Project 3 – Simulator Enhancement

Maria Luiza Hermann

1. Project Overview

The Power System Simulator is a Python-based tool for modeling and analyzing power systems through a modular, object-oriented design. Users can construct networks using components like buses, transmission lines, transformers, generators, and loads, and simulate system behavior under normal and faulted conditions using per-unit calculations. Core features include automated Ybus/Zbus matrix formation, Newton-Raphson power flow, and symmetrical/asymmetrical fault analysis with full sequence modeling.

To improve interpretability and user engagement, the simulator now includes graphical visualizations of voltage profiles, under the power flow and fault currents using matplotlib and networkx. These plots provide intuitive insights into system performance, line loading, and fault propagation. Designed as both a learning and analysis tool, the simulator reinforces key power system concepts while preparing users for real-world roles in planning, operations, and protection engineering. The key features of this simulator are:

- **Component-Based System Modeling:** Users can build networks using customizable components:
    - *Buses*: Automatically assigned as Slack, PV, or PQ based on connectivity and generators.
    - *Transmission Lines*: Use bundle and geometry data to calculate sequence impedances.
    - *Transformers*: Model all standard connection types (Y-Y, Y-Δ, Δ-Y, Δ-Δ) with configurable grounding.
    - *Loads and Generators*: Specify both real and reactive power levels; generator voltage control is supported.
- **Per-Unit System Calculations:** The simulator calculates all quantities using a consistent per-unit system normalized by a global Sbase (100 MVA), supporting voltage levels from distribution (20 kV) to transmission (230 kV and above).
- **Admittance Matrix Formation:** Constructs the complete Ybus, Ybus-positive, Ybus-negative, and Ybus-zero matrices by summing primitive admittances of all network elements.
- **Newton-Raphson Power Flow Solver:** Solves the nonlinear power flow equations by iteratively correcting voltage magnitudes and angles using a full Jacobian matrix. Supports PV, PQ, and Slack bus types with accurate ΔP and ΔQ mismatch handling.
- **Fault Analysis**
    - Symmetrical (3-phase) faults
    - Line-to-ground (LG) faults
    - Line-to-line (LL) faults
    - Double line-to-ground (LLG) faults

    - Fault studies use Zbus matrices for all three sequence networks and output post-fault voltages and fault current magnitudes.

- **Sequence Modeling:** Includes detailed positive, negative, and zero-sequence modeling for each component, which is critical for accurate fault analysis and stability evaluation.
- **Modularity and Extensibility:** Each component class is self-contained and extensible, allowing for future integration of features such as dynamic simulations, harmonic analysis, or protection logic.

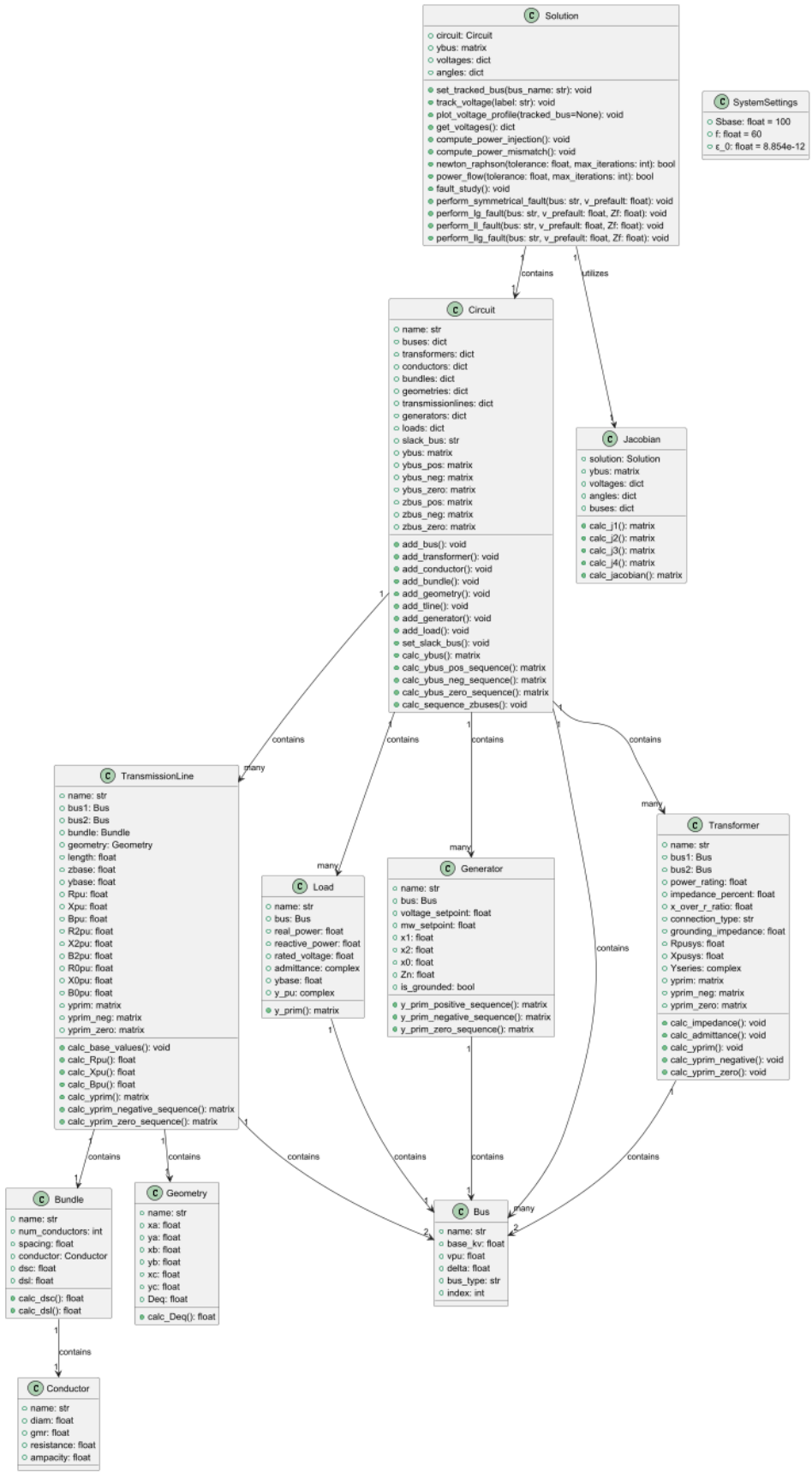
The enhancement Features are:

- **Voltage Profile:** This includes tracking of the voltage changes during the power flow and the fault analysis, and then plotting to be able to observe the results of these actions in the simulator.

