

# Optimal design of a HESS for peak shaving BEV fast charging demand at a highway rest stop

Semester Thesis

Sean Bone  
bones@ethz.ch

6. April 2022

Electrochemistry Laboratory (LEC)  
Paul Scherrer Institute

**Supervisors:**

Christian Peter (PSI)  
Prof. Dr. T.J. Schmidt (PSI, ETH)



---

## **Abstract**

The focus of this project is the cost-optimal design of Battery and Hydrogen Energy Storage Systems (BESS/HESS) for peak shaving the demand resulting from Battery-Electric Vehicle (BEV) fast-charging at a highway rest stop. To this end, a MATLAB simulation framework is implemented along with Simulink models for the simulation of the Energy Storage Systems in question. Finally, the simulations are evaluated for a variety of scenarios to determine the cost-optimal design of BESS/HESS peak-shaving installations.



---

# Contents

---

<b>Contents</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Reducing operational expenses via peak-shaving . . . . .	2
1.2 Hydrogen Energy Storage System . . . . .	3
1.3 Previous works . . . . .	4
1.4 Goals of this project . . . . .	4
<b>2 Framework architecture design</b>	<b>7</b>
2.1 Design definition . . . . .	7
2.2 Framework overview . . . . .	8
2.3 Example usage . . . . .	9
<b>3 Modelling approach</b>	<b>11</b>
3.1 Peak-shaving algorithm . . . . .	12
3.2 Estimation of minimum required storage capacity . . . . .	14
3.2.1 Proof of correctness . . . . .	15
3.3 Ideal ESS . . . . .	18
3.4 BESS physical simulation . . . . .	19
3.4.1 Simple BESS model . . . . .	19
3.4.2 BESS with improved efficiency modelling . . . . .	20
3.5 HESS physical simulation . . . . .	24
3.5.1 Simple HESS model . . . . .	24
3.5.2 HESS with improved efficiency modelling . . . . .	25
3.6 Calculation of CAPEX . . . . .	26
3.7 Calculation of OPEX . . . . .	26
<b>4 Discussion of results</b>	<b>29</b>
4.1 Methodology . . . . .	29
4.2 Scenario 1: power data from all of 2019 . . . . .	31

4.3	Scenario 2: most energy-intensive week of 2019 . . . . .	34
4.4	Scenario 3: most energy-intensive week of 2019 with 30% BEV share . .	36
<b>5</b>	<b>Summary and conclusions</b>	<b>39</b>
5.1	Outlook . . . . .	39
<b>A</b>	<b>User Guide to the ESPS framework</b>	<b>41</b>
A.1	Directory and package structure . . . . .	41
A.1.1	MATLAB packages . . . . .	41
A.1.2	I/O file format . . . . .	42
A.2	ESPS models . . . . .	44
A.2.1	Creating a new model . . . . .	44
A.3	ESPS simulations . . . . .	46
A.3.1	How models look for inputs . . . . .	46
A.3.2	Model input/output mapping . . . . .	46
A.3.3	Creating a new simulation . . . . .	47
A.4	Running simulations with ESPS . . . . .	48
A.4.1	<code>SequentialSimulation</code> . . . . .	48
A.4.2	Custom simulations . . . . .	49
A.4.3	Running grid searches . . . . .	51
	<b>Bibliography</b>	<b>53</b>

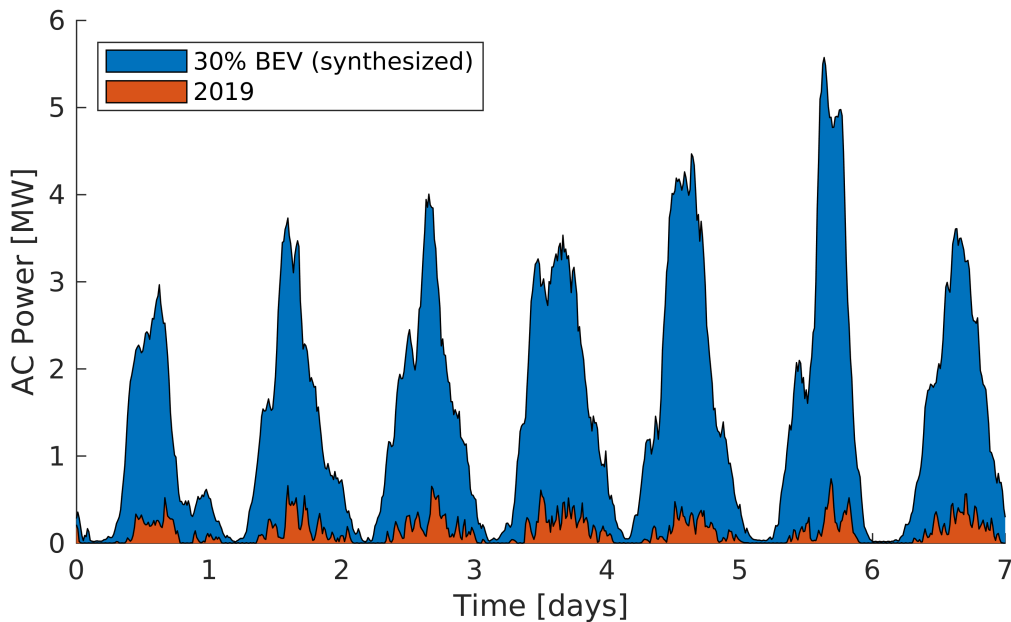
---

# Introduction

---

In this report we will consider the scenario of a highway rest stop equipped with fast-charging stations for battery-electric vehicles (BEV). A single BEV can draw up to 250kW when fast-charging: it is easy to imagine how having multiple fast-charging stations available could quickly lead to very high peak loads, especially since customers tend to cluster in the afternoon and evening time slots, rather than being distributed evenly over the 24 hour cycle.

Figure 1.1 illustrates such a scenario at a highway rest stop in Switzerland; the most intensive week of 2019 already tested the operational limits of the local transformer



**Figure 1.1:** Power draw at a highway rest stop in Switzerland during the most energy-intensive week of 2019 (orange) and extrapolated to a potential future scenario with a 30% BEV traffic share (blue) [1].

station, and as the share of electric vehicles increases, higher capacities will most likely be required.

The primary goal of the peak-shaving techniques explored in this report is to reduce the peak loads of the given demand profiles in favour of a more uniform power consumption. From the point of view of the utility companies, this reduces the strain on the electricity distribution and generation infrastructure, making more efficient use of the current capabilities and potentially improving the stability of supply. From the perspective of the highway rest-stop operator, this reduces operational expenses in the form of electricity bills.

### 1.1 Reducing operational expenses via peak-shaving

For the purposes of this work, the term “Operational EXpenses” (OPEX) refers to the electricity bills experienced by the highway rest stop in question. These are subdivided into multiple components, two of which we will take into account for this project:

**Energy charges** are calculated in terms of CHF/kWh, and tax the total amount of energy consumed by the station. This corresponds to the area below the power graph 1.1.

**Demand charges** are calculated in terms of CHF/kW/month, and tax the peak power draw (averaged in 15-minute time windows) reached over the course of the month. This corresponds to the highest peak of the power graph 1.1.

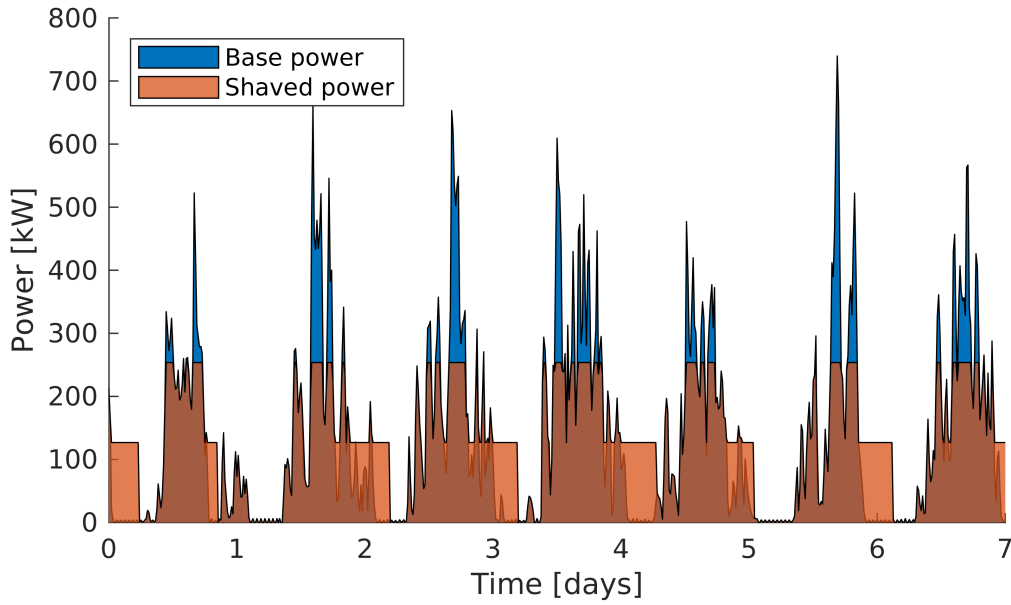
Demand charges are introduced by utility companies to incentivise consumers to better distribute their power consumption, since, as previously mentioned, lower peak loads lead to lower requirements on distribution and production infrastructure.

Figure 1.2 compares two power profiles with nearly identical total energy consumption, but differing OPEX. The peak-shaved power profile (orange overlay) has significantly lower demand charges, since its peak is lower than the baseline (shaded blue). It makes up for the lower consumption at peak hours by consuming more power overnight.

Such peak-shaving can be achieved by installing an Energy Storage System (ESS) on-site. This can be pre-charged when demand is low, and used to offset some of the load when demand is high, effectively lowering the peak power demand. The larger the storage and power capacities of the ESS, the lower the demand peaks can be shaved.

Any such ESS will naturally require an up-front investment which for the purposes of this work makes up the “CAPital EXpenses” (CAPEX). This naturally introduces a trade-off: the larger the ESS capacities, the more we could potentially reduce our expected OPEX over the lifespan of the installation, but we must make a larger up-front investment (CAPEX). The cost-optimal CAPEX/OPEX trade-off depends on several factors such as efficiency and physical dynamics of the ESS in question, cost of compo-





**Figure 1.2:** Comparison of a peak-shaved power profile versus baseline. The base power profile (shaded blue) is the most energy-intensive week of 2019. The peak-shaved profile (shaded orange) has the same total energy consumption, but lower demand charges.

nents for the ESS, electricity costs, power demand profiles we expect, degradation of the ESS infrastructure and more.

These trade-offs are the main focus of this project. In particular, we will be considering two ESS options: a Battery ESS (BESS) and a Hydrogen ESS (HESS).

## 1.2 Hydrogen Energy Storage System

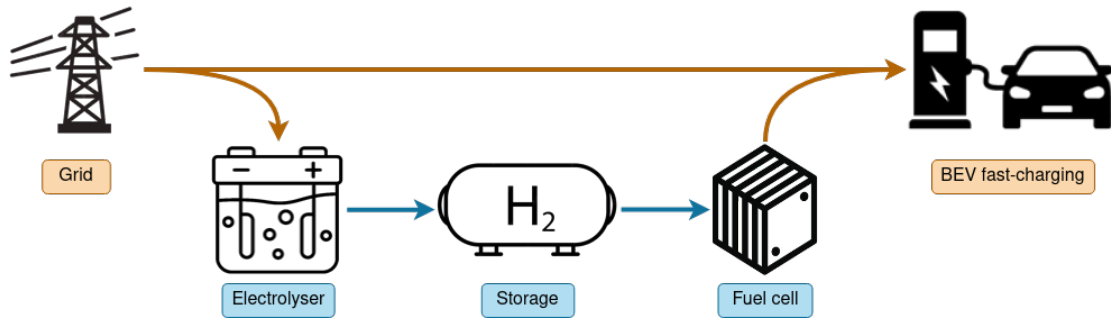
While the concept of a battery storage system is fairly intuitive, the same may not be true about a Hydrogen-based system. Figure 1.3 illustrates the fundamental components of any HESS:

**Electrolyser:** during charging phases, an electrolyser is used to separate water into hydrogen and oxygen gases.

**H<sub>2</sub> tank:** the hydrogen (and possibly oxygen) gas is subsequently stored in a pressurised tank.

**Fuel Cell:** finally, during discharging phases the H<sub>2</sub> is recombined with oxygen (either from storage or from the atmosphere) via a Proton Exchange Membrane (PEM) fuel cell, producing electricity.

A HESS has a lower round-trip efficiency than a battery-based system; however, it can scale to much larger storage capacities since the storage medium (H<sub>2</sub> tank) is significantly cheaper per kWh of installed capacity when compared to battery banks.



**Figure 1.3:** Overview of the components of a Hydrogen Energy Storage System (HESS). Icons from stock.adobe.com under non-commercial license.

The comparison of BESS and HESS systems in different use cases is another focal point of this project.

