emhass

Release 0.1.0

David HERNANDEZ

Nov 19, 2021

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

INTRODUCTION

This module was conceived as an energy management optimization tool for residential electric power consumption and production systems. The goal is to optimize the energy use in order to maximize self-consumption. The main study case is a household where we have solar panels, a grid connection and one or more controllable (deferrable) electrical loads. Including an energy storage system using batteries is also possible in the code. The package is 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.

The main dependencies of this project are PVLib to model power from a PV residential installation and the PuLP Python package to perform the actual optimizations using the Linear Programming approach.

The source code for this package is available here.

1.1 Installation

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.yaml` and `secrets.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
```

1.2 Usage

To run a command simply use the `emhass` 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` and `publish-data`
- `--config`: Define path to the config.yaml file
- `--costfun`: Define the type of cost function, options are: `profit`, `cost`, `self-consumption`

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

```
emhass --action 'dayahead-optim' --config '/home/user/emhass' --costfun 'profit'
```

Before running any valuable command you need to modify the `config.yaml` and `secrets.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.

1.3 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 `configuration.yaml`:

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

And 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 optimization is performed everyday at 5:30am and the data is published every 5 minutes. 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'
```

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'
```

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
```

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 have 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:
  trigger:
    - platform: numeric_state
      entity_id:
        - sensor.p_deferrable1
      above: 0.1
  action:
    - service: homeassistant.turn_on
      entity_id: switch.water_heater
```

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

