DailyLoadFlow by OPENLF

Introduction

This pdf is created to explain the scirpt named dailyloadflow.py in path I:\Konstantinos Sidiropoulos\Documentation\LoadFlow_reports.

The specified code creates a comprehensive LoadFlow analysis and saves the output in .xlsx form similar to Unicorns LoadFlows reports. It uses functions of Pypowsybl library alongside with libraries of python and creates a loop where a user can have a daily or hourly LoadFlow analysis.

The code has been tested and works for IGMs and CGMs 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)

All UCTE's had name structure ucte_filename = {Date}_{hour}_{File_type}_{country_code}{number}.{format}

1. Libraries and paths

The script:

➢ Imports the following libraries:

```
1. import pypowsybl.network as pp
2. import pypowsybl.loadflow as lf
3. import pandas as pd
4. import os
5. import pypowsybl.report as nf
6. import logging
```

Figure 1: Libraries import

pypowsybl.network: main data structure of pypowsybl. It contains all the data of a power network.

pypowsybl.loadflow: Used in order to run load flows on networks.

pandas: Used to process and merge electrical network data, enabling the transformation of load flow results into structured tables that can be exported to Excel for detailed analysis.

os: used for interacting with the file system.

pypowsybl.report: Used to generate and manage detailed reports on the execution of load flow analysis.

Logging: Shows messages in the console.

- **Def get_user_inputs():** Provides the possibility for user to specify his UCTEs folder path (**ucte_folder**) and the path he wants his LoadFlow reports to be saved (**output_folder**). User also can define which hours he wants to make LoadFlow analysis (**hours**) and specify the correct **Date**, **File_type** and **country_code** UCTE information to the console after running the script. If user specifies correctly the variables the script will continue processing. If not, it will finalize without error message.

```
1. def get_user_inputs():
2.     # Get the folder paths
3.     ucte_folder = input("Enter the path to your UCTE files folder: ")
4.     output_folder = input("Enter the path where output reports should be saved: ")
5.
6.     # Get the date, file type, country code, and format
7.     date = input("Enter the date (e.g., 20240717): ")
8.     file_type = input("Enter the file type (e.g., FO3): ")
9.     country_code = input("Enter the country code (e.g., GR): ")
10.    format = input("Enter the format (e.g., UCT): ")
11.
12.    # Define hours or ask user for specific hours
13.    hours = input("Enter the hours (comma-separated ex. 0030,0130 ... , or leave blank for
default 0030-2330): ")
14.    if not hours:
15.        hours = ['0030', '0130', '0230', '0330', '0430', '0530', '0630', '0730', '0830', '0930',
'1030', '1130','1230', '1330', '1430', '1530', '1630', '1730', '1830', '1930', '2030', '2130',
'2230', '2330']
17.    else:
18.        hours = hours.split(',')
19.
20.    # Get the number range or use default
21.    numbers = input("Enter the version numbers (comma-separated, or leave blank for default 0-
20): ")
22.    if not numbers:
23.        numbers = range(0, 21)
24.    else:
25.        numbers = [int(x) for x in numbers.split(',')]
26.
27.    return ucte_folder, output_folder, date, file_type, country_code, format, hours, numbers
28.
```

Figure 2: Paths and UCTE information

- Script creates a function named **adjust_prefixes()** for giving prefix to the current attribute. The concept is current to have always same prefix with active power and if active power is zero then to have the prefix of the reactive power. This way can identify which way the current is heading to between the two sides of the line.

