Greek Boundary Nodes

Introduction

This pdf is created to explain the scirpt named Boundary_diagrams.py in path I:\Konstantinos Sidiropoulos\Documentation\Greek Boundary Nodes.

Script loads 24 CGM's in UCTE format that correspond to the same day. Reads the boundary lines of the European network and keeps only the Greek boundary nodes with information about **Current (A)**, **Active Power(MW)** and **Reactive Power (MVAr)** that flows through the boundary line to the Greek boundary Node. Creates a final file in .xlsx form with a new column that gives the **timestamps**. File is sorted by id and timestamp in order user to see for each Greek boundary node the Power flows from 0030 to 2330 (HH:MM) without making any changes to the file. Automatically 3 diagrams for each Greek boundary node are created. X-axis contains the timestamps and Y-axis the Current (A), active power (MW) or reactive power (MVar) depending on the diagram.

The code has been tested and works for CGM's in UCTE format.

It was used in computer 10.91.100.15 with the following installations:

- Python version 3.12.0
- Pypowsybl library: User can simply install released versions of pypowsybl from PyPI using pip:

```
pip install pypowsybl
```

- Visual Studio Code: VSCodeUserSetup-x64-1.91.0.exe

Note: File_type was always FO = Day Ahead (D-1) or 2D = Two Days Ahead (D-2)
Note: ALL UCTE'S had name structure ucte_filename = {Date}_{hour}_{File_type}_{country_code}{number}.{format}

1. Libraries and paths

Script:

➢ Imports the following libraries:

```
1. import matplotlib.pyplot as plt
2. import pypowsybl.network as pp
3. import pypowsybl.loadflow as lf
4. import pandas as pd
5. import pypowsybl.report as nf
6. import math
7. import os
```

Figure 1: Libraries import

import matplotlib.pyplot as plt : Creates and customizes current, active and reactive power plots.

import pypowsybl.network as pp: Loads and interacts with the power grid model (CGM) in UCTE format.

import pypowsybl.loadflow as lf : Allows user to configure and run **AC load flow analysis** on the network model that is loaded.

import pandas as pd: Reads excel files, converts data types (**pd.to_numeric()**) and writes dataframes to excel.

import pypowsybl.report as nf : Creates a comprehensive report about the LoadFlow execution, available in the terminal.

import os : path manipulation, checking file existence, listing files in a directory, and deleting files.

import math : Creates dynamic limits in Y-axis of the plots based in .xlsx maximum and minimum values of current (A), active (MW) and reactive power(MVar) columns.

- **def get_user_inputs():**

Provides the possibility for user to specify the path where his UCTE files are (**ucte_folder**), the path where the .xlsx file will be saved (**output_folder**), and the path where his diagrams wiil be saved (**output_folder1**). Creates a variable where user enters the date of his UCTE files (**Date**) and variables where user specifies the time horizon (**File_type**), country code (**country_code**) and the format that is loaded (**format**). **Numbers** variable corresponds to the different versions of UCTE's files.

```
1. def get_user_inputs():
2.     """
3.     Get user inputs for folder paths, date, file type, country code, and numbers.
4.     """
5.     ucte_folder = input("Enter the path for UCTE folder (e.g.,
'C:/Users/k.sidiropoulos/Downloads/CGM_greek_nodes'): ").strip()
6.     output_folder = input("Enter the path for Excel output folder (e.g.,
'C:/Users/k.sidiropoulos/Downloads/CGM_greek_nodes/Daily_excel'): ").strip()
7.     output_folder1 = input("Enter the path for diagrams output folder (e.g.,
'C:/Users/k.sidiropoulos/Downloads/CGM_greek_nodes/Daily_excel/diagrams'): ").strip()
8.     Date = input("Enter the date in YYYYMMDD format (e.g., '20240717'): ").strip()
9.     File_type = input("Enter the file type (e.g., 'FO3'): ").strip()
10.    country_code = input("Enter the country code (e.g., 'UX'): ").strip()
11.    format = input("Enter the file format (e.g., 'UCT'): ").strip()
12.    numbers = input("Enter the range of numbers (e.g., '0-20'): ").strip()
13.
14.    # Convert 'numbers' input to a range
15.    try:
16.        start, end = map(int, numbers.split('-'))
17.        numbers = range(start, end + 1)
18.    except ValueError:
19.        print("Invalid range input. Using default range 0-20.")
20.        numbers = range(0, 21)
21.
22.    # Return user inputs
23.    return ucte_folder, output_folder, output_folder1, Date, File_type, country_code, format,
numbers
24.
```

