VencoPy

Release September 2021

German Aerospace Center (DLR)

Nov 12, 2021

GETTING STARTED

A data processing tool offering hourly demand and flexibility profiles for future electric vehicle fleets in an aggregated manner. VencoPy is developed at the Department of Energy Systems Analysis at the German Aerospace Center (DLR).

CHAPTER

ONE

ABOUT

- Authors: Niklas Wulff, Fabia Miorelli
- Contact: niklas.wulff@dlr.de

CHAPTER

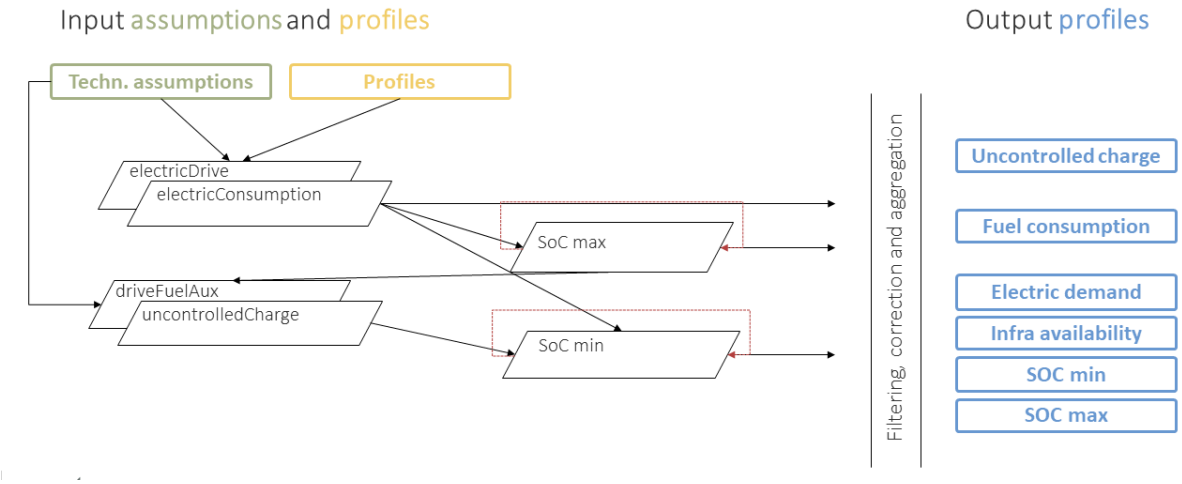
TWO

LINKS

- Source code: https://gitlab.com/dlr-ve/vencopy
- PyPI release: https://pypi.org/project/vencopy
- License: https://opensource.org/licenses/BSD-3-Clause

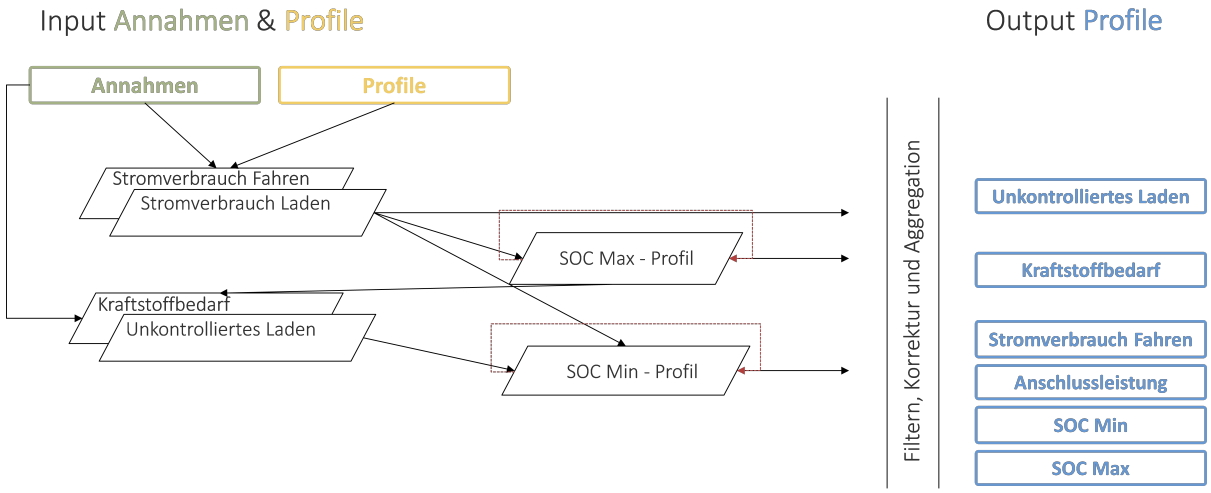
2.1 Introduction

Future electric vehicle fleets pose both challenges and opportunities for power systems. While increased power demand from transport electrification necessitates expansion of power supply, vehicle batteries can to a certain degree shift their charging load to times of high availability of power. The model-adequate description of power demand from future plug-in electric vehicle fleets is a pre-requisite for modelling sector-coupled energy systems and drawing respective policy-relevant conclusions. Vehicle Energy Consumption in Python (VencoPy) is a tool that provides boundary conditions for load shifting and vehicle-to-grid potentials based on transport demand data and techno-economic assumptions. It has so far been applied to the German travel survey (Mobilität in Deutschland) on a national scale to derive hourly load-shifting constraining profiles for the energy system optimization model REMix.



Zukünftige Flotten elektrisch angetriebener Fahrzeuge sind sowohl Chancen als auch Risiken für Energiesysteme. Auf der einen Seite erfordert eine Elektrifizierung des Verkehrs einen ausgeweiteten Kraftwerkspark, auf der anderen Seite können Batterien in gewissen Grenzen eine Flexibilitätsoption für das Stromsystem darstellen. Die angemessene Beschreibung der Stromnachfrage zukünftiger elektrischer Plug-in-Fahrzeugflotten ist eine Voraussetzung für die Modellierung sektorgekoppelter Energiesysteme und die Bereitstellung politikrelevanter Erkenntnisse. Vehicle Energy Consumption in Python (VencoPy) ist ein Tool, das Randbedingungen für das Ladeverhalten und möglicher Vehicle-to-Grid Potenziale basierend auf Mobilitätsdaten und techno-ökonomischen Annahmen berechnet. Es wurde bisher auf die

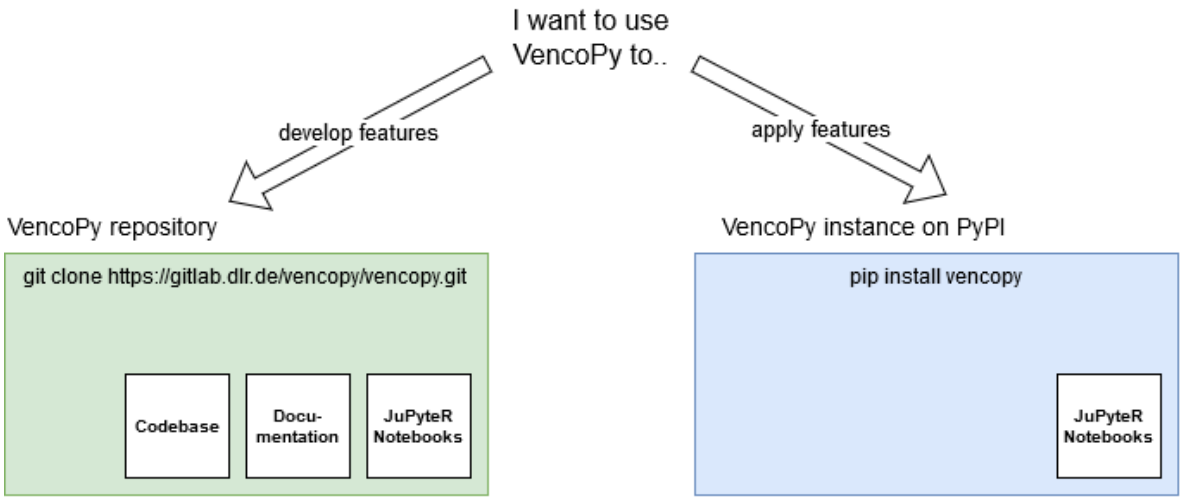
deutsche Verkehrserhebung “Mobilität in Deutschland” angewendet, um den Einfluss des Nutzerverhaltens auf das zukünftige Lastverschiebepotenzial und dessen Auswirkungen auf das deutsche Stromsystem zu untersuchen.



