Documentation for `qt_folium_map.py`

Overview

`qt_folium_map.py` is a Python module designed for creating a PyQt5-based application. This application displays an interactive map with markers for each bus, using Folium for map generation. It allows users to search, select, edit bus values, submit changes, run simulations, and receive feedback through a status bar.

Features

The application includes the following functionalities:

- **Search Box with Autocomplete**: For searching buses.
- **Dropdown Menu**: To select specific buses.
- **Form for Editing Bus Values**: Allows users to modify bus-related data.
- **Bus Data Interaction:** Enables users to search, select, and edit bus data.
- **Submit Button**: To apply changes made to the bus values.
- **Simulation Button**: To run simulations based on the current data.
- **Status Bar**: Displays messages and status updates to the user.

Setup and Installation

To run the application, follow these steps:

1. Install the required packages:

   ```
   pip install requirements.txt
   ```

2. Ensure the CSV files (`values.txt` and `sample_for_x_y.txt`) containing bus values and coordinates are in the same directory as this script.
3. Execute the script:

   ```
   python map_qt_folium.py
   ```

Imports and Global Variables

The script imports various modules necessary for its functionality:

- **PyQt5 Modules**: For creating the graphical user interface (GUI).
- **Folium**: Used for generating interactive maps.
- **Pandas and NumPy**: For data manipulation and calculations.
- **Other Utilities**: `os`, `sys`, `random`, `math`, and `win32com.client` for various utility functions.

Global variables and file paths are defined:

- `FILE_PATH`: Path to the DSS (Distribution System Simulator) files.
- `MAP_HTML_FILE:` MAP_HTML_FILE is a global constant that is used to save the map as an HTML file. If the file does not exist, it will be created.
- `MAX_ITER`, `MAX_CONTROL_ITER`: Constants for maximum iterations in simulations.

Functions

1) setup_opendss():

- **Purpose**: Sets up the OpenDSS engine and circuit.
- **Returns**: A tuple containing OpenDSS objects (`dssObj`, `dssText`, `dssCircuit`, `dssElement`, `dssSolution`).
- **Description**: This function initializes the OpenDSS engine, checks if it starts successfully, and sets up the necessary components for the circuit simulation. It compiles the circuit using the file specified in `FILE_PATH` and sets the maximum iterations.
- **Example Usage**:

```
dssObj, dssText, dssCircuit, dssElement, dssSolution= setup_opendss()
```

---

2) __main__ function:

Overview

This function is the entry point (`__main__`) of a Python application designed to visualize bus data in an electrical distribution system. It uses OpenDSS for simulation and PyQt5 for GUI.

Function Breakdown

1. **OpenDSS Setup**
    - **Function:** `setup_opendss()`
        - **Purpose:** Initializes the OpenDSS environment for electrical system simulation.
        - **Returns:** Objects for interacting with the OpenDSS environment (e.g., `dssObj`, `dssText`, `dssCircuit`, `dssElement`, `dssSolution`).
2. **Voltage Data Extraction**
    - **Variable:** `voltages`
    - **Code:** `dssCircuit.AllBusVmagPu`
        - **Purpose:** Extracts the per unit (pu) magnitudes of bus voltages from the circuit.
3. **Loading Bus Data**
    - **Function:** `load_bus_data(dssCircuit, dssElement, dssText, dssSolution)`
        - **Purpose:** Loads data related to loads, buses, generators, lines, and transformers from the OpenDSS simulation.
        - **Returns:** Tuple containing data about loads, bus coordinates, generators, lines, and transformers.
4. **Map Creation**
    - **Function:** `create_map(...)`
        - **Purpose:** Creates a map visualization using the extracted data.
        - **Parameters:** Data related to loads, bus coordinates, generators, lines, voltages, and transformers.
        - **Returns:** An object representing the created map.
5. **PyQt5 Application Setup**
    - **Code:** `app = QApplication(sys.argv)`
        - **Purpose:** Initializes a PyQt5 application.
6. **Main Window Configuration**
    - **Creation:** `main = QMainWindow()`
    - **Configuration:** Title, geometry, and status bar setup.
7. **Status Bar and Message Label**
    - **Purpose:** To display status messages and alerts.
    - **Components:** `QStatusBar` and `QLabel`.
8. **Web Engine View for Map**
    - **Code:** `view = QWebEngineView(main)`
    - **Purpose:** To load and display the HTML file containing the map.

