eELib

elenia

Apr 19, 2024

CONTENTS

The **eELib** (**e**lenia **E**nergy **L**ibrary) is the software tool for simulations concerning **future power systems for prosumers**. The library with its functionalities and models can be used for various simulative investigations regarding research or current challenges in the field of a distributed electrical power system.

The goal of the eELib is creating a model library that is suitable for solving energy-related questions around prosumers (consumers that are now also producing energy). This includes, among other things, the ...

- ... creation and consideration of different **energy supply scenarios** (on building, district and grid level, among others with different penetration levels of distributed facilities like PV).
- ... comparison of different **operating strategies for energy management systems**, including e.g. variable tariffs, multi-use concepts, operator models or schedule-based flexibility.
- ... investigation of the **impacts and interactions** of prosumer households (e.g., sector coupling and electrification) **with the power grid** to identify violations of grid limits.
- ... calculating the **economic values of different use cases and strategies** for components and systems.
- ... investigation of innovative marketing strategies of market players in the spot and balancing **power markets**.

CHAPTER

ONE

GETTING STARTED FOR USING THE EELIB

The eELib will soon be made available to install via PyPI - the Python Package Index. Until then, you need to use the installation via its Gitlab Page. The procedure for this is explained in the following subsection.

CHAPTER

TWO

GETTING STARTED FOR CONTRIBUTING TO THE EELIB

When working with the eELib, you should consider this documentation pages and the Gitlab Repository. The documentation gives a short overview of what the eELib is, how it is set up and how it can be used. The short introduction is provided by this start page and the *About eELib* section. Afterwards you should have a look at the *Wiki* part of the documentation. It provides info about how to

- Set up your environment to work with the eELib via an *Installation Guide*,
- Work with *Git* (version management control) and its different operations
- *Set up a scenario* and *run a simulation*
- Add new implementations like *models*

Additionally, the section *API reference* provides an overview of the setup of the eELib and its models and gives explanations to the classes and their methods, which are provided by the docstrings in the source code.

One major thing to know for working with and using the eELib is that we are **using a co-simulation tool** (mosaik) to run simulations, as the eELib itself just provides the implementation of models and no configuration to run a simulation. The sense of this is further explained in the *About eELib* section on *Coupling With mosaik*. Additionally, there is a short *mosaik introduction* and the linking of tutorials using mosaik given in the Wiki.

2.1 For contributing to the eELib, it is useful to follow these steps:

1. *Create a local copy of the Git repository* on your computer. Afterwards, make sure the eELib folder exists on your local hardware and the test scenarios can be executed without any error.
2. Get familiar on how you can *use Git for version control*.
3. Read the *About eELib* section to get a sense of the structure of the eELib and how simulations are carried out.
4. Familiarize with the test scenarios (for one single building and a small 8-bus low-voltage grid):
    - For running a simulation, we need four files: A **scenario script** and the scenario data, which is divided into a **model data** file, the **model connections** and a **grid file**. Additionally, corresponding input data like files for the input of pv generation or a household baseload need to be provided. The setup of these files and their combination is more deeply explained in the Wiki page on *Running a simulation*.
5. You should additionally have a look at a simulator and model for an exemplary model type. The idea of using simulators and models separately is explained in the *Coupling With mosaik and Architecture of a Simulation*``` part of the *About eELib section*. You can find further information on *Simulators* and *Models* and how they are set up in the Wiki pages. This should give you the relevant information to implement your own or advance existing models.

6. If you are interested in setting up (implementing) a new operating strategy for energy management systems (EMS), you should have a look at the explanations on *Implementing a new EMS strategy*. Following the steps mentioned there you should be able to set up your operating strategy and integrate it into your simulation.

2.1.1 About eELib

The models in eELib - like other models too - represent the real processes of existing components in a quasi-stationary / quasi-dynamic approach under the assumption of simplifications. The implementations of the eELib hold some characteristics, that will shortly be explained here.

Folder Structure

This is to give an overview of where different parts of the eELib are stored:

- **docs** : Files for the documentation with AutodocSphinx into the GitLab Pages style. Documentation is stored in `.rst` files within the source subfolder.
- **eelib** : This stores the main part of the elenia Energy Library, as it contains the models and all other functionalities.
  - **core** : Here all of the models are stored. This is divided into the devices (like PV system or electric vehicle), control models (like energy management system), grid models (for power flow calculation), forecasting, and market models (like intraday market).
  - **data** : This contains all models that are "just" retrieving input data (like a simple csv-reader). It also includes the functionalities for simulation data to be collected and assessed.
  - **utils**
    [Contains helper functions and classes, exemplarily for running a simulation or] evaluating and plotting/presenting outputs of simulations.
  - **testing** : Contains all of the testing for the models and functionalities of the library.
- **examples** : This folder gives data and scripts for some test scenarios and should provide an overview of how the eELib can be used. Some exemplary data is stored in the data subfolder which also contains the simulations results.

The highest folder also contains various files that are used for setup of the environment, gitlab communication, a README for information etc.

You can also familiarize with the folder structure of the *`eElib package`* by exploring the *API Reference*.

Plug-and-Play Style

The programming of the models should be implemented in such a way that it can also be used in real-world applications without any adjustments. This ensures that, for example, in the case of changed simulation scenarios or also in laboratory investigations, the same source code can simply be applied in a "**plug-and-play**" way.

Coupling With mosaik and Architecture of a Simulation

In order to make the above mentioned investigation cases possible, one needs to couple the different models of the library in scenarios to be created, e.g. a **PV** system with the energy management system (**EMS**). For this purpose, an *Orchestrator* is to be used, which performs the exchange of data sets between models, calls the calculation of the individual models and controls the general model flow as well as the coupling with a database. mosaik is intended for these tasks. Certain input values are assumed, data sets are calculated internally and output data sets are issued. See the *eELib docomentation part on mosaik* for more information. The eELib should definetily be usable with other orchestrators of a simulation, but the explanations in this documentation are done for mosaik and its simulation orchestration. Additionally, the eELib provides simulators to its models that serve the prupose of APIs for a simulation with mosaik.

The following graphic illustrates the architecture of a simulation and the usage of mosaik in it. For explanations to the terms used for the parties also have a look at the *glossary*. While the eELib provides implementations for the models and their simulators, mosaik is used for a simulation. Additionally a scenario script and data for the scenario (like parameterization of the model entities and linking of entities) have to be provided. Examples for these last two are provided within the eELib in an *examples* part.

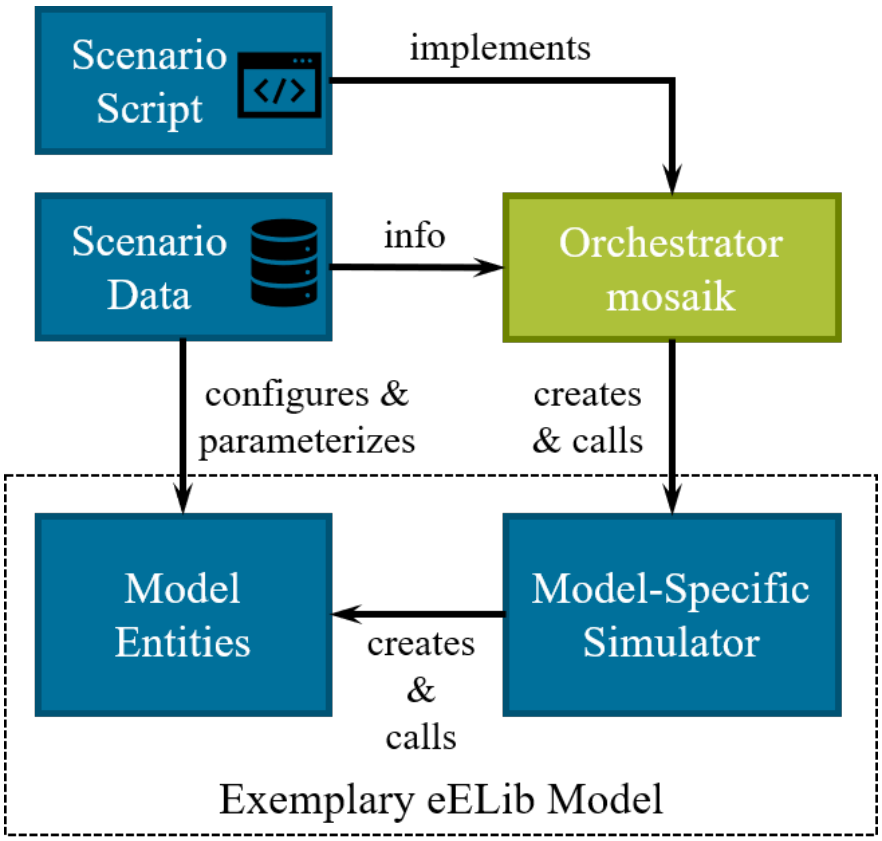


Fig. 1: The architecture of a simulation using mosaik with the eELib.

For a setup of a simulation, have a look at the following illustration. It depicts the physical and data connections between the different parts of the eELib.

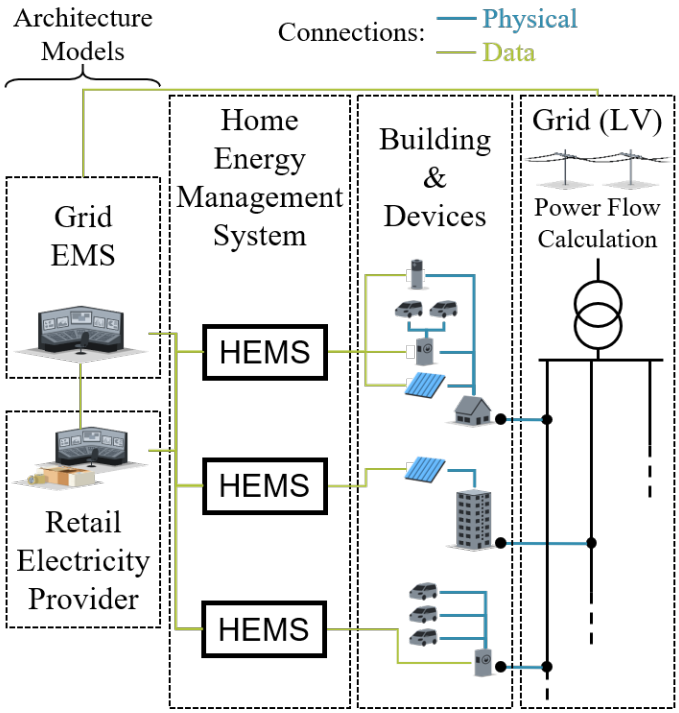


Fig. 2: The setup of a (grid) simulation with the eELib.

Event-Based Simulation

The computations within the eELib - using mosaik - are executed in an **event-based** manner. This implies the process of a simulation to depend on the triggering of events. Independent of a *step size* - the length of a simulation step in seconds - the process within one simulation step is executed by event triggering, as depicted in the following figure. To add to the explanation of mosaik, the execution of events depends on the triggering of such events. E.g. the simulation of a BSS is triggered by an EMS sending a power set value to the BSS. For this, mosaik knows about …

- … when each model has to be called for calculation.
- … which outputs for the models are send to which inputs for other models.

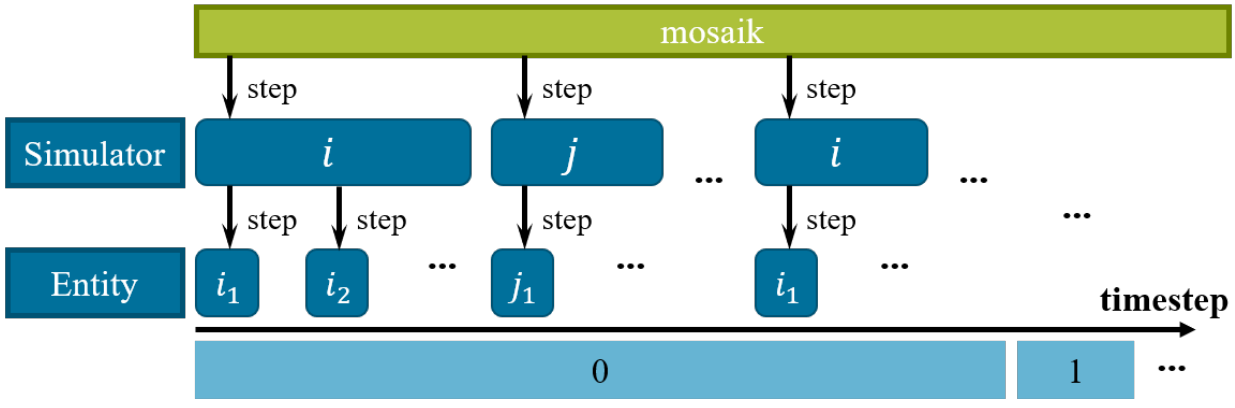


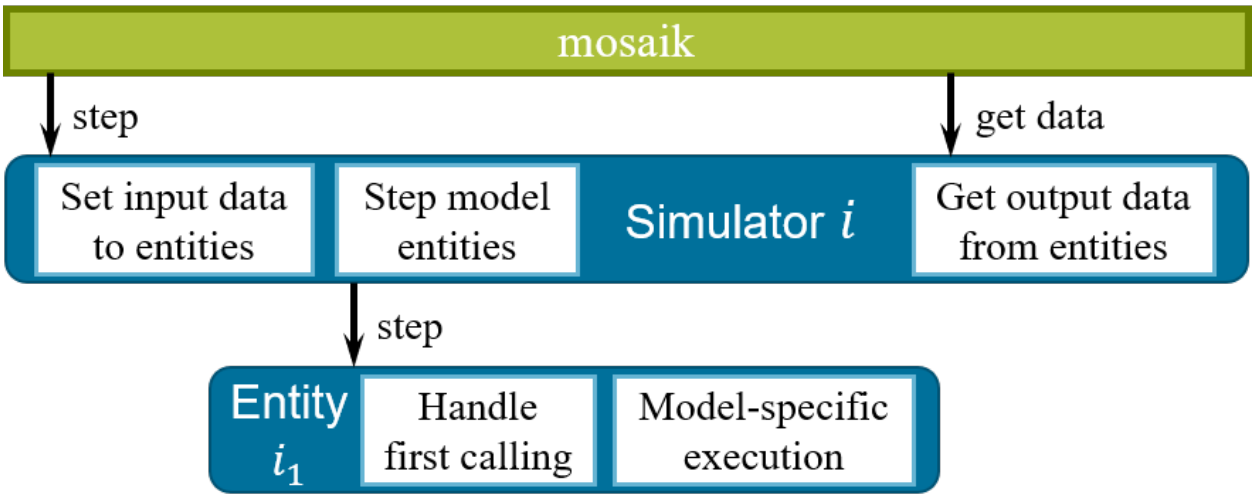
Fig. 3: The simulation process of event-based simulation.

1. mosaik calls the calculation of the entities via the simulator

2. models calculate output with the already given input
3. simulator returns the time, when the models have to be calculated next
4. mosaik sends the output to a connected entity (e.g. the power generation of a PV system to the HEMS)
5. mosaik then calls the calculation of the next entity for which all inputs have been collected (this is done by means of the word "triggering", so the finalized calculation of one entity triggers the calculation of another entity...)

This is done within a single timestep as long as data is send between the entities such that the renewed calculation of a model entity is triggered. mosaik calls the execution of the models in the way they are connected to each other and send values each way. If everything is finished for this timestep, mosaik advances to the next timestep and the simulation process carries on.

The implementation of the process within one simulation step can be better seen in the next figure. It is shown, that when a model is called, the inputs for the models are provided by mosaik and set by the simulator. Afterwards the models are stepped and ultimately the simulators are called to extract the (by mosaik requested) result outputs from the models.



2.1.2 Wiki

The Wiki provides a walkthrough for different tasks, as well as self-aid in case of common questions.

For a detailed start, we recommend fully reading this Wiki. It covers...

- **Basics** - Setup of a *programming environment*,
- **Usage** - Guides to *configure a scenario* and *run a simulation*,
- **Contribution** - Introduction to *git workflow* and adding *models*/*simulators*/*strategies*
- ... and much more like explanations to *Forecasts and Schedules* and a *glossary*

Installation and Setup

Installation of Python

1. Download Python version 3.10.X from Python Downloads
2. When the download is finished, double-click the installer.
3. Select *Install for all users* and click *Next*.
4. The default installation path is okay. Click *Next*.
5. In the *Customize Python* page, click on the *Python* node and select *Entire feature will be installed on local hard drive*. Make sure that *Add python.exe to Path* is enabled. Click *Next*.
    - If not, the Python Path has to be added to system variables by hand afterwards
6. When Windows asks you to allow the installation, do so. Wait for it to happen. Click *Finish*

Note: This will also install the Python package manager pip. For checking and if not, see Pip getting started (can also be used generally for working with pip).

Installation of Git

1. In order to work with version control management, you need to install Git
2. Visit the Git Getting Started Page and ...
3. Install Git by downloading the Installer and clicking through the setup
4. For Git steps/processes afterwards, have a look at the *Git Workflow* page

Installation and Setup of Python IDE (VSC)

Easier than using command window or PowerShell is the use of an IDE (integrated development environment) for Python, especially when working with the code.

1. Decide for an IDE. There are several good options: PyCharm, Visual Studio Code, Jupyter Notebook, IDLE, Spyder, Pydev
    - We recommend Visual Studio Code (VSC) for eELib, so this tutorial will be based on VSC
2. If needed, download VSC from the VSC Homepage and install it
3. Configuration of User Settings
    1. Install some Extensions (`Ctrl` + `Shift` + `X`)
        - Python (for working with python programming language)
        - autoDocstring (for simple creation of docstrings for classes and methods)
        - H5Web (for a quick and easy view of the HDF5 simulation output)
        - Other possibly helpful extensions: Black Formatter, Git Graph, GitLens, Rainbow CSV
    2. Settings -> Editor: Default Formatter -> set to "Black Formatter"
    3. Settings -> Text Editor -> Formatting -> Format on Save should be true
    4. Settings -> Features -> Notebook -> Format on Save should be true

5. For max line lenght of 100 set: Settings -> Text Editor -> Rulers -> click "Edit in settings.json" and then set "'editor.rulers': [100]" (you can instantly see the vertical line shift to the right when saving the file)

Cloning eELib Repository to your Local Workspace

1. Clone the Git Repository with VSC: When all folders are closed, select *Clone (Git) Repository*
   - https address: "https://gitlab.com/elenia1/elenia-energy-library"
2. You can also use GitBash (or other Git programs) for cloning (personal choice, up to you)

```
$ git clone https://gitlab.com/elenia1/elenia-energy-library
```

3. Afterwards follow the steps you will be guided through, as you probably need to give your username and password for Gitlab

---

Note: The path to the project folder will now be noted as *<Project Folder>*.

Setup Working in VSC with eELib

1. Open VSC and navigate to *<Project Folder>*
2. Open new Terminal: PowerShell is recommended (GitBash or Command Window are possible too, but not as mighty)
3. Create a virtual environment in the directory of your repository:
   1. Run `python -m venv <_VENV-PATH_>`
      - For virtual environment path <_VENV-PATH_>, we typically use `.venv` (which for this tutorial we will now exemplarily do so)
      - Accept VSC for acknowledging new environment, if it is detected
   2. Run the activation script for Powershell: `.venv\Scripts\Activate.ps1`
      - In case Scripts can't be executed, you have to adjust the Execution Policy by running `Set-ExecutionPolicy Bypass -Scope CurrentUser -Force` and try again
      - (Command window has different activation file 'activate.bat')
      - Check: If successful, the prompt should now start with `(.venv)`
      - Check whether the correct python interpreter is selected: `python --version` (Output: Python 3.10.X)
4. Install requirements into the virtual environment
   - If VSC explorer isn't already in the repository folder, you have to navigate there
   - Run `pip install -r requirements.txt`
5. If a new release of pip is available, you can update it via `python.exe -m pip install --upgrade pip`
6. Check: If you open a Python file, the selected virtual environment is listed in the lower right corner in the blue row *('.venv')*
7. Install Configurations for Processes, that are executed before each commit
   - Run `pre-commit install`

Test successfull installation

1. You can test the functionality and correct installation by running a testcase
   - Open the file `test_scenario_building.py` in the examples folder and click on the *Run* sign in the upper right corner (Quick start typicall possible with `F5`)
   - (Or run `python test_scenario_building.py` in the terminal)
2. If you are not able to run the `test_scenario` and get the error `no module named 'eelib'` ...
   - If your are in your <*Project Folder*>: Execute `pip install -e.`
   - Otherwise you have to insert the path: Execute `pip install -e <Project Folder>`

Git Workflow

This wiki page will give detailed instruction on how to work with the commands in Git, especially via Visual Studio Code (VSC)! It will not provide a full overview of how the git process works but informs about the necessary commands.

Create Personal Access Token

1. In Gitlab, via 'Edit profile', you go to 'Access Tokens' and create a Token.
   - The name is irrelevant and the expiration date can be set to a year.
   - You can select all scopes.
2. The Token has to be saved somewhere, because you have to give it in order to establish a connection from your local repository to the online repository.

Cloning

The next thing that has to be done is cloning the online Git repository onto your local computer.

1. You can do this via Visual Studio Code by typing in `gitcl` in the command line and following the processes.
   - Use either the URL or SSH-key from the repository.
   - When asked for username and password you have to give your **GitLab username** and the **Personal Access Token**.
   - This process can also be done via GitBash, when going to the target folder and typing in `git clone <REPO-URL>`.
2. It is recommended to end this process by saving the Personal Access Token in this Git project by running `git remote set-url origin https://oauth2:[PersonalAccessToken]@<GIT-REPO>` via GitBash, then your Git program will not ask for the Token every time.
3. You should also set your username and mail adress for when using Git by running `git config --global user.name "Your Name"` resp. `git config --global user.email "youremail@yourdomain.de"`.

Visual Studio Code

- Visual Studio Code allows easy version management via Git.
- You should use the window "Source Control" (left side) to always see the current changes you did on the code.
- When using the Extension Git Graph , you can also have an overview of all the changes that have been done in the repository (possibly by others too!).

Change the Branch

- Before any adaption to the code, you should always check your currently selected branch.
- The default is always the `main` branch, in which you should **not change anything!**
- Changing the selected branch can be done by the Command "Source Control -> Branches -> Switch to Another Branch" and then selecting that specific branch. Or you can use the command `check-out to` in order to switch to another branch.

Hint: You can only change the selected branch, when you currently have no changes in the files.

Saving Changes (Commit)

The next thing is saving changes, that were coded locally - this is done by 'committing' the code, which is still a local process but kind of saves the stage at the moment.

1. You have to "stage" the relevant code files by pressing the "plus" (+) sign.
2. Then type in a fitting commit message (What have you done? Short!).
3. Last thing is to hit the commit button ("check" sign).

Getting Changes from Online Repository (Pull)

When others made relevant code changes, which you might need, you can get their changes from the online repository by "pulling" them.