One of the main reasons for plotting voltage profiles during power flow analysis is to get a clear picture of how the system behaves as it approaches a steady-state solution. Voltage magnitudes at each bus need to stay within safe operating limits (usually around ±5% of the nominal voltage) to ensure the proper functioning of equipment and the overall reliability of the grid. By tracking how voltage changes at a specific bus over the course of iterative methods like Newton-Raphson, engineers can see whether the system is converging correctly and identify any buses that might be causing issues. This approach is commonly recommended in power system analysis literature, as it helps validate both the numerical solution and the system configuration (Grainger & Stevenson, 1994).

Plotting voltage profiles before and after fault studies is equally important. It allows engineers to observe how a fault—whether it's a three-phase short circuit or an unbalanced line-to-ground fault—impacts the system voltages, particularly at sensitive or critical buses. These voltage dips can indicate how severe the fault is and help in designing better protection schemes. By comparing pre-fault and post-fault voltages visually, one can evaluate the system's ability to withstand and recover from disturbances, a practice grounded in symmetrical component theory and fault analysis methods (Anderson, 1995). This kind of analysis supports decisions related to system reinforcement and is essential for improving system resilience.

2. Class Diagrams:


Solution
circuit: Circuit
ybus: matrix
voltages: dict
angles: dict
set_tracked_bus(bus_name: str): void
track_voltage(label: str): void
plot_voltage_profile(tracked_bus=None): void
get_voltages(): dict
compute_power_injection(): void
compute_power_mismatch(): void
newton_raphson(tolerance: float, max_iterations: int): bool
power_flow(tolerance: float, max_iterations: int): bool
fault_study(): void
perform_symmetrical_fault(bus: str, v_prefault: float): void
perform_lg_fault(bus: str, v_prefault: float, Zf: float): void
perform_ll_fault(bus: str, v_prefault: float, Zf: float): void
perform_llg_fault(bus: str, v_prefault: float, Zf: float): void
SystemSettings
Sbase: float = 100
f: float = 60
ε_0: float = 8.854e-12
contains
utilizes
Circuit
name: str
buses: dict
transformers: dict
conductors: dict
bundles: dict
geometries: dict
transmissionlines: dict
generators: dict
loads: dict
slack_bus: str
ybus: matrix
ybus_pos: matrix
ybus_neg: matrix
ybus_zero: matrix
zbus_pos: matrix
zbus_neg: matrix
zbus_zero: matrix
add_bus(): void
add_transformer(): void
add_conductor(): void
add_bundle(): void
add_geometry(): void
add_tline(): void
add_generator(): void
add_load(): void
set_slack_bus(): void
calc_ybus(): matrix
calc_ybus_pos_sequence(): matrix
calc_ybus_neg_sequence(): matrix
calc_ybus_zero_sequence(): matrix
calc_sequence_zbuses(): void
Jacobian
solution: Solution
ybus: matrix
voltages: dict
angles: dict
buses: dict
calc_j1(): matrix
calc_j2(): matrix
calc_j3(): matrix
calc_j4(): matrix
calc_jacobian(): matrix
TransmissionLine
name: str
bus1: Bus
bus2: Bus
bundle: Bundle
geometry: Geometry
length: float
zbase: float
ybase: float
Rpu: float
Xpu: float
Bpu: float
R2pu: float
X2pu: float
B2pu: float
R0pu: float
X0pu: float
B0pu: float
yprim: matrix
yprim_neg: matrix
yprim_zero: matrix
calc_base_values(): void
calc_Rpu(): float
calc_Xpu(): float
calc_Bpu(): float
calc_yprim(): matrix
calc_yprim_negative_sequence(): matrix
calc_yprim_zero_sequence(): matrix
Load
name: str
bus: Bus
real_power: float
reactive_power: float
rated_voltage: float
admittance: complex
ybase: float
y_pu: complex
y_prim(): matrix
Generator
name: str
bus: Bus
voltage_setpoint: float
mw_setpoint: float
x1: float
x2: float
x0: float
Zn: float
is_grounded: bool
y_prim_positive_sequence(): matrix
y_prim_negative_sequence(): matrix
y_prim_zero_sequence(): matrix
Transformer
name: str
bus1: Bus
bus2: Bus
power_rating: float
impedance_percent: float
x_over_r_ratio: float
connection_type: str
grounding_impedance: float
Rpusys: float
Xpusys: float
Yseries: complex
yprim: matrix
yprim_neg: matrix
yprim_zero: matrix
calc_impedance(): void
calc_admittance(): void
calc_yprim(): void
calc_yprim_negative(): void
calc_yprim_zero(): void
Bundle
name: str
num_conductors: int
spacing: float
conductor: Conductor
dsc: float
dsl: float
calc_dsc(): float
calc_dsl(): float
Geometry
name: str
xa: float
ya: float
xb: float
yb: float
xc: float
yc: float
Deq: float
calc_Deq(): float
Bus
name: str
base_kv: float
vpu: float
delta: float
bus_type: str
index: int
Conductor
name: str
diam: float
gmr: float
resistance: float
ampacity: float
many

- Bus Object

Bus Object		
Purpose	**Method**	**Explanation**
Initialize a new Bus object.	init(self, name: str, base_kv: float)	Sets up a bus with its defining properties.
	Attribute	**Distribution**
	self.name = name	The name of the bus, provided by the user when defining the object.
	self.base_kv = base_kv	The base_kv of the bus, provided by the user when defining the object.
	self.vpu = 1	The vpu of the bus assumed to be 1.
	self.delta = 0	The delta of the bus is assumed to be 0.
	self.bust_type = “PQ Bus”	The bus type always starts as a PQ bus and may change if connected to a generator.
	self.index = bus.instance_count	The index is a number given to each bus when added to keep track of amount and order.

