Method

|  |  |
| --- | --- |
| transformer\_agent.py | |
| Used to create a transformer class.  Input variables:  Number of households  Example of usage:  transformer = Transformer(num\_households=100)  transformer.initialize\_transformer()  capacity = transformer.get\_max\_capacity()  print(capacity) | |
| Function | Description |
| set\_power\_household() | This function calculates the power of a house connection in watts.  The equation for calculating this is:  Volt \* Ampere \* Phases  We assume that for a house connection the voltage is 230 volts, the amperage is 62 amps and one phase is used.  In this function, adjustments can be made if the house connections should have a higher power. |
| set\_customers\_contracted\_power() | The connected load which was calculated with the help of set\_power\_household is converted to kW.  The value is then multiplied by the number of households.  In our simulation, we currently assume that each house has an electric vehicle. Therefore, the number of households is also the number of electric vehicles. |
| get\_c\_diversity() | The diversity factor plays a role when using artificial load profiles; the diversity factor can also be called the simultaneity / coincidence factor. It is used to represent the simultaneous occurrence of the peak loads.  Find source for diversity factor:  https://en.wikipedia.org/wiki/Diversity\_factor |
| set\_transformer\_power\_capacity() | With the formula from the paper quoted below, we calculate the capacity of the transformer.  Influencing factors on capacity are the sum of contracted power values from all customers, the diversity factor, a safety margin, and an oversize factor. Since a transformer usually has a lifetime of several decades (source?), a safety margin and oversized power capacity are implemented. These are used to map the future increase in load that flows through the transformer.  https://www.sciencedirect.com/science/article/pii/S0960148117310649?via%3Dihub |
| initialize\_transformer() | This function runs the previously defined functions set\_power\_household(), set\_customer\_conrtacted\_power() and set\_transformer\_power\_capacity() to calculate the capacity of the transformer in one single action. |
| get\_max\_capacity() | This function outputs the capacity of the transformer class.  Calling this function should only be done after the initialize\_transformer() function has been completed, as the capacity value of the transformer contains a None value as starting value. |

|  |  |
| --- | --- |
| customer\_agent.py | |
| Used to create an electricity customer class having a standard load profile. This allows to map a base load on the transformer, which reduces the available charging power for electric vehicles.  Input variables:  Annual consumption household  Start date  End date  Example of usage:  customer = PowerCustomer(yearly\_cons\_household=3500,  start\_date=start\_date,  end\_date=end\_date)  customer.initialize\_customer()  customer.set\_current\_load(timestamp)  print(customer.current\_load\_kw) | |
| Function | Description |
| create\_cleaned\_h0\_profile() | Loads the file h0\_profile.csv from the input folder. This file is provided by BDEW online at <https://www.bdew.de/energie/standardlastprofile-strom/>.  This represents the average power values in kW of a household in Germany, standardized to 1000 kWh annual consumption, in a resolution of 15 minutes.  The H0 profile is then edited so that it is stored in one column, having the timestamp as index.  The year in the time stamp is set to the year of the start date.  The cleaned standard load profile is then saved as a file with the name cleaned\_h0\_profil.csv in the input folder. |
| set\_standard\_load\_profile() | Loads the processed H0 load profile from the input folder and assigns it to the customer class. |
| set\_scale() | Uses the variable yearly\_cons\_household, which represents the annual consumption of a household in kWh. For example, 3500 kWh. This consumption is then divided by 1000 to calculate the factor used in the set\_scaled\_load\_profile() function to scale the load profile. |
| set\_scaled\_load\_profile() | To scale the standard load profile, which is normalized to 1000 kWh yearly consumption, the scaling factor is taken from set\_scale and the standard load profile is multiplied by this scaling factor.  The result is a scaled standard load profile. |
| initialize\_customer() | This function is used to call the other functions, set\_standard\_load\_profile(), set\_scale() and set\_scaled\_load\_profile().  The function initialize\_customer() calculates the load profiles for a customer. |
| set\_current\_load() | This function has a timestamp as input parameter. The function outputs the load value in kW of the input timestamp. This load value in kW is taken from the scaled load profile. |

