
lemlab

Release 1.0

Sebastian D. Lumpp, Michel Zadé, Markus Doepfert

Nov 08, 2022

CONTENTS

1	General information	3
2	Introduction to LEMs	5
3	Getting started with lemlab	9
4	Configuring and executing scenarios	15
5	Real-time co-simulation	23
6	Analyzing results	25
7	Getting involved	31

**Authors**

Sebastian D. Lumpp, Michel Zadé, Markus Doepfert

Organization

Chair of Energy Economy and Application Technology, Technical University of Munich

Version

1.0

Date

28.05.2021

Copyright

The model code is licensed under the [GNU General Public License 3.0](#). This documentation is licensed under a [Creative Commons Attribution 4.0 International license](#).

GENERAL INFORMATION

Authors

Sebastian D. Lumpp, Michel Zadé, Markus Doepfert

Organization

Chair of Energy Economy and Application Technology, Technical University of Munich

Version

1.0

Date

01.06.2021

Copyright

The model code is licensed under the [GNU General Public License 3.0](#). This documentation is licensed under a [Creative Commons Attribution 4.0 International](#) license.

1.1 Description

lemlab is an open-source tool for the agent-based development and testing of local energy market applications offering:

- a fully open-source, agent-based local energy market modelling toolbox
- a modular and extendable design for easy adaptation to your own research questions
- real-time capabilities for the development and testing of hard- and software
- a database-agnostic approach that enables the integration of multiple database technologies
- integrated time-series data for several plant types (household loads, pv, wind, heat pumps, electric vehicles etc...)
- template functionality for load and generation forecasting, trading strategies, cutting-edge market clearing algorithms designed specifically for LEMs [paper under review] and much more...

1.2 Changes

01.06.2021 - first release

1.3 Dependencies

- [Python](#) please see lemlab-env.yaml for virtual environment configuration
- Any solver supported by pyomo. We suggest [gurobi](#) or [cplex](#)
- PostgreSQL

INTRODUCTION TO LEMS

This chapter’s purpose is to familiarize the user with what local energy markets (LEMs) actually are, how they work, who the participants are and basic terminology used in this research area.

2.1 The why and what of LEMs

The fundamental goal of markets is to efficiently match supply and the demand. While the definition of “efficiency” may vary, energy markets share this goal. Traditional energy markets are usually implemented on national levels. The transmission of electricity is treated as an entirely separate task in European liberalized markets. These traditional systems are facing challenges in the face of the rapid expansion of distributed energy resources (often in the form of fluctuating renewables) as well as the anticipated electrification of the mobility and heat sectors in the form of electric vehicles and heat pumps.

The European Union has included new rules in its “Directive on common rules for the internal electricity market” ((EU) 2019/944) “that enable active consumer participation, individually or through citizen energy communities, in all markets, either by generating, consuming, sharing or selling electricity, or by providing flexibility services through demand-response and storage. The directive aims to improve the uptake of energy communities and make it easier for citizens to integrate efficiently in the electricity system, as active participants.”

Specifically, the aim of these energy communities is to enable decentralized producers and consumers (prosumers) to participate “on equal footing with large participants”.

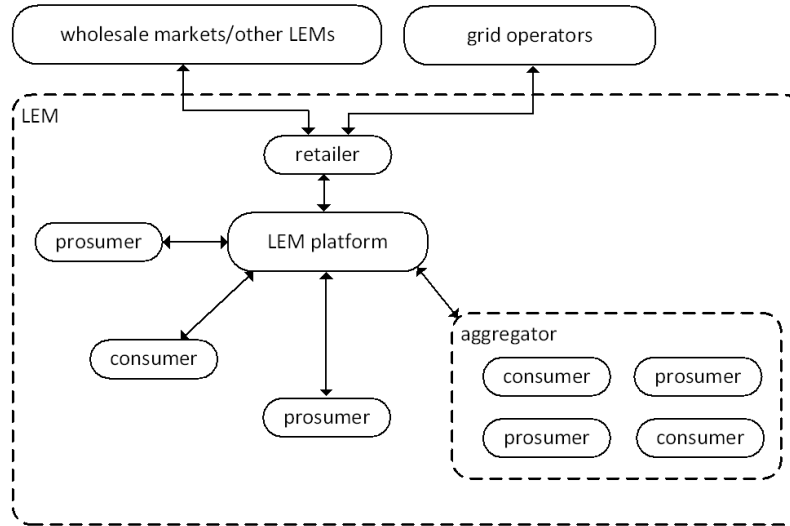
There is no doubt that developing local energy markets and effectively integrating these into existing structures presents a major engineering challenge. In addition to questions of market design, social welfare, and regulation, technical questions regarding infrastructure, organization, agent strategies, market gaming, price formation and many more threaten to overwhelm designers.

In order to tackle the challenges faced by engineers, designers, and regulators, we present the local energy market laboratory, **lemlab**, an open-source tool for the agent-based development and testing of local energy market applications.

2.2 What does a LEM look like?

There is as yet no consensus on what the best structure for a LEM is. For this reason we have attempted to keep our architecture as generic as possible in order to allow the user maximal flexibility in the system designs they wish to implement.

