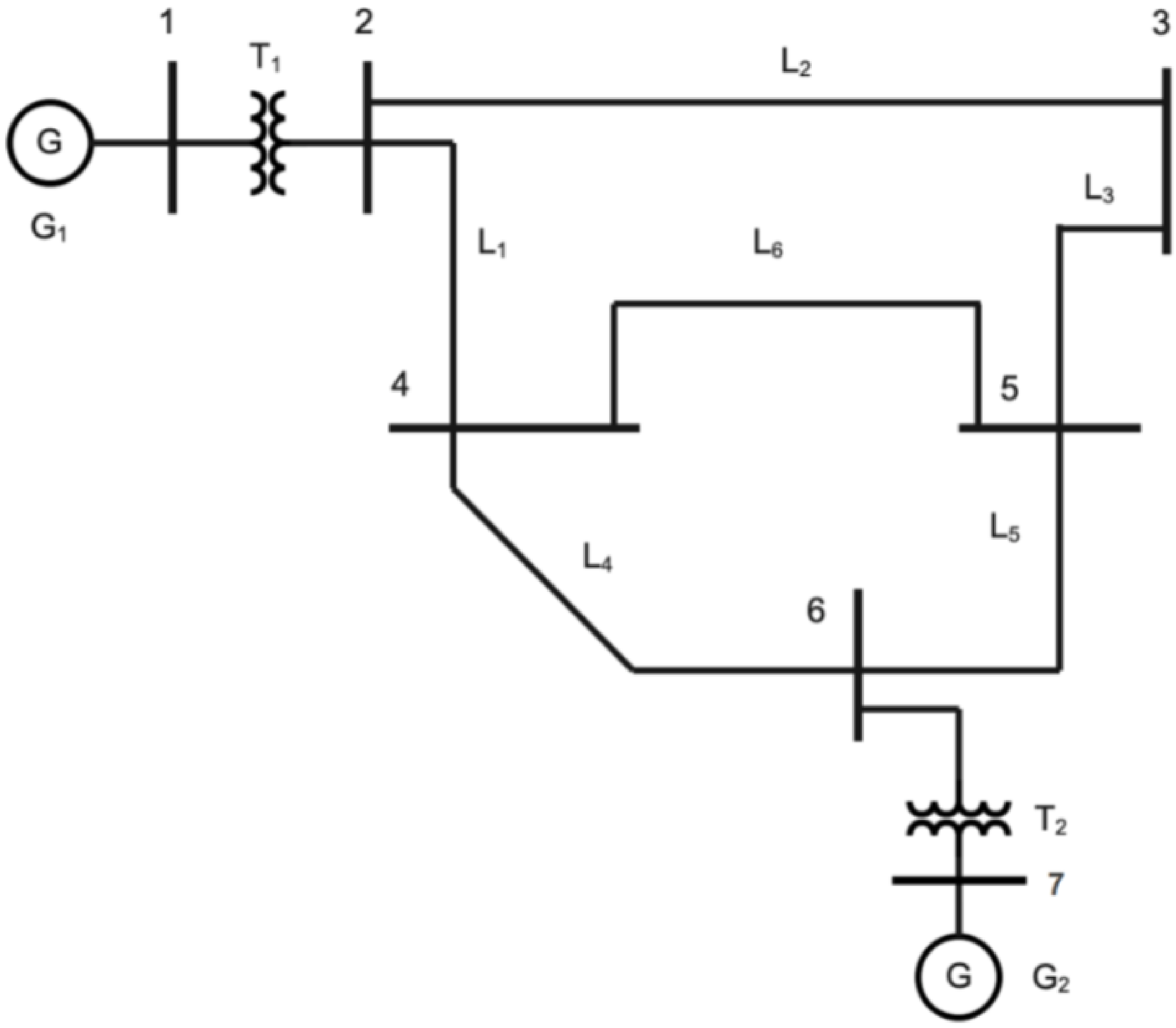
Project 2 Documentation

Group 7: Phan Khanh Do, Jeff Drentkiewicz

7-Node Looped Transmission System



Milestone 1: Creating Equipment Classes

Introduction In Milestone 1, you will create the equipment classes to determine the power system's network. There are three classes: Bus, Transformer, and TransmissionLine, along with their associated subclasses: Conductor, Bundle, and Geometry. These classes will help you model the components of a power system.

1. Bus Class

The Bus class will be used to create buses in your power system. When you create a bus in a power system simulation tool, you must give the bus a name and a nominal voltage level. Your Bus class should mimic this functionality.

Steps to implement:

1. Define the Bus class.

2. Add an __init__ method that takes name and base_kv as parameters.

3. Initialize the name and base_kv attributes.

4. Keep a count of bus instances to assign a unique index to each bus.

Python code:

Bus Class (Bus.py)

The Bus class represents nodes where components connect.

Attributes:

- name: Unique identifier for the bus.
- base_kv: Nominal voltage level of the bus.
- index: Auto-incremented index to uniquely identify each bus.

How it works:
Each time a Bus object is instantiated, it gets a unique index by incrementing the class variable bus_count. This is crucial for matrix operations in later milestones.

```
class Bus:

    bus_count = 0
    def __init__(self, name: str, base_kv: float):
        self.name = name
        self.base_kv = base_kv
        self.index = Bus.bus_count
        Bus.bus_count += 1
```

<u>Transformer Class:</u>

The Transformer class will model a transformer in your power system. Transformers connect two buses with specific parameters such as power rating, impedance, and X/R ratio.

Steps to implement:

1. Define the Transformer class.

2. Add an __init__ method that takes parameters for name, bus1, bus2, power_rating, impedance_percent, and x_over_r_ratio.

3. Initialize the transformer attributes and calculate the impedance and admittance values.

4. Implement methods to calculate impedance, admittance, and the admittance matrix (yprim)**.

Models a transformer connecting two buses.

Key Functions:

- calc_impedance (): Calculates per-unit impedance based on power rating and bus voltage.
- calc_rt_xt (): Splits total impedance into resistance and reactance using the X/R ratio.
- calc_admittance (): Calculates admittance as the inverse of impedance.
- calc_yprim (): Builds a 2x2 primitive admittance matrix (Yprim) for integration into the system's Ybus.

```
def calc_yprim(self):
    y_series = 1 / self.zt if self.zt != 0 else complex('inf')
    return [[y_series, -y_series], [-y_series, y_series]]
```

TransmissionLine Class

Represents a transmission line using conductor bundling and geometry data.

Key Functions:

- calc_line_parameters (): Computes series resistance, reactance, and shunt susceptance using physical properties and standard formulas (Module 7).
- calc_yprim (): Like transformer, calculates the primitive admittance matrix considering both series impedance and shunt admittance.

How it works:
Converts physical line specs into electrical parameters, scaled by line length, and outputs a Yprim matrix for system integration.

Milestone 2: Creating Circuit Class

In Milestone 2, we will create the Circuit class, which will serve as the core framework for managing and analyzing power system networks. The Circuit class will organize all power system components (buses, transformers, transmission lines, and more components to come) and provide methods for adding components, setting system parameters, and preparing the system for power flow analysis.

Circuit Class (Circuit.py)

Acts as a container for the entire power system.

Attributes:

- buses, transformers, tlines: Dictionaries to store system components.

Key Methods:

- add_bus (): Adds a Bus object.
- add_transformer (): Links two buses via a Transformer.
- add_tline (): Connects buses with a TransmissionLine.
- print_circuit_summary (): Outputs a readable overview of the circuit.

How it works:
The Circuit class ensures that all components are systematically stored and easily accessible for further computations, like forming the Ybus matrix in Milestone 4.

Milestone 3: Per-Unit Primitive Admittance Matrices.

In Milestone 3, we will enhance the Transformer and Transmission Line classes by modifying their impedance and admittance calculations to use the per-unit system. Additionally, we will implement a new method for each class to compute the primitive admittance matrix (Yprim). These updates will align with the industry-standard approach for modeling power systems and preparing for power flow analysis.

Key Concepts:

Per-unit simplifies calculations across components with different voltage and power ratings.

$$Z_{pu} = \frac{Z_{actual}}{Z_{base}}$$

For shunt components:

$$B_{pu} = B_{actual} \times Z_{base}$$

Primitive admittance matrices (Yprim) represent individual component contributions before assembling the full Ybus.

<u>Transformer PUS Class:</u>

Enhancements:

Incorporates primary (V1) and secondary (V2) voltage ratings.

Converts transformer impedance to the system base (100 MVA assumed).

Calculates Rpu and Xpu based on X/R ratio.

Uses pandas. DataFrame for clear Yprim representation, labeling rows/columns with bus names.

Key Methods:

calc_impedance ():
Adjusts impedance from transformer base to system base:

$$Z_{pu\_corrected} = Z_{pu} \times \left(\frac{Z_{base}}{Z_{base\_sys}} \right)$$

Converts the transformer's impedance from its own base (MVA rating) to the system base (100 MVA). It then separates this impedance into per-unit resistance (**Rpu**) and reactance (**Xpu**) using the X/R ratio.

calc_admittance ():
Returns series admittance as:

$$Y_{series} = \frac{1}{R_{pu} + jX_{pu}}$$

Calculates the series admittance as the inverse of the complex impedance.

calc_yprim ():
Constructs:

$$Y_{prim} = \begin{bmatrix} Y_{series} & -Y_{series} \\ -Y_{series} & Y_{series} \end{bmatrix}$$

Builds the **primitive admittance matrix (Yprim)** for the transformer. This 2x2 matrix represents how the transformer contributes admittance between its two connected buses, following standard symmetrical patterns.

```python
class Transformer:
    def __init__(self, name: str, bus1: Bus, bus2: Bus, power_rating: float, impedance_percent: float,
                 x_over_r_ratio: float, V1, V2):
        # Initialize transformer parameters
        self.name = name
        self.bus1 = bus1                  # Primary bus object
        self.bus2 = bus2                  # Secondary bus object
        self.power_rating = power_rating  # Transformer rated power (MVA)
        self.impedance_percent = impedance_percent  # Impedance as % of base impedance
        self.x_over_r_ratio = x_over_r_ratio        # X/R ratio of transformer
        self.V1 = V1                      # Primary side voltage (kV)
        self.V2 = V2                      # Secondary side voltage (kV)

        # Calculate per-unit resistance and reactance
        self.Rpu, self.Xpu = self.calc_impedance()

        # Calculate series admittance in per-unit
        self.Yseries = self.calc_admittance()

        # Generate primitive admittance matrix (Yprim)
        self.YPrim_matrix = self.calc_yprim()
```

When a Transformer object is created, it:

Calculates per-unit resistance (**Rpu**) and reactance (**Xpu**) using calc_impedance ().

Calculates series admittance (**Yseries**) with calc_admittance ().

Builds the **Yprim matrix** with calc_yprim().
Automates all key calculations upon initialization, so the transformer is ready for network analysis immediately.

```python
def calc_impedance(self):
    """
    Calculates transformer impedance in per-unit (P.U.S), converted to system base.
    """
    Z_base = (self.V1 ** 2) / self.power_rating    # Calculate base impedance on transformer base
    Z_pu = (self.impedance_percent / 100) * Z_base # Convert % impedance to per-unit on transformer base

    Z_base_sys = (self.bus1.base_kv ** 2) / 100    # System base impedance assuming 100 MVA base
    Z_pu_corrected = Z_pu * (Z_base / Z_base_sys)  # Adjust impedance to system base

    R_pu = Z_pu_corrected / np.sqrt(1 + (self.x_over_r_ratio ** 2))  # Calculate per-unit resistance
    X_pu = R_pu * self.x_over_r_ratio                                 # Calculate per-unit reactance
    return R_pu, X_pu
```

Calculate transformer base impedance using its voltage and MVA rating.

Convert % impedance to **per-unit** on the transformer base.

Adjust to system base (assumed 100 MVA).

Split the corrected per-unit impedance into **Rpu** and **Xpu** using the X/R ratio.

```python
def calc_admittance(self):
    """
    Calculates the series admittance (Y = 1/Z) in per-unit.
    """
    return 1 / complex(self.Rpu, self.Xpu) if self.Rpu or self.Xpu else float('inf')
```

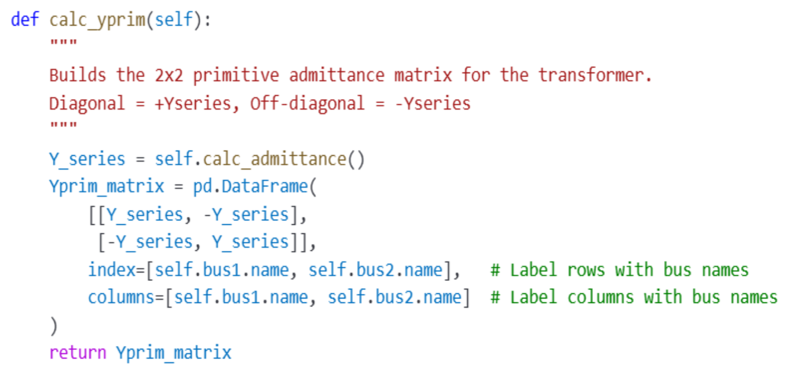
Uses the series admittance (Yseries).

