emipy

Release None

Jul 05, 2023

Contents

CHAPTER 1

Introduction and quick start guide

1.1 Introduction

Emipy is a python package to analyze industrial emission sources within Europe.
The package accesses data from the European Environmental Agency and Eurostat and allows to generate desired data sets. Data sets can be filtered with clearly structured build-in functions. Furthermore, functions are provided that allow a quick visualization of the data.

1.2 Installation

1.2.1 Requirements

Emipy requires:

1. Python (tested for version 3.7)
2. **Additional add-on modules:**
    1. matplotlib
    2. requests
    3. configparser
    4. pandas
    5. geopandas
    6. descartes
    7. openpyxl
    8. ruamel.yaml
    9. xlrd

3. The emipy package itself

1.2.2 Installation & Initialisation

If you are not familiar with Python, we suggest you follow our step by step installation guide:

1. Download and install the Anaconda distribution from the Anaconda page.
2. **Create a new environment. For this:**
    1. Start the console "Anaconda Prompt"
    2. **Create the environment via executing the following line in "Anaconda Prompt" or your terminal:**

```
conda create -n emipy python=3.7 matplotlib requests configparser␣
↪pandas geopandas \
descartes xlrd ruamel.yaml openpyxl
```

    3. Enter 'y' if asked to install all required packages.
    4. **Activate the environment with:**

```
conda activate emipy
```

    The environment is active when your active code line starts with "(emipy)" instead of "(base)".

    5. **install emipy via:**

```
pip install emipy
```

3. **Initialize a new emipy project. For this:**
    1. Open a Python console in the Anaconda Prompt console via entering and execute the following lines to load emipy and execute the function *init_emipy_project()* which will create a folder structure at the given path and download all necessary data.

Note: You have to change the path to the location, where you want the data to be stored! The inialization process may take a few minutes as large amounts of data is downloaded. Please be patient and let it run until finished completely.

Note: If you are using Windows, the path needs a special format. Python reads `"\"` as an escape, like `"\n"` for new line. You can either use `"\\"` or "/" instead of a single `"\"` or, alternatively you can put a "r" before the string to convert to a raw string. Python needs the single mark quotes around the path to recognize it as a String. Keep that in mind, for all further applications of the emipy functions!

```
import emipy as ep
ep.init_emipy_project('<some_path>')
exit()
```

Here, <some_path> is the name of the directory, where you want to put the data.
Be sure to put the **absolute path** and not a relative path here.
If the initialization function completed its task it prints the message 'The Initialization process is completed.'
If you do not receive this message check for typos and repeat executing the function.

Note: In principle you could also install emipy using only pip but it is advised to install the dependencies separately, since some packages (e.g. geopandas) don't install correctly in Windows when using only the version installed from pypi. In this case, you can install geopandas' dependency Fiona from the channel conda-forge.

1.3 Quick start

1. Start the IDE of your preference. If you are new, just execute `>jupyter notebook` in the Anaconda Prompt console. Make sure to have the jupyter package installed in the Anaconda environment that you are using. This should open a window in your browser. Click on "New" and select Python3. (Here is a short example for the Jupyter Notebook usage. You can also look at the documentation)
2. **Import the emipy module:**

```
import emipy as ep
```

3. **Load the data into your current session with:**

```
db = ep.read_db()
mb = ep.read_mb()
```

4. **and display it with:**

```
db.head()
mb.plot()
```

Note: Use one Notebook box for each display line (`db.head()` and `mb.plot()`). Jupyter Notebook displays just the last object of the box. Therefore it just shows the plot of mb but not the table db, if you write both into the same box.

CHAPTER 2

Data parameters

2.1 Pollution Data

The European Environmental Agency (EEA) provides a lot of information that are assigned to the pollution data. There are very intuitiv ones like the name of the country in that the pollutants are emitted or the year of the emission. But there are also very "specialized" ones like the *facility report ID*, or the *NACE-main economic activity code* (an economic classification, performed by Eurostat). Here we provide a short explanation of the most used parameters and how to access them in the f_db() function. For the time span of 2001 - 2017 we use the data, uploaded under the E-PRTR directive. Since 2017 the EEA combines the data from the E-PRTR data base with the data from IED(Industrial Emission Directive) and LCP (Large Combustion Plants). As a consequence, the structure of the data changes partly (for instance, FacilityReportID becomes FacilityInspireID). Therefore we decided to keep the new data as a seperate data table, adapted it to the needs of emipy but let structural changes stay. The possible filter arguments are listed in the second table. For more detailed information, take a look at the EEA webpage or the Eurostat webpage.

Column Name	Input Data Type	List Of Entries	Example
FacilityReportID	Integer or List of Integers	facilityreportid	f_db(db, FacilityReportID=1856)
CountryName	String or List of Strings	countryname	f_db(db, CountryName='Spain')
ReportingYear	Integer or List of Integers	reportingyear	f_db(db, ReportingYear=2015)
ReleaseMediumName	String or List of Strings	releasemediumname	f_db(db, ReleaseMediumName='Air')
PollutantName	String or List of Strings	pollutantname	f_db(db, PollutantName='Carbon dioxide (CO2)')
PollutantGroupName	String or List of Strings	pollutantgroupname	f_db(db, PollutantGroupName='Inorganic substances')
NACEMainEconomicActivityCode	String or List of Strings	nacemaineconomicactivitycode	f_db(db, NACEMainEconomicActivityCode='25.91')
NUTSRegionGeoCode	String or List of Strings	nutsregiongeocode	f_db(db, NUTSRegionGeoCode='AT11')
ParentCompanyName	String or List of Strings	parentcompanyname	f_db(db, ParentCompanyName='Lenzing AG')
FacilityName	String or List of Strings	facilityname	f_db(db, FacilityName='Lenzing AG')
City	String or List of Strings	city	f_db(db, City='Lenzing')
PostalCode	String or List of Strings	postalcode	f_db(db, PostalCode='4860')
CountryCode	String or List of Strings	countrycode	f_db(db, CountryCode='AT')
RBDGeoCode	String or List of Strings	rbdgeocode	f_db(db, RBDGeoCode='DK1')
RBDGeoName	String or List of Strings	rbdgeoname	f_db(db, RBDGeoName='Jutland and Funen')
NUTSRegionGeoName	String or List of Strings	nutsregiongeoname	f_db(db, NUTSRegionGeoName='')
NACEMainEconomicActivityName	String or List of Strings	nacemaineconomicactivityname	f_db(db, NACEMainEconomicActivityName='Manufacture of pulp')
MainIASectorCode	String or List of Strings	mainiasectorcode	f_db(db, MainIASectorCode='EPER_4')
MainIASectorName	String or List of Strings	mainiasectorname	f_db(db, MainIASectorName='Chemical industry')
MainIAActivityCode	String or List of Strings	mainiaactivitycode	f_db(db, MainIAActivityCode='EPER_5.1/5.2')
MainIAActivityName	String or List of Strings	mainiaactivityname	f_db(db, MainIAActivityName='Tanning of hides and skins')
PollutantReleaseID	Integer or List of Integers	pollutantreleaseid	f_db(db, PollutantReleaseID='16962')
ReleaseMediumCode	String or List of Strings	releasemediumcode	f_db(db, ReleaseMediumCode='AIR')
PollutantCode	String or List of Strings	pollutantcode	f_db(db, PollutantCode='ZN AND COMPOUNDS')
PollutantGroupCode	String or List of Strings	pollutantgroupcode	f_db(db, PollutantGroupCode='INORG')

Column Name	Input Data Type	List Of Entries	Example
FacilityReportID	String or List of Strings	facilityreportid_NewData	f_db(db, FacilityReportID='AT.CAED/900839031695 5.FACILITY')
ReportingYear	Integer or List of Integers	reportingyear_NewData	f_db(db, ReportingYear=2015)
PollutantName	String or List of Strings	pollutantname_NewData	f_db(db, PollutantName='Carbon dioxide (CO2)')
NACEMainEconomicActivityCode	String or List of Strings	nacemaineconomicactivitycode_NewData	f_db(db, NACEMainEconomicActivityCode='25.91')
NUTSRegionGeoCode	String or List of Strings	nutsregiongeocode_NewData	f_db(db, NUTSRegionGeoCode='AT11')
ParentCompanyName	String or List of Strings	parentcompanyname_NewData	f_db(db, ParentCompanyName='Lenzing AG')
FacilityName	String or List of Strings	facilityname_NewData	f_db(db, FacilityName='Lenzing AG')
City	String or List of Strings	city_NewData	f_db(db, City='Lenzing')
PostalCode	String or List of Strings	postalcode_NewData	f_db(db, PostalCode='4860')
CountryCode	String or List of Strings	countrycode_NewData	f_db(db, CountryCode='AT')
NUTSRegionGeoName	String or List of Strings	nutsregiongeoname_NewData	f_db(db, NUTSRegionGeoName='')
NACEMainEconomicActivityName	String or List of Strings	nacemaineconomicactivityname_NewData	f_db(db, NACEMainEconomicActivityName='Manufacture of pulp')
MainIAActivityCode	String or List of Strings	mainiaactivitycode_NewData	f_db(db, MainIAActivityCode='EPER_5.1/5.2')
MainIAActivityName	String or List of Strings	mainiaactivityname_NewData	f_db(db, MainIAActivityName='Tanning of hides and skins')
ReleaseMediumCode	String or List of Strings	releasemediumcode_NewData	f_db(db, ReleaseMediumCode='AIR')
PollutantCode	String or List of Strings	pollutantcode_NewData	f_db(db, PollutantCode='ZN AND COMPOUNDS')

2.2 Map Data

The map data are provided by Eurostat. The maps always show a complete view of Europe, but there are different parameters, that change the layout of the visualisation.