- Transformer Object

Transformer Object		
Purpose	**Method**	**Explanation**
Initialize a new Transformer object.	init(self, name: str, bus1: Bus, bus2: Bus, power_rating: float, impedance_percent: float, x_over_r_ratio: float, connection_type: str, grounding_impedance: float)	Sets up a transformer with its properties, power rating, impedance percent, x over r ratio, and associated buses.
	Attribute	**Distribution**
	self.name = name	The name of the transformer, provided by the user.
	self.bus1 = Bus	The first bus connected to the transformer.
	self.bus2 = Bus	The second bus connected to the transformer.

	self.power_rating = power_rating	The power rating in the transformer, provided by the user.
	self.impedance_percent = impedance_percent	The impedance percent in the transformer, provided by the user.
	self.x_over_r_ratio = x_over_r_ratio	The x over r ratio in the transformer, provided by the user.
	self.Rpusys, self.Xpusys = self.calc_impedance()	The Rpusys and Xpusys of the transformer, calculated using calc_impedance.
	self.Yseries = self.calc_admittance()	The Yseries of the transformer, calculated using calc_admittance.
	self.yprim = self.calc_yprim()	The yprim of the transformer, calculated using calc_yprim.
	self.connection_type = connection_type.upper()	The transformer connection type (e.g., “Y-Y”, “Y-DELTA”, etc.), used for zero-sequence modeling.
	self.Zn = grounding_impedance	The grounding impedance in ohms, provided by the user; converted to per-unit during zero-sequence calculations.
	self.yprim_neg = self.calc_yprim_negative()	The negative sequence Yprim matrix, identical to the positive sequence for this model.
	self.yprim_zero = self.calc_yprim_zero()	The zero sequence Yprim matrix, which varies based on connection type and grounding impedance.
Purpose	**Method**	**Description**
Calculates the Zbase and Ybase.	calc_impedance(self)	Calculates the r_pu and x_pu of the transformer, using the power rating, impedance percent and x over r ratio given by the user and the Sbase which is always assumed 100 MVA.
Calculates the admittance.	calc_admittance(self)	Calculates the Yseries of the transformer using the r_pu and x_pu calculated before.

Calculates the positive sequence Y prim per-unit.	calc_yprim(self)	Calculates the positive sequence of the yprim of the transformer using the Yseries calculated before.
Calculates the negative sequence Y prim per-unit.	calc_yprim_negative(self)	Calculates the negative sequence of the yprim of the transformer using the Yseries calculated before.
Calculates the zero sequence Y prim per-unit.	calc_yprim_zero(self)	Calculates the zero sequence of the yprim of the transformer using connection type. grounding impedance, and the Yseries calculated before; behavior differs by transformer configuration.

- Conductor Object

Conductor Object		
Purpose	**Method**	**Explanation**
Initialize a new Conductor object.	init(self, name: str, diam: float, gmr: float, resistance: float, ampacity: float)	Sets up a conductor with its defining properties.
	Attribute	**Distribution**
	self.name = name	The name of the conductor, provided by the user when defining the object.
	self.diam = diam	The conductor diameter in inches.
	self.gmr = gmr	The geometric mean radius (GMR) in feet.
	self.resistance = resistance	The resistance per unit length (measured in ohms per mile).
	self.ampacity = ampacity	The maximum current-carrying capacity of the conductor in amperes.

- Bundle Object

Bundle Object		
Purpose	**Method**	**Explanation**

Initialize a new Bundle object.	init(self, name: str, num_conductors: float, spacing: float, conductor: Conductor)	Sets up a bundle of conductors with defined properties. Calculates DSC and DSL upon initialization.
	Attribute	**Distribution**
	self.name = name	The name of the conductor, provided by the user when defining the object.
	self.num_conductors = num_conductors	The spacing between conductors in the bundle.
	self.conductor = conductor	A Conductor object representing the type of conductor used in the bundle.
	self.dsc = self.calc_dsc()	The equivalent self-GMD (DSC) of the bundle, calculated using the calc_dsc method.
	self.dsl = self.calc_dsl()	The equivalent mutual-GMD (DSL) of the bundle, calculated using the calc_dsl method.
Purpose	**Method**	**Description**
Calculate the DSL.	calc_dsl(self)	Calculates the geometric mean distance (DSL) for the bundle based on the number of conductors and their spacing.
Calculate the DSC.	calc_dsc(self)	Calculates the self-GMD (DSC) for the bundle based on conductor radius and spacing.

- Geometry Object

Geometry Object		
Purpose	**Method**	**Explanation**
Initialize a new Geometry object.	init(self, name: str , xa: float, ya: float, xb: float, yb: float, xc: float, yc: float):	Sets up a geometric configuration using three coordinate points. Calculates the equivalent distance (Deq) upon initialization.
	Attribute	**Distribution**
	self.name = name	The name of the geometry configuration, provided by the user.

	self.xa = xa, self.ya = ya	The x and y coordinates of point A.
	self.xb = xb, self.yb = y	The x and y coordinates of point B.
	self.xc = xc, self.yc = yc	The x and y coordinates of point C.
	self.Deq = self.calc_Deq()	The equivalent distance (Deq) of the configuration, calculated using the calc_Deq method.
Purpose	**Method**	**Description**
Calculates the Deq.	calc_Deq(self)	Calculates the equivalent distance (Deq) using the geometric mean of the distances between the three points.

- Transmission Line Object

Transmission Line Object		
Purpose	**Method**	**Explanation**
Initialize a new Transmission Line object.	init(self, name: str, bus1: Bus, bus2: Bus, bundle: Bundle, geometry: Geometry, length: float)	Sets up a transmission line with its properties, including bundle, geometry, length, and associated buses.
	Attribute	**Description**
	self.name = name	The name of the transmission line, provided by the user.
	Self.bus1 = bus1	The first bus connected to the transmission line.
	self.bus2 = bus2	The second bus connected to the transmission line.
	Self.bundle = bundle	A Bundle object representing the conductor bundle used in the transmission line.
	Self.geometry = geometry	A Geometry object representing the spatial configuration of the transmission line.
	self.length = length	The length of the transmission line in miles.

	self.zbase = calc_base_values()	The base impedance of the transmission line, calculated using calc_base_values.
	self.ybase = calc_base_values()	The base admittance of the transmission line, calculated using calc_base_values.
	self.Rpu = calc_Rpu()	The per-unit resistance of the transmission line, calculated using calc_Rpu.
	self.Xpu = calc_Xpu()	The per-unit reactance of the transmission line, calculated using calc_Xpu.
	self.Bpu = calc_Bpu()	The per-unit susceptance of the transmission line, calculated using calc_Bpu.
	self.R2pu = self.Rpu	The negative-sequence per-unit resistance (assumed same as positive).
	self.X2pu = self.Xpu	The negative-sequence per-unit reactance (same as positive
	self.B2pu = self.Bpu	The negative-sequence per-unit susceptance (same as positive)
	self.R0pu = 2.5 * self.Rpu	The zero-sequence per-unit resistance (approximated as 2.5x positive-sequence).
	self.X0pu = 2.5 * self.Xpu	The zero-sequence per-unit reactance (approximated as 2.5x positive-sequence)
	self.B0pu = self.Bpu	The zero-sequence per-unit susceptance (same as positive-sequence).
	self.yprim = calc_yprim()	The positive-sequence primitive admittance matrix of the transmission line.
	self.yprim_neg = self.calc_yprim_negative_sequence()	The negative-sequence primitive admittance matrix of the transmission line.

