# Contents

1 Introduction 1  
1.1 Objective ........................................... 1  
1.2 Why? ................................................. 2  
1.3 How to install this package? .......................... 2  
1.4 How to use it? ........................................ 3  

2 Structure 11  
2.1 Modules ............................................... 12  
2.2 Driving cycle module .................................. 12  
2.3 Mass module .......................................... 13  
2.4 Auxiliary energy module ............................... 13  
2.5 Motive energy module ................................ 14  

3 Modeling and assumptions 15  
3.1 Vehicle sizing ........................................ 15  
3.2 Tank-to-wheel energy consumption .................... 24  
3.3 Fuel blends .......................................... 32  
3.4 Fuel-related direct emissions .......................... 33  
3.5 Hot pollutants emissions ............................... 33  
3.6 Noise emissions ....................................... 36  
3.7 Vehicle inventory ...................................... 36  
3.8 Fuel pathways ........................................ 40  
3.9 Electricity mixes for battery charging and hydrogen production ........................................ 41  
3.10 Background inventory .................................. 42  

4 Validity tests 45  
4.1 Driving cycle, velocity and acceleration ............... 45  
4.2 Car and components masses ............................ 47  
4.3 Tank-to-wheel energy .................................. 50  
4.4 End-of-pipe CO2 emissions ............................. 51  

5 Technical Reference 53  
5.1 Car Input Parameter .................................... 53  
5.2 Array .................................................. 54  
5.3 Driving cycle .......................................... 55  
5.4 Energy consumption .................................... 55  
5.5 Car Model ............................................. 57
cartulator is a parameterized model that allows to generate and characterize life cycle inventories for different vehicle configurations, according to selected:

- powertrain technologies (9): petrol engine, diesel engine, electric motor, hybrid, plugin-hybrid, etc.,
- year of operation (2): 2000, 2010, 2017, 2040 (with the possibility to interpolate in between, and up to 2050)
- and sizes (7): Mini, Large, etc.

The methodology used to develop cartulator is explained in: cartulator: an open-source tool for prospective environmental and economic life cycle assessment of vehicles. When, Where and How can battery-electric vehicles help reduce greenhouse gas emissions? Romain Sacchi, Christian Bauer, Brian Cox, Christopher Mutel Environmental Modelling and Software (2020, submitted)

At the moment, the tool has a focus on passenger cars.

It is initially based on the model developed in Uncertain environmental footprint of current and future battery electric vehicles by Cox, et al (2018).

More specifically, cartulator generates Brightway2 and SimaPro inventories, but also directly provides characterized results against several midpoint indicators from the impact assessment method ReCiPe as well as life cycle cost indicators.

cartulator is a special in the way that it uses time- and energy-scenario-differentiated background inventories for the future, resulting from the coupling between the ecoinvent 3.6 database and the scenario outputs of PIK’s integrated assessment model REMIND. This allows to perform prospective study while consider future expected changes in regard to the production of electricity, cement, steel, heat, etc.

1.1 Objective

The objective is to produce life cycle inventories for vehicles in a transparent, comprehensive and quick manner, to be further used in prospective LCA of transportation technologies.
1.2 Why?

Many life cycle assessment (LCA) models of passenger cars exist. Yet, because LCA of vehicles, particularly for electric battery vehicles, are sensitive to assumptions made in regards to electricity mix used for charging, lifetime of the battery, etc., it has led to mixed conclusions being published in the scientific literature. Because the underlying calculations are kept undocumented, it is not always possible to explain the disparity in the results given by these models, which can contribute to adding confusion among the public.

Because carculator is kept as open as possible, the methods and assumptions behind the generation of results are easily identifiable and adjustable. Also, there is an effort to keep the different modules (classes) separated, so that improving certain areas of the model is relatively easy and does not require changing extensive parts of the code. In that regard, contributions are welcome.

Finally, beside being more flexible and transparent, carculator provides interesting features, such as:

• a stochastic mode, that allows fast Monte Carlo analyses, to include uncertainty at the vehicle level
• possibility to override any or all of the 200+ default input car parameters (e.g., number of passengers, drag coefficient) but also calculated parameters (e.g., driving mass).
• hot pollutants emissions as a function of the driving cycle, using HBEFA 4.1 data, further divided between rural, suburban and urban areas
• noise emissions, based on CNOSSOS-EU models for noise emissions and Noise footprint from personal land-based mobility by Cucurachi, et al (2019) for inventory modelling and mid- and endpoint characterization of noise emissions, function of driving cycle and further divided between rural, suburban and urban areas
• export of inventories as an Excel/CSV file, to be used with Brightway2 or Simapro, including uncertainty information. This requires the user to have ecoinvent 3.6 cutoff installed on the LCA software the car inventories are exported to.
• export inventories directly into Brightway2, as a LCIImporter object to be registered. Additionally, when run in stochastic mode, it is possible to export arrays of pre-sampled values using the presamples library to be used together with the Monte Carlo function of Brightway2.
• development of an online graphical user interface: carculator online

1.3 How to install this package?

carculator is a Python package, and is primarily to be used from within a Python 3.x environment. Because carculator is still at an early development stage, it is a good idea to install it in a separate environment, such as a conda environment:

conda create -n <name of the environment> python=3.7

Once your environment created, you should activate it:

conda activate <name of the environment>

And install the carculator library in your new environment via Conda:

pip install carculator

This will install the package and the required dependencies.
1.4 How to use it?

1.4.1 Static vs. Stochastic mode

Note: many examples are given in this notebook that you can run directly on your computer.

The inventories can be calculated using the most likely value of the given input parameters (“static” mode), but also using randomly-generated values based on a probability distribution for those (“stochastic” mode).

For example, the drivetrain efficiency of SUVs in 2017, regardless of the powertrain, is given the most likely value (i.e., the mode) of 0.38, but with a triangular probability distribution with a minimum and maximum of 0.3 and 0.4, respectively.

Creating car models in static mode will use the most likely value of the given parameters to dimension the cars, etc., such as:

```python
from carculator import *
cip = CarInputParameters()
cip.static()
dcts, array = fill_xarray_from_input_parameters(cip)
cm = CarModel(array)
cm.set_all()
```

Alternatively, if one wishes to work with probability distributions as parameter values instead:

```python
from carculator import *
cip = CarInputParameters()
cip.stochastic(800)
dcts, array = fill_xarray_from_input_parameters(cip)
cm = CarModel(array)
cm.set_all()
```

This effectively creates 800 iterations of the same car models, picking pseudo-random value for the given parameters, within the probability distributions defined. This allows to assess later the effect of uncertainty propagation on characterized results.

In both case, a CarModel object is returned, with a 4-dimensional array array to store the generated parameters values, with the following dimensions:

0. **Vehicle sizes (called “size”):**
   - Mini
   - Small
   - Lower medium
   - Medium
   - Large
   - SUV
   - Van

1. **Powertrains:**
   - ICEV-p, ICEV-d, ICEV-g: vehicles with internal combustion engines running on gasoline, diesel and compressed gas, respectively.
   - HEV-p, HEV-d: vehicles with internal combustion engines running on gasoline and diesel, assisted with an electric engine.
Carculator, Release 1.0.0

- PHEV-p, PHEV-d: vehicles with internal combustion engines running on gasoline and diesel, assisted with a plugin electric engine.
- BEV: battery electric vehicles.
- FCEV: fuel cell electric vehicles.

2. Year. Anything between 2000 and 2050.

3. Iteration number (length = 1 if static(), otherwise length = number of iterations).

\texttt{cm.set\_all()} generates a CarModel object and calculates the energy consumption, components mass, as well as exhaust and non-exhaust emissions for all vehicle profiles.

1.4.2 Custom values for given parameters

You can pass your own values for the given parameters, effectively overriding the default values.

For example, you may think that the base mass of the glider for large diesel and petrol cars is 1600 kg in 2017 and 1,500 kg in 2040, and not 1,500 kg as defined by the default values. It is easy to change this value. You need to create first a dictionary and define your new values as well as a probability distribution if needed:

\begin{verbatim}

dic_param = {
    ('Glider', ['ICEV-d', 'ICEV-p'], 'Large', 'glider base mass', 'triangular'): {
        (2017, 'loc'): 1600.0,
        (2017, 'minimum'): 1500.0,
        (2017, 'maximum'): 2000.0,
        (2040, 'loc'): 1500.0,
        (2040, 'minimum'): 1300.0,
        (2040, 'maximum'): 1700.0,
    }
}
\end{verbatim}

Then, you simply pass this dictionary to \texttt{modify\_xarray\_from\_custom\_parameters(<dic\_param or filepath>, array)}, like so:

\begin{verbatim}

cip = CarInputParameters()
cip.static()
dcts, array = fill\_xarray\_from\_input\_parameters(cip)
modify\_xarray\_from\_custom\_parameters(dic\_param, array)
cm = CarModel(array, cycle='WLTC')
cm.set\_all()
\end{verbatim}

Alternatively, instead of a Python dictionary, you can pass a file path pointing to an Excel spreadsheet that contains the values to change, following this template.

The following probability distributions are accepted: * “triangular” * “lognormal” * “normal” * “uniform” * “none”

1.4.3 Inter and extrapolation of parameters

Carculator creates by default car models for the year 2000, 2010, 2017 and 2040. It is possible to inter and extrapolate all the parameters to other years simply by writing:

\begin{verbatim}

array = array.interp(year=[2018, 2022, 2035, 2040, 2045, 2050], kwargs={"fill_value" : 'extrapolate'})
\end{verbatim}
However, we do not recommend extrapolating for years before 2000 or beyond 2050.

### 1.4.4 Changing the driving cycle

carculator gives the user the possibility to choose between several driving cycles. Driving cycles are determinant in many aspects of the car model: hot pollutant emissions, noise emissions, tank-to-wheel energy, etc. Hence, each driving cycle leads to slightly different results. By default, if no driving cycle is specified, the WLTC driving cycle is used. To specify a driving cycle, simply do:

```python
cip = CarInputParameters()
cip.static()
dcts, array = fill_xarray_from_input_parameters(cip)
cm = CarModel(array, cycle='WLTC 3.4')
cm.set_all()
```

In this case, the driving cycle *WLTC 3.4* is chosen (this driving cycle is in fact a sub-part of the WLTC driving cycle, mostly concerned with driving on the motorway at speeds above 80 km/h). Driving cycles currently available:

- WLTC
- WLTC 3.1
- WLTC 3.2
- WLTC 3.3
- WLTC 3.4
- CADC Urban
- CADC Road
- CADC Motorway
- CADC Motorway 130
- CADC
- NEDC

The user can also create custom driving cycles and pass it to the CarModel class:

```python
import numpy as np
x = np.linspace(1, 1000)
def f(x):
    return np.sin(x) + np.random.normal(scale=20, size=len(x)) + 70
cycle = f(x)
cm = CarModel(array, cycle=cycle)
```

### 1.4.5 Accessing calculated parameters of the car model

Hence, the tank-to-wheel energy requirement per km driven per powertrain technology for a SUV in 2017 can be obtained from the CarModel object:

```python
TtW_energy = cm.array.sel(size='SUV', year=2017, parameter='TtW energy', value=0) * 1/3600 * 100
plt.bar(TtW_energy.powertrain, TtW_energy)
```

(continues on next page)
Note that if you call the `stochastic()` method of the `CarInputParameters`, you would have several values stored for a given calculated parameter in the array. The number of values correspond to the number of iterations you passed to `stochastic()`.

For example, if you ran the model in stochastic mode with 800 iterations as shown in the section above, instead of one value for the tank-to-wheel energy, you would have a distribution of values:

```python
l_powertrains = TtW_energy.powertrain
[plt.hist(e, bins=50, alpha=.8, label=e.powertrain.values) for e in TtW_energy]
plt.ylabel('kWh/100 km')
plt.legend()
```

Any other attributes of the `CarModel` class can be obtained in a similar way. Hence, the following code lists all direct exhaust emissions included in the inventory of an petrol Van in 2017:
List of all the given and calculated parameters of the car model:

```python
list_param = cm.array.coords['parameter'].values.tolist()
```

Return the parameters concerned with direct exhaust emissions (we remove noise emissions):

```python
direct_emissions = [x for x in list_param if 'emission' in x and 'noise' not in x]
```

Finally, return their values and display the first 10 in a table:

```python
cm.array.sel(parameter=direct_emissions, year=2017, size='Van', powertrain='BEV').to_dataframe(name='direct emissions')
```

Or we could be interested in visualizing the distribution of non-characterized noise emissions, in joules:

```python
noise_emissions = [x for x in list_param if 'noise' in x]
data = cm.array.sel(parameter=noise_emissions, year=2017, size='Van', powertrain='ICEV-p', value=0).to_dataframe(name='noise emissions')['noise emissions']
data[data>0].plot(kind='bar')
plt.ylabel('joules per km')
```

1.4. How to use it?
1.4.6 Modify calculated parameters

As input parameters, calculated parameters can also be overridden. For example here, we override the *driving mass* of large diesel vehicles for 2010 and 2017:

```python
cm.array.loc['Large','ICEV-d', 'driving mass', [2010, 2017]] = [[2000],[2200]]
```

1.4.7 Characterization of inventories (static)

carculator makes the characterization of inventories easy. You can characterize the inventories directly from carculator against midpoint impact assessment methods.