Constructs a 2x2 symmetric matrix:

Diagonal elements = +Yseries.

Off-diagonal elements = -Yseries.

Labels rows and columns with bus names using pandas. DataFrame. Why: This matrix defines how the transformer interacts electrically between its two buses, ready for Ybus assembly.



```python
def calc_yprim(self):
    """
    Builds the 2x2 primitive admittance matrix for the transformer.
    Diagonal = +Yseries, Off-diagonal = -Yseries
    """
    Y_series = self.calc_admittance()
    Yprim_matrix = pd.DataFrame(
        [[Y_series, -Y_series],
         [-Y_series, Y_series]],
        index=[self.bus1.name, self.bus2.name],   # Label rows with bus names
        columns=[self.bus1.name, self.bus2.name]  # Label columns with bus names
    )
    return Yprim_matrix
```

Computes the series admittance as the
nverse of the complex per-unit
mpedance.
Admittance is needed to construct the
Yprim matrix.

<u>TransmissionLine_PUS Class:</u>

The TransmissionLine_PUS class represents a transmission line, converting its physical characteristics into per-unit electrical parameters.

Enhancements:

Computes R_{pu}, X_{pu}, Z_{pu} using physical properties (bundle & geometry). Applies standard per-unit conversion using:

$$Z_{base} = \frac{V_{base}^2}{S_{base}}$$

Builds the Yprim matrix considering both series impedance and shunt admittance split across both ends.

Initialization (__init__):

Accepts details such as connected buses, conductor bundle, geometry, length, and frequency. On initialization, it computes per-unit resistance (**Rpu**), reactance (**Xpu**), shunt susceptance (**Bpu**), series admittance, and the primitive admittance matrix.

calc_impedance (): Calculates per-unit resistance, reactance, and shunt susceptance. Calculates the actual resistance, reactance, and shunt susceptance based on physical formulas (using conductor and geometry data). These values are then converted to the per-unit system using the base impedance formula.

calc_admittance ():
Computes the series admittance as the reciprocal of the complex per-unit impedance.

$$Y_{series} = \frac{1}{R_{pu} + jX_{pu}}$$

calc_yprim ():
Constructs the **primitive admittance matrix (Yprim)** for the transmission line. This matrix accounts for both the series admittance and the distributed shunt admittance (split equally at both ends of the line).

$$Y_{prim} = \begin{bmatrix} Y_{series} + \frac{B_{pu}}{2} & -Y_{series} \\ -Y_{series} & Y_{series} + \frac{B_{pu}}{2} \end{bmatrix}$$

```python
class TransmissionLine:
    def __init__(self, name: str, bus1, bus2, bundle, geometry, length: float, frequency=60):
        # Initialize transmission line parameters
        self.name = name
        self.bus1 = bus1              # Starting bus object
        self.bus2 = bus2              # Ending bus object
        self.bundle = bundle          # Bundle object (defines conductor arrangement)
        self.geometry = geometry      # Geometry object (defines phase spacing)
        self.length = length          # Line length in miles
        self.frequency = frequency    # System frequency (default 60 Hz)

        # Compute per-unit R, X, and B values
        self.Rpu, self.Xpu, self.Bpu = self.calc_impedance()

        # Calculate series admittance (Y = 1/Z)
        self.Yseries = self.calc_admittance()

        # Generate primitive admittance matrix (Yprim)
        self.YPrim_matrix = self.calc_yprim()
```

Upon creation, it:

Computes per-unit resistance, reactance, and shunt susceptance.

Calculates series admittance.

Builds the Yprim matrix.

```python
def calc_impedance(self):
    """
    Calculates series resistance, reactance, and shunt susceptance in per-unit.
    """
    r_series = self.bundle.conductor.resistance / self.bundle.num_conductors  # Resistance per mile

    # Calculate reactance using logarithmic distance ratio
    x_series = 2 * math.pi * self.frequency * 2e-7 * math.log(self.geometry.Deq / self.bundle.DSL) * 1609.34

    # Calculate shunt susceptance using geometric spacing
    b_shunt = 2 * math.pi * self.frequency * (2 * math.pi * 8.854e-12) * math.log(
        self.geometry.Deq / self.bundle.DSC) * 1609.34

    # Convert actual values to per-unit using base impedance
    base_impedance = (self.bus1.base_kv ** 2) / 100
    Rpu = (r_series * self.length) / base_impedance
    Xpu = (x_series * self.length) / base_impedance
    Bpu = (b_shunt * self.length) * base_impedance  # Shunt susceptance scales differently

    return Rpu, Xpu, Bpu
```

Calculates physical resistance, reactance, and shunt susceptance based on conductor and geometry.

Converts these values into per-unit using system base impedance.

Returns **Rpu**, **Xpu**, and **Bpu**.

```python
def calc_admittance(self):
    """
    Computes series admittance in per-unit.
    """
    return 1 / complex(self.Rpu, self.Xpu) if self.Rpu or self.Xpu else float('inf')
```

Converts the per-unit impedance into admittance.

```python
def calc_yprim(self):
    """
    Builds the 2x2 primitive admittance matrix for the transmission line.
    Accounts for both series admittance and distributed shunt admittance.
    """
    Yprim_matrix = pd.DataFrame(
        [[self.Yseries + (self.Bpu / 2), -self.Yseries],
         [-self.Yseries, self.Yseries + (self.Bpu / 2)]],
        index=[self.bus1.name, self.bus2.name],
        columns=[self.bus1.name, self.bus2.name]
    )
    return Yprim_matrix
```

Calculates physical resistance, reactance, and shunt susceptance based on conductor and geometry.

Converts these values into per-unit using system base impedance.

Returns **Rpu**, **Xpu**, and **Bpu**.

Milestone 4: Ybus Admittance Matrix

In Milestone 4, we will extend the Circuit class to compute the system-wide Ybus matrix by creating a ybus attribute and implementing a calc_ybus () method. The Ybus matrix (nodal admittance matrix) is a crucial element in power flow analysis, as it represents the network's admittance relationships between buses. This milestone builds upon the previous implementation of primitive admittance matrices for transformers and transmission lines

Circuit_Ybus Class: The Circuit_Ybus class enhances the previous circuit structure by adding functionality to compute the Ybus matrix. It manages system components and performs matrix assembly based on standard power system analysis techniques.

Attributes:

buses, transformers, tlines, bundles, geometries:
Dictionaries storing all network components.

ybus:
Stores the final computed Ybus matrix as a pandas. DataFrame for readability.

add_bus (), add_transformer (), add_tline (), add_bundle ()

These methods allow dynamic construction of the network by adding buses, transformers, transmission lines, and bundles.

They ensure that dependencies (like bundles and conductors) are correctly linked before components are added.

calc_ybus () Method

This is the core function of Milestone 4. It performs the following steps:

Initialize Ybus Matrix:

Creates an N×N zero matrix, where N is the number of buses in the circuit.

Process Transformers:

Iterates through all transformers.

Extracts their **Yprim** matrices.

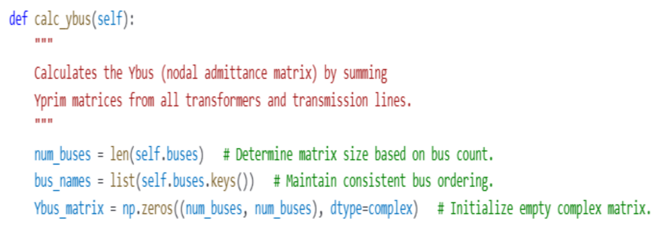
Maps the Yprim values into the appropriate positions in the Ybus matrix based on connected bus indices.

Process Transmission Lines:

Like transformers, it sums each transmission line’s **Yprim** contribution into the Ybus matrix.

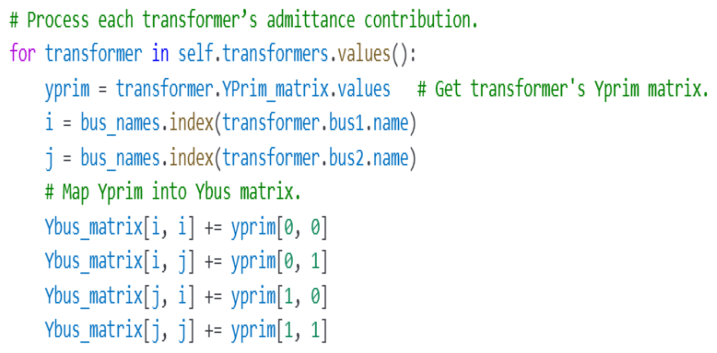
Store Final Ybus:

Convert the NumPy matrix into a labeled pandas. DataFrame uses bus names for easy interpretation and debugging.



```python
def calc_ybus(self):
    """
    Calculates the Ybus (nodal admittance matrix) by summing
    Yprim matrices from all transformers and transmission lines.
    """
    num_buses = len(self.buses)   # Determine matrix size based on bus count.
    bus_names = list(self.buses.keys())   # Maintain consistent bus ordering.
    Ybus_matrix = np.zeros((num_buses, num_buses), dtype=complex)   # Initialize empty complex matrix.
```

Initialize the Ybus Matrix Creates an empty square matrix of size N (where N is the number of buses). Each element is initialized to **0 + 0j** (complex number format). The Ybus matrix starts as zero because we’ll **accumulate** admittance values from each component (superposition principle).



```python
    # Process each transformer’s admittance contribution.
    for transformer in self.transformers.values():
        yprim = transformer.YPrim_matrix.values   # Get transformer's Yprim matrix.
        i = bus_names.index(transformer.bus1.name)
        j = bus_names.index(transformer.bus2.name)
        # Map Yprim into Ybus matrix.
        Ybus_matrix[i, i] += yprim[0, 0]
        Ybus_matrix[i, j] += yprim[0, 1]
        Ybus_matrix[j, i] += yprim[1, 0]
        Ybus_matrix[j, j] += yprim[1, 1]
```

Process All Transformers:

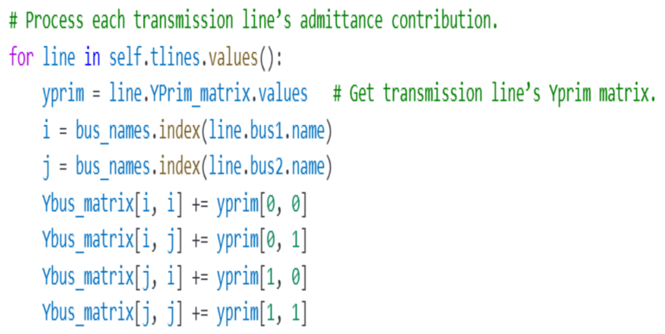
Each transformer affects two buses by adding admittance between them. This step ensures that influence is properly reflected in the system matrix.

Loops through each transformer in the circuit.

Retrieves its primitive admittance matrix (YPrim_matrix) as a NumPy array.

Finds the index positions (i and j) for the two buses connected by the transformer.

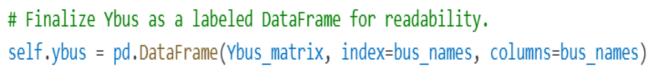
Adds the transformer’s admittance contributions into the correct locations in the Ybus matrix: Diagonal terms (i, i and j, j): Self-admittances. Off-diagonal terms (i, j and j, i): Mutual admittances (negative values).



```
# Process each transmission line's admittance contribution.
for line in self.tlines.values():
    yprim = line.YPrim_matrix.values   # Get transmission line's Yprim matrix.
    i = bus_names.index(line.bus1.name)
    j = bus_names.index(line.bus2.name)
    Ybus_matrix[i, i] += yprim[0, 0]
    Ybus_matrix[i, j] += yprim[0, 1]
    Ybus_matrix[j, i] += yprim[1, 0]
    Ybus_matrix[j, j] += yprim[1, 1]
```

Process All Transmission Lines:

Like transformers, this loop: Retrieves each transmission line's **Yprim matrix**. Locates which buses the line connects. Adds both series admittance and shunt admittance (embedded in Yprim) into the Ybus matrix. Transmission lines contribute to both diagonal and off-diagonal elements, representing how they connect buses and influence voltage/current relationships.



```
# Finalize Ybus as a labeled DataFrame for readability.
self.ybus = pd.DataFrame(Ybus_matrix, index=bus_names, columns=bus_names)
```

Convert to Pandas DataFrame: This improves readability, making it easier to interpret and debug the Ybus matrix by clearly identifying which buses correspond to which matrix entries. Converts the NumPy matrix into a labeled **panda DataFrame**, assigning bus names to rows and columns.

Milestone 5: Settings Class, Bus Refactoring, Generator & Load Class.

Milestone 5 introduces critical enhancements to our power system model by adding a Settings class, refining the Bus class, and implementing new Generator and Load classes. These updates will improve the modularity and realism of our power system representation, paving the way for power flow analysis.

Settings class:

```
class Settings:
    def __init__(self, frequency=60, base_power=100):
        self.frequency = frequency      # System frequency in Hz (default 60 Hz)
        self.base_power = base_power    # System base power in MVA (default 100 MVA)
```

Purpose:
Defines global system parameters such as frequency and base power. These values will be referenced throughout the project, especially for per-unit conversions and admittance calculations.

