KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

KUMASI, GHANA (MAIN CAMPUS)

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL/ELECTRONICS ENGINEERING

ELECTRICAL/ELECTRONICS ENGINEERING

EE 368: POWER SYSTEM ANALYSIS

GROUP SEVEN (7)

NEWTON RAPHSON ASSIGNMENT FOR AN N-BUS SYSTEM

DATE: August 2024

NAME
ESON-BENJAMIN, Phillip Vitus
FIADZO, Noble Kojo Likem
FIADZORGBE, Bright Kofi
FIATEPE, Elorm
FORSON, Kelvin Agyei
FRIMPONG, Andy
FRIMONG, Kwaku Agyei
FRIMPONG, Samuel Oteng
GBORNU, Emmanuel Edinam Kwame
GEMEGAH, Kobla Jeremy Delanyo
GHANSON, Raymond Kofi
GYAMFI, Andy Sarkodie
GYANE, Eugenia Etornam
GYIMAH, Emmanuel Kwasi Gyan
HASSEY, Berneth Kofi
HENAKU, Kwasi Yeboah
IBRAHIM, Abdul Fadil Niibontuo
KAMAL, Hamdan Timtooni
KARIKARI, Edward Kwaku
KEELSON-AIKINS, Samuel
ANYIK, Collins Achuyung

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Problem background

The Newton-Raphson method is a powerful numerical technique used for solving nonlinear equations and is particularly effective for power flow analysis in electrical networks. Traditionally, hand calculations for load flow analysis in power systems involve complex and tedious procedures, especially as the size of the system increases. Such methods are labor-intensive and prone to human error, making them less feasible for modern, large-scale systems.

1.2 Problem statement

To address the limitations of manual computations, this report presents a Python-based implementation of the Newton-Raphson method for load flow analysis. The code is designed to provide an efficient, accurate, and easy-to-use solution for computing power system parameters. By automating the process, the code significantly reduces computational time and human error, making it a valuable tool for engineers and researchers in the field.

1.3 Objectives

- Accurate load flow calculation
- Efficient convergence handling
- Modular code structure
- User readability in data output
- User flexibility in data input

2.0 CODE DOCUMENTATION

2.1 Overview

The load flow problem can be defined as the calculation of the voltage and angle at each bus of a given transmission system for specified generation and load conditions. The provided Python code implements the Newton-Raphson method to solve load flow problems in power systems. This method is a successive approximation procedure based on initial guesses of the unknowns and the use of Taylor's series expansion. The program allows users to input data either manually or generate it randomly for testing purposes. It calculates the bus voltages and angles, performs matrix operations, and prints the results. This report excludes the theory for load flow calculations.

2.2 Function descriptions

1. **get_admittance_matrix_rectangular(n)**

- Purpose: Prompts the user to input the admittance matrix in rectangular form (G + jB) for each element of the matrix.
- Process:
    - Initializes an n x n complex matrix Y.
    - Loops through each matrix element, asking the user to input the real (G) and imaginary (B) parts.
    - Stores the input as a complex number in the matrix Y.
    - Output: Returns the complete admittance matrix Y in rectangular form.

2. **get_admittance_matrix_polar(n)**

- Purpose: Prompts the user to input the admittance matrix in polar form (magnitude ∠ angle) for each element of the matrix.
- Process:
    - Initializes an n x n complex matrix Y.
    - Loops through each matrix element, asking the user to input the magnitude and angle in degrees.
    - Converts the polar form input to rectangular form and stores it as a complex number in the matrix Y.

- Output: Returns the complete admittance matrix Y in rectangular form.

3. **display_matrix(matrix, title)**

- Purpose: Nicely formats and prints the admittance matrix with its real and imaginary parts.
- Process:
    - Loops through each row of the matrix and prints the values with both real and imaginary parts, aligning them for readability.
    - Output: Prints the matrix with a given title.

4. **display_real_matrix(matrix, title)**

- Purpose: Nicely formats and prints matrices that consist of real numbers only.
- Process:
    - Loops through each row of the matrix and prints the values, aligning them for readability.
    - Output: Prints the matrix with a given title.

