emhass

Release 0.9.0

David HERNANDEZ

May 10, 2024

CONTENTS:

Welcome to the documentation of EMHASS. With this package written in Python you will be able to implement a real Energy Management System for your household. This software was designed to be easy configurable and with a fast integration with Home Assistant: https://www.home-assistant.io/

To get started go ahead and look at the installation procedure and usage instructions below.

CHAPTER

ONE

INTRO / QUICK START

EHMASS is a Python module designed to optimize your home energy interfacing with Home Assistant.

1.1 Introduction

EMHASS (Energy Management for Home Assistant) is an optimization tool designed for residential households. The package uses a Linear Programming approach to optimize energy usage while considering factors such as electricity prices, power generation from solar panels, and energy storage from batteries. EMHASS provides a high degree of configurability, making it easy to integrate with Home Assistant and other smart home systems. Whether you have solar panels, energy storage, or just a controllable load, EMHASS can provide an optimized daily schedule for your devices, allowing you to save money and minimize your environmental impact.

The complete documentation for this package is available here.

1.2 What is Energy Management for Home Assistant (EMHASS)?

EMHASS and Home Assistant provide a comprehensive energy management solution that can optimize energy usage and reduce costs for households. By integrating these two systems, households can take advantage of advanced energy management features that provide significant cost savings, increased energy efficiency, and greater sustainability.

EMHASS is a powerful energy management tool that generates an optimization plan based on variables such as solar power production, energy usage, and energy costs. The plan provides valuable insights into how energy can be better managed and utilized in the household. Even if households do not have all the necessary equipment, such as solar panels or batteries, EMHASS can still provide a minimal use case solution to optimize energy usage for controllable/deferrable loads.

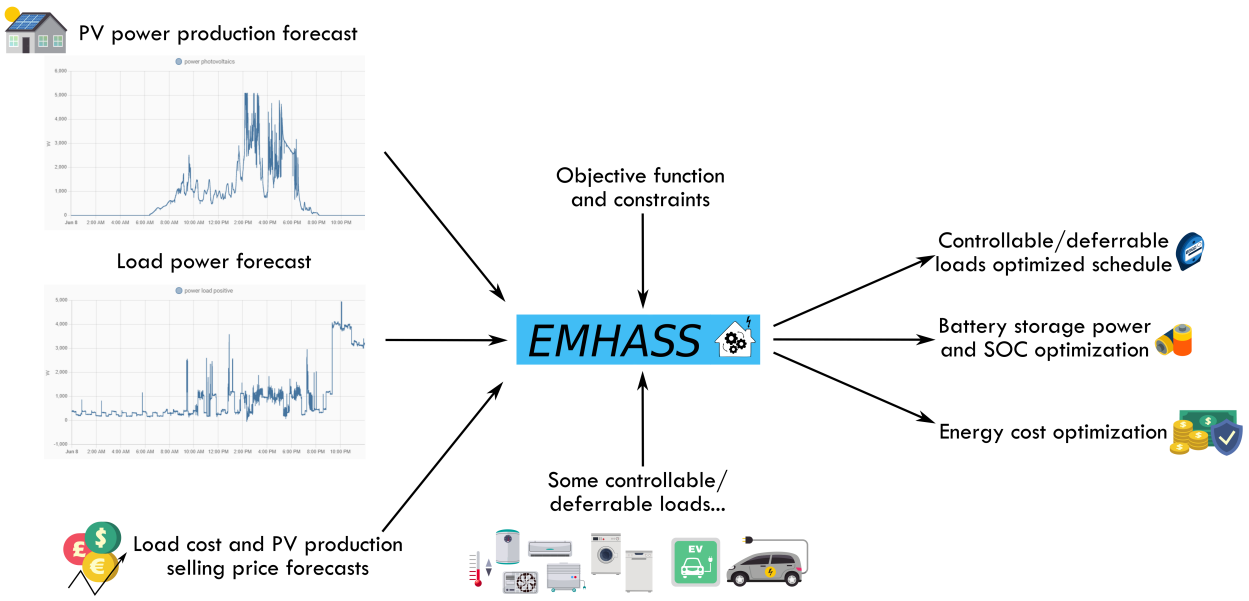
Home Assistant provides a platform for the automation of household devices based on the optimization plan generated by EMHASS. This includes devices such as batteries, pool pumps, hot water heaters, and electric vehicle (EV) chargers. By automating EV charging and other devices, households can take advantage of off-peak energy rates and optimize their EV charging schedule based on the optimization plan generated by EMHASS.

One of the main benefits of integrating EMHASS and Home Assistant is the ability to customize and tailor the energy management solution to the specific needs and preferences of each household. With EMHASS, households can define their energy management objectives and constraints, such as maximizing self-consumption or minimizing energy costs, and the system will generate an optimization plan accordingly. Home Assistant provides a platform for the automation of devices based on the optimization plan, allowing households to create a fully customized and optimized energy management solution.

Overall, the integration of EMHASS and Home Assistant offers a comprehensive energy management solution that provides significant cost savings, increased energy efficiency, and greater sustainability for households. By leveraging advanced energy management features and automation capabilities, households can achieve their energy management

objectives while enjoying the benefits of a more efficient and sustainable energy usage, including optimized EV charging schedules.

The package flow can be graphically represented as follows:



1.3 Configuration and Installation

The package is meant to be highly configurable with an object oriented modular approach and a main configuration file defined by the user. EMHASS was designed to be integrated with Home Assistant, hence it's name. Installation instructions and example Home Assistant automation configurations are given below.

You must follow these steps to make EMHASS work properly:

1) Define all the parameters in the configuration file according to your installation. See the description for each parameter in the **configuration** section.

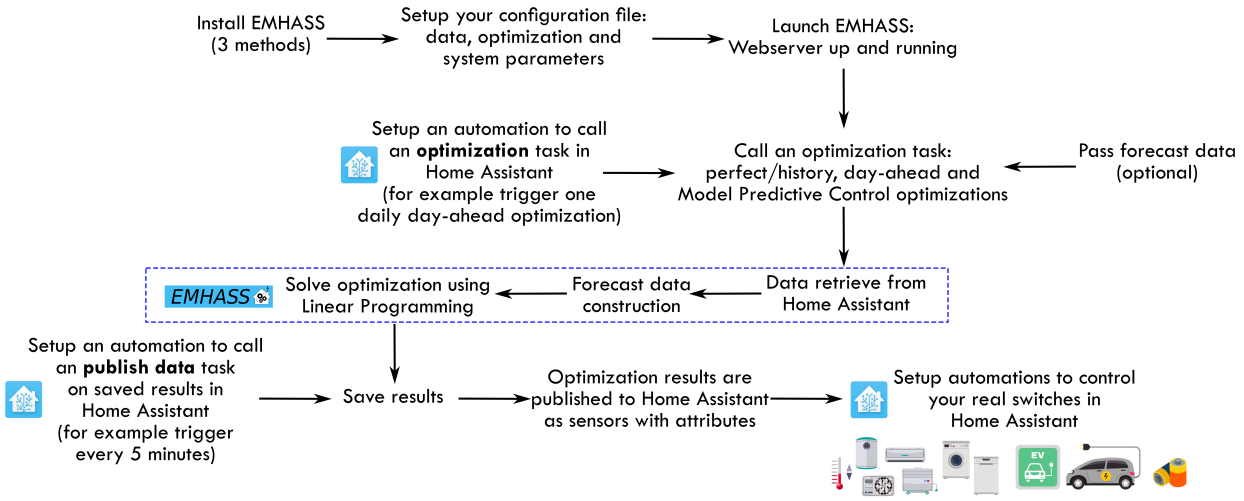
2) You most notably will need to define the main data entering EMHASS. This will be the `sensor.power_photovoltaics` for the name of the your hass variable containing the PV produced power and the variable `sensor.power_load_no_var_loads` for the load power of your household excluding the power of the deferrable loads that you want to optimize.

3) Launch the actual optimization and check the results. This can be done manually using the buttons in the web ui or with a `curl` command like this: `curl -i -H 'Content-Type:application/json' -X POST -d '{}' http://localhost:5000/action/dayahead-optim`.

4) If you're satisfied with the optimization results then you can set the optimization and data publish task commands in an automation. You can read more about this on the **usage** section below.

5) The final step is to link the deferrable loads variables to real switchs on your installation. An example code for this using automations and the shell command integration is presented below in the **usage** section.

A more detailed workflow is given below:



1.3.1 Method 1) The EMHASS add-on for Home Assistant OS and supervised users

For Home Assistant OS and HA Supervised users, I've developed an add-on that will help you use EMHASS. The add-on is more user friendly as the configuration can be modified directly in the add-on options pane and as with the standalone docker it exposes a web ui that can be used to inspect the optimization results and manually trigger a new optimization.

You can find the add-on with the installation instructions here: https://github.com/davidusb-geek/emhass-add-on

The add-on usage instructions can be found on the documentation pane of the add-on once installed or directly here: EMHASS Add-on documentation

These architectures are supported: `amd64`, `armv7`, `armhf` and `aarch64`.

1.3.2 Method 2) Using Docker in standalone mode

You can also install EMHASS using docker. This can be in the same machine as Home Assistant (if using the supervised install method) or in a different distant machine. To install first pull the latest image from docker hub:

```
docker pull davidusb/emhass-docker-standalone
```

You can also build your image locally. For this clone this repository, setup your `config_emhass.yaml` file and use the provided make file with this command:

```
make -f deploy_docker.mk clean_deploy
```

Then load the image in the .tar file:

```
docker load -i <TarFileName>.tar
```

Finally check your image tag with `docker images` and launch the docker itself:

```
docker run -it --restart always -p 5000:5000 -e LOCAL_COSTFUN="profit" -v $(pwd)/config_
↪emhass.yaml:/app/config_emhass.yaml -v $(pwd)/secrets_emhass.yaml:/app/secrets_emhass.
↪yaml --name DockerEMHASS <REPOSITORY:TAG>
```

- If you wish to keep a local, persistent copy of the EMHASS generated data, create a local folder on your device, then mount said folder inside the container.

```
mkdir -p $(pwd)/data #linux: create data folder on local device

docker run -it --restart always -p 5000:5000 -e LOCAL_COSTFUN="profit" -v $(pwd)/
↪config_emhass.yaml:/app/config_emhass.yaml -v $(pwd)/data:/app/data  -v $(pwd)/
↪secrets_emhass.yaml:/app/secrets_emhass.yaml --name DockerEMHASS <REPOSITORY:TAG>
```

If you wish to set the web_server's diagrams to a timezone other than UTC, set TZ environment variable on:

```
docker run -it --restart always -p 5000:5000  -e TZ="Europe/Paris"  -e LOCAL_COSTFUN=
↪"profit" -v $(pwd)/config_emhass.yaml:/app/config_emhass.yaml -v $(pwd)/secrets_emhass.
↪yaml:/app/secrets_emhass.yaml --name DockerEMHASS <REPOSITORY:TAG>
```

1.3.3 Method 3) Legacy method using a Python virtual environment

With this method it is recommended to install on a virtual environment. For this you will need `virtualenv`, install it using:

```
sudo apt install python3-virtualenv
```

Then create and activate the virtual environment:

```
virtualenv -p /usr/bin/python3 emhassenv
cd emhassenv
source bin/activate
```

Install using the distribution files:

```
python3 -m pip install emhass
```

Clone this repository to obtain the example configuration files. We will suppose that this repository is cloned to:

```
/home/user/emhass
```

This will be the root path containing the yaml configuration files (`config_emhass.yaml` and `secrets_emhass.yaml`) and the different needed folders (a `data` folder to store the optimizations results and a `scripts` folder containing the bash scripts described further below).

To upgrade the installation in the future just use:

```
python3 -m pip install --upgrade emhass
```

1.4 Usage

1.4.1 Method 1) Add-on and docker standalone

If using the add-on or the standalone docker installation, it exposes a simple webserver on port 5000. You can access it directly using your brower, ex: http://localhost:5000.

With this web server you can perform RESTful POST commands on multiple ENDPOINTS with prefix `action/*`:

- A POST call to `action/perfect-optim` to perform a perfect optimization task on the historical data.
- A POST call to `action/dayahead-optim` to perform a day-ahead optimization task of your home energy.

- A POST call to `action/naive-mpc-optim` to perform a naive Model Predictive Controller optimization task. If using this option you will need to define the correct `runtimeparams` (see further below).
- A POST call to `action/publish-data` to publish the optimization results data for the current timestamp.
- A POST call to `action/forecast-model-fit` to train a machine learning forecaster model with the passed data (see the dedicated section for more help).
- A POST call to `action/forecast-model-predict` to obtain a forecast from a pre-trained machine learning forecaster model (see the dedicated section for more help).
- A POST call to `action/forecast-model-tune` to optimize the machine learning forecaster models hyperparameters using bayesian optimization (see the dedicated section for more help).

A `curl` command can then be used to launch an optimization task like this: `curl -i -H 'Content-Type:application/json' -X POST -d '{}' http://localhost:5000/action/dayahead-optim`.

1.4.2 Method 2) Legacy method using a Python virtual environment

To run a command simply use the `emhass` CLI command followed by the needed arguments. The available arguments are:

- `--action`: That is used to set the desired action, options are: `perfect-optim`, `dayahead-optim`, `naive-mpc-optim`, `publish-data`, `forecast-model-fit`, `forecast-model-predict` and `forecast-model-tune`.
- `--config`: Define path to the config.yaml file (including the yaml file itself)
- `--costfun`: Define the type of cost function, this is optional and the options are: `profit` (default), `cost`, `self-consumption`
- `--log2file`: Define if we should log to a file or not, this is optional and the options are: `True` or `False` (default)
- `--params`: Configuration as JSON.
- `--runtimeparams`: Data passed at runtime. This can be used to pass your own forecast data to EMHASS.
- `--debug`: Use `True` for testing purposes.
- `--version`: Show the current version of EMHASS.

For example, the following line command can be used to perform a day-ahead optimization task:

```
emhass --action 'dayahead-optim' --config '/home/user/emhass/config_emhass.yaml' --
↪costfun 'profit'
```

Before running any valuable command you need to modify the `config_emhass.yaml` and `secrets_emhass.yaml` files. These files should contain the information adapted to your own system. To do this take a look at the special section for this in the documentation.

1.5 Home Assistant integration

To integrate with home assistant we will need to define some shell commands in the `configuration.yaml` file and some basic automations in the `automations.yaml` file. In the next few paragraphs we are going to consider the `dayahead-optim` optimization strategy, which is also the first that was implemented, and we will also cover how to publish the results. Then additional optimization strategies were developed, that can be used in combination with/replace the `dayahead-optim` strategy, such as MPC, or to expland the funcitonalities such as the Machine Learning method to predict your hosehold consumption. Each of them has some specificities and features and will be considered in dedicated sections.

1.5.1 Dayahead Optimization - Method 1) Add-on and docker standalone

In `configuration.yaml`:

```
shell_command:
  dayahead_optim: "curl -i -H \"Content-Type:application/json\" -X POST -d '{}' http://
↪localhost:5000/action/dayahead-optim"
  publish_data: "curl -i -H \"Content-Type:application/json\" -X POST -d '{}' http://
↪localhost:5000/action/publish-data"
```

1.5.2 Dayahead Optimization - Method 2) Legacy method using a Python virtual environment

In `configuration.yaml`:

```
shell_command:
  dayahead_optim: /home/user/emhass/scripts/dayahead_optim.sh
  publish_data: /home/user/emhass/scripts/publish_data.sh
```

Create the file `dayahead_optim.sh` with the following content:

```
#!/bin/bash
. /home/user/emhassenv/bin/activate
emhass --action 'dayahead-optim' --config '/home/user/emhass/config_emhass.yaml'
```

And the file `publish_data.sh` with the following content:

```
#!/bin/bash
. /home/user/emhassenv/bin/activate
emhass --action 'publish-data' --config '/home/user/emhass/config_emhass.yaml'
```

Then specify user rights and make the files executables:

```
sudo chmod -R 755 /home/user/emhass/scripts/dayahead_optim.sh
sudo chmod -R 755 /home/user/emhass/scripts/publish_data.sh
sudo chmod +x /home/user/emhass/scripts/dayahead_optim.sh
sudo chmod +x /home/user/emhass/scripts/publish_data.sh
```

1.5.3 Common for any installation method

In `automations.yaml`:

```
- alias: EMHASS day-ahead optimization
  trigger:
    platform: time
    at: '05:30:00'
  action:
  - service: shell_command.dayahead_optim
- alias: EMHASS publish data
  trigger:
  - minutes: /5
    platform: time_pattern
  action:
  - service: shell_command.publish_data
```

In these automations the day-ahead optimization is performed everyday at 5:30am and the data is published every 5 minutes.

The final action will be to link a sensor value in Home Assistant to control the switch of a desired controllable load. For example imagine that I want to control my water heater and that the `publish-data` action is publishing the optimized value of a deferrable load that I want to be linked to my water heater desired behavior. In this case we could use an automation like this one below to control the desired real switch:

```
automation:
- alias: Water Heater Optimized ON
  trigger:
  - minutes: /5
    platform: time_pattern
  condition:
  - condition: numeric_state
    entity_id: sensor.p_deferrable0
    above: 0.1
  action:
    - service: homeassistant.turn_on
      entity_id: switch.water_heater_switch
```

A second automation should be used to turn off the switch:

```
automation:
- alias: Water Heater Optimized OFF
  trigger:
  - minutes: /5
    platform: time_pattern
  condition:
  - condition: numeric_state
    entity_id: sensor.p_deferrable0
    below: 0.1
  action:
    - service: homeassistant.turn_off
      entity_id: switch.water_heater_switch
```

1.6 The publish-data specificities

The `publish-data` command will push to Home Assistant the optimization results for each deferrable load defined in the configuration. For example if you have defined two deferrable loads, then the command will publish `sensor.p_deferrable0` and `sensor.p_deferrable1` to Home Assistant. When the `dayahead-optim` is launched, after the optimization, a csv file will be saved on disk. The `publish-data` command will load the latest csv file and look for the closest timestamp that match the current time using the `datetime.now()` method in Python. This means that if EMHASS is configured for 30min time step optimizations, the csv will be saved with timestamps 00:00, 00:30, 01:00, 01:30, ... and so on. If the current time is 00:05, then the closest timestamp of the optimization results that will be published is 00:00. If the current time is 00:25, then the closest timestamp of the optimization results that will be published is 00:30.

The `publish-data` command will also publish PV and load forecast data on sensors `p_pv_forecast` and `p_load_forecast`. If using a battery, then the battery optimized power and the SOC will be published on sensors `p_batt_forecast` and `soc_batt_forecast`. On these sensors the future values are passed as nested attributes.

It is possible to provide custm sensor names for all the data exported by the `publish-data` command. For this, when using the `publish-data` endpoint just add some runtime parameters as dictionaries like this:

```
shell_command:
  publish_data: "curl -i -H \"Content-Type:application/json\" -X POST -d '{\"custom_load_
↪forecast_id\": {\"entity_id\": \"sensor.p_load_forecast\", \"unit_of_measurement\": \
↪"W\", \"friendly_name\": \"Load Power Forecast\"}}' http://localhost:5000/action/
↪publish-data"
```

These keys are available to modify: `custom_pv_forecast_id`, `custom_load_forecast_id`, `custom_batt_forecast_id`, `custom_batt_soc_forecast_id`, `custom_grid_forecast_id`, `custom_cost_fun_id`, `custom_deferrable_forecast_id`, `custom_unit_load_cost_id` and `custom_unit_prod_price_id`.

If you provide the `custom_deferrable_forecast_id` then the passed data should be a list of dictionaries, like this:

```
shell_command:
  publish_data: "curl -i -H \"Content-Type:application/json\" -X POST -d '{\"custom_
↪deferrable_forecast_id\": [{\"entity_id\": \"sensor.p_deferrable0\",\"unit_of_
↪measurement\": \"W\", \"friendly_name\": \"Deferrable Load 0\"},{\"entity_id\": \
↪"sensor.p_deferrable1\",\"unit_of_measurement\": \"W\", \"friendly_name\": \
↪"Deferrable Load 1\"}]}' http://localhost:5000/action/publish-data"
```

And you should be careful that the list of dictionaries has the correct length, which is the number of defined deferrable loads.

1.6.1 Computed variables and published data

Below you can find a list of the variables resulting from EMHASS computation, shown in the charts and published to Home Assistant through the `publish_data` command:

EMHAS variable	Definition	Home Assistant published sensor
P_PV	Forecasted power generation from your solar panels (Watts). This helps you predict how much solar energy you will produce during the forecast period.	sensor.p_pv_forecast
P_Load	Forecasted household power consumption (Watts). This gives you an idea of how much energy your appliances are expected to use.	sensor.p_load_forecast
P_defer = 0, 1, 2, ...]	Forecasted power consumption of deferrable loads (Watts). Deferable loads are appliances that can be managed by EMHASS. EMHASS helps you optimise energy usage by prioritising solar self-consumption and minimizing reliance on the grid or by taking advantage or supply and feed-in tariff volatility. You can have multiple deferable loads and you use this sensor in HA to control these loads via smart switch or other IoT means at your disposal.	sensor.p_deferrableX
P_grid_	Forecasted power imported from the grid (Watts). This indicates the amount of energy you are expected to draw from the grid when your solar production is insufficient to meet your needs or it is advantagous to consume from the grid.	-
P_grid_	Forecasted power exported to the grid (Watts). This indicates the amount of excess solar energy you are expected to send back to the grid during the forecast period.	-
P_batt	Forecasted (dis)charge power load (Watts) for the battery (if installed). If negative it indicates the battery is charging, if positive that the battery is discharging.	sensor.p_batt_forecast
P_grid	Forecasted net power flow between your home and the grid (Watts). This is calculated as P_grid_pos - P_grid_neg. A positive value indicates net export, while a negative value indicates net import.	sensor.p_grid_forecast
SOC_op	Forecasted battery optimized Status Of Charge (SOC) percentage level	sensor.soc_batt_forecast
unit_loa	Forecasted cost per unit of energy you pay to the grid (typically "Currency"/kWh). This helps you understand the expected energy cost during the forecast period.	sensor.unit_load_cost
unit_pro	Forecasted price you receive for selling excess solar energy back to the grid (typically "Currency"/kWh). This helps you understand the potential income from your solar production.	sensor.unit_prod_price
cost_pro	Forecasted profit or loss from your energy usage for the forecast period. This is calculated as unit_load_cost * P_Load - unit_prod_price * P_grid_pos. A positive value indicates a profit, while a negative value indicates a loss.	sensor.total_cost_profit_value
cost_fun	Forecasted cost associated with deferring loads to maximize solar self-consumption. This helps you evaluate the trade-off between managing the load and not managing and potential cost savings.	sensor.total_cost_fun_value
optim_stat	This contains the status of the latest execution and is the same you can see in the Log following an optimization job. Its values can be Optimal or Infeasible.	sensor.optim_status

1.7 Passing your own data

In EMHASS we have basically 4 forecasts to deal with:

- PV power production forecast (internally based on the weather forecast and the characteristics of your PV plant). This is given in Watts.
- Load power forecast: how much power your house will demand on the next 24h. This is given in Watts.
- Load cost forecast: the price of the energy from the grid on the next 24h. This is given in EUR/kWh.
- PV production selling price forecast: at what price are you selling your excess PV production on the next 24h. This is given in EUR/kWh.

The sensor containing the load data should be specified in parameter `var_load` in the configuration file. As we want to optimize the household energies, when need to forecast the load power conumption. The default method for this is a naive approach using 1-day persistence. The load data variable should not contain the data from the deferrable loads themselves. For example, lets say that you set your deferrable load to be the washing machine. The variable that you should enter in EMHASS will be: `var_load: 'sensor.power_load_no_var_loads'` and `sensor.power_load_no_var_loads = sensor.power_load - sensor.power_washing_machine`. This is supposing that the overall load of your house is contained in variable: `sensor.power_load`. The sensor `sensor.power_load_no_var_loads` can be easily created with a new template sensor in Home Assistant.

If you are implementing a MPC controller, then you should also need to provide some data at the optimization runtime using the key `runtimeparams`.

The valid values to pass for both forecast data and MPC related data are explained below.

1.7.1 Forecast data

It is possible to provide EMHASS with your own forecast data. For this just add the data as list of values to a data dictionary during the call to `emhass` using the `runtimeparams` option.