There are two levels where you can choose parameters. These are first the download of the map data and second the load procedure into your session.

During initialisation, emipy downloads, for every NUTS version, the map data with resolution 1:10 million. For storage size reasons, not all map files are downloaded. You can download additional map data with download_MapData(). See *Special Features* for the correct usage.

Statistical Unit	Publication Date	Resolution
NUTS 2021	01/02/2020	1:1 Million
		1:3 Million
		1:10 Million
		1:20 Million
		1:60 Million
NUTS 2016	14/03/2019	1:1 Million
		1:3 Million
		1:10 Million
		1:20 Million
		1:60 Milion
NUTS 2013	03/12/2015	1:1 Million
		1:3 Million
		1:10 Million
		1:20 Milion
		1:60 Milion
NUTS 2010	01/12/2012	1:1 Million
		1:3 Million
		1:10 Million
		1:20 Million
		1:60 Million
NUTS 2006	01/12/2008	1:1 Million
		1:3 Million
		1:10 Million
		1:20 Million
		1:60 Million
NUTS 2003	03/12/2005	1:1 Million
		1:3 Million
		1:10 Million
		1:20 Million

The following sub categories are downloaded for every publication year and resolution:

Spatial Type	NUTS_LVL	Projection
BN	None	3035
		3857
		4326
	Level 0	3035
		3857
		4326
	Level 1	3035
		3857
		4326
	Level 2	3035
		3857
		4326
	Level 3	3035
		3857
		4326

Continued on next page

Table 1 – continued from previous page

Spatial Type	NUTS_LVL	Projection
LB	None	3035
		3857
		4326
	Level 0	3035
		3857
		4326
	Level 1	3035
		3857
		4326
	Level 2	3035
		3857
		4326
	Level 3	3035
		3857
		4326
RG	None	3035
		3857
		4326
	Level 0	3035
		3857
		4326
	Level 1	3035
		3857
		4326
	Level 2	3035
		3857
		4326
	Level 3	3035
		3857
		4326

When loading the map data into your session, you can choose from the parameters *resolution*, *SpatialType*, *NUTS_LVL*, *m_year* and *projection*. *Resolution* and *m_year* do correspond to the above given resolutions and NUTS versions.

Spatialtype has three different options: RG (region), BD (boundary) and LB. For the emipy visualisation functions, the information, stored in the RG file are necessary. Therefore it is chosen by default. Mainly for layout configuration, you can choose BD to only show the borders.

Take into acount, that for the higher NUTS levels, the file just stores new occuring borders. So you would have to plot level 0, 1, 2 and then 3 on top of each other (or level None) to get a map with the complete level 3 borders. LB displays points for the regions.

NUTS_LVL is the Level of the NUTS-classification. You can choose from no level at all up to level 0, 1, 2 and 3. If you put the level on *None*, the loaded shp file contains all objects from the other levels.

Projection refers to the spatial projection of the displayed map. You can choose from EPSG: 4326, 3035, 3857. When the data is loaded into the session you can also transfer the corresponding reference system (crs) with geopandas or emipy.

The default setting is:

```
read_mb(path=None, resolution='10M', SpatialType='RG', NUTS_LVL=0, m_year=2016,␣
↪projection=4326)
```

CHAPTER 3

Tutorials

Here are a few tutorials for the start that explain the basic functions of emipy.

3.1 Generating data sets

At first import the package emipy and read the data base.
The programm stored the path to the project initialisation and automatically searches for the data there and loads it. You can aswell read explicit databases. For this, give the function *read_db()* the path in form of a String as an argument.

```
import emipy as ep

db = ep.read_db()
db.head()
```

A list of possible column names to filter for is displayed with:

```
db.columns
```

```
Index(['FacilityReportID', 'PollutantReleaseAndTransferReportID', 'FacilityID',
       'NationalID', 'ParentCompanyName', 'FacilityName', 'StreetName',
       'BuildingNumber', 'City', 'PostalCode', 'CountryCode', 'CountryName',
       'Lat', 'Long', 'RBDGeoCode', 'RBDGeoName', 'NUTSRegionGeoCode',
       'NUTSRegionGeoName', 'RBDSourceCode', 'RBDSourceName',
       'NUTSRegionSourceCode', 'NUTSRegionSourceName',
       'NACEMainEconomicActivityCode', 'NACEMainEconomicActivityName',
       'CompetentAuthorityName', 'CompetentAuthorityAddressStreetName',
       'CompetentAuthorityAddressBuildingNumber',
       'CompetentAuthorityAddressCity', 'CompetentAuthorityAddressPostalCode',
       'CompetentAuthorityAddressCountryCode',
       'CompetentAuthorityAddressCountryName',
       'CompetentAuthorityTelephoneCommunication',
       'CompetentAuthorityFaxCommunication',
       'CompetentAuthorityEmailCommunication',
       'CompetentAuthorityContactPersonName', 'ProductionVolumeProductName',
       'ProductionVolumeQuantity', 'ProductionVolumeUnitCode',
       'ProductionVolumeUnitName', 'TotalIPPCInstallationQuantity',
       'OperatingHours', 'TotalEmployeeQuantity', 'WebsiteCommunication',
       'PublicInformation', 'ConfidentialIndicator',
       'ConfidentialityReasonCode', 'ConfidentialityReasonName',
       'ProtectVoluntaryData', 'MainIASectorCode', 'MainIASectorName',
       'MainIAActivityCode', 'MainIAActivityName', 'MainIASubActivityCode',
       'MainIASubActivityName', 'ReportingYear', 'CoordinateSystemCode',
       'CoordinateSystemName', 'CdrReleased', 'Published',
       'PollutantReleaseID', 'ReleaseMediumCode', 'ReleaseMediumName',
       'PollutantCode', 'PollutantName', 'PollutantGroupCode',
       'PollutantGroupName', 'PollutantCAS', 'MethodBasisCode',
       'MethodBasisName', 'TotalQuantity', 'AccidentalQuantity', 'UnitCode',
       'UnitName'],
      dtype='object')
```

If you are interested in e.g. the countries that occur in your database you can receive a list with the *get_Countrylist()* function. There are more *get_xy()* functions to access the information in your data base. For more information take a look at the *processdata module description*.

```
ep.get_CountryList(db)
```

```
['Austria',
 'Belgium',
 'Cyprus',
 'Czech Republic',
 'Germany',
 'Denmark',
 'Estonia',
 'Spain',
 'Finland',
 'France',
 'Greece',
```

(continues on next page)

(continued from previous page)

```
 'Hungary',
 'Ireland',
 'Italy',
 'Lithuania',
 'Luxembourg',
 'Latvia',
 'Malta',
 'Netherlands',
 'Norway',
 'Poland',
 'Portugal',
 'Sweden',
 'Slovenia',
 'Slovakia',
 'United Kingdom',
 'Iceland',
 'Serbia',
 'Romania',
 'Bulgaria',
 'Switzerland',
 'Croatia']
```

The actual filtering happens with the function *f_db()*. You have to specifiy the database that you want to filter and the column names and column values that you want to filter for.

Note:

The following lines only create the DataFrame and do not display it. To display the data table, execute e.g. *data1.head()*.
For a better overview, you can use *data = ep.row_reduction(db)*. The new DataFrame is reduced to a list of columns. This list can be adjusted.

Let's filter for pollution in Germany:

```
data1 = ep.f_db(db, CountryName='Germany')
```

If you want to filter for multiple values in one column you have to insert a list.

```
data2 = ep.f_db(db, CountryName=['Germany', 'Switzerland', 'Austria'])
```

You can filter for multiple columns at the same time:

```
CountryName = ['Germany', 'Austria', 'Switzerland']
ReportingYear = [2014, 2015, 2016,2017]
PollutantName = ['Carbon dioxide (CO2)', 'Methane (CH4)']

data3 = ep.f_db(db, CountryName=CountryName, ReportingYear=ReportingYear,␣
↪PollutantName=PollutantName)
```

Note: Take into account that numbers are not from type string and therefore do not need quote markers around them.

For the precise values use the *get_xy()* function or alternativley, you can take a look at the *parameter table*.
You can also filter step by step. For this you would have to insert the filtered database into the filter function.

You can adjust two more arguments in *f_db()*.
If you want to take a look at the continent Europe, you have to exclude Exclaves that belong to European countries, like French Guiana.

```
data4 = ep.f_db(db, ExclaveExclude=True)
```

If you put *ReturnUnknown* on True the function returns a data table, which contains all entries that would be sorted out in the filter process but just do not possess enough information to pass the filter. If this table is empty, then it is a good sign.

```
data5 = ep.f_db(db, CountryName='Germany', ReturnUnknown=True)
```

Now you can generate your own data set of interest with a few lines of code. Since db is a DataFrame object, you can use all pandas functions as well, to personalize your data generation.