5. **display_vector(vector, title)**

- Purpose: Prints a vector of real numbers with a specified title.
- Process:
    - Loops through the vector elements and prints each element.
    - Output: Prints the vector with a given title.

6. **get_bus_data(n)**

- Purpose: Collects bus data from the user, including the bus type (Slack/PQ/PV) and associated parameters.
- Process:
    - For each bus, prompts the user to input the bus type.
    - Depending on the bus type, it collects the relevant data (voltage magnitude and angle for slack bus; active and reactive power for PQ bus; active power and voltage magnitude for PV bus).
    - Stores the data in a dictionary format.
    - Output: Returns a list of dictionaries containing the bus data for all buses.

7. **random_bus_data(n)**

- Purpose: Generates bus data randomly with some controlled randomness to ensure realistic values.
- Process:
    - Initializes the slack bus with fixed values.
    - For other buses, randomly assigns them as PQ or PV buses.
    - Assigns realistic values for active/reactive power, voltage magnitudes, and angles, introducing some randomness within defined limits.

- Handles situations where a PV bus reaches its reactive power limits and converts it to a PQ bus if necessary.
- Output: Returns a list of dictionaries containing the randomly generated bus data for all buses.

8. **random_admittance_matrix(n, sparsity=0.1)**

- Purpose: Generates a random admittance matrix that is diagonally dominant to ensure numerical stability.
- Process:
    - Initializes an n x n complex matrix Y.
    - For diagonal elements, assigns larger, more realistic values.
    - For off-diagonal elements, assigns smaller values to reflect weak coupling, based on the specified sparsity.
    - Ensures that the matrix is diagonally dominant to aid in convergence.
    - Output: Returns the randomly generated admittance matrix Y.

9. **def calculate_power_mismatch(bus_data, Y, V, theta):**

- Purpose: Computes the difference between the specified power (P_spec, Q_spec) and the calculated power (P_calc, Q_calc) at each bus, which is used to determine how far the current solution is from satisfying the load flow equations.
- Process:
    - Loops through each bus to calculate P_calc[i] and Q_calc[i] using the current estimates of bus voltages and angles.
    - Computes P_mismatch = P_spec - P_calc and Q_mismatch = Q_spec - Q_calc.
    - Combines the mismatches for all buses into a single mismatch vector.
    - Output: The mismatch vector is used to determine how much the voltages and angles need to be adjusted in the next iteration.

10. **def compute_jacobian_matrix(bus_data, Y, V, theta, PV_indices, PQ_indices, PV_PQ_indices):**

- Purpose: Constructs the Jacobian matrix, which represents the partial derivatives of the power equations with respect to the voltage magnitudes and angles. This matrix is essential for the Newton-Raphson iteration.
- Process:
    - Initializes the Jacobian matrix J.
    - Loops through the PQ and PV buses and computes the derivatives of the power mismatch equations with respect to the voltage magnitudes and angles.
    - Fills in the elements of the Jacobian matrix J based on these derivatives.
    - Output: The Jacobian matrix J is used to solve for the change in voltages and angles (`delta`) in each iteration.

- **def update_bus_voltages(bus_data, delta, V, theta, PV_indices, PQ_indices, PV_PQ_indices):**

- Purpose: Updates the voltage magnitudes and angles at each bus based on the results of the Newton-Raphson iteration.
- Process:
    - Solves the linear system delta = J_inv @ mismatch to obtain the changes in voltage magnitudes and angles.
    - Updates the angle theta[i] for all PQ and PV buses.
    - Updates the voltage magnitude V[i] for all PQ buses.
    - Output: The updated bus voltages and angles are used in the next iteration or as the final solution if convergence is achieved.

11. **main()**

- Purpose: Manages the overall execution of the program.
- Process:
    - Prompts the user to choose between manual input or random generation of data.
    - Collects the required data (admittance matrix and bus data).
    - Calls newton_raphson_load_flow() to perform the load flow analysis.
    - Output: Displays results and allows the user to restart or exit the program.