### 1.3 Previous works

The topic of this project has been tackled in several previous Semester Projects at the Paul Scherrer Institute. R. Lehner focused on identifying the bottlenecks in the utility infrastructure, developed synthesised demand profiles, and implemented basic Simulink models to simulate both battery and hydrogen storage systems [2]. S. Renggli focused on the synthesis of demand profiles based on real-world data and extrapolation into future scenarios [3]. Subsequently, M. Heer approached the question of optimal design of a BESS/HESS system by formulating mathematical optimisation problems [4].

Each approach had its limitations, however. The code base of the first two projects grew unwieldy, which limited its reusability and the reproducibility of the results. Meanwhile, Heer's optimisation approach [4] was limited by computation times, as the problem formulation inevitably lead to rapidly increasing computational complexity. This prevented the study of longer time horizons. Furthermore, the need to formulate the problem as a mathematical optimisation problem limited the complexity and ease of extension of the physical models.

The present project sets out to improve on the limitations of previous projects while building on the existing knowledge base and resources on the topic.

### 1.4 Goals of this project

Having established the necessary background and context, we are finally in a position to formulate the primary goals of this project:

- Implement a structured code framework, taking into account possible future projects based on the same code base;

- Structured management of inputs and outputs for better reproducibility of results;
- Re-implement the existing models within the new framework;
- Implement some improvements to the existing models, particularly with regards to the BESS physical simulation;
- Allow for larger simulations with longer time horizons while containing computation times;
- Finally, reproduce the results of previous projects by using the new code base verify the results and to propose cost-optimal designs for a BESS and a HESS.



## Chapter 2

---

# Framework architecture design

---

In this chapter we will focus on the design and implementation of the code framework. We will begin by looking at the design requirement definition and then move on to a high-level overview of the final framework. A more detailed user guide to the code can be found in Appendix A.

## 2.1 Design definition

### Design requirements

For a design proposal to be accepted, all requirements must be fulfilled:

1. Simulations must be easily reproducible and re-runnable.
2. Simulation parameters and inputs must be easily modifiable.
3. It must be easy to add new simulations and modify existing ones.
4. Each simulation run must return a "result structure" containing all necessary information to understand how the simulation results were derived.

By "easy" we mean that a user with basic programming knowledge and no familiarity with the software architecture should be able to carry out modifications without disproportionate time investment.

### Design goals

A design proposal should strive to include the following additional design goals, or at least be open to their addition:

1. New simulations should be able to incorporate parts of existing simulations.
2. Simulations should clearly state their required inputs and their outputs.

3. It should be possible to run an automated grid search, i.e. re-run the same simulation multiple times with varying input parameters.
4. Such grid searches should be executable in parallel, i.e. each simulation instance running in a separate thread or process.
5. Post-processing of simulations results (e.g. generating plots, excel files, ...) should be decoupled from actual simulation.
6. The code should be well-documented.

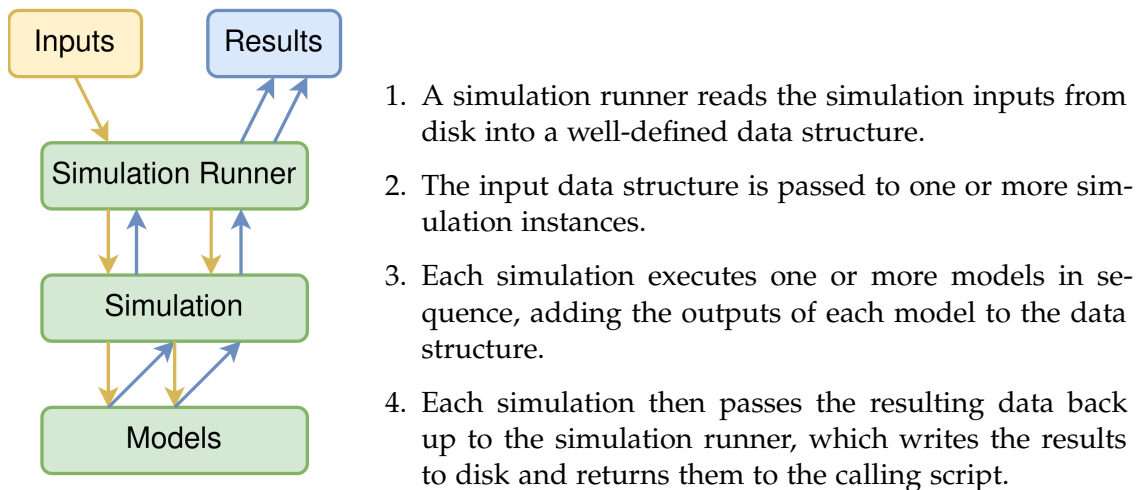
### Language choice

The framework was implemented in MATLAB. The primary reason for this choice is that it allows the use of Simulink models in simulation, which significantly lowers the barrier to entry for future students wanting to build upon the existing framework.

Viable free and open-source alternatives include GNU Octave, Python, Julia, or C++. These could offer improved performance and a better programming environment, however, at time of writing, there is no viable alternative to Simulink, which would force the users to “manually” implement the physical models in code.

## 2.2 Framework overview

Figure 2.1 gives a high-level overview of the implemented framework. The information flow when executing a typical simulation is as follows:



**Figure 2.1:** Overview of information flow in the ESPS framework.

This modular structure maintains a high degree of flexibility: each simulation runner can work with any simulation, and each simulation is agnostic of the inner workings

of the models it calls. This means a developer can implement a new simulation, and then run, for example, a grid-search without needing to re-implement the grid-search functionality.

At the same time, the framework imposes some constraints: each model must declare all of its inputs and outputs, and each simulation must declare the models it calls. This allows for better transparency to the end-user, and allows the framework to handle the input and output data structures.

Finally, the outputs of a simulation run are returned in a well-defined data structure, which includes all of the information necessary to reproduce the results (i.e. all of the inputs, outputs and metadata about the execution) as well as perform post-processing operations such as data visualisation (plotting).

At time of writing, three simulation runners have been implemented: `SimpleSimulationRunner`, which executes the given simulation once, `GridSearchRunner`, which runs multiple instances of the given simulation with varying inputs, and `ParallelGridSearch`, which also runs a grid search, but runs each simulation in a separate MATLAB parallel worker and therefore allows scaling to much larger searches.

## 2.3 Example usage

Listing 2.1 shows a simple example script which executes a parallel grid-search.

```

1 % Create an instance of the chosen simulation
2 sim = esps.simulations.SimEtaHESS();
3
4 % Create an instance of a simulation runner for our simulation
5 runner = esps.runners.ParallelGridSearch(sim);
6
7 % Read our input files
8 runner.readInputs("unifiedOneWeek.jsonc", "inputs/unified/");
9
10 % Sweep 10 values of the shavingAmount parameter from 0.2 to 1
11 runner.addSearchParam("shavingAmount", 0.2, 1, 10);
12
13 % Run the simulation
14 runner.run();
15
16 % Write the outputs to file
17 outDir = runner.writeOutputs();
18
19 % Fetch the results
20 outData = runner.getOutputData();
21
22 % Generate some plots
23 esps.post.GSFinancialPlots(outData, outDir, "HESS");

```

**Listing 2.1:** Example script using the ESPS framework to execute a 1-dimensional grid-search.





## Chapter 3

---

# Modelling approach

---

In this chapter, we will focus on the models used for the simulations and their implementation. Here the term “model” is being used somewhat loosely; by “model” we mean a component of simulation within the code framework, which includes but is not limited to physical simulation models. As a high-level summary, the most important models used in the simulations in this report are the following:

**Peak-shaving algorithm:** this is essentially an open-loop controller for the energy storage system, which decides when the ESS should charge or discharge.

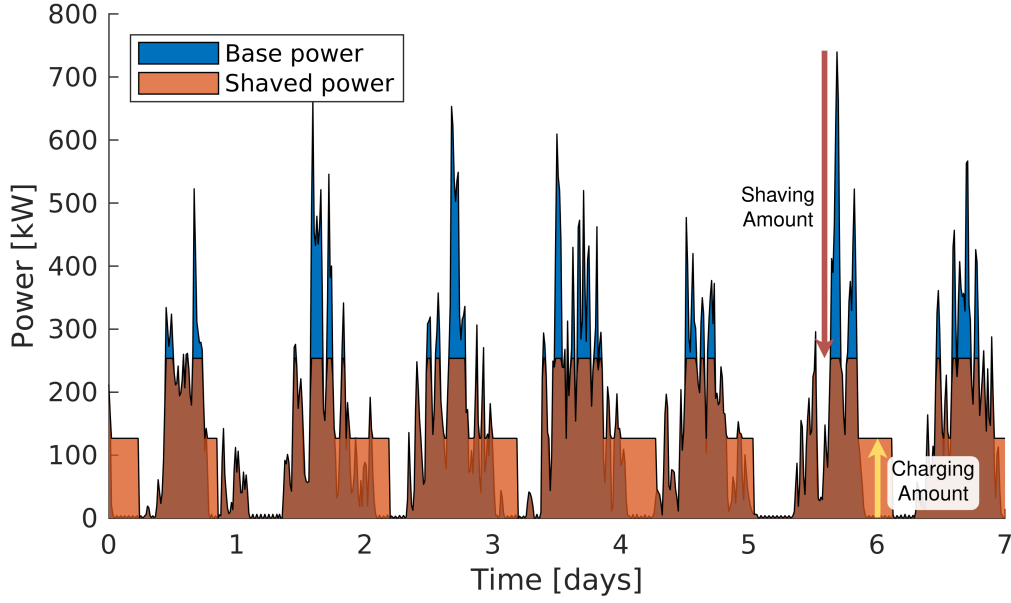
**Minimum capacity algorithm:** this algorithm is used to obtain an estimate of the capacity required for an ESS to be able to fulfill a peak-shaving plan.

**Ideal ESS model:** this model represents an “ideal” energy storage system, which has no inefficiencies and therefore no waste. Its results were used to evaluate the performance of more realistic systems.

**BESS models:** these models simulate the physical dynamics of a battery energy storage system when tackling a given peak-shaving plan.

**HESS models:** these models simulate the physical dynamics of a hydrogen energy storage system when tackling a given peak-shaving plan.

**CAPEX/OPEX models:** these calculate the CAPEX/OPEX of installing and operating an ESS for a given time period.



**Figure 3.1:** Illustration of the peak-shaving algorithm executed by an idealised ESS (with perfect efficiency). The arrows denote the function of the `shavingAmount` and `chargingAmount` parameters.

### 3.1 Peak-shaving algorithm

Given a time-dependent power profile  $P : [t_s; t_e] \mapsto \mathbb{R}^+$ , the peak-shaving algorithm is tasked with calculating a *power set-point* profile for the energy storage system. This determines how much power an energy storage system should be drawing or discharging such that the resulting power draw of the overall system has a lower peak load.

The power set-point is effectively an open-loop control input  $u_P : [t_s; t_e] \mapsto \mathbb{R}$ . By convention, we say that  $u_P(t) > 0$  indicates charging of the ESS, and  $u_P(t) < 0$  indicates discharging. This leads to the following intuitive definition of the resulting power  $\hat{P}(t)$ , which corresponds to the aggregate power flowing over the transformer station supplying the fast-charging station:

$$\hat{P}(t) := P(t) + P_{ESS}(u_P(t), P(t)), \quad (3.1)$$

where  $P_{ESS}(\cdot, \cdot)$  accounts for the real-world (or simulated) physical dynamics of a concrete ESS. In the ideal case of a “perfect” ESS, we would have  $P_{ESS}(u_P(t), P(t)) = u_P(t)$ .

The algorithm described in this section is implemented in the `PeakShaving` model in the ESPS framework.

The `shavingAmount` parameter  $\alpha_s$  determines the maximum power draw  $P_{high}$  we allow, and represents a fraction of the distance between the average of  $P$  and its maximum.

The `chargingAmount` parameter  $\alpha_c$  determines how much power  $P_{low}$  we are allowed to draw to charge the ESS, and represents a fraction of the upper target  $P_{high}$ :

$$\bar{P} := \frac{1}{t_e - t_s} \int_{t_s}^{t_e} P(t) dt, \quad (3.2)$$

$$P_{high} := \max_{t \in [t_s; t_e]} P(t) - \alpha_s \left( \max_{t \in [t_s; t_e]} P(t) - \bar{P} \right), \quad (3.3)$$

$$P_{low} := \alpha_c P_{high}. \quad (3.4)$$

See Figure 3.1 for an illustration of the function of the `shavingAmount` and `chargingAmount` parameters. The control input (power set point) is then calculated as follows:

$$u_P(t) := \begin{cases} P_{high} - P(t) & P(t) > P_{high}, \\ P_{low} - P(t) & P(t) < P_{low}, \\ P(t) & \text{otherwise.} \end{cases} \quad (3.5)$$

A value of  $\alpha_s = 1$  means we completely “flatten” the power profile from the top to its average value, whereas a value of 0 means we perform no peak-shaving at all. Conversely, a value of  $\alpha_c = 1$  means we completely “flatten” the power profile from the bottom to the upper target  $P_{high}$ , whereas a value of 0 means we never charge the ESS at all.