3.2 Visualize data sets

Let's start with generating a filtered data set:

```
import emipy as ep

db = ep.read_db()

CountryName = ['Germany', 'Austria', 'Switzerland']
ReportingYear = [2014, 2015, 2016, 2017]
PollutantName = ['Carbon dioxide (CO2)']

data1 = ep.f_db(db, CountryName=CountryName, ReportingYear=ReportingYear,
                PollutantName=PollutantName)
```

Now we can plot the CO2 volume against the reporting years:

```
ep.plot_PollutantVolume(data1, FirstOrder='ReportingYear', rot=0,
                        ylabel='Emission [kg]')
```

As you can see, the first order is equivalent to the x-axis of the plot and the first parameter that the data is sorted by. We can now take a deeper look into our data and sort it additionally by another order:

```
ep.plot_PollutantVolume(data1, FirstOrder='ReportingYear',
                        SecondOrder='CountryName', rot=0,
                        ylabel='Emission [kg]')
```

Keep in mind that the plot functions do not filter the data. If you would like to plot e.g. just the output from Austria you would have to create a new data set, and specifiy this as input in a new plot:

```
data2 = ep.f_db(data1, CountryName='Austria')
ep.plot_PollutantVolume(data2, FirstOrder='ReportingYear',
                        rot=0, ylabel='Emission [kg]')
```

Additionaly to the pollutant emmisions, you can analyse the change of the emmission over time. As this calculation needs information of the year before, the function can only provide this result for all but the first year in the data table.

```
ep.plot_PollutantVolumeChange(data1, FirstOrder='ReportingYear',
                              SecondOrder='CountryName', rot=0,
                              ylabel='Change of emission [kg]')
```

As a third option, you can plot normalised values. With the parameter norm, you can specify the First Order value, that the data is normed to. For a good example we create a new data table:

```
CountryName = ['Germany', 'Austria', 'Switzerland']
ReportingYear = [2014, 2015, 2016, 2017]
PollutantName=['Zinc and compounds (as Zn)', 'Nickel and compounds (as Ni)']

data3 = ep.f_db(db,CountryName=CountryName, ReportingYear=ReportingYear,␣
↪PollutantName=PollutantName)
```

(continues on next page)

(continued from previous page)

```
ep.plot_PollutantVolumeRel(data3, FirstOrder='PollutantName',
                           SecondOrder='ReportingYear', rot=0,
                           ylabel='Normalized emission')
```

If you want to customize the layout of the graphs, you can enter the known arguments of the PyPlot package into the functions. Since the functions return a matplotlib.axes object, you can access and modify the individual elements of the plots like in PyPlot.
The code below returns you the basic plots. For the layout changes, take a look into the Tutorial 2 notebook.

```
import matplotlib.pyplot as plt

fig1, fig1_axes = plt.subplots(2, 2)
fig1_axes[0,0] = ep.plot_PollutantVolume(data1, FirstOrder='ReportingYear',
                                         ax=fig1_axes[0,0], rot=0, ylabel='Emission␣
↪[kg]', color='r')
fig1_axes[1,0] = ep.plot_PollutantVolumeRel(data1, FirstOrder='ReportingYear',
                                            ax=fig1_axes[1,0], rot=0, ylabel=
↪'Normalized Emission')
fig1_axes[0,1] = ep.plot_PollutantVolumeChange(data1, FirstOrder='ReportingYear',
                                               ax=fig1_axes[0,1], rot=0, ylabel=
↪'Change of Emission [kg]')
fig1_axes[1,1] = ep.plot_PollutantVolume(data1, FirstOrder='ReportingYear', ax=fig1_
↪axes[1,1],
                                         SecondOrder='CountryName', rot=0, ylabel=
↪'Emission [kg]')

plt.tight_layout()
plt.show()
```

3.3 Using map data

The first thing that you will realise is, that there is not just one data set for the map like in the pollution data. There are different parameters that change the layout of the maps, therefore when reading the map data you can choose from these parameters. Nevertheless, there is a presetting, that gives you a map by the hand.

```
import emipy as ep
mb = ep.read_mb()

mb.plot()
```

Of special interest is the parameter *NUTS_LVL*, which is the level of the NUTS-ID's which are the codes for categorized regions. See the Eurostat page for more information.
We start with the following set up:

```
NUTS_LVL = '1'
resolution = '10M'
projection = '4326'
SpatialType = 'RG'
m_year = '2013'

mb = ep.read_mb(resolution=resolution, SpatialType=SpatialType, NUTS_LVL=NUTS_LVL, m_
↪year=m_year, projection=projection)
mb.plot().set(xlabel='Longitude', ylabel='Latitude')
```

The filtering happens with the function f_mb(). Depending on the NUTS level, you can filter for countries or the corresponding NUTS-ID. Additionally, there is the argument *ExclaveExclude* which you can put on True to exclude the exclaves and map continental europe.

```
mapdata1 = ep.f_mb(mb, ExclaveExclude=True)
mapdata1.plot().set(xlabel='Longitude', ylabel='Latitude')
```

Lets generate a map of Germany

```
mapdata2 = ep.f_mb(mb, CNTR_CODE='DE')
mapdata2.plot().set(xlabel='Longitude', ylabel='Latitude')
```

To map e.g. North Rhine-Westphalia you have to know, that the NUTS-ID is 'DEA' and can use it as a filter. You can look up the NUTS_ID' at the link above or take a look in the DataFrame mb.

```
mapdata3 = ep.f_mb(mb, NUTS_ID=['DEA'], CNTR_CODE='DE')
mapdata3.plot(aspect='equal').set(xlabel='Longitude', ylabel='Latitude')
```

To combine map data and pollution data you have two options. You can plot the pollution sources on the map or create a colormap of the pollution in the regions.
Let's start with mapping the CO2 sources in Germany and Austria in the year 2017.

```
import matplotlib.pyplot as plt

db = ep.read_db()

CountryName = ['Germany', 'Austria']
```

(continues on next page)

(continued from previous page)

```
ReportingYear = [2017]
PollutantName = ['Carbon dioxide (CO2)']

data4 = ep.f_db(db,CountryName=CountryName, ReportingYear=ReportingYear,␣
↪PollutantName=PollutantName)
mapdata4 = ep.f_mb(mb, CNTR_CODE=['DE','AT'])

fig1 = plt.figure()
ax1 = fig1.add_subplot(1, 1, 1)
#ax1 = mapdata1.plot(ax=ax1, color='lightgrey')
ax1 = ep.map_PollutantSource(data4, mapdata4, MarkerSize=200,
                             ax=ax1).set(xlabel='Longitude', ylabel='Latitude')
fig1.set_figheight(10)
fig1.set_figwidth(10)
```

If you uncomment everything, you'll get a complete map of Europe in light grey without emission sources, while Germany and Austria are highlighted and show their sources.
For the *map_PollutantSource()* you have to insert the data and map set. You can choose the MarkerSize, which is the size of the maximal output. The other sources are normalized to this value. If MarkerSize is put on zero or is not given at all, all marker have the same size.

map_PollutantSource() can return three different objects. The return is specified by the argument *ReturnMarker* which is [0] by default. If not choosen differently the function returns the axes-object, or the plot. ReturnMarker=1 returns the DataFrame with all data that are plotted. ReturnMarker=2 returns the DataFrame with all data that is not plotted. This might happen, when the coordinates of the data is bad and not inside the regions or not given at all.
You can also plot different pollutants and color them differently with the parameter *category*.

```
CountryName = ['Germany', 'Austria']
ReportingYear = [2017]
PollutantName = ['Carbon dioxide (CO2)', 'Nitrogen oxides (NOx/NO2)']

data5 = ep.f_db(db,CountryName=CountryName, ReportingYear=ReportingYear,␣
↪PollutantName=PollutantName)
mapdata5 = ep.f_mb(mb, CNTR_CODE=['DE','AT'])

fig2 = plt.figure()
ax1 = fig2.add_subplot(1, 1, 1)
ax1 = ep.map_PollutantSource(data5, mapdata5, MarkerSize=200, category='PollutantName
↪',
                             ax=ax1).set(xlabel='Longitude', ylabel='Latitude')
fig2.set_figheight(10)
fig2.set_figwidth(10)
```

To plot the emission of specific regions you can use the *map_PollutantRegions()* function. In the following example we plot the emission of CO2 in Austria on NUTS-level 2.

```
NUTS_LVL = '2'
Resolution = '10M'
projection = '4326'
SpatialType = 'RG'
m_year = '2013'

mb = ep.read_mb(resolution=resolution, SpatialType=SpatialType,
                NUTS_LVL=NUTS_LVL, m_year=m_year, projection=projection)


CountryName = ['Austria']
ReportingYear = [2017]
PollutantName = ['Carbon dioxide (CO2)']

data6 = ep.f_db(db,CountryName=CountryName, ReportingYear=ReportingYear,␣
↪PollutantName=PollutantName)
mapdata6 = ep.f_mb(mb, CNTR_CODE='AT')

fig3 = plt.figure()
ax1 = fig3.add_subplot(1, 1, 1)
ax1 = ep.map_PollutantRegions(data6, mapdata6, ax=ax1, legend=True)
plt.title("CO2 emission in Austria in the year 2017 in [kg]", fontsize=20)
fig3.set_figheight(10)
fig3.set_figwidth(20)
plt.xlabel('Longitude', fontsize=16)
plt.ylabel('Latitude', fontsize=16)
```

Since the returns of the functions are Axes-objects, you can use PyPlot functions and arguments to change the layout. You can also use Geopandas to personalize the plot generation because the map data is stored as a GeoDataFrame.

3.4 Export Data and Figures

You can export data tables and figures directly to the ExportData-folder of your project. Aditionally you can depleat unnecessary information and configurate your data table before export.

Let's start with creating the data of the Benelux states for the pollution of C02 and CH4 (Methane) in the time period 2010-2013.

```
import emipy as ep

db = ep.read_db()

Countrylist = ['Luxembourg', 'Belgium', 'Netherlands']
ReportingYear = [2010, 2011, 2012, 2013]
PollutantName = ['Carbon dioxide (CO2)', 'Methane (CH4)']

dataset1 = ep.f_db(db, CountryName=Countrylist, ReportingYear=ReportingYear,␣
↪PollutantName=PollutantName)
```

We can now export this data table with:

```
ep.export_db_to_csv(dataset1, filename='Benelux.csv')
```

emipy searches for the ExportData folder in the path given during the initiation process and stores the file with the described filename over there.
If you want to export the file to a different path, you can use the argument path to name the corresponding path.

```
ep.export_db_to_csv(dataset1, path=r'C:\User\User1\testpath', filename='Benelux2.csv')
ep.export_db_to_csv(dataset1, path=r'C:\User\User1\testpath\Benelux3.csv')
```

You can aswell export to other file types. The emipy export functions are based on the pandas export functions and imply their features:

```
ep.export_db_to_pickle(dataset1, filename='Benelux.pkl', compression='zip')
ep.export_db_to_excel(dataset1, filename='Benelux.xlsx')
```

Note:

Pandas needs an additional Package for the export to a xlsx file. In consequence we do too. Execute >*pip install openpyxl* in the Anaconda Prompt console.