	self.yprim_zero = self.calc_yprim_zero_sequence()	The zero-sequence primitive admittance matrix of the transmission line.
Purpose	**Method**	**Explanation**
Calculates the Zbase and Ybase.	calc_base_values(self)	Calculates the base impedance and admittance based on system voltage and base power.
Calculates the R per-unit.	calc_Rpu(self)	Calculates the per-unit resistance of the transmission line.
Calculates the X per-unit.	calc_Xpu(self)	Calculates the per-unit reactance of the transmission line using inductive properties.
Calculates the Z per-unit.	calc_Bpu(self)	Calculates the per-unit susceptance of the transmission line.
Forms the primitive Y matrix.	calc_yprim(self)	Creates the positive-sequence Yprim matrix.
Forms negative-sequence Yprim.	calc_yprim_negative_sequence(self)	Creates the negative-sequence Yprim matrix (same form as positive).
Forms zero-sequence Yprim.	calc_yprim_zero_sequence(self)	Creates the zero-sequence Yprim matrix using 2.5× R/X approximations and B0.

- Load Object

Load Object		
Purpose	**Method**	**Explanation**
Initialize a new Load object.	init(self, name: str, bus: Bus, real_power: float, reactive_power: float)	Sets up a load with its defining properties.
	Attribute	**Description**
	self.name = name	The name of the load, provided by the user when defining the object.
	self.bus = bus	The Bus object where the load is connected.

	self.real_power = real_power	The real power of the load (MW).
	self.reactive_power = reactive_power	The reactive power of the load (MVAR).
	Self.rated_voltage = bus.base_kv	The base voltage of the connected bus (kV).
	self.admittance = (self.real_power - 1j*self.reactive_power)/ (self.rated_voltage**2)	The complex admittance calculated from real and reactive power, not in per unit.
	self.ybase = SystemSettings.Sbase / self.rated_voltage**2	The admittance base, calculated from system base power and bus voltage.
	self.y_pu = self.admittance / self.ybase	The per-unit complex admittance of the load
Purpose	**Method**	**Explanation**
Creates the primitive admittance matrix.	y_prim(self)	Returns a 1×1 primitive admittance matrix for the load using its per-unit admittance. Used in Y-bus assembly.

- Generator Object

Generator Object		
Purpose	**Method**	**Explanation**
Initialize a new Generator object.	init(self, name: str, bus: Bus, voltage_setpoint: float, mw_setpoint: float)	Sets up a generator with its defining properties.
	Attribute	**Description**
	self.name = name	The name of the generator, provided by the user when defining the object.
	self.bus = bus	The bus connected to the generator, provided by the user when defining the object.
	self.voltage_setpoint = voltage_setpoint	The voltage setpoint of the generator provided by the user when defining the object.

	self.mw_setpoint = mw_setpoint	The mw setpoint of the generator provided by the user when defining the object.
	self.x1 = 0.12	Positive-sequence subtransient reactance (fixed at 0.12 pu).
	self.x1 = 0.14	Negative-sequence subtransient reactance (fixed at 0.14 pu).
	self.x0 = 0.05	Zero-sequence subtransient reactance (fixed at 0.05 pu).
	self.Zn = grounding_impedance	Grounding impedance in ohms, provided by the user. Converted to pu during zero-sequence admittance calculation.
	self.is_grounded = is_grounded	Boolean indicating whether the generator is grounded. If False, zero-sequence admittance is set to 0.
Purpose	**Method**	**Explanation**
Compute positive-sequence Yprim.	y_prim_positive_sequence(self)	Creates the generator's positive-sequence primitive admittance matrix using its subtransient reactance.
Compute negative-sequence Yprim.	y_prim_negative_sequence(self)	Creates the generator's negative-sequence primitive admittance matrix using its subtransient reactance.
Compute zero-sequence Yprim.	y_prim_zero_sequence(self)	Creates the generator's zero-sequence primitive admittance matrix based on grounding configuration and impedance.

- Circuit Object

Circuit Object		
Purpose	**Method**	**Explanation**
Initialize the Circuit object.	__init__(self, name: str)	Sets up a circuit with a name and initializes empty dictionaries for buses, transformers, conductors, bundles, geometries, transmission lines, generators, and loads. Also initializes the Y-bus matrix.

	Attribute	**Description**
	self.name = name	The name of the circuit, provided by the user.
	self.buses = {}	A dictionary storing Bus objects, indexed by their names.
	self.transformers = {}	A dictionary storing Transformer objects, indexed by their names.
	self.conductors = {}	A dictionary storing Conductor objects, indexed by their names.
	self.bundles = {}	A dictionary storing Bundle objects, indexed by their names.
	self.geometries = {}	A dictionary storing Geometry objects, indexed by their names.
	self.transmissionlines = {}	A dictionary storing TransmissionLine objects, indexed by their names.
	self.generators = {}	A dictionary storing Generator objects, indexed by their names.
	self.loads = {}	A dictionary storing Load objects, indexed by their names.
	self.slack_bus = None	Stores the name of the slack bus, if assigned.
	self.ybus = self.calc_ybus()	Calls the method to calculate the Y-bus matrix for the circuit.
	self.ybus_pos = self.calc_ybus_pos_sequence()	Positive-sequence Y-bus matrix.
	self.ybus_neg = self.calc_ybus_neg_sequence()	Negative-sequence Y-bus matrix.
	self.ybus_zero = self.calc_ybus_zero_sequence()	Zero-sequence Y-bus matrix.
	self.zbus_pos, self.zbus_neg, self.zbus_zero	Sequence Z-bus matrices derived from inverting corresponding Y-bus matrices.
Purpose	**Method**	**Explanation**