Note that this algorithm does not take into account the feasibility of the problem. If we set the charging parameter  $\alpha_c$  too low, the ESS will not be able to store enough energy and will run out during one of the discharging phases, resulting in higher demand charges. Similarly, if we are too ambitious with setting a high value of the shaving parameter  $\alpha_s$ , a situation may occur where it is simply impossible to store enough energy during the charging phases to completely shave the peaks, no matter how high we set  $\alpha_c$ .

The effectiveness with which peak-shaving can be performed, as quantified by the CAPEX/OPEX costs, depends on the exact dynamics and efficiencies of the ESS that executes it, the distribution of the power load  $P(t)$  as well as the various CAPEX/OPEX unit costs, and is therefore difficult to predict *a priori*. Instead, in this work a grid search will be executed over values of the parameters  $\alpha_s$  and  $\alpha_c$ , which will allow us to estimate the cost-optimal parameters for each ESS accounting for all of the aforementioned effects.

### 3.2 Estimation of minimum required storage capacity

Given an ESS power set point profile  $u_P$  generated by a peak-shaving algorithm such as the one described in the previous section, we wish to estimate the minimum required storage capacity for an ESS to execute said power set points. We make the following assumptions:

1. We have an ideal ESS, meaning an ESS with infinite power capacity (both charging and discharging), perfect efficiency, instant response to control input and no limits w.r.t. rate of change of power throughputs.
2. The storage medium begins with 100% charge. This is a safe assumption for the purposes of calculating the minimum required capacity, since if the ESS were to begin with less than 100% charge, the limiting factor would not be the capacity of the ESS but the availability of energy in charging phases, until we eventually reach 100% charge.

We propose the following algorithm to calculate the minimum required energy storage capacity  $E^*$ .

---

**Algorithm 1** Calculation of minimum required ESS capacity  $E^*$

---

```

 $T \leftarrow \text{sort\_asc} \{ \{\text{zeros of } u_P\} \cup \{t_s, t_e\} \}$ 
 $E^*, E^a \leftarrow 0$ 
for  $i = 1$  to length of  $T$  do
   $E^a \leftarrow E^a + \int_{t_{i-1}}^{t_i} u_P(t) dt$ 
   $E^a \leftarrow \min(0, E^a)$ 
   $E^* \leftarrow \max(E^*, |E^a|)$ 
end for

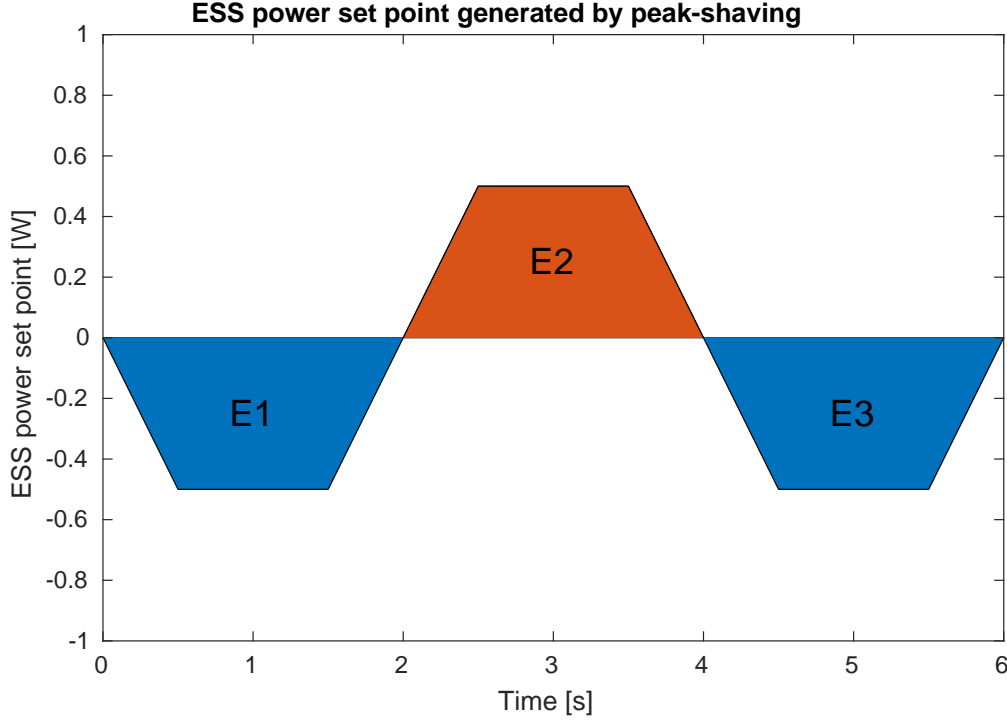
```

---

Here,  $T$  represents a sorted list containing the start and end points  $t_{i-1}, t_i$  of each charge/discharge phase. These are either sign changes in  $u_P(t)$  or one of the bounds  $t_s, t_e$  of the domain.  $E^*$  is the current estimate of the minimum required energy storage capacity, which is updated at each iteration.

Finally,  $E^a$  is an auxiliary variable representing the amount of energy required to bring the storage system back to full charge. If the storage is already at 100% charge, its value is zero; otherwise it is negative. Put differently, at any given point,  $E^a$  is the integral of all charging and discharging phases since the last time the storage system was at full charge. Since our ESS must be able to accommodate all of the discharging phases, the minimum required energy storage capacity is equal to the maximum magnitude  $|E^a|$  the ESS reaches over the course of its operations.

Algorithm 1 is implemented in the `MinESSCap` model in the ESPS framework.



**Figure 3.2:** A possible ESS power set point profile. Negative values indicate that the ESS should discharge.

### 3.2.1 Proof of correctness

In the following we will focus on proving the correctness of Algorithm 1 for the problem stated above. We will use a process of induction.

#### Base case

Consider Figure 3.2, which represents a generalised power set point profile in simplified form. We have three distinct *phases*, with phases 1 and 3 representing a discharge of the ESS and phase 2 a charging phase. Note that it follows from assumption 2 that the first phase is a discharge phase; if it weren't, we could simply skip the initial charging phase since we begin with 100% charge (and indeed this is what would occur with Algorithm 1). Also, consecutive phases will always alternate in type (charge/discharge), as otherwise they would be grouped into a single phase. Figure 3.2 is therefore representative of every possible three-phase power set point profile.

We define the total energy  $E_i$  (dis)charged over the course of phase  $i$  as follows:

$$\hat{E}_i := \int_{t_{i-1}}^{t_i} u_P(t) dt, \quad (3.6)$$

$$E_i := |\hat{E}_i|. \quad (3.7)$$

We now have three principal scenarios with regards to the minimum required storage capacity  $E^*$ :

$E_2 \geq E_1$ : the charging phase is sufficient to restore the ESS to full capacity before the second discharging phase. This means the minimum required capacity becomes  $E^* = \max(E_1, E_3)$ . Stepping through the algorithm's iterations we see how it comes to the correct conclusion:

1.  $E_1^a = \hat{E}_1$  and  $E_1^* = E_1$ .
2.  $E_2^a = \min(0, E_1^a + \hat{E}_2) = 0$  and  $E_2^* = \max(E_1^*, |E_2^a|) = E_1$ .
3.  $E_3^a = \min(0, \hat{E}_3) = \hat{E}_3$  and  $E_3^* = \max(E_2^*, |E_3^a|) = \max(E_1, E_3)$ .

$E_1 \geq E_2 \geq E_3$ : if the charging phase does not fully recharge the ESS before phase 3, then the two discharge phases become "coupled". In this case however, since  $E_2 \geq E_3$ , we know that the minimum required capacity is actually  $E^* = E_1$ , since the second discharge phase is effectively "nullified" by the charge phase. Stepping through the algorithm's iterations:

1.  $E_1^a = \hat{E}_1$  and  $E_1^* = E_1$ .
2.  $E_2^a = \min(0, E_1^a + \hat{E}_2) = \hat{E}_1 + \hat{E}_2$  and  $E_2^* = \max(E_1^*, |E_2^a|) = E_1$ .
3.  $E_3^a = \min(0, E_2^a + \hat{E}_3) = \hat{E}_1 + \hat{E}_2 + \hat{E}_3$  and  $E_3^* = \max(E_2^*, |E_3^a|) = \max(E_1, |\hat{E}_1 + \hat{E}_2 + \hat{E}_3|) = E_1$ .

$E_1 \geq E_2; E_3 \geq E_2$ : here we have a similar scenario as before, but because  $E_3 \geq E_2$ , the minimum capacity becomes  $E^* = E_1 + E_3 - E_2 = |\hat{E}_1 + \hat{E}_2 + \hat{E}_3|$ . It should be clear that the algorithm's iterations proceed as in the previous scenario, with the only difference being that now  $E_3^* = \max(E_1, |\hat{E}_1 + \hat{E}_2 + \hat{E}_3|) = |\hat{E}_1 + \hat{E}_2 + \hat{E}_3|$ .

This proves the correctness of the algorithm for any valid initial condition.

#### Induction step

Given that we have calculated the correct values of  $E_k^*$  and  $E_k^a$ , we wish to prove that Algorithm 1 correctly calculates  $E_{k+1}^*$  and  $E_{k+1}^a$ .

If the system is at full charge, i.e. if  $E_k^a = 0$ , we are effectively tackling a new sub-problem, independent of previous power set points. The final required capacity will be the maximum between the capacities required for the individual sub-problems. Therefore, given that the algorithm always updates  $E^*$  with a maximum ( $\max(E_k^*, E_{k+1}^*)$ ), we can refer to the previous section for correctness of the algorithm in this scenario.

If  $E_k^a < 0$ , we have two scenarios, depending on whether  $k + 1$  is a charging or discharging phase:

$\hat{E}_{k+1} > 0$ : phase  $k + 1$  is a charging phase, which means the required capacity should not change. Indeed, the magnitude of the auxiliary variable will increase:  $|E_{k+1}^a| =$

$\min(0, |E_k^a + E_{k+1}|) < |E_k^a|$  – this means the estimated minimum capacity remains the same too:  $E_{k+1}^* = \max(E_k^*, |E_{k+1}^a|) = \max(\max(E_{k-1}^*, |E_k^a|), |E_{k+1}^a|) = \max(E_{k-1}^*, |E_k^a|) = E_k^*$ .

$\hat{E}_{k+1} < 0$ : phase  $k + 1$  is a discharging phase. The auxiliary variable will be correctly updated:  $E_{k+1}^a = \min(0, E_k^a + \hat{E}_{k+1}) = E_k^a + \hat{E}_{k+1}$ . If after this update its magnitude exceeds the previous required capacity, the optimal required capacity is updated accordingly:  $E_{k+1}^* = \max(E_k^*, |E_{k+1}^a|)$ .

Having proven the correctness of Algorithm 1 for any valid initial condition, as well as for the step  $k \rightarrow k + 1$ , by mathematical induction it follows that Algorithm 1 is correct.

□

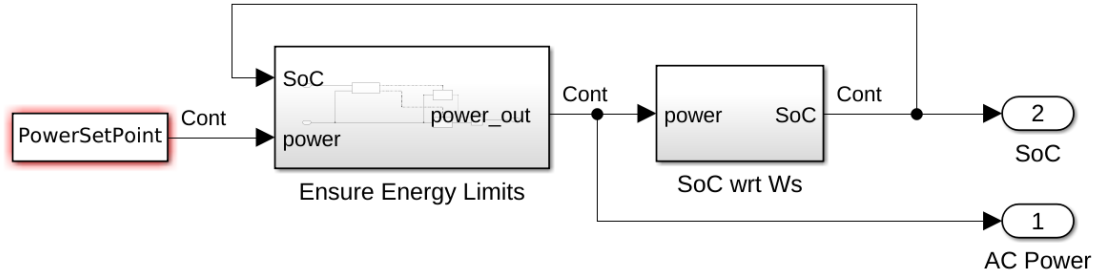


Figure 3.3: Overview of the IdealESS Simulink model.

### 3.3 Ideal ESS

In order to evaluate the performance of Hydrogen and Battery Energy Storage Systems, we would like to establish a “best-case” scenario. Thus, the first physical model we will look at is that of an “ideal” ESS which:

- has perfect efficiency when storing and releasing energy,
- has no limits on the power throughput when charging or discharging,
- has instant response time to control inputs,
- but has limited energy storage capacity.

In practice, the ideal ESS model gives us a lower bound on the OPEX we can achieve with a given control input  $u_p$ . Any realistic ESS would have inefficiencies, which translate to energy waste and increased energy costs. These inefficiencies can also increase demand charges, since we risk running out of usable stored energy sooner when discharging.

Figure 3.3 shows an overview of the Simulink implementation of this ideal ESS model. The State of Charge (SoC) is a scalar value in the range  $[0;1]$  which indicates the fraction of the total storage capacity that is currently being used. The energy limits are imposed as follows:

$$P_{IESS}(t) := \begin{cases} 0 & \text{SoC} \geq 1 \wedge u_p(t) \geq 0, \\ 0 & \text{SoC} \leq 0 \wedge u_p(t) \leq 0, \\ u_p(t) & \text{otherwise.} \end{cases} \quad (3.8)$$

In other words, we cannot further charge the ESS if we are already at maximum charge ( $\text{SoC} = 1$ ) and we cannot discharge the ESS if the energy storage is empty ( $\text{SoC} = 0$ ).