    - **File Loading:** `view.load(QUrl.fromLocalFile(os.path.abspath(MAP_HTML_FILE)))`
9. **Bus Editor Dock Panel**
    - **Creation:** `bus_editor = BusEditor(...)`
    - **Purpose:** Provides a user interface for editing bus data.
    - **Integration:** Added as a docked panel to the main window.
10. **Application Execution**
    - **Code:** `main.show()` and `sys.exit(app.exec_())`
    - **Purpose:** Displays the main window and starts the PyQt5 event loop.

Notes

- The function assumes the presence of certain global variables and functions (`cwd_before`, `setup_opendss()`, `load_bus_data()`, `create_map()`, `MAP_HTML_FILE`, `BusEditor`).
- Error handling is not explicitly included in this function. It's recommended to add try-except blocks, especially around file and network operations.
- The application's scalability and performance aspects depend on the implementation details of the functions and classes it calls.

3) calculate_line_loading(dssCircuit, dssText, dssSolution):

- **Purpose**: Calculates line loading values for the circuit.
- **Parameters**:
    - `dssCircuit`: The active circuit object from OpenDSS.
    - `dssText`: The text interface for OpenDSS commands.
    - `dssSolution`: The solution object for the circuit.
- **Returns**: A dictionary containing the line loading values for each line in the circuit.
- **Description**: This function performs calculations related to the line loading in the electrical circuit. It likely involves accessing and manipulating data from the `dssCircuit` and using the `dssSolution` to perform necessary computations. The exact nature of these calculations will be clarified upon further analysis of the code.
- Example Useage:

```
line_loadings = calculate_line_loading(dssCircuit, dssText, dssSolution)
```

---

4) load_bus_data(dssCircuit, dssElement, dssText, dssSolution)

Purpose:

This function is designed to load bus data from CSV files for a power distribution system simulated in OpenDSS. It creates and returns dictionaries containing bus values, bus coordinates, generator values, line values, and transformer values.

Parameters:

- `dssCircuit`: An instance of the OpenDSS circuit class.
- `dssElement`: An interface to access properties of circuit elements.
- `dssText`: Used for issuing commands to OpenDSS.
- `dssSolution`: Used for accessing solution properties in OpenDSS.

Process:

1. **Lines**:
    - Iterates over each line in the circuit.
    - Extracts the names of the buses connected to each end of the line.
    - Stores this data in the `line_values` dictionary.
2. **Line Loads**:
    - Calls `calculate_line_loading` to get the loading values of each line.
    - Merges these loading values into `line_values`.
3. **Loads**:
    - Iterates over each load in the circuit.
    - Extracts and stores load-related data (kV, kW, kvar) in `load_values`.
4. **Bus Coordinates**:
    - Iterates over each bus in the circuit.
    - Extracts and stores bus coordinates in `bus_coords`.
    - If coordinates are missing, reads from a CSV file.
5. **Generators**:
    - Iterates over each generator.
    - Extracts and stores generator-related data in `generator_values`.
6. **Transformers**:
    - Iterates over each transformer.
    - Extracts and stores transformer-related data in `transformer_values`.

Returns:

- A tuple containing dictionaries: `load_values`, `bus_coords`, `generator_values`, `line_values`, `transformer_values`.

---

5) Function: `add_pu_feedback_layer`

Purpose:

This function adds a layer to a Folium map to represent per unit (PU) values of each bus in a power distribution system.

Parameters:

- **`m`: A Folium map object.**
- **`bus_coords`: A dictionary containing bus coordinates.**
- **`given_voltages`: A dictionary containing the PU values of the buses.**
- **`threshold`: A tuple with lower and upper bounds for acceptable PU values.**

Process:

1. **Layer Creation:**
   - **Creates a feature group for PU feedback.**
2. **Marker Generation:**
   - **Iterates over `bus_coords`.**
   - **Creates a marker for each bus with color indicating whether the PU value is within the specified threshold.**
   - **Adds these markers to the PU feedback layer.**
3. **Layer Addition:**
   - **Adds the PU feedback layer and a layer control to the map.**

Returns:

- **`m`: The updated Folium map object with the PU feedback layer added.**

---

6. Function: `get_base_bus_name`

Purpose:

This function extracts the base bus name from a bus name that includes phase notations.