For example, to obtain characterized results against the midpoint impact assessment method ReCiPe for all cars:

```python
ic = InventoryCalculation(cm.array)
results = ic.calculate_impacts()
```

Hence, to plot the carbon footprint for all medium cars in 2017:

```python
results.sel(size='Medium', year=2017, impact_category='climate change', value=0).to_dataframe('impact').unstack(level=1)['impact'].plot(kind='bar', stacked=True)
plt.ylabel('kg CO2-eq./vkm')
plt.show()
```

Note that, for now, only the ReCiPe method is available for midpoint characterization. Also, once the instance of the *CarModel* class has been created, there is no need to re-create it in order to calculate additional environmental impacts (unless you wish to change values of certain input or calculated parameters, the driving cycle or go from static to stochastic mode).

1.4.8 Characterization of inventories (stochastic)

In the same manner, you can obtain distributions of results, instead of one-point values if you have run the model in stochastic mode (with 500 iterations and the driving cycle WLTC).
Carculator, Release 1.0.0

cip = CarInputParameters()
cip.stochastic(500)
dcts, array = fill_xarray_from_input_parameters(cip)
cm = CarModel(array, cycle='WLTC')
cm.set_all()
scope = {
    'powertrain':['BEV', 'PHEV'],
}
ic = InventoryCalculation(cm.array, scope=scope)
results = ic.calculate_impacts()
data_MC = results.sel(impact_category='climate change').sum(axis=3).to_dataframe(
    'climate change')
plt.style.use('seaborn')
data_MC.unstack(level=[0,1,2]).boxplot(showfliers=False, figsize=(20,5))
plt.xticks(rotation=70)
plt.ylabel('kg CO2-eq./vkm')

Many other examples are described in a Jupyter Notebook in the examples folder.

1.4.9 Export of inventories (static)

Inventories can be exported as:

- a Python list of exchanges
- a Brightway2 bw2io.importers.base_lci.LCIImporter object, ready to be imported in a Brightway2 environment
- an Excel file, to be imported in a Brightway2 environment
- a CSV file, to be imported in SimaPro 9.x.

```python
ic = InventoryCalculation(cm.array)
# export the inventories as a Python list
mylist = ic.export_lci()
# export the inventories as a Brightway2 object
import_object = ic.export_lci_to_bw()
# export the inventories as an Excel file (returns the file path of the created file)
filepath = ic.export_lci_to_excel(software_compatibility="brightway2", ecoinvent_version="3.7")
filepath = ic.export_lci_to_excel(software_compatibility="simapro", ecoinvent_version="3.6")
```

1.4. How to use it?
### 1.4.10 Export of inventories (stochastic)

If you had run the model in stochastic mode, the export functions return in addition an array that contains pre-sampled values for each parameter of each car, in order to perform Monte Carlo analyses in Brightway2.

```python
ic = InventoryCalculation(cm.array)

# export the inventories as a Python list
mylist, presamples_arr = ic.export_lci()

# export the inventories as a Brightway2 object
import_object, presamples_arr = ic.export_lci_to_bw()

# export the inventories as an Excel file (note that this method does not return the
# presamples array)
filepath = ic.export_lci_to_excel()
```

### 1.4.11 Import of inventories (static)

The background inventory is originally a combination between ecoinvent 3.6 and outputs from PIK’s REMIND model. Outputs from PIK’s REMIND are used to project expected progress in different sectors into ecoinvent. For example, the efficiency of electricity-producing technologies as well as the electricity mixes in the future for the main world regions are built upon REMIND outputs. The library used to create hybrid versions of the ecoinvent database from PIK’s REMIND is called premise. This means that, as it is, the inventory cannot properly link to ecoinvent 3.6 or 3.7 unless some transformation is performed before. These transformations are in fact performed when exporting the inventory. Hence, when doing:

```python
ic.export_lci_to_excel(ecoinvent_compatibility=True, ecoinvent_version="3.6")
```

the resulting inventory should properly link to the unmodified version of ecoinvent 3.6 cutoff. Should you wish to export an inventory to link with a IAM-modified version of ecoinvent, just export the inventory with the `ecoinvent_compatibility` argument set to `False`.

```python
ic.export_lci_to_excel(ecoinvent_compatibility=False, ecoinvent_version="3.6")
```

In that case, the inventory will only link to a custom ecoinvent database produced by `premise`.

But in any case, the following script should successfully import the inventory into a Brightway2 project:

```python
import brightway2 as bw
bw.projects.set_current("test_calculator")
import bw2io
fp = r"C:\file_path_to_the_inventory\lci-test.xlsx"

i = bw2io.ExcelImporter(fp)
i.apply_strategies()

if 'additional_biosphere' not in bw.databases:
    i.create_new_biosphere('additional_biosphere')
i.match_database("name_of_the_ecoinvent_db", fields=('name', 'unit', 'location',
              'reference product'))
i.match_database("biosphere3", fields=('name', 'unit', 'categories'))
i.match_database("additional_biosphere", fields=('name', 'unit', 'categories'))
i.match_database(fields=('name', 'unit', 'location'))

i.statistics()
i.write_database()
```
CHAPTER 2

Structure
2.1 Modules

Composed of eight modules to build the car models:

- Driving cycle module
- Mass module
- Auxiliary energy module
- Motive energy module
- Fuel-related emissions module
- Hot pollutant emissions module
- Non-exhaust emissions module
- Noise emissions module

Additionally, three modules are used to:

- configure energy systems for the background model (background systems module)
- build and solve the life cycle inventory of cars (inventory module)
- export the life cycle inventory of cars (export module)

2.2 Driving cycle module
2.3 Mass module

2.4 Auxiliary energy module
2.5 Motive energy module
CHAPTER 3

Modeling and assumptions

The modeling of passenger vehicles in the past, present and future is complex and relies on many assumptions. With calculator and calculator online, we wish to be transparent about those: assumptions and modeling approaches should ideally be easily critiqued and modified.

We try here to give a comprehensive list of assumptions and modeling choices, and describe how, as a user, you can change those.

Parameters’ names are indicated verbatim and are to be used in calculator. The can also be accessed and modified via its online graphical user interface calculator online, via the search bar in the Car Parameters section.

3.1 Vehicle sizing

calculator models vehicles along four dimensions:

- their powertrain (e.g., gasoline-run internal combustion engine, battery electric vehicle, etc.),
- their size (e.g., mini, medium, large, etc.),
- their year of production (2000, 2010, 2017 and 2040)
- and a parameter dimension (i.e., input and calculated parameters).

When calculator sizes the vehicles for the different powertrains, sizes and years, it starts with the input parameter’s value for the glider base mass, which is essentially an initial guess for the mass of the vehicle’s glider without anything on it.

Then it adds the following components and their associated mass:

- fuel mass: mass of the fuel in the fuel tank (only applicable to vehicles using liquid or gaseous fuels),
- fuel tank mass: mass of the fuel tank (empty),
- charger mass: mass of the onboard battery charger (for battery electric and plug-in hybrid vehicles only),
- converter mass: mass of the onboard electricity AC/DC converter (for battery electric and plug-in hybrid vehicles only),
• **inverter mass**: mass of the onboard electricity DC/AC converter (for battery electric and plugin hybrid vehicles only),

• **power distribution unit mass**: mass of the onboard power distribution unit (for battery electric and plugin hybrid vehicles only),

• **combustion engine mass**: mass of the internal combustion engine (if applicable),

• **electric engine mass**: mass of the electric motor (if applicable),

• **powertrain mass**: mass of the powertrain excluding the mass of the engine (e.g., transmission, drive shafts, differentials, etc.),

• **fuel cell stack mass**: mass of the fuel cell stack (only for fuel cell electric vehicles),

• **fuel cell ancillary BoP mass**: mass of the ancillary part of the Balance of Plant of the fuel cell stack (only for fuel cell electric vehicles),

• **fuel cell essential BoP mass**: mass of the essential part of the Balance of Plant of the fuel cell stack (only for fuel cell electric vehicles),

• **battery cell mass**: mass of the battery cells. Two types of batteries are distinguished: power and energy batteries,

• **battery BoP mass**: mass of the Balance of Plant of the battery.

Adding the mass of the glider to the mass of these components constitutes a first attempt at guessing the **curb mass** of the vehicle, that is its mass in working order, but without passengers and cargo. The **driving mass** of the vehicle is then obtained by summing the curb mass to the mass of the passengers (average passengers x average passenger mass) and cargo transported (cargo mass).

A second step consists into calculating the mass of the combustion and electric engine, based on the following relations:

- **power demand** (power) [kW] = power-to-mass ratio [kW/kg] x curb mass [kg]
- **electrical power demand** (electric power) [kW] = power demand (power) [kW] x (1 - combustion power share [%])

- **electric engine mass** [kW] = (electric power [kW] x electric mass per power [kg/kW]) + electric fixed mass [kg]

- **combustion power demand** (combustion power) [kW] = power [kW] x combustion power share [%]

- **combustion engine mass** [kW] = (combustion power [kW] x combustion mass per power [kg/kW]) + combustion fixed mass [kg]

As well as for the mass of the powertrain:

- **powertrain mass** [kg] = (power [kW] x powertrain mass per power [kg/kW]) + powertrain fixed mass [kg]

With the mass of these new components recalculated (electric engine mass, combustion engine mass and powertrain mass), the curb mass of the vehicle is calculated once again. This iterative process stops when the curb mass of the vehicle stabilizes (i.e., when recalculating the mass of the engine and powertrain does not lead to a change in the new curb mass of more than one percent).
Four initial input parameters are therefore of importance:

- **glider base mass**: the initial mass of the glider
- **power to mass ratio**: the power-to-mass ratio
- **combustion power share**: how much of the power is provided by an internal combustion engine
- **combustion mass per power**: the mass of the combustion engine per unit of power

For electric vehicles (i.e., BEV and FCEV), **combustion power share** = 0. For internal combustion engine vehicles (i.e., ICEV-p, ICEV-d and ICEV-g), **combustion power share** = 1 in the early years (until 2020). However, starting 2020 on, this value drops progressively to 0.85 by 2050, as we assumed a mild-hybridization of the powertrain to a level similar to that of non-plugin hybrids nowadays (i.e., HEV-p and HEV-d). While it is uncertain whether ICEVs will exist in the future, it was assumed that a way for them to comply with future emission standards was to be assisted by an electric engine. This mild-hybridization allows to reduce the size of the combustion engine and recover energy during braking.

For non-plugin hybrids, **combustion power share** is usually set at around 0.75.

For plugin hybrid vehicles, things are modeled differently: a purely electric vehicle is modeled, as well as a purely combustion-based vehicle. Later on, when the range of the purely-electric vehicle is calculated, a **electric utility ratio** is obtained, which is used to fusion both vehicles. This ratio, which is dependent on the range, is usually between 0.6 and 0.7. This means that plugin hybrid vehicles are made of between 60 and 70% of a purely electric vehicle and 30 to 40% of a purely combustion-based vehicle.

### 3.1.1 If I know already the curb mass of a vehicle, can I override its value?

With Carculator online:
Currently, it is not possible to modify directly the calculated parameter curb mass, as it would be recalculated. In order to do so, you need to use instead the Python library carculator (see next section). You can however modify any of the input parameters glider base mass, power to mass ratio, combustion power share and combustion mass per power used to calculate curb mass. To do so, type their name in the search field of the Parameters section.

With carculator:

Yes. After having created the CarModel() object and executed the set_all() method, you can override the calculated curb mass value. Here is an example for a diesel car of medium size in 2020:

```python
cm = CarModel(array, cycle='WLTC')
cm.set_all()
cm.array.loc[dict(parameter="curb mass",
               powertrain="ICEV-d",
               year=2020,
               size="Medium")]
      = 1600
```

### 3.1.2 How to prevent the mild-hybridization of ICEVs?

With carculator online:

In the Parameters section, search for combustion power share and add the parameter for the vehicles you wish to modify.