We begin by explaining the basic structure of a LEM according to *lemlab*:



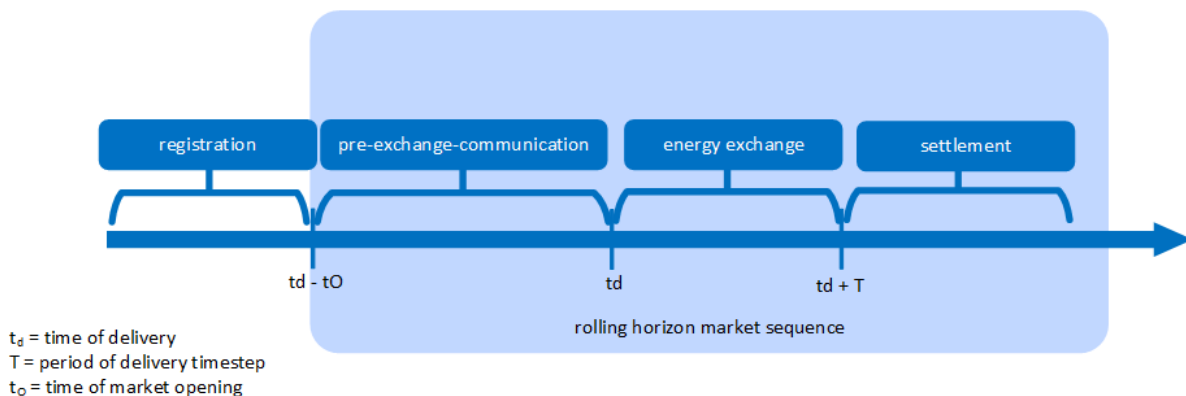
At the heart of the *lemlab* is the **prosumer**. In modern power systems, the end user is no longer just a consumer. The prosumer is the generalized market participant, combining production, consumption, and flexibility with a preference for local and sustainable energy. Of course a prosumer could be a pure consumer, a simple producer, they might be entirely inflexible or they might prefer their electrical energy sources to produce as much carbon dioxide as possible. In any case, we will still refer to them as a prosumer.

Prosumers place bids on a **LEM platform**. This platform combines all market functionality, beginning with user management, bid/offer collection and clearing, and ending with meter reading collection, market settlement and levy collection. **Aggregators** trade on a LEM on behalf of a group of prosumers. **Retailers** take care of coupling the LEM to the wholesale market.

That's pretty much it. Of course we can also model grid operators, transmission grids and wholesale markets as well as multiple LEMs. These are possible in *lemlab* and some are covered in this documentation. All are simply a matter of rearranging the building blocks *lemlab* supplies to construct the desired system.

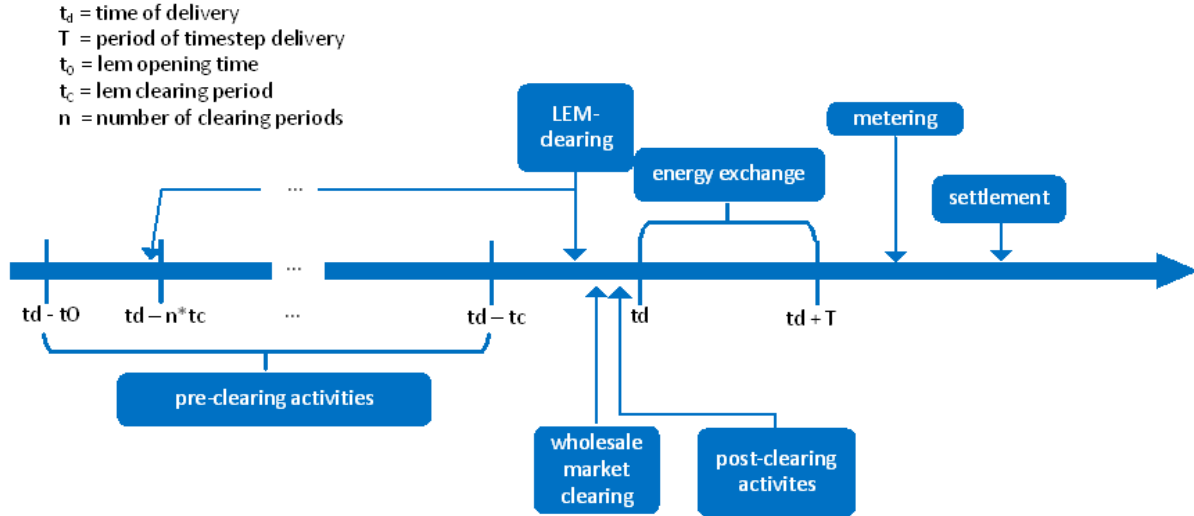
2.3 When does what happen in a LEM?

Similarly to LEM structure, there is no general consensus on the timeline of events in a LEM. We present here a, in our opinion, sensible timeline for a local energy market, based loosely on the general principles underlying European electricity markets.



We divide our market timeline into four fundamental time periods. The first is **registration**, which takes place before trading commences. We then enter the **rolling horizon market sequence**, which repeats for each delivery period. Each

day is divided into an arbitrary number of periods of energy exchange, referred to as **timestep of delivery** ($ts_delivery$) labelled as the period between td and $td + T$. We usually use 15 minute windows ($T=900s$), as the German electricity market is divided into 15 minute periods. During this time, the physical flow of energy takes place. All other activities either take place between market opening, $td - tO$, and the beginning of the $ts_delivery$ in question, td , and are known as **pre-exchange-activities**, or they take place after delivery, in which case they are known as **settlement** activities. Examples of pre-exchange activities are forecasting of demand and energy trading, while settlement activities include meter value logging, market settlement and balancing energy settlement.



The above figure shows a slightly more detailed timeline. We see pre-clearing activities taking place during each of n clearing periods before the timestep of energy delivery. Market clearing can take place during some or all of these periods. Post-clearing activities take place after clearing but before delivery and typically include the checking of clearing results. Metering and settlement activities are performed after the period of energy delivery is complete.

For a more detailed look into the construction of lemlab and the individual agents, please see chapters 3 and 4.

