geocoding

„geocodingETRS89.py“:

**Function Name:** get\_coordinates(address)

**Description:** This function takes an address as input and attempts to retrieve the corresponding geographic coordinates (longitude and latitude) using the Nominatim geocoding service. It also transforms these coordinates from the WGS84 (GPS) coordinate system to the ETRS89 / UTM Zone 33N coordinate system.

**Parameters:**

* address (str): The address for which coordinates need to be retrieved.

**Returns:**

* A tuple containing the UTM Zone 33N coordinates (UTM\_X and UTM\_Y) in meters as (utm\_x, utm\_y). If the geocoding is successful, it returns the coordinates; otherwise, it returns (None, None).

**Function Name:** process\_data(input\_csv, output\_csv)

**Description:** This function reads data from an input CSV file, processes each row, geocodes the address in the row to obtain UTM coordinates, and writes the original data along with the transformed UTM coordinates to an output CSV file.

**Parameters:**

* input\_csv (str): The path to the input CSV file containing data to be processed. The CSV file should have a specific structure with columns: "Country," "State," "City," "Address," and additional columns.
* output\_csv (str): The path to the output CSV file where the processed data with UTM coordinates will be written.

**Behavior:**

1. The function opens the input CSV file for reading and the output CSV file for writing.
2. It reads the input CSV data row by row, extracting relevant information such as country, state, city, and address.
3. It combines these pieces of information to create a full address string.
4. It calls the get\_coordinates function to geocode the full address and obtain UTM coordinates.
5. The original row data is then extended with two additional columns, "UTM\_X" and "UTM\_Y," containing the UTM coordinates.
6. The processed data is written to the output CSV file, including the newly added UTM coordinates.
7. The function prints "Processing completed" when finished.

**Note:**

* The code snippet at the end of the script demonstrates how to call the process\_data function by specifying the input and output CSV file paths. You can uncomment and customize these lines to execute the geocoding and data processing.

This function allows you to geocode addresses from an input CSV file, retrieve UTM coordinates, and create a new CSV file with the original data extended with UTM coordinates for further spatial analysis or mapping applications.

Net\_generation

net\_generation.py

The general functionality of the provided scripts and how they work together is to generate network layers for heating pipelines based on geospatial data. Here's an overview of how the process works:

1. **Data Loading and Configuration**:
   * The scripts begin by importing necessary libraries for geospatial data manipulation, such as GeoPandas and Shapely, and additional libraries for mathematical calculations and network analysis.
2. **Data Input**:
   * The user provides project-specific inputs by setting the projekt variable to specify which project (e.g., "Zittau" or "Görlitz") is being worked on. These inputs determine the source of geospatial data and producer location coordinates.
3. **Loading Geospatial Data**:
   * The script loads the following geospatial data layers and information:
     + OpenStreetMap (OSM) street layer in GeoJSON format, representing the road network.
     + Data points from a CSV file, which likely include information about heating requirements or other relevant data.
     + Coordinates for the producer's location (x\_coord and y\_coord).
4. **Generating Network Layers**:
   * The scripts use a set of functions, primarily located in the "simple\_MST.py" script, to generate different types of network layers for heating pipelines:
     + generate\_lines: Generates lines based on input data points and attributes.
     + generate\_return\_lines: Generates return lines for heating pipelines.
     + generate\_network\_fl: Generates network layers for flow lines (Vorlaufleitungen).
     + generate\_network\_rl: Generates network layers for return lines (Rücklaufleitungen).
     + generate\_mst: Generates Minimum Spanning Trees (MST) to optimize network connections.
5. **Coordinate Transformation**:
   * The generated network layers are set with the Coordinate Reference System (CRS) to ensure proper spatial alignment and compatibility.
6. **Exporting Network Layers**:
   * The final step involves exporting the generated network layers as GeoJSON files. These exported layers include:
     + Vorlaufleitungen (Flow Lines)
     + Rücklaufleitungen (Return Lines)
     + Heating Demand (Wärmebedarf) information
     + Erzeugeranlagen (Producer Facilities)
7. **Script Execution**:
   * The user executes the script, and based on the selected project, the script loads the appropriate data, generates network layers, sets the CRS, and exports the layers to GeoJSON files.

In essence, these scripts provide a framework for generating and analyzing network layers for heating pipelines. They take project-specific data, transform it into network layers, and export the results for further analysis or use in a geographical information system (GIS) environment. The process is modular and can be adapted to different projects by changing input parameters and data sources.

**Documentation for import\_and\_create\_layers.py:**

This script is responsible for loading geospatial data, generating network layers for heating pipelines, and exporting the results to GeoJSON files. It uses functions from the "simple\_MST" module to create network layers.

1. import\_osm\_street\_layer(osm\_street\_layer\_geojson\_file):
   * This function loads an OpenStreetMap (OSM) street layer from a GeoJSON file.
   * It returns a GeoDataFrame containing the street layer geometry.
2. generate\_lines(layer, distance, angle\_degrees, df=None):
   * This function generates lines based on input data points and attributes.
   * It calculates the Wärmebedarf (heat demand) based on the provided DataFrame (df) and creates offset lines.
   * Returns a GeoDataFrame containing the generated lines.
3. load\_layers(osm\_street\_layer\_geojson\_file, data\_csv\_file\_name, x\_coord, y\_coord):
   * This function loads various layers and data required for network generation.
   * It loads the street layer, data points from a CSV file, and the producer's location.
   * Returns the loaded layers and data as GeoDataFrames and DataFrames.
4. generate\_and\_export\_layers(osm\_street\_layer\_geojson\_file\_name, data\_csv\_file\_name, x\_coord, y\_coord, fixed\_angle=0, fixed\_distance=1):
   * This is the main function that orchestrates the process of generating network layers.
   * It calls the necessary functions to generate lines for different types of heating pipelines (HAST, Rücklauf, Vorlauf, Erzeugeranlagen).
   * Sets the Coordinate Reference System (CRS) and exports the resulting GeoDataFrames to GeoJSON files.

**Documentation for simple\_MST.py:**

This script contains various functions used for generating network layers, including Vorlaufleitungen (flow lines) and Rücklaufleitungen (return lines), as well as finding Minimum Spanning Trees (MST) for network optimization.

1. create\_offset\_points(point, distance, angle\_degrees):
   * This function calculates and returns an offset point from a given point based on a specified distance and angle in degrees.
2. find\_nearest\_line(point, line\_layer):
   * This function finds the nearest line in a given layer to a specified point.
   * It returns the nearest line geometry.
3. create\_perpendicular\_line(point, line):
   * This function creates a perpendicular line from a given point to a specified line.
   * It returns the perpendicular line as a LineString.
4. process\_layer\_points(layer, layer\_lines):
   * This function processes layer points and finds perpendicular lines to nearby lines.
   * It returns a set of street end points where perpendicular lines intersect nearby lines.
5. generate\_return\_lines(layer, distance, angle\_degrees, layer\_lines):
   * This function generates return lines for heating pipelines based on input layer points, distance, and angle.
   * It returns a set of street end points for return lines.
6. generate\_mst(points):
   * This function generates a Minimum Spanning Tree (MST) network from a set of points.
   * It constructs a graph where each point is a node, and edges are weighted based on the distance between points.
   * Returns the MST as a GeoDataFrame containing LineString geometries.
7. generate\_network\_fl(layer\_points\_fl, layer\_wea, layer\_lines):
   * This function generates network layers for Vorlaufleitungen (flow lines) in the heating network.
   * It calculates perpendicular lines for points in both layer\_points\_fl and layer\_wea.
   * Returns a GeoDataFrame containing the final network layer.
8. generate\_network\_rl(layer\_points\_rl, layer\_wea, fixed\_distance\_rl, fixed\_angle\_rl, layer\_lines):
   * This function generates network layers for Rücklaufleitungen (return lines) in the heating network.
   * It calculates perpendicular lines for points in both layer\_points\_rl and layer\_wea with specified fixed distance and angle.
   * Returns a GeoDataFrame containing the final network layer.

