplied current by the power supply is not enough.
- maintenance and replacements can be done to one box without disturbing the whole system.
- new models or Pi nodes can be added without reengineering the whole system.

### 3.1.2. Communication architecture between Raspberry Pi's

For the Raspberry Pi to synchronize with each other, there should be a communication architecture. A master-slave cluster setup through SSH is chosen as described in the report of subgroup A.2[8]. With this architecture, all slaves are assigned a specific model. The models are modified so that the script running on the models can control the GPIO pins directly. This approach is also chosen by subgroup A.2 **group_b**, who makes use of direct data control of the LED-strips.

Other options considered were:

- To transfer CSV files to slaves of Raspberry Pi's running the model. This could be done in theory with Secure Copy Protocol (SCP) through SSH. However, the functionality is being used by Mosaik during simulation, therefore SSH could not be used for file transfer. File transfer could be done after the simulation, but this requires extra coding which makes the software more complex. Furthermore, one of the requirements stated in the PoR was that the system must work within the existing Illuminator architecture and software (Mosaik based). Adding script after the simulation violate this requirement.

- To use HTTP to receive the output file generated at each simulation step. This file is then used to control the GPIO pins. HTTP could be a solution as it can operate simultaneously to SSH. However, it requires an HTTP-server with upload functionality and additional management to pause the Illuminator script, perform operation on the GPIO pins and then continue the script. This method requires multi-threading, signal handling, or inter-process communication, thereby increasing the implementation effort and system overhead.

Thus, one benefit above the last two option mentioned is that it offers real-time control. However, a downside is that by downloading the repository from GitHub as-is, one needs a Raspberry Pi (cluster setup) to run a tutorial or needs additional coding to bypass the errors.

### 3.1.3. Modelling of 3D Components

#### Battery

The battery is represented by a hollow, 3D printed battery with LED strip inside. The LED colours change from deep red to green to represent the State of Charge. With this, the battery is easy to implement with the Illuminator model. Its prototype can be seen in figure 3.2.

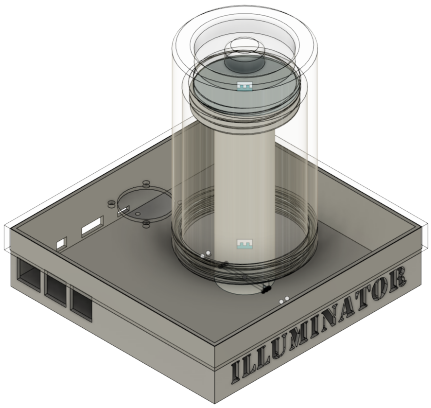



**Figure 3.2:** Prototype of the battery.

#### Load and Electric Vehicle

The model of the load and EV are abstract representation without upper or lower boundaries. Each model is displayed as a hollow 3D model (a house for a load and a car for an EV) with a LED strip inside. The LED colours show the relative energy consumption. Absolute values are displayed on the dashboard. Its prototype was not finished by the time of writing, but the prototype can be found by clicking this link. For an intuition, both the load and the EV prototype looks similar to the battery prototype: a 'Raspberry Pi box' with a clear object on top.

#### Heat Pump

The heat pump is modelled as a cuboid with a mini fan inside, representing a typical outdoor unit of a heat pump. This model is controlled through the GPIO and turns on if there is an active heat pump in the simulation. Its prototype can be seen in figure 3.3.

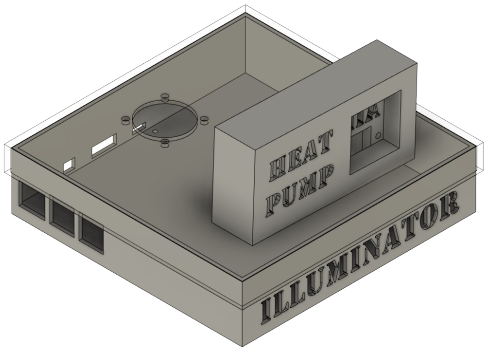



**Figure 3.3:** Prototype of the heat pump.

### Fossil Fuel

This model is designed as a smokestack with an ultrasonic mister that creates visible 'fume' to represent the emissions. This gives an intuitive visualization of the fossil fuel activities. The prototype can be seen in figure 3.4

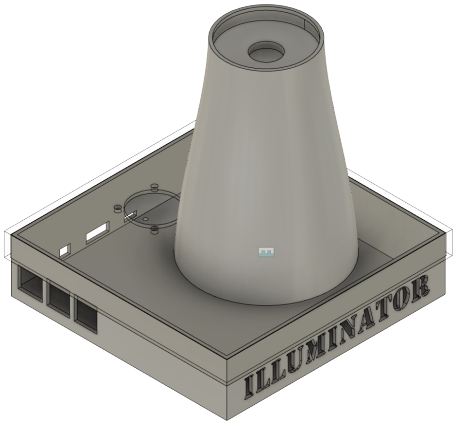



**Figure 3.4:** Prototype of the fossil fuel

### Solar Panel

Solar panel power production is presented by a LED. A photocell is embedded in a 3D printed panel to make interactivity possible. When exposed to light (e.g., a torch), new data lines will be appended to existing input data files. Afterwards, a new simulation scenario will start consisting of the new data lines. Data files will be transferred through SCP to the master so that the master can start the new simulation. The prototype of the solar panel can be seen in figure 3.5

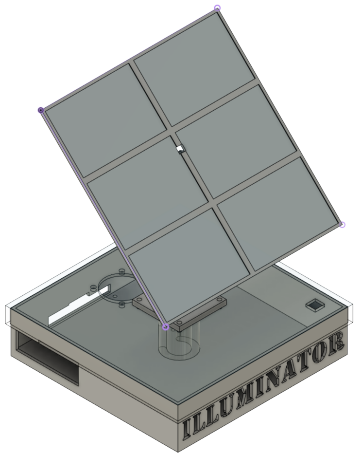
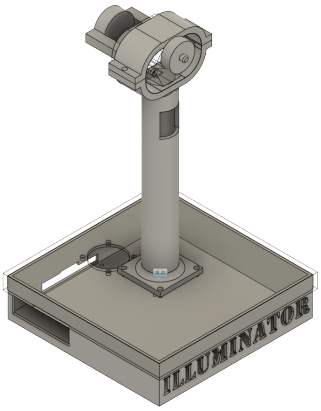


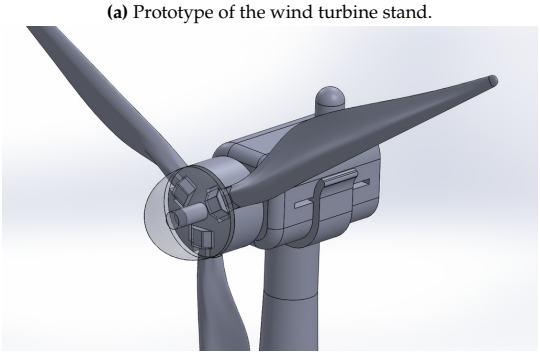

**Figure 3.5:** Prototype of the solar panel.

### Wind Mill

A wind turbine with 3D printed blades and a Stepper motor rotates proportional with the simulated produced energy. For the interactivity, a magnetic sensor detects a magnets on one of the blades when it is rotating. Same as the solar panel, new data lines will be appended to existing input data files and the simulations will start again for the new data lines. Data files will be transferred through SCP to the master before simulation starts. The prototype of the wind turbine can be seen in figure 3.6.




**(a)** Prototype of the wind turbine stand.



**(b)** Prototype of the wind turbine blades obtained from [10]

**Figure 3.6:** Prototype of the wind turbines.

## 3.2. Dashboard design

In addition to the 3D models, some parameters and models are better represented through a dashboard. The dashboard helps to bring together data from the simulation models in the Illuminator system, and for presenting it in a way that different audiences can understand and act upon.

Four separate dashboards were designed for four different user groups: policymakers, educators, researchers, and utility companies. Grafana was selected as the visualization platform for each dashboard due to its flexibility, real time data hold, and ideal integration with InfluxDB, the time-series database used to store the simulation output[11][12]. While alternative tools such as Node-RED[13] were also considered for their interface and integration capabilities, grafana was preferred by the project supervisors. The Raspberry Pi 3B+ has limited processing power, which results in slow performance. To address this, the Python code running on the Raspberry Pi accesses the InfluxDB instance hosted on a separate PC. Consequently, the InfluxDB token is configured to include the IP address of the PC to enable remote access. InfluxDB is used to collect data from different models and agents in the simulation, such as the PV model, Wind model, Fossil fuel, Battery, Load, and agents such as the Market Operator and Justice Agent. These outputs are logged and indexed in time, making them available to Grafana panels for visualization as shown in diagram figure 3.7. Grafana makes use of customizable panels such as time series plots and gauges.

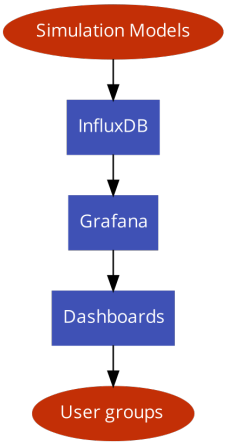



**Figure 3.7:** Diagram from models to users

### 3.2.1. Education

The educational dashboard was developed to support both students and members of the general public in building a better understanding of how energy systems work and how individual actions have an impact on broader sustainability goals. This version focuses on accessibility, clarity, and an easy to understand design.

The dashboard presents data on energy consumption, renewable generation (solar and wind) and battery usage. These elements will be visualized by time series plots, gauges, and statistical panels, each with a short descriptive text that explains their meaning.

### 3.2.2. Policymakers

The dashboard for policymakers is designed to give a clear overview of the performance of the energy system, focusing on indicators that are relevant to make decisions about sustainability and climate policy. Instead of technical details, this dashboard shows high-level trends and summaries that are easy to understand and to act upon.

The main panels include total energy demand and generation, green energy percentage, $CO_2$ emissions, battery charge level, fossil fuel use, and nuclear waste produced. For example, a time series graph compares energy demand and generation to help policymakers see whether the system is balanced. A gauge shows the percentage of energy coming from renewable sources, which helps track progress

toward clean energy goals. Another panel shows how much $CO_2$ has been emitted over time, based on fossil fuel use, while nuclear waste is shown as a separate time series to highlight the long-term environmental impact.

### 3.2.3. Utility companies

The dashboard for utility companies is designed to provide a detailed and operationally useful view of the energy system. It focuses on technical measures that are relevant for managing supply, demand, system efficiency, and emissions.

The main panels include total power generation and demand, renewable energy output, $CO_2$ emissions, battery state of charge, and detailed outputs from fossil and nuclear sources. For instance, a time series graph compares total generation against load demand, allowing operators to evaluate system balance and potential over- or underproduction.

### 3.2.4. Researchers

The dashboard for researchers is designed to give detailed access to all important simulation data needed. Unlike dashboards for general users or policymakers, this one includes more technical information and allows users to explore how the system behaves under different conditions. The goal is to help researchers study, compare, and validate different setups of energy systems.

The dashboard includes time series panels for model outputs, such as solar and wind generation, load demand, battery power, fossil fuel use, and controller actions. Researchers can view each signal to study the system's response over time. For example, they can see how the battery responds when the solar energy production falls.

Because of grafana's support for custom queries, filters, and data transformations it makes it easy to compare different scenarios. Researchers can select specific time ranges, zoom in on parts of the graph, or calculate new metrics (like averages) directly in the dashboard. This makes the dashboard useful for more detailed analysis.