2.2 Installation and Setup

2.2.1 Requirements and boundary conditions

VencoPy runs on Unix and Windows-based operating systems. It requires an installed version of python and the package, dependency and environment management tool conda as well as access to the internet for setting up the environment (downloading the required packages). VencoPy is consistent with the software application class 1 of the DLR software categorization. Versioning is based on major, minor and fix (X.Y.Z) changes versioning system via git-labels. You can use VencoPy as a user or contribute to the codebase and documentation as developer. Depending on that choice the installation and setup differs.



- Parse any mobility or EV data set with my own routines
- Make use of VencoPy class object functionalities
- Contribute to the codebase
- Contribute to the documentation

- Parse MiD data and use it for EV analyses
- Generate hourly EV mobility and parking profiles
- Assume charging column scenarios
- Calculate EV charging profiles with and without charging flexibility

```
vencopy (Name of Repository)
├── docs
│   └── ..
├── vencopy
│   ├── classes
│   ├── config
│   ├── output*
│   ├── scripts
│   └── tutorials
├── run.py
├── setup.py
└── ...
```

```
FOLDERNAME
├── config
│   ├── evaluatorConfig.yaml
│   ├── ..
├── output
│   ├── dataParser
│   ├── ..
├── tutorials
│   └── ..
└── run.py
```

* created on first run

2.2.2 Installation for users

As a user, you will apply VencoPy for answering analytical questions. Thus, you're mainly interested in applying VencoPy's built-in features and functions. On this level, you will not change the codebase within the VencoPy class objects - of course you can write your own data processing routines around those functions.

Install using the environment management system conda, open the conda console, create a new environment and activate it by typing:

```
conda create -n <your environment name> python=3.9
conda activate <your environment name>
```

Install VencoPy from the Python Package Index PyPI:

```
pip install vencopy
```

Navigate to a parent directory where you want to create your VencoPy user folder in and type:

```
vencopy
```

You will be prompted for a userfolder name, type it and hit enter. Your VencoPy user folder will now be created. It will look like this:

```
FOLDERNAME
├── config
│   ├── evaluatorConfig.yaml
│   ├── flexConfig.yaml
│   ├── globalConfig.yaml
│   ├── gridConfig.yaml
│   ├── localPathConfig.yaml
│   └── parseConfig.yaml
├── output
│   ├── tripConfig.yamldataParser
│   ├── evaluator
│   ├── flexEstimator
│   ├── gridModeler
│   └── tripDiaryBuilder
├── tutorials
│   └── ..
└── run.py
```

The configs in the config folder are the main interface between the user and the code. In order to learn more about them, check out our tutorials. For this you won't need any additional data.

To run VencoPy in full mode, you will need the data set Mobilität in Deutschland (German for mobility in Germany), you can request it here from the clearingboard transport: https://daten.clearingstelle-verkehr.de/order-form.html Currently, VencoPy is only tested with the B2 data set.

In your localPathConfig.yaml, please enter the path to your local MiD STATA folder, it will end on .../B2/STATA/. Now open your user folder in an IDE, configure your interpreter (environment) or type:

```
python run.py
```

and enjoy the tool!

2.2.3 Installation for developers

This part of the documentation holds a step-by-step installation guide for VencoPy.

1. Navigate to a folder to which you want to clone VencoPy. Clone the repository to your local machine using

```
git clone https://gitlab.com/dlr-ve/vencopy.git
```

2. Set-up your environment. For this, open a conda console, navigate to the folder of your VencoPy repo and enter the following command:

```
conda env create -f requirements.yml
conda activate VencoPy_env
```

3. Configure your config files if you want to use absolute links. This is only needed if you want to reference your own local data or want to post-process VencoPy results and write them to a model input folder somewhere on your drive. You will find your config file in your repo under /config/config.yaml Input filenames are set to the example files shipped with the repo. You may specify labels for file naming in the config under the key "labels".

4. You're now ready to run VencoPy for the first time by typing:

```
python run.py
```

5. Have fun calculating electric vehicle flexibility!

2.3 Getting Started and Tutorials

2.3.1 Tutorials overview

To get started with VencoPy a set of tutorials are provided for the user to learn about the different classes and how each of them can be customised. The tutorials are iPythonNotebooks to be opened with Jupyter Lab and can be found in the gitlab repository.

- Tutorial 1: Showcasing run.py
- Tutorial 2: The DataParser class
- Tutorial 3: The TripDiaryBuilder class (currently this tutorial is empty, it will be complemented following upcoming developments)
- Tutorial 4: The GridModeler class
- Tutorial 5: The FlexEstimator class

2.3.2 Tutorials setup

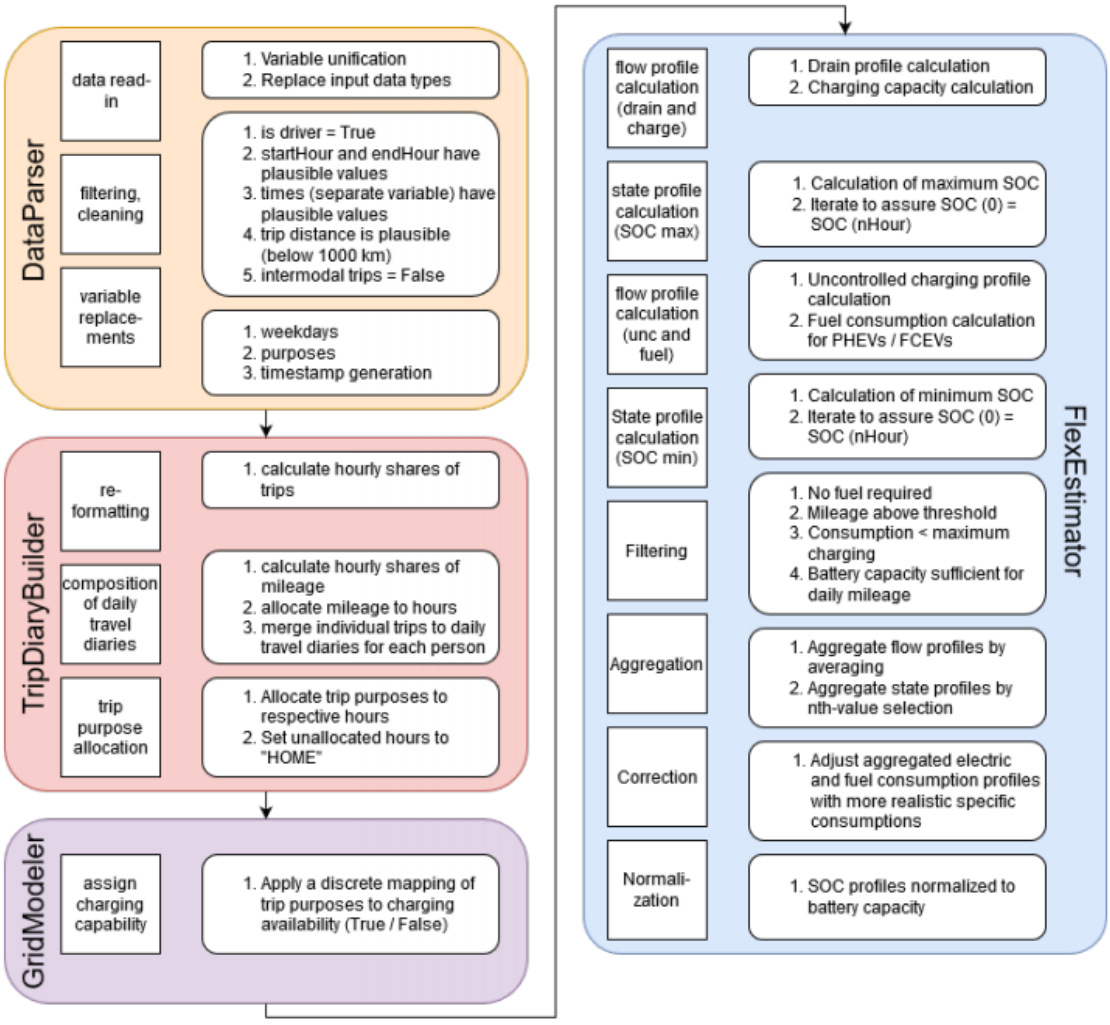
To start carrying out the tutorials, jupyterlab should be installed in the environment. If you cloned the repository doing 'pip install vencopy', the jupyterlab package will automatically be installed in your environment and you thus only need to activate your vencopy environment via the Anaconda Powershell Prompt ('conda activate <your environment name>'). If you want to run the tutorial in an environment, which already exists, you can first activate your desired environment ('conda activate <your environment name>') and then install the jupyterlab package ('conda install jupyterlab'). You might need to add the ipykernel to the environment to be able to run the jupyter notebooks with the tutorials. To do this type 'python -m ipykernel install –user –name=<your environment name>'. Now that the requirements are satisfied, you can either open the jupyter notebooks with the tutorials in an IDE that supports notebooks (e.g. VSCode) or open them in browser from the Anaconda Powershell Prompt ('jupyter lab –notebook-dir='<your local path to te

repository>' –browser=firefox'). Note: you might need to restart the jupyter notebooks kernel between the tutorials if you carry out multiple ones.