Current is saved in string form so with **convert_to_numeric()** saves the current including the prefix in numeric form. Process_df() compines those two functions.

```
1. def adjust_prefixes(df):
2.     def has_minus(value):
3.         return '-' in str(value)
4.
5.     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

#Convert I values to numeric
def convert_to_numeric(df):
    df['I'] = pd.to_numeric(df['I'], errors='coerce')
    return df

#Uses above functions at once
def process_df(df):
    df = adjust_prefixes(df)
    df = convert_to_numeric(df)
    return df

```

Figure 3: adjust_prefixes(df), convert_to_numeric(df) and process_df(df) functions

2. AC loadflow execution

def process_network_files_from_user_inputs(): function gathers user inputs (such as folder paths, date, file type, country code, hours, and version numbers) and uses these inputs to process the network files through the process_network_files function.

def process_network_files(date, hours, numbers, file_type, country_code, format, ucte_folder, output_folder): function iterates through the specified hours, finds the highest version UCTE file for each hour through find_highest_version_file function and then processes and saves the network file using the identified path and parameters. If filename in file path does not exist, error message is printed in console.

def find_highest_version_file(date, hour, numbers, file_type, country_code, format, ucte_folder): function iterates through possible version numbers to identify the highest version of a UCTE file for a given hour, checking if each file exists, and returns the highest version number and its corresponding file path.

```
def process_network_files_from_user_inputs():
    # Get user inputs
    ucte_folder, output_folder, date, file_type, country_code, format, hours, numbers =
get_user_inputs()

    # Process network files using user-defined inputs
    process_network_files(date, hours, numbers, file_type, country_code, format, ucte_folder,
output_folder)

def process_network_files(date, hours, numbers, file_type, country_code, format, ucte_folder,
output_folder):
    for hour in hours:
        highest_number, selected_ucte_path = find_highest_version_file(date, hour, numbers,
file_type, country_code, format, ucte_folder)
```

```
11.
12.         if selected_ucte_path:
13.             logging.info(f"Highest number version for {hour}: {highest_number}")
14.             process_and_save_network(selected_ucte_path, date, hour, file_type, country_code,
output_folder)
15.         else:
16.             logging.warning(f"No valid UCTE file found for {hour}.")
17.
18. def find_highest_version_file(date, hour, numbers, file_type, country_code, format,
ucte_folder):
19.     highest_number = -1
20.     selected_ucte_path = None
21.
22.     for number in numbers:
23.         ucte_filename = f'{date}_{hour}_{file_type}_{country_code}{number}.{format}'
24.         ucte_path = os.path.join(ucte_folder, ucte_filename)
25.
26.         if os.path.exists(ucte_path):
27.             if number > highest_number:
28.                 highest_number = number
29.                 selected_ucte_path = ucte_path
30.
31.     return highest_number, selected_ucte_path
32.
```

Figure 4: User inputs, Highest version files specification

def process_and_save_network(ucte_path ,date, hour, file_type, country_code, output_folder): function loads specified UCTE , configures various load flow settings, performs an AC load flow analysis on the network, logs the result, and prints the analysis report.

A comprehensive analysis of the LoadFlow execution is available in the terminal with **.Reporter()** and **print(str(reporter))**.

Get_provider_parameters() gives a list of all the available provider parameters for AC LoadFlow execution available in **'provider_parameters.xlsx'** file. Order is not used in the current script.

User defines the network parameters through **lf.Parameters()**.

```
1. def process_and_save_network(ucte_path ,date, hour, file_type, country_code, output_folder):
 2.     # Load network
 3.     network = pp.load(ucte_path)
 4.     reporter = nf.Reporter()
 5.
 6.     # Define load flow parameters
 7.     p = lf.Parameters(
 8.         distributed_slack=False ,
 9.         transformer_voltage_control_on=False ,
10.         phase_shifter_regulation_on=True ,
11.         shunt_compensator_voltage_control_on=True ,
12.         voltage_init_mode=None,
13.         use_reactive_limits=None,
14.         twt_split_shunt_admittance=None,
15.         read_slack_bus=None,
16.         write_slack_bus=None,
17.         balance_type=None,
18.         dc_use_transformer_ratio=None,
19.         countries_to_balance=None,
20.         connected_component_mode=None,
21.         dc_power_factor=None,
22.         provider_parameters={
```

```
        'maxOuterLoopIterations' : str(30) ,
        'lowImpedanceBranchMode ': 'REPLACE_BY_MIN_IMPEDANCE_LINE' ,
        'slackBusesIds'  : 'G5MEGA14'
    })

    # Run loadflow
    lf.run_ac(network, parameters=p, reporter=reporter)
    print(str(reporter))
    logging.info(f"LoadFlow completed for {hour}.")