```
automation:
  trigger:
    - platform: numeric_state
      entity_id:
        - sensor.p_deferrable1
      below: 0.1
  action:
    - service: homeassistant.turn_off
      entity_id: switch.water_heater
```

CHAPTER

TWO

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).

2.1 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.

2.2 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).

2.3 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.

2.3.1 Cost functions

The profit cost function:

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

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

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.

The energy from the grid cost:

In this case the cost function is computed as the cost of the energy comming from the grid. This is:

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

The self-consumption cost function:

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

The problem constraints are written as follows.

2.3.2 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.

2.3.3 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.

2.3.4 Perfect forecast optimization

This is the first of two 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.

2.3.5 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.

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.

We are now ready to configure our system using the proposed configuration file and link our package to Home Assistant!

2.4 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

THREE

A REAL STUDY CASE

In this section a study case is presented.

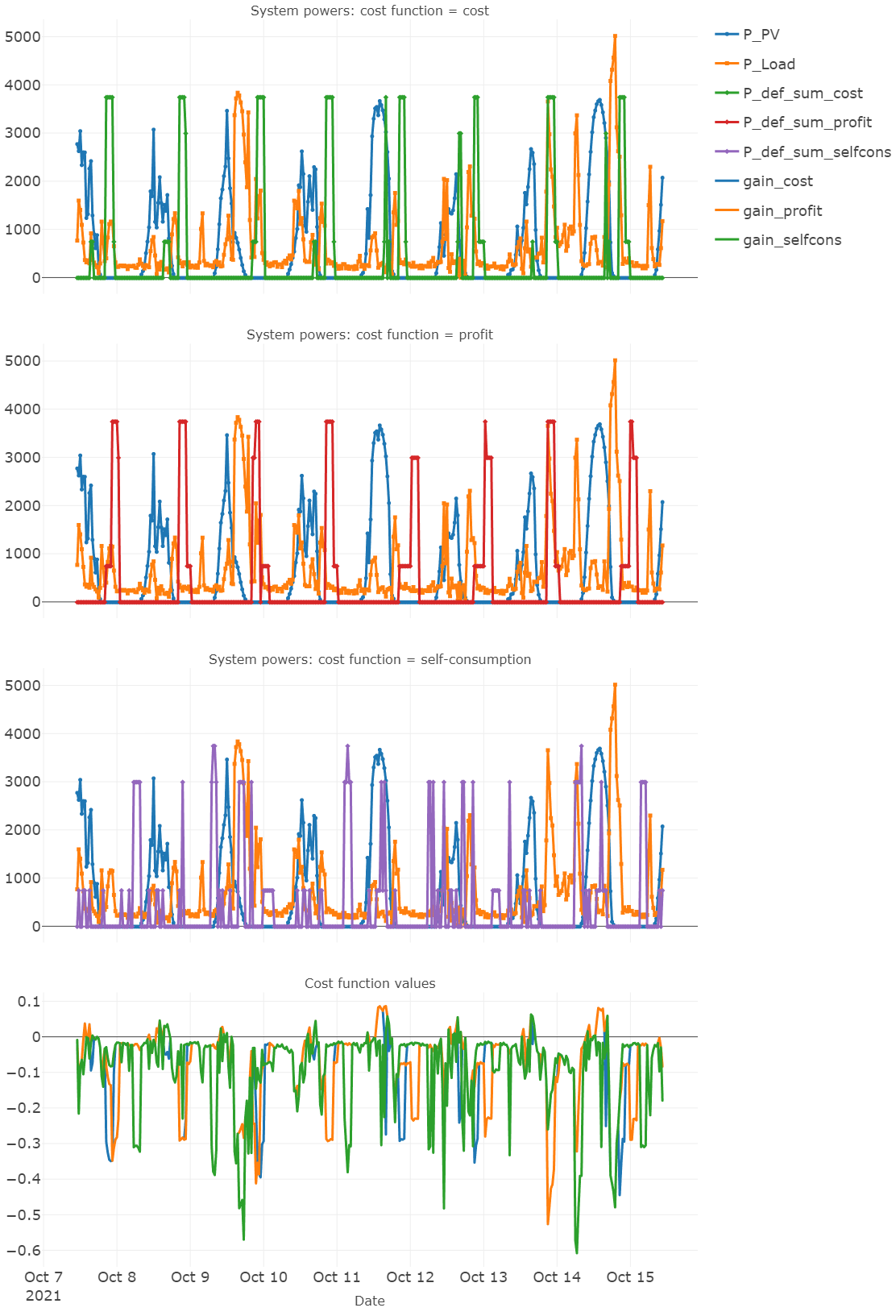
For this example a 5 kWp solar production is considered with two deferrable loads. No battery is considered. The configuration used is the default configuration proposed with EMHASS. We will use 7-day historical data for this optimization.

We compare the results obtained with the perfect optimization using the following command:

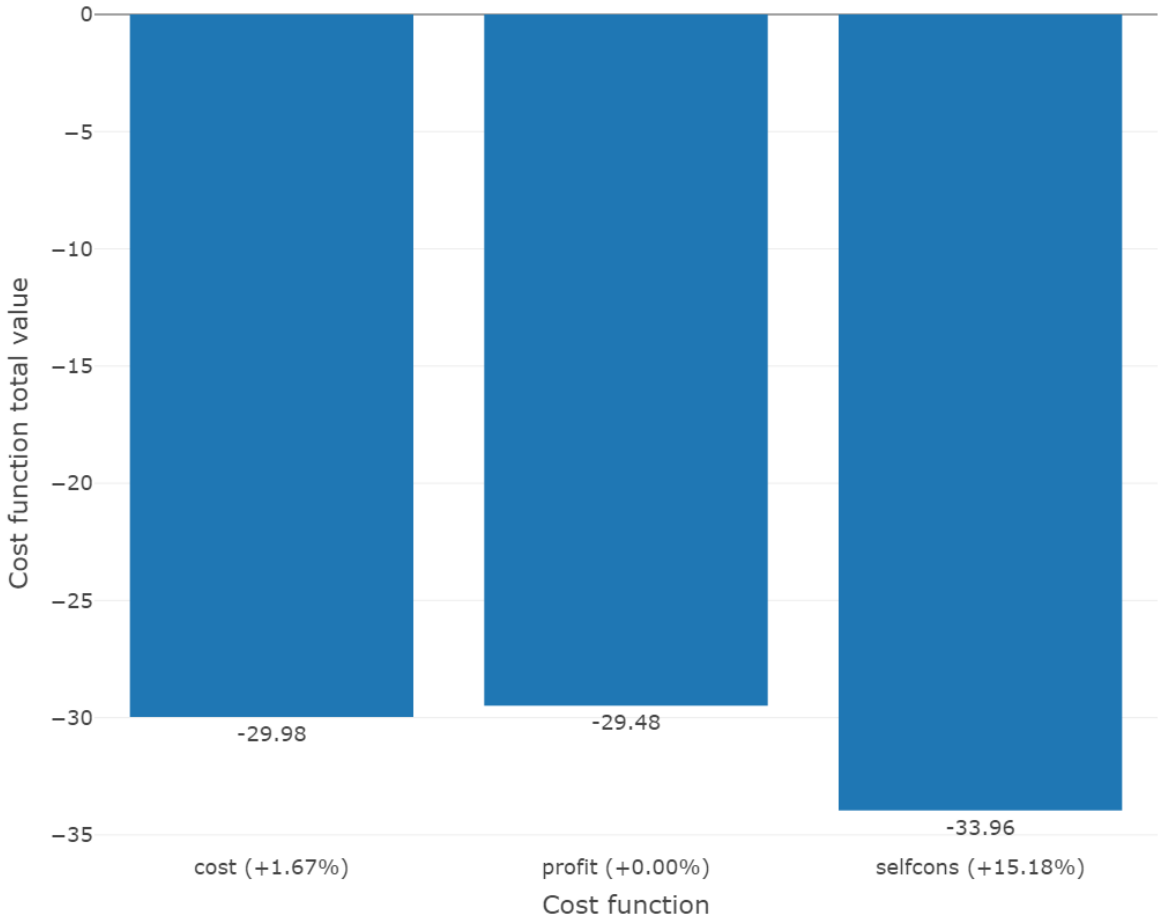
```
emhass --action 'perfect-optim' --config '/home/user/emhass' --costfun 'profit'
```

The `--costfun` is compared with all the possible options: `profit`, `cost`, `self-consumption`.

The obtained results are presented below. These results shows different behaviors of the optimization algorithm depending on the cost function.


System powers: cost function = cost
5000
4000
3000
2000
1000
0
P_PV
P_Load
P_def_sum_cost
P_def_sum_profit
P_def_sum_selfcons
gain_cost
gain_profit
gain_selfcons
System powers: cost function = profit
5000
4000
3000
2000
1000
0
System powers: cost function = self-consumption
5000
4000
3000
2000
1000
0
Cost function values
0.1
0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
Oct 7
2021
Oct 8
Oct 9
Oct 10
Oct 11
Oct 12
Oct 13
Oct 14
Oct 15
Date

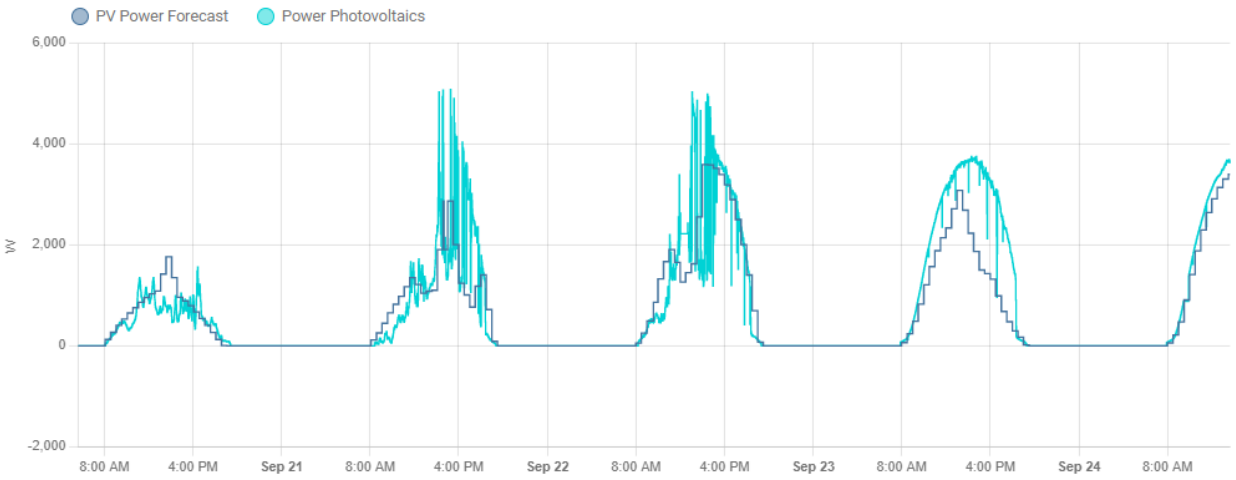
In this particular case, for comparison purposes, we compute the profit value for each cost function. The results are presented below:



We can see that for this particular case, the profit cost function is the most interesting if we focus on maximizing profit. However this can differ depending on your specific usage of your PV production. In some cases it could be interesting to maximize self-consumption, for example in off-grid applications.

