normal [LE,RO]1 [LO] [RE] [LE,RO], @chappos

pyPRIMA

Kais Siala
Houssame Houmy

Version 1.0.0

Jul 09, 2021

Contents

Chapter 1

User manual

1.1 Installation

Note: We assume that you are familiar with git and conda.

First, clone the git repository in a directory of your choice using a Command Prompt window:

```
$ ~\directory-of-my-choice> git clone https://github.com/tum-ens/pyPRIMA.git
```

We recommend using conda and installing the environment from the file `gen_mod.yml` that you can find in the repository. In the Command Prompt window, type:

```
$ cd pyPRIMA\env\
$ conda env create -f gen_mod.yml
```

Then activate the environment:

```
$ conda activate gen_mod
```

In the folder `code`, you will find multiple files:

File	Description
config.py	used for configuration, see below.
runme.py	main file, which will be run later using `python runme.py`.
lib\initialization.py	used for initialization.
lib\input_maps.py	used to generate input maps for the scope.
lib\generate-models.py	used to generate the model files from intermediate files.
lib\generate_intermediate_files.py	used to generate intermediate files from raw data.
lib\spatial_functions.py	contains helping functions related to maps, coordinates and indices.
lib\correction_functions.py	contains helping functions for data correction/cleaning.
lib\util.py	contains minor helping functions and the necessary python libraries to be imported.

1.2 config.py

This file contains the user preferences, the links to the input files, and the paths where the outputs should be saved. The paths are initialized in a way that follows a particular folder hierarchy. However, you can change the hierarchy as you wish.

1.2.1 Main configuration function

`config.`**`configuration`**`()`

This function is the main configuration function that calls all the other modules in the code.

Return (paths, param) The dictionary param containing all the user preferences, and the dictionary path containing all the paths to inputs and outputs.

Return type tuple(dict, dict)

`config.`**`general_settings`**`()`

This function creates and initializes the dictionaries param and paths. It also creates global variables for the root folder `root` and the system-dependent file separator `fs`.

Return (paths, param) The empty dictionary paths, and the dictionary param including some general information.

Return type tuple(dict, dict)

Note: Both *param* and *paths* will be updated in the code after running the function *`config.configuration`*.

Note: `root` points to the directory that contains all the inputs and outputs. All the paths will be defined relatively to the root, which is located in a relative position to the current folder.

The code differentiates between the geographic scope and the subregions of interest. You can run the first part of the script `runme.py` once and save results for the whole scope, and then repeat the second part using different subregions within the scope.

`config.`**`scope_paths_and_parameters`**`(`*paths*, *param*`)`

This function defines the path of the geographic scope of the output *spatial_scope* and of the subregions of interest *subregions*. It also associates two name tags for them, respectively *region_name* and *subregions_name*, which define the names of output folders. Both paths should point to shapefiles of polygons or multipolygons.

For *spatial_scope*, only the bounding box around all the features matters. Example: In case of Europe, whether a shapefile of Europe as one multipolygon, or as a set of multiple features (countries, states, etc.) is used, does not make a difference. Potential maps (theoretical and technical) will be later generated for the whole scope of the bounding box.

For *subregions*, the shapes of the individual features matter, but not their scope. For each individual feature that lies within the scope, you can later generate a summary report and time series. The shapefile of *subregions* does not have to have the same bounding box as *spatial_scope*. In case it is larger, features that lie completely outside the scope will be ignored, whereas those that lie partly inside it will be cropped using the bounding box of *spatial_scope*. In case it is smaller, all features are used with no modification.

year defines the year of the weather/input data, and *model_year* refers to the year to be modeled (could be the same as *year*, or in the future).

technology is a dictionary of the technologies (*Storage*, *Process*) to be used in the model. The names of the technologies should match the names which are used in assumptions_flows.csv, assumptions_processes.csv and assumptions_storage.csv.

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences.

Return (paths, param) The updated dictionaries paths and param.

Return type tuple of dict

Note: We recommend using a name tag that describes the scope of the bounding box of the regions of interest. For example, `'Europe'` and `'Europe_without_Switzerland'` will actually lead to the same output for the first part of the code.

1.2.2 User preferences

`config.`**`resolution_parameters`**(*param*)

This function defines the resolution of weather data (low resolution), and the desired resolution of output rasters (high resolution). Both are numpy array with two numbers. The first number is the resolution in the vertical dimension (in degrees of latitude), the second is for the horizontal dimension (in degrees of longitude).

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

Note: As of version 1.0.1, these settings should not be changed. Only MERRA-2 data can be used in the tool. Its spatial resolution is 0.5° of latitudes and 0.625° of longitudes. The high resolution is 15 arcsec in both directions.

`config.`**`grid_parameters`**(*param*)

This function defines parameters related to the grid to be used while cleaning the data.

- *quality* is a user assessment of the quality of the data. If the data is trustworthy, use 1, if it is not trustworthy at all, use 0. You can use values inbetween.
- *default* is a collection of default values for voltage, wires, cables, and frequency, to use when these data are missing.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

`config.`**`load_parameters`**(*param*)

This function defines the user preferences which are related to the load/demand. Currently, only one parameter is used, namely *default_sec_shares*, which sets the reference region to be used in case data for other regions is missing.

Parameters **param** (*dict*) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

`config.`**`processes_parameters`**(*param*)

This function defines parameters related to the processes in general, and to distributed renewable capacities in particular.

For *process*, only the parameter *cohorts* is currently used. It defines how power plants should be grouped according to their construction period. If *cohorts* is 5, then you will have groups of coal power plants from 1960, then another from 1965, and so on. If you do not wish to group the power plants, use the value 1.

For distributed renewable capacities, *dist_ren*, the following parameters are needed:

- *units* is a dictionary defining the standard power plant size for each distributed renewable technology in MW.
- *randomness* is a value between 0 and 1, defining the randomness of the spatial distribution of renewable capacities. The complementary value (1 - randomness) is affected by the values of the potential raster used for the distribution. When using a high resolution map, set *randomness* at a high level (close to 1), otherwise all the power plants will be located in a small area of high potential, close to each other.
- *default_pa_type* and *default_pa_availability* are two arrays defining the availability for each type of protected land. These arrays are used as default, along with the protected areas raster, in case no potential map is available for a distributed renewable technology.

