emobpy

Carlos Gaete-Morales

Jun 20, 2022

GETTING STARTED

emobpy is a Python tool that can create battery electric vehicle time series. Four different time series can be created: vehicle mobility time series, driving electricity consumption time series, grid availability time series and grid electricity demand time series. The vehicles mobility time series are created based on mobility statistics. For driving electricity consumption time series, the properties of vehicles can be selected from a database with several actual battery electric vehicles models. *emobpy* is developed by the research group Transformation of the Energy Economy at DIW Berlin (German Institute of Economic Research).

Note: Cite this article: Gaete-Morales, C., Kramer, H., Schill, WP. et al. An open tool for creating battery-electric vehicle time series from empirical data, emobpy. Sci Data 8, 152 (2021). https://doi.org/10.1038/s41597-021-00932-9

CHAPTER

ONE

ANNOUNCEMENT

Join our webinar and do-a-thon on 25/03/2022 at 10am CET. . Register your attendance here: https://www.eventbrite.com/e/265898719227

CHAPTER

TWO

COMMUNICATION CHANNELS

Please use the following channels to share ideas, propose enhancements or ask for support:

- Slack: https://emobpy.slack.com (Use our invite link)
- Gitlab issue: https://gitlab.com/emobpy/emobpy/issues

CHAPTER

THREE

INSTRUCTIONS

This tool has been tested in window 7, Ubuntu 18.04, Ubuntu 19.04 and Suse Linux. It is recommended to install the package in an dedicated Python environment with Python version 3.6+.

Installation:

```
pip install emobpy
```

CHAPTER

FOUR

USAGE

You can use our project template. It is a folder that contains files with mobility probabilities, assumptions in a rules file and python scripts that show different python classes and functions to start generating the time series. To get a copy of the template folder, we create a project folder. For instance, as shown below, our project name is *my_evs*.

```
emobpy create -n my_evs
```

Hint: When we create a project folder for the first time, emobpy also copies files to our system user folder. In windows is usually located in *c:/users/your_win_user/AppData/Local/emobpy*, while for Linux is */home/your_linux_user/.local/share/emobpy*. The files hosted contain actual battery electric vehicle models, weather time series hourly across a year for different countries, and driving cycles divided on urban, rural, and highways.

Then by using the command line, we access to project folder *my_evs*:

```
cd my_evs
```

We can run the python script that enables us to generate examples of time series.

```
python Step1Mobility.py
```

read the instruction file in my_evs folder

Jupyter notebook offers a more interactive learning. You can open the `Time-series_generation.ipynb` by running jupyter in your console.

```
jupyter notebook
```

In the example section of the documentation, the code is clearly explained. Go directly to the example here.

Remove library:

```
pip uninstall emobpy
```

4.1 Links

- Documentation: https://emobpy.readthedocs.io/
- Source code: https://gitlab.com/diw-evu/emobpy/emobpy
- PyPI releases: https://pypi.org/project/emobpy
- License: http://opensource.org/licenses/MIT
- Code DOI: https://doi.org/10.5281/zenodo.3675456
- Dataset DOI: https://doi.org/10.5281/zenodo.3931663
- Issues: https://gitlab.com/diw-evu/emobpy/emobpy/issues
- Slack chat: https://emobpy.slack.com (Use our invite link)

4.2 Authors

The developers are Carlos Gaete-Morales (lead) and Lukas Trippe.

CHAPTER

FIVE

CHANGELOG:

v0.6.2 (2021-12-05)

- Fix: Solved updating custom dataset issue. BEV database and weather data are now updated.
- Fix: Logging issue. Messages are now logged in the correct order.
- Fix: Insulation data had an incorrect key *steel* and was not recognized. Heating and cooling calculations are more accurate now.

v0.6.1 (2021-12-02)

- Fix: Availability class had an inadequate allocation of states (locations), causing faulty availability and charging time-series.
- Fix: Add seed to a random number generator (get_seed(seed)).
- Improved message for short hours time series that may cause empty time series.

v0.6.0 (2021-12-01)

- Implemented 1-minute time-step (see template eg3)
- Implemented 1-second time-step (see template eg4)
- Added average power in W per time-step for driving consumption time-series
- Added instant power in W per time-step for driving consumption time-series
- Improved logging. Now the logging is done in a file, while the console can be suppressed
- Results can be exported to "DIETERpy" https://diw-evu.gitlab.io/dieter_public/dieterpy/
- Results can be exported to "SimSES" https://www.ei.tum.de/ees/simses/

5.1 Description

CHAPTER

SIX

VEHICLE MOBILITY TIME SERIES

The vehicle mobility time series contains the location of a vehicle at each point in time. The locations vary according to the mobility of drivers. Possible locations are at home, workplace, shopping, errands, escort, leisure, or driving. When "Driving", the distance travelled is also provided in the time series. The time resolution can be established initially (our examples contains 15 minutes time steps). The daily number of trips, the departure time, the trip purpose, distance travelled, and duration of the trips are determined based on statistics of mobility surveys. Other considerations can also be set up. For instance, the number of working hours per day, the first and last destination of the day, can be established as "at home". The "driving" will always be placed in between two different locations.

CHAPTER

SEVEN

DRIVING ELECTRICITY CONSUMPTION TIME SERIES

The previous time series is used as input to the creation of driving electricity consumption time series. The energy required for every trip is calculated based on the ambient temperature and traction effort for the vehicle's movement. To simulate the travel conditions, driving cycles are taken into account. The tool counts with battery electric vehicle models that are currently in the market. A vehicle's model has to be selected to include the model's parameters and characteristics.

CHAPTER

EIGHT

GRID AVAILABILITY TIME SERIES

Grid availability time series consists of taking a driving electricity consumption time series and based on the locations. The model assigns charging stations. Different charging stations can be available for a vehicle, and they are chosen based on a probability distribution that adds up 100% for each location. The charging stations defined in this tool are “home”, “public”, “maker”, “workplace”, “fast” and “none”, although more user-defined charging stations can be established. The charging stations have an associated capacity per time interval, and “none” has zero capacity. Different scenarios of grid availability can be modelled.

CHAPTER

NINE

GRID ELECTRICITY DEMAND TIME SERIES

While a grid availability time series contains at each interval information of the charging stations available, such as the maximum power rating allocated to them, a grid electricity demand time series is the one that indicates the actual consumption of electricity from the grid to charge the battery of a vehicle according to its driving needs and grid availability. There are different options available to create a grid electricity demand time series. For example, "Immediate-Full capacity" is an option that informs the energy drawn from the grid at a maximum power rating of a respective charging station until the battery is fully charged or "Immediate-Balanced" option that creates a time series taking into account the duration of a vehicle is connected to a charging station and the energy required to get the battery fully charged, allowing to charge the battery at a lower capacity than the maximum capacity available.

9.1 Installation

To run emobpy, you should have a functioning installation of Python. It is a good idea to install emobpy within a conda environment. The result can best be visualized in a jupyter notebook. However, it is also possible to run emobpy through other python interpreter.

The following instructions describes in detail how to install Python, how to create a `conda` environment, and finally how to install emobpy and its dependencies.

This tool has been tested in window 7, Ubuntu 18.04, Ubuntu 19.04 and Suse Linux. Python version 3.6+.

9.1.1 Installation of Python with Conda

The easiest and most convenient way to install Python is to install Anaconda (or Miniconda). Download the latest version from their website and install it.

During the installation of Anaconda on **Windows**, you will be asked to specify several options. We recommend you to choose the following ones so that emobpy will run smoothly:

- Install Anaconda to a custom directory (such as `C:/Anaconda` or `D:/Anaconda`) and do not install it in the default folder, because this might increase the log-in and log-out out time;
- Do not use `C:/Programs/` or `C:/Program Files/` because this will require admin rights (recommended for permission restricted computers);
- During the installation, select "advanced option" and check both boxes (despite not recommended by the application). This will add this Conda Python Installation to the path and enables it as default python.

Create a new Conda environment

An conda environment is an isolated Python space. Different environments can contain different Python packages of different versions. In order to have reproducible and stable "working space", it is useful to create a new environment for emobpy.

Anaconda offers different ways to create a new environment:

1. Anaconda Navigator

Start the Anaconda Navigator and go on *Environments* and then *create*. Choose a name, tick the box next to *Python* and choose a Python version compatible with your GAMS installation (see box above).

2. Console

Open a console (Anaconda Prompt, CMD, PowerShell, Windows Terminal) and create a new conda environment with the name *yourenvname* and the exemplary Python version *X.X* (we recommend 3.6) with the following command:

```
$ conda create -n yourenvname python=X.X
```

Note: To verify the successful creation of your environment, type `conda info --envs` in your console.

For further information on how to edit and delete conda environments, we refer to the conda documentation.

9.1.2 Installation of emobpy

Now, your are ready to install emobpy. Make sure you have activated the correct environment:

```
$ conda activate yourenvname
```

You can install emobpy easily from PyPI:

```
$ pip install emobpy
```

Installing emobpy from PyPI ensures that all necessary linked packages are downloaded and installed.

Note: You can uninstall emobpy, while having activated the *yourenvname* environment, simply by executing `$ pip uninstall emobpy` in the console.