With carculator:

You can simply override the default value by "1" in array before passing it to CarModel():

```python
dict_param = {
    ('Powertrain', ('ICEV-d', 'ICEV-p', 'ICEV-g'), 'all', 'combustion_power_share', 'none'): {
        (2000, 'loc'): 1,
    }
}
```
You can also just override the default value of a specific powertrain of a specific size, for a specific year:

\[
\text{dict\_param} = \{\text{('Powertrain', 'ICEV-d', 'Medium', 'combustion power share', 'none'): \{(2017, 'loc'): 1\}}\}
\]

\[
\text{modify\_xarray\_from\_custom\_parameters(dict\_param, array)}
\]

### 3.1.3 How can I modify the battery capacity of a battery electric car?

Two parameters are of importance, energy battery mass [kg] and battery cell energy density [kWh/kg], so that:

\[
\text{battery cell mass [kg]} = \text{energy battery mass [kg]} \times \text{battery cell mass share [%]}
\]

\[
\text{energy stored [kWh]} = \text{battery cell energy density [kWh/kg]} \times \text{battery cell mass [kg]}
\]

Hence, by modifying either of them (or both), you can affect the capacity of the battery for a given size class.

With calculator online:

![Battery capacity modification interface](image)

With calculator:

You can simply override the default values in array before passing it to CarModel():

\[
\]

\[
\text{modify\_xarray\_from\_custom\_parameters(dict\_param, array)}
\]

(continues on next page)
The curb mass values obtained for the vehicles in 2000, 2010 and 2017 are calibrated against a passenger cars database Car2DB. The calibration of the curb mass for vehicles for the year 2000 is done against vehicles in the Car2DB database with a production year in the range of 1998-2002, against 2008-2012 and 2015-2018 for vehicles for the years 2010 and 2017, respectively. The value of the input parameter glider base mass was adjusted to fit the distribution shown in the plots below.

Calibration of vehicles’ curb mass for the year 2000

Calibration of vehicles’ curb mass for the year 2010
3.1. Vehicle sizing

Calibration of vehicles’ curb mass for the year 2017
For the year 2040, the value for input parameters glider base mass, combustion mass per power, power to mass ratio are adjusted according to the following studies:


### 3.1.4 What happens when I inter-/extrapolate to other years?

If the default years of 2000, 2010, 2017 and 2040 are of no interest, it is possible to inter-/extrapolate the vehicle models to any year between 2000 and 2050. When such inter-/extrapolation is done, all the physical input parameters’ values are inter-/extrapolated linearly.
With **carculator online**: 

In the Scope section, simply drag the desired years from the left frame to the right frame.

With **carculator**: 

After creating `array`, which is a `DataArray` object from the library `xarray`, it is possible to use the `.interp()` method, like so:

```python
array = array.interp(year=np.arange(2015, 2051, 5), kwars={'fill_value': 'extrapolate'})
```

Here, the years under study are from 2015 to 2050 by step of 5 years.

This is slightly different for cost input parameters’ values, which are usually following a decay-like cost curve, to account for a learning rate. Hence, parameters such as fuel tank cost per kg, fuel cell cost per kW, energy battery cost per kWh, power battery cost per kW, or combustion powertrain cost per kW would be of shape: $a\exp(b) + c$. Coefficients $a$, $b$ and $c$ are defined to fit the literature and projections.

Projection of energy battery cost per kWh for BEV and FCEV.

![Cost evolution for battery systems](image)

### 3.1. Vehicle sizing

23
3.2 Tank-to-wheel energy consumption

The tank-to-wheel energy consumption is the sum of:

- the motive energy needed to move the vehicle over 1 km
- the auxilliary energy needed to operate on-board equipment as well as to provide heating and cooling over 1 km

3.2.1 Motive energy

Once the vehicle and its powertrain has been sized, it is possible to calculate the motive energy required along a specific driving cycle to overcome the following forces:

- rolling resistance
- aerodynamic resistance
- air resistance
- road gradient resistance (if provided)

on top of the kinetic energy needed to move the vehicle.

To calculate the motive energy, the following parameters are needed:

- the driving mass of the vehicle
- its rolling resistance coefficient
- its aerodynamic drag coefficient
- its frontal area
- its tank-to-wheel efficiency (TtW efficiency)
- its recuperation efficiency
- and the power of its electric motor, if any (electric power)

To that amount of energy is subtracted the energy recuperated during braking, if the vehicle is equipped with an electric motor (to the extent of the power of the motor, discounted with a recuperation efficiency).

\[
\text{recuperation efficiency} [\%] = \text{drivetrain efficiency} [\%] \times \text{battery charge efficiency} [\%]
\]
Also, distance, velocity and acceleration are derived from the driving cycle.

In parallel, the TtW efficiency (the loss of energy between the energy storage and the wheels) is calculated as the product of the following efficiency parameters:

- battery discharge efficiency
- fuel cell system efficiency
- drivetrain efficiency
- engine efficiency

The motive energy is calculated as the sum of:

- rolling resistance \([\text{kg.m.s}^{-2}] = \text{driving mass [kg]} \times \text{rolling resistance coefficient [%]} \times 9.81 \text{ [m/s}^{2}])

- air resistance \([\text{kg.m.s}^{-2}] = \text{velocity}^{2} \times \text{frontal area [m}^{2}] \times \text{aerodynamic drag coefficient [%]} \times \text{air density [kg/m}^{3}] / 2)

- road gradient resistance \([\text{kg.m.s}^{-2}] = \text{driving mass [kg]} \times 9.81 \text{ [m/s}^{2}] \times \sin(\text{gradient})

- kinetic force \([\text{kg.m.s}^{-2}] = \text{acceleration [m/s}^{2}] \times \text{driving mass [kg]}

This gives:

- force required \([\text{kg.m.s}^{-2}] = \text{rolling resistance + air resistance + road gradient resistance + kinetic force}

Then, the gross power required is calculated as:

3.2. Tank-to-wheel energy consumption
• power [W or kg.m^2.s^-3] = force required [kg.m.s^-2] x velocity [m/s]

The recuperated power, via electro-braking is calculated as the decelerating power (when power is negative) comprised between 0 and the electric engine power *-1, times the recuperation efficiency:

• recuperated power [W] = power [W] * recuperation efficiency [%], when power between (-1 x electric engine power [W]) and 0

Finally, to obtain the motive energy the gross power minus the recuperated power (which is negative!) are summed along the driving cycle duration:

• motive energy [joules] = sum ((power [W or joules/s] + recuperated power [W or joules/s]) / distance [m] / TtW efficiency [%] / 1000 [j/kj])

The motive energy is divided by the TtW efficiency to obtain the amount of kilojoules needed in the tank (or battery) to move the vehicle over 1 km.

Here is plotted the second-by-second gross power requirement for a large-sized battery electric vehicle, along the WLTC driving cycle:

### 3.2.2 How can I add a road gradient?

By default, the vehicles are compared based on a driving cycle on a flat road.

It is however possible to pass a gradient series (which must provide a gradient degree for each second of the driving cycle) to the energy model:
3.2.3 Auxilliary energy

The auxilliary energy, that is the energy needed to operate onboard equipment and heating and cooling systems, is also calculated as the sum of the power demand over time.

This power demand entails:

- the average power demand for heating
- the average power demand for cooling
- the average power demand for onboard electronics

This power demand is modeled as:

\[
\text{auxilliary power demand [W]} = \text{auxilliary power base demand [W]} + (\text{heating thermal demand [W]} \times \text{heating energy consumption [0-1]}) + (\text{cooling thermal demand [W]} \times \text{cooling energy consumption [0-1]})
\]

auxilliary power demand is summed over the driving time defined by the driving cycle and divided by the engine efficiency.

The power demand for heating varies between 200 Watts and 350 Watts depending on the car size. The power demand for cooling varies between 200 Watts and 350 Watts depending on the car size.

Note that, unlike battery electric vehicles, internal combustion engine vehicles satisfy the power demand in heating without the additional use of energy, because heating energy consumption = 0.

3.2.4 Tank-to-wheel energy

The sum of the motive and the auxilliary energy gives the tank-to-wheel energy (TtW energy) of the vehicle.

Parameters such as battery discharge efficiency, fuel cell system efficiency, drivetrain efficiency, engine efficiency and therefore, indirectly, TtW efficiency, have been calibrated to obtain TtW energy figures that fit what is observed in reality.

For 2010 and 2017 vehicles, the tank-to-wheel energy use (TtW energy) and underlying parameters have been calibrated against the database from the Monitoring of CO2 emissions from passenger cars program from the European
Environment Agency. This database lists energy and emission measurement for each new passenger car registered in the European Union, based on the NEDC and WLTC driving cycles.

Tank-to-wheel energy calibration for 2010 vehicles

Tank-to-wheel energy calibration for 2017 vehicles
For the year 2000, such energy and emission measurement data was not available. Hence, we relied on the International Council on Clean Transportation data that provides historical time series on the measured fuel efficiency of diesel and petrol engines based on the WLTC driving cycle, including its evolution between 2000 and 2010 (-20%). Therefore, the underlying parameters of TtW efficiency have been adjusted to produce TtW energy figures about 20% more important than those observed in 2010.

Here is a comparison of the TtW energy based on the WLTC driving cycle for 2000, 2010 and 2017 vehicles:
Knowing the tank-to-wheel energy requirement allows to calculate the range (in km) of a vehicle on a full tank since:

\[
\text{range [km]} = \frac{\text{fuel mass [kg]} \times \text{LHV fuel MJ per kg [Mj/kg]} \times 1000}{\text{TtW energy}}
\]

In the case of battery electric vehicles and hybrid vehicles, things are similar:

\[
\text{range [km]} = \frac{\text{electric energy stored [kWh]} \times \text{battery DoD [%]} \times 3.6 \times 1000}{\text{TtW energy}}
\]

The following lower heating values (LHV) for the liquid and gaseous fuels, in Mj/kg, are used:

- conventional gasoline: 42.4
- conventional diesel: 42.8
- compressed natural gas: 55.5
- hydrogen: 120

Those can be changed by modifying the value of the \text{LHV fuel MJ per kg in array} before passing it to \text{CarModel}. For example, we can decrease the LHV of diesel:
dict_param = {('Powertrain', 'ICEV-d', 'all', 'LHV fuel MJ per kg', 'none'): {
  (2000, 'loc'): 44,
  (2010, 'loc'): 44,
  (2017, 'loc'): 44,
  (2040, 'loc'): 44
}}
modify_xarray_from_custom_parameters(dict_param, array)

### 3.2.5 How can I override the tank-to-wheel efficiency?

With **carculator online**:

You cannot directly override TtW efficiency. However, you can adjust any of the four parameters affecting TtW efficiency in the Tank-to-wheel efficiency section.

![Powertrain Efficiency Table](image)

With **carculator**:

After having created the CarModel() object and executed the `set_all()` method, you can override the calculated TtW efficiency value and recalculate TtW energy with the `calculate_ttw_energy()` method. Here is an example for a diesel car of medium size in 2020, for which we want to set the TtW efficiency at 30% (instead of 24%):

```python
cm = CarModel(array, cycle='WLTC')
cm.set_all()
cm.array.loc[dict(parameter="TtW efficiency",
  powertrain="ICEV-d",
  year=2020,
  size="Medium") = 0.3
cm.calculate_ttw_energy()
```

You can also adjust any of the input parameters that affect TtW efficiency, namely battery discharge efficiency (for battery electric cars only), fuel cell stack efficiency (for fuel cell cars only), engine efficiency and drivetrain efficiency.

### 3.2.6 If I know already the fuel consumption of a vehicle, can I override it?

With **carculator online**:

Currently, it is not possible to modify directly the parameter TtW energy, as it would be recalculated. In order to do so, you need to use instead the Python library carculator (see next section):

#### 3.2. Tank-to-wheel energy consumption
With `carculator`:

Yes. After having created the CarModel() object and executed the `set_all()` method, you can override the calculated TtW energy value (in kilojoules). Here is an example for a diesel car of medium size in 2020:

```python
cm = CarModel(array, cycle='WLTC')
cm.set_all()
cm.array.loc[dict(parameter="TtW energy",
    powertrain="ICEV-d",
    year=2020,
    size="Medium")] = 2800
```

### 3.3 Fuel blends

The user can define fuel blends. The following fuel types are available, along with their lower heating value (in MJ/kg) and CO2 emission factor (kg CO2/kg fuel).