Parameters **`param`** (`dict`) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

`config.`**`renewable_time_series_parameters`**(*param*)

This function defines parameters related to the renewable time series to be used in the models. In particular, the user can decide which *modes* to use from the files of the time series, provided they exist. See the repository tum-ens/renewable-timeseries for more information.

Parameters **`param`** (`dict`) – Dictionary including the user preferences.

Return param The updated dictionary param.

Return type dict

1.2.3 Paths

`config.`**`assumption_paths`**(*paths*)

This function defines the paths for the assumption files and the dictionaries.

- *assumptions_landuse* is a table with land use types as rows and sectors as columns. The table is filled with values between 0 and 1, so that each row has a total of 0 (no sectoral load there) or 1 (if there is a load, it will be distributed according to the shares of each sector).
- *assumptions_flows* is a table with the following columns:
  - *year*: data reference year.
  - *Process/Storage*: name of the process or storage type.
  - *Direction*: either `In` or `Out`. You can have multiple inputs and outputs, each one in a separate row.
  - *Commodity*: name of the input or output commodity.
  - *ratio*: ratio to the throughput, which is an intermediate level between input and output. It could be any positive value. The ratio of the output to the input corresponds to the efficiency.
  - *ratio-min* similar to *ratio*, but at partial load.
- *assumptions_processes* is a table with the following columns:
  - *year*: data reference year.
  - *Process*: name of the process usually given as the technology type.
  - *cap-lo*: minimum power capacity.
  - *cap-up*: maximum power capacity.
  - *max-grad*: maximum allowed power gradient (1/h) relative to power capacity.
  - *min-fraction*: minimum load fraction at which the process can run at.

  - *inv-cost*: total investment cost per power capacity (Euro/MW). It will be annualized in the model using an annuity factor derived from the wacc and depreciation period.
  - *fix-cost*: annual operation independent or fix cost (Euro/MW/a)
  - *var-cost*: variable cost per throughput energy unit(Euro/MWh) but excludes fuel costs.
  - *start-cost*: startup cost when the process is switch on from the off condition.
  - *wacc*: weighted average cost of capital. Percentage of cost of capital after taxes.
  - *depreciation*: deprecation period in years.
  - *lifetime*: lifetime of already installed capacity in years.
  - *area-per-cap*: area required per power capacity (m^2/MW).
  - *act-up*: maximal load (per unit).
  - *act-lo*: minimal load (per unit).
  - *on-off*: binary variable, 1 for controllable power plants, otherwise 0 (must-run).
  - *reserve-cost*: cost of power reserves (Euro/MW) (to be verified).
  - *ru*: ramp-up capacity (MW/MWp/min).
  - *rd*: ramp-down capacity (MW/MWp/min).
  - *rumax*: maximal ramp-up (MW/MWp/h).
  - *rdmax*: minimal ramp-up (MW/MWp/h).
  - *detail*: level of detail for modeling thermal power plants, modes 1-5.
  - *lambda*: cooling coefficient (to be verified).
  - *heatmax*: maximal heating capacity (to be verified).
  - *maxdeltaT*: maximal temperature gradient (to be verified).
  - *heatupcost*: costs of heating up (Euro/MWh_th) (to be verified).
  - *su*: ramp-up at start (to be verified).
  - *sd*: ramp-down at switch-off (to be verified).
  - *pdt*: (to be verified).
  - *hotstart*: (to be verified).
  - *pot*: (to be verified).
  - *pretemp*: temperature at the initial time step, per unit of the maximum operating temperature.
  - *preheat*: heat content at the initial time step, per unit of the maximum operating heat content.
  - *prestate*: operating state at the initial time step (binary).
  - *prepow*: available power at the initial time step (MW) (to be verified).
  - *precaponline*: online capacity at the initial time step (MW).
  - *year_mu*: average construction year for that type of power plants.
  - *year_stdev*: standard deviation from the average construction year for that type of power plants.
- *assumptions_storage* is a table with the following columns:
  - *year*: data reference year.
  - *Storage*: name of the storage usually given as the technology type.
  - *ep-ratio*: fixed energy to power ratio (hours).
  - *cap-up-c*: maximum allowed energy capacity (MWh)

  - *cap-up-p*: maximum allowed power capacity (MW)
  - *inv-cost-p*: total investment cost per power capacity (Euro/MW). It will be annualized in the model using an annuity factor derived from the wacc and depreciation period.
  - *inv-cost-c*: total investment cost per energy capacity (Euro/MWh). It will be annualized in the model using an annuity factor derived from the wacc and depreciation period.
  - *fix-cost-p*: annual operation independent or fix cost per power capacity (Euro/MW/a)
  - *fix-cost-c*: annual operation independent or fix cost per energy capacity (Euro/MWh/a)
  - *var-cost-p*: opertion dependent costs for input and output of energy per MWh_out stored or retreived (euro/MWh)
  - *var-cost-c*: operation dependent costs per MWh stored. This value can used to model technologies that have increased wear and tear proportional to the amount of stored energy.
  - *lifetime*: lifetime of already installed capacity in years.
  - *depreciation*: deprecation period in years.
  - *wacc*: weighted average cost of capital. Percentage of cost of capital after taxes.
  - *init*: initial storage content. Fraction of storage capacity that is full at the simulation start. This level has to be reached in the final timestep.
  - *var-cost-pi*: variable costs for charing (Euro/MW).
  - *var-cost-po*: variable costs for discharing (Euro/MW).
  - *act-lo-pi*: minimal share of active capacity for charging (per unit).
  - *act-up-pi*: maximal share of active capacity for charging (per unit).
  - *act-lo-po*: minimal share of active capacity for discharging (per unit).
  - *act-up-po*: maximal share of active capacity for discharging (per unit).
  - *act-lo-c*: minimal share of storage capacity (per unit).
  - *act-up-c*: maximal share of storage capacity (per unit).
  - *precont*: energy content of the storage unit at the initial time step (MWh) (to be verified).
  - *prepowin*: energy stored at the initial time step (MW).
  - *prepowout*: energy discharged at the initial time step (MW).
  - *ru*: ramp-up capacity (MW/MWp/min).
  - *rd*: ramp-down capacity (MW/MWp/min).
  - *rumax*: maximal ramp-up (MW/MWp/h).
  - *rdmax*: minimal ramp-up (MW/MWp/h).
  - *seasonal*: binary variable, 1 for seasonal storage.
  - *ctr*: binary variable, 1 if can be used for secondary reserve.
  - *discharge*: energy losses due to self-discharge per hour as a percentage of the energy capacity.
  - *year_mu*: average construction year for that type of storage.
  - *year_stdev*: standard deviation from the average construction year for that type of storage.