9.2 Creating a project folder

Before start creating time-series we have to create a new project folder. By creating a project we will copy all required files from a template folder. These files can be modified according to our research purpose.

In our example, we want to create a new project called *my_evs* and use the console to create the project folder. Open the console (or terminal) where you want to create the project folder (e.g. in Windows by right-clicking and shift and choosing *Open Windows Terminal here*), or navigate to the desired folder within the console.

Once you are in the desired folder and if emobpy was installed correctly, type the following command:

```
$ emobpy create -n my_evs
```

or:

```
$ emobpy create --name my_evs
```

Warning: The name of the project must not contain any blanks. This can lead to errors.

9.2.1 Further options

In addition, further settings can be made. For example, a seed can be set. This allows that the same calculations are always executed in the same way. This makes it easy to compare projects and simplifies further development. If no seed is given, it will be randomly generated.

Argument	Shortcut	Optional	Description
–name	-n	Required	Name of the project folder.
–template	-t	Optional	Select a specific example project.
–seed	-s	Optional	Set a seed to make calculations reproducible.

This will create a folder and file structure as follows.

```
├── my_evs
│   └── config_files
│       ├── DepartureDestinationTrip.csv
│       ├── DistanceDurationTrip.csv
│       ├── TripsPerDay.csv
│       ├── rules.yml
│   ├── Time-series_generation.ipynb
│   ├── Step1Mobility.py
│   ├── Step2DrivingConsumption.py
│   ├── Step3GridAvailability.py
│   ├── Step4GridDemand.py
│   ├── Visualize_and_Export.ipynb
```

To start creating the time series, follow the instructions on the next page.

9.3 Generate the time-series

After having created a project folder (previuos section), you are ready to run the model. This happens in four steps.

The information in this chapter can also be found in the Instruction.txt file that is created with the creation of a new project folder.

9.3.1 Base Case

If not already done, change to project folder:

```
$ cd my_evs
```

Warning: It is recommended to perform the following steps only if a conda environment has been installed and activated. Instructions can be found in the *Installation* page.

Method 1: Jupyer Notebook

```
$ jupyter notebook
```

Once the browser opens up, select and open `Time-series_generation.ipynb`.

Method 2: Python interprreter

Run the script in the following order:

```
$ python Step1Mobility.py
$ python Step2DrivingConsumption.py
$ python Step3GridAvailability.py
$ python Step4GridDemand.py
```

After finishing all the runs, open *Visualize_and_Export.ipynb* for visualization of results.

Warning: jupyter notebook must be installed, it can be installed `conda install jupyter`.

9.4 Examples

This section contains examples developed to help understand the features and functionalities of emobpy. To facilitate the implementation and execution of the examples, the tool enables us to create projects from templates that contain such examples. We expect to provide more examples further forward.

To obtain the template of an example we have to run the command line interface as follows:

```
$ dieterpy create_project -n myproject -t eg1
```

In the above code snippet `-n` is an argument to provide the name of our project *myproject* and `-t` is the argument to provide the name of the template to obtain *eg1*.

Hint: If no template (-t) is provided `dieterpy create_project -n myproject`, then the base case folder will be generated.

9.4.1 Base Case

This is the base case that will be created if no specific template is selected. It serves as a foundation for own modeling and to get an overview for the program.

To initialize the base case and create a project folder, no template needs to be specified:

```
$ emobpy create -n <give a name>
```

Hint: Before running this example, install and activate a dedicated environment (a conda environment is recommended).

The initialisation creates a folder and file structure as follows:

```
├── my_evs
│   └── config_files
│       ├── DepartureDestinationTrip.csv
│       ├── DistanceDurationTrip.csv
│       ├── TripsPerDay.csv
│       ├── rules.yml
│   ├── Time-series_generation.ipynb
│   ├── Step1Mobility.py
│   ├── Step2DrivingConsumption.py
│   ├── Step3GridAvailability.py
│   ├── Step4GridDemand.py
│   ├── Visualize_and_Export.ipynb
```

This base case consists of four *.py* files that run the modelling, a *.ipynb* to visualise the results and the *config_files* folder that contains mobility data.

File name	Description
`config_files/`	Mobility data files that can be changed in this folder.
`Step1Mobility.py`	Uses `emobpy.Mobility()` to create individual mobility time series with vehicle location and distance travelled.
`Step2DrivingConsumption.py`	Uses `emobpy.Consumption()` to assign vehicles and to model their consumption.
`Step3GridAvailability.py`	Uses `emobpy.Availability()` to create the grid availability time series.
`Step4GridDemand.py`	Uses `emobpy.Charging()` to calculate the grid electricity demand time series.
`Visualize_and_export.ipynb`	Jupyter Notebook File to view the results. See Visualization.
`Time-series_generation.ipynb`	Jupyter Notebook File to create and visualize all four time series (Recomended).

After initialisation, you have two options: Using jupyter notebook or the python interpreter directly.

Method 1: Using Jupyter notebook

```
$ jupyter notebook
```

It will open the notebook in your browser. The document contains all instructions.

> **Warning:** Make sure you have installed jupyter in your activated environment. To install it type in the console `conda install jupyter`

The jupyter notebook file could look like this, for example:

Method 2: Python interpreter

Run the script in the following order:

```
$ cd <given name>
$ python Step1Mobility.py
$ python Step2DrivingConsumption.py
$ python Step3GridAvailability.py
$ python Step4GridDemand.py
```

The results are saved as pickle files. To read them, two methods can be implemented. Using the DataBase class as described in the Visualize_and_Export.ipynb or by opening the pickle file directly. More information can be found in the pickle documentation.

The pickle file can be opened as follows:

```
pickle_in = open("data.pickle","rb")
data = pickle.load(pickle_in)
```

The jupyter notebook file *.ipynb* file could look like this:

9.4.2 BEV models

This example shows the different cars that are included in the database. A Sankey diagram can be used to clearly identify the energy usage for each car.

To initialize the example1 and create a project folder, the template *eg1* must be selected:

```
$ emobpy create -n <give a name> -t eg1
```

> **Warning:** Before running this example, install and activate an emobpy dedicated environment (conda recommended).

The initialisation creates a folder and file structure as follows.

```
├── my_evs
│   └── config_files
│       ├── DepartureDestinationTrip_Worker.csv
│       ├── DistanceDurationTrip.csv
```

(continues on next page)

(continued from previous page)

```
│       ├── rules.yml
│       ├── TripsPerDay.csv
│   ├── eg1.ipynb
```

Everything, running and visualization, happens in a single jupyter notebook file. In the default settings a single mobility profile is created and three consumption profiles for different car types. For each car a Sankey diagram is created to show the different energy usage of the cars.

The Jupyter notebook might look like this:

9.5 emobpy

9.5.1 emobpy package

emobpy.availability module

This module contains the Availability class that creates the grid availability time series. The class requires to be provided with the name of the driving consumption profile on which will build up the new time series. It will also require the charging station power rating and charging station probability distribution based on the location. The location can also be associated with the trip purpose or destination. The time series indicate the location as a state.

The Availability class reads the consumption profile. Every row contains a location, arriving time, duration, distance and energy consumption from battery for driving. Following the availability probability distribution, the tool looks at every row's location (state) and sample a charging station. After the charging stations have been allocated to every row, the tool tests if the allocation complies with the energy requirements. The state of charge (SOC) is determined assuming an immediate charging strategy. This strategy consists of charging at the maximum power rate whenever the current SOC is between 0 - 1. If the resulting SOC never goes below zero. Then the allocation is considered correct, and the charging availability time series is done; otherwise, a new allocation occurs.

The allocation of charging stations is carried out several times until a successful allocation is reached or the maximum number of attempts is attained. If the latter, the file name will contain FAIL word.

For more details see the article and cite:

```
@article{Gaete-Morales_2021,
author={Gaete-Morales, Carlos and Kramer, Hendrik and Schill, Wolf-Peter and Zerrahn,␣
↪Alexander},
title={An open tool for creating battery-electric vehicle time series from empirical␣
↪data, emobpy},
journal={Scientific Data}, year={2021}, month={Jun}, day={11}, volume={8}, number={1},␣
↪pages={152},
issn={2052-4463}, doi={10.1038/s41597-021-00932-9}, url={https://doi.org/10.1038/s41597-
↪021-00932-9}}
```

See also the examples in the documentation https://diw-evu.gitlab.io/emobpy/emobpy

class emobpy.availability.**Availability**(*inpt*, *db*)

Bases: `object`

Instance that represents a grid availability time series. It requires the driving consumption profile name (inpt) on which will build up the new time series and the database instace (db) where the consumption profiles are hosted.

Parameters

- **inpt** (*str*) – driving consumption profile name

- **db** (*DataBase()*) – class instance that contains the profiles

Example

```
GA = Availability('ev1_abc_tesla3_def', DB)
GA.set_scenario(charging_data)
GA.run()
GA.save_profile('path to folder')
```

run()

No input required. Once it finishes the following attributes can be called.

Attributes:

- kind
- input
- chargingdata
- battery_capacity
- charging_eff
- discharging_eff
- soc_init
- soc_min
- storage_altern
- profile
- timeseries
- success
- name
- proportion_ts_modified

save_profile(*folder*, *description=''*)

Saves object profile as a pickle file.