Centralizes key constants, ensuring consistency across all components and calculations.

Refactored Bus Class:

```
class Bus:
    bus_count = 0
    VALID_BUS_TYPES = ["Slack Bus", "PQ Bus", "PV Bus"]

    def __init__(self, name: str, base_kv: float, bus_type: str, vpu=1.0, delta=0.0):
        self.name = name                    # Bus identifier
        self.base_kv = base_kv              # Voltage base in kV
        self.bus_type = bus_type            # Type of bus (Slack, PQ, PV)
        self.vpu = vpu                      # Voltage magnitude in per-unit
        self.delta = delta                  # Voltage angle in degrees
        self.index = Bus.bus_count          # Unique index assigned
        Bus.bus_count += 1
        self._validate_bus_type()           # Validate bus type
```

Added bus_type to classify buses: **Slack Bus**: Reference bus, **PQ Bus**: Load bus, **PV Bus**: Generator bus. Added vpu (voltage magnitude) and delta (voltage angle), critical for power flow studies. Includes validation to ensure only correct bus types are assigned.

These attributes are essential for Newton-Raphson power flow analysis, where different bus types have different known and unknown variables.

Load class:

```
class Load:
    def __init__(self, name, bus, real_power=0.0, reactive_power=0.0):
        self.name = name                          # Load identifier
        self.bus = bus                            # Associated bus
        self.real_power = real_power              # Real power demand (MW)
        self.reactive_power = reactive_power      # Reactive power demand (MVar)
```

Models load connected to buses, specifying how much active and reactive power is consumed.
Loads define the **PQ injections** at PQ buses, directly impacting the power balance equations in power flow analysis.

Conductor Class:

```
class Conductor:
    def __init__(self, name, diam, GMR, resistance, ampacity):
        self.name = name                      # Conductor type/name (e.g., ACSR type)
        self.diam = diam                      # Diameter in inches
        self.GMR = GMR                        # Geometric Mean Radius (GMR) in feet
        self.resistance = resistance          # AC resistance (ohms/mile)
        self.ampacity = ampacity              # Current carrying capacity (Amps)
```

Stores physical and electrical properties of a conductor type.

Used within the Bundle class and transmission line calculations to derive series resistance and inductive reactance.

Geometry class:

```
class Geometry:
    def __init__(self, name, xa, ya, xb, yb, xc, yc):
        ...
        self.Deq()   # Automatically calculate equivalent spacing.
```

```
def Deq(self):
    # Calculates the equivalent spacing (Deq) based on phase coordinates.
```

Defines the spatial arrangement of transmission line phases.

Calculates **Deq (Equivalent Distance)** using geometric mean distances between phases, which is crucial for determining inductive reactance and capacitance.

Bundle Class:

```
class Bundle:
    def __init__(self, name, num_conductors, spacing, conductor):
        ...
        self.calcradii()   # Computes DSC and DSL based on bundling.
```

```
def calcradii(self):
    # Calculates:
    # DSC = Equivalent spacing for capacitance.
    # DSL = Equivalent spacing for inductance.
```

Models bundled conductors, adjusting GMR and spacing to reflect real-world transmission line configurations. Computes **DSC** (for capacitance) and **DSL** (for inductance), both of which are critical inputs when calculating transmission line parameters in per-unit.

Milestone 6: Power Injection Equations and Power Mismatch Initialization.

Milestone 6 focuses on implementing the power injection equations for each bus in the system and calculating the power mismatch, which is a critical step in performing power flow analysis. These calculations form the basis of iterative numerical methods, such as the Newton-Raphson method, used to solve power flow equations.

Objective:

Implement methods to calculate **power injections (P and Q)** at each bus using the **Ybus matrix** and bus voltage values.

Initialize data structures to prepare for iterative solutions like the **Newton-Raphson method**.

Set up the foundation for solving power flow problems by computing the difference between specified and calculated power — known as the **power mismatch**.

$$S_k = P_k + jQ_k = V_k \cdot I_k^*$$

$$I = Y_{\text{bus}} \cdot V$$

$$P_k = V_k \sum_{n=1}^{N} V_n \left|Y_{kn}\right| \cos(\theta_{kn} + \delta_n - \delta_k)$$

$$Q_k = V_k \sum_{n=1}^{N} V_n \left|Y_{kn}\right| \sin(\theta_{kn} + \delta_n - \delta_k)$$

$$\Delta P = P_{\text{spec}} - P_{\text{calc}}$$

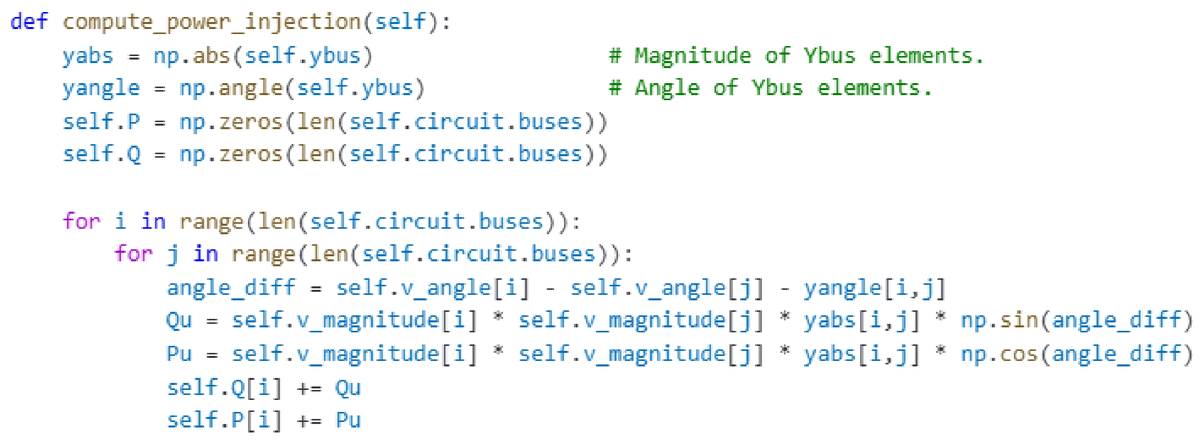
$$\Delta Q = Q_{\text{spec}} - Q_{\text{calc}}$$

```python
class PowerFlow:
    def __init__(self, circuit: Circuit):
        self.circuit = circuit
        self.ybus = circuit.ybus.values                     # Import Ybus matrix from the circuit.
        self.Sbase = circuit.s.base_power                   # System base power for per-unit conversion.
        self.v_magnitude = np.ones(len(circuit.buses))      # Initialize voltage magnitudes (1.0 p.u.).
        self.v_angle = np.zeros(len(circuit.buses))         # Initialize voltage angles (0 rad).
        self.mismatch = np.ones(11)                         # Placeholder for mismatch vector.
        self.P = np.zeros(len(circuit.buses))               # Initialize calculated P injections.
        self.Q = np.zeros(len(circuit.buses))               # Initialize calculated Q injections.
```

Sets up key attributes including Ybus, voltage profiles, power arrays, and mismatch storage. Voltage magnitudes and angles are initialized as flat start conditions. Links the PowerFlow object to the circuit, importing the Ybus matrix and system base power (Sbase). Initializes voltage magnitudes to 1.0 p.u. (flat start). Initializes voltage angles to 0 radians. Prepares arrays to store: Active power (P). Reactive power (Q). Power mismatch vector (mismatch).

```python
def _initialize_specified_power(self):
    self.specified_power = np.zeros(len(self.circuit.buses), dtype=complex)
    for i, bus in enumerate(self.circuit.buses.values()):
        # Sum generator contributions (positive P)
        for gn in self.circuit.generators:
            if self.circuit.generators[gn].bus == bus.name:
                gen = self.circuit.generators[gn]
                self.specified_power[i] += complex(gen.mw_setpoint, 0) / self.Sbase
        # Sum load contributions (negative P and Q)
        for ld in self.circuit.loads:
            if self.circuit.loads[ld].bus == bus.name:
                load = self.circuit.loads[ld]
                self.specified_power[i] += complex(-load.real_power, -load.reactive_power) / self.Sbase
```

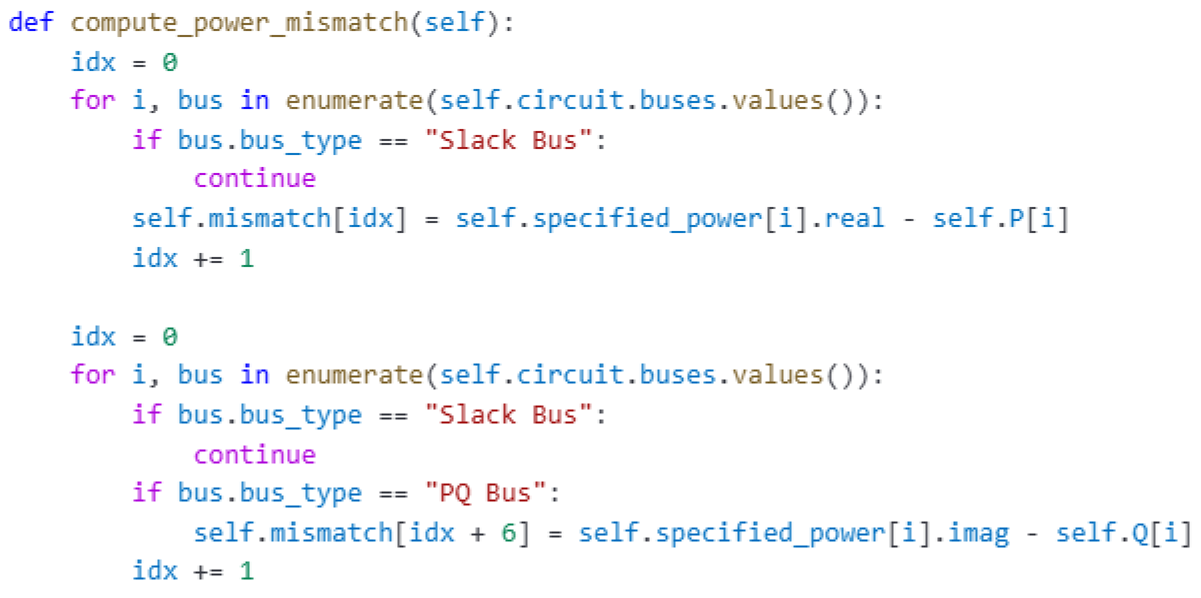
Converts generator setpoints and load demands into **per-unit specified powers** S=P+jQ for each bus. Loads are subtracted because they represent consumption. Loops through each bus. For each bus: Adds generator output (P generation, Q assumed 0 here). Subtract load demand (both P and Q) since loads consume power. Converts everything to per-unit by dividing by system base power. Why: This defines the target power values (P_spec and Q_spec) that the system should achieve after solving.



```
def compute_power_injection(self):
    yabs = np.abs(self.ybus)                    # Magnitude of Ybus elements.
    yangle = np.angle(self.ybus)                # Angle of Ybus elements.
    self.P = np.zeros(len(self.circuit.buses))
    self.Q = np.zeros(len(self.circuit.buses))

    for i in range(len(self.circuit.buses)):
        for j in range(len(self.circuit.buses)):
            angle_diff = self.v_angle[i] - self.v_angle[j] - yangle[i,j]
            Qu = self.v_magnitude[i] * self.v_magnitude[j] * yabs[i,j] * np.sin(angle_diff)
            Pu = self.v_magnitude[i] * self.v_magnitude[j] * yabs[i,j] * np.cos(angle_diff)
            self.Q[i] += Qu
            self.P[i] += Pu
```

Implements the **power injection equations** for active (P) and reactive (Q) power at each bus. Loops through all bus pairs to compute contributions using polar form of voltages and Ybus. This is core to power flow analysis, reflecting network behavior.

Convert the Ybus matrix into magnitude (yabs) and angle (yangle). Loops through each bus pair (i, j) to compute: Active power (P) using cosine terms. Reactive power (Q) using sine terms. Uses the standard power injection formulas from power system theory.



```
def compute_power_mismatch(self):
    idx = 0
    for i, bus in enumerate(self.circuit.buses.values()):
        if bus.bus_type == "Slack Bus":
            continue
        self.mismatch[idx] = self.specified_power[i].real - self.P[i]
        idx += 1

    idx = 0
    for i, bus in enumerate(self.circuit.buses.values()):
        if bus.bus_type == "Slack Bus":
            continue
        if bus.bus_type == "PQ Bus":
            self.mismatch[idx + 6] = self.specified_power[i].imag - self.Q[i]
        idx += 1
```

Loops through each bus. Skips Slack Bus because its power adjusts automatically. For all other buses: Calculates $\Delta P = P_{\text{spec}} - P_{\text{calc}}$. Calculates ΔQ only for PQ buses. Stores these values in the mismatch vector.

Milestone 7: Jacobian Matrix

This milestone focuses on constructing the Jacobian matrix, which is essential for solving power flow problems using the Newton-Raphson method. The matrix consists of partial derivatives of power mismatches with respect to voltage magnitudes and angles.