Add a bus to the circuit.	add_bus(self, bus: str, base_kv: float)	Adds a Bus object to the circuit using a specified name and base voltage level. Raises an error if the bus already exists.
Add a transformer to the circuit.	add_transformer(self, name: str, bus1_name: str, bus2_name: str, power_rating: float, impedance_percent: float, x_over_r_ratio: float)	Adds a Transformer object between two existing buses. Ensures both buses exist before creating the transformer.
Add a conductor type.	add_conductor(self, name: str, diam: float, gmr: float, resistance: float, ampacity: float)	Creates a new Conductor object with specified physical and electrical properties. Ensures unique conductor names.
Add a bundle type.	add_bundle(self, name: str, num_conductors: int, spacing: float, conductor_name: str)	Defines a bundle using an existing conductor type. Ensures that the conductor exists before creating the bundle.
Add a geometry type.	add_geometry(self, name: str, xa: float, ya: float, xb: float, yb: float, xc: float, yc: float)	Defines spatial positioning for conductors in a transmission line. Ensures unique geometry names.
Add a transmission line.	add_tline(self, name: str, bus1_name: str, bus2_name: str, bundle_name: str, geometry_name: str, length: float)	Adds a TransmissionLine object, ensuring the necessary buses, bundle, and geometry exist before creation.
Add a generator.	add_generator(self, name: str, bus: str, voltage_setpoint: float, mw_setpoint: float)	Adds a Generator object at a specified bus. Ensures the bus exists and sets bus type to "Slack Bus" if it is the first generator added.
Set the slack bus manually.	set_slack_bus(self, bus_name: str)	Assigns the slack bus for the circuit. Ensures that the bus exists and is connected to a generator. Updates the previous slack bus type to "PV Bus" if necessary.
Add a load to the circuit.	add_load(self, name: str, bus: str, real_power: float, reactive_power: float)	Adds a Load object to the specified bus with real and reactive power demands.
Build the Y-bus matrix.	calc_ybus(self)	Builds the overall admittance matrix by summing all transformer and transmission line admittances.
Build the positive-sequence Y-bus	calc_ybus_pos_sequence(self)	Builds the Y-bus matrix for the positive-sequence network, including generator contributions.

Build the negative-sequence Y-bus.	calc_ybus_neg_sequence(self)	Builds the Y-bus matrix for the negative-sequence network, including generator contributions.
Build the zero-sequence Y-bus.	calc_ybus_zero_sequence(self)	Builds the Y-bus matrix for the zero-sequence network, incorporating grounding and impedance effects.
Compute sequence Zbus matrices.	calc_sequence_zbuses(self)	Computes and stores the positive-, negative-, and zero-sequence impedance matrices by inverting the respective Y-bus matrices.

- Solution Class

Solution Class		
Purpose	**Method**	**Explanation**
Initialize the Solution class.	__init__(self, circuit: Circuit)	Sets up a solution class and uses the information from the circuit class.
	Attribute	**Description**
	self.circuit = circuit	Initializes the circuit that is being analysed by the solution class.
	self. ybus = circuit.ybus	Initializes Y-bus from the circuit.
	self.voltages, self.angles = self.get_voltages()	Initializes voltages and angles that will be calculated by get_voltages. Dictionary of per-unit voltage magnitudes for each bus and of voltage phase angles (in radians) for each bus.
	self.voltage_profile = {}	Initializes a dictionary that will

		keep values of voltage over time.
	self.tracked_bus = None	Initializes track bus that will be kept as the main bus during the voltage profile.
Purpose	**Method**	**Explanation**
Starts voltage and angles for the buses in the circuit.	get_voltages(self)	Returns two dictionaries: one for per-unit voltage magnitudes and another for voltage angles (in radians) for each bus, using data from the circuit object.
Enable voltage tracking at a specific bus	set_tracked_bus(self, bus_name)	Initializes voltage profile for the specified bus and logs the initial voltage.
Track voltage with label	track_voltage(self, label)	Stores the voltage at the tracked bus with a corresponding step label (e.g., 'Iteration 1').
Plot voltage profile of tracked bus	plot_voltage_profile(self, tracked_bus=None)	Prints and plots voltage magnitude history for the selected or default tracked bus.
Compute the real and reactive power injected at each bus based on current voltage estimates.	compute_power_injection(self)	Uses voltage magnitudes, angles, and Y-bus to calculate the complex power injection at each bus. Returns arrays for real (P) and reactive (Q) power.

Calculate mismatch between specified and computed power for buses.	compute_power_mismatch(self)	Computes mismatch vectors (ΔP and ΔQ) by subtracting calculated power from specified values. ΔP is for all non-slack buses, and ΔQ is only for PQ buses.
Run Newton-Raphson method to solve the power flow equations.	newton_raphson(self, tolerance=0.001, max_iterations=50)	Iteratively solves the nonlinear power flow equations using the Newton-Raphson method until the mismatch is within the tolerance or iteration limit is reached. Returns True if converged, otherwise False.
Wrapper function that starts the power flow solution process.	power_flow(self, tolerance=0.001, max_iterations=50)	Calls the Newton-Raphson method with specified parameters. Returns True if the solution converges, otherwise False.
Launch fault analysis menu.	fault_study(self)	CLI interface prompting user to select a type of fault study.
Perform 3-phase fault analysis.	perform_three_phase_fault(self)perform_symmetrical_fault(self, bus, v_prefault)	Computes the fault current and post-fault bus voltages for a balanced three-phase fault at a specified bus using the positive-sequence network.
Perform line-to-ground fault.	perform_lg_fault(self, bus, v_prefault, Zf)	Calculates the sequence currents and phase-to-neutral voltages at all buses for a single line-to-ground fault using

		all three sequence networks.
Perform line-to-line fault.	perform_ll_fault(self, bus, v_prefault, Zf)	Performs analysis of a line-to-line fault by solving the positive and negative sequence networks, and computing post-fault voltages at all buses.
Perform double-line-to-ground fault.	perform_llg_fault(self, bus, v_prefault, Zf)	Analyzes a fault between two lines and ground using all three sequence networks, and computes resulting fault currents and system voltages.

- Jacobian Object

Jacobian Class		
Purpose	**Method**	**Explanation**
Initialize the Jacobian class.	__init__(self, solution: Solution)	Sets up the Jacobian class using the solution object that contains Y-bus, voltage, angle, and circuit data.
	Attribute	**Description**
	self.solution = solution	Stores the Solution object used to access bus data and voltage estimates.
	self.ybus = solution.ybus	Initializes the admittance matrix (Y-bus) from the solution.
	self.voltages = solution.voltages	Initializes per-unit voltage magnitudes for each bus.
	self.angles = solution.angles	Initializes voltage phase angles (in radians) for each bus.
	self.buses = solution.circuit.buses	Initializes bus information from the Circuit object.