1. For that you must't have everything committed - so do that first.
2. To pull the current state of the branch from the online repository, simply click on the 3 points and select "Pull".
3. In case any merge conflicts occur, see step *Merge Changes Into Your Local Branch*

Sending Changes to Online Repository (Push)

- When having implemented relevant code changes, which other programmers/users might need, AFTER VALIDATING THEM you can (and should regularly) push your saved commits.
- This should only be done after pulling the online repository first.
- Also, all current changes have to be staged.
- To push, simply click on the 3 points and select "Push".

Merge Changes Into Your Local Branch

- Merging Changes is generally needed, when two programmers changed parts of the same code and Git does not know, which "solutions" it should select.
- This process should be done **carefully**, as nobody wants to discard changes, that have been done by others.
- Merging can be needed in two cases: when you merge another branch into your own local branch, or when you pull the online repository status into your local state.
- In the process, click on the files with "*Merge Conflicts*", go through the problems and try to find a solution that approriately takes both solutions into account.
- If you have concerns, ask for help!
- Merges with merge conflicts always have to be committed after the conflicts have been solved, and the file is saved and staged.

Create a New Branch for each Topic

The *work flow* is supposed to include the creation of a new branch for each problem. So in case you want to create a new model, you create a branch named `model/[model_name]` and work on this problem only in this branch. Or if you want to fix something existing, create a branch named `bugfix/[problem_name]` etc.

This is a way to structurize the current work and additions to the joint use of the library in the `main` branch.

Compare the State of two Branches

When working on a topic in a specific branch, it can be beneficial before a merge request to review the changes that one made. So after merging the current state of the `main` branch into your own branch, you can compare these two branches (without creating a merge request first) by going to your branch in the Gitlab repository and clicking `Compare`. This can also help to check whether the merge of the `main` branch into your branch worked.

Merge Your Changes into the main Branch

When you

1. completed a task in a specific branch (don't use the `main` branch for that!)
2. and tested your stuff,

you can make this accessible for others via the `main` branch.

For that you should …

1. … merge the current state of the main branch into your (feature) branch.
2. … push your changes into the remote branch.
3. … go to the GitLab Repository and create a merge request for your branch into `main`. There you should also assign a *developer* to shortly check your updates and may even assign a *reviewer*.

The merge will then be completed and afterwards your changes are also part of the `main` branch.

Mosaik

mosaik, according to the mosaik documentation is a "flexible Smart Grid co-simulation framework". It can combine various existing models to run large-scale scenarios - and this is what we intend it for in our use. mosaik can combine our prosumer models like electric vehicles, energy management systemens, grid calculations, and all the others. While we in our eELib focus on the implementation of power system models and energy management strategies, mosaik is used for simulatory behaviour, as explained in the *overview* .

For introduction to mosaik you can or should have a look at its tutorial. Recommended articles of this tutorial for the use within the eELib are:

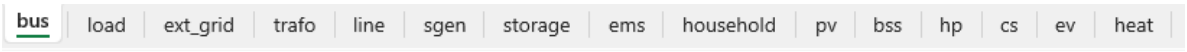
1. Integrating a simulation model into the mosaik ecosystem
2. Creating and running simple simulation scenarios
3. (optionally helpful) Adding a control mechanism to a scenario
4. (optionally helpful) Integrating a control mechanism

Configure a Scenario With an Excel File

> **Caution:** Still **WIP** and might not work.

Excel-file setup

1. Open Excel-file `(eelib\utils\simulation_setup\sim_config_data.xlsx)`. This one is part of the *`utils package`*.
2. The sheets **bus, load, ext_grid, trafo, line, sgen, storage** are used to configure the grid.

3. To **add a new entity** to the grid you create a new row and include the index name and the init_vals of the element. More information about the characteristics of the grid elements: https://pandapower.readthedocs.io/en/v2.13.1/elements.html

   (Make sure you have at least **one transformer**, **one external grid** and **buses are connected through lines**)

	name	vn_kv	type	in_service	x	y
0	bus_0	20	b	True	1	0
1	bus_1	20	b	True	2	0
2	bus_2	0.40	b	True	3	0
3	bus_3	0.40	b	True	4	0
4	bus_4	0.40	b	True	5	-1
5	bus_5	0.40	b	True	6	-1
6	bus_6	0.40	b	True	7	-1
7	bus_7	0.40	b	True	6	0
8	bus_8	0.40	b	True	7	0
9	bus_9	0.40	b	True	8	0
10	bus_10	0.40	b	True	9	0
11	bus_11	0.40	b	True	6	1
12	bus_12	0.40	b	True	7	1

4. After configurating the grid you can add the model entities.
    1. Open the sheet ems and define the number of ems and connect them to a bus
    2. Households, charging stations or pv can be connected to a bus too (but only one per bus)
    3. Adding a new entitiy of a model type is similar to add a new grid element (important is to set the connection to an ems or a bus and differentiate between csv_reader model or exact mode type
    4. Now set the init_vals for every entity of the model type

	bus	ems	name	type	output_type	p_rated	strategy	p_rated_csv	p_rated_profile	cos_phi	filename	header_rows	date_format	start_time
0		2	charging_station_0	exact	AC	11000	max_P							
1		4	charging_station_csv_0	csv				11000	11000	0.95	examples	2	YYYY-MM-DD HH:mm:	01.01.2020 00:00
2	12		charging_station_csv_1	csv				11000	11000	0.95	examples	2	YYYY-MM-DD HH:mm:	01.01.2020 00:00

Create .json files for scenario

1. Open jupyter notebook `eelib\\utils\\simulation_setup\\script_sim_setup.ipynb`
2. Run the corresponding cells to import the packages and set the input and output paths:

    To create a grid

    1. Read the sheets from the excel file
    2. Run the “create grid data file” cell
    3. An image of the grid is shown underneath and the `.json`-file is safed at `C:/Users/Puplic/Documents`

    To create model data file

    1. Read the model type sheets frome the excel file
    2. Run “create model data file” cell
    3. `.json`-file is safed at `C:/Users/Public/Documents`

    To create connection file

    1. Run the cell “Create model_grid_config file”

2. File is safed at `C:/Users/Public/Documents`

3. Now run a scenario file from examples folder, e.g. `test_scenario_grid` or `test_scenario_building`

Add a completely new model type

1. Create a new sheet for the model type in the excel file
2. Create entities and set the init_vals and the connection to a bus or ems for the new model type
3. Open the jupyter notebook `eelib\\utils\\simulation_setup\\script_sim_setup.ipynb`
    1. Read the new excel sheet

```
df_new_model_type = pd.read_excel(FILE_SCENARIO_INPUT, sheet_name="new_model_
↪type", index_col=0)
```

    2. Add new model entities to model_data file (if necessery differentiate between exact- and csv-model)

```
#create an empty list for the new model type
new_model_type = []
for i in df_new_model_type.index:
# Add all init_vals for model type
   new_model_type_data = {
      "datafile": df_new_model_type.at[i, "datafile"],
      "start_time": df_new_model_type.at[i, "start_time"],
      "date_format": df_new_model_type.at[i, "date_format"],
      "header_rows": int(df_new_model_type.at[i, "header_rows"]),
   }
   new_model_type.append(new_model_type_data)
# add new model type list to dictionary
model_data= { ...
            new_model_type : new_model_type
}
```

    3. Add new model type to `model_grid_config` file

```
# create an empty list for the new model type
new_model_types = []
# loop over all entities
for j in df_new_model_type.index:
      # if the entity is connected with bus or ems the name of the entity is␣
↪added to the list new_model_types
      if df_loads.at[i, "bus"] == df_new_model_type.at[j, "bus"] or ems_idx ==␣
↪df_new_model_type.at[j, "ems"]:
            new_model_type = df_new_model_type.at[j, "name"]
            new_model_types.append(new_model_type)
# add new model type list to dictionary
elements = {...
      new_model_type: new_model_types
}
```

4. Now run the cells to create new `.json` files

Set Up and Run a Simulation

What files are needed for a simulation?

Scenario script

Start the simulators, build the models, create the connections and start mosaik. Exemplary scripts for building, grid etc. can be found in the `examples` folder. The setup of these scripts is explained below in the *Scenario Script Configuration* section.

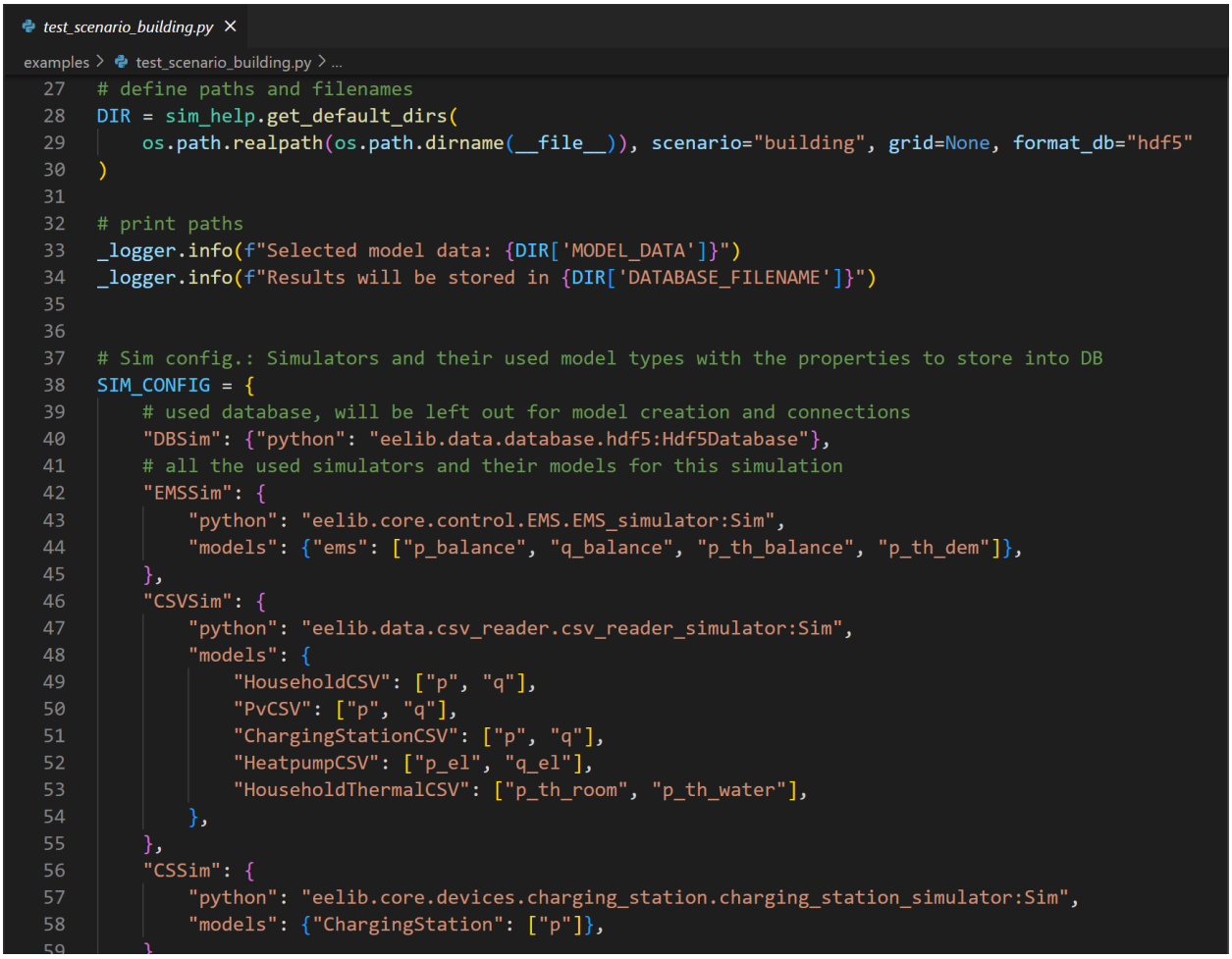


Fig. 4: examples/test_scenario_building.py (01/24)

Model data file

Information on number of models and their parameterization.

This is stored as a dict with fields for each model type, named by the class of the model. E.g. the field *"HouseholdCSV"* contains a list of all household baseloads that read from a csv file. In the example below there is just one baseload, and if more of these should be integrated, we could add a second dict with initialization parameters for this model inside of the given list.

One exception is the Energy Management System (EMS), as they are all sorted within the "ems" section. Their differentiation is coming from the field *"strategy"*, which is referring to the chosen operating strategy of the EMS.

Listing 1: examples/data/model_data_scenario/model_data_building.json (01/24)

```
{
   "ems": [
      {
            "strategy": "HEMS_default",
            "cs_strategy": "balanced"
      }
   ],
   "HouseholdCSV": [
      {
            "p_rated": 4500,
            "p_rated_profile": 4000,
            "cos_phi": 1.0,
            "datafile": "examples/data/load/4_persons_profile/load_34.csv",
            "date_format": "YYYY-MM-DD HH:mm:ss",
            "header_rows": 2,
            "start_time": "2014-01-01 00:00:00"
      }
   ],
...
```

Model connections

Information on the linking of model entities in the simulation. This includes the connections between grid buses, ems models, and the prosumer devices. It is stated, which devices are located at which grid connection point (by the naming of the corresponding load bus) and whether there is an EMS used. If no EMS is given at a loadbus (see *"0-load_1_2"* below), the devices are directly connection to the grid, but this should only be possible for some of the devices, as e.g. a single heat pump makes no sense.

Listing 2: examples/data/grid/grid_model_config.json (01/24)

```
{
   "0-load_1_1": {
      "ems": "HEMS_default_0",
      "load": [
            "HouseholdCSV_0"
      ],
      "household_thermal": [],
      "pv": [],
```

(continues on next page)

(continued from previous page)

```
      "bss": [],
      "hp": [],
      "cs": [
            "ChargingStation_0"
      ],
      "ev": [
            "EV_0"
      ]
   },
   "0-load_1_2": {
      "ems": "",
      "load": [
            "HouseholdCSV_1"
      ],
...
```

Grid file

In `.json` format (possibly created via pandapower). Provides detail on the given buses and their connections via grid lines (plus their parameterization).

Listing 3: examples/data/grid/example_grid_kerber.json (01/24)

```
{
  "_module": "pandapower.auxiliary",
  "_class": "pandapowerNet",
  "_object": {
    "bus": {
      "_module": "pandas.core.frame",
      "_class": "DataFrame",
      "_object": "{\"columns\":[\"name\",\"vn_kv\",\"type\",\"zone\",\"in_service\"],\
↪"index\":[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17],\"data\":[[\"Trafostation_OS\",
↪10.0,\"b\",null,true],[\"main_busbar\",0.4,\"b\",null,true],[\"MUF_1_1\",0.4,\"n\",
↪null,true],[\"loadbus_1_1\",0.4,\"b\",null,true],[\"KV_1_2\",0.4,\"b\",null,true],[\
↪"loadbus_1_2\",0.4,\"b\",null,true],[\"MUF_1_3\",0.4,\"n\",null,true],[\"loadbus_1_3\",
↪0.4,\"b\",null,true],[\"KV_1_4\",0.4,\"b\",null,true],[\"loadbus_1_4\",0.4,\"b\",null,
↪true],[\"MUF_1_5\",0.4,\"n\",null,true],[\"loadbus_1_5\",0.4,\"b\",null,true],[\"KV_1_
↪6\",0.4,\"b\",null,true],[\"loadbus_1_6\",0.4,\"b\",null,true],[\"MUF_2_1\",0.4,\"n\",
↪null,true],[\"loadbus_2_1\",0.4,\"b\",null,true],[\"KV_2_2\",0.4,\"b\",null,true],[\
↪"loadbus_2_2\",0.4,\"b\",null,true]]}",
      "orient": "split",
      "dtype": {
        "name": "object",
        "vn_kv": "float64",
        "type": "object",
        "zone": "object",
        "in_service": "bool"
      }
    },
...
```

Tip: All files can be created (more easily) with a *Scenario Configurator* (`.ipynb`) via an excel file. Or use existing files and adapt the parameterization.

Configuration of a Scenario Script

Note: All of these code-blocks derive from `examples/test_scenario_building.py` as of (01/24) if not stated otherwise.

Setup

Listing 4: `import` of used packages (examples, like mosaik or logging during the simulation)

```
import os
import mosaik
import eelib.utils.simulation_helper as sim_help
from eelib.model_connections.connections import get_default_connections
import logging
```

Listing 5: Setting of paths for simulation data and used model simulators

```
# define paths and filenames
DIR = sim_help.get_default_dirs(
    os.path.realpath(os.path.dirname(__file__)), scenario="building", grid=None, format_
↪db="hdf5"
)
```

Listing 6: Define simulators and models for the simulation

```
# Sim config.: Simulators and their used model types with the properties to store into DB
SIM_CONFIG = {
   # used database, will be left out for model creation and connections
   "DBSim": {"python": "eelib.data.database.hdf5:Hdf5Database"},
   # all the used simulators and their models for this simulation
   "EMSSim": {
      "python": "eelib.core.control.EMS.EMS_simulator:Sim",
      "models": {"ems": ["p_balance", "q_balance", "p_th_balance", "p_th_dem"]},
   },
   "CSVSim": {
      "python": "eelib.data.csv_reader.csv_reader_simulator:Sim",
      "models": {
            "HouseholdCSV": ["p", "q"],
            "PvCSV": ["p", "q"],
            "ChargingStationCSV": ["p", "q"],
            "HeatpumpCSV": ["p_el", "q_el"],
            "HouseholdThermalCSV": ["p_th_room", "p_th_water"],
      },
```

(continues on next page)

(continued from previous page)

```
    },
    ...
```

This is done in a format that fits both mosaik for orchestrating the simulation and also the handling of data and simulation setup. So in the *"SIM_CONFIG"* dict we first need to give all used simulators as the keys (e.g. *"CSVSim"* for all used csv readers). With the sub-key *"python"* we then provide the path to this simulator in our library (e.g. *"eelib.data.csv_reader.csv_reader_simulator:Sim"*). Under *"models"* we list all of the used model types within the simulation for this simulator (e.g. *"HouseholdCSV", "PvCSV"*) and the attributes of them that should be stored within the database (e.g. *["p", "q"]* for active and reactive power of the PV system).

Listing 7: Configure start and end time of the simulation, step length, and whether to use forecasts. Model data and connections are read and `SIM_CONFIG` is handed to mosaik.

```
# Configuration of scenario: time and granularity
START = "2020-01-01 00:00:00"
END = "2020-01-04 00:00:00"
USE_FORECAST = False
STEP_SIZE_IN_SECONDS = 900  # 1=sec-steps, 3600=hour-steps, 900=15min-steps, 600=10min-steps
N_SECONDS = int(
    (
        arrow.get(END, "YYYY-MM-DD HH:mm:ss") - arrow.get(START, "YYYY-MM-DD HH:mm:ss")
    ).total_seconds()
)
N_STEPS = int(N_SECONDS / STEP_SIZE_IN_SECONDS)
scenario_config = {
    "start": START,  # time of beginning for simulation
    "end": END,  # time of ending
    "step_size": STEP_SIZE_IN_SECONDS,
    "n_steps": N_STEPS,
    "use_forcast": USE_FORECAST,
    "bool_plot": False,
}

# Read Scenario file with data for model entities
with open(DIR["MODEL_DATA"]) as f:
    model_data = json.load(f)

# Read configuration file with data for connections between prosumer devices
model_connect_config = get_default_connections()

# Create world
world = mosaik.World(SIM_CONFIG, debug=True)
```

With this *mosaik.World* we start the orchestrator mosaik and afterwards set up the simulation using mosaik. For this, we mostly use mosaik functionalities within the helper functions of the *eelib.utils.simulation_helper* folder. Regarding the order, we mostly first handle the exceptions of the models, like the output database, and afterwards use the helper functions for all of the other models. Fore a deeper understanding of what is done with these steps, you should have a look at the implementation of these helper functions and possibly even the setup of mosaik for creating the simulation.

Start Simulators for all used Models

```
# start all simulators/model factories with mosaik for data given in SIM_CONFIG
dict_simulators = sim_help.start_simulators(
    sim_config=SIM_CONFIG, world=world, scenario_config=scenario_config
)
```

Initiate Model Entities using the created Simulators for each Device Type

```
# create all models based on given SIM_CONFIG
dict_entities = sim_help.create_entities(
    sim_config=SIM_CONFIG, model_data=model_data, dict_simulators=dict_simulators
)
```

Connect Entities

The connections for each entity are listed in the `model_connect_config` data. Now tell mosaik about the and let mosaik create graphs for how the calculation procedure of the simulation is to be executed.

```
# connect all models to each other
sim_help.connect_entities(
    world=world,
    dict_entities=dict_entities,
    model_connect_config=model_connect_config,
    dict_simulators=dict_simulators,
)
```

Run Simulation

```
world.run(until=scenario_config["n_steps"], print_progress=True)
```

Just go Ahead and Run a Simulation

- **You can run one of the `test_scenario` s in the `examples` folder**

  `building`: Just one single building with a bunch of different devices to see the operation of devices inside the household.

  `grid`: Simple low voltage grid (2 feeders with six resp. two household connection points) to get an estimation of the impact of different operating strategies on the local grid.

- While running the simulation, some stuff is logged like creation of simulators, entities, and connections. Additionally, mosaik plots the progress of the whole simulation (like how many time steps are finished calculating).

- Adapting the parameterization in the *simulation files* can yield quite different results.
- Running one of the simulations will create a `.hdf5` results data file in the folder `/examples/results`.
- You can view the information of this file via the H5Web Extension in Microsoft VSC and plot the profiles (stored under Series and the name of the corresponding simulator) of the used devices.
- In case you want, you can set the parameter *bool_plot* in the *scenario_config* dict to true (only do this for small simulations!), mosaik will then present 4 plots for the connections, the simulation process and the timely duration during the simulation.

Create your own simulation

1. Copy one of the *test scenarios* and delete all of the redundant simulators/devices/connections.
2. Set up corresponding *model* (and *grid*) data as well as a *model connection file*.

Configuration of a Model

A model, as explained in the *glossary*, represents a real process or component in form of code. It follows the general structure of *Input - internal calculation - output*, which is triggered by its corresponding *simulator* that needs to be existent for using the model in simulations. The model is just representing the calculation steps and an entity (see the *glossary*) is the instantiation of a model. So within a scenario it is possible to have multiple entities (like battery storages) for one model type (battery storage model).

This page explains the structure of a `_model.py` to enable you to create your own model. As there can be different implementations of a device type (like storage system), the can be more than one class within a `_model.py` (like a very detailled model and a generic model).

Note: All code-blocks derive from `charging_station_model.py` as of (01/24) if not stated otherwise.

Imports

Listing 8: `Import` relevant packages for the model calculation

```python
import warnings
import math
from eelib.utils import cos_phi_fix
```

Class definition

Listing 9: Short explanation, listing of all model parameters with their allowed values for initialization (+ method to `return` these).

```python
class ChargingStation:
    """Models a charging station for electric vehicles of different types."""

    # Valid values and types for each parameter
```

(continues on next page)

(continued from previous page)

```
    _VALID_PARAMETERS = {
        "p_rated": {"types": [float], "values": (0, math.inf)},
        "output_type": {"types": [str], "values": ["AC", "DC"]},
        "charge_efficiency": {"types": [float, int], "values": (0, 1)},
        "discharge_efficiency": {"types": [float, int], "values": (0, 1)},
        "cos_phi": {"types": [float, int], "values": (0, 1)},
    }

    @classmethod
    def get_valid_parameters(cls):
        """Returns dictionary containing valid parameter types and values.

        Returns:
            dict: valid parameters for this model
        """
        return cls._VALID_PARAMETERS
```

E.g. the rated power `p_rated` of a charging station has to be a floating point value on should be non-negative. The `efficiency` should have a value between zero and one (100%). These values are used when creating a model in simulations to check for correct values.