Objectives:

Implement the calculation of the **Jacobian matrix**, which is essential for solving nonlinear power flow equations using the **Newton-Raphson method**.

The Jacobian relates changes in bus voltages (magnitude and angle) to changes in active and reactive power mismatches.

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix} \begin{bmatrix} \Delta \delta \\ \Delta |V| \end{bmatrix}$$

$$J_1 = \frac{\partial P}{\partial \delta}$$
$$J_2 = \frac{\partial P}{\partial |V|}$$
$$J_3 = \frac{\partial Q}{\partial \delta}$$
$$J_4 = \frac{\partial Q}{\partial |V|}$$

This milestone is critical because the Newton-Raphson method relies on iteratively solving:

$$\mathbf{J} \cdot \Delta x = \text{Mismatch}$$

Code explaination:

1. Class Initialization:

```
class Jacobian:
    def __init__(self, buses, ybus):
        self.buses = buses      # Dictionary of bus objects.
        self.ybus = ybus        # Nodal admittance matrix (Ybus).
```

The Constructor stores:

- The system bus data (needed to identify PQ, PV, and Slack buses).
- The Ybus matrix, used in derivative calculations.

2. Calculating the Full Jacobian Matrix:

```
def calc_jacobian(self, angles, voltages):
```

angles: Current voltage angles for all buses.

voltages: Current voltage magnitudes for all buses.

```
PQ_buses = [bus for bus in self.buses.values() if bus.bus_type == 'PQ Bus']
PV_buses = [bus for bus in self.buses.values() if bus.bus_type == 'PV Bus']
all_buses = PQ_buses + PV_buses
```

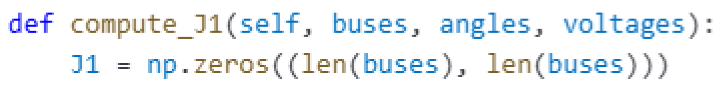
Identifies which buses contribute to which parts of the Jacobian:

- δ: Both PQ and PV buses.
- |V||: Only PQ buses (since PV buses control voltage magnitude).

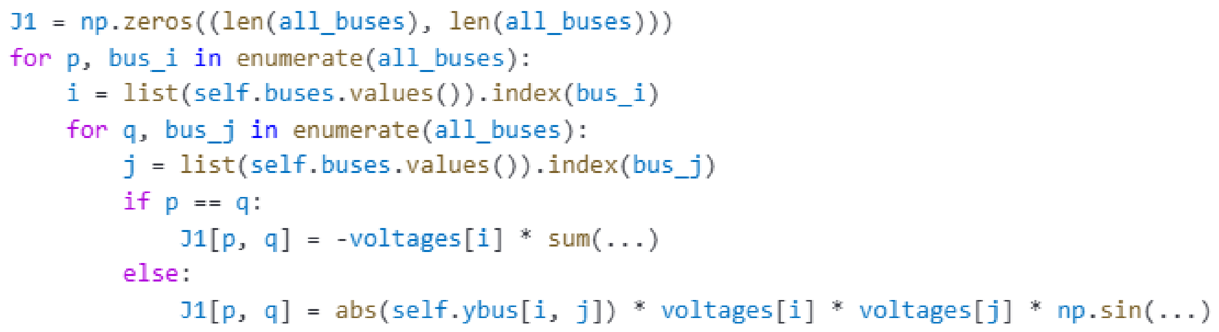
Submatrix Computation:

```
J1 = self.compute_J1(all_buses, angles, voltages)   # dP/dδ
J2 = self.compute_J2(all_buses, angles, voltages)   # dP/d|V|
J3 = self.compute_J3(all_buses, angles, voltages)   # dQ/dδ
J4 = self.compute_J4(all_buses, angles, voltages)   # dQ/d|V|
```

Submatrix: J1 (∂P / ∂δ): Shows how active power at each bus changes with respect to angle deviations.



```
def compute_J1(self, buses, angles, voltages):
    J1 = np.zeros((len(buses), len(buses)))
```



```
J1 = np.zeros((len(all_buses), len(all_buses)))
for p, bus_i in enumerate(all_buses):
    i = list(self.buses.values()).index(bus_i)
    for q, bus_j in enumerate(all_buses):
        j = list(self.buses.values()).index(bus_j)
        if p == q:
            J1[p, q] = -voltages[i] * sum(...)
        else:
            J1[p, q] = abs(self.ybus[i, j]) * voltages[i] * voltages[j] * np.sin(...)
```

Diagonal Terms (i = j):

$$J1_{ii} = -|V_i| \sum_{k \neq i} |V_k||Y_{ik}| \sin(\delta_i - \delta_k - \theta_{ik})$$

Off-Diagonal Terms (i ≠ j):

$$J1_{ij} = |V_i||V_j||Y_{ij}| \sin(\delta_i - \delta_j - \theta_{ij})$$

Diagonal Terms (p = q):

- For bus iii, how its own angle affects its active power output.
- Formula comes from differentiating the power injection equation w.r.t δi.
- The negative sign reflects the nature of increasing δ\deltaδ leading to reduced P in certain conditions.

Off-Diagonal Terms (p ≠ q):

- Reflects interaction between buses iii and jjj, i.e., how changing angle at bus j impacts P_i
- Depends on-line admittance magnitude and sine of angle differences.

<u>Submatrix: J2 (∂P / ∂|V|): </u>Captures sensitivity of active power to changes in voltage magnitude.

```
def compute_J2(self, buses, angles, voltages):
    J2 = np.zeros((len(buses), len(PQ_buses)))
```

```
J2 = np.zeros((len(all_buses), len(PQ_buses)))
...
if p == q:
    J2[p, q] = voltages[i] * abs(self.ybus[i,i]) * np.cos(...) + sum(...)
else:
    J2[p, q] = abs(self.ybus[i, j]) * voltages[i] * np.cos(...)
```

Diagonal Terms:

$$J2_{ii} = |V_i||Y_{ii}|\cos(\theta_{ii}) + \sum_{k \neq i} |V_k||Y_{ik}|\cos(\delta_i - \delta_k - \theta_{ik})$$

Off-Diagonal Terms:

$$J2_{ij} = |V_i||Y_{ij}|\cos(\delta_i - \delta_j - \theta_{ij})$$

Diagonal Terms:

It shows how increasing voltage magnitude at bus iii changes its active power injection. Includes self-admittance and contributions from connected buses.

Off-Diagonal Terms:

Interaction between voltage at bus j and power at bus i. Even though PPP is more sensitive to angles, voltage magnitude still plays a role, especially in voltage stability studies.

<u>Submatrix: J3 (∂Q / ∂δ):</u> Shows how reactive power responds to angle changes (only applies to PQ buses).

```
def compute_J3(self, buses, angles, voltages):
    J3 = np.zeros((len(PQ_buses), len(buses)))
```

```
J3 = np.zeros((len(PQ_buses), len(all_buses)))
...
if p == q:
    J3[p, q] = voltages[i] * sum(...)
else:
    J3[p, q] = -abs(self.ybus[i, j]) * voltages[i] * voltages[j] * np.cos(...)
```

Diagonal Terms:

$$J3_{ii} = |V_i| \sum_{k \neq i} |V_k||Y_{ik}| \cos(\delta_i - \delta_k - \theta_{ik})$$

Off-Diagonal Terms:

$$J3_{ij} = -|V_i||V_j||Y_{ij}| \cos(\delta_i - \delta_j - \theta_{ij})$$

Diagonal Terms:

- Reflects how reactive power at a PQ bus changes with its own angle.

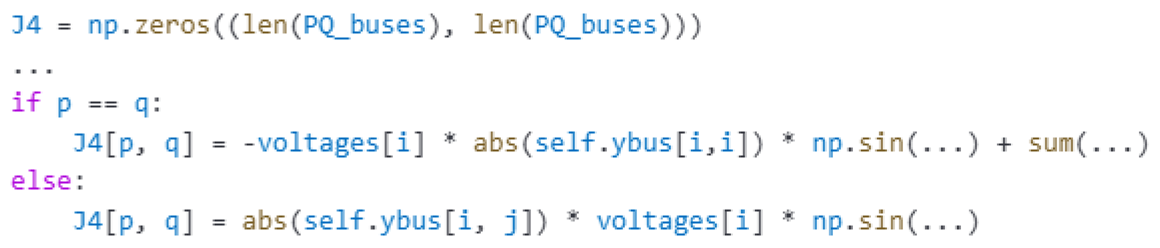
Off-Diagonal Terms:

Captures cross-sensitivities with other buses.

Reactive power is less sensitive to angle changes than active power, but this derivative is essential for accurate NR convergence.

Submatrix: J4 (∂Q / ∂|V|): Reflects how reactive power changes with voltage magnitude (only for PQ buses).

```
def compute_J4(self, buses, angles, voltages):
    J4 = np.zeros((len(PQ_buses), len(PQ_buses)))
```



```
J4 = np.zeros((len(PQ_buses), len(PQ_buses)))
...
if p == q:
    J4[p, q] = -voltages[i] * abs(self.ybus[i,i]) * np.sin(...) + sum(...)
else:
    J4[p, q] = abs(self.ybus[i, j]) * voltages[i] * np.sin(...)
```

Diagonal Terms:

$$J4_{ii} = -|V_i||Y_{ii}|\sin(\theta_{ii}) + \sum_{k\neq i} |V_k||Y_{ik}|\sin(\delta_i - \delta_k - \theta_{ik})$$

Off-Diagonal Terms:

$$J4_{ij} = |V_i||Y_{ij}|\sin(\delta_i - \delta_j - \theta_{ij})$$

Diagonal Terms:

Shows how reactive power at bus iii changes with its own voltage magnitude.

Off-Diagonal Terms:

- How changing voltage at neighboring buses impacts Q_i.

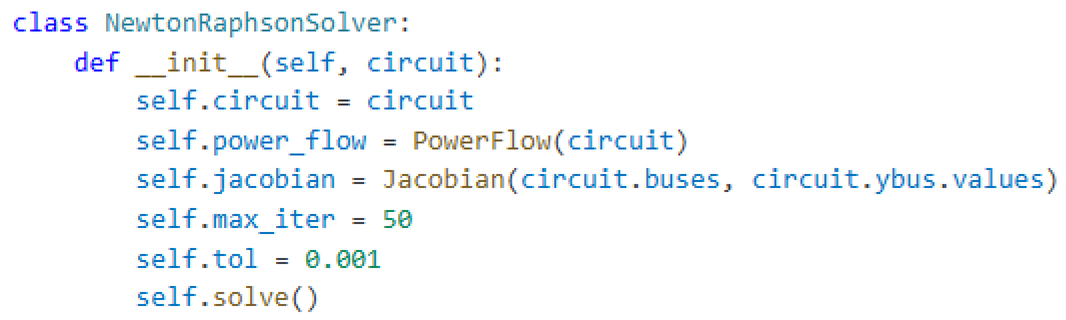
Voltage magnitude directly influences reactive power — this is the most dominant relationship in the Jacobian for Q.

Milestone 8: Newton Raphson

This milestone focuses on constructing the Jacobian matrix, which is essential for solving power flow problems using the Newton-Raphson method. The matrix consists of partial derivatives of power mismatches with respect to voltage magnitudes and angles.

This solver iteratively adjusts voltage magnitudes and angles to reduce the power mismatch vector to zero, using:

$$\Delta x = J^{-1} \cdot \text{Mismatch}$$



```
class NewtonRaphsonSolver:
    def __init__(self, circuit):
        self.circuit = circuit
        self.power_flow = PowerFlow(circuit)
        self.jacobian = Jacobian(circuit.buses, circuit.ybus.values)
        self.max_iter = 50
        self.tol = 0.001
        self.solve()
```

self.circuit: The full power system, already initialized with buses, lines, transformers, etc.

PowerFlow(circuit): Handles all calculations of:

- Specified power (P_{spec}, Q_{spec})
- Calculated power injections
- Mismatch vector (ΔP, ΔQ)

Jacobian(...): Constructs the Jacobian matrix dynamically during each iteration.

self.tol: Convergence threshold (e.g., 0.001 p.u).

self.max_iter: Ensures the solver doesn't run forever in case of divergence.

self.solve(): Kicks off the NR iteration immediately.

```
def solve(self):
    self.power_flow._initialize_specified_power()
```

Prepares the mismatch system by computing S_{spec}=P+jQ from the generators and loads.

Convert MW/MVar into per-unit values based on base power.

```
for iteration in range(self.max_iter):
    self.power_flow.compute_power_injection()
```

Compute P_{calc}, Q_{calc} from current voltages:

$$S_i = V_i \cdot \left(\sum_j Y_{ij} V_j \right)^*$$

```
self.power_flow.compute_power_mismatch()
mismatch = self.power_flow.mismatch
```

$$\text{Mismatch} = \begin{bmatrix} P_{\text{spec}} - P_{\text{calc}} \\ Q_{\text{spec}} - Q_{\text{calc}} \end{bmatrix}$$

```
if np.max(np.abs(mismatch)) < self.tol:
    print("Converged after", iteration, "iterations")
    break
```

Stops iterations if the **largest mismatch element** is below threshold.