All data is streamed from InfluxDB, which stores output from the simulation at each time step. This allows researchers to monitor and analyse the system while it runs, without needing to export data or write any additional code.

# 4

# Integration with the Illuminator

## 4.1. Implementation of the 3D models

### 4.1.1. Hardware integration

Each model is controlled by the GPIO pins of its respective Raspberry Pi. The models are powered by the power supply pins on the GPIO header or by the USB port. LED strips that are more than ten are powered by an external power supply to prevent overload.

### 4.1.2. Software integration

ALL GPIO interactions are located within the `step()` function of the Python script of each model. This function fetches the latest simulation values and directly translates them into hardware commands. Against the other method mentioned in section 3.1.2, this method guarantees up-to-date access to variables and real time simulation as stated in the PoR.

Models that have interactivity append new data lines to their input file. For models without interactivity, data of desired date is extrapolated and added as new data lines. After this, a new simulation can start for specified date and time to see the effect of the new appended data lines.

### 4.1.3. Assembly of the models

This section addresses the assembly of each model. The python code added in each model is available in appendix A.1

#### Battery

The electrical components that are used include:

- Twenty individual LEDs on the addressable LED strip WS2813[14] by NeoPixel.
- High speed level shifter[15] (essential due to strict timing of the LEDs).

The electric circuit can be seen in figure 4.1

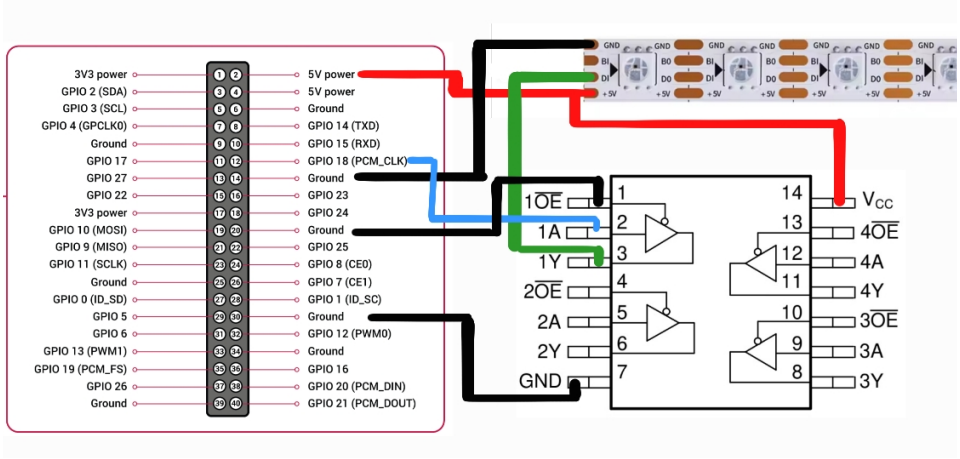
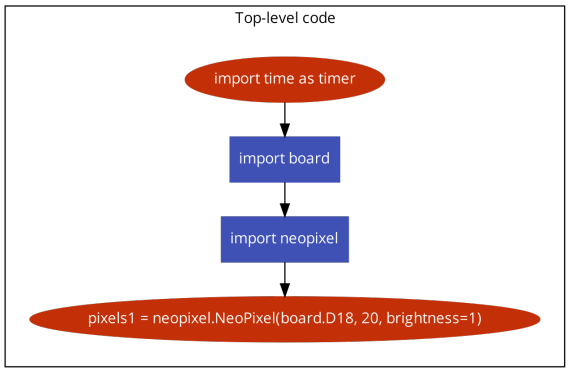
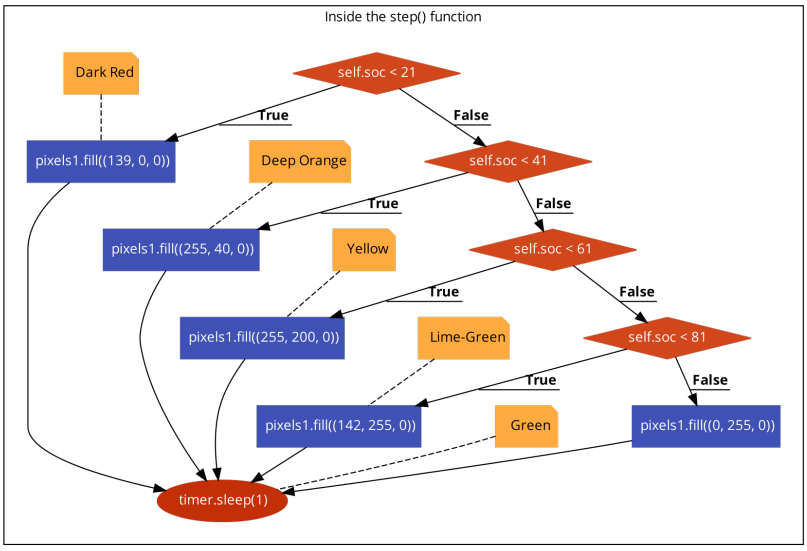



**Figure 4.1:** LED strip electrical connection. Images obtained from [4] and [15].

The variable `self.soc` is used every simulation step to assign colours to the LED strips. The colour gradient progresses in 20% increments from deep red, deep orange, yellow, lime-green and green. The flowchart can be seen in figure 4.2




**(a)** Top-level code flowchart of the battery.



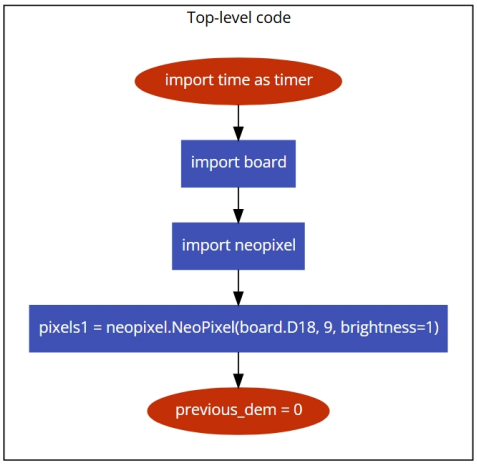

**(b)** Flowchart of the battery inside the `step()` function.

**Figure 4.2:** Flowchart of the battery.

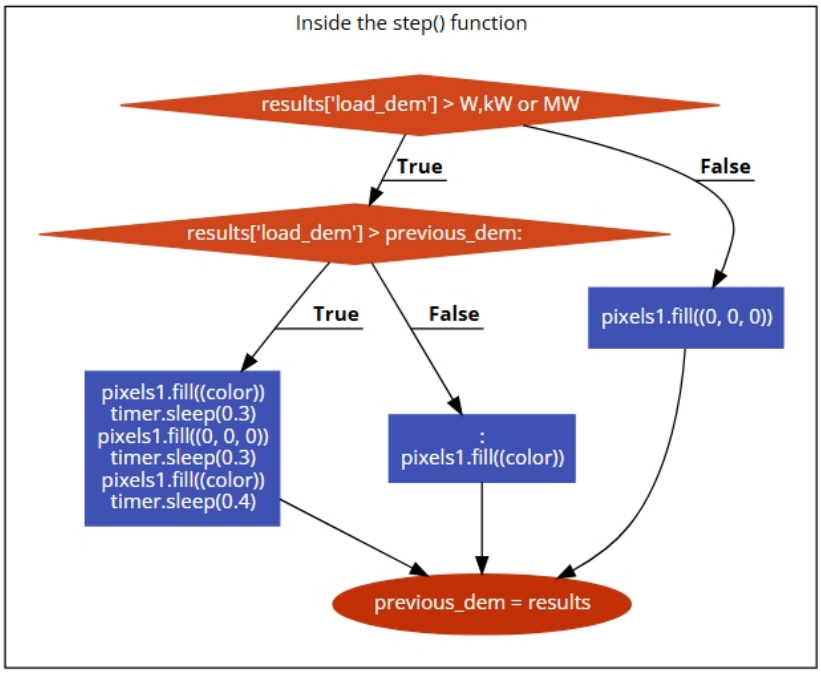
### Load and Electric Vehicle

The Load has nine LEDs, while the EV has seven LED. Both model has the same LED strip circuitry as the battery (figure 4.1).

There is a conceptual mapping for the LED display. Green is for base unit $W$ or $Wh$, yellow is for $kW$ or $kWh$ and red is for $MW$ or $MWh$ and above. LEDs blinking indicate load demand is increased, while constant illumination indicate constant or decreased load demand. The flowchart of the load can be seen in figure 4.3. The flowchart of the EV is identical, except that the variable `results['load_dem']` became results['load_EV'].




**(a)** Top-level code flowchart of the load.




**(b)** Flowchart of the load inside the `step()` function.

**Figure 4.3:** Flowchart of the load.

### Heat Pump

Electrical component needed are :

- a $5V$ mini fan [9].
- an NPN Bipolar Junction Transistor (BJT) for switching [16].

- a $1k\Omega$ resistor for the base resistor.

The datasheet of the NPN BJT gives a wide range of possible DC current gain. A safe base resistor value in this range is $1k\Omega$. The interconnection between the Raspberry Pi and the components can be seen in figure 4.4.

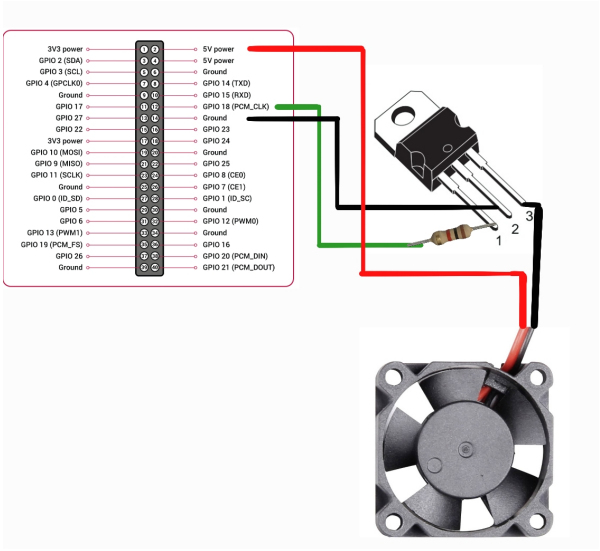
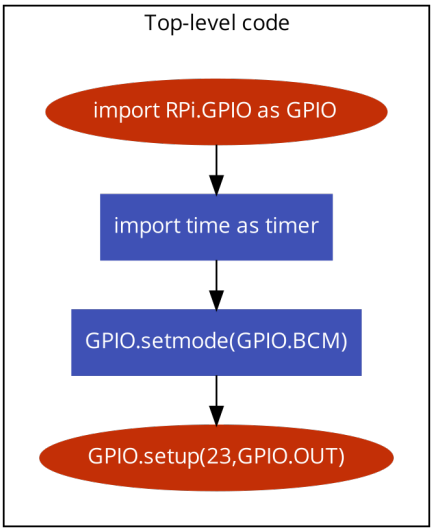
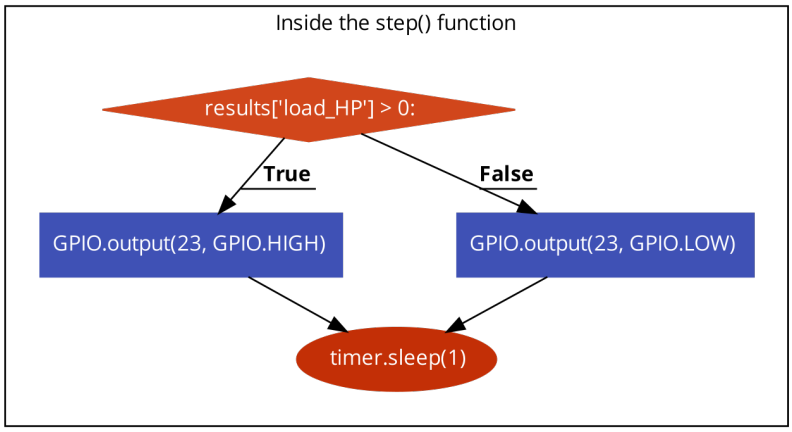



**Figure 4.4:** Heat pump electrical connection. Images obtained from [4], [9] and [16]

The fan is toggled on/off depending on the variable `results['load_HP']`. The flowchart can be seen in figure 4.5.