2.4 Architecture Documentation

2.4.1 General structure of the framework

The figure below shows the detailed VencoPy components in a developer diagram. The four components - implemented as Python classes - DataParser, TripDiaryBuilder, GridModeler and FlexEstimator can be clearly distinguished. A fifth component, Evaluator, used to create the plots for this work is not shown in the figure as it is not needed to calculate results from VencoPy. A brief description of the classes is presented below. For a more detailed algebraic description of the tool please refer to the *Publications* section.



2.4.2 Quality values

Table 1: Quality values

Value priority	Description
1. Learnability	The highest priority of VencoPy is to provide an easy-to-apply tool for scientists (not primarily developers) to estimate electric vehicle fleets' load shifting potential. Hence, easy to understand approaches, code structures and explicit formulations are favored.
2. Readability	The readability of the code, especially the linear flow structure of the main VencoPy file should be preserved. Source code and equations should be easy to read and understand. Splitting of statements is favored over convoluted one-liners. Significant learnability improvements (e.g. through an additional library), may motivate a deviation from this principle.
3. Reproducibility	The model has to be deterministic and reproducible both on the scientific and on the technical level. Hence, all example and test artifacts have to be part of the git repository. Source code revisions should be used to reference reproducible states of the source code.
4. Reliability	The software has to operate without crashes and should terminate early, if scientific requirements are not met. Testing and asserting expectations in code is encouraged. However, in its alpha-version no special error catching routines are implemented.
5.Performance	Performance is not a high priority, since runtimes are quite quick. However, basic performance considerations like efficient formulation and application of libraries and algorithms should be considered.

2.4.3 Organizational information

Table 2: requirements

Requirement	Context
Software Engineering Team (SET)	Niklas Wulff, Fabia Miorelli, Benjamin Fuchs
Stakeholders	Hans Christian Gils, Department of Energy Systems Analysis at Institute of Networked Energy Systems, DLR
Timeline	Alpha release in Q4 2020, Beta release in Q3 2021, for further releases planned, see *Release Timeline*
Open source ready	Features, dependencies and components which are contraindicative or at odds with an open source publication should not be used
Development tools	Source code and all artefacts are located in the DLR GITLAB repository for VencoPy including the software documentation. For development, the PyCharm community edition IDE and gitbash are used. For graphical depictions of software components and similar documentation draw.io is used.

2.5 VencoPy Classes

Below is a brief explanation of the four main VencoPy classes. For a more detailed explanation about the internal workings and the specific outputs of each function, you can click on the hyperlink on the function name.

2.5.1 Interface to the dataset: dataParser

The first step in the VencoPy framework for being able to estimate EV energy consumption implies accessing a travel survey data set, such as the MiD. This is carried out through a parsing interface to the original database. In the parsing interface to the data set, three main operations are carried out: the read-in of the travel survey trip data, stored in .dta or .csv files, filtering and cleaning of the original raw data set and a set of variable replacement operations to allow the composition of travel diaries in a second step. In order to have consistent entry data for all variables and for different data sets, all database entries are harmonised, which includes generating unified data types and consistent variable naming. The naming convention for the variables and their respective input type can be specified in the VencoPy-config file. Of the 22 variables, four variables are used for indexing, 11 variables characterize the trip time within the year, two variables are used for filtering and five variables characterize the trip itself. The representation of time may vary between travel surveys. Most travel surveys include motorised, non motorised as well as multi-modal trips. We only select trips that were carried out with a motorized individual vehicle as a driver. Similarly, trips with missing (e.g. missing tripID, missing start or end time etc.) or invalid information (e.g. implausible trip distance) are filtered out. Filters can be easily adapted to other travel survey numeric codes via the config-file. By applying a set of filters, the initial database is subset to only contain valid entries representing motorised trips. The last operation in the parsing of raw travel survey data sets is a harmonization step.

2.5.2 Daily travel diary composition: tripDiaryBuilder

In the second VencoPy class, the tripDiaryBuilder, individual trips at the survey day are consolidated into person-specific travel diaries comprising multiple trips (carried out by car). The daily travel diary composition consists of three main steps: Reformatting the database, allocating trip purposes and merging the obtained dataframe with other relevant variables from the original database. In the first step, reformatting, the time dimension is transferred from the raw data (usually in minutes) to the necessary output format (e.g. hours). Each trip is split into shares, which are then assigned to the respective hour in which they took place, generating an hourly dataframe with a timestamp instead of a dataframe containing single trip entries. Similarly, mileages driven and the trip purpose are allocated to their respective hour and merged into daily travel diaries. Trips are assumed to determine the respective person's stay in the consecutive hours up to the next trip and therefore are related to the charging availability between two trips. Trip purposes included in surveys may comprise trips carried out for work or education reasons, trips returning to home, trips to shopping facilities and other leisure activities. Currently, trips whose purpose is not specified are allocated to trips returning to the own household. At the end of the second VencoPy component TripDiaryBuilder, two intermediary data sets are available either directly from the class within Python or from the hard-drive as .csv files. The first one comprises mileage travel diaries d(v, t) and the second one comprises parking place types derived from trip purposes parkingType(v, t).

2.5.3 Charging infrastructure allocation: gridModeler

The charging infrastructure allocation makes use of a basic charging infrastructure model, which assumes the availability of charging stations when vehicles are parked. Since the analytical focus of the framework lies on a regional level (NUTS1-NUTS0), the infrastructure model is kept simple in the current version. Charging availability is allocated based on a binary TRUE/FALSE mapping to a respective trip purpose in the VencoPy-config. Thus, different scenarios describing different charging availabilities, e.g. at home or at home and at work etc. can be distinguished, but neither a regional differentiation nor a charging availability probability or distribution are assumed. At the end of the application of the GridModeler, a given parking purpose diary parkingType(v, t) is transferred into a binary grid connection diary connectgrid (v, t) with the same format but consisting only of TRUE/FALSE values.

2.5.4 Flexibility estimation: flexEstimator

There are three integral inputs to the estimation: 1. a profile describing hourly distances for each vehicle 2. a boolean set of profiles describing if a vehicle is connected to the grid at a given hour 3. techno-economic input assumptions After some filtering and iteration steps, this yields the minimum and maximum battery constraints. After these steps, six profiles are provided to the user: a battery drain profile (the electricity that flows out of the vehicle battery each hour for driving), a charging capacity profile (the maximum electricity available for charging in each hour), a minimum and a maximum SoC (upper and lower limits for the battery SoC), an uncontrolled charging profile (the electricity flow from grid to vehicle when no control is exerted) and a fuel consumption profile.

The first four profiles can be used as constraints for other models to determine optimal charging strategies, the fifth profile simulates a case, where charging is not controlled an EVs charge as soon as a charging possibility is available. Lastly, the sixth profile quantifies the demand for additional fuel for trips that cannot be only by electricity.

2.6 Functions

You can search for the different modules and funtions below:

- genindex
- modindex
- search

This part of the documentation holds the function definitions of VencoPy from the main run file ´run.py´ and the library files.

class vencopy.classes.dataParsers.**DataParser**(*configDict: dict*, *datasetID: str*, *loadEncrypted=False*)

Bases: object

addStrColumnFromVariable(*colName: str*, *varName: str*)

Replaces each occurence of a MiD/KiD variable e.g. 1,2,…,7 for weekdays with an explicitly mapped string e.g. 'MON', 'TUE',…,'SUN'.