Figure 2: Paths and UCTE information

2. Def main()

Script takes a list of variables with specified values from user with get_user_inputs().

Creates a loop to iterate through the list of hours variable and an empty dataframe where the results are going to be saved (**combined**).

Initializes the **highest_number** and **selected_ucte_path** variables in order to create a second inner loop that finds the more recent version of the same hour UCTE file.

Inside the loop there is a variable that constructs the UCTE file name dynamically:

ucte_filename = f'{Date}_{hour}_{File_type}_{country_code}{number}.{format}'

A variable that specifies the path to the UCTE file: **ucte_path = os.path.join(ucte_folder, ucte_filename)**

An **If** function that checks if the path exists and upgrades highest number and selected_ucte_path.

After checking versions from 0 to 20 script checks if there is a valid path and prints the highest number of version.

```
 1. def main():
 2.     # Get user inputs
 3.     ucte_folder, output_folder, output_folder1, Date, File_type, country_code, format, numbers =
get_user_inputs()
 4.     #Timestamps
 5.     hours = ['0030', '0130', '0230', '0330', '0430', '0530', '0630', '0730', '0830', '0930',
'1030', '1130',
 6.          '1230', '1330', '1430', '1530', '1630', '1730', '1830', '1930', '2030', '2130', '2230',
'2330']
 7.     combined = pd.DataFrame()  # Initialize combined DataFrame
 8.     #Iterate through each timestamp
 9.     for hour in hours:
10.         highest_number = -1
11.         selected_ucte_path = None
12.         #Iterate through numbers to check for highest UCTE version
13.         for number in numbers:
14.             # Construct the UCTE filename
15.             ucte_filename = f'{Date}_{hour}_{File_type}_{country_code}{number}.{format}'
16.             ucte_path = os.path.join(ucte_folder, ucte_filename)
17.
18.             if os.path.exists(ucte_path):
19.                 if number > highest_number:
20.                     highest_number = number
21.                     selected_ucte_path = ucte_path
22.
```

Figure 3: Loops

Loads the network and performs AC LoadFlow with load_and_run_loadflow(selected_ucte_path). Extracts Greek boundary nodes with their corresponding flows using **extract_boundary_nodes()**. **Combined** variable saves the dataframes for each UCTE file corresponding to Greek boundary nodes. If UCTE file does not exist error message is printed in the console and loop proceeds to the next hour.

```
1. if selected_ucte_path:
2.             print(f'Highest number version was: {highest_number}')
3.             network = load_and_run_loadflow(selected_ucte_path)
4.             filtered_data = extract_boundary_nodes(network, hour)
5.             combined = pd.concat([combined, filtered_data], ignore_index=True)
6.         else:
7.             print(f'No valid version found for hour: {hour}. Skipping this hour.')
8.             continue  # Skip this hour and move on to the next one
9.
```

Figure 4: Combined dataframe()

If combined variable contains dataframes, **output_file** variable is created containing the name and path for the constructed excel. **Save_combined_data()** creates the new excel and **generate_plots(hours, output_file, output_folder1)** the current, active and reactive power figures for each boundary node.

For empty dataframe an error message is printed.

```
    if not combined.empty:
        output_file = os.path.join(output_folder, f'GREEK_BOUNDARY_NODES_{Date}.xlsx')
        save_combined_data(combined, output_file)
        generate_plots(hours, output_file, output_folder1)
        print("Current, active power, and reactive power plotting completed. Files are saved to the output folder.")
    else:
        print("No valid data was processed. No output generated.")