Parameters

- **folder** (*str*) – Where the files will be stored. Folder is created in case it does not exist.
- **description** (*str, optional*) – Description which can be saved in object attribute. Defaults to ” “.

set_scenario(*charging_data*)

Sets given charging_data to object.

Parameters

charging_data (*dict*) – E.g.

```
{
    'prob_charging_point' :
        {'errands':  {'public':0.3,'none':0.7},
        'escort':   {'public':0.3,'none':0.7},
```

(continues on next page)

(continued from previous page)

```
        'leisure':  {'public':0.3,'none':0.7},
        'shopping': {'public':0.3,'none':0.7},
        'home':     {'public':0.3,'none':0.7},
        'workplace':{'public':0.0,'workplace':0.3,'none':0.7},
        'driving':  {'none':1.0}
        },
    'capacity_charging_point' :
        {'public':11,'home':1.8,'workplace':5.5,'none':0}
}
```

emobpy.availability.**add_column_datetime**(*df*, *totalrows*, *reference_date*, *t*)

Useful to convert the time series from hours index to datetime index.

Parameters

- **df** (*pd.DataFrame*) – Table on which datetime column should be added.
- **totalrows** (*int*) – Number of rows on which datetime column should be added.
- **reference_date** (*str*) – Starting date for adding. E.g. '01/01/2020'.
- **t** (*float*) – Float frequency, will be changed to string.

Returns

Table with added datetime column.

Return type

pd.DataFrame

emobpy.charging module

While a grid availability time series contains at each interval information of the charging stations available, such as the maximum power rating allocated to them, a grid electricity demand time series is the one that indicates the actual consumption of electricity from the grid to charge the battery of a vehicle according to its driving needs and grid availability. There are different options available to create a grid electricity demand time series. For example, "Immediate-Full capacity" is an option that informs the energy drawn from the grid at a maximum power rating of a respective charging station until the battery is fully charged or "Immediate-Balanced" option that creates a time series taking into account the duration of a vehicle is connected to a charging station and the energy required to get the battery fully charged, allowing to charge the battery at a lower capacity than the maximum capacity available.

For more details see the article and cite:

```
@article{Gaete-Morales_2021,
author={Gaete-Morales, Carlos and Kramer, Hendrik and Schill, Wolf-Peter and Zerrahn,␣
↪Alexander},
title={An open tool for creating battery-electric vehicle time series from empirical␣
↪data, emobpy},
journal={Scientific Data}, year={2021}, month={Jun}, day={11}, volume={8}, number={1},␣
↪pages={152},
issn={2052-4463}, doi={10.1038/s41597-021-00932-9}, url={https://doi.org/10.1038/s41597-
↪021-00932-9}}
```

See also the examples in the documentation https://diw-evu.gitlab.io/emobpy/emobpy

class emobpy.charging.**Charging**(*inpt*)

Bases: object

Parameters

self.__init__ (*input*) – input: string. File name of the input profile (not the path). The input should be in this case a grid availability profile name.

Methods in the following order:

- self.loadScenario(DataBase)
- self.setSubScenario(option)
- self.run()
- self.save_profile(folder, description=' ')

`load_scenario`(*database*)

Loads scenario data from given database into object.

Parameters

database (*DataBase()*) – E.g. manager = DataBase(). "manager" is a class instance that contains the profiles.

Raises

`ValueError` – Raised if charging profile can not be found in the database.

`run`()

No input required. Once it finishes the following attributes can be called.

Attributes:

- kind
- input
- change_battery_cap
- pointmissing
- success
- option
- profile
- timeseries
- name

`save_profile`(*folder*, *description=' '*)

Saves object profile as a pickle file.

Parameters

- **folder** (*str*) – Where the files will be stored. Folder is created in case it does not exist.
- **description** (*str, optional*) – Description which can be saved in object attribute. Defaults to " ".

`set_sub_scenario`(*option*)

Sets sub scenario in self.option.

Parameters

option (*str*) – 'immediate', 'balanced' or 'from_22_to_6_at_home'.

emobpy.charging.**add_column_datetime**(*df*, *totalrows*, *reference_date*, *t*)

Useful to convert the time series from hours index to datetime index.

Parameters

- **df** (*pd.DataFrame*) – Table on which datetime column should be added.
- **totalrows** (*int*) – Number of rows on which datetime column should be added.
- **reference_date** (*str*) – Starting date for adding. E.g. '01/01/2020'.
- **t** (*float*) – Float frequency, will be changed to string.

Returns
Table with added datetime column.

Return type
pd.DataFrame

emobpy.charging.**is_between**(*t*, *time_range*)

Checks if given value is between given time_range.

Parameters

- **t** (*float*) – Value to check.
- **time_range** (*list*) – Time range list. E.g. [1,100].

Returns
Value if t is between time_range.

Return type
bool

emobpy.charging.**represents_int**(*s*)

Check if argument is an int value.

Parameters
s (*any type*) – Value to check.

Returns
True if the argument is an int value.

Return type
bool

emobpy.constants module

List of some constants used for the calculations.

For more details see the article and cite:

```
@article{Gaete-Morales_2021,
author={Gaete-Morales, Carlos and Kramer, Hendrik and Schill, Wolf-Peter and Zerrahn,␣
↪Alexander},
title={An open tool for creating battery-electric vehicle time series from empirical␣
↪data, emobpy},
journal={Scientific Data}, year={2021}, month={Jun}, day={11}, volume={8}, number={1},␣
↪pages={152},
issn={2052-4463}, doi={10.1038/s41597-021-00932-9}, url={https://doi.org/10.1038/s41597-
↪021-00932-9}}
```

emobpy.consumption module

Based on the vehicle mobility time series, the driving electricity consumption (ii) time series is derived. This requires further input data, such as information on nominal motor power, curb weight, drag coefficient, and dimensions, which the tool includes for several current BEV models. Ambient temperature is also a significant parameter that affects the consumption of BEV. For that reason, emobpy is endowed with a database of hourly temperature for European countries with a registry of the last 17 years. Additionally, the vehicle cabin insulation characteristics are required; this data is not widely available and thus assumed independently of the BEV models database. Driving cycles are also important input parameters that are used to simulate every individual trip. The model includes two driving cycles, Worldwide Harmonized Light Vehicles Test Cycle (WLTC) and Environmental Protection Agency (EPA). This input data is already provided within the tool, and the user can select a particular BEV model, country weather, and driving cycle. Alternatively, emobpy also allows providing user-defined custom data.

For more details see the article and cite:

```
@article{Gaete-Morales_2021,
author={Gaete-Morales, Carlos and Kramer, Hendrik and Schill, Wolf-Peter and Zerrahn,␣
↪Alexander},
title={An open tool for creating battery-electric vehicle time series from empirical␣
↪data, emobpy},
journal={Scientific Data}, year={2021}, month={Jun}, day={11}, volume={8}, number={1},␣
↪pages={152},
issn={2052-4463}, doi={10.1038/s41597-021-00932-9}, url={https://doi.org/10.1038/s41597-
↪021-00932-9}}
```

See also the examples in the documentation https://diw-evu.gitlab.io/emobpy/emobpy

class emobpy.consumption.**BEVspecs**(*filename=None*)

Bases: object

average(*parameter*)

Returns average of a given parameter from self.data.

Parameters
parameter (*str*) – Parameter of which the average is required.

Returns
Average of the parameter.

Return type
float

dropna_model(*parameter*)

Delete all na in self.data for given parameter.

Parameters
parameter (*str*) – Parameter from which to delete.

get(*brand*, *model*, *year*, *parameter*)

Search for specific information in the vehicle database and returns it.

Parameters

- **brand** (*str*) – E.g 'Volkswagen'
- **model** (*str*) – E.g. 'ID.3'
- **year** (*int*) – E.g. 2020
- **parameter** (*str*) – E.g. 'acc_0_100_kmh'

Returns
Requested value. None if nothing was found.

Return type
[float]

`get_fallback_parameter`(*parameter*)

Get data for a given parameter if it is missing.

Parameters
parameter (*str*) – Parameter to get data value for.

Returns
Fallback data value for given parameter.

Return type
float

`maximum`(*parameter*)

Returns maximum of specific parameter for the object.

Parameters
parameter (*str*) – Parameter of which the maximum is required.

Returns
Maximum of the parameter.

Return type
float

`model`(*model*, *use_fallback=True*, *msg=True*)

Initializes ModelSpecs object, adds parameters and checks them.

Parameters

- **model** (*tuple*) – Data of model. E.g ('Volkswagen', 'ID.3', 2020)
- **use_fallback**
- **msg** (*bool, optional*) – Flag, whether to inform about missing parameters. Defaults to True.

Returns
Initialized object.

Return type
ModelSpecs object

`replacena_model`(*parameter*, *default*)

Replace all na in self.data for given parameter with default value.

Parameters

- **parameter** (*str*) – Parameter from which to delete.
- **default** (*float*) – Value to be used instead.

`save`()

Save self.data into .yml file.

`search_by_parameter`(*parameter='power'*, *first_x=10*, *brand_filter=[]*, *model_filter=[]*, *year_filter=[]*)

Searching for vehicles sorted in descending order of given parameter. It returns a Pandas DataFrame.