2.4 Basic terminology

agent	someone participating in the LEM
aggregator	an agent trading on behalf of several other agents
LEM	local energy market
pre-clearing activities	activities taking place before market clearing, e.g. forecasting and trading
pre-exchange activities	activities taking place before energy exchange, e.g. forecasting, trading, and checking clearing results
post-clearing activities	activities taking place before energy exchange but after market clearing, e.g. checking clearing results
prosumer	an agent with production and/or consumption, participating in the LEM
registration	the first step in participating in a LEM
retailer	an agent who couples the LEM with the wholesale markets
settlement	activities taking place after market clearing, e.g. market settlement and balancing settlement

GETTING STARTED WITH LEMLAB

3.1 General

In order to further enable the integration of ever-increasing shares of decentralized electricity generation and flexible prosumers, local/regional energy markets (LEM) are being investigated as a potential solution to maximize local matching of generation of consumption in order to relieve pressure on electrical grids by leveraging the inherent temporal flexibility of the smart prosumer.

Investigating the unique challenges involved in constructing stable and efficient LEMs usually involves the setting up of complex and extensive simulation environments, the collection of sufficient data and deep technical knowledge of all components involved. Only then can the desired solution be implemented and tested. Developing software or hardware tools for deployment in the field typically requires the development of hardware/software-in-the-loop (XiL) development environments in order to test prototypes and before field testing can commence.

lemlab was designed specifically with these use cases in mind. lemlab allows the user to simulate a LEM using a full agent-based modelling (ABM) in either simulation (SIM) or real-time (RTS) modes. This allows the rapid testing of algorithms as well as the real-time integration of hardware and software components.

3.2 Installation guide

lemlab is maintained using a combination of PyCharm, PostgreSQL, Gurobi and Anaconda. This installation guide will explain the procedure for this software combination. It is recommended that beginners follow this guide.

Install the following software

- PyCharm (Community or Professional)
- Anaconda Individual Edition
- PostgreSQL*
- Gurobi* or CPLEX. GLPK can be used although this is non-ideal.

Clone repository You can download or clone the repository to a local directory of your choice. You can use version control tools such as GitHub Desktop, Sourcetree, GitKraken or pure Git. The link for pure Git is:

```
git clone https://github.com/tum-ewk/lemlab.git
```

If using PyCharm, clone the repository to `./PyCharmProjects/lemlab/`

Create a virtual python environment

- Open the AnacondaPrompt.
- Type `conda env create -f ./PycharmProjects/lemlab/lemlab-env.yaml`

- Take care to set the correct (absolute) path to your cloned repository.

Activate the environment

- Open PyCharm
- Go to 'File->Open'
- Navigate to PyCharmProjects and open lemlab
- **When the project has opened, go to**
File->Settings->Project->Python Interpreter->Show all->Add->Conda Environment->Existing environment->Select folder->OK`

Install a solver (we recommend Gurobi)

- Go to gurobi.com
- Create an account with your university email
- When the account has been activated, log in and download the newest Gurobi solver.
- Go to Academia->Academic Program and Licenses
- Follow the installation instructions under "Individual Academic Licenses" to activate your copy of Gurobi

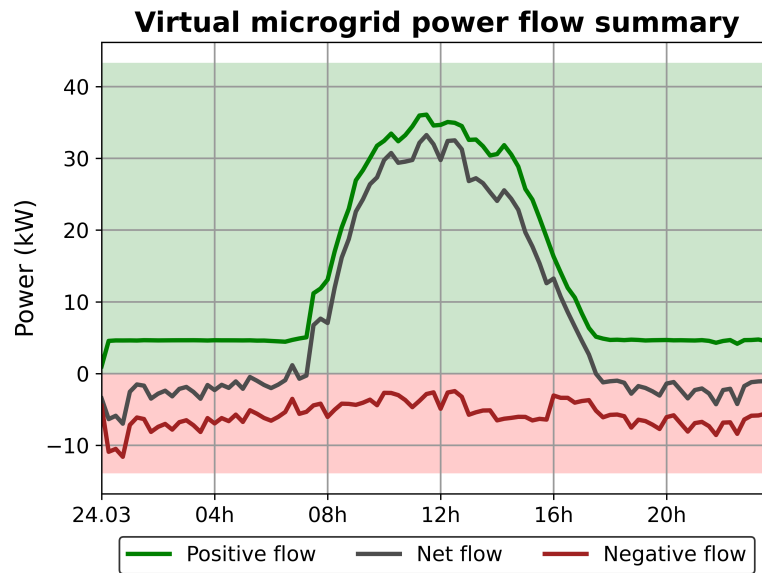
Install and configure PostgreSQL

- Install PostgreSQL from <https://www.postgresql.org/>
- if possible, select port 5432. If this is not possible, you will need to edit the configuration file before executing any simulations
- When your installation has been completed, launch pgAdmin 4
- Select your local server (PostgreSQL 13)
- **Open Login/Group Roles and create two new login roles as follows**
 1. **name: admin_lem**
password: admin privileges: can login
superuser
 2. **name: market_participant**
password: user privileges: can login

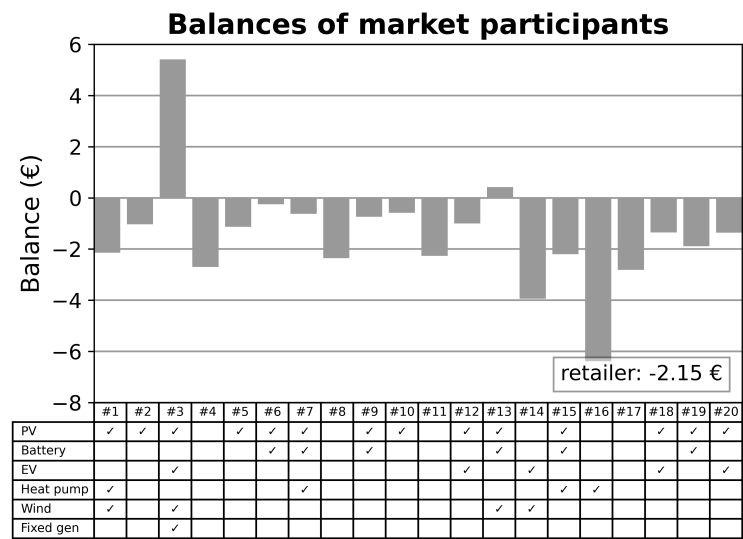
3.3 Test your installation

- navigate to ./PycharmProjects/lemlab/code_examples
- execute sim_1_create_scenario.py, followed by sim_3_run.py
- When the simulation has completed (this may take some time, depending on your system), analyze the results by executing sim_4_plot_results.py
- Look at the output plots under lemlab/simulation_results/test_sim/analyzer/