For example if using the add-on or the standalone docker installation you can pass this data as list of values to the data dictionary during the `curl` POST:

```
curl -i -H 'Content-Type:application/json' -X POST -d '{"pv_power_forecast":[0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 70, 141.22, 246.18, 513.5, 753.27, 1049.89, 1797.
↪93, 1697.3, 3078.93, 1164.33, 1046.68, 1559.1, 2091.26, 1556.76, 1166.73, 1516.63,␣
↪1391.13, 1720.13, 820.75, 804.41, 251.63, 79.25, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}' http:/
↪/localhost:5000/action/dayahead-optim
```

Or if using the legacy method using a Python virtual environment:

```
emhass --action 'dayahead-optim' --config '/home/user/emhass/config_emhass.yaml' --
↪runtimeparams '{"pv_power_forecast":[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪70, 141.22, 246.18, 513.5, 753.27, 1049.89, 1797.93, 1697.3, 3078.93, 1164.33, 1046.68,
↪ 1559.1, 2091.26, 1556.76, 1166.73, 1516.63, 1391.13, 1720.13, 820.75, 804.41, 251.63,␣
↪79.25, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}'
```

The possible dictionary keys to pass data are:

- `pv_power_forecast` for the PV power production forecast.
- `load_power_forecast` for the Load power forecast.
- `load_cost_forecast` for the Load cost forecast.
- `prod_price_forecast` for the PV production selling price forecast.

1.7.2 Passing other data

It is possible to also pass other data during runtime in order to automate the energy management. For example, it could be useful to dynamically update the total number of hours for each deferrable load (`def_total_hours`) using for instance a correlation with the outdoor temperature (useful for water heater for example).

Here is the list of the other additional dictionary keys that can be passed at runtime:

- `num_def_loads` for the number of deferrable loads to consider.
- `P_deferrable_nom` for the nominal power for each deferrable load in Watts.
- `def_total_hours` for the total number of hours that each deferrable load should operate.
- `def_start_timestep` for the timestep as from which each deferrable load is allowed to operate (if you don't want the deferrable load to use the whole optimization timewindow).
- `def_end_timestep` for the timestep before which each deferrable load should operate (if you don't want the deferrable load to use the whole optimization timewindow).
- `treat_def_as_semi_cont` to define if we should treat each deferrable load as a semi-continuous variable.
- `set_def_constant` to define if we should set each deferrable load as a constant fixed value variable with just one startup for each optimization task.
- `solcast_api_key` for the SolCast API key if you want to use this service for PV power production forecast.
- `solcast_rooftop_id` for the ID of your rooftop for the SolCast service implementation.
- `solar_forecast_kwp` for the PV peak installed power in kW used for the solar.forecast API call.
- `SOCtarget` for the desired target value of initial and final SOC.
- `publish_prefix` use this key to pass a common prefix to all published data. This will add a prefix to the sensor name but also to the forecasts attributes keys within the sensor.

1.8 A naive Model Predictive Controller

A MPC controller was introduced in v0.3.0. This is an informal/naive representation of a MPC controller. This can be used in combination with/as a replacement of the Dayahead Optimization.

A MPC controller performs the following actions:

- Set the prediction horizon and receding horizon parameters.
- Perform an optimization on the prediction horizon.
- Apply the first element of the obtained optimized control variables.
- Repeat at a relatively high frequency, ex: 5 min.

This is the receding horizon principle.

When applying this controller, the following `runtimeparams` should be defined:

- `prediction_horizon` for the MPC prediction horizon. Fix this at at least 5 times the optimization time step.
- `soc_init` for the initial value of the battery SOC for the current iteration of the MPC.
- `soc_final` for the final value of the battery SOC for the current iteration of the MPC.
- `def_total_hours` for the list of deferrable loads functioning hours. These values can decrease as the day advances to take into account receding horizon daily energy objectives for each deferrable load.

- `def_start_timestep` for the timestep as from which each deferrable load is allowed to operate (if you don't want the deferrable load to use the whole optimization timewindow). If you specify a value of 0 (or negative), the deferrable load will be optimized as from the beginning of the complete prediction horizon window.
- `def_end_timestep` for the timestep before which each deferrable load should operate (if you don't want the deferrable load to use the whole optimization timewindow). If you specify a value of 0 (or negative), the deferrable load optimization window will extend up to the end of the prediction horizon window.

A correct call for a MPC optimization should look like:

```
curl -i -H 'Content-Type:application/json' -X POST -d '{"pv_power_forecast":[0, 70, 141.
↪22, 246.18, 513.5, 753.27, 1049.89, 1797.93, 1697.3, 3078.93], "prediction_horizon":10,
↪ "soc_init":0.5,"soc_final":0.6}' http://192.168.3.159:5000/action/naive-mpc-optim
```

Example with :`def_total_hours`, `def_start_timestep`, `def_end_timestep`.

```
curl -i -H 'Content-Type:application/json' -X POST -d '{"pv_power_forecast":[0, 70, 141.
↪22, 246.18, 513.5, 753.27, 1049.89, 1797.93, 1697.3, 3078.93], "prediction_horizon":10,
↪ "soc_init":0.5,"soc_final":0.6,"def_total_hours":[1,3],"def_start_timestep":[0,3],
↪"def_end_timestep":[0,6]}' http://localhost:5000/action/naive-mpc-optim
```

1.9 A machine learning forecaster

Starting in v0.4.0 a new machine learning forecaster class was introduced. This is intended to provide a new and alternative method to forecast your household consumption and use it when such forecast is needed to optimize your energy through the available strategies. Check the dedicated section in the documentation here: https://emhass.readthedocs.io/en/latest/mlforecaster.html

1.10 Development

Pull request are very much accepted on this project. For development you can find some instructions here Development

1.11 Troubleshooting

Some problems may arise from solver related issues in the Pulp package. It was found that for arm64 architectures (ie. Raspberry Pi4, 64 bits) the default solver is not avaliable. A workaround is to use another solver. The `glpk` solver is an option.

This can be controlled in the configuration file with parameters `lp_solver` and `lp_solver_path`. The options for `lp_solver` are: 'PULP_CBC_CMD', 'GLPK_CMD' and 'COIN_CMD'. If using 'COIN_CMD' as the solver you will need to provide the correct path to this solver in parameter `lp_solver_path`, ex: '/usr/bin/cbc'.

1.12 License

CHAPTER

TWO

EMHASS & EMHASS-ADD-ON DIFFERENCES

User will pass parameters into EMHASS differently, based on running *Standalone* mode or *addon* Mode.
This page tries to help to resolve the common confusion between the two.
Its best to see EMHASS-Add-on as a Home Assistant Docker wrapper for EMHASS. However, because of this containerization, certain changes are made between the two modes.

2.1 Configuration & parameter differences

Both EMHASS & EMHASS-Add-on utilize `config_emhass.yaml` for receiving parameters.
Where they diverge is EMHASS-Add-ons additional use of `options.json`, generated by Home Assistants `Configuration Page`.
Any passed parameters given in `options.json` will overwrite the parameters hidden in the `config_emhass.yaml` file in EMHASS. *(this results in `config_emhass.yaml` used for parameter default fall back if certain required parameters were missing in `options.json`)*

The parameters naming convention has also been changed in `options.json`, designed to make it easier for the user to understand.

See bellow for a list of associations between the parameters from `config_emhass.yaml` and `options.json`:
You can view the current parameter differences in the `Utils.py` file under the `build_params` function.

config	config_emhass.yaml	options.json	options.json li
retrieve_hass_conf	freq	optimization_time_step	
retrieve_hass_conf	days_to_retrieve	historic_days_to_retrieve	
retrieve_hass_conf	var_PV	sensor_power_photovoltaics	
retrieve_hass_conf	var_load	sensor_power_load_no_var_loads	
retrieve_hass_conf	load_negative	load_negative	
retrieve_hass_conf	set_zero_min	set_zero_min	
retrieve_hass_conf	method_ts_round	method_ts_round	
params_secrets	solcast_api_key	optional_solcast_api_key	
params_secrets	solcast_rooftop_id	optional_solcast_rooftop_id	
params_secrets	solar_forecast_kwp	optional_solar_forecast_kwp	
params_secrets	time_zone	time_zone	
params_secrets	lat	Latitude	
params_secrets	lon	Longitude	
params_secrets	alt	Altitude	
optim_conf	set_use_battery	set_use_battery	
optim_conf	num_def_loads	number_of_deferrable_loads	
optim_conf	P_deferrable_nom	list_nominal_power_of_deferrable_loads	nominal_power

Table 1 – continued from previous page

config	config_emhass.yaml	options.json	options.json li
optim_conf	def_total_hours	list_operating_hours_of_each_deferrable_load	operating_hour
optim_conf	treat_def_as_semi_cont	list_treat_deferrable_load_as_semi_cont	treat_deferrabl
optim_conf	set_def_constant	list_set_deferrable_load_single_constant	set_deferrable_
optim_conf	weather_forecast_method	weather_forecast_method	
optim_conf	load_forecast_method	load_forecast_method	
optim_conf	delta_forecast	delta_forecast_daily	
optim_conf	load_cost_forecast_method	load_cost_forecast_method	
optim_conf	load_cost_hp	load_peak_hours_cost	
optim_conf	load_cost_hc	load_offpeak_hours_cost	
optim_conf	prod_price_forecast_method	production_price_forecast_method	
optim_conf	prod_sell_price	photovoltaic_production_sell_price	
optim_conf	set_total_pv_sell	set_total_pv_sell	
optim_conf	lp_solver	lp_solver	
optim_conf	lp_solver_path	lp_solver_path	
optim_conf	set_nocharge_from_grid	set_nocharge_from_grid	
optim_conf	set_nodischarge_to_grid	set_nodischarge_to_grid	
optim_conf	set_battery_dynamic	set_battery_dynamic	
optim_conf	battery_dynamic_max	battery_dynamic_max	
optim_conf	battery_dynamic_min	battery_dynamic_min	
optim_conf	weight_battery_discharge	weight_battery_discharge	
optim_conf	weight_battery_charge	weight_battery_charge	
optim_conf	def_start_timestep	list_start_timesteps_of_each_deferrable_load	start_timesteps
optim_conf	def_end_timestep	list_end_timesteps_of_each_deferrable_load	end_timesteps_
plant_conf	P_grid_max	maximum_power_from_grid	
plant_conf	module_model	list_pv_module_model	pv_module_mo
plant_conf	inverter_model	list_pv_inverter_model	pv_inverter_mo
plant_conf	surface_tilt	list_surface_tilt	surface_tilt
plant_conf	surface_azimuth	list_surface_azimuth	surface_azimut
plant_conf	modules_per_string,list_modules_per_string	modules_per_string	
plant_conf	strings_per_inverter	list_strings_per_inverter	strings_per_inv
plant_conf	Pd_max	battery_discharge_power_max	
plant_conf	Pc_max	battery_charge_power_max	
plant_conf	eta_disch	battery_discharge_efficiency	
plant_conf	eta_ch	battery_charge_efficiency	
plant_conf	Enom	battery_nominal_energy_capacity	
plant_conf	SOCmin	battery_minimum_state_of_charge	
plant_conf	SOCmax	battery_maximum_state_of_charge	
plant_conf	SOCtarget	battery_target_state_of_charge	

Descriptions of each parameter, can be found at:

- `Configuration file` on EMHASS
- `en.yaml` on EMHASS-Add-on

2.2 Passing in secret parameters

Secret parameters get passed differently, depending on which mode you choose. Alternative options are also present for passing secrets if running EMHASS separately from Home Assistant. *(I.e. not via EMHASS-Add-on)*

2.2.1 EMHASS (with standalone mode)

Running EMHASS in standalone mode's default workflow retrieves all secret parameters via a passed `secrets_emhass.yaml` file. An example template has been provided under the name `secrets_emhass(example).yaml`.

Alternative Options

For users who are running EMHASS with methods other than EMHASS-Add-on, secret parameters can be passed with the use of arguments and/or environment variables. *(instead of `secrets_emhass.yaml`)*

Some arguments include: `--url` and `--key`
Some environment variables include: `TIME_ZONE`, `LAT` , `LON`, `ALT`, `EMHASS_URL`, `EMHASS_KEY`

Note: As of writing, EMHASS standalone will override ARG/ENV secret parameters if file is present.

For more information on passing arguments and environment variables using docker, have a look at some examples from Configuration and Installation and EMHASS Development pages.

2.2.2 EMHASS-Add-on (addon mode)

By default the `URL` and `KEY` parameters have been set to `empty`/blank. This results in EMHASS calling to its Supervisor API to gain access locally. This is the easiest method, as there is no user input necessary.

However, if you wish to receive/send sensor data to a different Home Assistant environment, set url and key values in the `hass_url` & `long_lived_token` hidden parameters.

- `hass_url` example: `https://192.168.1.2:8123/`
- `long_lived_token` generated from the `Long-lived access tokens` section in your user profile settings

Secret Parameters such as: `time_zone`, `lon`, `lat` and `alt` are also automatically passed in via the Home Assistants environment. *(Values set in the Home Assistants config/general page)*
Note: Local currency could also be obtained via the Home Assistant environment, however as of writing, this functionality has not yet been developed.

Secret Parameters such as: `solcast_api_key`, `solcast_rooftop_id` and `solar_forecast_kwp` *(used by their respective `weather_forecast_method` parameter values)*, can also be set via hidden parameters in the configuration page.

CHAPTER

THREE

AN EMS BASED ON LINEAR PROGRAMMING

In this section we present the basics of the Linear Programming (LP) approach for a household Energy Management System (EMS).

3.1 Motivation

Imagine that we have installed some solar panels in our house. Imagine that we have Home Assistant and that we can control (on/off) some crucial power consumptions in our home. For example the water heater, the pool pump, a dispatchable dishwasher, and so on. We can also imagine that we have installed a battery like a PowerWall, in order to maximize the PV self-consumption. With Home Assistant we also have sensors that can measure the power produced by our PV plant, the global power consumption of the house and hopefully the power consumed by the controllable loads. Home Assistant has released the Energy Dashboard where we can viusalize all these variables in somme really good looking graphics. See: https://www.home-assistant.io/blog/2021/08/04/home-energy-management/

Now, how can we be certain of the good and optimal management of these devices? If we define a fixed schedule for our deferrable loads, is this the best solution? When we can indicate or force a charge or discharge on the battery? This is a well known academic problem for an Energy Management System.

The first and most basic approach could be to define some basic rules or heuristics, this is the so called rule-based approach. The rules could be some fixed schedules for the deferrable loads, or some threshold based triggering of the battery charge/discharge, and so on. The rule-based approach has the advantage of being simple to implement and robust. However, the main disadvantage is that optimality is not guaranteed.

The goal of this work is to provide an easy to implement framework where anyone using Home Assistant can apply the best and optimal set of instructions to control the energy flow in a household. There are many ways and techniques that can be found in the literature to implement optimized EMS. In this package we are using just one of those techniques, the Linear Programming approach, that will be presented below.

When I was designing and testing this package in my own house I estimated a daily gain between 5% and 8% when using the optimized approach versus a rule-based one. In my house I have a 5 kWp PV installation with a contractual grid supply of 9 kVA. I have a grid contract with two tariffs for power consumption for the grid (peak and non-peak hours) and one tariff for the excess PV energy injected to the grid. I have no battery installed, but I suppose that the margin of gain would be even bigger with a battery, adding flexibility to the energy management. Of course the disadvantage is the initial capital cost of the battery stack. In my case the gain comes from the fact that the EMS is helping me to decide when to turn on my water heater and the pool pump. If we have a good clear sky day the results of the optimization will normally be to turn them on during the day where solar production is present. But if the day is going to be really clouded, then is possible that the best solution will be to turn them on during the non-peak tariff hours, for my case this is during the night from 9pm to 2am. All these decisions are made automatically by the EMS using forecasts of both the PV production and the house power consumption.

Some other good packages and projects offer similar approaches to EMHASS. I can cite for example the good work done by my friends at the G2ELab in Grenoble, France. They have implemented the OMEGAlpes package that can also be used as an optimized EMS using LP and MILP (see: https://gricad-gitlab.univ-grenoble-alpes.fr/omegalpes/

omegalpes). But here in EMHASS the first goal was to keep it simple to implement using configuration files and the second goal was that it should be easy to integrate to Home Assistant. I am sure that there will be a lot of room for optimize the code and the package implementation as this solution will be used and tested in the future.

I have included a list of scientific references at the bottom if you want to deep into the technical aspects of this subject.

Ok, let's start by a resumed presentation of the LP approach.

3.2 Linear programming

Linear programming is an optimization method that can be used to obtain the best solution from a given cost function using a linear modeling of a problem. Typically we can also also add linear constraints to the optimization problem.

This can be mathematically written as:

$$\begin{aligned}
&\underset{x}{\text{Maximize}} \\
&\mathbf{c}^{\mathrm{T}}\mathbf{x} \\
&\text{subject to} \\
&A\mathbf{x} \leq \mathbf{b} \\
&\text{and} \\
&\mathbf{x} \geq \mathbf{0}
\end{aligned}$$

with $\mathbf{x}$ the variable vector that we want to find, $\mathbf{c}$ and $\mathbf{b}$ are vectors with known coefficients and $\mathbf{A}$ is a matrix with known values. Here the cost function is defined by $\mathbf{c}^{\mathrm{T}}\mathbf{x}$. The inequalities $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$ represent the convex region of feasible solutions.

We could find a mix of real and integer variables in $\mathbf{x}$, in this case the problem is referred as Mixed Integer Linear Programming (MILP). Typically this kind of problem use the branch and boud type of solvers or similars.

The LP has of course its set of advantages and disadvantages. The main advantage is the that if the problem is well posed and the region of feasible possible solutions is convex, then a solution is guaranteed and solving times are usually fast when compared to other optimization techniques (as dynamic programming for example). However we can easily fall into memory issues, larger solving times and convergence problems if the size of the problem is too high (too many equations).

3.3 Household EMS with LP

The LP problem for the household EMS is solved in EMHASS using different user-chosen cost functions.

Three main cost functions are proposed.

3.3.1 Cost functions

1/ The *profit* cost function:

In this case the cost function is posed to maximize the profit. The profit is defined by the revenues from selling PV power to the grid minus the cost of consumed energy from the grid. This can be represented with the following objective function:

$$\sum_{i=1}^{\Delta_{opt}/\Delta_t} -0.001 * \Delta_t * (unit_{LoadCost}[i] * P_{gridPos}[i] + prod_{SellPrice} * P_{gridNeg}[i])$$

For the special case of an energy contract where the totality of the PV produced energy is injected into the grid this will be:

$$> \sum_{i=1}^{\Delta_{opt}/\Delta_t} -0.001 * \Delta_t * (unit_{LoadCost}[i] * (P_{load}[i] + P_{defSum}[i]) + prod_{SellPrice} * P_{gridNeg}[i]) >$$

where Δ_{opt} is the total period of optimization in hours, Δ_t is the optimization time step in hours, $unit_{LoadCost_i}$ is the cost of the energy from the utility in EUR/kWh, P_{load} is the electricity load consumption (positive defined), P_{defSum} is the sum of the deferrable loads defined, $prod_{SellPrice}$ is the price of the energy sold to the utility, $P_{gridNeg}$ is the negative component of the grid power, this is the power exported to the grid. All these power are expressed in Watts.

2/ The energy from the grid *cost*:

In this case the cost function is computed as the cost of the energy coming from the grid. The PV power injected into the grid is not valorized. This is:

$$\sum_{i=1}^{\Delta_{opt}/\Delta_t} -0.001 * \Delta_t * unit_{LoadCost}[i] * P_{gridPos}[i]$$

Again, for the special case of an energy contract where the totality of the PV produced energy is injected into the grid this will be:

$$> \sum_{i=1}^{\Delta_{opt}/\Delta_t} -0.001 * \Delta_t * unit_{LoadCost}[i] * (P_{load}[i] + P_{defSum}[i]) >$$

3/ The *self-consumption* cost function:

This is a cost function designed to maximize the self-consumption of the PV plant.

Note: EMHASS has two methods for defining a self-consumption cost function: **bigm** and **maxmin**. In the current version, only the **bigm** method is used, as the maxmin method has convergence issues.

---

bigM self-consumption method

In this case, the cost function is based on the profit cost function, but the energy offtake cost is weighted more heavily than the energy injection revenue. This can be represented with the following objective function:

$$\sum_{i=1}^{\Delta_{opt}/\Delta_t} -0.001 * \Delta_t * (bigM * unit_{LoadCost}[i] * P_{gridPos}[i] + prod_{SellPrice} * P_{gridNeg}[i])$$

where bigM equals 1000. Adding this bigM factor will give more weight to the cost of grid offtake, or formulated differently: avoiding offtake through self-consumption will have strong influence on the calculated cost.

Please note that the bigM factor is not used in the calculated cost that comes out of the optimizer results. It is only used to drive the optimizer.

Maxmin self-consumption method (currently disabled)

The cost function is computed as the revenues from selling PV power to the grid, plus the avoided cost of consuming PV power locally (the latter means: valorizing the self-consumed cost at the grid offtake price).

The self-consumption is defined as:

$$> SC = \min(P_{PV}, (P_{load} + P_{defSum})) >$$

To convert this to a linear cost function, an additional continuous variable SC is added. This is the so-called maximin problem. The cost function is defined as:

$$> \sum_{i=1}^{\Delta_{opt}/\Delta_t} SC[i] >$$

With the following set of constraints:

$$> SC[i] \leq P_{PV}[i] >$$

and

$$> SC[i] \leq P_{load}[i] + P_{defSum}[i] >$$

All these cost functions can be chosen by the user with the `--costfun` tag with the `emhass` command. The options are: `profit`, `cost`, `self-consumption`. They are all set in the LP formulation as cost function to maximize.

The problem constraints are written as follows.

3.3.2 The main constraint: power balance

$$P_{PV_i} - P_{defSum_i} - P_{load_i} + P_{gridNeg_i} + P_{gridPos_i} + P_{stoPos_i} + P_{stoNeg_i} = 0$$

with P_{PV} the PV power production, $P_{gridPos}$ the positive component of the grid power (from grid to household), P_{stoPos} and P_{stoNeg} are the positive (discharge) and negative components of the battery power (charge).

Normally the PV power production and the electricity load consumption are considered known. In the case of a day-ahead optimization these should be forecasted values. When the optimization problem is solved the others power defining the power flow are found as a result: the deferrable load power, the grid power and the battery power.

3.3.3 Other constraints

Some other special linear constraints are defined. A constraint is introduced to avoid injecting and consuming from grid at the same time, which is physically impossible. Other constraints are used to control the total time that a deferrable load will stay on and the number of start-ups.

Constraints are also used to define semi-continuous variables. Semi-continuous variables are variables that must take a value between their minimum and maximum or zero.

A final set of constraints is used to define the behavior of the battery. Notably:

- Ensure that maximum charge and discharge powers are not exceeded.
- Minimum and maximum state of charge values are not exceeded.
- Force the final state of charge value to be equal to the initial state of charge.

The minimum and maximum state of charge limitations can be expressed as follows:

$$\sum_{i=1}^{k} \frac{P_{stoPos_i}}{\eta_{dis}} + \eta_{ch} P_{stoNeg_i} \leq \frac{E_{nom}}{\Delta_t} (SOC_{init} - SOC_{min})$$

and

$$-(\sum_{i=1}^{k} \frac{P_{stoPos_i}}{\eta_{dis}} + \eta_{ch} P_{stoNeg_i}) \leq \frac{E_{nom}}{\Delta_t} (SOC_{max} - SOC_{init})$$

where E_{nom} is the battery capacity in kWh, $\eta_{dis/ch}$ are the discharge and charge efficiencies and SOC is the state of charge.