Parameters

- **parameter** (*str*) – Vehicle parameter to compare. Defaults to 'power'

- **first_x** (*int*) – Number of vehicles to show. Defaults to 10.
- **brand_filter** (*int*) – Filter for brands. Defaults to [].
- **model_filter** (*int*) – Filter for models. Defaults to [].
- **year_filter** (*int*) – Filter for years. Defaults to [].

Returns
data (pd.DataFrame)

`show_models`(*brand=''*, *model=''*, *year=''*)

Shows a list of all cars from the database. Can be filtered by brand, model and year.

Parameters

- **brand** (*str, optional*) – Show only cars that match the brand. Defaults to ''.
- **model** (*str, optional*) – Show only cars that match the model. Defaults to ''.
- **year** (*str, optional*) – Show only cars that match the year. Defaults to ''.

`class` emobpy.consumption.`Consumption`(*inpt*, *ev_model*)

Bases: `object`

`load_setting_mobility`(*DataBase*)

Load certain attributes of the object with data from the transferred database.

Attributes:

- self.df
- self.t
- self.totalrows
- self.hours
- self.freq
- self.refdate
- self.energy_consumption
- self.states

Parameters
DataBase (*DataBase()*) – Database from which the data is to be loaded.

Raises
`ValueError` – A driving profile can not be found in the database.

`run`(*heat_insulation*, *weather_country='DE'*, *weather_year=2016*, *passenger_mass=75*, *passenger_sensible_heat=70*, *passenger_nr=1.5*, *air_cabin_heat_transfer_coef=10*, *air_flow=0.01*, *driving_cycle_type='WLTC'*, *road_type=0*, *wind_speed=0*, *road_slope=0*)

#TODO Docstring

Parameters

- **heat_insulation** (*object*) – [description]
- **weather_country (str, optional** – [description]. Defaults to "DE".
- **weather_year** (*int, optional*) – [description]. Defaults to 2016.
- **passenger_mass** (*int, optional*) – Passenger mass in kg. Defaults to 75.

- **passenger_sensible_heat** (*int, optional*) – Passenger sensible heat in W. Defaults to 70.
- **air_cabin_heat_transfer_coef** (*int, optional*) – Coefficient in W/(m2K). Defaults to 10.
- **air_flow** (*float, optional*) – Ranges from 0.02 (high ventilation) to 0.001 (minimum ventilation) in me/s.
- **Defaults to 0.01.**
- **driving_cycle_type** (*str, optional*) – [desc]. Defaults to "WLTC".
- **road_type** (*int, optional*) – See function rolling_resistance_coef(method='M1') if an integer then all trips
- **have the same value, if list must fit the length of the profile. Defaults to 0.**
- **wind_speed** (*int, optional*) – m/s if an integer then all trips have the same value, if list must fit the
- **length of the profile. Defaults to 0.**
- **road_slope** (*int, optional*) – Radians if an integer then all trips have the same value, if list must fit the
- **length of the profile. Defaults to 0.**

Raises

Exception – [description]

`save_profile`(*folder*, *description=''*)

Saves object profile as a pickle file.

Parameters

- **folder** (*str*) – Where the files will be stored. Folder is created in case it does not exist.
- **description** (*str, optional*) – Description which can be saved in object attribute. Defaults to " ".

`class` emobpy.consumption.`DrivingCycle`

Bases: `object`

`create_data`()

Create self.data from self.dc_df.

`driving_cycle`(*trip*, *model*, *full_driving_cycle=False*)

Calculates driving cycle from Trip and ModelSpecs object.

Parameters

- **trip** (*Trip*) – Trip for the driving cycle.
- **model** (*ModelSpecs*) – Vehicle Model for the driving cycle.
- **full_driving_cycle** (*bool, optional*) – [description]. Defaults to False.

`get_csv`(*csv_path='/home/docs/checkouts/readthedocs.org/user_builds/emobpy/envs/latest/lib/python3.8/site-packages/emobpy/data/driving_cycles.csv'*)

Load csv as dataframe into self.dc_df

Parameters

csv_path (*str, optional*) – Path of file. Defaults to os.path.join(MODULE_DATA_PATH, "driving_cycles.csv").

load_data()

Load data from self.datafile to self.data.

save_data()

Save self.tmpdata to file.

class emobpy.consumption.HeatInsulation(*default=False*)

Bases: object

compile()

Set all zones from self.zone_names to self.zone_layers.

show()

Prints Heat Insulation attributes.

class emobpy.consumption.MGefficiency(*filename=None*)

Bases: object

get_efficiency(*load_fraction*, *g_m_code*)

g_m_code: 1 -> motor, -1 -> generator #TODO DOCSTRING :Parameters: * **load_fraction** (*[type]*)

- **g_m_code** (*-1, 1*) – 1 for motor, -1 for generator

Raises

Exception – Raised if g_m_code is not 1 or -1.

Returns

[description]

Return type

type

class emobpy.consumption.ModelSpecs(*model*, *BEVspecs_instance*)

Bases: object

add(*parameters_dict*, *msg=True*)

Adds parameter and associated value to the object.

Parameters

- **parameters_dict** (*dict*) – Contains the name of the parameters and the corresponding value.
- **msg** (*bool, optional*) – Flag, whether to inform about added parameters. Defaults to True.

add_calculated_param()

Calculate all parameters that can be calculated from existing information.

add_fallback_data()

add_parameters()

Adds a value from the database to all parameters in self.db.parameters.

addtodb()

Adds parameters, which are not None, to database.

Returns

Returns list of None parameters, which can not be added to database.

Return type

None, List of None parameters

class emobpy.consumption.**Trip**(*trips*)

Bases: object

add_distance_duration(*distance*, *duration*)

Sets loaded distance and duration to self.distance and self.duration and saves mean_speed.

Parameters

- **distance** (*dict*) – Dictionary containing value and unit. E.g. {'value': 10.0, 'unit': 'km'}
- **duration** (*dict*) – Dictionary containing value and unit. E.g. {'value': 15.0, 'unit': 'min'}

get_mean_speed()

Calculates mean_speed ans saves it in object attribute.

class emobpy.consumption.**Trips**

Bases: object

add_trip(*trip*)

Adds single trip to trips object.[summary]

Parameters
trip (*Trip*) – Object to be added to the trips collection.

get_code()

Returns trip code based on quantity of trip plus 1.

Returns
Trip code.

Return type
int

get_trip(*code*)

Returns specific trip based on code.

Parameters
code (*int*) – Code of the requested trip.

Returns
Requested trip object.

Return type
Trip

class emobpy.consumption.**Weather**

Bases: object

static air_density_from_ideal_gas_law(*t*, *p*)

Calculate Air density from ideal gas law.

Parameters

- **t** (*float*) – Temperature in degree Celsius.
- **p** (*float*) – Pressure in mega bar.

Returns
Air density

Return type
array

`static calc_dew_point(t, h)`

Calculate dew point. :Parameters: * **t** (*float*) – air temperature in degree Celsius.

- **h** (*float*) – Relative humidity in percent.

Return type
array

`static calc_dry_air_partial_pressure(P, pv)`

Calculate dry air partial pressure.

Parameters

- **P** (*float*) – [description]
- **pv** (*float*) – [description]

Returns
Dry air partial pressure.

Return type
array

`calc_rel_humidity(Dp, T)`

Calculate humidity.

Parameters

- **Dp** (*array*) – Dew point temperature in degree Celsius.
- **T** (*array*) – Temperature in degree Celsius.

Returns
Relative humidity in percent.

Return type
array

`static calc_vapor_pressure(t)`

Calculate vapor pressure.

Parameters
t (*array*) – Dew point or air temperature in degree Celsius.

Returns
Vapor pressure array

Return type
array

`dewpoint(country_code, year)`

Loads selected dew point data in Kelvin into object.

Parameters

- **country_code** (*str*) – E.g. 'DE'.
- **year** (*int*) – E.g. 2016.

Returns
Dew point data.

Return type
list

`static download_weather_data`(*location=None*)

Download weather data from zenodo.

Parameters
location (*str, optional*) – Path to user path. Defaults to None.

Returns
Weather data.

Return type
list

`humidair_density`(*t*, *p*, *dp=None*, *h=None*)

Calculate humid air density.

Parameters

- **t** (*array*) – Temperature in degree Celsius.
- **p** (*array*) – Pressure in mbar.
- **dp** (*array, optional*) – Dew point temperature in degree Celsius. Defaults to None.
- **h** (*array, optional*) – Humidity in percent. Defaults to None.

Raises
`Exception` – Dp or h is missing.

Returns
Humid air density.

Return type
array

`pressure`(*country_code*, *year*)

Loads selected pressure data in Pascal into object.

Parameters

- **country_code** (*str*) – E.g. 'DE'.
- **year** (*int*) – E.g. 2016.

Returns
Pressure data.

Return type
list

`temp`(*country_code*, *year*)

Loads selected temperature data in Kelvin into object.

Parameters

- **country_code** (*str*) – E.g. 'DE'.
- **year** (*int*) – E.g. 2016.

Returns
Temperature data.

Return type
list

emobpy.consumption.**acceleration**(*V0*, *V2*)

Calculate and returns acceleration.

Parameters

- **V0** (*float*) – Old speed.
- **V2** (*float*) – New speed.

