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

pyGRETA

Kais Siala
Houssame Houmy
Sergio Alejandro Huezo Rodriguez

Version 1.0.1

Apr 20, 2022

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/pyGRETA.git
```

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

```
$ cd pyGRETA\env\
$ conda env create -f ren_ts_new.yml
```

Then activate the environment:

```
$ conda activate ren_ts_new
```

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\potential.py	contains functions related to the potential estimation.
lib\time_series.py	contains functions related to the generation of time series.
lib\regression.py	contains functions related to the regression.
lib\spatial_functions.py	contains helping functions related to maps, coordinates and indices.
lib\physical_models.py	contains helping functions for the physical/technological modeling.
lib\correction_functions.py	contains helping functions for data correction/cleaning.
lib\util.py	contains minor helping functions and the necessary python libraries to 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`**(*config_file*)

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

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

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*, *config_file*)

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

- 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 input data.
- *technology* defines the list of technologies that you are interested in. Currently, four technologies are defined: onshore wind `'WindOn'`, offshore wind `'WindOff'`, photovoltaics `'PV'`, concentrated solar power `'CSP'`.

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(dict, 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.

Note: As of version 1.1.0, it is possible to use different technologies in the same run, but not the same technology with different settings.

Warning: If you intend to use the wind correction feature relying on the Global Wind Atlas, it is recommended that *spatial_scope* covers **all** the countries that you are interested in, because the correction is done on a country-level. Also, you have to download the data from the Global Wind Atlas for each country that lies within the scope, even partially, and put it in the corresponding location.

1.2.2 User preferences

`config.`**`computation_parameters`**(*param*)

This function defines parameters related to the processing:

- *nproc* is an integer that limits the number of parallel processes (some modules in `potential.py` and `time_series.py` allow parallel processing).
- *CPU_limit* is a boolean parameter that sets the level of priority for all processes in the multiprocessesing. Leave `True` if you plan on using the computer while FLH and TS are being computed, `False` for fastest computation time.

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

Return param The updated dictionary param.

Return type dict

`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 arrays 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 1.1.0, 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.`**`csp_parameters`**(*param*)

This function sets the parameters for concentrated solar power in the dictionary *csp* inside param:

- *resource* is a dictionary including the parameters related to the resource potential:

  - *clearness_correction* is a factor that will be multiplied with the clearness index matrix to correct it. If no correction is required, leave it equal to 1.

- *technical* is a dictionary including the parameters related to the module:
  - *T_avg_HTF* is the average temperature in °C of the heat transfer fluid between the inlet and outlet of the solar field.
  - *loss_coeff* is the the heat loss coefficient in W/(m^{2}K), which does not depend on wind speed (relevant for `physical_models.calc_CF_solar`).
  - *loss_coeff_wind* is the the heat loss coefficient in W/(m^{2}K(m/s)^0.6), which depends on wind speed (relevant for `physical_models.calc_CF_solar`).
  - *Flow_coeff* is a factor smaller than 1 for the heat transfer to the HTF (Flow or heat removal factor).
  - *AbRe_ratio* is the ratio between the receiver area and the concentrator aperture.
  - *Wind_cutoff* is the maximum wind speed for effective tracking in m/s.
- *mask* is a dictionary including the parameters related to the masking:
  - *slope* is the threshold slope in percent. Areas with a larger slope are excluded.
  - *lu_suitability* is a numpy array of values 0 (unsuitable) or 1 (suitable). It has the same size as the array of land use types.
  - *pa_suitability* is a numpy array of values 0 (unsuitable) or 1 (suitable). It has the same size as the array of protected area categories.
- *weight* is a dictionary including the parameters related to the weighting:
  - *lu_availability* is a numpy array of values between 0 (completely not available) and 1 (completely available). It has the same size as the array of land use types.
  - *pa_availability* is a numpy array of values between 0 (completely not available) and 1 (completely available). It has the same size as the array of protected area categories.
  - *power_density* is the power density of CSP projects in MW/m^2.
  - *f_performance* is a number smaller than 1, taking into account all the other losses from the CSP module until the AC substation.

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

Return param The updated dictionary param.

Return type dict

`config.`**`file_saving_options`**(*param*)

This function sets some options for saving files.

- *savetiff* is a boolean that determines whether tif rasters for the potentials are saved (`True`), or whether only mat files are saved (`False`). The latter are saved in any case.
- *report_sampling* is an integer that sets the sample size for the sorted FLH values per region (relevant for `potential.reporting`).

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

Return param The updated dictionary param.

Return type dict

`config.`**`landuse_parameters`**(*param*)

This function sets the land use parameters in the dictionary *landuse* inside param:

- *type* is a numpy array of integers that associates a number to each land use type.

- *type_urban* is the number associated to urban areas (useful for `input_maps.generate_buffered_population`).
- *Ross_coeff* is a numpy array of Ross coefficients associated to each land use type (relevant for `physical_models.loss`).
- *albedo* is a numpy array of albedo coefficients between 0 and 1 associated to each land use type (relevant for reflected irradiation, see `physical_models.calc_CF_solar`).
- *hellmann* is a numpy array of Hellmann coefficients associated to each land use type (relevant for `correction_functions.generate_wind_correction`).

param param Dictionary including the user preferences.

type param dict

return param The updated dictionary param.

rtype dict

 # Landuse reclassification # 0 No data # 10 Cropland, rain-fed # 11 Herbaceous cover # 12 Tree or shrub cover # 20 Cropland, irrigated or post-flooding # 30 Mosaic cropland (>50%) / natural vegetation (tree, shrub, herbaceous cover) (<50%) # 40 Mosaic natural vegetation (tree, shrub, herbaceous cover) (>50%) / cropland (<50%) # 50 Tree cover, broadleaved, evergreen, closed to open (>15%) # 60 Tree cover, broadleaved, deciduous, closed to open (>15%) # 61 Tree cover, broadleaved, deciduous, closed (>40%) # 62 Tree cover, broadleaved, deciduous, open (15-40%) # 70 Tree cover, needleleaved, evergreen, closed to open (>15%) # 71 Tree cover, needleleaved, evergreen, closed (>40%) # 72 Tree cover, needleleaved, evergreen, open (15-40%) # 80 Tree cover, needleleaved, deciduous, closed to open (>15%) # 81 Tree cover, needleleaved, deciduous, closed (>40%) # 82 Tree cover, needleleaved, deciduous, open (15-40%) # 90 Tree cover, mixed leaf type (broadleaved and needleleaved) # 100 Mosaic tree and shrub (>50%) / herbaceous cover (<50%) # 110 Mosaic herbaceous cover (>50%) / tree and shrub (<50%) # 120 Shrubland # 121 Shrubland evergreen # 122 Shrubland deciduous # 130 Grassland # 140 Lichens and mosses # 150 Sparse vegetation (tree, shrub, herbaceous cover) (<15%) # 151 Sparse tree (<15%) # 152 Sparse shrub (<15%) # 153 Sparse herbaceous cover (<15%) # 160 Tree cover, flooded, fresh or brakish water # 170 Tree cover, flooded, saline water # 180 Shrub or herbaceous cover, flooded, fresh/saline/brakish water # 190 Urban areas # 200 Bare areas # 201 Consolidated bare areas # 202 Unconsolidated bare areas # 210 Water bodies # 220 Permanent snow and ice

`config.`**`offshore_wind_paramters`**(*param*)

This function sets the parameters for offshore wind in the dictionary *windoff* inside param:

- *resource* is a dictionary including the parameters related to the resource potential:
  - *res_correction* is either 1 (perform a redistribution of wind speed when increasing the resolution) or 0 (repeat the same value from the low resolution data). It is relevant for `correction_functions.generate_wind_correction`.
- *technical* is a dictionary including the parameters related to the wind turbine:
  - *w_in* is the cut-in speed in m/s.
  - *w_r* is the rated wind speed in m/s.
  - *w_off* is the cut-off wind speed in m/s.
  - *P_r* is the rated power output in MW.
  - *hub_height* is the hub height in m.
- *mask* is a dictionary including the parameters related to the masking:
  - *depth* is the threshold depth in meter (negative number). Areas that are deeper are excluded.
  - *pa_suitability* is a numpy array of values 0 (unsuitable) or 1 (suitable). It has the same size as the array of protected area categories.
- *weight* is a dictionary including the parameters related to the weighting:

- *power_density* is the power density of offshore wind projects in MW/m^2.
- *f_performance* is a number smaller than 1, taking into account all the other losses from the turbine generator until the AC substation.

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

Return param The updated dictionary param.

Return type dict

config.**onshore_wind_parameters**(*param*)

This function sets the parameters for onshore wind in the dictionary *windon* inside param:

- *resource* is a dictionary including the parameters related to the resource potential:
  - *res_correction* is either 1 (perform a redistribution of wind speed when increasing the resolution) or 0 (repeat the same value from the low resolution data). It is relevant for `correction_functions.generate_wind_correction`.
  - *topo_correction* is either 1 (perform a correction of wind speed based on the altitude and the Global Wind Atlas) or 0 (no correction based on altitude).
  - *topo_weight* is only relevant if *topo_correction* = 1. It defines how to weight the correction factors of each country. There are three options: `'none'` (all countries have the same weight), `'size'` (larger countries have a higher weight), or `'capacity'` (countries with a higher installed capacity according to IRENA have a higher weight).
- *technical* is a dictionary including the parameters related to the wind turbine:
  - *w_in* is the cut-in speed in m/s.
  - *w_r* is the rated wind speed in m/s.
  - *w_off* is the cut-off wind speed in m/s.
  - *P_r* is the rated power output in MW.
  - *hub_height* is the hub height in m.
- *mask* is a dictionary including the parameters related to the masking:
  - *slope* is the threshold slope in percent. Areas with a larger slope are excluded.
  - *lu_suitability* is a numpy array of values 0 (unsuitable) or 1 (suitable). It has the same size as the array of land use types.
  - *pa_suitability* is a numpy array of values 0 (unsuitable) or 1 (suitable). It has the same size as the array of protected area categories.
  - *buffer_pixel_amount* is an integer that defines the number of pixels making a buffer of exclusion around urban areas.
- *weight* is a dictionary including the parameters related to the weighting:
  - *lu_availability* is a numpy array of values between 0 (completely not available) and 1 (completely available). It has the same size as the array of land use types.
  - *pa_availability* is a numpy array of values between 0 (completely not available) and 1 (completely available). It has the same size as the array of protected area categories.
  - *power_density* is the power density of onshore wind projects in MW/m^2.
  - *f_performance* is a number smaller than 1, taking into account all the other losses from the turbine generator until the AC substation.

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

Return param The updated dictionary param.

Return type dict

config.**protected_areas_parameters**(*param*)

This function sets the parameters for protected areas in the dictionary *protected_areas* inside param:

- *type* is a numpy array of integers that associates a number to each protection type.
- *IUCN_Category* is an array of strings with names associated to each protection type (for your information).

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

Return param The updated dictionary param.

Return type dict

config.**time_series_parameters**(*param*)

This function determines the time series that will be created.

- *quantiles* is a list of floats between 100 and 0. Within each subregion, the FLH values will be sorted, and points with FLH values at a certain quantile will be later selected. The time series will be created for these points. The value 100 corresponds to the maximum, 50 to the median, and 0 to the minimum.
- *regression* is a dictionary of options for `regression.regression_coefficients`:
  - *solver* is the name of the solver for the regression.
  - *WindOn* is a dictionary containing a list of hub heights that will be considered in the regression, with a name tag for the list.
  - *WindOff* is a dictionary containing a list of hub heights that will be considered in the regression, with a name tag for the list.
  - *PV* is a dictionary containing a list of orientations that will be considered in the regression, with a name tag for the list.
  - *CSP* is a dictionary containing a list of settings that will be considered in the regression, with a name tag for the list.

  If all the available settings should be used, you can leave an empty list.
- *modes* is a dictionary that groups the quantiles and assigns names for each subgroup. You can define the groups as you wish. If you want to use all the quantiles in one group without splitting them in subgroups, you can write:

```
param["modes"] = {"all": param["quantiles"]}
```

- *combo* is a dictionary of options for `time_series.generate_stratified_timeseries`:
  - *WindOn* is a dictionary containing the different combinations of hub heights for which stratified time series should be generated, with a name tag for each list.
  - *WindOff* is a dictionary containing the different combinations of hub heights for which stratified time series should be generated, with a name tag for each list.
  - *PV* is a dictionary containing the different combinations of orientations for which stratified time series should be generated, with a name tag for each list.
  - *CSP* is a dictionary containing the different combinations of settings for which stratified time series should be generated, with a name tag for each list.

  If all the available settings should be used, you can leave an empty list.

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

Return param The updated dictionary param.

Return type dict

config.**weather_data_parameters**(*param*)

This function defines the coverage of the weather data *MERRA_coverage*, and how outliers should be corrected using *MERRA_correction*:

- *MERRA_coverage*: If you have downloaded the MERRA-2 data for the world, enter the name tag `'World'`. The code will later search for the data in the corresponding folder. It is possible to download the MERRA-2 just for the geographic scope of the analysis. In that case, enter another name tag (we recommend using the same one as the spatial scope).
- *MERRA_correction*: MERRA-2 contains some outliers, especially in the wind data. *MERRA_correction* sets the threshold of the relative distance between the yearly mean of the data point to the yearly mean of its neighbors.

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

Return param The updated dictionary param.

Return type dict

1.2.3 Paths

config.**emhires_input_paths**(*paths*, *tech*)

This function defines the path to the EMHIRES input file for each technology (only `'WindOn'`, `'WindOff'`, and `'PV'` are supported by EMHIRES).

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences.
- **tech** (*string*) – Name of the technology.

Return paths The updated dictionary paths.

Return type dict

config.**global_maps_input_paths**(*paths*, *param*)

This function defines the paths where the global maps are saved:

- *LU_global* for the land use raster
- *Topo_tiles* for the topography tiles (rasters)
- *Pop_global* for the global population raster
- *Bathym_global* for the bathymetry raster
- *Protected* for the shapefile of protected areas
- *GWA* for the country data retrieved from the Global Wind Atlas (missing the country code, which will be filled in a for-loop in :mod:correction_functions.calc_gwa_correction)
- *Countries* for the shapefiles of countries
- *EEZ_global* for the shapefile of exclusive economic zones of countries

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

Return paths The updated dictionary paths.

Return type dict

config.**irena_paths**(*paths*, *param*)

This function defines the paths for the IRENA inputs and outputs:

- *IRENA* is a csv file containing statistics for all countries and technologies for a specific *year*, created using a query tool of IRENA.

- *IRENA_dict* is a csv file to convert the code names of countries from the IRENA database to the database of the shapefile of countries.
- *IRENA_summary* is a csv file with a summary of renewable energy statistics for the countries 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.**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
- *SUB* for the raster of areas covered by subregions (both land and sea) within the scope
- *LU* for the land use raster within the scope
- *BATH* for the bathymetry raster within the scope
- *TOPO* for the topography raster within the scope
- *SLOPE* for the slope raster within the scope
- *PA* for the raster of protected areas within the scope
- *POP* for the population raster within the scope
- *BUFFER* for the raster of population buffer areas within the scope
- *CORR_GWA* for correction factors based on the Global Wind Atlas (mat file)
- *CORR_ON* for the onshore wind correction factors (raster)
- *CORR_OFF* for the offshore wind correction factors (raster)
- *AREA* for the area per pixel in m^2 (mat file)

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.
- *weather_data* is the output folder for the weather data of the spatial scope.
- *local_maps* is the output folder for the local maps of the spatial scope.
- *potential* is the output folder for the ressource and technical potential maps.
- *regional_analysis* is the output folder for the time series and the report of the subregions.
- *regression_in* is the folder where the regression parameters (FLH, fitting time series) are saved.
- *regression_out* is the output folder for the regression results.

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.

Return paths The updated dictionary paths.

Return type dict

config.**potential_output_paths**(*paths*, *param*, *tech*)

This function defines the paths of the files that will be saved in the folder for the potential outputs:

- *FLH* is the file with the full-load hours for all pixels within the scope (mat file).
- *mask* is the file with the suitable pixels within the scope (mat file).
- *FLH_mask* is the file with the full-load hours for the suitable pixels within the scope (mat file).
- *weight* is the power density for all the pixels in the scope (mat file).
- *FLH_weight* is the potential energy output for all the pixels in the scope (mat file).

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences.
- **tech** (*string*) – Name of the technology.

Return paths The updated dictionary paths.

Return type dict

config.**regional_analysis_output_paths**(*paths*, *param*, *tech*)

This function defines the paths of the files that will be saved in the folder for the regional analysis outputs:

- *Locations* is the shapefile of points that correspond to the selected quantiles in each subregion, for which the time series will be generated.
- *TS* is the csv file with the time series for all subregions and quantiles.
- *Region_Stats* is the csv file with the summary report for all subregions.
- *Sorted_FLH* is the mat file with the sorted samples of FLH for each subregion.
- *Regression_coefficients* is the path format for a csv files containing the regression coefficients found by the solver
- *Regression_TS* is the path format for a csv files with the regression resulting timeseries for the tech and settings

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*dict*) – Dictionary including the user preferences.
- **tech** (*string*) – Name of the technology.

Return paths The updated dictionary paths.

Return type dict

config.**regression_paths**(*paths*, *param*, *tech*)

This function defines the paths for the regression parameters:

- *FLH_regression* is a csv file containing FLH statistics for the subregions and the four technologies for a specific *year*, based on the previously created *IRENA_summary*.
- *TS_regression* is a csv file containing time series to be match for each subregion and technology, based on EMHIRES time series if available.

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

Return paths The updated dictionary paths.

Return type dict

config.**weather_input_folder**(*paths*, *param*)
This function defines the path *MERRA_IN* where the MERRA-2 data is saved. It depends on the coverage of the data and the year.

Parameters

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

Return paths The updated dictionary paths.

Return type dict

config.**weather_output_paths**(*paths*, *param*)
This function defines the paths to weather filesfor a specific *year*:

- *W50M* is the file for the wind speed at 50m in m/s.
- *CLEARNESS* is the file for the clearness index, e.g. the ratio between total ground horizontal radiation and total top-of-the-atmosphere horizontal radiation.
- *T2M* is the file for the temperature at 2m in Kelvin.

Parameters

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

Return paths The updated dictionary paths.

Return type dict

1.3 runme.py

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

```
import lib.correction_functions as cf
import lib.spatial_functions as sf
import lib.input_maps as im
import lib.potential as pl
from lib.log import logger
import initialization as ii
import lib.time_series as ts
import lib.regression as rg
import os
import psutil

if __name__ == "__main__":

    # logger.setLevel(logging.DEBUG)     # Comment out to get more information on the console

    if psutil.virtual_memory().available > 50*10**9:     # Check if memory size is large enough for multiprocessing
        multiprocessing = True
    else:
        multiprocessing = False
    logger.info('Multiprocessing: ' + str(multiprocessing))
```

(continues on next page)

(continued from previous page)

```
    configs = sorted(os.listdir('../configs'))
    for config in configs:       # Iterate over all config files for each country in folder 'configs'

        try:        # only interrupt current country run in case of failure
            logger.info('Started: ' + str(config))

            paths, param = ii.initialization(config)    # Initialize for each country with the corresponding config defined in folder 'configs'

            im.downloadGWA(paths, param)    # Download wind speed data from Global Wind Atlas
            im.generate_maps_for_scope(paths, param, multiprocessing)    # Generate input raster maps

            cf.generate_wind_correction(paths, param)

            for tech in param["technology"]:
                logger.info("Tech: " + tech)
                if tech == "Biomass":
                    im.generate_livestock(paths,param)
                    pl.generate_biomass_production(paths, param, tech)
                    pl.report_biomass_potentials(paths, param, tech)

                else:
                    # Generate potential maps and reports
                    pl.calculate_full_load_hours(paths, param, tech, multiprocessing)
                    pl.mask_potential_maps(paths, param, tech)
                    pl.weight_potential_maps(paths, param, tech)
                    pl.report_potentials(paths, param, tech)

                    # Generate time series
                    # ts.find_representative_locations(paths, param, tech)
                    # ts.generate_time_series_for_representative_locations(paths, param, tech)
                    # ts.generate_time_series_for_specific_locations(paths, param, tech)

            # for tech in param["technology"]:
                # logger.info("Tech: " + tech)

                # Generate regression coefficients for FLH and TS model matching
                # rg.get_regression_coefficients(paths, param, tech)