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.


PV Power Forecast
Power Photovoltaics
6,000
4,000
2,000
0
-2,000
W
8:00 AM
4:00 PM
Sep 21
8:00 AM
4:00 PM
Sep 22
8:00 AM
4:00 PM
Sep 23
8:00 AM
4:00 PM
Sep 24
8:00 AM

CHAPTER

FOUR

CONFIGURATION FILE

In this section we will explain all the parts of the `config.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)

4.1 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'

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

The parameters in the `secrets.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

4.2 Optimization configuration parameters

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

- 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
- treat_def_as_semi_cont: Define if we should treat each deferrable load as a semi-continuous variable. Semi-continuous variables are variables that must take a value between their minimum and maximum or zero. 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
- 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 0.1464.
- load_cost_hc: The cost of the electrical energy from the grid during non-peak hours in €/kWh. Defaults to 0.096359.
- prod_sell_price: The paid price for energy injected to the grid from excedent PV production in €/kWh. Defaults to 0.065.
- 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.

4.3 System configuration parameters

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

- P_grid_max: The maximum power that can be supplied by the utility grid in Watts. 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/ The complete list of supported modules and inverter models can be found here: https://pvlib-python.readthedocs.io/en/stable/generated/pvlib.pvsystem.retrieve_sam.html

- module_model: The PV module model. For example: 'CSUN_Eurasia_Energy_Systems_Industry_and_Trade_CSUN295_60M'
- inverter_model: The PV inverter model. For example: 'Fronius_International_GmbH__Fronius_Primo_5_0_1_208_240__240V_'
- surface_tilt: The tilt angle of your solar panels. Defaults to 30.
- surface_azimuth: The azimuth of your PV installation. Defaults to 205.
- modules_per_string: The number of modules per string. Defaults to 16.
- strings_per_inverter: The number of used strings per inverter. Defaults to 1.

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 minimun 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

FIVE

API REFERENCE

5.1 emhass.command_line module

emhass.command_line.**dayahead_forecast_optim**(*input_data_dict*, *logger*)
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

Returns The output data of the optimization

Return type pd.DataFrame

emhass.command_line.**main**()
Define the main command line entry function.

emhass.command_line.**perfect_forecast_optim**(*input_data_dict*, *logger*)
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

Returns The output data of the optimization

Return type pd.DataFrame

emhass.command_line.**publish_data**(*input_data_dict*, *logger*)
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

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

Return type pd.DataFrame

emhass.command_line.**setUp**(*config_path*, *costfun*, *logger*)
Set up some of the data needed for the different actions.