Returns
Acceleration.

Return type
float

emobpy.consumption.**acceleration_array**(*speed_array*)

Calculates and returns acceleration array from speed_array. The acceleration of the adjoining values is calculated.

Parameters
speed_array (*ndarray*) – Array with speed values.

Returns
Array with acceleration values.

Return type
ndarray

emobpy.consumption.**bar_progress**(**args*)

Prints actual progress in format: "Downloading: 80% [8 / 10] kilobyte" :Parameters: * **current** (*int*) – Current download.

- **total** (*int*) – Total number of downloads.

emobpy.consumption.**consumption_progress**(*current*, *total*)

Prints message about consumption progress.

Parameters

- **current** (*int*) – Current index.
- **total** (*int*) – Total number of loops.
- **width** (*int, optional*) – Not used. Defaults to 80.

emobpy.consumption.**include_weather**(*pf*, *refdate*, *temp_arr*, *pres_arr*, *dp_arr*, *H*, *r_ha*)

Adds weather data to given DataFrame.

Parameters

- **pf** (*pd.DataFrame*) – DataFrame where weather data should be added.
- **refdate** (*str*) – E.g. '01/01/2020'.
- **temp_arr** (*ndarray*) – Temperature in degree Celsius.
- **pres_arr** (*ndarray*) – Pressure in mbar.
- **dp_arr** (*ndarray*) – Dewpoint data in degree Celsius.
- **H** (*ndarray*) – Humidity data.
- **r_ha** (*ndarray*) – Air density in kg/m3.

Returns
[description]

Return type
pd.DataFrame

emobpy.consumption.**inertial_mass**(*curb_weight*, *gear_ratio*)

Calculates and returns inertial mass.

Parameters

- **curb_weight** (*float*) – Curb weight of the car.
- **gear_ratio** (*float*) – Gear ratio of the car.

Returns
Inertial mass of the car.

Return type
float

emobpy.database module

This module contains data organisation classes to read, load and edit the resulting time series. See also the examples in the documentation https://diw-evu.gitlab.io/emobpy/emobpy

For more details see the article and cite:

```
@article{Gaete-Morales_2021,
author={Gaete-Morales, Carlos and Kramer, Hendrik and Schill, Wolf-Peter and Zerrahn,␣
↪Alexander},
title={An open tool for creating battery-electric vehicle time series from empirical␣
↪data, emobpy},
journal={Scientific Data}, year={2021}, month={Jun}, day={11}, volume={8}, number={1},␣
↪pages={152},
issn={2052-4463}, doi={10.1038/s41597-021-00932-9}, url={https://doi.org/10.1038/s41597-
↪021-00932-9}}
```

class emobpy.database.**DataBase**(*folder*)

Bases: `object`

DataBase object useful to manage many.

important attribute: self.db : It is a dictionary that contains all profiles. The dictionary keys are the name of the profile

Every profile in this dict has nested dictionary. The keys depend on the type of profile. Common keys:

self.db[“name of the profile”][“kind”] that can be [“driving”, “availability”, “charging”]
self.db[“name of the profile”][“input”] that is a string only for [“availability”, “charging”] profiles

self.__init__(folder)
folder: path as string of folder where profiles are hosted.

getdb()

Run self.loadfiles() and return imported database.

Returns
Loaded database object.

Return type
DataBase

`loadfiles`(*loaddir=''*)

Load profiles and host in a directory other than the "folder". So that directory must be indicated (loaddir). In this way profiles from many directories can be loaded.

Parameters
loaddir (*str, optional*) – Directory to load from. Defaults to ''.

`loadfiles_batch`(*loaddir=''*, *batch=10*, *nr_workers=4*, *kind=''*, *add_variables=[]*)

Load datafiles into DataBase object for further usage.

Parameters

- **loaddir** (*str, optional*) – Directory to load from. Defaults to ''.
- **batch** (*int, optional*) – Number of batches to load. Defaults to 10.
- **nr_workers** (*int, optional*) – Number of workers to load. Defaults to 4.
- **kind** (*str, optional*) – Data kind to load. E.g 'consumption'. Defaults to ''.
- **add_variables** (*list, optional*) – New variables to load. Defaults to [].

`static loadpkl`(*f*, *variables*, *kind*)

Load from pickle file.

Parameters

- **f** (*str*) – Path to pickle file.
- **variables** (*str*) – Variables which should be loaded.
- **kind** (*[type]*) – Data kind to load.

Returns
Loaded object.

Return type
DataBase

`remove`(*name*)

Remove part of database.

Parameters
name (*str*) – Key which is to be deleted.

`update`()

Run self.laodfiles() to load files from database "folder".

`class emobpy.database.DataManager`

Bases: `object`

Data Manager to load and save files.

`loaddb`(*dbfilepath*, *profilesdir*)

Load database from pickle file.

Parameters

- **dbfilepath** (*str*) – Path to pickle file.
- **profilesdir** (*str*) – Path to profiles directory.

Returns
Loaded database from pickle file.

Return type
object

`savedb`(*obj*, *dbdir='db_files'*)

Save database to pickle file.

Parameters

- **obj** (*object*) – Database to be saved.
- **dbdir** (*str, optional*) – Path to database directory. Defaults to 'db_files'.

emobpy.export module

This module contains a class that can be used to export results to CSV file in a format that is useful for modelling BEV in the power system model DIETER and DIETERpy https://diw-evu.gitlab.io/dieter_public/dieterpy.

The documentation contains examples of Export class https://diw-evu.gitlab.io/emobpy/emobpy

`class emobpy.export.Export`(*groupbyhr=True*, *rows=None*, *kwto='MW'*)

Bases: `object`

exp = Export() exp.loaddata(db), where db is an instance of DataBase Class. exp.to_csv() exp.save_files('csv')

`loaddata`(*db*)

Load data from database into self.data.

Parameters
db () – Database to be loaded.

`save_files`(*repository=''*)

Saves object as an csv.

Parameters
repository (*str, optional*) – Path to repository. Defaults to "".

`to_csv`()

Change object data to csv structure.

emobpy.functions module

Fuctions used for Consumption class

`emobpy.functions.EFFICIENCYregenerative_braking`(*acceleration*)

#TODO DOCSTRING

Parameters
acceleration (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**balance**(*db*, *tscode*, *include=None*)

#TODO DOCSTRING

Parameters

- **db** (*[type]*) – [description]
- **tscode** (*[type]*) – [description]
- **include** (*[type], optional*) – [description]. Defaults to None.

Raises

Exception – [description]

Returns

[description]

Return type

[type]

emobpy.functions.**cp**(*T*)

#TODO DOCSTRING

Parameters

T (*[type]*) – [description]

Returns

[description]

Return type

[type]

emobpy.functions.**htc_air_out**(*vehicle_speed*, *limit=5*)

#TODO DOCSTRING

Parameters

- **vehicle_speed** (*[type]*) – [description]
- **limit** (*int, optional*) – [description]. Defaults to 5.

Returns

[description]

Return type

[type]

emobpy.functions.**p_generatorin**(*p_wheel*, *transmission_eff*, *regenerative_braking_eff*)

#TODO DOCSTRING

Parameters

- **p_wheel** (*[type]*) – [description]
- **transmission_eff** (*[type]*) – [description]
- **regenerative_braking_eff** (*[type]*) – [description]

Returns

[description]

Return type

[type]

emobpy.functions.**p_generatorout**(*p_generator_in*, *generator_eff*)

#TODO DOCSTRING

Parameters

- **p_generator_in** (*[type]*) – [description]
- **generator_eff** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**p_gravity**(*vehicle_mass*, *g*, *v*, *slop_angle=0*)

#TODO DOCSTRING

Parameters

- **vehicle_mass** (*[type]*) – [description]
- **g** (*[type]*) – [description]
- **v** (*[type]*) – [description]
- **slop_angle** (*int, optional*) – [description]. Defaults to 0.

Returns
[description]

Return type
[type]

emobpy.functions.**p_motorin**(*p_motor_out*, *motor_eff*)

#TODO DOCSTRING

Parameters

- **p_motor_out** (*[type]*) – [description]
- **motor_eff** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**p_motorout**(*p_wheel*, *transmission_eff*)

#TODO DOCSTRING

Parameters

- **p_wheel** (*[type]*) – [description]
- **transmission_eff** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**p_wheel**(*p_rollingresistance*, *p_airdrag*, *p_gravity*, *p_inertia*)

#TODO DOCSTRING

Parameters

- **p_rollingresistance** (*[type]*) – [description]
- **p_airdrag** (*[type]*) – [description]
- **p_gravity** (*[type]*) – [description]
- **p_inertia** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**pairdrag**(*air_density*, *frontal_area*, *drag_coeff*, *v*, *wind_speed=0*)

#TODO DOCSTRING Reference: Wang, J.; Besselink, I.; Nijmeijer, H. Electric Vehicle Energy Consumption Modelling and Prediction Based on Road Information. World Electr. Veh. J. 2015, 7, 447-458. https://doi.org/10.3390/wevj7030447

Parameters

- **air_density** (*[type]*) – [description]
- **frontal_area** (*[type]*) – [description]
- **drag_coeff** (*[type]*) – [description]
- **v** (*[type]*) – [description]
- **wind_speed** (*int, optional*) – Wind speed in direction of the vehicle.. Defaults to 0.