Parameters

- **colName** – Name of the column in self.data where the explicit string info is stored
- **varName** – Name of the VencoPy internal variable given in config/parseConfig['dataVariables']

Returns None

addStrColumns(*weekday=True*, *purpose=True*)

Adds string columns for either weekday or purpose.

Parameters

- **weekday** – Boolean identifier if weekday string info should be added in a separate column
- **purpose** – Boolean identifier if purpose string info should be added in a separate column

Returns None

checkDatasetID(*datasetID: str*, *parseConfig: dict*) → str

General check if data set ID is defined in parseConfig.yaml

Parameters

- **datasetID** – list of strings declaring the datasetIDs to be read in

- **parseConfig** – A yaml config file holding a dictionary with the keys 'pathRelative' and 'pathAbsolute'

Returns Returns a string value of a mobility data

checkFilterDict()
Checking if all values of filter dictionaries are of type list. Currently only checking if list of list str not typechecked all(map(self.__checkStr, val). Conditionally triggers an assert.

Returns None

compileVariableList() → list
Clean up the replacement dictionary of raw data file variable (column) names. This has to be done because some variables that may be relevant for the analysis later on are only contained in one raw data set while not contained in another one. E.g. if a trip is an intermodal trip was only assessed in the MiD 2017 while it wasn't in the MiD 2008. This has to be mirrored by the filter dict for the respective data set.

Returns List of variables

composeStartAndEndTimestamps()

Returns Returns start and end time of a trip

composeTimestamp(*data: Optional[pandas.core.frame.DataFrame] = None*, *colYear: str = 'tripStartYear'*, *colWeek: str = 'tripStartWeek'*, *colDay: str = 'tripStartWeekday'*, *colHour: str = 'tripStartHour'*, *colMin: str = 'tripStartMinute'*, *colName: str = 'timestampStart'*) → numpy.datetime64

Parameters

- **data** – a data frame
- **colYear** – year of start of a particular trip
- **colWeek** – week of start of a particular trip
- **colDay** – weekday of start of a particular trip
- **colHour** – hour of start of a particular trip
- **colMin** – minute of start of a particular trip
- **colName** –

Returns Returns a detailed time stamp

convertTypes()
Convert raw column types to predefined python types as specified in parseConfig['inputDTypes'][datasetID]. This is mainly done for performance reasons. But also in order to avoid index values that are of type int to be cast to float. The function operates only on self.data and writes back changes to self.data

Returns None

createReplacementDict(*datasetID: str*, *dictRaw: dict*) → dict
Creates the mapping dictionary from raw data variable names to VencoPy internal variable names as specified in parseConfig.yaml for the specified data set.

Parameters

- **datasetID** – list of strings declaring the datasetIDs to be read in
- **dictRaw** – Contains dictionary of the raw data

Returns Dictionary with internal names as keys and raw data column names as values.

`filter()`
Wrapper function to carry out filtering for the four filter logics of including, excluding, greaterThan and smallerThan. If a filterDict is defined with a different key, a warning is thrown. The function operates on self.data class-internally.

Returns None

`filterAnalysis(filterData: pandas.core.frame.DataFrame)`
Function supplies some aggregate info of the data after filtering to the user Function does not change any class attributes

Parameters `filterData` –

Returns None

`filterConsistentHours()`
Filtering out records where starting hour is after end hour but trip takes place on the same day. These observations are data errors.

Returns No returns, operates only on the class instance

`harmonizeVariables()`
Harmonizes the input data variables to match internal VencoPy names given as specified in the mapping in parseConfig['dataVariables']. So far mappings for MiD08 and MiD17 are given. Since the MiD08 doesn't provide a combined household and person unique identifier, it is synthesized of the both IDs.

Returns None

`harmonizeVariablesGenericIdNames()`

`loadData()`
Loads data specified in self.rawDataPath and stores it in self.rawData. Raises an exception if a invalid suffix is specified in self.rawDataPath. READ IN OF CSV HAS NOT BEEN EXTENSIVELY TESTED BEFORE BETA RELEASE.

Returns None

`loadEncryptedData(pathToZip, pathInZip)`
Since the MiD data sets are only accessible by an extensive data security contract, VencoPy provides the possibility to access encrypted zip files. An encryption password has to be given in parseConfig.yaml in order to access the encrypted file. Loaded data is stored in self.rawData

Parameters

- `pathToZip` – path from current working directory to the zip file or absolute path to zipfile
- `pathInZip` – Path to trip data file within the encrypted zipfile

Returns None

`process()`
Wrapper function for harmonising and filtering the dataset.

`removeNA(variables: list)`
Removes all strings that can be capitalized to 'NA' from the list of variables

Parameters `variables` – List of variables of the mobility dataset

Returns Returns a list with non NA values

`returnDictBottomKeys(baseDict: dict, lst: Optional[list] = None)` → list
Returns the lowest level keys of baseDict and returns all of them as a list. The parameter lst is used as interface between recursion levels.

Parameters

- **baseDict** – Dictionary of variables
- **lst** – empty list, used as interface between recursion levels

Returns Returns a list with all the bottom level dictionary keys

returnDictBottomValues(*baseDict: dict*, *lst: list = []*) → list
Returns a list of all dictionary values of the last dictionary level (the bottom) of baseDict. The parameter lst is used as an interface between recursion levels.

Parameters

- **baseDict** – Dictionary of variables
- **lst** – empty list, is used as interface to next recursion

Returns Returns a list with all the bottom dictionary values

selectColumns()
Function to filter the rawData for only relevant columns as specified by parseConfig and cleaned in self.compileVariablesList(). Stores the subset of data in self.data

Returns None

setExcludeFilter(*excludeFilterDict: dict*, *dataIndex*) → pandas.core.frame.DataFrame
Read-in function for exclude filter dict from parseConfig.yaml

Parameters

- **excludeFilterDict** – Dictionary of exclude filters defined in parseConfig.yaml
- **dataIndex** – Index for the data frame

Returns Returns a filtered data frame with exclude filters

setGreaterThanFilter(*greaterThanFilterDict: dict*, *dataIndex*)
Read-in function for greaterThan filter dict from parseConfig.yaml

Parameters

- **greaterThanFilterDict** – Dictionary of greater than filters defined in parseConfig.yaml
- **dataIndex** – Index for the data frame

Returns

setIncludeFilter(*includeFilterDict: dict*, *dataIndex*) → pandas.core.frame.DataFrame
Read-in function for include filter dict from parseConfig.yaml

Parameters

- **includeFilterDict** – Dictionary of include filters defined in parseConfig.yaml
- **dataIndex** – Index for the data frame

Returns Returns a data frame with individuals using car as a mode of transport

setSmallerThanFilter(*smallerThanFilterDict: dict*, *dataIndex*) → pandas.core.frame.DataFrame
Read-in function for smallerThan filter dict from parseConfig.yaml

Parameters

- **smallerThanFilterDict** – Dictionary of smaller than filters defined in parseConfig.yaml
- **dataIndex** – Index for the data frame

Returns Returns a data frame of trips covering a distance of less than 1000 km

updateEndTimestamp()

Returns

updateEndTimestamps() → numpy.datetime64

Returns Returns start and end time of a trip

updateFilterDict() → None
Internal function to parse the filter dictionary of a specified data set from parseConfig.yaml

Returns None

class vencopy.classes.dataParsers.**ParseKiD**(*configDict: dict*, *datasetID: str*)
Bases: *vencopy.classes.dataParsers.DataParser*

addStrColumns(*weekday=True*, *purpose=True*)
Adds string columns for either weekday or purpose.

Parameters

- **weekday** – Boolean identifier if weekday string info should be added in a separate column
- **purpose** – Boolean identifier if purpose string info should be added in a separate column

Returns None

convertTypes()
Convert raw column types to predefined python types as specified in parseConfig['inputDTypes'][datasetID]. This is mainly done for performance reasons. But also in order to avoid index values that are of type int to be cast to float. The function operates only on self.data and writes back changes to self.data

Returns None

excludeHours()
Removes trips where both start and end trip time are missing

loadData()
Loads data specified in self.rawDataPath and stores it in self.rawData. Raises an exception if a invalid suffix is specified in self.rawDataPath. READ IN OF CSV HAS NOT BEEN EXTENSIVELY TESTED BEFORE BETA RELEASE.