Let's create a figure and use map data to visualize our data:

```
NUTS_LVL = '2'
resolution = '10M'
projection = '4326'
SpatialType = 'RG'
m_year = '2013'

mb = ep.read_mb(resolution=resolution, SpatialType=SpatialType, NUTS_LVL=NUTS_LVL, m_
↪year=m_year, projection=projection)

mapdata1 = ep.f_mb(mb, CNTR_CODE=['BE', 'LU', 'NL'])

import matplotlib.pyplot as plt

fig1, ax = plt.subplots(2, 2, figsize=(8.27, (1.5/3)*11.69))
ep.plot_PollutantVolume(dataset1, ax=ax[0,0], FirstOrder='ReportingYear', SecondOrder=
↪'CountryName', rot=0).set(xlabel='Reporting Year', ylabel='Emission [kg]')
ep.plot_PollutantVolumeChange(dataset1, ax=ax[0,1], FirstOrder='ReportingYear',␣
↪SecondOrder='CountryName', rot=0).set(xlabel='Reporting Year', ylabel='Change of␣
↪Emission [kg]')
ep.map_PollutantSource(dataset1,mapdata1, ax=ax[1,0], MarkerSize=100).set(xlabel=
↪'Longitude', ylabel='Latitude')
```

(continues on next page)

(continued from previous page)

```
ep.map_PollutantRegions(dataset1, mapdata1, ax=ax[1,1], legend=True).set(xlabel=
↪'Longitude', ylabel='Latitude')

fig1.set_figheight(10)
fig1.set_figwidth(20)
```

The export of the figures is based on matplotlib.pyplot.savefig and has the same features for the export, but automatically saves the figure to the ExportFolder, if not stated otherwise.

```
ep.export_fig(fig1, filename='Benelux.png')
ep.export_fig(fig1, filename='Benelux.pdf', facecolor='w', edgecolor='w')
ep.export_fig(fig1, filename='Benelux.svg', facecolor='w', edgecolor='w')
```

Emipy provides functions for the export to calliope. Calliope is a multi-scale energy systems modelling framework.

3.5 Special Features

There are several functions that enhance the usage of emipy. In this tutorial we take a look at these functions and work threw their use cases.

3.5.1 Change root path

At first let's take a look at the configuration options of emipy. When initialise an emipy project, we defined a path to the root of the project. This path is stored in a config file and is used when the data is loaded in a session or auto exported to the export folder.

When you use emipy in another environment than the one in which you initialised emipy, your root path is not set to the projects folder. If you want to continue your work in this project, or simply don't want to download all data again, you can adapt this path.

```
import emipy as ep

ep.change_RootPath(r'Your\individual\path\to\your\project')
```

3.5.2 Downloading additional map data

During the initialization, emipy downloads map data from Eurostat. There is not just one map, but a lot of different ways to visualize the countries.

Emipy downloads the predefined set with resolution factor 1:10 Million, but you can download additional map data if wanted. For the download you can choose from the resolution (1:1,3,10,20,60 Million) and emipy downloads the map data for all publication years, projections and NUTS-LVL into the projects MappingData folder. With *read_mb()* you can make further choices of the way, your map is diplayed. For more details see *Data parameters*.

The *resolution* can be an Integer or a list of Integers which values has to be the resolution factor. You can put the parameter *clear* to True, to clear the MappingData folder before downloading. This prevents doublication of data and reduces the memory size.

```
example_res =[3,60]
ep.download_MapData(r'The\path\to\the\root\of\your\project', resolution=example_res,␣
↪clear=True)
```

3.5.3 Change Units

When working with the pollution data, you might want to change the unit of the emission. The function *change_unit()* reads the current unit for all entries and recalculates the emission value to the new unit and stores the new unit code in the DataFrame.

```
db = ep.read_db()
data1 = ep.f_db(db,CountryName='Germany')
data2 = ep.change_unit(data1, unit='megaton')
```

3.5.4 Data Table Adaption

For the export you might not need all of the 73 columns and want to increase your overview. You can exclude columns and reduce the columns to those which you are interested in.

The function *row_reduction()* uses information from the config file to determine which columns are defined as columns of interest. You can change these configuration with *change_ColumnsOfInterest()*. You can use the paramter *total* to change the complete list, *add* to add column names or *sub* to substract from present column names. If you put the parameter *reset* on True, you reset the list of column names to the initial state.

```
data3 = ep.row_reduction(data1)
total=['CountryCode', 'CountryName']
ep.change_ColumnsOfInterest(total=total, add=None, sub=None, reset=False)
data4 = ep.row_reduction(data3)
ep.change_ColumnsOfInterest(total=None, add='Lat', sub=None, reset=False)
data5 = ep.row_reduction(data3)
ep.change_ColumnsOfInterest(total=None, add=None, sub='CountryCode', reset=False)
data6 = ep.row_reduction(data3)
ep.change_ColumnsOfInterest(total=None, add=None, sub=None, reset=True)
data7 = ep.row_reduction(data3)
```

There is also an option to rename the columns at large scale. The working principle is the same as the column reduction but you have to insert dicts instead of strings or lists.

```
addition={'Lat': 'Latitude'}
ep.change_RenameDict(total=None, add=addition, sub=None, reset=False)
data8 = ep.rename_columns(data3)
```

(continues on next page)

(continued from previous page)

```
ep.change_RenameDict(total=None, add=None, sub=None, reset=True)
data9 = ep.rename_columns(data3)
```

3.5.5 Emission information

So far you have produced filtered data bases and plots of these data bases. But perhaps you want to get the information of your plot in form of a data table.

```
data10 = ep.get_PollutantVolume(data2, FirstOrder='ReportingYear')
data11 = ep.get_PollutantVolumeRel(data2, FirstOrder='ReportingYear')
data12 = ep.get_PollutantVolumeChange(data2, FirstOrder='ReportingYear')
```

In comparison to your data base, this table has summed up all emissions for your order parameter. The usage of the order parameter is the same as in the plot functions.

3.5.6 NACE-Code selection

The economical classification of the entries with the NACE-Code is not consistent over time. The European Union performed a revision of the NACE-Classification NACE 1.1 to NACE 2. In consequence, the entries for the years 2001 and 2004 are encoded in the old classification, while the newer entries are encoded by NACE 2.
Emipy provides a function that performes an transition of the old codes to the new, based on the transition tables, provided by Eurostat.

```
db2 = ep.read_db()
db = ep.perform_NACETransition(db2)
```

The transition does not allow an unique assignment of new codes, which is why the new codes may be stored as list of multiple codes. In a consequence, entries might pass your filter, but are not truly part of your requested data. You might want to check for these entries, if they really are part of your economic field.
There exist entries in the old data, that have 2 different NACE 1.1 Code, which have no assignment in the transition tables. For these cases we decided for a assignment.
The NACECode 27.35 is translated to 24.10 since the NACEMainEconomicActivityName of both sounds very similar. 74.84 is translated to 59.20, 63.99, 74.10, 74.90, 77.40, 82.30, 82.91, 82.99. Here the NACEMainEconomicActivityName is the same as for 74.87 and we copied the transition from this NACECode.
You can finde the NACE-Codes in the NACE Rev.2 starting at page 63. Choosing the right code enables you to filter for your request. *NACEMainEconomicActivityCode* needs a string with the complete NACE Code like '01.46' or a list of these Codes.

```
data13 = ep.f_db(db, NACEMainEconomicActivityCode='35.11')
```

Some groups of NACE codes are stored in the config file. You can access them with *get_NACECode_filter()*. If the parameter *specify* is None, which it is by default, you receive a list of dictionaries which have the NACE Codes as list

corresponding to the key name. You can put *specify* to on of the keys to receive the value, the list of NACE Codes.

```
print(ep.get_NACECode_filter())
NACECode = ep.get_NACECode_filter(specify = 'animal production')
data14 = ep.f_db(db2, NACEMainEconomicActivityCode=NACECode)
```

You can create your own NACE-Code lists with *change_NACECode_filter()*. This works very much like *change_RenameDict()*. You can add and subtract single key/value pairs, or replace the complete dictionary. For the right syntax, make sure your codes are 5 characters long and seperated by a comma.

```
ep.change_NACECode_filter(add={'metalmanufaction':'24.51,24.52,24.53,24.54'})
ep.change_NACECode_filter(sub={'metalmanufaction':'24.51,24.52,24.53,24.54'})
```

3.5.7 Impurity analysis

The emission of specific pollutants comes with emission of other pollutants. In consequence you do not have pure emissions, but impurities to your target pollutant. To analyse these, emipy provides the functions *get_ImpurityVolume()* and *plot_ImpurityVolume()*.

You can specify your analysis with a few parameters. At first you specify your data that is to analyse with calling db. Then you name the target pollutant, which is the impured one. For the plot function you also have to specify the impurity molecule that you are looking for.