Returns
[description]

Return type
float

emobpy.functions.**pinertia**(*inertial_mass*, *vehicle_mass*, *acceleration*, *v*)

#TODO DOCSTRING

Parameters

- **inertial_mass** (*[type]*) – [description]
- **vehicle_mass** (*[type]*) – [description]
- **acceleration** (*[type]*) – [description]
- **v** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**plot_multi**(*data*, *cols=None*, *spacing=0.1*, ***kwargs*)

#TODO DOCSTRING

Parameters

- **data** (*[type]*) – [description]
- **cols** (*[type], optional*) – [description]. Defaults to None.
- **spacing** (*float, optional*) – [description]. Defaults to .1.

Returns
[description]

Return type
[type]

emobpy.functions.**prollingresistance**(*rolling_resistance_coeff*, *vehicle_mass*, *g*, *v*, *slop_angle=0*)

Calculates and returns polling resistance.

#TODO DOCSTRING :Parameters: * **rolling_resistance_coeff** (*[type]*) – [description]

- **vehicle_mass** (*[type]*) – [description]
- **g** (*[type]*) – [description]
- **v** (*[type]*) – [description]
- **slop_angle** (*int, optional*) – [description]. Defaults to 0.

Returns
Polling resistance.

Return type
float

emobpy.functions.**q_person**(*q_sensible*, *persons=1*)

#TODO DOCSTRING

Parameters

- **q_sensible** (*[type]*) – [description]
- **persons** (*int, optional*) – [description]. Defaults to 1.

Returns
[description]

Return type
[type]

emobpy.functions.**q_transfer**(*zone_layer*, *zone_area*, *layer_conductivity*, *layer_thickness*, *t_air_cabin*, *t_air_out*, *vehicle_speed*, *air_cabin_heat_transfer_coef=10*)

#TODO DOCSTRING

Parameters

- **zone_layer** (*[type]*) – [description]
- **zone_area** (*[type]*) – [description]
- **layer_conductivity** (*[type]*) – [description]
- **layer_thickness** (*[type]*) – [description]

- **t_air_cabin** (*[type]*) – [description]
- **t_air_out** (*[type]*) – [description]
- **vehicle_speed** (*[type]*) – [description]
- **air_cabin_heat_transfer_coef** (*int, optional*) – [description]. Defaults to 10.

Returns
[description]

Return type
[type]

emobpy.functions.**q_ventilation**(*density_air*, *flow_air*, *Cp_air*, *temp_air*)

#TODO DOCSTRING

Density_air: kg/m3, Flow_air: m3/s, Cp_air: J/(kg*K), Temp_air: degC :Parameters: * **density_air** (*[type]*) – [description]

- **flow_air** (*[type]*) – [description]
- **Cp_air** (*[type]*) – [description]
- **temp_air** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**qhvac**(*D*, *T_out*, *T_targ*, *cabin_volume*, *flow_air*, *zone_layer*, *zone_area*, *layer_conductivity*, *layer_thickness*, *vehicle_speed*, *Q_sensible=70*, *persons=1*, *P_out=1013.25*, *h_out=60*, *air_cabin_heat_transfer_coef=10*)

#TODO DOCUMENTATION Q indexes 0: Qtotal, 1: Q_in_per, 2: Q_in_vent, 3: Q_out_vent, 4: Q_tr

Parameters

- **D** (*method*) – [description]
- **T_out** (*float*) – [description]
- **T_targ** (*int*) – [description]
- **cabin_volume** (*float*) – [description]
- **flow_air** (*float*) – [description]
- **zone_layer** (*ndarray*) – [description]
- **zone_area** (*ndarray*) – [description]
- **layer_conductivity** (*ndarray*) – [description]
- **layer_thickness** (*ndarray*) – [description]
- **vehicle_speed** (*ndarray*) – [description]
- **Q_sensible** (*int, optional*) – [description]. Defaults to 70.
- **persons** (*int, optional*) – [description]. Defaults to 1.
- **P_out** (*float, optional*) – [description]. Defaults to 1013.25.
- **h_out** (*int, optional*) – [description]. Defaults to 60.

- **air_cabin_heat_transfer_coef** (*int, optional*) – [description]. Defaults to 10.

Returns
[description]

Return type
[type]

emobpy.functions.**resistances**(*zone_layer*, *zone_area*, *layer_conductivity*, *layer_thickness*, *vehicle_speed*, *air_cabin_heat_transfer_coef*)

[summary] #TODO DOCSTRING

Parameters

- **zone_layer** (*[type]*) – [description]
- **zone_area** (*[type]*) – [description]
- **layer_conductivity** (*[type]*) – [description]
- **layer_thickness** (*[type]*) – [description]
- **vehicle_speed** (*[type]*) – [description]
- **air_cabin_heat_transfer_coef** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.functions.**rolling_resistance_coeff**(*method='M1'*, ***kwargs*)

Returns calculated rolling resistance coeff depending on method.

M1 depends on v:velocity (km/h), temp: degC and road_type: int -> Wang et al. M2 depends on v:velocity (km/h), tire_type: int, road_type: int -> Rahka et al.

M1 options:
v: km/h temp: degC road_type

0: ordinary car tires on concrete, new asphalt, cobbles small new, coeff: 0.01 - 0.015 1: car tires on gravel - rolled new, on tar or asphalt, coeff: 0.02 2: car tires on cobbles - large worn, coeff: 0.03 3: car tire on solid sand, gravel loose worn, soil medium hard, coeff: 0.04 - 0.08 4: car tire on loose sand, coeff: 0.2 - 0.4

M2 options:
v: km/h tire_type

0: Radial, c2:0.0328, c3: 4.575 1: Bias ply, c2:0.0438, c3: 6.100

road_type
Concrete: excellent 0: 1.00, good 1: 1.50, poor 2: 2.00 Asphalt: good 3: 1.25, fair 4: 1.75, poor 5: 2.25 Macadam: good 6: 1.50, fair 7: 2.25, poor 8: 3.75 Cobbles: ordinary 9: 5.50, poor 10: 8.50 Snow: 2 inch 11: 2.50, 4 inch 12: 3.75 Dirt: Smooth 13: 2.50, sandy 14: 3.75 Sand not implemented range 6.00 - 30.00

Parameters
method (*str, optional*) – [description]. Defaults to 'M1'.

Raises
Exception – Raised if Method is not M1 or M2.

Returns
[description]

Return type
[type]

emobpy.functions.**rolling_resistance_coeff_M1**(*temp*, *v*, *road_type=0*)

Returns calculated rolling resistance coeff for M1.

Parameters

- **temp** (*float*) – Temperature ein degree celsius.
- **v** (*float*) – Speed in km/h.
- **road_type** (*int, optional*) **– 0: ordinary car tires on concrete, new asphalt, cobbles small new, coeff: 0.01 - 0.015 (Default)**
  1: car tires on gravel - rolled new, on tar or asphalt, coeff: 0.02 2: car tires on cobbles - large worn, coeff: 0.03 3: car tire on solid sand, gravel loose worn, soil medium hard, coeff: 0.04 - 0.08 4: car tire on loose sand, coeff: 0.2 - 0.4

  reference: Wang, J.; Besselink, I.; Nijmeijer, H. Electric Vehicle Energy Consumption Modelling and Prediction Based on Road Information. World Electr. Veh. J. 2015, 7, 447-458. https://doi.org/10.3390/wevj7030447

Returns
Rolling resistance coefficient

Return type
int

emobpy.functions.**rolling_resistance_coeff_M2**(*v*, *tire_type=0*, *road_type=4*)

Returns calculated rolling resistance coeff for M2.

Parameters

- **v** (*float*) – Speed in km/h.
- **tire_type** (*int, optional*) – 0: Radial, c2:0.0328, c3: 4.575 1: Bias ply, c2:0.0438, c3: 6.100 (Default)
- **road_type** (*int, optional*) – [description]. Defaults to 4. Concrete: excellent 0: 1.00, good 1: 1.50, poor 2: 2.00 Asphalt: good 3: 1.25, fair 4: 1.75, poor 5: 2.25 Macadam: good 6: 1.50, fair 7: 2.25, poor 8: 3.75 Cobbles: ordinary 9: 5.50, poor 10: 8.50 Snow: 2 inch 11: 2.50, 4 inch 12: 3.75 Dirt: Smooth 13: 2.50, sandy 14: 3.75 Sand not implemented range 6.00 - 30.00 reference: Rahka et al. 2001. Vehicle Dynamics Model for Predicting Maximum Truck Acceleration Levels. https://doi.org/10.1061/(ASCE)0733-947X(2001)127:5(418)

Returns
Rolling resistance coefficient

Return type
int

emobpy.functions.**vehicle_mass**(*curb_weight*, *passengers_weight*)

Calculates and returns vehicle mass.

Parameters

- **curb_weight** (*float*) – Curb weight of the vehicle.
- **passengers_weight** (*float*) – Passengers weight.

Returns
Vehicle mass.