Purpose	Method	Explanation
Compute submatrix J1 (∂P/∂δ).	calc_j1(self, pv_pq_buses, all_bus)	Calculates the partial derivatives of real power injections with respect to bus voltage angles for PV and PQ buses.
Compute submatrix J2 (∂P/∂V).	calc_j2(self, pv_pq_buses, pq_buses, all_bus)	Calculates the partial derivatives of real power injections with respect to voltage magnitudes for PV and PQ buses.
Compute submatrix J3 (∂Q/∂δ).	calc_j3(self, pv_pq_buses, pq_buses, all_bus)	Calculates the partial derivatives of reactive power injections with respect to bus voltage angles for PQ buses.
Compute submatrix J4 (∂Q/∂V).	calc_j4(self, pq_buses, all_bus)	Calculates the partial derivatives of reactive power injections with respect to voltage magnitudes for PQ buses.
Assemble the full Jacobian matrix.	calc_jacobian(self)	Combines submatrices J1, J2, J3, and J4 to form the full Jacobian matrix used in Newton-Raphson iterations.

- System Settings Object

System Settings Object		
Purpose	**Method**	**Explanation**
Initialize all the system settings values.	Sbase = 100	Sets the system base [MVA]
	f = 60	Sets the frequency that will be used by the system in Hz.
	ε_0 = 8.854 * 10 ** -12	Sets the permittivity of free space (F/m)

3. Example Cases

<u>Validation of the Plot Voltage Profile 2 Different Examples</u>

The Solution object has been enhanced with the ability to track and plot the voltage magnitude of a specified bus throughout the Newton-Raphson power flow process and during fault analysis. This functionality helps visualize how the system converges during iterative load flow and how a bus voltage responds before and after a fault is applied. The enhancement provides deeper insight into convergence stability, sensitivity of critical buses, and system response to disturbances, thereby supporting both debugging and operational planning.

The Solution object includes the following attributes and methods related to voltage tracking:

- voltage_profile: Dictionary storing per-unit voltage magnitudes with labeled timestamps for a selected bus.
- tracked_bus: The name of the bus being monitored.
- set_tracked_bus(bus_name): Initializes the tracking process for a selected bus.
- track_voltage(label): Records the absolute voltage value of the tracked bus with an associated label (e.g., “Iteration 1”, “Before Fault”, “After Fault”).
- plot_voltage_profile(): Plots and prints the voltage trajectory for the tracked bus across the power flow and fault analysis stages.

Solution Process

Initialization:

- The circuit is initialized with voltage tracking enabled at a selected bus (e.g., Bus5).
- The Newton-Raphson algorithm is executed. After each iteration, the updated voltage at the tracked bus is recorded using track_voltage("Iteration i").
- Once the power flow converges, the solution proceeds with the fault analysis using the fault_study() method.

Voltage Logging Across Simulation:

- Before a fault is applied, the current steady-state voltage is recorded using track_voltage("Before Fault").
- After the fault is applied and bus voltages are recalculated using symmetrical component methods, the new voltage is logged using track_voltage("After Fault").
- This process is repeated for multiple types of faults (LG, LL, LLG), providing a labeled sequence of voltage states.

Problem Definition 1

The sample system used to validate the Newton-Raphson algorithm includes:

- Bus1: Slack Bus (20 kV)
- Bus2–Bus6: PQ Buses (230 kV)
- Bus7: PV Bus (18 kV)
- Transmission Lines:
  - 6 lines using "Partridge" conductor, 2-conductor bundle, and fixed geometry
  - Lengths range from 10 to 35 miles
- Transformers:
  - T1: Connects Bus1 to Bus2, 125 MVA, delta-Y grounded, 1 Ω grounding
  - T2: Connects Bus6 to Bus7, 200 MVA, delta-Y grounded, 999999 Ω grounding
- Generators:

    - G1 at Bus1: 20 MW, 100 MVAR
    - G2 at Bus7: 18 MW, 200 MVAR
- Loads:
    - L1 at Bus3: 110 MW, 50 MVAR
    - L2 at Bus4: 100 MW, 70 MVAR
    - L3 at Bus5: 100 MW, 65 MVAR

Use the main code below for this:

```
from Circuit import Circuit
from Solution import Solution

   # Step 1: create test circuit
   circuit1 = Circuit("Test Circuit")

   # Step 2: Load your Circuit
   # ADD TRANSMISSION LINES
   circuit1.add_conductor("Partridge", 0.642, 0.0217, 0.385, 460)
   circuit1.add_bundle("Bundle1", 2, 1.5, "Partridge")
   circuit1.add_geometry("Geometry1", 0, 0, 18.5, 0, 37, 0)

   circuit1.add_tline("Line1", "Bus2", "Bus4", "Bundle1", "Geometry1", 10)
   circuit1.add_tline("Line2", "Bus2", "Bus3", "Bundle1", "Geometry1", 25)
   circuit1.add_tline("Line3", "Bus3", "Bus5", "Bundle1", "Geometry1", 20)
   circuit1.add_tline("Line4", "Bus4", "Bus6", "Bundle1", "Geometry1", 20)
   circuit1.add_tline("Line5", "Bus5", "Bus6", "Bundle1", "Geometry1", 10)
   circuit1.add_tline("Line6", "Bus4", "Bus5", "Bundle1", "Geometry1", 35)

   # ADD TRANSMORMERS
   circuit1.add_transformer("T1", "Bus1", "Bus2", 125, 8.5, 10, "delta-y", 1)
   circuit1.add_transformer("T2", "Bus6", "Bus7", 200, 10.5, 12, "delta-y",
999999)

   # ADD GENERATORS
   circuit1.add_generator("G1", "Bus1", 20, 100, 0, True)
   circuit1.add_generator("G2", "Bus7", 18, 200, 1, True)

   # ADD LOAD
   circuit1.add_load("L1", "Bus3", 110, 50)
   circuit1.add_load("L2", "Bus4", 100, 70)
   circuit1.add_load("L3", "Bus5", 100, 65)

   # Step 3: Initialize Solution object
   solution = Solution(circuit1)

   # Step 4: Start voltage tracking for a specific bus
   solution.set_tracked_bus("Bus5")

   # Step 5: Run power flow (Newton-Raphson)
   solution.power_flow()  # Tracking will happen inside

   # Step 5: Run fault analysis (user input required during execution)
   solution.fault_study()
```

```
# Step 6: Plot voltage profile at the tracked bus
solution.plot_voltage_profile()
```

Expected Output

Iteration 1:
Max mismatch = 2.000000

Iteration 2:
Max mismatch = 0.127868

Iteration 3:
Max mismatch = 0.003035

Iteration 4:
Max mismatch = 0.000001

Iteration 5:
Max mismatch = 0.000000

Converged!