- *assumptions_commodities* is a table with the following columns:
  - *year*: data reference year.
  - *Commodity*: name of the commodity.
  - *Type_urbs*: type of the commodity according to urbs' terminology.
  - *Type_evrys*: type of the commodity according to evrys' terminology.

  - *price*: commodity price (euro/MWh).
  - *max*: maximum annual commodity use (MWh).
  - *maxperhour*: maximum commodity use per hour (MW).
  - *annual*: total value per year (MWh).
  - *losses*: losses (to be verified).
- *assumptions_transmission* is a table with the following columns:
  - *Type*: type of transmission.
  - *length_limit_km*: maximum length of the transmission line in km, for which the assumptions are valid.
  - *year*: data reference year.
  - *Commodity*: name of the commodity to be transported along the transmission line.
  - *eff_per_1000km*: transmission efficiency after 1000km in percent.
  - *inv-cost-fix*: length independent investment cost (euro).
  - *inv-cost-length*: length dependent investment cost (euro/km).
  - *fix-cost-length*: fixed annual cost dependent on the length of the line (euro/km/a).
  - *var-cost*: variable costs per energy unit transmitted (euro/MWh)
  - *cap-lo*: minimum required power capacity (MW).
  - *cap-up*: maximum allowed power capacity (MW).
  - *wacc*: weighted average cost of capital. Percentage of cost of capital after taxes.
  - *depreciation*: deprecation period in years.
  - *act-lo*: minimum capacity (MW/MWp).
  - *act-up*: maximum capacity (MW/MWp).
  - *angle-up*: maximum phase angle ramp-up (to be verified).
  - *PSTmax*: maximum phase angle difference.
- *dict_season_north* is a table with the following columns:
  - *Month*: number of the month (1-12).
  - *Season*: corresponding season.
- *dict_daytype* is a table with the following columns:
  - *Weak day*: name of the weekday (Monday-Sunday).
  - *Type*: either *Working day*, *Saturday*, or *Sunday*.
- *dict_sectors* is a table with the following columns:
  - *EUROSTAT*: name of the entry in the EUROSTAT table.
  - *Model_sectors*: corresponding sector (leave empty if irrelevant).
- *dict_counties* is a table with the following columns:
  - *IRENA*: names of countries in IRENA database.
  - *Counties shapefile*: names of countries in the countries shapefile.
  - *NAME_SHORT*: code names for the countries as used by the code.
  - *ENTSO-E*: names of countries in the ENTSO-E dataset.
  - *EUROSTAT*: names of countries in the EUROSTAT table.

- *dict_line_voltage* is a table with the following columns:
  - *voltage_kV*: sorted values of possible line voltages.
  - *specific_impedance_Ohm_per_km*: specific impedance (leave empty if unknown).
  - *loadability*: loadability factor according to the St Clair's curve (leave empty if unknown).
  - *SIL_MWh*: corresponding surge impedance load (leave empty if unknown).
- *dict_technologies* is a table with the following columns:
  - *IRENA*: names of technologies in the IRENA database.
  - *FRESNA*: names of technologies in the FRESNA database.
  - *Model names*: names of technologies as used in the model.

Parameters **`paths`** (`dict`) – Dictionary including the paths.

Returns The updated dictionary paths.

Return type dict

`config.`**`grid_input_paths`**(*paths*)

This function defines the paths where the transmission lines (inputs) are located.

Parameters **`paths`** (`dict`) – Dictionary including the paths.

Return paths The updated dictionary paths.

Return type dict

`config.`**`load_input_paths`**(*paths*)

This function defines the paths where the load related inputs are saved:

- *sector_shares* for the sectoral shares in the annual electricity demand.
- *load_ts* for the load time series.
- *profiles* for the sectoral load profiles.

Parameters **`paths`** (`dict`) – Dictionary including the paths.

Return paths The updated dictionary paths.

Return type dict

`config.`**`local_maps_paths`**(*paths*, *param*)

This function defines the paths where the local maps will be saved:

- *LAND* for the raster of land areas within the scope
- *EEZ* for the raster of sea areas within the scope
- *LU* for the land use raster within the scope
- *PA* for the raster of protected areas within the scope
- *POP* for the population raster within the scope

Parameters

- **`paths`** (`dict`) – Dictionary including the paths.
- **`param`** (`dict`) – Dictionary including the user preferences.

Return paths The updated dictionary paths.

Return type dict

`config.`**`output_folders`**(*paths*, *param*)

This function defines the paths to multiple output folders:

- *region* is the main output folder.
- *local_maps* is the output folder for the local maps of the spatial scope.
- *sites* is the output folder for the files related to the modeled sites.
- *load* is the output folder for the subregions-independent, load-related intermediate files.
- *load_sub* is the output folder for the subregions-dependent, load-related intermediate files.
- *grid* is the output folder for the subregions-independent, grid-related intermediate files.
- *grid_sub* is the output folder for the subregions-dependent, grid-related intermediate files.
- *regional_analysis* is the output folder for the regional analysis of renewable energy.
- *proc* is the output folder for the subregions-independent, process-related intermediate files.
- *proc_sub* is the output folder for the subregions-dependent, process-related intermediate files.
- *urbs* is the output folder for the urbs model input file.
- *evrys* is the output folder for the evrys model input files.

All the folders are created at the beginning of the calculation, if they do not already exist.