```

Figure 5: Boundary nodes extraction process

3. Def load_and_run_loadflow(selected_ucte_path):

Script loads the most recent UCTE file version of a specific hour (**pp.load(selected_ucte_path)**) and defines the parameters for performing AC LoadFlow (**lf.Parameters()**).

NOTE: All parameters for performing AC LoadFlow available in: https://powsybl.readthedocs.io/projects/pypowsybl/en/stable/reference/loadflow/parameters.html#pypowsybl.loadflow.Parameters

NOTE: Provider_parameters available in the excel named provider_parameters.xlsx in I:\Konstantinos Sidiropoulos\Documentation\Greek Boundary Nodes. User can add the follow orders to get anytime the provider parameters:

```
It =lf.get_provider_parameters() # OPTIONAL --> GET A LIST OF AVAILABLE provider PARAMETERS TO EXECUTE AC LOADFLOW
        It.to_excel('provider_parameters.xlsx')
```

Figure 6:Provider parameters

nf.reporter() and **print(str(reporter))** functions, provide in the terminal important information about the LoadFlow execution (number of iterations, slack bus id, PV to PQ nodes). Script returns the loadflow execution in the **network** variable.

```
def load_and_run_loadflow(selected_ucte_path):
        """
          Load the network from the UCTE file and run the AC load flow.
        """
        #Load the UCTE
        network = pp.load(selected_ucte_path)
        reporter = nf.Reporter()
        #PERFORMING AC LOADFLOW, specifying the parameters
        p = lf.Parameters(
                distributed_slack=False ,
                transformer_voltage_control_on=False ,
                phase_shifter_regulation_on=True ,
                shunt_compensator_voltage_control_on=True ,
                voltage_init_mode=None,
                use_reactive_limits=None,
                twt_split_shunt_admittance=None,
                read_slack_bus=None,
```

```
                write_slack_bus=None,
                balance_type=None,
                dc_use_transformer_ratio=None,
                countries_to_balance=None,
                connected_component_mode=None,
                dc_power_factor=None,
                provider_parameters={
            'maxOuterLoopIterations' : str(30) ,
            'lowImpedanceBranchMode': 'REPLACE_BY_MIN_IMPEDANCE_LINE'
                })
        lf.run_ac(network , parameters = p , reporter= reporter) # User can use report_node =
reporter instead reporter = reporter.
        #Print the provider parameters
        print(str(reporter))

        return network

```

Figure 7: AC LoadFlow

4. def extract_boundary_nodes(network, hour):

X_nodes = **network.get_dangling_lines(attributes = ['bus_breaker_bus_id', 'i' , 'p' , 'q'])** :

X_nodes variable retrieves all the boundary lines (X-node – real boundary Node) from the network. Extracts certain attributes such as I, P, Q from the real boundary node side and the id of the real boundary node ('bus_breaker_bus_id' ,'i' , 'p' , 'q').

Script adds a new index column (**.reset_index()**) which renames to 'id'. This way provides the id's of the boundary lines.

Converts id's to string and replaces ' ' with '_' using the follow function:

X_nodes['id'] = X_nodes['id'].astype(str).str.replace(' ', '_').

Renames the columns (**.rename()**).

Filters the data to contain only those boundary lines where their real boundary node string name starts with 'G': **X_nodes[X_nodes['bus_breaker_id'].str.startswith(('G'))].copy()**. This way script isolates the Greek boundary nodes with their I, P, Q attributes from the boundary line. Results are saved in **filtered** variable.

Removes whitespaces (.str.strip()) and adds a new column named 'Timestamp' which contains the value of the current **hour** variable (**filtered['Timestamp'] = hour**).

Uses **convert_to_numeric()** and **adjust_prefixes()** to give numeric values to all the specified columns (except id's) and prefix to current.