These functions collectively support the generation of heating network layers, including flow and return lines, and the optimization of these networks using Minimum Spanning Trees (MST).

heat\_requirement

**Script Name:** heat\_requirement\_BDEW.py

**Description:** This script is designed for calculating the heat demand profile (Lastgang) for a given year based on various parameters, including temperature data and load profile coefficients. The calculations are performed at both daily and hourly intervals.

**Functions:**

1. **import\_TRY(filename)**
   * Description: This function imports temperature data from a specific file in the TRY format and extracts the hourly temperature values.
   * Parameters:
     + filename (str): The path to the TRY (Temperature Reference Year) data file.
   * Returns:
     + temperature (numpy.ndarray): An array containing the hourly temperature values.
2. **generate\_year\_months\_days\_weekdays(year)**
   * Description: Generates arrays representing days of the year, months, days of the month, and weekdays (1-7) for a given year.
   * Parameters:
     + year (int): The year for which the arrays should be generated.
   * Returns:
     + Four arrays representing days\_of\_year, months, days, and daily\_weekdays.
3. **calculate\_daily\_averages(temperature)**
   * Description: Calculates daily average temperatures from hourly temperature data.
   * Parameters:
     + temperature (numpy.ndarray): An array containing hourly temperature values.
   * Returns:
     + daily\_avg\_temperature (numpy.ndarray): An array containing daily average temperatures.
4. **calculate\_quarter\_hourly\_intervals(year)**
   * Description: Generates an array of quarter-hourly intervals for the specified year.
   * Parameters:
     + year (int): The year for which quarter-hourly intervals should be generated.
   * Returns:
     + hourly\_intervals (numpy.ndarray): An array of quarter-hourly intervals.
5. **get\_coefficients(lastprofiltyp, subtyp, daily\_data)**
   * Description: Retrieves load profile coefficients (A, B, C, D, mH, bH, mW, bW) for a specific load profile.
   * Parameters:
     + lastprofiltyp (str): The main load profile type.
     + subtyp (str): The subtype within the main load profile type.
     + daily\_data (pandas.DataFrame): A DataFrame containing load profile coefficients.
   * Returns:
     + Coefficient values as individual floats.
6. **get\_weekday\_factor(daily\_weekdays, lastprofiltyp, subtyp, daily\_data)**
   * Description: Retrieves weekday factors for a specific load profile, given the weekdays and load profile information.
   * Parameters:
     + daily\_weekdays (numpy.ndarray): An array representing weekdays (1-7) for each day.
     + lastprofiltyp (str): The main load profile type.
     + subtyp (str): The subtype within the main load profile type.
     + daily\_data (pandas.DataFrame): A DataFrame containing load profile coefficients.
   * Returns:
     + weekday\_factors (numpy.ndarray): An array of weekday factors for each day.
7. **berechnung\_lastgang(weather\_data, JWB\_kWh, lastprofiltyp, subtyp, Feiertage, year=2019)**
   * Description: Calculates the heat demand profile (Lastgang) for a specified year using temperature data and load profile coefficients.
   * Parameters:
     + weather\_data (str): The path to the temperature data file (TRY or other).
     + JWB\_kWh (float): The annual heating energy demand in kilowatt-hours (kWh).
     + lastprofiltyp (str): The main load profile type.
     + subtyp (str): The subtype within the main load profile type.
     + Feiertage (numpy.ndarray): An array of holidays in 'datetime64[D]' format.
     + year (int, optional): The year for which the Lastgang should be calculated (default is 2019).
   * Returns:
     + hourly\_intervals (numpy.ndarray): An array of quarter-hourly intervals.
     + hourly\_heat\_demand (numpy.ndarray): An array of hourly heat demand values (kW).
     + hourly\_temperature (numpy.ndarray): An array of hourly temperature values (°C).
8. **Jahresdauerlinie(hourly\_intervals, hourly\_heat\_demand)**
   * Description: Plots the annual duration curve (Jahresdauerlinie) for the calculated heat demand profile.
   * Parameters:
     + hourly\_intervals (numpy.ndarray): An array of quarter-hourly intervals.
     + hourly\_heat\_demand (numpy.ndarray): An array of hourly heat demand values (kW).
9. **calculate(JWB\_kWh=10000, lastprofiltyp="HMF", subtyp="03", year=2021)**
   * Description: Entry point function for calculating the Lastgang.
   * Parameters (with defaults):
     + JWB\_kWh (float): The annual heating energy demand in kilowatt-hours (default is 10,000 kWh).
     + lastprofiltyp (str): The main load profile type (default is "HMF").
     + subtyp (str): The subtype within the main load profile type (default is "03").
     + year (int): The year for which the Lastgang should be calculated (default is 2021).
   * Returns:
     + The calculated hourly\_intervals, hourly\_heat\_demand, and hourly\_temperature arrays.

**Note:**

* The script provides functions to calculate the Lastgang based on temperature data, load profile coefficients, and various input parameters.
* The calculate function can be used as the entry point to customize the calculation for different scenarios.

**Usage:**

1. Set the desired input parameters (e.g., JWB\_kWh, lastprofiltyp, subtyp, year, and weather\_data) in the calculate function.
2. Call the calculate function to obtain the Lastgang results.
3. Optionally, you can plot the Jahresdauerlinie using the Jahresdauerlinie function.

To execute the script, uncomment the last line in the script and specify the desired parameters for the calculation.

**Script Name:** heat\_requirement\_VDI4655.py

**Libraries Used:**

* pandas (as pd): For handling data in tabular format.
* numpy (as np): For numerical calculations.
* matplotlib.pyplot (as plt): For creating plots.