- **Parameters:**
  - `bus_name_with_phases`: **A string representing the bus name with phase notations.**
- **Process:**
  1. **Phase Notation Removal:**
     - **Splits the bus name by '.' and returns the first part if a '.' is present.**
     - **If no '.' is present, returns the original bus name.**

Returns:

- **A string representing the base bus name without phase notations.**

---

7) Function: add_custom_legend:

Purpose:

This function's primary purpose is to visually represent key information on the map, such as differentiating between loads and generators using distinct colors.

- **Parameters:**
  - `m`: **A Folium map object. This is the map to which the legend will be added.**
- **Process:**
  1. **Defining the HTML Template for the Legend:**
     a. **A multi-line string `template` is defined, which contains the HTML code for the legend. This HTML code includes:**
        i. **Styling (position, size, border, font size) to ensure the legend is visible and appropriately placed on the map.**
        ii. **The content of the legend, which includes an icon and a text label for each item (e.g., Load, Generator). The icons are colored circles (`<i class="fa fa-circle" style="color:..."></i>`) with corresponding descriptions, allowing users to easily identify different elements on the map.**

8) Function: `create_map:`

The `create_map` function is designed to create an interactive map using Folium, a Python library for visualizing geospatial data. This map displays various elements of a power distribution system, including transmission lines, buses (loads), generators, and transformers.

Purpose:

This function visualizes geospatial data related to a power distribution system on a map, providing an intuitive and interactive way to understand the physical layout and characteristics of the system's components.

Parameters:

- `load_values`**: A dictionary containing data about loads.**
- `bus_coords`**: A dictionary containing coordinates of buses.**
- `generator_values`**: A dictionary containing data about generators.**
- `line_values`**: A dictionary containing data about transmission lines.**
- `voltages`**: A dictionary containing voltage data (not explicitly used in the function).**
- `transformer_values`**: A dictionary containing data about transformers.**

Process:

1. **Initialization:**
    - **Initializes lists to store all latitude and longitude values (`all_lats`, `all_lons`).**
    - **Assumes `MAP_HTML_FILE` is defined elsewhere, which specifies the file name for the output map.**
2. **Map Creation:**
    - **Creates a Folium map object (`m`) centered at the coordinates of the first bus.**
3. **Adding Transmission Lines:**
    - **Iterates over `line_values`.**
    - **Creates PolyLine objects for each transmission line, with popups showing line details. Lines are randomly colored either red or green.**
4. **Adding Load Buses:**
    - **Iterates over `load_values`.**
    - **Adds CircleMarker objects to the map for each load, with popups showing load details. These markers are blue.**
5. **Adding Generators:**
    - **Iterates over `generator_values`.**
    - **Adds CircleMarker objects to the map for each generator, with popups showing generator details. These markers are orange.**
6. **Adding Transformers:**
    - **Iterates over `transformer_values`.**

    - **Adds CircleMarker objects for transformers with popups showing transformer details. These markers are purple.**
7. **Adjusting Map Bounds:**
    - **Calculates the southwest and northeast bounds of the map based on all latitude and longitude values.**
    - **Adjusts the map to fit these bounds.**
8. **Adding Custom Legend:**
    - **Calls `add_custom_legend` to add a legend to the map, explaining the color coding of different elements.**
9. **Saving the Map:**
    - **Saves the map as an HTML file (`MAP_HTML_FILE`).**

Returns:

- **`m`: The Folium map object with all elements added.**

Usage:

- **This function is utilized to create a detailed and interactive map of a power distribution system. It's especially useful for visualizing the geographical layout and the interconnections between different components of the system.**

Example:

- **After defining the necessary dictionaries (`load_values`, `bus_coords`, etc.), calling `create_map(load_values, bus_coords, generator_values, line_values, voltages, transformer_values)` will generate a map visualizing these elements.**

---

9) `run_simulation` Function:

Overview

The `run_simulation` function conducts an electrical system simulation using OpenDSS, applies changes to the load values, solves the circuit, and updates a graphical map representation of the system. It is crucial for updating and visualizing the impact of load changes in an electrical distribution system.