Initialization method `__init__()`

Listing 10: takes parameter values as inputs

```
def __init__(
    self,
    ename: str,
    p_rated: int,
    output_type: str = "AC",
    charge_efficiency: float = 0.99,
    discharge_efficiency: float = 0.99,
    cos_phi: float = 1.0,
    step_size=60 * 15,  # step size in seconds
):
```

Listing 11: creates entity of this model by setting the properties and initializing attributes

```
     # Set attributes of init_vals to static properties
     self.ename = ename
     self.p_rated = p_rated  # rated active power (AC/DC) [W]
     self.output_type = output_type  # source AC/DC [-]
     self.discharge_efficiency = discharge_efficiency  # discharging efficiency [-]
     self.charge_efficiency = charge_efficiency  # charging efficiency [-]
     self.cos_phi = cos_phi

     # initialize dynamic output properties
     self.p = 0  # Active Power (after control) [W]
```

(continues on next page)

(continued from previous page)

```
    self.q = 0  # Reactive Power (after control) [W]
    self.p_device = {}  # active power for every vehicle [W]
    self.p_min = 0  # Minimal active Power [W]
    self.p_max = 0  # Maximal active Power [W]

    ...

    # save time step length and current time step
    self.step_size = step_size
    self.time = 0
```

Model methods

Listing 12: Type-specific function like calculation of power limits, aging, efficiency, adaption of stored energy etc.

```
def _calc_power_limits(self):
    """Calculate the power limits for the charging station with the input thats coming from the
    electric vehicles.

    Raises:
        ValueError: If the power limits of at least one connected ev do not work together.
    """

    # calculate current efficiency depending on the direction of power flow
    self._calc_current_efficiency()

    # in case no ev is connected to cs - no active power flexibility
    self.p_min = 0
    self.p_max = 0
    for ev_id, ev_data in self.ev_data.items():
        # check for each ev if connected - consider their limits, efficiency and nominal power
        if ev_data.appearance:
            # check if min. and max. power are correct
            if ev_data.p_min > ev_data.p_max:
                raise ValueError(f"Min. and max. power of ev {ev_id} do not comply.")
            # handle the power limits
            self.p_min = max(
                self.p_min + ev_data.p_min / self.efficiency,
                -self.p_rated,
            )
            self.p_max = min(
                self.p_max + ev_data.p_max / self.efficiency,
                self.p_rated,
            )
```

(continues on next page)

(continued from previous page)

```
...
```

`step()` method

Within the `step()` method, the calculation processes are executed for the model entity.

This example is coming from the `storage_model.py` (01/24).

Listing 13: For handling of the processes of the model (its calculation) within a time step

```
def step(self, time):
    """Performs simulation step of eELib battery model.
    Calculates all of the Dynamic Properties based on the Input Properties.

    Args:
        time (int): Current simulation time
    """
```

Listing 14: At first: Always handling of a new time step (if entity was called for first time, do some processes once, like adapting energy). This is needed for storages etc., but may not be needed for other model types.

```
    # handle current time step
    if not self.time == time:
        self.time = time

        # adapt energy content from last time step ( + self-discharge)
        if self.p >= 0:  # charging
            self.e_bat_step_volume = (
                self.p * self.charge_efficiency * (self.step_size / 3600)
            )
        else:  # discharging
            self.e_bat_step_volume = (
                self.p / self.discharge_efficiency * (self.step_size / 3600)
            )
        self.e_bat += self.e_bat_step_volume

        # Calculate battery cycles
        self.bat_cycles += abs(self.e_bat_step_volume / self.e_cycle)

        # Calculate battery state of health and aging properties
        if self.status_aging:
            self.__calculate_aging_status()
```

Listing 15: Call model-specific methods in supposed order

```
259     # Set active power and energy within limits
260     self.__set_power_within_limit()
261     self.__set_energy_within_limit()
262     self.soc = self.e_bat / self.e_bat_usable
263
264     self.__calc_charging_efficiency()
265
266     self.__calc_power_limits()
```

Checklist for adding / enhancing a model

What changes?

adapting current implementation?
Try to make use of the existing properties and methods of the model

adding new implementation (e.g. new method) or need for new properties?

1. Add the part of code to the model
2. Write proper comments and documentation (docstrings for every method!)
3. Write a corresponding test function!

New packages have been added?
add them to the `requirements.txt` file

Where to add?

New model attributes need to be …

1. … added to the `META` of the simulator.
2. If they are also input data, add them to the `model_data` of the test scenarios (`examples/data/model_data_scenario`) as well as the `VALID_PARAMETERS` of the model.

New connections between properties of different models …

… need to be integrated to the `model_connections/model_connect_config.json` file, simply paste them in the direction of the connection.

… in case of connections in both direction, the setup of strong (first sent value) and weak (at beginning only done with default value) *connection* has to be set in `model_connections/connect_directions_config.json` file. Always use lower-case letters for the model names!

New models need to be integrated …

1. … into the test scenario scripts (in the `SIM_CONFIG`).
2. … into the `model_data` of the test scenarios (`examples/data/model_data_scenario`).
3. … into the `model_connections/model_connect_config.json` file with their connections to/from other models. If the direction of the connection is of importance, there may be a need to adapt the *simulation_helper* functions *connect_entities()* and *connect_entities_of_two_model_types()*.
4. … into the simulator `META` data.

5. ... with a unit test file that checks all the relevant model functionalities.

Configuration of a Simulator

The simulator of a model, as explained in the *glossary*, works as out interface to communicate between the orchestrator of the simulation and the model entities. This means, that the simulator for the PV system model sends input values to the pv systems, initiates the model calculations and collects output values to return to the orchestrator (e.g. for sending the power values to the connected energy management systems). As there may be multiple entities of a model connected to a single simulator, you only need one simulator of a model type for a simulation, as this simulator makes the communication for all entities of its model type. Hence, a `_simulator.py` is always paired with a *_model.py*.

This page explains the structure of a `_simulator.py` to enable you to create a simulator for your own model.

Note: All code-blocks derive from `charging_station_simulator.py` as of (01/24) if not stated otherwise.

Import mosaik and model

Listing 16: `Import` of the mosaik simulator base class `mosaik_api_v3` and the corresponding model + additional imports

```
import mosaik_api_v3
from eelib.core.devices.charging_station import charging_station_model
import eelib.utils.validation as vld
from copy import deepcopy
```

Listing of model `META`

Listing 17: State of simulator (whether it is time-discrete or event-based/hybrid), see the *glossary*

```
META = {
    "type": "hybrid",
```

Listing 18: Listing of **ALL** attributes for each modeltype: At first, all attributes of the model class...

```
    "models": {
        "ChargingStation": {
            "public": True,
            "params": ["init_vals"],
            "attrs": [
                "p",
                "step_size",
                "time",
                ...
```

Listing 19: … **input attributes** also listed in "trigger" list …

```
        "trigger": [
            "p_set",
            "ev_data",
        ],  # input attributes
```

Listing 20: … **output attributes** also listed in ""non-persistent" list.

```
        "non-persistent": [
            "p",
            "p_min",
            "p_max",
            "cs_data",
        ],  # output attributes
```

Initialization of simulator class

Listing 21: Constructor __init__ for this simulator and the initialization function init()

```
class Sim(mosaik_api_v3.Simulator):
    """Simulator class for eELib charging station model."""

    def __init__(self):
        """Constructs an object of the Charging-Station:Sim class."""

        super(Sim, self).__init__(META)

        ...

    def init(self, sid, scenario_config, time_resolution=1.0):
        """Initializes parameters for an object of the Charging-Station:Sim class."""

        ...
```

Following are the so called **core functions** *create()*, *step()* and *get_data()* that are found in every _simulator.py.

Caution: All **core functions** of the simulators are called by mosaik and need to be existent!

Creation of model entities in `create()`

Listing 22: Creation of model entities by assigning ID and name, creating it by its model type, and storing the entity data

```
111 def create(self, num, model_type, init_vals):
112     """Creates entities of the eELib charging station model."""
113
114     # generated next unused ID for entity
115     next_eid = len(self.entities)
116
117     # create empty list for created entities
118     entities_orchestrator = []
119
120     for i in range(next_eid, next_eid + num):
121         # create entity by specified name and ID
122         ename = "%s%s%d" % (model_type, "_", i)
123         full_id = self.sid + "." + ename
124
125         # get class of specific model and create entity with init values after␣
    ↪validation
126         entity_cls = getattr(charging_station_model, model_type)
127         vld.validate_init_parameters(entity_cls, init_vals[i])
128         entity = entity_cls(
129             ename,
130             **init_vals[i],
131             step_size=self.scenario_config["step_size"],
132
133         # add info to the simulators entity-list and current entities
134         self.entities[ename] = {
135             "ename": ename,
136             "etype": model_type,
137             "model": entity,
138             "full_id": full_id,
139         }
140         entities_orchestrator.append({"eid": ename, "type": model_type})
141
142     return entities_orchestrator
```

Stepping of models in `step()`

Withint the `step()` method, `inputs` coming from mosaik are set into the model entity attributes they are intended for. Afterwards, the entities are called and their calculation is executed with the models' `step()` method.

This example is coming from the `storage_simulator.py` (01/24).

Listing 23: Take input data and set values of entities, then call the model calculation

```
175 # assign property values for each entity and attribute with entity ID
176 # process input signals: for the entities (eid), attr is a dict for attributes to be set
```

(continues on next page)

(continued from previous page)

```
for eid, attrs in inputs.items():
    # for the attributes, setter is a dict for entities with corresponding set values
    for attr, setter in attrs.items():
        # set values corresponding to its type
        ...

# call step function for each entity in the list
for ename, entity_dict in self.entities.items():
    entity_dict["model"].step(time)
```

Note: You might `return` the next timestep for when this model should be called again. This is needed for time-based models (e.g. when reading data from a csv file, the next value should be read and send out) and optional for event-based models (e.g. the adaption of energy levels for a storage model is needed, but a heatpump could maybe just be triggered by a new set value).

Handling of output data in `get_data()`

Listing 24: From a defined set of output properties given by the `outputs` parameter from mosaik, the values of the transmitting entities are read and stored into a data dict that is returned

```
for transmitter_ename, attrs in outputs.items():
    # get name for current entity and create dict field
    entry = self.entities[transmitter_ename]
    if transmitter_ename not in self.output_cache:
        self.output_cache[transmitter_ename] = {}

    # loop over all targeted attributes and check if info is available
    for attr in attrs:
        if attr not in self.meta["models"][type(entry["model"]).__name__]["attrs"]:
            raise ValueError("Unknown output attribute: %s" % attr)

        # create empty field for cache and output data
        if attr not in self.output_cache[transmitter_ename]:
            self.output_cache[transmitter_ename][attr] = {}

        output_data_to_save = getattr(entry["model"], attr)

        # check if this value changed
        ...

    # check if nothing is to be send out - send output 1 step later to avoid waiting for␣
↪data
    if not flag_output_changed:
        if self.time == self.scenario_config["n_steps"] - 1:  # is last time step?
            data["time"] = self.time + 1
        else:
```

(continues on next page)

(continued from previous page)

```
            data = {"time": self.time}

    return data
```

For each time step, the output data is continuously stored and compared to the lastly sent data (`output_cache`) such that if nothing new is to be send out, only the time step will be send. In case of the last time step, we need to adjust the signal by transmitting not the current but the next time step to avoid a loop of endless calling between the simulators.

Implementing an EMS strategy

Energy Management Strategies are handled by Energy Management Systems (EMS). To implement a new strategy, additions in multiple places are necessary.

1. Adapt the `EMS_model.py` file and implement the operating strategy

Create a new class that is inheriting from the general *HEMS* class. Don't forget to add the strategy in the valid parameter of the HEMS ABC class.

Listing 25: eelib/core/control/EMS/EMS_model.py (01/24)

```python
class HEMS_default(HEMS):
    """Default strategy for Energy Management System.
    Should be copied and adapted for the use of a specific EMS concept.
    """

    @classmethod
    def get_valid_parameters(cls):
        """Returns dictionary containing valid parameter types and values.

        Returns:
            dict: valid parameters for this model
        """

        # use parent's parameter list and modify them for this class
        result = HEMS.get_valid_parameters().copy()
        result.update({})
        return result

    def __init__(self, ename: str, step_size: int = 900, **kwargs):
        """Initializes the eELib HEMS default model.

        Args:
            ename (str): name of the entity to create
            step_size (int): length of a simulation step in seconds
            **kwargs: initial values for the HEMS entity

        Raises:
            ValueError: Error if selected strategy does not comply with model type.
        """

```

(continues on next page)

(continued from previous page)

```
        # check given strategy
        if "strategy" in kwargs.keys() and kwargs["strategy"] != "HEMS_default":
            raise ValueError("Created a HEMS_default entity with strategy not 'HEMS_
↪default'!")
        else:
            kwargs["strategy"] = "HEMS_default"  # set strategy if not already given

        # call init function of super HEMS class
        super().__init__(ename=ename, step_size=step_size, **kwargs)
```

Add a `step()` function that first calls the HEMS *`step()`* and afterwards implement the functionalities of your operating strategy

Listing 26: eelib/core/control/EMS/EMS_model.py (01/24)

```
def step(self, time):
    """Calculates power set values for each connected component according to the␣
↪strategy.

    Args:
        time (int): Current simulation time
    """

    # execute general processes (aggregation of power values etc.)
    super().step(time)

    # from here on: execute strategy-specific processes
    ...
```

2. Add the strategy and its input to the `model_data` of the scenarios

Listing 27: examples/data/model_data_scenario/model_data_building.json (01/24)

```
{
    "ems": [
        {
            "strategy": "HEMS_default",
            "cs_strategy": "balanced"
        }
    ],
    ...
```

3. Add your EMS class to the `model_connections/model_connect_config.json` file

Add the data that is **sent out**…

Listing 28: eelib/model_connections/model_connect_config.json (01/24)

```
"HEMS_default": {
    "PVLib": [["p_set_pv", "p_set"]],
    "BSS": [
        [
            "p_set_storage",
            "p_set"
        ]
    ],
    "ChargingStation": [
        [
            "p_set_charging_station",
            "p_set"
        ]
    ],
    "Heatpump": [
        [
            "p_th_set_heatpump",
            "p_th_set"
        ]
    ],
    "grid_load": [["p_balance", "p_w"], ["q_balance", "q_var"]]
},
```

… but also to every model, which **data is sent to your HEMS!**

Listing 29: e.g. but not only the BSS

```
"BSS": {
    "HEMS_default": [
        [
            "p_discharge_max",
            "p_min"
        ],
        [
            "p_charge_max",
            "p_max"
        ],
        [
            "p",
            "p"
        ]
    ],
    ...
},
```

1. Add the model with its name and (input/output) attributes to the `META` of the `EMS_simulator`

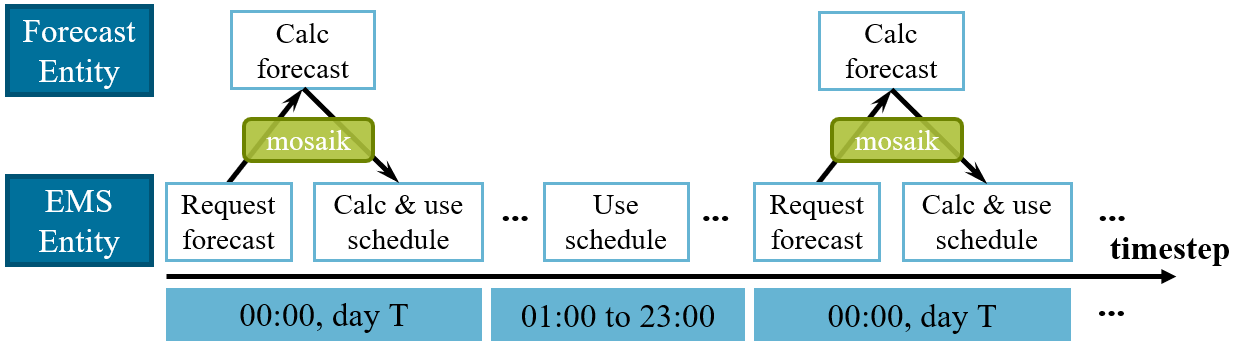
Listing 30: eelib/core/control/EMS/EMS_simulator.py (01/24)

```
14 META = {
15     "type": "hybrid",
16     "models": {
```

```
50         "HEMS_default": {
51             "public": True,
52             "params": ["init_vals"],
53             "attrs": [
54                 "q",
55                 "p",
56                 "p_max",
57                 "p_min",
58                 "p_set_storage",
59                 "p_set_charging_station",
60                 "p_set_pv",
61                 "p_balance",
62                 "q_balance",
63                 "bss_data",
64                 "pv_data",
65                 "cs_data",
66                 "p_th",
67             ],
```

Forecasts and Schedules

Implemented into the eELib is a possibility to calculate a *forecast* and *schedule* for devices, EMSs, and the grid. For this, a forecast model is given to calculate requested forecasts. Additionally, the calculation of schedules can be implemented in each (control) model.



Integration into scenario

Forecasts and schedules can be integrated into simulations like the `examples/test_scenario_***.py` files examplarily show. In there, one has to declare the forecast simulator to mosaik via the `SIM_CONFIG`. Additionally, one has to set the parameter `USE_FORECAST` in the scenario configuration to true.

Listing 31: simulator configuration [test_scenario_building.py 01/24]

```
# Sim config.: Simulators and their used model types with the properties to store into DB
SIM_CONFIG = {
    # used database, will be left out for model creation and connections
    "DBSim": {"python": "eelib.data.database.hdf5:Hdf5Database"},
    # forecast, will be left out for model creation and connections
    "ForecastSim": {
        "python": "eelib.core.control.forecast.forecast_simulator:Sim",
        "models": {"Forecast": []},
    },
```

Listing 32: scenario configuration [test_scenario_building.py 01/24]

```
# Configuration of scenario: time and granularity
START = "2020-01-01 00:00:00"
END = "2020-01-04 00:00:00"
STEP_SIZE_IN_SECONDS = 900  # 1=sec-steps, 3600=hour-steps, 900=15min-steps, 600=10min-
↪steps
USE_FORECAST = True
```

After this, the forecast simulator and model will be created and the connections to the models will be instantiated.

Forecast Model

The forecast model and its simulator are implemented in the folder `eelib/core/control/forecast`. The implemented behaviour is quite simple and straight-forward, as all model entities of the simulation are stored within the forecast entity (as deep-copies) via the `add_forecasted_entity()` method. This allows to create forecasts by simply iterating over all requested forecasts for all entities, stepping the model for the requested timesteps and collecting the values (of the model entity) for the requested time steps. After that, the forecast model simply returns those calculated forecasts.

Listing 33: forecast calculation [forecast_model.py 01/24]

```
# check if request for a forecast was sent
if self.forecast_request == {}:
    self.forecast = {}  # no forecast requested, simply return
else:
    # clear earlier forecasts
    self.forecast = {}

    # go by all entities to create a forecast for
    for forecast_getter_id, forecast_req_eid_dict in self.forecast_request.items():
        # create empty dict for forecasts connected to this request entity
        self.forecast[forecast_getter_id] = {}
        # check if no forecast requested
```

(continues on next page)

(continued from previous page)

```
76      if forecast_req_eid_dict == {}:
77          continue
78
79      # go by all forecasted model entities
80      for forecast_eid, forecast_info in forecast_req_eid_dict.items():
81          # check if forecast can be done for this model type
82          if forecast_eid in self.forecast_eid.keys():
83              # create structure for forecast of each attribute for this entity
84              forecast_save = {}
85              for attr in forecast_info["attr"]:
86                  forecast_save[attr] = []
87
88              # store the copy of this model entity to execute the stepping
89              entity = self.forecast_eid[forecast_eid]
90
91              # run the model for each time step and collect the calculated attr values
92              for t in forecast_info["t"]:
93                  entity.step(t)
94                  for attr in forecast_info["attr"]:
95                      forecast_save[attr].append(getattr(entity, attr))
```

Integration into (Control) Models

For the forecast model to take effect, the forecasts have to be requested by other models, e.g. the energy management system. Additionally, the calculated forecasts should afterwards be used for strategic behaviour (in operating strategies).

First of all, the mosaik needs to create a connection between the model and the forecast model. This is done in the `connect_to_forecast()` function of the simulation helpers. Here, there is a connection added from EMS to the forecast model for the `forecast_request` attribute, while a `forecast` attribute is send back the other way around. All other models are simply added to the forecast entity such that forecasts can be calculated.

Listing 34: forecast connection function [simulation_helper.py 01/24]

```
470 def connect_to_forecast(
471     world: object,
472     dict_entities: dict,
473     dict_simulators: dict,
474     forecast: object,
475     forecast_sim: object,
476 ):
477     """Create connections for the forecasts to work.
478     Includes mosaik connections to ems model and adding of the model entities to the␣
    ↪forecasts list.
479
480     Args:
481         world (object): mosaik world object to orchestrate the simulation process
482         dict_entities (dict): dict of all used model entity objects
483         dict_simulators (dict): dict of all used simulators with their ModelFactory-objects
484         forecast (object): forecast model entity
485         forecast_sim (object): simulator for the forecast model
486     """
```

(continues on next page)

(continued from previous page)

```

   # create connections for each entity of each model type
   for model_name, ent_list in dict_entities.items():
      for entity in ent_list:
         # for ems create connections to forecast entity
         if "ems" in model_name or "EMS" in model_name:
            world.connect(entity, forecast, "forecast_request")
            world.connect(
               forecast,
               entity,
               "forecast",
               weak=True,
               initial_data={"forecast": {forecast.full_id: {}}},
            )
         # for other models (devices) add those entities to the forecast entity list
         else:
            forecast_sim.add_forecasted_entity(
               forecast.eid,
               {entity.full_id: dict_simulators[model_name].get_entity_by_id(entity.eid)}
↪,
            )
```

Additionally, forecasts have to be requested by the ems, which should be done only in the corresponding time steps.

Listing 35: forecast request by EMS [EMS_model.py 01/24]

```
# request forecasts if needed
if self.use_forecast and self.calc_forecast:
   self.forecast_request = {}
   for model_type, entity_list in self.controlled_eid_by_type.items():
      if model_type in self.forecasted_attrs.keys():
         # add forecast request for every entity of this model type
         for e_full_id in entity_list.keys():
            self.forecast_request[e_full_id] = {
               "attr": self.forecasted_attrs[model_type],
               "t": range(self.forecast_start, self.forecast_end),
            }
```

Due to the connection by mosaik, the forecasts are calculated and afterwards send back, such that they should be processed. It is now also possible to calculate schedules with set values for the devices that no forecast can be directly extracted from (charging station, heatpump, battery).