Returns None

process()
Wrapper function for harmonising and filtering the dataset.

updateEndTimestamp()

Returns

class vencopy.classes.dataParsers.**ParseMiD**(*configDict: dict*, *datasetID: str*, *loadEncrypted=False*)
Bases: *vencopy.classes.dataParsers.DataParser*

class vencopy.classes.tripDiaryBuilders.**FillHourValues**(*data*, *rangeFunction*)
Bases: object

class vencopy.classes.tripDiaryBuilders.**TripDiaryBuilder**(*configDict: dict*, *ParseData*, *datasetID: str = 'MiD17'*, *debug: bool = False*)

Bases: object

assignDriving(*driveData: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Assign hours where driveData != 0/NA to 'driving'

Parameters **driveData** – driving data

Returns Returns driving data with 'driving' instead of hours having 0/NA

calcDistanceShares(*data: pandas.core.frame.DataFrame*, *duration: pandas.core.series.Series*, *timestampSt: str*, *timestampEn: str*) → tuple

Parameters

- **data** – list of strings declaring the datasetIDs to be read in
- **duration** – duration of a trip
- **timestampSt** – start time of a trip
- **timestampEn** – end time of a trip

Returns Return a data frame of distance covered by each trip in an hour or more

calcFullHourTripLength(*duration: pandas.core.series.Series*, *numberOfFullHours: pandas.core.series.Series*, *tripLength: pandas.core.series.Series*, *shareStartHour: pandas.core.series.Series*, *shareEndHour: pandas.core.series.Series*) → pandas.core.series.Series
Calculates the share of the full trip hours. E.g. the fullHourTripLength of a trip starting at 1:45 and ending at 4:30 is 120 minutes / 165 minutes ~ 0.73.

Parameters

- **duration** – Series holding durations of all trips
- **numberOfFullHours** – Series holding the number of full hours of all trips
- **tripLength** – Series of trip lengths

Returns Returns a Series of full hour trip lengths for all trips

calcHourShareEnd(*timestampEnd: pandas.core.series.Series*, *duration: pandas.core.series.Series*, *isSameHourTrip: pandas.core.series.Series*) → pandas.core.series.Series

Parameters

- **timestampEnd** – end time of a trip
- **duration** – duration of a trip
- **isSameHourTrip** – data frame containing same start time of various trips

Returns Returns a data frame of share of individual trip for trips completed in an hour and more w.r.t end time of the trip

calcHourShareStart(*timestampStart: pandas.core.series.Series*, *timestampEnd: pandas.core.series.Series*, *duration: pandas.core.series.Series*) → Tuple[pandas.core.series.Series, pandas.core.series.Series]

Parameters

- **timestampStart** – start time of a trip
- **timestampEnd** – end time of a trip
- **duration** – duration of a trip

Returns Returns a data frame of share of individual trip for trips completed in an hour and more w.r.t start time of the trip

calcHourlyShares(*data: pandas.core.frame.DataFrame*, *ts_st: str*, *ts_en: str*) → pandas.core.frame.DataFrame

Calculates the shares of first hour, the share of full hours and the full hour trip length.

Parameters

- **data** – Trip data
- **ts_st** – String specifying the trip start column
- **ts_en** – String specifying the trip end column

Returns data frame consisting additional information regarding share of a trip, number of full hours and lenght of each trip

calculateConsistentHourlyShares(*data: pandas.core.frame.DataFrame*)

Wrapper around calcHourlyShares also filtering inconsistent trips.

Parameters **data** – Trip data

Returns Filtered trip data

determinePurposeStartHour(*departure: numpy.datetime64*, *arrival: numpy.datetime64*) → int

Determines the start hour of a parking activity depending on previous trip end time and next trip start time

Parameters

- **departure** – Start time of next trip after parking
- **arrival** – End time of previous trip

Returns Returns start hour of a parking activity

fillDataframe(*hourlyArray: pandas.core.frame.DataFrame*, *fillFunction*) → pandas.core.frame.DataFrame

fillDayPurposes(*tripData: pandas.core.frame.DataFrame*, *purposeDataDays: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Main purpose diary builder function. Root of low performance of tripDiaryBuilder. Will be improved in future releases.

Parameters

- **tripData** – data frame holding all the information about individual trip
- **purposeDataDays** – DataFrame with 24 (hour) columns holding 0s and 'DRIVING' for trip hours (for hours where majority of time is driving)

Returns Returns a data frame of individual trip with it's hourly activity or parking purpose

initiateColRange(*row: pandas.core.series.Series*)

Returns a range object with start and end hour as limits describing only full hours e.g. for a trip from 18:30 to 21:20 the hours 19 and 20 (21 will be included in the range but disregarded in the following)

Parameters **row** – A trip observation

Returns range of full hours

initiateHourDataframe(*indexCol*, *nHours: int*) → pandas.core.frame.DataFrame
Sets up an empty dataframe to be filled with hourly data.

Parameters

- **indexCol** – List of column names
- **nHours** – integer giving the number of columns that should be added to the dataframe

Returns data frame with columns given and nHours additional columns appended with 0s

mergeTrips(*tripData: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Merge multiple individual hourly trip distances into one diary consisting of multiple trips

Parameters **tripData** – Input trip data with hourly distances of all hourly trips

Returns Merged trip distance diaries

numberOfFullHours(*timestampStart: pandas.core.series.Series*, *timestampEnd: pandas.core.series.Series*) → pandas.core.frame.DataFrame
Calculates the number of full hours in each trip. E.g. a trip taking 1:42 has 1 full hour and a 30-minute trip has 0 full hours.

Parameters

- **timestampStart** – Start timestamps of trips
- **timestampEnd** – End times of trips

Returns Returns a data frame of number of full hours of all trips

tripDistanceAllocation(*globalConfig: dict*) → pandas.core.frame.DataFrame
Wrapper function for the conversion of trip distance values (in a column) to hourly trip distance diaries.

Parameters **globalConfig** – Dictionary holding relative paths, filenames and run labels

Returns Trip distance diary as a pd.DataFrame

tripDuration(*timestampStart: numpy.datetime64*, *timestampEnd: numpy.datetime64*) → numpy.datetime64

tripPurposeAllocation()
Wrapper function for trip purpose allocation. Falsely non-allocated parking purposes are replaced by "HOME".

Returns None

writeOut(*globalConfig: dict*, *dataDrive: pandas.core.frame.DataFrame*, *dataPurpose: pandas.core.frame.DataFrame*, *datasetID: str = 'MiD17'*)
General writeout utility for tripDiaries

Parameters

- **globalConfig** – global config storing relative paths, filenames and run labels
- **dataDrive** – Driving distance diary for each survey participant
- **dataPurpose** – Parking purpose diary for each survey participant
- **datasetID** – ID used for filenames

Returns None

class vencopy.classes.gridModelers.**GridModeler**(*configDict: dict*, *datasetID: str*)
Bases: object

assignGridViaProbabilities(*model: str*, *fastChargingHHID*)
Not tested, preleiminary version

Parameters

- **model** – Input for assigning probability according to models presented in gridConfig
- **fastChargingHHID** – List of household trips for fast charging

Returns Returns a dataFrame holding charging capacity for each trip assigned with probability distribution

assignSimpleGridViaPurposes()
Method to translate hourly purpose profiles into hourly profiles of true/false giving the charging station availability in each hour for each individual vehicle.

Returns None

getFastChargingList()
Returns a list of household trips having consumption greater than 80% (40 kWh) of battery capacity (50 kWh)

getRandomNumberForModel1(*purpose*)
Not tested, preleiminary version

Assigns a random number between 0 and 1 for all the purposes, and allots a charging station according to the probability distribution :param purpose: Purpose of each hour of a trip :return: Returns a charging capacity for a purpose based on probability distribution 1

getRandomNumberForModel2(*purpose*)
Not tested, preleiminary version

Assigns a random number between 0 and 1 for all the purposes, and allots a charging station according to the probability distribution :param purpose: Purpose of each hour of a trip :return: Returns a charging capacity for a purpose based on probability distribution model 2

getRandomNumberForModel3(*purpose*)
Not tested, preleiminary version

Assigns a random number between 0 and 1 for all the purposes, and allots a charging station according to the probability distribution :param purpose: Purpose of each hour of a trip :return: Returns a charging capacity for a purpose based on probability distribution model 3 (fast charging)

writeOutGridAvailability()
Function to write out the boolean charging station availability for each vehicle in each hour to the output file path.