Why max(abs(.))? This is the **infinity norm**, ensuring **no bus** has unacceptable errors.

Jacobian Construction and Solution

```
J = self.jacobian.calc_jacobian(self.power_flow.v_angle, self.power_flow.v_magnitude)
```

$$\begin{bmatrix} \frac{\partial P}{\partial \delta} & \frac{\partial P}{\partial |V|} \\ \frac{\partial Q}{\partial \delta} & \frac{\partial Q}{\partial |V|} \end{bmatrix}$$

```
try:
    delta_X = np.linalg.solve(J, mismatch)
```

Solves the linear system J·Δx=Mismatch

delta_X contains corrections for:

- Δ of PV and PQ buses.
- |V| of PQ buses only.

```
except np.linalg.LinAlgError:
    delta_X = np.linalg.pinv(J) @ mismatch
    print("Jacobian singular, used pseudo-inverse.")
```

Uses pseudo-inverse if Jacobian is ill-conditioned or singular.

Update State Variables:

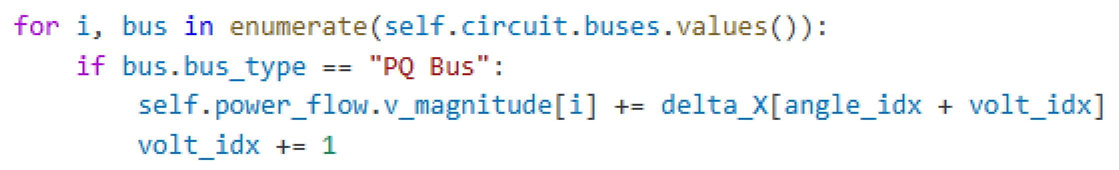
Update Angles:

```
angle_idx = 0
volt_idx = 0

for i, bus in enumerate(self.circuit.buses.values()):
    if bus.bus_type != "Slack Bus":
        self.power_flow.v_angle[i] += delta_X[angle_idx]
        angle_idx += 1
```

Only non-Slack buses have angle corrections.

Slack angle is fixed (typically δ=0), serving as system reference.

Update Voltage Magnitudes:



```
for i, bus in enumerate(self.circuit.buses.values()):
    if bus.bus_type == "PQ Bus":
        self.power_flow.v_magnitude[i] += delta_X[angle_idx + volt_idx]
        volt_idx += 1
```

Only PQ buses have voltage magnitude corrections.

PV bus voltage is fixed (controlled by the generator).

Indexing ensures correct alignment between delta_X values and bus updates.

Milestone 9, 10, 11 Combination.

Milestone 9: Symmetrical Fault

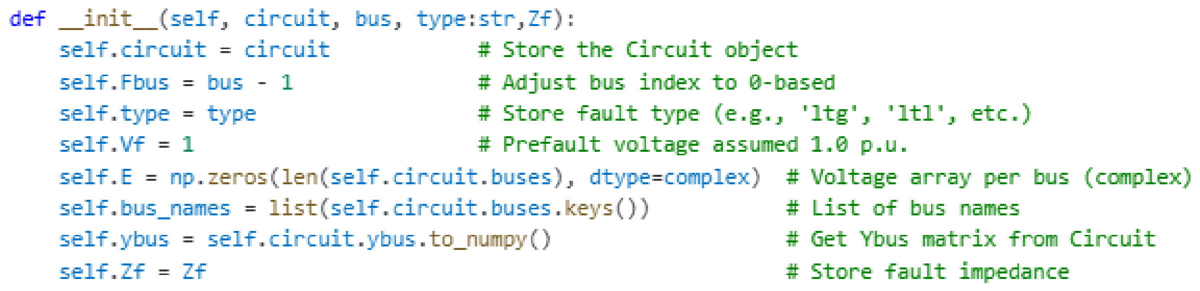
This milestone extends the power system analysis framework to include symmetrical fault analysis (three-phase faults). Students will refactor their existing solver to support both power flow and fault analysis modes, update generator modeling to account for subtransient reactance, and calculate fault currents and post-fault bus voltages using the Zbus matrix derived from Ybus.

Milestone 10: Negative- and Zero-Sequence Impedance Matrices

In this milestone, you will extend your symmetrical component modeling capability by creating negative- and zero-sequence impedance matrices for the bus admittance model developed in earlier milestones. This lays the groundwork for unbalanced fault analysis and sequence network construction in future assignments. You will implement functions that construct the negative- and zero-sequence bus impedance matrices by modifying the existing data structure used for the positive-sequence network. This milestone emphasizes reuse of your Milestones 2-5 code where appropriate, focusing on the mathematical and topological differences in the negative- and zero-sequence networks**.**

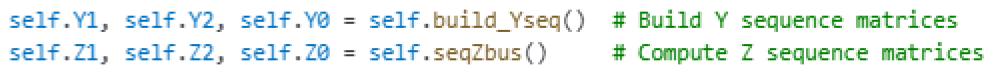
Milestone 11: Asymmetrical Faults

Asymmetrical faults, such as single line-to-ground (SLG), line-to-line (LL), and double line-to-ground (DLG) faults, are among the most common types of disturbances in power systems. Unlike balanced (symmetrical) faults, these faults require the use of symmetrical component analysis to determine fault currents and post-fault bus voltages accurately. In this milestone, you will build on the sequence impedance matrices developed in Milestone 10 to simulate these fault types using a systematic and modular approach. Mastery of asymmetrical fault analysis is essential for understanding protection coordination and evaluating the impact of unbalanced conditions on system performance.

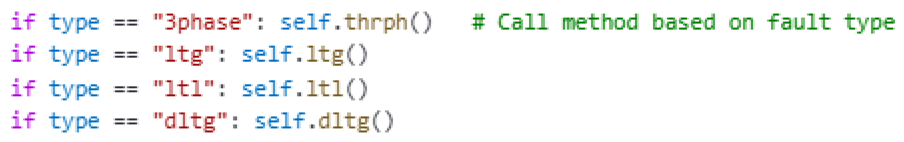


```python
def __init__(self, circuit, bus, type:str,Zf):
    self.circuit = circuit           # Store the Circuit object
    self.Fbus = bus - 1              # Adjust bus index to 0-based
    self.type = type                 # Store fault type (e.g., 'ltg', 'ltl', etc.)
    self.Vf = 1                      # Prefault voltage assumed 1.0 p.u.
    self.E = np.zeros(len(self.circuit.buses), dtype=complex)  # Voltage array per bus (complex)
    self.bus_names = list(self.circuit.buses.keys())          # List of bus names
    self.ybus = self.circuit.ybus.to_numpy()                  # Get Ybus matrix from Circuit
    self.Zf = Zf                                              # Store fault impedance
```

We’ve saved input data, initialized arrays, and got Ybus from circuit.

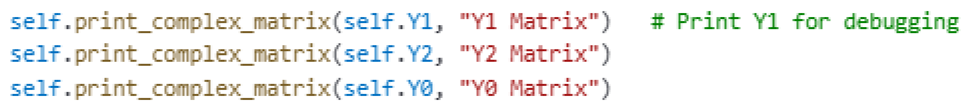


```python
self.Y1, self.Y2, self.Y0 = self.build_Yseq()  # Build Y sequence matrices
self.Z1, self.Z2, self.Z0 = self.seqZbus()     # Compute Z sequence matrices
```



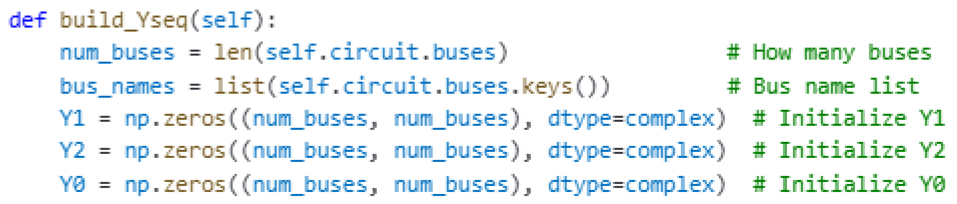
```python
if type == "3phase": self.thrph()   # Call method based on fault type
if type == "ltg": self.ltg()
if type == "ltl": self.ltl()
if type == "dltg": self.dltg()
```

This automatically runs the correct fault calculation based on type input.



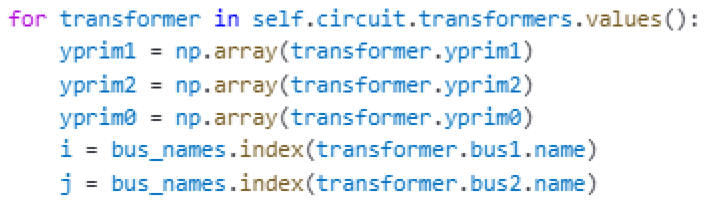
```python
self.print_complex_matrix(self.Y1, "Y1 Matrix")   # Print Y1 for debugging
self.print_complex_matrix(self.Y2, "Y2 Matrix")
self.print_complex_matrix(self.Y0, "Y0 Matrix")
```

Prints sequence Y matrices (optional debugging/verification).



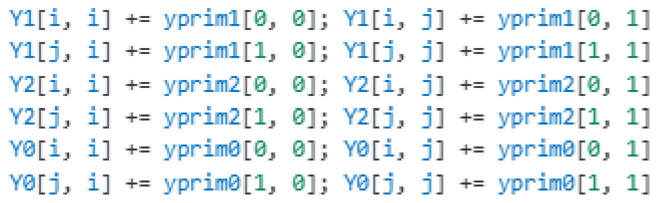
```python
def build_Yseq(self):
    num_buses = len(self.circuit.buses)                # How many buses
    bus_names = list(self.circuit.buses.keys())        # Bus name list
    Y1 = np.zeros((num_buses, num_buses), dtype=complex)  # Initialize Y1
    Y2 = np.zeros((num_buses, num_buses), dtype=complex)  # Initialize Y2
    Y0 = np.zeros((num_buses, num_buses), dtype=complex)  # Initialize Y0
```

Creates empty matrices for Y1, Y2, Y0.



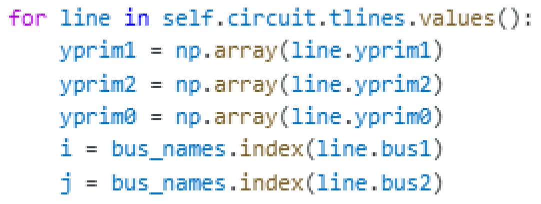
```python
for transformer in self.circuit.transformers.values():
    yprim1 = np.array(transformer.yprim1)
    yprim2 = np.array(transformer.yprim2)
    yprim0 = np.array(transformer.yprim0)
    i = bus_names.index(transformer.bus1.name)
    j = bus_names.index(transformer.bus2.name)
```

Loops through transformers → grabs each sequence primitive matrix and maps buses i, j.



```python
Y1[i, i] += yprim1[0, 0]; Y1[i, j] += yprim1[0, 1]
Y1[j, i] += yprim1[1, 0]; Y1[j, j] += yprim1[1, 1]
Y2[i, i] += yprim2[0, 0]; Y2[i, j] += yprim2[0, 1]
Y2[j, i] += yprim2[1, 0]; Y2[j, j] += yprim2[1, 1]
Y0[i, i] += yprim0[0, 0]; Y0[i, j] += yprim0[0, 1]
Y0[j, i] += yprim0[1, 0]; Y0[j, j] += yprim0[1, 1]
```

Adds transformer admittances into Y1, Y2, Y0.



```python
for line in self.circuit.tlines.values():
    yprim1 = np.array(line.yprim1)
    yprim2 = np.array(line.yprim2)
    yprim0 = np.array(line.yprim0)
    i = bus_names.index(line.bus1)
    j = bus_names.index(line.bus2)
```

Same process for transmission lines.

```
Y1[i, i] += yprim1[0, 0]; Y1[i, j] += yprim1[0, 1]
Y1[j, i] += yprim1[1, 0]; Y1[j, j] += yprim1[1, 1]
Y2[i, i] += yprim2[0, 0]; Y2[i, j] += yprim2[0, 1]
Y2[j, i] += yprim2[1, 0]; Y2[j, j] += yprim2[1, 1]
Y0[i, i] += yprim0[0, 0]; Y0[i, j] += yprim0[0, 1]
Y0[j, i] += yprim0[1, 0]; Y0[j, j] += yprim0[1, 1]
```

Adds transmission line admittances into Y1, Y2, Y0.

```
for gen in self.circuit.generators.values():
    yprim1 = np.array(gen.yprim1)
    yprim2 = np.array(gen.yprim2)
    yprim0 = np.array(gen.yprim0)
    i = bus_names.index(gen.bus)
    Y1[i, i] += yprim1
    Y2[i, i] += yprim2
    Y0[i, i] += yprim0
```

Same for generators (diagonal admittance added).