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences *region_name* and *subregions_name*.

Returns The updated dictionary paths.

Return type dict

`config.`**`output_paths`**(*paths*, *param*)

This function defines the paths to multiple output files.

Sites:

- *sites_sub* is the CSV output file listing the modeled sites and their attributes.

Load:

- *stats_countries* is the CSV output file listing some load statistics on a country level.
- *load_ts_clean* is the CSV output file with cleaned load time series on a country level.
- *cleaned_profiles* is a dictionary of paths to the CSV file with cleaned load profiles for each sector.
- *df_sector* is the CSV output file with load time series for each sector on a country level.
- *load_sector* is the CSV output file with yearly electricity demand for each sector and country.
- *load_landuse* is the CSV output file with load time series for each land use type on a country level.
- *intersection_subregions_countries* is a shapefile where the polygons are the outcome of the intersection between the countries and the subregions.
- *stats_country_parts* is the CSV output file listing some load statistics on the level of country parts.
- *load_ts_clean* is the CSV output file with load time series on the level of subregions.

Grid:

- *grid_expanded* is a CSV file including a reformatted table of transmission lines.
- *grid_filtered* is a CSV file obtained after filtering out erronous/useless data points.
- *grid_corrected* is a CSV file obtained after correcting erronous data points.
- *grid_filled* is a CSV file obtained after filling missing data with default values.

- *grid_cleaned* is a CSV file obtained after cleaning the data and reformatting the table.
- *grid_shp* is a shapefile of the transmission lines.
- *grid_completed* is a CSV file containing the aggregated transmission lines between the subregions and their attributes.

Renewable processes:

- *IRENA_summary* is a CSV file with a summary of renewable energy statistics for the countries within the scope.
- *locations_ren* is a dictionary of paths pointing to shapefiles of possible spatial distributions of renewable power plants.
- *potential_ren* is a CSV file with renewable potentials.

Other processes and storage:

- *process_raw* is a CSV file including aggregated information about the power plants before processing it.
- *process_filtered* is a CSV file obtained after filtering out erronous/useless data points.
- *process_joined* is a CSV file obtained after joining the table with default attribute assumptions (like costs).
- *process_completed* is a CSV file obtained after filling missing data with default values.
- *process_cleaned* is a CSV file obtained after cleaning the data and reformatting the table.
- *process_regions* is a CSV file containing the power plants for each subregion.
- *storage_regions* is a CSV file containing the storage devices for each subregion.
- *commodities_regions* is a CSV file containing the commodities for each subregion.

Framework models:

- *urbs_model* is the urbs model input file.
- *evrys_model* is the evrys model input file.

Parameters

- **`paths`** (*`dict`*) – Dictionary including the paths.
- **`param`** (*`dict`*) – Dictionary including the user preferences *region_name*, *subregions_name*, and *year*.

Returns The updated dictionary paths.

Return type dict

`config.`**`processes_input_paths`**(*paths*, *param*)

This function defines the paths where the process-related inputs are located:

- *IRENA*: IRENA electricity statistics (useful to derive installed capacities of renewable energy technologies).
- *dist_ren*: dictionary of paths to rasters defining how the potential for the renewable energy is spatially distributed. The rasters have to be the same size as the spatial scope.
- *FRESNA*: path to the locally saved FRESNA database.

Parameters

- **`paths`** (*`dict`*) – Dictionary including the paths.
- **`param`** (*`dict`*) – Dictionary including the parameter *year*.

Return paths The updated dictionary paths.

Return type dict

`config.`**`renewable_time_series_paths`**(*paths*, *param*)

This function defines the paths where the renewable time series (inputs) are located. *TS_ren* is itself a dictionary with the keys *WindOn*, *WindOff*, *PV*, *CSP* pointing to the individual files for each technology.

Parameters

- **`paths`** (*`dict`*) – Dictionary including the paths.
- **`param`** (*`dict`*) – Dictionary including the parameters *region_name*, *subregions_name*, and *year*.

Return paths The updated dictionary paths.

Return type dict

1.3 runme.py

`runme.py` calls the main functions of the code:

```python
from lib.initialization import initialization
from lib.generate_intermediate_files import *
from lib.correction_functions import *
from lib.generate_models import *

if __name__ == "__main__":
    paths, param = initialization()

    ## Clean raw data
    clean_residential_load_profile(paths, param)
    clean_commercial_load_profile(paths, param)
    clean_industry_load_profile(paths, param)
    clean_agriculture_load_profile(paths, param)
    clean_streetlight_load_profile(paths, param)
    clean_GridKit_Europe(paths, param)
    clean_sector_shares_Eurostat(paths, param)
    clean_load_data_ENTSOE(paths, param)
    distribute_renewable_capacities_IRENA(paths, param)
    clean_processes_and_storage_FRESNA(paths, param)

    ## Generate intermediate files
    generate_sites_from_shapefile(paths, param)
    generate_load_timeseries(paths, param)
    generate_transmission(paths, param)
    generate_intermittent_supply_timeseries(paths, param)
    generate_processes(paths, param)
    generate_storage(paths, param)
    generate_commodities(paths, param)

    ## Generate model files
    generate_urbs_model(paths, param)
    generate_evrys_model(paths, param)
```

1.4 Recommended input sources

1.4.1 Load time series for countries

ENTSO-E publishes (or used to publish - the service has been discontinued as of November 2019) hourly load profiles for each country in Europe that is part of ENTSO-E.

1.4.2 Sectoral load profiles

The choice of the load profiles is not too critical, since the sectoral load profiles will be scaled according to their shares in the yearly demand, and their shapes edited to match the hourly load profile. Nevertheless, examples of load profiles for Germany can be obtained from the BDEW.

1.4.3 Sector shares (demand)

The sectoral shares of the annual electricity demand can be obtained from Eurostat. The table reference is *nrg_105a*. Follow these instructions to obtain the file as needed by the code:

- GEO: Choose all countries, but not EU
- INDIC_NRG: Choose all indices
- PRODUCT: Electrical energy (code 6000)
- TIME: Choose years
- UNIT: GWh
- Download in one single csv file

1.4.4 Power plants and storage units

The powerplantmatching package within FRESNA extracts a standardized power plant database that combines several other databases covering Europe. In this repository, all non-renewable power plants, all storage units, and some renewable power plants (e.g. geothermal) are obtained from this database. Since the capacities for most renewable technologies are inaccurate, they are obtained from another source (see below).