**(a)** Top-level code flowchart of the heat pump.




**(b)** Flowchart of the heat pump inside the `step()` function.

**Figure 4.5:** Flowchart of the heat pump.

### Fossil Fuel

The used components are:

- an ultrasonic mister[17].
- an NPN BJT for switching the component on and off[16].
- a $1k\Omega$ resistor for the base resistor, which is same as for the fossil fuel model.

The ultrasonic mister has a mechanical switch to turn the mister on/off. To control the switch with the GPIO pins, a NPN BJT is placed between the pin of the switch that is connected to ground and the pin which turns the mister on. The interconnection of the components is displayed in figure 4.6. In practice, this design proved to be not very robust. Therefore, an alternative configuration will be evaluated to determine if it offers better performance. This revised setup is shown in figure 4.7. In this configuration, the Invalid Switch is enabled and the NPN BJT controls the power to the mister.

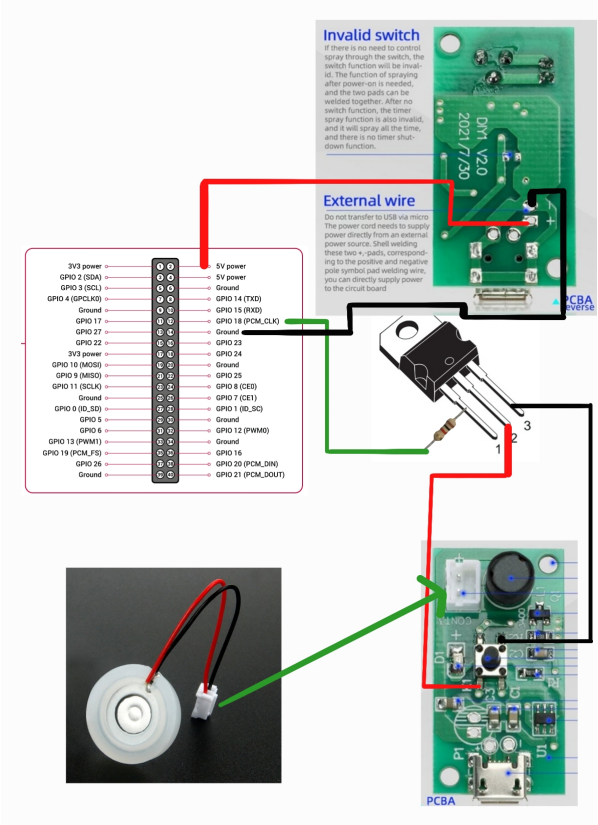



**Figure 4.6:** Ultrasonic mister electrical connection. Images obtained from [4], [17] and [16].

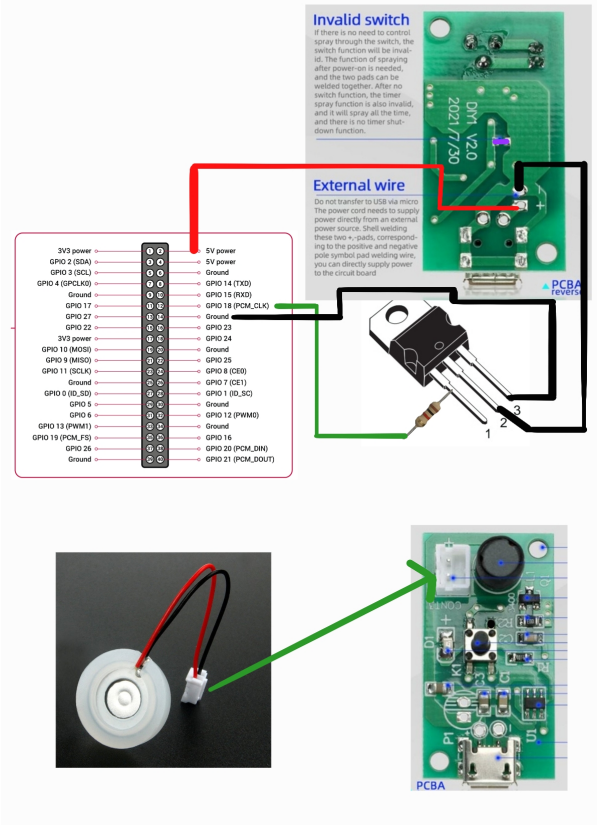



**Figure 4.7:** Revised ultrasonic mister electrical connection. Images obtained from [4], [17] and [16].

The mister is toggled on/off depending on the variable `self.outputs['gen_pow']`. The flowchart is identical to the heat pump (figure 4.5), except that the variable controlling the logic, which is now `self.outputs['gen_pow']`.

### Solar Panel

The components used include:

- same LED circuitry components as the battery (section 4.1.3).
- photocell [18] and analogue to digital converter (ADC) [19].

For the interactivity aspect, the photocell in the middle of the 3D-printed panel measures the ambient light. This measurement represent the `G_Gh` variable in the `step()` function. The variable `G_Dh` is set equal to `G_Gh` and `G_Bn` is set equal to zero for simplicity. Thus, the solar zenith angle $\theta_z$ is set to 90° making the irradiance conversion valid: $G_{Gh} = G_{bn}cos(\theta_z) + G_{Dh}$. The other variables in the input file are imported from the OpenWeatherMap API key.

The photocell measurement is converted to a digital signal and is read by a GPIO pin. The circuitry can be seen in figure 4.8. The reversed polarity at the photocell makes the minimum value represent a 'no light' scenario. The flowchart of the code added in the PV model is displayed in figure 4.9. A separate Python script has been added for the interactivity. The script should be ran separately to update the input TXT files. The flowchart of the interactivity can be seen figure 4.10.

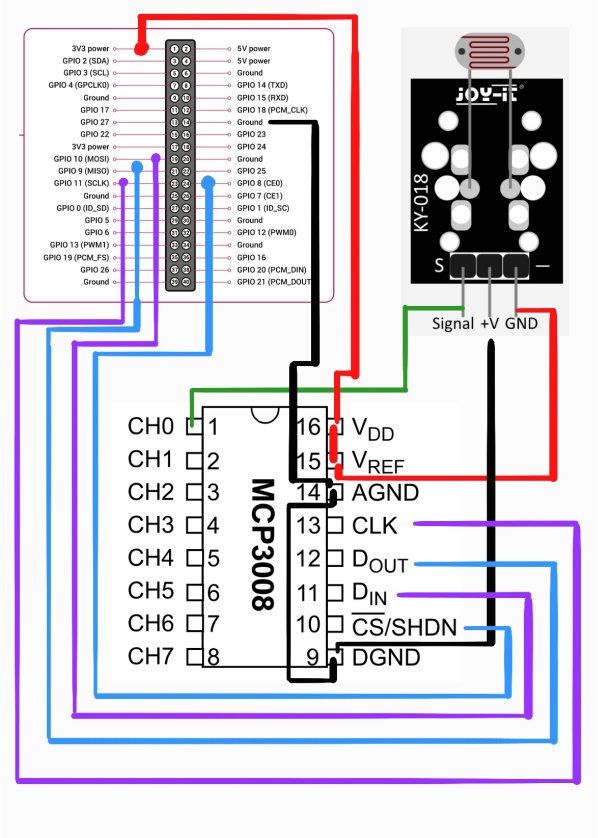
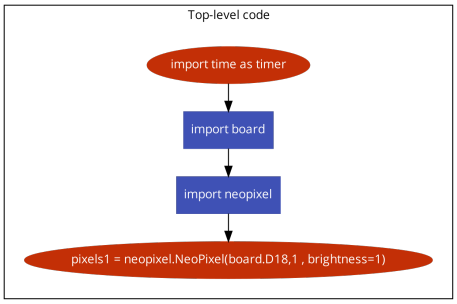
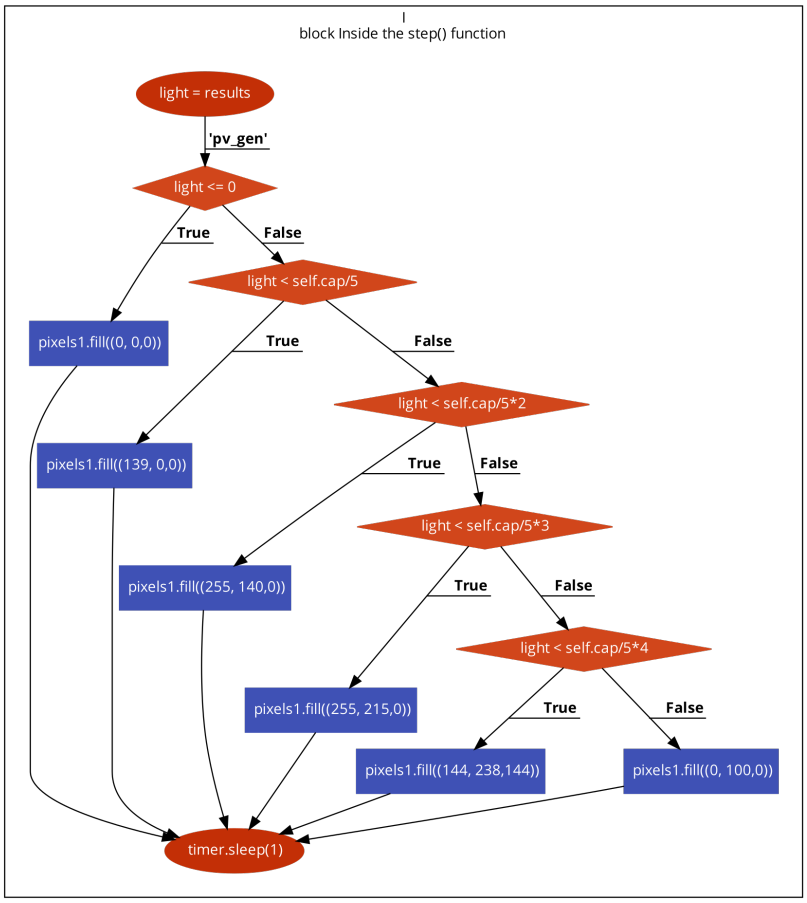