Function returns the new structured dataframe to **filtered** variable.

```
def extract_boundary_nodes(network, hour):
    """
    Extract X-Nodes (boundary nodes) from the network and filter Greek boundary nodes.
    """
    #Get the X-nodes of the UCTE
    X_nodes = network.get_dangling_lines(attributes = ['bus_breaker_bus_id' ,'i' , 'p' , 'q'
])  # TAKE ALL THE BOUNDARY LINES OF THE CGM
    X_nodes.reset_index(inplace=True)
```

```
    X_nodes.rename(columns={'index': 'id'}, inplace=True)
    X_nodes['id'] = X_nodes['id'].astype(str).str.replace(' ', '_')
    X_nodes.rename(columns={'i': 'I', 'p': 'P', 'q': 'Q' , 'bus_breaker_bus_id':
'bus_breaker_id' }, inplace=True)
    # we study THE P , Q ,I FROM THE GREEK BOUNDARY NODES
    filtered = X_nodes[X_nodes['bus_breaker_id'].str.startswith(('G')) ].copy()
    filtered.columns = filtered.columns.str.strip()
    filtered['Timestamp'] = hour
    #Give prefix to current and numeric form
    filtered = adjust_prefixes(filtered)
    filtered = convert_to_numeric(filtered)
    return filtered

```

Figure 8: Greek boundary nodes

5. def adjust_prefixes(df) and def convert_to_numeric(df)

Script creates a function named **adjust_prefixes()** for giving prefix to the current attribute (Current magnitude).

def has_minus(value): checks whether a given value contains a minus sign by converting the value to a string and checking if '-' sign is present. Returns True/False.

df['P_minus'] = **df['P'].apply(lambda x: '-' if has_minus(x) else '')** : Adds new column named 'P_minus' that stores '-' or ' ' using lambda function after True/False import of has_minus(x).

df['Q_minus'] **= df['Q'].apply(lambda x: '-' if has_minus(x) else '')**: Adds new column named 'Q_minus' that stores '-' or ' ' using lambda function after True/False import of has_minus(x).

df['I'] = **df.apply()**: Creates a lambda function on each row of the dataframe.

Current has always same prefix with active power and if active power is zero then has the prefix of the reactive power. Script drops the temporary columns and returns the modified dataframe.

```
def adjust_prefixes(df):

    def has_minus(value):
        return '-' in str(value)

    df['P_minus'] = df['P'].apply(lambda x: '-' if has_minus(x) else '')
    df['Q_minus'] = df['Q'].apply(lambda x: '-' if has_minus(x) else '')

    df['I'] = df.apply(
        lambda row: f"{row['P_minus']}{row['I']}" if row['P'] != 0 and row['P_minus'] else
                    f"{row['Q_minus']}{row['I']}" if row['P'] == 0 and row['Q_minus'] else
                            row['I'],
        axis=1
     )

    df.drop(columns=['P_minus', 'Q_minus'], inplace=True)
    return df
```

Figure 9: Adjust_prefixes()

Script creates a function **convert_to_numeric()** to convert specific columns of a DataFrame to numeric data types (such as integers or floats). This way we ensure modified Current column will have numeric values. Script converts all the columns to numeric data types for full coverage.

```
1. def convert_to_numeric(df):
2.             df['P'] = pd.to_numeric(df['P'], errors='coerce')
3.             df['Q'] = pd.to_numeric(df['Q'], errors='coerce')
4.             df['I'] = pd.to_numeric(df['I'], errors='coerce')
5.             df['Timestamp'] = pd.to_numeric(df['Timestamp'], errors='coerce')
6.             return df
```

Figure 10: Convert_to_numeric()

6. Def save_combined_data(combined, output_file):

Script ensures timestamps correspond to hours variable list, sorts the data and saves them in an excel named **'GREEK_BOUNDARY_NODES_{Date}.xlsx'**

combined['Timestamp'] = combined['Timestamp'].apply(lambda x: f'{int(x):04d}'): Script ensures every row has 4 digit timestamp string value (replaces '30' to '0030' , '130' to '0130' etc.)

combined.sort_values(by=['bus_breaker_id' , 'Timestamp'], inplace = True): Sorts the data by their boundary bus breaker id and the timestamp (A to Z, '0030' to '2330').

Creates **output_file** variable that contains the path (**output_folder**) and the name of the combined dataframe (**os.path.join()**).