Forcing the final state of charge value to be equal to the initial state of charge can be expressed as follows:

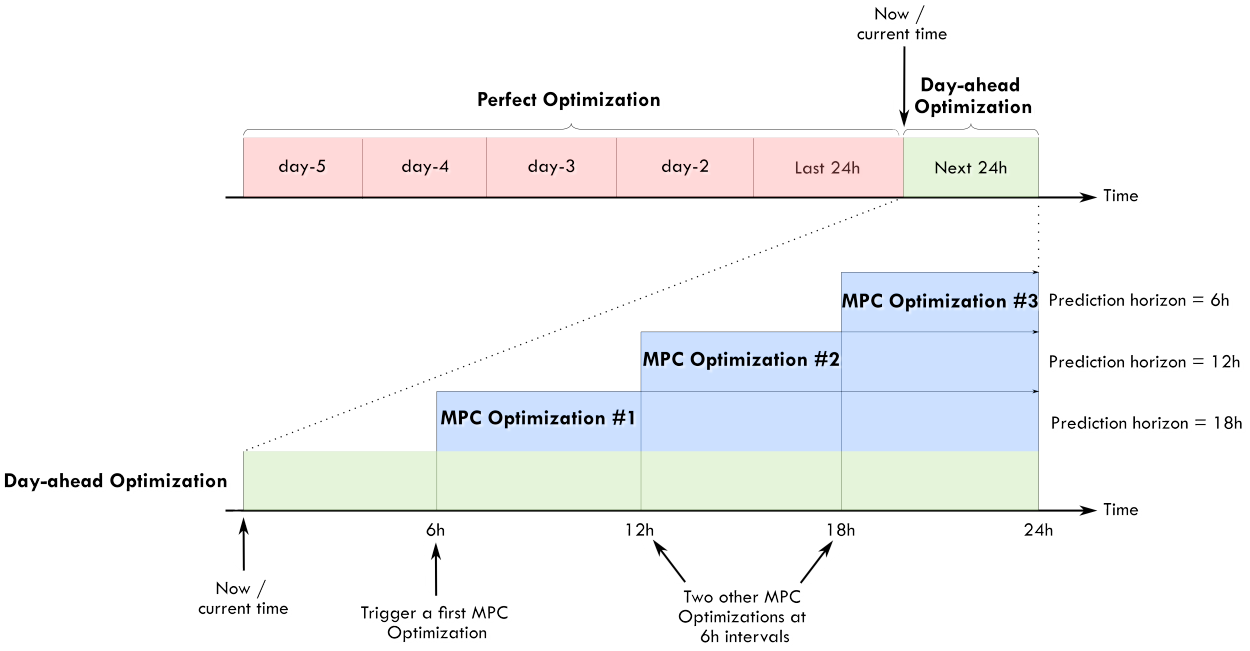
$$\sum_{i=1}^{k} \frac{P_{stoPos_i}}{\eta_{dis}} + \eta_{ch} P_{stoNeg_i} = \frac{E_{nom}}{\Delta_t} (SOC_{init} - SOC_{final})$$

3.4 The EMHASS optimizations

There are 3 different optimization types that are implemented in EMHASS.

- A perfect forecast optimization.
- A day-ahead optimization.
- A Model Predictive Control optimization.

The following example diagram may help us understand the time frames of these optimizations:



3.4.1 Perfect forecast optimization

This is the first type of optimization task that are proposed with this package. In this case the main inputs, the PV power production and the house power consumption, are fixed using historical values from the past. This mean that in some way we are optimizing a system with a perfect knowledge of the future. This optimization is of course non-practical in real life. However this can be give us the best possible solution of the optimization problem that can be later used as a reference for comparison purposes. On the example diagram presented before, the perfect optimization is defined on a 5-day period. These historical values will be retrieved from the Home Assistant database.

3.4.2 Day-ahead optimization

In this second type of optimization task the PV power production and the house power consumption are forecasted values. This is the action that should be performed in a real case scenario and is the case that should be launched from Home Assistant to obtain an optimized energy management of future actions. This optimization is defined in the time frame of the next 24 hours.

As the optimization is bounded to forecasted values, it will also be bounded to uncertainty. The quality and accuracy of the optimization results will be inevitably linked to the quality of the forecast used for these values. The better the forecast error, the better accuracy of the optimization result.

3.4.3 Model Predictive Control (MPC) optimization

This is an informal/naive representation of a MPC controller.

This type of controller performs the following actions:

- Set the prediction horizon and receding horizon parameters.
- Perform an optimization on the prediction horizon.
- Apply the first element of the obtained optimized control variables.
- Repeat at a relatively high frequency, ex: 5 min.

On the example diagram presented before, the MPC is performed on 6h intervals at 6h, 12h and 18h. The prediction horizon is progressively reducing during the day to keep the one-day energy optimization notion (it should not just be a fixed rolling window as, for example, you would like to know when you want to reach the desired `soc_final`). This type of optimization is used to take advantage of actualized forecast values during throughout the day. The user can of course choose higher/lower implementation intervals, keeping in mind the contraints below on the `prediction_horizon`.

When applying this controller, the following `runtimeparams` should be defined:

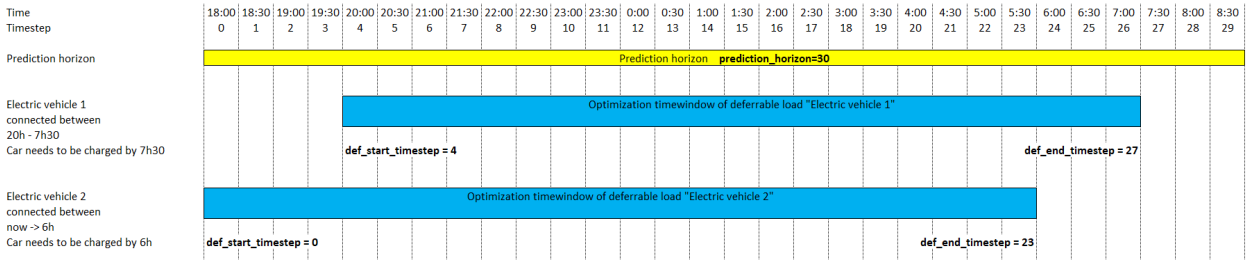
- `prediction_horizon` for the MPC prediction horizon. Fix this at at least 5 times the optimization time step.
- `soc_init` for the initial value of the battery SOC for the current iteration of the MPC.
- `soc_final` for the final value of the battery SOC for the current iteration of the MPC.
- `def_total_hours` for the list of deferrable loads functioning hours. These values can decrease as the day advances to take into account receding horizon daily energy objectives for each deferrable load.
- `def_start_timestep` for the timestep as from which each deferrable load is allowed to operate (if you don't want the deferrable load to use the whole optimization timewindow). If you specify a value of 0 (or negative), the deferrable load will be optimized as from the beginning of the complete prediction horizon window.
- `def_end_timestep` for the timestep before which each deferrable load should operate (if you don't want the deferrable load to use the whole optimization timewindow). If you specify a value of 0 (or negative), the deferrable load will be optimized over the complete prediction horizon window.

In a practical use case, the values for `soc_init` and `soc_final` for each MPC optimization can be taken from the initial day-ahead optimization performed at the beginning of each day.

3.4.4 Time windows for deferrable loads

Since v0.7.0, the user has the possibility to limit the operation of each deferrable load to a specific timewindow, which can be smaller than the prediction horizon. This is done by means of the `def_start_timestep` and `def_end_timestep` parameters. These parameters can either be set in the configuration screen of the Home Assistant EMHASS add-on, in the config_emhass.yaml file, or provided as runtime parameters.

Taking the example of two electric vehicle that need to charge, but which are not available during the whole prediction horizon:



For this example, the settings could look like this: Either in the Home Assistant add-on config screen:

list_nominal_power_of_deferrable_loads, The nominal power for each deferrable load in Watts

```
- nominal_power_of_deferrable_loads: 7360
- nominal_power_of_deferrable_loads: 7360
```

This is a list of elements, the number of elements. (number of items = number_of_deferrable_loads)

list_operating_hours_of_each_deferrable_load, The total number of hours that each deferrable load should operate

```
- operating_hours_of_each_deferrable_load: 4
- operating_hours_of_each_deferrable_load: 2
```

A list of elements. (number of items = number_of_deferrable_loads)

list_start_timesteps_of_each_deferrable_load, The timestep as from which each deferrable load is allowed to operate. Operation before this timestep is not allowed.

```
- start_timesteps_of_each_deferrable_load: 4
- start_timesteps_of_each_deferrable_load: 0
```

(Default 0). A list of elements. If value 0 is given, the deferrable load will be optimized as from the beginning of the complete prediction horizon window. (number of items = number_of_deferrable_loads)

list_end_timesteps_of_each_deferrable_load,The timestep before which each deferrable load should operate. Operation after this timestep is not allowed.

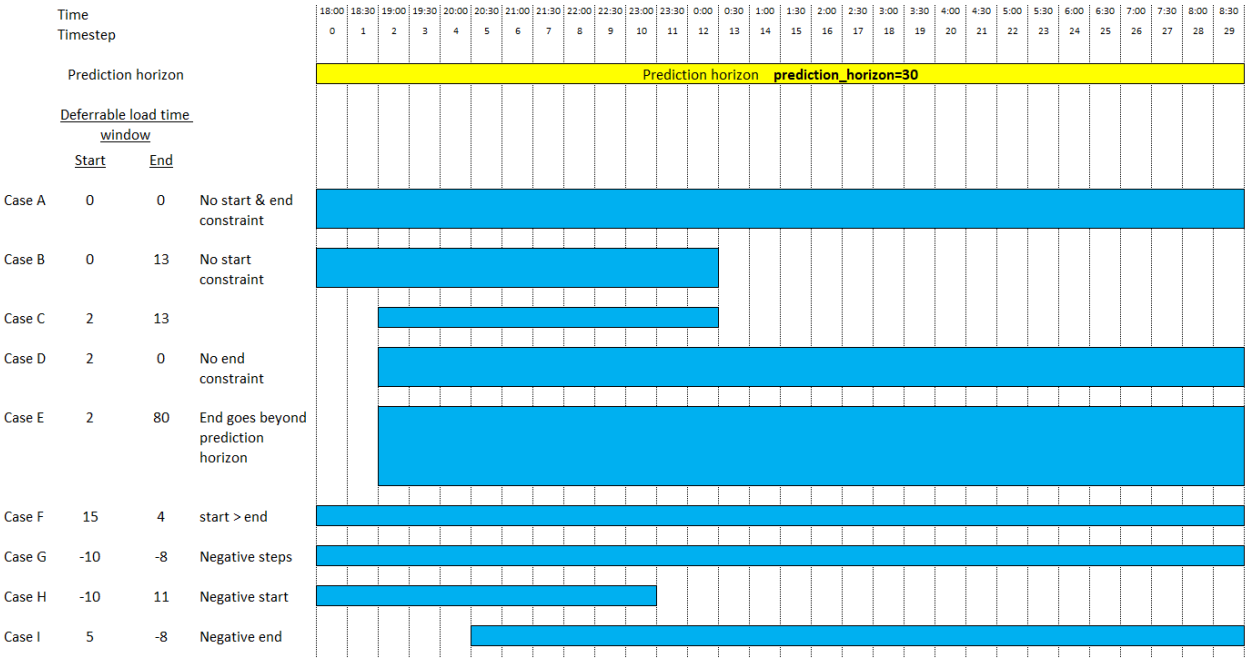
```
- end_timesteps_of_each_deferrable_load: 27
- end_timesteps_of_each_deferrable_load: 23
```

(Default 0). A list of elements. If value 0 is given, the deferrable load optimization window will extend up to the end of the prediction horizon window. (number of items = number_of_deferrable_loads)

Either as runtime parameter:

```
curl -i -H 'Content-Type:application/json' -X POST -d '{"prediction_horizon":30, "def_
↪total_hours":[4,2],"def_start_timestep":[4,0],"def_end_timestep":[27,23]}' http://
↪localhost:5000/action/naive-mpc-optim
```

Please note that the proposed deferrable load time windows will be submitted to a validation step & can be automatically corrected. Possible cases are depicted below:



3.5 References

- Camille Pajot, Lou Morriet, Sacha Hodencq, Vincent Reinbold, Benoit Delinchant, Frédéric Wurtz, Yves Maréchal, Omegalpes: An Optimization Modeler as an EfficientTool for Design and Operation for City Energy Stakeholders and Decision Makers, BS'15, Building Simulation Conference, Roma in September 24, 2019.
- Gabriele Comodi, Andrea Giantomassi, Marco Severini, Stefano Squartini, Francesco Ferracuti, Alessandro Fonti, Davide Nardi Cesarini, Matteo Morodo, and Fabio Polonara. Multi-apartment residential microgrid with electrical and thermal storage devices: Experimental analysis and simulation of energy management strategies. Applied Energy, 137:854–866, January 2015.
- Pedro P. Vergara, Juan Camilo López, Luiz C.P. da Silva, and Marcos J. Rider. Security-constrained optimal energy management system for threephase residential microgrids. Electric Power Systems Research, 146:371–382, May 2017.
- R. Bourbon, S.U. Ngueveu, X. Roboam, B. Sareni, C. Turpin, and D. Hernandez-Torres. Energy management optimization of a smart wind power plant comparing heuristic and linear programming methods. Mathematics and Computers in Simulation, 158:418–431, April 2019.

CHAPTER

FOUR

THE FORECAST MODULE

EMHASS will basically need 4 forecasts to work properly:

- PV power production forecast (internally based on the weather forecast and the characteristics of your PV plant). This is given in Watts.
- Load power forecast: how much power your house will demand on the next 24h. This is given in Watts.
- Load cost forecast: the price of the energy from the grid on the next 24h. This is given in EUR/kWh.
- PV production selling price forecast: at what price are you selling your excess PV production on the next 24h. This is given in EUR/kWh.

There are methods that are generalized to the 4 forecast needed. For all there forecasts it is possible to pass the data either as a passed list of values or by reading from a CSV file. With these methods it is then possible to use data from external forecast providers.

Then there are the methods that are specific to each type of forecast and that proposed forecast treated and generated internally by this EMHASS forecast class. For the weather forecast a first method (`scrapper`) uses a scrapping to the ClearOutside webpage which proposes detailed forecasts based on Lat/Lon locations. Another method (`solcast`) is using the SolCast PV production forecast service. A final method (`solar.forecast`) is using another external service: Solar.Forecast, for which just the nominal PV peak installed power should be provided. Search the forecast section on the documentation for examples on how to implement these different methods.

The `get_power_from_weather` method is proposed here to convert from irradiance data to electrical power. The PVLib module is used to model the PV plant. A dedicated webapp will help you search for your correct PV module and inverter: https://emhass-pvlib-database.streamlit.app/

The specific methods for the load forecast are a first method (`naive`) that uses a naive approach, also called persistance. It simply assumes that the forecast for a future period will be equal to the observed values in a past period. The past period is controlled using parameter `delta_forecast`. A second method (`mlforecaster`) uses an internal custom forecasting model using machine learning. There is a section in the documentation explaining how to use this method.

Note: This custom machine learning model is introduced from v0.4.0. EMHASS proposed this new `mlforecaster` class with `fit`, `predict` and `tune` methods. Only the `predict` method is used here to generate new forecasts, but it is necessary to previously fit a forecaster model and it is a good idea to optimize the model hyperparameters using the `tune` method. See the dedicated section in the documentation for more help.

For the PV production selling price and Load cost forecasts the privileged method is a direct read from a user provided list of values. The list should be passed as a runtime parameter during the `curl` to the EMHASS API.

4.1 PV power production forecast

The default method for PV power forecast is the scrapping of weather forecast data from the https://clearoutside.com/ website. This is obtained using `method=scrapper`. This site proposes detailed forecasts based on Lat/Lon locations. This method seems quite stable but as with any scrape method it will fail if any changes are made to the webpage API. The weather forecast data is then converted into PV power production using the `list_pv_module_model` and `list_pv_inverter_model` paramters defined in the configuration.

A second method uses the SolCast solar forecast service. Go to https://solcast.com/ and configure your system. You will need to set `method=solcast` and basically use two parameters `solcast_rooftop_id` and `solcast_api_key` that should be passed as parameters at runtime. This will be limited to 10 API requests per day, the granularity will be 30 min and the forecast is updated every 6h. If needed, better performances may be obtained with paid plans: https://solcast.com/pricing/live-and-forecast.

For example:

```
curl -i -H "Content-Type:application/json" -X POST -d '{"solcast_rooftop_id":"<your_
↪system_id>","solcast_api_key":"<your_secret_api_key>"}' http://localhost:5000/action/
↪dayahead-optim
```

A third method uses the Solar.Forecast service. You will need to set `method=solar.forecast` and use just one parameter `solar_forecast_kwp` (the PV peak installed power in kW) that should be passed at runtime. This will be using the free public Solar.Forecast account with 12 API requests per day and 1h data resolution. As with SolCast, there are paid account services that may results in better forecasts.

For example, for a 5 kWp installation:

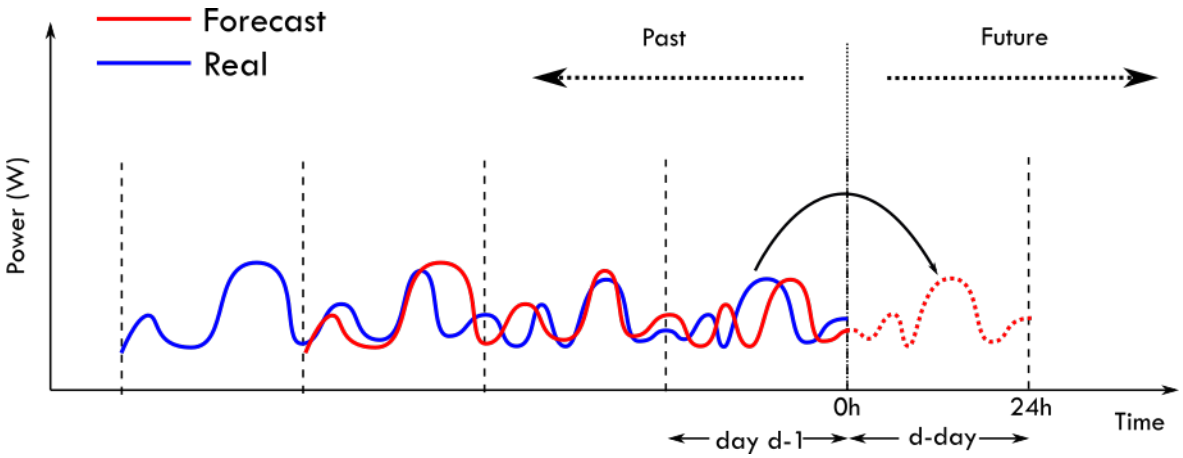
```
curl -i -H "Content-Type:application/json" -X POST -d '{"solar_forecast_kwp":5}' http://
↪localhost:5000/action/dayahead-optim
```

Note: If you use the Solar.Forecast or Solcast methods, or explicitly pass the PV power forecast values (see below), the list_pv_module_model and list_pv_inverter_model paramters defined in the configuration will be ignored.

4.2 Load power forecast

The default method for load forecast is a naive method, also called persistence. This is obtained using `method=naive`. This method simply assumes that the forecast for a future period will be equal to the observed values in a past period. The past period is controlled using parameter `delta_forecast` and the default value for this is 24h.

This is presented graphically here:



Note: New in EMHASS v0.4.0: machine learning forecast models!

Starting with v0.4.0, a new forecast framework is proposed within EMHASS. It provides a more efficient way to forecast the power load consumption. It is based on the `skforecast` module that uses `scikit-learn` regression models considering auto-regression lags as features. The hyperparameter optimization is proposed using bayesian optimization from the `optuna` module. To use this change to `method=mlforecaster` in the configuration.

The API provides fit, predict and tune methods.

The following is an example of a trained model using a KNN regressor:

The naive persistance model performs very well on the 2 day test period, however is well out-performed by the KNN regressor when back-testing on the complete training set (10 months of 30 minute time step data).

The hyperparameter tuning using bayesian optimization improves the bare KNN regressor from $R^2 = 0.59$ to $R^2 = 0.75$. The optimized number of lags is 48.

See the *machine learning forecaster* section for more details.

4.3 Load cost forecast

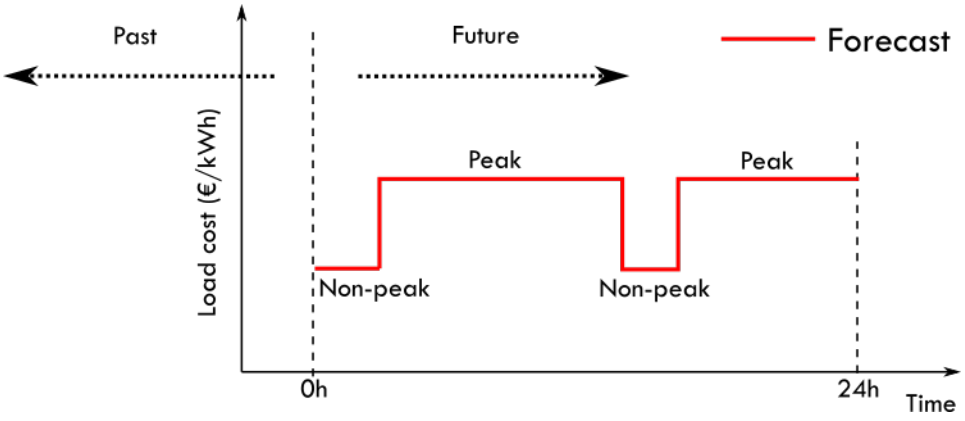
The default method for load cost forecast is defined for a peak and non-peak hours contract type. This is obtained using `method=hp_hc_periods`.

When using this method you can provide a list of peak-hour periods, so you can add as many peak-hour periods as possible.

As an example for a two peak-hour periods contract you will need to define the following list in the configuration file:

```
- list_hp_periods:
    - period_hp_1:
        - start: '02:54'
        - end: '15:24'
    - period_hp_2:
        - start: '17:24'
        - end: '20:24'
- load_cost_hp: 0.1907
- load_cost_hc: 0.1419
```

This example is presented graphically here:



4.4 PV production selling price forecast

The default method for this forecast is simply a constant value. This can be obtained using `method=constant`.

Then you will need to define the `prod_sell_price` variable to provide the correct price for energy injected to the grid from excedent PV production in €/kWh.

4.5 Passing your own forecast data

For all the needed forecasts in EMHASS two other methods allows the user to provide their own forecast value. This may be used to provide a forecast provided by a more powerful and accurate forecaster. The two methods are: `csv` and `list`.

For the `csv` method you should push a csv file to the `data` folder. The CSV file should contain no header and the timestamped data should have the following format:

```
2021-04-29 00:00:00+00:00,287.07
2021-04-29 00:30:00+00:00,274.27
2021-04-29 01:00:00+00:00,243.38
...
```

For the list method you just have to add the data as a list of values to a data dictionnary during the call to `emhass` using the `runtimeparams` option.

The possible dictionnary keys to pass data are:

- `pv_power_forecast` for the PV power production forecast.
- `load_power_forecast` for the Load power forecast.
- `load_cost_forecast` for the Load cost forecast.
- `prod_price_forecast` for the PV production selling price forecast.

For example if using the add-on or the standalone docker installation you can pass this data as list of values to the data dictionnary during the `curl` POST:

```
curl -i -H "Content-Type: application/json" -X POST -d '{"pv_power_forecast":[0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 70, 141.22, 246.18, 513.5, 753.27, 1049.89, 1797.
↪93, 1697.3, 3078.93, 1164.33, 1046.68, 1559.1, 2091.26, 1556.76, 1166.73, 1516.63,␣
↪1391.13, 1720.13, 820.75, 804.41, 251.63, 79.25, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}' http:/
↪/localhost:5000/action/dayahead-optim
```

You need to be careful here to send the correct amount of data on this list, the correct length. For example, if the data time step is defined to 1h and you are performing a day-ahead optimization, then this list length should be of 24 data points.

4.5.1 Example using: SolCast forecast + Amber prices

If you're using SolCast then you can define the following sensors in your system:

```
sensors:

  - platform: rest
    name: "Solcast Forecast Data"
    json_attributes:
      - forecasts
    resource: https://api.solcast.com.au/rooftop_sites/yyyy/forecasts?format=json&api_
↪key=xxx&hours=24
    method: GET
    value_template: "{{ (value_json.forecasts[0].pv_estimate)|round(2) }}"
    unit_of_measurement: "kW"
    device_class: power
    scan_interval: 8000
    force_update: true

  - platform: template
    sensors:
      solcast_24hrs_forecast :
        value_template: >-
          {%- set power = state_attr('sensor.solcast_forecast_data', 'forecasts') |␣
↪map(attribute='pv_estimate') | list %}
          {%- set values_all = namespace(all=[]) %}
          {% for i in range(power | length) %}
           {%- set v = (power[i] | float |multiply(1000) ) | int(0) %}
            {%- set values_all.all = values_all.all + [ v ] %}
          {%- endfor %} {{ (values_all.all)[:48] }}
```

With this you can now feed this SolCast forecast to EMHASS along with the mapping of the Amber prices.

A MPC call may look like this for 4 deferrable loads:

```
    post_mpc_optim_solcast: "curl -i -H \"Content-Type: application/json\" -X POST -d '{\
↪"load_cost_forecast\":{{(
          ([states('sensor.amber_general_price')|float(0)] +
          state_attr('sensor.amber_general_forecast', 'forecasts') |map(attribute='per_
↪kwh')|list)[:48])
          }}, \"prod_price_forecast\":{{(
          ([states('sensor.amber_feed_in_price')|float(0)] +
          state_attr('sensor.amber_feed_in_forecast', 'forecasts')|map(attribute='per_kwh
```

(continues on next page)

(continued from previous page)

```
↪')|list)[:48])
          }}, \"pv_power_forecast\":{{states('sensor.solcast_24hrs_forecast')
          }}, \"prediction_horizon\":48,\"soc_init\":{{(states('sensor.powerwall_charge
↪')|float(0))/100
          }},\"soc_final\":0.05,\"def_total_hours\":[2,0,0,0]}' http://localhost:5000/
↪action/naive-mpc-optim"
```

Thanks to @purcell_labs for this example configuration.