Parameters

- **config_path** (*str*) – The absolute path where the config.yaml file is located
- **costfun** (*str*) – The type of cost function to use for optimization problem
- **logger** (*logging object*) – The passed logger object

Returns A dictionnary with multiple data used by the action functions

Return type dict

5.2 emhass.forecast module

class emhass.forecast.**forecast**(*retrieve_hass_conf: dict*, *optim_conf: dict*, *plant_conf: dict*, *config_path: str*, *logger: logging.Logger*, *opt_time_delta: Optional[int] = 24*)

Bases: object

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

The weather forecast is obtained from two methods. In the first method the tools proposed in the PVLib package are use to retrieve data from the GFS global model. However this API proposed by PVLib is highly experimental and prone to changes. A second method uses a scrapper to the ClearOutside webpage which proposes detailed forecasts based on Lat/Lon locations. This methods seems quite stable but as any scrape method it will fail if any changes are made to the webpage API.

The 'get_power_from_weather' method is proposed here to convert from irradiance data to electrical power. Again PVLib is used to model the PV plant. For the load forecast two methods are available.

The first method allows the user to use a CSV file with their own forecast. With this method a more powerful external package for time series forecast may be used to create your own detailed load forecast.

The CSV 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 second method is a naive method, 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'.

get_load_forecast(*days_min_load_forecast: Optional[int] = 3*, *method: Optional[str] = 'naive'*, *csv_path: Optional[str] = '/data/data_load_forecast.csv'*) → pandas.core.series.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 'csv' to load a CSV file or 'naive' for a persistance model, defaults to 'naive'
- **csv_path** (*str, optional*) – The path to the CSV file used when method = 'csv', defaults to "/data/data_load_forecast.csv"

Returns The DataFrame containing the electrical load power in Watts

Return type pd.DataFrame

get_power_from_weather(*df_weather: pandas.core.frame.DataFrame*) → pandas.core.series.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.

Returns The DataFrame containing the electrical power in Watts

Return type pd.DataFrame

get_weather_forecast(*method: Optional[str] = 'scrapper'*) → pandas.core.frame.DataFrame
Get and generate weather forecast data.

Parameters **method** (*str, optional*) – The desired method, options are 'scrapper' and 'pvlib', defaults to 'scrapper'

Returns The DataFrame containing the forecasted data

Return type pd.DataFrame

5.3 emhass.optimization module

class emhass.optimization.**optimization**(*retrieve_hass_conf: dict*, *optim_conf: dict*, *plant_conf: dict*, *days_list: pandas.core.indexes.datetimes.date_range*, *costfun: str*, *config_path: str*, *logger: logging.Logger*, *opt_time_delta: Optional[int] = 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:

- get_load_unit_cost
- perform_optimization
- perform_perfect_forecast_optim
- perform_dayahead_forecast_optim