1.4.5 Renewable installed capacities

Renewable electricity capacity and generation statistics are obtained from the Query Tool of IRENA. The user has to create a query that includes all countries (but no groups of countries, such as continents), all technologies (but no groups of technology) for a particular year and name the file `IRENA_RE_electricity_statistics_allcountries_alltech_YEAR.csv`. This dataset has a global coverage, however it does not provide the exact location of each project. The code includes an algorithm to distribute the renewable capacities spatially.

1.4.6 Renewable potential maps

These maps are needed to distribute the renewable capacities spatially, since IRENA does not provide their exact locations. You can use any potential maps, provided that they have the same extent as the geographic scope. Adjust the resolution parameters in `config.py` accordingly. Such maps can be generated using the GitHub repository tum-ens/pyGRETA.

1.4.7 Renewable time series

Similarly, the renewable time series can be generated using the GitHub repository tum-ens/pyGRETA. This repository is particularly is the model regions are unconventional.

1.4.8 Transmission lines

High-voltage power grid data for Europe and North America can be obtained from GridKit, which used OpenStreetMap as a primary data source. In this repository, we only use the file with the lines (links.csv). In general, the minimum requirements for any data source are that the coordinates for the line vertices and the voltage are provided.

1.4.9 Other assumptions

Currently, other assumptions are provided in tables filled by the modelers. Ideally, machine-readable datasets providing the missing information are collected and new modules are written to read them and extract that information.

1.5 Recommended workflow

The script is designed to be modular and split into three main modules: `lib.correction_functions`, `lib.generate_intermediate_files`, and *`lib.generate_models`*.

Warning: The outputs of each module serve as inputs to the following module. Therefore, the user will have to run the script sequentially.

The recommended use cases of each module will be presented in the order in which the user will have to run them.

1. *Correction and cleaning of raw input data*
2. *Generation of intermediate files*
3. *Generation of model input files*

The use cases associated with each module are presented below.

It is recommended to thoroughly read through the configuration file *config.py* and modify the input paths and computation parameters before starting the *runme.py* script. Once the configuration file is set, open the *runme.py* file to define what use case you will be using the script for.

1.5.1 Correction and cleaning of raw input data

Each function in this module is designed for a specific data set (usually mentioned at the end of the function name). The pre-processing steps include filtering, filling in missing values, correcting/overwriting erronous values, aggregating and disaggregating entries, and deleting/converting/renaming the attributes.

At this stage, the obtained files are valid for the whole geographic scope, and do not depend on the model regions.

1.5.2 Generation of intermediate files

The functions in this module read the cleaned input data, and adapts it to the model regions. They also expand the attributes based on assumptions to cover all the data needs of all the supported models. The results are saved in individual CSV files that are model-independent. These files can be shared with modelers whose models are not supported, and they might be able to adjust them according to their model input requirements, and use them.

1.5.3 Generation of model input files

Here, the input files are adapted to the requirements of the supported model frameworks (currently urbs and evrys). Input files as needed by the scripts of urbs and evrys are generated at the end of this step.

Chapter 2

Theory

This chapters explains how the load time series are disaggregated spatially and according to sectors, then aggregated again according to the desired model regions.

2.1 Purpose

Load time series are widely available, but the published datasets are usually restricted to predefined spatial regions such as countries and their administrative subdivisions. The generate_load_timeseries() function takes the datasets which are available for these regions and disaggregate them according to a set of parameters, before aggregating them at a different spatial level. It is then possible to obtain time series for any region.

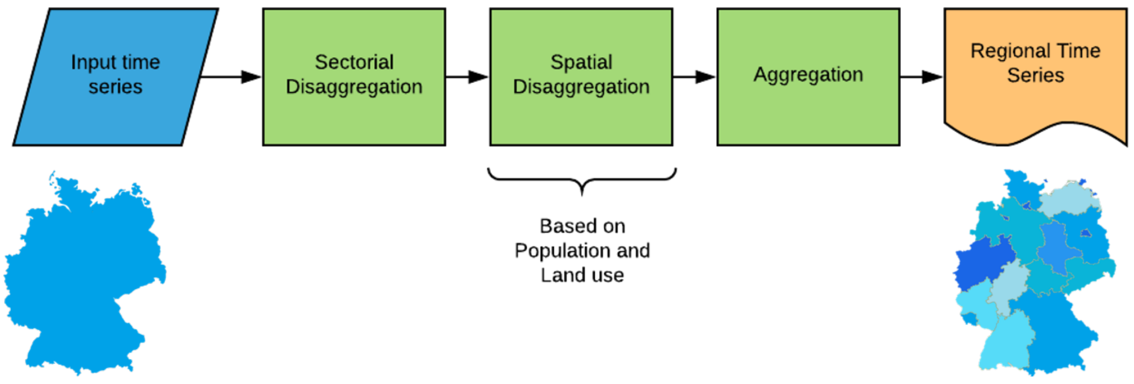

Fig. 1: Description of lib.generate_intermediate_files.generate_load_timeseries

2.2 Inputs

The main inputs of this script are:

- Load time series of the countries or regions to be disaggregated (hourly).
- Shapefiles of the countries or regions to be disaggregated
- Shapefiles of the regions of interest (subregions)
- Assumptions (land use and sector correspondence)
- Load profiles of the different sectors
- Raster of the population and land use correspondent to the country or region

2.3 Sectoral disaggregation

The load is assumed to be perfectly divided into four distinct sectors (load sources):