NEWTON-RAPHSON SOLUTION SUMMARY
==========

Converged in 5 iterations

Final Bus Angles (radians):
Bus1: 0.000000 rad
Bus2: -0.077577 rad
Bus3: -0.095161 rad
Bus4: -0.082034 rad
Bus5: -0.084293 rad
Bus6: -0.069086 rad
Bus7: 0.037416 rad

Final Bus Voltages (p.u.):
Bus1: 1.000000 p.u.
Bus2: 0.937102 p.u.
Bus3: 0.920804 p.u.
Bus4: 0.930048 p.u.
Bus5: 0.927005 p.u.
Bus6: 0.939857 p.u.

Bus7: 1.000000 p.u.

Final Power Mismatch:
Bus2: ΔP = 0.0000
Bus3: ΔP = -0.0000
Bus4: ΔP = -0.0000
Bus5: ΔP = -0.0000
Bus6: ΔP = 0.0000
Bus7: ΔP = 0.0000
Bus2: ΔQ = -0.0000
Bus3: ΔQ = 0.0000
Bus4: ΔQ = 0.0000
Bus5: ΔQ = -0.0000
Bus6: ΔQ = -0.0000

Final Jacobian Matrix:
∂δ Bus2 ∂δ Bus3 ∂δ Bus4 ∂δ Bus5 ∂δ Bus6 ∂δ Bus7 ∂V Bus2 ∂V Bus3 ∂V Bus4 ∂V Bus5 ∂V Bus6
∂P Bus2 112.434920 -28.130357 -70.739557 0.000000 -0.000000 -0.000000 36.071035 -9.378510 -24.446745 0.000000 0.000000
∂P Bus3 -27.809320 62.275559 -0.000000 -34.466239 -0.000000 -0.000000 -10.265178 20.724426 0.000000 -12.582029 0.000000
∂P Bus4 -70.534039 0.000000 125.783339 -19.979363 -35.269937 -0.000000 -24.934757 0.000000 42.851539 -6.979612 -12.786448
∂P Bus5 -0.000000 -34.711585 -19.949927 124.916345 -70.254833 -0.000000 0.000000 -11.850296 -7.053764 42.704319 -25.658167
∂P Bus6 0.000000 0.000000 -35.569281 -70.955661 124.106016 -17.581074 0.000000 0.000000 -11.935035 -23.697223 38.774318
∂P Bus7 0.000000 0.000000 0.000000 0.000000 -17.897148 17.897148 0.000000 0.000000 0.000000 0.000000 0.444942
∂Q Bus2 -33.802234 8.635765 22.736638 -0.000000 -0.000000 -0.000000 119.981535 -30.549791 -76.060143 0.000000 -0.000000
∂Q Bus3 9.619518 -21.283126 -0.000000 11.663608 -0.000000 -0.000000 -29.675878 66.545745 -0.000000 -37.180195 -0.000000
∂Q Bus4 23.366407 -0.000000 -41.853973 6.470137 12.017429 -0.000000 -75.268273 0.000000 133.738674 -21.552587 -37.526930
∂Q Bus5 -0.000000 10.911795 6.560336 -41.587131 24.115000 -0.000000 -0.000000 -37.697057 -21.450436 133.350200 -74.750578
∂Q Bus6 -0.000000 -0.000000 11.100151 21.967452 -36.442303 3.374700 0.000000 0.000000 -38.244579 -76.542885 132.047805

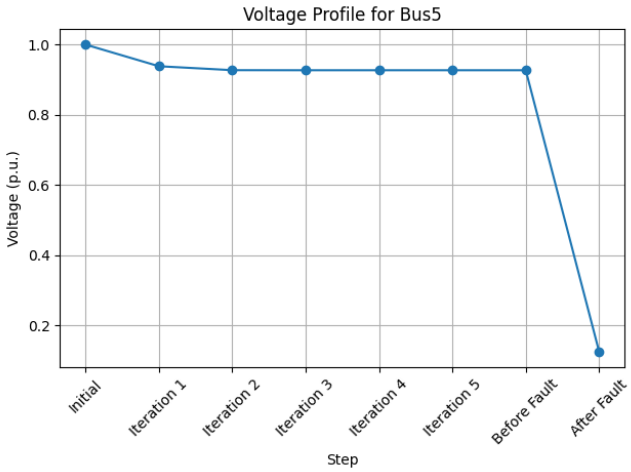
FAULT ANALYSIS
==========

Select fault type:

1. Three-phase fault
2. Line-to-ground fault
3. Line-to-line fault
4. Double-line-to-ground fault
Enter choice (1-4): 1
Available buses: ['Bus1', 'Bus2', 'Bus3', 'Bus4', 'Bus5', 'Bus6', 'Bus7']
Enter the bus name where the fault occurs: Bus3

>>> Performing symmetrical 3-phase fault analysis

Subtransient fault current at Bus3: 11.0929 p.u. ∠-72.51°
Post-fault Phase A voltage at Bus1: 0.4569 p.u. ∠-4.00°
Post-fault Phase A voltage at Bus2: 0.1483 p.u. ∠-7.41°
Post-fault Phase A voltage at Bus3: 0.0000 p.u. ∠0.00°
Post-fault Phase A voltage at Bus4: 0.1556 p.u. ∠-4.94°
Post-fault Phase A voltage at Bus5: 0.1256 p.u. ∠-4.05°
Post-fault Phase A voltage at Bus6: 0.1723 p.u. ∠-6.72°
Post-fault Phase A voltage at Bus7: 0.4245 p.u. ∠-3.87°
Voltage Profile for Bus5:
Initial → 1.000000 p.u.
Iteration 1 → 0.938429 p.u.
Iteration 2 → 0.927199 p.u.
Iteration 3 → 0.927005 p.u.
Iteration 4 → 0.927005 p.u.
Iteration 5 → 0.927005 p.u.
Before Fault → 0.927005 p.u.
After Fault → 0.125572 p.u.