- 'electrolysis'
- 'smr - natural gas'
- 'smr - natural gas with CCS'
- 'smr - biogas'
- 'smr - biogas with CCS'
- 'coal gasification'
- 'wood gasification'
- 'wood gasification with CCS'

#### 3.3.1 Natural gas technologies

- 'cng' (55.5 MJ/kg, 2.65 kg CO2/kg CNG)
- 'biogas' (55.5 MJ/kg, 2.65 kg CO2/kg CNG)
- 'syngas' (55.5 MJ/kg, 2.65 kg CO2/kg CNG)

#### 3.3.2 Diesel technologies

- 'diesel' (42.8 MJ/kg, 3.14 kg CO2/kg)
- 'biodiesel - algae' (31.7 MJ/kg, 2.85 kg CO2/kg)
- 'biodiesel - cooking oil' (31.7 MJ/kg, 2.85 kg CO2/kg)
- 'synthetic diesel' (43.3 MJ/kg, 3.16 kg CO2/kg)

#### 3.3.3 Petrol technologies

- 'petrol' (42.4 MJ/kg, 3.18 kg CO2/kg)
- 'bioethanol - wheat straw' (26.8 MJ/kg, 1.91 kg CO2/kg)
- 'bioethanol - maize starch' (26.8 MJ/kg, 1.91 kg CO2/kg)
• ‘bioethanol - sugarbeet’ (26.8 MJ/kg, 1.91 kg CO2/kg)
• ‘bioethanol - forest residues’ (26.8 MJ/kg, 1.91 kg CO2/kg)
• ‘synthetic gasoline’ (42.4 MJ/kg, 3.18 kg CO2/kg)

Once the fuel blend is defined, the range is calculated once again, now considering the new energy amount stored in the tank. Therefore, a car solely running on bio-ethanol will have a reduced range, increasing the fuel consumption and emissions related to the growing of crops and supply of fuel. The tailpipe CO2 emissions may not necessarily increase as biofuels have generally lower CO2 emission factors.

It is important to note that CO2 emissions of biogenic origin from biofuels are characterized with a similar Global Warming Potential factor as those for conventional fossil fuels. However, CO2 uptake is considered during biomass growth.

### 3.4 Fuel-related direct emissions

Carbon dioxide emissions from fuel combustion are calculated based on the fuel blend defined by the user (see above).

\[
\text{carbon dioxide emission [kg/km]} = \text{CO2}_\text{fuel} \times \text{fuel mass [kg]} \times \frac{\text{share}_\text{fuel}}{\text{range [km]}}
\]

This is calculated for every fuel type found in the blend (primary and secondary fuel).

Sulfur dioxide emissions are calculated based on the fuel consumption and the sulfur content of the fuel. The sulfur content of the fuel is defined for over 90 countries. It is assumed that countries which currently have a fuel with a high sulfur content will have it down to 50 ppm by 2050. The past and current sulfur content in fuels for European countries is provided by the Handbook Emission Factors for Road Transport. For other countries, it is provided by Xier et al. 2020.

This maps shows the sulfur content the model considers in 2020 for on-road diesel fuel. ![Sulfur content in on-road diesel fuel in 2020](https://github.com/romainsacchi/carculator/raw/master/docs/diesel_sulfur_map.png)

Other emissions based on fuel combustion are considered, from Spielmann et al., Transport Services Data v.2 (2007). However those only apply when conventional diesel or conventional gasoline is burnt:

- Cadmium
- Chromium and Chromium VI
- Copper
- Nickel
- Selenium
- Zinc

### 3.5 Hot pollutants emissions

Carculator quantifies the emissions of the following substances:

- Hydrocarbons
- Carbon monoxide
- Nitrogen oxides
• Particulate matters
• Methane
• NMVOC
• Lead
• Sulfur dioxide
• Dinitrogen oxide
• Ammonia
• Benzene
• Xylene
• Toluene
• Formaldehyde
• Acetyldehyde

It does so by correlating the emission of a substance with the fuel consumption of the vehicle for each second of the driving cycle.

The emission of substances function of the fuel consumption is sourced from the Handbook Emission Factors for Road Transport for vehicles of various emission standards (from Euro-0 to Euro-6d).

Here is such correlation (g of pollutant emitted versus MJ of energy consumed) plotted for diesel-run vehicles:
3.5. Hot pollutants emissions
Given the years selected, the corresponding emission factors are chosen:

- before 1993: Euro-0
- between 1993 and 1997: Euro-1
- between 2001 and 2005: Euro-3
- between 2006 and 2010: Euro-4
- between 2011 and 2014: Euro-5
- between 2015 and 2017: Euro-6-ab
- between 2017 and 2019: Euro-6-c
- between 2019 and 2020: Euro-6-d-temp
- 2020 and beyond: Euro-6-d

Emissions are summed over the duration of the driving cycle. Furthermore, some driving cycles have distinct parts corresponding to different driving environments: urban, suburban, highway, etc. These driving environments are used to further split emissions and be more precise on the fate of the substances and the exposure of the population.

### 3.6 Noise emissions

Given the driving cycle, where speed [km/h] is given along time [s], noise levels (in dB) are calculated for each of the 8 octaves (or frequency ranges) to obtain propulsion and rolling noise levels, based on the CNOSSOS model.

For electric engines, special coefficients apply.

Also, electric cars are added a warning signal of 56 dB at speed levels lower than 20 km/h. Hybrid cars are assumed to use an electric engine up to a speed level of 30 km/h, beyond which the combustion engine is used. The sum of the propulsion and rolling noise levels is converted to noise power (in joules) and divided by the distance driven to obtain the noise power par km driven (joules/km), for each octave.

Noise emissions are further compartmented into urban, sub-urban and rural geographical environments based on speed intervals given by the driving cycle. The study from Cucurachi et al. 2014 is used to characterize noise emissions against midpoint and endpoint indicators, expressed in Person-Pascal-second and DALY's, respectively.

Overall, propulsion noise emissions dominate in urban environments, thereby justifying the use of electric cars in that regard. In sub-urban and rural environments, rolling noise emissions dominate above a speed level around 50 km/h.

It is important to note that although calculator differentiates noise coefficients by powertrain (internal combustion engine, electric and hybrid), it is not possible to differentiate them by size class. Therefore, the noise produced by a small vehicle will be similar to that produced by a large vehicle.

Finally, the noise coefficients used correspond to day time exposure only.

### 3.7 Vehicle inventory

This section presents the vehicle inventory once its size, mass, energy consumption and emissions are known.
Table 1: Vehicle inventory

<table>
<thead>
<tr>
<th>Applies to</th>
<th>Component</th>
<th>Formula</th>
<th>Dataset name</th>
<th>Location</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Glider</td>
<td>glider base mass / lifetime kilometers</td>
<td>market for glider, passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>All</td>
<td>Glider lightweight</td>
<td>(glider base mass x lightweighting) / lifetime kilometers</td>
<td>lightweighting</td>
<td>GLO</td>
<td>PSI</td>
</tr>
<tr>
<td>All</td>
<td>Maintenance</td>
<td>curb mass / 1600 / 150000</td>
<td>maintenance, passenger car</td>
<td>RER</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Dismantling of electric car</td>
<td>curb mass x (1 - combustion power share) / 1180 / lifetime kilometers</td>
<td>market for manual dismantling of used electric passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>ICEV/HEV</td>
<td>Dismantling of internal combustion engine car</td>
<td>curb mass x combustion power share / 1600 / lifetime kilometers</td>
<td>market for manual dismantling of used passenger car with internal combustion engine</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Battery charger</td>
<td>charger mass / lifetime kilometers</td>
<td>market for charger, electric passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Battery converter</td>
<td>converter mass / lifetime kilometers</td>
<td>market for converter, for electric passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Battery motor</td>
<td>electric engine mass / lifetime kilometers</td>
<td>market for electric motor, electric passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Battery inverter</td>
<td>inverter mass / lifetime kilometers</td>
<td>market for inverter, for electric passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Battery power distribution unit</td>
<td>power distribution unit mass / lifetime kilometers</td>
<td>market for power distribution unit, for electric passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>BEV/FCEV</td>
<td>Dismantling of electric powertrain</td>
<td>charger mass + converter mass + inverter mass + power distribution unit mass + electric engine mass + fuel cell stack mass + fuel cell ancillary BoP mass + fuel cell essential BoP mass + battery cell mass + battery BoP mass) / lifetime kilometers</td>
<td>market for used powertrain from electric passenger car, manual dismantling</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Applies to</th>
<th>Component</th>
<th>Formula</th>
<th>Dataset name</th>
<th>Location</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICEV/HEV</td>
<td>Internal combustion engine</td>
<td>((\text{combustion engine mass} + \text{&quot;powertrain mass&quot;}) / \text{lifetime kilometers})</td>
<td>market for internal combustion engine, passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>FCEV</td>
<td>Fuel cell stack, ancillary BoP</td>
<td>fuel cell ancillary BoP mass / lifetime kilometers</td>
<td>Ancillary BoP</td>
<td>GLO</td>
<td>Evangelisti et al. 2017. Journal of Cleaner Production 142. DOI: 10.1016/j.jclepro.2016.11.159</td>
</tr>
<tr>
<td>FCEV</td>
<td>Fuel cell stack</td>
<td>fuel cell stack mass / lifetime kilometers</td>
<td>Stack</td>
<td>GLO</td>
<td>Evangelisti et al. 2017. Journal of Cleaner Production 142. DOI: 10.1016/j.jclepro.2016.11.159</td>
</tr>
<tr>
<td>BEV/FCEV/HEV/PHEV</td>
<td>Battery BoP mass (\times (1 + \text{battery lifetime replacements})) / lifetime kilometers</td>
<td>Battery BoP</td>
<td>GLO</td>
<td>Schmidt et al. 2019. Environ. Sci. Technol. 53. DOI: 10.1021/acs.est.8b05313</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Applies to</th>
<th>Component</th>
<th>Formula</th>
<th>Dataset name</th>
<th>Location</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEV/FCEV</td>
<td>Battery cell</td>
<td>( \text{battery cell mass} \times (1 + \text{battery lifetime replacements}) / \text{lifetime kilometers} )</td>
<td>Battery cell NMC/LFP/NCA</td>
<td>GLO</td>
<td>Schmidt et al. 2019. Environ. Sci. Technol. 53. DOI: 10.1021/acs.est.8b05313</td>
</tr>
<tr>
<td>ICEV-d/p</td>
<td>Fuel tank</td>
<td>( \text{fuel tank mass} / \text{lifetime kilometers} )</td>
<td>polyethylene production, high density, granulate</td>
<td>RER</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>ICEV-g</td>
<td>Fuel tank</td>
<td>( \text{fuel tank mass} / \text{lifetime kilometers} )</td>
<td>glass fibre reinforced plastic production, polyamide, injection moulded</td>
<td>RER</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>FCEV</td>
<td>Fuel tank</td>
<td>( \text{fuel tank mass} / \text{lifetime kilometers} )</td>
<td>Fuel tank, compressed hydrogen gas, 700 bar</td>
<td>GLO</td>
<td>Evangelisti et al. 2017. Jour- nal of Cleaner Production 142. DOI: 10.1016/j.jclepro.2016.11.159</td>
</tr>
<tr>
<td>FCEV</td>
<td>Hydrogen supply</td>
<td>( \text{fuel mass/range} )</td>
<td>Hydrogen, gaseous, 700 bar, from electrolysis/SMR NG w CCS/SMR NG wo CCS, at H2 fuelling station</td>
<td>RER</td>
<td></td>
</tr>
<tr>
<td>ICEV-g</td>
<td>Compressed gas supply</td>
<td>( \text{fuel mass/range} )</td>
<td>market for natural gas, from high pressure network/biogas/methane, from electrochemical methanation</td>
<td>GLO/RER</td>
<td>ecoinvent cutoff 3.6/PSI</td>
</tr>
<tr>
<td>ICEV-d</td>
<td>Diesel supply</td>
<td>( \text{fuel mass/range} )</td>
<td>market for diesel/biodiesel/synthetic diesel</td>
<td>RER</td>
<td>ecoinvent cutoff 3.6/PSI</td>
</tr>
<tr>
<td>ICEV-p</td>
<td>Gasoline supply</td>
<td>( \text{fuel mass/range} )</td>
<td>market for petrol, low-sulfur/bioethanol/synthetic gasoline</td>
<td>RER/PSI</td>
<td>ecoinvent cutoff 3.6/PSI</td>
</tr>
<tr>
<td>All</td>
<td>Road wear emissions</td>
<td>( \text{driving mass} \times 1e-08 )</td>
<td>market for road wear emissions, passenger car</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>All</td>
<td>Tyre wear emissions</td>
<td>( \text{driving mass} \times 6e-08 )</td>
<td>market for brake wear emissions, passenger cars</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Applies to</th>
<th>Component</th>
<th>Formula</th>
<th>Dataset name</th>
<th>Location</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Road</td>
<td>driving mass x 5.37e-7</td>
<td>market for road</td>
<td>GLO</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>ICEV/HEV/ PHEV emissions</td>
<td>(CO2 per kg fuel x fuel mass)/range</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICEV/HEV/ PHEV emissions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>Noise</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 3.8 Fuel pathways