                # Generate times series for combinations of technologies and locations
                # ts.generate_time_series_for_regions(paths, param, tech)
        except Exception:
            logger.info("General exception noted!", exc_info=True)
```

1.4 Recommended input sources

For a list of GIS data sources, check this wikipedia article.

1.4.1 Weather data from MERRA-2

The most important inputs within this model are the weather time series. These are taken from the Modern-Era Retrospective Analysis for Research and Applications, version 2 (MERRA-2), which is the latest atmospheric reanalysis of the modern satellite era produced by NASA's Global Modeling and Assimilation Office (GMAO) [5]. The parameters taken from MERRA-2 are:

- Global Horizontal Irradiance (*GHI*): Downward shortwave radiation received by a surface horizontal to the ground (*SWGDN* in MERRA-2 nomenclature).
- Top of the Atmosphere Irradiance (*TOA*): Downward shortwave radiation at the top of the atmosphere (*SWTDN* in MERRA-2 nomenclature).
- Air temperature 2 meters above the ground (*T2M*).
- Northward wind velocity at 50 meters (*V50M*).
- Eastward wind velocity at 50 meters (*U50M*).

The *GHI* and *TOA* data are time-averaged hourly values given in W/m while *T2M* data are instantaneous values in Kelvin. *V50M* and *U50M* are instantaneous hourly values given in m/s.

The spatial arrangement of the data consists of a global horizontal grid structure with a resolution of 576 points in the longitudinal direction and 361 points in the latitudinal direction, resulting in pixels of 5/8° longitude and 1/2° latitude [1].

It is possible to download MERRA-2 dataset for the whole globe or just for a subset of your region of interest. Depending on the *MERRA_coverage* parameter in config.py, the script can accept both datasets. Note that downloading the coverage for the whole globe is easier but will require a significant amount of space on your drive (coverage of the whole globe requires 13.6 Gb for one year).

In both cases, please follow these instructions to download the MERRA-2 dataset:

1. In order to download MERRA-2 data using the FTP server, you first need to create an Eathdata account (more on that on their website).
2. Navigate to the link for the FTP sever here.
3. In *Data Product*, choose tavg1_2d_slv_NX and select the *Parameters* T2M, U50M, V50M to downaload the temperature and the wind speed datasets.
4. In *Spatial Search*, enter the coordinates of the bounding box around your region of interest or leave the default values for the whole globe. To avoid problems at the edge of the MERRA-2 cells, use the following set of formulas:

$$
\begin{aligned}
minLat &= \left\lfloor \frac{s + 0.25}{0.5} \right\rfloor \cdot 0.5 - \epsilon \\
maxLat &= \left\lceil \frac{n - 0.25}{0.5} \right\rceil \cdot 0.5 + \epsilon \\
minLon &= \left\lfloor \frac{w + 0.3125}{0.625} \right\rfloor \cdot 0.625 - \epsilon \\
maxLon &= \left\lceil \frac{e - 0.3125}{0.625} \right\rceil \cdot 0.625 + \epsilon
\end{aligned}
$$

   where *[s n w e]* are the southern, northern, western, and eastern bounds of the region of interest, which you can read from the shapefile properties in a GIS software, and
   $epsilon$ a small number.
5. In *Temporal Order Option*, choose the year(s) of interest.

6. Leave the other fields unchanged (no time subsets, no regridding, and NetCDF4 for the output file format).
7. Repeat the steps 4-6 for the *Data Product* tavg1_2d_rad_Nx, for which you select the *Parameters* SWGDN and SWTDN, the surface incoming shortwave flux and the top of the atmosphere incoming shortwave flux.
8. Follow the instructions in the website to actually download the NetCDF4 files from the urls listed in the text files you obtained.

If you follow these steps to download the data for the year 2015, you will obtain 730 NetCDF files, one for each day of the year and for each data product.

1.4.2 Raster of Mean Wind Speed

High resolution (250m) country-wise rasters of mean wind speed from Global wind atlas website will be automatically downloaded by the tool.

1.4.3 Raster of land use

Another important input for this model is the land use type. A land use map is useful in the sense that other parameters can be associated with different landuse types, namely:

- Urban areas
- Ross coefficients
- Hellmann coefficients
- Albedo
- Suitability
- Installation cost
- etc.

For each land use type, we can assign a value for these parameters which affect the calculations for solar power and wind speed correction. The global land use raster for which `lib.input_maps.generate_landuse` has been written can be downloaded from the ESA CCI website. However, this new version requires additional data processing. The spatial resolution of the land use raster downloaded is 300m, but the resolution used in the model is 250m. So the landuse raster should be resampled in a GIS software. QGIS can be used easily for doing this.

1.4.4 Shapefile of the region of interest

The strength of the tool relies on its versatility, since it can be used for any user-defined regions provided in a shapefile. If you are interested in administrative divisions, you may consider downloading the shapefiles from the website of the Global Administration Divisions (GADM). You can also create your own shapefiles using a GIS software.

Warning: In any case, you need to have at least one attribute called *NAME_SHORT* containing a string (array of characters) designating each sub-region.

1.4.5 Shapefile of countries

A shapefile of all the countries of the world is also needed. It can be downloaded again from GADM. The attribute "GID_0" contains the ISO 3166-1 Alpha-3 codes of the countries, and is currently hard coded in the script.

Warning: If you want to use another source or other code names, you need to edit the name of the attribute "GID_0" and the dictionary *dict_countries.csv*.

1.4.6 Shapefile of Exclusive Economic Zones (EEZ)

A shapefile of the maritime boundaries of all countries is available at the website of the Flanders Marine Institute (VLIZ). It is used to identify offshore areas.

1.4.7 Shapefile of Internal Waters

A shapefile of the internal waters boundaries of all countries is available at the website of the Flanders Marine Institute (VLIZ). It is used to identify offshore areas.

1.4.8 Raster of topography / elevation data

A high resolution raster (15 arcsec = 1/240° longitude and 1/240° latitude) made of 24 tiles can be downloaded from viewfinder panoramas. Multiple files will be downloaded from this source. They can all be merged and resampled to the resolution of the model (250m) using QGIS, similar to the landuse raster.

1.4.9 Raster of bathymetry

A high resolution raster (60 arcsec) of bathymetry can be downloaded from the website of the National Oceanic and Atmospheric Administration (NOAA). The one used in the database is ETOPO1 Ice Surface, cell-registered.

1.4.10 Shapefile of protected areas

Any database for protected areas can be used with this tool, in particular the World Database on Protected Areas published by the International Union for Conservation of Nature (IUCN). The shapefile has many attributes, but only one is used in the tool: "IUCN_CAT". If another database is used, an equivalent attribute with the different categories of the protection has to be used and `config.py` has to be updated accordingly.

1.4.11 Airports Coordinates

List of airports around the world can be downloaded as a csv file from open data (openflights).

1.4.12 Shapefiles from OSM data

Open Street Map data can be downloaded as shapefiles from geofabrik. The shapefiles for roads, railways and landuse are used in this model. These shapefiles have many attributes, but only one is used in the tool: "fclass". For roads shapefile, the "fclass" types "Motoways, motorways_link, primary, primary-link, secondary, secondary-link, trunk, trunk-link" are filtered prior to using the model. For railways shapefile, no filtering is necessary. For landuse shapefile, the "fclass" types "commercial, industrial, quarry, military, park, recreation_ground" are filtered prior to using the model. If another exclusion criteria is used, `config.py` has to be updated accordingly.

1.4.13 Raster of Settlement Footprint

A high resolution raster (0.32 arcsec or 10m) of World settlement footprint can be downloaded from open source. The downloaded multiple files need to be merged and resampled to the desired resolution (250m) of the model prior to the run.

1.4.14 Shapefile of HydroLakes

Any database for Lakes can be used with this tool, in particular from HydroSheds. No preprocessing is necessary for this dataset.

1.4.15 Shapefile of HydroRivers

Any database for Rivers can be used with this tool, in particular from HydroSheds. No preprocessing is necessary for this dataset.

1.4.16 Data of Crop Production

Annual crop production data of all countries in the world can be downloaded from the website of Food and Agriculture Organization of United States (FAOSTAT). While downloading, the latest year and "Production Quantity" should be selected as filters.

1.4.17 Data of Forestry Production

Annual forestry production data of all countries in the world can be downloaded from the website of Food and Agriculture Organization of United States (FAOSTAT). While downloading, the latest year and "Production Quantity" should be selected as filters.

1.4.18 Shapefile of Livestock density

Any dataset for Livestock density can be used with this tool, in particular the rasters created from the data of Food and Agriculture Organization of United States for various animals (FAO GLW3). These files are available at high resolution (5 arc-minutes). The model can read these high resolution rasters and resample them to the resolution of the model.

1.5 Recommended workflow

The script is designed to be modular and split into four main modules: `lib.input_maps`, `lib.potential`, `lib.time_series`, and `lib.regression`.

> **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. *Input raster maps*
2. *Potential maps and reports*
3. *Time series for quantiles and user-defined locations*
4. *Regression*

5. *Stratified time series*

The use cases associated with each module with examples of their outputs 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 Input raster maps

The `lib.input_maps` module is used to generate data (mostly raster maps, but also arrays in MAT files) for the spatial scope defined by the user. These data sets include:

- Weather data
- Land and sea masking
- Bathymetry
- Topography
- Slope
- Area
- Land use and buffer masking
- Protected areas and their buffer masking
- Boarder Buffer masking
- Airports and buffer masking
- Roads Buffer masking
- Railway lines Buffer masking
- OSM defined areas like mining, military zones Buffer masking
- Settlement regions Buffer masking
- HydroLakes Buffer masking