```

Figure 5: AC LoadFlow execution

All parameters description for AC LoadFlow execution are available in:

https://powsybl.readthedocs.io/projects/pypowsybl/en/stable/reference/loadflow/parameters.html#pypowsybl.loadflow.Parameters

Final script processes electrical components using different functions that are explained below. Saves in excel named **output_filename** the electrical components with their corresponding load flows.

```
# Process DataFrames for different components
    nodes = process_bus_sheet(network)
    current_limits = process_current_limits(network)
    lines_final = process_lines(network, nodes, current_limits)
    transformers = process_transformers(network, nodes, current_limits)
    x_nodes = process_x_nodes(network, nodes, current_limits)
    switches = process_switches(network)

    # Define output file name
    output_filename = f'{date}_{hour}_{file_type}_{country_code}_0_OPENLF_REPORT.xlsx'
    output_path = os.path.join(output_folder, output_filename)

    # Save to Excel
    save_to_excel(output_path, nodes, transformers, lines_final, x_nodes, switches)

```

Figure 6: Electrical components and final excel

3. Buses loadflow report

def process_bus_sheet(network):

Script Compines 3 functions of Pypowsybl's documentation in order to create the LoadFlow report for Buses and selects the attributes of each one of the functions based on Unicorn's LoadFlow reports.

Note: use get_bus_breaker_view_buses() instead of get_buses(). Get_buses() function gives a unique name corresponding to a group of buses with the same voltage level () , which is not helpful in LoadFlow report project.

Information about all the available attributes:

- https://powsybl.readthedocs.io/projects/pypowsybl/en/stable/reference/api/pypowsybl.network.Network.get_generators.html

- https://powsybl.readthedocs.io/projects/pypowsybl/en/stable/reference/api/pypowsybl.network.Network.get_loads.html

Reset_index() & **rename()** : Creates an 'id' attribute for each one of the functions based on their labels. Script checks if id column has been created and if not then raises a value error.

Script creates a new column for each function which contains the first 8 characters of the id's using **df['id_8'] = df['id'].astype(str).str[:8]**.

Script merges the functions in pairs based on their **'id_8'** column and adds suffixes to distinguish the same named columns using (**nodes = pd.merge(buses, generators, on='id_8', suffixes=('_bus', '_gen'), how='outer')**).

```
"""""BUS SHEET CREATION"""""
#BUS attributes selection
buses = network.get_bus_breaker_view_buses(attributes = ['v_mag' , 'v_angle'])
loads = network.get_loads(attributes=['p' , 'q'])
generators = network.get_generators(attributes=[ 'target_v' ,'p' , 'q' ,'max_q' , 'min_q' , 'voltage_regulator_on' ])

## Reset indices and rename columns for merging
buses.reset_index(inplace=True)
buses.rename(columns={'index': 'id'}, inplace=True)
generators.reset_index(inplace=True)
generators.rename(columns={'index': 'id', }, inplace=True)
loads.reset_index(inplace=True)
loads.rename(columns={'index': 'id'}, inplace=True)


for df, name in zip([generators, loads, buses], ["generators", "loads", "buses"]):
    if 'id' not in df.columns:
        raise ValueError(f"'{name}' CSV file is missing the 'id' column.")
for df in [generators, loads, buses ]:
      df['id_8'] = df['id'].astype(str).str[:8]

nodes = pd.merge(buses, generators, on='id_8', suffixes=('_bus', '_gen'), how='outer')
nodes = pd.merge(nodes, loads, on='id_8', suffixes=('', '_load' ), how='outer')
nodes.drop(columns=['id_8', 'id_gen' ,  'id'  ], inplace=True)