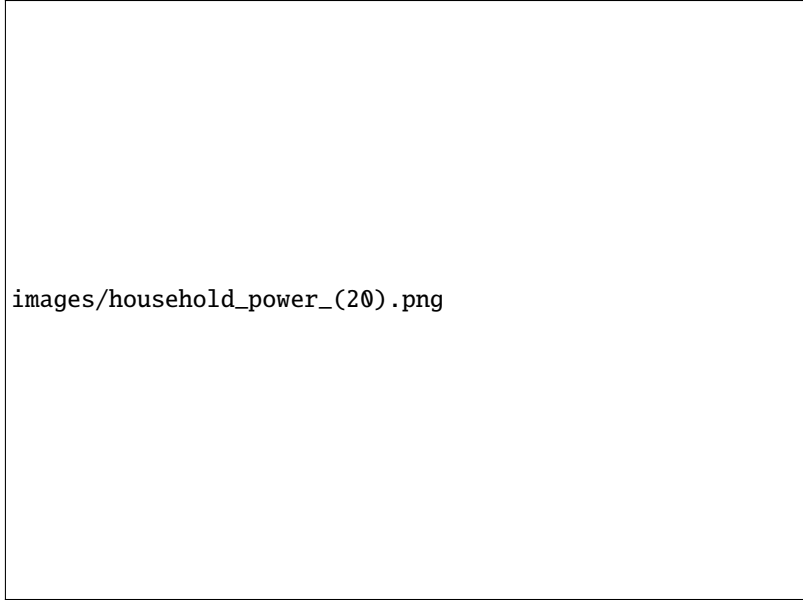
Your installation was successful if you see plots similar to the following:



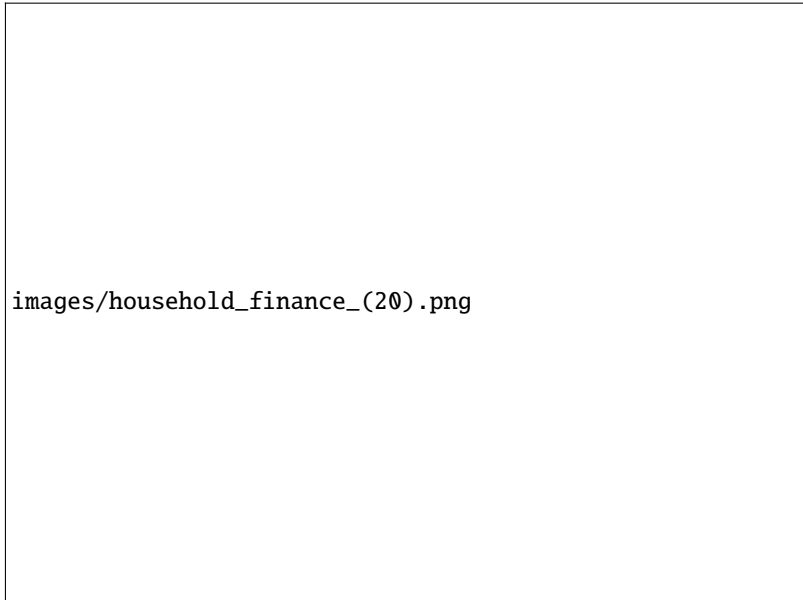
images/mcp_ex_ante_da.png



images/price_type_ex_ante_da.png



images/household_power_(20).png



images/household_finance_(20).png

3.4 Structure

3.5 Workflow

3.6 Input Data

CONFIGURING AND EXECUTING SCENARIOS

The following sections explain how to configure new scenarios using the provided scenario generator. As of version 1.0 it is possible to create scenarios with single-family homes only. Future versions will also include multi-dwelling units, commercial and service buildings as well as industry.

4.1 The config file

The config file specifies all parameters that are needed for the setup up. Two config files can be found in the example folder, which can be used as starter for simulation (SIM) or real-time (RTS) modes. Furthermore, the example folder contains standard files to create, run and analyze new scenarios.

Each config file is divided into the following categories:

- simulation
- lem
- supplier
- prosumer
- producer
- aggregator
- db_connections

4.1.1 simulation

All general parameters that the simulation requires to run are gathered in “simulation”. These differ depending on, if the simulation is real-time or not.

4.1.2 lem

The section contains all information regarding the setup of the local energy market (LEM). The parameters include, for example, how the market is cleared, what transaction types are to be modeled and what types of qualities are traded on the lem. Please note, that the market can be cleared using different methods at once, however, the market’s behavior is based on the very first provided method.

4.1.3 supplier

The supplier represents the entity that ensures the balance of the market by selling additional energy and buying surplus energy. The price at which the supplier sells/buys energy sets the boundaries between which the participants will trade as they would never be willing to pay more than what is guaranteed by the supplier. Likewise they would not sell their energy at a lower price than the supplier's minimum price.

The simplest possible supplier simply creates a market price floor and ceiling by entering unlimited buy and sell offers at fixed prices. Limited coupling capacity can be represented by limiting these buy and sell offers.

4.1.4 prosumer

This is the most extensive section as it contains all information for the individual prosumers. The prosumer class also includes simple consumers. In this case, they are modeled as prosumers that do not own any generation or flexibility capabilities. It is mainly divided into *general settings*, *plant configuration* and *market agent configuration*.