2.3 Code Snippets

The function **compute_jacobian_matrix** is integral to the Newton-Raphson method used for load flow analysis in power systems. This function computes the Jacobian matrix, which plays a pivotal role in the iterative process of finding the correct voltage magnitudes and phase angles across the buses in the system. The Jacobian matrix is derived from the partial derivatives of the power equations with respect to the system's state variables: voltage magnitudes and phase angles. The accuracy of this matrix directly influences the convergence rate and stability of the Newton-Raphson method. Below is the implementation of the Jacobian matrix computation from the code:

```
    def compute_jacobian_matrix(bus_data, Y, V, theta, PV_indices, PQ_indices, PV_PQ_indices):

n = len(bus_data)

PV_indices = [i for i in range(n) if bus_data[i]['type'] == 'PV']

PQ_indices = [i for i in range(n) if bus_data[i]['type'] == 'PQ']

num_PV_PQ = len(PV_PQ_indices)


# Initialize calculated active and reactive power arrays

P_calc = np.zeros(n)

Q_calc = np.zeros(n)
```

```
    # Calculate P and Q for each bus
    for i in range(n):
        for j in range(n):
            P_calc[i] += V[i] * V[j] * (Y[i, j].real * np.cos(theta[i] - theta[j]) + Y[i, j].imag
* np.sin(theta[i] - theta[j]))
            Q_calc[i] += V[i] * V[j] * (Y[i, j].real * np.sin(theta[i] - theta[j]) - Y[i, j].imag
* np.cos(theta[i] - theta[j]))


    # Initialize the Jacobian matrix
    J = np.zeros((num_PV_PQ + len(PQ_indices), num_PV_PQ + len(PQ_indices)))


    # Fill the Jacobian matrix
    for row, i in enumerate(PV_PQ_indices):
        for col, j in enumerate(PV_PQ_indices):
            if i == j:
                J[row, col] = -Q_calc[i] - (V[i]**2) * Y[i, i].imag  # H term
                if i in PQ_indices:
                    pq_row = num_PV_PQ + PQ_indices.index(i)
                    J[pq_row, col] = P_calc[i] - (V[i]**2) * Y[i, i].real  # L term
                    J[row, pq_row] = P_calc[i] / V[i] + V[i] * Y[i, i].real  # N term
                    J[pq_row, pq_row] = Q_calc[i] / V[i] - V[i] * Y[i, i].imag  # M term
            else:
                J[row, col] = V[i] * V[j] * (Y[i, j].real * np.sin(theta[i] - theta[j]) - Y[i,
j].imag * np.cos(theta[i] - theta[j]))  # H term
                if i in PQ_indices:
                    pq_row = num_PV_PQ + PQ_indices.index(i)
                    J[pq_row, col] = -V[i] * V[j] * (Y[i, j].real * np.cos(theta[i] - theta[j]) +
Y[i, j].imag * np.sin(theta[i] - theta[j]))  # L term
                if j in PQ_indices:
                    J[row, num_PV_PQ + PQ_indices.index(j)] = V[i] * V[j] * (Y[i, j].real *
np.cos(theta[i] - theta[j]) + Y[i, j].imag * np.sin(theta[i] - theta[j]))  # N term
                    if i in PQ_indices:
                        pq_row = num_PV_PQ + PQ_indices.index(i)
                        J[pq_row, num_PV_PQ + PQ_indices.index(j)] = V[i] * (Y[i, j].real *
np.sin(theta[i] - theta[j]) - Y[i, j].imag * np.cos(theta[i] - theta[j]))  # M term


    return J
```

- Power Calculation: The function first calculates the active (P) and reactive (Q) power at each bus based on the bus voltage magnitudes and angles. These calculations are derived from the real and imaginary components of the bus admittance matrix.
- Jacobian Matrix Initialization: The Jacobian matrix is initialized with dimensions corresponding to the number of buses minus the slack bus. The function handles both PV (voltage-controlled) and PQ (load) buses, which require different treatments within the matrix.
- Matrix Population: The Jacobian matrix is populated by iterating over each bus. Diagonal elements (where bus indices match) account for the self-derivatives, while off-diagonal elements represent the interaction between different buses. Special considerations are made for PQ buses to incorporate both voltage magnitude and angle variations.
- Final Output: The function returns the Jacobian matrix, which is then used in the Newton-Raphson iterations to update the voltage magnitudes and angles until the solution converges.