```

Figure 7: Merge

Drops the pairing keys (**nodes.drop()**) and keeps only **id_bus** containing the merging id information of the buses, generators and loads. Fills with zero values the empty cells for P, Q of generators and loads (**.fillna(0)**). Renaming columns based on Unicorns LoadFlow reports (**nodes.rename()**).

```
#Fill empty cells and rename columns
    nodes.drop(columns=['id_8', 'id_gen' , 'id' ], inplace=True)
    columns = ['p' , 'q' , 'p_load' , 'q_load']
    nodes[columns] = nodes[columns].fillna(0) # Drop zero values to empty cells
    rename_columns = {
        'target_v': 'Reference Voltage',
        'p': 'Pgen',
        'q': 'Qgen',
        'p_load': 'Pload',
        'q_load': 'Qload',
        'id_bus' : 'BUS'
    }
```

```
    nodes.rename(columns=rename_columns, inplace=True)

    return nodes

```

Figure 8: Final Bus structure

4. Lines and current limits loadflow reports

def process_current_limits(network): Creates dataframe with all current limits of Lines, Transformers, X-lines.

Network.Get_operational_limits() : Function that gives the current limits of each electrical component of the network.

Reset() & rename(): Creates an element_id column based on the index of the function.

```
def process_current_limits(network):
    current_limits = network.get_operational_limits()
    current_limits.reset_index(inplace=True)
    current_limits.rename(columns={'index': 'element_id'}, inplace=True)
    return current_limits

```

Figure 9: Current limits structure

def process_lines(network, nodes, current_limits): Creates specific structured dataframe of all the lines of the UCTE file.

Network.get_lines() : Gets the lines of the network with specific attributes accordingly to Unicorn's line excel sheet.

ALL Attributes of the Get_lines() available in :

https://powsybl.readthedocs.io/projects/pypowsybl/en/stable/reference/api/pypowsybl.network.Network.get_lines.html

Reset & rename(): Creates an id column based on the index of get.lines().

Script creates two functions corresponding to the two side's of the line and copies the essential attributes of the get_lines() function. Also creates new column named 'side' as unicorn's loadflow reports data structure.

```
def process_lines(network, nodes, current_limits):
    lines = network.get_lines(attributes=['bus_breaker_bus1_id', 'i1', 'p1', 'q1', 'i2', 'p2', 'q2', 'bus_breaker_bus2_id'])
    lines.reset_index(inplace=True)
    lines.rename(columns={'index': 'id'}, inplace=True)

     # Create DataFrame for '1' side attributes
    lines_1_df = lines[['id', 'i1', 'p1', 'q1' ,'bus_breaker_bus1_id' ]].copy()
    lines_1_df.rename(columns={'i1': 'I', 'p1': 'P', 'q1': 'Q' , 'bus_breaker_bus1_id' : 'BUS'}, inplace=True)
    lines_1_df['side'] = 1

    # Create DataFrame for '2' side attributes
```

```
12.     lines_2_df = lines[['id', 'i2', 'p2', 'q2' , 'bus_breaker_bus2_id' ]].copy()
13.     lines_2_df.rename(columns={'i2': 'I', 'p2': 'P', 'q2': 'Q' , 'bus_breaker_bus2_id' :'BUS' },
inplace=True)
14.     lines_2_df['side'] = 2
15.
```

Figure 5: Lines modifications

Wrong data merging with get_operational_limits() is avoided by filtering the dataset of the function to keep only rows with side values ‘ONE’ OR ‘TWO’ based on the fact that every line has the same current limits in both sides.

Functions are merged by their common id’s as a pairing key and undesirable attributes are dropped.

```
1. # Combine both DataFrames and merge with operational limits
2. reduced_operational_limits_df = current_limits[(current_limits['side'] == 'ONE') |
(current_limits['side'] == 'TWO')]
3. merged_lines_1_df = pd.merge(lines_1_df, reduced_operational_limits_df, left_on='id',
right_on='element_id', how='left')
4. merged_lines_2_df = pd.merge(lines_2_df, reduced_operational_limits_df, left_on='id',
right_on='element_id', how='left')
5.
6. #Drop unnecessary columns
7. merged_lines_1_df.drop(columns=['element_id' , 'element_type' , 'side_y' , 'name' , 'type' ,
'acceptable_duration' ], inplace=True)
8. merged_lines_2_df.drop(columns=['element_id'  , 'element_type' , 'side_y' , 'name' , 'type' ,
'acceptable_duration'], inplace=True)
9.
```

Figure 10: Lines and reduced_operational_limis_df

Replaces in ‘id’ attribute, blanks with ‘_’ for same structure with Unicorn’s reports (**astype(str).str.replace(' ', '_')**). Use of process_df() function in order current to have prefix and numeric form.