The setup starts with **general settings**, which contain, for example, the number of prosumers that are part of the LEM.

The **household settings** represent the fixed household consumption and their method of forecasting the demand. The demand can be modeled using either a uniform value for all households or a distribution to simulate differently sized households.

The **pv settings** determine the share of households with PV as well as the according sizing method and forecast model.

The **battery settings** can be set either dependently or independently of the PV settings allowing users to own a battery without having to own a PV-system as well. The settings mainly contain the battery characteristics as well as the charging method.

The **electric vehicle settings** encompass the share as well as the basic parameters of an electric vehicle and the corresponding forecasting method to predict the availability and SoC.

The **fixed gen generator settings** offer the possibility to implement a constant generation, which can serve as base-load generation, which could model run-of-river or CHP generation.

The **model predictive control settings** specify how the mpc will forecast the local electricity price and what its trading horizon should be.

The **market agent settings** define the prosumer's trading strategy in the market. Additionally, it can be specified, if the prosumer is willing to pay a premium for electricity of a higher quality, e.g. if he is willing to pay 20 % more for local energy.

The **metering settings** are mainly relevant for real-time simulations as lemlab allows to operate Hardware-in-the-Loop (HiL) LEMs. They allow to simulate that meter readings arrive either late or never.

4.1.5 producer

The producer is a simple prosumer with only one generator. This allows the inclusion of a large producer, for example a community wind or PV farm.

4.1.6 aggregator

The aggregator can be used to simulate an agent that trades on the market for several prosumers. The settings allow to specify which types of prosumers should be aggregated. Furthermore, the forecast method and trading horizon of the aggregator can be set. Similar to the individual prosumers, the aggregator can be configured with a price premium that it is willing to pay for energy of higher quality.

4.1.7 db_connections

The database connections contain the setup of the admin, i.e. the manager of the LEM, as well as the one of the market participants. Depending on the setup, specified by the user, various database platforms can be used.

4.2 Adding input data

The modeling of market participants requires various input files that give each prosumer concrete values for their configuration. Some of the files have a specific naming method that must be abode to and is divided into the following subfolders found under *input_data*:

- balancing_prices
- ev
- ex_post_pricing
- fixedgen
- households
- levy_prices
- pv

4.2.1 balancing_prices

lemmlab allows to model balancing prices either as constant or varying values. When the prices are supposed to vary a time-series file needs to be provided, which contains the positive and negative balancing prices for every time step. The file needs to be csv and there is no specific naming scheme to follow. The name of the file is to be provided in the config file of the simulation in the *lem* section. The form of the data is shown in the following table.

Format: csv (table)

Naming scheme: None

Column names	timestamp	price_balancing_energy_positive	price_balancing_energy_negative
Unit	unix timestamp	€/kWh*	€/kWh*
Data type	integer	float	float
Description	current times-tamp	balancing price for procured positive energy	balancing price for fed-in negative energy

*€ can be substituted with any other currency

4.2.2 ev

The folder contains all driving profiles for the EVs. Every EV in the simulation is randomly assigned a driving profile for the simulation. This occurs in the scenario creation of `scenario_manager.py`. The file is a csv file and has no naming scheme as they are randomly chosen.

Format: csv (table)

Naming scheme: None

Column names	timestamp	availability	distance_driven
Unit	unix timestamp	None	km
Data type	integer	boolean	integer
Description	current timestamp	1: EV available 0: EV not available	driven distance since last departure

4.2.3 ex_post_pricing

The files within the folder describe the clearing for ex-post methods. In the ex-post methods the price for each kWh is based on the supply-demand-ratio within the LEM for each time step. The file contains a dictionary with two keys, which specify the price for the various supply-demand-ratios. When the ratio lies between two explicitly specified ratios the price is interpolated using the two closest values. The name of the file needs to be identical with the name of the method specified in the config file under *type_pricing_ex_post*.

Format: json (dictionary)

Naming scheme: “[name of pricing type].json”

Dict keys	price	supply_demand_ratio
Unit	€/kWh*	None
Data type	float	float
Description	price per kWh	ratio between supply and demand within the LEM

4.2.4 fixedgen

The files contain the power output of the fixed generation. The file is a csv file and has no naming scheme as they are randomly chosen. The power output is specified as p.u. between 0 and 1 to allow differently sized fixed generation.

Format: csv (table)

Naming scheme: None

Column names	timestamp	power
Unit	unix timestamp	p.u.
Data type	integer	float
Description	current timestamp	power output specified per unit between [0,1]

4.2.5 households

The folder contains the household profiles that contain the discrete energy use over the specified time period. Each time stamp has a specific energy consumption in Wh. This demand is seen as inflexible and needs to be served at all times. The file is a csv and has a specific naming conventions, which needs to be followed for the automatic scenario creator to identify the file.

Format: csv

Naming scheme: “hh_[total demand in kWh]_[nth profile with the same demand].csv”

Column names	timestamp	power
Unit	unix timestamp	Wh
Data type	integer	integer
Description	current timestamp	energy consumption

4.2.6 levy_prices

Similar to *balancing_prices* the folder contains files that specify levies for each time step. The file needs to be csv but there is no naming scheme that needs to be adhered since the file to use for the simulation needs to be written in the config file. The file is only used when file-based levies are specified as it is also possible to specify fixed levies in the config file.

Format: csv Naming scheme: None

Column names	timestamp	price_energy_levies_positive	price_energy_levies_negative
Unit	unix timestamp	€/kWh*	€/kWh*
Data type	integer	float	float
Description	current timestamp	levies for energy fed into the grid	levies for energy taken from the grid

4.2.7 pv (incomplete)

The PV files contain the normalized power output of different PV systems. Similar to *fixedgen* the PV profile is randomly chosen when the prosumer is created within `scenario_manager.py`. Therefore, there is no specific naming scheme to follow for now. However, this will change in upcoming releases once the weather data will be implemented. Therefore, this subsection is still incomplete.