**Main Functions and Purpose:**

1. **import\_TRY(filename)**
   * Imports temperature and cloud cover data from a TRY (Temperature Reference Year) file.
   * Returns arrays containing temperature and cloud cover values.
2. **import\_csv(filename)**
   * Imports data from a CSV file.
   * Returns a DataFrame containing the data.
3. **generate\_year\_months\_days\_weekdays(year)**
   * Generates arrays representing days of the year, months, days of the month, and weekdays for a given year.
   * Returns these arrays.
4. **calculate\_daily\_averages(temperature, cloud\_cover)**
   * Calculates daily average temperature and cloud cover from hourly data.
   * Returns arrays containing daily average values.
5. **calculate\_quarter\_hourly\_intervals(year)**
   * Generates an array of quarter-hourly intervals for a specified year.
   * Returns this array.
6. **quarter\_hourly\_data(data)**
   * Creates an array with quarter-hourly data for a year.
   * Returns this array.
7. **standardized\_quarter\_hourly\_profile(year, building\_type, days\_of\_year, type\_days)**
   * Calculates a standardized quarter-hourly profile for a specific building type, taking into account factors like season, time of day, and cloud cover.
   * Returns quarter-hourly intervals and demand profiles for electricity, heating, and hot water.
8. **berechnung\_lastgang(weather\_data, factors, building\_type, anzahl\_personen\_haushalt, JEB\_Strom\_kWh, JEB\_Heizwärme\_kWh, JEB\_Trinkwarmwasser\_kWh, Feiertage, klimazone="9", year=2019)**
   * Calculates the energy demand profiles (electricity, heating, and hot water) for a specified year based on weather data, building type, and other parameters.
   * Returns quarter-hourly intervals and normalized demand profiles.
9. **Jahresdauerlinie(weather\_data, factors, building\_type, anzahl\_personen\_haushalt, JEB\_Strom\_kWh, JEB\_Heizwärme\_kWh, JEB\_Trinkwarmwasser\_kWh, Feiertage, klimazone="9", year=2019)**
   * Plots the annual duration curve for the energy demand profiles.
10. **calculate(JEB\_Heizwärme\_kWh, JEB\_Trinkwarmwasser\_kWh, JEB\_Strom\_kWh=1, building\_type="MFH", personen\_pro\_haushalt=2, year=2019, Klimazone="9")**
    * Main entry point for calculating energy demand profiles.
    * Calls berechnung\_lastgang to calculate demand profiles and then plots the annual duration curve using Jahresdauerlinie.

**Usage:**

1. Set the desired input parameters such as building type, number of occupants, energy demands, year, and weather data file.
2. Call the calculate function to calculate energy demand profiles.
3. The annual duration curve will be plotted to visualize the results.

To execute the script, uncomment the last lines in the script and specify the desired parameters for the calculation. This script can be used to analyze and visualize energy demand patterns for different scenarios.

Osm\_data

import\_osm\_building\_layer.py

This Python script, named import\_osm\_building\_layer.py, is designed to import and process OpenStreetMap (OSM) building data from a specified area. It uses the geopandas, pandas, and numpy libraries to handle geospatial data and perform various data filtering and calculations. Below is a documentation of the script:

**Script Overview:**

The script performs the following tasks:

1. Imports OSM building data from a specific area using an Overpass query and saves it as a GeoJSON file.
2. Reads the GeoJSON file and adds a 'full\_address' column.
3. Filters the building data based on a list of ignored building types.
4. Converts coordinates to a different coordinate reference system (CRS).
5. Calculates the area of each building and adds columns for specific heat demand, the number of floors, and annual heating demand.
6. Exports the filtered and calculated data as a new GeoJSON file.

**Functions:**

1. **import\_and\_filter\_building()**
   * Downloads OSM building data using an Overpass query and saves it as a GeoJSON file.
   * Reads the GeoJSON file and adds a 'full\_address' column.
   * Filters the building data based on addresses from a CSV file.
   * Saves the filtered data as a new GeoJSON file.
2. **filter\_building\_data(geojson\_file, output\_file)**
   * Reads an input GeoJSON file containing building data.
   * Filters out buildings based on specified ignored building types.
   * Exports the filtered data to a new GeoJSON file.
3. **calculate\_building\_data(geojson\_file, output\_file)**
   * Reads an input GeoJSON file containing building data.
   * Converts the coordinate reference system (CRS) to EPSG 25833.
   * Calculates the area of each building in square meters.
   * Adds columns for specific heat demand, the number of floors, and annual heating demand based on random values.
   * Exports the extended GeoDataFrame to a new GeoJSON file.

**Usage:**

1. Ensure that you have the necessary Python libraries (geopandas, pandas, and numpy) installed.
2. Customize the Overpass query to specify the desired area and OSM data types.
3. Set the paths for input and output GeoJSON files.
4. Uncomment the desired function calls at the end of the script (e.g., import\_and\_filter\_building(), filter\_building\_data(...), or calculate\_building\_data(...)).

**Note:**

* The script can be used to filter and analyze OSM building data for specific types of buildings and calculate additional information, such as heating demand.

Please adjust the script's parameters and functions to suit your specific needs and the area of interest for your OSM building data analysis.

Import\_osm\_data\_geojson.py

**Script Overview:**

The script is designed to download and convert OpenStreetMap (OSM) data into GeoJSON format, focusing on specific elements (e.g., roads or buildings) within a specified city area. It includes functions to build an Overpass query, download data, process the data into GeoJSON features, and save the resulting GeoJSON file.

**Functions:**

1. **build\_query(city\_name, tags, element\_type="way")**
   * Generates an Overpass query to fetch OSM data based on specified criteria, such as city name, tags (key-value pairs), and element type (default is "way" for roads).
   * Returns the constructed query as a string.
2. **download\_data(query, element\_type)**
   * Executes the Overpass query to retrieve OSM data.
   * Converts the fetched data into GeoJSON features.
   * Supports two element types: "way" (for roads) and "building" (for buildings).
   * Returns a GeoJSON FeatureCollection containing the downloaded data.
3. **json\_serial(obj)**
   * JSON serializer for handling objects that are not natively serializable to JSON.
   * Converts Decimal objects to floating-point numbers.
   * Used in the save\_to\_file function.
4. **save\_to\_file(geojson\_data, filename)**
   * Saves the GeoJSON data to a specified file with proper formatting.
   * Utilizes the json\_serial function for serializing Decimal objects.
5. **run\_here()**
   * Defines default parameters for city name and tags (e.g., highway type).
   * Builds an Overpass query, downloads OSM data, and saves it as a GeoJSON file.
   * This function can be uncommented and executed to run the script with predefined settings.

**Usage:**

1. Ensure you have the required Python libraries (overpy, json, decimal, and geojson) installed.
2. Customize the city\_name, tags, and element\_type in the run\_here() function to target the specific OSM data you need.
3. Run the script to download and save the OSM data as GeoJSON.

**Note:**

* The script allows you to tailor your data extraction by specifying the city, OSM tags, and element type (way or building).
* You can uncomment and execute the run\_here() function to run the script with predefined parameters.

This script is a useful tool for retrieving and converting OSM data into GeoJSON format, making it suitable for various geospatial applications and analyses.

Wärmeversorgungsgebiete.py

**Script Overview:**

The script aims to determine heat supply areas based on specific building data and characteristics. It employs clustering techniques to group buildings into quartiers (areas) and then allocates overlapping areas between quartiers with different heat supply methods. The final result is saved as a GeoJSON file, indicating the supply method for each quartier.