**Figure 4.8:** Photocell electrical connection. Images obtained from [4], [19] and [18].




**(a)** Top-level code flowchart of the solar panel module.




**(b)** Flowchart of the solar panel inside the `step()` function.

**Figure 4.9:** Flowchart of the solar panel model.

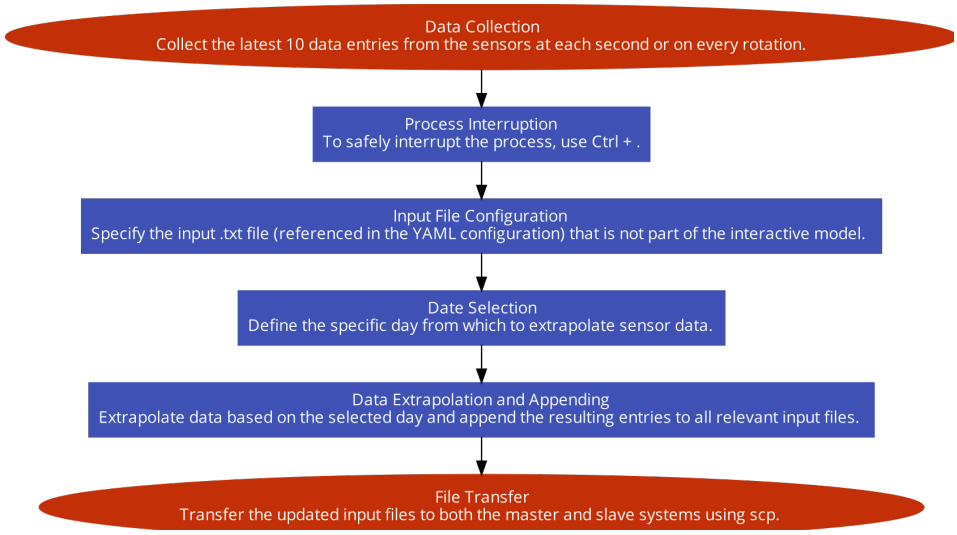



**Figure 4.10:** Interactive flowchart of solar panel and wind turbine. Python code can be found in appendix A.1.7

### Wind Mill

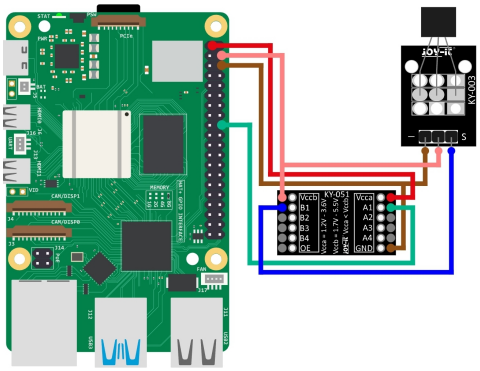
The components used include:

- a stepper motor to rotate the blades [20].
- Hall-effect sensor[21] for detecting the magnet located on the blade.
- a level shifter[22] for the sensor. The sensor itself is 3.3V, but the LED on the board is $5V$ Thus the power and logic level data must be $5V$.

The variables `self.p_rated` and `results['wind_gen']` in the `step()` function are used to calculate the RPM of the stepper motor.

For the interactivity aspect, the Hall-effect sensor detects the magnet on the blade. The time between the detections are converted to wind speed variable `u` in the input file.

The circuitry of the wind turbine can be seen in figure 4.11 and the flowchart of the code added inside the wind model is displayed in figure 4.12. Similar to the solar panel module, a separate Python script has been added for the interactivity. The script should be run separately to update the input TXT files. The flowchart of the interactivity can be seen figure 4.10.



**(a)** Motor electrical connection. Image obtained from [23].

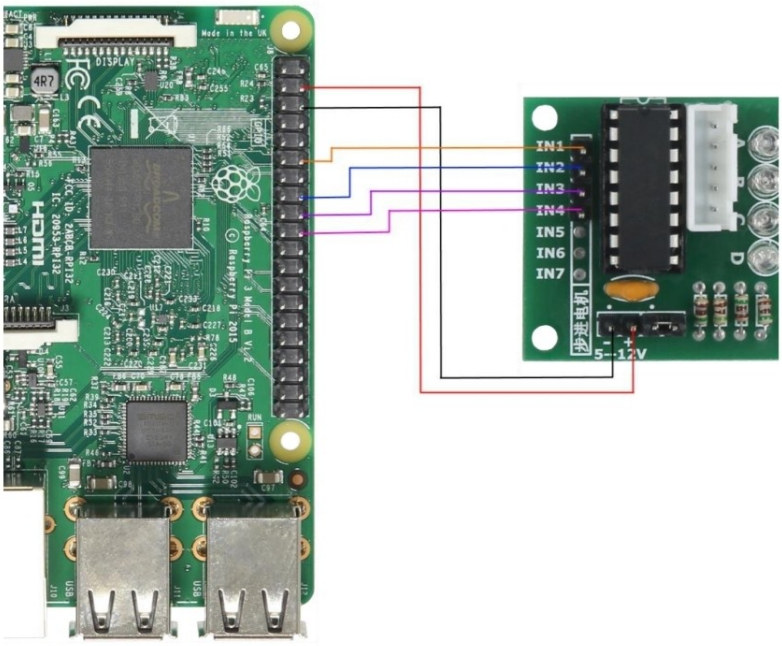



Bild 5: Verkabelung mit Raspberry Pi

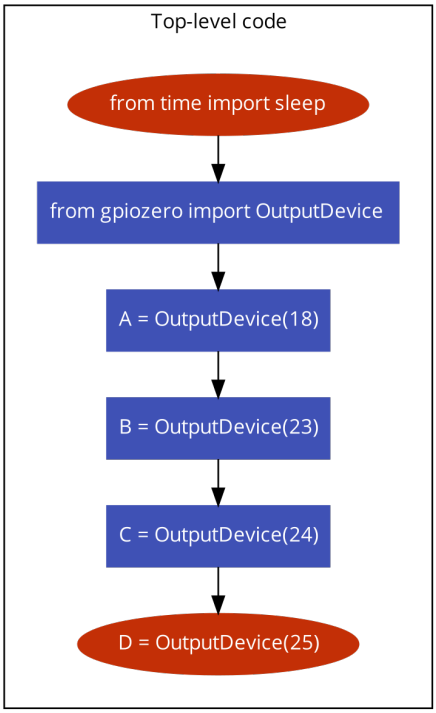


| Raspberry Pi PIN | Treiber-Board PIN |
|---|---|
| PIN 12 (BCM 18) | IN1 |
| PIN 16 (BCM 23) | IN2 |
| PIN 18 (BCM 24) | IN3 |
| PIN 22 (BCM 25) | IN4 |
| PIN 4 (5v Power) | + |
| PIN 6 (Ground) | - |

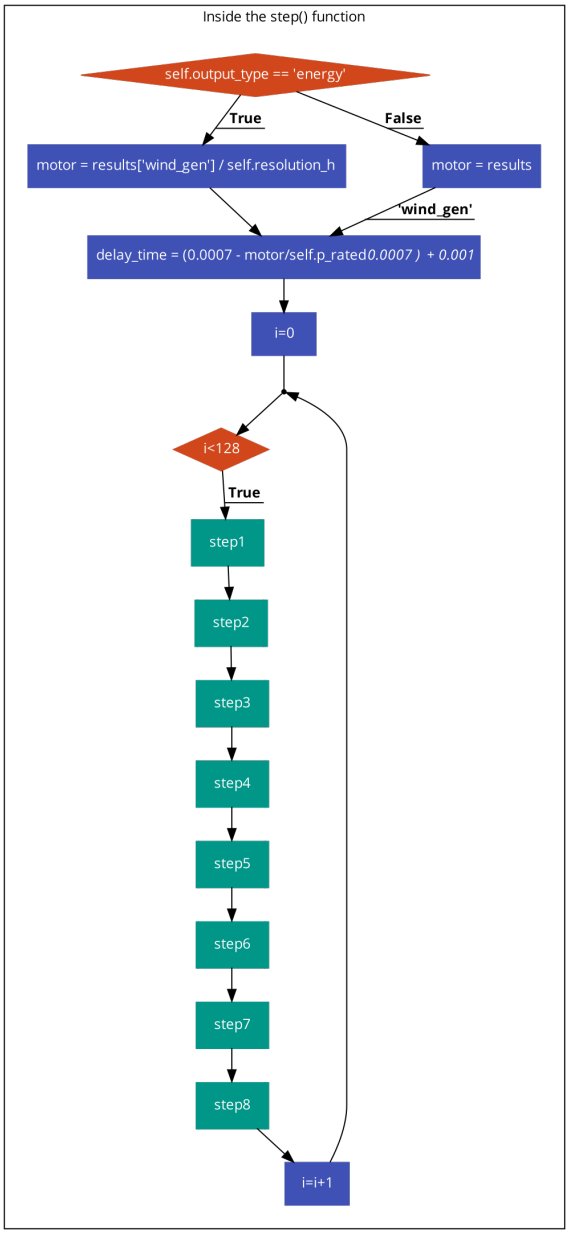
Tabelle 2: PIN-Verbindung zwischen Rasperry Pi und Treiber-Board

**(b)** Hall-effect sensor electrical circuit. Image obtained from [20]

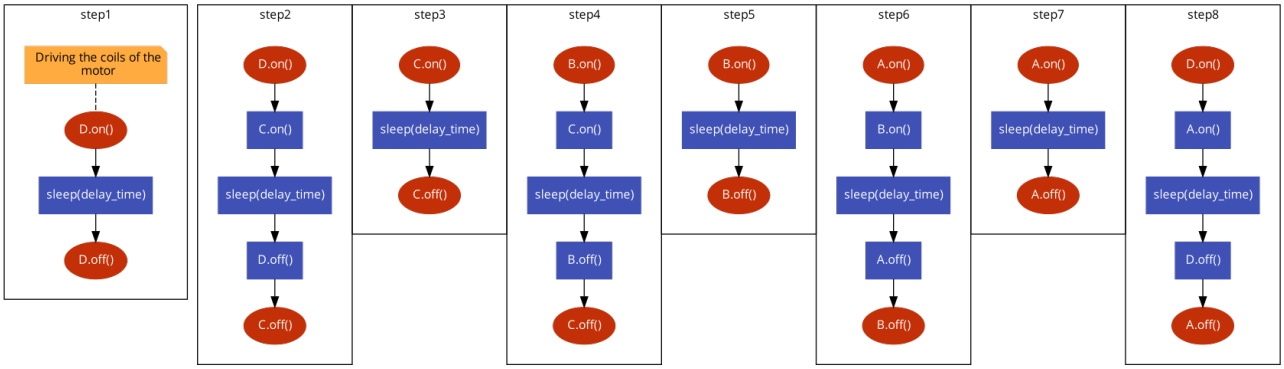
**Figure 4.11:** Wind turbine electrical connection.




**(a)** Top-level code flowchart of the wind turbine module.




**(b)** Flowchart of the wind turbine inside the `step()` function.




**(c)** Flowchart of the wind turbine motor functions inside the `step()` function.

**Figure 4.12:** Flowchart of the wind turbine model.

## 4.2. Dashboard

To get real-time visualizations of simulation results, a data pipeline is implemented to automatically transfer outputs from the simulation models and agents in the Illuminator framework into InfluxDB, the time-series database used for the dashboards. This is done by extending the collector.py script, which totals the results from the simulation environment and making a change in the engine.py code.

Inside engine.py the simulation scenario includes a `results_show` dictionary that configures which output channels are active. Originally, the `dashboard_show` was set to False, disabling the InfluxDB connection. For this project, this is switched to True, as shown in appendix B.0.1, to activate real-time dashboard logging.

The implementation activates conditionally via the `dashboard_show`, allowing the same simulation to run with or without visualization. When allowed, the simulation connects to an InfluxDB instance using the influxdb-client-python library. The configuration specifies the target bucket, organization, and authentication token for secure access. This is shown in appendix B.0.2.

At each simulation time step, the collector's step() function creates a pandas DataFrame that collects outputs from all connected models. Each row of the DataFrame corresponds to a single point in time and includes all relevant fields, such as `PV1-0.time-based_0-pv_gen` or `"Battery1-0.time-based_0-soc"`. These fields are then transformed into InfluxDB-compatible "points" with timestamps and written to the `energy_metrics` measurement. The data is then ready to be used in grafana. This piece of the code can be seen in the appendix B.0.3. The flowchart of the process from models to the grafana dashboards is shown in figure figure 4.13.