Listing 36: handling of forecasts and calculation of schedules in EMS [EMS_model.py 01/24]

```
# CALC SCHEDULE WITH UNCONTROLLED (CSV) DEVICES
if self.forecast != {} and self.calc_forecast is True:
   # calculate the residual load schedule including all not controllable devices
   schedule_residual_uncontrollable = schedule_help.residual_calc_schedule_
↪uncontrollable(...)

   # calc schedules for charging station
```

(continues on next page)

(continued from previous page)

```
    schedule_help.cs_calc_schedule_uncontrolled(...)

    # calc schedules for heatpump
    th_residual_forecast = schedule_help.thermal_calc_forecast_residual(...)
    schedule_help.hp_calc_schedule(...)

    # get schedule from battery storage based on residual load schedule
    schedule_help.bss_calc_schedule(...)
```

FAQ & Glossary

Glossary

Model
: Representation of real processes (assuming simplifications), using the general procedure of "*Input - internal calculation - output*".

Orchestrator
: Coordination of the whole simulation procedure and coupling of the models (**mosaik**).

Simulator
: API/interface for communication between the orchestrator mosaik and the entities of the specific model.

Entity
: Created instance of a model type. In "ename", "etype", "eid" etc. the "e" stands for entity.

PSC
: We use the **passive sign convention** (german: **Verbraucherzaehlpfeilsystem**), therefore loads are positive while generation is negative.

Forecast
: Prediction of a behaviour for a defined time horizon in the future, e.g. power values for the upcoming 24h steps for a household base load from a csv reader.

Schedule
: Calculated set values for a defined time horizon in the future, e.g. target power values for the upcoming 24h steps for a battery storage system.

time-based
: Used for time-discrete simulation of models, e.g. if just some data is collected to be transmitted to other models.

event-based/hybrid
: Used for triggered simulation of models, e.g. if model calculation can be triggered by inputs like the receiving of a power set value.

weak and strong connections for mosaik simulations
: As you may be tasked with adding a new model or directly adding new connections between existing models, you may come across connections between models in multiple directions - either directly or within a circle of multiple models. When that happens, you have to add the order of the connections (into `model_connections/connect_directions_config.json`) for mosaik to know the order in which the models have to be called. Weak connections are first replaced by None-values such that their recieving model is first called and afterwards the strong connection is taken.

Parameters used in test scenarios

START
simulation starting time (e.g. 2023-01-01 00:00:00)

END
simulation ending time (e.g. 2023-01-02 00:00:00 for a full day calculation)

N_STEPS
number of steps that should be simulated, calculated from start and end time (e.g. 96 steps for a day with 15 min time steps)

STEP_SIZE_IN_SECONDS
length of one (pre-defined) time step in seconds (e.g. 15 min steps: 15*60 sec = 900 sec)

USE_FORECAST
True-False value whether to consider forecasts in the simulation or not

FAQ

What is the process of building Model-Factories?
`examplesim = world.start("ExampleSim", eidprefix="Model")`

The resulting `examplesim` is an entity of the class `mosaik.scenario.ModelFactory`. It can create and store the entities of an example model.

What can one do in case of an ImportError when running the example scenarios?
`pip install -e .`

What about an error like `Could not import module:  No module named 'XXX' --> No module named 'XXX'`
when running the example scenarios (after merging main into your branch)?

Maybe some new package was added to the requirements, try to install them again by `pip install -r requirements.txt` and see wether any packages were newly installed.

What attributes does a model have?
For that you should have a look at the `META` of the simulator or optionally within the model class itself.

Listing 37: e.g. EMS_simulator.py (01/24)

```
# SIMULATION META DATA
META = {
    "type": "hybrid",
    "models": {
        "HEMS": {
            "public": True,
            "params": ["init_vals"],
            "attrs": [
                "q",
                "p",
                "p_max",
                "p_min",
...
```

What inputs does a model have?
Have a look at the `VALID_PARAMETERS` stored in each model class:

Listing 38: e.g. EMS_model.py (01/24)

```
# Valid values and types for each parameter that apply for all subclasses
_VALID_PARAMETERS = {
    "cs_strategy": {
        "types": [str],
        "values": ["max_p", "balanced", "night_charging", "solar_charging"],
    },
    "bss_strategy": {
        "types": [None, str],
        "values": [None, "reduce_curtailment"],
    },
}
```

Exemplarily the `model_data` of the test scenarios set (at least some of) these parameters and their initial values for the models:

Listing 39: e.g. model_data_building.json (01/24)

```
{
    "ems": [
        {
            "cs_strategy": "balanced"
        }
    ],
```

What connections does a model have?

For that the eELib provides the `model_connections/model_connect_config.json` file, where all of the `FROM-TO-CONNECTIONS` are given for each model type implemented in the eELib:

Listing 40: e.g. model_connect_config.json (01/24)

```
"HEMS_default": {
    "PVLib": [["p_set_pv", "p_set"]],
    "BSS": [
        [
            "p_set_storage",
            "p_set"
        ]
    ],
    "ChargingStation": [
        [
            "p_set_charging_station",
            "p_set"
        ]
    ],
    "Heatpump": [
        [
            "p_th_set_heatpump",
            "p_th_set"
        ]
    ],
    "grid_load": [["p_balance", "p_w"], ["q_balance", "q_var"]]
```

(continues on next page)

(continued from previous page)

```
},
```

Listing 41: e.g. model_connect_config.json (01/24)

```
"BSS": {
    "HEMS_default": [
        [
            "p_discharge_max",
            "p_min"
        ],
        [
            "p_charge_max",
            "p_max"
        ],
        [
            "p",
            "p"
        ]
    ],
},
```

Extracted from these two examples above can be the sending of values for attributes `p`, `p_min` and `p_max` from `BSS` to `HEMS` and the return of a value for `p_set`.

Units

Make use of the SI-units!!

- **Power**: W
- **Time (e.g. simulation time)**: s
- **Energy**: Wh
- ...

2.1.3 API Reference

The API reference provides detailed descriptions of eElib's classes and methods. This is taken from the implementations of the models, which can be taken from the *public Gitlab-Repository <https://gitlab.com/elenia1/elenia-energy-library>*.

eelib

This stores the main part of the elenia Energy Library, as it contains the models and all other functionalities.

Subpackages

eelib.core

Here, all of the models are stored. This is divided into the devices (like PV system or electric vehicle), control models (like energy management system), grid models (like grid control) and market models (like electricity providers).

Subpackages

eelib.core.control

Subpackages

eelib.core.control.EMS

Submodules

eelib.core.control.EMS.EMS_bss_helper

Helper functions for handling of battery storage systems in EMS.

Author: elenia@TUBS

Module Contents

Functions

bss_calc_balance(→ float)	Calculates the set power values of storage based on given p_balance.
bss_strategy_reduce_curtailment(p_balance, p_max, p_min)	This strategy reduces curtailment of the solar power system
bss_set_energy_within_limit(→ float)	Check that battery energy limits are not surpassed.
bss_calc_e_bat(→ float)	Calculates the energy level of the forecasted battery storage system.
bss_calc_p_limits(→ tuple[float, float])	Calculates the bss active power limits for the forecast using input "bss_e_bat_usable".

bss_calc_balance(*p_balance: float*, *p_max: float*, *p_min: float*) → float

Calculates the set power values of storage based on given p_balance. Resulting in charging and discharging values.

Parameters

- **p_balance** (*float*) – Input p_balance which storage tries to level

- **p_min** (*float*) – Discharge rate of storage
- **p_max** (*float*) – Charge rate of storage

Returns
calculated power set value of bss to level p_balance

Return type
float

Note: This function was created based on the function __set_power_within_limit() in storage_model.

---

bss_strategy_reduce_curtailment(*p_balance: float*, *p_max: float*, *p_min: float*, *bss_charging_limit: float = 0.6*)

This strategy reduces curtailment of the solar power system by reducing the bss' charging power. This decreases self-sufficiency but also the curtailment due to EEG2021 (e.g. feed-in is only allowed at max. 70% of installed power for uncontrolled solar power systems). This operation strategy should not be used in combination with electric vehicle charging. Suitable limits are 30%-60%.

Parameters

- **p_balance** (*float*) – input balance which bss needs to level
- **p_max** (*float*) – maximum charging power of bss
- **p_min** (*float*) – maximum discharging power of bss
- **bss_charging_limit** (*float*) – Charging limit of batter storage. Defaults to 0.6.

Returns
reduced charging power for battery storage

Return type
int

bss_set_energy_within_limit(*bss_data: eelib.data.BSSData*) → float

Check that battery energy limits are not surpassed. Similiar to function in storage_model.py.

Parameters
bss_data (BSSData) – extract from bss dataclass

Returns
energy level of the bss after scheduling step

Return type
float

Note: This function was created based on the function __set_energy_within_limit() in storage_model.

---

bss_calc_e_bat(*step_size: int*, *bss_data: eelib.data.BSSData*) → float

Calculates the energy level of the forecasted battery storage system.

Parameters

- **step_size** (*int*) – simulation step_size
- **bss_data** (BSSData) – extract of dataclass for currently scheduled bss

Returns
energy level of forecasted bss

Return type
float

Note: This function was created based on the function `step()` in `storage_model`.

bss_calc_p_limits(*step_size: int*, *bss_data: eelib.data.BSSData*) → tuple[float, float]

Calculates the bss active power limits for the forecast using input ‘’bss_e_bat_usable”.

Parameters

- **step_size** (*int*) – simulation step_size
- **bss_data** (BSSData) – extract of dataclass for currently scheduled bss

Returns
maximum charging power of bss float: minimum charging (resp. max. discharging) power of bss

Return type
float

Note: This function was created based on the function `__calc_power_limits()` in `storage_model`.

eelib.core.control.EMS.EMS_car_helper

Helper functions for handling of cars in EMS.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Functions

car_calc_e_bat(→ float)	Calculates the energy amount of electric car after power influence (from last step).
car_set_energy_within_limit(→ float)	Check that battery energy limits are not surpassed.
car_calc_power_limits(→ tuple[float, float])	Calculate the maximum (dis)charging power for the car battery.
car_calc_power(→ float)	Calculate the power at the current time step within the (dis)charging limits.

car_calc_e_bat(*step_size: int*, *ev_data: eelib.data.EVData*) → float

Calculates the energy amount of electric car after power influence (from last step).

Parameters

- **step_size** (*int*) – simulation step_size

- **ev_data** (EVData) – contains all information of electric vehicles

Returns
new energy level of electric vehicle (for next step)

Return type
float

NOTE: This function was created based on the function step() in car_model

car_set_energy_within_limit(*ev_data: eelib.data.EVData*) → float

Check that battery energy limits are not surpassed.

Parameters
ev_data (EVData) – contains all information of electric vehicles

Returns
updated energy level of electric vehicle

Return type
float

NOTE: This function was created based on the function set_energy_within_limit() in
car_model.

car_calc_power_limits(*step_size: int*, *ev_data: eelib.data.EVData*) → tuple[float, float]

Calculate the maximum (dis)charging power for the car battery. depending on the power limits and the current stored energy.

Parameters

- **step_size** (*int*) – simulation step_size
- **ev_data** (EVData) – contains all information of electric vehicles

Returns
minimum charging (resp. max. discharging) power of EV float: maximum (charging) power of EV

Return type
float

NOTE: This function was created based on the function _calc_power_limits() in car_model.

car_calc_power(*step_size: int*, *consumption_step: float*, *p_set: float*, *ev_data: eelib.data.EVData*) → float

Calculate the power at the current time step within the (dis)charging limits.

Parameters

- **step_size** (*int*) – simulation step_size
- **consumption_step** (*float*) – consumption at the current time step
- **p_set** (*float*) – set value for charging power of car
- **ev_data** (EVData) – contains all information of electric vehicles

Returns
electric power of the vehicle

Return type
float

Note: This function was created based on the function `__calc_power()` in `car_model`.

eelib.core.control.EMS.EMS_cs_helper

Helper functions for handling of charging stations in EMS.

Author: elenia@TUBS

Module Contents

Functions

cs_strategy_balanced_charging()	The forecast-based charging strategy estimates the total standing time of the connected
cs_strategy_night_charging()	The night charging strategy extends the charging process similar to forecast-based
cs_strategy_solar_charging()	The solar charging strategy is initiated within times of solar energy surplus. The
cs_distribute_charging_power(→ dict)	Distributes the charging power across all connected cars.
cs_calc_power_limits(→ tuple[float, float])	Calculate the power limits for the charging station with the input thats coming from the
cs_calc_current_efficiency(→ float)	Calculates the charging efficiency for charging station from static properties.

cs_strategy_balanced_charging()

The forecast-based charging strategy estimates the total standing time of the connected EV at home and then calculates the minimal charging power needed to fully charge the EV up to the moment of the next departure.

cs_strategy_night_charging()

The night charging strategy extends the charging process similar to forecast-based charging. The strategy is solely activated and processed overnight between 20:00 and 05:00 when grid load is typically lower in residential areas. When the EV is not available overnight, maximum power charging is instead activated.

cs_strategy_solar_charging()

The solar charging strategy is initiated within times of solar energy surplus. The cs model receives the information of such a surplus from its control model (e.g. EMS).

cs_distribute_charging_power(*forecast: dict*, *t: int*, *cs_data: eelib.data.CSData*) → dict

Distributes the charging power across all connected cars. For this, the distribution is done evenly unless the power limits of the cars are exceeded.

Parameters

- **forecast** (*dict*) – collected forecast from EMS
- **t** (*int*) – control variable in range of current forecast

- **cs_data** (CSData) – contains all information of charging station

Raises

- **ValueError** – If Vehicles charging power does not match the power of the charging station
- **ValueError** – If set charging power does not fit the power limits of the connected evs
- **ValueError** – If charging station has power value although no car is connected
- **ValueError** – If sum of vehicle power does not match power

Returns
individual charging power for all connected cars

Return type
dict

Note: This function was created based on the function `_distribute_charging_power()` in `charging_station_model`.

---

cs_calc_power_limits(*forecast: dict*, *t: int*, *cs_data: eelib.data.CSData*) → tuple[float, float]

Calculate the power limits for the charging station with the input thats coming from the electric vehicles.

Parameters

- **forecast** (*dict*) – collected forecast from EMS
- **t** (*int*) – control variable in range of current forecast
- **cs_data** (CSData) – contains all information of charging stations

Raises
ValueError – If the power limits of at least one connected ev do not work together.

Returns
maximum charging power of charging station float: maximum discharging (resp. minimum charging) power of charging station

Return type
float

Note: This function was created based on the function `_calc_power_limits()` in `charging_station_model`.

---

cs_calc_current_efficiency(*cs_data: eelib.data.CSData*) → float

Calculates the charging efficiency for charging station from static properties. Based on the active power.

Parameters
cs_data (CSData) – contains all information of charging station

Returns
efficiency for dis/charging process of cs

Return type
float

Note: This function was created based on the function `_calc_current_efficiency()` in `charging_station_model`.

---

eelib.core.control.EMS.EMS_hp_helper

Helper functions for handling of heatpumps in EMS.

Author: elenia@TUBS

Module Contents

Functions

hp_set_power_with_limits(→ float)	Set the thermal power of a heatpump with regard to its three limits (max, min, min_on).
set_hp_time(→ int)	Update on-/off-time of heatpump with regard to its current power output.
calc_hp_power_limits(hp_p_rated, hp_p_min_th_rel, hp_state)	Set thermal power limits of heatpump with regard to state.
set_hp_state(→ str)	Update state of heatpump with regard to its on- or off-time and minimum state time.

hp_set_power_with_limits(*hp_p_th_max: float*, *hp_p_th_min_on: float*, *hp_p_th_min: float*, *hp_p_th_target: float*) → float

Set the thermal power of a heatpump with regard to its three limits (max, min, min_on).

Parameters

- **hp_p_th_max** (*float*) – maximum thermal power
- **hp_p_th_min_on** (*float*) – minimum thermal power in on-state
- **hp_p_th_min** (*float*) – minimum thermal power overall
- **hp_p_th_target** (*float*) – desired power value

Returns
thermal power set value according to power limits

Return type
float

Note: This function was created based on the function `step()` in `heatpump_model`.

set_hp_time(*step_size: int*, *p_th_set: float*, *hp_time_on: int*, *hp_time_off: int*) → int

Update on-/off-time of heatpump with regard to its current power output.

Parameters

- **step_size** (*int*) – current simulation step size
- **p_th_set** (*float*) – current thermal power output
- **hp_time_on** (*int*) – current time in on-state [s]

- **hp_time_off** (*int*) – current time in off-state [s]

Returns

(1) updated time in on-state [s]

(2) updated time in off-state [s]

Return type
int, int

Note: This function was created based on the function `step()` in `heatpump_model`.

`calc_hp_power_limits`(*hp_p_rated: float*, *hp_p_min_th_rel: float*, *hp_state: str*)

Set thermal power limits of heatpump with regard to state.

Parameters

- **hp_p_rated** (*float*) – rated thermal power of heatpump
- **hp_p_min_th_rel** (*float*) – relative minimum thermal power in on-state (regarding rated)
- **hp_state** (*str*) – current state of the heatpump

Returns

(1) maximum thermal power

(2) minimum thermal power in on-state

(3) minimum thermal power

Return type
float, float, float

Note: This function was created based on the function `__calc_thermal_limits()` in `heatpump_model`.

`set_hp_state`(*p_th_set: float*, *hp_time_on: int*, *hp_time_off: int*, *hp_time_min: int*) → str

Update state of heatpump with regard to its on- or off-time and minimum state time.

Parameters

- **p_th_set** (*float*) – current thermal power output
- **hp_time_on** (*int*) – current time in on-state [s]
- **hp_time_off** (*int*) – current time in off-state [s]
- **hp_time_min** (*int*) – minimum time for one state [s]

Returns
updated state of the heatpump

Return type
str

Note: This function was created based on the function `step()` in `heatpump_model`.

---

eelib.core.control.EMS.EMS_model

The eElib EMS model was developed as an energy management system on the building level and is therefore connected to all of the building's components. The control gathers all of the component's power flows, calculates the energy balance at the building node and the component's flexibility, and then sends out power set values in each step. Hereby, available solar power is favored to grid consumption, as well as charging / discharging limits of the charging stations, and batteries are considered.

Regarding charging stations, different charging strategies are implemented: Default / Maximum power charging, balanced charging, solar charging and night charging.

Author: elenia@TUBS

Module Contents

Classes

HEMS	Models an home energy managament system on building level.
HEMS_default	Default strategy for Energy Management System.
GCP_Aggregator_HEMS	Aggregation of power flows for depiction of uncontrolled devices in a building.
HEMS_forecast_base	Baseclass for EMS strategies using forecasts (and optimizations).
HEMS_forecast_default	Default forecast strategy for Energy Management System.
HEMS_forecast_opt	Optimization-based forecast/schedule strategy for Energy Management System.

class HEMS(*ename: str*, *step_size: int*, *cs_strategy: str = 'max_p'*, *bss_strategy: str = 'surplus'*)

Bases: abc.ABC

Models an home energy managament system on building level.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

add_controlled_entity(*entities_dict: dict*)

Adds a controlled entity to the corresponding EMS's lists (controlled components and type-specific).

Parameters
entities_dict (*dict*) – Input list with all models of a type (e.g. charging_station) which will be connected to the controller.

_get_component_by_type(*type: str*)

Gets all controlled components of passed type.

Parameters
type (*str*) – Query to determine all controlled components of this type.

Returns
Returns all entities of queried type.

Return type
dict

abstract step(*time: int*)

Handles stepping of energy managament systems by checking power values.

Parameters
time (*int*) – Current simulation time

_calc_power_energy(*time: int*)

Gathers current power flows, aggregates limits, and calculates energy balances.

Parameters
time (*int*) – Current simulation time

_set_pv_max()

Calculate pv system setpoints for all devices by using the maximum power.

_set_cs_p()

Calculate charging station setpoints using the chosen charging strategy.

_set_hp_p_th()

Calculate heatpump thermal power setpoints by reducing needed power at this step.

_set_bss_p()

Calculate battery storage system setpoints for all devices using the chosen strategy.

_reset_financial_values()

Reset all financial values (at the beginning of a new timestep).

_handle_incoming_signals()

Retrieve tariff settings for this timestep from an incoming tariff signal.

_calc_financial_output()

Calculate financial values like cost, profit and revenue for this timestep.

class HEMS_default(*ename: str*, *step_size: int = 900*, ***kwargs*)

Bases: *HEMS*

Default strategy for Energy Management System. Should be copied and adapted for the use of a specific EMS concept.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time: int*)

Calculates power set values for each connected component according to the strategy.

Parameters
time (*int*) – Current simulation time

class GCP_Aggregator_HEMS(*ename: str*, *step_size: int = 900*, ***kwargs*)

Bases: *HEMS*

Aggregation of power flows for depiction of uncontrolled devices in a building.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*)

Aggregation of power values.

Parameters
time (*int*) – Current simulation time

class HEMS_forecast_base(*ename: str*, *step_size: int*, *forecast_horizon_hours: int = 24*, *forecast_frequency_hours: int = 24*, *forecast_type: str = 'household_only'*, *cs_strategy: str = 'max_p'*, *bss_strategy: str = 'surplus'*)

Bases: *HEMS*

Baseclass for EMS strategies using forecasts (and optimizations).

forecast_types

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_request_forecast()

Set models with their attributes and time horizon for forecast request.

_set_schedule_values()

Take schedules of the devices and convert them into set values.

_set_power_in_limits(*power: float*, *p_min: float*, *p_max: float*) → float

Set a power value with regard to a maximum and minimum value, if given.

Parameters

- **power** (*float*) – current power value to start with
- **p_min** (*float*) – minimum power limit
- **p_max** (*float*) – maximum power limit

Returns
power value within the limiting bounds

Return type
float

_set_schedule_values_pv()

Take schedules of pv systems and convert them to set values by using the power limits.

_set_schedule_values_cs()

Take schedules of charging stations and convert to set values by using the limits.

_set_schedule_values_hp()

Take schedules of heatpumps and convert them to set values by using the power limits.

_set_schedule_values_bss()

Take schedules of battery storage systems and convert to set values using the limits.

abstract step(*time: int*)

Gathers current energy flows and calculates energy balances.

Parameters
time (*int*) – Current simulation time

class HEMS_forecast_default(*ename: str*, *step_size: int = 900*, ***kwargs*)

Bases: *HEMS_forecast_base*

Default forecast strategy for Energy Management System. Should be copied and adapted for the use of a specific EMS concept.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time: int*)

Calculates power set values for each connected component according to the strategy.

Parameters
time (*int*) – Current simulation time

class HEMS_forecast_opt(*ename: str*, *step_size: int = 900*, ***kwargs*)

Bases: *HEMS_forecast_base*

Optimization-based forecast/schedule strategy for Energy Management System.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time: int*)

Calculates power set values for each connected component according to the strategy.

Parameters
time (*int*) – Current simulation time

eelib.core.control.EMS.EMS_simulator

Mosaik interface for the eELib energy management system (EMS) model. Simulator for communication between orchestrator (mosaik) and EMS entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib EMS model.