Compines **merge_lines()** functions to create a final line dataset and uses **.sort_values()** , **.reset_index()** , .**rename()** and .**fillna(0)** to have visual optimized output.

```
 1. #Add prefixes, numeric form
 2. merged_lines_1_df['id'] = merged_lines_1_df['id'].astype(str).str.replace(' ', '_')
 3. merged_lines_2_df['id'] = merged_lines_2_df['id'].astype(str).str.replace(' ', '_')
 4. merged_lines_1_df = process_df(merged_lines_1_df)
 5. merged_lines_2_df = process_df(merged_lines_2_df)
 6.
 7. #Concatenate, sort and rename
 8. combined_lines_df = pd.concat([merged_lines_1_df, merged_lines_2_df], ignore_index=True)
 9. combined_lines_df.drop_duplicates(inplace=True)
10. combined_lines_df.sort_values(by=['id', 'side_x'], ascending=[True, True], inplace=True)
11. combined_lines_df.reset_index(drop=True, inplace=True)
12. columns_L = ['I' , 'P' , 'Q']
13. combined_lines_df[columns_L] = combined_lines_df[columns_L].fillna(0) # Drop zero values to
empty cells
14. rename_columns = {
15.     'value' : 'I_limit'
16.     }
17.     combined_lines_df.rename(columns = rename_columns , inplace = True)
18.
```

Figure 11: Lines structure

Merges **lines** with **nodes** with 'BUS' attribute of both functions as a pairing key to include voltage and theta attributes in the final lines report. Fills empty cells with zero values and reorders the columns to match the one's from Unicorns LoadFlows reports.

```
1.  ##Add voltage magnitude and theta to both sides of the line
2.  voltage_theta = nodes[['BUS', 'v_mag', 'v_angle']]
3.  lines_final = pd.merge(voltage_theta , combined_lines_df ,  on='BUS', how='left')
4.  columns_to_check = ['I', 'P', 'Q', 'side_x', 'I_limit']
5.  mask = lines_final[columns_to_check].isna().all(axis=1)
6.  lines_final = lines_final[~mask]
7.  Order = ['id', 'side_x', 'BUS', 'v_mag', 'v_angle', 'I', 'I_limit', 'P', 'Q']
8.  lines_final = lines_final[Order]
9.
10. return lines_final
```

Figure 12: Lines final LoadFlow report

5. Transformers loadflow report

def process_transformers(network, nodes, current_limits): Creates a specific structured loadflow report of Transformers.

Script uses **get_2_windings_transformers()** with specific attributes for similar output as Unicorn's LoadFlow reports. To guide through all the attributes of the Transformers:

https://powsybl.readthedocs.io/projects/pypowsybl/en/stable/reference/api/pypowsybl.network.Network.update_2_windings_transformers.html

Creates two functions that simulate the sides of the transformers. Those functions are going to be saved as rows in the final .xlsx file. Copies the essential attributes for each side of the transformer and creates the 'side' column as unicorn's loadflow reports data structure.

