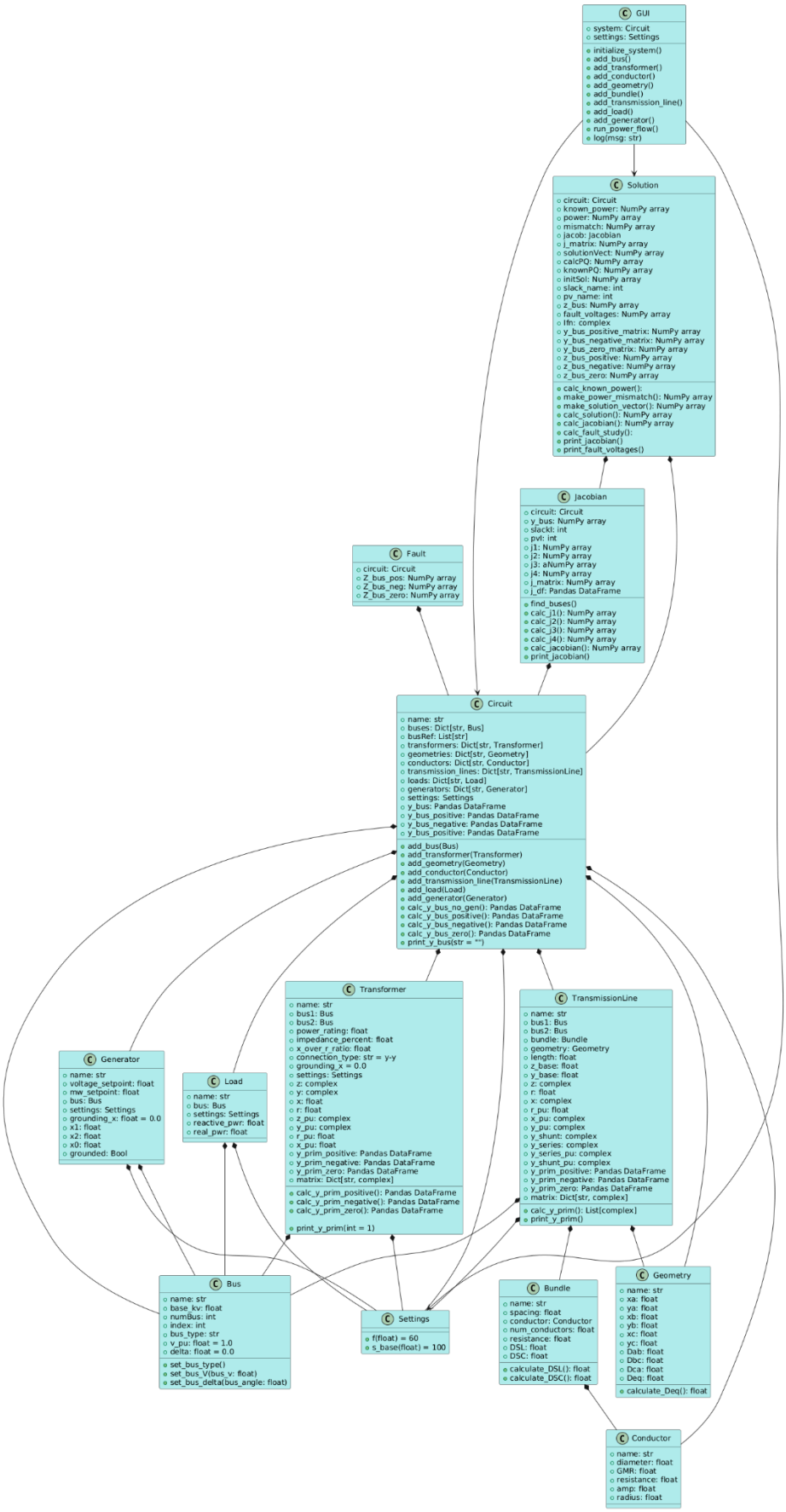
Circuit Simulator

UML



Settings

The Settings class sets global variables for the power base and frequency in a power system.

Attributes

- f (float): The frequency of the power system.
- s_base (float): The power base of the system.

Methods

- __init__(self, f = 60, s_base = 100): initializes a Settings object with the given parameters; if no frequency is provided, defaults to 60 Hz; if no power base is provided, defaults to 100 MVA.

Properties

- f(self) → float: sets frequency of the power system.
- s_base(self) → float: sets the power base of the power system.

Bus

The Bus class represents a bus in a power system, with attributes for its name, base voltage, bus type, voltage magnitude, voltage angle, and an index that uniquely identifies each bus instance.

Attributes

- numBus (int): A class-level variable that keeps track of the total number of Bus instances created. It is incremented each time a new Bus object is instantiated.
- name (str): The name of the bus.
- base_kv (float): The base voltage of the bus in kilovolts.
- index (int): A unique index assigned to each bus instance, based on the order of creation.
- bus_type(string): classifies a bus as PQ, PV, or slack
- v_pu (float): bus voltage magnitude
- delta (float): bus voltage angle

Methods

- __init__(self, name: str, base_kv: float): Initializes a Bus object with the provided parameters, increments the numBus class variable, and assigns the current value of numBus to the index attribute, ensuring each bus has a unique index.

- set_bus_type(self) → None: Assigns the bus type provided in the constructor. A bus can be classified as PQ, PV, or a slack bus and the program will throw an error if a different name is provided for the bus type.

- set_bus_V(self, bus_v: float) → None: Assigns the voltage provided as an argument.

- set_bus_delta(self, bus_angle: float) → None: Assigns the angle provided as an argument.

Conductor

The Conductor class represents an electrical conductor with attributes for its physical and electrical properties.