Attributes

META
ADAPTION_TOLERANCE

META

ADAPTION_TOLERANCE

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib EMS model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Yields
object – Initializes a mosaik event to return set_data.

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the EMS:Sim class.

Parameters

- **sid** (*str*) – ID of the created entity of the simulator (e.g. EMSSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – fitting of the step size to the simulation scenario step size.

Returns
description of the simulator

Return type
meta

`create`(*num*, *model_type*, *init_vals=None*)

Creates entities of the eELib EMS model. Core function of mosaik.

Parameters

- **num** (*int*) – number of models to be created
- **model_type** (*str*) – type of created entity (e.g. "HEMS")
- **init_vals** (*list*) – list with initial values for each EMS entity, defaults to None

Returns
created entities

Return type
dict

`get_entity_by_id`(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib EMS model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict*) – allocation of set values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises
TypeError – if value_dict containing set values has an unknown format

Returns
next time step (when orchestrator calls again)

Return type
int

`get_data`(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each control model

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

add_controlled_entity(*control_entity_eid: str*, *entity_dict: dict*)

Adds entities (e.g. pv systems) to the specific control unit entity.

Parameters

- **control_entity_eid** (*str*) – entity id of control unit entity
- **entity_dict** (*dict*) – Dictionary of created models to be added to the control unit entity

eelib.core.control.EMS.schedule_helper

Helper functions for the creation of schedules in EMS.

Pyomo released under 3-clause BSD license

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Functions

calc_forecast_residual(→ numpy.ndarray)	Calculates the residual forecast for the "p"-values of all devices' forecasts.
calc_forecast_thermal_residual(→ numpy.ndarray)	Calculates the residual forecast for the thermal demand ("p_th"-values).
bss_calc_schedule(→ dict)	Calculates the schedule for the p_set values of the bss.
pv_calc_schedule(→ dict)	Calculates the schedule for all pv systems using the forecast (maximum power).
hp_calc_schedule(→ dict)	Calculates the schedule for the thermal power set values of the heatpump.
cs_calc_schedule_uncontrolled(step_size, forecast, ...)	Calculates the schedule for all connected charging stations and its connected cars.
calc_schedule_opt(, tariff, cs_data, bss_data, ...)	Calculates an optimal schedule for all connected devices.

Attributes

_logger

_logger

calc_forecast_residual(*forecast: dict*, *forecast_horizon: int*) → numpy.ndarray

Calculates the residual forecast for the "p"-values of all devices' forecasts.

Parameters

- **forecast** (*dict*) – input forecasts to calculate schedule
- **forecast_horizon** (*int*) – time steps in which forecast is created

Returns
residual electrical power values from forecasts for attribute "p" for devices

Return type
np.ndarray

calc_forecast_thermal_residual(*forecast: dict*, *forecast_horizon: int*) → numpy.ndarray

Calculates the residual forecast for the thermal demand ("p_th"-values).

Parameters

- **forecast** (*dict*) – input forecasts to calculate schedule
- **forecast_horizon** (*int*) – time steps in which forecast is created

Returns
residual thermal demand values from forecast

Return type
np.ndarray

bss_calc_schedule(*step_size: int*, *schedule_p_res: list*, *bss_data: dict*) → dict

Calculates the schedule for the p_set values of the bss.

Parameters

- **step_size** (*int*) – simulation step_size
- **schedule_p_res** (*list*) – Input schedule of residual load.
- **bss_data** (*dict*) – battery information at start of forecast with structure {id: BSSData}

Returns
Schedule with `p_set` values of the bss in the forecast horizon.

Return type
dict

pv_calc_schedule(*forecast: dict*, *pv_data: dict*) → dict

Calculates the schedule for all pv systems using the forecast (maximum power).

Parameters

- **forecast** (*dict*) – collected forecast from EMS
- **pv_data** (*dict*) – contains all information of pv systems with structure {id: PVData}

Returns
Schedule with power set values of the pv systems in the forecast horizon.

Return type
dict

hp_calc_schedule(*step_size: int*, *th_demand_profile: list*, *energy_demand_th: float*, *hp_data: dict*) → dict

Calculates the schedule for the thermal power set values of the heatpump.

Parameters

- **step_size** (*int*) – simulation step_size
- **th_demand_profile** (*list*) – input profile of the thermal power demand.
- **energy_demand_th** (*float*) – current thermic energy demand
- **hp_data** (*dict*) – contains all information of heatpumps with structure {id: HPData}

Returns
Schedule with thermal and electrical set values of the hps in the forecast horizon.

Return type
dict

cs_calc_schedule_uncontrolled(*step_size: int*, *forecast: dict*, *forecast_horizon: int*, *cs_data: dict*)

Calculates the schedule for all connected charging stations and its connected cars. Uses the forecast of the appereance and the cars consumption. NOTE: For this schedule the car is always charged with max. power.

Parameters

- **step_size** (*int*) – simulation step size in seconds
- **forecast** (*dict*) – collected forecast from EMS
- **forecast_horizon** (*int*) – length of the forecast period
- **cs_data** (*dict*) – contains all information of charging stations with structure {id: CSData}

Returns
Schedule with power set values of the charging stations in the forecast horizon.

Return type
dict

calc_schedule_opt(*step_size: int*, *forecast: dict*, *forecast_horizon: int*, *opt: eelib.data.OptimOptions = OptimOptions()*, *tariff: eelib.data.TariffSignal = TariffSignal()*, *cs_data: dict = {}*, *bss_data: dict = {}*, *pv_data: dict = {}*, *hp_data: dict = {}*, *control_signals: eelib.data.ControlSignalEMS = ControlSignalEMS()*) → dict

Calculates an optimal schedule for all connected devices.

Parameters

- **step_size** (*int*) – simulation step_size
- **forecast** (*dict*) – collected forecasts from EMS
- **forecast_horizon** (*int*) – timesteps for which forecast should be calculated
- **opt** (OptimOptions) – information about optimization setup (whats used and what not)
- **tariff** (TariffSignal) – electricity price signal. Defaults to TariffSignal().
- **cs_data** (*dict*) – info of charging stations, like {id: CSData}. Defaults to {}.
- **bss_data** (*dict*) – info of battery storage systems, like {id: BSSData}. Defaults to {}.
- **pv_data** (*dict*) – info of pv systems, like {id: PVData}. Defaults to {}.
- **hp_data** (*dict*) – info of heatpumps, like {id: HPData}. Defaults to {}.
- **control_signals** (ControlSignalEMS) – Control signals sent to EMS with lists for power limits.

Raises
ValueError – constructed optimization problem could not be solved by solver

Returns
Schedule with power set values of the connected devices in the forecast horizon.

Return type
dict

eelib.core.control.grid

Submodules

eelib.core.control.grid.grid_ems_model

The eElib grid EMS model is some sort of an implementation for the grid operator, as it is the grid energy management system. The control unit gathers the power flow results of the grid and can send out different kind of control signals, like set values or prices.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

GridEMS	Models a grid energy managament system.

class GridEMS(*ename: str*, *strategy: str*, *grid_model_config: dict*, *use_14a_enwg: bool = True*, *grid_tariff_model: str = 'flat-rate'*, *energy_price_static: float = 0.08*, *capacity_fee_dem: float = 0.0*, *capacity_fee_gen: float = 0.0*, *capacity_fee_horizon_sec: int = 3600 * 24*, *step_size: int = 900*)

Bases: `object`

Models a grid energy managament system.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*)

Gathers current grid status, handles it and calculates output values.

Parameters
time (*int*) – Current simulation time

_create_grid_tariff_signal()

Create the grid tariff signal to send to retail electricity providers and households.

Raises
ValueError – In case grid tariff model is not implemented

eelib.core.control.grid.grid_ems_simulator

Mosaik interface for the eELib grid energy management system model. Simulator for communication between orchestrator (mosaik) and grid EMS entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib grid EMS model.

Attributes

META
ADAPTION_TOLERANCE

META

ADAPTION_TOLERANCE

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib grid EMS model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Yields
object – Initializes a mosaik event to return set_data.

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the grid_ems:Sim class.

Parameters

- **sid** (*str*) – ID of the created entity of the simulator (e.g. GridEMSSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – fitting of the step size to the simulation scenario step size.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals=None*)

Creates entities of the eELib grid EMS model. Core function of mosaik.

Parameters

- **num** (*int*) – number of models to be created
- **model_type** (*str*) – type of created entity (e.g. "grid_ems")
- **init_vals** (*list*) – list with initial values for each grid EMS entity, defaults to None

Returns
created entities

Return type
dict

get_entity_by_id(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

step(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib grid EMS model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict*) – allocation of set values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Returns
next time step (when orchestrator calls again)

Return type
int

get_data(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each control model

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

add_controlled_entity(*control_entity_eid: str*, *entity_dict: dict*)

Adds entities (e.g. pv systems) to the specific control unit entity.

Parameters

- **control_entity_eid** (*str*) – entity id of control unit entity
- **entity_dict** (*dict*) – Dictionary of created models to be added to the control unit entity

eelib.core.devices

Subpackages

eelib.core.devices.car

Submodules

eelib.core.devices.car.car_model

eELib car models the mobility and charging behavior of EV. The EV's electricity demand is derived from profiles generated by the open-source tool emopby. The model calculates its soc in every step, based on the cars consumption and possible charging power derived from charging station.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

EV	Models an electric vehicle and its change of soc depending on driving and charging.

Attributes

TOLERANCE_OVERCHARGED

TOLERANCE_OVERCHARGED

class EV(*ename: str*, *start_time: str*, *file_emobpy: str*, *step_size: int = 60 * 15*, *n_steps: int = 1*, *soc_init: float = 0.5*, *set_emobpy_val: bool = True*, *soc_min: float = 0.02*, *e_max: int = 50000*, *p_nom_discharge_max: int = -11000*, *p_nom_charge_max: int = 11000*, *dcharge_efficiency: float = 0.95*, *charge_efficiency: float = 0.99*)

Models an electric vehicle and its change of soc depending on driving and charging.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

__set_emobpy_data(*n_steps: int*)

Collect data from the emobpy pickle file and save it into the model.

Parameters
n_steps (*int*) – number of steps for the simulation

Raises

- **IndexError** – When the time window of the pickle file does not match the simulation window
- **ValueError** – when the time resolutions of simulation and pickle file do not match

step(*time*)

Performs simulation step of eELib ev model.

Parameters
time (*int*) – Current simulation time

__check_appearance()

Check whether the EV is currently connected to the charging point or not.

__calc_next_arrival()

Calculates when the EV is arriving (the index) for the next time.

__calc_appearance_duration()

Calculate the number of time steps that the EV is still going to be connected to the charging station from now on until the next trip.

__calc_power()

Calculate the power at the current time step within the (dis)charging limits.

__set_energy_within_limit()

Check that battery energy limits are not surpassed.

__calc_power_limits()

Calculate the maximum (dis)charging power for the ev battery depending on the power limits and the current stored energy.

__calc_bev_consumption_period()

Calculate the electr. consumption of the BEV while at home for the next trip.

eelib.core.devices.car.car_simulator

Mosaik interface for the eELib car model. Simulator for communication between orchestrator (mosaik) and car entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib car model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib car model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Raises
ValueError – Unknown output attribute, when not described in META of simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the Car:Sim class.

Parameters

- **sid** (*str*) – Id of the created instance of the load simulator (e.g. CarSim-0).
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size.
- **time_resolution** (*float*) – fitting of the step size to the simulation scenario step size.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates instances of the eELib car model.

Parameters

- **num** (*int*) – Number of car models to be created
- **model_type** (*str*) – Description of the created instance (here "Car")
- **init_vals** (*list*) – List (length=num) with initial values for each car model

Returns
created entities

Return type
dict

get_entity_by_id(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

step(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib-car model.

Parameters

- **time** (*int*) – Current simulation time according to step size. Given by MOSAIK.
- **inputs** (*dict*) – Is a dictionary, that maps entity IDs to data dictionaries which map attribute names to lists of values. Given by MOSAIK.
- **max_advance** (*int, optional*) – Is the simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises

- **ValueError** – Error if incoming input has wrong format or includes more than one set value
- **TypeError** – value_dict has unknown format

Returns
New time stamp (time increased by step size)

Return type
int

get_data(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – Dictionary with data outputs from each car model

Raises
ValueError – Error if attribute not in model metadata

Returns
Dictionary with simulation outputs

Return type
dict

eelib.core.devices.charging_station

Submodules

eelib.core.devices.charging_station.charging_station_model

eElib charging station model is built to manage the charging processes of EVs.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

ChargingStation	Models a charging station for electric vehicles of different types.

class ChargingStation(*ename: str*, *p_rated: int*, *output_type: str = 'AC'*, *charge_efficiency: float = 1.0*, *discharge_efficiency: float = 1.0*, *cos_phi: float = 1.0*, *step_size=60 * 15*)

Models a charging station for electric vehicles of different types.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_calc_power_limits()

Calculate the power limits for the charging station with the input thats coming from the electric vehicles.

Raises

ValueError – If the power limits of at least one connected ev do not work together.

_distribute_charging_power()

Distributes the charging power self.p across all connected cars. For this, the distribution is done evenly unless the power limits of the cars are exceeded.

Raises

- **ValueError** – If Vehicles charging power does not match the power of the charging station
- **ValueError** – If set charging power does not fit the power limits of the connected evs
- **ValueError** – If charging station has power value although no car is connected

_calc_current_efficiency()

For the current timestep and based on the active power flow calculate the present efficiency for the charging station.

step(*time*)

Performs simulation step of eELib cs model, using the power limits, set values and assigning power to charged cars.

Parameters

time (*int*) – Current simulation time in seconds

eelib.core.devices.charging_station.charging_station_simulator

Mosaik interface for the eELib charging station model. Simulator for communication between orchestrator (mosaik) and charging station entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib charging station model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib charging station model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Raises
ValueError – Unknown output attribute, when not described in META of simulator.

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the Charging-Station:Sim class.

Parameters

- **sid** (*str*) – Id of the created instance of the simulator (e.g. CSSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – Time resolution of current scenario.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates entities of the eELib charging station model. Core function of mosaik.

Parameters

- **num** (*int*) – Number of cs models to be created
- **model_type** (*str*) – type of created instance (e.g. “charging_station”)
- **init_vals** (*list*) – list with initial values for each charging_station entity

Returns
created entities

Return type
dict

get_entity_by_id(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib charging_station model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises

- **ValueError** – Error if more than one set value is tried to be given to entity
- **TypeError** – value_dict has unknown format

Returns
next time step (when orchestrator calls again)

Return type
int

`get_data`(*outputs*)

Gets the data for the next connected model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each entity

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

eelib.core.devices.heatpump

Submodules

eelib.core.devices.heatpump.heatpump_model

eElib heat pump model for calculation of the thermal generation and electrical load of a heat pump. The model approximates the thermal calculations with power values and their impact on temperatures.

Author: elenia@TUBS

Module Contents

Classes

Heatpump	Models a heatpump.

class Heatpump(*ename: str*, *p_rated_th=10000*, *modulation=1*, *p_min_th_rel=0.5*, *time_min=2100*, *cop=3.60584*, *cos_phi: float = 1.0*, *step_size=60 * 15*)

Models a heatpump.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

__set_state()

Set the state of the heatpump with regard to the current power on on/off time.

__calc_thermal_limits()

Calculate the thermal maximum and minimum power of the heatpump.

step(*time*)

Calculating of current thermal and active power values, as well as the minimum and maximal values while differentiating between modulation or on/off heatpump.

Parameters
time (*int*) – Current simulation time

eelib.core.devices.heatpump.heatpump_simulator

Mosaik interface for the eELib heatpump model. Simulator for communication between orchestrator (mosaik) and heatpump entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

Sim	Simulator class for eELib heatpump model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: `mosaik_api_v3.Simulator`

Simulator class for eELib heatpump model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Raises
ValueError – Unknown output attribute, when not described in META of simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the Heatpump:Sim class.

Parameters

- **sid** (*str*) – ID of the created entity of the simulator (e.g. LoadSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – Time resolution of current scenario.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates entities of the eELib load model. Core function of mosaik.

Parameters

- **num** (*int*) – number of load models to created
- **model_type** (*str*) – type of created instance (e.g. “household”)
- **init_vals** (*list*) – list with initial values for each load entity

Returns
created entities

Return type
dict

`get_entity_by_id`(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib load model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models (NOT NEEDED FOR LOAD MODEL)
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises

- **ValueError** – Error if more than one set value is tried to be given to entity
- **TypeError** – value_dict has unknown format

Returns
next time step (when orchestrator calls again)

Return type
int

`get_data`(*outputs*)

Gets the data for the next connected model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each entity

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

eelib.core.devices.pv

Submodules

eelib.core.devices.pv.pv_lib_model

eElib pvlib photovoltaic model. The pv module and inverter are both modeled together.

Caution: It contains weather data from different years between 2005 and 2016 and handles them as if they all belong to the same simulated year.

See https://pvlib-python.readthedocs.io/en/stable/index.html for further reference: William F. Holmgren, Clifford W. Hansen, and Mark A. Mikofski. 'pvlib python: a python package for modeling solar energy systems.' Journal of Open Source Software, 3(29), 884, (2018). https://doi.org/10.5281/zenodo.8368494

Author: elenia@TUBS

Module Contents

Classes

PVLibBase	Baseclass model for pv lib model implementations.
PVLib	An electrical photovoltaic system model from pv lib.
PVLibExact	An electrical photovoltaic system model from pv lib using specific modules and inverters.

class PVLibBase(*ename: str*, *latitude: float = 57.38*, *longitude: float = 36.23*, *start_time: str = '2016-01-01 00:00:00'*, *timezone: str = 'Europe/Berlin'*, *azimuth: float = 180*, *tilt: float = 35.0*, *cos_phi: float = 0.9*, *step_size: int = 60 * 15*)

Bases: `abc.ABC`

Baseclass model for pv lib model implementations.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_retrieve_weather_step(*time: int*)

Calculate current weather data.

Parameters
time (*int*) – Current simulation time step

_set_power()

Set active power from set value or to maximum and within limits, set reactive power.

abstract step(*time: int*)

Handles stepping of pvlib entity.

Parameters
time (*int*) – Current simulation time

class PVLib(*ename: str*, *p_rated: float = 10000*, *latitude: float = 57.38*, *longitude: float = 36.23*, *start_time: str = '2016-01-01 00:00:00'*, *timezone: str = 'Europe/Berlin'*, *azimuth: float = 180*, *tilt: float = 35.0*, *gamma_pdc: float = -0.004*, *cos_phi: float = 0.9*, *inverter_efficiency: float = 0.96*, *losses_standby: float = 3.6*, *min_power: float = 56.0*, *step_size: int = 60 * 15*)

Bases: *PVLibBase*

An electrical photovoltaic system model from pv lib.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time: int*)

Performs simulation step of eELib pv_lib model. Simply extracts the value corresponding to the current simulation time.

Parameters
time (*int*) – Current simulation time

class PVLibExact(*ename: str*, *latitude: float = 57.38*, *longitude: float = 36.23*, *altitude_m: float = 75*, *module_name: str = 'SunPower_SPR_220__PVL____2006_'*, *num_modules_per_string: int = 9*, *num_strings: int = 4*, *inverter_name: str = 'SMA_America__SB5000US__240V_'*, *num_inverters: int = 2*, *start_time: str = '2016-01-01 00:00:00'*, *timezone: str = 'Europe/Berlin'*, *azimuth: float = 180*, *tilt: float = 35*, *cos_phi: float = 0.9*, *step_size: int = 60 * 15*)

Bases: *PVLibBase*

An electrical photovoltaic system model from pv lib using specific modules and inverters.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time: int*)

Performs simulation step of eELib pv_lib model. Simply extracts the value corresponding to the current simulation time.

Parameters
time (*int*) – Current simulation time

eelib.core.devices.pv.pv_lib_simulator

Mosaik interface for the eELib pvlib pv model. Simulator for communication between orchestrator (mosaik) and pv_lib entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

Sim	Simulator class for eELib pv_lib model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib pv_lib model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the PVLib:Sim class.

Parameters

- **sid** (*str*) – Id of the created instance of the pv_lib simulator (e.g. PVLibSim-0)

- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size.
- **time_resolution** (*float*) – Time resolution of current mosaik scenario.

Returns
meta description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates instances of the eELib pv_lib model.

Parameters

- **num** (*int*) – Number of pv_lib models to be created
- **model_type** (*str*) – Description of the created pv_lib instance
- **init_vals** (*list*) – List (length=num) with initial values for each pv_lib model

Returns
return created entities

Return type
dict

get_entity_by_id(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

step(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib pv_lib model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises

- **ValueError** – Error if more than one set value is tried to be given to entity
- **TypeError** – if value_dict containing set values has an unknown format

Returns
next time step (when orchestrator calls again)

Return type
int

get_data(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – Dictionary with data outputs from each pv_lib model

Raises
ValueError – Error if attribute not in model metadata

Returns
Dictionary with simulation outputs

Return type
dict

eelib.core.devices.storage

Submodules

eelib.core.devices.storage.storage_model

eElib battery storage system (BSS) model. The battery and inverter are both modeled together. Among other things , dynamic efficiency is taken into account.

Author: elenia@TUBS

Module Contents

Classes

BSS	An electrical storage system model can be instantiated with default values or with a map of

class BSS(*ename: str*, *soc_init=0.5*, *e_bat_rated=10000*, *p_rated_discharge_max=-8000*, *p_rated_charge_max=8000*, *discharge_efficiency_init=0.95*, *charge_efficiency_init=0.95*, *status_curve=False*, *loss_rate=0.02*, *dod_max=0.95*, *status_aging=False*, *soh_init=1.0*, *soh_cycles_max=0.8*, *bat_cycles_max=5000*, *bat_cycles_init=0*, *step_size=60 * 15*, *bat2ac_efficiency=None*, *ac2bat_efficiency=None*)

An electrical storage system model can be instantiated with default values or with a map of initial values. An energy management model can be used to determine `p_set` values.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

`__set_power_within_limit()`

Sets the active power limit of the battery based on its rated charge and discharge capacities and the current active power setpoint.

`__set_energy_within_limit()`

Check that battery energy limits are not surpassed.

`__calc_charging_efficiency()`

Sets dynamic (dis)charging efficiency levels ("charge_efficiency" / "discharge_efficiency").

`__calc_power_limits()`

Sets (dis)charging power limits.

`step`(*time*)

Performs simulation step of eELib battery model. Calculates all of the Dynamic Properties based on the Input Properties.

Parameters
time (*int*) – Current simulation time

`__calculate_aging_status()`

Calculates the state of health (soh) and then checks if the soh limit has been reached.

soh = soh_init - #cycles * aging_of_one_cycle
aging_of_one_cycle = (soh_init - soh_end) / #max_cycles

Raises
ValueError – When minimum soh has been reached for the battery

`__calc_discharge_efficiency()`

Calculation of dynamic discharging efficiency (`bat2ac_efficiency`).

`__calc_charge_efficiency()`

Calculation of dynamic charging efficiency (`ac2bat_efficiency`).

eelib.core.devices.storage.storage_simulator

Mosaik interface for the eELib storage model. Simulator for communication between orchestrator (mosaik) and storage entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib storage model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: `mosaik_api_v3.Simulator`

Simulator class for eELib storage model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the Storage:Sim class.