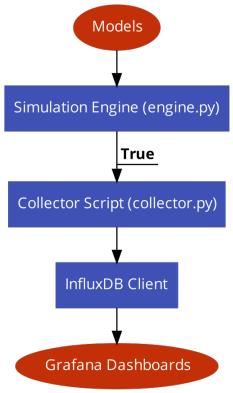



**Figure 4.13:** Flowchart model to dashboard

# 5 Results

## 5.1. Evaluation of the 3D models

3D models developed in this project are evaluated based on their robustness, integration with the Illuminator software and their conformity with the PoR. All models operate under real time conditions and visualize simulations results. Electrical output results are not directly available since the model communicate their states visually (e.g., LEDs, rotation, mist).

An overview of the models with their features is presented in table 5.1 below and in figure 5.1 till figure 5.4.

**Table 5.1:** Overview of Developed 3D Models

| **Model** | **Functionality** | **Physical Output** | **Interactivity** | **Robustness** |
|---|---|---|---|---|
| Battery | LED strip visualizes SoC | LED colour gradient | None | High |
| Electric Vehicle | LED strip visualizes demand | LED blinking and colour | None | High |
| House (Load) | Same as EV | LED blinking and colour | None | High |
| Heat Pump | Fan toggled on/off | Fan rotation | None | High |
| Fossil Generator | Ultrasonic mister toggled on/off | Visible fume | None | Medium |
| Solar Panel | Led strip visualizes generated energy<br>Photoresistor sensor | LED colour gradient | Yes | High |
| Wind Turbine | Motor-driven blades<br>Hall-effect sensor | Blade rotation | Yes | Medium |


**Figure 5.1:** Complete physical model of the fossil fuel.

**Figure 5.2:** Complete physical model of the wind turbine.

**Figure 5.3:** Complete physical model of the heat pump.

**(a)** Front side of the complete physical model of the solar panel.

**(b)** Back side of the complete physical model of the solar panel.

**Figure 5.4:** Complete physical model of the solar panel.

All prototypes STL and Python files are publicly available through the following links:

- STL Files: https://drive.google.com/drive/folders/1VaiVhr4sBUBDyJrtzEl0uZoCbZxp6wHW?usp=sharing
- Python Scripts: https://github.com/Kyr-updog/Illuminator-BAP-EE-2025/tree/main

Despite the models being functional and adhering to the PoR, there are several areas of improvements:

- The wind turbine blades tend to detach after few rotations. Thus, the adapter connecting the motor to the blades could be improved.
- Model compactness could be improved, particularly the internal organization of the components and cable routing. Once the design has been validated, the components may be mounted on a printed circuit board (PCB) for improved integration and reliability.
- The NeoPixel LED strips demand higher currents, $48mA$ per LED. To avoid unstable operations, a buck converter (e.g., LM2596TVADJG) is recommended.

## 5.2. Dashboard

The Grafana dashboards developed for the Illuminator demonstrator is evaluated in terms of clarity and usefulness for four different groups: educators, policymakers, researchers, and utility companies. Each dashboard is connected to simulation data via InfluxDB and provides visual insights based on the needs and technical levels of its audience. Tutorial 1 from the Illuminator software is used to make the example dashboards, where the fossil fuel model and nuclear model from group A.3 [7] were added.

### 5.2.1. Education

The educational dashboard is created to communicate crucial aspects of the energy system in a way that is accessible to students and the general public. The dashboard connects to the simulation data and shows this in a compact set of intuitive visual elements.

The dashboard in figure 5.5 includes a combination of gauges, time series plots, and statistical panels to present real-time information on renewable energy production, battery status, consumption levels, and energy demand. Each visualization is designed to show a specific message requiring little interpretive effort from the viewer, using colour and layout for interpretation.

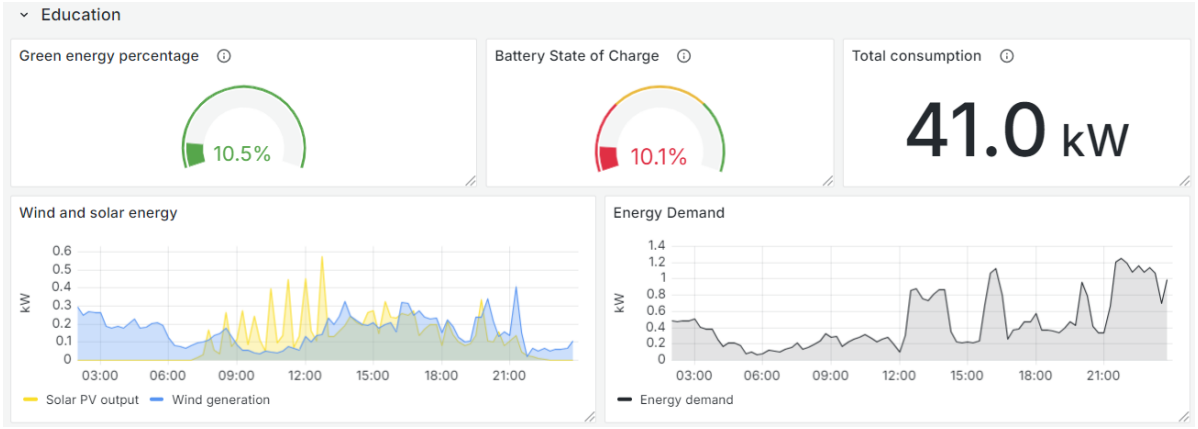



**Figure 5.5:** Education dashboard

The top left gauge shows the share of energy consumption currently being met by renewable sources (solar PV and wind). This allows users to quickly see how much of the system's demand is supplied by green energy. All gauges show the last provides value, so when the simulation is running the value will keep adapting.

The battery state of charge is displayed by a circular gauge with a colour scale (red to green) to show the urgency or sufficiency of the energy stored.

Generation from both solar and wind sources are plotted. This time series visualization shows the time related variability of renewables, with solar generation rising and falling with daylight hours, and wind showing more irregular patterns.

Energy demand is plotted, allowing users to observe changes and identify peak usage periods.

Each indicator includes an information icon next to its title to support users who may be unfamiliar with the terms used. When clicked, the icon reveals a short explanation that clarifies the meaning of the metric displayed. This makes sure that

### 5.2.2. Policymakers

The policymakers dashboard was created to provide data for decision makers overseeing energy and climate policy. It shows simplified technical details that translate the simulation data into clear and accessible visuals that show trends and monitor progress toward sustainability goals.

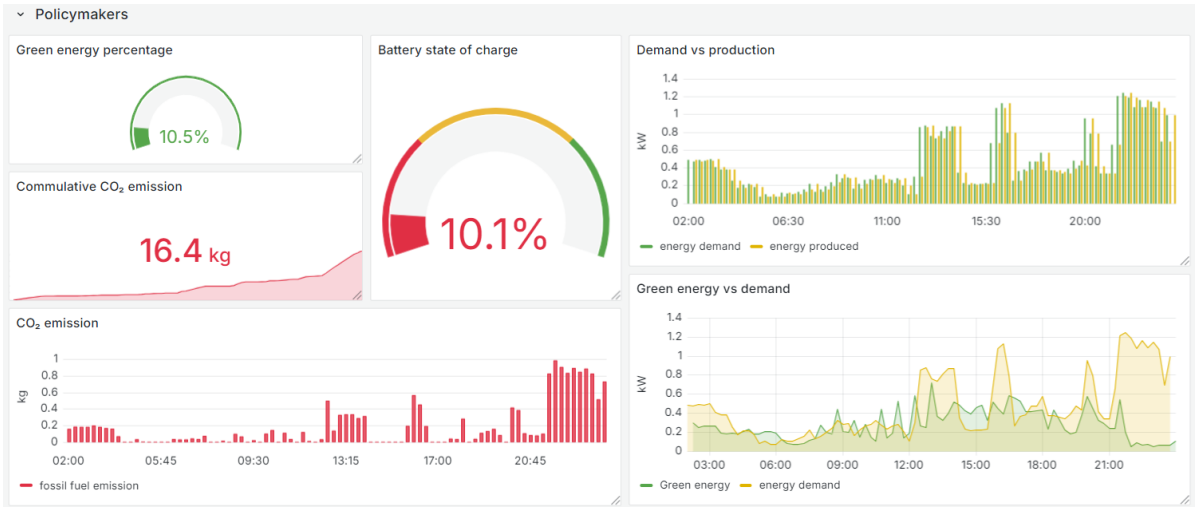



**Figure 5.6:** Policymakers dashboard

The dashboard shown in figure 5.6 visualizes the total energy demand and generation over time. This comparison shows whether the energy system is in balance or if there are mismatches, which are

relevant when considering the stability and sufficiency of energy supply. Policymakers can use this information to estimate whether additional generation or other resources are needed for future demand reliability.

The dashboard also includes the green energy percentage and battery state of charge like the educational dashboard.

$CO_2$ emissions are displayed using both a cumulative counter and a time series graph, showing the environmental impact of fossil based generation. Making this information visible supports decisions about decarbonization targets.

Another time series graph illustrates the production of green energy with the total energy demand. This visualization helps to evaluate the level to which renewables meet consumption needs.

### 5.2.3. Utility companies

The utility companies Dashboard is designed to support operational awareness and system performance monitoring for grid operators and energy providers. It provides a detailed, technically rich view of the energy system, helping users manage production, tracks demand and assess emissions in a single integrated interface.

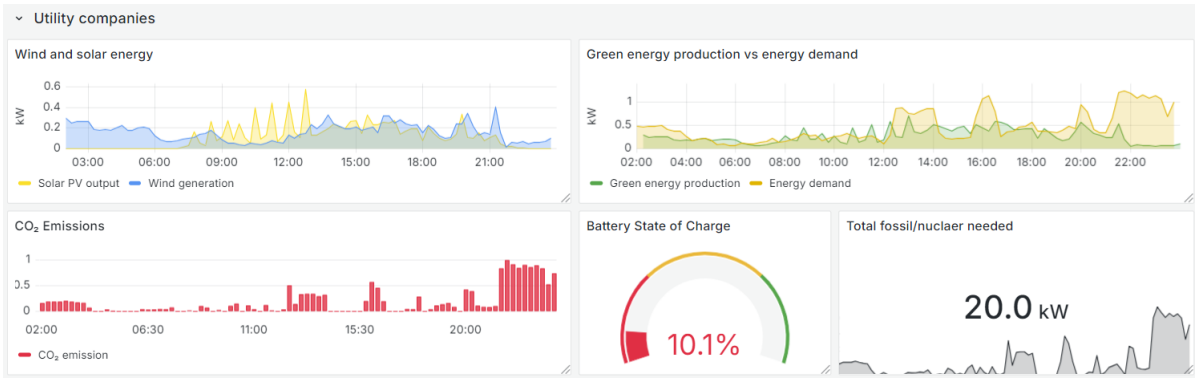



**Figure 5.7:** Utility companies dashboard

The dashboard in figure figure 5.7 shows a time series comparison of wind and solar PV output which enables operators to track intermittent renewable generation. This combined view helps assess the reliability and integration patterns of green energy sources.

The "Green energy production vs energy demand" panel provides a direct comparison between renewable supply and total demand. This visualizes periods where green energy could not meet load requirements, helping utilities quantify the renewable gap and determine the extent of the required fossil fuel or nuclear compensation.