Format: csv

Naming scheme: None

Column names	timestamp	power
Unit	unix timestamp	p.u.
Data type	integer	float
Description	current timestamp	power output specified per unit between [0,1]

4.2.8 weather (incomplete)

The weather files are linked to *households* and *pv*. Future releases will also link the weather to the heat supply (e.g. heat pump and CHP). As the files currently do not exist, this section merely serves as information for the reader that further information will be added in future releases.

Format: json

Naming scheme: tba

4.3 Creating a new scenario

This section explains how to create a new scenario using *scenario_manager.py* with the aid of the example file *sim_1_create_scenario.py* in the subfolder *code_examples*.

sim_1_create_scenario.py:

```
import lemlab

if __name__ == "__main__":
    sim_name = "test_sim"

    scenario = lemlab.Scenario()
    scenario.new_scenario(path_specification="sim_0_config.yaml",
                        scenario_name=f"{sim_name}")
```

New scenario are created by first calling an instance of the scenario manager. Afterwards, the function *new_scenario* requires the relative path of the config file that is to be used for the simulation as well as a name of the scenario. A short text appears in the terminal when the creation of the scenario is completed. The scenario will be saved in the subfolder *scenarios* under the given scenario name.

4.4 Editing an existing scenario

In principal there are two methods to edit an existing scenario. The first is manually by editing the config file within a scenario. The second is automatically by opening a scenario config file using code, which allows to serialize scenario editing on the basis of an existing scenario. Both methods will be explained with the aid of the example file *sim_2_edit_scenario.py* in the subfolder *code_examples*, which contains the latter method. Please note that not all settings can be changed as some are fundamental for a scenario. In these cases it is best to create a new scenario.

sim_2_edit_scenario.py:

```
import lemlab
from ruamel.yaml import YAML

if __name__ == "__main__":

    sim_name = "test_sim"

    # create new config file from which to edit scenario
    with open(f"../scenarios/{sim_name}/config.yaml") as config_file:
        config = YAML().load(config_file)
        config["aggregator"]["active"] = True
```

(continues on next page)

(continued from previous page)

```

with open(f"../scenarios/{sim_name}/config_edited.yaml", 'w+') as file:
    YAML().dump(config, file)

# generate new scenario from edited config
scenario = lemlab.Scenario()
scenario.edit_scenario(path_new_config=f"../scenarios/{sim_name}/config_edited.yaml",
                      name_new_scenario="test_sim_with_agg")

```

4.4.1 Manual editing

Manual editing is most suitable when only one new scenario is to be generated based on an existing scenario as it is a fast method. Simply navigate to the scenario that you wish to edit and create a copy of the config file within the folder. Open the copy and edit all settings that you wish to change. Since the new config file already exists, you can skip the middle part of the code shown above. All you need to do is to create an instance of the scenario and use the function `edit_scenario`. The function requires you to specify the path of the edited config file as well as the name of the new scenario. The scenario manager will then create a new scenario based on the existing one.

4.4.2 Automatic editing

Automatic editing is most suitable when several scenarios need to be generated from an existing one as it allows the use of for-loops. The method differs only slightly from the manual one and is shown in the above code. Since an edited config file was not created it needs to be done within the code. To do so, the config file of the existing scenario needs to be imported. Afterwards, the settings can be changed. For example, in the example file the aggregator was set to True to activate it and trade for the specified prosumers on the market. Naturally, more than one parameter can be changed at once. The rest of the code is identical with the manual method.

4.5 Executing a scenario

The execution of a scenario is independent of whether it is real-time or not. However, real-time simulations have a few more features which will be explained separately.

4.5.1 Non-real-time scenarios

As both methods require the same code to be run, they will be explained with the aid of the example file `sim_3_run.py` in the subfolder `code_examples`. The file shows the execution code for a non-real-time simulation. The almost identical code for real-time simulations can be found in `rts_3_start.py`

sim_3_run.py:

```

import lemlab

if __name__ == "__main__":
    sim_name = "test_sim"

    simulation = lemlab.ScenarioExecutor(path_scenario=f"../scenarios/{sim_name}",
                                         path_results=f"../simulation_results/{sim_name}"
                                         ↪)
    simulation.run()

```

To run a simulation you first need to create an instance of the scenario executor. The instance requires the relative path of the scenario as well as the path where to store the simulation results. By using *run* the simulation is started.

4.5.2 Real-time scenarios

Due to their nature, real-time simulations offer a few more features. After a simulation is started, it is possible to pause the simulation to adjust some settings. An example on how the simulation is paused is shown in *rts_4_pause.py*. Afterwards, the simulation can be continued as shown in *rts_5_restart.py*. Please note that the edits should occur in between clearing points as it can otherwise cause issues when trying to restart the simulation. Furthermore, it is possible to obtain visual information using *rts_6_plot_live.py*. This will export the current results of the simulation to be plotted using the scenario analyzer, which is explained in [Analyzing results](#). To stop the simulation use the code provided in *rts_7_stop.py*.

REAL-TIME CO-SIMULATION

documentation coming soon

ANALYZING RESULTS

lemlab comes with a variety of preconfigured plots that allow a first analysis. Additionally it is possible to create your own plots and display them in the lemlab format.

6.1 Using the analysis toolbox

All plotting capabilities are bundled in *scenario_analyzer.py*. The file contains two classes. ScenarioAnalyzer contains all functions to plot various aspects of a scenario to get a deep insight into the results. ScenarioPlotter is the configuration class for lemlab, which gives all plots the same uniform look. It can also be used to create additional plots by the user and still maintain the same look.