Returns None

class vencopy.classes.flexEstimators.**FlexEstimator**(*configDict: dict*, *ParseData*, *datasetID: str*)
Bases: object

aggregate()
Wrapper function to aggregate profiles from individual vehicle level to fleet level. This is done in two categories: Aggregating to one representative weekday with 24 hours and aggregating for a representative week. Within these categories mean and weighted mean values are calculated for comparison. Simple means are only calculated for 24 hour profiles. For state of charge profiles (soc max and soc min), simple estimations of minimum and maximum state-of-charge profiles are carried out. This is done based on https://elib.dlr.de/92151/1/Dissertation_Diego_Luca_de_Tena.pdf by selecting the nth maximum or minimum value. See https://doi.org/10.3390/en14144349 for further explanations.

aggregateDiffVariable(*data: pandas.core.frame.DataFrame*, *by: str*, *weights: pandas.core.series.Series*, *hourVec: list*) → pandas.core.series.Series
A separate weighted aggregation function differentiating by the variable defined as a string in str. Weights as given in MiD.

Parameters

- **data** – list of strings declaring the datasetIDs to be read in
- **by** – String specifyzing a variable.
- **weights** – Weight vector as given in the MiD
- **hourVec** – hour vector specifying the hours used for VencoPy analysis

Returns

aggregateProfilesMean(*profilesIn: pandas.core.frame.DataFrame*) → pandas.core.series.Series
This method aggregates all single-vehicle profiles that are considered to one fleet profile.

Parameters **profilesIn** – Dataframe of hourly values of all filtered profiles

Returns Returns a Dataframe with hourly values for one aggregated profile

aggregateProfilesWeight(*profiles: pandas.core.frame.DataFrame*, *weights: pandas.core.frame.DataFrame*) → pandas.core.series.Series
Aggregation of profiles considering the specific weights given for the household person IDs. No rescaling of the weights is carried out so far. The function calculateWeightedAverage() is specified in globalFunctions.py

Parameters

- **profilesIn** – Dataframe of hourly values of all filtered profiles
- **weights** – Returns a Dataframe with hourly values considering the weight of individual trip

Returns

baseProfileCalculation()
Wrapper function for first part of flexibility estimation calculating the six resulting profiles for all individual vehicles. The number of iterations for calcChargeMaxProfiles() and calcChargeMinProfiles() can be specified here.

Returns None

booleanMapping(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Replaces given strings with python values for true or false.

Parameters **df** – Dataframe holding strings defining true or false values

Returns Dataframe holding true and false

calcChargeMaxProfiles(*chargeProfiles: pandas.core.frame.DataFrame*, *consumptionProfiles: pandas.core.frame.DataFrame*, *nIter: int*) → pandas.core.frame.DataFrame
Calculates all maximum SoC profiles under the assumption that batteries are always charged as soon as they are plugged to the grid. Values are assured to not fall below SoC_min * battery capacity or surpass SoC_max * battery capacity. Relevant profiles are chargeProfile and consumptionProfile. An iteration assures the boundary condition of chargeMaxProfile(0) = chargeMaxProfile(len(profiles)). The number of iterations is given as parameter.

Parameters

- **chargeProfiles** – Indexed dataframe of charge profiles.
- **consumptionProfiles** – Indexed dataframe of consumptionProfiles.
- **flexConfig** – YAML config holds all relative paths and filenames for flexEstimators.py
- **scalarsProc** – DataFrame holding information about profile length and number of hours.

- **nIter** – Number of iterations to assure that the minimum and maximum value are approximately the same

Returns Returns an indexed DataFrame with the same length and form as chargProfiles and consumptionProfiles, containing single-profile SOC max values for each hour in each profile.

calcChargeMinProfiles(*chargeProfiles: pandas.core.frame.DataFrame*, *consumptionProfiles: pandas.core.frame.DataFrame*, *driveProfilesFuelAux: pandas.core.frame.DataFrame*, *nIter: int = 3*) → pandas.core.frame.DataFrame

Calculates minimum SoC profiles assuming that the hourly mileage has to exactly be fulfilled but no battery charge is kept inspite of fulfilling the mobility demand. It represents the minimum charge that a vehicle battery has to contain in order to fulfill all trips. An iteration is performed in order to assure equality of the SoCs at beginning and end of the profile.

Parameters

- **chargeProfiles** – Charging profiles with techno-economic assumptions on connection power.
- **consumptionProfiles** – Profiles giving consumed electricity for each trip in each hour assuming specified consumption.
- **driveProfilesFuelAux** – Auxilliary fuel demand for fulfilling trips if purely electric driving doesn't suffice.
- **scalarsProc** – Number of profiles and number of hours of each profile.
- **nIter** – Gives the number of iterations to fulfill the boundary condition of the SoC equalling in the first and in the last hour of the profile.

Returns Returns an indexed DataFrame containing minimum SOC values for each profile in each hour in the same format as chargeProfiles, consumptionProfiles and other input parameters.

calcChargeProfiles(*plugProfiles: pandas.core.frame.DataFrame*, *flexConfig*) → pandas.core.frame.DataFrame

Calculates the maximum possible charge power based on the plug profile assuming the charge column power given in the scalar input data file (so far under Panschluss).

Parameters

- **plugProfiles** – indexed boolean profiles for vehicle connection to grid
- **flexConfig** – YAML config which holds all relative paths and filenames for flexEstimators.py

Returns Returns scaled plugProfile in the same format as plugProfiles.

calcChargeProfilesUncontrolled(*chargeMaxProfiles: pandas.core.frame.DataFrame*, *scalarsProc: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Calculates uncontrolled electric charging based on SoC Max profiles for each hour for each profile.

Parameters

- **chargeMaxProfiles** – Dataframe holding timestep dependent SOC max values for each profile.
- **scalarsProc** – VencoPy Dataframe holding meta-information about read-in profiles.

Returns Returns profiles for uncontrolled charging under the assumption that charging occurs as soon as a vehicle is connected to the grid up to the point that the maximum battery SOC is reached or the connection is interrupted. DataFrame has the same format as chargeMaxProfiles.

calcDrainProfiles(*driveProfiles: pandas.core.frame.DataFrame*, *flexConfig: dict*) → pandas.core.frame.DataFrame

Calculates electrical consumption profiles from drive profiles assuming specific consumption (in kWh/100 km) given in scalar input data file.

Parameters

- **driveProfiles** – indexed profile file
- **flexConfig** – YAML config which holds all relative paths and filenames for flexEstimators.py

Returns Returns a dataframe with consumption profiles in kWh/h in same format and length as driveProfiles but scaled with the specific consumption assumption.

calcDriveProfilesFuelAux(*chargeMaxProfiles: pandas.core.frame.DataFrame*, *chargeProfilesUncontrolled: pandas.core.frame.DataFrame*, *driveProfiles: pandas.core.frame.DataFrame*, *flexConfig*, *scalarsProc: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Calculates necessary fuel consumption profile of a potential auxilliary unit (e.g. a gasoline motor) based on gasoline consumption given in scalar input data (in l/100 km). Auxilliary fuel is needed if an hourly mileage is higher than the available SoC Max in that hour.

Parameters

- **chargeMaxProfiles** – Dataframe holding hourly maximum SOC profiles in kWh for all profiles
- **chargeProfilesUncontrolled** – Dataframe holding hourly uncontrolled charging values in kWh/h for all profiles
- **driveProfiles** – Dataframe holding hourly electric driving demand in kWh/h for all profiles.
- **flexConfig** – YAML config which holds all relative paths and filenames for flexEstimators.py
- **scalarsProc** – Dataframe holding meta-infos about the input

Returns Returns a DataFrame with single-profile values for back-up fuel demand in the case a profile cannot completely be fulfilled with electric driving under the given consumption and battery size assumptions.

calcElectricPowerProfiles(*consumptionProfiles: pandas.core.frame.DataFrame*, *driveProfilesFuelAux: pandas.core.frame.DataFrame*, *filterCons: pandas.core.frame.DataFrame*, *filterIndex*) → pandas.core.frame.DataFrame

Calculates electric power profiles that serve as outflow of the fleet batteries.

Parameters

- **consumptionProfiles** – Indexed DataFrame containing electric vehicle consumption profiles.
- **driveProfilesFuelAux** – Indexed DataFrame containing
- **flexConfig** – YAML config which holds all relative paths and filenames for flexEstimators.py
- **filterCons** – Dataframe containing one boolean filter value for each profile
- **scalarsProc** – Dataframe containing meta information of input profiles
- **filterIndex** – Can be either 'indexCons' or 'indexDSM' so far. 'indexDSM' applies stronger filters and results are thus less representative.