$CO_2$ emissions are shown in a time-series bar graph. The "Total fossil/nuclear needed" indicator summarizes the day's requirement at 20.0 kW, offering a straightforward measure of carbon-intensive backup generation.

### 5.2.4. Researchers

The Researcher Dashboard provides in depth access to the simulation output, supporting the analysis, validation, and comparison of different energy system configurations.

The dashboard is built for flexibility. Using Grafana's interactive tools, researchers can freely configure panels, apply filters, and execute custom queries to display exactly the metrics they need. The example layout is shown in figure 5.8. It shows a collection of six time series panels for variables such as battery SoC, power output, load demand, green energy generation and $CO_2$. However, researchers are not limited to this configuration. They can easily add, remove, or reformat panels to suit different scenarios or analysis objectives.

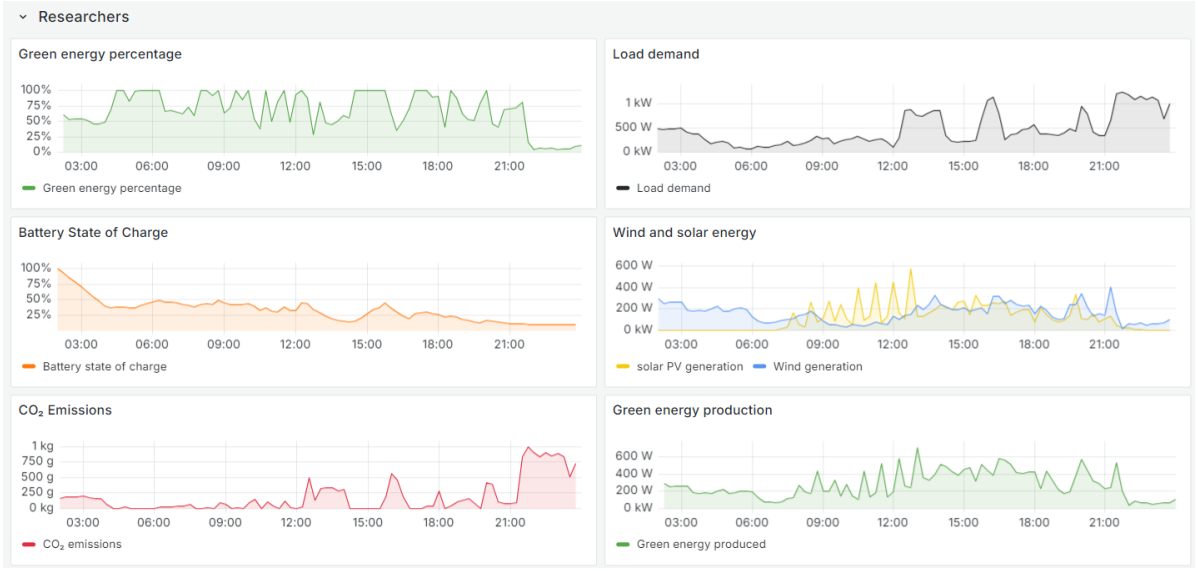



**Figure 5.8:** Researchers dashboard

# 6

# Conclusion & Discussion

## 6.1. Project results

This project successfully achieved the KPIs and specifications outlined in the PoR. The Mosaic-based Illuminator architecture now includes a modular 3D physical models that reflect energy system behaviour in real time, combined with audience-specific Grafana dashboards. The dashboard together with the solar panel and wind turbine add an interactive element to the table-top model. Design decisions and implementation strategies emphasize user-friendliness, plug-and-play modularity, engagement and educational value.

## 6.2. Areas for improvement

Limitations and their improvements include:

- The current interactivity implementation is functional but basic. It requires manual input file manipulation and execution of separate scripts. Options for user-driven inputs (e.g. dropdowns, toggles, or sliders to allow real-time scenario tuning) could enhance educational and engagement value.
- The use of BD911 transistor is an overkill for the fan and the ultrasonic mister. More efficient alternatives provides robustness (e.g. 2N2222).
- The ultrasonic mister's connection is unstable and unpredictable. An alternative electrical connection or alternative mister is recommended.
- Due to direct GPIO control within the modified model scripts, the software cannot be run on standard PCs without GPIO interfaces. A bypass or abstraction layer should be developed for cross-platform compatibility.
- GPIO pins are currently assigned manually. This restricts scalability and parallelization. Implementing dynamic pin assignment could enable multiple models to operate concurrently on a single Raspberry Pi, provided their combined current demand remains within safe limits.
- Integration of an HTTP-based control interface to exchange output CSV files, which is technically feasible within Mosaik's flexible simulation framework. This approach could eliminate the need to modify model code directly and would also enable cross-platform compatibility, improved scalability, and more efficient testing workflows.
- Advanced interactivity in Grafana dashboards. Although Grafana effectively visualizes key indicators, its dashboards could be made more interactive. Features such as scenario switching would enhance its value. Particularly for educational or policy-focused presentations.
- While dashboards were tailored for four distinct user groups, additional customization could enhance relevance. A modular Grafana design with user selectable views could address this need.

## 6.3. Future work

Future development of the for the visualization of the model include:

- Implementing dynamic pin configuration at runtime, possibly with a centralized pin manager or specify pin configuration in the YAML file.
- Redesigning the 3D 'Raspberry Pi box' for compactness and better cable management (e.g., using compact PCBs instead of jumper cables).
- Developing a simulation abstraction layer for cross-platform compatibility.
- Exploring user-driven dashboards that allow scenario selection and interactive inputs.
- Improving the mechanical stability of dynamic models, particularly the wind turbine. A sturdier mechanical attachment or alternative motor mechanism could be the solution.
- Adding a 5V buck-boost converter, such as the LM2596TVADJG, to maintain voltage stability and prevent overloads.
- Currently, the functionality for sending data to InfluxDB is implemented within the `collector.py` script. This setup limits modularity and flexibility. It prevents predefined dashboards from being used effectively, as the data must be filtered manually each time. Integration, build into the Illuminator software, could redirect the data automatically to specific dashboards, improving efficiency and usability.

## 6.4. Final remarks

This project demonstrates the feasibility and impact of combining physical models with digital visualization. It makes the challenges of energy transition intuitive to understand through physical-digital visualization and engagement.

By adhering to the PoR, the outcomes presented here are not only functionally but also pedagogically valuable. They also provide a modular foundation for future scaling, refinement, and broader use in both educational and stakeholder-focused contexts.

# A

# 3D Model

## A.1. Implementation

This link brings you to the GitHub repository of the whole project

### A.1.1. Battery

```
#Top-level code
import time as timer
import board
import neopixel

pixels1 = neopixel.NeoPixel(board.D18, 20, brightness=1)

#Code in step() function
        if self.soc < 21:
            pixels1.fill((139, 0, 0))
        elif self.soc < 41:
            pixels1.fill((255, 40, 0))
        elif self.soc < 61:
            pixels1.fill((255, 200, 0))
        elif self.soc < 81:
            pixels1.fill((142, 255, 0))
        else:
            pixels1.fill((0, 255, 0))
        timer.sleep(1)
```

### A.1.2. Load and electric vehicle

**Listing A.1:** Load

```
#Top-level code
import time as timer
import board
import neopixel

pixels1 = neopixel.NeoPixel(board.D18, 9, brightness=1)
previous_dem = 0

#Code in step() function
         #Green is for base unit W or Wh, yellow is for kW or kWh and red is for
             MW or MWhh and above.
        #LEDs blinking indicated load demand is increased, while constant
            illumination indicates constant or decreased load demand.
```

```
        global previous_dem
        if results['load_dem'] > 1000000:
            if results['load_dem'] > previous_dem:
                pixels1.fill((139, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((139, 0, 0))
                timer.sleep(0.4)
            else:
                pixels1.fill((139, 0, 0))
        elif results['load_dem'] > 1000:
            if results['load_dem'] > previous_dem:
                pixels1.fill((255, 200, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((255, 200, 0))
                timer.sleep(0.4)
            else:
                pixels1.fill((255, 200, 0))
        elif results['load_dem'] > 0:
            if results['load_dem'] > previous_dem:
                pixels1.fill((0, 255, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 255, 0))
                timer.sleep(0.4)
            else:
                pixels1.fill((0, 255, 0))
        else:
            pixels1.fill((0, 0, 0))
        previous_dem = results['load_dem']
```

**Listing A.2:** Load$_EV$

```

#Top-level code
import time as timer
import board
import neopixel

pixels1 = neopixel.NeoPixel(board.D18, 7, brightness=0.8)
previous_dem = 0

#Code in step() function
        #Green is for base unit W or Wh, yellow is for kW or kWh and red is for MW or MWhh and above.
        #LEDs blinking indicated load demand is increased, while constant illumination indicates constant or decreased load demand.
        global previous_dem
        if results['load_EV'] > 1000000:
            if results['load_EV'] > previous_dem:
                pixels1.fill((139, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((139, 0, 0))
                timer.sleep(0.4)
            else:
```

```
                pixels1.fill((139, 0, 0))
        elif results['load_EV'] > 1000:
            if results['load_EV'] > previous_dem:
                pixels1.fill((255, 200, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((255, 200, 0))
                timer.sleep(0.4)
            else:
                pixels1.fill((255, 200, 0))
        elif results['load_EV'] > 0:
            if results['load_EV'] > previous_dem:
                pixels1.fill((0, 255, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 0, 0))
                timer.sleep(0.3)
                pixels1.fill((0, 255, 0))
                timer.sleep(0.4)
            else:
                pixels1.fill((0, 255, 0))
        else:
            pixels1.fill((0, 0, 0))
        previous_dem = results['load_EV']
```

### A.1.3. Heat pump

```
        if results['load_HP'] > 0:
            GPIO.output(23, GPIO.HIGH)
        else:
            GPIO.output(23, GPIO.LOW)
        timer.sleep(1)
```

### A.1.4. Fossil fuel

```
        if self.outputs['gen_pow'] > 0:
            GPIO.output(23, GPIO.HIGH)
        else:
            GPIO.output(23, GPIO.LOW)
        timer.sleep(1)
```

### A.1.5. Solar panel

```
        pixels1 = neopixel.NeoPixel(board.D18,1 , brightness=1)

        light = results['pv_gen']
        print (light, self.cap)
        if light <= 0:
           pixels1.fill((0, 0,0))
           print("Light Level:",light)
        elif light < self.cap/5:
           pixels1.fill((139, 0,0))
           print("Light Level:",light)
        elif light < self.cap/5*2:
           pixels1.fill((255, 140,0))
           print("Light Level:",light)
        elif light < self.cap/5*3:
           pixels1.fill((255, 215,0))
           print("Light Level:",light)
```

```
        elif light < self.cap/5*4:
           pixels1.fill((144, 238,144))
           print("Light Level:",light)
        else:
           pixels1.fill((0, 100,0))
           print("Light Level:",light)
        timeeee.sleep(1)
```

### A.1.6. Wind mill

```
        if self.output_type == 'energy':
            motor = results['wind_gen'] / self.resolution_h
        else:
            motor = results['wind_gen']
        print (motor, self.p_rated)

        delay_time = (0.0007 - motor/self.p_rated*0.0007 )  + 0.001  # 1
            millisecond  #0.0017
        print("Light Level:", motor, delay_time, self.p_rated)


        # Driving the coils of the motor
        def step1():
            D.on()
            sleep(delay_time)
            D.off()

        def step2():
            D.on()
            C.on()
            sleep(delay_time)
            D.off()
            C.off()

        def step3():
            C.on()
            sleep(delay_time)
            C.off()

        def step4():
            B.on()
            C.on()
            sleep(delay_time)
            B.off()
            C.off()

        def step5():
            B.on()
            sleep(delay_time)
            B.off()

        def step6():
            A.on()
            B.on()
            sleep(delay_time)
            A.off()
            B.off()

        def step7():
            A.on()
            sleep(delay_time)
```

```
            A.off()

        def step8():
            D.on()
            A.on()
            sleep(delay_time)
            D.off()
            A.off()

        # Perform one fourth of a rotation
        for _ in range(128):
            step1()
            step2()
            step3()
            step4()
            step5()
            step6()
            step7()
            step8()
```