Different fuel pathways can be selected for a given powertrain type. The table below lists them.
### Table 2: Fuel pathways

<table>
<thead>
<tr>
<th>Applies to</th>
<th>Fuel type</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICEV-p</td>
<td>Gasoline, low-sulfur</td>
<td>Regular gasoline, from the refining of crude oil.</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>ICEV-p</td>
<td>Bioethanol - forest residues</td>
<td>Second generation bioethanol. Francesco Cozzolini. 2018. PSI.</td>
<td></td>
</tr>
<tr>
<td>ICEV-p</td>
<td>Bioethanol - sugarbeet</td>
<td>First generation bioethanol. Francesco Cozzolini. 2018. PSI.</td>
<td></td>
</tr>
<tr>
<td>ICEV-p</td>
<td>Bioethanol - maize starch</td>
<td>First generation bioethanol. Francesco Cozzolini. 2018. PSI.</td>
<td></td>
</tr>
<tr>
<td>ICEV-d</td>
<td>Diesel</td>
<td>Regular diesel, from the refining of crude oil.</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
<tr>
<td>ICEV-d</td>
<td>Biodiesel - algae</td>
<td>Third generation biofuel from micro-algae. Francesco Cozzolini. 2018. PSI.</td>
<td></td>
</tr>
<tr>
<td>ICEV-d</td>
<td>Biodiesel - cooking oil</td>
<td>Second generation biodiesel from used cooking oil. Francesco Cozzolini. 2018. PSI.</td>
<td></td>
</tr>
<tr>
<td>ICEV-d</td>
<td>Synthetic</td>
<td>Based on a Fischer-Tropsch process. van der Giesen et al. 2014.</td>
<td></td>
</tr>
<tr>
<td>ICEV-g</td>
<td>Compressed natural gas</td>
<td>High pressure.</td>
<td>ecoinvent cutoff 3.6</td>
</tr>
</tbody>
</table>

### 3.9 Electricity mixes for battery charging and hydrogen production

The `carculator` has national electricity mixes for more than 80 countries, gathered from the following sources:

- European Union State members and the UK: EU Reference Scenario 2016

• African countries: TEMBA model

• Other countries: IEA World Energy outlook 2017

Unless a specific electricity mix is indicated by the user, such national mixes are used when modeling the energy chain for battery and fuel cell electric vehicles (BEV, FCEV), for battery charging and the production of hydrogen via electrolysis, respectively.

Knowing the production year of the vehicle, considered to be its first year of use, as well as its annual mileage, the number of years of use is calculated. Hence, the electricity mix used is the kilometer-distributed mix over the years of use of the vehicle.

If the annual mileage of the vehicle is evenly distributed throughout its lifetime, the electricity mix used therefore equals the average of the year-by-year national mixes comprised between Year 0 and Year 0 + the number of years of use.

3.10 Background inventory

Besides datasets adapted from the literature, vehicle inventories also rely on a number of datasets provided by the database ecoinvent cutoff 3.6 such as “market for glider, passenger car”, “market for diesel”, etc.

However calculator does not directly use the database as is: the database and its datasets are modified according to projections provided by the Integrated Assessment Model REMIND.

REMIND provides projections for different regions in the world until 2150, following different energy scenarios, described here.

Projection outputs include the expected change over time in efficiency for power plants, steel making, cement production, etc.

Using the Python library premise, we produce a number of ecoinvent databases with the inclusion of REMIND projections, so that future improvements in electricity production, among others, propagate into the datasets involved in the vehicles’ inventories.

calculator comes with pre-calculated impact values for ecoinvent datasets from the following databases:

• 2005 - ecoinvent-REMIND, SSP2-Base
• 2010 - ecoinvent-REMIND, SSP2-Base
• 2020 - ecoinvent-REMIND, SSP2-Base
• 2030 - ecoinvent-REMIND, SSP2-Base
• 2040 - ecoinvent-REMIND, SSP2-Base
• 2050 - ecoinvent-REMIND, SSP2-Base
• 2005 - ecoinvent-REMIND, SSP2-PkBudg1100
• 2010 - ecoinvent-REMIND, SSP2-PkBudg1100
• 2020 - ecoinvent-REMIND, SSP2-PkBudg1100
• 2030 - ecoinvent-REMIND, SSP2-PkBudg1100
• 2040 - ecoinvent-REMIND, SSP2-PkBudg1100
• 2050 - ecoinvent-REMIND, SSP2-PkBudg1100
Depending on the year of analysis and the energy scenario demanded, carculator picks the corresponding datasets. If year of analysis in between the available years is demanded, a linear interpolation is used.

With carculator online, the results provided only use the “SSP2-Base” energy scenario of REMIND, projecting a global atmospheric temperature increase by 3.5 degrees Celsius by 2100.
4.1 Driving cycle, velocity and acceleration

Beside custom driving cycles, there are eleven default driving cycles to select from:

- WLTC
- WLTC 3.1
- WLTC 3.2
- WLTC 3.3
- WLTC 3.4
- CADC Urban
- CADC Road
- CADC Motorway
- CADC Motorway 130
- CADC
- NEDC

They are needed to calculate a number of things, such as:

- velocity, driving distance, driving time, and acceleration,
- but also hot pollutant and noise emissions.

Manually, such parameters can be obtained the following way:

```python
import pandas as pd
import numpy as np

# Retrieve the driving cycle WLTC 3 from the UNECE
driving_cycle = pd.read_excel('http://www.unece.org/fileadmin/DAM/trans/doc/2012/wp29grpe/WLTP-DHC-12-07e.xls',
                             verbatim=False)
```

(continues on next page)
Carculator, Release 1.0.0

```python
# Calculate velocity (km/h -> m/s)
velocity = driving_cycle['km/h'].values * 1000 / 3600

# Retrieve driving distance (-> km)
driving_distance = velocity.sum() * 1000

# Retrieve driving time (-> s)
driving_time = len(driving_cycle.values)

# Retrieve acceleration by calculating the delta of velocity per time interval of 2 -> seconds
acceleration = np.zeros_like(velocity)
acceleration[1:-1] = (velocity[2:] - velocity[:-2]) / 2
```

Using `carculator`, these parameters can be obtained the following way:

```python
from carculator.energy_consumption import EnergyConsumptionModel
ecm = EnergyConsumptionModel('WLTC')

# Access the driving distance
ecm.velocity.sum() * 1000

# Access the driving time
len(ecm.velocity)

# Access the acceleration
ecm.acceleration
```

Both approaches should return identical results:

```python
print(np.array_equal(velocity, ecm.velocity))
print(driving_distance == ecm.velocity.sum()*1000)
print(driving_time == len(ecm.velocity))
print(np.array_equal(acceleration, ecm.acceleration))
```

True
True
True
True

And the acceleration returned by carculator should equal the values given by the UNECE:

```python
np.array_equal(np.around(ecm.acceleration, 4), np.around(driving_cycle['m/s²'].values, 4))
```

True

Which can also be verified visually:

```python
plt.plot(driving_cycle['m/s²'].values, label='UNECE')
plt.plot(acceleration, label='Manually calculated')
plt.plot(ecm.acceleration, label='carculator', alpha=0.6)
plt.legend()
plt.ylabel('m/s²')
plt.xlabel('second')
```
4.2 Car and components masses

CarModel sizes and “builds” the vehicles. The vehicles attributes are accessed in the array attribute of the CarModel class. Filters like vehicle size class, year of manufacture and powertrain technology are convenient to use. A relevant calculated parameter is the driving mass, as it is determinant for the energy required to overcome rolling resistance, the drag, but also the energy required to move the vehicle over a given distance – kinetic energy, which is altogether defined as the tank to wheel energy, stored under the parameter TtW_energy.

Parameters such as total cargo mass, curb mass and driving mass, can be obtained the following way, for a 2017 battery electric SUV:

```python
cm.array.sel(size='SUV', powertrain='BEV', year=2017, parameter=['cargo mass','curb mass', 'driving mass']).values
```

array([[ 20. ],
       [1719.56033224],
       [1874.56033224]])

One can check whether total cargo mass is indeed equal to cargo mass plus the product of the number of passengers and the average passenger weight:

```python
total_cargo, cargo, passengers, passengers_weight = cm.array.sel(size='SUV',
                                                                 powertrain='BEV',
                                                                 year=2017,
                                                                 parameter=['total cargo mass','cargo mass','average passengers', 'average passenger mass']).values
print('Total cargo of {} kg, with a cargo mass of {} kg, and {} passengers of {} kg.Individual weight of {} kg.'.format(total_cargo[0], cargo[0], passengers[0], passengers_weight[0]))
print(total_cargo == cargo+(passengers * passengers_weight))
```

Total cargo of 155.0 kg, with a cargo mass of 20.0 kg, and 1.8 passengers of 75.0 kg.
However, most of the driving mass is explained by the curb mass:

```python
plt.pie(np.squeeze(cm.array.sel(size='SUV', powertrain='BEV', year=2017,
                          parameter=['total cargo mass', 'curb mass']).values).tolist(), labels=['Total cargo mass', 'Curb mass'])
plt.show()
```

Here is a split between the components making up for the curb mass. One can see that, in the case of a battery electric SUV, most of the weight comes from the glider as well as the battery cells. On an equivalent diesel powertrain, the mass of the glider base is comparatively more important:

```python
l_param=['fuel mass','charger mass','converter mass','glider base mass','inverter mass','power distribution unit mass',
         'combustion engine mass','electric engine mass','powertrain mass','fuel cell stack mass',
         'fuel cell ancillary BoP mass','fuel cell essential BoP mass','battery cell mass','battery BoP mass','fuel tank mass']

colors = ['yellowgreen','red','gold','lightskyblue','white','lightcoral','blue','pink','darkgreen','yellow','grey','violet','magenta','cyan', 'green']

BEV_mass = np.squeeze(cm.array.sel(size='SUV', powertrain='BEV', year=2017,
                                    parameter=l_param).values)
percent = 100.*BEV_mass/BEV_mass.sum()

f = plt.figure(figsize=(15,10))
ax = f.add_subplot(121)
ax.pie(BEV_mass, colors=colors, startangle=90, radius=1.2)
labels = ['
    {0} - (1:1.2f) %'.format(i,j) for i,j in zip(l_param, percent)]
```

(continues on next page)
sort_legend = True
if sort_legend:
    patches, labels, dummy = zip(*sorted(zip(patches, labels, BEV_mass),
                                         key=lambda x: x[2],
                                         reverse=True))

ax.legend(patches, labels, loc='upper left', bbox_to_anchor=(-0.1, 1.),
          fontsize=8)

ICEV_d_mass = np.squeeze(cm.array.sel(size='SUV', powertrain='ICEV-d', year=2017,
                                      parameter=l_param).values)
percent = 100. * ICEV_d_mass / ICEV_d_mass.sum()

ax2 = f.add_subplot(122)
patches, texts = ax2.pie(ICEV_d_mass, colors=colors, startangle=90, radius=1.2)
ax2.set_title('ICE-d SUV')
labels = ['{0} - {1:1.2f}%'.format(i, j) for i, j in zip(l_param, percent)]

sort_legend = True
if sort_legend:
    patches, labels, dummy = zip(*sorted(zip(patches, labels, ICEV_d_mass),
                                         key=lambda x: x[2],
                                         reverse=True))

ax2.legend(patches, labels, loc='upper left', bbox_to_anchor=(-0.1, 1.),
           fontsize=8)

plt.subplots_adjust(wspace=1)
plt.show()

The curb mass returned by calculator for the year 2010 and 2017 is further calibrated against manufacturers’ data.
per vehicle size class and powertrain technology. For example, we use the car database Car2db (https://car2db.com/) and load all the vehicles produced between 2015 and 2019 (11,500+ vehicles) to do the curb mass calibration for 2017 vehicles. The same exercise is done with vehicles between 2008 and 2012 to calibrate the curb mass of given by carculator for vehicles in 2010.