**Functions:**

1. **versorgungsgebiet\_bestimmen(flaechenspezifischer\_waermebedarf, schwellwert\_waermenetz, schwellwert\_wasserstoff)**
   * Determines the heat supply method for each quartier based on specific heat demand and predefined thresholds.
   * Returns the heat supply method as a string: "Wärmenetzversorgung," "Wasserstoffversorgung," or "Einzelversorgungslösung."
2. **clustering\_quartiere\_hdbscan(gdf, buffer\_size=10, min\_cluster\_size=30, min\_samples=1, schwellwert\_waermenetz=90, schwellwert\_wasserstoff=60)**
   * Performs clustering of buildings into quartiers using the HDBSCAN algorithm.
   * Calculates quartier-level heat demand and supply method.
   * Resolves overlapping areas between quartiers with different supply methods.
   * Returns a GeoDataFrame with quartier-level information.
3. **postprocessing\_hdbscan(quartiere)**
   * Conducts post-processing to further resolve overlapping areas between quartiers.
   * Combines quartiers with the same supply method that intersect.
   * Returns a GeoDataFrame with post-processed quartier information.
4. **allocate\_overlapping\_area(quartiere)**
   * Allocates overlapping areas between quartiers with different supply methods.
   * Ensures that each overlapping area belongs to one quartier based on heat demand.
   * Returns a GeoDataFrame with allocated quartier information.
5. **calculate\_building\_data(gdf, output\_filename)**
   * Calculates additional building data, including area, specific heat demand, number of stories, and annual heat demand.
   * Saves the updated building data as a GeoJSON file.
   * Returns the updated GeoDataFrame.
6. **run\_here()**
   * Executes the main workflow of the script.
   * Loads building data from a GeoJSON file.
   * Calculates building data, quartiers, and post-processes the results.
   * Saves the final GeoJSON file with allocated quartiers.

**Usage:**

1. Make sure you have the required Python libraries (numpy, pandas, geopandas, sklearn, and hdbscan) installed. To install hdbscan, follow the instructions provided in the script comments.
2. Customize the parameters and file paths as needed within the run\_here() function.
3. Run the script to perform clustering, post-processing, and allocation of quartiers.

**Note:**

* The script provides a comprehensive workflow for determining heat supply areas based on building characteristics.
* It utilizes HDBSCAN clustering to group buildings into quartiers and resolves overlapping areas.
* Ensure you have your building data in GeoJSON format and specify the input and output file paths accordingly.
* The script is designed for flexibility and customization, allowing you to adjust clustering and threshold parameters to suit your specific use case.

This script can be a valuable tool for urban planning, energy distribution, and infrastructure development by efficiently identifying and allocating heat supply areas within a city or region.

net\_generation\_pandapipes

„net\_test.py“

**Script Overview:**

The script uses the Pandapipes library for hydraulic modeling to generate and analyze pipeline networks. It imports and utilizes functions for network generation, correction of flow directions, and optimization of pipeline parameters, including diameter types and parameters. Additionally, it includes functions for plotting network information on a map.

**Functions:**

1. **initialize\_test\_net()**
   * Initializes and generates a test network using the Pandapipes library.
   * Creates various network components such as junctions, pipes, pumps, heat exchangers, and flow control elements.
   * Sets properties, parameters, and connections for each component.
   * Runs a pipeflow simulation to calculate hydraulic conditions.
   * Returns the generated network as a Pandapipes network object.
2. **config\_plot(net, ax, show\_junctions=True, show\_pipes=True, show\_flow\_controls=True, show\_heat\_exchangers=True, show\_plot=False)**
   * Configures and displays the network's information on a map.
   * Displays junctions, pipes, flow control elements, and heat exchangers with relevant details such as pressure, temperature, flow rate, and pipe characteristics.
   * Provides options to control which network elements are displayed.
   * Requires a Pandapipes network object (net) and an axes object (ax) for plotting.
   * Set show\_plot to True to display the plot.
3. **generate\_net(qext\_w=83000, pipe\_creation\_mode="type")**
   * Generates a pipeline network based on geospatial data using Pandapipes.
   * Imports geospatial data for various network components (vorlauf, rücklauf, HAST, Erzeugeranlagen) from GeoJSON files.
   * Creates a Pandapipes network, corrects flow directions, optimizes diameter types, and optimizes diameter parameters for heat exchangers and flow control elements.
   * Returns the generated and optimized network.
4. **generate\_test\_net()**
   * Generates a simplified test network using the initialize\_test\_net() function.
   * Configures and displays the test network with various optimization stages using the config\_plot() function.

**Usage:**

1. Ensure you have the required libraries installed. You can install Pandapipes, Geopandas, and other dependencies as needed.
2. Pandapipes is a powerful library for hydraulic modeling of pipe networks, commonly used for water, gas, or district heating systems.
3. This script serves as a basic example of how to create and simulate a network using Pandapipes.
4. You can expand upon this example by adding more components and customizing the network properties according to your specific modeling needs.
5. Customize the network generation and optimization parameters within the generate\_net() and generate\_test\_net() functions according to your specific network data and requirements.
6. Run the script to generate and analyze the pipeline network. The script will display network information on a map.

**Note:**

* Pandapipes is a powerful library for hydraulic modeling of pipeline networks, while Geopandas is used for geospatial data handling.
* This script provides an example of how to import geospatial data, create a pipeline network, correct flow directions, and optimize pipeline parameters.
* You can adapt this script to work with your own geospatial data and customize it to perform more advanced network modeling and analysis.

This script serves as a practical tool for generating, visualizing, and optimizing pipeline networks using Pandapipes and Geopandas. It is suitable for applications in hydraulic engineering, network design, and optimization.

„net\_simulation\_calculation.py“

**Script Overview:**

The script utilizes the Pandapipes library for hydraulic modeling to create, analyze, and optimize heating pipeline networks. It includes functions for generating network components, correcting flow directions, optimizing pipeline parameters (including diameter and type), creating controllers, and exporting network data to GeoJSON format.

**Functions:**

1. **get\_line\_coords\_and\_lengths(gdf)**
   * Extracts line coordinates and lengths from a GeoDataFrame.
   * Calculates the length of each line in the GeoDataFrame.
   * Returns lists of line coordinates and lengths.
2. **get\_all\_point\_coords\_from\_line\_cords(all\_line\_coords)**
   * Retrieves unique point coordinates from a list of line coordinates.
   * Returns a list of unique point coordinates.
3. **create\_network(gdf\_vorlauf, gdf\_rl, gdf\_hast, gdf\_wea, qext\_w, ...)**
   * Generates a heating pipeline network based on geospatial data using Pandapipes.
   * Imports geospatial data for various network components (vorlauf, rücklauf, HAST, Erzeugeranlagen) from GeoJSON files.
   * Creates a Pandapipes network, corrects flow directions, and optimizes diameter parameters for pipes.
   * Creates heat exchangers and circulation pumps with specified parameters.
   * Returns the generated network.
4. **create\_controllers(net, qext\_w, target\_temperature)**
   * Creates controllers for heat exchangers in the network.
   * Sets up controllers to adjust the heat exchanger's external heat input (qext\_w) based on the target temperature.
   * Returns the updated network with controllers.
5. **correct\_flow\_directions(net)**
   * Corrects flow directions in the network to ensure physically consistent results.
   * Swaps junctions for pipes with negative velocities.
   * Re-calculates the pipe flow to obtain updated results.
   * Returns the network with corrected flow directions.
6. **optimize\_diameter\_parameters(initial\_net, element="pipe", v\_max=2.2, v\_min=1.8, dx=0.001)**
   * Optimizes pipeline parameters (diameter) based on specified velocity constraints.
   * Adjusts pipe diameters iteratively until velocities are within the defined range (v\_max and v\_min).
   * Returns the network with optimized pipeline parameters.
7. **optimize\_diameter\_types(net, v\_max=1.0)**
   * Optimizes pipeline parameters (pipe type) based on specified velocity constraints.
   * Adjusts pipe types to achieve target velocities (v\_max).
   * Returns the network with optimized pipe types.
8. **export\_net\_geojson(net)**
   * Exports the network's geographical data, including pipes and pipe attributes, to a GeoJSON file.
   * Converts line coordinates to LineString geometries and sets the coordinate reference system (CRS).
   * Exports the GeoDataFrame to a GeoJSON file.
9. **calculate\_worst\_point(net)**
   * Calculates the worst point in the heating network, defined as the heat exchanger with the lowest pressure difference between the forward and return lines.
   * Returns the minimum pressure difference (dp\_min) and the corresponding heat exchanger index (idx\_dp\_min).