- HydroRivers Buffer masking
- Livestock density

All these maps are needed before the potential or time series modules can be used for a specific spatial scope.

1.5.2 Potential maps and reports

The `lib.potential` module serves to generate potential raster maps for all four technologies supported by the script. This module generates a Full-Load Hour (FLH) raster map, masking and masked rasters for unsuitable and protected areas, and a weighting and weighted raster used for energy and power potential calculations. It also generates a CSV report containing metrics for each subregion:

- Available number of pixels, before and after masking
- Available area in in km^2
- FLH mean, median, max, min values, before and after masking
- FLH standard deviation after masking
- Power Potential in GW, before and after weighting
- Energy Potential in TWh in total, after weighting, and after masking and weighting
- Sorted sample of FLH values for each region

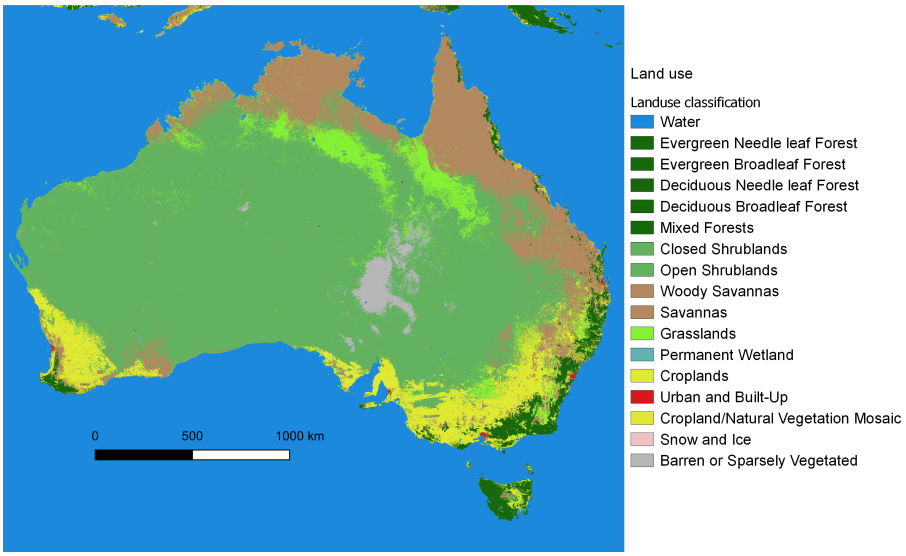

Fig. 1: Land Use Raster Map - Australia

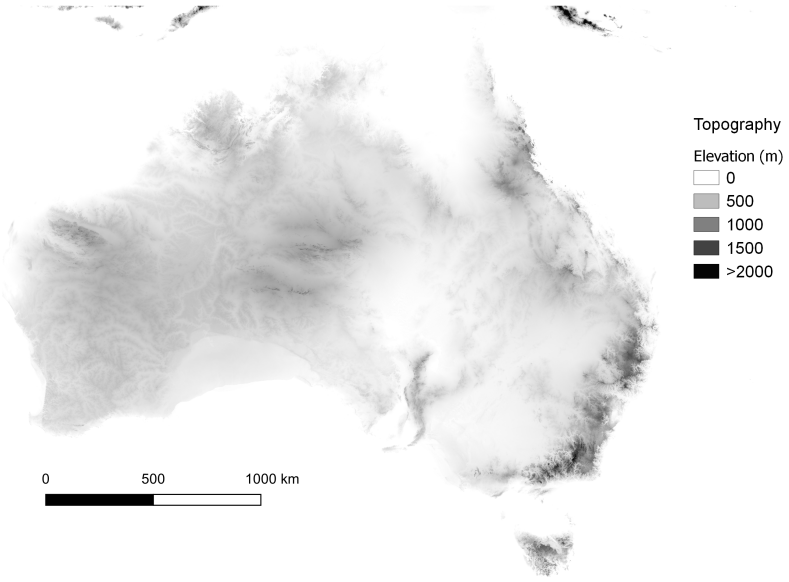

Fig. 2: Topography Raster Map - Australia

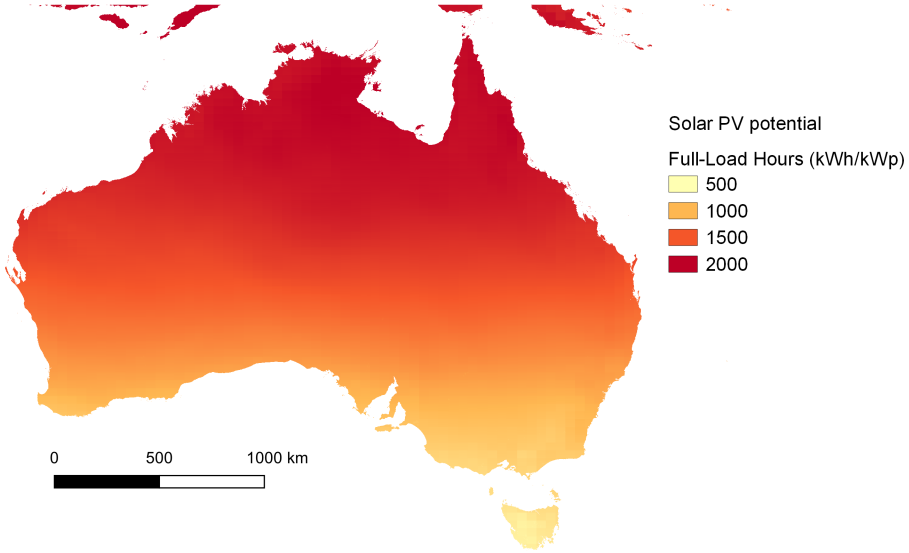

Fig. 3: FLH of solar PV - Australia

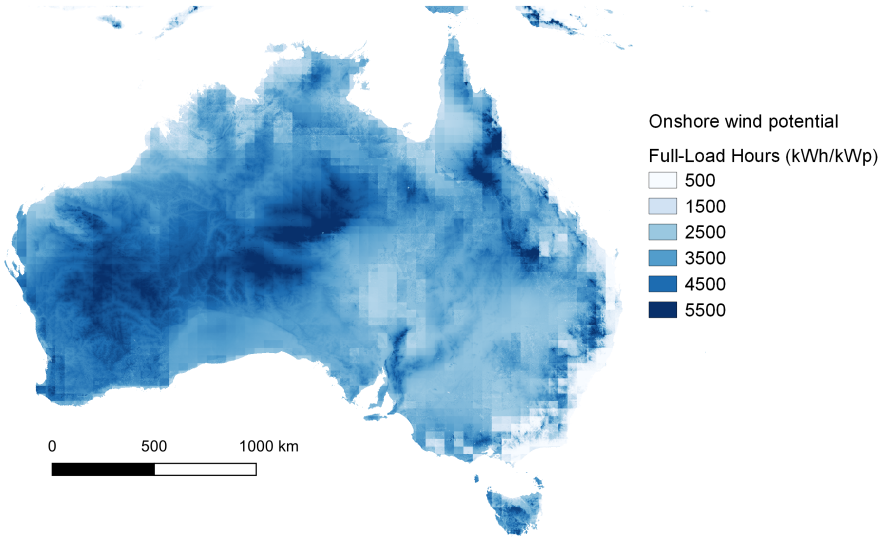

Fig. 4: FLH of onshore wind - Australia

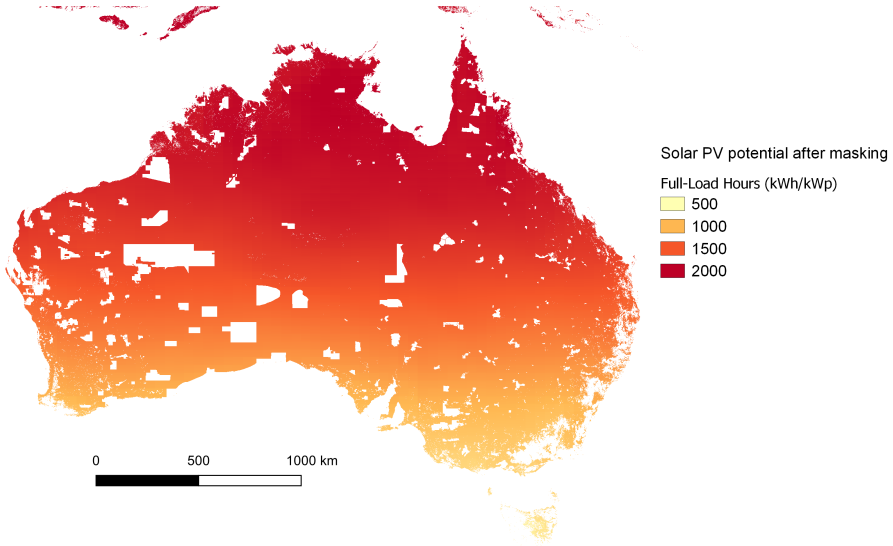

Fig. 5: FLH of solar PV after masking - Australia

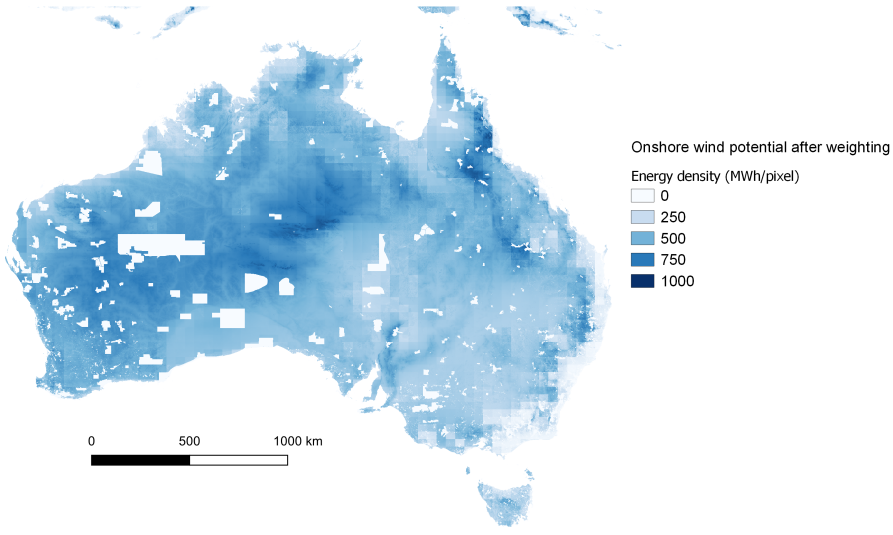

Fig. 6: Energy output of onshore wind after weighting - Australia

Sample of potential report:

Region	Available area (km^2)
Region A	4315.7
Region B	2128.3
Region C	561.3
Region D	100953.1
Region E	10.2
Region F	2829.8

FLH mean	FLH max	FLH min	Masked FLH mean	Masked FLH max	Masked FLH min	Power potential (G
1638.4	1686.3	1578.0	1644.3	1686.3	1589.7	6.5
1682.9	1699.7	1601.6	1684.2	1695.0	1613.7	1.4
1849.7	1853.4	1833.8	1849.6	1853.3	1840.8	0.9
2017.6	2090.5	1986.8	2018.0	2086.1	1986.8	183.7
1856.8	1857.1	1856.5	1856.8	1857.1	1856.5	0.0
1729.5	1772.2	1659.1	1731.4	1772.2	1659.1	4.8

1.5.3 Time series for quantiles and user-defined locations

The `lib.time_series` module allows to generate time series for quantiles as well as user-defined locations based on the FLH raster maps generated in the previously mentioned module. It is therefore important for the FLH raster maps to be generated first, in order to locate the quantiles. However, generating time series for user-defined locations does not require the potential maps to be generated beforehand.

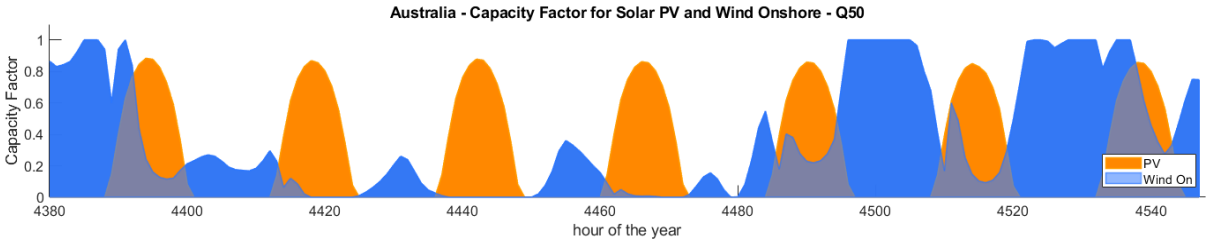

Fig. 7: Wind Onshore and Solar PV capacity factor time series for quantile 50 - Australia

1.5.4 Regression

Once a set of time series for different settings (hub heights for wind technologies, orientations for solar PV) is generated, the `lib.regression` module allows the user to find a combination of settings and quantiles in order to match a known FLH value and a given (typical) time series. The output is a set of regression coefficients that should be multiplied with the time series.

1.5.5 Stratified time series

Part of the `lib.time_series` module, the `lib.time_series.generate_time_series_for_regions` function reads the regression coefficients and the generated time series, and combines them into user-defined *modes* (combinations of quantiles) and *combos* (combinations hub height or orientations settings).

Example: - Graphic of Modes and Combos

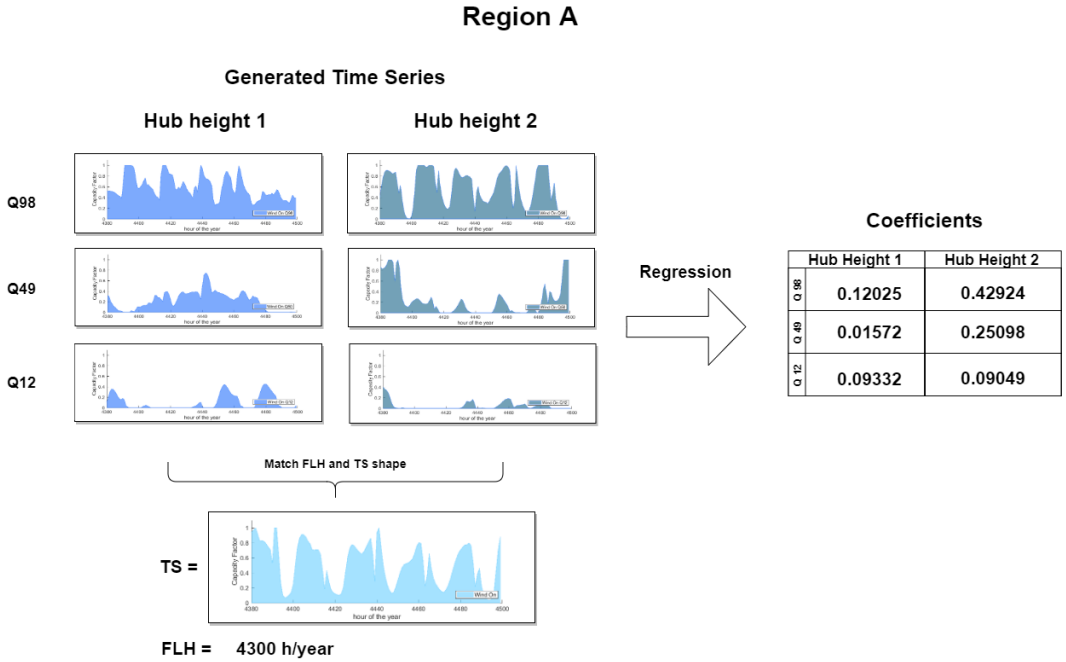

Fig. 8: Regression Coefficients - Process example Region A

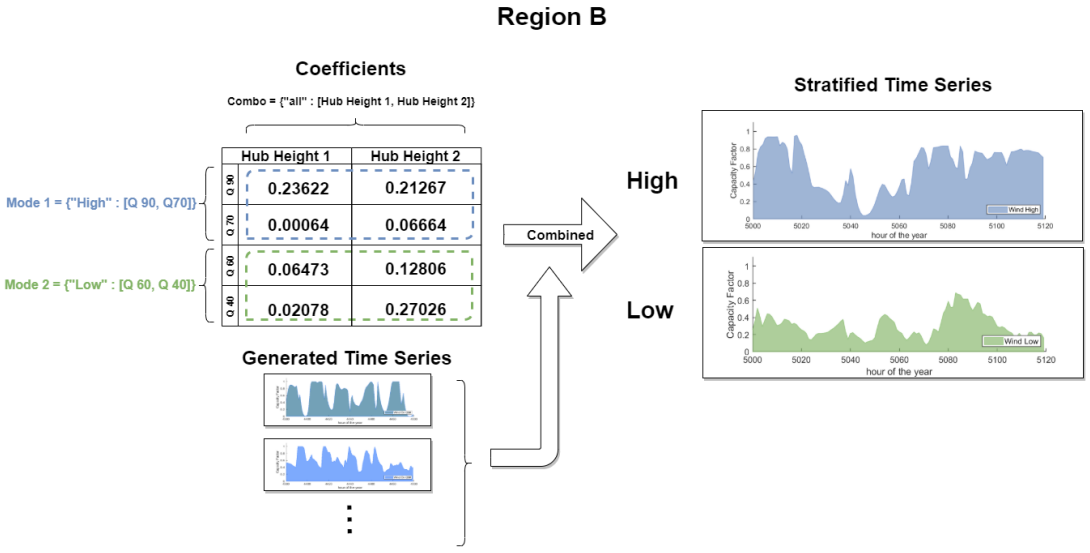

Fig. 9: Stratified Time Series - Process example Region A

Chapter 2

Theory

2.1 Solar

2.1.1 Solar Angles

While the output power of the Sun is usually considered as a constant, the amount of power arriving at the Earth's surface varies according to the time, location, weather, and relative position of the Earth with respect to the Sun. Besides, the available data needed for a solar power calculation is usually given for a horizontal surface and most of the PV systems are placed in a tilted position. Therefore, it is necessary to calculate a set of parameters describing the Sun's relative position with respect to the position of the system being irradiated. These parameters are calculated for points located at the center of every pixel (with high resolution) within the extension under analysis and for every hour of the year.

Declination Angle δ

This angle varies during the year due to the tilt of the Earth's axis, which is 23.45° tilted, so the declination ranges between -23.4° and 23.45° through the year. The declination could be interpreted as the latitude where the Sun's rays perpendicularly strike the Earth's surface at solar noon. This value is the same for all the locations within the globe for a given day and is calculated as follows [11]:

$$\delta = \arcsin\left(0.3978 \sin\left(\frac{2\pi N}{365.25} - 1.4 + 0.0355 \sin\left(\frac{2\pi N}{365.25} - 0.0489\right)\right)\right)$$

where N is the day of the year.

Solar Time

The time is important to define the position of the Sun in the sky. However, it is easier to use the time if it is converted into solar time. To do so, a few corrections are needed. The equation of time is an empirical equation which corrects the error caused by the axial tilt of the Earth and the eccentricity of its orbit [11]:

$$EOT = -0.128 \sin\left(\frac{360}{365.25} N - 2.8\right) - 0.165 \sin\left(\frac{720}{365.25} N + 19.7\right)$$

When the time is given in GMT, as it is for this model, it is also necessary to take into account the longitude of the location, hence the time correction:

$$TC = EOT + longitude/15$$

where the factor 15 accounts for the geographical span of each time zone (15° of longitude). With this correction, the local solar time is calculated as:

$$LST = T_{GMT} + TC$$

where LST is the local solar time. An even more apporpriate time measure for solar calculations is the hour angle ω This converts hours into degrees which indicate how the Sun moves in the sky relatively to the Earth, where the solar noon is 0°, the anlges after the noon positive, and before the noon negative.

$$\omega = 15(LST - 12)$$

Another important quantity is the duration of the day, which is delimited by the sunrise and the sunset. The sunrise and sunset have the same value, however, the sunrise is considered negative and the sunset positive. They depend on the day of the year and the location on the Earth (denoted by the declination and the latitude ϕ respectively) and they are calculated for a horizontal surface as follows [7]:

$$\omega_s = \arccos(-\tan\phi\tan\delta)$$

Due to self-shading, a tilted plane might be exposed to different sunrise and sunset's values. Also, if the plane is not facing the equator, the sunrise and sunset angles will be numerically different for such surfaces. The following equations consider this orientation changes for the sunrise and sunset values of a tilted plane [7].

$$a = \frac{\cos\phi}{\tan\beta} + \sin\phi$$

$$b = \tan\delta\cos\phi\cos\gamma - \frac{\sin\phi}{\tan\beta}$$

$$\omega_s' = \cos\left[\frac{ab \pm \sin\gamma\sqrt{a^2 - b^2 + \sin^2\gamma}}{a^2 + \sin^2\gamma}\right]$$

where γ is the azimuthal orientation of the panel and β is the tilt of the panel (for this model, chosen as the optimal tilt according to the latitude). These equations might give higher values than the real sunrise and sunset values. This would imply that the Sun rises over the tilted plane before it has risen over the horizon or that when the Sun sets, there is still light striking the plane. As this is wrong, the sunrise and sunset values for a horizontal plane must be compared with the values for a tilted plane and the lower values (for both sunrise and sunset) must be selected.

$$\omega_0 = \min(\omega_s, \omega_s')$$

Incidence Angle θ

As stated before, PV panels are not normally parallel to the Earth's surface, so it is necessary to calculate the incidence angle of the Sun's rays striking the surface of the panel. Nevertheless, a set of angles must be calculated in order to calculate the incidence angle.

The elevation angle α or altitude angle measures the angular distance between the Sun and the horizon. It ranges from 0° at the sunrise to 90° at the noon (the value at the noon varies depending on the day of the year) [12].

$$\alpha = \arcsin[\sin\delta\sin\phi + \cos\delta\cos\phi\cos\omega]$$

The azimuth angle A_z is an angular measurement of the horizontal position of the Sun. It could be seen as a compass direction with 0° to the North and 180° to the South. The range of values of the azimuth angle varies over the year, going from 90° at the sunrise to 270° at the sunset during the equinoxes. The equation for the azimuth depends on the time of the day. For the solar morning, it is [12]:

$$Az_{am} = \arccos\left(\frac{\sin\delta\cos\phi - \cos\delta\sin\phi\cos\omega}{\cos\alpha}\right)$$

and for the afternoon:

$$Az_{pm} = 360 - Az_{am}$$

With the already calculated angles, it is possible to calculate the incidence angle, which is the angle between the surface's normal and the Sun's beam radiation [10]:

$$\begin{aligned}\theta_i = \arccos(&\sin\delta\sin\phi\cos\beta\\ &- \sin\delta\cos\phi\sin\beta\cos\gamma\\ &+ \cos\delta\cos\phi\cos\beta\cos\omega\\ &+ \cos\delta\sin\phi\sin\beta\cos\gamma\cos\omega\\ &+ \cos\delta\sin\beta\sin\gamma\sin\omega)\end{aligned}$$

Tracking

When one-axis tracking is active, the tilt angle β and the azimuthal orientation γ of the panel change constantly as the panel follows the sun. In this model a tilted on-axis tracking with east-west tracking is considered. The rotation of the plane around the axis is deffned by the rotation angle R, it is calculated in order to achieve the smallest incidence angle for the plane by the following equations [13]:

$$X = \frac{-\cos\alpha \sin(A_z - \gamma_a)}{-\cos\alpha\cos(A_z - \gamma_a)\sin\beta_a + \sin\alpha\cos\beta_a}$$

$$\Psi = \begin{cases} 0, & \text{if } X = 0, \text{ or if } X > 0 \wedge (A_z - \gamma_a) > 0, \text{ or if } X < 0 \wedge (A_z - \gamma_a) < 0 \\ 180, & \text{if } X < 0 \wedge (A_z - \gamma_a) > 0 \\ -180, & \text{if } X > 0 \wedge (A_z - \gamma_a) < 0 \end{cases}$$

for the previous equations, β_a and γ_a are considered as the tilt and azimuthal orientation of the tracking axis respectively. The variable Ψ places R in the correct trigonometric quadrant. For the selection of Ψ, the difference $(A_z - \gamma_a)$ must be considered as the angular displacement with the result within the range of -180°to 180°. Once the rotation angle is calculated, the tilt and azimuthal orientation of the panel are calculated as follows:

$$\beta = \arccos(\cos R \cos\beta_a)$$

$$\gamma = \begin{cases} \gamma_a + \arcsin\left(\frac{\sin R}{\sin\beta}\right), & \text{for } \beta_a \neq 0, -90 \leq R \leq 90 \\ \gamma_a - 180 - \arcsin\left(\frac{\sin R}{\sin\beta}\right), & \text{for } -180 \leq R < -90 \\ \gamma_a + 180 - \arcsin\left(\frac{\sin R}{\sin\beta}\right), & \text{for } 90 < R \leq -90 \end{cases}$$