|  |  |
| --- | --- |
| mobility\_data.py | |
| Used to create a class called MobilityDataAggregator. This class contains functions to prepare mobility profiles for the electric vehicles. The preparation includes the selection of the correct time range, the aggregation to 15-minute time intervals and initial data cleaning.  Input variables:  raw\_mobility\_data: pd.DataFrame,  start\_date: str,  end\_date: str  Example of usage:  data\_aggregator = MobilityDataAggregator(raw\_mobility\_data=raw\_mobility\_data,  start\_date=self.start\_date,  end\_date=self.end\_date)  print(data\_aggregator.df\_processed) | |
| Function | Description |
| \_create\_df\_limited\_time() | Gets the dataframe of the mobility data for one electric vehicle as input. This dataframe is a dask dataframe. A dask dataframe is used to speed up processes. The first step is to check whether the start date and the end date are outside the range of the mobility data. If this is the case, an error message is returned.  Next, a filter is applied to the mobility data so that only mobility data that is needed for the simulation is stored. |
| \_aggregate\_15\_min\_steps() | The reduced mobility data is aggregated to 15 minute steps.  The timestamp at the beginning of each 15 minute interval is included in the resampled resulting interval.  Aggregations are done as follows:   * ECONSUMPTION: the sum is taken because total values are needed for the 15 minutes. * TRIPNUMBER: the minimum value is taken. Trip numbers are a consecutive ascending number within a mobility profile. If the car is parked, one trip ends and the next one starts. This means that if the trip number increases within the 15 minutes, the trip number is set to the smallest value and therefore the starting trip number. * ID\_PANELSESSION: is set to the maximum. The value varies between 0=ignition, 1=driving and 2=engine turn-off. If the value changes from ignition to driving within a 15-minute interval, the value is set to driving and the entire quarter of an hour is considered to be driving. If it changes from driving to engine turn-off, the entire quarter hour is set to engine turn-off. ECONSUMPTION may still happen. If all three panel sessions occur within a quarter hour, the quarter hour is set to engine turn-off. * ID\_TERMINAL is the same in each aggregation, since this represents the Car\_ID. Set to the first occurring value in the dataframe column. * CLUSTER varies between 0=everywhere else, 1=home and 2=work. Gives the location of the car. Is set to the minimum. This has the effect that the car is still labelled with a 0, although it has already arrived at home in the 15 minute interval. The same applies to work. A car must therefore have been at a location for at least 15 minutes to receive a corresponding label. * DELTAPOS is the change in position in m since the last measurement. Here, the sum is formed during aggregation, since the absolute distance travelled within the 15 minute interval is relevant. |
| \_data\_cleaning() | Two main steps are carried out in datacleaning. Firstly, NAN values in the TRIPNUMBER and ECONSUMPTION columns are dropped from the mobility data. Secondly, the timestamps in the mobility data are checked for the correct format. If the removal of the NAN values or the checking of the timestamp fails, the mobility data is not assigned to any electric vehicle and the script will proceed with the next agent. This happens because both ECONSUMPTION and the timestamps play a crucial role in the following simulation. |
| prepare\_mobility\_data() | Calls the functions \_create\_df\_limited\_time(),\_aggregate\_15\_min\_steps() and \_data\_cleaning(). |
| set\_median\_trip\_len() | Calculates the median length of trips for the added mobility data. |

Beschreibung Mobilitätsdaten:

Es werden von diesen Spalten nur 7 Spalten benutzt. Um das Laden der Mobilitätsdaten schneller zu machen, werden diese Spalten in der Funktion zum Laden spezifiziert. Um die Geschwindigkeit des Ladens weiterhin zu erhöhen wird in der Ladefunktion bereits der Datentyp jeder Spalte definiert. Bei den Spalten handelt es sich um folgende Spalten mit dem dazugehörigen Datentypen {'TIMESTAMP': str, 'TRIPNUMBER': int, 'DELTAPOS': float, 'CLUSTER': int, 'ECONSUMPTIONKWH': float, 'ID\_PANELSESSION': int, 'ID\_TERMINAL': int}

Da der Zeitstempel in String Format gespeichert wird, wird dieser nachträglich im timestamp Format geparst.

909 Dateien mit Mobilitätsprofilen.

REWORK!

Start running the model with opening the main.py file. Here the start date, end date, number of agents (number of electric vehicles) and model runs can be specified.

The number of steps the model has to take will be calculated automatically.

In the main.py file the model.py file will be opened and the class ChargingModel is called.

The ChargingModel has number of agents (number of electric vehicles), start date and end date as input variables. These are already taken from the main.py file and passed from there to the ChargingModel.

It then starts creating a scheduler. In our ChargingModel the SimultaneousActivation class for a scheduler from the mesa library is used because the agents interact at the same timestamp with each other. Therefore, they all need to be activated at the same time.

Currently there is an assertion implemented to stop a user to input more than 698 agents. Usually the maximum value, without assigning the same mobility profiles to two or more electric vehicles, would be 701, since there are 701 private cars in the mobility data set. (Double check if we can use 701)

Starting from the number of agents there will be car models generated. The generation of car models is based on statistics of the Kraftfahrtbundesamt “Fahrzeugzulassungen (FZ)” from 1st October 2022 (<https://www.kba.de/DE/Statistik/Fahrzeuge/Bestand/Vierteljaehrlicher_Bestand/viertelj%C3%A4hrlicher_bestand_node.html>)

The number of created models for the agents is then randomly chosen following the distribution of the official statistics. This means running two times the same simulation will not have exactly the same number of each model the previous run had. However, the models used in one run are saved in a text file in the folder results with the name car\_models\_timestamp to recreate the used models. The models are stored in a list.