- Commercial
- Industrial
- Residential
- Agricultural

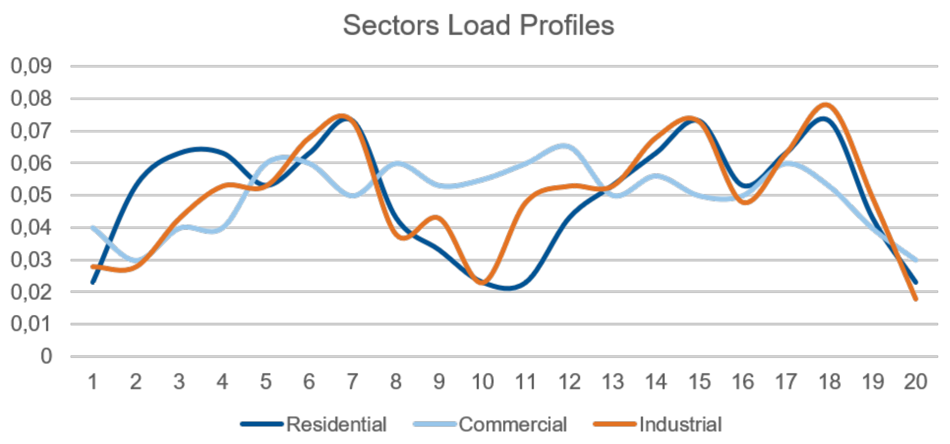

Fig. 2: Sectoral load profiles

Sectoral load shares:

Region	Industry	Commerce	Residential	Agriculture
A	0.41%	0.28%	0.29%	0.02%
B	0.31%	0.30%	0.38%	0.01%
C	0.44%	0.30%	0.25%	0.01%

An hourly load profile for each sector has been predefined for one week, the same load profile is assumed to repeat over the year. These load profiles are scaled and normalized based on sectoral load shares for each region(assumed to be constant throughout the spatial scope), by multiplying the load profiles by their corresponding share and normalizing their hourly sum to be equal to 1.

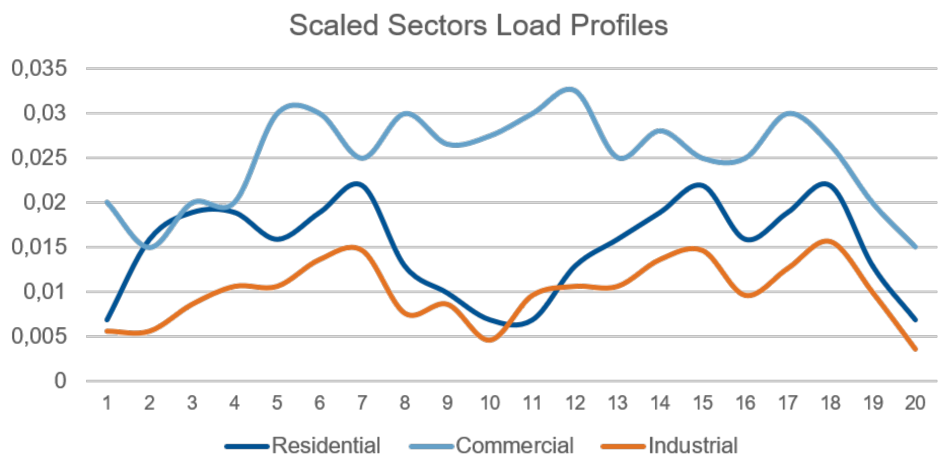

Fig. 3: Scaled sectoral load profiles

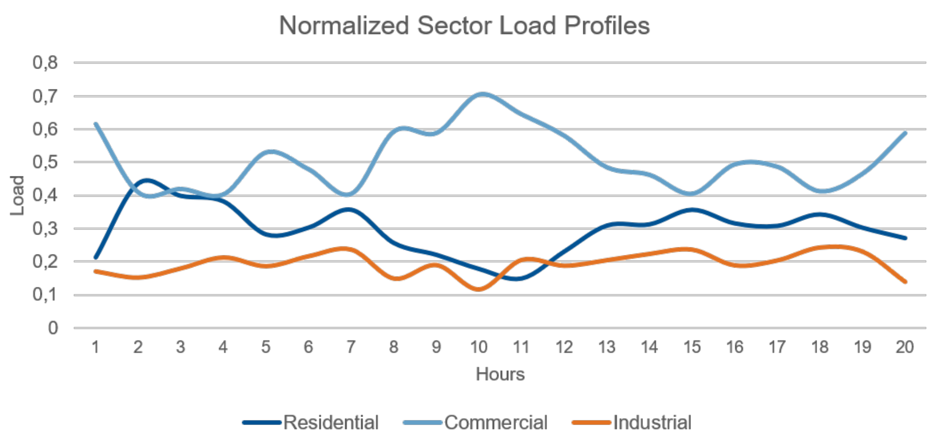

Fig. 4: Normalized sectoral load profiles

Once the load profiles are normalized, we can multiply them with the actual load time series to obtain the load timeseries for each sector.

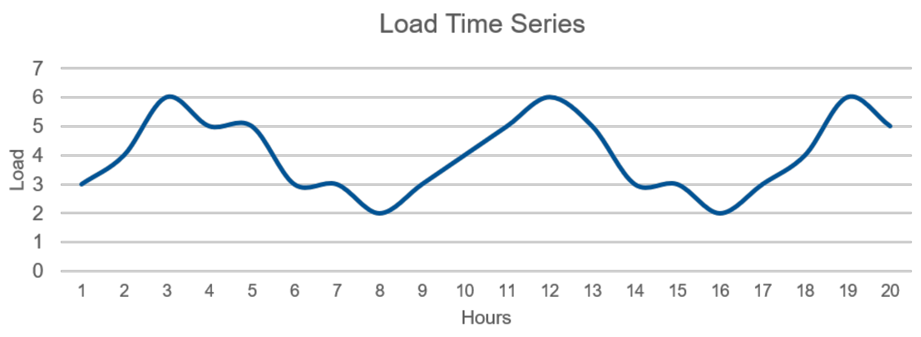

Fig. 5: Load time series

2.4 Spatial disaggregation

The next step is the spatial disaggregation, based on the land use and population concentration rasters. First, each land-use type is assigned a sectoral load percentage corresponding to the load components of the land use category. Then, the population concentration raster is used to calculate the population of each pixel.