4.5.2 Example combining multiple SolCast configurations

If you have multiple rooftops, for example for east-west facing solar panels, then you will need to fuze the sensors providing the different forecasts on a single one using templates in Home Assistant. Then feed that single sensor data passing the data as a list when calling the shell command.

Here is a sample configuration to achieve this, thanks to @gieljnssns for sharing.

The two sensors using rest sensors:

```
- platform: rest
  name: "Solcast Forecast huis"
  json_attributes:
    - forecasts
  resource: https://api.solcast.com.au/rooftop_sites/xxxxxxxxxxc/forecasts?format=json&
↪api_key=xxxxxxxxx&hours=24
  method: GET
  value_template: "{{ (value_json.forecasts[0].pv_estimate)|round(2) }}"
  unit_of_measurement: "kW"
  device_class: power
  scan_interval: 86400
  force_update: true

- platform: rest
  name: "Solcast Forecast garage"
  json_attributes:
    - forecasts
  resource: https://api.solcast.com.au/rooftop_sites/xxxxxxxxxxc/forecasts?format=json&
↪api_key=xxxxxxxxx&hours=24
  method: GET
  value_template: "{{ (value_json.forecasts[0].pv_estimate)|round(2) }}"
  unit_of_measurement: "kW"
  device_class: power
  scan_interval: 86400
  force_update: true
```

Then two templates, one for each sensor:

```
    solcast_24hrs_forecast_garage:
      value_template: >-
        {%- set power = state_attr('sensor.solcast_forecast_garage', 'forecasts') |␣
↪map(attribute='pv_estimate') | list %}
        {%- set values_all = namespace(all=[]) %}
```

(continues on next page)

(continued from previous page)

```
        {% for i in range(power | length) %}
          {%- set v = (power[i] | float |multiply(1000) ) | int(0) %}
          {%- set values_all.all = values_all.all + [ v ] %}
        {%- endfor %} {{ (values_all.all)[:48] }}

    solcast_24hrs_forecast_huis:
      value_template: >-
        {%- set power = state_attr('sensor.solcast_forecast_huis', 'forecasts') |␣
↪map(attribute='pv_estimate') | list %}
        {%- set values_all = namespace(all=[]) %}
        {% for i in range(power | length) %}
          {%- set v = (power[i] | float |multiply(1000) ) | int(0) %}
          {%- set values_all.all = values_all.all + [ v ] %}
        {%- endfor %} {{ (values_all.all)[:48] }}
```

And the fusion of the two sensors:

```
    solcast_24hrs_forecast:
      value_template: >-
        {% set a = states("sensor.solcast_24hrs_forecast_garage")[1:-1].split(',') | map(
↪'int') | list %}
        {% set b = states("sensor.solcast_24hrs_forecast_huis")[1:-1].split(',') | map(
↪'int') | list %}
        {% set ns = namespace(items = []) %}
        {% for i in range(a | length) %}
          {% set ns.items = ns.items + [ a[i]  + b[i]  ] %}
        {% endfor %}
        {{ ns.items }}
```

And finally the shell command:

```
dayahead_optim: "curl -i -H \"Content-Type:application/json\" -X POST -d '{\"pv_power_
↪forecast\":{{states('sensor.solcast_24hrs_forecast')}}}' http://localhost:5001/action/
↪dayahead-optim"
```

4.5.3 Example using the Nordpool integration

The Nordpool integration provides spot market electricity prices (consumption and production) for the Nordic, Baltic and part of Western Europe. An integration for Home Assistant can be found here: https://github.com/custom-components/nordpool

After setup the sensors should appear in Home Assistant for raw `today` and `tomorrow` values.

The subsequent shell command to concatenate `today` and `tomorrow` values can be for example:

```
shell_command:
  trigger_nordpool_forecast: "curl -i -H \"Content-Type: application/json\" -X POST -d '
↪{\"load_cost_forecast\":{{((state_attr('sensor.nordpool', 'raw_today') | map(attribute=
↪'value') | list  + state_attr('sensor.nordpool', 'raw_tomorrow') | map(attribute='value
↪') | list))[now().hour:][:24] }},\"prod_price_forecast\":{{((state_attr('sensor.
↪nordpool', 'raw_today') | map(attribute='value') | list  + state_attr('sensor.nordpool
↪', 'raw_tomorrow') | map(attribute='value') | list))[now().hour:][:24]}}}' http://
↪localhost:5000/action/dayahead-optim"
```

4.6 Now/current values in forecasts

When implementing MPC applications with high optimization frequencies it can be interesting if at each MPC iteration the forecast values are updated with the real now/current values measured from live data. This is useful to improve the accuracy of the short-term forecasts. As shown in some of the references below, mixing with a persistance model make sense since this type of model performs very good at low temporal resolutions (intra-hour).

A simple integration of current/now values for PV and load forecast is implemented using a mixed one-observation presistence model and the one-step-ahead forecasted values from the current passed method.

This can be represented by the following equation at time $t = k$:

$$P_{PV}^{mix} = \alpha \hat{P}_{PV}(k) + \beta P_{PV}(k-1)$$

Where P_{PV}^{mix} is the mixed power forecast for PV prodduction, $\hat{P}_{PV}(k)$ is the current first element of the original forecast data, $P_{PV}(k-1)$ is the now/current value of PV production and α and β are coefficients that can be fixed to reflect desired dominance of now/current values over the original forecast data or viceversa.

The `alpha` and `beta` values can be passed in the dictionnary using the `runtimeparams` option during the call to `emhass`. If not passed they will both take the default 0.5 value. These values should be fixed following your own analysis on how much weight you want to put on measured values to be used as the persistance forecast. This will also depend on your fixed optimization time step. As a default they will be at 0.5, but if you want to give more weight to measured persistance values, then you can try lower α and rising β, for example: `alpha=0.25`, `beta=0.75`. After this you will need to check with the recored history if these values fits your needs.

4.7 References

- E. Lorenz, J. Kuhnert, A. Hammer, D. Heinemann, Photovoltaic (PV) power predictions with PV measurements, satellite data and numerical weather predictions. Presented at CM2E, Energy & Environment Symposium, Martinique, 2014.
- Maimouna Diagne, Mathieu David, Philippe Lauret, John Boland, NicolasSchmutz, Review of solar irradiance forecasting methods and a proposition for small-scale insular grids. Renewable and Sustainable Energy Reviews 27 (2013) 65–76.
- Bryan Lima, Sercan O. Arik, Nicolas Loeff, Tomas Pfister, Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting. arXiv:1912.09363v3 [stat.ML] 27 Sep 2020.

CHAPTER

FIVE

THE MACHINE LEARNING FORECASTER

Starting with v0.4.0, a new forecast framework is proposed within EMHASS. It provides a more efficient way to forecast the power load consumption. It is based on the `skforecast` module that uses `scikit-learn` regression models considering auto-regression lags as features. The hyperparameter optimization is proposed using bayesian optimization from the `optuna` module.

This API provides three main methods:

- fit: to train a model with the passed data. This method is exposed with the `forecast-model-fit` end point.
- predict: to obtain a forecast from a pre-trained model. This method is exposed with the `forecast-model-predict` end point.
- tune: to optimize the models hyperparameters using bayesian optimization. This method is exposed with the `forecast-model-tune` end point.

5.1 A basic model fit

To train a model use the `forecast-model-fit` end point.

Some paramters can be optionally defined at runtime:

- `days_to_retrieve`: the total days to retrieve from Home Assistant for model training. Define this in order to retrieve as much history data as possible.

Note: The minimum number of `days_to_retrieve` is hard coded to 9 by default. But it is adviced to provide more data for better accuracy by modifying your Home Assistant recorder settings.

- `model_type`: define the type of model forecast that this will be used for. For example: `load_forecast`. This should be an unique name if you are using multiple custom forecast models.
- `var_model`: the name of the sensor to retrieve data from Home Assistant. Example: `sensor.power_load_no_var_loads`.
- `sklearn_model`: the `scikit-learn` model that will be used. For now only this options are possible: `LinearRegression`, `ElasticNet` and `KNeighborsRegressor`.
- `num_lags`: the number of auto-regression lags to consider. A good starting point is to fix this as one day. For example if your time step is 30 minutes, then fix this to 48, if the time step is 1 hour the fix this to 24 and so on.
- `split_date_delta`: the delta from now to `split_date_delta` that will be used as the test period to evaluate the model.
- `perform_backtest`: if `True` then a back testing routine is performed to evaluate the performance of the model on the complete train set.

The default values for these parameters are:

```
runtimeparams = {
    "days_to_retrieve": 9,
    "model_type": "load_forecast",
    "var_model": "sensor.power_load_no_var_loads",
    "sklearn_model": "KNeighborsRegressor",
    "num_lags": 48,
    "split_date_delta": '48h',
    "perform_backtest": False
}
```

A correct `curl` call to launch a model fit can look like this:

```
curl -i -H "Content-Type:application/json" -X POST -d '{}' http://localhost:5000/action/
↪forecast-model-fit
```

As an example, the following figure shows a 240 days load power data retrieved from EMHASS and that will be used for a model fit:

After applying the `curl` command to fit the model the following information is logged by EMHASS:

```
2023-02-20 22:05:22,658 - __main__ - INFO - Training a KNN regressor
2023-02-20 22:05:23,882 - __main__ - INFO - Elapsed time: 1.2236599922180176
2023-02-20 22:05:24,612 - __main__ - INFO - Prediction R2 score: 0.2654560762747957
```

As we can see the R^2 score for the fitted model on the 2 day test perdiod is 0.27. A quick prediction graph using the fitted model should be available in the webui:

Visually the prediction looks quite acceptable but we need to evaluate this further. For this we can use the `"perform_backtest": True` option to perform a backtest evaluation using this syntax:

```
curl -i -H "Content-Type:application/json" -X POST -d '{"perform_backtest": "True"}'␣
↪http://localhost:5000/action/forecast-model-fit
```

The results of the backtest will be shown in the logs:

```
2023-02-20 22:05:36,825 - __main__ - INFO - Simple backtesting
2023-02-20 22:06:32,162 - __main__ - INFO - Backtest R2 score: 0.5851552394233677
```

So the mean backtest metric of our model is $R^2 = 0.59$.

Here is the graphic result of the backtesting routine:

5.2 The predict method

To obtain a prediction using a previously trained model use the `forecast-model-predict` end point.

```
curl -i -H "Content-Type:application/json" -X POST -d '{}' http://localhost:5000/action/
↪forecast-model-predict
```

If needed pass the correct `model_type` like this:

```
curl -i -H "Content-Type:application/json" -X POST -d '{"model_type": "load_forecast"}'␣
↪http://localhost:5000/action/forecast-model-predict
```

The resulting forecast DataFrame is shown in the webui.

It is possible to publish the predict method results to a Home Assistant sensor. By default this is desactivated but it can be activated by using runtime parameters.

The list of parameters needed to set the data publish task is:

- `model_predict_publish`: set to `True` to activate the publish action when calling the `forecast-model-predict` end point.
- `model_predict_entity_id`: the unique `entity_id` to be used.
- `model_predict_unit_of_measurement`: the `unit_of_measurement` to be used.
- `model_predict_friendly_name`: the `friendly_name` to be used.

The default values for these parameters are:

```
runtimeparams = {
    "model_predict_publish": False,
    "model_predict_entity_id": "sensor.p_load_forecast_custom_model",
    "model_predict_unit_of_measurement": "W",
    "model_predict_friendly_name": "Load Power Forecast custom ML model"
}
```

5.3 The tuning method with Bayesian hyperparameter optimization

With a previously fitted model you can use the `forecast-model-tune` end point to tune its hyperparameters. This will be using bayeasian optimization with a wrapper of `optuna` in the `skforecast` module.

You can pass the same parameter you defined during the fit step, but `var_model` has to be defined at least. According to the example, the syntax will be:

```
curl -i -H "Content-Type:application/json" -X POST -d '{"var_model": "sensor.power_load_
↪no_var_loads"}' http://localhost:5000/action/forecast-model-tune
```

This will launch the optimization routine and optimize the internal hyperparamters of the `scikit-learn` regressor and it will find the optimal number of lags. The following are the logs with the results obtained after the optimization for a KNN regressor:

```
2023-02-20 22:06:43,112 - __main__ - INFO - Backtesting and bayesian hyperparameter␣
↪optimization
2023-02-20 22:25:29,987 - __main__ - INFO - Elapsed time: 1126.868682384491
```

(continues on next page)

(continued from previous page)

```
2023-02-20 22:25:50,264 - __main__ - INFO - ### Train/Test R2 score comparison ###
2023-02-20 22:25:50,282 - __main__ - INFO - R2 score for naive prediction in train␣
↪period (backtest): 0.22525145245617462
2023-02-20 22:25:50,284 - __main__ - INFO - R2 score for optimized prediction in train␣
↪period: 0.7485208725102304
2023-02-20 22:25:50,312 - __main__ - INFO - R2 score for non-optimized prediction in␣
↪test period: 0.7098996657492629
2023-02-20 22:25:50,337 - __main__ - INFO - R2 score for naive persistance forecast in␣
↪test period: 0.8714987509894714
2023-02-20 22:25:50,352 - __main__ - INFO - R2 score for optimized prediction in test␣
↪period: 0.7572325833767719
```

This is a graph comparing these results:

The naive persistance load forecast model performs very well on the 2 day test period with a $R^2 = 0.87$, however is well out-performed by the KNN regressor when back-testing on the complete training set (10 months of 30 minute time step data) with a score $R^2 = 0.23$.

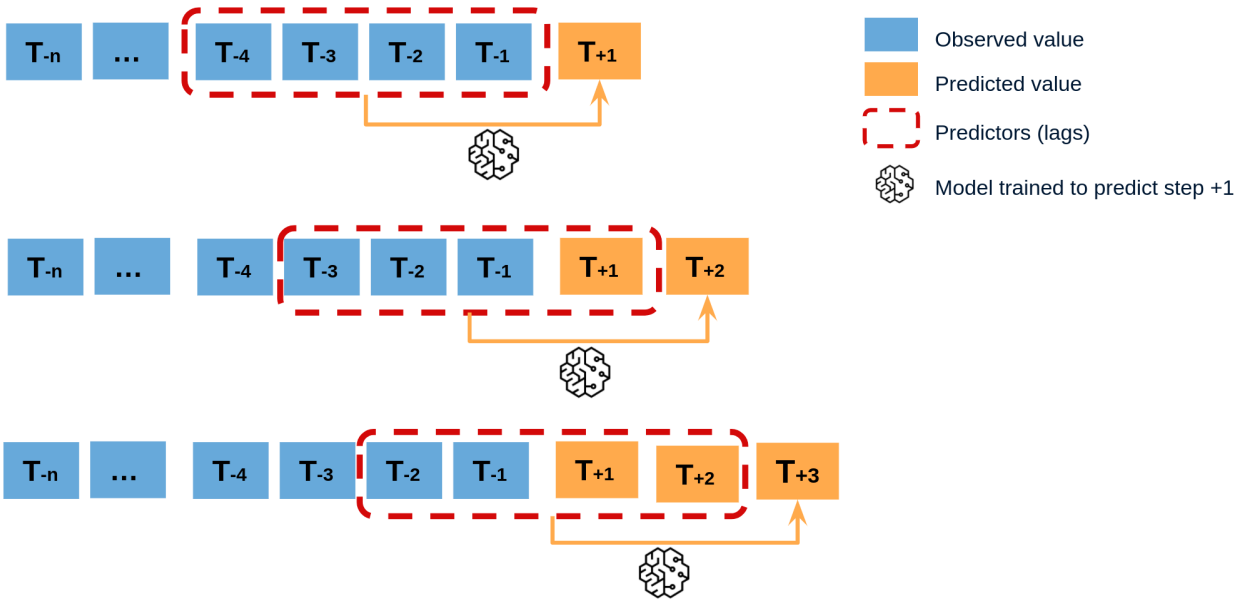
The hyperparameter tuning using bayesian optimization improves the bare KNN regressor from $R^2 = 0.59$ to $R^2 = 0.75$. The optimized number of lags is 48.

Warning: The tuning routine can be computing intense. If you have problems with computation times, try to reduce the `days_to_retrieve` parameter. In the example shown, for a 240 days train period, the optimization routine took almost 20 min to finish on an amd64 Linux architecture machine with a i5 processor and 8 Gb of RAM. This is a task that should be performed once in a while, for example every week.

5.4 How does this works?

This machine learning forecast class is based on the `skforecast` module. We use the recursive autoregresive forecaster with added features.

I will borrow this image from the `skforecast` documentation that help us understand the working principles of this type of model.



With this type of model what we do in EMHASS is to create new features based on the timestamps of the data retrieved from Home Assistant. We create new features based on the day, the hour of the day, the day of the week, the month of the year, among others.

What is interesting is that these added features are based on the timestamps, they always known in advance and useful for generating forecasts. These are the so-called future known covariates.

In the future we may test to expand using other possible known future covariates from Home Assistant, for example a known (forecasted) temperature, a scheduled presence sensor, etc.

5.5 Going further?

This class can be gebneralized to actually forecasting any given sensor variable present in Home Assistant. It has been tested and the main initial motivation for this development was for a better load power consumption forecasting. But in reality is has been coded in a flexible way so that you can control what variable is used, how many lags, the amount of data used to train the model, etc.

So you can really go further and try to forecast other types of variables and possible use the results for some interesting automations in Home Assistant. If doing this, was is important is to evaluate the pertinence of the obtained forecasts. The hope is that the tools proposed here can be used for that purpose.

CHAPTER

SIX

THE MACHINE LEARNING REGRESSOR

Starting with v0.9.0, a new framework is proposed within EMHASS. It provides a machine learning module to predict values from a csv file using different regression models.

This API provides two main methods:

- **fit**: To train a model with the passed data. This method is exposed with the `regressor-model-fit` end point.
- **predict**: To obtain a prediction from a pre-trained model. This method is exposed with the `regressor-model-predict` end point.

6.1 A basic model fit

To train a model use the `regressor-model-fit` end point.

Some paramters can be optionally defined at runtime:

- `csv_file`: The name of the csv file containing your data.
- `features`: A list of features, you can provide new values for this.
- `target`: The target, the value that has to be predicted.
- `model_type`: Define the name of the model regressor that this will be used for. For example: `heating_hours_degreeday`. This should be an unique name if you are using multiple custom regressor models.
- `regression_model`: The regression model that will be used. For now only this options are possible: `LinearRegression`, `RidgeRegression`, `LassoRegression`, `RandomForestRegression`, `GradientBoostingRegression` and `AdaBoostRegression`.
- `timestamp`: If defined, the column key that has to be used for timestamp.
- `date_features`: A list of 'date_features' to take into account when fitting the model. Possibilities are `year`, `month`, `day_of_week` (monday=0, sunday=6), `day_of_year`, `day`(day_of_month) and `hour`

6.1.1 Examples:

```
runtimeparams = {
    "csv_file": "heating_prediction.csv",
    "features": ["degreeday", "solar"],
    "target": "heating_hours",
    "regression_model": "RandomForestRegression",
    "model_type": "heating_hours_degreeday",
    "timestamp": "timestamp",
    "date_features": ["month", "day_of_week"]
    }
```

A correct `curl` call to launch a model fit can look like this:

```
curl -i -H "Content-Type:application/json" -X POST -d  '{"csv_file": "heating_prediction.
↪csv", "features": ["degreeday", "solar"], "target": "hour", "regression_model":
↪"RandomForestRegression", "model_type": "heating_hours_degreeday", "timestamp":
↪"timestamp", "date_features": ["month", "day_of_week"], "new_values": [12.79, 4.766, 1,
↪ 2] }' http://localhost:5000/action/regressor-model-fit
```

A Home Assistant `rest_command` can look like this:

```
fit_heating_hours:
  url: http://127.0.0.1:5000/action/regressor-model-fit
  method: POST
  content_type: "application/json"
  payload: >-
    {
    "csv_file": "heating_prediction.csv",
    "features": ["degreeday", "solar"],
    "target": "hours",
    "regression_model": "RandomForestRegression",
    "model_type": "heating_hours_degreeday",
    "timestamp": "timestamp",
    "date_features": ["month", "day_of_week"]
    }
```

After fitting the model the following information is logged by EMHASS:

```
2024-04-17 12:41:50,019 - web_server - INFO - Passed runtime parameters: {'csv_file':
↪'heating_prediction.csv', 'features': ['degreeday', 'solar'], 'target': 'heating_hours
↪', 'regression_model': 'RandomForestRegression', 'model_type': 'heating_hours_degreeday
↪', 'timestamp': 'timestamp', 'date_features': ['month', 'day_of_week']}
2024-04-17 12:41:50,020 - web_server - INFO -  >> Setting input data dict
2024-04-17 12:41:50,021 - web_server - INFO - Setting up needed data
2024-04-17 12:41:50,048 - web_server - INFO -  >> Performing a machine learning␣
↪regressor fit...
2024-04-17 12:41:50,049 - web_server - INFO - Performing a MLRegressor fit for heating_
↪hours_degreeday
2024-04-17 12:41:50,064 - web_server - INFO - Training a RandomForestRegression model
2024-04-17 12:41:57,852 - web_server - INFO - Elapsed time for model fit: 7.
↪78800106048584
2024-04-17 12:41:57,862 - web_server - INFO - Prediction R2 score of fitted model on␣
↪test data: -0.5667567505914477
```

6.2 The predict method

To obtain a prediction using a previously trained model use the `regressor-model-predict` end point.

The list of parameters needed to set the data publish task is:

- `mlr_predict_entity_id`: The unique `entity_id` to be used.
- `mlr_predict_unit_of_measurement`: The `unit_of_measurement` to be used.
- `mlr_predict_friendly_name`: The `friendly_name` to be used.
- `new_values`: The new values for the features (in the same order as the features list). Also when using date_features, add these to the new values.
- `model_type`: The model type that has to be predicted

6.2.1 Examples:

```
runtimeparams = {
    "mlr_predict_entity_id": "sensor.mlr_predict",
    "mlr_predict_unit_of_measurement": None,
    "mlr_predict_friendly_name": "mlr predictor",
    "new_values": [8.2, 7.23, 2, 6],
    "model_type": "heating_hours_degreeday"
}
```

Pass the correct `model_type` like this:

```
curl -i -H "Content-Type:application/json" -X POST -d '{"new_values": [8.2, 7.23, 2, 6],
↪"model_type": "heating_hours_degreeday" }' http://localhost:5000/action/regressor-
↪model-predict
```

or

```
curl -i -H "Content-Type:application/json" -X POST -d  '{"mlr_predict_entity_id":
↪"sensor.mlr_predict", "mlr_predict_unit_of_measurement": "h", "mlr_predict_friendly_
↪name": "mlr predictor", "new_values": [8.2, 7.23, 2, 6], "model_type": "heating_hours_
↪degreeday" }' http://localhost:5000/action/regressor-model-predict
```

A Home Assistant `rest_command` can look like this:

```
predict_heating_hours:
  url: http://localhost:5001/action/regressor-model-predict
  method: POST
  content_type: "application/json"
  payload: >-
   {
    "mlr_predict_entity_id": "sensor.predicted_hours",
    "mlr_predict_unit_of_measurement": "h",
    "mlr_predict_friendly_name": "Predicted hours",
    "new_values": [8.2, 7.23, 2, 6],
    "model_type": "heating_hours_degreeday"
    }
```

After predicting the model the following information is logged by EMHASS:

```
2024-04-17 14:25:40,695 - web_server - INFO - Passed runtime parameters: {'mlr_predict_
↪entity_id': 'sensor.predicted_hours', 'mlr_predict_unit_of_measurement': 'h', 'mlr_
↪predict_friendly_name': 'Predicted hours', 'new_values': [8.2, 7.23, 2, 6], 'model_type
↪': 'heating_hours_degreeday'}
2024-04-17 14:25:40,696 - web_server - INFO -  >> Setting input data dict
2024-04-17 14:25:40,696 - web_server - INFO - Setting up needed data
2024-04-17 14:25:40,700 - web_server - INFO -  >> Performing a machine learning␣
↪regressor predict...
2024-04-17 14:25:40,715 - web_server - INFO - Performing a prediction for heating_hours_
↪degreeday
2024-04-17 14:25:40,750 - web_server - INFO - Successfully posted to sensor.predicted_
↪hours = 3.716600000000001
```

The predict method will publish the result to a Home Assistant sensor.