The ideal ESS described in this section is implemented in the `IdealESS` model in the ESPS framework.



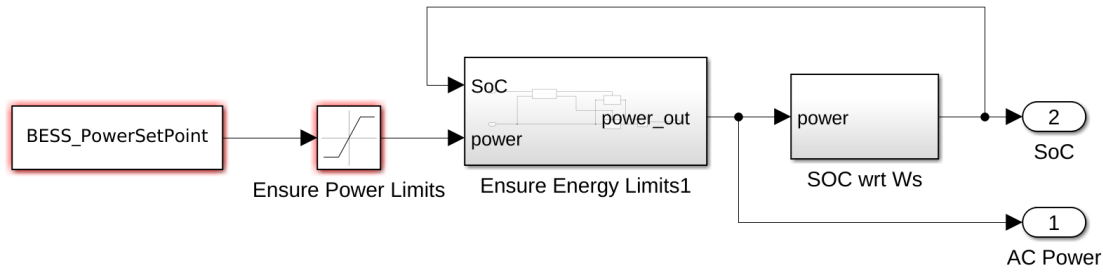


Figure 3.4: Overview of the SimpleBESS Simulink model.

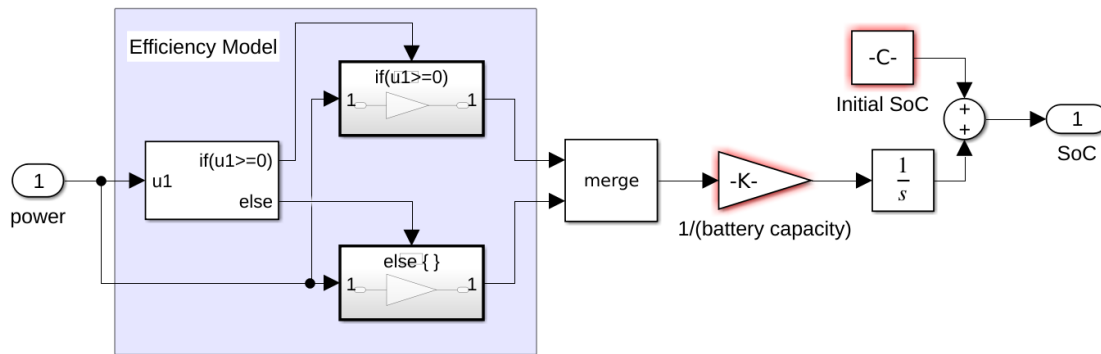


Figure 3.5: Overview of the SimpleBESS Simulink State of Charge model.

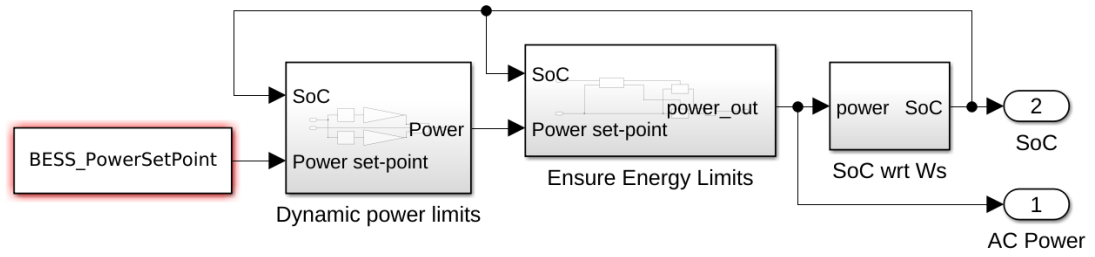
## 3.4 BESS physical simulation

This section will cover the physical modelling of Battery Energy Storage Systems. There are currently two models implemented: the first, *SimpleBESS*, is based on previous works [2], whereas the second, *EtaBESS*, was developed as part of this project and features improved modelling of efficiency, operating constraints and available capacity as a function of (dis)charge rate.

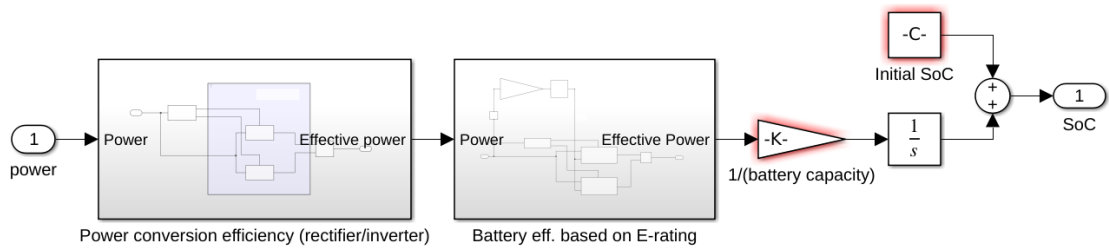
### 3.4.1 Simple BESS model

The *SimpleBESS* is based on work by R. Lehner in a prior Semester Project at PSI [2], and is similar to the ideal ESS model introduced in the previous section, with the following modifications:

- *SimpleBESS* has limited input and output powers, imposed with a saturation filter;
- constant efficiency factors are introduced in the SoC model for charging and discharging phases.



**Figure 3.6:** Overview of the EtaBESS Simulink model.



**Figure 3.7:** Overview of the EtaBESS Simulink State of Charge model.

#### 3.4.2 BESS with improved efficiency modelling

The `EtaBESS` model improves on `SimpleBESS` in the following ways:

- scaling of the power limits based on the State of Charge;
- efficiency modelling of power conversion electronics via look-up tables;
- modelling of internal battery efficiency dependant on the E-rate of the charge/discharge process.

##### Power limit scaling

As the battery's State of Charge approaches its maximum, the effective power with which we can continue to charge it tapers off. Conversely, when the battery is almost completely discharged, the effective power we can continue to draw also drops significantly.

`EtaBESS` models these effects by introducing look-up tables which dynamically scale the maximum charge and discharge powers as a function of the current SoC of the battery. Table 3.1 lists the values of the look-up tables as percentages of the maximum power limits (which are calculated as in `SimpleBESS`). Intermediate values are evaluated using linear interpolation.

**Table 3.1:** Power limit scaling look-up tables in the `EtaBESS` model. Intermediate values calculated by linear interpolation. Values provided by project supervisor.

SoC	Charge limit [%]	Discharge limit [%]
0	100	0
0.2	100	0
0.4	100	100
0.6	100	100
0.8	0	100
1.0	0	100

**Table 3.2:** Power conversion efficiency look-up tables in the `EtaBESS` model. Values provided by project supervisor and based on empirical measurements. See Figure 3.8.

Power [%]	Rectifier eff. [%]	Inverter eff. [%]
0	1	1
6.7	73.5	73.5
20	87.5	87.5
33.3	91.5	91.5
46.7	92	92
60	94.1	94.1
100	94.1	94.1

### Power conversion efficiency

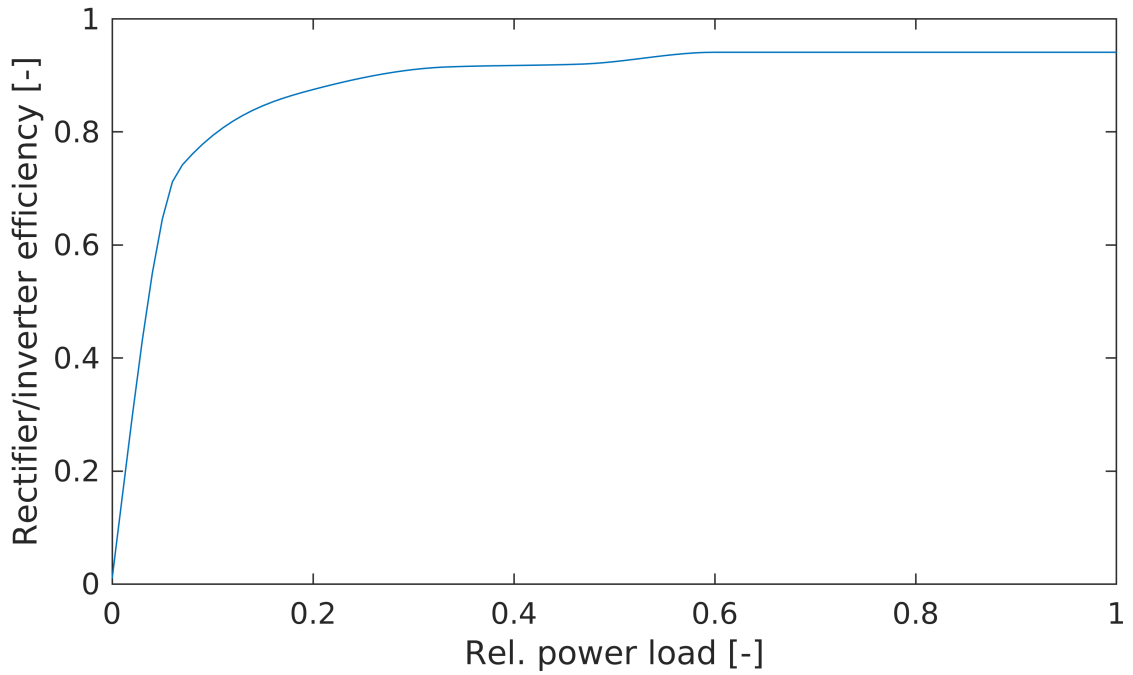
During charging phases, a rectifier is converting the incoming AC power to DC; conversely, an inverter converts DC to AC when discharging. These components cause losses in the usable energy, and their efficiency varies depending on how much power is being converted relative to the component's maximum rated capacity.

`EtaBESS` models these effects by introducing look-up tables which evaluate the rectifier/inverter efficiency as a function of their maximum rated power. The resulting efficiency then scales the power input/output to the battery.

Table 3.2 lists the values of the look-up tables, where the power value is a percentage of the maximum power rating of the component. The tables were resampled to 100 evenly-spaced values using piecewise cubic Hermite interpolation (PCHIP) before use in simulation. The LUTs were then evaluated using the "Nearest" algorithm in Simulink to improve the stability of the simulation.

### Battery efficiency as a function of E-rate

An E-rate describes the rate at which a battery is discharged relative to its maximum energy capacity [5]. An E-rate of  $1\text{h}^{-1}$  means that the discharge power will discharge the entire battery in one hour. An E-rate of  $2\text{h}^{-1}$  means the battery will last thirty minutes.

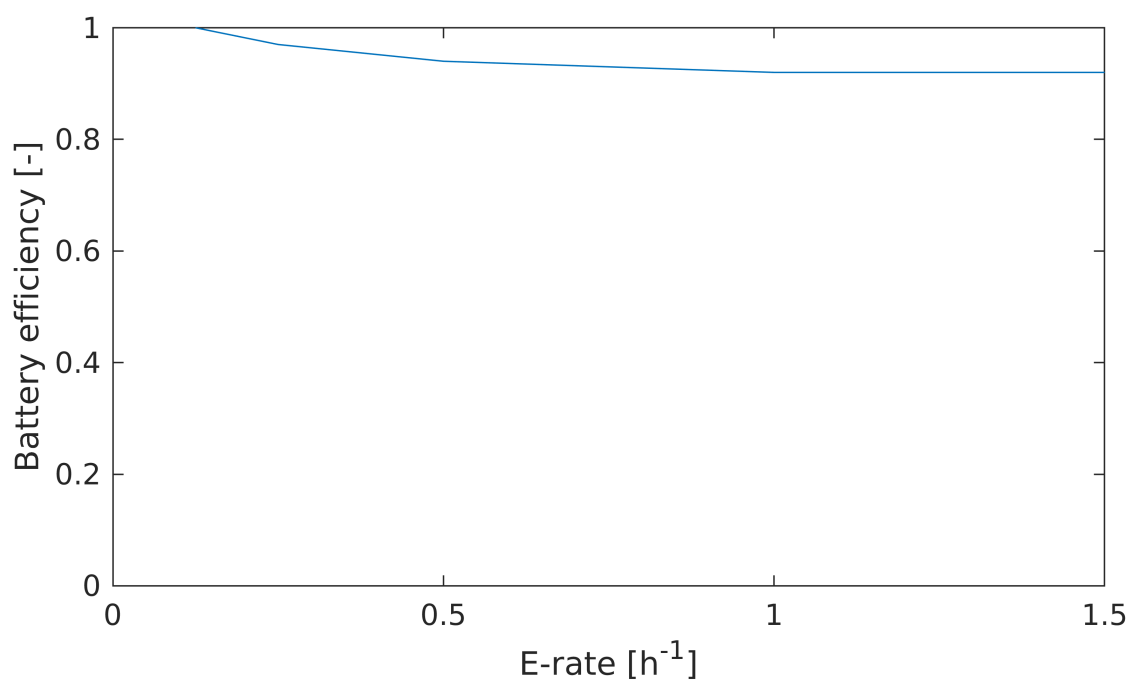


**Figure 3.8:** Efficiency of AC/DC and DC/AC power conversion. See Table 3.2.

**Table 3.3:** Battery efficiency look-up tables in the `EtaBESS` model. Values based on [6] and interpolated linearly. A value of 100% means the available capacity is 100% of the nominal capacity; this was translated to efficiency in our model. See Figure 3.9.