Returns Returns electric demand from driving filtered and aggregated to one fleet.

calcProfileSelectors(*chargeProfiles: pandas.core.frame.DataFrame*, *consumptionProfiles: pandas.core.frame.DataFrame*, *driveProfiles: pandas.core.frame.DataFrame*, *driveProfilesFuelAux: pandas.core.frame.DataFrame*, *randNos: pandas.core.frame.DataFrame*, *fuelDriveTolerance*, *isBEV: bool*) → pandas.core.frame.DataFrame

This function calculates two filters. The first filter, filterCons, excludes profiles that depend on auxiliary fuel with an option of a tolerance (bolFuelDriveTolerance) and those that don't reach a minimum daily average for mileage (bolMinDailyMileage). A second filter filterDSM excludes profiles where charging throughout the day supplies less energy than necessary for the respective trips (bolConsumption) and those where the battery doesn't suffice the mileage (bolSuffBat).

Parameters

- **chargeProfiles** – Indexed DataFrame giving hourly charging profiles
- **consumptionProfiles** – Indexed DataFrame giving hourly consumption profiles
- **driveProfiles** – Indexed DataFrame giving hourly electricity demand profiles for driving.
- **driveProfilesFuelAux** – Indexed DataFrame giving auxiliary fuel demand.
- **randNos** – Indexed Series giving a random number between 0 and 1 for each profiles.
- **fuelDriveTolerance** – Give a threshold value how many liters may be needed throughout the course of a day in order to still consider the profile.
- **isBEV** – Boolean value. If true, more 2030 profiles are taken into account (in general).

Returns The bool indices are written to one DataFrame in the DataManager with the columns randNo, indexCons and indexDSM and the same indices as the other profiles.

correct()

Wrapper function to correct all electric and fuel demand profiles with more realistic specific consumption values.

Returns None

correctProfiles(*profile: pandas.core.series.Series*, *profType*) → pandas.core.series.Series

This method scales given profiles by a correction factor. It was written for VencoPy scaling consumption data with the more realistic ARTEMIS driving cycle.

Parameters

- **flexConfig** – YAML config which holds all relative paths and filenames for flexEstimators.py
- **profile** – Dataframe of profile that should be corrected
- **profType** – A list of strings specifying if the given profile type is an electric or a fuel profile. profType has to have the same length as profiles.

Returns

createRandNo(*driveProfiles: pandas.core.frame.DataFrame*, *setSeed=1*)

Creates a random number between 0 and 1 for each profile based on driving profiles.

Parameters

- **driveProfiles** – Dataframe holding hourly electricity consumption values in kWh/h for all profiles
- **setSeed** – Seed for reproducing stochasticity. Scalar number.

Returns Returns an indexed series with the same indices as dirveProfiles with a random number between 0 and 1 for each index.

`filter()`
Wrapper function to carry out filtering and selection procedures. A tolerance for needing additional fuel to carry out trips can be specified to keep profiles in the analyzed data basis.

Returns None

`filterConsProfiles(`*profile: pandas.core.frame.DataFrame*, *filterCons: pandas.core.frame.DataFrame*, *critCol*`)` → pandas.core.frame.DataFrame
Filter out all profiles from given profile types whose boolean indices (so far DSM or cons) are FALSE.

Parameters

- **profile** – Dataframe of hourly values for all filtered profiles
- **filterCons** – Identifiers given as list of string to store filtered profiles back into the DataManager
- **critCol** – Criterium column for filtering

Returns Stores filtered profiles in the DataManager under keys given in dmgrNames

`findIndexCols(`*data: pandas.core.frame.DataFrame*, *nHours: int*`)` → list
Identifies columns that contain index strings rather than numeric data. It does so by checking for column names equal to the numbers from 0 to the number of hours.

Parameters

- **data** – Pandas DataFrame where index columns should be found
- **nHours** – Integer giving the analyzed number of hours

Returns List of index columns

`indexDriveAndPlugData(`*driveData: pandas.core.frame.DataFrame*, *plugData: pandas.core.frame.DataFrame*, *dropIdxLevel: str*, *nHours: int*`)` → tuple
Wrapper function for indexing drive and plug profiles so that value columns are all made up of hourly data.

Parameters

- **driveData** – Hourly drive data for individual vehicles as floats
- **plugData** – Hourly plug data for individual vehicles as boolean
- **dropIdxLevel** – Column to be dropped
- **nHours** – Integer specifying the number of values columns

Returns Tuple of indexed drive and plug profiles as pandas DataFrames

`indexProfile(`*data: pandas.core.frame.DataFrame*, *nHours: int*`)` → pandas.core.frame.DataFrame
Takes raw data as input and indices different profiles with the specified index columns und an unstacked form.

Parameters

- **driveProfiles_raw** – Dataframe of raw drive profiles in km with as many index columns as elements of the list in given in indices. One column represents one timestep, e.g. hour.
- **plugProfiles_raw** – Dataframe of raw plug profiles as boolean values with as many index columns as elements of the list in given in indices. One column represents one timestep e.g. hour.
- **indices** – List of column names given as strings.

Returns Two indexed dataframes with index columns as given in argument indices separated from data columns

indexWeights(*weights: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Reduces the dtype from string to float if possible and sets the index of the weights to align with the indices of drive and plug profiles (genericID and tripStartWeekday).

Parameters **weights** – dataframe containing the MiD trip weights of the original trips

Returns An indexed pandas DataFrame of the MiD trip weights

mergeDataToWeightsAndDays(*ParseData*)
Function to merging weekday and trip weight data to driving and plugging data of respective personHHIDs. It is assumed that trips occur on one daz and trip weights are equal for all trips of one genericID

Parameters **ParseData** – Class instance of type DataParser

Returns None

normalize()
Normalization of soc profiles with regard to the battery capacity.

normalizeProfiles(*socMin: pandas.core.series.Series*, *socMax: pandas.core.series.Series*) → tuple
Normalizes given profiles with a given scalar reference.

Parameters

- **scalars** – Dataframe containing technical assumptions e.g. battery capacity
- **socMin** – Minimum SOC profile subject to normalization
- **socMax** – Minimum SOC profile subject to normalization
- **normReferenceParam** – Reference parameter that is taken for normalization. This has to be given in scalar input data and is most likely the 'Battery_capacity'.

Returns Writes the normalized profiles to the DataManager under the specified keys

procScalars(*driveProfiles_raw*, *plugProfiles_raw*, *driveProfiles: pandas.core.frame.DataFrame*, *plugProfiles: pandas.core.frame.DataFrame*)
Calculates some scalars from the input data such as the number of hours of drive and plug profiles, the number of profiles etc.

Parameters

- **driveProfiles** – Input drive profile input data frame with timestep specific driving distance in km
- **plugProfiles** – Input plug profile input data frame with timestep specific boolean grid connection values

Returns Returns a dataframe of processed scalars including number of profiles and number of hours per profile

readInputBoolean(*filePath*) → pandas.core.frame.DataFrame
Wrapper function for reading boolean data from CSV.

Parameters **filePath** – Relative path to CSV file

Returns Returns a dataframe with boolean values

readInputCSV(*filePath*) → pandas.core.frame.DataFrame
Reads input and cuts out value columns from a given CSV file.

Parameters **filePath** – Relative file path to CSV file

Returns Pandas dataframe with raw input from CSV file

readVencoInput(*datasetID: str*) → tuple

Initializing action for VencoPy-specific config-file, path dictionary and data read-in. The config file has to be a dictionary in a .yaml file containing three categories: pathRelative, pathAbsolute and files. Each category must contain itself a dictionary with the pathRelative to data, functions, plots, scripts, config and tsConfig. Absolute paths should contain the path to the output folder. Files should contain a path to scalar input data, and the two timeseries files inputDataDriveProfiles and inputDataPlugProfiles.

Parameters **config** – A yaml config file holding a dictionary with the keys 'pathRelative' and 'pathAbsolute'

Returns Returns four dataframes: A path dictionary, scalars, drive profile data and plug profile data, the latter three ones in a raw data format.

run()

Wrapper function for the whole flexibility estimation workflow containing the six above described wrapper functions.