### A.1.7. Interactivity

**Listing A.3:** Interactive code of solar panel

```


####### Interactivity ######

import signal

import spidev

from collections import deque

from pvlib import solarposition
from pvlib.location import Location
import pandas as pd
from datetime import datetime, date
import time as timee


import requests

import subprocess




API_KEY = "456db28b094c717c8b7e5a3e0d4fb4d0"
LAT = 52.0116    # Delft latitude
LON = 4.3571     # Delft longitude


spi = spidev.SpiDev()
spi.open(0,0)
spi.max_speed_hz=1350000

location = (LAT, LON)  # Delft (lat, lon)

# Create location object
```

```
site = Location(LAT, LON)


data_time = deque(maxlen=10)
data_G_Gh = deque(maxlen=10)
data_G_Dh = deque(maxlen=10)
data_G_Bn = deque(maxlen=10)
data_Ta = deque(maxlen=10)
data_hs = deque(maxlen=10)
data_FF = deque(maxlen=10)
data_Az = deque(maxlen=10)


#interrupt function
def handle_sigquit(signum, frame):

    new_rows = []
    new_row_load = []
    new_row_wind = []

    #specify the input .txt file that is not the interactive model but is in the .
        yaml file.

    #model 1
    df = pd.read_csv('/home/Raspinator/Illuminator/examples/Tutorial1/data/
        winddata_NL.txt', skiprows=1, parse_dates=['time'])
    df.set_index('time', inplace=True)


    # Extract only .txt file time data
    txt_date = datetime.strptime('2012-06-01 00:00:00', '%Y-%m-%d %H:%M:%S')  #
        time where data should be extrapolated from
    df_day = df.loc[str(txt_date.date())].reset_index()

    # Combine today's date with original time
    new_date = datetime.now().date()

    df_day['time'] = [datetime.combine(new_date, t.time()) for t in df_day['time'
        ]]
    df_day.set_index('time',inplace=True)
    wind_df_day = df_day


    #model 2
    df = pd.read_csv('/home/Raspinator/Illuminator/examples/Tutorial1/data/
        load_data.txt', skiprows=1, parse_dates=['time'])
    df.set_index('time', inplace=True)



    txt_date = datetime.strptime('2012-06-01 00:00:00', '%Y-%m-%d %H:%M:%S')  #
        time where data should be extrapolated from
    df_day = df.loc[str(txt_date.date())].reset_index()


    new_date = datetime.now().date()

    df_day['time'] = [datetime.combine(new_date, t.time()) for t in df_day['time'
        ]]
    df_day.set_index('time',inplace=True)

```

```
92
93
94
95
96      #Arange the rows that should be added to all input .txt file.
97      for i in range(len(list(data_Az))):
98          row = f"{list(data_time)[i]},{list(data_G_Gh)[i]},{list(data_G_Dh)[i]},{
                list(data_G_Bn)[i]},{list(data_Ta)[i]},{list(data_hs)[i]},{list(data_FF
                )[i]},{list(data_Az)[i]}"
99          new_rows.append(row)
100
101         # Find closest row for interpolation
102         target_time = pd.to_datetime(list(data_time)[i])
103
104         wind_df_interp = wind_df_day.resample('1T').interpolate('time')
105         row = wind_df_interp.reindex([target_time], method='nearest', tolerance=pd
                .Timedelta('14m'))
106         new_row_wind.append(row)
107
108
109
110         df_interp = df_day.resample('1T').interpolate('time')
111         row = df_interp.reindex([target_time], method='nearest', tolerance=pd.
                Timedelta('14m'))
112         new_row_load.append(row)
113
114
115     #append the iterpolated and interactive model data to the input .txt file.
116     with open('/home/Raspinator/Illuminator/examples/Tutorial1/data/
            pv_data_Rotterdam_NL-15min.txt', "a") as file:
117         for row in new_rows:
118             file.write(row + "\n" )
119
120
121     with open('/home/Raspinator/Illuminator/examples/Tutorial1/data/load_data.txt'
            , "a") as file:
122         for row in new_row_load:
123             file.write(row.to_csv(header=False, index=True).strip() + "\n" )
124
125     with open('/home/Raspinator/Illuminator/examples/Tutorial1/data/winddata_NL.
            txt', "a") as file:
126         for row in new_row_wind:
127             file.write(row.to_csv(header=False, index=True).strip()   +  "\n")
128
129     # file must be send to master and wind turbine slave
130     subprocess.run([
131         "scp",
132         "/path/to/local/file",                              #Fill in local file
                path
133         "Raspinator@remote_ip:/path/to/remote/destination"  #fill in remote
                username, ipadress and file destination
134     ])
135
136
137     code = '''
138     #run the simulation
139     CONFIG_FILE = '/home/Raspinator/Illuminator/examples/Tutorial1/
            Tutorial_Power_Balance_a.yaml'
140     simulation_RES = Simulation(CONFIG_FILE)
141     simulation_RES.set_model_param(model_name='CSVload', parameter='file_path',
            value='/home/Raspinator/Illuminator/examples/Tutorial1/data/load_data.txt')
```

```
    simulation_RES.set_model_param(model_name='CSV_pv', parameter='file_path',
        value='/home/Raspinator/Illuminator/examples/Tutorial1/data/
        pv_data_Rotterdam_NL-15min.txt')
    simulation_RES.set_model_param(model_name='CSV_wind', parameter='file_path',
        value='/home/Raspinator/Illuminator/examples/Tutorial1/data/winddata_NL.txt
        ')

    new_settings = {'Wind1': {'p_rated': 0.3}, # power in kW
                    'Load1': {'houses': 5}, # number of houses
                    'PV1':{'cap': 500} # installed capacity in W
                    }

    simulation_RES.edit_models(new_settings)
    print(list(data_time)[0], list(data_time)[-1])
    simulation_RES.set_scenario_param('start_time', str(list(data_time)[0]))
    simulation_RES.set_scenario_param('end_time', str(list(data_time)[-1]))
    simulation_RES.set_scenario_param('time_resolution', 1)

    # run the simulation
    simulation_RES.run()
    '''
    with open('simulation_script.py', 'w') as f:
        f.write(f"replace str(list(data_time)[0]) with {str(list(data_time)[0])}")
        f.write(f" replace str(list(data_time)[0]) with {str(list(data_time)[-1])}
            ")
        f.write(code)

    #send only to master.
    subprocess.run([
        "scp",
        "/path/to/local/file",                              #Fill in local file
            path
        "Raspinator@remote_ip:/path/to/remote/destination"  #fill in remote
            username, ipadress and file destination
    ])


#create interrupt function
signal.signal(signal.SIGQUIT, handle_sigquit)

#read photoresistor
def read_channel(channel):
    adc=spi.xfer2([1,  (8+channel) << 4 ,0 ])
    data = ((adc[1]&3) <<8 ) +adc[2]
    return data




#take the last 10samples of the solar pannel
print("Running. Press Ctrl+\\ (backslash) to send SIGQUIT.")
while True:
    print(list(data_G_Gh))
    url = f"https://api.openweathermap.org/data/2.5/weather?lat={LAT}&lon={LON}&
        appid={API_KEY}&units=metric"
    response = requests.get(url)
    now = datetime.now()
    if not list(data_time):
        data_time.append(datetime.now().replace(microsecond=0))
        data_G_Gh.append(read_channel(0)/1023*900)
        data_G_Dh.append(read_channel(0)/1023*900)
```

```
195         data_G_Bn.append(0)
196         data_Ta.append(response.json()['main']['temp'])
197         data_hs.append(solarposition.get_solarposition(pd.Timestamp('now', tz='
                Europe/Amsterdam'), location[0], location[1])['elevation'].iloc[0])
198         data_FF.append(response.json()['wind']['speed'])
199         data_Az.append(solarposition.get_solarposition(pd.Timestamp('now', tz='
                Europe/Amsterdam'), location[0], location[1])['azimuth'].iloc[0])
200         timee.sleep(0.40)
201     elif now.replace(microsecond=0) != list(data_time)[-1]:
202         data_time.append(datetime.now().replace(microsecond=0))
203         data_G_Gh.append(read_channel(0)/1023*900)
204         data_G_Dh.append(read_channel(0)/1023*900)
205         data_G_Bn.append(0)
206         data_Ta.append(response.json()['main']['temp'])
207         data_hs.append(solarposition.get_solarposition(pd.Timestamp('now', tz='
                Europe/Amsterdam'), location[0], location[1])['elevation'].iloc[0])
208         data_FF.append(response.json()['wind']['speed'])
209         data_Az.append(solarposition.get_solarposition(pd.Timestamp('now', tz='
                Europe/Amsterdam'), location[0], location[1])['azimuth'].iloc[0])
210         timee.sleep(0.40)
211     else:
212         dummy = 0
```