Function Signature

```
def run_simulation(load_values, generator_values, changed_loads, map_obj):
```

Parameters

1. **load_values (dict): A dictionary containing bus values. This parameter represents the initial load values before any changes are made.**
2. **generator_values: The parameter is passed to the function but is not used in the current implementation. It suggests that the function may be intended to handle generator data in the future.**
3. **changed_loads (dict): A dictionary with keys as load names and values as dictionaries indicating the changes (in kW and kVAR) to be applied to those loads.**
4. **map_obj: An object representing the current state of the map visualization. This object is expected to be modifiable to reflect changes in the simulation.**

Process and Implementation

1. **Applying Load Changes:**
   - **Iterates over `changed_loads` and applies the changes to the corresponding loads in the OpenDSS simulation.**
   - **Changes include modifications to kW and kVAR values of loads.**
2. **Circuit Solution:**
   - **Calls `dssSolution.Solve()` to solve the circuit with the new load values.**
   - **Checks for convergence of the solution and prints the outcome.**
3. **Voltage Updates:**
   - **Extracts new voltage values (`new_voltages`) in per unit (pu) after the circuit solution.**
4. **Map Visualization Update:**
   - **Calls `add_pu_feedback_layer` to update the map (`map_obj`) with the new voltage values.**
   - **Saves the updated map as an HTML file (`MAP_HTML_FILE`).**
   - **Triggers a refresh of the map view in the GUI through `refresh_map_view()`.**

Additional Notes

- **The function assumes the existence of several global variables and functions, such as `dssCircuit`, `dssSolution`, `add_pu_feedback_layer`, `MAP_HTML_FILE`, and `refresh_map_view`.**
- **The `generator_values` parameter is currently not utilized in the function.**

---

10) `refresh_map_view`

Overview

The `refresh_map_view` function is designed to reload a QWebEngineView instance that displays a map. This is particularly useful in GUI applications where the map needs to be updated dynamically.

Implementation Details

- **Global Variable: The function uses a global variable `view`, which is assumed to be an instance of `QWebEngineView`.**
- **Loading the Map: The map is loaded from a file (assumed to be named `MAP_HTML_FILE` and located in the current directory or a known path) into the `view` instance.**

Usage

- **This function should be called whenever the map visualization needs to be updated in the GUI, such as after data changes or simulation runs.**

---

11. `extract_numbers`

Overview

The `extract_numbers` function extracts all integers from a given string and returns them as a tuple. This function is useful for processing strings that contain numerical data embedded within.

Parameters

- **s (str): A string from which integers will be extracted.**

Implementation Details

- Regular Expression: Uses the `re.findall` function with a regular expression (`r"\d+"`) to find all occurrences of one or more digits in the string.

- Conversion and Return: Converts the extracted strings to integers and returns them as a tuple.

Usage

- Ideal for parsing strings to extract numerical information, such as in data processing or sorting scenarios.

---

12. `custom_sort`

Overview

`custom_sort` serves as a custom sorting key function, primarily used in sorting operations where elements need to be sorted based on numerical values extracted from the elements.

Parameters

- **item:** The item to be sorted, typically a string from which numbers are to be extracted for sorting purposes.

Implementation Details

- **Sorting Key:** Utilizes the `extract_numbers` function to extract numerical values from `item`.
- **Return Value:** The tuple of integers extracted from `item` serves as the sorting key.

Usage

- Can be passed as a key function to sorting methods like `sorted()` or list's `sort()` to sort elements based on embedded numerical values.

---

13. BusEditor Class:

1. Class Overview

The `BusEditor` class is a complex widget designed for editing bus values and running simulations in a graphical user interface (GUI) environment. It inherits from `QWidget` and provides a comprehensive set of features for interacting with bus data. This documentation provides a detailed breakdown of the class, its methods, and functionalities.