The example code to run the analyzer can be found in the code examples in *rts_8_plot_results.py* for real-time and in *sim_4_plot_results.py* for non-real-time simulations. Both files first create an instance of the scenario analyzer by providing the path of the simulation results that are to be observed. Additional arguments specify whether the figures should be shown directly in the IDE and if they are to be saved as png-files in the subfolder *analyzer* of the provided scenario. The command *run_analysis()* calls all analysis functions within the ScenarioAnalyzer class, which will be explained in detail below. All functions can also be called separately by using their respective name.

sim_4_plot_results.py:

```
import lemlab

if __name__ == "__main__":
    sim_name = "test_sim"

    analysis = lemlab.ScenarioAnalyzer(path_results=f"../simulation_results/{sim_name}",
                                       show_figures=True,
                                       save_figures=True)

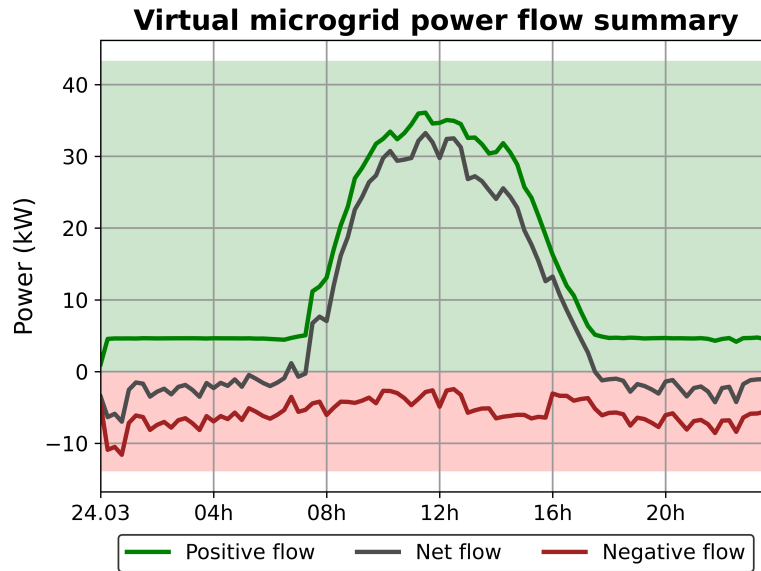
    analysis.run_analysis()
```

6.1.1 Virtual feeder flow

`plot_virtual_feeder_flow()`

The plot shows the virtual power flow within the microgrid for the entire simulation period. The flow is split into the negative flow, which represents all loads that are present in the grid and the positive flow, which sums all local generation within the microgrid. The difference between the two is the net flow and represents the power that is either drawn from the higher level grid during times of higher demand than production or fed into the grid vice-versa.

The function requires no input to create the plot.



6.1.2 Market clearing price

```
plot_mcp(type_market)
```

The plot contains the clearing price for every time step of the simulation. It both contains the individual results in green and the weighted average in red. The more vivid the green circles of the individual results are the more bid-offers matches were cleared at that price.

The plot has one optional argument *type_market*, which can be used to specify which of the simulated market is to be plotted. If no argument is specified, the first type of market is displayed as it is the main market. For further information about the types of market, see lem and the example config files.

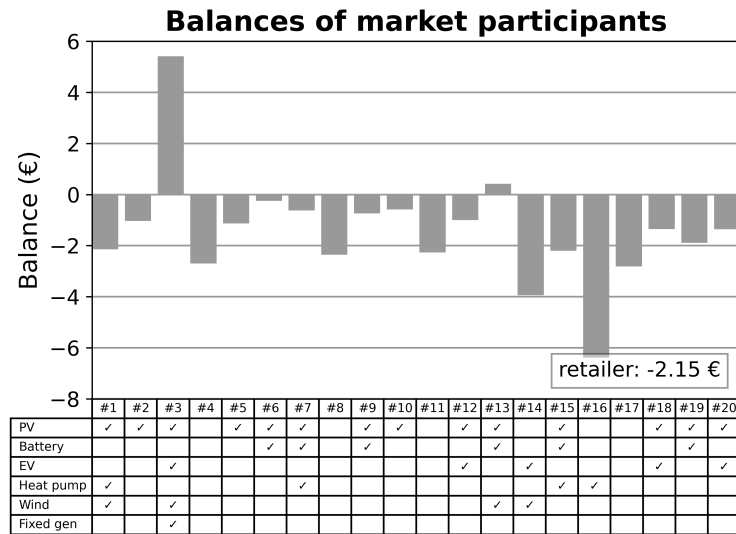
images/mcp_ex_ante_da.png

6.1.3 Market balances

```
plot_balance()
```

The plot shows the market balances of each market participant for the entire simulation period. The supplier's balance is displayed in the bottom corner. The balances of the prosumers are shown as bars alongside the information, which types of devices they own. If the balance is positive, it means that the prosumers earned money during this period while they spent money, if the balance is negative. Positive balances can occur, for example, when a prosumer has as a PV plant and battery.

The function requires no input to create the plot.




6.1.4 Price versus quality

```
plot_price_quality(type_market)
```

The plot displays the weighted market clearing price over the simulation period as well as the share of different qualities in the microgrid. In the below figure these are *local* and *green & local* energy.

The plot has one optional argument *type_market*, which can be used to specify which of the simulated market is to be plotted. If no argument is specified, the first type of market is displayed as it is the main market. For further information about the types of market, see lem and the example config files.