Then the incidence angle is calculated using the new β and γ angles. For two-axis tracking the beta angle is considered as the complementary angle to the altitude angle while the azimuthal orientation angle γ is considered as $A_z - 180$.

2.1.2 Solar Power

To calculate the solar power we must start with the solar constant. However, the irradiance striking the top of the Earth's atmosphere (TOA) varies over the year. This is due to the eccentricity of the Earth's orbit and its tilted axis. The TOA is calculated according to the following equation [10]:

$$TOA = G_{sc}\left[1 + 0.03344\cos\left(\frac{2\pi N}{365.25} - 0.048869\right)\right]\sin\alpha$$

where G_{sc} is the solar constant (1367 W/m2) and N is the day of the year. This equation escalates the solar constant by multiplying it with a factor related to the eccentricity of the Earth's orbit and the sinus of the altitude angle of the Sun to finally get the extraterrestrial horizontal radiation.

To calculate the amount of horizontal radiation at the surface of the Earth, the attenuation caused by the atmosphere must be considered. A way to measure this attenuation is by using the clearness index k_t. This value is the ratio between the extraterrestrial horizontal radiation and the radiation striking the Earth's surface:

$$k_t = \frac{GHI_{M2}}{TOA_{M2}}$$

where GHI_{M2} and TOA_{M2} are the global horizontal irradiance and the top of the atmosphere radiation extracted from MERRA-2 data. Furthermore, the GHI is made-up by diffuse and beam radiation:

$$GHI = G_b + G_d$$

where G_b is the beam radiation, which is the solar radiation that travels directly to the Earth's surface without any scattering in the atmosphere, and G_d stands forfor the diffuse radiation, the radiation that comes to a surface from all directions as its trajectory is changed by the atmosphere. These two components have different contributions to the total irradiance on a tilted surface, so it is necessary to distinguish between them. This can be done using

the correlation of Erbs et al [4], which calculates the ratio R of the beam and diffuse radiation as a function of the clearness index.

$$R = \begin{cases} 1 - 0.09k_t, & \text{for } k_t \leq 0.22 \\ 0.9511 - 0.1604k_t + 4.388k_t^2 - 16.638k_t^3 + 12.336k_t^4, & \text{for } 0.22 > k_t \leq 0.8 \\ 0.165, & \text{for } k_t > 0.8 \end{cases}$$

Furthermore, diffuse radiation could be divided into more components. The HDKR model [3][10], developed by Hay, Davies, Klucher, and Reindl in 1979 assumes isotropic diffuse radiation, which means that the diffuse radiation is uniformly distributed across the sky. However, it also considers a higher radiation intensity around the Sun, the circumsolar diffuse radiation, and a horizontal brightening correction. To use the HDKR model, some factors must be defined first [10]. The ratio of incident beam to horizontal beam:

$$R_b = \frac{\cos\theta_i}{\sin\alpha}$$

The anisotropy index for forward scattering circumsolar diffuse irradiance:

$$A_i = (1 - R)k_t$$

The modulating factor for horizontal brightening correction:

$$f = \sqrt{1 - R}$$

Then the total radiation incident on the surface is calculated with the next equations:

$$GHI = k_t TOA$$

$$G_T = GHI\Big[(1 - R + RA_i)R_b + R(1 - A_i)\Big(\frac{1 + \cos\beta}{2}\Big)\Big(1 + f\sin^3\frac{\beta}{2}\Big) + \rho_g\Big(\frac{1 - \cos\beta}{2}\Big)\Big]$$

Where ρ_g is the ground reflectance or albedo and it is related to the land use type of the location under analysis. The first term of this equation corresponds to the beam and circumsolar diffuse radiation, the second to the isotropic and horizon brightening radiation, and the last one to the incident ground-reflected radiation.

2.1.3 PV

Temperature losses

In a PV panel, not all the radiation absorbed is converted into current. Some of this radiation is dissipated into heat. Solar cells, like all other semiconductors, are sensitive to temperature. An increase of temperature results in a reduction of the band gap of the solar cell which is translated into a reduction of the open circuit voltage. The overall effect is a reduction of the power output of the PV system. To calculate the power loss of a solar cell it is necessary to know its temperature. This can be expressed as a function of the incident radiation and the ambient temperature [9]:

$$T_{cell} = T_{amb} + kG_T$$

where T_{amb} is the ambient temperature and k is the Ross coefficient, which depends on the characteristics related to the module and its environment. It is defined based on the land use type of the region where the panel is located. With the temperature of the panel, the fraction of the irradiated power which is lost can be calculated as:

$$Loss_T = (T_{cell} - T_r)T_k$$

where T_r is the rated temperature of the module according to standard test conditions and T_k is the heat loss coefficient. Both values are usually given on the data sheets of the PV modules.

PV Capacity Factor Calculation

It is the ratio of the actual power output to the theoretical maximum output which is normally considered as $1000W/m^2$. The temperature loss is also considered for this calculation:

$$CF_{PV} = \frac{G_T(1 - Loss_T)}{1000}$$

Ground coverage ratio (GCR)

It is also important to consider the area lost due to the space between the modules or due to the modules shading adjacent modules. This is done with the GCR which is the ratio of the module area to the total ground area.

$$GCR = \frac{1}{\cos\beta + |\cos A_z| \cdot \left(\frac{\sin\beta}{\tan\alpha}\right)}$$

2.1.4 CSP

For its popularity and long development history, the parabolic trough technology was chosen to model Concentrated Solar power.

Convection Losses

The receiver of parabolic troughs are kept in a vacuum glass tube to prevent convection as much as possible. Radiative heat losses are still present and ultimatly results in convective losses between the glass tube and the air. These heat losses are increased when wind is blowing around the receiver. The typical heat losses for a receiver can be estimated through the following empirical equation [3]:

$$Q_{Loss} = A_r(U_{L_{cst}} + U_{L_{Wind}} \cdot V_{Wind}^{0.6})(T_i - T_a)$$

where A_r is the outer area of the receiver, $U_{L_{cst}}$ correspond to a loss coefficient at zero wind speed, $U_{L_{Wind}}$ is a loss coefficient dependent on the wind speed V_{Wind}, T_i is the average heat transfer fluid temperature, and T_a is the ambient temperature.

Typical values for the $U_{L_{cst}}$ and $U_{L_{Wind}}$ are 1.06 kW/m^2K and 1.19 $kW/(m^2K(m/s)^{0.6})$ respectively

Flow Losses

Flow loss coefficient or heat removal factor F_r is the ratio between the actual heat transfer to the maximum heat transfer possible between the receiver and the heat transfer fluid (HTF). These losses result from the difference between the temperature of the receiver and the temperature of the HTF and are dependent on the heat capacity and the flow rate of the HTF. A typical value for parabolic troughs is 95%.

CSP Capacity Factor Calculation

The capacity factor of a solar field is the ratio of the actual useful heat collected to the theoretical maximum heat output of 1000 W/m^2. It is given by the formula:

$$CF_{csp} = \frac{F_r(S - Q_{Loss})}{1000}$$

Where S is the component of the DNI captured by the collector at an angle (based on one axis traking), Q_{Loss} is the heat convection losses, and F_r is the heat removal factor.

2.2 Wind

2.2.1 Wind Speed

In order to use a single value for the following wind power calculations, a norm is calculated as if both variables were vectors, so the absolute velocity is:

$$W_{50M} = \sqrt{V50M^2 + U50M^2}$$

However, the wind velocities extracted from MERRA-2 are given for a height of 50 meters, which does not correspond to the hub height of the wind turbines; therefore, they must be extrapolated.

2.2.2 Wind Shear

While the wind is hardly affected by the Earth's surface at a height of about one kilometer, at lower heights in the atmosphere the friction of the Earth's surface reduces the speed of the wind [2]. One of the most common expressions describing this phenomenon is the Hellmann exponential law, which correlates the wind speed at two different heights [6].

$$v = v_0 \left(\frac{H}{H_0} \right)^{\alpha}$$

Where v is the wind speed at a height H, v_0 is the wind speed at a height H_0 and α is the Hellmann coefficient, which is a function of the topography and air stability at a specific location.

2.2.3 Wind Power

The wind turbines convert the kinetic energy of the air into torque. The power that a turbine can extract from the wind is described by the following expression [8]:

$$P = \frac{1}{2} \rho A v^3 C_p$$

where ρ is the density of the air, v is the speed of the wind and C_p is the power cofficient. As it is shown in the previous equation, the energy in the wind varies proportionally to the cube of the wind's speed. Therefore, the power output of wind turbines is normally described with cubic power curves. However, there are some regions within those curves which have special considerations.

Cut-in wind speed

The wind turbines start running at a wind speed between 3 and 5 m/s to promote torque and acceleration. Therefore, there is no power generation before this velocity.

Rated Wind Speed

The power output of a wind turbine rises with the wind speed until the power output reaches a limit defined by the characteristics of the electric generator. Beyond this wind speed, the design and the controllers of the wind turbine limit the power output so this does not increase with further increases of wind speed.

Cut-out wind speed

The wind turbines are programmed to stop at wind velocities above their rated maximum wind speed(usually around 25 m/s) to avoid damage to the turbine and its surroundings, so there is no power generation after this velocity.

Wind Onshore and Offshore Capacity factor

Finally, the capacity factors, which are the ratios of the actual power output to the theoretical maximum output (rated wind speed), are calculated according to the previously presented regions:

$$CF = \begin{cases} \frac{W_{hub}^3 - W_{in}^3}{W_r^3 - W_{in}^3}, & \text{for } W_{in} < W_{hub} < W_r \\ 1, & \text{for } W_r \leq W_{hub} \leq W_{out} \\ 0, & \text{for } W_{hub} < W_{in} | W_{hub} > W_{out} \end{cases}$$

where W_{in} is the cut-out wind speed, and W_r is the rated wind speed. The area is not included in the previous equations as it does not change in both generation states (actual and theoretical maximum power). While the density could vary for both states, the overall impact of a change in density is negligible compared to the wind speed and therefore is not included in the calculation.

Chapter 3

Implementation

You can run the code by typing:

```
$ python runme.py
```

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

3.1 initialization.py

3.2 input_maps.py

lib.input_maps.**downloadGWA**(*paths*, *param*)
This function downloads wind speed data from Global Wind Atlas (www.globalwindatlas.info) if it does not already exist

Parameters

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

Returns The wind data is saved directly in the desired paths.

Return type None

lib.input_maps.**generate_area**(*paths*, *param*)
This function retreives the coordinates of the spatial scope and computes the pixel area gradient of the corresponding raster.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the path to the output file.
- **param** (*dict*) – Dictionary of dictionaries containing spatial scope coordinates and desired resolution.

Returns The mat file for AREA is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_area_offshore**(*paths*, *param*)
This function retreives the coordinates of the spatial scope and computes the pixel area gradient of the corresponding raster.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the path to the output file.

- **param** (*dict*) – Dictionary of dictionaries containing spatial scope coordinates and desired resolution.

Returns The mat file for AREA is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_array_coordinates**(*paths*, *param*, *W50M*)
ToDo: All of this docstring This function reads the daily NetCDF data (from MERRA-2) for SWGDN, SWTDN, T2M, U50m, and V50m, and saves them in matrices with yearly time series with low spatial resolution. Depending on the *MERRA_correction* parameter this function will also call clean_weather_data() to remove data outliers. This function has to be run only once.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the MERRA-2 input files *MERRA_IN*, and to the desired output locations for *T2M*, *W50M* and *CLEARNESS*.
- **param** (*dict*) – Dictionary including the year, the spatial scope, and the MERRA_correction parameter.

Returns The files T2M.mat, W50M.mat, and CLEARNESS.mat are saved directly in the defined paths, along with their metadata in JSON files.

Return type None

lib.input_maps.**generate_bathymetry**(*paths*, *param*)
This function reads the global map of bathymetry, resizes it, and creates a raster out of it for the desired scope. The values are in meter (negative in the sea).

Parameters

- **paths** (*dict*) – Dictionary including the paths to the global bathymetry raster *Bathym_global* and to the output path *BATH*.
- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and the georeference dictionary.

Returns The tif file for *BATH* is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_land**(*paths*, *param*)
This function reads the shapefile of the subregions within the scope, and creates a raster out of it.

Parameters

- **paths** (*dict*) – Dictionary including the paths *SUB*, *LAND*, *EEZ*.
- **param** (*dict*) – Dictionary including the geodataframe of the shapefile, the number of features, the coordinates of the bounding box of the spatial scope, and the number of rows and columns.

Returns The tif file for *SUB* is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_landuse**(*paths*, *param*)
This function reads the global map of land use, and creates a raster out of it for the desired scope. There are 17 discrete possible values from 0 to 16, corresponding to different land use classes. See `config.py` for more information on the land use map.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the global land use raster *LU_global* and to the output path *LU*.

- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and the georeference dictionary.

Returns The tif file for *LU* is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_livestock**(*paths*, *param*)
This function reads the global maps of each livestock density, resizes it, and creates a raster out of it for the desired scope. The values are in number of animals per sq.km.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the global livestock rasters *LS_global* and to the output path *LS*.
- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and the georeference dictionary.

Returns The tif files for *LS* is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_maps_for_scope**(*paths*, *param*, *multiprocessing*)
This function calls the individual functions that generate the maps for the geographic scope.

Parameters

- **paths** (*dict*) – Dictionary including the paths.
- **param** (*bool*) – Dictionary including the user preferences.
- **multiprocessing** – Determines if multiprocessing is applied.

Returns The maps are saved directly in the desired paths.

Return type None

lib.input_maps.**generate_osm_areas**(*paths*, *param*)
This function reads the osm land use shapefile, identifies several areas, and excludes pixels around them based on a user-defined buffers *buffer_pixel_amount*. It creates a masking raster of boolean values (0 or 1) for the scope. Zero means the pixel is excluded, one means it is suitable. The function is useful in case there is a policy to exclude renewable energy projects next to certain type of areas.

Parameters

- **paths** (*dict*) – Dictionary including the path to the osm land-use shapefile, and to the output path BUFFER.
- **param** (*dict*) – Dictionary including the user-defined buffers (buffer_pixel_amount) and the georeference dictionary.

Returns The tif file for BUFFER is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_protected_areas**(*paths*, *param*)
This function reads the shapefile of the globally protected areas, adds an attribute whose values are based on the dictionary of conversion (protected_areas) to identify the protection category, then converts the shapefile into a raster for the scope. The values are integers from 0 to 10.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the shapefile of the globally protected areas, to the land raster of the scope, and to the output path PA.
- **param** (*dict*) – Dictionary including the dictionary of conversion of protection categories (protected_areas).

Returns The tif file for PA is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_protected_areas_offshore**(*paths*, *param*)
This function reads the shapefile of the globally protected areas, adds an attribute whose values are based on the dictionary of conversion (protected_areas) to identify the protection category, then converts the shapefile into a raster for the scope. The values are integers from 0 to 10.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the shapefile of the globally protected areas, to the land raster of the scope, and to the output path PA.
- **param** (*dict*) – Dictionary including the dictionary of conversion of protection categories (protected_areas).

Returns The tif file for PA is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_sea**(*paths*, *param*)
This function reads the shapefiles of the countries (land areas) and of the exclusive economic zones (sea areas) within the scope, and creates two rasters out of them.

Parameters

- **paths** (*dict*) – Dictionary including the paths *LAND* and *EEZ*.
- **param** (*dict*) – Dictionary including the geodataframes of the shapefiles, the number of features, the coordinates of the bounding box of the spatial scope, and the number of rows and columns.

Returns The tif files for *LAND* and *EEZ* are saved in their respective paths, along with their metadata in JSON files.

Return type None

lib.input_maps.**generate_settlements**(*paths*, *param*)

This function reads the global map of settlements, and creates a raster out of it for the desired scope. See config.py for more information on the settlements map.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the global settlements raster *WSF_global* and to the output path *WSF*.
- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and the georeference dictionary.

Returns The tif file for *WSF* is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_slope**(*paths*, *param*, *A_TOPO*)
This function reads the topography raster for the scope, and creates a raster of slope out of it. The slope is calculated in percentage, although this can be changed easily at the end of the code.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the topography map of the scope *TOPO* and to the output path *SLOPE*.
- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and the georeference dictionary.

Returns The tif file for SLOPE is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_topography**(*paths*, *param*)
This function reads the tiles that make the global map of topography, picks those that lie completely or partially in the scope, and creates a raster out of them for the desired scope. The values are in meter.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the tiles of the global topography raster *Topo_tiles* and to the output path *TOPO*.
- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and the georeference dictionary.

Returns The tif file for *TOPO* is saved in its respective path, along with its metadata in a JSON file.

Return type None

lib.input_maps.**generate_weather_files**(*paths*, *param*)
This function reads the daily NetCDF data (from MERRA-2) for SWGDN, SWTDN, T2M, U50m, and V50m, and saves them in matrices with yearly time series with low spatial resolution. Depending on the *MERRA_correction* parameter this function will also call clean_weather_data() to remove data outliers. This function has to be run only once.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the MERRA-2 input files *MERRA_IN*, and to the desired output locations for *T2M*, *W50M* and *CLEARNESS*.
- **param** (*dict*) – Dictionary including the year, the spatial scope, and the MERRA_correction parameter.

Returns The files T2M.mat, W50M.mat, and CLEARNESS.mat are saved directly in the defined paths, along with their metadata in JSON files.

Return type None

lib.input_maps.**generate_weather_offshore_files**(*paths*, *param*)
This function reads the daily NetCDF data (from MERRA-2) for U50m, and V50m, and saves them in matrices with yearly time series with low spatial resolution. Depending on the *MERRA_correction* parameter this function will also call clean_weather_data() to remove data outliers. This function has to be run only once.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the MERRA-2 input files *MERRA_IN*, and to the desired output locations for *T2M*, *W50M* and *CLEARNESS*.
- **param** (*dict*) – Dictionary including the year, the spatial scope, and the MERRA_correction parameter.

Returns The file W50M.mat is saved directly in the defined paths, along with their metadata in JSON files.

Return type None

3.3 potential.py

lib.potential.**calc_FLH_solar**(*hours*, *args*)
This function computes the full-load hours for all valid pixels specified in *ind_nz* in *param*. Due to parallel processing, most of the inputs are collected in the list *args*.

Parameters

- **hours** (*numpy array*) – Filtered day hour ranks in a year (from 0 to 8759).
- **args** (*list*) – List of arguments: * *param* (dict): Dictionary including multiple parameters such as the status bar limit, the name of the region, and others for calculating the hourly capacity factors. * *tech* (str): Name of the technology. * *rasterData* (dict): Dictionary of numpy arrays containing land use types, Ross coefficients, albedo coefficients, and wind speed correction for every point in *reg_ind*. * *merraData* (dict): Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.

Return FLH Full-load hours over the year for the technology.

Return type numpy array

lib.potential.**calc_FLH_windoff**(*hours*, *args*)
This function computes the full-load hours for all valid pixels specified in *ind_nz* in *param*. Due to parallel processing, most of the inputs are collected in the list *args*.

Parameters

- **hours** (*numpy array*) – Hour ranks in a year (from 0 to 8759).
- **args** (*list*) – List of arguments: * *param* (dict): Dictionary including multiple parameters such as the status bar limit, the name of the region, and others for calculating the hourly capacity factors. * *tech* (str): Name of the technology. * *rasterData* (dict): Dictionary of numpy arrays containing land use types, Ross coefficients, albedo coefficients, and wind speed correction for every point in *reg_ind*. * *merraData* (dict): Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.

Return FLH Full-load hours over the year for the technology.

Return type numpy array

lib.potential.**calc_FLH_windon**(*param*, *tech*, *rasterData*, *merraData*, *GWA_array*, *b_xmin*, *b_xmax*, *b_ymin*, *b_ymax*, *x_gwa*, *y_gwa*, *pixles*, *list_results*)
This function computes the full-load hours for all valid pixels specified in *ind_nz* in *param*. Due to parallel processing, most of the inputs are collected in the list *args*.

Parameters

- **hours** (*numpy array*) – Hour ranks in a year (from 0 to 8759).
- **args** (*list*) – List of arguments: * *param* (dict): Dictionary including multiple parameters such as the status bar limit, the name of the region, and others for calculating the hourly capacity factors. * *tech* (str): Name of the technology. * *rasterData* (dict): Dictionary of numpy arrays containing land use types, Ross coefficients, albedo coefficients, and wind speed correction for every point in *reg_ind*. * *merraData* (dict): Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.

Return FLH Full-load hours over the year for the technology.

Return type numpy array

lib.potential.**calc_gcr**(*Crd_all*, *m_high*, *n_high*, *res_desired*, *GCR*)
This function creates a GCR weighting matrix for the desired geographic extent. The sizing of the PV system is conducted on a user-defined day for a shade-free exposure to the sun during a given number of hours.

Parameters

- **Crd_all** (*list*) – Desired geographic extent of the whole region (north, east, south, west).
- **m_high** (*int*) – Number of rows.
- **n_high** (*int*) – Number of columns.
- **res_desired** (*list*) – Resolution of the high resolution map.
- **GCR** (*dict*) – Dictionary that includes the user-defined day and the duration of the shade-free period.

Return A_GCR GCR weighting raster.

Return type numpy array

lib.potential.**calculate_full_load_hours**(*paths*, *param*, *tech*, *multiprocessing*)
This function calculates the yearly FLH for a technology for all valid pixels in a spatial scope. Valid pixels are land pixels for WindOn, PV and CSP, and sea pixels for WindOff. The FLH values are calculated by summing up hourly capacity factors.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the input weather data, land, sea and land use rasters, and correction rasters.
- **param** (*dict*) – Dictionary of dictionaries containing the spatial scope, and technology and computation parameters.
- **tech** (*str*) – Technology under study.
- **multiprocessing** (*bool*) – Determines if the computation uses multiprocessing (True/False)

Returns The raster of FLH potential is saved as mat and tif files, along with the json metadata file.

Return type None

lib.potential.**get_merra_raster_data**(*paths*, *param*, *tech*)
This function returns a tuple of two dictionaries containing weather and correction rasters for specified technology.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the input weather and raster data.
- **param** (*dict*) – Dictionary of dictionaries containing land use, Ross coefficients, albedo, and Hellmann coefficients.
- **tech** (*str*) – Technology under study.

Return (merraData, rasterData) Dictionaries for the weather data and for the correction data.

Return type tuple (dict, dict)

lib.potential.**mask_potential_maps**(*paths*, *param*, *tech*)
This function first reads the rasters for land use, slope, bathymetry, and protected areas for the scope. Based on user-defined assumptions on their suitabilities, it generates a masking raster to exclude the unsuitable pixels. Both the mask itself and the masked potential rasters can be saved.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing user-defined parameters for masking, protected areas, and landuse.
- **param** (*dict*) – Dictionary of dictionaries containing the paths to the land use, protected areas, slope and bathymetry, in addition to output paths.
- **tech** (*str*) – Technology under study.

Returns The files for the mask and the masked FLH are saved as tif and mat files, along with their metadata json files.

Return type None

lib.potential.**redistribution_array**(*param*, *merraData*, *i*, *j*, *xmin*, *xmax*, *ymin*, *ymax*, *GWA_array*, *x_gwa*, *y_gwa*)

What does this function do?

Parameters

- **param** –
- **merraData** –
- **i** –
- **j** –
- **xmin** –
- **xmax** –
- **ymin** –
- **ymax** –
- **GWA_array** –
- **x_gwa** –
- **y_gwa** –

Return reMerra

Return type numpy array

Aim: Increase the resolution of the MERRA wind data by using Global Wind Atlas data. For this reason, the low resolution MERRA data is redistributed by the energy distribution of the higher resolution Global Wind Atlas data

Algorithm:

1) Import wind data from Global Wind Atlas
2) Select pixels that are within one MERRA pixle
3) Convert from wind speed to energy
4) Redistribute MERRA data

ToDo: Where do the borders/limits come from? -> not shape file ?!?

lib.potential.**report_potentials**(*paths*, *param*, *tech*)

This function reads the FLH files and the subregion shapefile, and creates a CSV file containing various statistics:

- Available number of pixels, before and after masking
- Available area in in km^2
- FLH mean, median, max, min values, before and after masking
- FLH standard deviation after masking
- Power Potential in GW, before and after weighting
- Energy Potential in TWh in total, after weighting, and after masking and weighting
- Sorted sample of FLH values for each region

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to FLH, Masking, Weighting, and Area rasters.
- **param** (*dict*) – Dictionary of dictionaries containing technology parameters and sampling parameters.
- **tech** (*str*) – Technology under study.

Returns The CSV files with the report and the sorted FLH are saved directly in the desired paths, along with the corresponding metadata in JSON files.

Return type None

lib.potential.**sampled_sorting**(*Raster*, *sampling*)
This function returns a list with a defined length of sorted values sampled from a numpy array.

Parameters

- **Raster** (*numpy array*) – Input raster to be sorted.
- **sampling** (*int*) – Number of values to be sampled from the raster, defines length of output list.

Return s List of sorted values sampled from *Raster*.