4.3 Tank-to-wheel energy

The European Commission monitors all new registered cars for emissions and energy consumption according to the WLTC driving cycle (v.3). See: https://www.eea.europa.eu/data-and-maps/data/co2-cars-emission-16

However, this database does not directly give energy consumption. But we can use CO2 emission measurements with the lower heating value of the corresponding fuel to back-calculate the energy consumption. Here is an example, where the 2017 vehicle fuel consumption is calibrated against 15,000,000+ measurements found in the EU database for vehicles registered in 2018 (the 2017 database could not be used as emissions were not measured according to WLTC just yet).
4.4 End-of-pipe CO2 emissions

Similarly, we can plot the CO2 measurements from the EU emissions monitoring database against the values returned by carculator for fossil fuel-powered vehicles.
There seems to be a general alignment between measured figures from the EU emissions monitoring programme and the figures produced by carculator.
5.1 Car Input Parameter

class carculator.car_input_parameters.CarInputParameters(parameters=None, extra=None, limit=None)

A class used to represent vehicles with associated type, size, technology, year and parameters.

This class inherits from NamedParameters, located in the klausen package. It sources default parameters for all vehicle types from a dictionary in default_parameters and format them into an array following the structured described in the klausen package.

Variables

- **sizes (list)** – List of string items e.g., ['Large', 'Lower medium', 'Medium', 'Mini', 'SUV', 'Small', 'Van']
- **powertrains (list)** – List of string items e.g., ['BEV', 'FCEV', 'HEV-p', 'ICEV-d', 'ICEV-g', 'ICEV-p', 'PHEV-c', 'PHEV-e']
- **parameters (list)** – List of string items e.g., ['Benzene', 'CH4', 'CNG tank mass intercept',...]
- **years (list)** – List of integers e.g., [2017, 2040]
- **metadata (dict)** – Dictionary for metadata.
- **values (dict)** – Dictionary for storing values, of format {'param':[value]}
- **iterations (int)** – Number of iterations executed by the method stochastic(). None if static() used instead.

add_car_parameters (parameters)

Split data and metadata according to klausen convention.

The parameters are split into the metadata and values attributes of the CarInputParameters class by the add_parameters() method of the parent class.

Parameters **parameters (dict)** – A dictionary that contains parameters.
5.2 Array

carculator.array.fill_xarray_from_input_parameters(cip, sensitivity=False, scope=None)

Create an xarray labeled array from the sampled input parameters.

This function extracts the parameters’ names and values contained in the parameters attribute of the CarInputParameters class in car_input_parameters and insert them into a multi-dimensional numpy-like array from the xarray package (http://xarray.pydata.org/en/stable/).

Parameters

• cip – Instance of the CarInputParameters class in car_input_parameters.

• scope – a dictionary to narrow down the scope of vehicles to consider

Returns tuple, xarray.DataArray

• tuple (size_dict, powertrain_dict, parameter_dict, year_dict)

• array

Dimensions of array:

0. Vehicle size, e.g. “small”, “medium”. str.
2. Year. int.
3. Samples.

carculator.array.modify_xarray_from_custom_parameters(fp, array)

Override default parameters values in xarray based on values provided by the user.

This function allows to override one or several default parameter values by providing either:

• a file path to an Excel workbook that contains the new values

• or a dictionary

The dictionary must be of the following format:

```python
{
    (parameter category,
    powertrain,
    size,
    parameter name,
    uncertainty type): {
        (year, 'loc'): value,
        (year, 'scale'): value,
        (year, 'shape'): value,
        (year, 'minimum'): value,
        (year, 'maximum'): value
    }
}
```

For example:
Parameters $\mathbf{fp}$ ($\text{str or dict}$) – File path of workbook with new values or dictionary.

### 5.3 Driving cycle

carculator.driving_cycles.get_standard_driving_cycle (name='WLTC')

Get driving cycle data as a Pandas Series.

Driving cycles are given as km/h per second up to 3200 seconds.

Parameters name (str) – The name of the driving cycle. WLTC (Worldwide harmonized Light vehicles Test Cycles) is chosen by default if :param name: left unspecified.

name should be one of:

- WLTC
- WLTC 3.1
- WLTC 3.2
- WLTC 3.3
- WLTC 3.4
- CADC Urban
- CADC Road
- CADC Motorway
- CADC Motorway 130
- CADC
- NEDC

Returns A pandas DataFrame object with driving time (in seconds) as index, and velocity (in km/h) as values.

Return type pandas.Series

### 5.4 Energy consumption

class carculator.energy_consumption.EnergyConsumptionModel (cycle, $\rho_{air}=1.204$, gradient=\text{None})

Calculate energy consumption of a vehicle for a given driving cycle and vehicle parameters.

Based on a selected driving cycle, this class calculates the acceleration needed and provides two methods:
• `aux_energy_per_km()` calculates the energy needed to power auxiliary services

• `motive_energy_per_km()` calculates the energy needed to move the vehicle over 1 km

Acceleration is calculated as the difference between velocity at \( t_2 \) and velocity at \( t_0 \), divided by 2. See for example: [http://www.unece.org/fileadmin/DAM/trans/doc/2012/wp29grpe/WLTP-DHC-12-07e.xls](http://www.unece.org/fileadmin/DAM/trans/doc/2012/wp29grpe/WLTP-DHC-12-07e.xls)

Parameters

- **cycle** ([pandas.Series](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html)) — Driving cycle. Pandas Series of second-by-second speeds (km/h) or name (str) of cycle e.g., “WLTC”, “WLTC 3.1”, “WLTC 3.2”, “WLTC 3.3”, “WLTC 3.4”, “CADC Urban”, “CADC Road”, “CADC Motorway”, “CADC Motorway 130”, “CADC”, “NEDC”.

- **rho_air** ([float](https://docs.python.org/3/library/stdtypes.html#builtins.float)) — Mass per unit volume of air. Set to (1.225 kg/m³) by default.

- **gradient** ([numpy.ndarray](https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html)) — Road gradient per second of driving, in degrees. None by default. Should be passed as an array of length equal to the length of the driving cycle.

Variables

- **rho_air** ([float](https://docs.python.org/3/library/stdtypes.html#builtins.float)) — Mass per unit volume of air. Value of 1.204 at 23°C (test temperature for WLTC).


- **acceleration** ([numpy.ndarray](https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html)) — Time series of acceleration, calculated as increment in velocity per interval of 1 second, in meter per second².

**aux_energy_per_km(aux_power, efficiency=1)**

Calculate energy used other than motive energy per km driven.

Parameters

- **aux_power** ([int](https://docs.python.org/3/library/stdtypes.html#builtins.int)) — Total power needed for auxiliaries, heating, and cooling (W)

- **efficiency** ([float](https://docs.python.org/3/library/stdtypes.html#builtins.float)) — Efficiency of electricity generation (dimensionless, between 0.0 and 1.0). Battery electric vehicles should have efficiencies of one here, as we account for battery efficiencies elsewhere.

Returns total auxiliary energy in kJ/km

Return type float

**motive_energy_per_km(driving_mass, rr_coef, drag_coef, frontal_area, ttw_efficiency, recuperation_efficiency=0, motor_power=0)**

Calculate energy used and recuperated for a given vehicle per km driven.

**param driving_mass** Mass of vehicle (kg)

**type driving_mass** int

**param rr_coef** Rolling resistance coefficient (dimensionless, between 0.0 and 1.0)

**type rr_coef** float

**param drag_coef** Aerodynamic drag coefficient (dimensionless, between 0.0 and 1.0)

**type drag_coef** float

**param frontal_area** Frontal area of vehicle (m²)

**type frontal_area** float

**param ttw_efficiency** Efficiency of translating potential energy into motion (dimensionless, between 0.0 and 1.0)
type ttw_efficiency float

param recuperation_efficiency Fraction of energy that can be recuperated (dimensionless, between 0.0 and 1.0). Optional.

type recuperation_efficiency float

param motor_power Electric motor power (watts). Optional.

type motor_power int

Power to overcome rolling resistance is calculated by:

\[ g v M C_r \]

where \( g \) is 9.81 (m/s²), \( v \) is velocity (m/s), \( M \) is mass (kg), and \( C_r \) is the rolling resistance coefficient (dimensionless).

Power to overcome air resistance is calculated by:

\[ \frac{1}{2} \rho_{\text{air}} v^3 A C_d \]

where \( \rho_{\text{air}} \) is 1.225 (kg/m³), \( v \) is velocity (m/s), \( A \) is frontal area (m²), and \( C_d \) is the aerodynamic drag coefficient (dimensionless).

returns net motive energy (in kJ/km)

rtype float

5.5 Car Model

class carculator.model.CarModel(array, mappings=None, cycle=None, gradient=None)

This class represents the entirety of the vehicles considered, with useful attributes, such as an array that stores all the vehicles parameters.

Variables

- array (xarray.DataArray) – multi-dimensional numpy-like array that contains parameters’ value(s)
- mappings (dict) – Dictionary with names correspondence
- ecm (coarse.energy_consumption.EnergyConsumptionModel) – instance of EnergyConsumptionModel class for a given driving cycle

adjust_cost ()

This method adjusts costs of energy storage over time, to correct for the overly optimistic linear interpolation between years.

adjust_fuel_mass ()

This method adjusts the fuel mass over the years, to correct for the linear interpolation between years.

calculate_cost_impacts (sensitivity=False, scope=None)

This method returns an array with cost values per vehicle-km, sub-divided into the following groups:

- Purchase
- Maintentance
- Component replacement
- Energy
- Total cost of ownership

**Returns** A xarray array with cost information per vehicle-km

**Return type** xarray.core.dataarray.DataArray

calculate_ttw_energy()

This method calculates the energy required to operate auxiliary services as well as to move the car. The sum is stored under the parameter label “TtW energy” in *self.array*.

create_PHEV()

PHEV-p/d is the range-weighted average between PHEV-c-p/PHEV-c-d and PHEV-e.

drop_hybrid()

This method drops the powertrains PHEV-c-p, PHEV-c-d and PHEV-e as they were only used to create the PHEV powertrain. .returns: Does not return anything. Modifies *self.array* in place.

set_all(drop_hybrids=True, electric_utility_factor=None)

This method runs a series of other methods to obtain the tank-to-wheel energy requirement, efficiency of the car, costs, etc.

- set_component_masses()
- set_car_masses()
- set_power_parameters()

are inter-dependent. *powertrain_mass* depends on *power*, *curb_mass* is affected by changes in *powertrain_mass*, *combustion engine mass* and *electric engine mass*, and *power* is a function of *curb_mass*. The current solution is to loop through the methods until the increment in driving mass is inferior to 0.1%.

**Parameters**

- *drop_hybrids* – boolean. True by default. If False, the underlying vehicles used to build plugin-hybrid vehicles remain present in the array.

- *electric_utility_factor* – array. If an array is passed, its values are used to override the electric utility factor for plugin hybrid vehicles. If not, this factor is calculated using a relation described in *set_electric_utility_factor()*

**Returns** Does not return anything. Modifies *self.array* in place.

set_auxiliaries()

Calculates the power needed to operate the auxiliary services of the vehicle (heating, cooling).

The demand for heat and cold are expressed as a fraction of the heating and cooling capacities

set_battery_fuel_cell_replacements()

This methods calculates the fraction of the replacement battery needed to match the vehicle lifetime.

---

**Note:** if *car lifetime = 200000 (km)* and *battery lifetime = 190000 (km)* then
replacement battery = 0.05

---

**Note:** It is debatable whether this is realistic or not. Car owners may not decide to invest in a new battery if the remaining lifetime of the car is only 10000 km. Also, a battery lifetime may be expressed in other terms, e.g., charging cycles.

set_car_masses()

Define *curb mass*, *driving mass*, and *total cargo mass*.

- *curb mass* is the mass of the vehicle and fuel, without people or cargo.
• **total cargo mass** is the mass of the cargo and passengers.
• **driving mass** is the curb mass plus total cargo mass.

**Note:** driving mass = total cargo mass + driving mass

---

`set_electric_utility_factor(uf=None)`

Set the electric utility factor according to a study from Plötz et al. 2017 which correlated the share of km driven in electric-mode to the capacity of the battery. The argument `uf` is used to override this relation, if needed. `uf` must be a ratio between 0 and 1, for each.