Counting the pixels, and land use occurrences inside of region for which the sectoral load timeseries has been calculated, the sectoral load for the Industry, commercial, and agricultural can be retrieved for each pixel of that region. Similarly, the residential load timeseries can be assigned to each pixel based on the population contained in the said pixel. The spatial disaggregation results in the assignment to every pixel inside a given region to be assigned a specific sectoral load timeseries.

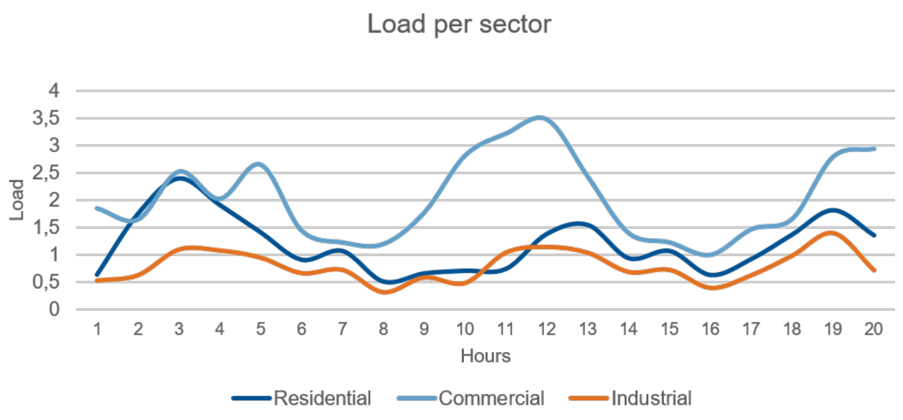

Fig. 6: Sectoral load time series

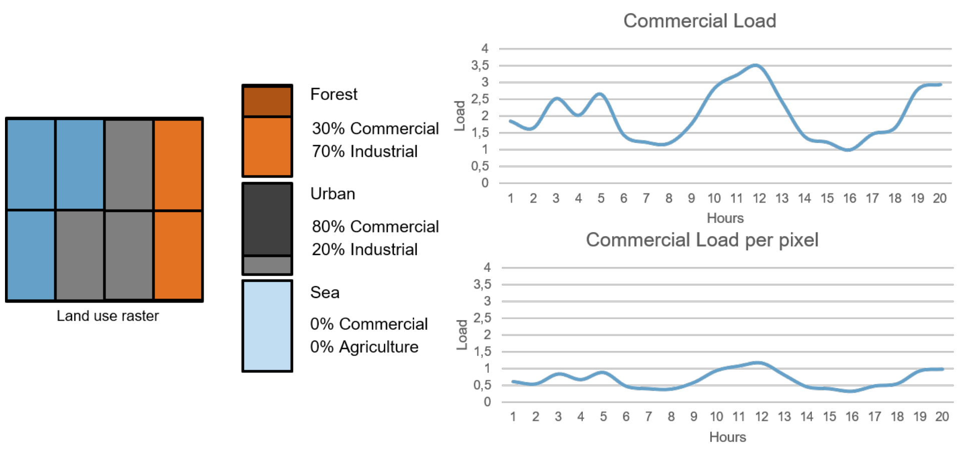

Fig. 7: Example - Commerce sector spatial disaggregation

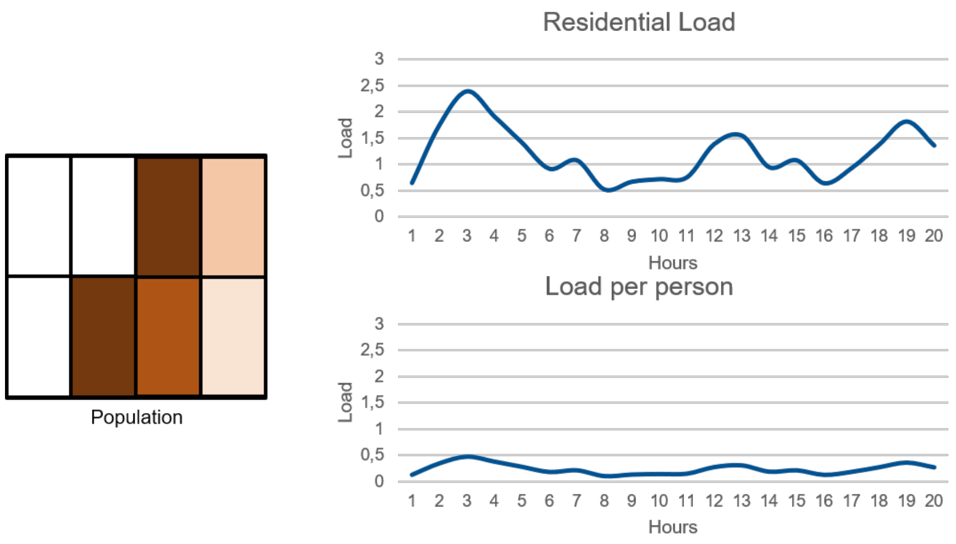

Fig. 8: Example - Residential sector spatial disaggregation

2.5 Re-aggregation

The result of the sectoral and spatial disaggregation, performed in the first two sections can be used to retrieve the sectoral load timeseries and, therefore, the general load time series of any desired region by summing up the loads of every pixel contained within the region. If a subregion spans more than one region or country, it is divided into subregions restrained to each of those countries.

Chapter 3

Implementation

Start with the configuration:

You can run the code by typing:

```
$ python runme.py
```

`runme.py` calls the main functions of the code, which are explained in the following sections.

3.1 initialization.py

Helping functions for the models are included in `generate_intermediate_files.py`, `correction_functions.py`, `spatial_functions.py`, and `input_maps.py`.

3.2 generate_intermediate_files.py

3.3 correction_functions.py

3.4 spatial_functions.py

3.5 input_maps.py

Utility functions as well as imported libraries are included in `util.py`.

3.6 util.py

`lib.util.`**`assign_values_based_on_series`**(*series*, *dict*)

This function fills a series based on the values of another series and a dictionary. The dictionary does not have to be sorted, it will be sorted before assigning the values. However, it must contain a key that is greater than any value in the series. It is equivalent to a function that maps ranges to discrete values.