Return type list

lib.potential.**weight_potential_maps**(*paths*, *param*, *tech*)
This function weights the power potential by including assumptions on the power density and the available area. Therefore, it reads the rasters for land use and protected areas for the scope. Based on user-defined assumptions on their availabilities, it generates a weighting raster to exclude the unsuitable pixels. Both the weight itself and the weighted potential rasters can be saved.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing user-defined parameters for weighting, protected areas, and landuse.
- **param** (*dict*) – Dictionary of dictionaries containing the paths to the land use, protected areas, area, in addition to output paths.
- **tech** (*str*) – Technology under study.

Returns The files for the weight and the weighted FLH are saved as tif and mat files, along with their metadata json files.

Return type None

3.4 time_series.py

lib.time_series.**calc_TS_solar**(*hours*, *args*)
This function computes the hourly PV and CSP capacity factor for the desired quantiles.

Parameters

- **hours** (*numpy array*) – Hour ranks of the year (from 0 to 8759).
- **args** (*list*) – List of arguments:
  - *param* (dict): Dictionary including multiple parameters such as the status bar limit, the name of the region, and others for calculating the hourly capacity factors.
  - *tech* (str): Name of the technology.
  - *rasterData* (dict): Dictionary of numpy arrays containing land use types, Ross coefficients, albedo coefficients, and wind speed correction for every point in *reg_ind*.

- *merraData* (dict): Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.

Return TS Array of time series for the desired quantiles for each subregion.

Return type numpy array

lib.time_series.**calc_TS_windoff**(*hours*, *args*)
This function computes the hourly onshore and offshore wind capacity factor for desired quantiles.

Parameters

- **hours** (*numpy array*) – Hour ranks of the year (from 0 to 8759).
- **args** (*list*) – List of arguments:
  - *param* (dict): Dictionary including multiple parameters such as the status bar limit, the name of the region, and others for calculating the hourly capacity factors.
  - *tech* (str): Name of the technology.
  - *rasterData* (dict): Dictionary of numpy arrays containing the wind speed correction for every point in *reg_ind*.
  - *merraData* (dict): Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.

Return TS Array of time series for the desired quantiles for each subregion.

Return type numpy array

lib.time_series.**calc_TS_windon**(*point*, *args*)
This function computes the hourly onshore and offshore wind capacity factor for desired quantiles.

Parameters

- **hours** (*numpy array*) – Hour ranks of the year (from 0 to 8759).
- **args** (*list*) – List of arguments:
  - *param* (dict): Dictionary including multiple parameters such as the status bar limit, the name of the region, and others for calculating the hourly capacity factors.
  - *tech* (str): Name of the technology.
  - *rasterData* (dict): Dictionary of numpy arrays containing the wind speed correction for every point in *reg_ind*.
  - *merraData* (dict): Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.

Return TS Array of time series for the desired quantiles for each subregion.

Return type numpy array

lib.time_series.**combinations_for_time_series**(*paths*, *param*, *tech*)
This function reads the list of generated regression coefficients for different hub heights and orientations, compares it to the user-defined modes and combos and returns a list of lists containing all the available combinations. The function will return a warning if the user input and the available time series are not congruent.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the regression output folder.
- **param** (*dict*) – Dictionary of dictionaries containing the year, the user defined combos, and subregions name.
- **tech** (*str*) – Technology under study.

Return combinations List of combinations of settings to be used in stratified time series.

Return inputfiles List of regression outputs to be used in generating the stratified time series.

Return type tuple (list, list)

Raises

- **No coefficients** – If regression coefficients are not available, a warning is raised.
- **Missing coefficients** – If regression coefficients are missing based on user-defined combos and mode, a warning is raised.

lib.time_series.**find_representative_locations**(*paths*, *param*, *tech*)

This function reads the masked FLH raster and finds the coordinates and indices of the pixels for the user-defined quantiles for each region. It creates a shapefile containing the position of those points for each region, and two MAT files with their coordinates and indices.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing path values for FLH MAT files, region statistics, and output paths.
- **param** (*dict*) – Dictionary of dictionaries containing the user-defined quantiles, FLH resolution, and spatial scope.
- **tech** (*str*) – Technology under study.

Returns The shapefile with the locations and the two MAT files for the coordinates and the indices are saved directly in the given paths, along with their corresponding metadata in JSON files.

Return type None

lib.time_series.**generate_time_series_for_regions**(*paths*, *param*, *tech*)

This function reads the coefficients obtained from the regression function as well as the generated time series for the combinations of hub heights / orientations and quantiles, to combine them according to user-defined *modes* (quantile combination) and *combos* (hub heights / orientation combinations) and saves the results (time series) in a CSV file.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the regression coefficients and the time series.
- **param** (*dict*) – Dictionary of dictionaries containing the list of subregions, the modes, and the combos.
- **tech** (*str*) – Technology under study.

Returns The stratified time series for each region, mode, and combo are saved directly in the given path, along with the metadata in a JSON file.

Return type None

lib.time_series.**generate_time_series_for_representative_locations**(*paths*, *param*, *tech*)

This function generates yearly capacity factor time-series for the technology of choice at quantile locations generated in find_locations_quantiles. The timeseries are saved in CSV files.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing paths to coordinate and indices of the quantile locations.
- **param** (*dict*) – Dictionary of dictionaries containing processing parameters.
- **tech** (*str*) – Technology under study.

Returns The CSV file with the time series for all subregions and quantiles is saved directly in the given path, along with the corresponding metadata in a JSON file.

Return type None

lib.time_series.**generate_time_series_for_specific_locations**(*paths*, *param*, *tech*)

This function generates yearly capacity factor time-series for the technology of choice at user defined locations. The timeseries are saved in CSV files.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing paths output desired locations.
- **param** (*dict*) – Dictionary of dictionaries containing processing parameters, and user-defined locations.
- **tech** (*str*) – Technology under study.

Returns The CSV file with the time series for all subregions and quantiles is saved directly in the given path, along with the corresponding metadata in a JSON file.

Return type None

Raises

- **Point locations not found** – Is raised when the dictionary containing the points names and locations is empty.
- **Points outside spatial scope** – Some points are not located inside of the spatial scope, therefore no input maps are available for the calculations

3.5 regression.py

lib.regression.**check_regression_model**(*paths*, *tech*)

This function checks the regression model parameters for nan values, and returns the FLH and TS model dataframes. If missing values are present in the input CSV files, the users are prompted if they wish to continue or can modify the corresponding files.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the FLH and TS model regression CSV files.
- **tech** (*str*) – Technology under study.

Return (FLH, TS_reg) Tuple of pandas dataframes for FLH and TS.

Return type Tuple of pandas dataframes

lib.regression.**clean_FLH_regression**(*paths*, *param*)

This function creates a CSV file containing the model FLH used for regression. If the region is present in the IRENA database, then the FLH are extracted directly from there. In case it is not present, a place holder for the regions is written in the csv file and it is the user's responsibility to fill in an appropriate value. The function will warn the user, and print all regions that are left blank.

Parameters

- **param** (*dict*) – Dictionary of dictionaries containing the list of regions.
- **paths** (*dict*) – Dictionary of dictionaries containing the paths to *IRENA_summary*, *IRENA_dict*.

Return missing List of string of the missing regions. The CSV file for the the FLH needed for the regression is saved directly in the given path, along with the corresponding metadata in a JSON file.

Return type list of str

Raises **Missing Regions** – No FLH values exist for certain regions.

lib.regression.**clean_TS_regression**(*paths*, *param*, *tech*)

This function creates a CSV file containing the model time series used for regression. If the region is present in the EMHIRES text files then the TS is extracted directly from it. If the region is not present in the EMHIRES text files, the highest FLH generated TS is used instead and is scaled to match IRENA FLH if the IRENA FLH are available.

Parameters

- **paths** (*dict*) – Dictionary containing paths to EMHIRES text files.
- **param** (*dict*) – Dictionary containing the *FLH_regression* dataframe, list of subregions contained in shapefile, and year.

Returns The time series used for the regression are saved directly in the given path, along with the corresponding metadata in a JSON file.

Return type None

Raises

- **Missing FLH** – FLH values are missing for at least one region. No scaling is applied to the time series for those regions.
- **Missing EMHIRES** – EMHIRES database is missing, generated timeseries will be used as model for all regions.

lib.regression.**combinations_for_regression**(*paths*, *param*, *tech*)

This function reads the list of generated time series for different hub heights and orientations, compares it to the user-defined combinations and returns a list of lists containing all the available combinations. The function will return a warning if the user input and the available time series are not congruent.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the regional analysis output folder.
- **param** (*dict*) – Dictionary of dictionaries containing the subregions name, year, and user-defined combinations.
- **tech** (*str*) – Technology under study.

Return combinations List of combinations for regression.

Return type list

Raises

- **missing data** – If no time series are available for this technology, a warning is raised.
- **missing combination** – If a hub height or orientation is missing based on user-defined combinations, a warning is raised.

lib.regression.**get_regression_coefficients**(*paths*, *param*, *tech*)

This function solves the following optimization problem: A combination of quantiles, hub heights or orientations is to be found, so that the error to a given historical time series (e.g. from EMHIRES for European countries) is minimized, while constraining the FLH to match a given value (for example from IRENA). The settings of the combinations can be defined by the user.

The function starts by identifying the existing settings (hub heights, orientations) and quantiles. If the combinations of time series requested by the user cannot be found, a warning is raised.

It later runs the optimization and identifies the subregions for which a solution was found. If the optimization is infeasible (too high or too low FLH values compared to the reference to be matched), the time series with the closest FLH to the reference value is used in the final output.

The output consists of coefficients between 0 and 1 that could be multiplied later with the individual time series in `time_series.generate_stratified_timeseries`. The sum of the coefficients for each combination is equal to 1.

Parameters

- **paths** (*dict*) – Dictionary including the paths to the time series for each subregion, technology setting, and quantile, to the output paths for the coefficients.
- **param** (*dict*) – Dictionary including the dictionary of regression parameters, quantiles, and year.
- **tech** (*str*) – Technology under study.

Returns The regression parameters (e.g. IRENA FLH and EMHIRES TS) are copied under *regression_in* folder, and the regression coefficients are saved in a CSV file under *regression_out* folder, along with the metadata in a JSON file.

Return type None

Raises

- **Missing Data** – No time series present for technology *tech*.
- **Missing Data for Setting** – Missing time series for desired settings (hub heights / orientations).

lib.regression.**pyomo_regression_model**()
This function returns an abstract pyomo model of a constrained least square problem for time series fitting to match model FLHs and minimize difference error with model time series.

Return model Abstract pyomo model.

Return type pyomo object

lib.regression.**read_generated_TS**(*paths*, *param*, *tech*, *settings*, *subregion*)
This function returns a dictionary containing the available time series generated by the script based on the desired technology and settings.

Parameters

- **paths** (*dict*) – Dictionary including output folder for regional analysis.
- **param** (*dict*) – Dictionary including list of subregions and year.
- **tech** (*str*) – Technology under study.
- **settings** – List of lists containing setting combinations.
- **subregion** (*str*) – Name of the subregion.

Return GenTS Dictionary of time series indexed by setting and quantile.

Return type dict

lib.regression.**regmodel_load_data**(*paths*, *param*, *tech*, *settings*, *subregion*)
This function returns a dictionary used to initialize a pyomo abstract model for the regression analysis of each region.

Parameters

- **paths** (*dict*) – Dictionary of dictionaries containing the paths to the CSV time series files.
- **param** (*dict*) – Dictionary of dictionaries contating IRENA's region list, FLHs and EMHIRES model timeseries.
- **tech** (*str*) – Technology under study.
- **settings** (*list*) – List of all the settings (hub heights/orientations) to be used in the regression.
- **subregion** (*str*) – Name of subregion.

Return data Dictionary containing regression parameters.

Return type dict

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

3.6 correction_functions.py

`lib.correction_functions.`**`clean_IRENA_summary`**(*paths*, *param*)

This function reads the IRENA database, format the output for selected regions and computes the FLH based on the installed capacity and yearly energy production. The results are saved in CSV file.

Parameters

- **`param`** (*dict*) – Dictionary of dictionaries containing list of subregions, and year.
- **`paths`** (*dict*) – Dictionary of dictionaries containing the paths to the IRENA country name dictionary, and IRENA database.

Returns The CSV file containing the summary of IRENA data for the countries within the scope is saved directly in the desired path, along with the corresponding metadata in a JSON file.

Return type None

`lib.correction_functions.`**`clean_weather_data`**(*p*, *paths*, *param*)

This function detects data outliers in the weather input .mat files. An outlier is a data point, for which the absolute value of the difference between the yearly average value and the mean of the direct neighbors (Moore neighborhood) is higher than a user-defined threshold *MERRA_correction_factor*. It replaces the hourly values with the hourly values of the mean of the neighbors, and overwrites the original .mat file.

Parameters

- **`paths`** (*dict*) – Dictionary including the path to the weather .mat files.
- **`param`** (*dict*) – Dictionary including the threshold value *MERRA_correction_factor*.

Returns The file weather .mat files are overwritten after the correction.

Return type None

`lib.correction_functions.`**`generate_wind_correction`**(*paths*, *param*)

This function creates a matrix of correction factors for onshore and/or offshore wind. There are different types of correction:

- Gradient correction: Adjusts for the hub height of the wind turbines, based on the Hellmann coefficients of each land use type. This correction applies always.
- Resolution correction: Performs a redistribution of wind speed when increasing the resolution based on land use types, while ensuring that the average of each MERRA-2 cell at 50m is still the same. This correction is optional, and is activated if *res_correction* is 1. If not activated, the same value from the low resolution is repeated.
- Topographic/Orographic correction: Takes into account the elevation of the terrain, because MERRA-2 usually underestimates the wind speed in mountains. This correction is optional, uses data from the Global Wind Atlas for all countries in the scope, and is activated only for onshore wind if *topo_correction* is 1

Parameters

- **`paths`** (*dict*) – Dictionary of dictionaries containing the paths to the land, land use, and topography rasters, and to the output files CORR_ON and CORR_OFF.
- **`param`** (*dict*) – Dictionary of dictionaries containing user-preferences regarding the wind correction, landuse, hub height, weather and desired resolutions.

Returns The rasters for wind correction CORR_ON and/or CORR_OFF are saved directly in the user-defined paths, along with their metadata in JSON files.

Return type None

3.7 spatial_functions.py

lib.spatial_functions.**adjust_resolution**(*array*, *res_data*, *res_desired*, *aggfun=None*)
description

lib.spatial_functions.**aggregate_x_dim**(*array*, *res_data*, *res_desired*, *aggfun*)
description

lib.spatial_functions.**aggregate_y_dim**(*array*, *res_data*, *res_desired*, *aggfun*)
description

lib.spatial_functions.**array2raster**(*newRasterfn*, *rasterOrigin*, *pixelWidth*, *pixelHeight*, *array*)
This function saves array to geotiff raster format based on EPSG 4326.

Parameters

- **newRasterfn** (*string*) – Output path of the raster.
- **rasterOrigin** (*list of two floats*) – Latitude and longitude of the Northwestern corner of the raster.