`set_electricity_consumption()`

This method calculates the total electricity consumption for BEV and plugin-hybrid vehicles. Returns: Does not return anything. Modifies `self.array` in place.

`set_fuel_cell_parameters()`

Specific setup for fuel cells, which are mild hybrids. Must be called after `set_power_parameters()`.

`set_hot_emissions()`

Calculate hot pollutant emissions based on driving cycle. The driving cycle is passed to the `HotEmissionsModel` class and `get_emissions_per_powertrain()` return emissions per substance per second of driving cycle. Returns: Does not return anything. Modifies `self.array` in place.

`set_noise_emissions()`

Calculate noise emissions based on driving cycle. The driving cycle is passed to the `NoiseEmissionsModel` class and `get_sound_power_per_compartment()` returns emissions per compartment type (“rural”, “non-urban” and “urban”) per second of driving cycle.

Returns: Does not return anything. Modifies `self.array` in place.

`set_power_parameters()`

Set electric and combustion motor powers based on input parameter `power to mass ratio`.

---

### 5.6 Noise Model

`class carculator.noise_emissions.NoiseEmissionsModel(cycle, cycle_name)`

Calculate propulsion and rolling noise emissions for combustion, hybrid and electric vehicles, based on CNOS-SOS model.

**Parameters**
- `cycle (pandas.Series)` – Driving cycle. Pandas Series of second-by-second speeds (km/h) or name (str) of cycle e.g., “WLTC”, “WLTC 3.1”, “WLTC 3.2”, “WLTC 3.3”, “WLTC 3.4”, “CADC Urban”, “CADC Road”, “CADC Motorway”, “CADC Motorway 130”, “CADC”,”NEDC”.

`get_sound_power_per_compartment(powertrain_type)`

Calculate sound energy (in J/s) over the driving cycle duration from sound power (in dB). The sound energy sums are further divided into geographical compartments: urban, suburban and rural.

- **urban**: from 0 to 50 km/h
- **suburban**: from 51 km/h to 80 km/h
- **rural**: above 80 km/h

**Returns** Sound energy (in Joules) per km driven, per geographical compartment.
propulsion_noise(powertrain_type)
Calculate noise from propulsion engine and gearbox. Model from CNOSSOS-EU project (http://publications.jrc.ec.europa.eu/repository/bitstream/JRC72550/cnossos-eu%20jrc%20reference%20report_final_on%20line%20version_10%20august%202012.pdf)
For electric cars, special coefficients are applied from (Pallas et al. 2016)
Also, for electric cars, a warning signal of 56 dB is added when the car drives at 20 km/h or lower.

Returns A numpy array with propulsion noise (dB) for all 8 octaves, per second of driving cycle

rolling_noise()

Returns A numpy array with rolling noise (dB) for each 8 octaves, per second of driving cycle

5.7 Hot pollutants emissions

class carculator.hot_emissions.HotEmissionsModel(cycle, cycle_name)
Calculate hot pollutants emissions based on HBEFA 4.1 data, function of fuel consumption for vehicles with a combustion engine.

Parameters cycle (pandas.Series) – Driving cycle. Pandas Series of second-by-second speeds (km/h) or name (str) of cycle e.g., “WLTC”, “WLTC 3.1”, “WLTC 3.2”, “WLTC 3.3”, “WLTC 3.4”, “CADC Urban”, “CADC Road”, “CADC Motorway”, “CADC Motorway 130”, “CADC”, “NEDC”.

get_hot_emissions(powertrain_type, euro_class, energy_consumption, yearly_km)
Calculate hot pollutants emissions given a powertrain type (i.e., diesel, petrol, CNG) and a EURO pollution class, per air sub-compartment (i.e., urban, suburban and rural). Note that Nh3 and N2O emissions do not depend on the speed level. For those, average values observed across different traffic situations are used instead. Also includes cold start emissions. Cold start emissions are also given by HBEFA 4.1 and expressed in given in g/start. Cold start emissions are divided by the average trip length in Europe (20 km), to normalize them per km.

The emission sums are further divided into air compartments: urban, suburban and rural.

• urban: from 0 to 50 km/k
• suburban: from 51 km/h to 80 km/h
• rural: above 80 km/h

Parameters
• powertrain_type (str) – “diesel”, “petrol” or “CNG”
• euro_class (float) – integer, corresponding to the EURO pollution class
• energy_consumption (xarray) – tank-to-wheel energy consumption for each second of the driving cycle
• yearly_km – annual mileage, to calculate cold start emissions
Returns  Pollutants emission per km driven, per air compartment.

Return type  numpy.array

carculator.hot_emissions.get_hot_emission_factors()
Hot emissions factors extracted for trucks from HBEFA 4.1 detailed by size, powertrain and EURO class for each substance.

carculator.hot_emissions.get_non_hot_emission_factors()
Non hot emissions factors (cold start, evaporation, soak emissions) extracted for trucks from HBEFA 4.1 detailed by size, powertrain and EURO class for each substance.

5.8 Inventory calculation

class  carculator.inventory.InventoryCalculation(array, scope=None, background_configuration=None, scenario='SSP2-Base', method='recipe', method_type='midpoint')

Build and solve the inventory for results characterization and inventory export

Vehicles to be analyzed can be filtered by passing a scope dictionary. Some assumptions in the background system can also be adjusted by passing a background_configuration dictionary.

```python
scope = {
    'powertrain': ['BEV', 'FCEV', 'ICEV-p'],
}
bc = {'country': 'CH', # considers electricity network losses for Switzerland
      'custom electricity mix': [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                                [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                                [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                                [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
      # in this case, 100% nuclear for the second year
      'fuel blend': {
        'cng': { # specify fuel blend for compressed gas
                  'primary fuel': {
                      'type': 'biogas',
                      'share': [0.9, 0.8, 0.7, 0.6], # shares per year. Must total 1.
                  },
        'diesel': { # locally produced (i.e., same location as country of use).
                     'primary fuel': {
                                     'type': 'synthetic diesel',
                     }
        }
      }
```

(continues on next page)
The custom electricity mix key in the background_configuration dictionary defines an electricity mix to apply, under the form of one or several array(s), depending on the number of years to analyze, that should total 1, of which the indices correspond to:

- [0]: hydro-power
- [1]: nuclear
- [2]: natural gas
- [3]: solar power
- [4]: wind power
- [5]: biomass
- [6]: coal
- [7]: oil
- [8]: geothermal
If none is given, the electricity mix corresponding to the country specified in `country` will be selected. If no country is specified, Europe applies.

The `primary` and `secondary` fuel keys contain an array with shares of alternative petrol fuel for each year, to create a custom blend. If none is provided, a blend provided by the Integrated Assessment model REMIND is used, which will depend on the REMIND energy scenario selected.

Here is a list of available fuel pathways:

- electrolysis
- smr - natural gas
- smr - natural gas with CCS
- smr - biogas
- smr - biogas with CCS
- coal gasification
- wood gasification
- wood gasification with CCS
- wood gasification with EF
- wood gasification with EF with CCS
- atr - natural gas
- atr - natural gas with CCS
- atr - biogas
- atr - biogas with CCS
- cng
- biogas - sewage sludge biogas
- biowaste syngas
- diesel
- biodiesel - algae biodiesel
- cooking oil synthetic diesel
- synthetic diesel - energy allocation
- petrol
- bioethanol - wheat straw
- bioethanol - maize starch
- bioethanol - sugarbeet
- bioethanol - forest residues
- synthetic gasoline

**Variables**

- `array (CarModel.array)` – array from the CarModel class
- `scope` – dictionary that contains filters for narrowing the analysis
- `background_configuration` – dictionary that contains choices for background system
- `scenario` – REMIND energy scenario to use ("SSP2-Baseline": business-as-usual, "SSP2-PkBudg1100": limits cumulative GHG emissions to 1,100 gigatons by 2100, "static": no forward-looking modification of the background inventories).

"SSP2-Baseline" selected by default.

```python
create_electricity_market_for_battery_production()
```

This function fills in the column in `self.A` concerned with the electricity mix used for manufacturing battery cells.

```python
create_electricity_market_for_fuel_prep()
```

This function fills the electricity market that supplies battery charging operations and hydrogen production through electrolysis.

```python
create_fuel_markets(fuel_type, primary=None, secondary=None, tertiary=None, primary_share=None, secondary_share=None, tertiary_share=None)
```

This function creates markets for fuel, considering a given blend, a given fuel type and a given year. It also adds separate electricity input in case hydrogen from electrolysis is needed somewhere in the fuel supply chain.
define_electricity_mix_for_fuel_prep()
This function defines a fuel mix based either on user-defined mix, or on default mixes for a given country. The mix is calculated as the average mix, weighted by the distribution of annually driven kilometers. :return:

define_fuel_blends()
This function defines fuel blends from what is passed in background_configuration. It populates a dictionary self.fuel_blends that contains the respective shares, lower heating values and CO2 emission factors of the fuels used. :return:

export_lci (presamples=True, ecoinvent_compatibility=True, ecoinvent_version='3.7', db_name='carculator db', forbidden_activities=None, create_vehicle_datasets=True)
Export the inventory as a dictionary. Also return a list of arrays that contain pre-sampled random values if stochastic() of CarModel class has been called.

Parameters
- presamples – boolean.
- ecoinvent_compatibility – bool. If True, compatible with ecoinvent. If False, compatible with REMIND-ecoinvent.
- ecoinvent_version – str. “3.5”, “3.6” or “uvek”
- create_vehicle_datasets – bool. Whether vehicles datasets (as structured in ecoinvent) should be created too.

Returns inventory, and optionally, list of arrays containing pre-sampled values.

Return type list

export_lci_to_bw (presamples=True, ecoinvent_compatibility=True, ecoinvent_version='3.7', db_name='carculator db', forbidden_activities=None, create_vehicle_datasets=True)
Export the inventory as a brightway2 bw2io.importers.base_lci.LCIImporter object with the inventory in the data attribute.

# get the inventory
i, _ = ic.export_lci_to_bw()

# import it in a Brightway2 project
i.match_database('ecoinvent 3.6 cutoff', fields=('name', 'unit', 'location', 'reference product'))
i.match_database("biosphere3", fields=('name', 'unit', 'categories'))
i.match_database(fields=('name', 'unit', 'location', 'reference product'))
i.match_database(fields=('name', 'unit', 'categories'))

# Create an additional biosphere database for the few flows that do not exist in "biosphere3"
i.create_new_biosphere("additional_biosphere", relink=True)

# Check if all exchanges link
i.statistics()

# Register the database
i.write_database()

Returns LCIImport object that can be directly registered in a brightway2 project.

Return type bw2io.importers.base_lci.LCIImporter
Export the inventory as an Excel file (if the destination software is Brightway2) or a CSV file (if the destination software is Simapro) file. Also return the file path where the file is stored.

Parameters

- **directory** *(str)* – directory where to save the file.
- **ecoinvent_compatibility** – If True, compatible with ecoinvent. If False, compatible with REMIND-ecoinvent.
- **ecoinvent_version** – “3.6”, “3.5” or “uvek”
- **software_compatibility** – “brightway2” or “simapro”

Returns file path where the file is stored.

Return type *str*

**find_inputs** *(value_in, value_out, find_input_by=’name’, zero_out_input=False)*

Finds the exchange inputs to a specified functional unit :param find_input_by: can be ‘name’ or ‘unit’ :param value_in: value to look for :param value_out: functional unit output :return: indices of all inputs to FU, indices of inputs of intereste :rtype: tuple

**get_A_matrix**()

Load the A matrix. The A matrix contains exchanges of products (rows) between activities (columns).

Returns A matrix with three dimensions of shape (number of values, number of products, number of activities).

Return type *numpy.ndarray*

**get_B_matrix**()

Load the B matrix. The B matrix contains impact assessment figures for a give impact assessment method, per unit of activity. Its length column-wise equals the length of the A matrix row-wise. Its length row-wise equals the number of impact assessment methods.

Parameters

- **method** – only “recipe” and “ilcd” available at the moment.
- **level** – only “midpoint” available at the moment.

Returns an array with impact values per unit of activity for each method.

Return type *numpy.ndarray*

**get_dict_impact_categories**()

Load a dictionary with available impact assessment methods as keys, and assessment level and categories as values.