Additional paramters for the **get_** function are *FirstOrder*, *ReleaseMediumName*, *absolute*, *FacilityFocus*, *impurity* and *statistics*. With *FirstOrder*, you specify the column at which your data is sorted. The default is FacilityReportID, since it is very intuitive to look for the impurities of specific facilities. Nontheless, you could also choose for example NUTSRegionGeoCode, to make a region analysis, rather than a facility analysis.

```
data = ep.f_db(db, CountryName='Germany')
data16 = ep.get_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', FirstOrder=
↪'NUTSRegionGeoCode')
ep.plot_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', impurity='Carbon␣
↪monoxide (CO)', FirstOrder='NUTSRegionGeoCode')
```

You can change the *ReleaseMediumName* to Water or Soil, if your target pollutant is emitted in another medium than air. The function returns the emission value of your impurity in relation to the emission of your target. If you want to get the absolute value, you can change the *absolute* parameter to True.

```
data17 = ep.get_ImpurityVolume(db=data, target='Carbon dioxide (CO2)',␣
↪ReleaseMediumName='Air', absolute=True)
```

If your *FirstOrder* is something else than FacilityReportID, the *FacilityFocus* parameter becomes interesting. If this parameter is True, only impurities emitted in facilities that aswell emit your target pollutant are considered. You can put the parameter to False, to turn this feature off and analyse the impurities of all facilities in your Order parameter.

```
data18 = ep.get_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', FirstOrder=
↪'NUTSRegionGeoCode', FacilityFocus=False)
ep.plot_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', impurity='Carbon␣
↪monoxide (CO)', FirstOrder='NUTSRegionGeoCode', FacilityFocus=False)
```

If you have not specified the impurity, you will get a table with all present impurities. You can specify your impurity, to receive only your impurity of interest.

```
data19 = ep.get_ImpurityVolume(db=data, target='Carbon dioxide (CO2)')
data20 = ep.get_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', impurity=
↪'Carbon monoxide (CO)')
```

You can also set the parameter *statistics* to False or True to either simply get or plot your impurity values or to get or plot the statistics of these. The default is False.

```
ep.get_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', impurity='Carbon␣
↪monoxide (CO)')
ep.plot_ImpurityVolume(db=data, target='Carbon dioxide (CO2)', impurity='Carbon␣
↪monoxide (CO)', statistics=True)
```

When creating a data table with *get_ImpurityVolume()* you will often have NaN as entries. This happens because different facilities have different impurities. So a NaN value means, that there is no impurity of this pollutant type listet in the data base. This does not mean, that there is definitly no impurity. Impurities can be below a certain threshold value and therefore not listet in the E-PRTR.
If *statistics* is True, *plot_ImpurityVolume()* automatically ignores NaN values. When you plot your simple impurity values, you can set the parameter *PlotNA* to False. Then the na values are not plotted.

The following example illustrates emipy's functionality for impurity analysis.

```
db = ep.read_db()

data1 = ep.f_db(db, ReportingYear=2015, CountryName='France', NUTSRegionGeoCode='FR30
↪',
                PollutantName='Carbon dioxide (CO2)')
data1 = ep.change_unit(data1, unit='megaton')
data2 = ep.f_db(db, ReportingYear=2015, CountryName='France', NUTSRegionGeoCode='FR30
↪')

testdata=ep.get_PollutantVolume(data1, FirstOrder='FacilityReportID').sort_values(
↪'TotalQuantity', ascending=False)
testdata2=ep.get_ImpurityVolume(data2, target='Carbon dioxide (CO2)', impurity=
↪'Nitrogen oxides (NOx/NO2)', absolute=True).sort_values('TotalQuantity',␣
↪ascending=False)
testdata2.loc[:,'Nitrogen oxides (NOx/NO2)'] /= 1000000

testdata3=ep.get_ImpurityVolume(data2, target='Carbon dioxide (CO2)', impurity=
↪'Nitrogen oxides (NOx/NO2)').sort_values('TotalQuantity', ascending=False)
```

(continues on next page)

(continued from previous page)

```
testdata3.loc[:,'Nitrogen oxides (NOx/NO2)'] *= 1000
testdata4=ep.get_ImpurityVolume(data2, target='Carbon dioxide (CO2)', impurity=
↪'Nitrogen oxides (NOx/NO2)', statistics=True)
testdata4.loc[:,'Nitrogen oxides (NOx/NO2)'] *= 1000
testdata4 = testdata4.drop('count')

fig3, ax = plt.subplots(2, 2)

testdata.plot(x='FacilityReportID', y='TotalQuantity', ax=ax[0, 0], kind='bar')
testdata2.plot(x='FacilityReportID', y='Nitrogen oxides (NOx/NO2)', ax=ax[0,1], kind=
↪'bar')
testdata3.plot(x='FacilityReportID', y='Nitrogen oxides (NOx/NO2)', ax=ax[1,0], kind=
↪'bar')
testdata4.plot(y='Nitrogen oxides (NOx/NO2)', ax=ax[1,1], kind='bar', rot=0)

plt.tight_layout()
ep.export_fig(fig3, filename='impurity_analysis.pdf')
```

3.5.8 Calliope Export

The following code generates csv and yaml files for the use with the energy modelling framework Calliope.
The csv file contains information about the supply constraint of the carbon dioxide source

```
db = emipy.read_db()
source = f_db(db, CountryName='Germany', ReportingYear=2017, PollutantCode='CO2',␣
↪City='Dueren')
emipy.export_calliope(source)
```

The generated yaml file contains the technology definition. In the default setting the costs for CO_2 provision are set to 70€ (which can be specified using the *sc* variable).

```
3519569:
techs:
    co2_supply:
        essentials:
            name: CO2 Supply
            color: '#0b95ef'
            parent: supply
            carrier: co2
        constraints:
            resource: file=emipy2calliope.csv:3519569
            energy_cap_max: 20776.255707762557
            lifetime: 1
        costs:
            monetary:
                interest_rate: 0
                om_prod: 0.07
coordinates:
```

(continues on next page)

(continued from previous page)

```
    lat: 50.776516546
    lon: 6.48949128038
```

These are the notebooks of the tutorials:

```
download Tutorial 1 Data Generation
download Tutorial 2 Visualize Data
download Tutorial 3 Using Map Data
download Tutorial 4 Export Data
download Tutorial 5 Special Features
```

CHAPTER 4

Modules

4.1 Module rawdata

This module contains all functions, necessary for the inititation of an emipy project.

emipy.rawdata.**change_RootPath**(*path*)
Changes the path of the root to the project in the configuration.ini file.

Parameters **path** (*String*) – Path to the project, which is to access.

Returns

Return type None.

emipy.rawdata.**download_MapData**(*path*, *resolution=10*, *clear=False*, *chunk_size=128*)
Download shp files to given path

Parameters

- **path** (*String*) – Path to the root of the project.
- **resolution** (*int/list, optional*) – Defines the resolution, that the downloaded maps have. Selectable are 1,3,10,20,60. The default is 10.
- **clear** (*Boolean*) – If put on True, the function clears the MappingData Directory, before downloading the data., The default is False.
- **chunk_size** (*TYPE, optional*) – DESCRIPTION. The default is 128.

Returns

Return type None.

emipy.rawdata.**download_NACE_TransitionTables**(*path*)
Creates, if necessary, the folder TransitionData in the given path and downloads NACE transition tables from Eurostat to save them in the folder TransitionData.

Parameters **path** (*String*) – Path to the root of the project.

Returns

Return type None.

emipy.rawdata.**download_PollutionData**(*path*, *chunk_size=128*)
Downloads the pollution data into given path.

Parameters

- **path** (*String*) – Path to the root of the project.
- **chunk_size** (*TYPE, optional*) – DESCRIPTION. The default is 128.

Returns

Return type None.

emipy.rawdata.**generate_PollutionData_2**(*path*)
Generates the new data set. For this, the function downloads the .pkl file from the emipy repository, decompresses it and renames the columns.

Parameters **path** (*String*) – Path to the root of the project.

Returns **data2** – The new data base in the format, that enables the emipy functions to act on it.

Return type DataFrame

emipy.rawdata.**get_RootPath**()
Returns the current root path, stored in the config file.

Returns **path** – Current path to the root of the project, stored in the config file.

Return type String

emipy.rawdata.**init_emipy_project**(*path*, *resolution=10*, *force_rerun=False*)
Executes the initiation of a new project. Downloads all needed data to the given path and merges data of interest.

Parameters

- **path** (*String*) – Path to root of project.
- **force_rerun** (*Boolean, optional*) – Forces the programm to rerun the merging routine, if True. The default is False.
- **resolution** (*int/list, optional*) – Defines the resolution, that the downloaded maps have. Selectable are 1,3,10,20,60. The default is 10.

Returns

Return type None.

emipy.rawdata.**merge_PollutionData**(*path*, *force_rerun=False*)
Inserts tables with different data into each other.

Parameters

- **path** (*String*) – Path to the root of the project.
- **force_rerun** (*Boolean, optional*) – If true, the function executes the routine even if the destination folder already contains corresponding files.. The default is False.

Returns

Return type None.

emipy.rawdata.**pickle_RawData**(*path*, *force_rerun=False*)
loads files of interest, converts them into .pkl file and saves them in the same path.

Parameters

- **path** (*String*) – Path to the root of the project.
- **force_rerun** (*Boolean, optional*) – If true, the function executes the routine even if the destination folder already contains corresponding files.. The default is False.

Returns

Return type None.

4.2 Module processdata

This module contains all functions to produce the data set of interest.

emipy.processdata.**change_ColumnsOfInterest**(*total=None*, *add=None*, *sub=None*, *reset=False*)

Changes the list of column names in the config file, that are of interest.