2. Constructor: `__init__`

Parameters:

- **`load_values`: A dictionary containing load values.**
- **`generator_values`: A dictionary containing generator values.**
- **`message_label`: A `QLabel` object for displaying messages.**
- **`map_obj`: An object representing the map.**

Functionality:

1. **Initialization and Layout Setup:**
    - **Inherits from `QWidget`.**
    - **Creates a `QVBoxLayout` for organizing child widgets vertically.**
2. **Search Box Configuration:**
    - **A label and a QLineEdit are added for searching loads.**
    - **A QCompleter is used to suggest load names as the user types.**
3. **Dropdown for Load Selection:**
    - **A `QComboBox` is populated with sorted load names.**
    - **The dropdown menu allows the user to select a load.**
4. **Selected Load Display:**
    - **A `QLabel` is used to display the currently selected load with distinct styling.**
5. **Change Bus Values Section:**
    - **Two `QLabel` widgets display the current `kw` and `kvar` values.**
    - **`QDoubleSpinBox` widgets allow the user to enter percentage changes.**
6. **Submit Changes Button:**
    - **A `QPushButton` is used for submitting the changes made to the bus values.**
7. **Global Adjustment Section:**
    - **Checkboxes for `kw` and `kvar` selection.**
    - **A `QSpinBox` to specify the global adjustment percentage.**
    - **A button to apply these changes globally.**
8. **Add PV System Button:**
    - **A button to open a dialog for adding a PV system to the selected load.**
9. **Run Simulation Button:**
    - **Initiates the simulation with the current parameters.**

3. Class Methods

3.1 `populate_values()`

- **Purpose: Updates the display with the selected load's values.**
- **Functionality: Retrieves the current load's data and updates labels.**

3.2 `on_load_selected()`

- **Purpose: Handles load selection from the search box.**
- **Functionality: Updates input fields with the selected load's values.**

3.3 `on_load_selected_from_completer(selected_load)`

- **Purpose: Handles load selection via the completer.**
- **Trigger: Activated when a load is selected from the search box.**
- **Functionality: Sets dropdown index and updates input fields.**

3.4 `submit_changes()`

- **Purpose: Submits the changes made to the load values.**
- **Functionality: Calculates and stores the new `kw` and `kvar` values.**

3.5 `show_temporary_message(message, duration=1000)`

- **Purpose: Displays a temporary message.**
- **Functionality: Sets the message label's content and style, then clears it after a duration.**

3.6 `apply_global_adjustment()`

- **Purpose: Applies a global adjustment to all loads.**
- **Functionality: Adjusts `kw` and `kvar` values based on the specified percentage.**

3.7 `show_pv_dialog()`

- **Purpose: Displays the dialog for adding a PV system.**
- **Functionality: Collects PV system details and shows a success message.**

3.8 `run_simulation()`

- **Purpose: Executes the simulation with the current values.**
- **Functionality: Applies the changes to `load_values` and runs the simulation.**

4. Widget Layout and Design

The BusEditor class uses a vertical box layout (QVBoxLayout) to arrange its child widgets. This layout choice ensures a streamlined, top-to-bottom flow of the interface, making it intuitive for users to follow and interact with.

5. Summary of Key Functionalities

- Load Management: Allows searching, selecting, and viewing load details.
- Value Editing: Enables modification of `kw` and `kvar` values.
- Global Adjustments: Provides a feature to adjust all loads simultaneously.
- PV System Integration: Facilitates the addition of PV systems.
- Simulation Control: Offers the capability to run simulations based on current data.

5. Additional Notes

- **UI Elements:** The class heavily relies on Qt widgets for its interface.
- Data Handling: It manages data flow between the UI and the underlying data structures.
- User Interaction: Designed to provide a user-friendly interface for managing and simulating bus values.

---

14. PvSystemDialog Class

The `PvSystemDialog` class, derived from `QDialog`, is designed for entering photovoltaic (PV) system parameters in a GUI application. It allows users to input details such as the PV system's name, number of phases, kVA, kV, maximum power point (Pmpp), and irradiance.

Constructor: `__init__`

Parameters:

- **`parent`: Optional parent widget, default is `None`.**

Functionality:

1. **Initialization: Inherits from `QDialog` and sets the window title to "Define PV System".**
2. **Layout: Uses `QFormLayout` for arranging input widgets.**

3. **Input Widgets:**
    - **`QLineEdit` for PV system name, kVA, kV, Pmpp, and irradiance.**
    - **`QSpinBox` for selecting the number of phases (1 to 3).**
4. **Selected Load Display: Shows the selected load from the parent widget.**
5. **Save Button: A button for saving the input data, connected to the dialog's `accept` method.**

Usage

Instantiate the `PvSystemDialog` and use its `exec_()` method to display the dialog. After user input, the entered data can be retrieved from the dialog's fields.