E-rate [ $\text{h}^{-1}$ ]	Charge eff. [%]	Discharge eff. [%]
1.5	92	92
1	92	92
0.5	94	94
0.25	97	97
0.125	100	100

The efficiency with which a battery can be charged or discharged depends on the E-rate with which it is charged or discharged. `EtaBESS` models this with an additional pair of look-up tables relating E-rate to efficiency. This efficiency is applied in series with the efficiency of power conversion previously described. Table 3.3 lists the values of the look-up tables, which are based on [6].



**Figure 3.9:** Internal battery efficiency as a function of E-rate. See Table 3.3.

### 3.5 HESS physical simulation

In this section we will briefly cover the physical modelling of Hydrogen Energy Storage Systems. As in the previous section, there is a `SimpleHESS` model which was adapted from previous works [2], and a second version `EtaHESS` which improves on the simpler version.

#### 3.5.1 Simple HESS model

Unlike the BESS models, which directly integrate the power input/output from the storage system to calculate the change in state-of-charge, in a HESS model we must first convert power to/from Hydrogen gas mass flow.

##### Electrolyser

The electrolyser consumes power to produce hydrogen gas. In `SimpleHESS`, the mass flow  $\dot{m}_{H2,ELY}(t)$  resulting from power input  $u_P(t)$  is calculated as follows:

$$\dot{m}_{H2,ELY}(t) = \frac{u_P(t) \cdot \eta_{ELY} \cdot \rho_{H2}}{W_{H2}}, \quad (3.9)$$

where  $\eta_{ELY} = 0.65$  is the electrolyser efficiency (including AC/DC power conversion),  $\rho_{H2} = 0.0899 \text{ kg/Nm}^3$  is the normalised density of  $H_2$  gas and  $W_{H2} = 3 \text{ kWh/Nm}^3$  is the energy density of  $H_2$  gas.

##### Fuel Cell

The fuel cell consumes hydrogen gas to produce electricity. In `SimpleHESS`, the mass flow  $\dot{m}_{H2,FC}(t)$  the fuel cell draws from the tank to produce the required power output  $u_P(t)$  is calculated as follows:

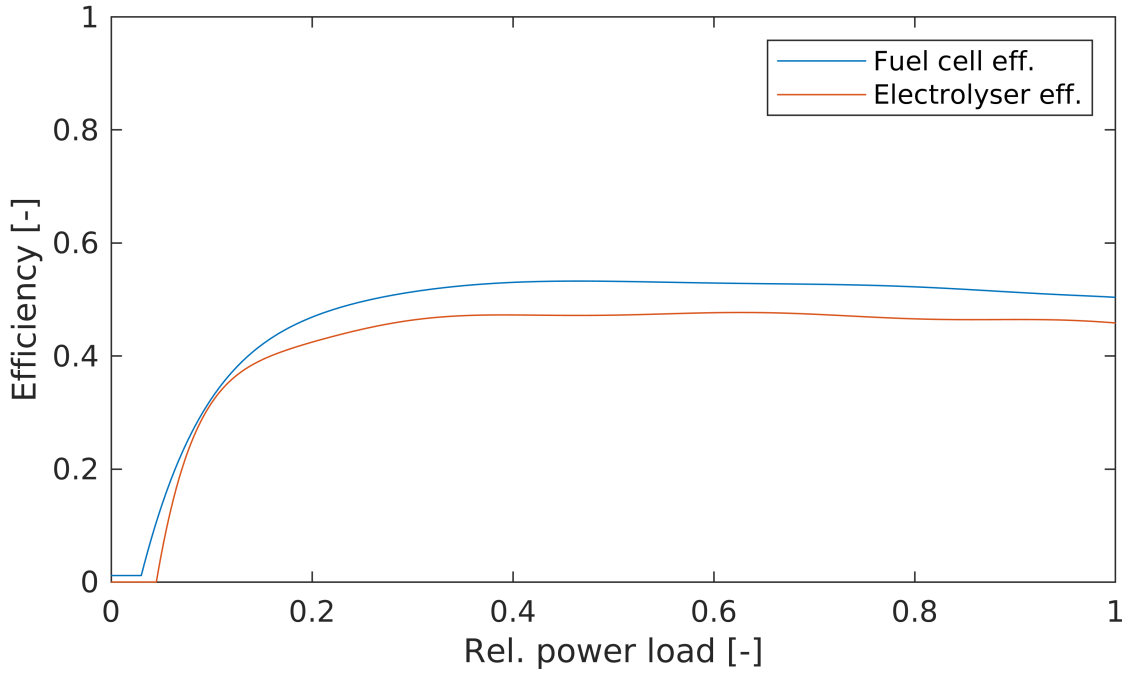
$$\dot{m}_{H2,FC}(t) = \frac{u_P(t) \cdot \rho_{H2}}{\eta_{FC} \cdot W_{H2}}, \quad (3.10)$$

where  $\eta_{FC} = 0.65$  is the fuel cell efficiency (including DC/AC power conversion). It is worth noting that the power set point  $u_P$  is negative when the fuel cell is in operation, and, accordingly, the resulting negative mass flow drains the tank.

##### H2 storage tank

The mass content of  $H_2$  gas in the storage tank is simply calculated by integrating the mass flow resulting from either the electrolyser or fuel cell over time. The pressure in the tank is calculated using ideal gas relations:

$$p_{H2}(t) = p_0 + \frac{R_{S,H2} \cdot T_{amb}}{V_{tank}} \int_0^t \dot{m}_{H2}(\tau) d\tau, \quad (3.11)$$



**Figure 3.10:** Efficiency curves of fuel cell ( $\eta_{FC}$ ) and electrolyser ( $\eta_{ELY}$ ) in the EtaHESS model.

where  $R_{S,H_2} = 4123.2 \text{ Ws/kg/K}$  is the specific constant for  $H_2$  gas,  $T_{amb} = 300 \text{ K}$  is the ambient temperature and  $V_{tank}$  is the volume of the storage tank.

### 3.5.2 HESS with improved efficiency modelling

SimpleHESS models the efficiencies of fuel cell and electrolyser as constant values. In reality, their efficiencies can vary greatly depending on the power load on the components. The EtaHESS replaces the constant efficiencies in equations 3.10 and 3.11 with look-up tables that express efficiency as a function of relative power load.

Figure 3.10 illustrates the values of the efficiency look-up tables used for fuel cell and electrolyser.

**Table 3.4:** Component prices for HESS and BESS.

Component	Current (high)	Future (low)	Unit	Sources
Battery storage	800	300	[CHF/kWh]	[7], [8]
H2 tank	37	7.4	[CHF/kWh]	[9]
Fuel Cell	1700	230	[CHF/kW]	[10], [11]
Electrolyser	1000	300	[CHF/kW]	[12], [13]

### 3.6 Calculation of CAPEX

Table 3.4 lists the costs of individual components of the battery and hydrogen storage systems per unit capacity. Both current and expected future prices are reported.

### 3.7 Calculation of OPEX

In our scenario, the operational expenditures correspond to electricity costs. Of those, the two primary components considered are energy charges, which depend on the total amount of energy consumed, and demand charges, which are based on the highest monthly 15-minute-averaged power load:

$$\text{OPEX}_{\text{Energy}} = \varepsilon_E \cdot C_E \cdot \frac{t_{\text{OPEX}}}{t_e - t_s} \int_{t_s}^{t_e} \hat{P}(\tau) d\tau, \quad (3.12)$$

$$\text{OPEX}_{\text{Demand}} = \varepsilon_D \cdot C_D \cdot t_{\text{OPEX}} \sum_{\text{months}} \max_{\text{month}} \frac{\tilde{P}(\tau)}{t_{\text{month}}}, \quad (3.13)$$

$$\text{OPEX} = \text{OPEX}_{\text{Energy}} + \text{OPEX}_{\text{Demand}}. \quad (3.14)$$

Here,  $t_{\text{OPEX}}$  refers to the time horizon over which we wish to evaluate the OPEX costs, which may differ from the time horizon  $t_e - t_s$  of the physical simulation.  $C_E = 0.0739 \text{ CHF/kWh}$  is the unit price of electrical energy [14] and  $C_D$  is the monthly unit demand charge price. The latter will have values ranging from 6 CHF/kW/m to 30 CHF/kW/m depending on the scenario.  $\tilde{P}(\cdot)$  refers to the aggregate power demand ( $\hat{P}(\cdot)$  as defined in Equation 3.1) averaged over 15-minute time windows.

There is one additional aspect to the calculation of operational expenses, and that is *usage time* (*Benutzungsdauer*, BD), which is calculated as follows [14]:

$$\text{BD} = \frac{\text{Total yearly energy [kWh]}}{\text{Avg. monthly peak [kW]}}. \quad (3.15)$$

If the usage time is above the threshold  $T_{\text{BD}} = 3500 \text{ h}$ , the fees for energy and demand change. This is implemented in our OPEX model with the  $\varepsilon_E$  and  $\varepsilon_D$  multipliers, the



values of which are based on electricity pricing in Switzerland at time of writing [14]:

$$\varepsilon_E := \begin{cases} 1 & \text{BD} < T_{BD}, \\ 0.540 & \text{BD} \geq T_{BD}; \end{cases} \quad (3.16)$$

$$\varepsilon_D := \begin{cases} 1 & \text{BD} < T_{BD}, \\ 2.122 & \text{BD} \geq T_{BD}. \end{cases} \quad (3.17)$$

Note that this method is based on the assumption that, as demand charges rise in the future, the high-usage rates will remain proportional to the low-usage rates by the same ratio. It remains to be clarified whether this is accurate but, as we will see when discussing the results, this approach significantly limits the effectiveness of peak-shaving techniques.



## Chapter 4

---

# Discussion of results

---

In this chapter we will look at the simulations that were run using the new code framework and models and analyse the results obtained.

Three scenarios were studied:

**Scenario 1** uses real-world power data from a rest stop in Switzerland for 2019.

**Scenario 2** uses only the most energy-intensive week from the 2019 dataset as the input power profile.

**Scenario 3** uses data from the same week as scenario 2, but extrapolated to a 30% BEV share, which is meant to be representative of a future scenario.

Each dataset was provided by the principal project supervisor C. Peter.

### 4.1 Methodology

For each scenario, a grid-search was executed over the shaving amount ( $\alpha_s$ ) and charging amount ( $\alpha_c$ ) parameters of the peak-shaving algorithm (see section 3.1), with values ranging from 0.2 to 1.0. CAPEX values were calculated as per section 3.6 for both current (high) and future (low) component prices.

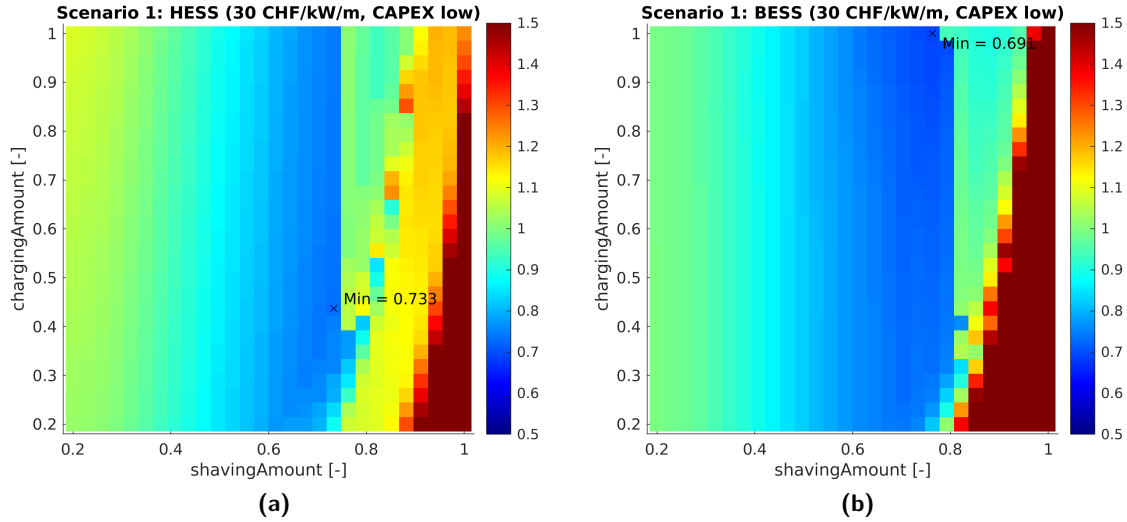
The resolution for this grid search was 30x30 (900 points) on scenarios 2 and 3. Scenario 1, being significantly more computationally intensive, was evaluated at a lower resolution of 28x28 (784 points) and took around 47.5 hours to simulate on a Ryzen 7 1700 8-core CPU, with 8 Matlab parallel workers. It is worth noting that such high resolutions were only achievable by using the parallel grid search functionality of the implemented framework; executing the same simulation sequentially would have required approximately 392 hours (16 days).