Output_file is saved in .xlsx form (**.to_excel()**). It will be processed further for plots creation.

```
 1. def save_combined_data(combined, output_file):
 2.     """
 3.     Save the combined DataFrame to an Excel file.
 4.     """
 5.     #Ensure timestamp is 4 digit
 6.     combined['Timestamp'] = combined['Timestamp'].apply(lambda x: f'{int(x):04d}')
 7.     #Sort data and save to excel
 8.     combined.sort_values(by=['bus_breaker_id', 'Timestamp'], inplace=True)
 9.     combined.to_excel(output_file, index=False)
10.
```

Figure 11: Daily boundary flows

7. def generate_plots(hours, output_file , output_folder):

pd.read_excel(output_file, dtype={'Timestamp': str}: Script creates **combine** variable that reads the output_file in excel form and ensures that 'Timestamp' values are strings in order not to lose the leading zeros (ex. '0030' to '30'). If file does not exist prints an error message.

combine['Timestamp'].str.zfill(4): Ensures that all timestamps are zero-padded to four digits.

time_point_to_index = {time: idx for idx, time in enumerate(hours)}: Variable maps each string time point to numeric index because plots require numeric values in their X-axis ticks.

combine['Timestamp_index'] = combine['Timestamp'].map(time_point_to_index): Converts timestamps in 'Timestamp' column into their corresponding numeric indices. Saves results in a new 'Timestamp_index' column.

bus_breaker_ids = combine['bus_breaker_id'].unique(): Saves in **bus_breaker_ids** variable unique bus_breaker_id's from dataframe, which will use to title the diagrams and create a loop.

Script creates an **if** function to check for invalid time points and if there are, prints error messages.

```
def generate_plots(hours, output_file , output_folder):
    """
    Generate and save plots for current, active power, and reactive power for each bus breaker ID.
    """
    #Read the Greek X-nodes
    try:
       combine = pd.read_excel(output_file, dtype={'Timestamp': str})
    except FileNotFoundError:
        print(f"Error: {output_file} not found.")
        exit(1)
    # Ensure that all timestamps have a consistent 4-character format (e.g., '0030', '1130', etc.)
    combine['Timestamp'] = combine['Timestamp'].str.zfill(4)
    # Create a mapping of time points to numeric indices (0 for '0030', 1 for '0130', etc.)
    time_point_to_index = {time: idx for idx, time in enumerate(hours)}
    # Convert the 'Timestamp' column to the corresponding numeric index using the mapping
    combine['Timestamp_index'] = combine['Timestamp'].map(time_point_to_index)

    # Get unique bus breaker IDs
    bus_breaker_ids = combine['bus_breaker_id'].unique()

    # Check if there are any unmapped timestamps
    if combine['Timestamp_index'].isna().any():
        print("Warning: Some timestamps could not be mapped. Please check your data.")
        print(combine[combine['Timestamp_index'].isna()]['Timestamp'].unique())

```

Figure 12: Final data structure

Script generates plots for current, active and reactive power for each one of the unique id's of the boundary nodes (bus_breaker_id's).

for bus_breaker_id in bus_breaker_ids: Loop iterates through each unique bus_breaker_id.

filtered_df = combine[combine['bus_breaker_id'] == bus_breaker_id]: Creates a variable inside of the loop to keep only the rows with specific bus_breaker_id.

boundary_line_id = filtered_df['id'].iloc[0]: Variable keeps the first value of filtered_df 'id' column that corresponds to the boundary line id of the bus_breaker_id. This Is useful for titling the PNG file.

Creates a variable named **ticks** that specifies the number of values that plot will show in Y-axis.