Parameters

- **sid** (*str*) – Id of the created instance of the storage simulator (e.g. StorageSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size.
- **time_resolution** (*float*) – Time resolution of current mosaik scenario.

Returns
meta description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates instances of the eELib storage model.

Parameters

- **num** (*int*) – Number of storage models to be created
- **model_type** (*str*) – Description of the created mosaik-storage instance
- **init_vals** (*list*) – List (length=num) with initial values for each storage model

Returns
return created entities

Return type
dict

`get_entity_by_id`(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib storage model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises

- **ValueError** – Error if more than one set value is tried to be given to entity
- **TypeError** – if value_dict containing set values has an unknown format

Returns
next time step (when orchestrator calls again)

Return type
int

`get_data`(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – Dictionary with data outputs from each storage model

Raises
ValueError – Error if attribute not in model metadata

Returns
Dictionary with simulation outputs

Return type
dict

eelib.core.forecast

Submodules

eelib.core.forecast.forecast_model

eELib model for creation of forecast.

Author: elenia@TUBS

Module Contents

Classes

Forecast	Models a forecast creation/calculation.

Attributes

_logger

_logger

class Forecast(*ename: str*, *step_size: int*)

Models a forecast creation/calculation.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time: int*)

Perform simulation step for the forecast model.

Parameters
time (*int*) – Current simulation time

Raises
TypeError – Forecast is requested for an entity that is not added to forecast list

add_forecasted_entity(*entity_dict: dict*)

Adds a entity for the forecast to the corresponding forecast' lists.

Parameters

entity_dict (*dict*) – Input list with all entities of a type (e.g. pv) for which a forecast will be made.

eelib.core.forecast.forecast_simulator

Mosaik interface for the eELib energy management system (EMS) forecast. Simulator for communication between orchestrator (mosaik) and forecast.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

Sim	Simulator class for eELib forecast model.

Attributes

META
ADAPTION_TOLERANCE

META

ADAPTION_TOLERANCE

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib forecast model.

Parameters

mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializer for Prognosis:Sim class.

Parameters

- **sid** (*str*) – ID of the created entity of the simulator (e.g. PrognosisSim-0)

- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – fitting of the step size to the simulation scenario step size.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type='forecast'*, *init_vals=None*)

Creates entities of the eELib forecast model. Core function of mosaik.

Parameters

- **num** (*int*) – number of models to be created
- **model_type** (*str*) – type of created entity, defaults to "forecast"
- **init_vals** (*list*) – list with initial values for each EMS entity, defaults to None

Returns
created entities

Return type
dict

get_entity_by_id(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

step(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib forecast model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict*) – allocation of set values to specific models
- **max_advance** (*int*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises
TypeError – if input has unknown format

Returns
next time step (when orchestrator calls again)

Return type
int

get_data(*outputs: dict*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each prognosed model

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

add_forecasted_entity(*prognosed_entity_id: str*, *entity_dict: dict*)

Adds entities (e.g. pv systems) to the specific forecast unit.

Parameters

- **prognosed_entity_id** (*str*) – Dictionary of forecast entity
- **entity_dict** (*dict*) – Dictionary of created models to be added to the forecast entity

eelib.core.grid

Subpackages

eelib.core.grid.pandapower

Submodules

eelib.core.grid.pandapower.pandapower_model

eElib pandapower grid model (oriented at mosaik_pandapower). Grid import could happen through Json files, and the file should exist in the working directory.

In pandapower library there exist a list of standard grids for direct import and simulation: https://pandapower.readthedocs.io/en/v2.1.0/networks.html

Author: elenia@TUBS

Module Contents

Classes

Pandapower	An electrical grid model in pandapower style.

Attributes

OUTPUT_ATTRS
RENAMING_ATTRS

OUTPUT_ATTRS

RENAMING_ATTRS

class Pandapower(*ename: str*, *gridfile: str*, *sim_start: str*)

Bases: `object`

An electrical grid model in pandapower style.

_VALID_PARAMETERS

LIM

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_load_case(*path*)

Loads a pandapower grid. The grid should be ready in a json file or an example grid.

Parameters
path (*str*) – file path to the grid file

Raises
ValueError – if given grid directory/file cannot be opened

_get_slack()

Create entity of the slack bus.

Returns
zero with index of slack bus

Return type
tuple

_get_buses()

Create entities of the buses.

Returns
indices of busses with their nominal voltage in kV

Return type
list

_get_loads()

Create load entities.

Returns
list of load with tuples about bus index, active/reactive power, scaling factor and their status

Return type
list

_get_lines()

Create branches entities.

Returns
list of grid lines with tuples about bus indices, their length, impedance/reactance/capacitance, maximum current and their status

Return type
list

_get_trafos()

Create trafo entities.

Returns
list of trafos with tuples about bus indices, max. power, nominal voltages, short circuit voltages, iron losses, open-loop current, phase shift, tap status and their status

Return type
list

set_inputs(*etype*, *idx*, *data*, *static*)

Setting the input from the simulator.

Parameters

- **etype** (*string*) – entity type of element to be set
- **idx** (*int*) – index of element in element list
- **data** (*dict*) – input data to be set, key (parameter) and corresponding value
- **static** (*dict*) – static values of this entity

Raises
ValueError – Error if no possible entity type was given

_powerflow()

Conduct power flow for the saved grid with its options from pandapower.

step(*time*)

Performs simulation step of eELib pandapower grid model.

Parameters
time (*int*) – Current simulation time

_store_power_flow_results()

Retrieve the results of the power flow and store them.

_get_powerflow_result_component(*eid*)

Retrieve the results of the power flow for a specific component.

Parameters
eid (*str*) – Entity ID of the component to be searched for

Returns
dict with the results from the power flow for the component

Return type
(dict)

`_get_grid_status()`

Analyse the status of the grid and its component at the current step in time.

`_get_status_bus(comp_idx: int)` → dict

Analyse the status of a grid bus at the current step in time.

Parameters
comp_idx (*int*) – index of the (bus) component in the corresponding grid.bus table

Returns
Status for the bus including limits and congestion status

Return type
dict

`_get_status_line(comp_idx: int, comp_type: str)` → dict

Analyse the status of a grid line at the current step in time.

Parameters

- **comp_idx** (*int*) – index of the (line) component in the grid.line/.trafo table
- **comp_type** (*str*) – whether it is a trafo or a line

Returns
description

Return type
dict

`_calc_ptdf()`

Calculate the DC Power Transfer Distribution Matrix of the Grid. Using function from pandapower.pypower. Only works after the calculation of a powerflow, as the "_ppc" matrix is calculated there. Orientation is as follows: One list for every branch (consequence), one element in each list for every bus (cause). E.g. for a 3-Bus radial grid the matrix will look like this: [[0, -1, -1], [0, 0, -1], [0, 0, 0]].

`_calc_vpif()`

Calculate the AC Voltage Power Impact Factor Matrix of the (current) Grid. The jacobian matrix is calculated in every iteration of pandapower powerflow calculation. The matrix of the last iteration is stored and will be accessed here as a sensitivity of adaptions. Only works after the calculation of a powerflow, as the "_ppc" matrix is calculated there. Direction corresponds to PSC (power consumption decreases voltage magnitude). Orientation is as follows: One list for every bus (consequence, adjustment of vm_pu), one element in each list for every bus times two (cause, first adaption of active power P afterwards adaption of reactive power Q). E.g. for a 2-bus grid the matrix will look like this: [[B1dP->B1dVm, B2dP->B1dVm, B1dQ->B1dVm, B2dQ->B1dVm], … … [B1dP->B2dVm, B2dP->B2dVm, B1dQ->B2dVm, B2dQ->B2dVm]].

eelib.core.grid.pandapower.pandapower_simulator

Mosaik interface for the eELib pandapower grid model. Simulator for communication between orchestrator (mosaik) and pandapower grid entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib pandapower grid model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: `mosaik_api_v3.Simulator`

Simulator class for eELib pandapower grid model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*, *trigger=False*)

Initializes parameters for an object of the Pandapower:Sim class.

Parameters

- **sid** (*str*) – Id of the entity of the pandapower grid simulator (e.g. PandapowerSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size.
- **time_resolution** (*float*) – Time resolution of current mosaik scenario.
- **trigger** (*boolean*) – Whether power flow is triggered by events

Returns
meta description of the simulator

Return type
meta

`create`(*num*, *model_type*, *gridfile*, *sim_start=None*)

Creates instances of the eELib pandapower model.

Parameters

- **num** (*int*) – Number of pandapower models to be created
- **model_type** (*str*) – Description of the created eELib-pandapower instance
- **gridfile** (*str*) – directory of the file with the grid data
- **sim_start** (*str*) – date of the simulation start

Returns
return created entities

Return type
dict

`setup_done`()

Adjust entity IDs and activate components at the end of the grid setup.

Yields
related_components – grid components that are in relation to one another (e.g. bus and connected line)

`get_entity_by_id`(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib pandapower grid model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Returns
next time step (when orchestrator calls again)

Return type
int

`get_data`(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – Dictionary with data outputs from each storage model

Raises

- **ValueError** – Error if attribute not in model metadata
- **KeyError** – Error if component entity could not be found in pandapower grid

Returns
Dictionary with simulation outputs

Return type
dict

eelib.core.market

Subpackages

eelib.core.market.retail_electricity_provider

Submodules

eelib.core.market.retail_electricity_provider.rep_model

eELib retail electricity provider models the behavior of provider for electricity. This basically is reduced to a provider for households and maybe small business companies.

Author: elenia@TUBS

Module Contents

Classes

RetailElectricityProvider	Models a retail electricity provider with tariff signals and market information.

class RetailElectricityProvider(*ename: str*, *step_size: int = 60 * 15*, *n_steps: int = 24 * 4*, *tariff_type: str = 'static'*, *elec_price: float = 0.35*, *feedin_tariff: float = 0.07*)

Models a retail electricity provider with tariff signals and market information.

market_info_attrs = ['price_weighted_avg', 'price_low', 'price_high', 'price_last']

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

`add_market_entity`(*market_dict: dict*)

Adds a market entity to the corresponding list.

Parameters
market_dict (*dict*) – Input list with all market models

`step`(*time*)

Performs simulation step of eELib REP model.

Parameters
time (*int*) – Current simulation time

`_calc_tariff_output`()

Combine values and market info for a tariff signal for this time step.

Raises

- **ValueError** – If variable tariffs are chosen (not yet implemented!)
- **ValueError** – If dynamic tariffs are chosen (not yet implemented!)

`__request_forecast`()

Set markets with the needed attributes and time horizon for forecast request.

eelib.core.market.retail_electricity_provider.rep_simulator

Mosaik interface for the eELib retail electricity provider model. Simulator for communication between orchestrator (mosaik) and REP entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

Sim	Simulator class for eELib retail electricity provider model.

Attributes

ADAPTION_TOLERANCE
META

ADAPTION_TOLERANCE

META

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib retail electricity provider model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Raises
ValueError – Unknown output attribute, when not described in META of simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the REP:Sim class.

Parameters

- **sid** (*str*) – Id of the created instance of the load simulator (e.g. REPSim-0).
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size.
- **time_resolution** (*float*) – fitting of the step size to the simulation scenario step size.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates instances of the eELib retail electricity provider model.

Parameters

- **num** (*int*) – Number of REP models to be created
- **model_type** (*str*) – Description of the created instance (here "RetailElectricityProvider")
- **init_vals** (*list*) – List (length=num) with initial values for each REP model

Returns
created entities

Return type
dict

get_entity_by_id(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib retail electricity provider model.

Parameters

- **time** (*int*) – Current simulation time according to step size. Given by MOSAIK.
- **inputs** (*dict*) – Is a dictionary, that maps entity IDs to data dictionaries which map attribute names to lists of values. Given by MOSAIK.
- **max_advance** (*int, optional*) – Is the simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Raises
TypeError – value_dict has unknown format

Returns
New time stamp (time increased by step size)

Return type
int

`get_data`(*outputs*)

Gets the data for the next concatenated model. Core function of mosaik.

Parameters
outputs (*dict*) – Dictionary with data outputs from each REP model

Raises
ValueError – Error if attribute not in model metadata

Returns
Dictionary with simulation outputs

Return type
dict

`add_market_entity`(*market_entity_eid: str*, *market_dict: dict*)

Adds market entities (e.g. market csv reader) to the specific REP entity.

Parameters

- **market_entity_eid** (*str*) – entity id of REP entity
- **market_dict** (*dict*) – Dictionary of market models to be added to the REP entity

eelib.data

This contains all models that are "just" retrieving input data (like a simple csv-reader). It also includes the functionalities for simulation data to be collected and assessed and dataclasses for structured exchange of data between the models.

Subpackages

eelib.data.csv_reader

Submodules

eelib.data.csv_reader.csv_reader_model

eElib csv-reader model for reading-in .csv-files to imitate different model types (e.g. PV, household or heatpump).

Author: elenia@TUBS

Module Contents

Classes

CSVReader	Parent class for reading-in .csv-files.
GenericCSV	CSV-Data-Reader for profiles. Inherits from class csv_reader.
RatedCSV	CSV-Data-Reader for profiles. Inherits from class csv_reader.
PowerCSV	CSV-Data-Reader for power profiles. Inherits from class csv_reader.
HouseholdCSV	CSV-Data-Reader for household load profiles. Inherits from class csv_reader.
PvCSV	CSV-Data-Reader for pv-generation profiles. Inherits from class CSVReader.
HeatpumpCSV	CSV-Data-Reader for heatpump load profiles.
ChargingStationCSV	CSV-Data-Reader for charging_station profiles. Inherits from class CSVReader.
HouseholdThermalCSV	CSV-Data-Reader for thermal demand profiles. Inherits from class RatedCSV.
MarketIntradayContinuousCSV	CSV-Data-Reader for intraday continuous market data. Inherits from class CSVReader.

class CSVReader(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*)

Parent class for reading-in .csv-files.

Raises

- **FileNotFoundError** – when .csv-file cannot be opened (e.g. path or data corrupted)
- **ValueError** – when start date is not in .csv-file
- **IndexError** – when end of .csv-file is reached before simulation end

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_open_csv() → None

Opens the .csv-file and loads data set.

Raises
FileNotFoundError – when .csv-file cannot be read

_check_data() → None

Checks if row in opened .csv-file contains the simulation start date.

Raises
ValueError – Start date is not in .csv-file

_resample_data() → None

step(*time*) → None

Performs simulation step of eELib csv_reader model, which is returning the averaged read values in the .csv-file.

Parameters
time (*int*) – Current simulation time in seconds.

class GenericCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*)

Bases: *CSVReader*

CSV-Data-Reader for profiles. Inherits from class csv_reader. Sets each column of the csv file as attribute.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*) → None

Performs simulation step of generic csv reader model, which is setting the attr according to the columns in csv file.

Parameters
time (*int*) – Current simulation time in seconds.

class RatedCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*)

Bases: *CSVReader*

CSV-Data-Reader for profiles. Inherits from class csv_reader. Adds support for scaling to p_rated.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_apply_p_rated() → None

Apply scaling factor (p_rated/p_rated_profile).

class PowerCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1.0*)

Bases: *RatedCSV*

CSV-Data-Reader for power profiles. Inherits from class csv_reader.

Adds support for calculating reactive power.
Assumptions:
- first column is p
- second column is q, if csv has more than one column

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

_set_reactive_power() → None

Calculate reactive power if not given in csv file and if cos_phi is set.

step(*time*)

Performs simulation step of eELib load model, which is returning the read active/reactive power value of the .csv-file.

Parameters
time (*int*) – Current simulation time in seconds.

class HouseholdCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1.0*)

Bases: *PowerCSV*

CSV-Data-Reader for household load profiles. Inherits from class csv_reader.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

class PvCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1.0*)

Bases: *PowerCSV*

CSV-Data-Reader for pv-generation profiles. Inherits from class CSVReader.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*)

Performs simulation step of eELib load model, which is returning the read active/reactive power value of the .csv-file.

Parameters
time (*int*) – Current simulation time in seconds.

class HeatpumpCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 4000*, *p_rated_profile: int = 4000*, *cos_phi: float = 0.95*)

Bases: *PowerCSV*

CSV-Data-Reader for heatpump load profiles.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*)

Performs simulation step of eELib load model, which is returning the read active/reactive power value of the .csv-file.

Parameters
time (*int*) – Current simulation time in seconds.

class ChargingStationCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1.0*)

Bases: *PowerCSV*

CSV-Data-Reader for charging_station profiles. Inherits from class CSVReader.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

class HouseholdThermalCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*, *p_rated: int = 20000*, *p_rated_profile: int = 20000*)

Bases: *RatedCSV*

CSV-Data-Reader for thermal demand profiles. Inherits from class RatedCSV.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*)

Performs simulation step of eELib charging_station model, which is returning the read active power value of the .csv-file.

Parameters
time (*int*) – Current simulation time in seconds.

class MarketIntradayContinuousCSV(*ename: str*, *datafile: str*, *header_rows: int*, *start_time: str*, *date_format: str = 'YYYY-MM-DD HH:mm:ss'*, *delimiter: str = ','*, *step_size: int = 60 * 15*)

Bases: *CSVReader*

CSV-Data-Reader for intraday continuous market data. Inherits from class CSVReader.

Implementation as a price-taking market: The participants have no influence or impact on the market price and participants have to accept the prevailing market price. As decentralized power systems are investigated, this assumption is fine as the whole intraday market includes many sellers (with the identical product 'electricity'). See e.g. https://www.economicsonline.co.uk/definitions/price-taker.html/ for more information.

The prices are given in the unit 'EUR/MWh'.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

step(*time*)

Performs simulation step of model, which is returning the read value of the .csv-file.

Parameters
time (*int*) – Current simulation time in seconds.

eelib.data.csv_reader.csv_reader_simulator

Mosaik interface for the eELib csv-reader model. Simulator for communication between orchestrator (mosaik) and csv-reader entities.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib csv-reader model.

Attributes

META

META

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib csv-reader model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Raises
ValueError – Unknown output attribute, when not described in META of simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializes parameters for an object of the CSV-Reader:Sim class.

Parameters

- **sid** (*str*) – ID of the created entity of the simulator (e.g. LoadSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – Time resolution of current scenario.

Returns
description of the simulator

Return type
meta

`create`(*num*, *model_type*, *init_vals*)

Creates entities of the eELib csv-reader model. Core function of mosaik.

Parameters

- **num** (*int*) – number of load models to be created
- **model_type** (*str*) – type of created instance (e.g. "household")
- **init_vals** (*list*) – list with initial values for each csv-reader entity

Returns
created entities

Return type
dict

`get_entity_by_id`(*entity_id: str*)

Searches for a requested entity id and gives back the entity model.

Parameters
entity_id (*str*) – id of the entity to be searched for

Returns
entity model if found, None otherwise

Return type
object

`step`(*time*, *inputs*, *max_advance*)

Performs simulation step calling the eELib csv-reader model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models (NOT NEEDED FOR CSV-READER MODEL)
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Returns
next time step (when orchestrator calls again)

Return type
int

`get_data`(*outputs*)

Gets the data for the next connected model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each entity

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

eelib.data.database

Submodules

eelib.data.database.hdf5

Interface for the event-based HDF5 database.

Within this interface mosaik functionalities are used. Copyright (c) LGPL

Author: elenia@TUBS

Module Contents

Classes

Hdf5Database	HDF5 Database class

Attributes

meta

meta

class Hdf5Database

Bases: `mosaik_api.Simulator`

HDF5 Database class Inherits mosaik simulator class for management.

init(*sid*, *time_resolution*, *scenario_config*, *grid_model_config=None*, *additional_groups: dict = None*, *series_path=(None, None)*)

Initialize HDF5 Database Entity.

Parameters

- **sid** (*str*) – string id of the entity to be created
- **time_resolution** (*float*) – timely resolution of the simulation scenario
- **scenario_config** (*dict*) – configurations of the simulation
- **grid_model_config** (*dict*) – configuration of order of the series in the resulting database. If not given a default structure is used.
- **additional_groups** (*dict*) – additional groupings not defined in the model config If not given all unmatched components are stored in a `misc` grouping. Nested dictionaries and arrays as values are permissible. Assignment of a component to one of these groups is only

possible if it is not already assigned to another group provided in grid_model_config. In that case, assignment is determined by component eid string including one of the lowest level strings (group names) provided here OR matching the regular expression, if the lowest level strings are regexes. Additionally, names defined first have priority. An example of this is as follows: {"E1": ["S1", "S2"], "E2": "S3", "E3": {"T1":"S8", "T2":[r'^ABC-(.*)', "S9"]}}

- **series_path** (*tuple*) – path for the series to be stored. Defaults to (None, None)

Returns
meta description of the simulator

Return type
meta

create(*num*, *model*, *filename*, *buf_size=1000*, *dataset_opts=None*)

Creates instances of the HDF5 output database.

Parameters

- **num** (*int*) – Number of hdf5 db models to be created
- **model** (*str*) – Description of the created hdf5 db entity
- **filename** (*str*) – Directory of the hdf5 file
- **buf_size** (*int*) – Number of elements to be stored before writing into DB. Defaults to 1000
- **dataset_opts** (*dict*) – Potential set of options for the dataset. Defaults to None.

Raises

- **ValueError** – Error if more than one hdf5 DB is to be created
- **ValueError** – Error if an unknown model type (not Database) is given

Returns
discription of the hdf5 DB entity with ID, type and relations

Return type
dict

setup_done()

Check if setup of the DB is done.

Yields
list – related entity for the simulator (given by mosaik)

step(*time*, *inputs*, *max_advance*)

Performs simulation step saving/buffering data and storing it into the DB. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict*) – allocation of return values to specific models
- **max_advance** (*int*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Returns
next time step (when orchestrator calls again)

Return type
int

`set_meta_data`(*data*)

Set the meta data of the DB.

Parameters
data (*dict*) – data elements to be stored (attribute with corresponding value)

`set_static_data`(*data*)

Store static (un-changeable) data into the hdf5 DB.

Parameters
data (*dict*) – data elements to be stored (entity id with dict of attributes and values)

`_store_relations`()

Query relations graph and store it in the database.

Yields
list – related entities of simulator (given by mosaik)

`_create_dataset`(*src_id*, *attr*, *dtype*, *ds_size*, *buf*, *buf_size*)

Create a dataset for the attribute `attr` of entity `src_id`. The dataset will use the type `dtype` and will have the size `ds_size`. Also initialize the buffer `buf` with size `buf_size`.

Parameters

- **src_id** (*str*) – entity for the dataset to be stored in hdf5 file
- **attr** (*str*) – attribute to be saved
- **dtype** (*type*) – Type of the attribute
- **ds_size** (*int*) – number of data elements to be stored
- **buf** (*dict*) – storing of buffered elements within a list
- **buf_size** (*int*) – number of elements to be buffered before storing

`_get_group_generic`(*eid*, *supergroup*)

Generic method for creating and/or finding a group within a supergroup.