3. seqZbus() function

```
def seqZbus(self):
    Z1 = np.linalg.inv(self.Y1)
    Z2 = np.linalg.inv(self.Y2)
    Z0 = np.linalg.inv(self.Y0)
    return Z1, Z2, Z0
```

4. thrph() → three-phase fault

```
def thrph(self):
    for generator in self.circuit.generators.values():
        i = self.bus_names.index(generator.bus)
        self.ybus[i,i] += 1/(1j*generator.Xsub)   # Add generator impedance into Ybus diagonal
```

Modify Ybus to include generator subtransient reactance.

```
self.Zbus = np.linalg.inv(self.ybus)         # Compute Zbus from updated Ybus
self.If = self.Vf/self.Zbus[self.Fbus,self.Fbus]   # Calculate fault current
```

$$Y_{bus}[i,i] += \frac{1}{jX''}$$

$$I_f = \frac{V_f}{Z_{nn}}$$

```
for k in range(len(self.circuit.buses)):
    self.E[k] = (1 - (self.Zbus[k,self.Fbus]/self.Zbus[self.Fbus,self.Fbus]))*self.Vf
```

$$E_k = 1 - \frac{Z_{kn}}{Z_{nn}} V_f$$

5. ltg() – Line-to-ground fault:

```
i = self.Fbus
If = self.Vf / (self.Z0[i, i] + self.Z1[i, i] + self.Z2[i, i] + 3 * self.Zf)
```

$$I_f = \frac{V_f}{Z_0 + Z_1 + Z_2 + 3Z_f}$$

6. ltl() – Line-to-line fault:

```
self.If1 = self.Vf / (self.Z2[i,i] + self.Z1[i,i] + self.Zf)
self.If2 = -self.If1
self.If0 = 0
```

$$I_1 = \frac{V_f}{Z_1 + Z_2 + Z_f}, \quad I_2 = -I_1, \quad I_0 = 0$$

7. dltg() – Double line-to-ground fault:

```
I1 = V_f / (Z1 + Z2*(Z0+3Zf)/(Z2+Z0+3Zf))
```

$$I_1 = \frac{V_f}{Z_1 + \frac{Z_2(Z_0+3Z_f)}{Z_2+Z_0+3Z_f}}$$

$$I_2 = -I_1 \frac{Z_0 + 3Z_f}{Z_0 + 3Z_f + Z_2}, \; I_0 = -I_1 \frac{Z_2}{Z_0 + 3Z_f + Z_2}$$

8. VkFn() function:

```
def VkFn(self):
    self.Vk = np.zeros((3,len(self.circuit.buses)),dtype=complex)
    self.I_seq = np.array([self.If0, self.If1, self.If2])
```

Stores voltage per sequence.

```
  for k in range(len(self.circuit.buses)):
      Z_matrix = np.diag([self.Z0[k, self.Fbus], self.Z1[k, self.Fbus], self.Z2[k, self.Fbus]])
      self.Vk[:, k] = np.array([0, self.Vf, 0], dtype=complex) - Z_matrix @ self.I_seq
```

Voltage at each bus per sequence:

$$V_{k,seq} = V_{prefault} - Z_{kn} I_{seq}$$

9. PhVC() function:

```
def PhVC(self):
    self.Vphk = np.zeros((3, len(self.circuit.buses)), dtype=complex)
    self.IphF = np.zeros((3,1),dtype=complex)
    a = 1*np.exp((1j*2*np.pi/3))
    A = np.array(([1,1,1],[1,a**2,a],[1,a,a**2]))
```

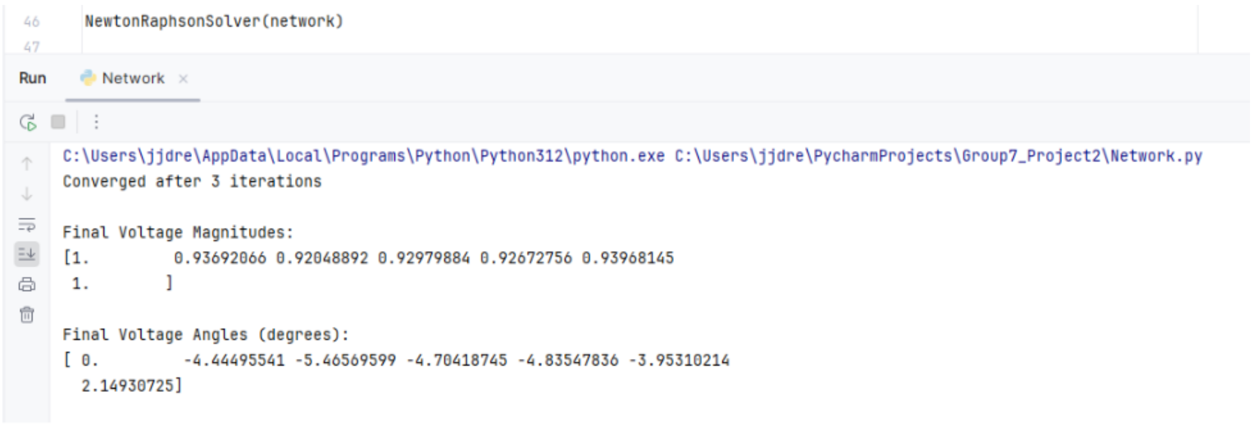
Defines Fortescue matrix A.

```
  for k in range(len(self.circuit.buses)):
      self.Vphk[:, k] = A @ self.Vk[:, k]
  self.IphF = A @ self.I_seq
```

Transform sequence voltages and currents → phase domain.

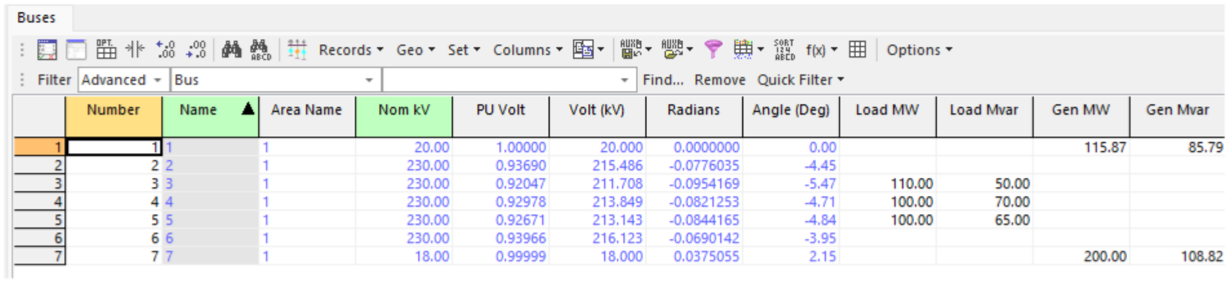
10. SFprint() and AFprint() functions:

11. Python code and PowerWorld Result Comparision.



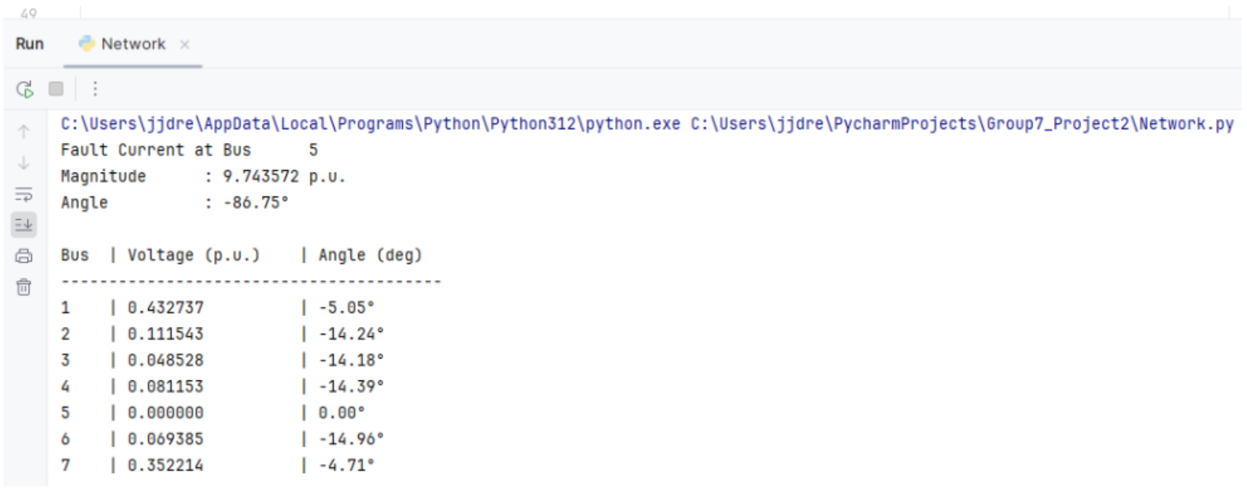
```
NewtonRaphsonSolver(network)
```

```
C:\Users\jjdre\AppData\Local\Programs\Python\Python312\python.exe C:\Users\jjdre\PycharmProjects\Group7_Project2\Network.py
Converged after 3 iterations

Final Voltage Magnitudes:
[1.         0.93692066 0.92048892 0.92979884 0.92672756 0.93968145
 1.        ]

Final Voltage Angles (degrees):
[ 0.         -4.44495541 -5.46569599 -4.70418745 -4.83547836 -3.95310214
  2.14930725]
```

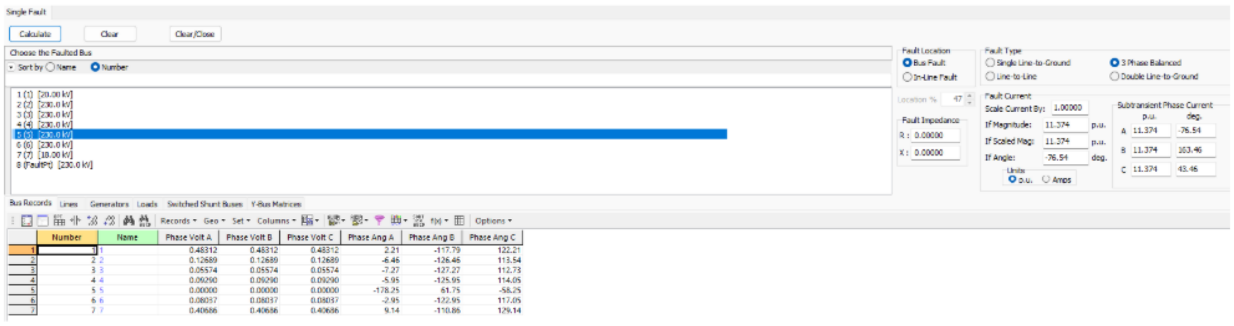


Filter: Advanced | Bus

	Number	Name	Area Name	Nom kV	PU Volt	Volt (kV)	Radians	Angle (Deg)	Load MW	Load Mvar	Gen MW	Gen Mvar
1	1	1	1	20.00	1.00000	20.000	0.0000000	0.00			115.87	85.79
2	2	2	1	230.00	0.93690	215.486	-0.0776035	-4.45				
3	3	3	1	230.00	0.92047	211.708	-0.0954169	-5.47	110.00	50.00		
4	4	4	1	230.00	0.92978	213.849	-0.0821253	-4.71	100.00	70.00		
5	5	5	1	230.00	0.92671	213.143	-0.0844165	-4.84	100.00	65.00		
6	6	6	1	230.00	0.93966	216.123	-0.0690142	-3.95				
7	7	7	1	18.00	0.99999	18.000	0.0375055	2.15			200.00	108.82



```
C:\Users\jjdre\AppData\Local\Programs\Python\Python312\python.exe C:\Users\jjdre\PycharmProjects\Group7_Project2\Network.py
Fault Current at Bus      5
Magnitude       : 9.743572 p.u.
Angle           : -86.75°

Bus  | Voltage (p.u.)     | Angle (deg)
-----------------------------------------
1    | 0.432737           | -5.05°
2    | 0.111543           | -14.24°
3    | 0.048528           | -14.18°
4    | 0.081153           | -14.39°
5    | 0.000000           | 0.00°
6    | 0.069385           | -14.96°
7    | 0.352214           | -4.71°
```



Choose the Faulted Bus

Sort by ( ) Name [✓ Number]

```
1 (1)  [20.00 kV]
2 (2)  [230.0 kV]
3 (3)  [230.0 kV]
4 (4)  [230.0 kV]
5 (5)  [230.0 kV]
6 (6)  [230.0 kV]
7 (7)  [18.00 kV]
8 (FaultPt)  [230.0 kV]
```

Fault Location: [✓ Bus Fault] ( ) In-Line Fault

Location % 47

Fault Type: ( ) Single Line-to-Ground ( ) Line-to-Line [✓ 3 Phase Balanced] ( ) Double Line-to-Ground

Fault Impedance: R : 0.00000 X : 0.00000

Fault Current

Scale Current By: 1.00000

If Magnitude: 11.374 p.u.

If Scaled Mag: 11.374 p.u.

If Angle: -76.54 deg.