Return type
float

emobpy.init module

emobpy.init.**copy_to_user_data_dir**()

This function combines the generation of user data directory and marges the user-defined data This usually runs when we create a project folder from command line. If we install emobpy and do not run the funtion create_project then it is likely that 'emobpy user data folder' has not been created and an error will ocour when using Consumption class. Check out the folder in # linux: ~/.local/share/emobpy # Windows: C:/Users/<USER>/AppData/Roaming/emobpy if embopy user data folder does not exist then run this function: `from emobpy.init import copy_to_user_data_dir; copy_to_user_data_dir()`

emobpy.init.**create_project**(*project_name*, *template*)

Creates project based on selected template and copies these files.

Parameters

- **project_name** (*str*) – Chosen project name.
- **template** (*str*) – Chosen template.

Raises

- **Exception** – Template arguments not valid.
- **Exception** – Chosen folder does not exist.

emobpy.mobility module

Mobility class consists of creating individual driver time series that contains vehicle location and distance travelled at every time step. The vehicle's locations are a group of common places obtained from mobility data.

Our approach consists of using three probability distributions from which to extract relevant features by samplings, such as the number of trips per day, the destination, departure time, trip distance and trip duration. Time steps and total time for every time series are parameters that have to be provided in advance.

To reach consistent and feasible mobility patterns, a set of rules can also be provided, for instance, establishing home as the destination of the last trip of the day.

Two different approaches have been developed—commuters and free-time drivers (non-commuters). Commuters are drivers that go during the weekdays to the workplace.

An instance of a mobility class contains a driving profile and a time series of one driver. The instance can be saved in a pickle file to access it later on and create other time series, such as consumption time series, grid availability, and eventually the grid demand.

For more details see the article and cite:

```
@article{
Gaete-Morales2021,
author={Gaete-Morales, Carlos and Kramer, Hendrik and Schill, Wolf-Peter and Zerrahn,␣
↪Alexander},
title={An open tool for creating battery-electric vehicle time series from empirical␣
↪data, emobpy},
```

(continues on next page)

(continued from previous page)

```
journal={Scientific Data},
year={2021},
month={Jun},
day={11},
volume={8},
number={1},
pages={152},
abstract={There is substantial research interest in how future fleets of battery-
↪electric vehicles will interact with the power sector. Various types of energy␣
↪models are used for respective analyses. They depend on meaningful input parameters,␣
↪in particular time series of vehicle mobility, driving electricity consumption, grid␣
↪availability, or grid electricity demand. As the availability of such data is highly␣
↪limited, we introduce the open-source tool emobpy. Based on mobility statistics,␣
↪physical properties of battery-electric vehicles, and other customizable assumptions,␣
↪it derives time series data that can readily be used in a wide range of model␣
↪applications. For an illustration, we create and characterize 200 vehicle profiles for␣
↪Germany. Depending on the hour of the day, a fleet of one million vehicles has a␣
↪median grid availability between 5 and 7 gigawatts, as vehicles are parking most of␣
↪the time. Four exemplary grid electricity demand time series illustrate the smoothing␣
↪effect of balanced charging strategies.},
issn={2052-4463},
doi={10.1038/s41597-021-00932-9},
url={https://doi.org/10.1038/s41597-021-00932-9}
}
```

class emobpy.mobility.**Mobility**(*config_folder='config_files'*)

Bases: `object`

To create a mobility time series, after creating the Mobility class instance call the methods in the following order:

- self.set_params(param)
- self.set_stats(stat_ntrip, stat_dest, stat_km)
- self.set_rules(rules_key, rules_file)
- self.run()
- self.save_profile(destination_folder)

Parameters

config_folder (*string*) – Folder name where the configuration files are hosted. Configuration files names are specified when setting stats and rules.

run()

This function returns a driving profile. No input possible. Returns nothing. Once it finishes the following attributes can be called:

- kind
- name_prefix
- refdate
- hours
- t

- user_defined
- df1
- df2
- df3
- user_rules
- states
- numb_weeks
- totalrows
- name
- profile
- timeseries

`save_profile`(*folder*, *description=' '*)

Saves a profile from the current object as a pickle file.

Parameters

- **folder** (*str*) – Where the files will be stored. Folder is created in case it does not exist.
- **description** (*str, optional*) – Profile description. Defaults to " ".

`set_params`(*name_prefix=''*, *total_hours=168*, *time_step_in_hrs=0.5*, *category='user_defined'*, *reference_date='01/01/2020'*)

Defines the attributes given to the class objects.

Parameters

- **name_prefix** (*str*) – It is an identifier that takes part of the file name. The user can freely choose it, e.g. 'worker'. Defaults to ''
- **total_hours** (*int*) – Defaults 168 hrs (one week). Maximum value 8760.
- **time_step_in_hrs** (*float*) – It is the time step in hours. Defaults to 0.5 . Options are 0.25, 0.5, 1. Recommennded values below 1.
- **category** (*str*) – This is an identifier. A column in the time series will contain this string e.g. 'worker'. Defauts to 'user_defined'.
- **reference_date** (*str, optional*) – With this date the time series will start. All consecutive time steps will follow this date, e.g. '01/01/2020'. Defaults to "01/01/2020".

`set_rules`(*rule_key=None*, *rules_path='rules.yml'*)

A set of rules can be defined and provided in a YAML file. Several customized rules can be allocated in only one file. Hence, the rule_key points to the set of rules that wanted to be used in the current run. If no rule_key is provided, then the tool copy a rules dictionary with default values.

If you want to know all the possibles rules to create your own rules file afterwards, type .initial_conf() and then .rules, e.g. Mobility.rules

Parameters

- **rule_key** (*str, optional*) – e.g. 'user_defined'. Defaults to None.
- **rules_path** (*str, optional*) – Path to rules file. Defaults to "rules.yml" in config_files folder.

Raises

`Exception` – Raised if .setStats(args) is not called before.

set_stats(*stat_ntrip_path*, *stat_dest_path*, *stat_km_duration_path*)

Name of files that contain the required probabilities to generate a mobility time series. They are three: DepartureDestinationTrip, DistanceDurationTrip, and TripsPerDay.

The DepartureDestinationTrip file must contain a wide data format (https://en.wikipedia.org/wiki/Wide_and_narrow_data). The first two columns are days and departure time in hours. The following columns represent the expected destinations where 'home' must be one of the destinations. At least 2 destinations (columns) should be included. As emobpy takes into account weekend days as well, the column days must contain 'weekdays', 'sunday' and 'saturday'. Suppose your data does not differentiate from these days. Repeat the probabilities in each of the three options.

The DistanceDurationTrip file must contain a wide data format. The first column is distance in kilometers. The following columns are the duration of the trip in minutes. The tool determines the distance of trips and the duration. Then calculates the average velocity. Make sure that all possible combinations in your joint probabilities lead to consistent average velocities.

The TripsPerDay file must contain a wide data format. The first column is 'trip' and represnts the number of trips per day. The column weekday and weekend contain the probability distribution. Bear in mind that zero trip per day represent not trips at all, two trips means a trip to some destination and then a retun trip. One trip is not possible, and considered a limitation of the tool, as the model needs to starts the following day from home again. Always set 1 trip with zero as probability value. Also the maximun number of trip per day is limited with the time steps selected. If the time step is one hour, then having a distribution with 8 trips per day could be challenging for the model, and it might hang off.

Parameters

- **stat_ntrip_path** (*str*) – e.g. 'DepartureDestinationTrip.csv'
- **stat_dest_path** (*str*) – e.g. 'DistanceDurationTrip.csv'
- **stat_km_duration_path** (*str*) – 'TripsPerDay.csv

Raises

Exception – Raised, if .set_params(args) is not called before.

emobpy.mobility.**add_column_datetime**(*df*, *totalrows*, *reference_date*, *t*)

Useful to convert the time series from hours index to datetime index.

Parameters

- **df** (*pd.DataFrame*) – Table on which datetime column should be added.
- **totalrows** (*int*) – Number of rows on which datetime column should be added.
- **reference_date** (*str*) – Starting date for adding. E.g. '01/01/2020'.
- **t** (*float*) – Float frequency, will be changed to string.

Returns

Table with added datetime column.

Return type

pd.DataFrame

emobpy.mobility.**bar_progress**(*current*, *total*)

Prints actual progress in format: "Progress: 80% [8 / 10] days".

Parameters

- **current** (*int*) – Current day.
- **total** (*int*) – Total number of days.

emobpy.mobility.**cmp**(*arg1*, *string_operator*, *arg2*)

Perform comparison operation according to operator module. This function is used on meet_all_conditions().

Parameters

- **arg1** (*object*) – First argument.
- **string_operator** (*'<', '<=', '==', '!=', '>=', '>'*) – String operator which can be used.
- **arg2** (*object*) – Second Argument.

Returns
Result of comparison.

Return type
bool

emobpy.mobility.**create_tour_nb**(*unsortedtrips*, *previous_dest*, *lasttriparrival*, *t*)

#TODO DOCSTRING Create tour tuple .

Parameters

- **unsortedtrips** (*[type]*) – [description]
- **previous_dest** (*[type]*) – [description]
- **lasttriparrival** (*[type]*) – [description]
- **t** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.mobility.**creation_unsorted_trips_nb**(*Rnd_week_trips*, *km_array*, *duration_array*, *km_dur_prob_array*, *km_sorted*, *duration_sorted*, *t*, *time_purpose_array*, *daycode*)

#TODO DOCSTRING It returns a unsorted time dataframe with trips. It includes number of trips, departure time, destination (purpose).