Documentation for OpenDSS Data Generation Module (generate_training_data.py)

This documentation provides a comprehensive guide to the module designed for generating data for machine learning models using the OpenDSS engine. The module leverages the OpenDSS engine to simulate electrical networks, allowing users to modify load and generator parameters and collect simulation data. This data is then stored in a CSV file, making it suitable for machine learning applications.

Overview

The module consists of various functions, each designed to perform specific tasks such as setting up OpenDSS, modifying parameters, running simulations, collecting data, and storing this data in a CSV format. The process flows sequentially, starting from the OpenDSS setup to data collection and storage.

Dependencies

- `win32com.client`: **Used for interfacing with the OpenDSS engine.**
- `csv`: **For storing data in CSV format.**
- `random`: **To generate random numbers for parameter modification.**
- `pandas`: **For handling data in a structured format.**

Function Documentation

1. `setup_opendss(dss_path)`

Purpose: Initializes the OpenDSS engine with a specified circuit file.

Parameters:

- `dss_path`: **Path to the DSS file of the circuit.**

Returns:

- **A tuple containing references to the main OpenDSS COM object, text interface, active circuit, and solution object.**

Details:

- **Starts the OpenDSS engine and loads the specified circuit.**
- **Sets maximum iterations for simulation.**

2. `modify_load_parameters()`:

Purpose: Modifies the parameters of a specified load in the circuit.

Parameters:

- `dssCircuit`**: Active circuit object.**
- `load_name`**: Name of the load to be modified.**
- `kw_pct_change`**: Percentage change in kW.**
- `kvar_pct_change`**: Percentage change in kVAr.**

Details:

- **Adjusts the kW and kVAr of the specified load based on the given percentage changes.**

3. `modify_generator_parameters()`:

- Similar to `modify_load_parameters`, but for generators.

4. `store_original_parameters(dssCircuit)`

Purpose: Stores the original parameters of loads and generators.

Parameters:

- `dssCircuit`**: Active circuit object.**

Returns:

- **Two dictionaries containing the original kW and kVAr values for each load and generator.**

5. `reset_to_original_parameters(dssCircuit, original_loads, original_gens)`

Purpose: Resets the loads and generators to their original parameters.

Parameters:

- **`dssCircuit`: Active circuit object.**
- **`original_loads`: Dictionary of original load parameters.**
- **`original_gens`: Dictionary of original generator parameters.**

6. `collect_data_for_ml(dssCircuit, dssSolution)`

Purpose: Collects data from the circuit simulation for machine learning.

Parameters:

- **`dssCircuit`: Active circuit object.**
- **`dssSolution`: Solution object for the circuit.**

Returns:

- **Two dictionaries: `features` (containing load and generator parameters) and `labels` (containing bus voltages).**

7. `store_to_csv(data, file_name)`

Purpose: Stores the collected data in a CSV file.

Parameters:

- **`data`: Data to be stored.**
- **`file_name`: Name of the CSV file.**

8. `main()`

Purpose: Main function to orchestrate the data generation process.

Details:

- **Calls `setup_opendss` to initialize OpenDSS.**
- **Stores original parameters of loads and generators.**
- **Preparing for Randomization**
  - **load_kw_range = (-80, 80) # Percent change. …….. etc**
  - **Purpose: Defines the range of changes for load and generator parameters.**
  - **Detail: These ranges are used to randomly modify the parameters of loads and generators. The values represent the percentage change for loads and actual change in kW and kVAr for generators.**

- **Runs a specified number of simulations, each time randomly modifying load and generator parameters.**
- **Collects and stores data after each simulation.**
- **Data is finally saved in a CSV file using `pandas`.**

Additional Notes

- **The `main` function defines various parameters such as the number of simulations to run, the range of random changes for loads and generators, and the percentage of loads and generators to modify.**
- **After each simulation, the loads and generators are reset to their original parameters to ensure independence of simulations.**
- **The data collection is conditional upon the convergence of the simulation solution.**