Units: [✓ p.u.] ( ) Amps

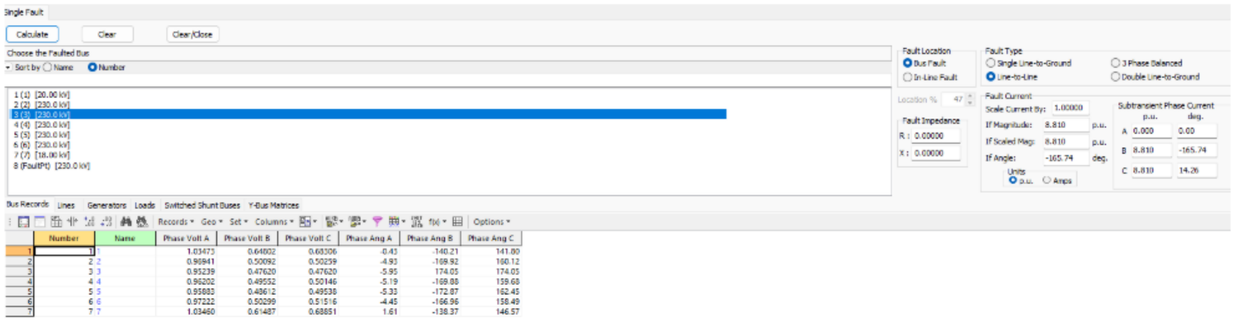
Subtransient Phase Current

	p.u.	deg.
A	11.374	-76.54
B	11.374	163.46
C	11.374	43.46

	Number	Name	Phase Volt A	Phase Volt B	Phase Volt C	Phase Ang A	Phase Ang B	Phase Ang C
1	1	1	0.48312	0.48312	0.48312	2.21	-117.79	122.21
2	2	2	0.12689	0.12689	0.12689	-6.46	-126.46	113.54
3	3	3	0.05574	0.05574	0.05574	-7.27	-127.27	112.73
4	4	4	0.09290	0.09290	0.09290	-5.95	-125.95	114.05
5	5	5	0.00000	0.00000	0.00000	-178.25	61.75	-58.25
6	6	6	0.08037	0.08037	0.08037	-2.95	-122.95	117.05
7	7	7	0.40686	0.40686	0.40686	9.14	-110.86	129.14

<u>Fault(network,3,"ltl",0) - Line to line fault - bus 3</u>

```
================ Phase Voltages at Buses =================

Bus bus1:

  Phase A: 1.0463 ∠ 0.18°

  Phase B: 0.6702 ∠ -146.36°

  Phase C: 0.6115 ∠ 142.99°

--------------------------------------------

Bus bus2:

  Phase A: 1.0474 ∠ 0.17°

  Phase B: 0.5619 ∠ -168.59°

  Phase C: 0.5082 ∠ 167.73°

--------------------------------------------

Bus bus3:

  Phase A: 1.0476 ∠ 0.17°

  Phase B: 0.5238 ∠ -179.83°

  Phase C: 0.5238 ∠ -179.83°

--------------------------------------------

Bus bus4:
  Phase A: 1.0475 ∠ 0.17°
  Phase B: 0.5622 ∠ -168.56°
  Phase C: 0.5082 ∠ 167.69°
--------------------------------------------
Bus bus5:
  Phase A: 1.0476 ∠ 0.17°
  Phase B: 0.5527 ∠ -170.74°
  Phase C: 0.5094 ∠ 170.30°
--------------------------------------------
Bus bus6:
  Phase A: 1.0476 ∠ 0.17°
  Phase B: 0.5695 ∠ -167.08°
  Phase C: 0.5079 ∠ 165.85°
--------------------------------------------
Bus bus7:
  Phase A: 1.0474 ∠ 0.17°
  Phase B: 0.6542 ∠ -148.63°
  Phase C: 0.5939 ∠ 145.38°
--------------------------------------------
================ Fault Phase Currents =================
Phase A: 0.0000 ∠ 0.00°
Phase B: 7.5415 ∠ -175.99°
Phase C: 7.5415 ∠ 4.01°
```



Single Fault

Choose the Faulted Bus

Sort by ( ) Name [✓ Number]

1 (1) [20.00 kV]
2 (2) [230.0 kV]
3 (3) [230.0 kV]
4 (4) [230.0 kV]
5 (5) [230.0 kV]
6 (6) [230.0 kV]
7 (7) [18.00 kV]
8 (FaultPt) [230.0 kV]

Fault Location: [✓ Bus Fault] ( ) In-Line Fault

Location %: 47

Fault Type: ( ) Single Line-to-Ground [✓ Line-to-Line] ( ) 3 Phase Balanced ( ) Double Line-to-Ground

Fault Impedance
R : 0.00000
X : 0.00000

Fault Current
Scale Current By: 1.00000
If Magnitude: 8.810 p.u.
If Scaled Mag: 8.810 p.u.
If Angle: -165.74 deg.
Units: [✓ p.u.] ( ) Amps

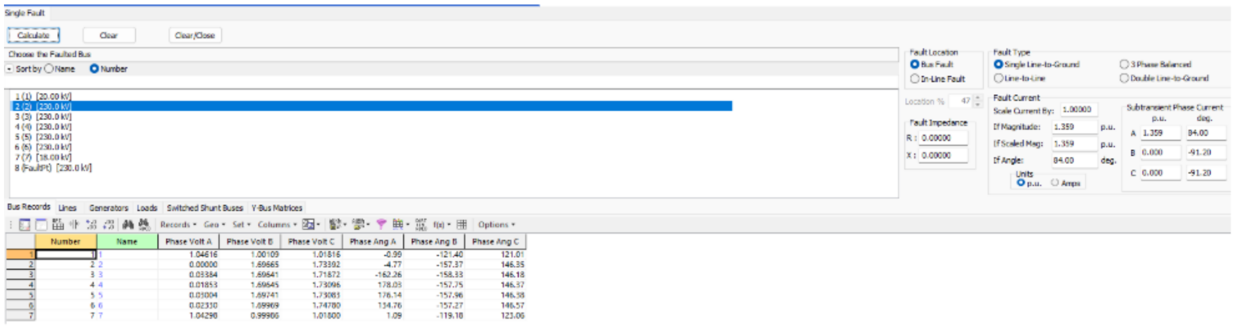
Subtransient Phase Current

	p.u.	deg.
A	0.000	0.00
B	8.810	-165.74
C	8.810	14.26

	Number	Name	Phase Volt A	Phase Volt B	Phase Volt C	Phase Ang A	Phase Ang B	Phase Ang C
1	1	1	1.03473	0.64802	0.68306	-0.43	-140.21	141.80
2	2	2	0.96941	0.50092	0.50259	-4.93	-169.92	160.12
3	3	3	0.95239	0.47620	0.47620	-5.95	174.05	174.05
4	4	4	0.96202	0.49552	0.50146	-5.19	-169.88	159.68
5	5	5	0.95883	0.48612	0.49538	-5.33	-172.87	162.45
6	6	6	0.97222	0.50299	0.51516	-4.45	-166.96	158.49
7	7	7	1.03460	0.61487	0.68851	1.61	-138.37	146.57

<u>Fault(network,2,"ltg",0) - single line to ground fault bus 2</u>

```
================ Phase Voltages at Buses =================

Bus bus1:

  Phase A: 1.0680 ∠ -0.06°

  Phase B: 1.0129 ∠ -121.81°

  Phase C: 1.0140 ∠ 121.78°

--------------------------------------------

Bus bus2:

  Phase A: 0.0000 ∠ 179.89°

  Phase B: 1.8619 ∠ -152.72°

  Phase C: 1.8694 ∠ 152.29°

--------------------------------------------

Bus bus3:

  Phase A: 0.0149 ∠ 161.82°

  Phase B: 1.8681 ∠ -152.92°

  Phase C: 1.8781 ∠ 152.33°

--------------------------------------------

Bus bus4:

  Phase A: 0.0122 ∠ 161.85°
  Phase B: 1.8662 ∠ -152.86°
  Phase C: 1.8755 ∠ 152.32°
--------------------------------------------
Bus bus5:
  Phase A: 0.0215 ∠ 161.85°
  Phase B: 1.8691 ∠ -152.95°
  Phase C: 1.8796 ∠ 152.34°
--------------------------------------------
Bus bus6:
  Phase A: 0.0237 ∠ 161.88°
  Phase B: 1.8676 ∠ -152.90°
  Phase C: 1.8775 ∠ 152.32°
--------------------------------------------
Bus bus7:
  Phase A: 1.0666 ∠ 0.00°
  Phase B: 1.0131 ∠ -121.76°
  Phase C: 1.0132 ∠ 121.76°
--------------------------------------------
================ Fault Phase Currents =================
Phase A: 1.4869 ∠ 89.57°
Phase B: 0.0000 ∠ -170.54°
Phase C: 0.0000 ∠ -161.57°
============================================================
```



Choose the Faulted Bus

Sort by ( ) Name [✓ Number]

1 (1) [20.00 kV]
2 (2) [230.0 kV]
3 (3) [230.0 kV]
4 (4) [230.0 kV]
5 (5) [230.0 kV]
6 (6) [230.0 kV]
7 (7) [18.00 kV]
8 (FaultPt) [230.0 kV]

Fault Location: [✓ Bus Fault] ( ) In-Line Fault

Location % 47

Fault Impedance R: 0.00000 X: 0.00000

Fault Type: [✓ Single Line-to-Ground] ( ) Line-to-Line ( ) 3 Phase Balanced ( ) Double Line-to-Ground

Fault Current

Scale Current By: 1.00000

If Magnitude: 1.359 p.u.

If Scaled Mag: 1.359 p.u.

If Angle: 84.00 deg.

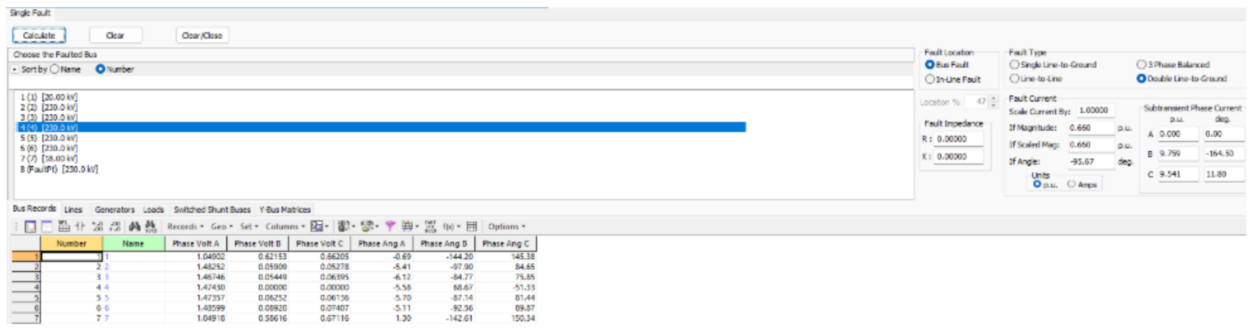
Units: [✓ p.u.] ( ) Amps

Subtransient Phase Current	p.u.	deg.
A	1.359	84.00
B	0.000	-91.20
C	0.000	-91.20

	Number	Name	Phase Volt A	Phase Volt B	Phase Volt C	Phase Ang A	Phase Ang B	Phase Ang C
1	1	1	1.04616	1.00109	1.01816	-0.99	-121.40	121.01
2	2	2	0.00000	1.69665	1.73392	-4.77	-157.37	146.35
3	3	3	0.03384	1.69641	1.71872	-162.26	-158.33	146.18
4	4	4	0.01853	1.69645	1.73096	178.03	-157.75	146.37
5	5	5	0.03004	1.69741	1.73083	176.14	-157.96	146.58
6	6	6	0.02330	1.69969	1.74780	134.76	-157.27	146.57
7	7	7	1.04298	0.99986	1.01800	1.09	-119.18	123.06

<u>Fault(network,4,"dltg",0) - double line to ground fault bus 4</u>

```
================ Phase Voltages at Buses =================

Bus bus1:

  Phase A: 1.0493 ∠ 2.13°

  Phase B: 0.6861 ∠ -148.12°

  Phase C: 0.5672 ∠ 145.24°

--------------------------------------------

Bus bus2:

  Phase A: 1.5930 ∠ 2.03°

  Phase B: 0.1063 ∠ -115.32°

  Phase C: 0.0847 ∠ 27.68°

--------------------------------------------

Bus bus3:

  Phase A: 1.5958 ∠ 1.99°

  Phase B: 0.1067 ∠ -113.80°

  Phase C: 0.0871 ∠ 26.78°

--------------------------------------------

Bus bus4:

  Phase A: 1.5921 ∠ 2.16°
  Phase B: 0.0632 ∠ -127.31°
  Phase C: 0.0632 ∠ -7.31°
----------------------------------------
Bus bus5:
  Phase A: 1.5953 ∠ 1.99°
  Phase B: 0.1088 ∠ -113.84°
  Phase C: 0.0882 ∠ 28.11°
----------------------------------------
Bus bus6:
  Phase A: 1.5941 ∠ 1.97°
  Phase B: 0.1247 ∠ -112.71°
  Phase C: 0.0997 ∠ 36.03°
----------------------------------------
Bus bus7:
  Phase A: 1.0506 ∠ 2.23°
  Phase B: 0.6726 ∠ -150.39°
  Phase C: 0.5488 ∠ 147.93°
----------------------------------------
================ Fault Phase Currents =================
Phase A: 0.5665 ∠ -154.60°
Phase B: 7.7474 ∠ -171.93°
Phase C: 8.2044 ∠ 4.24°
========================================================
```