Parameters

- **Rnd_week_trips** (*[type]*) – [description]
- **km_array** (*[type]*) – [description]
- **duration_array** (*[type]*) – [description]
- **km_dur_prob_array** (*[type]*) – [description]
- **km_sorted** (*[type]*) – [description]
- **duration_sorted** (*[type]*) – [description]
- **t** (*[type]*) – [description]
- **time_purpose_array** (*[type]*) – [description]
- **daycode** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.mobility.**days_week_sequence**(*reference_date*, *weeks*)

This determines the day of the week of the reference date and sets the day-sequence. It helps to differentiate the statistics from week working days to weekends.

Parameters

- **reference_date** (*str*) – Reference date in form of '01/01/2020'
- **weeks** (*dict*) – Contains tag, tag type and day code per day ID.

Returns
List of day IDs (Integers) indicating the sequence of the week.

Return type
list

emobpy.mobility.**distance_duration_sample**(*km_array_*, *duration_array_*, *km_dur_prob_array*, *km_sorted_*, *duration_sorted*)

#TODO DOCSTRING Randomly choose a distance duration based on distance and probability distribution. :Parameters: * **km_array_** (*[type]*) – [description]

- **duration_array_** (*[type]*) – [description]
- **km_dur_prob_array** (*[type]*) – [description]
- **km_sorted_** (*[type]*) – [description]
- **duration_sorted** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.mobility.**distances_collection**(*nr_trips*, *km_array*, *duration_array*, *km_dur_prob_array*, *km_sorted*, *duration_sorted*, *edge_dist=- 1*, *edge_dur=- 1*, *edge_nr=0*)

#TODO DOCSTRING Sample distances from nba - grams.

Parameters

- **nr_trips** (*[type]*) – [description]
- **km_array** (*[type]*) – [description]
- **duration_array** (*[type]*) – [description]
- **km_dur_prob_array** (*[type]*) – [description]
- **km_sorted** (*[type]*) – [description]
- **duration_sorted** (*[type]*) – [description]
- **edge_dist** (*int, optional*) – [description]. Defaults to -1.
- **edge_dur** (*int, optional*) – [description]. Defaults to -1.
- **edge_nr** (*int, optional*) – [description]. Defaults to 0.

Returns
[description]

Return type
[type]

emobpy.mobility.**expand_table**(*df*, *factor*, *time_step*)

Expands a table (DataFrame) by a factor. The table can then be more granular. :Parameters: * **df** (*[pd.DataFrame]*) – Table that is to be extended.

- **factor** (*float*) – By how many times the table should be extended.
- **time_step** (*float*) – The new time_step that should apply.

Returns
Table in an expanded format.

Return type
[pd.DataFrame]

emobpy.mobility.**make_interval**(*x*, *factor*, *time_step*)

Make a new interval for the given factor.

Parameters

- **x** (*float*) – Start value.
- **factor** (*int*) – Number of repetition.
- **time_step** (*float*) – Time step for each repetition.

Returns
[description]

Return type
[type]

emobpy.mobility.**make_list**(*x*, *factor*)

Make a list of integers representing a number x .

Parameters

- **x** (*float*) – Number which is repeated in the list.
- **factor** (*int*) – Len of the produced list.

Returns
List which contains value x for factor times.

Return type
list

emobpy.mobility.**nb_isin**(*A*, *C*)

#TODO DOCSTRING Return nb in - place of nb in c .

Parameters

- **A** (*[type]*) – [description]
- **C** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.mobility.**rand_choice_nb**(*arr*, *prob*)

Choose random sample from numpy array with a explicit probability.

Parameters

- **arr** (*numpy.array*) – A 1D numpy array of values to sample from.
- **prob** (*numpy.array*) – A 1D numpy array of probabilities for the given samples.

Returns
A random sample from the given array with a given probability.

Return type
int or float

emobpy.mobility.**time_purpose_iter**(*array_tp*, *daycode*, *deltime*)

#TODO DOCSTRING Return a random value for time purpose

Parameters

- **array_tp** (*[type]*) – [description]
- **daycode** (*[type]*) – [description]
- **deltime** (*[type]*) – [description]

Returns
[description]

Return type
[type]

emobpy.plot module

This module contains a class that can be used to visualise the data. There are different visualisation functions.

class emobpy.plot.**NBplot**(*db*)

Bases: object

Work in Jupyter notebooks only. Set of plots for a single time series and groups. Three kind of plots:

- self.sgplot_dp(tscode) for driving profiles
- self.sgplot_ga(tscode) for grid availability profiles
- self.sgplot_ged(tscode) for grid electricity demand profiles
- tscode: time series code (string of profile name)

self.__init__(db)
db is an instance of a DataBase class that contains the time series.

overview(*tscode*, *date_range=None*, *to_html=False*, *path=None*)

Still in Development.

Do not use!

date_range=[‘01/01/2020 00:00:00’,’01/06/2020 23:00:00’]

`sankey`(*tscode*, *include=None*, *to_html=False*, *path=None*)

Plot of sankey diagram which shows the energy consumption flows.

Parameters

- **tscode** (*str*) – Time series code. Eg. 'ev_user_W3_85e59_avai_65t2p'
- **include** (*int*) – Index which part to include. Defaults to None.
- **to_html** (*bool*) – Save as a html file. Defaults to False.
- **path** (*str*) – Path if plot should be saved to file. Defaults to None.

Returns
Plot object.

Return type
plotly.plot

`sgplot_dp`(*tscode*, *rng=None*, *to_html=False*, *path=None*)

Plot of a single driving profile.

Parameters

- **tscode** (*str*) – Time series code. Eg. 'ev_user_W3_85e59_avai_65t2p'
- **rng** (*tuple*) – (a,b) index if only part of timeseries should be copied. Defaults to None.
- **to_html** (*bool*) – Save as a html file. Defaults to False.
- **path** (*str*) – Path if plot should be saved to file. Defaults to None.

Returns
Plot object.

Return type
plotly.plot

`sgplot_ga`(*tscode*, *rng=None*, *to_html=False*, *path=None*)

Plot of a single grid availability profile.

Parameters

- **tscode** (*str*) – Time series code. Eg. 'ev_user_W3_85e59_avai_65t2p'
- **rng** (*tuple*) – (a,b) index if only part of timeseries should be copied. Defaults to None.
- **to_html** (*bool*) – Save as a html file. Defaults to False.
- **path** (*str*) – Path if plot should be saved to file. Defaults to None.

Returns
Plot object.

Return type
plotly.plot

`sgplot_ged`(*tscode*, *rng=None*, *to_html=False*, *path=None*)

Plot of grid electricity demand profiles associated with the same grid availability profile.

Parameters

- **tscode** (*str*) – Time series code. Eg. 'ev_user_W3_85e59_avai_65t2p'
- **rng** (*tuple*) – (a,b) index if only part of timeseries should be copied. Defaults to None.
- **to_html** (*bool*) – Save as a html file. Defaults to False.

- **path** (*str*) – Path if plot should be saved to file. Defaults to None.

Returns
Plot object.

Return type
plotly.plot

emobpy.tools module

This module contains a tool functions used to create a new project.

class emobpy.tools.**Unit**(*val*, *unit*, *description='No description given.'*)

Bases: object

This class represents a unit. In this way a value and a unit can be saved and with class methods the unit can easily be changed. The default value used by the programme is always the 1 value in unit_scale.

convert_to(*new_unit*, *msg=False*)

convert_to_default_value()

info()

emobpy.tools.**check_for_new_function_name**(*attribute_error_name*)

In an earlier update function names have been changed from camelCase to snake_case. To prevent users confusing this function raises a specific AttributeError of the user trys to access to old function name, which does not exist anymore.

emobpy.tools.**parallel_func**(*dc*, *queue=None*, *queue_lock=None*, *function=None*, *kargs={}*)

#TODO DOCSTRING

Parameters

- **dc** (*[type]*) – [description]
- **queue** (*[type], optional*) – [description]. Defaults to None.
- **queue_lock** (*[type], optional*) – [description]. Defaults to None.
- **function** (*[type], optional*) – [description]. Defaults to None.
- **kargs** (*dict, optional*) – [description]. Defaults to {}.

Returns
[description]

Return type
[type]

emobpy.tools.**parallelize**(*function=None*, *inputdict: Optional[dict] = None*, *nr_workers=1*, ***kargs*)

Parallelize function to run program faster. The queue contains tuples of keys and objects, the function must be consistent when getting data from queue.

Parameters

- **function** (*function, optional*) – Function that is to be parallelized. Defaults to None.
- **inputdict** (*dict, optional*) – Contains numbered keys and as value any object. Defaults to None.
- **nr_workers** (*int, optional*) – Number of workers, so their tasks can run parallel. Defaults to 1.

Returns
Dictionary the given functions creates.

Return type
dict

emobpy.tools.**set_seed**()

Sets seed at the beginning of any python script or jupyter notebook. That allows to repeat the same calculations with exactly same results, without any random noise.

PYTHON MODULE INDEX

e

INDEX

Q

R

S

T

U

V

W