Parameters

- **eid** (*str*) – entity id corresponding to the group to be found/created
- **supergroup** (*Group*) – the supergroup where the search/creation occurs

Returns
the resulting group

Return type
Group

`_get_group_default`(*eid*)

Get or create group for entity `eid` based on the default structure.

Parameters
eid (*str*) – entity identification string

Raises
ValueError – if the given eid format does not match the predefined pattern

Returns
hdf5 file group element for data to be stored into

Return type
Group

`_create_groups_preset`(*supergroup*, *config*, *dictionary=None*)

Recursive method to create groups based on a preset dictionary.

Parameters

- **supergroup** (*Group*) – supergroup where creation occurs. Initial input is where all the series are stored.
- **config** (*dict*) – the configuration dictionary.
- **dictionary** (*dict*) – if provided, the given dictionary is updated with pairs of group names and Group objects for the groups at the lowest level.

`_find_group_preset`(*eid*, *supergroup*)

Recursive method to find a group within a supergroup.

Parameters

- **eid** (*str*) – entity id corresponding to the group to be found
- **supergroup** (*Group*) – the supergroup where the search occurs

Raises
ValueError – if the given eid format does not match the predefined pattern

Returns
the resulting group OR `None` if it doesn't exist

Return type
Group

`_get_group_additional`(*eid*)

Get or create group for entity `eid` in a group among those added in addition to model config, if applicable.

Parameters
eid (*str*) – entity id corresponding to the group to be found

Returns
the first possible group OR `None` if it isn't applicable

Return type
Group

`_store_dict`(*group*, *dictionary*, *dataset_name*)

Store a dictionary as a dataset in a given group.

Parameters

- **dictionary** (*dict*) – dictionary to be stored
- **group** (*Group*) – the group to store the data in
- **dataset_name** (*str*) – the name of the dataset

`_store_json`(*group*, *data*, *dataset_name*)

Store json-like data as a dataset in a given group.

Parameters

- **data** (*Any*) – data to be stored
- **group** (*Group*) – the group to store the data in

- **dataset_name** (*str*) – the name of the dataset

_get_entity_path(*eid*)

Get the database path to a searched entity by its ID.

Parameters
eid (*str*) – identification number of the entity to be searched

Returns
path to the entity

Return type
str

_save_attrs(*g*, *attrs*)

Write an attribute with its value into the database.

Parameters

- **g** (*db element*) – _description_
- **attrs** (*dict*) – data elements to be stored (attribute with corresponding value)

eelib.data.dataclass

Submodules

eelib.data.dataclass._base

Base Class for Dataclasses used in eELib.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

BaseData	Baseclass for Dataclasses.

class BaseData

Bases: abc.ABC

Baseclass for Dataclasses.

check_adaption_tolerance(*adaption_tolerance: float*, *dict_cache: dict*, *dict_to_check: dict = None*) → bool

Compares the input data with its cache based on the individual values.

Parameters

- **adaption_tolerance** (*float*) – fixed adaption tolerance for comparison of values

- **dict_cache** (*dict*) – cached dataclass to be compared
- **dict_to_check** (*dict*) – Subdict to be checked, substitutes dataclass if existent. Defaults to None.

Raises
TypeError – if format for dataclass value is unknown

Returns
True if value of new dataclass has changed

Return type
bool

eelib.data.dataclass.control

Dataclasses for control signals used in eELib.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

ControlSignalEMS	Dataclass for Control Signals sent to an Energy Management System.

class ControlSignalEMS

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Control Signals sent to an Energy Management System. The lists always have to be of the same length!

steps (list): timesteps where control signal is active p_max (list): maximum active power for those time steps [kW] p_min (list): minimum active power for those time steps [kW] penalty_cost (float): penalty costs in case of non-compliancy [EUR/kW]

steps: list[int]

p_max: list[float]

p_min: list[float]

penalty_cost: float

eelib.data.dataclass.devices

Dataclasses for devices' data used in eELib.

Author: elenia@TUBS

Module Contents

Classes

BaseDeviceData	Baseclass for Dataclasses of the devices.
BSSData	Dataclass for eELib BSS model.
EVData	Dataclass for eELib EV model.
CSData	Dataclass for eElib charging station model.
HPData	Dataclass for eElib heatpump model.
PVData	Dataclass for eElib pv(lib) model.

class BaseDeviceData

Bases: *eelib.data.dataclass._base.BaseData*

Baseclass for Dataclasses of the devices.

p (float): current active power p_min (float): minimum active power limit p_max (float): maximum active power limit

p: float

p_min: float

p_max: float

class BSSData

Bases: *BaseDeviceData*

Dataclass for eELib BSS model.

e_bat_rated (float): rated energy amount of bss p_rated_charge_max (float): maximum charge power of bss p_rated_discharge_max (float): maximum discharge power of bss loss_rate (float): self-discharge (per month) of bss self_discharge_step (float): self-discharge rate (per simulation step) of bss dod_max (float): maximum depth of discharge status_aging (float): whether to account for aging of battery soh_cycles_max (float): state of health at end of lifetime bat_cycles_max (float): maximum cycles to reach end of lifetime soc_min (float): lower soc level of bss bat2ac_efficiency (list[float]): exponential discharging efficiency values ac2bat_efficiency (list[float]): exponential charging efficiency values

charge_efficiency (float): charging efficiency discharge_efficiency (float): discharging efficiency soh (float): state of health bat_cycles (int): number of cycles accumulated by the battery e_bat (float): current energy amount stored e_bat_usable (float): usable energy amount of bss considering ageing

e_bat_rated: int

p_rated_discharge_max: int

p_rated_charge_max: int

loss_rate: float

self_discharge_step: float

dod_max: float

status_aging: bool

soh_cycles_max: float

bat_cycles_max: int

soc_min: float

bat2ac_efficiency: list[float] | None

ac2bat_efficiency: list[float] | None

charge_efficiency: float

discharge_efficiency: float

soh: float

bat_cycles: int

e_bat: float

e_bat_usable: float

class EVData

Bases: *BaseDeviceData*

Dataclass for eELib EV model.

soc_min (float): minimum soc level of car e_max (float): rated energy level of car battery p_nom_charge_max (float): maximum charging power of car p_nom_discharge_max (float): maximum discharging power of car charge_efficiency (float): charge efficiency of car dcharge_efficiency (float): discharge efficiency of car

e_bat (float): energy amount of the car appearance (bool): where car is available at charging point or not

soc_min: float

e_max: float

p_nom_discharge_max: float

p_nom_charge_max: float

dcharge_efficiency: float

charge_efficiency: float

e_bat: float

appearance: bool

class CSData

Bases: *BaseDeviceData*

Dataclass for eElib charging station model.

charge_efficiency (float): charge efficiency of charging station discharge_efficiency (float): discharge efficiency of charging station p_rated (float): rated power of charging station

ev_data (dict): information about all the connected electric vehicles q (float): current reactive power

discharge_efficiency: float

charge_efficiency: float

p_rated: float

ev_data: dict[str:EVData]

q: float

class HPData

Bases: *BaseDeviceData*

Dataclass for eElib heatpump model.

p_rated (float): rated thermal power of heatpumps p_min_th_rel (float): relative minimum power (of rated power) for modulating heatpumps time_min (int): minimum on/off-time for heatpumps

time_on (int): current on-time for heatpumps time_off (int): current off-time for heatpumps state (str): current state of heatpumps (on, off, must-on or must-off) cop (float): current coefficient of power for heatpumps p_th (float): current thermal power p_th_min (float): minimum thermal power limit p_th_min_on (float): minimum thermal power limit in on-state p_min_on (float): minimum electrical power limit in on-state p_th_max (float): maximum thermal power limit q (float): current reactive power

p_rated_th: float

p_min_th_rel: float

time_min: int

time_on: int

time_off: int

state: str

cop: float

p_th: float

p_th_min: float

p_th_min_on: float

p_min_on: float

p_th_max: float

q: float

class PVData

Bases: *BaseDeviceData*

Dataclass for eElib pv(lib) model.

p_rated (float): rated power of system

q (float): current reactive power

p_rated: float

q: float

eelib.data.dataclass.options

Dataclasses for options used in eELib.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Classes

OptimOptions	Dataclass for Options of an Energy Management System Optimization.

class OptimOptions

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Options of an Energy Management System Optimization.

consider_thermal (bool): whether to consider the thermal balance for the optimization thermal_energy_start (float): thermal energy demand before first schedule step thermal_energy_restriction (bool): restrict the thermal energy within this energy range [kWh_th] th_e_penalty (float): penalty for th. energy demand surpassing the restricting limits

ev_direct_cha (bool): incent. dir. charging with max. power with min. desc. obj. func. value bss_direct_cha (bool): incent. dir. charging using a min. descending obj. function value dir_cha_desc_factor (float): descending obj. func. factor for charging power (bss, ev)

ev_end_energy_penalty (float): penalty factor if ev not entirely filled at the last step bss_end_energy_incentive (bool): incent. stored bss energy at last step (avoid disch. into grid)

round_int (int): number of integers to round the optimization results to

consider_thermal: bool

thermal_energy_start: float

thermal_energy_restriction: float

th_e_penalty: float

ev_direct_cha: bool

bss_direct_cha: bool

dir_cha_desc_factor: float

ev_end_energy_penalty: float

bss_end_energy_incentive: bool

round_int: int

eelib.data.dataclass.tariff

Dataclasses for tariffs and market data used in eELib.

Author: elenia@TUBS

Module Contents

Classes

MarketData	Dataclass for Market Data.
Tariff	Dataclass for Prosumer Tariff.
TariffSignal	Dataclass for Tariff that is sent as a Signal to a Household.
GridTariff	Dataclass for Grid Usage Fees.
GridTariffSignal	Dataclass for Signal regarding Grid Usage Fees.

class MarketData

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Market Data.

price_weighted_avg (float): weighted average price price_low (float): lowest market price price_high (float): highest market price price_last (float): last market price

price_weighted_avg: float

price_low: float

price_high: float

price_last: float

class Tariff

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Prosumer Tariff.

elec_price (float): electricity consumption price info [EUR/kWh]. Defaults to 35ct/kWh feedin_tariff (float): electricity feed-in price info [EUR/kWh]. Defaults to 7ct/kWh capacity_fee_dem (float): Price for max. power demand at connection [EUR/kW]. Defaults to 0 capacity_fee_gen (float): Price for max. power generation at connection [EUR/kW]. Defaults to 0 capacity_fee_horizon_sec (int): Horizon for capacity fee [seconds]. Defaults to 86400 (1 day)

elec_price: float

feedin_tariff: float

capacity_fee_dem: float

capacity_fee_gen: float

capacity_fee_horizon_sec: int

class TariffSignal

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Tariff that is sent as a Signal to a Household.

bool_is_list (bool): Whether signal is for multi-step tariff. Defaults to False elec_price (float, list): electricity consumption price info [EUR/kWh]. Defaults to 0.35 feedin_tariff (float, list): electricity feed-in price info [EUR/kWh]. Defaults to 0.07 steps (list): timesteps for the corresponding list of prices. Defaults to empty list capacity_fee_dem (float): Price for max. power demand at connection [EUR/kW]. Defaults to 0 capacity_fee_gen (float): Price for max. power generation at connection [EUR/kW]. Defaults to 0 capacity_fee_horizon_sec (int): Horizon for capacity fee [seconds]. Defaults to 86400 (1 day)

bool_is_list: bool

elec_price: float | list

feedin_tariff: float | list

steps: list

capacity_fee_dem: float

capacity_fee_gen: float

capacity_fee_horizon_sec: int

class GridTariff

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Grid Usage Fees.

grid_tariff_model (str): which control model is selected for households. Defaults to “flat-rate” energy_price (float): Energy-dependent price for electr. consump. [EUR/kWh]. Defaults to 8ct/kWh capacity_fee_dem (float): Price for max. power demand at connection [EUR/kW]. Defaults to 0 capacity_fee_gen (float): Price for max. power generation at connection [EUR/kW]. Defaults to 0 capacity_fee_horizon_sec (int): Horizon for capacity fee [seconds]. Defaults to 86400 (1 day)

grid_tariff_model: str

energy_price: float

capacity_fee_dem: float

capacity_fee_gen: float

capacity_fee_horizon_sec: int

class GridTariffSignal

Bases: *eelib.data.dataclass._base.BaseData*

Dataclass for Signal regarding Grid Usage Fees.

bool_is_list (bool): Whether signal is for multi-step tariff. Defaults to False energy_price (float, list): Energy-dependent price for el. cons. [EUR/kWh]. Defaults to 8ct/kWh steps (list): timesteps for the corresponding list of prices. Defaults to empty list capacity_fee_dem (float): Price for max. power demand at connection [EUR/kW]. Defaults to 0 capacity_fee_gen (float): Price for max. power generation at connection [EUR/kW]. Defaults to 0 capacity_fee_horizon_sec (int): Horizon for capacity fee [seconds]. Defaults to 86400 (1 day)

bool_is_list: bool

energy_price: float | list

steps: list

capacity_fee_dem: float

capacity_fee_gen: float

capacity_fee_horizon_sec: int

eelib.data.influx_reader

Submodules

eelib.data.influx_reader.influx_reader_model

eElib Influx-reader models for reading-in from Influx database to imitate different model types. Specializations for certain model types like Household series.

Author: elenia@TUBS

Module Contents

Classes

GenericInflux	Class for reading data from an influxdb.
RatedInflux	Influx-Data-Reader for profiles. Inherits from class GenericInflux.
PowerInflux	Influx-Data-Reader for power profiles. Inherits from class RatedInflux.
HouseholdInflux	Influx-Data-Reader for household load profiles. Inherits from class PowerInflux.
PvInflux	Influx-Data-Reader for pv load profiles. Inherits from class PowerInflux.

Attributes

_logger

_logger

class GenericInflux(*ename: str*, *step_size: int*, *measurement_name: str*, *tags: dict[str, str]*, *fields: list[str]*, *start_time: str*, *end_time: str*, *influx_url: str | None = None*, *influx_token: str | None = None*, *influx_org: str | None = None*, *influx_bucket: str | None = None*)

Class for reading data from an influxdb.

_VALID_PARAMETERS

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

__del__()

Close the client after the reader's job is done.

static generate_tag_filter(*tags: dict[str, str] = None*) → str

Generates tag filters for Influx query.

Parameters
tags (*dict[str, str], optional*) – Tags to generate filter.

Returns
Tag selection part of Influx query.

Return type
str

generate_influx_query(*measurement_name: str*, *start_time: str*, *end_time: str*, *resolution: int*, *tags: dict[str, str] = None*, *fields: list[str] = None*) → str

Generates influx query based on given parameters.

Parameters

- **measurement_name** (*str*) – Name of the series.
- **start_time** (*str*) – Start time of the series slice.
- **end_time** (*str*) – End time of the series slice.
- **resolution** (*int*) – Time resolution used for aggregation.
- **tags** (*dict[str, str], optional*) – Tags wirth keys and values. Defaults to None.
- **fields** (*list[str], optional*) – Fields of the series. Defaults to None.

Returns
Influx query

Return type
str

_read_db(*measurement_name: str*, *tags: dict[str, str]*, *fields: list[str]*, *start_time: str*, *end_time: str*, *resolution: int*)

Filters data in the database based on given parameters and returns the result as a dataframe.

Parameters

- **measurement_name** (*str*) – Name to filter _measurement
- **tags** (*dict[str, str]*) – Tags to filter, e.g. "type": "load"
- **fields** (*list[str]*) – Fields to select
- **start_time** (*str*) – Range start time
- **end_time** (*str*) – Range end time
- **resolution** (*int*) – time resolution to aggregate

Returns
query result with DateTime index and one column for each field.

Return type
pd.DataFrame

step(*time*)

Performs simulation step of eELib influx_reader model.

Parameters
time (*int*) – Current simulation time

Raises
IndexError – when end of data is reached before simulation end

class RatedInflux(*ename: str*, *step_size: int*, *measurement_name: str*, *tags: dict[str, str]*, *fields: list[str]*, *start_time: str*, *end_time: str*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *influx_url: str | None = None*, *influx_token: str | None = None*, *influx_org: str | None = None*, *influx_bucket: str | None = None*)

Bases: *GenericInflux*

Influx-Data-Reader for profiles. Inherits from class GenericInflux. Adds support for scaling to p_rated.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

`_apply_p_rated`() → None

Apply scaling factor (`p_rated`/`p_rated_profile`).

`class PowerInflux`(*ename: str*, *step_size: int*, *measurement_name: str*, *tags: dict[str, str]*, *fields: list[str]*, *start_time: str*, *end_time: str*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1.0*, *influx_url: str | None = None*, *influx_token: str | None = None*, *influx_org: str | None = None*, *influx_bucket: str | None = None*)

Bases: *RatedInflux*

Influx-Data-Reader for power profiles. Inherits from class RatedInflux.

Adds support for calculating reactive power. Assumptions: - first field is p - second field is q, if more than one field is given

`classmethod get_valid_parameters`()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

`_set_reactive_power`() → None

Calculate reactive power if not given as field of that series in influx and if cos_phi is set.

`step`(*time*)

Performs simulation step of eELib load model, which is returning the read active/reactive power value of the series read from Influx database.

Parameters
time (*int*) – Current simulation time in seconds.

`class HouseholdInflux`(*ename: str*, *step_size: int*, *measurement_name: str*, *fields: list[str]*, *start_time: str*, *end_time: str*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1*, *influx_url: str | None = None*, *influx_token: str | None = None*, *influx_org: str | None = None*, *influx_bucket: str | None = None*)

Bases: *PowerInflux*

Influx-Data-Reader for household load profiles. Inherits from class PowerInflux. Data in Influx Database must have a tag with key value pair {'type': 'load'}.

`classmethod get_valid_parameters`()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

`class PvInflux`(*ename: str*, *step_size: int*, *measurement_name: str*, *fields: list[str]*, *start_time: str*, *end_time: str*, *p_rated: int = 4500*, *p_rated_profile: int = 4500*, *cos_phi: float = 1*, *influx_url: str | None = None*, *influx_token: str | None = None*, *influx_org: str | None = None*, *influx_bucket: str | None = None*)

Bases: *PowerInflux*

Influx-Data-Reader for pv load profiles. Inherits from class PowerInflux. Data in Influx Database must have a tag with key value pair {'type': 'pv'}.

classmethod get_valid_parameters()

Returns dictionary containing valid parameter types and values.

Returns
valid parameters for this model

Return type
dict

eelib.data.influx_reader.influx_reader_simulator

Mosaik interface for the eELib influx-reader model. Simulator for communication between orchestrator (mosaik) and influx-reader entities.

Author: elenia@TUBS

Module Contents

Classes

Sim	Simulator class for eELib influx-reader model.

Functions

set_meta_attrs(init_vals)	Method to dynamically set the attributes of the simulator.

Attributes

META

META

set_meta_attrs(*init_vals*)

Method to dynamically set the attributes of the simulator. Should be called before the simulation begins.

Parameters
init_vals (*dict*) – initial variables and their values.

class Sim

Bases: mosaik_api_v3.Simulator

Simulator class for eELib influx-reader model.

Parameters
mosaik_api_v3 (*module*) – defines communication between mosaik and simulator

Raises
ValueError – Unknown output attribute, when not described in META of simulator

init(*sid*, *scenario_config*, *time_resolution=1.0*)

Initializer for influx-Reader:Sim class.

Parameters

- **sid** (*str*) – ID of the created entity of the simulator (e.g. LoadSim-0)
- **scenario_config** (*dict*) – scenario configuration data, like resolution or step size
- **time_resolution** (*float*) – Time resolution of current scenario.

Returns
description of the simulator

Return type
meta

create(*num*, *model_type*, *init_vals*)

Creates entities of the eELib influx-reader model. Core function of mosaik.

Parameters

- **num** (*int*) – number of load models to be created
- **model_type** (*str*) – type of created instance (e.g. "household")
- **init_vals** (*list*) – list with initial values for each influx-reader entity

Returns
created entities

Return type
dict

step(*time*, *inputs*, *max_advance=1*)

Performs simulation step calling the eELib influx-reader model. Core function of mosaik.

Parameters

- **time** (*int*) – current simulation time (given by mosaik)
- **inputs** (*dict, optional*) – allocation of return values to specific models
- **max_advance** (*int, optional*) – simulation time until the simulator can safely advance it's internal time without causing any causality errors.

Returns
next time step (when orchestrator calls again)

Return type
int

get_data(*outputs*)

Gets the data for the next connected model. Core function of mosaik.

Parameters
outputs (*dict*) – dictionary with data outputs from each entity

Raises
ValueError – error if attribute not in model metadata

Returns
dictionary with simulation outputs

Return type
dict

eelib.data.input

Submodules

eelib.data.input.extract

Helper module to extract data from different sources.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Functions

extract_htw(→ dict[str, pandas.DataFrame])	Extract data from public htw Matlab files.

extract_htw(*mat_file: str*, *idx: int | range | Sequence | None = None*) → dict[str, pandas.DataFrame]

Extract data from public htw Matlab files.

Parameters

- **mat_file** (*str*) – String to path of HTW Matlab file.
- **idx** (*Optional[int | range | Sequence], optional*) – Idx to extract, defaults to None.

Returns
Dict of DataFrames.

Return type
dict[str, pd.DataFrame]

eelib.data.input.load

Helper module. Influx Db Connector to load data to Influx Database.

Author: elenia@TUBS

Module Contents

Classes

InfluxDbStorage	Influx Db Connector class to write pandas dataframes.

Attributes

data_path

class InfluxDbStorage(*influx_url: str | None = None*, *influx_token: str | None = None*, *influx_org: str | None = None*, *influx_bucket: str | None = None*)

Influx Db Connector class to write pandas dataframes.

__del__()

Delete client connection on delete.

write_pandas_dataframe(*df: pandas.DataFrame*, *name: str*, *tags: dict[str, str]*) → None

Write pandas dataframe to influx database.

Parameters

- **df** (*pd.DataFrame*) – Df to write to Influx.
- **name** (*str*) – Name of the series.
- **tags** (*dict[str, str]*) – Tags to add when writing to Influx.

data_path = '.'

Submodules

eelib.data.collector

Simple collector simulator based on the one defined within the mosaik tutorial: https://mosaik.readthedocs.io/en/latest/
Prints all inputs immediately.

@author: elenia@TUBS

Module Contents

Classes

Collector	Collector simualtor to provide simple prints in each step.

Attributes

META

META

class Collector

Bases: mosaik_api_v3.Simulator

Collector simualtor to provide simple prints in each step.

init(*sid*, *time_resolution*)

Init of a simulator instance.

Parameters

- **sid** (*_type_*) – _description_
- **time_resolution** (*_type_*) – _description_

Returns
description

Return type
type

create(*num*, *model*)

Create mosaik sim of the collector.

Parameters

- **num** (*_type_*) – _description_
- **model** (*_type_*) – _description_

Raises
RuntimeError – If more than one simulators should be initialized.

Returns
eid and simulator

Return type
dict

step(*time*, *inputs*, *max_advance*)

Step method of the collector simulator. Prints all inputs.

Parameters

- **time** (*_type_*) – _description_
- **inputs** (*_type_*) – _description_
- **max_advance** (*_type_*) – _description_

Returns
max advance until next call of the simulator.