Choose the Faulted Bus

Sort by [ ] Name [✓ Number]

1 (1) [20.00 kV]
2 (2) [230.0 kV]
3 (3) [230.0 kV]
4 (4) [230.0 kV]
5 (5) [230.0 kV]
6 (6) [230.0 kV]
7 (7) [18.00 kV]
8 (FaultPt) [230.0 kV]

Fault Location: [✓ Bus Fault] [ ] In-Line Fault

Location % 47

Fault Impedance
R : 0.00000
X : 0.00000

Fault Type: [ ] Single Line-to-Ground [ ] Line-to-Line [ ] 3 Phase Balanced [✓ Double Line-to-Ground]

Fault Current
Scale Current By: 1.00000
If Magnitude: 0.660 p.u.
If Scaled Mag: 0.660 p.u.
If Angle: -95.67 deg.
Units [✓ p.u.] [ ] Amps

Subtransient Phase Current	p.u.	deg.
A	0.000	0.00
B	9.759	-164.50
C	9.541	11.80

	Number	Name	Phase Volt A	Phase Volt B	Phase Volt C	Phase Ang A	Phase Ang B	Phase Ang C
1	1	1	1.04902	0.62153	0.66205	-0.69	-144.20	145.38
2	2	2	1.48252	0.05909	0.05278	-5.41	-97.90	84.65
3	3	3	1.46746	0.05449	0.06395	-6.12	-84.77	75.85
4	4	4	1.47430	0.00000	0.00000	-5.58	68.67	-51.33
5	5	5	1.47357	0.06252	0.06136	-5.70	-87.14	81.44
6	6	6	1.48599	0.08920	0.07407	-5.11	-92.56	89.87
7	7	7	1.04918	0.58616	0.67116	1.30	-142.61	150.34

Y1 Matrix:

```
----------------------------------------------------------------------------------------------------
-----
1.4633 + -22.9662j      -1.4633 + 14.6329j      0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j

-1.4633 + 14.6329j      37.7775 + -127.0505j    -10.3755 + 32.1379j     -25.9387 + 80.3447j     0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j

0.0000 + 0.0000j    -10.3755 + 32.1379j     23.3448 + -72.2267j     0.0000 + 0.0000j    -12.9693 + 40.1724j     0.0000 + 0.0000j    0.0000 + 0.0000j

0.0000 + 0.0000j    -25.9387 + 80.3447j     0.0000 + 0.0000j    46.3191 + -143.3520j    -7.4111 + 22.9556j      -12.9693 + 40.1724j     0.0000 + 0.0000j

0.0000 + 0.0000j    0.0000 + 0.0000j    -12.9693 + 40.1724j     -7.4111 + 22.9556j      46.3191 + -143.3520j    -25.9387 + 80.3447j     0.0000 + 0.0000j

0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    -12.9693 + 40.1724j     -25.9387 + 80.3447j     40.4899 + -139.4432j    -1.5818 + 18.9818j

0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    -1.5818 + 18.9818j      1.5818 + -27.3152j
----------------------------------------------------------------------------------------------------
-----
```

Y2 Matrix:

```
----------------------------------------------------------------------------------------------------
-----

1.4633 + -21.7758j          -1.4633 + 14.6329j          0.0000 + 0.0000j    0.0000 +
0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j

-1.4633 + 14.6329j          37.7775 + -127.0505j       -10.3755 + 32.1379j         -
25.9387 + 80.3447j         0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j

0.0000 + 0.0000j    -10.3755 + 32.1379j         23.3448 + -72.2267j         0.0000 +
0.0000j    -12.9693 + 40.1724j         0.0000 + 0.0000j    0.0000 + 0.0000j

0.0000 + 0.0000j    -25.9387 + 80.3447j         0.0000 + 0.0000j    46.3191 + -
143.3520j       -7.4111 + 22.9556j          -12.9693 + 40.1724j         0.0000 +
0.0000j

0.0000 + 0.0000j    0.0000 + 0.0000j    -12.9693 + 40.1724j         -7.4111 +
22.9556j          46.3191 + -143.3520j       -25.9387 + 80.3447j         0.0000 +
0.0000j

0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    -12.9693 + 40.1724j
-25.9387 + 80.3447j         40.4899 + -139.4432j       -1.5818 + 18.9818j

0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j
0.0000 + 0.0000j    -1.5818 + 18.9818j          1.5818 + -26.1247j

----------------------------------------------------------------------------------------------------
-----


Y0 Matrix:

----------------------------------------------------------------------------------------------------
-----

0.0000 + -20.0000j          0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j
0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j

0.0000 + 0.0000j    14.5257 + -44.9281j         -4.1502 + 12.8552j          -10.3755 +
32.1379j          0.0000 + 0.0000j    0.0000 + 0.0000j    0.0000 + 0.0000j
```

0.0000 + 0.0000j -4.1502 + 12.8552j   9.3379 + -28.8406j   0.0000 + 0.0000j -5.1877 + 16.0689j   0.0000 + 0.0000j 0.0000 + 0.0000j

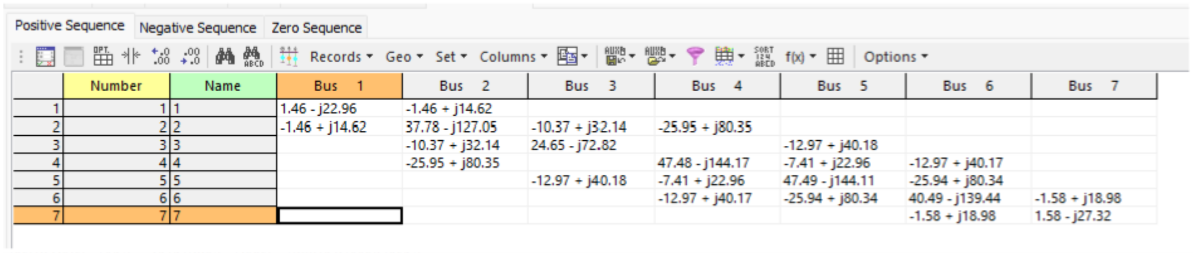
0.0000 + 0.0000j -10.3755 + 32.1379j   0.0000 + 0.0000j 18.5276 + -57.2684j   -2.9644 + 9.1823j -5.1877 + 16.0689j   0.0000 + 0.0000j

0.0000 + 0.0000j 0.0000 + 0.0000j -5.1877 + 16.0689j   -2.9644 + 9.1823j 18.5276 + -57.2684j   -10.3755 + 32.1379j   0.0000 + 0.0000j

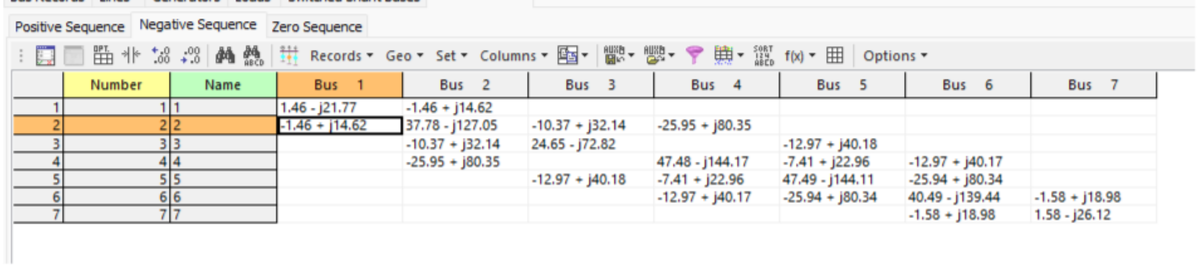
0.0000 + 0.0000j 0.0000 + 0.0000j 0.0000 + 0.0000j -5.1877 + 16.0689j -10.3755 + 32.1379j   15.5632 + -48.1511j   0.0000 + 0.0000j

0.0000 + 0.0000j 0.0000 + 0.0000j 0.0000 + 0.0000j 0.0000 + 0.0000j 0.0000 + 0.0000j 0.0000 + 0.0000j 3.1576 + -0.5116j

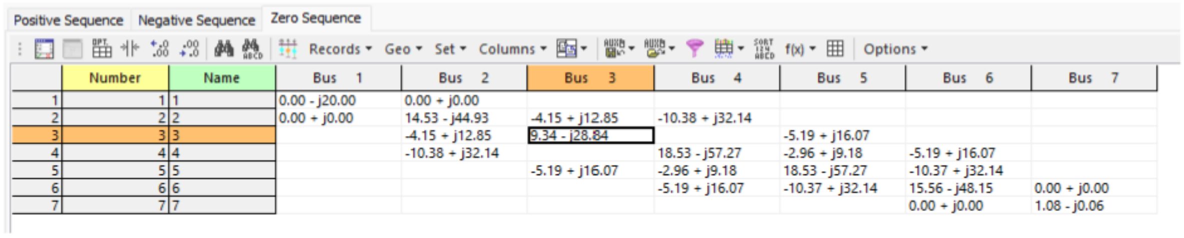
----


Positive Sequence | Negative Sequence | Zero Sequence

	Number	Name	Bus 1	Bus 2	Bus 3	Bus 4	Bus 5	Bus 6	Bus 7
1	1	1	1.46 - j22.96	-1.46 + j14.62					
2	2	2	-1.46 + j14.62	37.78 - j127.05	-10.37 + j32.14	-25.95 + j80.35			
3	3	3		-10.37 + j32.14	24.65 - j72.82		-12.97 + j40.18		
4	4	4		-25.95 + j80.35		47.48 - j144.17	-7.41 + j22.96	-12.97 + j40.17	
5	5	5			-12.97 + j40.18	-7.41 + j22.96	47.49 - j144.11	-25.94 + j80.34	
6	6	6				-12.97 + j40.17	-25.94 + j80.34	40.49 - j139.44	-1.58 + j18.98
7	7	7						-1.58 + j18.98	1.58 - j27.32


Positive Sequence | Negative Sequence | Zero Sequence

	Number	Name	Bus 1	Bus 2	Bus 3	Bus 4	Bus 5	Bus 6	Bus 7
1	1	1	1.46 - j21.77	-1.46 + j14.62					
2	2	2	-1.46 + j14.62	37.78 - j127.05	-10.37 + j32.14	-25.95 + j80.35			
3	3	3		-10.37 + j32.14	24.65 - j72.82		-12.97 + j40.18		
4	4	4		-25.95 + j80.35		47.48 - j144.17	-7.41 + j22.96	-12.97 + j40.17	
5	5	5			-12.97 + j40.18	-7.41 + j22.96	47.49 - j144.11	-25.94 + j80.34	
6	6	6				-12.97 + j40.17	-25.94 + j80.34	40.49 - j139.44	-1.58 + j18.98
7	7	7						-1.58 + j18.98	1.58 - j26.12


Positive Sequence | Negative Sequence | Zero Sequence

	Number	Name	Bus 1	Bus 2	Bus 3	Bus 4	Bus 5	Bus 6	Bus 7
1	1	1	0.00 - j20.00	0.00 + j0.00					
2	2	2	0.00 + j0.00	14.53 - j44.93	-4.15 + j12.85	-10.38 + j32.14			
3	3	3		-4.15 + j12.85	9.34 - j28.84		-5.19 + j16.07		
4	4	4		-10.38 + j32.14		18.53 - j57.27	-2.96 + j9.18	-5.19 + j16.07	
5	5	5			-5.19 + j16.07	-2.96 + j9.18	18.53 - j57.27	-10.37 + j32.14	
6	6	6				-5.19 + j16.07	-10.37 + j32.14	15.56 - j48.15	0.00 + j0.00
7	7	7						0.00 + j0.00	1.08 - j0.06

12. Summary.

Milestones 9, 10, and 11 focused on extending the power system simulation framework to incorporate advanced fault analysis and current calculations using symmetrical components and sequential networks. Building on the previous milestones establishing the Ybus matrix, load flow, and system structure, these final stages added a layer of complexity by integrating asymmetric fault analysis and fault current calculations.

In Milestone 9, we implemented the construction of positive, negative, and zero sequence lead matrices (Y1, Y2, Y0) for the entire system, including contributions from transmission lines, transformers, and generators. By inverting these matrices, the sequence impedance matrices (Z1, Z2, Z0) were calculated, providing the necessary foundation for asymmetric fault analysis.

In Milestone 10, we developed fault simulation algorithms for four fault types: three-phase (balanced) faults, phase-to-earth faults, phase-to-earth faults, and double phase-to-earth faults. Each fault scenario involved solving standard fault flow equations using the derived string networks, followed by calculating the post-fault bus voltage in both the string domain and the phase domain using the Fortescue transform. This milestone ensured accurate modeling of unbalanced faults in the network.

In Milestone 11, the simulation results were validated and presented, including the calculated fault currents, bus voltages, and phase voltages for each fault scenario. Debugging features such as a matrix visualization function were added to verify the internal calculations of the string network. The outputs from this milestone confirmed the theoretical consistency of the model with standard power system fault analysis methods.