The physical simulations were run using the `EtaBESS` and `EtaHESS` models described in sections 3.4 and 3.5 respectively. The initial state of charge was set to 20%, which corresponds to the minimum value for both models.

#### 4. DISCUSSION OF RESULTS

---

In a subsequent step, OPEX values were calculated by adding a third dimension to the grid-search, for five values of demand charge prices ranging from 6 CHF/kW/month (approximately representative of the current pricing in Switzerland) to 30 CHF/kW/month. Such high demand charge values could realistically be expected in the future, as the transition towards renewable energies progresses, and have indeed already been reached in some areas of the United States (California in particular) [2]. All OPEX results were scaled to a time horizon of 10 years (3650 days) with constant energy and demand charge prices over time.



**Figure 4.1:** Scenario 1: HESS (left) and BESS (right) relative cost heat-maps for a base demand charge of 30 CHF/kW/month and low CAPEX prices.

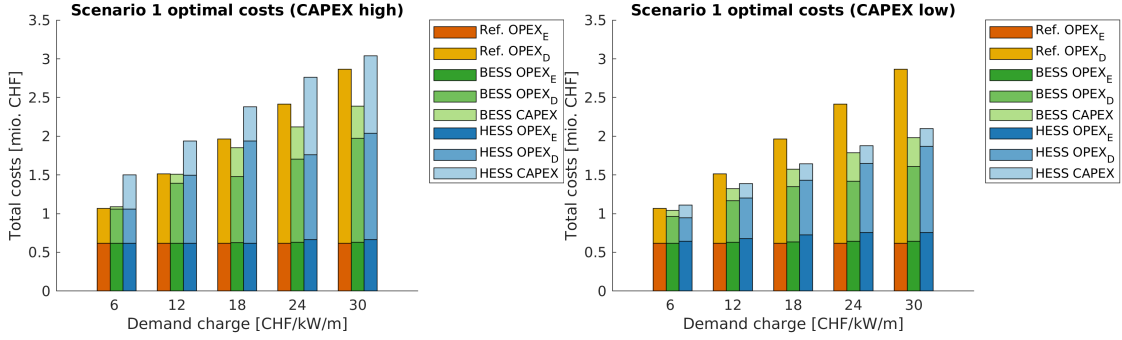
## 4.2 Scenario 1: power data from all of 2019

Figure 4.1 shows heat-maps visualising the *relative cost* of a BESS installation when compared to no ESS installation (and thus unmodified OPEX costs). A value of 1 (green hue) means that installing and operating a BESS for 10 years costs the same as operating the station without any peak-shaving for the same duration. Higher values (warm colours) indicate that the BESS solution is more expensive, whereas lower values (colder colours) indicate potential savings. Finally, a marker indicates the lowest value, i.e. the cost-optimal solution for the given ESS, demand charge and CAPEX values.

Note that in the interest of brevity, only certain parameter combination heat-maps are shown in this chapter. Refer to Appendix ?? for a full list.

In general, higher values of the shaving amount parameter lead to lower demand charge costs (and hence OPEX), but require a larger storage capacity, and so increase the CAPEX of our installation. High values of the shaving amount also require higher charging amounts to be beneficial: if we have a large storage capacity, but don't pre-charge it sufficiently to cover all of the peaks, we lose the benefit of peak-shaving, and are left paying for the installation and higher energy costs (due to inefficiencies in the ESS). For extremely high shaving amounts, it may not even be physically possible to pre-charge the ESS sufficiently. Therefore, the exact location of the trade-off boundary depends not only on the ESS dynamics, but also on the properties of the power demand profile being used for the simulations.

Notice how the minimum in Figure 4.1b has a charging amount of 1: this is because the BESS incurs no additional costs for increasing the charging amount, and so has



**Figure 4.2:** Scenario 1 summary: costs of optimal solutions per ESS for varying demand charge values, low and high CAPEX. Orange bars represent baseline OPEX (no peak-shaving), green bars are with BESS, and blue bars with HESS. Each bar has three segments, from bottom to top: energy costs, demand costs, and CAPEX.

no reason not to charge as much as possible. The CAPEX of the BESS are indeed calculated based solely on the required capacity, which is affected primarily by the shaving amount rather than the charging amount. This may seem counter-intuitive, but is a direct consequence of how the minimum required ESS capacity is calculated (see Section 3.2). The situation is different for the HESS, since higher charging amounts require a larger electrolyser, which in turn increases the CAPEX of the installation.

Observe in Figure 4.1b the sharp discontinuity in costs for the BESS installation at a shaving amount value of approximately 0.8. This is due to the usage time calculation described in Section 3.7: higher shaving amounts lead to lower peak loads and higher energy consumption (due to inefficiencies in the ESS), which overall produce higher usage time values. When we reach a usage time of 3500 hours, we switch to the modified tariffs, which in most cases produce higher costs.

The hydrogen system (Figure 4.1a) also features this discontinuity, however, it occurs for lower shaving amounts, since the HESS has lower round-trip efficiency than an equivalent battery system.

In general, the effectiveness of both peak-shaving systems is significantly limited by the usage time policy, when evaluated as described in Section 3.7.

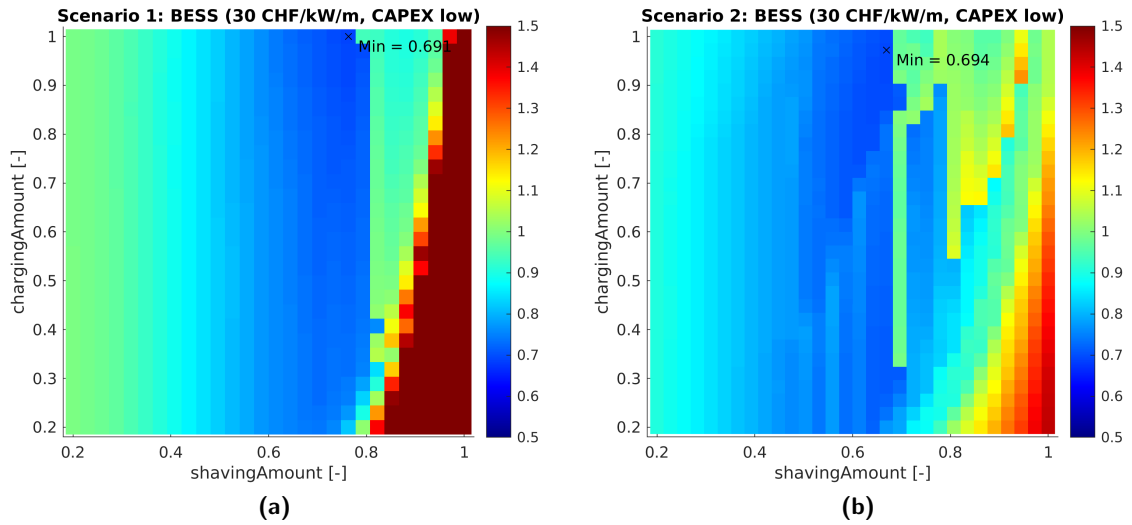
Figure 4.2 summarises the results for the first scenario. The “Baseline OPEX” bars indicates the total OPEX costs we expect over a 10-year time horizon with no peak-shaving installation. The bars for BESS and HESS indicate the costs of the cost-optimal solution for each ESS. The term “cost-optimal solution” refers to the combination of shaving and charging amounts which minimises the total costs. In other words, each of the previously introduced heat-maps has a corresponding bar in figure 4.2.

As we would expect, higher values of unit demand charge prices lead to a more competitive optimal solution for any ESS, since the financial benefit of peak-shaving is accentuated. Similarly, lowering the component costs of an ESS installation (i.e. CAPEX)

means we can afford larger installations, higher amounts of peak shaving and more savings on OPEX.

In conclusion, we find that for this scenario:

- at current demand charges and component prices, neither BESS or HESS is cost-competitive compared to the baseline costs;
- for higher demand charge values and current component prices, the BESS can become competitive, but the HESS is still too expensive;
- the HESS can become competitive with lower component prices and higher demand charge values, but is still outperformed by the BESS (albeit narrowly).



**Figure 4.3:** Comparison of equivalent heat-maps from scenario 1 (left) and scenario 2 (right).

### 4.3 Scenario 2: most energy-intensive week of 2019

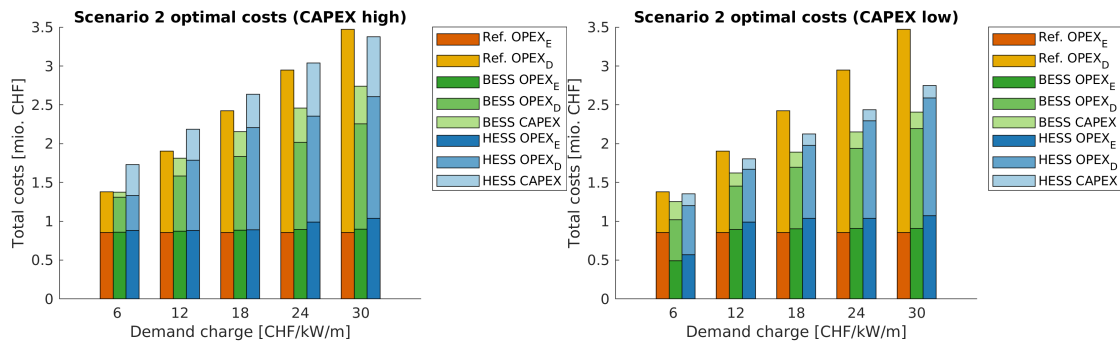
In this scenario, only the most energy-intensive single week of 2019 was used for the physical simulations. The shorter time horizon makes for much faster simulations (less than a minute, compared to the almost 40 minutes of scenario 1), but sacrifices solution quality.

Figure 4.3 serves as an illustration of this difference in solution quality: it shows the heat-maps resulting from a one-year (left) and a one-week (right) physical simulation, with the same ESS, demand charges and CAPEX values. The one-week plot is visibly more irregular; the optimisation surface has many more local minima and maxima which are “smoothed over” by the longer simulation. The risk here is that one of these local minima is mis-identified as a globally optimal solution which does not hold true in long-term, real-world operation.

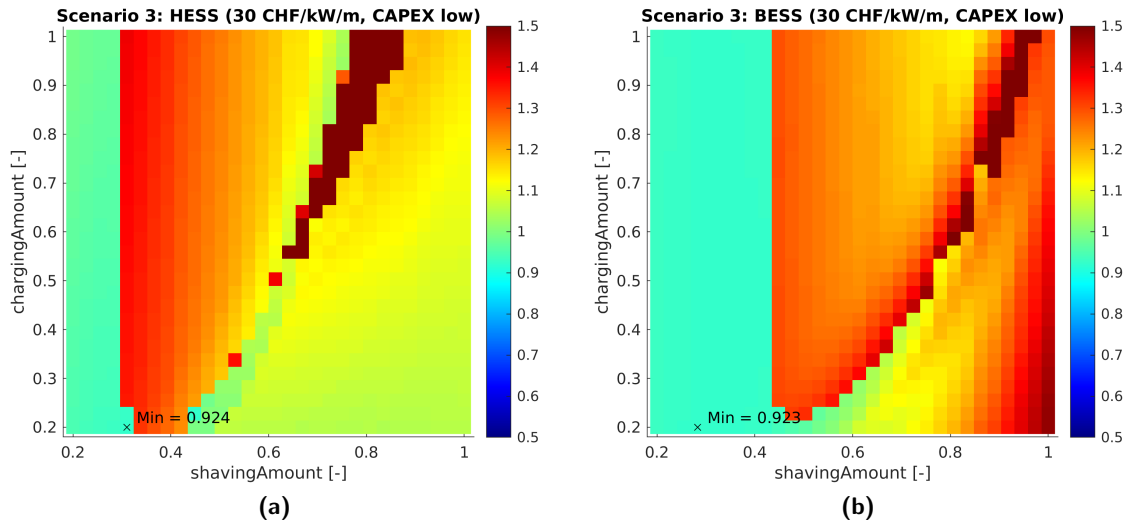
Figure 4.4 summarises the results for Scenario 2. In general, it paints a more optimistic picture when compared to the previous scenario, as the data used was from the most energy-intensive week of the year and therefore has higher total consumption and peak loads than the average week of 2019.

However, it also highlights one of the aforementioned “local minima”: for low CAPEX values and base demand charges of 6 CHF/kW/month, the optimisation finds a minimum above the high-usage threshold, as highlighted by the fact that the energy costs are lower than baseline. This minimum is not present in scenario 1 (Figure 4.2) and so is most likely not representative of long-term operation.





**Figure 4.4:** Scenario 2 summary: costs of optimal solutions per ESS for varying demand charge values, low and high CAPEX. Orange bars represent baseline OPEX (no peak-shaving), green bars are with BESS, and blue bars with HESS. Each bar has three segments, from bottom to top: energy costs, demand costs, and CAPEX.