Parameters

- **total** (*List/String, optional*) – Replaces the column names at all with the given list. If total is a string the names have to be seperated by a ",". The default is None.
- **add** (*List/String, optional*) – Adds the given column names to the existing ones. If add is a string the names have to be seperated by a ",".The default is None.
- **sub** (*List/String, optional*) – Subtracts the given column names from the existing ones. If sub is a string the names have to be seperated by a ",".The default is None.
- **reset** (*Boolean, optional*) – Resets the list of column names to the presettings. The default is False.

Returns **config['COLUMNSOFINTEREST']** – Updated list of columnsofinterest.

Return type dict

emipy.processdata.**change_NACECode_filter**(*total=None*, *add=None*, *sub=None*)

Changes the NACE code dict in the config file.

Parameters

- **total** (*Dict, optional*) – Replacement dictionary that replaces the complete NACE code dict. The default is None.
- **add** (*Dict, optional*) – Dictionary that gets added to the NACE code dict. The default is None.
- **sub** (*Dict, optional*) – Dictionary that is substracted from the NACE code dict. The default is None.

Returns

Return type None

emipy.processdata.**change_RenameDict**(*total=None*, *add=None*, *sub=None*, *reset=False*)

Changes the column name dict in the config file and returns the actual column names dict.

Parameters

- **total** (*Dict, optional*) – Replacement dictionary that replaces the complete column name dict. The default is None.
- **add** (*Dict, optional*) – Dictionary that gets added to the column name dict. The default is None.

- **sub** (*Dict, optional*) – Dictionary that is substracted from the column name dict. The default is None.
- **reset** (*Boolean, optional*) – If True, the column name dict gets resetted to the standard settings. The default is False.

Returns **config['COLUMNNAMES']** – actualised column name dictionary.

Return type dict

emipy.processdata.**change_unit**(*db*, *unit=None*)

Changes the units of the emission in the table and adapts the numbers of TotalQuantity and AccidentalQuantity in the according way. If no unit is given, no changes are applied.

Parameters

- **db** (*DataFrame*) – DataFrame which units are to be changed.
- **unit** (*string, optional*) – New unit name. The default is None.

Returns **data_out** – DataFrame with changed emission units.

Return type DataFrame

emipy.processdata.**export_db_to_csv**(*db*, *path=None*, *filename=None*, ***kwargs*)

Stores the DataFrame given in the input as a .csv file to the given path, or if the path is not given to the ExportData folder in the root path with the given filename.

Parameters

- **db** (*DataFrame*) – Filtered database, that is to export.
- **path** (*String, optional*) – Path under which the DataFrame is stored.
- **filename** (*String, optional*) – If the path is not given, this is the file name under which the DataFrame ist stored in the ExportData folder of the project
- **kwargs** (*Type, optional*) – pandas.to_csv() input arguments

Returns

Return type None

emipy.processdata.**export_db_to_excel**(*db*, *path=None*, *filename=None*, ***kwargs*)

Stores the DataFrame given in the input as a .xlsx file to the given path, or if the path is not given to the ExportData folder in the root path with the given filename.

Parameters

- **db** (*DataFrame*) – Filtered database, that is to export.
- **path** (*String, optional*) – Path under which the DataFrame is stored.
- **filename** (*String, optional*) – If the path is not given, this is the file name under which the DataFrame ist stored in the ExportData folder of the project
- **kwargs** (*Type, optional*) – pandas.to_excel() input arguments

Returns

Return type None

emipy.processdata.**export_db_to_pickle**(*db*, *path=None*, *filename=None*, ***kwargs*)

Stores the DataFrame given in the input as a .pkl file to the given path, or if the path is not given to the ExportData folder in the root path with the given filename.

Parameters

- **db** (*DataFrame*) – Filtered database, that is to export.
- **path** (*String, optional*) – Path under which the DataFrame is stored.
- **filename** (*String, optional*) – If the path is not given, this is the file name under which the DataFrame ist stored in the ExportData folder of the project
- **kwargs** (*Type, optional*) – pandas.to_pickle() input arguments

Returns

Return type None

emipy.processdata.**f_db**(*db*, *FacilityReportID=None*, *CountryName=None*, *ReportingYear=None*, *ReleaseMediumName=None*, *PollutantName=None*, *PollutantGroupName=None*, *NACEMainEconomicActivityCode=None*, *NUTSRegionGeoCode=None*, *ParentCompanyName=None*, *FacilityName=None*, *City=None*, *PostalCode=None*, *CountryCode=None*, *RBDGeoCode=None*, *RBDGeoName=None*, *NUTSRegionGeoName=None*, *NACEMainEconomicActivityName=None*, *MainIASectorCode=None*, *MainIASectorName=None*, *MainIAActivityCode=None*, *MainIAActivityName=None*, *PollutantReleaseID=None*, *ReleaseMediumCode=None*, *PollutantCode=None*, *PollutantGroupCode=None*, *ExclaveExclude=False*, *ReturnUnknown=False*)

Takes DataFrame and filters out data, according to input parameters.

Parameters

- **db** (*DataFrame*) – Input DataFrame.
- **FacilityReportID** (*Int/String/List, optional*) – List of FacilityReportID's to be maintained. In the data from 2001-2017 this entry is an integer. Therefore we have to use integers or a list of integers for the filtering. In the data from 2017-2019 this is stored as a string. Therefore we have to use a string or a list of strings for the filtering. The default is None.
- **CountryName** (*String/List, optional*) – List of countries to be maintained. The default is None.
- **ReportingYear** (*String/List, optional*) – List of reporting years to be maintained. The default is None.
- **ReleaseMediumName** (*String/List, optional*) – List of release medium names to be maintained. The default is None.
- **PollutantName** (*String/List, optional*) – List of pollutant names to be maintained. The default is None.
- **PollutantGroupName** (*String/List, optional*) – List of polllutant group names to be maintained. The default is None.
- **NACEMainEconomicActivityCode** (*String/List, optional*) – List of NACE main economic activity codes to be maintained. The default is None.
- **NUTSRegionGeoCode** (*String/List, optional*) – List of NUTS region geocodes to be maintained. The default is None.
- **ParentCompanyName** (*String/List, optional*) – List of Parent company names to be maintained. The default is None.
- **FacilityName** (*String/List, optional*) – List of facility names to be maintained. The default is None.

- **City** (*String/list, optional*) – List of cities to be maintained. The default is None.
- **PostalCode** (*String/List, optional*) – List of postal codes to be a´maintained. The default is None.
- **CountryCode** (*String/List, optional*) – List of country codes to be maintained. The default is None.
- **RBDGeoCode** (*String/List, optional*) – List of River Basin District geo codes to be maintained. The default is None.
- **RBDGeoName** (*String/List, optional*) – List of River Basin District geo names to be maintained. The default is None.
- **NUTSRegionGeoName** (*String/List, optional*) – List of NUTS region geo names to be maintained. The default is None.
- **NACEMainEconomicActivityName** (*String/List, optional*) – List of NACE main economic activity names to be maintained. The default is None.
- **MainIASectorCode** (*String/List, optional*) – List of Investment Association sector codes to be maintained. The default is None.
- **MainIASectorName** (*String/List, optional*) – List of Investmend Association sector names to be maintained. The default is None.
- **MainIAActivityCode** (*String/List, optional*) – List of Investmend Association activity codes to be maintained. The default is None.
- **MainIAActivityName** (*String/List, optional*) – List of Investmend Association activity names to be maintained. The default is None.
- **PollutantReleaseID** (*Int/List, optional*) – List of pollutant release IDs to be maintained. The default is None.
- **ReleaseMediumCode** (*String/List, optional*) – List of realease medium codes to be maintained. The default is None.
- **PollutantCode** (*String/List, optional*) – List of pollutant codes to be maintained. The default is None.
- **PollutantGroupCode** (*String/List, optional*) – List of pollutant group codes to be maintained. The default is None.
- **ExclaveExclude** (*Boolean, optional*) – If True, exclaves that are unique NUTS-LVL1 regions are excluded. The default is False.
- **ReturnUnknown** (*Boolean, optional*) – If True, function returns DataFrame that is sorted out due to not enough information for the filter process. The default is False.

Returns

- **db** (*DataFrame*) – DataFrame after filter process.
- **dbna** (*DataFrame*) – DataFrame that is filtered out, but has na values for the filter column. If they are filtered out correctly is not known.

emipy.processdata.**f_mb**(*mb*, *NUTS_ID=None*, *CNTR_CODE=None*, *NAME_LATN=None*, *ExclaveExclude=False*)

Filters the geometry data of the DataFrame by the specifications of the input.

Parameters

- **mb** (*DataFrame*) – Input DataFrame.

- **NUTS_ID** (*String/List, optional*) – NUTS:ID assigned from eurostat. The default is None.
- **CNTR_CODE** (*String/List, optional*) – Country code. The default is None.
- **NAME_LATN** (*String/List, optional*) – Name of Region, classified by eurostat. The default is None.

Returns **mb** – DataFrame with geometry data of the specified conditions.

Return type DataFrame

emipy.processdata.**get_CNTR_CODEList**(*mb*)
returns list of all possible CountryCodes in the given DataFrame.

Parameters **mb** (*DataFrame*) – Data of interest.

Returns **CNTR_CODEList** – list of all Country codes present in the current DataFrame.

Return type list

emipy.processdata.**get_CountryList**(*db*)
Returns a list of all appearing countries in given dataframe.

Parameters **db** (*DataFrame*) – Data in which is looked for unique countries.

Returns **CountryList** – List of unique countries.

Return type List

emipy.processdata.**get_NACECode_filter**(*specify=None*)
If not specified, this function returns a dict with all stored NACECODE dictionaries. If specified, it returns the corresponding NACECODES as a list.

Parameters **specify** (*String/List of Strings, optional*) – Specify for wich economical categories you want to have the NACECODES. You can get a list of all selection options, with executing this function with specify=None. The default is None.