**Usage:**

1. Ensure you have the required libraries installed. You can install Pandapipes, Pandapower, Geopandas, and other dependencies as needed.
2. Customize the network generation parameters, including geospatial data, controller settings, and optimization criteria within the create\_network() and create\_controllers() functions.
3. Run the script to generate, analyze, and optimize the heating pipeline network. It can also correct flow directions and export network data for further analysis.

**Note:**

* This script provides a comprehensive framework for creating, controlling, and optimizing heating pipeline networks using Pandapipes and Geopandas.
* You can adapt this script to work with your own geospatial data, network design, and optimization requirements.
* It includes functionality to ensure physically consistent results, optimize pipeline parameters, and export network data for visualization and analysis.

This script is a valuable tool for engineers and researchers working on heating network design, optimization, and analysis. It facilitates the creation of realistic network models and allows for the fine-tuning of network parameters to meet specific performance criteria.

„net\_simualtion.py“

Das Skript "net\_simulation.py" ist für die Simulation von Heiznetzwerksystemen mit der Verwendung der Python-Bibliothek "pandapipes" verantwortlich. Es enthält Funktionen zum Generieren von Profilen, zur Initialisierung von Netzwerken, zur Durchführung von zeitbasierten Simulationen, zur Aktualisierung von Steuerungen und zur Berechnung sowie Speicherung von Ergebnissen. Im Folgenden wird jede Hauptfunktion und ihr Zweck im Skript erläutert:

**1. generate\_profiles\_from\_geojson**

Die Funktion generate\_profiles\_from\_geojson generiert Heizlastprofile auf Grundlage eines GeoJSON-Datensatzes, der Gebäudeinformationen und Heizbedarfsdaten enthält. Die Funktion kann verschiedene Berechnungsmethoden verwenden, um die Heizlastprofile zu erstellen.

* gdf\_HAST: Ein GeoDataFrame, das Gebäudeinformationen und Heizbedarfsdaten enthält.
* building\_type: Der Gebäudetyp, für den die Heizlastprofile berechnet werden sollen.
* calc\_method: Die Berechnungsmethode, die für die Erstellung der Profile verwendet werden soll (z. B. VDI 4655 oder BDEW).

**2. initialize\_net\_geojson**

Die Funktion initialize\_net\_geojson initialisiert ein Heiznetzwerk unter Verwendung von GeoJSON-Daten. Sie setzt Steuerungen, Flussrichtungen und andere Netzwerkparameter.

* gdf\_vl, gdf\_rl, gdf\_HAST, gdf\_WEA: GeoDataFrames, die die Vorlauf-, Rücklauf-, Gebäude- und Wärmequellendaten enthalten.
* qext\_w: Die externe Wärmezufuhr.
* return\_temperature, supply\_temperature, flow\_pressure\_pump, lift\_pressure\_pump: Verschiedene Parameter für die Netzwerkinitialisierung.
* diameter\_mm, pipetype, k, alpha, pipe\_creation\_mode: Parameter für Rohrleitungsattribute und -erstellung.

**3. init\_timeseries\_opt**

Die Funktion init\_timeseries\_opt bereitet die zeitbasierte Simulation für das Netzwerk vor. Sie legt die Controller-Daten und die Zieltemperatur fest.

* net: Das initialisierte Heiznetzwerk.
* qext\_w\_profiles: Profile für externe Wärmezufuhr über die Zeit.
* time\_steps: Zeitstufen der Simulation.
* target\_temperature: Die Zieltemperatur für die Return-Temperatur-Steuerungen.

**4. update\_const\_controls**

Die Funktion update\_const\_controls aktualisiert während der Simulation die konstanten Steuerungen mit neuen Daten.

* net: Das Heiznetzwerk.
* qext\_w\_profiles: Profile für externe Wärmezufuhr über die Zeit.
* time\_steps: Zeitstufen der Simulation.
* start, end: Indizes für den Zeitraum der Aktualisierung.

**5. update\_return\_temperature\_controller**

Die Funktion update\_return\_temperature\_controller aktualisiert die Zieltemperatur für den Return-Temperatur-Regler.

* net: Das Heiznetzwerk.
* temperature\_target: Die neue Zieltemperatur für die Return-Temperatur-Regler.

**6. update\_supply\_temperature\_controls**

Die Funktion update\_supply\_temperature\_controls aktualisiert die Steuerungen für die Versorgungstemperatur, wenn entsprechende Daten vorhanden sind.

* net: Das Heiznetzwerk.
* supply\_temperature: Die Versorgungstemperaturdaten.
* time\_steps: Zeitstufen der Simulation.
* start, end: Indizes für den Zeitraum der Aktualisierung.

**7. create\_log\_variables**

Die Funktion create\_log\_variables erstellt eine Liste von zu protokollierenden Variablen während der Zeitreihensimulation.

**8. thermohydraulic\_time\_series\_net**

Die Funktion thermohydraulic\_time\_series\_net führt eine zeitbasierte Simulation für das Heiznetzwerk durch. Sie berücksichtigt unterschiedliche Einstellungen für Steuerungen und Temperaturen.

* net: Das initialisierte Heiznetzwerk.
* yearly\_time\_steps: Zeitstufen für die Simulation.
* qext\_w\_profiles: Profile für externe Wärmezufuhr über die Zeit.
* start, end: Indizes für den Zeitraum der Simulation.
* supply\_temperature: Daten zur Versorgungstemperatur (optional).
* target\_temp: Die Zieltemperatur für die Return-Temperatur-Steuerungen.

**9. calculate\_results**

Die Funktion calculate\_results berechnet verschiedene Ergebnisse aus der Zeitreihensimulation, wie Massenfluss, Temperaturen und Druckdaten.

* net: Das Heiznetzwerk.
* net\_results: Die Ergebnisse der Simulation.

**10. save\_results\_csv**

Die Funktion save\_results\_csv speichert die Simulationsergebnisse in einer CSV-Datei.

* time\_steps: Zeitstufen der Simulation.
* qext\_kW: Externe Wärmezufuhr in kW.
* flow\_temp\_circ\_pump: Vorlauftemperatur der Umwälzpumpe.
* return\_temp\_circ\_pump: Rücklauftemperatur der Umwälzpumpe.
* filename: Der Dateiname für die CSV-Datei.

**11. import\_results\_csv**

Die Funktion import\_results\_csv importiert zuvor gespeicherte Simulationsergebnisse aus einer CSV-Datei.

* filename: Der Dateiname der CSV-Datei.