6.3 Storing CSV files

6.3.1 Standalone container - how to mount a .csv files in data_path folder

If running EMHASS as Standalone container, you will need to volume mount a folder to be the `data_path`, or mount a single .csv file inside `data_path`

Example of mounting a folder as data_path *(.csv files stored inside)*

```
docker run -it --restart always -p 5000:5000 -e LOCAL_COSTFUN="profit" -v $(pwd)/data:/
↪app/data -v $(pwd)/config_emhass.yaml:/app/config_emhass.yaml -v $(pwd)/secrets_emhass.
↪yaml:/app/secrets_emhass.yaml --name DockerEMHASS <REPOSITORY:TAG>
```

Example of mounting a single csv file

```
docker run -it --restart always -p 5000:5000 -e LOCAL_COSTFUN="profit" -v $(pwd)/data/
↪heating_prediction.csv:/app/data/heating_prediction.csv -v $(pwd)/config_emhass.yaml:/
↪app/config_emhass.yaml -v $(pwd)/secrets_emhass.yaml:/app/secrets_emhass.yaml --name␣
↪DockerEMHASS <REPOSITORY:TAG>
```

6.3.2 Add-on - How to store data in a csv file from Home Assistant

Change data_path

If running EMHASS-Add-On, you will likley need to change the `data_path` to a folder your Home Assistant can access. To do this, set the `data_path` to `/share/` in the addon *Configuration* page.

Store sensor data to csv

Notify to a file

```
notify:
  - platform: file
    name: heating_hours_prediction
    timestamp: false
    filename: /share/heating_prediction.csv
```

Then you need an automation to notify to this file

```
alias: "Heating csv"
id: 157b1d57-73d9-4f39-82c6-13ce0cf42
trigger:
  - platform: time
    at: "23:59:32"
action:
  - service: notify.heating_hours_prediction
    data:
      message: >
        {% set degreeday = states('sensor.degree_day_daily') |float %}
        {% set heating_hours = states('sensor.heating_hours_today') |float | round(2) %}
        {% set solar = states('sensor.solar_daily') |float | round(3) %}
        {% set time = now() %}

          {{time}},{{degreeday}},{{solar}},{{heating_hours}}
```

CHAPTER

SEVEN

EXAMPLE CONFIGURATIONS

In this section example configurations are presented as study cases using real data.

7.1 First test system: a simple system with no PV and two deferrable loads

In this example we will consider a simple system with no PV installation and just two deferrable loads that we want to optimize their schedule.

For this the following parameters can be added to the `secrets.yaml` file: `solar_forecast_kwp: 0`. And also we will set the PV forecast method to `method='solar.forecast'`. This is a simple way to just set a vector with zero values on the PV forecast power, emulating the case where there is no PV installation. The other values on the configuration file are set to their default values.

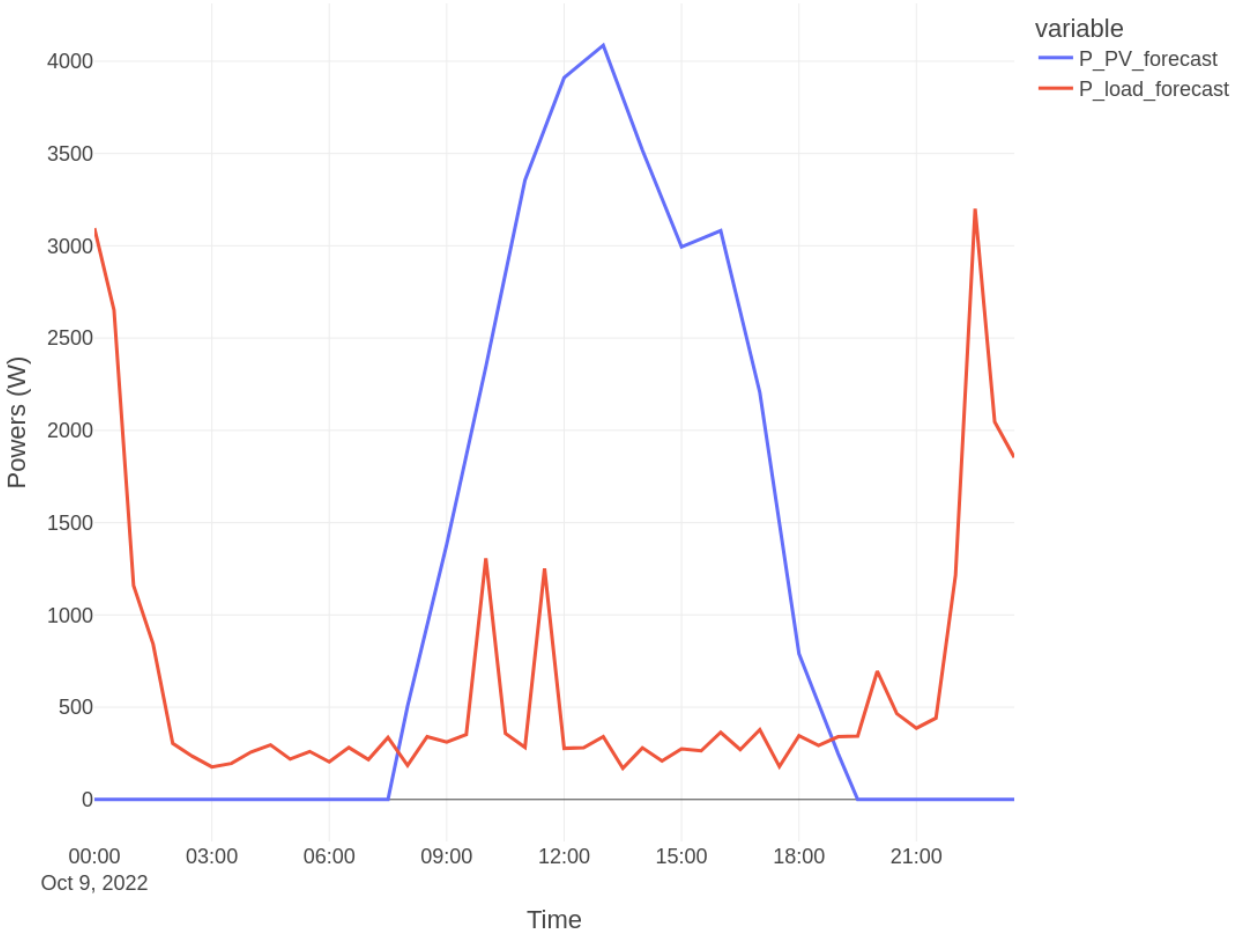
7.1.1 Day-ahead optimization

Let's performa a day-ahead optimization task on this simple system. We want to schedule our two deferrable loads.

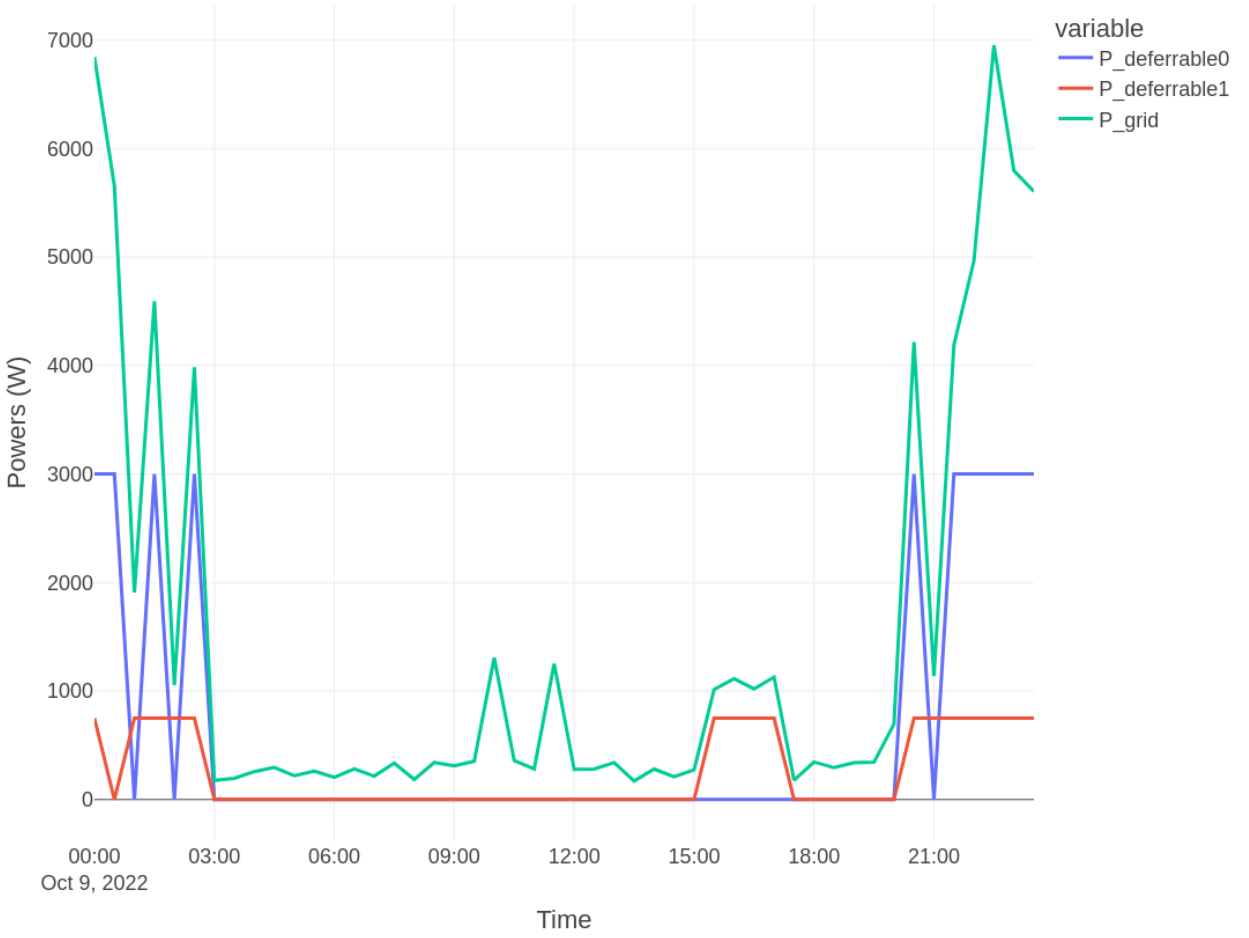
For this we use the following command (example using the legacy EMHASS Python module command line):

```
emhass --action 'dayahead-optim' --config '/home/user/emhass/config_emhass.yaml' --
↪costfun 'profit'
```

The retrieved input forecasted powers are shown below:



Finally, the optimization results are:



For this system the total value of the obtained cost function is -5.38 EUR.

7.2 A second test system: a 5kW PV installation and two deferrable loads

Let's add a 5 kWp solar production with two deferrable loads. No battery is considered for now. The configuration used is the default configuration proposed with EMHASS.

We will first consider a perfect optimization task, to obtain the optimization results with perfectly know PV production and load power values for the last week.

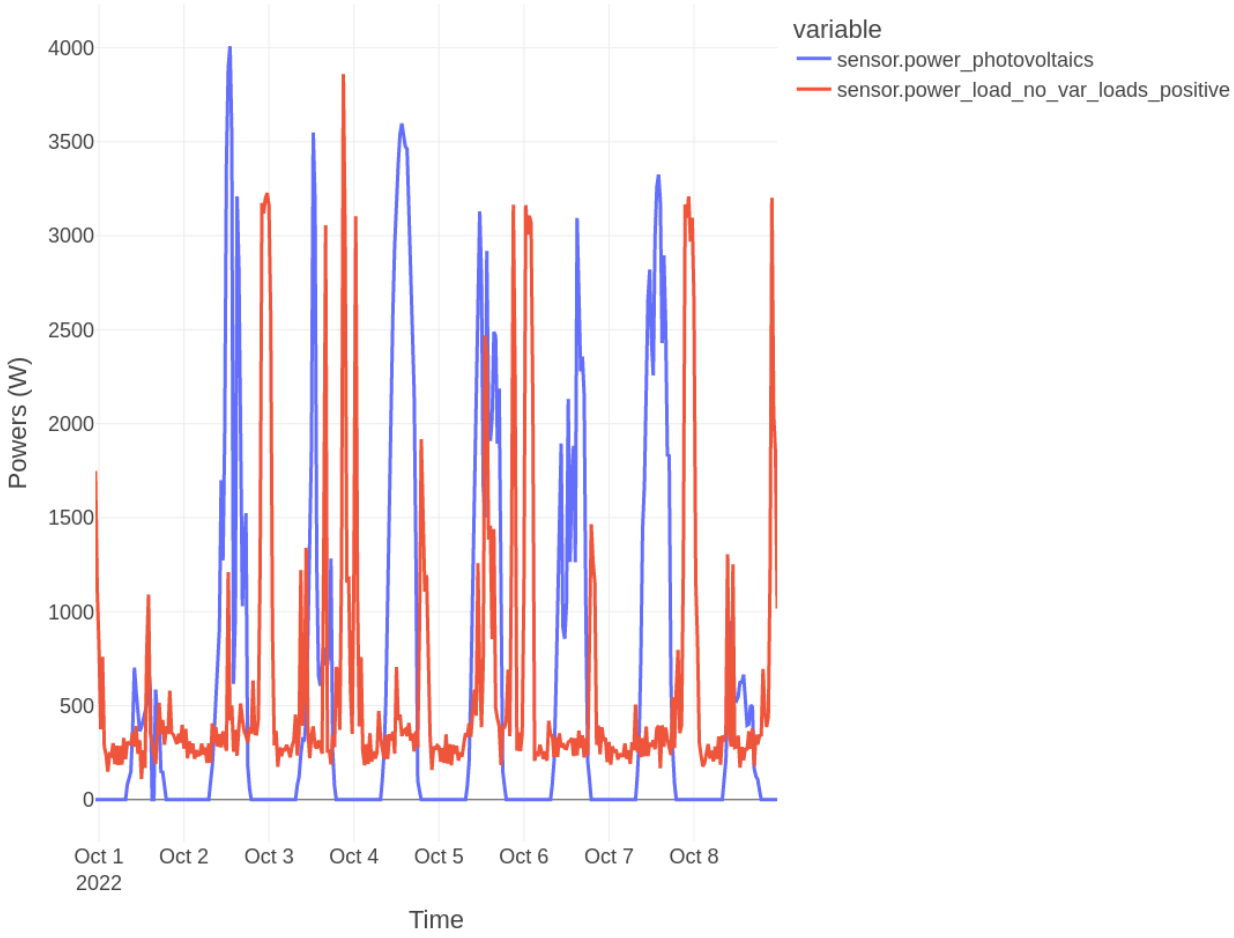
7.2.1 Perfect optimization

Let's perform a 7-day historical data optimization.

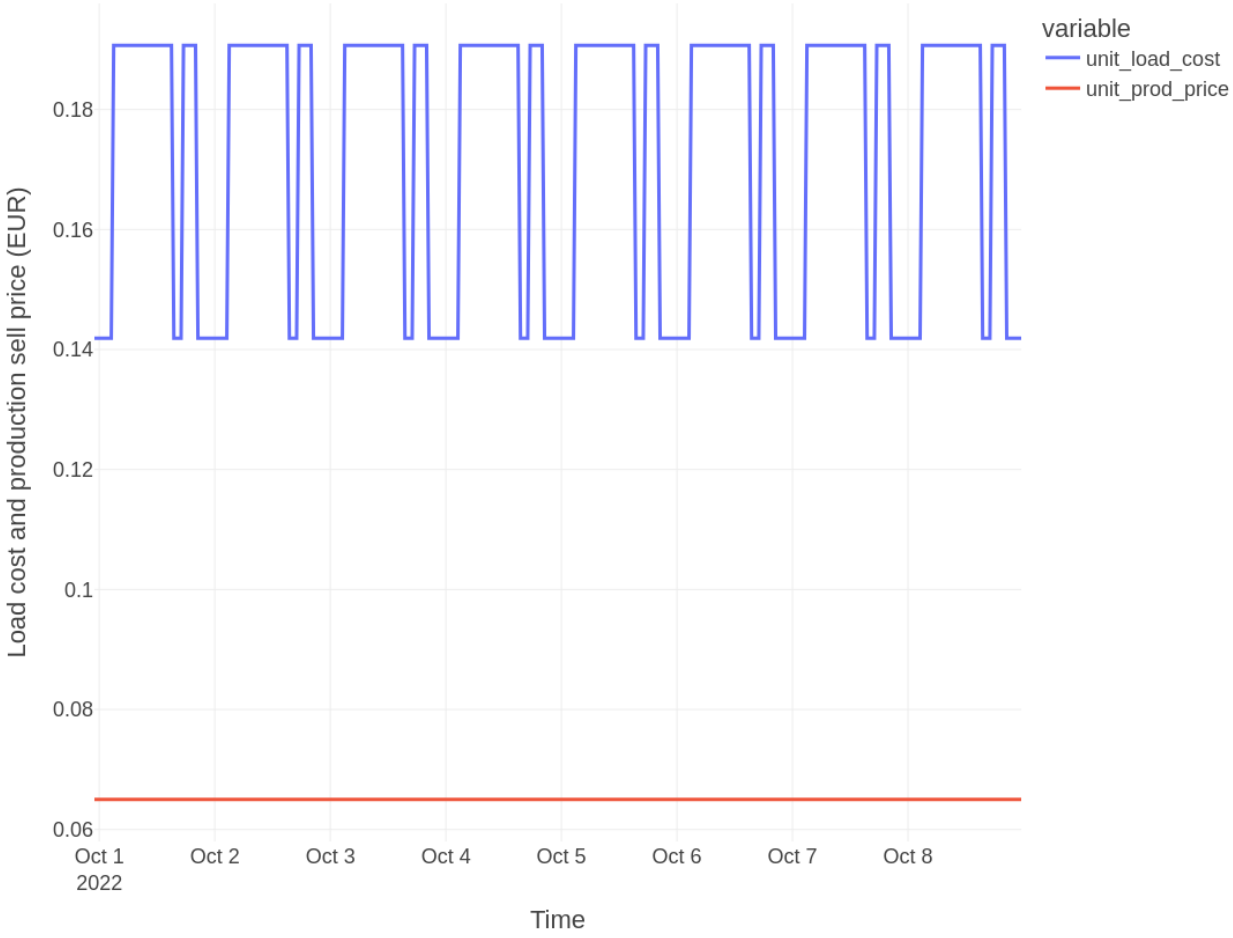
For this we use the following command (example using the legacy EMHASS Python module command line):

```
emhass --action 'perfect-optim' --config '/home/user/emhass/config_emhass.yaml' --
↪costfun 'profit'
```

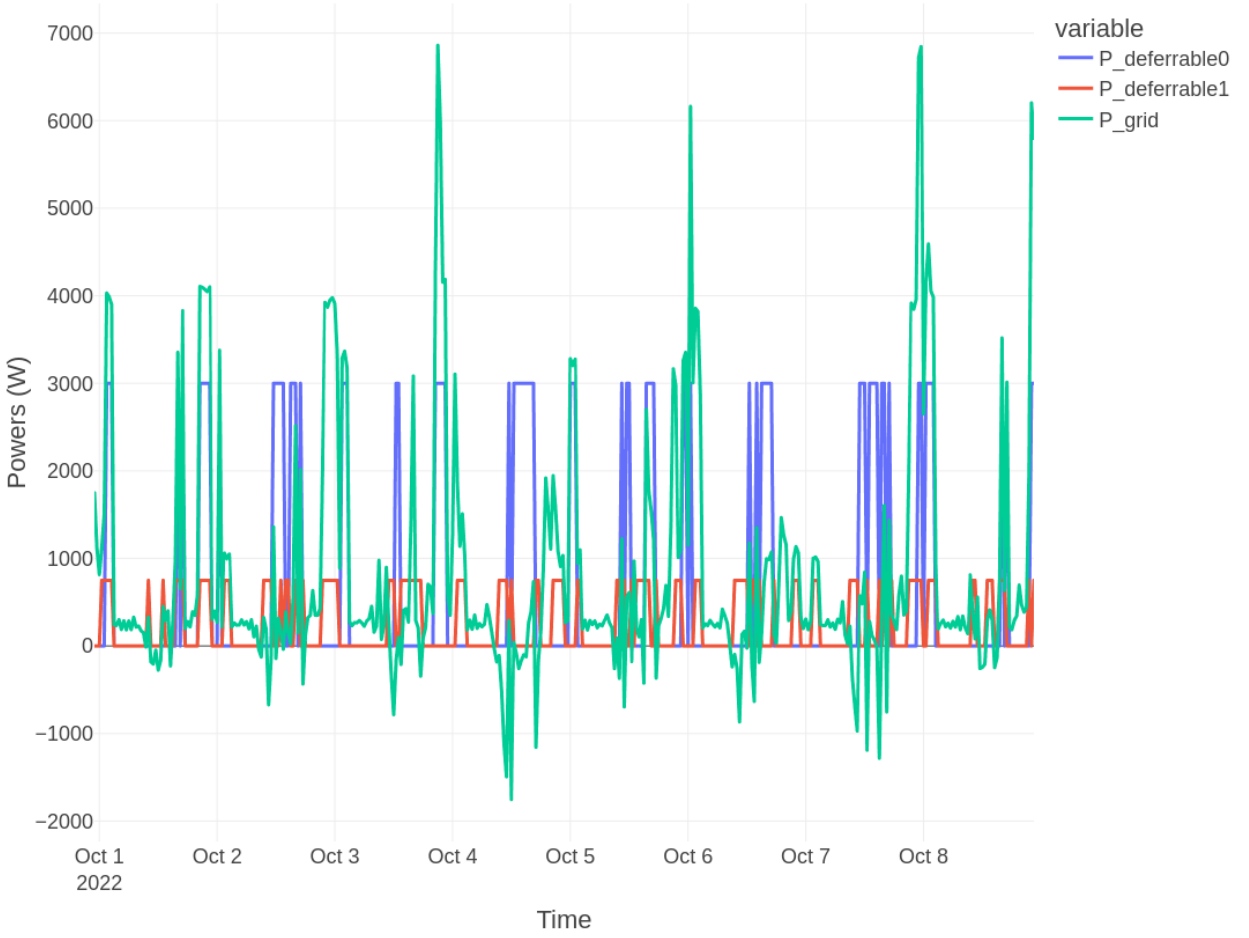
The retrieved input powers are shown below:



The input load cost and PV production selling prices are presented in the following figure:



Finally, the optimization results are:



For this 7-day period, the total value of the cost function was -26.23 EUR.

7.2.2 Day-ahead optimization

As with the simple system we will now perform a day-ahead optimization task. We use again the `dayahead-optim` action or end point.