Parameters

- **`series`** (*`pandas series`*) – Series with input values that will be mapped.
- **`dict`** (*`dictionary`*) – Dictionary defining the limits of the ranges that will be mapped.

Return result Series with the mapped discrete values.

Return type pandas series

`lib.util.`**`changem`**(*A*, *newval*, *oldval*)

This function replaces existing values *oldval* in a data array *A* by new values *newval*.

oldval and *newval* must have the same size.

Parameters

- **`A`** (`numpy array`) – Input matrix.
- **`newval`** (`numpy array`) – Vector of new values to be set.
- **`oldval`** (`numpy array`) – Vector of old values to be replaced.

Return Out The updated array.

Return type numpy array

`lib.util.`**`create_json`**(*filepath*, *param*, *param_keys*, *paths*, *paths_keys*)

This function creates a metadata JSON file containing information about the file in filepath by storing the relevant keys from both the param and path dictionaries.

Parameters

- **`filepath`** (`string`) – Path to the file for which the JSON file will be created.
- **`param`** (`dict`) – Dictionary of dictionaries containing the user input parameters and intermediate outputs.
- **`param_keys`** (`list of strings`) – Keys of the parameters to be extracted from the *param* dictionary and saved into the JSON file.
- **`paths`** (`dict`) – Dictionary of dictionaries containing the paths for all files.
- **`paths_keys`** (`list of strings`) – Keys of the paths to be extracted from the *paths* dictionary and saved into the JSON file.

Returns The JSON file will be saved in the desired path *filepath*.

Return type None

`lib.util.`**`display_progress`**(*message*, *progress_stat*)

This function displays a progress bar for long computations. To be used as part of a loop or with multiprocessing.

Parameters

- **`message`** (`string`) – Message to be displayed with the progress bar.
- **`progress_stat`** (`tuple(int, int)`) – Tuple containing the total length of the calculation and the current status or progress.

Returns The status bar is printed.

Return type None

`lib.util.`**`expand_dataframe`**(*df*, *column_names*)

This function reads a dataframe where columns with known *column_names* have multiple values separated by a semicolon in each entry. It expands the dataframe by creating a row for each value in each of these columns.

Parameters

- **`df`** (`pandas dataframe`) – The original dataframe, with multiple values in some entries.
- **`column_names`** (`list`) – Names of columns where multiple values have to be separated.

Return df_final The expanded dataframe, where each row contains only one value per column.

Return type pandas dataframe

`lib.util.`**`field_exists`**`(`*field_name*, *shp_path*`)`

This function returns whether the specified field exists or not in the shapefile linked by a path.

Parameters

- **`field_name`** (*`str`*) – Name of the field to be checked for.
- **`shp_path`** (*`str`*) – Path to the shapefile.

Returns `True` if it exists or `False` if it doesn't exist.

Return type bool

`lib.util.`**`get_sectoral_profiles`**`(`*paths*, *param*`)`

This function reads the raw standard load profiles, repeats them to obtain a full year, normalizes them so that the sum is equal to 1, and stores the obtained load profile for each sector in the dataframe *profiles*.

Parameters

- **`paths`** (*`dict`*) – Dictionary containing the paths to *dict_daytype*, *dict_season*, and to the raw standard load profiles.
- **`param`** (*`dict`*) – Dictionary containing the *year* and load-related assumptions.

Return profiles The normalized load profiles for the sectors.

Return type pandas dataframe

`lib.util.`**`resizem`**`(`*A_in*, *row_new*, *col_new*`)`

This function resizes regular data grid, by copying and pasting parts of the original array.

Parameters

- **`A_in`** (*`numpy array`*) – Input matrix.
- **`row_new`** (*`integer`*) – New number of rows.
- **`col_new`** (*`integer`*) – New number of columns.

Return A_out Resized matrix.

Return type numpy array

`lib.util.`**`reverse_lines`**`(`*df*`)`

This function reverses the line direction if the starting point is alphabetically after the end point.

Parameters **`df`** (*`pandas dataframe`*) – Dataframe with columns 'Region_start' and 'Region_end'.

Returns df_final The same dataframe after the line direction has been reversed.

Return type pandas dataframe

`lib.util.`**`timecheck`**`(`**args*`)`

This function prints information about the progress of the script by displaying the function currently running, and optionally an input message, with a corresponding timestamp. If more than one argument is passed to the function, it will raise an exception.

Parameters **`args`** (*`string`*) – Message to be displayed with the function name and the timestamp (optional).

Returns The time stamp is printed.

Return type None

Raise Too many arguments have been passed to the function, the maximum is only one string.

Finally, the module `generate_models.py` contains formating functions that create the input files for the urbs and evrys models.

3.7 generate_models module.py

`lib.generate_models.`**`generate_evrys_model`**(*paths*, *param*)

This function reads all the intermediate CSV files, adapts the formatting to the structure of the evrys Excel input file, and combines the datasets into one dataframe. It writes the dataframe into an evrys input Excel file. The function would still run even if some files have not been generated. They will simply be skipped.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the intermediate files *sites_sub*, *commodities_regions*, *process_regions*, *grid_completed*, *storage_regions*, *load_regions*, *potential_ren*, and to the output *evrys_model*.
- **param** (*dict*) – Dictionary of user preferences, including *model_year* and *technology*.

Returns The XLSX model input file is saved directly in the desired path.

Return type None

`lib.generate_models.`**`generate_urbs_model`**(*paths*, *param*)

This function reads all the intermediate CSV files, adapts the formatting to the structure of the urbs Excel input file, and combines the datasets into one dataframe. It writes the dataframe into an urbs input Excel file. The function would still run even if some files have not been generated. They will simply be skipped.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the intermediate files *sites_sub*, *commodities_regions*, *process_regions*, *assumptions_flows*, *grid_completed*, *storage_regions*, *load_regions*, *potential_ren*, and to the output *urbs_model*.
- **param** (*dict*) – Dictionary of user preferences, including *model_year* and *technology*.

Returns The XLSX model input file is saved directly in the desired path.

Return type None

Python Module Index

Index