3.0 TESTING AND RESULTS

3.1 Guidelines for program initialization and execution

This Python code implements the Newton-Raphson method to solve load flow problems in power systems. It calculates the bus voltages and angles in a power network, ensuring that the network remains balanced by iterating until the mismatches in power calculations are below a specified tolerance. The code supports both manual and randomly generated inputs, allowing for flexible testing and analysis of various N-bus systems.

How to Run the Code:

- Install Required Libraries:
  - Ensure that you have Python installed with NumPy available. You can install NumPy via pip if it isn't already installed: pip install numpy
- Execute the Script:
  - Run the Python script in your preferred environment (e.g., VS Code, Jupyter Notebook, command line)
- Input Method Selection:
  - When prompted, choose how you wish to provide input data:
    - Enter 1 for manual input.
    - Enter 2 for randomly generated input.
- Provide Admittance Matrix and Bus Data:
  - If you chose manual input:
    - Select the format for the admittance matrix (rectangular or polar).
    - Enter the required values for the admittance matrix.
    - Provide the bus data for each bus in the system.
  - If you chose random input:
    - Specify the number of buses.
    - The program will generate the admittance matrix and bus data automatically.
- Observe Results:
  - The program will execute the Newton-Raphson load flow analysis, displaying the results from the first and last iterations, including the bus voltages, angles, the Jacobian matrix, and the power mismatch vector.
  - The choice of displaying the results from only the **first** and **last** iterations are due to desire for brevity in the display of results.

- Handle Non-Convergence:
    - If the program fails to converge within the specified number of iterations, it will notify you and provide options to either retry with new data or exit the program.

- Flat Start Assumption: The code assumes a flat start for initial guesses, meaning all buses except the Slack bus start with a voltage magnitude of 1.0 per unit and an angle of 0 degrees.
- Error Handling: The code includes error handling for non-convergence scenarios, providing feedback and options to the user.

3.2 Test cases

To validate the accuracy and efficiency of the implemented Newton-Raphson method for load flow analysis, three different N-bus systems have been selected as test cases. Each system varies in complexity and bus types, allowing for a comprehensive evaluation of the algorithm's performance under different conditions.

The test cases are structured as follows:

- Test Case 1: A simple 2-bus system with a Slack and a PV bus. This case serves as a basic validation of the code's ability to handle a minimal system with a straightforward power flow scenario.
- Test Case 2: A more complex 3-bus system, incorporating a Slack bus, a PV bus, and a PQ bus. This case tests the algorithm's capability to solve for both voltage magnitudes and phase angles in a system with varying load and generation conditions.
- Test Case 3: A 3-bus system with one Slack bus and two PQ buses. This scenario challenges the algorithm to solve a system where all non-slack buses have both active and reactive power loads, testing the robustness of the code in a more demanding configuration.
- Test Case 4: A 4-bus system with one Slack bus and three PV buses. This provides a robust assessment of the algorithm’s capabilities

A tolerance of **0.00001** is used for all iterations and test cases. Choosing a particular tolerance value strikes a balance between solution accuracy and computational efficiency. This particular value ensures reliable convergence in a reasonable number of iterations. Reducing the level will increase accuracy but at the cost of more computation time. If convergence is reached, the code displays only the results of the first and last iteration as well as the number of iterations taken to converge.

The expected results, derived from known values and hand-solved calculations, will be provided in the Appendix for reference and comparison with the computational results. Due to long and lengthy solutions, these hand solved solutions have been computed over single or double iterations only. The results from the code however, will directly be displayed here. They are displayed in the conventional manner however in an instance a PV bus is present, entry of priority will be given to the PQ bus as seen for case 2. The results still remain accurate.