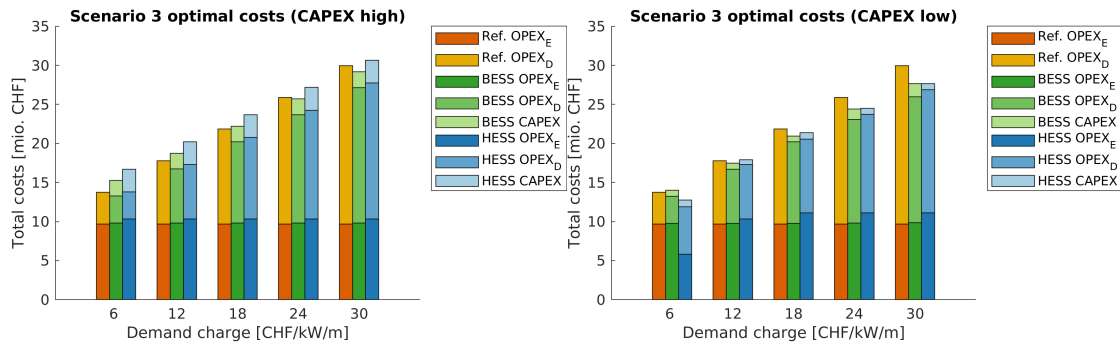
**Figure 4.5:** Scenario 3: comparison of HESS and BESS for low CAPEX and 30 CHF/kW/month demand charges.

### 4.4 Scenario 3: most energy-intensive week of 2019 with 30% BEV share

In the third and final scenario investigated, a synthesised dataset was used – based on the single week from the previous example, but extrapolated to a 30% traffic share of battery-electric vehicles [1]. This produces much higher peak loads and overall energy consumption. As a result, we would expect peak-shaving to perform better than in previous scenarios, since there are more potential benefits. Furthermore, a hydrogen system is expected to scale to these proportions better than a battery system, since a hydrogen system can accommodate the significantly larger storage capacity at lower prices.

Unfortunately, as exemplified by the heat-maps in Figure 4.5, the usage time policy severely limits the amount of peak-shaving that can be effectively performed, since the baseline usage time of the un-shaved power profile is also much higher. As a result, both hydrogen and battery energy storage systems struggle to reach the break-even point with current component prices, as can be seen in Figure 4.6, and with lower component prices they are still less effective (in relative terms) than in the previous scenarios.

#### 4.4. Scenario 3: most energy-intensive week of 2019 with 30% BEV share



**Figure 4.6:** Scenario 3 summary: costs of optimal solutions per ESS for varying demand charge values, low and high CAPEX. Orange bars represent baseline OPEX (no peak-shaving), green bars are with BESS, and blue bars with HESS. Each bar has three segments, from bottom to top: energy costs, demand costs, and CAPEX.



## Chapter 5

---

# Summary and conclusions

---

Over the course of this Semester Project, a MATLAB code framework was developed for the simulation of energy-storage-based peak-shaving of electrical power demand. The specific setting investigated was that of a highway rest-stop equipped with BEV fast-charging stations.

Using the new framework, the optimal design of a HESS and BESS was determined for a variety of present-day and future scenarios, based on measured data from a real-world rest-stop as well as synthesised data.

These simulations demonstrate the effectiveness of the new framework, which makes it possible to run grid-searches over multiple parameters to find the cost-optimal solution. Furthermore, these grid-searches can be executed in parallel, which allows scaling to significantly larger simulations, search spaces, and resolutions.

To summarise the results of the simulations, with the current pricing of electricity and ESS components (batteries, electrolyser, fuel cell and hydrogen tank), neither of the energy storage systems considered (battery and hydrogen) is cost-effective for peak-shaving the power load of BEV fast-charging stations at a highway rest stop. However, as demand charges increase and component costs fall to predicted values, both systems can become effective. The battery system is in general cheaper than an equivalent hydrogen system, however, as the traffic share of BEVs increases in the future and, consequently, the required storage capacity increases, hydrogen systems are expected to outperform battery storage. The effectiveness of peak-shaving is significantly limited by the high-usage policy as it was implemented for this project, particularly for high-load scenarios.

## 5.1 Outlook

It remains to be clarified whether the high-usage policy of electricity pricing as described in Section 3.7 accurately reflects the expected future evolution. For the purposes of this project, the high-usage demand charges increase proportionally to the

low-usage demand charges; however, it has been shown that this would penalise peak-shaving in the future. Keeping instead the high-usage demand charges more or less constant while increasing the low-usage charges could greatly incentivise peak-shaving via energy storage, particularly as loads increase.

In all the simulations performed for this project, the ESS components were assumed to function at maximum performance for the entire simulation time horizon. In reality, ESS components (batteries, electrolyzers and fuel cell) would endure gradual degradation depending on their operation, reducing their efficiency. Implementing some form of component degradation could significantly improve the accuracy of the simulations, and play a role in the optimal design process.

One of the findings of this project was how much of an impact the simulation time horizon can have on the quality of results. As a result, it would be beneficial to re-evaluate the findings from the high-load scenario (scenario 3) with a longer time horizon to reduce the sensitivity to initial conditions and sub-optimal local solutions.

Finally, the current project focused primarily on the optimal design of a HESS, and used what amounts to an open-loop controller with pre-computed control inputs. However, real-world implementation of an ESS would require some form of closed-loop control. The effectiveness of different control strategies could be evaluated, such as PID or model-predictive controllers.

## Appendix A

---

# User Guide to the ESPS framework

---

This appendix serves as an overview and user guide to the ESPS (Energy Storage for Peak Shaving) framework.

### A.1 Directory and package structure

Listing A.1 shows an overview of the directory structure of the project, with some annotations describing the function of important directories.

```
1 bevfc-peak-shaving
2 |-- +esps          -> ESPS package top-level
3 |   |-- +models    -> Contains concrete Model implementations
4 |   |-- +post       -> Contains post-processing utility functions
5 |   |-- +runners    -> Contains concrete Simulation Runners
6 |   |-- +simulations -> Contains concrete Simulations
7 |   |-- +unitTests
8 |   |-- +utils      -> Contains general utility functions
9 |   \-- +validationTests
10 |-- +examples      -> Contains examples for the ESPS framework
11 |-- inputs         -> Directory where simulation inputs are stored
12 |   \-- unified     -> Input files for every simulation
13 |-- +main          -> Utility scripts for easy execution
14 |-- outputs        -> Directory where simulation outputs are stored
15 \-- slx            -> Directory where Simulink models are stored
```

**Listing A.1:** Directory structure of the ESPS project

#### A.1.1 MATLAB packages

Packages in MATLAB are a way of organising code into namespaces. Any folder with a name starting with + will be interpreted as a package by MATLAB. As long as the package folder is in the MATLAB path, any functions, classes or subpackages will be

accessible to MATLAB scripts with dot-notation. For example, if the current working directory is `bevfc-peak-shaving`:

```

1 % Function runEtaHESS defined in +main/runEtaHESS.m
2 >> main.runEtaHESS();
3 [...]
4 % Class SimEtaHESS defined in +esps/+simulations/SimEtaHESS.m
5 >> m = esps.simulations.SimEtaHESS();
6
7 % We can import a class from a package
8 >> import esps.runners.ParallelGridSearch
9
10 % Class ParallelGridSearch is now directly accessible, e.g. we can
11 % show a list of its public methods
12 >> methods ParallelGridSearch
13 ParallelGridSearch      getSearchGrid      setUserName
14 addSearchParam          readInputs      validateData
15 findUniqueOutputDirectory run            writeOutputs
16 getOutputData           runAndWrite
17
18 % Show a list of public properties of a class
19 >> properties esps.SimulationDataIO
20     inputFileDirectory
21     outputFileDirectory

```

**Listing A.2:** Examples for MATLAB packages

### A.1.2 I/O file format

Inputs and outputs for simulations are stored in the JSON+C format. JSON (JavaScript Object Notation) is a format commonly used to store data structures in text files. MATLAB natively supports converting structure arrays to/from JSON strings with the `jsonencode` and `jsondecode` functions. For the purposes of this project, a **simple parser** was written to allow the use of JSON+C (JSON with comments) in MATLAB.

```

1 { // example.jsonc
2   /*
3    * An example of a valid input file.
4    */
5   "inputs": {
6     // Parameters for peak-shaving algorithm
7     "shavingAmount": 0.75,
8     "chargingAmount": 0.5,
9     "FILEIN_time": "oneWeek_time.csv",
10    "FILEIN_powerDemand": "oneWeek_ACChargingPower.csv",
11  }
12 }

```

**Listing A.3:** Example of a valid input JSON+C file with both multi-line and single-line comments.



Listing A.3 shows an example of a short input file. When parsed, this will create a structure array with a single field named `inputs`, which in turn contains several fields for the different parameters and inputs to be used in simulation.

Listing A.4 shows an example of what the output of a simulation might look like. In addition to the inputs, the structure array now has two additional fields. `meta` contains information about how the simulation was executed, such as code version, simulation runner and simulation used, start/end times, and, in the case of a grid search (parallel or otherwise), the names of the search parameters.

`results` is an array containing one entry for each model of the simulation, in the order in which they were executed. Each element of the results array has a `ModelName` field, containing the name of the model and an `output` field which contains all of the data output by the model.

```

1 {
2   "inputs": {
3     "shavingAmount": 0.2,
4     "chargingAmount": 0.22962962962962963,
5     "FILEIN_time": "scenario01_time.csv",
6     "FILEIN_powerDemand": "scenario01_ACChargingPower.csv"
7   },
8   "meta": {
9     "SimulationRunnerName": "ParallelGridSearch",
10    "CodeVersion": "v1.0.100",
11    "UserName": "sean",
12    "SearchParams": [
13      "shavingAmount",
14      "chargingAmount"
15    ],
16    "StartTime": "26-Mar-2022 20:06:00",
17    "SimulationName": "SimCompare",
18    "EndTime": "26-Mar-2022 21:22:50"
19  },
20  "results": [
21    {
22      "ModelName": "PeakShaving",
23      "output": {
24        "totalShavedEnergy": 1.32632770286208E+12,
25        "totalChargedEnergy": 2.9901067212800166E+12,
26        "FILEOUT_ESSetPoint": "ESSetPoint.csv"
27      }
28    },
29    {
30      "ModelName": "MinESSCap",
31      "output": {
32        "minRequiredESSCapacity": 1.2648217394046636E+8
33      }
34    }
35  ]
36 }

```

---

**Listing A.4:** Example of an output JSON+C file resulting from a grid-search.

---

### FILEIN and FILEOUT

Some inputs, such as time series, can be very large and would overcrowd the JSON+C file. To avoid this, `FILEIN` and `FILEOUT` fields can be used to read and write large data sets to and from CSV files.

When parsing an input file, any field whose name starts with the prefix `FILEIN_` will be interpreted as being the name of a CSV file to read. So, in the example from Listing A.3, the field `FILEIN_time` will cause the framework to read the file `dataFiles/scenario01_time.csv` (where the `dataFiles` directory is located in the same folder as the input file) using MATLAB's builtin `readmatrix` function and store its contents in a new field named `inputs.time`.

When writing outputs, the reverse process takes place: the `FILEIN_time` field will cause the framework to write the contents of the `time` field to the file `dataFiles/scenario01_time.csv` (this time located in the directory of the output file) using MATLAB's builtin `writematrix` function and remove the `time` field before converting the output structure array to JSON format.

`FILEOUT` fields behave in the same manner as `FILEIN` fields, but are used in the `results` array for model outputs.

## A.2 ESPS models

Models are the fundamental building block of ESPS simulations. Concrete models are stored in the `+esps/+models/` directory and must extend the `esps.Model` class.

### A.2.1 Creating a new model

Here we will briefly look at the steps required to create a new ESPS model:

- Begin by duplicating one of the existing models: for this example we will copy `+esps/+models/DummyModelA.m` to `+esps/+models/MyNewModel.m`.
- Change the class name of the model, on line 1 of the new script, to match the file name of the script – in our case `MyNewModel`. It is important that the class name match the file name exactly (this is imposed by MATLAB).
- The `requiredInputNames` property is a string array of input fields for the model. Models may only read inputs that are explicitly declared in this list. Modify the list as required; it is strongly recommended you use comments to clearly indicate the unit and function of each input.

- The `outputNames` property is a string array of output fields for the model. A model may only set an output field if it has been explicitly declared in the `outputNames` array, and must set each declared output every time it is run. The only exception are `FILEOUT` fields, which will be given a default file name if declared but not set. Modify the list as required; it is strongly recommended you use comments to clearly indicate the unit and function of each input.
- The `run_` method is the core of our new model. It is called by the framework when the model is executed. In here, use `self.dataHandler.getAllInputs()` to get a structure array with one field for each declared input. Alternatively, use `self.dataHandler.getInput("inputName")` to get the inputs one at a time. After performing operations on the inputs, use `self.dataHandler.setOutput("outputName", value)` to set each declared output.