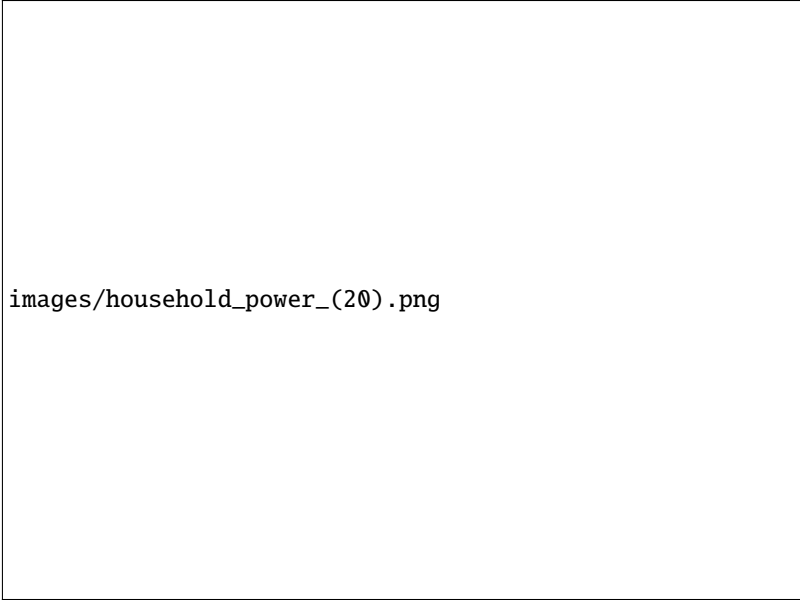
images/price_type_ex_ante_da.png

6.1.5 Household plots

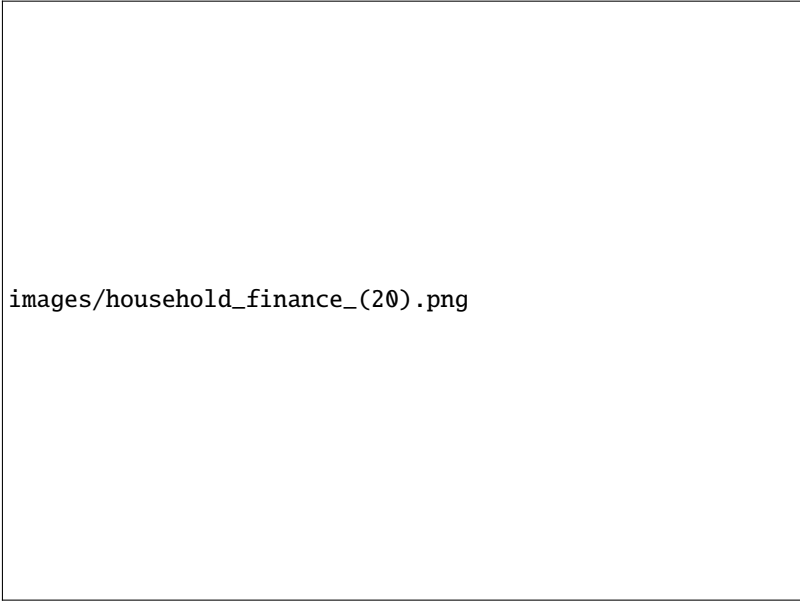
```
plot_household(type_household, id_user)
```

The household plots offer further insight into the individual prosumers. The first plot shows the power profile of the respective prosumer. It shows the individual consumers and generators as well as the power flow through the main meter. The second plot shows the corresponding balance for every time step of the simulation. The balance is split into revenue and fixed and varying costs. The fixed costs contain both the levies as well as balance costs while the varying costs are the costs for purchasing electricity on the market.

The function has two optional arguments *type_household* and *id_user* to allow to plot specific prosumers. *type_household* requires a tuple of 5 boolean values. Each boolean value represents the presence/lack (1/0) of one type of device. The order is the following (PV, Battery, EV, Heat pump, Fixed generation). For example, (1, 0, 1, 0, 0) means that a prosumer with a PV plant and an EV is to be plotted. The advantage for the user is that the function will automatically check if such a prosumer exists. If that is the case, it will be plotted, otherwise the prosumer with the most devices will be plotted. The second optional argument *id_user* allows the user to specify which exact user is to be plotted. The value can either be inserted as integer if numeric values are used as user IDs or otherwise as string.



images/household_power_(20).png



images/household_finance_(20).png

6.1.6 Costs per type prosumer

```
plot_balance_per_type(all_types)
```

To be done once the exact information is decided on.

6.2 Creating your own plots

The scenario analyzer merely serves as first start into the analysis of created scenarios. Depending on the topic to be investigated, additional plots are required to fully understand the market's behavior under the given setup. Naturally, these plots can also be created outside of the lemlab environment. All simulation results are found in the subfolder *scenario_results* under the scenario name. However, it is also possible to create the new plots in the lemlab design.

To create your own plot within the lemlab environment you can include it as function in the class *ScenarioAnalyzer*, however, this is not mandatory. Regardless of whether you want to include it or not the workflow is the same. After extracting the data to be analyzed an instance of the *ScenarioPlotter* needs to be created. This will call lemlab's style and create a figure and axes object. Graphs are to be added to the axes object (e.g. `ax.plot()` or `ax.scatter()`). Once all plots were added *figure_setup* is called to provide the additional figure information such as the title.

figure_setup:

```
figure_setup(title, xlabel, ylabel, ylabel_right, legend_labels, xlims, xticks_style)
```

All parameters of *figure_setup* are optional. The specific instructions on how to use the function can be found in the code. Here only a few parameters will be discussed. *ylabel_right* is only to be used if two y-axes exist. *xlims* specifies the range in which to plot. *xticks_style* specifies the style of the x-ticks. The available styles are "numeric" and "date". If no style is provided there are no x-ticks added to the x-axis. Afterwards, the plot can be displayed using matplotlib:

```
matplotlib.pyplot.show()
```

If you want to save the figure, it is possible to use the built-in function `__save_figure()` of the class *ScenarioAnalyzer* as long as the plot is created within *ScenarioAnalyzer*.

GETTING INVOLVED

@Sebastian

Join us for fun and pizza!