Test Case 1:

Bus	Type	Specified Active Power	Specified Reactive Power	Voltage specification
1	Slack	1	0.3	1.02+j0
2	PV	-1.3	-	1.03

Admittance matrix in polar form:

- Diagonal elements: $20\angle -70°$
- Off-diagonal elements:12 $\angle$100°

Results from code output terminal:

```
--- Results from the First Iteration ---


Jacobian Matrix (J):
    12.415668


Inverse of the Jacobian Matrix (J_inv):
     0.080543


Power Mismatch Vector:
Element 1: -6.367766


Updated Voltage Magnitudes and Angles after the First Iteration:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.030000 pu, θ = -29.385943 degrees


Convergence achieved in 4 iterations.


--- Results from the Last Iteration ---
```

```
Final Voltages and Angles:

Bus 1: V = 1.020000 pu, θ = 0.000000 degrees

Bus 2: V = 1.030000 pu, θ = -32.745035 degrees


Jacobian Matrix (J):

    9.270832


Inverse of the Jacobian Matrix (J_inv):

    0.107865


Power Mismatch Vector:

Element 1: -0.000009


Updated Voltage Magnitudes and Angles after the Last Iteration:

Bus 1: V = 1.020000 pu, θ = 0.000000 degrees

Bus 2: V = 1.030000 pu, θ = -32.745035 degrees
```

Test Case 2:

Bus	Type	Specified Active Power	Specified Reactive Power	Voltage specification
1	Slack	1	0.3	1.02+j0
2	PV	-1.3	-	1.03
3	PQ	0.8	1	-

Admittance matrix in polar form:

- Diagonal elements: $20\angle-70°$
- Off-diagonal elements:$12\ \angle 100°$
- Results from code output terminal:

Results from code output terminal:

```
--- Results from the First Iteration ---
```

```
Jacobian Matrix (J):
     24.587892     -12.172224      -2.146291
    -12.172224      24.226271       9.409061
      2.146291      -4.271745      13.361434

Inverse of the Jacobian Matrix (J_inv):
      0.054136       0.025560      -0.009303
      0.027200       0.049560      -0.030531
     -0.000000       0.011739       0.066576

Power Mismatch Vector:
Element 1: -4.221475
Element 2: -1.768658
Element 3: 6.432418

Updated Voltage Magnitudes and Angles after the First Iteration:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.407480 pu, θ = -22.853393 degrees
Bus 3: V = 1.030000 pu, θ = -19.112662 degrees

Convergence achieved in 5 iterations.

--- Results from the Last Iteration ---

Final Voltages and Angles:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.262775 pu, θ = -18.283646 degrees
Bus 3: V = 1.030000 pu, θ = -18.009065 degrees
```

```
Jacobian Matrix (J):
    26.513145     -15.385209      -2.639138
   -15.360006      28.970647       9.272171
     2.782073     -10.110897      24.531895

Inverse of the Jacobian Matrix (J_inv):
     0.053855       0.027054      -0.004432
     0.026953       0.044035      -0.013744
     0.005001       0.015081       0.035601

Power Mismatch Vector:
Element 1: 0.000090
Element 2: -0.000000
Element 3: -0.000002

Updated Voltage Magnitudes and Angles after the Last Iteration:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.262775 pu, θ = -18.283646 degrees
Bus 3: V = 1.030000 pu, θ = -18.009065 degrees
```

Test Case 3:

Bus	Type	Specified Active Power	Specified Reactive Power	Voltage specification
1	Slack	1	0.3	1.02+j0
2	PQ	0.8	1	-
3	PQ	1.2	0.6	-

Admittance matrix in polar form:

- Diagonal elements: $20\angle -70°$

- Off-diagonal elements:12 ∠100°

Results from code output terminal:

```
--- Results from the First Iteration ---


Jacobian Matrix (J):
   23.871740    -11.817693     9.471574    -2.083778
  -11.817693     23.871740    -2.083778     9.471574
   -4.209232      2.083778    13.715965   -11.817693
    2.083778     -4.209232   -11.817693    13.715965


Inverse of the Jacobian Matrix (J_inv):
    0.037573      0.011625    -0.101607    -0.089864
    0.011625      0.037573    -0.089864    -0.101607
    0.030741      0.024345     0.174339     0.138069
    0.024345      0.030741     0.138069     0.174339


Power Mismatch Vector:
Element 1: -1.831171
Element 2: -1.431171
Element 3: 6.077888
Element 4: 5.677888


Updated Voltage Magnitudes and Angles after the First Iteration:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 2.752421 pu, θ = -69.513407 degrees
Bus 3: V = 2.740471 pu, θ = -68.649568 degrees


Convergence achieved in 8 iterations.
```

```
--- Results from the Last Iteration ---

Final Voltages and Angles:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.546019 pu, θ = -23.337843 degrees
Bus 3: V = 1.540467 pu, θ = -22.956380 degrees

Jacobian Matrix (J):
    43.919646   -28.110659    11.092483    -5.149885
   -28.176743    43.997606    -4.775103    11.316053
   -15.549819     5.149885    29.702177   -18.248331
     4.775103   -15.032579   -18.225545    29.340504

Inverse of the Jacobian Matrix (J_inv):
     0.029601     0.017030    -0.014812    -0.010585
     0.016753     0.029515    -0.010822    -0.015174
     0.024105     0.018413     0.041827     0.023144
     0.018740     0.023788     0.022847     0.042407

Power Mismatch Vector:
Element 1: -0.000030
Element 2: -0.000028
Element 3: -0.000000
Element 4: -0.000000

Updated Voltage Magnitudes and Angles after the Last Iteration:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.546019 pu, θ = -23.337843 degrees

Bus 3: V = 1.540467 pu, θ = -22.956380 degrees
```

Test Case 4:

Bus	Type	Specified Active Power	Specified Reactive Power	Voltage specification
1	Slack	1	0.3	1.02+j0
2	PV	-1.2	-	1.03
3	PV	-1.5	-	1.04
4	PV	-0.8	-	1.05

Admittance matrix in polar form:

- Diagonal elements: $20\angle -70°$
- Off-diagonal elements:$12\ \angle 100°$

Results from code output terminal:

```
--- Results from the First Iteration ---


Jacobian Matrix (J):

    37.855616    -12.659113    -12.780835

   -12.659113     38.100242    -12.904921

   -12.780835    -12.904921     38.342505


Inverse of the Jacobian Matrix (J_inv):

     0.039883      0.020039      0.020039

     0.020039      0.039692      0.020039

     0.020039      0.020039      0.039505


Power Mismatch Vector:

Element 1: -1.782017

Element 2: -2.180479

Element 3: -1.580726
```

```
Updated Voltage Magnitudes and Angles after the First Iteration:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.030000 pu, θ = -8.390431 degrees
Bus 3: V = 1.040000 pu, θ = -8.819636 degrees
Bus 4: V = 1.050000 pu, θ = -8.127334 degrees

Convergence achieved in 3 iterations.

--- Results from the Last Iteration ---

Final Voltages and Angles:
Bus 1: V = 1.020000 pu, θ = 0.000000 degrees
Bus 2: V = 1.030000 pu, θ = -8.536256 degrees
Bus 3: V = 1.040000 pu, θ = -8.969738 degrees
Bus 4: V = 1.050000 pu, θ = -8.270885 degrees

Jacobian Matrix (J):
    37.409165    -12.675479    -12.770352
   -12.642037     37.567581    -12.876485
   -12.791049    -12.931473     37.936643

Inverse of the Jacobian Matrix (J_inv):
     0.041006      0.021046      0.020947
     0.020990      0.040913      0.020953
     0.020981      0.021042      0.040565

Power Mismatch Vector:
Element 1: -0.000013
Element 2: -0.000014

Element 3: -0.000013


Updated Voltage Magnitudes and Angles after the Last Iteration:

Bus 1: V = 1.020000 pu, θ = 0.000000 degrees

Bus 2: V = 1.030000 pu, θ = -8.536256 degrees

Bus 3: V = 1.040000 pu, θ = -8.969738 degrees

Bus 4: V = 1.050000 pu, θ = -8.270885 degrees
```