Calls three times **plot_data()** function with different each time attributes specification for I, P, Q plot creation.

```
    #Iterate through each bus_breaker_id to create the plots
    for bus_breaker_id in bus_breaker_ids:
        filtered_df = combine[combine['bus_breaker_id'] == bus_breaker_id]
        boundary_line_id = filtered_df['id'].iloc[0]  # Assume each bus_breaker_id has a single ID for titling the plots
        ticks = 15

        # Pass boundary_line_id into plot_data for file naming
       plot_data(filtered_df, combine,'I', bus_breaker_id, 'Current (A)', f'Current Plot for {bus_breaker_id}',output_folder, hours, time_point_to_index, f'{boundary_line_id}_current_plot', ticks)
```

```
09.
10.         plot_data(filtered_df, combine, 'P', bus_breaker_id, 'Active Power (MW)', f'Active Power
Plot for {bus_breaker_id}',output_folder, hours, time_point_to_index,
f'{boundary_line_id}_active_power_plot' , ticks)
11.
12.         plot_data(filtered_df, combine ,'Q', bus_breaker_id, 'Reactive Power (MVAr)', f'Reactive
Power Plot for {bus_breaker_id}',output_folder, hours, time_point_to_index,
f'{boundary_line_id}_reactive_power_plot' , ticks)
```

Figure 13: Plots creation for each unique bus_breaker_id().

8. **def plot_data(filtered_df, combine, column, bus_breaker_id, ylabel, title, output_folder, hours, time_point_to_index, file_suffix , ticks):**

Script sets the size of the plots (**plt.figure()**) and proceeds only if there are no missing data in predefined 'Timestamp_index' column (**if not filtered_df['Timestamp_index'].isna().all()**).

Variables **max_val**, **min_val** save the maximum and minimum value of current of specified bus_breaker_id and are rounded **up** to the nearest hundrend (Maximum current value) and **down** to the nearest hundrend (Minimum current value) using:

math.ceil(combine['column'].max() / 100) * 100: Rounds up to the nearest hundrend.

math.floor(combine['column'].min() / 100) * 100: Rounds down to the nearest hundrend.

value_range = max_val - min_val : Calculates the range of column values.

step = calculate_step_size(value_range , ticks): Calculation of step between two values in Y-axis.

max_limit, min_limit = calculate_limits(max_val, min_val, step): Calculation of Max/Min values that will appear in Y-axis.

Script creates the scatter plot **(plt.scatter()**) where **X-axis** is **filtered_df['Timestamp_index']** and **Y-axis** is **filtered_df['column']** (' ', replaced by 'I', 'P' or 'Q' depending on the diagram). Each point in scatter plot is styled with black dots (**color = 'black'**). Size and edge color are defined by the attributes **s**, **edgecolor** inside plt.scatter() function.

Plot, X-axis, Y-axis titles are defined using **plt.title()**, **plt.xlabel()**, **plt.ylabel()**.

plt.xticks(ticks=list(time_point_to_index.values()), labels=time_points, rotation=90): X-axis ticks are set to be the numeric values of the Timestamp_index. Labels will be vertical (**rotation**) and display the actual time_points.

Range of the values of Y-axis (Max/Min) and increasing/decreasing step for values that tick marks are defined by **plt.ylim(min_limit , max_limit)** and **plt.yticks(range(min_limit, max_limit+1, step))**.

Script ensures points in scatter plot are visually in front of grid lines with **plt.grid()**, and adds a horizontal black line in y=0 with **plt.axhline()**.