Note: 'Side' column value of TRANSFORMER_1_df is 2 and not 1 by purpose. The reason is that only this way transformer data are matching with those of Unicorns. It is not a mistake and should not be changed by user, because easily user can see the loadflow reports of Unicorn and OpenLF have the same side and BUS values (SOS).

```
 1.def process_transformers(network, nodes, current_limits):
 2.   #Attributes selection
 3.   transformers = network.get_2_windings_transformers(attributes =['rated_u1' , 'rated_u2' ,
'bus_breaker_bus1_id' , 'p1' , 'q1' , 'i1', 'p2' , 'q2' , 'i2' , 'bus_breaker_bus2_id'] )
 4.   transformers.reset_index(inplace=True)
 5.   transformers.rename(columns={'index': 'id'}, inplace=True)
 6.
 7.   #Create dataframe for '1', '2' side attributes
 8.   transformer_1_df = transformers[[ 'id','i1', 'p1', 'q1' ,'bus_breaker_bus1_id' , 'rated_u1'
]].copy()
 9.   transformer_1_df.rename(columns={'i1': 'I', 'p1': 'P', 'q1': 'Q' , 'bus_breaker_bus1_id' :
'BUS' , 'rated_u1' : 'Base Voltage'}, inplace=True)
10.   transformer_1_df['side'] = 2
11.
12.   transformer_2_df = transformers[[  'id','i2', 'p2', 'q2' , 'bus_breaker_bus2_id' ,
'rated_u2'  ]].copy()
13.   transformer_2_df.rename(columns={'i2': 'I', 'p2': 'P', 'q2': 'Q' , 'bus_breaker_bus2_id'
:'BUS' , 'rated_u2' : 'Base Voltage'}, inplace=True)
14.   transformer_2_df['side'] = 1
15.
```

Figure 13: Transformers structure

Merges functions with **current_limits()** using their common ‘id’ attribute as a pairing key (**pd.merge()**).

Drops off the non essential attributes of current_limits().

```
1. #Merge dataframes with operational limits and drop unnecessary columns
2. transformer_1_df = pd.merge(transformer_1_df, current_limits, left_on='id',
right_on='element_id', how='left')
3. transformer_2_df = pd.merge(transformer_2_df, current_limits, left_on='id',
right_on='element_id', how='left')
4. transformer_1_df.drop(columns=['element_id' , 'element_type' , 'side_y' ,   'name' , 'type' ,
'acceptable_duration' ], inplace=True)
5. transformer_2_df.drop(columns=['element_id'  , 'element_type' , 'side_y' , 'name' , 'type' ,
'acceptable_duration'], inplace=True)
6.
```

Figure 14: Merge dataframes

Replaces ‘ ‘ with ‘_’ for total match with the id’s of Unicorn’s LoadFlow reports. Gives prefix to the current and converts it to numeric form. Compines transformers functions to create a final line dataset and uses .sort_values() , .reset_index() , .rename() and .fillna(0) to have visual optimized output.

```
 1. #Add prefix, numeric form and replace ' ' with '_'
 2.    transformer_1_df['id'] = transformer_1_df['id'].astype(str).str.replace(' ', '_')
 3.    transformer_2_df['id'] = transformer_2_df['id'].astype(str).str.replace(' ', '_')
 4.    transformer_1_df = process_df(transformer_1_df)
 5.    transformer_2_df = process_df(transformer_2_df)
 6.
 7.    #Concatenate sides and sort
 8.    Transformers = pd.concat([transformer_1_df, transformer_2_df], ignore_index=True)
 9.    Transformers.drop_duplicates(inplace=True)
10.    Transformers.sort_values(by=['id', 'side_x'], ascending=[True, True], inplace=True)
11.    Transformers.reset_index(drop=True, inplace=True)
12.    columns_L = ['I' , 'P' , 'Q']
13.    Transformers[columns_L] = Transformers[columns_L].fillna(0)
14.    rename_columns = {
15.    'value' : 'I_limit'
16.  }
17.    Transformers.rename(columns = rename_columns , inplace = True)
18.
```