**Listing A.4:** Interactive code of wind turbine

```
1
2
3 ###### Interactivity
      ################################################################
4
5
6 from gpiozero import Button
7 from datetime import datetime, date
8 import time as timee
9
10 import signal
11 from collections import deque
12 import subprocess
13
14 import pandas as pd
15
16
17 data_time = deque(maxlen=10)
18 data_u = deque(maxlen=10)
19
20 sensor = Button(24, pull_up=True)
21
22 #interrupt function
23 def handle_sigquit(signum, frame):
24
25     new_rows = []
26     new_row_load = []
27     new_row_sun = []
28
29     #specify the input .txt file that is not the interactive model but is in the .
           yaml file.
30
31     #model 1
32     df = pd.read_csv('/home/Raspinator/Illuminator/examples/Tutorial1/data/
           pv_data_Rotterdam_NL-15min.txt', skiprows=1, parse_dates=['time'])
33     df.set_index('time', inplace=True)
```

```
34
35
36      # Extract only .txt file time data
37      txt_date = datetime.strptime('2012-06-01 00:00:00', '%Y-%m-%d %H:%M:%S')
38      df_day = df.loc[str(txt_date.date())].reset_index()
39
40      # Combine today's date with original time
41      new_date = datetime.now().date()
42
43      df_day['time'] = [datetime.combine(new_date, t.time()) for t in df_day['time'
            ]]
44      df_day.set_index('time',inplace=True)
45      sun_df_day = df_day
46
47
48      #model 2
49      df = pd.read_csv('/home/Raspinator/Illuminator/examples/Tutorial1/data/
            load_data.txt', skiprows=1, parse_dates=['time'])
50      df.set_index('time', inplace=True)
51
52
53
54      txt_date = datetime.strptime('2012-06-01 00:00:00', '%Y-%m-%d %H:%M:%S')
55      df_day = df.loc[str(txt_date.date())].reset_index()
56
57
58      new_date = datetime.now().date()
59
60      df_day['time'] = [datetime.combine(new_date, t.time()) for t in df_day['time'
            ]]
61      df_day.set_index('time',inplace=True)
62
63
64
65
66
67      #Arange the rows that should be added to all input .txt file.
68      diffs = [t2 - t1 for t1, t2 in zip(data_time, data_time[1:])]
69      for i in range(len(diffs)):
70          row = f"{list(data_time)[i]},{diffs[i]}"
71          new_rows.append(row)
72
73          # Find closest row
74          target_time = pd.to_datetime(list(data_time)[i])
75
76          sun_df_interp = sun_df_day.resample('1T').interpolate('time')
77          row = sun_df_interp.reindex([target_time], method='nearest', tolerance=pd.
                Timedelta('14m'))
78          new_row_sun.append(row)
79
80
81
82          df_interp = df_day.resample('1T').interpolate('time')
83          row = df_interp.reindex([target_time], method='nearest', tolerance=pd.
                Timedelta('14m'))
84          new_row_load.append(row)
85
86
87      #append the iterpolated and interactive model data to the input .txt file.
88      with open('/home/Raspinator/Illuminator/examples/Tutorial1/data/winddata_NL.
            txt ', "a") as file:
```

```
89          for row in new_rows:
90              file.write(row + "\n" )
91
92
93      with open('/home/Raspinator/Illuminator/examples/Tutorial1/data/load_data.txt'
            , "a") as file:
94          for row in new_row_load:
95              file.write(row.to_csv(header=False, index=True).strip() + "\n" )
96
97      with open('/home/Raspinator/Illuminator/examples/Tutorial1/data/
            pv_data_Rotterdam_NL-15min.txt', "a") as file:
98          for row in new_row_sun:
99              file.write(row.to_csv(header=False, index=True).strip()   +  "\n")
100
101     # file must be send to master and solar panel slave
102     subprocess.run([
103         "scp",
104         "/path/to/local/file",                              #Fill in local file
                path
105         "Raspinator@remote_ip:/path/to/remote/destination"  #fill in remote
                username, ipadress and file destination
106     ])
107
108
109
110     code = '''
111     #run the simulation
112     CONFIG_FILE = '/home/Raspinator/Illuminator/examples/Tutorial1/
            Tutorial_Power_Balance_a.yaml'
113     simulation_RES = Simulation(CONFIG_FILE)
114     simulation_RES.set_model_param(model_name='CSVload', parameter='file_path',
            value='/home/Raspinator/Illuminator/examples/Tutorial1/data/load_data.txt')
115     simulation_RES.set_model_param(model_name='CSV_pv', parameter='file_path',
            value='/home/Raspinator/Illuminator/examples/Tutorial1/data/
            pv_data_Rotterdam_NL-15min.txt')
116     simulation_RES.set_model_param(model_name='CSV_wind', parameter='file_path',
            value='/home/Raspinator/Illuminator/examples/Tutorial1/data/winddata_NL.txt
            ')
117
118     new_settings = {'Wind1': {'p_rated': 0.3}, # power in kW
119                     'Load1': {'houses': 5}, # number of houses
120                     'PV1':{'cap': 500} # installed capacity in W
121                     }
122
123     simulation_RES.edit_models(new_settings)
124     print(list(data_time)[0], list(data_time)[-1])
125     simulation_RES.set_scenario_param('start_time', str(list(data_time)[0]))
126     simulation_RES.set_scenario_param('end_time', str(list(data_time)[-1]))
127     simulation_RES.set_scenario_param('time_resolution', 1)
128
129     # run the simulation
130     simulation_RES.run()
131     '''
132     with open('simulation_script.py', 'w') as f:
133         f.write(f"replace str(list(data_time)[0]) with {str(list(data_time)[0])}")
134         f.write(f" replace str(list(data_time)[0]) with {str(list(data_time)[-1])}
                ")
135         f.write(code)
136
137     #send only to master.
138     subprocess.run([
```

```
        "scp",
        "/path/to/local/file",                              #Fill in local file
            path
        "Raspinator@remote_ip:/path/to/remote/destination"  #fill in remote
            username, ipadress and file destination
    ])



#create interrupt function
signal.signal(signal.SIGQUIT, handle_sigquit)




#take the last 10samples of the wind_mill
# This function is executed when a signal is detected (falling edge).
def ausgabeFunktion():
    print(list(data_u))
    now = datetime.now()
    if not list(data_time):
        data_time.append(now.replace(microsecond=0))
        timee.sleep(0.40)
    elif now.replace(microsecond=0) != list(data_time)[-1]:
        data_time.append(now.replace(microsecond=0))
        timee.sleep(0.40)
    else:
        dummy = 0

# The 'outputFunction' function is bound to the 'when_pressed' event of the sensor
    .
sensor.when_pressed = ausgabeFunktion
```

# B

# Dashboard

### B.0.1. Changes engine.py

```
    collector = world.start('Collector',
    time_resolution=_time_resolution,
    start_date=_start_time,
    results_show={
        'write2csv': True,
        'dashboard_show': True,  #     changed from False
        'Finalresults_show': False,
        'database': False,
        'mqtt': False
    },
    output_file=_results_file)
```

### B.0.2. Dashboard influxDB connection

```
     # Add influxDB info
            token = "Lfw7MDESxhk3NoeK3a8_bygrZB3U-2gHc6Vr-
                CxSHbyT8XjcZL_aq_SMHoEKdWJssZgXlrG4vBEqxXHMQtap-w=="
            org = "Illuminator"
            bucket = "Illuminator"
            url = "http://localhost:8086"

            #Connect to InfluxDB
            client = InfluxDBClient(url=url, token=token, org=org)
            write_api = client.write_api(write_options=SYNCHRONOUS)
```

### B.0.3. Changes collector.py

```
    for index, row in df.iterrows():
    point = Point("energy_metrics").tag("source", "illuminator")
    for col, val in row.items():
        point = point.field(col, float(val))
    point = point.time(index, WritePrecision.NS)
    write_api.write(bucket=bucket, org=org, record=point)
```

# Bibliography


[1] T. Rist and M. Masoodian, "Promoting sustainable energy consumption behavior through interactive data visualizations," *Multimodal Technologies and Interaction*, vol. 3, no. 3, p. 56, 2019. DOI: 10.3390/mti3030056. [Online]. Available: https://www.mdpi.com/2414-4088/3/3/56.

[2] K. L. Vavra, V. Janjic-Watrich, K. Loerke, L. M. Phillips, S. P. Norris, and J. Macnab, "Visualization in science education," *Alberta Science Education Journal*, vol. 41, no. 1, pp. 22–30, 2011. [Online]. Available: https://sc.teachers.ab.ca/SiteCollectionDocuments/Vol.%2041,%20No.%201%20January%202011.pdf#page=24https://sc.teachers.ab.ca/SiteCollectionDocuments/Vol.%2041,%20No.%201%20January%202011.pdf#page=24.

[3] J. G. Dr. M. (Milos) Cvetkovic, *Energy system integrator demonstrator*, Bachelor's graduation project, Delft Univeristy of Techology, 2025.

[4] Raspberry Pi Foundation, *Raspberry Pi Documentation: GPIO Pins*, https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#gpio-pins, Accessed: 2025-06-13, 2024.

[5] GeeksforGeeks. "Architecture of raspberry pi." (Nov. 2023), [Online]. Available: https://www.geeksforgeeks.org/architecture-of-raspberry-pi/ (visited on 06/14/2025).

[6] I. Team, *Illuminator models*, Accessed: 2025-06-08, 2025. [Online]. Available: https://illuminator-team.github.io/Illuminator/references/models.html.

[7] A. Zhang and K. Rahimatulla, "The netherlands illuminated: Designing and simulating an intuitive model of the dutch transmission grid," Delft University of Technology, Tech. Rep., 2025, Bachelor's Thesis.

[8] B. Dorland and A. Vermeer, "Embedded system communication for the hardware integration of the illuminator simulation tool," Delft University of Technology, Tech. Rep., 2025, Bachelor's Thesis.

[9] JOY-IT, *Datasheet: SBC-FAN-303010 – Active Fan for Raspberry Pi / Rock Pi / Banana Pi*, Accessed: 2025-06-11, 2025. [Online]. Available: https://asset.conrad.com/media10/add/160267/c1/-/en/001720600DS02/datablad-1720600-joy-it-sbc-fan-303010-actieve-ventilator-geschikt-voor-serie-raspberry-pi-rock-pi-banana-pi-zwart.pdf.

[10] 3DPrintTech Design, *Raspberry Pi Fan Duct for 30mm Fan*, https://www.thingiverse.com/thing:2007394, Accessed: 2025-06-14, 2017.

[11] G. Labs, *Grafana documentation*, Accessed: 2025-04-29, 2025. [Online]. Available: https://grafana.com/docs/grafana/latest/.

[12] InfluxData, *InfluxDB Documentation*, Available: https://docs.influxdata.com/influxdb/latest/, 2024.

[13] N.-R. Contributors, *User guide*, Accessed: 2025-04-29, 2025. [Online]. Available: https://nodered.org/docs/user-guide/.

[14] Worldsemi, *WS2813 Intelligent Control LED Integrated Circuit Datasheet*, Accessed: 2025-06-11, 2017. [Online]. Available: https://radioled.com/pdf/pdf/WS2813.pdf.

[15] Texas Instruments, *SN74HCT125 Quadruple Bus Buffer Gates With 3-State Outputs Datasheet*, Accessed: 2025-06-11, 2003. [Online]. Available: https://www.ti.com/lit/ds/symlink/sn74hct125.pdf.

[16] STMicroelectronics, *Bd911: Npn power transistor datasheet*, https://www.st.com/resource/en/datasheet/bd911.pdf, Accessed: 2025-06-13, 2005.

[17] OTRONIC, *Ultrasonic Mist Maker/Damper/Humidifier Datasheet*, https://www.otronic.nl/en/ultrasonic-mist-maker-damper-humidifier.html, Operating voltage: 5V/300mA, Frequency: 108kHz; accessed June11,2025, 2025.

[18] Joy-IT, *KY-018 Photocell (LDR) Module Datasheet*, Datasheet, Available: https://fabacademy.org/2024/labs/puebla/students/andrea-hortega/assets/images/week11/KY-018-Joy-IT.pdf, Jun. 2017.

[19] Microchip Technology Inc., *MCP3004/MCP3008 2.7V 10-Bit A/D Converters with SPI Interface*, Datasheet, Available: https://cdn-shop.adafruit.com/datasheets/MCP3008.pdf. Accessed: 2025-06-13, Jan. 2008.

[20] Joy-IT, *SBC-Moto1 Raspberry Pi Add-on PCB – User Safety Instructions*, Manual, Available: https://asset.conrad.com/media10/add/160267/c1/-/de/001503742ML01/user-safety-instructions-1503742-joy-it-sbc-moto1-raspberry-pi-add-on-pcb-suitable-for-single-board-pcs-raspberry-pi-raspberry-pi-2-b-raspberry-pi.pdf. Accessed: 2025-06-13, 2024.

[21] Joy-IT, *KY-003 Hall Magnetic Sensor Module Manual*, Manual, Available: https://www.gotronic.fr/pj2-sen-ky003-manual-1932.pdf. Accessed: 2025-06-13, Aug. 2018.

[22] Joy-IT / SIMAC Electronics GmbH, *Com-ky051vt voltage translator / level shifter – datasheet*, Published 03 August 2023, Joy-IT, Neukirchen-Vluyn, Germany, Aug. 2023. [Online]. Available: https://joy-it.net/files/files/Produkte/COM-KY051VT/COM-KY051VT_Datasheet_2023-08-03.pdf.

[23] Joy-IT, *KY-003 Hall Magnetic Sensor Module – Joy-IT Sensor Kit*, Online, Available: https://sensorkit.joy-it.net/en/sensors/ky-003. Accessed: 2025-06-13, n.d.