Avoids overlapping of the labels with **plt.tight_layout()**, saves figure (**plt.savefig()**) in **.png** file and closes the figure (**plt.close()**) in order script to continue providing figures for Current, active and reactive power in all bus_break_id's.

```
 1. def plot_data(filtered_df, combine, column, bus_breaker_id, ylabel, title, output_folder, hours,
time_point_to_index, file_suffix , ticks):
 2.     """
 3.     Plot data (Current, Active Power, or Reactive Power) and save the plot.
 4.     """
 5.     plt.figure(figsize=(10, 6)) # figure with specified size
 6.
 7.     if not filtered_df['Timestamp_index'].isna().all():
 8.         # Get the maximum and minimum values of the column (e.g., I, P, Q) and round to the
nearest hundred
 9.         max_val = math.ceil(combine[column].max() / 100) * 100
10.         min_val = math.floor(combine[column].min() / 100) * 100
11.         #Range of values
12.         value_range = max_val - min_val
13.         # Determine the step size for y-axis ticks based on the value range
14.         step = calculate_step_size(value_range, ticks)
15.         # Calculate the limits for the y-axis based on the step size
16.         max_limit, min_limit = calculate_limits(max_val, min_val, step)
17.
18.         # Plot the values using a scatter plot with black dots
19.         plt.scatter(filtered_df['Timestamp_index'], filtered_df[column], color='black', s=50,
edgecolor='black', zorder=5)
20.         # Plot title and axis labels
21.         plt.title(title)
22.         plt.xlabel('Timestamp')
23.         plt.ylabel(ylabel)
24.         # Define the x-axis ticks as the mapped time points and rotate the labels for better
visibility
25.         plt.xticks(ticks=list(time_point_to_index.values()), labels=hours, rotation=90)
26.         # Set the y-axis limits and add ticks based on the calculated step size
27.         plt.ylim(min_limit, max_limit)
28.         plt.yticks(range(min_limit, max_limit+1, step))
29.         # Add grid lines behind the plot and a horizontal line at y=0 for reference
30.         plt.grid(True, zorder=0)
31.         plt.axhline(y=0, color='black', linestyle='--', linewidth=1)
32.         # Automatically adjust the layout to prevent overlapping elements
33.         plt.tight_layout()
34.         # Save the plot to the specified folder with the corresponding bus breaker ID and file
suffix
35.         plot_path = os.path.join(output_folder, f'{bus_breaker_id}_{file_suffix}.png')
36.         plt.savefig(plot_path)
37.         plt.close()
38.
```

Figure 14: plot_data

9. Def calculate_step_size(range, ticks), calculate_limits(max, min, step) and main()

Script creates two functions for calculation of the step and of the limits (Max/Min values) in Y-axis.

def calculate_step_size(range, ticks): **range** variable corresponds to the difference between max and min value of a specified column. **Ticks** parameter determines the step size.

raw_step = range / (ticks * 100) and decimal_part = raw_step - math.floor(raw_step): Calculates the step, divides by 100 and keeps the decimal part.

if **decimal_part** > 0.50 then **step = math.ceil(raw_step) * 100** : Script rounds up raw_step to the next hundred and keeps this value as the step.

else: step = math.floor(raw_step) * 100: Scipts rounds down raw_step to the below hundred.

if step == 0: For small ranges (less than 700) step with the above orders is zero. In this case script sets step to 50.

Script ensures with this function that the step size is appropriate for covering the range while keeping the plot readable and well-distributed.

```
1. def calculate_step_size(range, ticks):
2.     """
3.     Calculate the step size for the plot's y-axis based on the range.
4.     """
5.     if ticks <= 0:
6.         raise ValueError("Number of ticks should be greater than zero.")
7.
8.     # Divide the range by the total number of ticks to get the raw step size
9.     raw_step = range / (ticks * 100)
10.
11.    # Determine the decimal portion of the step size to decide between rounding up or downn
12.    decimal_part = raw_step - math.floor(raw_step)
13.
14.    # If more than 50% of the next step, round up, otherwise round down
15.    if decimal_part > 0.50:
16.        step = math.ceil(raw_step) * 100
17.    else:
18.        step = math.floor(raw_step) * 100
19.    # define a minimum step size if necessary
20.    if step == 0 :
21.        step = 50
22.
23.    return step
24.
```

Figure 15: Calculate_step_size()

def calculate_limits(max, min, step): Calculates Max/Min values that will occur in the final plots.

max_limit = step * math.ceil(max / step): Divides max value with step and rounds up to the higher integer. Multiplies result with step and value is saved in max_limit variable.

min_limit = step * math.floor(min / step): Divides min value with step and and rounds down the to the below integer. Multiplies result with step and value is saved in mix_limit variable.

```
1. def calculate_limits(max, min, step):
2.     max_limit = step * math.ceil(max / step)
3.     min_limit = step * math.floor(min / step)
4.     return max_limit, min_limit
```

Figure 16: Calculate_limits()

Final, script calls main() for code execution.

```
1. if __name__ == "__main__":
2.     main()
3.
```

Figure 17: Code execution