Figure 15: Combine transformers sides

Merges Transformers() with nodes() in order to include voltage and theta attributes of their common ‘BUS’ id’s. Drops the inactive transformers, fills the empty cells with zero values and reorders the columns.

```
 1. #Add voltage magnitude and theta to each side of the transformer
 2. filtered_merged_df_transformer = nodes[['BUS', 'v_mag', 'v_angle']]
 3. transformers = pd.merge(filtered_merged_df_transformer , Transformers ,  on='BUS', how='left')
 4. transformers = transformers[transformers['I'] != 0]
 5. transformers.reset_index(drop=True, inplace=True)
 6. columns_to_check = ['I', 'P', 'Q', 'side_x', 'I_limit']
 7. mask = transformers[columns_to_check].isna().all(axis=1)
 8. transformers = transformers[~mask]
```

```
 9. Order = ['id', 'side_x', 'BUS', 'Base Voltage', 'v_mag', 'v_angle', 'I', 'I_limit', 'P', 'Q']
10. transformers = transformers[Order]
11.
12. return transformers
```

Figure 16: Final transformers structure

6. X_Nodes loadflow and switches load flow reports

def process_x_nodes(network, nodes, current_limits):

Same proccess also for the X-Nodes_lines(). Script uses **get_dangling_lines()** with the essential attributes and merges with current_limits() and buses() for including I_limit , voltage and theta attributes.

Note: **Get_dangling_lines()** contains all the essential information for X-Nodes and X-Lines and so there is no need for user to search in BUS or Line sheets for boundary node inforrmation

```
 1. def process_x_nodes(network, nodes, current_limits):
 2.     #Attributes selection
 3.     X_nodes_lines = network.get_dangling_lines(attributes = [ 'bus_breaker_bus_id', 'i' , 'p' ,
'q','boundary_v_mag' , 'boundary_v_angle', 'boundary_p', 'boundary_q'])
 4.     X_nodes_lines.reset_index(inplace=True)
 5.     X_nodes_lines.rename(columns={'index': 'id'}, inplace=True)
 6.
 7.     #Merge with current limits and drop, rename columns
 8.     x_nodes = pd.merge(X_nodes_lines , current_limits, left_on='id', right_on='element_id',
how='left' )
 9.     x_nodes['id'] = x_nodes['id'].astype(str).str.replace(' ', '_')
10.     x_nodes.drop(columns=['element_id' , 'element_type' , 'side' , 'name' , 'type' ,
'acceptable_duration' ], inplace=True)
11.     x_nodes.rename(columns={'i': 'I', 'p': 'P', 'q': 'Q' , 'value' : 'I_limit' ,
'bus_breaker_bus_id' : 'BUS'}, inplace=True)
12.
13.     #Add prefix to I, numeric form and fill 0 values to empty cells
14.     x_nodes = process_df(x_nodes)
15.     columns_X = ['I' , 'P' , 'Q' , 'boundary_v_mag' , 'boundary_v_angle' , 'boundary_p' ,
'boundary_q' ]
16.     x_nodes[columns_X] = x_nodes[columns_X].fillna(0)
17.
18.     #Add voltage magnitude and theta to X-Nodes side.
19.     merge_X_nodes = nodes[['BUS', 'v_mag', 'v_angle']]
20.     X_Nodes = pd.merge( merge_X_nodes , x_nodes , on='BUS', how='left')
21.     columns_to_check = ['I', 'P', 'Q', 'I_limit']
22.     mask = X_Nodes[columns_to_check].isna().all(axis=1)
23.     X_Nodes = X_Nodes[~mask]
24.     Order = ['id', 'BUS', 'v_mag', 'v_angle', 'I', 'I_limit', 'P', 'Q' , 'boundary_v_mag' ,
'boundary_v_angle' , 'boundary_p' , 'boundary_q' ]
25.     X_Nodes = X_Nodes[Order]
26.
27.     return X_Nodes
28.
```

Figure 17: X-Nodes

def process_switches(network):

Script uses .get_switches() with specific attributes to take information about the switches of the network.

```
1. def process_switches(network):
```

```
2.     switches = network.get_switches(attributes=['bus_breaker_bus1_id', 'kind', 'open',
'retained', 'bus_breaker_bus2_id'])
3.     return switches
4.
```

Figure 18: switches structure

Script ensures that process_network_files_from_user_inputs() is executed only when the script is run directly, not when it's imported as a module in another script.

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

Figure 19: Code running point