Diese Dokumentation bietet einen Überblick über die Funktionalität des Skripts "net\_simulation.py" und soll die Verwendung und Anpassung für Ihre speziellen Anforderungen erleichtern. Weitere detaillierte Informationen finden sich in den Funktionen selbst sowie in den Kommentaren im Skript.

„controllers.py“

The "controllers.py" script defines two custom controllers for heating network simulations using the pandapipes library. These controllers are designed to control pressure and return temperature within the heating network. Below, we will provide detailed explanations of each controller and its functionality.

## 1. WorstPointPressureController Class

The WorstPointPressureController class is responsible for controlling the pressure at the worst point in the heating network. It extends the BasicCtrl class from pandapower.control.basic\_controller.

### Constructor Parameters:

* net: The heating network model.
* worst\_point\_idx: The index of the worst point (heat exchanger) in the network.
* circ\_pump\_pressure\_idx: The index of the circulation pump pressure control element (default is 0).
* target\_dp\_min\_bar: The target minimum pressure difference in bar.
* tolerance: Tolerance for pressure control.
* proportional\_gain: Proportional gain for pressure control.
* \*\*kwargs: Additional keyword arguments for the base class constructor.

### Methods:

#### time\_step(self, net, time\_step)

* Resets the iteration counter and returns the current time step.

#### is\_converged(self, net)

* Checks if the pressure control has converged within tolerance.

#### control\_step(self, net)

* Implements the pressure control logic to adjust the pump pressure based on the target pressure difference.

## 2. ReturnTemperatureController Class

The ReturnTemperatureController class controls the return temperature at the heat exchanger's inlet. It extends the BasicCtrl class from pandapower.control.basic\_controller.

### Constructor Parameters:

* net: The heating network model.
* heat\_exchanger\_idx: The index of the heat exchanger in the network.
* target\_temperature: The target return temperature in degrees Celsius.
* tolerance: Tolerance for temperature control.
* min\_velocity: Minimum flow velocity for mass flow calculation.
* max\_velocity: Maximum flow velocity for mass flow calculation.
* \*\*kwargs: Additional keyword arguments for the base class constructor.

### Methods:

#### time\_step(self, net, time\_step)

* Resets the iteration counter and clears the previous temperature data.

#### update\_state(self, net)

* Updates the controller's state with the current inlet temperatures.

#### calculate\_mass\_flow\_limits(self, net)

* Calculates the minimum and maximum mass flow limits based on the heat exchanger's diameter and velocity constraints.

#### is\_converged(self, net)

* Checks if the return temperature control has converged within tolerance.

#### control\_step(self, net)

* Implements the return temperature control logic to adjust the mass flow rate at the heat exchanger's inlet.

### Attributes:

* initial\_target\_temperature: The initial target return temperature.
* min\_mass\_flow: The calculated minimum mass flow based on velocity constraints.
* max\_mass\_flow: The calculated maximum mass flow based on velocity constraints.
* at\_min\_mass\_flow\_limit: Indicates if the controller is at the minimum mass flow limit.
* at\_max\_mass\_flow\_limit: Indicates if the controller is at the maximum mass flow limit.

These custom controllers provide advanced control capabilities for heating network simulations using the pandapipes library, allowing users to optimize and fine-tune the performance of their heating systems.

„stanet\_import\_pandapipes.py“

The "stanet\_import\_pandapipe.py" script is used to import data from a specific CSV file representing a heating network configuration and convert it into a pandapipes network model. The script also performs coordinate transformation and incorporates additional data for heat exchangers. Below is a detailed explanation of the script's functionality.

## Importing Required Libraries

The script begins by importing necessary Python libraries:

* pandas (pd): Used for data manipulation.
* pandapipes (pp): Used for creating the heating network model.
* numpy (np): Used for numerical operations.
* pyproj.Transformer: Used for coordinate transformation.
* sys: Used to append a custom directory to the system path to import a module.

## Function: create\_net\_from\_stanet\_csv

This function is the main entry point for importing and converting the data from the CSV file into a pandapipes network model. It takes several parameters:

* file\_path: The path to the CSV file containing the heating network data.
* supply\_temperature: The supply temperature of the heating network.
* flow\_pressure\_pump: The pressure setpoint for flow circulation pumps.
* lift\_pressure\_pump: The pressure setpoint for lift circulation pumps.

### Importing and Processing CSV Data

1. The script reads the entire CSV file into a list of lines while ignoring any erroneous lines, as it reads the entire file as a single column.
2. The data is categorized into dictionaries based on different object types such as 'KNO' (nodes), 'LEI' (pipes), 'WAE' (heat exchangers), 'HEA' (heat sources), and 'ZAE' (consumers).
3. For each object type, the script creates a DataFrame, ensuring consistency in the number of columns between the header and data rows.

### Coordinate Transformation

1. The script defines a coordinate transformation from EPSG:31465 to EPSG:25833 using the pyproj.Transformer class.
2. It applies this transformation to the coordinates of nodes, pipes, and heat exchangers.

### Pandapipes Network Creation

1. A new empty pandapipes network model is created with "water" as the fluid type.
2. For each node in the CSV data, a junction is created in the pandapipes network, with properties such as pressure and temperature.
3. For each pipe, a pipe element is created in the pandapipes network, including properties like pipe length, diameter, and thermal conductivity.
4. For each heat source, a circulation pump with constant pressure is created in the pandapipes network, connecting the supply and return junctions.
5. For each heat exchanger, a flow control element and a heat exchanger element are created in the pandapipes network. The flow control element is placed between the heat exchanger's inlet and a new junction with intermediate coordinates.
6. The heat exchanger element is then placed between the intermediate junction and the heat exchanger's outlet.
7. The heat requirements for each heat exchanger are calculated using an external function (heat\_requirement\_BDEW.calculate) and incorporated into the model.

### Pandapipes Simulation

1. The script initiates a pandapipes network simulation using the pp.pipeflow function.
2. The resulting pandapipes network, time steps, and heat requirement data are returned for further analysis or simulation.

## Additional Notes

* The script includes error handling to catch exceptions during the data import process and prints any error messages encountered.
* The transformation between coordinate reference systems (CRS) assumes that the CSV data is in EPSG:31465, and it converts the coordinates to EPSG:25833.
* The script calculates heat requirements for heat exchangers, considering building profiles and consumption data from an external function (heat\_requirement\_BDEW.calculate). These values are used as parameters for heat exchangers in the pandapipes network.
* The code is partially commented to explain key sections of the script.