The optimization results are:

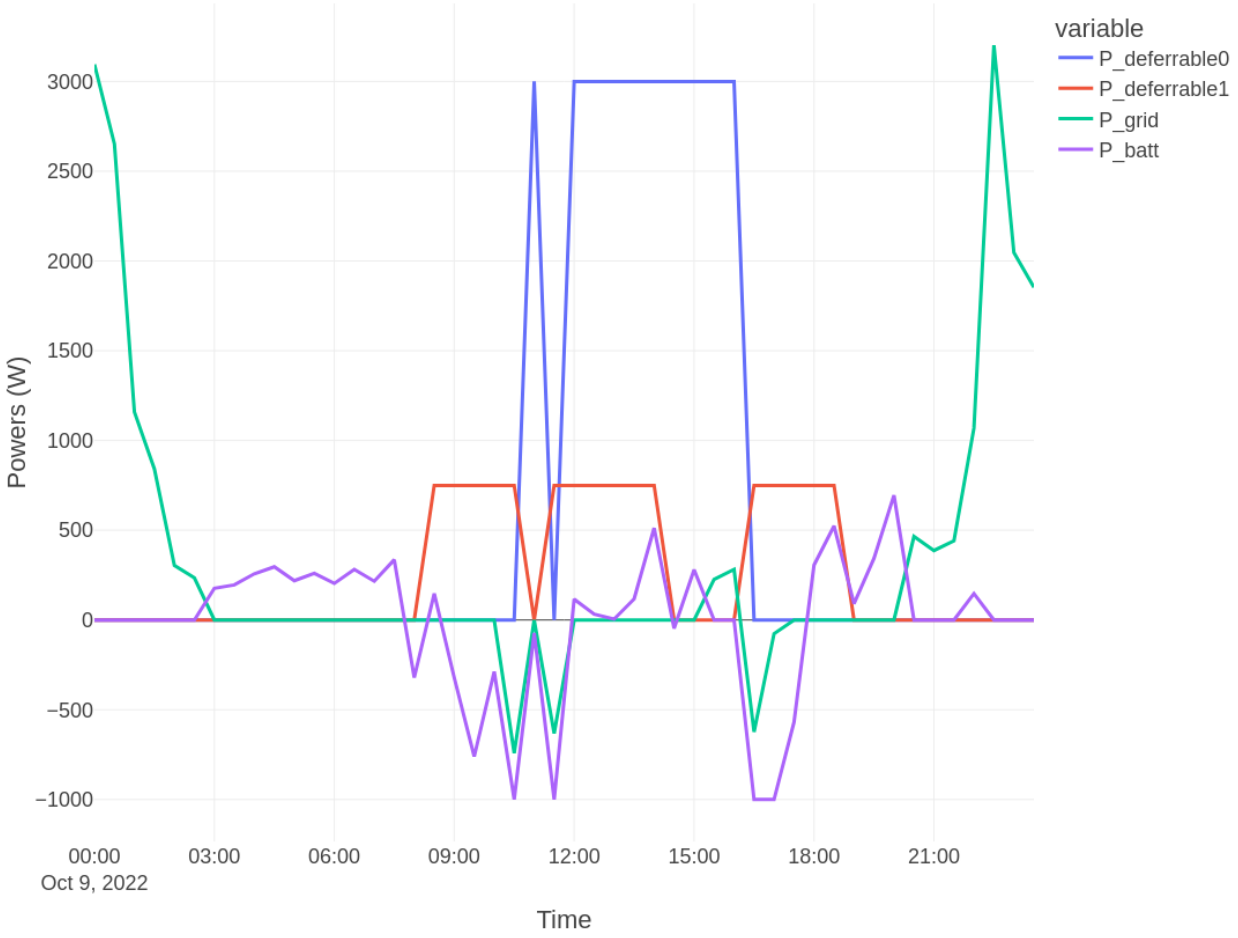
For this system the total value of the obtained cost function is -1.56 EUR. We can note the important improvement on the cost function value whenn adding a PV installation.

7.3 A third test system: a 5kW PV installation, a 5kWh battery and two deferrable loads

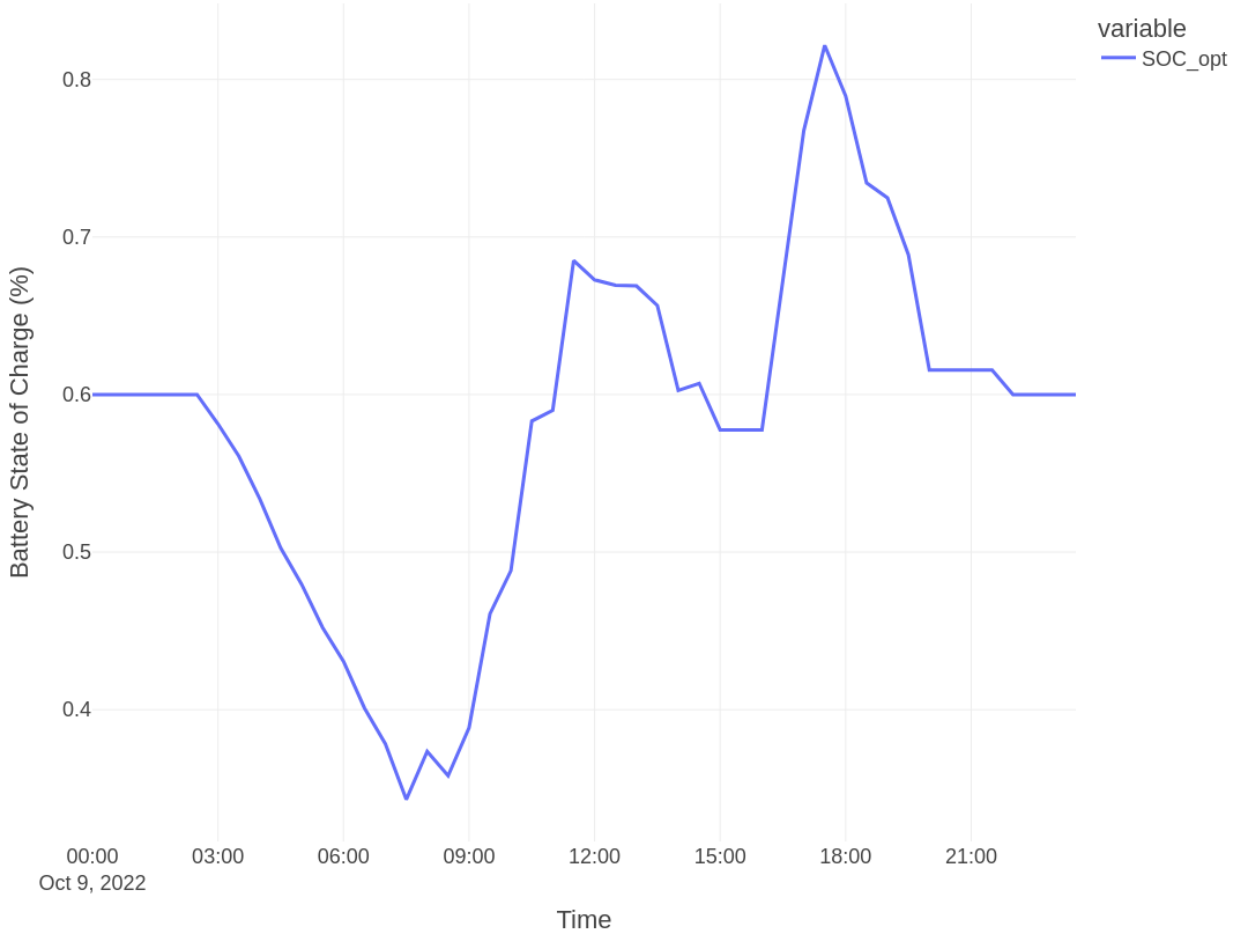
Now we will consider a complet system with PV and added batteries. To add the battery we will set `set_use_battery: true` in the `optim_conf` section of the `config_emhass.yaml` file.

In this case we want to schedule our deferrable loads but also the battery charge/discharge. We use again the `dayahead-optim` action or end point.

The optimization results are:



The battery state of charge plot is shown below:



For this system the total value of the obtained cost function is -1.23 EUR, a substantial improvement when adding a battery.

7.4 Configuration example to pass data at runtime

As we showed in the forecast module section, we can pass our own forecast data using lists of values passed at runtime using templates. However, it is possible to also pass other data during runtime in order to automate the energy management.

For example, let's suppose that for the default configuration with two deferrable loads we want to correlate and control them to the outside temperature. This will be used to build a list of the total number of hours for each deferrable load (`def_total_hours`). In this example the first deferrable load is a water heater and the second is the pool pump.

We will begin by defining a temperature sensor on a 12 hours sliding window using the filter platform for the outside temperature:

```
- platform: filter
  name: "Outdoor temperature mean over last 12 hours"
  entity_id: sensor.temp_air_out
  filters:
    - filter: time_simple_moving_average
```

(continues on next page)

(continued from previous page)

```
      window_size: "12:00"
      precision: 0
```

Then we will use a template sensor to build our list of the total number of hours for each deferrable load:

```
  - platform: template
    sensors:
      list_operating_hours_of_each_deferrable_load:
        value_template: >-
          {% if states("sensor.outdoor_temperature_mean_over_last_12_hours") < "10" %}
            {{ [5, 0] | list }}
          {% elif states("sensor.outdoor_temperature_mean_over_last_12_hours") >= "10"␣
↪and states("sensor.outdoor_temperature_mean_over_last_12_hours") < "15" %}
            {{ [4, 0] | list }}
          {% elif states("sensor.outdoor_temperature_mean_over_last_12_hours") >= "15"␣
↪and states("sensor.outdoor_temperature_mean_over_last_12_hours") < "20" %}
            {{ [4, 6] | list }}
          {% elif states("sensor.outdoor_temperature_mean_over_last_12_hours") >= "20"␣
↪and states("sensor.outdoor_temperature_mean_over_last_12_hours") < "25" %}
            {{ [3, 9] | list }}
          {% else %}
            {{ [3, 12] | list }}
          {% endif %}
```

The values for the total number of operating hours were tuned by trial and error throughout a whole year. These values work fine for a 3000W water heater (the first value of the list) and a 750W pool pump (the second value in the list).

Finally my two shell commands for EMHASS will look like:

```
shell_command:
  dayahead_optim: "curl -i -H \"Content-Type: application/json\" -X POST -d '{\"def_
↪total_hours\":{{states('sensor.list_operating_hours_of_each_deferrable_load')}}}'␣
↪http://localhost:5000/action/dayahead-optim"
  publish_data: "curl -i -H \"Content-Type: application/json\" -X POST -d '{}' http://
↪localhost:5000/action/publish-data"
```

The dedicated automations for these shell commands can be for example:

```
- alias: EMHASS day-ahead optimization
  trigger:
    platform: time
    at: '05:30:00'
  action:
  - service: shell_command.dayahead_optim
- alias: EMHASS publish data
  trigger:
  - minutes: /5
    platform: time_pattern
  action:
  - service: shell_command.publish_data
```

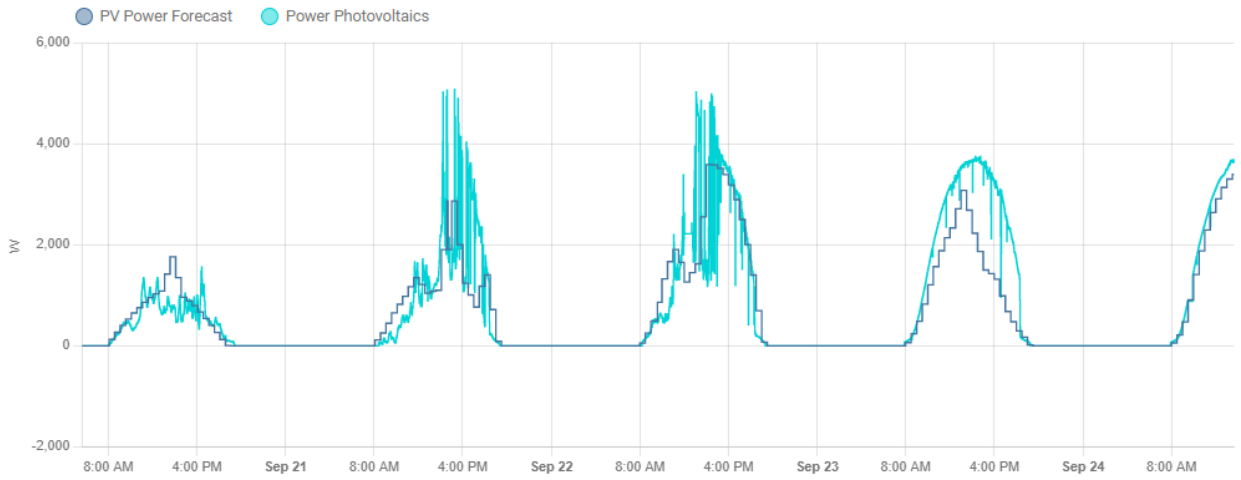
And as a bonus, an automation can be set to relaunch the optimization task automatically. This is very useful when restarting Home Assistant and when updating the EMHASS add-on:

```
- alias: Relaunch EMHASS tasks after HASS restart
  trigger:
  - platform: homeassistant
    event: start
  - platform: state
    entity_id: update.emhass_update
    to: 'off'
    for:
      minutes: 10
  action:
  - service: shell_command.dayahead_optim
  - service: notify.sms_free
    data_template:
      title: EMHASS relaunched optimization
      message: Home assistant restarted or the EMHASS add-on was updated and the␣
↪optimization task was automatically relaunched
```

7.5 Some real forecast data

The real implementation of EMHASS and its efficiency depends on the quality of the forecasted PV power production and the house load consumption.

Here is an extract of the PV power production forecast with the default PV forecast method from EMHASS: a web scarpping of the clearoutside page based on the defined lat/lon location of the system. These are the forecast results of the GFS model compared with the real PV produced data for a 4 day period.



CHAPTER

EIGHT

CONFIGURATION FILE

In this section we will explain all the parts of the `config_emhass.yaml` needed to properly run EMHASS.

We will find three main parts on the configuration file:

- The parameters needed to retrieve data from Home Assistant (retrieve_hass_conf)
- The parameters to define the optimization problem (optim_conf)
- The parameters used to model the system (plant_conf)

8.1 Retrieve HASS data configuration

These are the parameters that we will need to define to retrieve data from Home Assistant. There are no optional parameters. In the case of a list, an empty list is a valid entry.

- `freq`: The time step to resample retrieved data from hass. This parameter is given in minutes. It should not be defined too low or you will run into memory problems when defining the Linear Programming optimization. Defaults to 30.
- `days_to_retrieve`: We will retrieve data from now and up to days_to_retrieve days. Defaults to 2.
- `var_PV`: This is the name of the photovoltaic produced power sensor in Watts from Home Assistant. For example: 'sensor.power_photovoltaics'.
- `var_load`: The name of the household power consumption sensor in Watts from Home Assistant. The deferrable loads that we will want to include in the optimization problem should be substracted from this sensor in HASS. For example: 'sensor.power_load_no_var_loads'
- `load_negative`: Set this parameter to True if the retrived load variable is negative by convention. Defaults to False.
- `set_zero_min`: Set this parameter to True to give a special treatment for a minimum value saturation to zero for power consumption data. Values below zero are replaced by nans. Defaults to True.
- `var_replace_zero`: The list of retrieved variables that we would want to replace nans (if they exist) with zeros. For example:
  - 'sensor.power_photovoltaics'
- `var_interp`: The list of retrieved variables that we would want to interpolate nans values using linear interpolation. For example:
  - 'sensor.power_photovoltaics'
  - 'sensor.power_load_no_var_loads'
- `method_ts_round`: Set the method for timestamp rounding, options are: first, last and nearest.

A second part of this section is given by some privacy-sensitive parameters that should be included in a `secrets_emhass.yaml` file alongside the `config_emhass.yaml` file.

The parameters in the `secrets_emhass.yaml` file are:

- `hass_url`: The URL to your Home Assistant instance. For example: https://myhass.duckdns.org/
- `long_lived_token`: A Long-Lived Access Token from the Lovelace profile page.
- `time_zone`: The time zone of your system. For example: Europe/Paris.
- `lat`: The latitude. For example: 45.0.
- `lon`: The longitude. For example: 6.0
- `alt`: The altitude in meters. For example: 100.0

8.2 Optimization configuration parameters

These are the parameters needed to properly define the optimization problem.

- `set_use_battery`: Set to True if we should consider an energy storage device such as a Li-Ion battery. Defaults to False.
- `delta_forecast`: The number of days for forecasted data. Defaults to 1.
- `num_def_loads`: Define the number of deferrable loads to consider. Defaults to 2.
- `P_deferrable_nom`: The nominal power for each deferrable load in Watts. This is a list with a number of elements consistent with the number of deferrable loads defined before. For example:
  - 3000
  - 750
- `def_total_hours`: The total number of hours that each deferrable load should operate. For example:
  - 5
  - 8
- `def_start_timestep`: The timestep as from which each deferrable load is allowed to operate (if you don't want the deferrable load to use the whole optimization timewindow). If you specify a value of 0 (or negative), the deferrable load will be optimized as from the beginning of the complete prediction horizon window. For example:
  - 0
  - 1
- `def_end_timestep`: The timestep before which each deferrable load should operate. The deferrable load is not allowed to operate after the specified timestep. If a value of 0 (or negative) is provided, the deferrable load is allowed to operate in the complete optimization window). For example:
  - 0
  - 3
- `treat_def_as_semi_cont`: Define if we should treat each deferrable load as a semi-continuous variable. Semi-continuous variables (`True`) are variables that must take a value that can be either their maximum or minimum/zero (for example On = Maximum load, Off = 0 W). Non semi-continuous (which means continuous) variables (`False`) can take any values between their maximum and minimum. For example:
  - True

  - True

- `set_def_constant`: Define if we should set each deferrable load as a constant fixed value variable with just one startup for each optimization task. For example:
  - False
  - False
- `weather_forecast_method`: This will define the weather forecast method that will be used. The options are 'scrapper' for a scrapping method for weather forecast from clearoutside.com and 'csv' to load a CSV file. When loading a CSV file this will be directly considered as the PV power forecast in Watts. The default CSV file path that will be used is '/data/data_weather_forecast.csv'. Defaults to 'scrapper' method.
- `load_forecast_method`: The load forecast method that will be used. The options are 'csv' to load a CSV file or 'naive' for a simple 1-day persistance model. The default CSV file path that will be used is '/data/data_load_forecast.csv'. Defaults to 'naive'.
- `load_cost_forecast_method`: Define the method that will be used for load cost forecast. The options are 'hp_hc_periods' for peak and non-peak hours contracts and 'csv' to load custom cost from CSV file. The default CSV file path that will be used is '/data/data_load_cost_forecast.csv'. The following parameters and definitions are only needed if load_cost_forecast_method='hp_hc_periods':
  - `list_hp_periods`: Define a list of peak hour periods for load consumption from the grid. This is useful if you have a contract with peak and non-peak hours. For example for two peak hour periods:
    * period_hp_1:
      · start: '02:54'
      · end: '15:24'
    * period_hp_2:
      · start: '17:24'
      · end: '20:24'
  - `load_cost_hp`: The cost of the electrical energy from the grid during peak hours in €/kWh. Defaults to 0.1907.
  - `load_cost_hc`: The cost of the electrical energy from the grid during non-peak hours in €/kWh. Defaults to 0.1419.
- `prod_price_forecast_method`: Define the method that will be used for PV power production price forecast. This is the price that is payed by the utility for energy injected to the grid. The options are 'constant' for a constant fixed value or 'csv' to load custom price forecast from a CSV file. The default CSV file path that will be used is '/data/data_prod_price_forecast.csv'.
- `prod_sell_price`: The paid price for energy injected to the grid from excedent PV production in €/kWh. Defaults to 0.065. This parameter is only needed if prod_price_forecast_method='constant'.
- `set_total_pv_sell`: Set this parameter to true to consider that all the PV power produced is injected to the grid. No direct self-consumption. The default is false, for as system with direct self-consumption.
- `lp_solver`: Set the name of the linear programming solver that will be used. Defaults to 'COIN_CMD'. The options are 'PULP_CBC_CMD', 'GLPK_CMD' and 'COIN_CMD'.
- `lp_solver_path`: Set the path to the LP solver. Defaults to '/usr/bin/cbc'.
- `set_nocharge_from_grid`: Set this to true if you want to forbidden to charge the battery from the grid. The battery will only be charged from excess PV.
- `set_nodischarge_to_grid`: Set this to true if you want to forbidden to discharge the battery power to the grid.

- `set_battery_dynamic`: Set a power dynamic limiting condition to the battery power. This is an additional constraint on the battery dynamic in power per unit of time, which allows you to set a percentage of the battery nominal full power as the maximum power allowed for (dis)charge.
- `battery_dynamic_max`: The maximum positive (for discharge) battery power dynamic. This is the allowed power variation (in percentage) of battery maximum power per unit of time.
- `battery_dynamic_min`: The maximum negative (for charge) battery power dynamic. This is the allowed power variation (in percentage) of battery maximum power per unit of time.
- `weight_battery_discharge`: An additional weight (currency/ kWh) applied in cost function to battery usage for discharge. Defaults to 0.00
- `weight_battery_charge`: An additional weight (currency/ kWh) applied in cost function to battery usage for charge. Defaults to 0.00

8.3 System configuration parameters

These are the technical parameters of the energy system of the household.

- `P_from_grid_max`: The maximum power that can be supplied by the utility grid in Watts (consumption). Defaults to 9000.
- `P_to_grid_max`: The maximum power that can be supplied to the utility grid in Watts (injection). Defaults to 9000.

We will define the technical parameters of the PV installation. For the modeling task we rely on the PVLib Python package. For more information see: https://pvlib-python.readthedocs.io/en/stable/ A dedicated webapp will help you search for your correct PV module and inverter names: https://emhass-pvlib-database.streamlit.app/ If your specific model is not found in these lists then solution (1) is to pick another model as close as possible as yours in terms of the nominal power. Solution (2) would be to use SolCast and pass that data directly to emhass as a list of values from a template. Take a look at this example here: https://emhass.readthedocs.io/en/latest/forecasts.html#example-using-solcast-forecast-amber-prices

- `module_model`: The PV module model. For example: 'CSUN_Eurasia_Energy_Systems_Industry_and_Trade_CSUN295_60M'. This parameter can be a list of items to enable the simulation of mixed orientation systems, for example one east-facing array (azimuth=90) and one west-facing array (azimuth=270). When finding the correct model for your installation remember to replace all the special characters in the model name by '_'. The name of the table column for your device on the webapp will already have the correct naming convention.
- `inverter_model`: The PV inverter model. For example: 'Fronius_International_GmbH__Fronius_Primo_5_0_1_208_240__240 This parameter can be a list of items to enable the simulation of mixed orientation systems, for example one east-facing array (azimuth=90) and one west-facing array (azimuth=270). When finding the correct model for your installation remember to replace all the special characters in the model name by '_'. The name of the table column for your device on the webapp will already have the correct naming convention.
- `surface_tilt`: The tilt angle of your solar panels. Defaults to 30. This parameter can be a list of items to enable the simulation of mixed orientation systems, for example one east-facing array (azimuth=90) and one west-facing array (azimuth=270).
- `surface_azimuth`: The azimuth of your PV installation. Defaults to 205. This parameter can be a list of items to enable the simulation of mixed orientation systems, for example one east-facing array (azimuth=90) and one west-facing array (azimuth=270).
- `modules_per_string`: The number of modules per string. Defaults to 16. This parameter can be a list of items to enable the simulation of mixed orientation systems, for example one east-facing array (azimuth=90) and one west-facing array (azimuth=270).

- `strings_per_inverter`: The number of used strings per inverter. Defaults to 1. This parameter can be a list of items to enable the simulation of mixed orientation systems, for example one east-facing array (azimuth=90) and one west-facing array (azimuth=270).

If your system has a battery (set_use_battery=True), then you should define the following parameters:

- `Pd_max`: The maximum discharge power in Watts. Defaults to 1000.
- `Pc_max`: The maximum charge power in Watts. Defaults to 1000.
- `eta_disch`: The discharge efficiency. Defaults to 0.95.
- `eta_ch`: The charge efficiency. Defaults to 0.95.
- `Enom`: The total capacity of the battery stack in Wh. Defaults to 5000.
- `SOCmin`: The minimun allowable battery state of charge. Defaults to 0.3.
- `SOCmax`: The maximum allowable battery state of charge. Defaults to 0.9.
- `SOCtarget`: The desired battery state of charge at the end of each optimization cycle. Defaults to 0.6.

CHAPTER

NINE

API REFERENCE

9.1 emhass.command_line module

emhass.command_line.**dayahead_forecast_optim**(*input_data_dict: dict*, *logger: Logger*, *save_data_to_file: bool | None = False*, *debug: bool | None = False*) → DataFrame

Perform a call to the day-ahead optimization routine.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging object*) – The passed logger object
- **save_data_to_file** (*bool, optional*) – Save optimization results to CSV file
- **debug** (*bool, optional*) – A debug option useful for unittests

Returns
The output data of the optimization

Return type
pd.DataFrame

emhass.command_line.**forecast_model_fit**(*input_data_dict: dict*, *logger: Logger*, *debug: bool | None = False*) → Tuple[DataFrame, DataFrame, *MLForecaster*]

Perform a forecast model fit from training data retrieved from Home Assistant.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging.Logger*) – The passed logger object
- **debug** (*Optional[bool], optional*) – True to debug, useful for unit testing, defaults to False

Returns
The DataFrame containing the forecast data results without and with backtest and the *mlforecaster* object

Return type
Tuple[pd.DataFrame, pd.DataFrame, mlforecaster]

emhass.command_line.**forecast_model_predict**(*input_data_dict: dict*, *logger: Logger*, *use_last_window: bool | None = True*, *debug: bool | None = False*, *mlf: MLForecaster | None = None*) → DataFrame

Perform a forecast model predict using a previously trained skforecast model.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging.Logger*) – The passed logger object
- **use_last_window** (*Optional[bool], optional*) – True if the 'last_window' option should be used for the custom machine learning forecast model. The 'last_window=True' means that the data that will be used to generate the new forecast will be freshly retrieved from Home Assistant. This data is needed because the forecast model is an auto-regressive model with lags. If 'False' then the data using during the model train is used. Defaults to True
- **debug** (*Optional[bool], optional*) – True to debug, useful for unit testing, defaults to False
- **mlf** (*Optional[mlforecaster], optional*) – The 'mlforecaster' object previously trained. This is mainly used for debug and unit testing. In production the actual model will be read from a saved pickle file. Defaults to None

Returns
The DataFrame containing the forecast prediction data

Return type
pd.DataFrame

emhass.command_line.**forecast_model_tune**(*input_data_dict: dict*, *logger: Logger*, *debug: bool | None = False*, *mlf:* MLForecaster *| None = None*) → Tuple[DataFrame, *MLForecaster*]

Tune a forecast model hyperparameters using bayesian optimization.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging.Logger*) – The passed logger object
- **debug** (*Optional[bool], optional*) – True to debug, useful for unit testing, defaults to False
- **mlf** (*Optional[mlforecaster], optional*) – The 'mlforecaster' object previously trained. This is mainly used for debug and unit testing. In production the actual model will be read from a saved pickle file. Defaults to None

Returns
The DataFrame containing the forecast data results using the optimized model

Return type
pd.DataFrame

emhass.command_line.**main**()

Define the main command line entry function.