Attributes

- name (str): The name of the conductor.
- diameter (float): The outside diameter of the conductor in inches.
- GMR (float): The geometric mean radius of the conductor at 60 Hz.
- resistance (float): The electrical resistance of the conductor.
- amp (float): The current-carrying capacity (amperage) of the conductor.
- radius (float): The radius of the conductor in inches, calculated by converting the diameter to radius.

Methods

- __init__(self, name: str, diameter: float, GMR: float, resistance: float, amp: float): Initializes a Conductor object with the provided parameters and calculates and updates the radius by dividing the diameter by 24 to convert it to inches.

Bundle

The Bundle class represents a bundle of conductors used in power systems. It includes methods to calculate the equivalent distances for series inductance (DSL) and shunt capacitance (DSC).

Attributes

- name (str): The name of the bundle.
- num_conductors (float): The number of conductors in the bundle.
- spacing (float): The spacing between conductors.
- conductor (Conductor): An instance of the Conductor class.
- resistance (float): The resistance of the Bundle
- DSL (float): The equivalent distance for series inductance
- DSC (float): The equivalent distance for shunt capacitance

Methods

- __ init __(name: str, num_conductors: float, spacing: float, conductor: Conductor): Initializes Bundle object with given parameters, validates spacing and num_conductor parameters, and calculates the Bundle's resistance, DSL, and DSC. If there is an invalid parameter, it raises an error.

- calculate_DSL(self) → float: Calculates and returns the equivalent distance for series inductance (DSL) based on the number of conductors; if there are more than 4 conductors, an error is raised.

- calculate_DSC(self) → float: Calculates and returns the equivalent distance for shunt capacitance (DSC) based on the number of conductors. If there are more than 4 conductors, an error is raised.

Geometry

The Geometry class represents the geometric configuration of a transmission line, with methods to calculate the equivalent distance (Deq) based on the coordinates of three points.

Attributes

- name (str): The name of the geometry.
- xa (float): The x-coordinate of point A.
- ya (float): The y-coordinate of point A.
- xb (float): The x-coordinate of point B.
- yb (float): The y-coordinate of point B.
- xc (float): The x-coordinate of point C.
- yc (float): The y-coordinate of point C.
- Deq (float): The equivalent distance calculated based on the coordinates of points A, B, and C.
- Dab(float): The distance between points A and B.
- Dbc(float): The distance between points B and C.
- Dca(float): The distance between points C and A.

Methods

- __init__(self, name: str, xa: float, ya: float, xb: float, yb: float, xc: float, yc: float): initializes a Geometry object with the given parameters and calls the calculate_Deq method to find the equivalent distance and updates the variable Deq.

- calculate_Deq(self) → float: Calculates and returns the equivalent distance (Deq) based on the distances between the points A, B, and C.

Transmission Line

The TransmissionLine class represents a transmission line in a power system, with methods to calculate its series impedance, admittance, Y-matrix, and base values.

Attributes

- name (str): The name of the transmission line.
- bus1 (Bus): The starting bus of the transmission line.
- bus2 (Bus): The ending bus of the transmission line.
- bundle (Bundle): The bundle of conductors used in the transmission line.
- geometry (Geometry): The geometric configuration of the transmission line.
- length (float): The length of the transmission line.
- z_base (float): The impedance base for the transmission line.
- y_base (float): The admittance base for the transmission line.
- r (float): The series resistance of the transmission line.
- x (complex): The series reactance of the transmission line.
- z (complex): The series impedance of the transmission line.
- y_shunt (complex): The shunt susceptance of the transmission line.
- y_series (complex): The series admittance of the transmission line.
- r_pu (float): The series resistance of the transmission line in per-unit.
- x_pu (complex): The series reactance of the transmission line in per-unit.
- z_pu (complex): The series impedance of the transmission line in per-unit.
- y_shunt_pu (complex): The shunt susceptance of the transmission line in per-unit.
- y_series_pu (complex): The series admittance of the transmission line in per-unit.
- y_prim (Pandas DataFrame): The primitive admittance matrix as a Pandas DataFrame.
- matrix (Dictionary): The primitive admittance matrix as a dictionary.

Methods

- __init__(self, name: str, bus1: Bus, bus2: Bus, bundle: Bundle, geometry: Geometry, length: float): initializes a TransmissionLine object with the given parameters; calls methods to calculate series impedance, admittance, Y-matrix, and base values.

- calc_y_prim(self) → Pandas DataFrame: Calculates primitive admittance matrix for the transmission line; uses a dictionary to assign values to the proper location, then stores in a DataFrame. Returns the y_prim DataFrame.

- print_y_prim(self): Prints the Y-matrix of the transmission line in a formatted table using the Pandas DataFrame.

Transformer Class

The Transformer class represents an electrical transformer with attributes and methods to calculate various electrical parameters.

Attributes

- name (str): The name of the transformer.
- bus1 (Bus): The starting bus of the transformer.
- bus2 (Bus): The ending bus of the transformer.
- power_rating (float): The power rating of the transformer in MVA.
- impedance_percent (float): The impedance of the transformer as a percentage.
- x_over_r_ratio(float): imaginary divided by the real part of the transformer impedance
- settings (Settings): The current settings of the transformer.
- z (float): The series impedance of the transformer.
- y (float): The series admittance of the transformer.
- r (float): The series resistance of the transformer.
- x (float): The series reactance of the transformer.
- x_pu(float): The series reactance of the transformer in per-unit.
- r_pu(float): The series resistance of the transformer in per-unit.
- z_pu(float): The series impedance of the transformer in per-unit.
- y_pu(float): The series admittance of the transformer in per-unit.
- y_prim(Pandas DataFrame): The primitive admittance matrix of the transformer as a Pandas DataFrame.
- matrix (Dict[float,float]]): The dictionary representation of the primitive admittance matrix.

Methods

- __init__(self, name: str, bus1: Bus, bus2: Bus, power_rating: float, impedance_percent: float, x_over_r_ratio: float): Initializes the transformer with the given parameters and calculates initial values for impedance and admittance.

- calc_y_prim(self) → Pandas DataFrame: Calculates the primitive admittance matrix as a NumPy array, converts to a Pandas DataFrame, and updates the y_prim and matrix attributes. Returns y_prim.

- print_yprim(self) → None: Prints the admittance matrix in a tabular format using pandas in per-unit format.

Generator Class

The Generator class represents an electrical generator connected to a bus in a power system. It contains important parameters such as the voltage setpoint, MW (megawatt) setpoint, and the bus to which the generator is connected.

Attributes

- name (str): Name of the generator.
- voltage_setpoint (float): The voltage setpoint of the generator, measured in per unit (pu).
- mw_setpoint (float): The power setpoint of the generator in megawatts (MW).
- bus (Bus):The bus to which the generator is connected.
- settings (Settings): An instance of the Settings class defining the global base power level.
- X1 (float): The positive sequence reactance of the generator.
- X2 (float): The negative sequence reactance of the generator.
- X0 (float): The zero sequence reactance of the generator.

Load Class

The Load class represents an electrical load connected to a bus within a power system. It contains essential information about the load, such as its real power, reactive power, and the bus to which it is connected.

Attributes

- name (str): The name of the load.
- bus (Bus): The bus to which the load is connected.
- real_pwr (float): The real power consumed by the load, measured in megawatts(MW).
- reactive_pwr (float): The reactive power consumed by the load, measured in mega volt-amperes reactive (MVAR).
- settings (Settings): An instance of the Settings class defining the global base power level.

Circuit Class

The Circuit class represents an electrical circuit, containing collections of buses, transformers, geometries, conductors, and transmission lines. It provides methods to add these components to the circuit while ensuring no duplicates are added.

Attributes

- name (str): The name of the circuit.
- buses (Dict[str, Bus]): A dictionary to store Bus objects, keyed by their names.
- busRef(List[str]): list of the bus dictionary in order to keep track of indexing within large ybus
- transformers (Dict[str, Transformer]): A dictionary to store Transformer objects, keyed by their names.
- geometries (Dict[str, Geometry]): A dictionary to store Geometry objects, keyed by their names.
- conductors (Dict[str, Conductor]): A dictionary to store Conductor objects, keyed by their names.
- transmissionlines (Dict[str, TransmissionLine]): A dictionary to store TransmissionLine objects, keyed by their names.
- settings (Settings): A Settings object that stores the circuit's base power and frequency values.
- loads (Dict[str, Load]): A dictionary to store Load objects, keyed by their names.
- generators (Dict[str, Load]): A dictionary to store Generator objects, keyed by their names.
- y_bus (Pandas DataFrame): A Pandas Dataframe to store the circuit's admittance matrix.

Methods

- __init__(self, name: str): Initializes a Circuit object with the provided parameters and initializes empty dictionaries for buses, transformers, geometries, conductors, and transmission lines

- add_bus(self, bus: Bus) → None: Adds a Bus object to Circuit object; checks if bus with same name exists and prints error message if so; otherwise, adds the bus to the buses dictionary.

- add_transformer(self, transformer: Transformer) → None: Adds a Transformer object to the circuit; checks if a transformer with the same name already exists in the circuit and prints error message if so; otherwise, adds the transformer to the transformers dictionary.

- add_geometry(self, geometry: Geometry) → None: Adds a Geometry object to the circuit; checks if a geometry with the same name already exists in the circuit and prints error message if so; otherwise, adds the geometry to the geometries dictionary.

- add_conductor(self, conductor: Conductor) → None: Adds a Conductor object to the circuit; checks if a conductor with the same name already exists in the circuit and prints error message if so; otherwise, adds the geometry to the conductors dictionary.

- add_transmissionline(self, transmissionline: TransmissionLine) → None: Adds a TransmissionLine object to the circuit; checks if a transmission line with the same name already exists in the circuit and prints error message if so; otherwise, adds the transmission line to the transmissionlines dictionary.

- add_load(self, load: Load) → None: Adds a Load object to the circuit; checks if a load with the same name already exists in the circuit and prints error message if so; otherwise, adds the load to the loads dictionary.
- add_generator(self, generator: Generator) → None: Adds a Generator object to the circuit; checks if a generator with the same name already exists in the circuit and prints error message if so; otherwise, adds the generator to the generators dictionary.

- calc_y_bus(self) → Pandas DataFrame: Constructs an admittance matrix for the circuit for all buses. Returns the matrix as a Pandas DataFrame.

- print_y_bus(self) → None: Prints the admittance matrix for the circuit; checks if the y_bus matrix has been constructed and prints an error message if not; otherwise, prints the y_bus DataFrame.

Solution Class

The Solution class is designed to perform power flow calculations for an electrical circuit using a Newton-Raphson algorithmic approach or fault analysis based on user selection. It uses numerical methods to compute power mismatches, adjust voltage and phase angles, and iteratively solve for steady-state power system solutions.

Attributes

- Circuit (Circuit): Stores the circuit object passed during initialization.
- known_power (NumPy array): Known power values.
- power (NumPy array): General power values.
- mismatch (NumPy array): Stores power mismatch calculations.
- jacob (Jacobian): Stores Jacobian object initialized from Circuit object passed during initialization.
- j_matrix (NumPy array): Jacobian matrix for the power flow calculations.
- solutionVec (NumPy array)t: Solution vector initialized as zeros.
- calcPQ (NumPy array): Stores calculated power values.
- knownPQ (NumPy array): Stores known power values after processing.
- initSol (NumPy array): Stores the initial solution vector values.
- slack_name (str): Stores the name of the bus associated with the slack generator.
- pv_name (str): Stores the name of the bus associated with an additional generator (the PV bus).
- y_bus_positive_matrix (NumPy array): Stores the positive sequence y-bus matrix.
- y_bus_negative_matrix (NumPy array): Stores the negative sequence y-bus matrix.
- y_bus_zero_matrix (NumPy array): Stores the zero sequence y-bus matrix.
- z_bus_positive (NumPy array): Stores the positive sequence z-bus matrix.
- z_bus_negative (NumPy array): Stores the negative sequence z-bus matrix.
- z_bus_zero (NumPy array): Stores the zero sequence z-bus matrix.

Methods

- calc_known_power(self) → None: Computes and stores known power values from loads and generators in the circuit; adjusts values based on circuit settings and prints the final known power values.

- calc_mismatch(self) → NumPy array: Computes the difference between known power values and calculated power values; filters out buses based on type (pv, slack). Returns the adjusted mismatch array.

- calc_solutionRef(self) → NumPy array: Generates the initial solution vector for voltage and phase angle values; applies numerical corrections using the Jacobian matrix and mismatch values. Returns the updated solution vector.

- calc_jacobian(self) → NumPy array: Calls the Jacobian class method to compute the Jacobian matrix. Returns the calculated Jacobian matrix.

- calc_solution(self) → NumPy array: Iteratively solves for power system voltage and phase angle values using the Newton-Raphson algorithm; adjusts values across 50 iterations or until convergence tolerance is met. Returns a 2D NumPy array containing the final voltage magnitudes and phase angles.

- print_jacobian(self) → None: Calls the Jacobian class method to print the computed Jacobian matrix.

- calc_z_bus_postive(self) → NumPy array: Calculates the positive sequence y-bus matrix, including the subtransient reactances of the generators.

- calc_z_bus_negative(self) → NumPy array: Calculates the negative sequence y-bus matrix, including the subtransient reactances of the generators.

- calc_z_bus_zero(self) → NumPy array: Calculates the zero sequence y-bus matrix, including the subtransient reactances of the generators.

Jacobian Class

The Jacobian class calculates the Jacobian matrix for power flow analysis using numerical methods. The Jacobian matrix is crucial in iterative load flow solutions for adjusting voltage and phase angle variables.

Attributes

- Circuit (Circuit): Stores the electrical circuit object containing system data passed in during initalization.
- Y_bus (NumPy array): Admittance matrix for power system analysis.
- slackI (int): Index of the slack bus in the circuit.
- pvI (int): Index of the PV bus in the circuit.
- j_matrix (NumPy array): The final Jacobian matrix used in numerical calculations.
- j_df: (Pandas DataFrame): DataFrame representation of the Jacobian matrix for better visualization.
- Size (int): Number of buses in the power system.
- j1 (NumPy array): Submatrix used in Jacobian calculation.
- j2 (NumPy array): Submatrix used in Jacobian calculation.
- j3 (NumPy array): Submatrix used in Jacobian calculation.
- j4 (NumPy array): Submatrix used in Jacobian calculation.

Methods

find_buses(self) → None: Determines the indices of slack and PV buses for matrix adjustments prints the identified indices.

calc_jacobian(self) → NumPy array: Computes the full Jacobian matrix by assembling four submatrices (J1, J2, J3, J4). Returns the final Jacobian matrix.

calc_j1(self) → NumPy array: Computes the partial derivatives related to active power with respect to bus angle (dP/dTheta); removes slack bus row and colum. Returns the J1 submatrix.

calc_j2(self) → NumPy array: Computes the partial derivatives related to active power with respect to bus voltage (dP/dV); removes slack bus row and column, and PV bus column. Returns the J2 submatrix.

calc_j3(self) → NumPy array: Computes the partial derivatives related to reactive power with respect to bus angle (dQ/dTheta); removes slack bus row and column, and PV bus row. Returns the J3 submatrix.

calc_j4(self) → NumPy array: Computes the partial derivatives related to reactive power with respect to bus voltage (dQ/dV); removes slack bus row and column, and PV bus row and column. Returns the J4 submatrix.

print_jacobian(self) → None: Converts the final Jacobian matrix into a Pandas DataFrame; prints the matrix with formatted values for readability.

Test

The Test file validates the functionality of the component class files (Transformer, Transmission LIne, Bundle, Geometry, and Bus). Each section of code creates an instance of each component, calls the calculation methods, and prints the results as well as the expected, hand-calculated values for visual inspection and validation.

- The first section, under heading “Transformer Validation”, creates a Transformer object and validates its properties and calculations. It checks the transformer's name, bus connections, power rating, impedance percentage, and ratio. It also calculates and prints the transformer's impedance (z) and admittance (y) as well as the expected values for this instance.

- The subsequent section, under heading “Conductor Validation”, creates a Conductor object and prints its properties, including name, diameter, geometric mean radius (GMR), resistance, and amperage as well as the expected properties for validation.

- The third section, under heading “Bus Validation”, creates two Bus objects and validates their properties, including name, base voltage, and index.

- The next section, under heading “Bundle Validation”, creates a Bundle object and prints its properties, including name, number of conductors, spacing, and derived series capacitance (DSC) and shunt inductance (DSL).

- The fifth section, under heading “Geometry Validation”, creates a Geometry object and validates its properties, including name and coordinates as well as the equivalent distance (Deq).

- The sixth section, under heading “Transmission Line Validation”, creates a TransmissionLine object and validates its properties, including name, length, impedance (Z), admittance (Y), and susceptance (B). It also prints the transmission line's admittance matrix using the included print_yprim method.

- The final section, under heading “Circuit Validation”, creates a Circuit object and validates its constructor and properties, checking if there is an empty bus dictionary and the correct name. Next, the methods are validated, including adding buses, transformers, and transmission lines,

Main

This file tests the Solution code; a seven-bus Circuit is created called circuit that includes 2 Transformer objects, 2 Generator objects, 6 TransmissionLine objects, and 3 Load objects attached to the 7 Bus objects. The Settings object is initialized with a global power base of 100 MW and a frequency of 60 Hz.

The admittance matrix, y_bus, is calculated and printed using the Circuit methods calc_y_bus() and print_y_bus. The values displayed in the terminal can be visually compared to the file jacobian.xlsx found in the folder PowerWorld Files.

Next, the solver is initialized with the name solution, a Solution object with parameter circuit. Then, using the Solution methods calc_known_power() and calc_power_mismatch(), the mismatch vector, comparing mismatches between power and angles, is computed. The length of the mismatch() is printed for validation, correctly displaying as 11.

Then, the power flow is solved using the Newton-Raphson algorithm programmed in the Solution method calc_solution(). The output reads the number of iterations needed to solve (11), and that the solution has been successfully reached.

Finally, a fault study is run with a fault on Bus 3, utilizing the Fault methods calc_3_phase_bal(), print_fault_voltages(), calc_line_to_line(), calc_single_line_to_ground(), and calc_double_line_to_ground(). These can be visually inspected with the fault voltages produced by PowerWorld, as found in the file Bus3Fault.xlsx in the PowerWorld Files folder.

Fault Class

The Fault class is designed to analyze different types of faults in a power system using the admittance and impedance matrices of the circuit. It calculates fault currents and voltages for various fault scenarios, including three-phase balanced faults, single line-to-ground faults, line-to-line faults, and double line-to-ground faults.

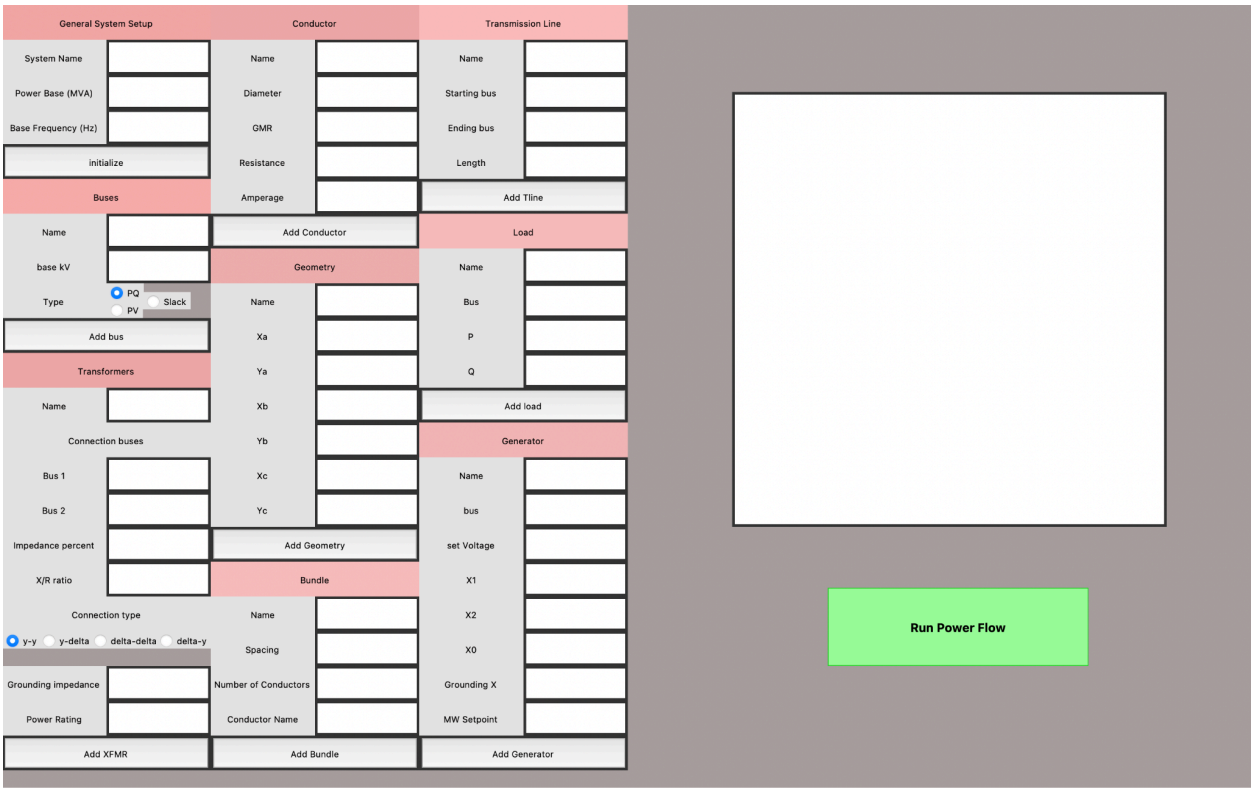
Attributes

- Circuit (Circuit): Stores the electrical circuit object containing system data passed in during initialization.
- z_bus_positive (NumPy array): Stores the positive sequence z-bus matrix.
- z_bus_negative (NumPy array): Stores the negative sequence z-bus matrix.
- z_bus_zero (NumPy array): Stores the zero sequence z-bus matrix.
- z_bus (NumPy array): Stores the z-bus matrix with no generator impedances for 3 phase balanced fault analysis.
- y_bus_matrix (NumPy array): Stores the y-bus matrix with no generator impedances for 3 phase balanced fault analysis.
- y_bus_matrix_positive (NumPy array): Stores the positive sequence y-bus matrix.
- y_bus_matrix_negative (NumPy array): Stores the negative sequence y-bus matrix.
- y_bus_matrix_zero (NumPy array): Stores the zero sequence y-bus matrix.
- slack_name (str): Stores the name of the bus associated with the slack generator.
- pv_name (str): Stores the name of the bus associated with an additional generator (the PV bus).

Methods

- calc_3_phase_bal (fault_bus: str, Zf: float = 0) → NumPy array, complex: Calculates the fault currents and voltages for a three-phase balanced fault at the specified bus.
- calc_line_to_line (fault_bus: str, Zf: float = 0): Calculates the fault currents and voltages for a line-to-line fault at the specified bus.
- calc_single_line_to_ground (fault_bus: str, fault_v: float = 1, Zf: float = 0):
- calc_double_line_to_ground (fault_bus: str, Zf: float = 0):
- calc_fault_voltages (Vf, Z0, Z1, Z2, I0, I1, I2) → NumPy array: Calculates the sequence voltages at the faulted bus.
- print_fault_voltages ():
- sequence_to_phase (sequence: List) → NumPy array: Converts a list of sequence components to phase components using the transformation matrix.

GUI Class



Enhancement Overview

The enhancement I developed for Project 3 is a Graphical User Interface (GUI) for the Newton-Raphson power flow solver. This GUI leverages the modular design of the backend solver and provides a more user-friendly and intuitive experience for system creation and simulation.

Each component within the solver — such as buses, transformers, conductors, geometries, bundles, transmission lines, loads, and generators — has dedicated input fields where users can specify the necessary characteristics (e.g., transmission line length, load MW and MVar, generator voltage setpoints).

Once the system is fully constructed, the Run Power Flow button executes the backend solver, displaying final bus voltages and angles in a formatted output window.

Purpose

After working with the backend power flow solver purely through hard-coded main.py files, it became clear that manually entering system setups required a higher attention to detail and was prone to more errors than interacting with a more visual program like PowerWorld.

The purpose of the GUI enhancement is to:

- Improve user experience by allowing users to build the system interactively.
- Reduce user error by providing structured input fields.

- Streamline system creation by allowing users to add components through button clicks rather than manual Python code.
- Enhance usability for users with less coding experience.

This GUI enables users to build even complex systems faster and more accurately compared to writing manual input scripts.

Inputs and Outputs

Inputs:
Inputs are entered into textboxes and radio buttons in the GUI. Typical inputs include:

- Bus: Name (e.g., bus1), Base kV, Bus Type (PQ, PV, Slack)
- Transformer: Bus names, impedance, X/R ratio, connection type
- Conductor: Diameter, GMR, resistance, ampacity
- Geometry: Coordinates for phase conductors
- Bundle: Number of conductors, spacing
- Transmission Line: Start bus, end bus, length
- Load: Associated bus, real power (P), reactive power (Q)
- Generator: Bus, voltage setpoint, reactances (X1, X2, X0), grounding impedance

Note: All bus names must follow the format busX, generators as Gen X, and loads as loadX to match backend dictionary keys.

Outputs:

- System setup log
- Validation messages (e.g., missing buses, invalid inputs)
- Final bus voltage magnitudes and phase angles
- Newton-Raphson solver convergence status and iteration count
- Intermediate outputs such as Y-bus matrix summaries, known powers, and Jacobian setup

References

- Used ChatGPT to learn how to build and organize a Tkinter GUI for modular component addition.
- https://stackoverflow.com/questions/2310130/how-to-write-gui-in-python

Instructions for Running and Testing

Setup:

- Download or clone the repository branch containing the GUI files and backend modules (Settings.py, Circuit.py, Bus.py, etc.).
- Open a terminal window (Mac/Linux) or command prompt (Windows).

Running the Program:

- Instead of pressing "Run" inside PyCharm, type the following command manually into the terminal:
  python3 GUI.py

  This will launch the interactive GUI window.
  Important: Full-screen the window to access all input sections easily.

Building the System:

- Start by initializing the system (enter system name, base frequency, and power base).
- Add each component by filling out the respective fields and clicking their buttons.
- Always add at least one Slack bus before running power flow.
- Once your system is built, click the "Run Power Flow" button to compute the voltages and angles.

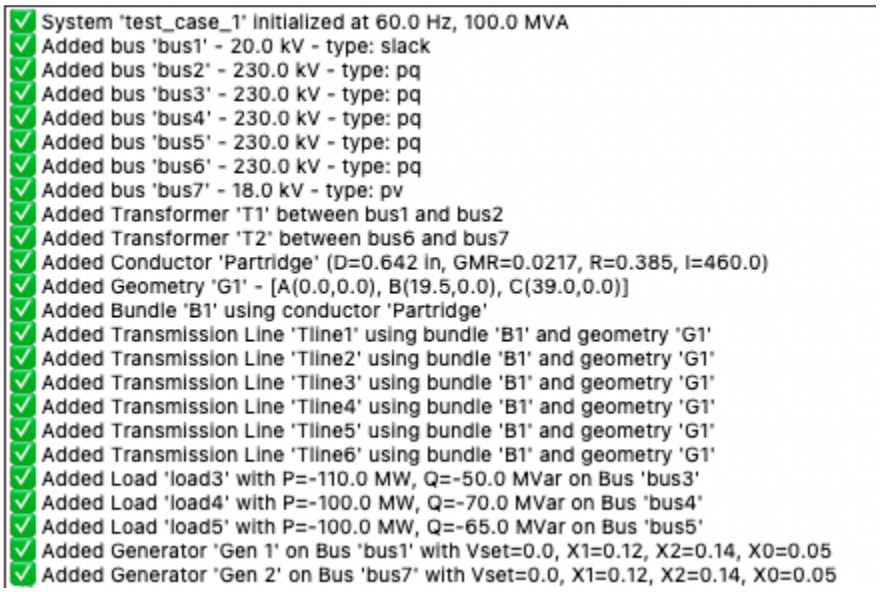
Key Assumptions:

- Only one Bundle and one Geometry are primarily used by default.
- Transmission lines use the last-added bundle and geometry automatically.
- Multiple bundles and geometries can be created, but the latest entry will be used unless customized further.

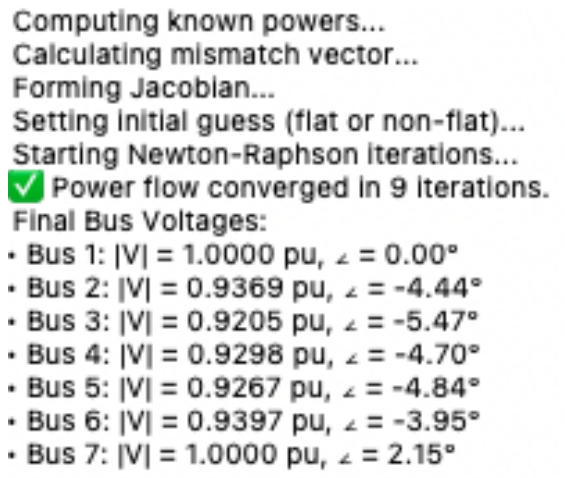
Validation Cases

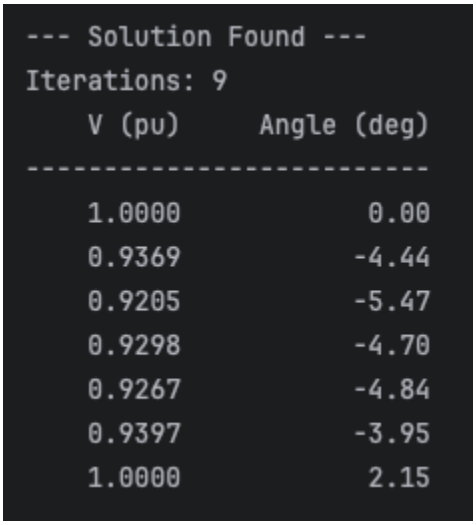
Case 1: Test Case 1 — 7-Bus System

System Setup:



The above information is displayed to the user when designing a power system. For efficiency, I made sure not to make a mistake when adding components, however, my GUI has error checking for things like duplicate bus names, adding transmission lines onto buses that don't exist, and not allowing the user to start adding components until power base and frequency are decided on.





After adding all necessary components, the user can then hit the green ‘run power flow’ button and the output box will provide the above screenshot for systems that converge. Conversely,

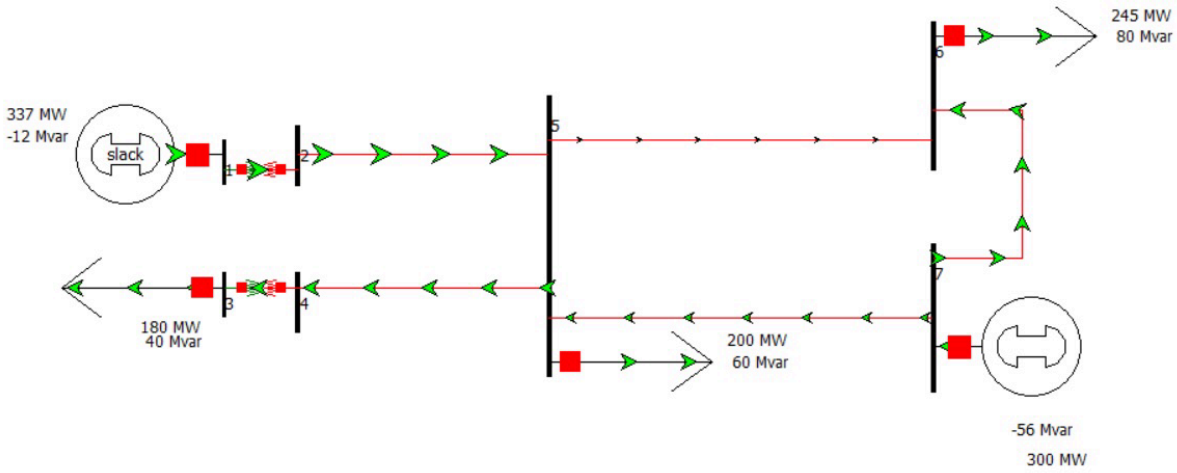
running this system without the GUI outputs identical values, which are provided in the right figure above.

Bus Number	Name	Area Name	Nom kV	PU Volt	Volt (kV)	Angle (Deg)
1	1	1	20	1	20	0
2	2	1	230	0.93693	215.494	-4.45
3	3	1	230	0.9205	211.715	-5.47
4	4	1	230	0.92981	213.856	-4.7
5	5	1	230	0.92674	213.15	-4.84
6	6	1	230	0.93969	216.129	-3.95
7	7	1	18	0.99999	18	2.15

Provided above are the exported final bus voltages and angles from PowerWorld. These values align with my solved system in Python, indicating the system is working correctly.

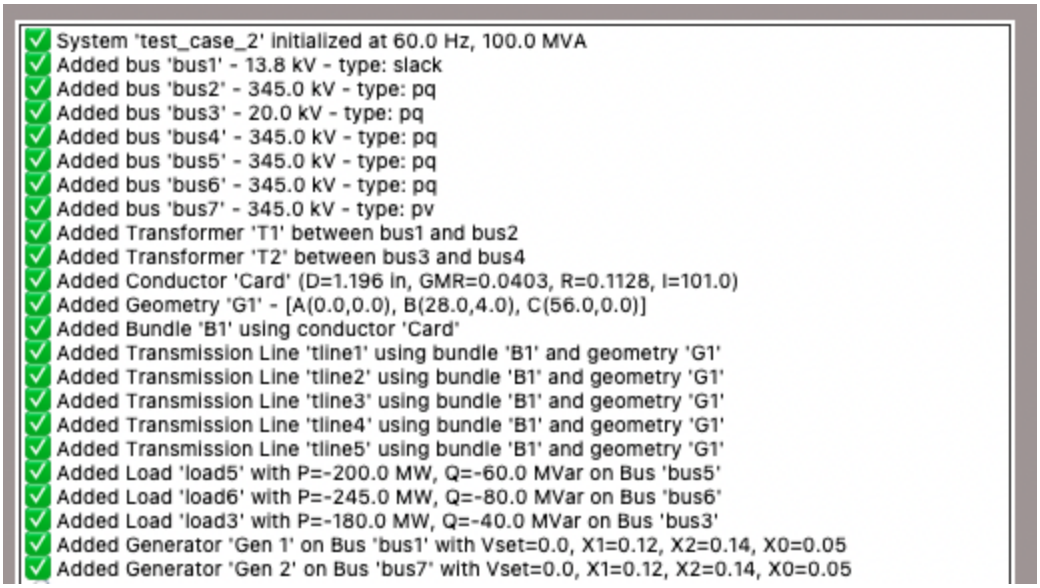
Case 2: Test Case 2 — 7-Bus System (Different Configuration)

System Setup:



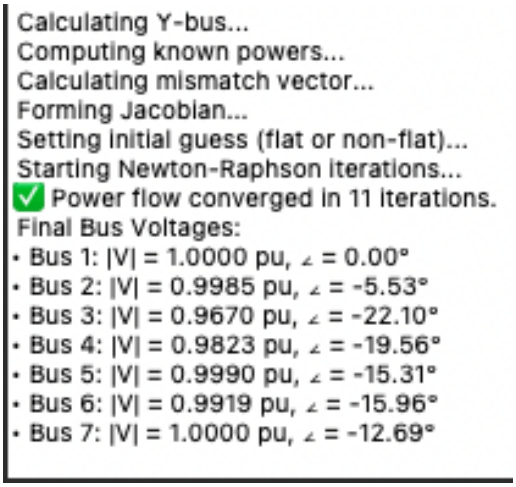
Provided above is a screenshot from Powerworld to visualize the second system I set up to validate my system for case 2.

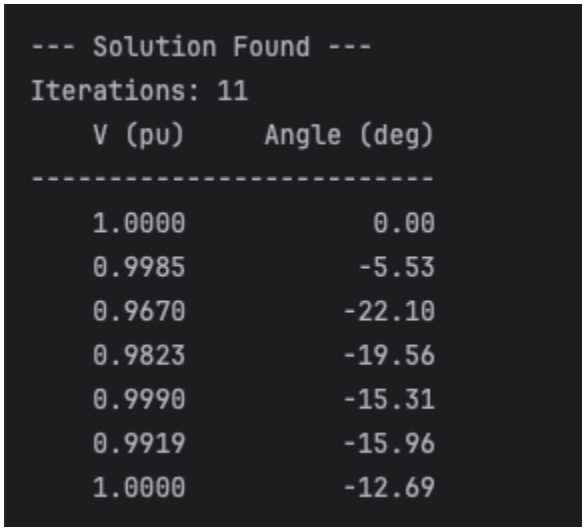
Similarly, here is what is displayed to the user as they would input this case into the GUI.



As expected, similar outputs are produced for this second system as the circuit is being set up.

After all components have been added, the run power flow provides the following output in the output box. As well as running this system without the GUI, it provides identical values.



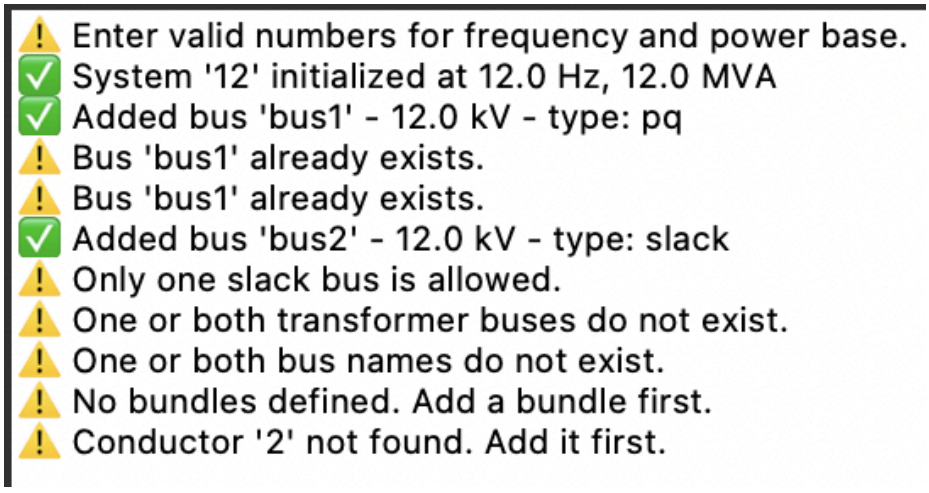


Finally, to validate these results, provided below are the Excel-exported final bus voltages and angles, which can be seen to match up with the previous results.

Bus Number	Name	Area Name	Nom kV	PU Volt	Volt (kV)	Angle (Deg)
1	1	1	13.8	1	13.8	0
2	2	1	345	0.99854	344.496	-5.53
3	3	1	20	0.96704	19.341	-22.1
4	4	1	345	0.98232	338.9	-19.56
5	5	1	345	0.99898	344.649	-15.31
6	6	1	345	0.99188	342.197	-15.96
7	7	1	345	1.00001	345.003	-12.69

Edge Cases:

With the system validated, I decided to add in some error detection. A key characteristic of these edge cases is rather than sending the user a large error message and terminating the program, the system will warn the user and not allow that component to be added until the message is addressed. Provided below are some of the edge case warning messages I incorporated within my system.



- If the user tries to initialize the system with a frequency like ‘sixty Hz’, the system will consider that invalid, similarly for the power base.
- If the user tries to add a bus with a name that already exists, the system will warn the user and not allow that bus to be added. Continually, only one Slack bus will be permitted per system
- For transmission lines and transformers, the system will not allow the component to be added if the bus the user inputted does not exist. As well as if the user has not defined a bundle, geometry, and/or conductor for the transmission lines to use.