Return type
int

eelib.model_connections

Submodules

eelib.model_connections.connections

Module to load connection config.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Functions

get_default_connections(→ dict)	Load default connection config from model_connect_config.json file.
get_connection_directions_config(→ dict)	Load set directions for connections from connect_directions_config.json file.

Attributes

conf

get_default_connections() → dict

Load default connection config from model_connect_config.json file.

Returns
Model connect config as Python dict.

Return type
dict

get_connection_directions_config() → dict

Load set directions for connections from connect_directions_config.json file.

Returns

Connect direction config as Python dict, with fields "ALWAYS_WEAK", "ALWAYS_STRONG",
and "STRONG_DIR".

Return type
dict

conf

eelib.utils

Contains helper functions and classes.

e.g. provide unified colors, return file content in a necessary format

Subpackages

eelib.utils.ancillary_services

Submodules

eelib.utils.ancillary_services.voltage_control_concepts

Methods for various concepts for voltage control of inverter-based devices.

Author: elenia@TUBS

Module Contents

Functions

cos_phi_fix(p, cos_phi)	Calculate reactive power output from active power and a fix cosinus phi (power factor).

cos_phi_fix(*p: float*, *cos_phi: float*)

Calculate reactive power output from active power and a fix cosinus phi (power factor). Negative active power (generation!) leads to positive reactive power (inductive behaviour). Positive active power (demand!) leads to negative reactive power (capacitive behaviour).

Parameters

- **p** (*float*) – active power of device (>0 is demand, <0 is generation)
- **cos_phi** (*float*) – power factor, value between 0 (not included!) and 1

Raises

ValueError – If cos_phi is outside of range from 0 to 1

Returns

reactive power output from device

Return type

float

eelib.utils.eval

Here the functionalities for evaluating and plotting/presenting outputs of simulations are stored, mostly in accessible and modifiable python scripts or jupyter notebooks.

Submodules

eelib.utils.eval.evaluation_utils

Useful helper methods for evaluating .hdf5 results in jupyter notebooks.

Author: elenia@TUBS

Module Contents

Functions

_extract_datasets_from_group(group[, parent_name, pattern])	Recursively extract data from hdf5 file.
hdf5_file_as_pandas(path[, datetime_col, pattern])	Return a dataframe representation of the hdf5 file.
get_config(→ dict)	Return scenario configuration of the hdf5 file.
convert_hdf5_to_csv(input_path, output_path[, sep, ...])	Converts an hdf5 file with the proper format into a csv file. Uses non-compact representation
timestep_to_datetime(timestep, zero_datetime, step_size)	Converts a (numpy list of) timestep(s) to a (numpy list of) actual date(s).
save_figure(fig, ax, filename, path[, figsize, dpi, ...])	Saves a Matplotlib figure with standardized sizing and format.
_read_config(hdf5_data)	Return dictionary corresponding to Config grouping.

_extract_datasets_from_group(*group*, *parent_name=''*, *pattern=None*)

Recursively extract data from hdf5 file.

Parameters

- **group** (*Group*) – The group where the search is currently occuring.
- **parent_name** (*str, optional*) – The parent name of that group.
- **pattern** (*str*) – regex pattern. If given, only the part matching the first grouping in the pattern remains as the name for each timeseries. Otherwise, the full path of the timeseries is used as its name.

Raises

ValueError – if pattern is given and at least one name does not match it

Returns

the fully formed subgroup for the current group.

Return type

dict

hdf5_file_as_pandas(*path: str*, *datetime_col=False*, *pattern=None*)

Return a dataframe representation of the hdf5 file.

Parameters

- **path** (*str*) – hdf5 file path.
- **datetime_col** (*bool, optional*) – determines whether a datetime column is created using the Config section of the database.
- **pattern** (*str*) – regex pattern. If given, only the part matching the first grouping in the pattern remains as the name for each timeseries. Otherwise, the full path of the timeseries is used as its name.

Returns

The resulting dataframe.

Return type

Dataframe

get_config(*path: str*) → dict

Return scenario configuration of the hdf5 file.

Parameters
path (*str*) – path of the hdf5 file.

Returns
scenario configuration dict.

Return type
dict

convert_hdf5_to_csv(*input_path: str*, *output_path: str*, *sep=','*, *na_rep=''*, *datetime_col=False*)

Converts an hdf5 file with the proper format into a csv file. Uses non-compact representation unless specified.

Parameters

- **input_path** (*str*) – path for input hdf5 file.
- **output_path** (*str*) – path for the output csv file.
- **sep** (*str*) – String of length 1. Field delimiter for the output file. Defaults to ‘,’.
- **na_rep** (*str*) – Missing data representation. Defaults to ‘’.
- **datetime_col** (*bool*) – Whether the function adds a column to dataframe to display the actual date and time in addition to timesteps. Only works if compact=True. Defaults to False

timestep_to_datetime(*timestep*, *zero_datetime: datetime.datetime*, *step_size: int*)

Converts a (numpy list of) timestep(s) to a (numpy list of) actual date(s).

Parameters

- **timestep** – a timestep or a numpy array of timesteps.
- **zero_datetime** (*datetime*) – the datetime corresponding to timestep 0. Inclusion of zero in the list is not mandatory.
- **step_size** (*int*) – size of each timestep in seconds.

Returns
calculated date(s) corresponding to timestep(s).

Return type
numpy.datetime64

save_figure(*fig*, *ax*, *filename*, *path*, *figsize=(15, 5)*, *dpi=300*, *format='svg'*, *rasterized=True*)

Saves a Matplotlib figure with standardized sizing and format.

Parameters

- **fig** (*Figure*) – Matplotlib figure to be saved.
- **ax** (*Axes*) – Matplotlib ax.
- **filename** (*str*) – Name of the output file.
- **path** (*str*) – path of the output file.
- **figsize** (*tuple*) – Size of the figure in inches (width, height).
- **dpi** (*int*) – Dots per inch for image resolution.
- **format** (*str*) – Output file format (e.g., ‘svg’, ‘png’, ‘jpg’, etc.).
- **rasterized** (*bool*) – Whether to rasterize vector elements (True) or not (False).

_read_config(*hdf5_data*)

Return dictionary corresponding to Config grouping.

Parameters
hdf5_data – data read from an hdf5 file using h5py.

Returns
the Config as stored.

Return type
dict

Submodules

eelib.utils.colormap

Provides an elenia colormap to use standardized colors across the project.

Author: elenia@TUBS

Module Contents

Functions

register_and_use_elenia_colormap()	Register the elenia colormap and set it as matplotlib.pyplot's default.

Attributes

PRIM_ELENIA_CMAP
SEC_ELENIA_CMAP
ELENIA_CMAP
PRIM_DISC_CMAP
SEC_DISC_CMAP
DISC_CMAP
PRIM_INT_CMAP
SEC_INT_CMAP
RED_GREEN_INT_CMAP
RED_GREEN_RED_INT_CMAP

PRIM_ELENIA_CMAP = ['#005374', '#00709B', '#66B4D3', '#BE1E3C', '#711C2F']

SEC_ELENIA_CMAP = ['#711C2F', '#D46700', '#E3B225', '#ACC13A', '#6D8300', '#00534A', '#003F57', '#00709B',...

ELENIA_CMAP

PRIM_DISC_CMAP

SEC_DISC_CMAP

DISC_CMAP

PRIM_INT_CMAP

SEC_INT_CMAP

RED_GREEN_INT_CMAP

RED_GREEN_RED_INT_CMAP

register_and_use_elenia_colormap()

Register the elenia colormap and set it as matplotlib.pyplot's default.

eelib.utils.logging_helpers

Logging helper with custom format and functions to set logger.

Author: elenia@TUBS

Module Contents

Classes

CustomFormatter	CustomFormatter with different colors for logging levels.

Functions

clear_logger()	Function to remove existing loggers.
set_console_logger([level])	Set console output logger.
set_file_logger([level, filename])	Set file logger.

Attributes

LOGGING_FORMAT

LOGGING_FORMAT = '%(asctime)s | %(levelname)-9s| %(threadName)s | %(name)s:%(lineno)d - %(message)s'

class CustomFormatter(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, ***, *defaults=None*)

Bases: `logging.Formatter`

CustomFormatter with different colors for logging levels. Specialization of logging.Formatter.

grey = '\x1b[38;20m'

yellow = '\x1b[33;20m'

red = '\x1b[31;20m'

bold_red = '\x1b[31;1m'

reset = '\x1b[0m'

format_str

FORMATS

format(*record*)

Overrides format method.

Parameters
record (*_type_*) – record to log.

Returns
Formatted log string.

Return type
str

clear_logger()

Function to remove existing loggers.

set_console_logger(*level: int = logging.DEBUG*)

Set console output logger. All messages will be printed to StdOut.

Parameters
level (*int*) – Logging level. Defaults to logging.DEBUG.

set_file_logger(*level: int = logging.DEBUG*, *filename: str = 'eELib.log'*)

Set file logger. All messages will be written to the file with given filename.

Parameters

- **level** (*int*) – Logging level. Defaults to logging.DEBUG.
- **filename** (*str*) – Logfile filename. Defaults to "eELib.log".

eelib.utils.read_pickle

Reads .pickle files.

Author: elenia@TUBS

Module Contents

Functions

read_pickle(path)	Read the whole data from a stored pickle file.

read_pickle(*path: str*)

Read the whole data from a stored pickle file.

Parameters
path (*str*) – path to pickle file

Raises
FileNotFoundError – Error if given pickle file path is not existent

Returns
stored data from pickle file

Return type
data

eelib.utils.resample

Methods for inter- or extrapolating timeseries data (in pandas Dataframe format).

Author: elenia@TUBS

Module Contents

Functions

get_resolution_pandas_timeseries(→ int)	Computes the time resolution of a given timeseries.
resample_pandas_timeseries_agg(→ pandas.DataFrame)	Resample a given DataFrame to a target time resolution.

Attributes

_logger

_logger

get_resolution_pandas_timeseries(*timeseries: pandas.DataFrame | pandas.Series*) → int

Computes the time resolution of a given timeseries.

TODO (1) handle if not DataFrame or Series (2) handle if not a DatetimeIndex

Parameters
timeseries (*pd.DataFrame | pd.Series*) – pandas Timeseries to determine resolution

Returns
time resolution in seconds

Return type
int

resample_pandas_timeseries_agg(*df: pandas.DataFrame*, *target_resolution: int*, *ffill_columns: list = []*, *distribute_columns: list = []*, *key: str = ''*) → pandas.DataFrame

Resample a given DataFrame to a target time resolution.

Columns that must be uniformly distributed or just filled forward

Parameters

- **df** (*pd.DataFrame*) – The df to be resampled.
- **target_resolution** (*int*) – The resolution to be sampled in (seconds).
- **ffill_columns** (*list*) – Columns to be filled forward. Defaults to [].
- **distribute_columns** (*list*) – Columns to be distributed. Defaults to [].
- **key** (*str*) – Key to describe the series (just for logging). Defaults to “”.

Returns
the resampled DataFrame

Return type
pd.DataFrame

eelib.utils.simulation_helper

Helper methods for retrieving simulation objects from lists or listing available entitities.

Author: elenia@TUBS

Module Contents

Functions

get_entity_by_id(e_list, e_id)	Provides an entity with regard to a given id from the entity list.
get_entity_by_id_from_dict_list(e_dict_list, e_id)	Provides an entity with regard to a given id from the entity list.
create_plots(world, dir_graphs, bool_plot[, ...])	Create some plots after the simulation.
start_simulators(→ dict)	Start all model factories for the simulators listed in SIM_CONFIG and one for each model.
create_entities(→ dict)	Create all models based on simulators and their models dict given in sim_config.
create_entities_of_model(→ list)	Create model entities for a given ModelFactory object with regard to model name and input.
get_grid_components(→ dict)	Collect all components of a specific grid.
check_model_connect_config(model_connect_config	Check the layout and input of a connection configuration for models.
connect_entities(world, dict_entities, ...)	Connect all model entities to each other based on the created models and their connections.
set_connection_direction(→ tuple[str, str])	Orders the weak-strong direction of two models by its model names to begin with.
connect_entities_in_grid(grid_model_config, grid, ...)	Connect all entities for a simulation containing a power grid.
connect_entities_to_db(sim_config, world, database, ...)	Connects all models (model entities) to the database.
connect_grid_to_db(sim_config_grid, world, database, ...)	Connects all grid components to the database.
connect_to_forecast(world, dict_entities, ...)	Create connections for the forecasts to work.
connect_entities_of_two_model_types(world, ...[, ...])	Connect the elements of two lists in both directions (if needed) via mosaik.
get_default_dirs(→ dict)	Gathers all needed directories resp. its paths based on eELib's default structure.

Attributes

_logger

_logger

get_entity_by_id(*e_list: list*, *e_id: str*)

Provides an entity with regard to a given id from the entity list.

Parameters

- **e_list** (*list*) – list of entities to be searched in
- **e_id** (*str*) – id of the entity that we search for

Returns

(1) model entity if found, otherwise None

(2) dict for the model entity (eid, full_id, type) if found, otherwise None

Return type
entity, dict

get_entity_by_id_from_dict_list(*e_dict_list: dict*, *e_id: str*)

Provides an entity with regard to a given id from the entity list.

Parameters

- **e_dict_list** (*dict*) – dict with lists of entities (sorted by type) to be searched in
- **e_id** (*str*) – id of the entity that we search for

Returns

(1) model entity if found, otherwise None

(2) dict for the model entity (eid, full_id, type) if found, otherwise None

Return type
entity, dict

create_plots(*world: object*, *dir_graphs: str*, *bool_plot: bool*, *slices_execution: list = [0, 4]*)

Create some plots after the simulation. Especially the execution_graph take some time to create and should not be created for every simulation.

Parameters

- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **dir_graphs** (*str*) – direction where to save the create plots
- **bool_plot** (*bool*) – whether to create the plots or not
- **slices_execution** (*list*) – list of length 2 to assign start and end index of the execution graph to plot. Defaults to [0, 4].

start_simulators(*sim_config: dict*, *world: object*, *scenario_config: dict*) → dict

Start all model factories for the simulators listed in SIM_CONFIG and one for each model.

Parameters

- **sim_config** (*dict*) – Information about the used simulators and their models
- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **scenario_config** (*dict*) – parameters for the simulation scenario

Returns
Contains all ModelFactorys sorted by the name of the corresponding model

Return type
dict

create_entities(*sim_config: dict*, *model_data: dict*, *dict_simulators: dict*, *grid_model_config: dict = {}*) → dict

Create all models based on simulators and their models dict given in sim_config.

Parameters

- **sim_config** (*dict*) – Information about the used simulators and their models
- **model_data** (*dict*) – dict containing all information about the to-be-created model entities
- **dict_simulators** (*dict*) – dict of all used simulators with their ModelFactory-objects
- **grid_model_config** (*dict*) – contains assigning the devices (by its type) to loads in the grid

Raises
KeyError – If a model is to be created but does not exist for this simulator (ModelFactory)

Returns
contains all created entities sorted by their model type

Return type
dict

create_entities_of_model(*model_factory: object*, *model_name: str*, *init_vals: list*) → list

Create model entities for a given ModelFactory object with regard to model name and input.

Parameters

- **model_factory** (*object*) – ModelFactory for a model
- **model_name** (*str*) – name of the model to create entities for
- **init_vals** (*list*) – initial values of the model (attributes to set)

Raises

- **KeyError** – If the ModelFactory has no such model type
- **TypeError** – If the initial values were given in the wrong format

Returns
all created model entities in one list (empty if no entity to create)

Return type
list

get_grid_components(*grid_comps_list: list*) → dict

Collect all components of a specific grid.

Parameters
grid_comps_list (*list*) – list of all grid components to collect from

Returns
containing all grid components separated by type

Return type
dict

check_model_connect_config(*model_connect_config: list*)

Check the layout and input of a connection configuration for models.

Parameters
model_connect_config (*list*) – list of the model properties to connect

Raises

- **TypeError** – If the connection config is no list
- **TypeError** – If an element of the connection config is neither a string nor a tuple
- **TypeError** – If an connection config element is a tuple but not consisting of two strings

connect_entities(*world: object*, *dict_entities: dict*, *model_connect_config: dict*, *dict_simulators: dict*)

Connect all model entities to each other based on the created models and their connections.

Parameters

- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **dict_entities** (*dict*) – dict of all used model entity objects as lists, sorted by their type

- **model_connect_config** (*dict*) – dict of all connected attributes between each model type
- **dict_simulators** (*dict*) – dict of all used simulators with their ModelFactory-objects

set_connection_direction(*name_from_init: str*, *name_to_init: str*) → tuple[str, str]

Orders the weak-strong direction of two models by its model names to begin with.

Parameters

- **name_from_init** (*str*) – name of model in initial from-position
- **name_to_init** (*str*) – name of model in initial to-position

Raises

ValueError – if no decision was found for connection of these two models

Returns

names of models, first FROM- and second TO-position

Return type

tuple[str, str]

connect_entities_in_grid(*grid_model_config: dict*, *grid: object*, *grid_loads: list*, *world: object*, *model_connect_conf: dict*, *dict_entities: dict*, *dict_simulators: dict*, *architecture_models: list = ['RetailElectricityProvider', 'GridEMS']*)

Connect all entities for a simulation containing a power grid. Includes connection between the devices in the grid and between devices (or EMS) and grid loads.

Parameters

- **grid_model_config** (*dict*) – contains assigning the devices (by its type) to loads in the grid
- **grid** (*object*) – power grid entity
- **grid_loads** (*list*) – list of all load objects in the power grid
- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **model_connect_conf** (*dict*) – dict of all connected attributes between each model type
- **dict_entities** (*dict*) – dict of all used model entity objects as lists, sorted by their type
- **dict_simulators** (*dict*) – dict of all used simulators with their ModelFactory-objects
- **architecture_models** (*list*) – all superordinate models to connect to each other and every ems

Raises

- **ValueError** – If more than one grid ems is created.
- **ValueError** – If more than one retail electricity provider is created.
- **KeyError** – No entity was created for a desired connection in the grid.
- **KeyError** – Given grid load name from model_connect_conf not found in grid loads list.
- **ValueError** – There are electric vehicles but no charging stations at a connection point.
- **KeyError** – A grid load is planned to be connected to two different entities.

connect_entities_to_db(*sim_config: dict*, *world: object*, *database: object*, *dict_entities: dict*)

Connects all models (model entities) to the database.

Parameters

- **sim_config** (*dict*) – Information about the used simulators and their models
- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **database** (*object*) – model object of the database for the simulation
- **dict_entities** (*dict*) – dict of all used model entity objects

connect_grid_to_db(*sim_config_grid: dict*, *world: object*, *database: object*, *dict_comps: dict*)

Connects all grid components to the database.

Parameters

- **sim_config_grid** (*dict*) – Information about the used grid simulator and the components with a list of their output attributes
- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **database** (*object*) – model object of the database for the simulation
- **dict_comps** (*dict*) – dict of all used grid component entity objects, sorted by their type

connect_to_forecast(*world: object*, *dict_entities: dict*, *dict_simulators: dict*, *forecast: object*, *forecast_sim: object*, *connect_models_names: list*)

Create connections for the forecasts to work. Includes mosaik connections to ems model and adding of the model entities to the forecasts list.

Parameters

- **world** (*object*) – mosaik world object to orchestrate the simulation process
- **dict_entities** (*dict*) – dict of all used model entity objects
- **dict_simulators** (*dict*) – dict of all used simulators with their ModelFactory-objects
- **forecast** (*object*) – forecast model entity
- **forecast_sim** (*object*) – simulator for the forecast model
- **connect_models_names** (*list*) – model names to directly connect to forecast entity

connect_entities_of_two_model_types(*world: object*, *entity_list_strong: list*, *entity_list_weak: list*, *model_connect_config_from_strong: list*, *model_connect_config_from_weak: list = []*, *sim_entities_weak: object = None*)

Connect the elements of two lists in both directions (if needed) via mosaik. Also add elements to list of controlled entities if the "weak" entity is an ems.

Parameters

- **world** (*object*) – mosaik World object
- **entity_list_strong** (*list*) – entities that have "strong" outgoing connections
- **entity_list_weak** (*list*) – entities that have "weak" outgoing connections
- **model_connect_config_from_strong** (*list*) – list of properties (str or tuple) to connect the two models types - from strong to weak
- **model_connect_config_from_weak** (*list*) – list of properties (str or tuple) to connect the two models types - from weak to strong. Defaults to [].
- **sim_entities_weak** (*object*) – Simulator of the "weak" entities for the adding of entities to the ems controlled entities. Defaults to None.

Raises

- **TypeError** – if given mosaik world is not correct (does not have connect method)
- **ValueError** – if no strong connection properties are given but there are weak connections

get_default_dirs(*dir_base*, *scenario: str = 'building'*, *grid: str = 'example_grid_kerber.json'*, *format_db: str = 'hdf5'*) → dict

Gathers all needed directories resp. its paths based on eELib's default structure.

Parameters

- **dir_base** (*str*) – string with path to root director
- **scenario** (*str*) – Type of used scenario (like building, grid etc.). Defaults to "building".
- **grid** (*str*) – name of the grid file to use. Defaults to "example_grid_kerber.json".
- **format_db** (*str*) – format of the database. Defaults to None.

Raises
FileNotFoundError – If model data for current simulation scenario not found

Returns
all relevant paths for a simulation

Return type
dict

eelib.utils.validation

Provide a dictionary of type and value validation rules and validate given parameters or print a readable string of the given set of rules.

Author: elenia@TUBS
Copyright 2024 elenia
This file is part of eELib, which is free software under the terms of the GNU GPL Version 3.

Module Contents

Functions

validate_init_parameters(model_class, input_param_dict)	Validate the type and value of a list of parameters according to a provided dictionary.
format_valid_parameters_dictionary(valid_paran	Take the valid parameters dictionary of a model and transform it into a readable string

validate_init_parameters(*model_class*, *input_param_dict: dict*)

Validate the type and value of a list of parameters according to a provided dictionary.

Parameters

- **model_class** (*class*) – a class representing a component. This should contain a method get_valid_parameters() that returns a dictionary.
- **input_param_dict** (*dict*) – a dictionary of parameter name value pairs to be assessed and validated.

Raises

- **TypeError** – Invalid parameter input types
- **ValueError** – Invalid parameter input values

format_valid_parameters_dictionary(*valid_parameters: dict*)

Take the valid parameters dictionary of a model and transform it into a readable string
format for the purpose of displaying it.

Parameters
valid_parameters (*dict*) – a dictionary representing valid parameter types and values

Returns
Message string with the collected errors from input validation

Return type
str

2.1.4 Disclaimer / Authors

Author: elenia@TUBS

If you intend to cite the eELib, please use this:

(TBD) Carsten Wegkamp, Henrik Wagner, Eike Niehs, Julien Essers, Marcel Luedecke, Mattias Hadlak, Bernd Engel: "eELib: Open-Source Model Library for Prosumer Power Systems and Energy Management Strategies", Open Source Modelling and Simulation of Energy Systems (OSMSES) 2024, Vienna, Austria, 2024

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

c

d

e

m

U

INDEX

Symbols

A

B

C

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y