Overall, the "stanet\_import\_pandapipe.py" script facilitates the conversion of CSV-based heating network data into a pandapipes network model, enabling further analysis and simulations of the heating network's behavior.

heat\_generators

„heat\_generator\_classes.py“

Das Skript "heat\_generator\_classes.py" enthält Python-Funktionen und Klassen, die für die Berechnung und Analyse von verschiedenen Wärmeerzeugungssystemen verwendet werden. Hier ist eine inhaltliche Beschreibung der wichtigsten Funktionen und Klassen im Skript:

1. **annuität()**: Diese Funktion berechnet die Annuität für verschiedene Kostenkomponenten eines Wärmeerzeugungssystems über die Nutzungsdauer. Sie wird in verschiedenen Wärmeerzeugungsklassen für die Wirtschaftlichkeitsberechnung verwendet und berücksichtigt Investitionskosten, Energiebedarf, Energiekosten und weitere Parameter.
2. **HeatPump-Klasse**: Diese Klasse stellt eine allgemeine Wärmepumpe dar und enthält Methoden zur Berechnung des Leistungskoeffizienten (COP) und der Wärmegewinnungskosten (WGK) für Wärmepumpensysteme. Sie enthält auch eine Methode zur Berechnung der Abwärme, die von der Wärmepumpe erzeugt wird.
3. **RiverHeatPump-Klasse**: Diese Unterklasse von HeatPump ist spezialisiert auf Wärmepumpensysteme, die Flusswasser als Wärmequelle nutzen. Sie enthält Methoden zur Berechnung der Wärme- und Stromproduktion sowie der WGK für solche Systeme.
4. **WasteHeatPump-Klasse**: Diese Unterklasse von HeatPump ist für Abwärmepumpensysteme vorgesehen und berechnet die Leistung und Kosten solcher Systeme basierend auf Abwärmequellen.
5. **Geothermal-Klasse**: Diese Unterklasse von HeatPump ist auf geothermische Wärmequellen spezialisiert. Sie enthält Methoden zur Berechnung der geothermischen Leistung und Kosten sowie zur Bestimmung der optimalen Betriebsstunden.
6. **CHP-Klasse**: Diese Klasse repräsentiert kombinierte Wärme- und Stromerzeugungssysteme, wie Blockheizkraftwerke (BHKW) und Holzgas-BHKW. Sie berechnet die Leistung, den Brennstoffverbrauch und die WGK für solche Systeme.
7. **BiomassBoiler-Klasse**: Diese Klasse stellt Biomassekessel dar und berechnet deren Leistung, Brennstoffbedarf und WGK.
8. **GasBoiler-Klasse**: Diese Klasse repräsentiert Gasheizkessel und berechnet deren Leistung, Brennstoffbedarf und WGK.
9. **SolarThermal-Klasse**: Diese Klasse ist für Solarthermie-Systeme verantwortlich. Sie berechnet die Wärmegewinnungskosten und berücksichtigt die Art des Solarkollektors und die Speichergröße.

Zusammengefasst ermöglicht das Skript "heat\_generator\_classes.py" die Berechnung und Analyse verschiedener Wärmeerzeugungssysteme unter Berücksichtigung von Investitionskosten, Energieeffizienz, Wirtschaftlichkeit und Umweltauswirkungen. Es bietet eine flexible Grundlage zur Untersuchung und Bewertung verschiedener Technologien zur Wärmeerzeugung.

Die Funktionen calculate\_factors, Berechnung\_Erzeugermix und optimize\_mix sind Teil des Skripts und dienen dazu, verschiedene Berechnungen und Optimierungen für Wärmeerzeugungssysteme durchzuführen. Hier ist eine Beschreibung der Funktionen:

1. calculate\_factors(Kapitalzins, Preissteigerungsrate, Betrachtungszeitraum): Diese Funktion berechnet die Faktoren q, r und T auf Grundlage von Kapitalzins, Preissteigerungsrate und Betrachtungszeitraum. Diese Faktoren werden in verschiedenen Berechnungen verwendet und repräsentieren den Zinsfaktor, den Preissteigerungsfaktor und die Betrachtungsdauer. Sie werden aus den gegebenen Prozentwerten umgewandelt und zurückgegeben.
2. Berechnung\_Erzeugermix(tech\_order, initial\_data, calc1, calc2, TRY, COP\_data, Gaspreis, Strompreis, Holzpreis, BEW, variables=[], variables\_order=[], kapitalzins=5, preissteigerungsrate=3, betrachtungszeitraum=20): Diese Funktion berechnet verschiedene Kennzahlen und Ergebnisse für einen Mix von Wärmeerzeugungstechnologien. Sie verwendet eine Reihe von Parametern, darunter die Reihenfolge der Technologien (tech\_order), Anfangsdaten (initial\_data), Berechnungsfaktoren (calc1 und calc2), den Jahresenergieverbrauch (TRY), Leistungskoeffizientendaten für Wärmepumpen (COP\_data), Gaspreis, Strompreis, Holzpreis, BEW, und verschiedene optionale Variablen für die Anpassung der Technologien.

Die Funktion führt eine Vielzahl von Berechnungen durch, um den Wärme- und Stromertrag, die Wirtschaftlichkeit, die Emissionen und andere Kennzahlen für jeden Technologietyp im Mix zu bestimmen. Sie verwendet auch die zuvor definierten Faktoren q, r und T, um die Ergebnisse über den Betrachtungszeitraum zu diskontieren und an die Preisentwicklung anzupassen. Die berechneten Ergebnisse werden in einem Python Dictionary (general\_results) gespeichert und am Ende zurückgegeben.

1. optimize\_mix(tech\_order, initial\_data, calc1, calc2, TRY, COP\_data, Gaspreis, Strompreis, Holzpreis, BEW, kapitalzins, preissteigerungsrate, betrachtungszeitraum): Diese Funktion ist anscheinend für die Optimierung des besten Technologiemixes für die Wärmeerzeugung verantwortlich. Sie verwendet die gleichen Parameter wie Berechnung\_Erzeugermix, einschließlich der Liste der Technologien (tech\_order) und der anderen Eingangsdaten. Die genaue Implementierung der Optimierung ist in der gegebenen Funktion nicht ersichtlich, da dies von den spezifischen Anforderungen und Zielen abhängen würde.

Zusammenfassend ermöglichen diese Funktionen die umfassende Analyse und Optimierung von Wärmeerzeugungssystemen unter Berücksichtigung verschiedener Parameter wie Kapitalkosten, Preissteigerungsraten, Technologieoptionen und Betrachtungszeiträume. Sie helfen bei der Bewertung der Wirtschaftlichkeit und Umweltauswirkungen verschiedener Technologiemixe für die Wärmeerzeugung.

„Solarthermie.py“

1. **Einleitung:** Dieses Python-Skript wurde von Dipl.-Ing. (FH) Jonas Pfeiffer erstellt, um den Ertrag von Solarthermieanlagen in einem Wärmenetz zu berechnen. Es nutzt die Berechnungsgrundlage "ScenoCalc Fernwärme 2.0" von <https://www.scfw.de/>. Solarthermieanlagen wandeln Sonnenenergie in Wärme um, die dann zur Beheizung von Gebäuden oder zur Warmwasserbereitung verwendet wird.
2. **Import von Bibliotheken:** Das Skript beginnt mit dem Import von verschiedenen Python-Bibliotheken, die später im Code verwendet werden. Dazu gehören math, numpy, und datetime. Einige spezifische Funktionen aus heat\_generators.Solarstrahlung werden ebenfalls importiert. Diese Funktionen sind für die Berechnung der Solarstrahlung erforderlich.
3. **Die Hauptfunktion "Berechnung\_STA":** Diese Funktion ist das Herzstück des Skripts und wird verwendet, um den Ertrag der Solarthermieanlage zu berechnen. Sie erfordert eine Vielzahl von Eingabeparametern:
   * Bruttofläche\_STA: Die Bruttofläche der Solarthermieanlage.
   * VS: Das Volumen des Speichers.
   * Typ: Der Typ der Solarthermieanlage (Flachkollektor oder Vakuumröhrenkollektor).
   * Last\_L: Eine Liste von Lasten über einen bestimmten Zeitraum.
   * VLT\_L: Eine Liste von Vorlauftemperaturen über denselben Zeitraum.
   * RLT\_L: Eine Liste von Rücklauftemperaturen über denselben Zeitraum.
   * TRY: Ein Tupel von Listen, die Temperatur, Windgeschwindigkeit, Direktstrahlung und Globalstrahlung enthalten.
   * time\_steps: Eine Liste von Zeitstempeln, die den Zeitverlauf der Berechnungen darstellen.
   * calc1 und calc2: Indices, die den Zeitbereich innerhalb der TRY-Daten angeben, der für die Berechnung verwendet werden soll.
   * duration: Die Gesamtdauer der Berechnungen.
   * Weitere optionale Parameter wie Tsmax, Longitude, STD\_Longitude, Latitude, usw.