.. code-block:: python

```python
{'recipe': {'midpoint': ['freshwater ecotoxicity',
                           'human toxicity', 'marine ecotoxicity', 'terrestrial ecotoxicity', 'metal depletion',
                           'agricultural land occupation', 'climate change', 'fossil depletion', 'freshwater eutrophication',
                           'ionising radiation', 'marine eutrophication', 'natural land transformation',
                           'ozone depletion', 'particulate matter formation', 'photochemical oxidant formation',
                           'terrestrial acidification', 'urban land occupation', 'water depletion',
                           'human noise', 'primary energy, non-renewable', 'primary energy, renewable']}
```

5.8. Inventory calculation
Returns dictionary
Return type dict

get_dict_input()
Load a dictionary with tuple (“name of activity”, “location”, “unit”, “reference product”) as key, row/column indices as values.
Returns dictionary with label:index pairs.
Return type dict

get_index_of_flows(items_to_look_for, search_by='name')
Return list of row/column indices of self.A of labels that contain the string defined in items_to_look_for.

Parameters
• items_to_look_for – string
• search_by – “name” or “compartment” (for elementary flows)

Returns list of row/column indices
Return type list

get_index_vehicle_from_array(items_to_look_for, items_to_look_for_also=None, method='or')
Return list of row/column indices of self.array of labels that contain the string defined in items_to_look_for.

Parameters items_to_look_for – string to search for

Returns list

get_results_table(split, sensitivity=False)
Format an xarray.DataArray array to receive the results.

Parameters split – “components” or “impact categories”. Split by impact categories only applicable when “endpoint” level is applied.

Returns xarray.DataArray

get_rev_dict_input()
Reverse the self.inputs dictionary.

Returns reversed dictionary
Return type dict

get_split_indices()
Return list of indices to split the results into categories.

Returns list of indices
Return type list

get_sulfur_content(location, fuel, year)
Return the sulfur content in the fuel. If a region is passed, the average sulfur content over the countries the region contains is returned.

:param location: str. A country or region ISO code
:param fuel: str. “diesel” or “gasoline”
select_heat_supplier(heat_supplier)

The heat supply is an important aspect of direct air capture. Here, we can change the supplier of heat.

set_actual_range()

Set the actual range considering the blend. Liquid bio-fuels and synthetic fuels typically have a lower calorific value. Hence, the need to recalculate the vehicle range. Modifies parameter range of array in place

set_inputs_in_A_matrix(array)


Parameters
array – array from CarModel class

set_inputs_in_A_matrix_for_export(array)


Parameters
array – array from CarModel class

5.9 Inventory export

class carculator.export.ExportInventory(array, indices, db_name=’carculator export’)

Export the inventory to various formats

load_mapping_36_to_uvek()

Load mapping dictionary between ecoinvent 3.6 and UVEK

load_mapping_36_to_uvek_for_simapro()

Load mapping dictionary between ecoinvent 3.6 and UVEK for Simapro name format

load_tags()

Loads dictionary of tags for further use in BW2

write_lci(presamples, ecoinvent_compatibility, ecoinvent_version, vehicle_specs, forbidden_activities=None)

Return the inventory as a dictionary If if there several values for one exchange, uncertainty information is generated. If presamples is True, returns the inventory as well as a presamples matrix. If presamples is False, returns the inventory with characterized uncertainty information. If ecoinvent_compatibility is True, the inventory is made compatible with ecoinvent. If False, the inventory is compatible with the REMIND-ecoinvent hybrid database output of the premise library.

Returns a dictionary that contains all the exchanges

Return type dict

write_lci_to_bw(presamples, ecoinvent_compatibility, ecoinvent_version, forbidden_activities, vehicle_specs=None)

Return a LCIImporter object with the inventory as data attribute.

Returns LCIImporter object to be imported in a Brightway2 project

Return type bw2io.base_lci.LCIImporter

write_lci_to_excel(ecoindent_compatibility, ecoinvent_version, software_compatibility, vehicle_specs=None, directory=None, filename=None, forbidden_activities=None, export_format=’file’)

Export a file that can be consumed by the software defined in software_compatibility. Alternatively, exports
a string representation of the file (in case the inventory should be downloaded from a browser, for example)

Parameters
• directory (str or pathlib.Path) – str. path to export the file to.
• ecoinvent_compatibility (bool) – bool. If True, the inventory is compatible with ecoinvent. If False, the inventory is compatible with REMIND-ecoinvent.
• ecoinvent_version (str) – str. “3.5”, “3.6” or “uvek”
• software_compatibility (str) – str. “brightway2” or “simapro”
• format – can be “file” or “string”. If “file”, returns a file path where the file is stored. If “string”, returns a string. :type format: str :returns: returns the file path of the exported inventory. :rtype: str.

carculator.export.load_mapping_37_to_35()
Load mapping dictionary between ecoinvent 3.7 and 3.5

carculator.export.load_mapping_37_to_36()
Load mapping dictionary between ecoinvent 3.7 and 3.6

carculator.export.load_references()
Load a dictionary with references of datasets

carculator.export.load_uvek_transport_distances()
Load a dictionary with transport distances for inventory export to UVEK database

5.10 Background systems

class carculator.background_systems.BackgroundSystemModel
Retrieve and build dictionaries that contain important information to model in the background system:
• gross electricity production mixes from nearly all countries in the world, from 2015 to 2050.
• cumulative electricity transformation/transmission/distribution losses from high voltage to medium and low voltage.
• share of biomass-derived fuel in the total consumption of liquid fuel in the transport sector. Source: REMIND.

get_biofuel_share()
Retrieve shares of biofuel consumption from REMIND and shape them into an xarray.

Parameters country (str. 2-digit ISO country code.) – Country to return the biofuel share for.

Returns An xarray with ‘country’ and ‘year’ as dimensions

Return type xarray.core.dataarray.DataArray

get_electricity_losses()
Retrieve cumulative electricity losses from high to medium and low voltage. Source: ecoinvent v.3.6.

Returns dictionary

Return type dict

get_electricity_mix()
Retrieve electricity mixes and shape them into an xarray. Source:
• for European countries (EU Reference Scenario 2016),
• for African countries (TEMBA model)
• and for other countries (IEA World Energy outlook 2017)

**Returns** An axarray with ‘country’ and ‘year’ as dimensions

**Return type** xarray.core.dataarray.DataArray

---

**get_region_mapping()**

Retrieve mapping between ISO country codes and REMIND regions.

**Returns** dictionary

**Return type** dict

---

**get_sulfur_content_in_fuel()**

Retrieve sulfur content per kg of petrol and diesel. For CH, DE, FR, AU and SE, the concentration values come from HBEFA 4.1, from 1909 to 2020 (extrapolated to 2050).


There is an assumption made: countries that have high-sulfur content fuels (above 50 ppm in 2019) are assumed to improve over time to reach 50 ppm by 2050.

**Parameters**

- **country** (str. 2-digit ISO country code.) – Country to return the sulfur concentration for.

**Returns** An axarray with ‘country’ and ‘year’ as dimensions

**Return type** xarray.core.dataarray.DataArray
C

carculator.array, 54
    carculator.background_systems, 68
    carculator.driving_cycles, 55
    carculator.energy_consumption, 55
    carculator.export, 67
    carculator.hot_emissions, 60
    carculator.inventory, 61
    carculator.model, 57
    carculator.noise_emissions, 59
Index

A
add_car_parameters() (carculator.car_input_parameters.CarInputParameters method), 53
adjust_cost() (carculator.model.CarModel method), 57
adjust_fuel_mass() (carculator.model.CarModel method), 57
aux_energy_per_km() (carculator.energy_consumption.EnergyConsumptionModel method), 56

B
BackgroundSystemModel (class in carculator.background_systems), 68

C
calculate_cost_impacts() (carculator.model.CarModel method), 57
calculate_ttw_energy() (carculator.model.CarModel method), 58
carculator.array (module), 54
carculator.background_systems (module), 68
carculator.driving_cycles (module), 55
carculator.energy_consumption (module), 55
carculator.export (module), 67
carculator.hot_emissions (module), 60
carculator.inventory (module), 61
carculator.model (module), 57
carculator.noise_emissions (module), 59
CarInputParameters (class in carculator.car_input_parameters), 53
CarModel (class in carculator.model), 57
create_electricity_market_for_battery_production() (carculator.inventory.InventoryCalculation method), 63
create_fuel_markets() (carculator.inventory.InventoryCalculation method), 63
create_PHEV() (carculator.model.CarModel method), 58

drop_hybrid() (carculator.model.CarModel method), 58

D
define_electricity_mix_for_fuel_prep() (carculator.inventory.InventoryCalculation method), 63
define_fuel_blends() (carculator.inventory.InventoryCalculation method), 64

E
export_lci() (carculator.inventory.InventoryCalculation method), 64
export_lci_to_bw() (carculator.inventory.InventoryCalculation method), 64
export_lci_to_excel() (carculator.inventory.InventoryCalculation method), 64
ExportInventory (class in carculator.export), 67

F
fill_xarray_from_input_parameters() (in module carculator.array), 54
find_inputs() (carculator.inventory.InventoryCalculation method), 65

G
get_A_matrix() (carculator.inventory.InventoryCalculation method),
get_B_matrix() (carculator.inventory.InventoryCalculation method), 69
get_biofuel_share() (carculator.background_systems.BackgroundSystemModel method), 69
get_dict_impact_categories() (carculator.inventory.InventoryCalculation method), 66
get_dict_input() (carculator.inventory.InventoryCalculation method), 66
get_electricity_losses() (carculator.background_systems.BackgroundSystemModel method), 68
get_electricity_mix() (carculator.background_systems.BackgroundSystemModel method), 68
get_hot_emission_factors() (in module carculator.hot_emissions), 61
get_hot_emissions() (carculator.hot_emissions.HotEmissionsModel method), 60
get_index_of_flows() (carculator.inventory.InventoryCalculation method), 66
get_index_vehicle_from_array() (carculator.inventory.InventoryCalculation method), 66
get_non_hot_emission_factors() (in module carculator.hot_emissions), 61
get_region_mapping() (carculator.background_systems.BackgroundSystemModel method), 69
get_results_table() (carculator.inventory.InventoryCalculation method), 66
get_rev_dict_input() (carculator.inventory.InventoryCalculation method), 66
get_sound_power_per_compartment() (carculator.noise_emissions.NoiseEmissionsModel method), 59
get_split_indices() (carculator.inventory.InventoryCalculation method), 66
get_standard_driving_cycle() (in module carculator.driving_cycles), 55
get_sulfur_content() (carculator.inventory.InventoryCalculation method), 66
get_sulfur_content_in_fuel() (carculator.background_systems.BackgroundSystemModel method), 67
HotEmissionsModel (class in carculator.hot_emissions), 60
InventoryCalculation (class in carculator.inventory), 61
load_mapping_36_to_uvek() (carculator.export.ExportInventory method), 67
load_mapping_36_to_uvek_for_simapro() (carculator.export.ExportInventory method), 67
load_mapping_37_to_35() (in module carculator.export), 68
load_mapping_37_to_36() (in module carculator.export), 68
load_references() (in module carculator.export), 68
load_tags() (carculator.export.ExportInventory method), 67
load_uvek_transport_distances() (in module carculator.export), 68
modify_xarray_from_custom_parameters() (in module carculator.array), 54
motive_energy_per_km() (carculator.energy_consumption.EnergyConsumptionModel method), 56
NoiseEmissionsModel (class in carculator.noise_emissions), 59
propulsion_noise() (carculator.noise_emissions.NoiseEmissionsModel method), 60
rolling_noise() (carculator.noise_emissions.NoiseEmissionsModel method), 60
select_heat_supplier() (carculator.inventory.InventoryCalculation method), 66
set_actual_range() (carculator.inventory.InventoryCalculation method), 67
set_all() (carculator.model.CarModel method), 58
set_auxiliaries() (carculator.model.CarModel method), 58
set_battery_fuel_cell_replacements() (carculator.model.CarModel method), 58
set_car_masses() (carculator.model.CarModel method), 58
set_electric_utility_factor() (carculator.model.CarModel method), 59
set_electricity_consumption() (carculator.model.CarModel method), 59
set_fuel_cell_parameters() (carculator.model.CarModel method), 59
set_hot_emissions() (carculator.model.CarModel method), 59
set_inputs_in_A_matrix() (carculator.inventory.InventoryCalculation method), 67
set_inputs_in_A_matrix_for_export() (carculator.inventory.InventoryCalculation method), 67
set_noise_emissions() (carculator.model.CarModel method), 59
set_power_parameters() (carculator.model.CarModel method), 59

write_lci() (carculator.export.ExportInventory method), 67
write_lci_to_bw() (carculator.export.ExportInventory method), 67
write_lci_to_excel() (carculator.export.ExportInventory method), 67