- **pixelWidth** (*integer*) – Pixel width (might be negative).
- **pixelHeight** (*integer*) – Pixel height (might be negative).
- **array** (*numpy array*) – Array to be converted into a raster.

Returns The raster file will be saved in the desired path *newRasterfn*.

Return type None

lib.spatial_functions.**calc_geotiff**(*Crd_all*, *res_desired*)
This function returns a dictionary containing the georeferencing parameters for geotiff creation, based on the desired extent and resolution.

Parameters

- **Crd_all** (*numpy array*) – Coordinates of the bounding box of the spatial scope.
- **res_desired** (*list*) – Desired data resolution in the vertical and horizontal dimensions.

Return GeoRef Georeference dictionary containing *RasterOrigin*, *RasterOrigin_alt*, *pixelWidth*, and *pixelHeight*.

Return type dict

lib.spatial_functions.**calc_region**(*region*, *Crd_reg*, *res_desired*, *GeoRef*)
This function reads the region geometry, and returns a masking raster equal to 1 for pixels within and 0 outside of the region.

Parameters

- **region** (*Geopandas series*) – Region geometry
- **Crd_reg** (*list*) – Coordinates of the region
- **res_desired** (*list*) – Desired high resolution of the output raster
- **GeoRef** (*dict*) – Georeference dictionary containing *RasterOrigin*, *RasterOrigin_alt*, *pixelWidth*, and *pixelHeight*.

Return A_region Masking raster of the region.

Return type numpy array

lib.spatial_functions.**crd2ind**(*Crd_points*, *Crd_all*, *resolution*)
This function converts latitude and longitude of points in high resolution rasters into indices.

Parameters

- **Crd_points** (*tuple of arrays*) – Coordinates of the points in the vertical and horizontal dimensions.
- **Crd_all** (*numpy array*) – Array of coordinates of the bounding box of the spatial scope.
- **resolution** (*list*) – Data resolution in the vertical and horizontal dimensions.

Return Ind_points Tuple of arrays of indices in the vertical and horizontal axes.

Return type list of arrays

lib.spatial_functions.**crd_merra**(*Crd_regions*, *res_weather*)
This function calculates coordinates of the bounding box covering MERRA-2 data.

Parameters

- **Crd_regions** (*numpy array*) – Coordinates of the bounding boxes of the regions.
- **res_weather** (*list*) – Weather data resolution.

Return Crd Coordinates of the bounding box covering MERRA-2 data for each region.

Return type numpy array

lib.spatial_functions.**define_spatial_scope**(*scope_shp*)
This function reads the spatial scope shapefile and returns its bounding box.

Parameters **scope_shp** (*Geopandas dataframe*) – Spatial scope shapefile.

Return box List of the bounding box coordinates.

Return type list

lib.spatial_functions.**ind2crd**(*Ind_points*, *Crd_all*, *resolution*)
This function converts indices of points in high resolution rasters into longitude and latitude coordinates.

Parameters

- **Ind_points** (*tuple of arrays*) – Tuple of arrays of indices in the vertical and horizontal axes.
- **Crd_all** (*numpy array*) – Array of coordinates of the bounding box of the spatial scope.
- **resolution** (*list*) – Data resolution in the vertical and horizontal dimensions.

Return Crd_points Coordinates of the points in the vertical and horizontal dimensions.

Return type list of arrays

lib.spatial_functions.**ind_global**(*Crd*, *res_desired*)
This function converts longitude and latitude coordinates into indices on a global data scope, where the origin is at (-90, -180).

Parameters

- **Crd** (*numpy array*) – Coordinates to be converted into indices.
- **res_desired** (*list*) – Desired resolution in the vertical and horizontal dimensions.

Return Ind Indices on a global data scope.

Return type numpy array

lib.spatial_functions.**ind_merra**(*Crd*, *Crd_all*, *res*)
This function converts longitude and latitude coordinates into indices within the spatial scope of MERRA-2 data.

Parameters

- **Crd** (*numpy array*) – Coordinates to be converted into indices.
- **Crd_all** (*numpy array*) – Coordinates of the bounding box of the spatial scope.

- **res** (*list*) – Resolution of the data, for which the indices are produced.

Return Ind Indices within the spatial scope of MERRA-2 data.

Return type numpy array

lib.spatial_functions.**subset**(*A*, *crd*, *param*)
This function retrieves a subset of the global MERRA-2 coverage based on weather resolution and the bounding box coordinates of the spatial scope.

Parameters

- **A** (*numpy array*) – Weather data on a global scale.
- **param** (*dict*) – Dictionary of parameters containing MERRA-2 coverage and the name of the region.

Return subset The subset of the weather data contained in the bounding box of *spatial_scope*.

Return type numpy array

3.8 physical_models.py

lib.physical_models.**angles**(*hour*, *reg_ind*, *Crd_all*, *res_desired*, *orient*)
This function creates multiple matrices for the whole scope, that represent the incidence, hour angles, declination, elevation, tilt, azimuth and orientation angles of every pixel with the desired resolution.

Parameters

- **hour** (*int*) – Hour rank in a year (from 0 to 8759).
- **reg_ind** (*tuple of arrays*) – indices of valid pixels within the spatial scope (pixels on land).
- **Crd_all** (*list*) – Coordinates of the bounding box of the spatial scope.
- **res_desired** (*list*) – Desired high resolution in degrees.
- **orient** (*int*) – Azimuth orientation of the module in degrees.

Return (phi, omega, delta, alpha, beta, azi, orientation) Rasters of latitude, hour, declination, elevation, tilt, azimuth and orientation angles.

Return type tuple of arrays

lib.physical_models.**calc_CF_solar**(*hour*, *reg_ind*, *param*, *merraData*, *rasterData*, *tech*)
This function computes the hourly capacity factor for PV and CSP technologies for all valid pixels within the spatial scope for a given hour.

Parameters

- **hour** (*integer*) – Hour within the year (from 0 to 8759).
- **reg_ind** (*tuple of arrays*) – indices of valid pixels within the spatial scope (pixels on land).
- **param** (*dict*) – Dictionary including the desired resolution, the coordinates of the bounding box of the spatial scope, and technology parameters.
- **merraData** (*dict*) – Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.
- **rasterData** (*dict*) – Dictionary of numpy arrays containing land use types, Ross coefficients, albedo coefficients, and wind speed correction for every point in *reg_ind*.
- **tech** (*str*) – Name of the technology (`'PV'` or `'CSP'`).

Return (CF_pv, CF_csp) the capacity factors for all the points during that hour for PV and CSP.

Return type tuple (numpy array, numpy array)

lib.physical_models.**calc_CF_windoff**(*hour*, *reg_ind*, *turbine*, *m*, *n*, *merraData*, *rasterData*)
This function computes the hourly capacity factor for onshore and offshore wind for all valid pixels within the spatial scope for a given hour.

Parameters

- **hour** (*integer*) – Hour within the year (from 0 to 8759).
- **reg_ind** (*tuple of arrays*) – indices of valid pixels within the spatial scope (pixels on land for onshore wind, on sea for offshore wind).
- **turbine** (*dict*) – Dictionary including the turbine parameters (cut-in, cut-off and rated wind speed).
- **m** (*int*) – number of rows.
- **n** (*int*) – number of columns.
- **merraData** (*dict*) – Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.
- **rasterData** (*dict*) – Dictionary of numpy arrays containing the wind speed correction for every point in *reg_ind*.

Return CF Capacity factors for all the valid points during that hour.

Return type numpy array

lib.physical_models.**calc_CF_windon**(*hours*, *turbine*, *merraData*, *rasterData*)
This function computes the hourly capacity factor for onshore and offshore wind for all valid pixels within the spatial scope for a given hour.

Parameters

- **hour** (*integer*) – Hour within the year (from 0 to 8759).
- **reg_ind** (*tuple of arrays*) – indices of valid pixels within the spatial scope (pixels on land for onshore wind, on sea for offshore wind).
- **turbine** (*dict*) – Dictionary including the turbine parameters (cut-in, cut-off and rated wind speed).
- **m** (*int*) – number of rows.
- **n** (*int*) – number of columns.
- **merraData** (*dict*) – Dictionary of numpy arrays containing the weather data for every point in *reg_ind*.
- **rasterData** (*dict*) – Dictionary of numpy arrays containing the wind speed correction for every point in *reg_ind*.

Return CF Capacity factors for all the valid points during that hour.

Return type numpy array

lib.physical_models.**coefficients**(*beta*, *ratio*, *R_b*, *A_i*, *f*)
This function creates three weighting matrices for the spatial scope with the desired resolution, that correspond to the gains/losses caused by tilting to each component of the incident irradiance (direct, diffuse, and reflected).

Parameters

- **beta** (*numpy array*) – Raster of tilt angles.
- **ratio** (*numpy array*) – Diffuse fraction of global horizontal solar radiation using the Erbs model.
- **R_b** (*numpy array*) – Ratio of incident beam to horizontal beam in the HDKR model.

- **A_i** (*numpy array*) – Anisotropy index for forward scattering circumsolar diffuse irradiance in the HDKR model.
- **f** (*numpy array*) – Modulating factor for horizontal brightening correction.

Return (F_direct, F_diffuse, F_reflected) Rasters of direct, diffuse and reflected ratios of irradiance.

Return type tuple of arrays

lib.physical_models.**global2diff**(*k_t*, *dims*)
This function estimates the global-to-diffuse irradiance ratio using the Erb model.

Parameters

- **k_t** (*numpy array*) – Raster of clearness indices.
- **dims** (*tuple*) – Dimensions of the output (similar to the dimension of the angles).

Return A_ratio Raster of global-to-diffuse irradiance ratios.

Return type numpy array

lib.physical_models.**loss**(*G_tilt_h*, *TEMP*, *A_Ross*, *pv*)
This function creates a temperature loss weighting matrix for the spatial scope.

Parameters

- **G_tilt_h** (*numpy array*) – Raster of incident irradiance on the tilted panel.
- **TEMP** (*numpy array*) – Raster of ambient temperatures in °C
- **A_Ross** (*numpy array*) – Raster of Ross coefficients for temperature sensitivity.
- **pv** (*dict*) – Dictionary containing PV-specific parameters for loss coefficient and rated temperature.

Return LOSS_TEMP raster of weighting temperature loss.

Return type numpy array

lib.physical_models.**toa_hourly**(*alpha*, *hour*)
This function returns the top of the atmosphere normal irradiance based on the solar constant, hour rank, and incidence angle.

Parameters

- **alpha** (*numpy array*) – Raster of elevation angles.
- **hour** (*int*) – Hour rank of the year (from 0 to 8759).

Return TOA_h Raster of the normal top of the atmosphere irradiance.

Return type numpy array

lib.physical_models.**tracking**(*axis*, *A_phi*, *A_alpha*, *A_beta*, *A_azimuth*)
This function computes the tilt angle and orientation based on the type of tracking, incidence, elevation tilt and azimuth angles.

Parameters

- **axis** (*int*) – Number of tracking axes (0, 1, 2). The value `0` means no tracking (fixed rack), `1` means single-axis tracking in east-west dimension, and `2` means double-axis tracking.
- **A_phi** (*numpy array*) – Raster of latitude angle.
- **A_alpha** (*numpy array*) – Raster of elevation angle.
- **A_beta** (*numpy array*) – Raster of tilt angle.
- **A_azimuth** (*numpy array*) – Raster of azimuth angle.

Return (A_orient, A_beta) Tuple of rasters for orientationa and tilt angles for specified tracking type.

Return type tuple of arrays

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

3.9 util.py

`lib.util.`**`arccosd`**(*digit*)
This function calculates the inverse cosine of a number.

Parameters **`digit`** (*`float`*) – Number between -1 and 1.

Returns The inverse cosine of the number in degrees.

Return type float

`lib.util.`**`arcsind`**(*digit*)
This function calculates the inverse sine of a number.

Parameters **`digit`** (*`float`*) – Number between -1 and 1.

Returns The inverse sine of the number in degrees.

Return type float

`lib.util.`**`arctand`**(*digit*)
This function calculates the inverse tangent of a number.

Parameters **`digit`** (*`float`*) – Number.

Returns The inverse tangent of the number in degrees.

Return type float

`lib.util.`**`changeExt2tif`**(*filepath*)
This function changes the extension of a file path to .tif.

Parameters **`filepath`** (*`str`*) – Path to the file.

Returns New path with .tif as extension.

Return type str

`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.`**`char_range`**(*c1*, *c2*)
This function creates a generator to iterate between the characters *c1* and *c2*, including the latter.

Parameters

- **`c1`** (*`char`*) – First character in the iteration.
- **`c2`** (*`char`*) – Last character in the iteration (included).

Returns Generator to iterate between the characters *c1* and *c2*.

Return type python generator

lib.util.**cosd**(*alpha*)
This function calculates the cosine of an angle in degrees.

Parameters **alpha** (*float*) – Angle in degrees.

Returns The cosine of the angle.

Return type float

lib.util.**create_json**(*filepath*, *param*, *param_keys*, *paths*, *paths_keys*)
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.**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.**hourofmonth**()
This function calculates the rank within a year of the first hour of each month.

Returns The rank of the first hour of each month.

Return type list

lib.util.**ind2sub**(*array_shape*, *ind*)
This function converts linear indices to subscripts.

Parameters

- **array_shape** (*tuple (int, int)*) – Dimensions of the array (# of rows, # of columns).
- **ind** – Linear index.

Returns Tuple of indices in each dimension (row index, column index).

Return type tuple(int, int)

lib.util.**intersection**(*lst1*, *lst2*)

This function calculates the intersection between two lists.

Parameters

- **lst1** (*list*) – First list of elements.
- **lst2** (*list*) – Second list of elements.

Return lst3 The unique elements that exist in both lists, without repetition.

Return type list

lib.util.**limit_cpu**(*check*)

This functions sets the priority of a process for CPU time and RAM allocation at two levels: average or below average.

Parameters **check** (*boolean*) – If `True`, the process is set a below average priority rating allowing other programs to run undisturbed. if `False`, the process is given the same priority as all other user processes currently running on the machine, leading to faster calculation times.

Returns The priority of the process is set.

Return type None

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.**sind**(*alpha*)

This function calculates the sine of an angle in degrees.

Parameters **alpha** (*float*) – Angle in degrees.

Returns The sine of the angle.

Return type float

lib.util.**sumnorm_MERRA2**(*A*, *m*, *n*, *res_low*, *res_desired*)

This function calculates the average of high resolution data if it is aggregated into a lower resolution.

Parameters

- **A** (*numpy array*) – High-resolution data.
- **m** (*int*) – Number of rows in the low resolution.
- **n** (*int*) – Number of columns in the low resolution.

- **res_low** (*numpy array*) – Numpy array with 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).
- **res_desired** (*numpy array*) – Numpy array with 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).

Return s Aggregated average of *A* on the low resolution.

Return type numpy array

lib.util.**tand**(*alpha*)
This function calculates the tangent of an angle in degrees.

Parameters **alpha** (*float*) – Angle in degrees.

Returns The tangent of the angle.

Return type float

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.

Bibliography

[1] MERRA-2: File Specification. Note No. 9. URL: http://gmao.gsfc.nasa.gov/pubs/office_notes.

[2] Danish Wind Industry Association. *Wind energy reference manual*. 2003.

[3] John A. Duffie and William A. Beckman. *Solar engineering of thermal processes*. Wiley, Hoboken, New Jersey, fourth edition edition, 2013. ISBN 9780470873663. URL: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10683270, doi:10.1002/9781118671603.

[4] D. G. Erbs, S. A. Klein, and J. A. Duffie. Estimation of the diffuse radiation fraction for hourly, daily and monthly-average global radiation. *Solar Energy*, 28(4):293–302, 1982. doi:10.1016/0038-092X(82)90302-4.

[5] Ronald Gelaro, Will McCarty, Max J. Suárez, Ricardo Todling, Andrea Molod, Lawrence Takacs, Cynthia A. Randles, Anton Darmenov, Michael G. Bosilovich, Rolf Reichle, Krzysztof Wargan, Lawrence Coy, Richard Cullather, Clara Draper, Santha Akella, Virginie Buchard, Austin Conaty, Arlindo M. da Silva, Wei Gu, Gi-Kong Kim, Randal Koster, Robert Lucchesi, Dagmar Merkova, Jon Eric Nielsen, Gary Partyka, Steven Pawson, William Putman, Michele Rienecker, Siegfried D. Schubert, Meta Sienkiewicz, and Bin Zhao. The modern-era retrospective analysis for research and applications, version 2 (merra-2). *Journal of Climate*, 30(14):5419–5454, 2017. doi:10.1175/JCLI-D-16-0758.1.

[6] Martin Kaltschmitt, Wolfgang Streicher, and Andreas Wiese. *Renewable Energy: Technology, and Environment Economics*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 9783540709473. URL: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10229340, doi:10.1007/3-540-70949-5.

[7] S. A. Klein. Calculation of monthly average insolation on tilted surfaces. *Solar Energy*, 19(4):325–329, 1977. doi:10.1016/0038-092X(77)90001-9.

[8] Gilbert M. Masters. *Renewable and efficient electric power systems*. Wiley-Interscience, Hoboken, 2004. ISBN 9780471280606. doi:10.1002/0471668826.

[9] Laura Maturi, Giorgio Belluardo, David Moser, and Matteo Del Buono. Bipv system performance and efficiency drops: overview on pv module temperature conditions of different module types. *Energy Procedia*, 48:1311–1319, 2014. doi:10.1016/j.egypro.2014.02.148.

[10] Douglas T. Reindl. *Estimating Diffuse Radiation On Horizontal Surfaces And Total Radiation On Tilted Surfaces: Thesis*. PhD thesis, 1988. URL: https://minds.wisconsin.edu/bitstream/1793/47660/1/0001.pdf.

[11] K. Scharmer and J. Greif. *The European solar radiation atlas: Database and Exploitation Software*. Volume 2. Presses de l Ecole des Mines, Paris, 2000. ISBN 2911762223.

[12] B. Stine William and Geyer Michael. Powerfromthesun.net. 2014.

[13] F. Marion William and P. Dobos Aron. Rotation angle for the optimum tracking of one-axis trackers. 2013. URL: https://www.nrel.gov/docs/fy13osti/58891.pdf.

Python Module Index

c

l

Index

H

I

L

M

P

R

S

T

W