get_load_unit_cost(*df_final: pandas.core.frame.DataFrame*) → pandas.core.frame.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_input_data** (*pd.DataFrame*) – The DataFrame containing all the input data retrieved from hass

Returns The input DataFrame with one additionnal column appended containing the load cost by unit of time

Return type pd.DataFrame

perform_dayahead_forecast_optim(*df_input_data: pandas.core.frame.DataFrame*, *P_PV: pandas.core.series.Series*, *P_load: pandas.core.series.Series*) → pandas.core.frame.DataFrame

Perform a day-ahead optimization task using real forecast data.

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_optimization(*data_opt: pandas.core.frame.DataFrame*, *P_PV: numpy.array*, *P_load: numpy.array*, *unit_load_cost: numpy.array*) → pandas.core.frame.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

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

Return type pd.DataFrame

perform_perfect_forecast_optim(*df_input_data: pandas.core.frame.DataFrame*) → pandas.core.frame.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.

Returns opt_res: A DataFrame containing the optimization results

Return type pandas.DataFrame

5.4 emhass.retrieve_hass module

class emhass.retrieve_hass.**retrieve_hass**(*hass_url: str*, *long_lived_token: str*, *freq: pandas._libs.tslibs.timedeltas.Timedelta*, *time_zone: datetime.timezone*, *config_path: str*, *logger: logging.Logger*)

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

get_data(*days_list: pandas.core.indexes.datetimes.date_range*, *var_list: list*, *minimal_response: Optional[bool] = False*, *significant_changes_only: Optional[bool] = False*) → 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: pandas.core.frame.DataFrame*, *idx: int*, *entity_id: str*, *unit_of_measurement: str*, *friendly_name: str*) → 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.

prepare_data(*var_load: str*, *load_negative: Optional[bool] = False*, *set_zero_min: Optional[bool] = True*, *var_replace_zero: Optional[list] = None*, *var_interp: Optional[list] = 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

5.5 emhass.utils module

emhass.utils.**get_days_list**(*days_to_retrieve: int*) → pandas.core.indexes.datetimes.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_logger**(*fun_name: str*, *config_path: str*, *file: Optional[bool] = True*) → Tuple[logging.Logger, logging.StreamHandler]
Create a simple logger object.

Parameters

- **fun_name** (*str*) – The Python function object name where the logger will be used
- **config_path** (*str*) – The path to the yaml configuration file
- **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**() → str
Get the root absolute path of the working directory.

Returns The root path

Return type str

emhass.utils.**get_root_2pardir**() → str
Get the root absolute path of the working directory using two pardir commands.

Returns The root path

Return type str

emhass.utils.**get_yaml_parse**(*config_path: str*) → Tuple[dict, dict, dict]
Perform parsing of the config.yaml file.

Parameters **config_path** (*str*) – The path to the yaml configuration file

Returns A tuple with the dictionaries containing the parsed data

Return type tuple(dict)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX