ECE 2774 - Advanced Power Systems Analysis

Project 2 - Group 8

Cecilia Espadas & Maria Luiza Hermann

1. Project Overview

The Power System Simulator is a Python-based tool designed to model and analyze power systems using a modular, object-oriented framework. It allows users to construct a network from individual components—including buses, transmission lines, transformers, generators, and loads—and simulate system behavior under both normal and faulted conditions. The simulator supports per-unit calculations and builds Ybus and Zbus matrices for power flow and fault analysis.

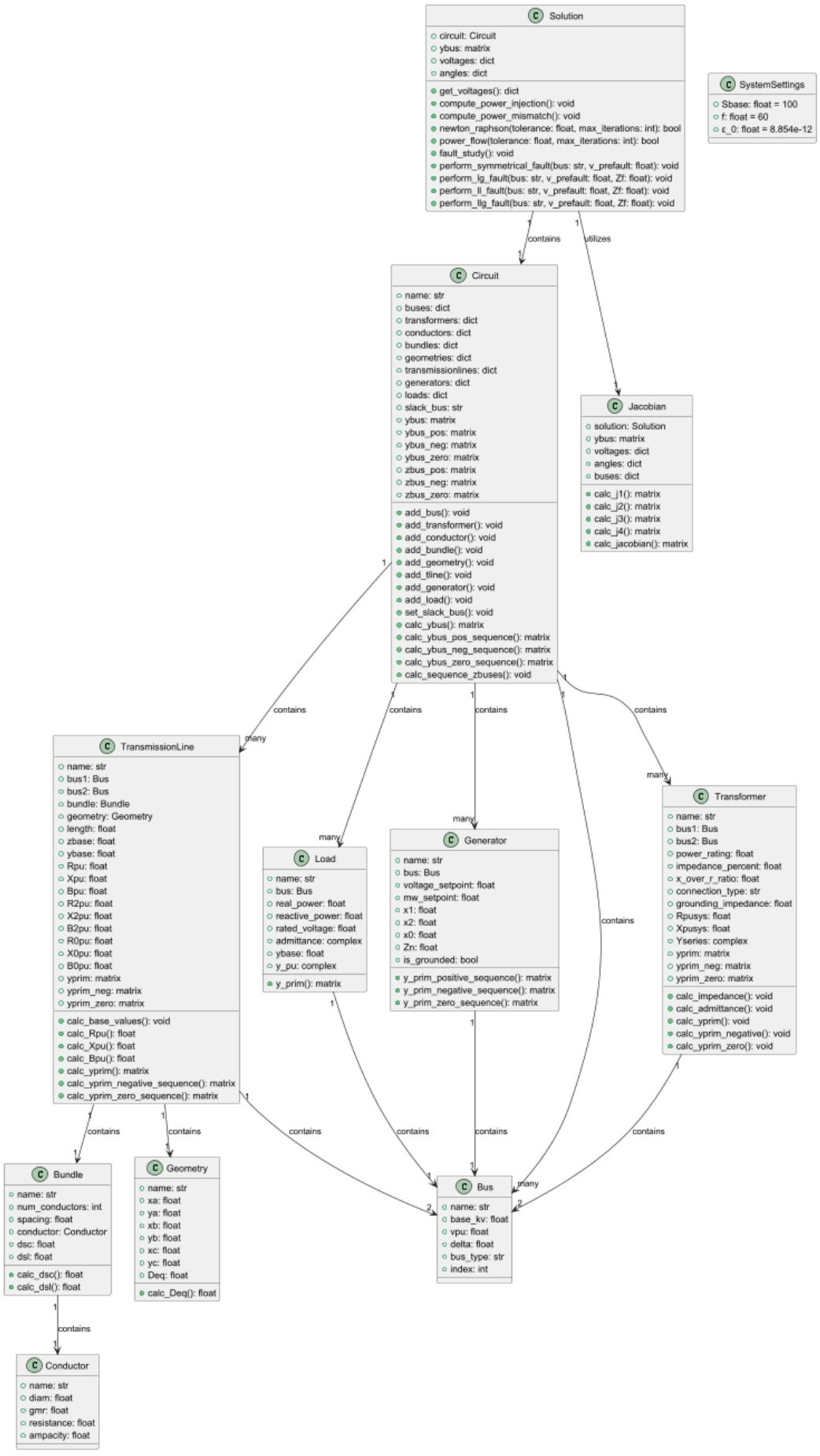
The purpose of the simulator is to give users a deeper understanding of how power systems behave by directly engaging with the equations and logic behind the analysis. Instead of using analysis softwares, students can build their own networks and apply methods like Newton-Raphson power flow and symmetrical or asymmetrical fault studies. This approach reinforces the relationships between system elements, system configuration, and real-world performance. 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.

Modern power systems are large, interconnected networks that require detailed analysis to ensure reliable operation. Solving these systems often involves nonlinear equations, iterative methods, and sequence network modeling to evaluate how the grid responds under different operating or faulted conditions. This simulator allows the user to run steady-state power flow and fault analysis while seeing every part of the process. It can model realistic scenarios like transformer grounding, short-circuit behavior, and generator response to faults. These features make it useful not only for understanding how a system behaves under normal conditions but also for seeing how faults propagate through the network. In a real-world setting, tools like this are used to evaluate voltage profiles, manage reactive power, design protective schemes, and verify system stability. By working directly with this simulator, users build the kind of system-level thinking and technical fluency needed for careers in transmission planning, protection engineering, and operations.

2. Class Diagrams


Solution
circuit: Circuit
ybus: matrix
voltages: dict
angles: dict
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

➢ 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 =	The negative sequence

	self.calc_yprim_negative()	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.
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.
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	perform_three_phase_fault	Computes the fault current

analysis.	(self)perform_symmetrical _fault(self, bus, v_prefault)	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	Sbase = 100	Sets the system base

settings values.		[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. Relevant Equations

- **Transformer**

Impedance Calculation (Zpu)

$$Z_{pu} = (\frac{S_{based}}{S_{rated}}) \times (\frac{Z_{percent}}{100}) \times e^{i\theta}$$

S_{based} is the system base power (typically 100 MVA).

S_{rated} is the transformer's rated power.

$Z_{percent}$ is the transformer's impedance in percentage.

$\theta = tan^{-1}(\frac{X}{R})$ is the impedance angle based on the X/R ratio.

$R_{pu} = Re(Z_{pu}) \quad X_{pu} = Im(Z_{pu})$

Admittance Calculation

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

R_{pu} = per-unit resistance of the transformer

X_{pu} = per-unit reactance of the transformer

Y_{series} = per-unit series admittance

Positive- and Negative-Sequence Y-Bus (Primitive Admittance Matrix) Formation

$$Y_{prim}^{(+/-)} = \begin{bmatrix} Y_{series} & -Y_{series} \\ -Y_{series} & Y_{series} \end{bmatrix}$$

Zero-Sequence Y-Bus (Primitive Admittance Matrix) Formations

$$Z_{n,pu} = \frac{Z_n}{Z_{pu}}$$

$$Y_g = \frac{1}{3Z_{n,pu}}$$

Z_{pu} = base impedance of the transformer

Z_n = grounding impedance in ohms (user-defined)

$Z_{n,pu}$ = grounding impedance converted to per-unit

Y_g = per-unit equivalent grounding admittance

- Y-Y
  - $Y_{11} = Y_{series} + Y_g, Y_{22} = Y_{series} + Y_g, Y_{12} = -Y_{series}$
- Y-Delta
  - $Y_{11} = Y_g, Y_{22} = 0, Y_{12} = 0$
- Delta-Y
  - $Y_{11} = 0, Y_{22} = Y_g, Y_{12} = 0$
- Delta-Delta
  - $Y_{11} = Y_{22} = Y_{12} = 0$

$Y_{prim} = \begin{bmatrix} Y_{11} & -Y_{12} \end{bmatrix}, \begin{bmatrix} -Y_{12} & Y_{22} \end{bmatrix}$

- **Bundle Object**

Equivalent Geometric Mean Radius (D_{SL})

N	D_{SL}
1	GMR_c
2	$\sqrt{GMR_c * d}$
3	$\sqrt[3]{GMR_c * d^2}$
4	$1.091 * \sqrt[4]{GMR_c * d^3}$

N = number of conductors in a bundle
GMR_c = geometric mean radius of the subconductor (feet)
d = distance between adjacent subconductors (feet)

Equivalent Radius (D_{SC})

N	D_{Sc}
1	r_c
2	$\sqrt{r_c * d}$
3	$\sqrt[3]{r_c * d^2}$
4	$1.091 * \sqrt[4]{r_c * d^3}$

N = number of conductors in a bundle
r_c = radius of the subconductor (feet)
Where r_c is the conductor diameter in inches divided by 24 in/ft
d = distance between adjacent subconductors (feet)

- **Geometry Object**

Equivalent Geometric Mean Distance (D_{eq})

$$D_{ab} = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

$$D_{bc} = \sqrt{(x_c - x_b)^2 + (y_c - y_b)^2}$$

$$D_{ca} = \sqrt{(x_a - x_c)^2 + (y_a - y_c)^2}$$

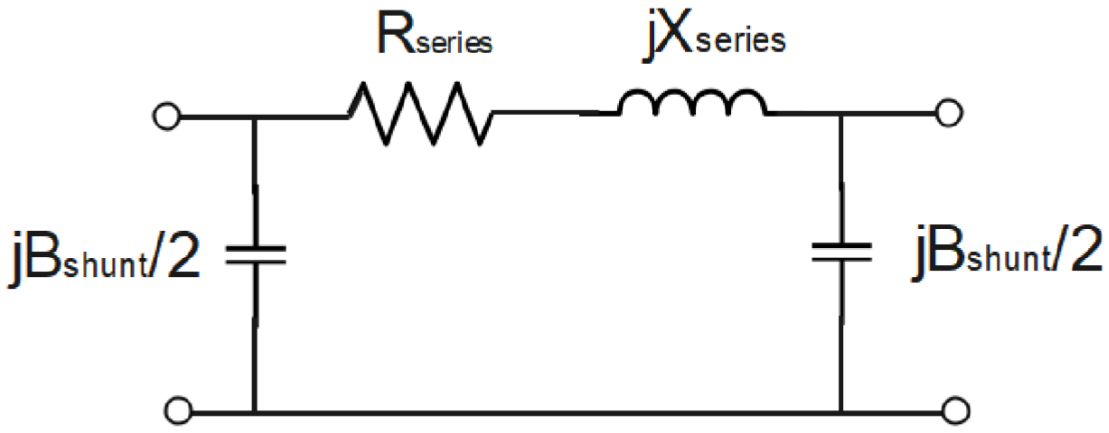
$$D_{eq} = \sqrt[3]{D_{ab} * D_{bc} * D_{ca}}$$

x_a= The x coordinate for phase A on a 2D plane (feet)
y_a= The y coordinate for phase A on a 2D plane (feet)
x_b= The x coordinate for phase B on a 2D plane (feet)
y_b= The y coordinate for phase B on a 2D plane (feet)
x_c= The x coordinate for phase C on a 2D plane (feet)
y_c= The y coordinate for phase C on a 2D plane (feet)

D_{ab} = Distance between phase A and phase B (feet)
D_{bc} = Distance between phase B and phase C (feet)
D_{ca} = Distance between phase C and phase A (feet)

D_{eq} = Equivalent geometric mean distance (feet)

- **Transmission Line Object**



Resistance (R)

$$R_{series} = l * \frac{R_c}{n_c}$$

ℓ = length of the transmission line (miles)
n_c = number of sub conductors in each bundle
R_c = resistance of the sub conductors in the bundled phase (Ω/mile)

Series Reactance (X)

$$X_{series} = 2\pi f * 2 \times 10^{-7} * ln(\frac{D_{eq}}{D_{SL}}) * 1609.34$$

f = operating frequency (set to a default of 60) (Hz)
D_{eq} = geometric mean distance of the phases (feet)
D_{SL} = equivalent geometric mean radius of the phases (feet)

Shunt Susceptance (B)

$$B_{shunt} = 2\pi f * \frac{2\pi * 8.854 \times 10^{-12}}{ln(\frac{D_{eq}}{D_{SC}})} * 1609.34$$

f = operating frequency (set to a default of 60) (Hz)
D_{eq} = geometric mean distance of the phases (feet)
D_{SC} = equivalent radius of the phases (feet)

Base Impedance (Z_{base})

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

V_{base} = voltage base (kV)
S_{base} = base power (MVA)

Base Admittance (Y_{base})

$$Y_{base} = \frac{1}{Z_{base}}$$

Z_{base} = base impedance (Ω)

Per Unit Resistance (R_{pu})

$$R_{pu} = \frac{R_{series}}{Z_{base}}$$

R_{series} = series resistance (Ω)
Z_{base} = base impedance (Ω)

Per Unit Reactance(X_{pu})

$$X_{pu} = \frac{X_{series}}{Z_{base}}$$

X_{series} = series reactance (Ω)
Z_{base} = base impedance (Ω)

Per Unit Reactance(B_{pu})

$B_{pu} = \frac{B_{shunt}}{Y_{base}}$

B_{shunt} = shunt susceptance (S)

Y_{base} = base impedance (S)

Series Impedance (Z_{series}) in per unit

$Z_{series} = R_{pu} + jX_{pu}$

R_{series} = series resistance (Ω)

X_{series} = series reactance (Ω)

Series Admittance (Y_{series}) in per unit

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

R_{series} = series resistance (Ω)

X_{series} = series reactance (Ω)

Shunt Admittance (Y_{shunt}) in per unit

$Y_{shunt} = jB_{pu}$

B_{shunt} = shunt susceptance (S)

Positive-, Negative-, and Zero-Sequence Primitive Admittance Matrix (Y_{prim})

$Y_{prim} = \left[Y_{series} + \frac{Y_{shunt}}{2}, \ -Y_{series}\right], \left[-Y_{series}, \ Y_{series} + \frac{Y_{shunt}}{2}\right]$

Y_{series} = series admittance (S)

Y_{series} = shunt admittance (S)

- **Generator Object**

Per-Unit Grounding Impedance ($Z_{n,pu}$)

$Z_{n,pu} = Z_n \frac{S_{base}}{V_{base}^2}$

$Z_{n,pu}$ = per unit grounding impedance

Z_n = grounding impedance, user defined (Ω)

S_{base} = system base power (MVA)

V_{base} = generator terminal voltage (kV)

Positive-Sequence Admittance (Y_1)

$Y_1 = \frac{1}{jX_1}$

Y_1 = per unit positive-sequence admittance

X_1 = per unit positive-sequence subtransient reactance

Negative-Sequence Admittance (Y_2)

$Y_2 = \frac{1}{jX_2}$

Y_2 = per unit negative-sequence admittance

X_2 = per unit negative-sequence subtransient reactance

Zero-Sequence Admittance (Y_0)

$$Y_0 = \frac{1}{jX_0 + 3Z_{n,pu}}$$

Y_0 = per unit zero-sequence admittance

X_0 = per unit zero-sequence subtransient reactance

$Z_{n,pu}$ = per unit grounding impedance

If *is_grounded* = *False*, $Y_0 = 0$

- **Load Class**

Load Admittance (Y)

$$Y = \frac{P - jQ}{V_{rated}^2}$$

Y = Complex admittance of the load (S)

P = Real power of the load (MW)

Q= Reactive power of the load (MVAR)

V_{rated} = Rated line-to-line voltage at the bus (kV)

Admittance Base (Y_{base})

$$Y_{base} = \frac{S_{base}}{V_{rated}^2}$$

Y_{base} = Base admittance (S)

V_{rated} = Rated line-to-line voltage at the bus (kV)

S_{base} = system base power (MVA)

Per Unit Admittance (Y_{pu})

$$Y_{pu} = \frac{Y}{Y_{base}}$$

Y_{pu} = Load admittance in per unit

Y = actual admittance of the load (S)

Y_{base} = Base admittance (S)

- **Jacobian Class**

Jacobian Submatrix (J1)

$$J1_{kn} = \frac{\partial P_k}{\partial \delta_n} = V_k Y_{kn} V_n sin(\delta_k - \delta_n - \theta_{kn}) \; if \; n \neq k$$

$$J1_{kn} = \frac{\partial P_k}{\partial \delta_k} = -V_k \sum_{n=1,\, n \neq k}^{N} Y_{kn} V_n sin(\delta_k - \delta_n - \theta_{kn}) \; if \; n = k$$

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

Jacobian Submatrix (J2)

$$J2_{kn} = \frac{\partial P_k}{\partial V_n} = V_k Y_{kn} cos(\delta_k - \delta_n - \theta_{kn}) \; if \; n \neq k$$

$$J2_{kk} = \frac{\partial P_k}{\partial V_k} = V_k Y_{kk} cos\theta_{kk} + \sum_{n=1}^{N} Y_{kn} V_n cos(\delta_k - \delta_n - \theta_{kn}) \; if \; n = k$$

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

Jacobian Submatrix (J3)

$$J3_{kn} = \frac{\partial Q_k}{\partial \delta_n} = -V_k Y_{kn} V_n cos(\delta_k - \delta_n - \theta_{kn}) \; if \; n \neq k$$

$$J3_{kn} = \frac{\partial Q_k}{\partial \delta_k} = V_k \sum_{n=1,\, n\neq k}^{N} Y_{kn} V_n cos(\delta_k - \delta_n - \theta_{kn}) \; if \; n = k$$

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

Jacobian Submatrix (J4)

$$J4_{kn} = \frac{\partial Q_k}{\partial V_n} = V_k Y_{kn} sin(\delta_k - \delta_n - \theta_{kn}) \; if \; n \neq k$$

$$J4_{kk} = \frac{\partial Q_k}{\partial V_k} = -V_k Y_{kk} sin\theta_{kk} + \sum_{n=1}^{N} Y_{kn} V_n sin(\delta_k - \delta_n - \theta_{kn}) \; if \; n = k$$

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

Full Jacobian Matrix (J)

$$J = [J1, J2], [J3, J4]$$

- **Solution Class**

Power Injection Equations

Real Power (P)

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

P_k = the real power for the k bus in per units

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

Reactive Power (Q)

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

Q_k = the reactive power for the k bus in per units

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

Power Mismatch Equations

Real Power Mismatch (ΔP)

$\Delta P = P_{specified} - P_{calculated}$ where $P_{calculated} = P_k$

$$P_{specified} = \sum_{generators\ at\ bus} \frac{P_G}{S_{base}} - \sum_{loads\ at\ bus} \frac{P_L}{S_{base}}$$

P_G= the MW set point for the generator (MW)

P_L= the real power consumed at the load (MW)

S_{base} = base power (MVA)

Reactive Power Mismatch (ΔQ)

$\Delta Q = Q_{specified} - Q_{calculated}$ where $Q_{calculated} = P_k$

$$Q_{specified} = \sum_{generators\ at\ bus} \frac{Q_G}{S_{base}} - \sum_{loads\ at\ bus} \frac{Q_L}{S_{base}}$$

Q_G= the MVAR set point for the generator (MVAR)

Q_L= the reactive power consumed at the load (MVAR)

S_{base} = base power (MVA)

Newton Raphson Algorithm

Step 1: Calculate the power mismatches

Step 2: Calculate the Jacobian

Step 3: Solve for the updates in δ and V

Solve the Linear System

$[\Delta\delta], [\Delta V] = [\,[J1, J2], [J3, J4]\,]^{-1} * [\Delta P], [\Delta Q]$

$\Delta\delta$ = update in phase angle in radians

ΔV = update in voltage in per unit

ΔP = real power mismatch in per unit

ΔQ = reactive power mismatch in per unit

Step 4: Calculate the updated δ and V

Updated Voltages and Phase Angles

$x(i+1) = [\delta(i+1)], [V(i+1)] = [\delta(i)], [V(i)] + [\Delta\delta], [\Delta V]$

$\delta(i+1)$ = updated phase angle in radians, i+1

$V(i+1)$ = updated voltage in per units, i+1

$\delta(i)$ = phase angle in radians, iteration i

$V(i)$ = voltage in per units, iteration i

$\Delta\delta$ = update in phase angle in radians

ΔV = update in voltage in per unit

Symmetrical (Three-Phase) Fault

Fault Current (I_f)

$$I_f = \frac{V_{prefault}}{Z^1_{nn}}$$

I_f= Subtransient fault current (line-to-neutral)

$V_{prefault}$= pre fault voltage in per units, assumed to be 1

Z^1_{nn}= Positive-sequence self-impedance at the faulted bus

Post-Fault Voltage at Bus k (V_k)

$V_k = V_{prefault}(1 - \frac{Z^1_{kn}}{Z^1_{nn}})$

$V_{prefault}$ = pre fault voltage in per units, assumed to be 1

Z^1_{nn} = Positive-sequence self-impedance at the faulted bus

Z^1_{kn} = Positive-sequence mutual impedance between bus k and bus n

Asymmetrical Faults

Assumptions

$V_{prefault}$ = 1.0 p.u.

Z_f = 0 for a bolted fault

All voltages and currents are in per unit

Bus labels:

n = faulted bus

k = any bus in the system

Line-to-Ground (LG) Fault

Total Fault Impedance (Z_{total})

$Z_{total} = Z^1_{nn} + Z^2_{nn} + Z^0_{nn} + 3Z_f$

Z_{total} = total impedance for line-to-ground faults

Z^1_{nn}, Z^2_{nn}, Z^0_{nn} = sequence self-impedance at faulted bus

Z_f = fault impedance

Sequence Currents

$I_1 = I_2 = I_0 = \frac{V_{prefault}}{Z_{total}}$, $I_f = I_1 + I_2 + I_0$

I_1, I_2, I_0 = sequence currents

I_f = subtransient fault current (line-to-neutral)

$V_{prefault}$ = pre fault voltage in per units

Z_{total} = total impedance for line-to-ground faults

Sequence Voltages at Bus k

$V^1_k = V_{prefault} - Z^1_{kn} * I_1$

$V^2_k = - Z^2_{kn} * I_2$

$V^0_k = - Z^0_{kn} * I_0$

V^1_k, V^2_k, V^0_k = sequence voltages at bus k

I_1, I_2, I_0 = sequence currents

$V_{prefault}$ = pre fault voltage in per units

Z^1_{kn}, Z^2_{kn}, Z^0_{kn} = sequence mutual impedance between bus k and faulted bus

Phase-to-Neutral Voltage (V_k)

$V_k = V^1_k + V^2_k + V^0_k$

V_k = total post-fault voltage at bus k

V_k^1, V_k^2, V_k^0 = sequence voltages at bus k

Line-to-Line (LL) Fault

Total Fault Impedance (Z_{total})

$$Z_{total} = Z_{nn}^1 + Z_{nn}^2 + Z_f$$

Z_{total} = total impedance for line-to-ground faults

Z_{nn}^1, Z_{nn}^2 = sequence self-impedance at faulted bus

Z_f = fault impedance

Sequence Currents

$$I_1 = -I_2 = \frac{V_{prefault}}{Z_{total}},\ I_0 = 0,\ I_f = \sqrt{3} * I_1$$

I_1, I_2, I_0 = sequence currents

I_f = subtransient fault current (line-to-neutral)

$V_{prefault}$ = pre fault voltage in per units

Z_{total} = total impedance for line-to-ground faults

Sequence Voltages at Bus k

$$V_k^1 = V_{prefault} - Z_{kn}^1 * I_1$$

$$V_k^2 = -Z_{kn}^2 * I_2$$

$$V_k^0 = -Z_{kn}^0 * I_0$$

V_k^1, V_k^2, V_k^0 = sequence voltages at bus k

I_1, I_2, I_0 = sequence currents

$V_{prefault}$ = pre fault voltage in per units

Z_{kn}^1, Z_{kn}^2, Z_{kn}^0 = sequence mutual impedance between bus k and faulted bus

Phase-to-Neutral Voltage (V_k)

$$V_k = V_k^1 + V_k^2 + V_k^0$$

V_k = total post-fault voltage at bus k

V_k^1, V_k^2, V_k^0 = sequence voltages at bus k

Double Line-to-Ground (LL) Fault

Total Fault Impedances per Sequence

$$Z_{total}^1 = Z_{nn}^1 - \left[\frac{Z_{nn}^2(Z_{nn}^0+3Z_f)}{Z_{nn}^2+Z_{nn}^0+3Z_f}\right],\ Z_{total}^2 = \frac{Z_{nn}^0+3Z_f}{Z_{nn}^2+Z_{nn}^0+3Z_f},\ Z_{total}^0 = \frac{Z_{nn}^2}{Z_{nn}^2+Z_{nn}^0+3Z_f}$$

Z_{total}^1, Z_{total}^2, Z_{total}^0 = total sequence impedance at faulted bus

Z_{nn}^1, Z_{nn}^2, Z_{nn}^0 = sequence self-impedance at faulted bus

Z_f = fault impedance

Sequence Currents

$$I_1 = \frac{V_{prefault}}{Z_{total}^1},\ I_2 = -I_1 * Z_{total}^2,\ I_0 = -I_1 * Z_{total}^0,\ I_f = \sqrt{3} * I_1$$

I_1, I_2, I_0 = sequence currents
I_f = subtransient fault current (line-to-neutral)
$V_{prefault}$ = pre fault voltage in per units
Z^1_{total}, Z^2_{total}, Z^0_{total} = total sequence impedance at faulted bus

Sequence Voltages at Bus k

$$V^1_k = V_{prefault} - Z^1_{kn} * I_1$$
$$V^2_k = - Z^2_{kn} * I_2$$
$$V^0_k = - Z^0_{kn} * I_0$$

V^1_k, V^2_k, V^0_k = sequence voltages at bus k
I_1, I_2, I_0 = sequence currents
$V_{prefault}$ = pre fault voltage in per units
Z^1_{kn}, Z^2_{kn}, Z^0_{kn} = sequence mutual impedance between bus k and faulted bus

Phase-to-Neutral Voltage (V_k)

$$V_k = V^1_k + V^2_k + V^0_k$$

V_k = total post-fault voltage at bus k
V^1_k, V^2_k, V^0_k = sequence voltages at bus k

4. Example cases

Validation of Resistance, Reactance, and Susceptance

The TransmissionLine Object models and analyzes the electrical characteristics of a power transmission line. This Object computes key per-unit parameters: resistance per unit (R_{pu}), reactance per unit (X_{pu}), and susceptance per unit (B_{pu}). These parameters are crucial for power system analysis and help with the evaluation of line performance, losses, and stability.

A transmission line's impedance and admittance values influence voltage regulation, power flow, and overall efficiency in the power grid. By calculating these values per unit, the Object normalizes them to a standard base for easy comparison and integration into power system studies. The TransmissionLine Object is designed with the following attributes:

name: Name of the transmission line.

bus1: The sending-end bus (Bus object).

bus2: The receiving-end bus (Bus object).

bundle: The bundle configuration of conductors (Bundle object).

geometry: The spatial configuration of the transmission line (Geometry object).

length: The total length of the transmission line in miles.

zbase: The base impedance for per-unit calculations.

ybase: The base admittance for per-unit calculations.

Rpu: The per-unit resistance of the transmission line.

Xpu: The per-unit reactance of the transmission line.

Bpu: The per-unit susceptance of the transmission line.

yprim: The primitive admittance matrix for the transmission line.

Solution Process

The Object computes the following per-unit values:

- **calc_base_values():**
  - Computes the base impedance ($Z_{base} = \frac{V_{base}^2}{S_{base}}$) and base admittance ($Y_{base} = \frac{1}{Z_{base}}$).
  - Uses the rated voltage of bus2 and a global system base Sbase.
- **calc_Rpu():**
  - Computes the per-unit resistance based on conductor resistance and bundle configuration.
  - Formula:
    - $R_{series} = l * \frac{R_c}{n_c}$
    - $R_{pu} = \frac{R_{series}}{Z_{base}}$
- **calc_Xpu():**
  - Computes the per-unit reactance using conductor spacing and system frequency.
  - Formula:
    - $X_{series} = 2\pi f * 2 \times 10^{-7} * ln(\frac{D_{eq}}{D_{SL}}) * 1609.34$
    - $X_{pu} = \frac{X_{series}}{Z_{base}}$
- **calc_Bpu():**
  - Computes the per-unit shunt susceptance using conductor spacing and system frequency.
  - Formula:
    - $B_{shunt} = 2\pi f * \frac{2\pi * 8.854 \times 10^{-12}}{ln(\frac{D_{eq}}{D_{SC}})} * 1609.34$
    - $B_{pu} = \frac{B_{shunt}}{Y_{base}}$
- **calc_yprim():**
  - Constructs the primitive admittance matrix using calculated per-unit values.
  - The matrix accounts for both series impedance and shunt admittance.

Problem Definition

The example transmission line model consists of the following components:

- Bus 1 (Sending End): 230 kV
- Bus 2 (Receiving End): 230 kV
- Transmission Line: "Line 1"
    - Length: 10 miles
- Conductor: "Partridge"
    - Diameter: 0.642 inches
    - Geometric Mean Radius (GMR): 0.0217 feet
    - Resistance: 0.385 Ω/mile
    - Ampacity: 460 A
- Bundle Configuration: "Bundle 1"
    - Number of conductors: 2
    - Spacing: 1.5 feet
- Geometry Configuration: "Geometry 1"
    - Conductor Y placement: Ya = 0, Yb = 0, Yc = 0
    - Conduct X placement: Xa = 0, Xb = 18.5, Xc = 37

Use the main code below for this:

```
import math

import numpy as np

import pandas as pd

from Conductor import Conductor

from Bundle import Bundle

from Geometry import Geometry

from Bus import Bus

from Global import Global

# Define conductor

conductor1 = Conductor("Partridge", 0.642, 0.0217, 0.385, 460)

# Define bundle configuration

bundle1 = Bundle("Bundle 1", 2, 1.5, conductor1)

# Define geometry

geometry1 = Geometry("Geometry 1", 0, 0, 18.5, 0, 37, 0)
```

```
# Define buses

bus1 = Bus("Bus 1", 230)

bus2 = Bus("Bus 2", 230)

# Define transmission line

line1 = TransmissionLine("Line 1", bus1, bus2, bundle1, geometry1, 10)

# Output calculations

print(f"Transmission Line: {line1.name}")

print(f"From: {line1.bus1.name} To: {line1.bus2.name}, Length: {line1.length}
miles")

print(f"Base Impedance: {line1.zbase}, Base Admittance: {line1.ybase}")

print(f"Per-Unit Values -> R: {line1.Rpu}, X: {line1.Xpu}, B: {line1.Bpu}")

print("Primitive Admittance Matrix:")

print(line1.yprim)
```

Expected Output

Transmission Line: Line 1

From: Bus 1 To: Bus 2, Length: 10 miles

Base Impedance: 529.0, Base Admittance: 0.001890359168241966

Per-Unit Values -> R: 0.0036389413988657847, X: 0.011150810943831139, B: 0.03753603852127484

Primitive Admittance Matrix:

	Bus 1	Bus 2
Bus 1	26.449163-81.029422j	-26.449163+81.048190j
Bus 2	-26.449163+81.048190j	26.449163-81.029422j

Validation of Transformer Object

The Transformer Object models and analyzes the electrical characteristics of a power transformer. This Object computes key per-unit parameters: resistance per unit (R_{pu}), reactance per unit (X_{pu}), and

series admittance (Y_{series}). These parameters are essential for power system analysis, helping to evaluate transformer performance, losses, and stability.

A transformer's impedance and admittance values influence voltage regulation, power flow, and overall efficiency in the power grid. By calculating these values per unit, the Object normalizes them to a standard base for easy comparison and integration into power system studies. The Transformer Object is designed with the following attributes:

- **name:** Name of the transformer.
- **bus1:** The primary-side bus (Bus object).
- **bus2:** The secondary-side bus (Bus object).
- **power_rating:** The transformer's rated power in MVA.
- **impedance_percent:** The impedance of the transformer as a percentage of base impedance.
- **x_over_r_ratio:** The ratio of reactance to resistance.
- **Rpusys:** The per-unit resistance of the transformer.
- **Xpusys:** The per-unit reactance of the transformer.
- **Yseries:** The series admittance of the transformer.
- **yprim:** The primitive admittance matrix for the transformer.

Solution Process

The Object computes the following per-unit values:

- **calc_base_values()**
  - Computes the base impedance and base admittance.
  - Formula: $Z_{base} = \frac{V_{base}^2}{S_{base}}, Y_{base} = \frac{1}{Z_{base}}$
- **calc_Rpu()**
  - Computes the per-unit resistance based on impedance percentage and X/R ratio.
  - Formula:
    - $Z_{pu} = (\frac{Z_{percent}}{100})Z_{base}$
    - $R_{pu} = \frac{Z_{pu}}{\sqrt{1+(X/R)^2}}$
- **calc_Xpu()**
  - Computes the per-unit reactance using impedance percentage and X/R ratio.
  - Formula:
    - $X_{pu} = R_{pu} * \frac{X}{R}$
- **calc_Yseries()**
  - Computes the series admittance using the calculated per-unit impedance.
  - Formula:
    - $Y_{series} = \frac{1}{R_{pu}+jX_{pu}}$
- **calc_yprim()**

- Constructs the primitive admittance matrix using the calculated per-unit values. The matrix accounts for both series impedance and shunt admittance.
- Formula:
  - $Y_{prim} = \left[Y_{series}, \ -Y_{series}\right], \left[-Y_{series}, Y_{series}\right]$

Problem Definition

A transformer model consists of the following components:

- Bus 1 (Primary Side): 20 kV
- Bus 2 (Secondary Side): 230 kV
- Transformer: "T1"
- Power Rating: 100 MVA
- Impedance: 8.5% of base impedance
- X/R Ratio: 10

Use the main code below for this:

```
import numpy as np

import pandas as pd

from Bus import Bus

from Global import Global



# Define Buses

bus1 = Bus("Bus1", 20)

bus2 = Bus("Bus2",230)



# Define Transformer

transformer1 = Transformer("T1", bus1, bus2, 100, 8.5, 10)



# Output Calculations

print("Transformer Name:", transformer1.name)

print("Connected Buses:", transformer1.bus1, "<-->", transformer1.bus2)

print("Power Rating:", transformer1.power_rating, "MVA")
```

```
print("Per-unit Resistance (Rpu):", transformer1.Rpusys)

print("Per-unit Reactance (Xpu):", transformer1.Xpusys)

print("Series Admittance (Yseries):", transformer1.Yseries)

print("Yprim matrix:")

print(transformer1.yprim)
```

Expected Output

Transformer Name: T1

Connected Buses: Bus1 <--> Bus2

Power Rating: 100 MVA

Per-unit Resistance (Rpu): 0.00849402834982856

Per-unit Reactance (Xpu): 0.0849402834982856

Series Admittance (Yseries): (0.09558015644011158-0.9558015644011158j)

Yprim matrix:

	Bus1	Bus2
Bus1	0.09558-0.9558j	-0.09558+0.9558j
Bus2	-0.09558+0.9558j	0.09558-0.9558j

Validation of the Admittance Matrix Formation

The Circuit Object computes the bus admittance matrix (Ybus) representing the steady-state admittance relationships between all buses in the power system. This matrix is fundamental for power flow and fault analysis and encapsulates the electrical connectivity and admittance characteristics of all network components including transmission lines, transformers, and shunt elements.

The admittance matrix quantifies how voltages at different buses influence current injections, making it critical for system analysis, planning, and operational decision-making. The Ybus is formed in per-unit and normalized using the system base. The Circuit Object uses the following attributes and structure:

- **buses:** Dictionary of all buses in the system (Bus objects).
- **lines:** Dictionary of all transmission lines (TransmissionLine objects).
- **transformers:** Dictionary of all transformers (Transformer objects), if present.

- **loads:** Dictionary of loads connected to the buses (optional).
- **generators:** Dictionary of generators in the system (optional).
- **ybus:** Pandas DataFrame containing the complex Ybus matrix in per-unit.

Solution Process

The object forms the Ybus matrix using the following method: calc_ybus(). It constructs the complete Ybus matrix using the primitive admittance matrices of each line and transformer in the system.

Steps include:

1. Initialize an empty matrix of size n x n where n = number of buses.
2. For each transmission line:
    a. Retrieve its yprim matrix.
    b. Add the primitive admittance matrix values to the rows and column corresponding to their buses
3. For each transformer (if modeled), follow a similar process using its yprim.
4. Final matrix is stored as a Pandas DataFrame indexed by bus names.

Problem Definition

The example transmission line model consists of the following components:

- Bus 1: 230 kV
- Bus 2: 230 kV
- Bus 3: 18 kV
- Transmission Line: "Line 1"
    - Length: 10 miles
    - Bus 1 to Bus 2
- Transformer: “T1”
    - Bus 2 to Bus 3
    - Rating: 200 MVA
    - Impedance: 10.5%
    - X/R: 12
    - Connection: Delta-Y grounded
    - Grounding impedance: 1 Ω
- Conductor: "Partridge"
    - Diameter: 0.642 inches
    - Geometric Mean Radius (GMR): 0.0217 feet
    - Resistance: 0.385 Ω/mile
    - Ampacity: 460 A
- Bundle Configuration: "Bundle 1"
    - Number of conductors: 2
    - Spacing: 1.5 feet
- Geometry Configuration: "Geometry 1"

    - Conductor Y placement: Ya = 0, Yb = 0, Yc = 0
    - Conduct X placement: Xa = 0, Xb = 18.5, Xc = 37
- Global Base Value:
    - S_{base} = 100 MVA

Use the main code below for this:

```
from Circuit import Circuit

circuit1 = Circuit("Test Circuit")

# ADD BUSES

circuit1.add_bus("Bus1", 230)

circuit1.add_bus("Bus2", 230)

circuit1.add_bus("Bus3", 18)



# ADD TRANSMISSION LINE

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", "Bus1", "Bus2", "Bundle1", "Geometry1", 10)



# ADD TRANSMORMER

circuit1.add_transformer("T1", "Bus2", "Bus3", 200, 10.5, 12, "delta-y", 1)



# Calculate Ybus

ybus = circuit1.calc_ybus()

print("Ybus Matrix:")

print(ybus)
```

Expected Output

Ybus Matrix:

	Bus1	Bus2	Bus3
Bus1	26.449163-81.029422j	-26.449163+ 81.048190j	0.000000+ 0.000000j
Bus2	-26.449163+81.048190j	28.030982-100.011246j	-1.581819+18.981824j
Bus3	0.000000+ 0.000000j	-1.581819+ 18.981824j	1.581819-18.981824j

Validation of the Power Flow and Newton Raphson Algorithm

The Solution Object implements the Newton-Raphson method to solve the power flow problem for a given circuit. It iteratively solves for bus voltage magnitudes and angles under steady-state conditions, using power mismatches and the Jacobian matrix to update the state vector. The method is widely used due to its fast convergence and robustness for solving nonlinear power system equations.

By calculating active (P) and reactive (Q) power mismatches and iteratively correcting them, the algorithm ensures that all generator, load, and network constraints are satisfied. The resulting voltages and angles are critical for downstream fault analysis and grid stability studies.The Solution Object is designed with the following attributes:

- **circuit:** The Circuit object containing buses, lines, transformers, loads, and generators.
- **ybus:** The bus admittance matrix used for power flow calculations.
- **voltages:** Dictionary storing voltage magnitudes (in per-unit) at each bus.
- **angles:** Dictionary storing voltage angles (in radians) at each bus.

Solution Process

The object solves the power flow problem using the following method: power_flow(). This is the wrapper method that calls the Newton-Raphson algorithm with configurable tolerance and iteration limits: newton_raphson(tolerance=0.001, max_iterations=50)

This method solves for voltage magnitudes and angles using the following steps:

1. Initialization: Start with flat voltage profile ($1.0 \angle 0$) and use circuit-specified generator/load data.
2. Mismatch Calculation: Compute ΔP and ΔQ using actual minus specified power.
    a. Real Power (P)

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

    P_k = the real power for the k bus in per units

    $V_{k'}$ V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

b. Reactive Power (Q)

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

Q_k = the reactive power for the k bus in per units

V_k, V_n = the voltages at bus k and bus n in per unit

Y_{kn} = the admittance between buses k and n (S)

δ_k, δ_n = the phase angles at buses k and n in radians

θ_{kn} = the angle of the admittance element Y_{kn} in radians

c. Real Power Mismatch (ΔP)

$\Delta P = P_{specified} - P_{calculated}$ where $P_{calculated} = P_k$

$$P_{specified} = \sum_{generators\ at\ bus} \frac{P_G}{S_{base}} - \sum_{loads\ at\ bus} \frac{P_L}{S_{base}}$$

P_G = the MW set point for the generator (MW)

P_L = the real power consumed at the load (MW)

S_{base} = base power (MVA)

d. Reactive Power Mismatch (ΔQ)

$\Delta Q = Q_{specified} - Q_{calculated}$ where $Q_{calculated} = P_k$

$$Q_{specified} = \sum_{generators\ at\ bus} \frac{Q_G}{S_{base}} - \sum_{loads\ at\ bus} \frac{Q_L}{S_{base}}$$

Q_G = the MVAR set point for the generator (MVAR)

Q_L = the reactive power consumed at the load (MVAR)

S_{base} = base power (MVA)

3. Jacobian Formation: Construct Jacobian matrix using partial derivatives of power equations.
4. State Update: Solve J·Δx = mismatch and update voltages and angles.
   a. Solve the Linear System

   $[\Delta\delta], [\Delta V] = [\,[J1, J2], [J3, J4]\,]^{-1} * [\Delta P], [\Delta Q]$

   $\Delta\delta$ = update in phase angle in radians

   ΔV = update in voltage in per unit

   ΔP = real power mismatch in per unit

   ΔQ = reactive power mismatch in per unit

   b. Updated Voltages and Phase Angles

   $x(i+1) = [\delta(i+1)], [V(i+1)] = [\delta(i)], [V(i)] + [\Delta\delta], [\Delta V]$

   $\delta(i+1)$ = updated phase angle in radians, i+1

   $V(i+1)$ = updated voltage in per units, i+1

   $\delta(i)$ = phase angle in radians, iteration i

   $V(i)$ = voltage in per units, iteration i

   $\Delta\delta$ = update in phase angle in radians

   ΔV = update in voltage in per unit

5. Convergence Check: Stop if max mismatch < tolerance, otherwise repeat.

Problem Definition

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



# create test circuit

circuit1 = Circuit("Test Circuit")



# ADD BUSES

circuit1.add_bus("Bus1", 20)

circuit1.add_bus("Bus2", 230)

circuit1.add_bus("Bus3", 230)

circuit1.add_bus("Bus4", 230)

circuit1.add_bus("Bus5", 230)
```

```
circuit1.add_bus("Bus6", 230)

circuit1.add_bus("Bus7", 18)



# 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)



print(f"Bus1 type: {circuit1.buses['Bus1'].bus_type}")  # Should print "Slack
Bus"

print(f"Bus2 type: {circuit1.buses['Bus2'].bus_type}")  # Should print "PQ Bus"

print(f"Bus7 type: {circuit1.buses['Bus7'].bus_type}")  # Should print "PV Bus"




solution = Solution(circuit1)



solution.power_flow()
```

Expected Output

Bus1 type: Slack Bus
Bus2 type: PQ Bus
Bus7 type: PV Bus

Iteration 1:

Max mismatch = 2.000000

Iteration 2:

Max mismatch = 0.127868

Iteration 3:

Max mismatch = 0.003035

Iteration 4:

Max mismatch = 0.000001

Converged!

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

Converged in 4 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	112.434933	-28.130360	-70.739566	0.000000	-0.000000	-0.000000	36.071037	-9.378511	-24.446748	0.000000	0.000000
∂P Bus3	-27.809324	62.275568	-0.000000	-34.466244	-0.000000	-0.000000	-10.265178	20.724429	0.000000	-12.582030	0.000000
∂P Bus4	-70.534048	0.000000	125.783355	-19.979365	-35.269942	-0.000000	-24.934757	0.000000	42.851543	-6.979612	-12.786448
∂P Bus5	-0.000000	-34.711589	-19.949930	124.916362	-70.254842	-0.000000	0.000000	-11.850297	-7.053764	42.704323	-25.658167
∂P Bus6	0.000000	0.000000	-35.569285	-70.955670	124.106030	-17.581075	0.000000	0.000000	-11.935036	-23.697226	38.774319
∂P Bus7	0.000000	0.000000	0.000000	0.000000	-17.897149	17.897149	0.000000	0.000000	0.000000	0.000000	0.444941
∂Q Bus2	-33.802240	8.635767	22.736642	-0.000000	-0.000000	-0.000000	119.981543	-30.549793	-76.060148	0.000000	-0.000000
∂Q Bus3	9.619518	-21.283127	-0.000000	11.663609	-0.000000	-0.000000	-29.675880	66.545750	-0.000000	-37.180198	-0.000000
∂Q Bus4	23.366409	-0.000000	-41.853977	6.470138	12.017430	-0.000000	-75.268278	0.000000	133.738683	-21.552588	-37.526933
∂Q Bus5	-0.000000	10.911797	6.560337	-41.587136	24.115002	-0.000000	-0.000000	-37.697060	-21.450437	133.350210	-74.750584
∂Q Bus6	-0.000000	-0.000000	11.100153	21.967456	-36.442309	3.374700	0.000000	0.000000	-38.244581	-76.542889	132.047814

Validation of Symmetrical Fault Analysis

The Solution object includes a method to perform symmetrical three-phase fault analysis at any specified bus. This type of fault assumes balanced conditions and uses only the positive-sequence impedance

matrix to calculate fault currents and post-fault voltages. Symmetrical faults are essential for establishing the maximum short-circuit currents that protective equipment must withstand.

The Solution object is designed with the following attributes:

- circuit: The Circuit object containing all system components (buses, lines, transformers, loads, generators).
- zbus_pos: The positive-sequence impedance matrix derived from the system.
- voltages: Dictionary storing voltage magnitudes (in per-unit) at each bus.
- angles: Dictionary storing voltage angles (in radians) at each bus.

Solution Process

The fault analysis is executed via perform_symmetrical_fault(bus_name, v_prefault) which performs the following steps:

1. Initialization:Assume prefault voltage $V_{prefault} = 1.0 \angle 1°$ per unit. Extract the positive-sequence Zbus matrix.

2. Fault Current Calculation:

$$I_f = \frac{V_{prefault}}{Z_{nn}^{(1)}}$$

   - I_f: Subtransient fault current, in per unit.
   - $Z_{nn}^{(1)}$: Self-impedance of the faulted bus in the positive-sequence Zbus matrix.

3. Bus Voltage Calculation (Post-fault):

$$V_k = V_{prefault}(1 - \frac{Z_{kn}}{Z_{nn}})$$

   - V_k:Post-fault voltage at bus k.
   - Z_{kn}: Mutual impedance between bus k and faulted bus n.
   - Z_{nn}: Self-impedance at faulted bus.
   - $V_{prefault}$: Assumed $1.0 \angle 1°$ per unit.

4. Output Formatting:
   - The fault current magnitude and angle (in degrees) are printed.
   - Post-fault voltages are printed for all buses in magnitude and angle.

Problem Definition

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



   # create test circuit

   circuit1 = Circuit("Test Circuit")



   # ADD BUSES

   circuit1.add_bus("Bus1", 20)

   circuit1.add_bus("Bus2", 230)

   circuit1.add_bus("Bus3", 230)

   circuit1.add_bus("Bus4", 230)

   circuit1.add_bus("Bus5", 230)

   circuit1.add_bus("Bus6", 230)

   circuit1.add_bus("Bus7", 18)



   # 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)



    print(f"Bus1 type: {circuit1.buses['Bus1'].bus_type}")  # Should print
"Slack Bus"

    print(f"Bus2 type: {circuit1.buses['Bus2'].bus_type}")  # Should print "PQ
Bus"

    print(f"Bus7 type: {circuit1.buses['Bus7'].bus_type}")  # Should print "PV
Bus"



    solution = Solution(circuit1)
```

```
solution.fault_study()
```

Expected Output

Bus1 type: Slack Bus
Bus2 type: PQ Bus
Bus7 type: PV Bus

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: Bus1

>>> Performing symmetrical 3-phase fault analysis

Subtransient fault current at Bus1: 13.0026 p.u. ∠-83.07°
Post-fault Phase A voltage at Bus1: 0.0000 p.u. ∠0.00°
Post-fault Phase A voltage at Bus2: 0.3288 p.u. ∠13.22°
Post-fault Phase A voltage at Bus3: 0.3747 p.u. ∠12.36°
Post-fault Phase A voltage at Bus4: 0.3665 p.u. ∠11.63°
Post-fault Phase A voltage at Bus5: 0.3967 p.u. ∠10.44°
Post-fault Phase A voltage at Bus6: 0.4093 p.u. ∠8.74°
Post-fault Phase A voltage at Bus7: 0.5854 p.u. ∠3.21°

Validation of Asymmetrical Fault Analysis

The Solution object implements asymmetrical fault analysis for unbalanced fault conditions, including line-to-ground (LG), line-to-line (LL), and double line-to-ground (LLG) faults. Unlike symmetrical faults, these fault types involve unequal impacts on the three phases and require the use of all three symmetrical component networks—positive, negative, and zero sequence. The analysis determines the subtransient fault current and post-fault bus voltages for each case using the impedance matrices derived from the Zbus model.

The Solution object is designed with the following attributes:

- circuit: The Circuit object containing buses, lines, transformers, loads, and generators.
- zbus_pos: Positive-sequence impedance matrix.
- zbus_neg: Negative-sequence impedance matrix.
- zbus_zero: Zero-sequence impedance matrix.
- voltages: Dictionary storing voltage magnitudes (in per-unit) at each bus.
- angles: Dictionary storing voltage angles (in radians) at each bus.

Solution Process

1. Initialization:
    - Assume prefault voltage $V_{prefault} = 1.0$ p.u. and phase angle radians.
    - Retrieve the corresponding Zbus matrices: positive, negative, and zero.
    - Set fault impedance $Z_f = 0$ (bolted fault).
2. Fault Current Calculation:
    - Line-to-Ground (LG) Fault:

$$I_1 = I_2 = I_0 = \frac{V_{prefault}}{Z_1+Z_2+Z_0+3Z_f} \; ; \; I_f = I_1 + I_2 + I_0$$

    - Line-to Line (LL) Fault:

$$I_1 = \frac{V_{prefault}}{Z_1+Z_2+Z_f} \; ; \; I_2 = -I_1 \; ; \; I_0 = 0 \; ; \; I_f = \sqrt{3}I_1$$

    - Double Line-to-Ground Fault:

$$I_1 = \frac{V_{prefault}}{Z_1+(\frac{Z_2(Z_0+3Z_f)}{Z_2+Z_0+3Z_f})} \; ; \; I_2 = -I_1(\frac{(Z_0+3Z_f)}{Z_2+Z_0+3Z_f}) \; ; \; I_0 = -I_1(\frac{Z_2}{Z_2+Z_0+3Z_f}) \; ; \; I_f = \sqrt{3}I_1$$

3. Post-Fault Voltage Calculation:

$$V_k = V_1^{(k)} + V_2^{(k)} + V_0^{(k)}$$

    - $V_1^{(k)} = V_{prefault} - I_1 Z_{1kn}$
    - $V_2^{(k)} = -I_2 Z_{2kn}$
    - $V_3^{(k)} = -I_3 Z_{3kn}$
4. Output Formatting:
    - The fault current magnitude and angle (in degrees) are printed.
    - Post-fault voltages are printed for all buses, with magnitude and phase angle.

Problem Definition

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



   # create test circuit

   circuit1 = Circuit("Test Circuit")



   # ADD BUSES

   circuit1.add_bus("Bus1", 20)

   circuit1.add_bus("Bus2", 230)

   circuit1.add_bus("Bus3", 230)

   circuit1.add_bus("Bus4", 230)

   circuit1.add_bus("Bus5", 230)

   circuit1.add_bus("Bus6", 230)

   circuit1.add_bus("Bus7", 18)



   # 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)



    print(f"Bus1 type: {circuit1.buses['Bus1'].bus_type}")  # Should print
"Slack Bus"

    print(f"Bus2 type: {circuit1.buses['Bus2'].bus_type}")  # Should print "PQ
Bus"

    print(f"Bus7 type: {circuit1.buses['Bus7'].bus_type}")  # Should print "PV
Bus"
```

```
solution = Solution(circuit1)


solution.fault_study()

solution.fault_study()

solution.fault_study()
```

Expected Output

Bus1 type: Slack Bus
Bus2 type: PQ Bus
Bus7 type: PV Bus

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): 2
Available buses: ['Bus1', 'Bus2', 'Bus3', 'Bus4', 'Bus5', 'Bus6', 'Bus7']
Enter the bus name where the fault occurs: Bus1

>>> Performing line-to-ground (LG) fault analysis

Phase A fault current: 14.1031 p.u. ∠-84.28°
Post-fault Phase A voltage at Bus1: 0.0000 p.u. ∠0.00°
Post-fault Phase A voltage at Bus2: 0.4810 p.u. ∠9.02°
Post-fault Phase A voltage at Bus3: 0.5157 p.u. ∠8.76°
Post-fault Phase A voltage at Bus4: 0.5095 p.u. ∠8.33°
Post-fault Phase A voltage at Bus5: 0.5322 p.u. ∠7.76°
Post-fault Phase A voltage at Bus6: 0.5414 p.u. ∠6.82°
Post-fault Phase A voltage at Bus7: 0.6703 p.u. ∠3.25°

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): 3
Available buses: ['Bus1', 'Bus2', 'Bus3', 'Bus4', 'Bus5', 'Bus6', 'Bus7']
Enter the bus name where the fault occurs: Bus1

>>> Performing line-to-line fault analysis

Line current between A and B (Iab): 10.6231 p.u. ∠-82.53°
Post-fault Phase A voltage at Bus1: 1.0567 p.u. ∠-0.48°
Post-fault Phase A voltage at Bus2: 1.0456 p.u. ∠-0.68°
Post-fault Phase A voltage at Bus3: 1.0438 p.u. ∠-0.71°
Post-fault Phase A voltage at Bus4: 1.0443 p.u. ∠-0.69°
Post-fault Phase A voltage at Bus5: 1.0434 p.u. ∠-0.69°
Post-fault Phase A voltage at Bus6: 1.0434 p.u. ∠-0.66°
Post-fault Phase A voltage at Bus7: 1.0406 p.u. ∠-0.50°

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): 4
Available buses: ['Bus1', 'Bus2', 'Bus3', 'Bus4', 'Bus5', 'Bus6', 'Bus7']
Enter the bus name where the fault occurs: Bus1

>>> Performing double line-to-ground (LLG) fault analysis

Line current between A and B (Iab): 15.9547 p.u ∠-84.24°
Post-fault Phase A voltage at Bus1: 0.8762 p.u ∠2.84°
Post-fault Phase A voltage at Bus2: 0.7250 p.u ∠3.96°
Post-fault Phase A voltage at Bus3: 0.7448 p.u ∠3.89°
Post-fault Phase A voltage at Bus4: 0.7415 p.u ∠3.73°
Post-fault Phase A voltage at Bus5: 0.7547 p.u ∠3.51°
Post-fault Phase A voltage at Bus6: 0.7606 p.u ∠3.15°
Post-fault Phase A voltage at Bus7: 0.8379 p.u ∠1.66°