Die Funktion beginnt mit der Anpassung der Eingangsdaten an die gewünschte Zeitauflösung und berechnet dann den Ertrag der Solaranlage über den angegebenen Zeitraum. Dabei werden die Temperaturen, Strahlungswerte und andere Parameter berücksichtigt.

1. **Berechnung der Kollektorwirkungsgrade:** Abhängig vom gewählten Kollektortyp (Flachkollektor oder Vakuumröhrenkollektor) und den spezifischen Konstanten für diesen Typ werden die Kollektorwirkungsgrade berechnet. Diese Wirkungsgrade sind entscheidend für die Berechnung des Ertrags.
2. **Berechnung der Verluste:** Das Skript berechnet verschiedene Arten von Verlusten, darunter Verluste in Verbindungsleitungen, Verluste in internen Rohrleitungen und Speicherverluste. Diese Verluste werden während der Berechnung des Ertrags berücksichtigt.
3. **Berechnung des Ertrags und der Speicherladung:** Die Hauptberechnung erfolgt innerhalb einer Schleife, die durch die Zeitpunkte in time\_steps iteriert. In jeder Iteration werden die Kollektorleistung, die Verluste und die Speicherladung neu berechnet. Der Ertrag der Solaranlage wird berechnet und in der Ausgabe erfasst.
4. **Rückgabe der Ergebnisse:** Am Ende der Berechnung gibt die Funktion die Gesamtwärmemenge und die Wärmeleistung der Solaranlage in Form von NumPy-Arrays zurück.

Dieses Skript ist äußerst nützlich für Ingenieure und Planer, die Solarthermieanlagen in Wärmenetzen planen oder optimieren möchten. Es ermöglicht die detaillierte Berechnung des Ertrags unter Berücksichtigung einer Vielzahl von Faktoren und kann zur Simulation verschiedener Szenarien und zur Optimierung von Solarthermieanlagen verwendet werden. Es stellt sicher, dass die Solaranlage unter den gegebenen Bedingungen effizient arbeitet und hilft, den Beitrag zur nachhaltigen Energieerzeugung zu maximieren.

„Solarstrahlung.py“

Das Skript "Solarstrahlung.py" dient der Berechnung der Solarstrahlung auf einer schrägen Oberfläche. Es verwendet verschiedene meteorologische und geografische Parameter, um den Einfluss von Sonnenstrahlung auf Kollektoren oder Solarpaneele zu analysieren. Hier sind die Schlüsselaspekte des Skripts:

#### Import von Bibliotheken:

Das Skript beginnt mit dem Import der benötigten Bibliotheken, insbesondere NumPy, das für mathematische Berechnungen verwendet wird.

#### Grad-Radian-Konversion:

Es wird eine Konstante DEG\_TO\_RAD definiert, um Grad in Radian umzurechnen. Diese Konversion wird in späteren Berechnungen verwendet.

#### Funktion deg\_to\_rad:

Eine Funktion deg\_to\_rad wird definiert, um Grad in Radian umzurechnen. Dies wird in den Berechnungen für Winkel benötigt.

#### Hauptfunktion Berechnung\_Solarstrahlung:

Dies ist die Hauptfunktion des Skripts, die die Solarstrahlung auf der schrägen Oberfläche berechnet. Sie erfordert eine Vielzahl von Eingabeparametern, darunter:

* Globalstrahlung\_L: Die Globalstrahlung auf horizontaler Oberfläche (W/m²).
* D\_L: Die Direktstrahlung auf horizontaler Oberfläche (W/m²).
* Tag\_des\_Jahres\_L: Der Tag des Jahres.
* time\_steps: Eine Liste von Zeitstempeln, um den Zeitverlauf der Berechnungen darzustellen.
* Longitude: Die geografische Länge (°).
* STD\_Longitude: Die Zeitzone (°).
* Latitude: Die geografische Breite (°).
* Albedo: Die Albedo der Oberfläche (Reflexionskoeffizient).
* IAM\_W und IAM\_N: Dictionaries mit Einstrahlungs-Winkelabhängigkeitsdaten für die EW- und NS-Richtung.
* EWCaa: Der Azimutwinkel des Kollektors (°).
* CTA: Der Neigungswinkel des Kollektors (°).

Die Funktion führt dann eine Reihe von Berechnungen durch, um Folgendes zu ermitteln:

* Die Sonnenzeit unter Berücksichtigung der geografischen Länge und Zeitkorrektur.
* Die Sonnendeklination und den Stundenwinkel der Sonne.
* Den Sonnenzenitwinkel und Azimutwinkel der Sonne.
* Den Einfallswinkel der Sonnenstrahlung auf den Kollektor.
* Die Einstrahlungsfaktoren IAM\_EW und IAM\_NS unter Berücksichtigung des Einfallswinkels.
* Das Verhältnis der Strahlungsintensität auf dem geneigten Kollektor zur horizontalen Oberfläche.
* Die Direktstrahlung auf horizontaler Oberfläche und die diffuse Strahlung.
* Den atmosphärischen Diffusanteil Ai.
* Die Gesamtstrahlung auf der schrägen Oberfläche GT\_H\_Gk.
* Die direkte Strahlung auf der schrägen Oberfläche GbT.
* Die diffuse Strahlung auf der schrägen Oberfläche GdT\_H\_Dk.
* Das Produkt der Einstrahlungsfaktoren K\_beam.

#### Zusammenfassung der Ergebnisse:

Die Funktion gibt schließlich vier wichtige Ergebnisse zurück:

1. GT\_H\_Gk: Die Gesamtstrahlung auf der schrägen Oberfläche, einschließlich des direkten und diffusen Beitrags sowie des durch Albedo reflektierten Beitrags.
2. K\_beam: Das Produkt der Einstrahlungsfaktoren für die EW- und NS-Richtung, das den Einfluss des Einfallswinkels auf die Strahlung beschreibt.
3. GbT: Die Direktstrahlung auf der schrägen Oberfläche.
4. GdT\_H\_Dk: Die diffuse Strahlung auf der schrägen Oberfläche.

### Verwendungszweck:

Dieses Skript ist nützlich für Ingenieure, die sich mit der Leistungsprognose von Solarthermieanlagen oder Photovoltaiksystemen befassen. Es ermöglicht eine detaillierte Analyse der Strahlungseinflüsse auf schräge Kollektoren, um den Ertrag genauer zu bestimmen und die Anlagenausrichtung zu optimieren.