Returns **NACElist** – If specify is None it returns a Dict with all stored NACECODE dictionarys. If specify is not None it returns the according NACECODES in a list.

Return type Dict/List

emipy.processdata.**get_NACECode_filter_industry**(*group=None*)
Creates a list of NACE codes corresponding to the selected industry sectors.

Parameters **group** (*String, optional*) – industry sector. The default is None.

Returns **NACECode** – list of NACE codes corresponding to the specified industry sectors.

Return type List

emipy.processdata.**get_PollutantList**(*db*)
Returns a list of all appearing pollutant names in given dataframe.

Parameters **db** (*DataFrame*) – Data in which is looked for unique pollutant names.

Returns **PollutantList** – List of unique pollutant names.

Return type List

emipy.processdata.**get_YearList**(*db*)
Returns a list of all appearing reporting years in given dataframe.

Parameters **db** (*DataFrame*) – Data in which is looked for unique reporting years.

Returns **YearList** – List of unique reporting years.

Return type List

emipy.processdata.**perform_NACETransition**(*db*, *NewNACE=2*, *path=None*)
Changes the NACE_1_1 Codes of the input DataFrame into NACE_2 Codes.

Parameters

- **db** (*DataFrame*) – Input DataFrame with partly entries that are coded with NACE_1_1.
- **NewNACE** (*Int, optional*) – The target NACE-code. The default is 2.
- **path** (*String, optional*) – Path to the root of your project. If None is given, emipy searches for the path, stored in the config file. The default is None.

Returns **final** – The input DataFrame with changed NACE-codes if necessary.

Return type DataFrame

emipy.processdata.**read_db**(*path=None*, *NewData=False*)
Loads complete pollution record.

Parameters

- **path** (*String, optional*) –

  Path to the file, that is to be loaded. The file has to be a .pkl file. If None is given, the function loads the data file, stored in the emipy project, that is specified in the config file. The default is 'None'.
- **NewData** (*Boolean, optional*) – If this is set to True, the data base with data from 2017 - 2019 is loaded instead of the one with data from 2001 - 2017. The default is False.

Returns **db** – Pollution record for either the years 2001-2017 or 2017-2019.

Return type DataFrame

emipy.processdata.**read_mb**(*path=None*, *resolution='10M'*, *SpatialType='RG'*, *NUTS_LVL=0*, *m_year=2016*, *projection=4326*)
Reads the shp file with the specifications given in the input to load the map data.

Parameters

- **path** (*String, optional*) – Path of the file, that is to be loaded. The file has to be a .shp file. If None is given, the function loads the shp file in the emipy project, that is specified in the config file. The default is None.
- **resolution** (*String*) – Resolution of the map. The default is '10M'.
- **SpatialType** (*String*) – Format of data presentation. The default is 'RG'.
- **NUTS_LVL** (*Int, optional*) – NUTS-classification level, defined by the eurostat. The default is 0.
- **m_year** (*Int*) – Year of publication of the geographical data. The default is 2016.
- **projection** (*Int*) – Projection on the globe. The default is 4326.

Returns **mb** – DataFrame with geometry data.

Return type DataFrame

emipy.processdata.**rename_columns**(*db*)
Renames column names of the DataFrame, specified by the "COLUMNNAMES" dict in the config file.

Parameters **db** (*DataFrame*) – DataFrame which's column names should be changed.

Returns **db** – DataFrame with changed column names.

Return type DataFrame

emipy.processdata.**row_reduction**(*db*)
Reduces DataFrame to columns specified in the config file.

Parameters **db** (*DataFrame*) – DataFrame which data shall be reduced.

Returns **db** – DataFrame with reduced number of columns.

Return type DatFrame

4.3 Module visualizedata

This module contains all functions to visualize the data set.

emipy.visualizedata.**add_MarkerSize**(*gdf*, *MaxMarker*)
adds column MarkerSize to GeoDataFrame. If MaxMarker=0, all markers have size 1. Else, they are normalized to max value and multiplied by value of MaxMarker.

Parameters

- **gdf** (*GeoDataFrame*) – GeoDataFrame, which gets additional column.
- **MaxMarker** (*Int*) – defines the marker size of the biggest marker. If 0, all markers have same size.

Returns **gdf** – GeoDataFrame with added column 'MarkerSize'.

Return type GeoDataFrame

emipy.visualizedata.**change_proj**(*gdf*, *OutProj=None*)
Changes The projection of the input GeoDataFrame to the projection defined with OutProj. If no CRS is given for the geometry, the function tries to recover information from gdf.

Parameters

- **gdf** (*GeoDataFrame*) – Data that CRS is to be changed.
- **OutProj** (*Datatype, optional*) – Code for target output projection. See http://pyproj4.github.io/pyproj/stable/api/crs/crs.html#pyproj.crs.CRS.from_user_input for input possibilities. The default is None.

Returns **gdf** – Data with new projection in the geometry.

Return type GeoDataFrame

emipy.visualizedata.**create_GDFWithRightProj**(*dfgdf*, *OutProj=None*)
Converts DataFrame into GeoDataFrame and changes the projection if new projection is given as input.

Parameters

- **dfgdf** (*DataFrame/GeoDataFrame*) – Data that is about to be converted into a GeoDataFrame and experience a projection change if wanted.
- **OutProj** (*String, optional*) – Target projection of the geometry of the data. The default is None.

Returns **gdf** – Data stored as GeoDataFrame and with eventually changed geometry CRS.

Return type GeoDataFrame

emipy.visualizedata.**exclude_DataOutsideBorders**(*borders*, *gdf*)
seperates data, that are inside and outside given borders

Parameters

- **borders** (*list*) – x,y min/max.
- **gdf** (*GeoDataFrame*) – GeoDataFrame that is to process.

Returns

- **gdft** (*DataFrame*) – GeoDataFrame with data inside the borders.
- **gdff** (*DataFrame*) – GeoDataFrame with data outside the borders.

emipy.visualizedata.**export_fig**(*fig*, *path=None*, *filename=None*, ***kwargs*)

Exports the choosen figure to a given path or to the export folder of the project if no path is given.

Parameters

- **fig** (*figure*) – The figure that is to export.
- **path** (*String, optional*) – Path under which the file is stored. The filename has to be included. The default is None.
- **filename** (*String, optional*) – Filename under which the figure is stored in the Export folder of the project. The default is None.
- ****kwargs** (*TYPE*) – Matplotlib.savefig() input arguments.

Returns

Return type None.

emipy.visualizedata.**get_ImpurityVolume**(*db*, *target*, *FirstOrder='FacilityReportID'*, *ReleaseMediumName='Air'*, *absolute=False*, *FacilityFocus=True*, *impurity=None*, *statistics=False*)

Creates a table with the impurities of the target pollutant, sorted by the FirstOrder parameter. Putting the absolute parameter to True, gives absolute values instead of relative.

Parameters

- **db** (*DataFrame*) – Data to look for impurities.
- **target** (*String*) – Pollutant name of the pollutant, which is not seen as impurity.
- **FirstOrder** (*String, optional*) – Order to sort the impurities by. E.g. NACERegionGeoCode, FacilityReportID, NACEMainEconomicActivityCode. The default is 'FacilityReportID'.
- **ReleaseMediumName** (*String, optional*) – The release medium name in which the target is emitted and in which can be impurities. The default is 'Air'.
- **absolute** (*Boolean, optional*) – If this parameter is set on False, this function returns the impurity relative to the target pollutant emission. If it is set on True, the absolute emission value is returned. The default is False.
- **FacilityFocus** (*Boolean, optional*) – If this parameter is true, only the impurities in the facilities in which the target is emittet is taken in to consideration. If it is false, all data are taken into consideration. The default is True.
- **impurity** (*String, optional*) – With this parameter, you can specify the impurity pollutant you want to return. Otherwise, all present impurities are shown. The default is None.
- **statistics** (*Boolean, optional*) – If this argument is True, the statistics (determined by .describe()) of the output DataFrame are returned, instead of the usual impurity table. The default is False.

Returns

- **d2** (*DataFrame*) – Data table with the rows beeing the different present order values, and in the columns their respective emission of the target pollutant and the absolute emission of the impurities.
- **d3** (*DataFrame*) – Data table with the rows beeing the different present order values, and in the columns their respective emission of the target pollutant and the relative emission of the impurities.

emipy.visualizedata.**get_PollutantVolume**(*db*, *FirstOrder=None*, *SecondOrder=None*)

Sorts the input data table, to the named order parameters, which are all possible column names.

Parameters

- **db** (*DataFrame*) – input data table.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is None.
- **SecondOrder** (*TYPE, optional*) – Name of column, the data are sorted in the second order. The default is None.

Returns **data** – Data table, sorted to the announced order parameters.

Return type DataFrame

emipy.visualizedata.**get_PollutantVolumeChange**(*db*, *FirstOrder=None*, *SecondOrder=None*)

Derives the pollutant volume change to the previous year.

Parameters

- **db** (*DataFrame*) – the filtered input DataFrame.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is None.
- **SecondOrder** (*String, optional*) – Name of column, the data are sorted in the second order. The default is None.

Returns **data** – The change of TotalQuantity to the previous data entry

Return type DataFrame

emipy.visualizedata.**get_PollutantVolumeRel**(*db*, *FirstOrder=None*, *SecondOrder=None*, *normtop=None*, *normtov=None*)

Normalises the volume values to one specific value. This value is either the present max value of the returned data table or is specifed by normtop(osition) or normtov(alue).