3.3 Error Handling and User Interaction in Non-Convergence Scenarios

In the event that the Newton-Raphson load flow algorithm does not achieve convergence within the maximum number of allowed iterations, the code is designed to handle this scenario gracefully. Specifically, if convergence is not achieved:

- First Iteration Results Display: The code automatically outputs the results from the first iteration. This includes the Jacobian matrix, its inverse, the power mismatch vector, and the updated voltage magnitudes and angles after the first iteration. This information is crucial for understanding the initial behavior of the system and the effectiveness of the starting estimates. This also aids in hand solving to ensure if first iteration is being calculated accurately.
- Non-Convergence Notification: After the maximum number of iterations is reached, the code alerts the user that convergence was not achieved. It provides a comprehensive list of potential reasons for non-convergence, which might include:
    - An inadequate initial guess for voltage magnitudes or angles.
    - A tolerance level that is too strict, making convergence difficult to attain.
    - Poor or unrealistic network data, such as inaccurate bus data or line impedances.
    - Numerical instability within the Jacobian matrix.
    - The maximum number of iterations being insufficient for the system to converge.
- User Prompt for Data Regeneration: The user is then prompted to decide whether they wish to generate new data and restart the simulation or to proceed by viewing the results of the last iteration despite the lack of convergence. This interactive feature allows the user to make an informed decision on how to proceed based on the partial results obtained during the computation.

By incorporating these error-handling mechanisms and interactive features, the code ensures that the user is well-informed and has control over the next steps, even in the event of non-convergence.

Sample from the code output terminal:

```
--- Results from the First Iteration ---


Jacobian Matrix (J):
    16.975878     -5.766392      7.476801
    -6.877237     15.529206     60.763302
    -6.469731      6.419096     -6.527807


Inverse of the Jacobian Matrix (J_inv):
     0.091082     -0.001919      0.086462
     0.081184      0.011573      0.200716
    -0.010439      0.013282     -0.041511


Power Mismatch Vector:
Element 1: -37.935969
Element 2: -34.644670
Element 3: 11.549840


Updated Voltage Magnitudes and Angles after the First Iteration:
Bus 1: V = 1.000000 pu, θ = 0.000000 degrees
Bus 2: V = 0.961182 pu, θ = -133.694304 degrees
Bus 3: V = 0.504674 pu, θ = -67.908749 degrees


Maximum iterations reached without convergence.


Possible Reasons for Non-Convergence:
```

```
1. Poor initial guess for voltage magnitudes or angles.
2. Tolerance level might be too strict, making it difficult to achieve convergence.
3. Poor network data (e.g., unrealistic bus data or line impedances).
4. Numerical instability in the Jacobian matrix.
5. Max iterations reached.

Would you like to generate new data? (yes/no)
Choose NO to view the results of last iteration regardless
Choose YES to forfeit results and go back to main menu
```

4.0 ANALYSIS AND DISCUSSION

4.1 Analysis of results

For each of the four test cases, the results obtained from the Newton Raphson Load flow algorithm in the Appendix have been carefully compared with the expected values derived from hand calculation. The analysis shows that:

- Test Case 1 (2-Bus System with Slack and PV Bus):
  - The computed bus voltages and angles align precisely with the expected values. This demonstrates the accuracy of the algorithm for a simple system configuration.
- Test Case 2 (3-Bus System with Slack, PV, and PQ Buses):
  - The results match the hand-calculated values, confirming the reliability of the algorithm when handling more complex systems with a mix of bus types.
- Test Case 3 (3-Bus System with Slack and Two PQ Buses):
  - The algorithm's output is consistent with the expected results, further validating its effectiveness in scenarios involving multiple PQ buses.
- Test Case 4 (4-Bus System with Slack and Three PV Buses):
  - The algorithm's output is consistent with the expected results, further validating its effectiveness in scenarios involving multiple number of buses.

For all test cases, the solutions converged within few number of iterations. In cases where PV buses are involved, convergence is achieved in a lesser number of iterations. Slight variations between expected and calculated values may be due to rounding off error.