This function may take several arguments as inputs. You can type *emhass –help* to see the list of options:

- action: Set the desired action, options are: perfect-optim, dayahead-optim, naive-mpc-optim, publish-data, forecast-model-fit, forecast-model-predict, forecast-model-tune
- config: Define path to the config.yaml file
- costfun: Define the type of cost function, options are: profit, cost, self-consumption
- log2file: Define if we should log to a file or not

- params: Configuration parameters passed from data/options.json if using the add-on
- runtimeparams: Pass runtime optimization parameters as dictionnary
- debug: Use True for testing purposes

emhass.command_line.**naive_mpc_optim**(*input_data_dict: dict*, *logger: Logger*, *save_data_to_file: bool | None = False*, *debug: bool | None = False*) → DataFrame

Perform a call to the naive Model Predictive Controller optimization routine.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging object*) – The passed logger object
- **save_data_to_file** (*bool, optional*) – Save optimization results to CSV file
- **debug** (*bool, optional*) – A debug option useful for unittests

Returns
The output data of the optimization

Return type
pd.DataFrame

emhass.command_line.**perfect_forecast_optim**(*input_data_dict: dict*, *logger: Logger*, *save_data_to_file: bool | None = True*, *debug: bool | None = False*) → DataFrame

Perform a call to the perfect forecast optimization routine.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging object*) – The passed logger object
- **save_data_to_file** (*bool, optional*) – Save optimization results to CSV file
- **debug** (*bool, optional*) – A debug option useful for unittests

Returns
The output data of the optimization

Return type
pd.DataFrame

emhass.command_line.**publish_data**(*input_data_dict: dict*, *logger: Logger*, *save_data_to_file: bool | None = False*, *opt_res_latest: DataFrame | None = None*) → DataFrame

Publish the data obtained from the optimization results.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging object*) – The passed logger object
- **save_data_to_file** (*bool, optional*) – If True we will read data from optimization results in dayahead CSV file

Returns
The output data of the optimization readed from a CSV file in the data folder

Return type
pd.DataFrame

emhass.command_line.**regressor_model_fit**(*input_data_dict: dict*, *logger: Logger*, *debug: bool | None = False*) → None

Perform a forecast model fit from training data retrieved from Home Assistant.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging.Logger*) – The passed logger object
- **debug** (*Optional[bool], optional*) – True to debug, useful for unit testing, defaults to False

emhass.command_line.**regressor_model_predict**(*input_data_dict: dict*, *logger: Logger*, *debug: bool | None = False*, *mlr: MLRegressor | None = None*) → None

Perform a prediction from csv file.

Parameters

- **input_data_dict** (*dict*) – A dictionnary with multiple data used by the action functions
- **logger** (*logging.Logger*) – The passed logger object
- **debug** (*Optional[bool], optional*) – True to debug, useful for unit testing, defaults to False

emhass.command_line.**set_input_data_dict**(*emhass_conf: dict*, *costfun: str*, *params: str*, *runtimeparams: str*, *set_type: str*, *logger: Logger*, *get_data_from_file: bool | None = False*) → dict

Set up some of the data needed for the different actions.

Parameters

- **emhass_conf** (*dict*) – Dictionary containing the needed emhass paths
- **costfun** (*str*) – The type of cost function to use for optimization problem
- **params** (*str*) – Configuration parameters passed from data/options.json
- **runtimeparams** (*str*) – Runtime optimization parameters passed as a dictionary
- **set_type** (*str*) – Set the type of setup based on following type of optimization
- **logger** (*logging object*) – The passed logger object
- **get_data_from_file** (*bool, optional*) – Use data from saved CSV file (useful for debug)

Returns
A dictionnary with multiple data used by the action functions

Return type
dict

9.2 emhass.forecast module

class emhass.forecast.**Forecast**(*retrieve_hass_conf: dict*, *optim_conf: dict*, *plant_conf: dict*, *params: str*, *emhass_conf: dict*, *logger: Logger*, *opt_time_delta: int | None = 24*, *get_data_from_file: bool | None = False*)

Bases: `object`

Generate weather, load and costs forecasts needed as inputs to the optimization.

In EMHASS we have basically 4 forecasts to deal with:

- PV power production forecast (internally based on the weather forecast and the characteristics of your PV plant). This is given in Watts.
- Load power forecast: how much power your house will demand on the next 24h. This is given in Watts.
- PV production selling price forecast: at what price are you selling your excess PV production on the next 24h. This is given in EUR/kWh.
- Load cost forecast: the price of the energy from the grid on the next 24h. This is given in EUR/kWh.

There are methods that are generalized to the 4 forecast needed. For all there forecasts it is possible to pass the data either as a passed list of values or by reading from a CSV file. With these methods it is then possible to use data from external forecast providers.

Then there are the methods that are specific to each type of forecast and that proposed forecast treated and generated internally by this EMHASS forecast class. For the weather forecast a first method (*scrapper*) uses a scrapping to the ClearOutside webpage which proposes detailed forecasts based on Lat/Lon locations. This method seems stable but as with any scrape method it will fail if any changes are made to the webpage API. Another method (*solcast*) is using the SolCast PV production forecast service. A final method (*solar.forecast*) is using another external service: Solar.Forecast, for which just the nominal PV peak installed power should be provided. Search the forecast section on the documentation for examples on how to implement these different methods.

The *get_power_from_weather* method is proposed here to convert from irradiance data to electrical power. The PVLib module is used to model the PV plant.

The specific methods for the load forecast are a first method (*naive*) that uses a naive approach, also called persistance. It simply assumes that the forecast for a future period will be equal to the observed values in a past period. The past period is controlled using parameter *delta_forecast*. A second method (*mlforecaster*) uses an internal custom forecasting model using machine learning. There is a section in the documentation explaining how to use this method.

Note: This custom machine learning model is introduced from v0.4.0. EMHASS proposed this new *mlforecaster* class with *fit*, *predict* and *tune* methods. Only the *predict* method is used here to generate new forecasts, but it is necessary to previously fit a forecaster model and it is a good idea to optimize the model hyperparameters using the *tune* method. See the dedicated section in the documentation for more help.

For the PV production selling price and Load cost forecasts the privileged method is a direct read from a user provided list of values. The list should be passed as a runtime parameter during the *curl* to the EMHASS API.

I reading from a CSV file, it should contain no header and the timestamped data should have the following format:

2021-04-29 00:00:00+00:00,287.07

2021-04-29 00:30:00+00:00,274.27

2021-04-29 01:00:00+00:00,243.38

...

The data columns in these files will correspond to the data in the units expected for each forecasting method.

cloud_cover_to_irradiance(*cloud_cover: Series*, *offset: int | None = 35*) → DataFrame

Estimates irradiance from cloud cover in the following steps.

1. Determine clear sky GHI using Ineichen model and climatological turbidity.
2. Estimate cloudy sky GHI using a function of cloud_cover
3. Estimate cloudy sky DNI using the DISC model.
4. Calculate DHI from DNI and GHI.

(This function was copied and modified from PVLib)

Parameters

- **cloud_cover** (*pd.Series*) – Cloud cover in %.
- **offset** (*Optional[int], optional*) – Determines the minimum GHI., defaults to 35

Returns
Estimated GHI, DNI, and DHI.

Return type
pd.DataFrame

get_forecast_days_csv(*timedelta_days: int | None = 1*) → date_range

Get the date range vector of forecast dates that will be used when loading a CSV file.

Returns
The forecast dates vector

Return type
pd.date_range

get_forecast_out_from_csv_or_list(*df_final: DataFrame*, *forecast_dates_csv: date_range*, *csv_path: str*, *data_list: list | None = None*, *list_and_perfect: bool | None = False*) → DataFrame

Get the forecast data as a DataFrame from a CSV file.

The data contained in the CSV file should be a 24h forecast with the same frequency as the main 'freq' parameter in the configuration file. The timestamp will not be used and a new DateTimeIndex is generated to fit the timestamp index of the input data in 'df_final'.

Parameters

- **df_final** (*pd.DataFrame*) – The DataFrame containing the input data.
- **forecast_dates_csv** (*pd.date_range*) – The forecast dates vector
- **csv_path** (*str*) – The path to the CSV file

Returns
The data from the CSV file

Return type
pd.DataFrame

get_load_cost_forecast(*df_final: DataFrame*, *method: str | None = 'hp_hc_periods'*, *csv_path: str | None = 'data_load_cost_forecast.csv'*, *list_and_perfect: bool | None = False*) → DataFrame

Get the unit cost for the load consumption based on multiple tariff periods. This is the cost of the energy from the utility in a vector sampled at the fixed freq value.

Parameters

- **df_final** (*pd.DataFrame*) – The DataFrame containing the input data.
- **method** (*str, optional*) – The method to be used to generate load cost forecast, the options are 'hp_hc_periods' for peak and non-peak hours contractsand 'csv' to load a CSV file, defaults to 'hp_hc_periods'
- **csv_path** (*str, optional*) – The path to the CSV file used when method = 'csv', defaults to "data_load_cost_forecast.csv"

Returns

The input DataFrame with one additionnal column appended containing the load cost for each time observation.

Return type

pd.DataFrame

get_load_forecast(*days_min_load_forecast: int | None = 3*, *method: str | None = 'naive'*, *csv_path: str | None = 'data_load_forecast.csv'*, *set_mix_forecast: bool | None = False*, *df_now: ~pandas.core.frame.DataFrame | None = Empty DataFrame Columns: [] Index: []*, *use_last_window: bool | None = True*, *mlf: ~emhass.machine_learning_forecaster.MLForecaster | None = None*, *debug: bool | None = False*) → Series

Get and generate the load forecast data.

Parameters

- **days_min_load_forecast** (*int, optional*) – The number of last days to retrieve that will be used to generate a naive forecast, defaults to 3
- **method** (*str, optional*) – The method to be used to generate load forecast, the options are 'naive' for a persistance model, 'mlforecaster' for using a custom previously fitted machine learning model, 'csv' to read the forecast from a CSV file and 'list' to use data directly passed at runtime as a list of values. Defaults to 'naive'.
- **csv_path** (*str, optional*) – The path to the CSV file used when method = 'csv', defaults to "/data/data_load_forecast.csv"
- **set_mix_forecast** (*Bool, optional*) – Use a mixed forcast strategy to integra now/current values.
- **df_now** (*pd.DataFrame, optional*) – The DataFrame containing the now/current data.
- **use_last_window** (*Bool, optional*) – True if the 'last_window' option should be used for the custom machine learning forecast model. The 'last_window=True' means that the data that will be used to generate the new forecast will be freshly retrieved from Home Assistant. This data is needed because the forecast model is an auto-regressive model with lags. If 'False' then the data using during the model train is used.
- **mlf** (*mlforecaster, optional*) – The 'mlforecaster' object previously trained. This is mainly used for debug and unit testing. In production the actual model will be read from a saved pickle file.
- **debug** (*Bool, optional*) – The DataFrame containing the now/current data.

Returns

The DataFrame containing the electrical load power in Watts

Return type
pd.DataFrame

static get_mix_forecast(*df_now: DataFrame*, *df_forecast: DataFrame*, *alpha: float*, *beta: float*, *col: str*) → DataFrame

A simple correction method for forecasted data using the current real values of a variable.

Parameters

- **df_now** (*pd.DataFrame*) – The DataFrame containing the current/real values
- **df_forecast** (*pd.DataFrame*) – The DataFrame containing the forecast data
- **alpha** (*float*) – A weight for the forecast data side
- **beta** (*float*) – A weight for the current/real values sied
- **col** (*str*) – The column variable name

Returns
The output DataFrame with the corrected values

Return type
pd.DataFrame

get_power_from_weather(*df_weather: ~pandas.core.frame.DataFrame*, *set_mix_forecast: bool | None = False*, *df_now: ~pandas.core.frame.DataFrame | None = Empty DataFrame Columns: [] Index: []*) → Series

Convert wheater forecast data into electrical power.

Parameters

- **df_weather** (*pd.DataFrame*) – The DataFrame containing the weather forecasted data. This DF should be generated by the 'get_weather_forecast' method or at least contain the same columns names filled with proper data.
- **set_mix_forecast** (*Bool, optional*) – Use a mixed forcast strategy to integra now/current values.
- **df_now** (*pd.DataFrame*) – The DataFrame containing the now/current data.

Returns
The DataFrame containing the electrical power in Watts

Return type
pd.DataFrame

get_prod_price_forecast(*df_final: DataFrame*, *method: str | None = 'constant'*, *csv_path: str | None = 'data_prod_price_forecast.csv'*, *list_and_perfect: bool | None = False*) → DataFrame

Get the unit power production price for the energy injected to the grid.This is the price of the energy injected to the utility in a vector sampled at the fixed freq value.

Parameters

- **df_input_data** (*pd.DataFrame*) – The DataFrame containing all the input data retrieved from hass
- **method** (*str, optional*) – The method to be used to generate the production price forecast, the options are 'constant' for a fixed constant value and 'csv'to load a CSV file, defaults to 'constant'
- **csv_path** (*str, optional*) – The path to the CSV file used when method = 'csv', defaults to "/data/data_load_cost_forecast.csv"

Returns
The input DataFrame with one additionnal column appended containing the power production price for each time observation.

Return type
pd.DataFrame

`get_weather_forecast`(*method: str | None = 'scrapper'*, *csv_path: str | None = 'data_weather_forecast.csv'*) → DataFrame

Get and generate weather forecast data.

Parameters
method (*str, optional*) – The desired method, options are 'scrapper', 'csv', 'list', 'solcast' and 'solar.forecast'. Defaults to 'scrapper'.

Returns
The DataFrame containing the forecasted data

Return type
pd.DataFrame

9.3 emhass.machine_learning_forecaster module

`class emhass.machine_learning_forecaster.MLForecaster`(*data: DataFrame*, *model_type: str*, *var_model: str*, *sklearn_model: str*, *num_lags: int*, *emhass_conf: dict*, *logger: Logger*)

Bases: `object`

A forecaster class using machine learning models with auto-regressive approach and featuresbased on timestamp information (hour, day, week, etc).

This class uses the *skforecast* module and the machine learning models are from *scikit-learn*.

It exposes three main methods:

- *fit*: to train a model with the passed data.
- *predict*: to obtain a forecast from a pre-trained model.
- *tune*: to optimize the models hyperparameters using bayesian optimization.

`static add_date_features`(*data: DataFrame*) → DataFrame

Add date features from the input DataFrame timestamp

Parameters
data (*pd.DataFrame*) – The input DataFrame

Returns
The DataFrame with the added features

Return type
pd.DataFrame

`fit`(*split_date_delta: str | None = '48h'*, *perform_backtest: bool | None = False*) → Tuple[DataFrame, DataFrame]

The fit method to train the ML model.

Parameters

- **split_date_delta** (*Optional[str], optional*) – The delta from now to *split_date_delta* that will be used as the test period to evaluate the model, defaults to '48h'
- **perform_backtest** (*Optional[bool], optional*) – If *True* then a back testing routine is performed to evaluate the performance of the model on the complete train set, defaults to False

Returns
The DataFrame containing the forecast data results without and with backtest

Return type
Tuple[pd.DataFrame, pd.DataFrame]

static generate_exog(*data_last_window*, *periods*, *var_name*)

Generate the exogenous data for future timestamps.

static neg_r2_score(*y_true*, *y_pred*)

The negative of the r2 score.

predict(*data_last_window: DataFrame | None = None*) → Series

The predict method to generate forecasts from a previously fitted ML model.

Parameters
data_last_window (*Optional[pd.DataFrame], optional*) – The data that will be used to generate the new forecast, this will be freshly retrieved from Home Assistant. This data is needed because the forecast model is an auto-regressive model with lags. If not passed then the data used during the model train is used, defaults to None

Returns
A pandas series containing the generated forecasts.

Return type
pd.Series

tune(*debug: bool | None = False*) → DataFrame

Tuning a previously fitted model using bayesian optimization.

Parameters
debug (*Optional[bool], optional*) – Set to True for testing and faster optimizations, defaults to False

Returns
The DataFrame with the forecasts using the optimized model.

Return type
pd.DataFrame

9.4 emhass.optimization module

class emhass.optimization.**Optimization**(*retrieve_hass_conf: dict*, *optim_conf: dict*, *plant_conf: dict*, *var_load_cost: str*, *var_prod_price: str*, *costfun: str*, *emhass_conf: dict*, *logger: Logger*, *opt_time_delta: int | None = 24*)

Bases: object

Optimize the deferrable load and battery energy dispatch problem using the linear programming optimization technique. All equipement equations, including the battery equations are hence transformed in a linear form.

This class methods are:

- perform_optimization
- perform_perfect_forecast_optim
- perform_dayahead_forecast_optim
- perform_naive_mpc_optim

perform_dayahead_forecast_optim(*df_input_data: DataFrame*, *P_PV: Series*, *P_load: Series*) → DataFrame

Perform a day-ahead optimization task using real forecast data. This type of optimization is intented to be launched once a day.

Parameters

- **df_input_data** (*pandas.DataFrame*) – A DataFrame containing all the input data used for the optimization, notably the unit load cost for power consumption.
- **P_PV** (*pandas.DataFrame*) – The forecasted PV power production.
- **P_load** (*pandas.DataFrame*) – The forecasted Load power consumption. This power should not include the power from the deferrable load that we want to find.

Returns
opt_res: A DataFrame containing the optimization results

Return type
pandas.DataFrame

perform_naive_mpc_optim(*df_input_data: DataFrame*, *P_PV: Series*, *P_load: Series*, *prediction_horizon: int*, *soc_init: float | None = None*, *soc_final: float | None = None*, *def_total_hours: list | None = None*, *def_start_timestep: list | None = None*, *def_end_timestep: list | None = None*) → DataFrame

Perform a naive approach to a Model Predictive Control (MPC). This implementaion is naive because we are not using the formal formulation of a MPC. Only the sense of a receiding horizon is considered here. This optimization is more suitable for higher optimization frequency, ex: 5min.

Parameters

- **df_input_data** (*pandas.DataFrame*) – A DataFrame containing all the input data used for the optimization, notably the unit load cost for power consumption.
- **P_PV** (*pandas.DataFrame*) – The forecasted PV power production.
- **P_load** (*pandas.DataFrame*) – The forecasted Load power consumption. This power should not include the power from the deferrable load that we want to find.
- **prediction_horizon** (*int*) – The prediction horizon of the MPC controller in number of optimization time steps.
- **soc_init** (*float*) – The initial battery SOC for the optimization. This parameter is optional, if not given soc_init = soc_final = soc_target from the configuration file.
- **soc_final** – The final battery SOC for the optimization. This parameter is optional, if not given soc_init = soc_final = soc_target from the configuration file.
- **def_total_hours** (*list*) – The functioning hours for this iteration for each deferrable load. (For continuous deferrable loads: functioning hours at nominal power)
- **def_start_timestep** (*list*) – The timestep as from which each deferrable load is allowed to operate.

- **def_end_timestep** (*list*) – The timestep before which each deferrable load should operate.

Returns
opt_res: A DataFrame containing the optimization results

Return type
pandas.DataFrame

perform_optimization(*data_opt: DataFrame*, *P_PV: array*, *P_load: array*, *unit_load_cost: array*, *unit_prod_price: array*, *soc_init: float | None = None*, *soc_final: float | None = None*, *def_total_hours: list | None = None*, *def_start_timestep: list | None = None*, *def_end_timestep: list | None = None*, *debug: bool | None = False*) → DataFrame

Perform the actual optimization using linear programming (LP).

Parameters

- **data_tp** (*pd.DataFrame*) – A DataFrame containing the input data. The results of the optimization will be appended (decision variables, cost function values, etc)
- **P_PV** (*numpy.array*) – The photovoltaic power values. This can be real historical values or forecasted values.
- **P_load** (*np.array*) – The load power consumption values
- **unit_load_cost** (*np.array*) – The cost of power consumption for each unit of time. This is the cost of the energy from the utility in a vector sampled at the fixed freq value
- **unit_prod_price** (*np.array*) – The price of power injected to the grid each unit of time. This is the price of the energy injected to the utility in a vector sampled at the fixed freq value.
- **soc_init** (*float*) – The initial battery SOC for the optimization. This parameter is optional, if not given soc_init = soc_final = soc_target from the configuration file.
- **soc_final** – The final battery SOC for the optimization. This parameter is optional, if not given soc_init = soc_final = soc_target from the configuration file.
- **def_total_hours** (*list*) – The functioning hours for this iteration for each deferrable load. (For continuous deferrable loads: functioning hours at nominal power)
- **def_start_timestep** (*list*) – The timestep as from which each deferrable load is allowed to operate.
- **def_end_timestep** (*list*) – The timestep before which each deferrable load should operate.

Returns
The input DataFrame with all the different results from the optimization appended

Return type
pd.DataFrame

perform_perfect_forecast_optim(*df_input_data: DataFrame*, *days_list: date_range*) → DataFrame

Perform an optimization on historical data (perfectly known PV production).

Parameters

- **df_input_data** (*pandas.DataFrame*) – A DataFrame containing all the input data used for the optimization, notably photovoltaics and load consumption powers.
- **days_list** (*list*) – A list of the days of data that will be retrieved from hass and used for the optimization task. We will retrieve data from now and up to days_to_retrieve days

Returns
opt_res: A DataFrame containing the optimization results

Return type
pandas.DataFrame

static validate_def_timewindow(*start: int*, *end: int*, *min_steps: int*, *window: int*) → Tuple[int, int, str]

Helper function to validate (and if necessary: correct) the defined optimization window of a deferrable load.

Parameters

- **start** (*int*) – Start timestep of the optimization window of the deferrable load
- **end** (*int*) – End timestep of the optimization window of the deferrable load
- **min_steps** (*int*) – Minimal timesteps during which the load should operate (at nominal power)
- **window** (*int*) – Total number of timesteps in the optimization window

Returns
start_validated: Validated start timestep of the optimization window of the deferrable load

Return type
int

Returns
end_validated: Validated end timestep of the optimization window of the deferrable load

Return type
int

Returns
warning: Any warning information to be returned from the validation steps

Return type
string

9.5 emhass.retrieve_hass module

class emhass.retrieve_hass.**RetrieveHass**(*hass_url: str*, *long_lived_token: str*, *freq: Timedelta*, *time_zone: timezone*, *params: str*, *emhass_conf: dict*, *logger: Logger*, *get_data_from_file: bool | None = False*)

Bases: object

Retrieve data from Home Assistant using the restful API.

This class allows the user to retrieve data from a Home Assistant instance using the provided restful API (https://developers.home-assistant.io/docs/api/rest/)

This class methods are:

- get_data: to retrieve the actual data from hass
- prepare_data: to apply some data treatment in preparation for the optimization task
- post_data: Post passed data to hass

static get_attr_data_dict(*data_df: DataFrame*, *idx: int*, *entity_id: str*, *unit_of_measurement: str*, *friendly_name: str*, *list_name: str*, *state: float*) → dict

get_data(*days_list: date_range*, *var_list: list*, *minimal_response: bool | None = False*, *significant_changes_only: bool | None = False*, *test_url: str | None = 'empty'*) → None

Retrieve the actual data from hass.