Listing A.5 shows an example of a concrete ESPS model implementation.

```

1 classdef MyNewModel < esps.Model
2     properties(Constant,Access=protected)
3         requiredInputNames = [...
4             "time",...           % [s] Array of time coordinates
5             "velocity",...       % [m/s] Array of velocity values over time
6             "initialPosition"... % [m] Initial position
7         ];
8         outputNames = [...
9             "position",... % [m] Position of the object over time
10            "FILEOUT_position"...
11        ];
12    end
13
14    methods(Access=protected)
15        function run_(self)
16            % Get inputs
17            in = self.dataHandler.getAllInputs();
18
19            % Perform some calculations
20            position = ones(length(in.time), 1) * in.initialPosition;
21            position = position + cumtrapz(in.time, in.velocity);
22
23            % Write outputs
24            self.dataHandler.setOutput("position", position);
25            % Notice how FILEOUT_position is declared but not set;
26            % the framework will give it a default value of "position.csv"
27        end
28    end
29 end

```

**Listing A.5:** Example of a concrete ESPS model implementation.

### Simulink models

Simulink models can be called from within an ESPS model. See `+esps/+models/SimpleBESS.m` for a working example.

## A.3 ESPS simulations

Simulations in the ESPS framework consist essentially of a collection of models. They are responsible for executing each model and ensuring each model receives the correct inputs. Concrete simulations are stored in the `+esps/+simulations/` directory and must extend the `esps.Simulation` class.

### A.3.1 How models look for inputs

In order to better understand how simulations work, it is useful to first understand how models navigate the inputs/results structure array to find their required fields. When a model calls `self.dataHandler.getInput("inputName")`, the following steps take place:

1. If there are results from models previously executed by this simulation, iterate over them in reverse order (newest to oldest).
2. If the output of a previous model matches the name of the requested input, return that value.
3. If there were no outputs from previous models, or if they were all checked, look for a field among the inputs that has a name matching the requested input.
4. If the input could not be found, throw an error.

This simple algorithm is implemented in the `esps.utils.fetchField` function, which is also useful for post-processing of the data.

Using this procedure to look for model inputs means we can use the outputs of one model as input for a subsequent model in the same simulation.

### A.3.2 Model input/output mapping

Simulations can alter the input/output process of models by performing input/output mapping in their `getModels` method.

By calling `modelInstance.mapInput("oldName", "newName")` on an instance of a model, we can effectively “rename” an input of the model. Now, when that instance’s `run_` method (described in the previous section) calls `self.dataHandler.getInput("oldInput")`, the value of the input field `newName` will be returned instead.

In a similar vein, outputs of model instances can be renamed with `modelInstance.mapOutput("oldName", "newName")`.

This is useful for example to execute the same model multiple times within the same simulation, but with different outputs each time. See `+esps/+simulations/SimOPEX.m` for a working example.

### A.3.3 Creating a new simulation

Here we will briefly look at the steps required to create a new ESPS simulation:

- Begin by duplicating one of the existing simulations: for this example we will copy `+esps/+simulations/DummySimulation.m` to `+esps/+simulations/MyNewSim.m`.
- Change the class name of the simulation, on line 1 of the new script, to match the file name of the script – in our case `MyNewSim`. It is important that the class name match the file name exactly (this is imposed by MATLAB).
- Change the `simulationName` property to match the class name – in our case `MyNewSim`.
- The `modelNames` property is a string array of model names. Each model name must match the name of a concrete model implementation in `+esps/+models/`.
- The `run_` method is called by the framework when the simulation is executed. Typically, this method amounts to calling `self.getModelNames()`, looping over the resulting model instance array, and calling the `.run()` method on each. If so, consider using `esps.simulations.SequentialSimulation` as your base class rather than `esps.Simulation` directly. See `esps.simulations.SimOPEX` for a working example.
- The `getModelNames` method returns a list of model instances. The type of each instance must match the names declared in the `modelNames` property. This is also where input/output mapping occurs.

Listing A.6 shows an example of a concrete ESPS simulation implementation. Notice that the same could be achieved by using `esps.simulations.SequentialSimulation` as base class rather than `esps.Simulation` directly. See `esps.simulations.SimOPEX` for a working example.

```

1 classdef MyNewSim < esps.Simulation
2     properties(Access=protected)
3         simulationName = "MyNewSim";
4         modelNames = ["MyNewModel", "MyNewModel"];
5     end
6
7     methods(Access=protected)
8         function outData = run_(self, inputData)
9             % Execute each model in sequence
10            outData = inputData;
11            modelInstances = self.getModelNames();
12            for k = 1 : length(modelInstances)

```

```

13         outData = modelInstances{k}.run(outData);
14     end
15 end
16
17     function modelInstances = getModels(self)
18         modelInstances = getModels@esps.Simulation(self);
19
20         % IO mapping for the bike
21         modelInstances{1}.mapInput("velocity", "velocity_bike");
22         modelInstances{1}.mapInput("initialPosition", ...
23 "initialPosition_bike");
24         modelInstances{1}.mapOutput("position", "position_bike");
25         modelInstances{1}.mapOutput("FILEOUT_position", ...
26 "FILEOUT_position_bike");
27
28         % IO mapping for the car
29         modelInstances{2}.mapInput("velocity", "velocity_car");
30         modelInstances{2}.mapInput("initialPosition", ...
31 "initialPosition_car");
32         modelInstances{2}.mapOutput("position", "position_car");
33         modelInstances{2}.mapOutput("FILEOUT_position", ...
34 "FILEOUT_position_car");
35     end
36 end

```

**Listing A.6:** Example of a concrete ESPS simulation implementation.

## A.4 Running simulations with ESPS

### A.4.1 SequentialSimulation

The `SequentialSimulation` is an easy way to quickly test one or more models. The code in Listing A.7 is implemented in `+examples/runMyModel.m`.

```

1 function runMyModel()
2     % Create instance of a simulation object
3     sim = esps.simulations.SequentialSimulation();
4     sim.addModels(["MyNewModel"]);
5     sim.setName("MyNewModelSim");
6
7     % Create an instance of a simulation runner for our simulation
8     runner = esps.runners.SimpleSimulationRunner(sim);
9
10    % Read our input files
11    runner.readInputs("myNewInputs.jsonc", "inputs/myNewInputs/");
12
13    % Run the simulation
14    runner.run();
15

```

```

16     % Write the outputs to file
17     runner.writeOutputs();
18 end

```

**Listing A.7:** Running a SequentialSimulation.

### A.4.2 Custom simulations

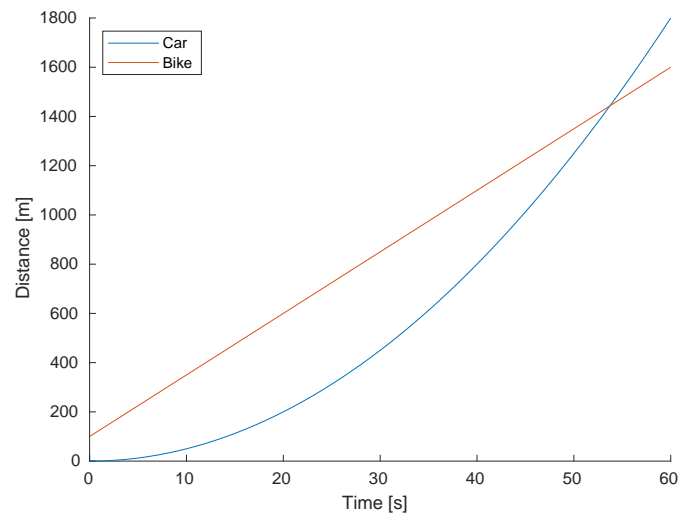
But what if we want to compare a bicycle moving at constant velocity with an accelerating car? We can use the custom simulation we implemented in Listing A.6, which uses the new model twice, with different inputs. Then, plot the results to see the comparison. The code in Listing A.8 is implemented in `+examples/runMySim.m`.

```

1 function runMySim()
2     % Create instance of a simulation object
3     sim = esps.simulations.MyNewSim();
4
5     % Create an instance of a simulation runner for our simulation
6     runner = esps.runners.SimpleSimulationRunner(sim);
7
8     % Read our input files
9     runner.readInputs("myNewInputs.jsonc", "inputs/myNewInputs/");
10
11    % Run the simulation
12    runner.run();
13
14    % Write the outputs to file
15    runner.writeOutputs();
16
17    % Get the results
18    outData = runner.getOutputData();
19    outData = outData{1};
20    time = esps.utils.fetchField(outData, "time");
21    position_car = esps.utils.fetchField(outData, "position_car" );
22    position_bike = esps.utils.fetchField(outData, "position_bike");
23
24    % Plot the results
25    f = figure(); hold on;
26    plot(time, position_car); plot(time, position_bike);
27    legend(["Car", "Bike"], 'Location', 'northwest');
28    ylabel("Distance [m]"); xlabel("Time [s]");
29 end

```

**Listing A.8:** Running a custom simulation and plotting the results.



**Figure A.1:** Result of the script in Listing A.8.



### A.4.3 Running grid searches

To run a grid search, all we need to do is change line 6 of our script to use the `esps.runners.GridSearchRunner` class, and specify our parameter sweeps with `runner.addSearchParam`. `runner.addSearchParam("v", a, b, n)` will generate the following  $n$  grid points spanning the range  $[a; b]$ :

$$v_j = a + \frac{b-a}{(n-1)}j, \quad j = 0, \dots, n-1$$

Listing A.9 shows an example of a one-dimensional grid-search.

```

1 function runMySimGS()
2     % Create instance of a simulation object
3     sim = esps.simulations.MyNewSim();
4
5     % Create an instance of a simulation runner for our simulation
6     runner = esps.runners.GridSearchRunner(sim);
7
8     % Read our input files
9     runner.readInputs("myNewInputs.jsonc", "inputs/myNewInputs/");
10
11    % Set up a sweep of the car's initial position from 0 to 100 with ...
12    10 points
13    runner.addSearchParam("initialPosition-car", 0, 100, 10);
14
15    % Run the simulation
16    runner.run();
17
18    % Write the outputs to file
19    runner.writeOutputs();
20 end

```

**Listing A.9:** Running a one-dimensional grid-search.

### Parallel grid searches

To run a grid search in parallel, simply use `esps.runners.ParallelGridSearch` instead of `esps.runners.GridSearchRunner`. Optionally for particularly large grid-searches, you may wish to use `runner.runAndWrite()`, which is more memory-efficient than `runner.run()` followed by `runner.writeOutputs()`.



---

## Bibliography

---

- [1] Christian Peter. *Renewable Management and Real-Time Control Platform (ReMaP)*. Tech. rep. Bern: Swiss Federal Office of Energy SFOE, Nov. 2021.
- [2] Robin Lehner et al. *Semester Project: Intermediate Energy Storage for Electric Vehicle Fast Charging*. Zürich, Jan. 2019.
- [3] Samuel Renggli, Christian Peter, and Felix Büchi. *Prosumage system in a peak shaving application*. Zürich, Jan. 2020.
- [4] Mario Heer et al. *Optimal Design of a Prosumage (Producer and Consumer) System in a Peak Shaving Application*. Zürich, July 2021.
- [5] MIT Electric Vehicle Team. *A Guide to Understanding Battery Specifications*. Dec. 2008. URL: [http://web.mit.edu/evt/summary\\_battery\\_specifications.pdf](http://web.mit.edu/evt/summary_battery_specifications.pdf).
- [6] Jonghyeon Kim and Julia Kowal. “Development of a Matlab/Simulink Model for Monitoring Cell State-of-Health and State-of-Charge via Impedance of Lithium-Ion Battery Cells”. In: *Batteries* 8.2 (2022). ISSN: 2313-0105. DOI: [10.3390/batteries8020008](https://doi.org/10.3390/batteries8020008). URL: <https://www.mdpi.com/2313-0105/8/2/8>.
- [7] M. Koller. *Batteriespeicher*. EKZ, 2018.
- [8] Wesley J Cole and Allister Frazier. *Cost Projections for Utility-Scale Battery Storage*. Tech. rep. Golden, CO (United States): National Renewable Energy Laboratory (NREL), June 2019. DOI: [10.2172/1529218](https://doi.org/10.2172/1529218).
- [9] Charlotte van Leeuwen and Andreas Zauner. *Innovative large-scale energy storage technologies and Power-to-Gas concepts after optimisation*. Tech. rep. Store&GO, Apr. 2018.
- [10] V Contini et al. *Stationary and Emerging Market Fuel Cell System Cost Analysis*. Tech. rep. Columbus, OH: Battelle / DOE, 2016.
- [11] G. Kleen and E. Padgett. *Durability-Adjusted Fuel Cell System Cost*. Tech. rep. DOE, 2021.

- [12] O. Schmidt et al. "Future cost and performance of water electrolysis: An expert elicitation study". In: *International Journal of Hydrogen Energy* 42.52 (Dec. 2017), pp. 30470–30492. ISSN: 03603199. DOI: [10.1016/j.ijhydene.2017.10.045](https://doi.org/10.1016/j.ijhydene.2017.10.045).
- [13] IRENA. *Green Hydrogen Cost Reduction: Scaling up Electrolysers to Meet the 1.5C Climate Goal*. Abu Dhabi, 2020.
- [14] BKW. *Preisinformation für Gross- und Industriekunden*. Bern, Jan. 2022.