4.2 Implementation issues

In the implementation of the power flow analysis code, several issues were identified:

- Incorrect Jacobian Matrix Construction:
  Reactive power elements were mistakenly included for PV buses in the Jacobian matrix, resulting in an oversized matrix. This caused prolonged iterations and inaccurate results.
- Formula Errors in Jacobian Matrix:
  Errors in computing partial derivatives for active power \(P_i \) with respect to voltage magnitude \(V_i \) and reactive power \(Q_i \) with respect to voltage angle \(\theta_i \) led to incorrect Jacobian values, affecting the accuracy of the power flow analysis.

- Singular Matrix Issues:
  High sparsity levels in the random admittance matrix caused singular matrix errors during Jacobian matrix computation, leading to problems in solving the system.

These issues have been identified and solved to ensure proper operation of the algorithm.

5.0 CONCLUSIONS AND RECOMMENDATIONS

5.1 Challenges and Improvements

Challenge

- One significant challenge encountered is that for very large power systems with numerous buses, the computation time required to generate results can be extensive. Additionally, if the solution fails to converge, the time required to diagnose, display non convergence messages and handle the error is also considerable. This impacts the overall efficiency of the analysis

Solution

- By utilizing parallel processing libraries and modules in python such as *'threading'* or *'multiprocessing'*. Through this, large tasks are broken down into smaller, independent sub tasks that can be executed concurrently. This will improve the performance and efficiency of the code.

Challenge

- Another challenge observed was in assuming a flat voltage start for all initial iterations. This does not present a major problem as the Newton-Rapshon method is not very sensitive to initial guesses. However, using a fixed, flat voltage guess for the initial calculations can hinder the algorithm's convergence by leading to slow or failed convergence, reducing flexibility, and potentially resulting in non-convergence if the guess is outside the feasible range.

Solution

- The algorithm can be updated to allow user input for the initial voltage guess, which includes validating the input, adapting the guess as needed, and refining convergence criteria. This approach improves convergence efficiency and accuracy by leveraging user insights and providing a more suitable starting point.

The most challenging parts of this assignment, however, were:

- Gaining enough proficiency in various Python functionalities essential for the assignment presented a learning curve. Mastering these required not only understanding core programming concepts but also familiarizing oneself with advanced features and libraries.
- Ensuring accurate calculations of the Jacobian matrix.
- Fixing one issue often led to the emergence of new errors, such as indexing errors and singular matrix errors. This iterative debugging process was time-consuming as most participants were novices.

5.2 Enhancing the performance of the Newton-Raphson method

The Newton-Raphson method is one of the most popular and efficient methods for load flow analysis. It uses an iterative procedure to find the solution of the nonlinear equations by linearizing them around an initial guess and updating the guess with the error correction. The method converges rapidly and can handle large and complex systems with high accuracy. However, it also has some drawbacks, such as the **need for a good initial guess**, the **possibility of divergence or oscillation**, and **the high computational cost and memory requirement**. To enhance the performance of the Newton-Raphson method and address its drawbacks in load flow analysis, several strategies can be employed:

- Implement techniques to generate better initial guesses. This can be done by using a flat voltage start, starting with a simpler, more robust method like the Gauss-Seidel method to get a reasonably good initial guess before switching to Newton-Raphson for faster convergence and employing historical data from previous or similar load flows to provide a more informed initial guess.
- Using safeguards to prevent divergence or oscillation by adjusting the step size dynamically, incorporating line search or trust region strategies that control the update step based on the error magnitude or system behavior, or applying damping factors to the update step, especially in cases where rapid changes in the solution occur, to smooth out the iteration process and prevent oscillations.
- Optimize the computational process by utilizing sparse matrix storage and manipulation techniques, employing more efficient linear solvers, such as LU decomposition with partial pivoting, and distributing the computation across multiple processors or cores to handle large-scale systems more efficiently, reducing overall computation time.

APPENDIX

Appendix A

GROUP 7 PSA NR ASSIGNMENT.py

Appendix B

Case1.pdf

Case 3.pdf

Test Case 2.pdf