Parameters

- **days_list** (*pandas.date_range*) – A list of days to retrieve. The ISO format should be used and the timezone is UTC. The frequency of the data_range should be freq='D'
- **var_list** (*list*) – The list of variables to retrive from hass. These should be the exact name of the sensor in Home Assistant. For example: ['sensor.home_load', 'sensor.home_pv']
- **minimal_response** (*bool, optional*) – Retrieve a minimal response using the hass restful API, defaults to False
- **significant_changes_only** (*bool, optional*) – Retrieve significant changes only using the hass restful API, defaults to False

Returns
The DataFrame populated with the retrieved data from hass

Return type
pandas.DataFrame

> **Warning:** The minimal_response and significant_changes_only options are experimental

post_data(*data_df: DataFrame*, *idx: int*, *entity_id: str*, *unit_of_measurement: str*, *friendly_name: str*, *type_var: str*, *from_mlforecaster: bool | None = False*, *publish_prefix: str | None = ''*) → None

Post passed data to hass.

Parameters

- **data_df** (*pd.DataFrame*) – The DataFrame containing the data that will be posted to hass. This should be a one columns DF or a series.
- **idx** (*int*) – The int index of the location of the data within the passed DataFrame. We will post just one value at a time.
- **entity_id** (*str*) – The unique entity_id of the sensor in hass.
- **unit_of_measurement** (*str*) – The units of the sensor.
- **friendly_name** (*str*) – The friendly name that will be used in the hass frontend.
- **type_var** (*str*) – A variable to indicate the type of variable: power, SOC, etc.
- **publish_prefix** (*str, optional*) – A common prefix for all published data entity_id.

prepare_data(*var_load: str*, *load_negative: bool | None = False*, *set_zero_min: bool | None = True*, *var_replace_zero: list | None = None*, *var_interp: list | None = None*) → None

Apply some data treatment in preparation for the optimization task.

Parameters

- **var_load** (*str*) – The name of the variable for the household load consumption.
- **load_negative** (*bool, optional*) – Set to True if the retrived load variable is negative by convention, defaults to False
- **set_zero_min** (*bool, optional*) – A special treatment for a minimum value saturation to zero. Values below zero are replaced by nans, defaults to True

- **var_replace_zero** (*list, optional*) – A list of retrived variables that we would want to replace nans with zeros, defaults to None
- **var_interp** (*list, optional*) – A list of retrived variables that we would want to interpolate nan values using linear interpolation, defaults to None

Returns
The DataFrame populated with the retrieved data from hass and after the data treatment

Return type
pandas.DataFrame

9.6 emhass.utils module

emhass.utils.**build_params**(*params: dict*, *params_secrets: dict*, *options: dict*, *addon: int*, *logger: Logger*) → dict

Build the main params dictionary from the loaded options.json when using the add-on.

Parameters

- **params** (*dict*) – The main params dictionary
- **params_secrets** (*dict*) – The dictionary containing the secret protected variables
- **options** (*dict*) – The load dictionary from options.json
- **addon** (*int*) – A "bool" to select if we are using the add-on
- **logger** (*logging.Logger*) – The logger object

Returns
The builded dictionary

Return type
dict

emhass.utils.**get_days_list**(*days_to_retrieve: int*) → date_range

Get list of past days from today to days_to_retrieve.

Parameters
days_to_retrieve (*int*) – Total number of days to retrieve from the past

Returns
The list of days

Return type
pd.date_range

emhass.utils.**get_forecast_dates**(*freq: int*, *delta_forecast: int*, *timedelta_days: int | None = 0*) → DatetimeIndex

Get the date_range list of the needed future dates using the delta_forecast parameter.

Parameters

- **freq** (*int*) – Optimization time step.
- **delta_forecast** (*int*) – Number of days to forecast in the future to be used for the optimization.
- **timedelta_days** (*Optional[int], optional*) – Number of truncated days needed for each optimization iteration, defaults to 0

Returns
A list of future forecast dates.

Return type
pd.core.indexes.datetimes.DatetimeIndex

emhass.utils.**get_injection_dict**(*df: DataFrame*, *plot_size: int | None = 1366*) → dict

Build a dictionary with graphs and tables for the webui.

Parameters

- **df** (*pd.DataFrame*) – The optimization result DataFrame
- **plot_size** (*Optional[int], optional*) – Size of the plot figure in pixels, defaults to 1366

Returns
A dictionary containing the graphs and tables in html format

Return type
dict

emhass.utils.**get_injection_dict_forecast_model_fit**(*df_fit_pred: DataFrame*, *mlf:* MLForecaster) → dict

Build a dictionary with graphs and tables for the webui for special MLF fit case.

Parameters

- **df_fit_pred** (*pd.DataFrame*) – The fit result DataFrame
- **mlf** (MLForecaster) – The MLForecaster object

Returns
A dictionary containing the graphs and tables in html format

Return type
dict

emhass.utils.**get_injection_dict_forecast_model_tune**(*df_pred_optim: DataFrame*, *mlf:* MLForecaster) → dict

Build a dictionary with graphs and tables for the webui for special MLF tune case.

Parameters

- **df_pred_optim** (*pd.DataFrame*) – The tune result DataFrame
- **mlf** (MLForecaster) – The MLForecaster object

Returns
A dictionary containing the graphs and tables in html format

Return type
dict

emhass.utils.**get_logger**(*fun_name: str*, *emhass_conf: dict*, *save_to_file: bool | None = True*, *logging_level: str | None = 'DEBUG'*) → Tuple[Logger, StreamHandler]

Create a simple logger object.

Parameters

- **fun_name** (*str*) – The Python function object name where the logger will be used
- **emhass_conf** (*dict*) – Dictionary containing the needed emhass paths
- **save_to_file** (*bool, optional*) – Write log to a file, defaults to True

Returns
The logger object and the handler

Return type
object

emhass.utils.**get_root**(*file: str*, *num_parent: int | None = 3*) → str

Get the root absolute path of the working directory.

Parameters

- **file** – The passed file path with __file__
- **num_parent** (*int, optional*) – The number of parents levels up to desired root folder

Returns
The root path

Return type
str

emhass.utils.**get_yaml_parse**(*emhass_conf: dict*, *use_secrets: bool | None = True*, *params: str | None = None*) → Tuple[dict, dict, dict]

Perform parsing of the config.yaml file.

Parameters

- **emhass_conf** (*dict*) – Dictionary containing the needed emhass paths
- **use_secrets** (*bool, optional*) – Indicate if we should use a secrets file or not. Set to False for unit tests.
- **params** (*str*) – Configuration parameters passed from data/options.json

Returns
A tuple with the dictionaries containing the parsed data

Return type
tuple(dict)

emhass.utils.**set_df_index_freq**(*df: DataFrame*) → DataFrame

Set the freq of a DataFrame DateTimeIndex.

Parameters
df (*pd.DataFrame*) – Input DataFrame

Returns
Input DataFrame with freq defined

Return type
pd.DataFrame

emhass.utils.**treat_runtimeparams**(*runtimeparams: str*, *params: str*, *retrieve_hass_conf: dict*, *optim_conf: dict*, *plant_conf: dict*, *set_type: str*, *logger: Logger*) → Tuple[str, dict]

Treat the passed optimization runtime parameters.

Parameters

- **runtimeparams** (*str*) – Json string containing the runtime parameters dict.
- **params** (*str*) – Configuration parameters passed from data/options.json
- **retrieve_hass_conf** (*dict*) – Container for data retrieving parameters.
- **optim_conf** (*dict*) – Container for optimization parameters.

- **plant_conf** (*dict*) – Container for technical plant parameters.
- **set_type** (*str*) – The type of action to be performed.
- **logger** (*logging.Logger*) – The logger object.

Returns
Returning the params and optimization parameter container.

Return type
Tuple[str, dict]

CHAPTER

TEN

EMHASS DEVELOPMENT

There are multiple different approaches to developing EMHASS.
The choice depends on EMHASS mode (standalone/add-on) and preference (Python venv/DevContainer/Docker).
Below are some development workflow examples:
Note: It is preferred to run both addon mode, standalone mode and unitest once before submitting and pull request.

10.1 Step 1 - Fork

With your preferred Git tool of choice:
Fork the EMHASS github repository into your own account, then clone the forked repository into your local development platform. (ie. PC or Codespace)

10.2 Step 2 - Develop

To develop and test code choose one of the following methods:

10.2.1 Method 1 - Python Virtual Environment

We can use python virtual environments to build, develop and test/unitest the code. This method works well with standalone mode.

confirm terminal is in the root `emhass` *directory before starting*

Install requirements

```
python3 -m pip install -r requirements.txt #if arm try setting --extra-index-url=https://
↪www.piwheels.org/simple
```

Create a developer environment:

```
python3 -m venv .venv
```

Activate the environment:

- linux:

  ```
  source .venv/bin/activate
  ```

- windows:

```
.venv\Scripts\activate.bat
```

An IDE like VSCode should automatically catch that a new virtual env was created.

Install the *emhass* package in editable mode:

```
python3 -m pip install -e .
```

Set paths with environment variables:

- Linux

```
export OPTIONS_PATH="${PWD}/options.json" && export USE_OPTIONS="True" ##optional␣
↪to test options.json
export CONFIG_PATH="${PWD}/config_emhass.yaml"
export SECRETS_PATH="${PWD}/secrets_emhass.yaml"
export DATA_PATH="${PWD}/data/"
```

- windows

```
set "OPTIONS_PATH=%cd%/options.json"  & ::  optional to test options.json
set "USE_OPTIONS=True"                & ::  optional to test options.json
set "CONFIG_PATH=%cd%/config_emhass.yaml"
set "SECRETS_PATH=%cd%/secrets_emhass.yaml"
set "DATA_PATH=%cd%/data/"
```

Make sure `secrets_emhass.yaml` *has been created and set. Copy* `secrets_emhass(example).yaml` *for an example.*

Run EMHASS

```
python3 src/emhass/web_server.py
```

Run unitests

```
python3 -m unitest -v -RP -s ./tests -p 'test_*.py'
```

unitest will need to be installed prior

10.2.2 Method 2: VS-Code Debug and Run via DevContainer

In VS-Code, you can run a Docker DevContainer to set up a virtual environment. There you can edit and test EMHASS.

The recommended steps to run are:

- Open forked root (`emhass`) folder inside of VS-Code
- VS-Code will ask if you want to run in a dev-container, say yes *(Dev Container must be set up first). (Shortcut:* `F1 > Dev Containers: Rebuild and Reopen in Container`*)*
- Edit some code…
- Compile emhass by pressing `control+shift+p > Tasks: Run Task > EMHASS Install`. This has been set up in the tasks.json file. - Before *run & debug*, re-run `EMHASS Install` task every time a change has been made to emhass.
- Launch and debug the application via selecting the `Run and Debug` tab /`Ctrl+Shift+D > EMHASS run Addon`. This has been set up in the Launch.json .

  - You will need to modify the `EMHASS_URL` *(http://HAIPHERE:8123/)* and `EMHASS_KEY` *(PLACEKEYHERE)* inside of Launch.json that matches your HA environment before running.
  - If you want to change your parameters, you can edit options.json file before launch.
  - you can also choose to run `EMHASS run` instead of `EMHASS run Addon`. This acts more like standalone mode an removes the use of options.json. *(user sets parameters in config_emhass.yaml instead)*

- You can run all the unitests by heading to the `Testing` tab on the left hand side.
  This is recommended before creating a pull request.

10.2.3 Method 3 - Docker Virtual Environment

With Docker, you can test EMHASS in both standalone and add-on mode via modifying the build argument: `build_version` with values: `standalone`, `addon-pip`, `addon-git`, `addon-local`.
Since emhass-add-on is using the same docker base, this method is good to test the add-on functionality of your code. *(addon-local)*

Depending on your choice of running standalone or addon, `docker run` will require different passed variables/arguments to function. See following examples:

Note: Make sure your terminal is in the root `emhass` directory before running the docker build.

Docker run add-on via with local files:

addon-local copies the local emhass files (from your device) to compile and run in addon mode.

```
docker build -t emhass/docker --build-arg build_version=addon-local .

docker run -it -p 5000:5000 --name emhass-container -e LAT="45.83" -e LON="6.86" -e ALT=
↪"4807.8" -e TIME_ZONE="Europe/Paris" emhass/docker --url YOURHAURLHERE --key␣
↪YOURHAKEYHERE
```

Note:

- `addon` mode can have secret parameters passed in at run via variables `-e`, arguments (`--key`,`--url`) or via `secrets_emhass.yaml` with a volume mount
- on file change, you will need to re-build and re-run the Docker image/container in order for the change to take effect. (excluding volume mounted configs)
- if you are planning to modify the configs: options.json, secrets_emhass.yaml or config_emhass.yaml, you can volume mount them with `-v`:

  ```
  docker build -t emhass/docker --build-arg build_version=addon-local .

  docker run -it -p 5000:5000 --name emhass-container -v $(pwd)/options.json:/app/
  ↪options.json -e LAT="45.83" -e LON="6.86" -e ALT="4807.8" -e TIME_ZONE="Europe/
  ↪Paris" emhass/docker --url YOURHAURLHERE --key YOURHAKEYHERE
  ```

  This allows the editing of config files without re-building the Docker Image. On config change, restart the container to take effect:

  ```
  docker stop emhass-container

  docker start emhass-container
  ```

Docker run Standalone with local files:

standalone copies the local emhass files (from your device) to compile and run in standalone mode.

```
docker build -t emhass/docker --build-arg build_version=standalone .

docker run -it -p 5000:5000 --name emhass-container -v $(pwd)/config_emhass.yaml:/app/
↪config_emhass.yaml -v $(pwd)/secrets_emhass.yaml:/app/secrets_emhass.yaml emhass/docker
```

Standalone mode can use `secrets_emhass.yaml` *to pass secret parameters (overriding secrets provided by ARG/ENV's). Copy* `secrets_emhass(example).yaml` *for an example.*

Docker run add-on with Git or pip:

If you would like to test with the current production/master versions of emhass, you can do so via pip or Git. With Git, you can also specify other repos/branches outside of `davidusb-geek/emhass:master`.

addon-pip will be the closest environment to the production emhass-add-on.
However, both come with the disadvantage of not easily being able to edit the emhass package itself.

Docker run add-on git

```
docker build -t emhass/docker --build-arg build_version=addon-git .

docker run -it -p 5000:5000 --name emhass-container -e LAT="45.83" -e LON="6.86" -e ALT=
↪"4807.8" -e TIME_ZONE="Europe/Paris" -v $(pwd)/options.json:/app/options.json emhass/
↪docker --url YOURHAURLHERE --key YOURHAKEYHERE
```

To test a repo and branch outside of `davidusb-geek/emhass:master`: *(Utilizing build args* `build_repo` *and* `build_branch`*)*
Linux:

```
repo=https://github.com/davidusb-geek/emhass.git
branch=master

docker build -t emhass/docker --build-arg build_version=addon-git --build-arg build_repo=
↪$repo --build-arg build_branch=$branch .

docker run -it -p 5000:5000 --name emhass-container -e LAT="45.83" -e LON="6.86" -e ALT=
↪"4807.8" -e TIME_ZONE="Europe/Paris" -v $(pwd)/options.json:/app/options.json emhass/
↪docker --url YOURHAURLHERE --key YOURHAKEYHERE
```

Docker run add-on pip:

```
docker build -t emhass/docker --build-arg build_version=addon-pip .

docker run -it -p 5000:5000 --name emhass-container -e LAT="45.83" -e LON="6.86" -e ALT=
↪"4807.8" -e TIME_ZONE="Europe/Paris" -v $(pwd)/options.json:/app/options.json emhass/
↪docker --url YOURHAURLHERE --key YOURHAKEYHERE
```

To build with specific pip version, set with build arg: `build_pip_version`:

```
docker build -t emhass/docker --build-arg build_version=addon-pip --build-arg build_pip_
↪version='==0.7.7' .
```

(continues on next page)

(continued from previous page)

```
docker run -it -p 5000:5000 --name emhass-container -e LAT="45.83" -e LON="6.86" -e ALT=
↪"4807.8" -e TIME_ZONE="Europe/Paris" -v $(pwd)/options.json:/app/options.json emhass/
↪docker --url YOURHAURLHERE --key YOURHAKEYHERE
```

You can add or remove file volume mounts with the `-v` *tag, this should override the file in the container (ex. options.json)*

EMHASS older then 0.7.9

For older versions of EMHASS, you may wish to specify the *config*, *data* and *options* paths to avoid errors:

```
docker run ... -e OPTIONS_PATH='/app/options.json' -e CONFIG_PATH='/app/config_emhass.
↪yaml' -e DATA_PATH='/app/data/'  ...
```

For example pip:

```
docker build -t emhass/docker --build-arg build_version=addon-pip .

docker run -it -p 5000:5000 --name emhass-container -e LAT="45.83" -e LON="6.86" -e ALT=
↪"4807.8" -e TIME_ZONE="Europe/Paris"  -e CONFIG_PATH='/app/config_emhass.yaml' -e DATA_
↪PATH='/app/data/' -e OPTIONS_PATH='/app/options.json' -v $(pwd)/options.json:/app/
↪options.json emhass/docker --url YOURHAURLHERE --key YOURHAKEYHERE
```

Sync with local data folder

For those who wish to mount/sync the local `data` folder with the data folder from the docker container, volume mount the data folder with `-v` .

```
docker run ... -v $(pwd)/data/:/app/data ...
```

You can also mount data (ex .csv) files separately

```
docker run... -v $(pwd)/data/heating_prediction.csv:/app/data/ ...
```

Issue with TARGETARCH

If your docker build fails with an error related to `TARGETARCH`. It may be best to add your devices architecture manually:

Example with armhf architecture

```
docker build ... --build-arg TARGETARCH=armhf --build-arg os_version=raspbian ...
```

For `armhf` *only, create a build-arg for* `os_version=raspbian`

Delete built Docker image

We can delete the Docker image and container via:

```
docker rm -f emhass-container #force delete Docker container

docker rmi emhass/docker #delete Docker image
```

Other Docker Options

Rapid Testing
As editing and testing EMHASS via docker may be repetitive (rebuilding image and deleting containers), you may want to simplify the removal, build and run process.

For rapid Docker testing, try a command chain:
Linux:

```
docker build -t emhass/docker --build-arg build_version=addon-local . && docker run --rm␣
↪-it -p 5000:5000 -v $(pwd)/secrets_emhass.yaml:/app/secrets_emhass.yaml --name emhass-
↪container emhass/docker
```

The example command chain rebuilds Docker image, and runs new container with newly built image. `--rm` has been added to the `docker run` to delete the container once ended to avoid manual deletion every time.
This use case may not require any volume mounts (unless you use secrets_emhass.yaml) as the Docker build process will pull the latest versions of the configs as it builds.

Environment Variables
Running addon mode, you can also pass location, key and url secret parameters via environment variables.

```
docker build -t emhass/docker --build-arg build_version=addon-local .

docker run -it -p 5000:5000 --name emhass-container -e URL="YOURHAURLHERE" -e KEY=
↪"YOURHAKEYHERE" -e LAT="45.83" -e LON="6.86" -e ALT="4807.8" -e TIME_ZONE="Europe/Paris
↪" emhass/docker
```

This allows the user to set variables prior to build Linux:

```
export EMHASS_URL="YOURHAURLHERE"
export EMHASS_KEY="YOURHAKEYHERE"
export TIME_ZONE="Europe/Paris"
export LAT="45.83"
export LON="6.86"
export ALT="4807.8"

docker build -t emhass/docker --build-arg build_version=addon-local .

docker run -it -p 5000:5000 --name emhass-container -e EMHASS_KEY -e EMHASS_URL -e TIME_
↪ZONE -e LAT -e LON -e ALT emhass/docker
```

10.2.4 Example Docker testing pipeline

If you are wishing to test your changes compatibility, check out this example as a template:

Linux:
Assuming docker and git installed

```
#setup environment variables for test
export repo=https://github.com/davidusb-geek/emhass.git
export branch=master
#Ex. HAURL=https://localhost:8123/
export HAURL=HOMEASSISTANTURLHERE
export HAKEY=HOMEASSISTANTKEYHERE

git clone $repo
cd emhass
git checkout $branch
```

```
#testing addon (build and run)
docker build -t emhass/docker --build-arg build_version=addon-local .
docker run --rm -it -p 5000:5000 --name emhass-container -v $(pwd)/data/heating_
↪prediction.csv:/app/data/heating_prediction.csv -v $(pwd)/options.json:/app/options.
↪json -e LAT="45.83" -e LON="6.86" -e ALT="4807.8" -e TIME_ZONE="Europe/Paris" emhass/
↪docker --url $HAURL --key $HAKEY
```

```
#run actions on a separate terminal
curl -i -H 'Content-Type:application/json' -X POST -d '{"pv_power_forecast":[0, 70, 141.
↪22, 246.18, 513.5, 753.27, 1049.89, 1797.93, 1697.3, 3078.93], "prediction_horizon":10,
↪ "soc_init":0.5,"soc_final":0.6}' http://localhost:5000/action/naive-mpc-optim
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪perfect-optim
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪dayahead-optim
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪forecast-model-fit
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪forecast-model-predict
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪forecast-model-tune
curl -i -H "Content-Type:application/json" -X POST -d  '{"csv_file": "heating_prediction.
↪csv", "features": ["degreeday", "solar"], "target": "hour", "regression_model":
↪"RandomForestRegression", "model_type": "heating_hours_degreeday", "timestamp":
↪"timestamp", "date_features": ["month", "day_of_week"], "new_values": [12.79, 4.766, 1,
↪ 2] }' http://localhost:5000/action/regressor-model-fit
curl -i -H "Content-Type:application/json" -X POST -d  '{"mlr_predict_entity_id":
↪"sensor.mlr_predict", "mlr_predict_unit_of_measurement": "h", "mlr_predict_friendly_
↪name": "mlr predictor", "new_values": [8.2, 7.23, 2, 6], "model_type": "heating_hours_
↪degreeday" }' http://localhost:5000/action/regressor-model-predict
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪publish-data
```

```
#testing standalone (build and run)
docker build -t emhass/docker --build-arg build_version=standalone .
```

(continues on next page)

(continued from previous page)

```
#make secrets_emhass
cat <<EOT > secrets_emhass.yaml
hass_url: $HAURL
long_lived_token: $HAKEY
time_zone: Europe/Paris
lat: 45.83
lon: 6.86
alt: 4807.8
EOT
docker run --rm -it -p 5000:5000 --name emhass-container -v $(pwd)/data/heating_
↪prediction.csv:/app/data/heating_prediction.csv -v $(pwd)/config_emhass.yaml:/app/
↪config_emhass.yaml -v $(pwd)/secrets_emhass.yaml:/app/secrets_emhass.yaml emhass/
↪docker
```

```
#run actions on a separate terminal
curl -i -H 'Content-Type:application/json' -X POST -d '{"pv_power_forecast":[0, 70, 141.
↪22, 246.18, 513.5, 753.27, 1049.89, 1797.93, 1697.3, 3078.93], "prediction_horizon":10,
↪ "soc_init":0.5,"soc_final":0.6}' http://localhost:5000/action/naive-mpc-optim
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪perfect-optim
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪dayahead-optim
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪forecast-model-fit
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪forecast-model-predict
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪forecast-model-tune
curl -i -H "Content-Type:application/json" -X POST -d  '{"csv_file": "heating_prediction.
↪csv", "features": ["degreeday", "solar"], "target": "hour", "regression_model":
↪"RandomForestRegression", "model_type": "heating_hours_degreeday", "timestamp":
↪"timestamp", "date_features": ["month", "day_of_week"], "new_values": [12.79, 4.766, 1,
↪ 2] }' http://localhost:5000/action/regressor-model-fit
curl -i -H "Content-Type:application/json" -X POST -d  '{"mlr_predict_entity_id":
↪"sensor.mlr_predict", "mlr_predict_unit_of_measurement": "h", "mlr_predict_friendly_
↪name": "mlr predictor", "new_values": [8.2, 7.23, 2, 6], "model_type": "heating_hours_
↪degreeday" }' http://localhost:5000/action/regressor-model-predict
curl -i -H 'Content-Type:application/json' -X POST -d {} http://localhost:5000/action/
↪publish-data
```

User may wish to re-test with tweaked parameters such as `lp_solver`, `weather_forecast_method` and `load_forecast_method`, in `config_emhass.yaml` *(standalone)* or `options.json` *(addon)*, to broaden the testing scope. *see EMHASS & EMHASS-Add-on differences for more information on how these config_emhass & options files differ*

Note: may need to set `--build-arg TARGETARCH=YOUR-ARCH` in docker build

10.3 Step 3 - Pull request

Once developed, commit your code, and push to your fork. Then submit a pull request with your fork to the davidusb-geek/emhass@master repository.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

N

O

P

R

S

T

V