Returns None

setUnconsideredBatProfiles(*chargeMaxProfiles: pandas.core.frame.DataFrame*, *chargeMinProfiles: pandas.core.frame.DataFrame*, *filterCons: pandas.core.frame.DataFrame*, *minValue*, *maxValue*)

Sets all profile values with filterCons = False to extreme values. For SoC max profiles, this means a value that is way higher than SoC max capacity. For SoC min this means usually 0. This setting is important for the next step of filtering out extreme values.

Parameters

- **chargeMaxProfiles** – Dataframe containing hourly maximum SOC profiles for all profiles
- **chargeMinProfiles** – Dataframe containing hourly minimum SOC profiles for all profiles
- **filterCons** – Dataframe containing one boolean value for each profile
- **minValue** – Value that non-reasonable values of SoC min profiles should be set to.
- **maxValue** – Value that non-reasonable values of SoC max profiles should be set to.

Returns Writes the two profiles files 'chargeMaxProfilesDSM' and 'chargeMinProfilesDSM' to the DataManager.

socProfileSelection(*profilesMin: pandas.core.frame.DataFrame*, *profilesMax: pandas.core.frame.DataFrame*, *filter*, *alpha*) → tuple

Selects the nth highest value for each hour for min (max profiles based on the percentage given in parameter 'alpha'. If alpha = 10, the 10%-biggest (10%-smallest) value is selected, all other values are disregarded. Currently, in the Venco reproduction phase, the hourly values are selected independently of each other. min and max profiles have to have the same number of columns.

Parameters

- **profilesMin** – Profiles giving minimum hypothetic SOC values to supply the driving demand at each hour
- **profilesMax** – Profiles giving maximum hypothetic SOC values if vehicle is charged as soon as possible
- **filter** – Filter method. Currently implemented: 'singleValue'

- **alpha** – Percentage, giving the amount of profiles whose mobility demand can not be fulfilled after selection.

Returns Returns the two profiles 'socMax' and 'socMin' in the same time resolution as input profiles.

socSelectionVar(*dataMin: pandas.core.frame.DataFrame*, *dataMax: pandas.core.frame.DataFrame*, *by: str*, *filter: str*, *alpha: int*) → tuple

SOC selection function to aggregate state profiles from individual vehicle to fleet level.

Parameters

- **dataMin** – Pandas Dataframe of hourly SOC min profiles for each household person ID
- **dataMax** – Pandas Dataframe of hourly SOC max profiles for each household person ID
- **by** – index level to differentiate selections by. Given as a string.
- **filter** – Filter method given as a string. Currently only 'singleValue' is implemented
- **alpha** – Percentile value to filter out extreme minimum and maximum soc values. E.g. 10 selects the 90th percentile for SOC max and the 10th percentile for SOC min values in each hour. These 24 values most likely do not belong to the same profile.

Returns Returns a tuple of estimated fleet socMin and socMax profiles for nHour x len(set(dataMin.loc[:, by]) values. E.g. if running for 24 hour profiles additionally differentiating weekdays, this yields 168 values per resulting profile.

writeOut()

Generic write-out function for estimated flexibility profiles. Profiles are all written to output/data as specified in the globalConfig and ammended by "vencoPyOutput", the runlabel as specified in the globalConfig as well as the datasetID. Output profiles are all written to one single file.

Returns None

vencopy.classes.flexEstimators.**random**() → x in the interval [0, 1).

vencopy.scripts.globalFunctions.**calculateWeightedAverage**(*col*, *weightCol*)

vencopy.scripts.globalFunctions.**createFileString**(*globalConfig: dict*, *fileKey: str*, *datasetID: Optional[str] = None*, *manualLabel: str = ''*, *filetypeStr: str = 'csv'*)

Generic method used for fileString compilation throughout the VencoPy framework. This method does not write any files but just creates the file name including the filetype suffix.

Parameters

- **globalConfig** – global config file for paths
- **fileKey** – Manual specification of fileKey
- **datasetID** – Manual specification of data set ID e.g. 'MiD17'
- **manualLabel** – Optional manual label to add to filename
- **filetypeStr** – filetype to be written to hard disk

Returns Full name of file to be written.

vencopy.scripts.globalFunctions.**createOutputFolders**(*configDict: dict*)

Function to crete vencopy output folder and subfolders

Param config dictionary

Returns None

vencopy.scripts.globalFunctions.**loadConfigDict**(*configNames: tuple*, *basePath*)
Generic function to load and open yaml config files

Parameters **configNames** – Tuple containing names of config files to be loaded

Returns Dictionary with opened yaml config files

vencopy.scripts.globalFunctions.**mergeDataToWeightsAndDays**(*diaryData*, *ParseData*)

vencopy.scripts.globalFunctions.**mergeVariables**(*data*, *variableData*, *variables*)
Global VencoPy function to merge MiD variables to trip distance, purpose or grid connection data.

Parameters

- **data** – trip diary data as given by tripDiaryBuilder and gridModeler
- **variableData** – Survey data that holds specific variables for merge
- **variables** – Name of variables that will be merged

Returns The merged data

vencopy.scripts.globalFunctions.**writeProfilesToCSV**(*profileDictOut*, *globalConfig: dict*, *localPathConfig: dict*, *singleFile=True*, *datasetID='MiD17'*)
Function to write VencoPy profiles to either one or five .csv files in the output folder specified in outputFolder.

Parameters

- **outputFolder** – path to output folder
- **profileDictOut** – Dictionary with profile names in keys and profiles as pd.Series containing a VencoPy profile each to be written in value
- **singleFile** – If True, all profiles will be appended and written to one .csv file. If False, five files are written
- **strAdd** – String addition for filenames

Returns None

2.7 Input-Output

In the VencoPy inputs and outputs, we can differentiate between configuration files, disk files and classes outputs. Moreover, we can distinguish between the overall framework IO and the classes IO.

2.7.1 General Framework Input-Output

Inputs:

- National travel surveys
- Mobility patterns from traffic models

Outputs:

- Refer to the outputs of the flexEstimator class for more details

2.7.2 Classes Input-Output

- dataParser
- tripDiaryBuilder
- gridModeler
- flexEstimator

2.8 Codestyle Guideline

VencoPy sticks to the codestyle conventions used in the department of energy system analysis at DLR mainly put forward by Benjamin Fuchs. It uses PEP-8 with the exception of lowerCamelCase for variable and method names and UpperCamelCase for class objects.

2.9 Publications

An overview of the main publications using and applying VencoPy as main tool is available below:

- Wulff et al. (2021), *Vehicle Energy Consumption in Python (VencoPy): Presenting and Demonstrating an Open Source Tool to Calculate Electric Vehicle Charging Flexibility*. Energies, MDPI. 2021; 14(14):4349. https://doi.org/10.3390/en14144349 .
- Wulff et al. (2020), *Comparing Power-System and User-Oriented Battery Electric Vehicle Charging Representation and its Impact on Energy System Modeling*. Energies, MDPI. 2020; 13(5):1093. https://doi.org/10.3390/en13051093 .

2.10 Release Timeline

A provisional development timeline is provided below.

Table 3: Historic and future release timeline of VencoPy

Planned quarter	Release version	Planned features
Q4 2020	0.0.X - Alpha release	Core function of VencoPy flexibility estimation
Q3 2021	0.1.X - Beta release	Interface to MiD (parser), trip diary building, grid modeling, structure overhaul, config separation
Q4 2021	0.2.X	Grid modeling refinements, potentially easier config handling, potentially DataClasses for flexibilityEstimation, potentially inheritance in dataParsers
Q1 2022	0.3.X	KiD data parser, code improvements, data validation capabilities in evaluator, performance improvements of trip diary builder

2.11 How to Contribute

2.11.1 General information

VencoPy is a tool that is open for your contribution. If you are interested in contributing to the code, the documentation, have a dataset that may act as an input to VencoPy or want to share other contributions, you are greatfully invited to share them. For this, please read, sign and send the Contributors License Agreement (CLA) to Niklas Wulff.

When contributing to this repository, please discuss the change you wish to make via issue, email, or any other method with the owners of this repository before making a change. Please check our code style, before making any changes to the code.

2.11.2 Maintenance plan

The software of VencoPy will in its current release version be provided as-is. Minor bug fixes may be released by the software engineering team (SET, see *Architecture Documentation*) within the alpha and beta release cycles. Currently, no major support can be offered to the software. However, feel free to reach out if you have any questions.

PYTHON MODULE INDEX

V

INDEX

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W