Parameters

- **db** (*DataFrame*) – input data table.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is None.
- **SecondOrder** (*String, optional*) – Name of column, the data are sorted in the second order. The default is None.
- **normtop** (*list, optional*) – With this parameter you can choose a entry of your data table, that the entries should be normalised too. The first item of the list has to be one value of the FirstOrder. If SecondOrder is called, the second value has to be a value of the SecondOrder. The default is None.

- **normtov** (*float, optional*) – With this parameter you can define a value, that the PollutantVolume entries are normalised to. The default is None

Returns data – Data table sorted to the announced parameters. The values are normed to one specific max value. If normtop and normtov are both unequal None, no normalization is applied, since there is no concrete value, that can be normed to.

Return type DataFrame

emipy.visualizedata.**get_mb_borders**(*mb*)

Generates a list with the borders of the objects of a GeoDataFrame.

Parameters mb (*GeoDataFrame*) – Table of geo objects which over all borders are wanted.

Returns borders – The x,y min/max values.

Return type List

emipy.visualizedata.**map_PollutantRegions**(*db*, *mb*, *ReturnMarker=0*, **args*, ***kwargs*)

Visualizes the pollutant emission of regions with a color map. The classification of regions is selected with the choice of mb. If ReturnMarker is put on 1, the function returns a DataFrame with the plotted data. If ReturnMarker is put on 2, the function returns the DataFrame with Data that have no complementary NUTSID in the mapdata.

Parameters

- **db** (*DataFrame*) – Pollution data that are plotted.
- **mb** (*TYPE*) – Map data for plotting. The region selection corresponds to the selection of mb.
- **ReturnMarker** (*int*) – If it has the value 0, the function returns the plot. If put on 1, the function returns a DataFrame with all data that are plotted. If put on 2, the function returns a DataFrame with all data that are not plotted, because their NUTS_ID is not present in the mapdata.
- ***args** (*TYPE*) – Geopandas.plot() input arguments.
- ****kwargs** (*TYPE*) – Geopandas.plot() input arguments.

Returns

- **ax** (*Axes*) – Axes with colormap of the pollution emission.
- **dbp** (*DataFrame*) – Data that are plotted
- **dbna** (*DataFrame*) – Data that are not plotted, because the NUTS_ID is not present in the mapdata.

emipy.visualizedata.**map_PollutantSource**(*db*, *mb*, *category=None*, *MarkerSize=0*, *OutProj=None*, *ReturnMarker=0*, **args*, ***kwargs*)

maps pollutant sources given by db on map given by mb.

Parameters

- **db** (*DataFrame/GeoDataFrame*) – Data table on pollutant sources.
- **mb** (*DataFrame*) – geo data table.
- **category** (*String*) – The column name of db, which gets new colors for every unique entry.
- **MarkerSize** (*Int*) – maximal size of the largest marker.

- **OutProj** (*DataType*) – Code for targeted output projection. See http://pyproj4.github.io/pyproj/stable/api/crs/crs.html#pyproj.crs.CRS.from_user_input for input possibilities. The default is None.
- **ReturnMarker** (*Int*) – If put on 1, the function returns a DataFrame with all data that are plotted. If put on 2, the function returns a DataFrame with all data that are not plotted, because their coordinates are outside the geo borders.
- ***args** (*TYPE*) – Geopandas.plot() input arguments.
- ****kwargs** (*TYPE*) – Geopandas.plot() input arguments.

Returns

- **ax** (*Axes*) – Plot with pollutant sources on map.
- **gdfp** (*GeoDataFrame*) – GeoDataFrame with all sources that are within geo borders and therefore plotted.
- **gdfd** (*GeoDataFrame*) – GeoDataFrame with all sources that are outside geo borders and therefore dropped.

emipy.visualizedata.**plot_ImpurityVolume**(*db*, *target*, *impurity*, *FirstOrder='FacilityReportID'*, *ReleaseMediumName='Air'*, *absolute=False*, *FacilityFocus=True*, *statistics=False*, *PlotNA=True*, **args*, ***kwargs*)

Plots the impurities for the different FirstOrder values or the statistics of the entries.

Parameters

- **db** (*DataFrame*) – The data to be plotted.
- **target** (*String*) – The target pollutant which is impured.
- **impurity** (*String*) – The impurity which is to be analyzed.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is 'FacilityReportID'.
- **ReleaseMediumName** (*TYPE, optional*) – The release medium name in which the target is emitted and in which can be impurities. The default is 'Air'.
- **absolute** (*Boolean, optional*) – If this parameter is set on False, this function returns the impurity relative to the target pollutant emission. If it is set on True, the absolute emission value is returned. The default is False.
- **FacilityFocus** (*Boolean, optional*) – If this parameter is true, only the impurities in the facilities in which the target is emittet is taken in to consideration. If it is false, all data are taken into consideration. The default is True.
- **statistics** (*Boolean, optional*) – If this parameter is True, the statistics of the data are plotted. If it is False, the actual values are plotted. The default is False.
- **PlotNA** (*Boolean, optional*) – This argument is a option for discarding the na values if plotting the impurities. The default is True.
- ***args** (*TYPE*) – pandas.plot() input variables.
- ****kwargs** (*TYPE*) – pandas.plot() input variables.

Returns **ax** – Plot of the impurities in db, or the statistics of these impurities.

Return type Axes

emipy.visualizedata.**plot_PollutantVolume**(*db*, *FirstOrder=None*, *SecondOrder=None*, *stacked=False*, **args*, ***kwargs*)

Plots the filtered data set. The first order is the x-axis, the second order is a differentiation of the y-values.

Parameters

- **db** (*DataFrame*) – The data to be plotted.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is None.
- **SecondOrder** (*String, optional*) – Name of column, the data are sorted in the second order. The default is None.
- **stacked** (*Boolean, optional*) – Stacks the bars for second order. The default is False.
- ***args** (*TYPE*) – pandas.plot() input variables.
- ****kwargs** (*TYPE*) – pandas.plot() input variables.

Returns **ax** – Plot of the data in db, sorted by FirstOrder and SecondOrder if given.

Return type Axes

emipy.visualizedata.**plot_PollutantVolumeChange**(*db*, *FirstOrder=None*, *SecondOrder=None*, *stacked=False*, **args*, ***kwargs*)

Plots the volume change of the data set. The first order is the x-axis, the second order is a differentiation of the y-values.

Parameters

- **db** (*DataFrame*) – The data to be plotted.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is None.
- **SecondOrder** (*String, optional*) – Name of column, the data are sorted in the second order.. The default is None.
- **stacked** (*Boolean, optional*) – Stacks the bars for second order. The default is False.
- ***args** (*TYPE*) – pandas.plot() input variables.
- ****kwargs** (*TYPE*) – pandas.plot() input variables.

Returns **ax** – Plot of the data in db, sorted by FirstOrder and SecondOrder if given.

Return type Axes

emipy.visualizedata.**plot_PollutantVolumeRel**(*db*, *FirstOrder=None*, *SecondOrder=None*, *normtop=None*, *normtov=None*, *stacked=False*, **args*, ***kwargs*)

Plots the normed pollutant volume of the data set, The first order is the x-axis, the second order is a differentiation of the y-values.

Parameters

- **db** (*DataFrame*) – The data to be plotted.
- **FirstOrder** (*String, optional*) – Name of column, the data are sorted in the first order. The default is None.

- **SecondOrder** (*String, optional*) – Name of column, the data are sorted in the second order.. The default is None.
- **normtop** (*list, optional*) – With this parameter you can choose a entry of your data table, that the entries should be normalised too. The first item of the list has to be one value of the FirstOrder. If SecondOrder is called, the second value has to be a value of the SecondOrder. The default is None.
- **normtov** (*float, optional*) – With this parameter you can define a value, that the PollutantVolume entries are normalised to. The default is None.
- **stacked** (*Boolean, optional*) – Stacks the bars for second order. The default is False.
- ***args** (*TYPE*) – pandas.plot() input variables.
- ****kwargs** (*TYPE*) – pandas.plot() input variables.

Returns **ax** – Plot of the data in db, sorted by FirstOrder and SecondOrder if given.

Return type Axes

4.4 Module exporttocalliope

emipy.exporttocalliope.**export_calliope**(*data*, *path=None*, *yamlfilename='emipy2calliope.yaml'*, *csvfilename='emipy2calliope.csv'*, *sc=0.07*)

Exports the data to a csv file readable by the calliope project.

Parameters

- **data** (*DataFrame*) – Data that are to be exported.
- **path** (*String, optional*) – Path to the storage place. If None is given, emipy uses the path, stored in the config file. The default ist None.
- **yamlfilename** (*String, optional*) – filename for the yaml file. The default is emipy2calliope.yaml.
- **csvfilename** (*String, optional*) – filename for the csv file. The default is emipy2calliope.csv.
- **sc** (*int, optional*) – monetary cost factor. The default is 0.07.

Returns

Return type None.

emipy.exporttocalliope.**get_default_config**()

Returns the default configuration

Returns **d** – dictionary with default configuration.

Return type dict

emipy.exporttocalliope.**prepare_csv**(*data*)

Creates a DataFrame with the emission volume of the different facilitys distributed over a time series.

Parameters **data** (*DataFrame*) – Data from which a time series is to be generated.

Returns

- **df** (*DataFrame*) – Time series of the pollutant emission for all present facility ID's.

- **coords** (*dict*) – Dictionary that stores the coordinates of each facility.
- **FacilityIDDict** (*dict*) – Dictionary that stores the facility names of each facility.

CHAPTER 5

Get in touch

5.1 Find us on PyPi

Our package is hosted by PyPi.

5.2 Find us on Gitlab

The package development takes place on Gitlab.

5.3 Contact

If you want to reach out for us please contact p.boettcher@fz-juelich.de.

Python Module Index

e

Index

M

P

R