After creating the list of electric vehicle models, a transformer will be created with the TransformerClass. The TransformerClass can be found in transformer\_agent.py. Depending on the number of households the maximum capacity of the transformer is calculated.

Then the ElectricVehicle agents are added to the Model. The ElectricVehicle class is created in car\_agent.py file. When adding a new agent, two functions will be executed immediately. The complete\_initialization() and add\_mobility\_data().

Complete initialization has several steps:

1. Load car value dict. This is a file in the input folder with the name car\_values.json. Every car model has its own entry in this json file with the variables: battery\_capacity, charging\_power\_ac, charging\_power\_dc, number. All data can be found in the following table.
2. Set battery capacity. This is taken from the car value dict.
3. Set car size. Because we have 10 different car models, we assume there are 10 different car sizes. For setting the car size we sort the car models according to their battery capacity ascending, where the car having the smallest battery capacity is the smallest car.
4. Set number of car models. This is taken from the car value dict.
5. Set charging power ac. This is taken from the car value dict.
6. Set charging power dc. This is taken from the car value dict.

All these values are stored as attributes of the ElectricVehicle class.

The second function that is called immediately is add\_mobility\_data().

1. Load matching df. The function trys to open a file having the car\_id, the median trip length of the mobility file (Important notice: This is only calculated for the simulation time range, e.g. between start date and end date), and a decile label. Decile label means that the median trip length are also labelled from 1 to 10, where 1 is the shortest median trip length and 10 is the longest median trip length. If this can not be loaded, because it does not exist at the first run, it will be created once, this takes a while and is done in the auxiliary.py file.
2. Create potential matches. This function checks all car\_ids from mobility files that have already been picked and assigned to cars and removes them from the pool of potential mobility files for the current agent. There is also an option to assign mobility files again, if there is no mobility file left to be chosen. (Note: this cannot happen because of the assertion in the class, but if the assertion is removed, there can be theoretical unlimited agents)
3. Create final match. Uses the car size of the agent, taken from the battery capacity and searches for a name of a mobility file having a similar size. A similar size is defined by the absolute difference between car size and median trip length size. Means a car with size 4 will look for a mobility file with median trip length 4. If there is no mobility file with 4 left, the car will look for 3 and 5 and vis-à-vis. All names of possible mobility files are stored in a list and one of these possible mobility file names will then be chosen randomly. (Note: This is another step of randomization, but should not have any impact, because of car 2 is matched with mobility file having size 2) (double check?)
4. Create file path. Will create in auxiliary.py the file path to load the matching file for the corresponding agent.
5. Load mobility data. Will load the mobility csv file using the MobilityDataAggregator class in mobility\_data.py file.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Absolut** | **Relative** | **Battery Capacity in kwh** |  |
| RENAULT ZOE | 84.450 | 10 | 52 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/renault/renault-zoe/ |
| TESLA MODEL 3 | 56.902 | 6,8 | 75 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/tesla/tesla-model-3/ |
| VW UP | 50.859 | 6 | 32,3 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/vw/vw-e-up/ |
| VW ID.3 | 48.483 | 5,8 | 58 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/vw/vw-id-3/ |
| SMART FORTWO | 47.683 | 5,7 | 16,7 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/smart/smart-eq-fortwo/ |
| HYUNDAI KONA | 40.374 | 4,8 | 64 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/hyundai/hyundai-kona-elektro/ |
| BMW I3 | 39.013 | 4,6 | 37,9 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/bmw/bmw-i3/ |
| FIAT 500 | 29.035 | 3,5 | 37,3 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/fiat/fiat-500-elektro/ |
| VW GOLF | 26.891 | 3,2 | 32 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/vw/golf/vii-facelift/266575/#technische-daten |
| VW ID.4, ID.5 | 25.831 | 3,1 | 77 | https://www.adac.de/rund-ums-fahrzeug/autokatalog/marken-modelle/vw/vw-id-4/ |
|  |  |  |  |  |
| **BEV TOP 10** | **449.521** | **53,5** |  |  |
| **Elektro (BEV) insgesamt** | **840.645** | **100** |  |  |

Interaction: (REWORK Explanation!)

# find all agents with the highest priority  
# add the charging power one after another to a sub total charging power for that priority  
# check continuously if the sub total charging power for that priority is higher than the max\_capacity  
# if it is higher all cars of that priority needs to be reduced AND all other charging values  
# need to be set to 0  
  
# else  
# go to the next priority and start the process again  
# BUT set the charging value only of the agents to 0 that have lower priority than the current  
# priority, or check if agents are in a list, if they are not in the list, set the charging power  
# to 0  
  
# AFTER THIS revert the charge for all cars that are either set to 0 or have reduced charging power  
# charge again  
# if charging reduction is split to all charging cars with same priority  
# it have to be maximum capped at last charging value  
# the rest of charging value that is then used can be distributed to all others