Problem Definition 1

The sample system used to validate the Newton-Raphson algorithm includes:

- Bus1: Slack Bus (20 kV)

- Bus2–Bus6: PQ Buses (230 kV)
- Bus7: PV Bus (18 kV)
- Transmission Lines:
  - 6 lines using "Partridge" conductor, 2-conductor bundle, and fixed geometry
  - Lengths range from 10 to 35 miles
- Transformers:
  - T1: Connects Bus1 to Bus2, 125 MVA, delta-Y grounded, 1 Ω grounding
  - T2: Connects Bus6 to Bus7, 200 MVA, delta-Y grounded, 999999 Ω grounding
- Generators:
  - G1 at Bus1: 20 MW, 100 MVAR
  - G2 at Bus7: 18 MW, 200 MVAR
- Loads:
  - L1 at Bus3: 110 MW, 50 MVAR
  - L2 at Bus4: 100 MW, 70 MVAR
  - L3 at Bus5: 100 MW, 65 MVAR

Use the main code below for this:

```
from Circuit import Circuit
from Solution import Solution

   # Step 1: create test circuit
   circuit1 = Circuit("Test Circuit")

   # Step 2: Load your Circuit
   # ADD TRANSMISSION LINES
   circuit1.add_conductor("Partridge", 0.642, 0.0217, 0.385, 460)
   circuit1.add_bundle("Bundle1", 2, 1.5, "Partridge")
   circuit1.add_geometry("Geometry1", 0, 0, 18.5, 0, 37, 0)

   circuit1.add_tline("Line1", "Bus2", "Bus4", "Bundle1", "Geometry1", 10)
   circuit1.add_tline("Line2", "Bus2", "Bus3", "Bundle1", "Geometry1", 25)
   circuit1.add_tline("Line3", "Bus3", "Bus5", "Bundle1", "Geometry1", 20)
   circuit1.add_tline("Line4", "Bus4", "Bus6", "Bundle1", "Geometry1", 20)
   circuit1.add_tline("Line5", "Bus5", "Bus6", "Bundle1", "Geometry1", 10)
   circuit1.add_tline("Line6", "Bus4", "Bus5", "Bundle1", "Geometry1", 35)

   # ADD TRANSMORMERS
   circuit1.add_transformer("T1", "Bus1", "Bus2", 125, 8.5, 10, "delta-y", 1)
   circuit1.add_transformer("T2", "Bus6", "Bus7", 200, 10.5, 12, "delta-y",
999999)

   # ADD GENERATORS
   circuit1.add_generator("G1", "Bus1", 20, 100, 0, True)
   circuit1.add_generator("G2", "Bus7", 18, 200, 1, True)

   # ADD LOAD
   circuit1.add_load("L1", "Bus3", 110, 50)
   circuit1.add_load("L2", "Bus4", 100, 70)
   circuit1.add_load("L3", "Bus5", 100, 65)
```

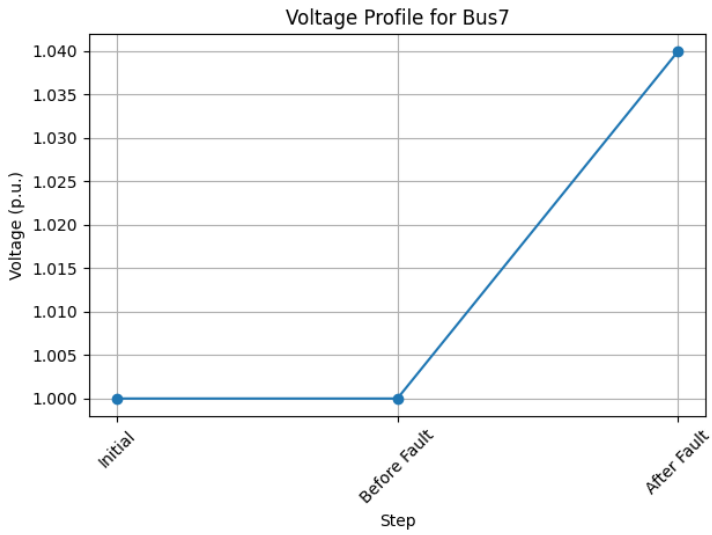
```
# Step 3: Initialize Solution object
solution = Solution(circuit1)

# Step 4: Start voltage tracking for a specific bus
solution.set_tracked_bus("Bus5")

# Step 5: Run power flow (Newton-Raphson)
# solution.power_flow()  # Tracking will happen inside

# Step 5: Run fault analysis (user input required during execution)
solution.fault_study()

# Step 6: Plot voltage profile at the tracked bus
solution.plot_voltage_profile()
```

Expected output:

FAULT ANALYSIS
==========

1. Three-phase fault
2. Line-to-ground fault
3. Line-to-line fault
4. Double-line-to-ground fault
Enter choice (1-4): 3
Enter the bus name where the fault occurs: Bus3

>>> Performing line-to-line (LL) fault analysis
Line current between A and B (Iab): 9.2564 p.u. ∠-72.04°
Post-fault Phase A voltage at Bus1: 1.0395 p.u. ∠-0.23°
Post-fault Phase A voltage at Bus2: 1.0372 p.u. ∠-0.40°
Post-fault Phase A voltage at Bus3: 1.0365 p.u. ∠-0.43°
Post-fault Phase A voltage at Bus4: 1.0369 p.u. ∠-0.41°
Post-fault Phase A voltage at Bus5: 1.0368 p.u. ∠-0.42°
Post-fault Phase A voltage at Bus6: 1.0373 p.u. ∠-0.40°
Post-fault Phase A voltage at Bus7: 1.0399 p.u. ∠-0.26°
Voltage Profile for Bus7:
Initial → 1.000000 p.u.
Before Fault → 1.000000 p.u.
After Fault → 1.039916 p.u.



4. **References:**

Grainger, J. J., & Stevenson, W. D. (1994). *Power system analysis*. McGraw-Hill.

Anderson, P. M. (1995). *Analysis of faulted power systems*. IEEE Press.

Other possible results to visualize:

1. Bus Voltage Classification Map

After running power flow, classify and color buses:

- Green: 0.95–1.05 p.u.
- Yellow: 0.9–0.95 or 1.05–1.1 p.u.
- Red: Outside normal range

Most utility grids require bus voltages to remain within 0.95–1.05 per unit. Going outside this band can damage equipment or trigger protection systems. **Fast anomaly detection**: Highlighting buses in red/yellow/green allows operators or students to immediately spot problems.

2. Network Graph Visualization (networkx)

Plot the system layout as a graph, with:

- **Nodes** as buses

- **Edges** as transmission lines / transformers
- **Node color** based on bus voltage (once Solution has solved it)

Debugging tool: You can quickly detect modeling mistakes — e.g., unconnected buses, incorrect transformer connections, or missing elements.