Advanced Power Systems Simulator
Language: Python
Group Members: Luke Barnett, Jacqueline Dillon, Ryan Sieber

Section 1: Object Classes

These Classes are used to represent various elements of power systems and their mathematical/physical properties.

Bus Class

This class creates a Bus object. Busses refer to any connection in a power system that has a specific nodal voltage related to its connections to other electrical components in the system

Syntax:

Setting bus data:

```
setbusdata(self, name, type: str, real_P: float, Q_or_V: float):

Example:
MainGrid.setBusData("Bus1", "Slack Bus", 0, 0)
```

Input Argument Name	Description	Value in example
Name	String name reference to generator	“Bus1”
type	Bus type, can be slack, load, or voltage controlled	“Slack Bus”
Real_P	The bus’s real power	0
Q or V	Either the bus’s per unit voltage V, or its reactive power Q depending on the bus type	0

Class Code: located in repository as Bus.py

```
# Bus Class

class Bus:

    def __init__(self, name: str):

        # Counter to track how many exist
        counter = 0

        # Create the bus with its name
        def __init__(self, name: str):
            self.name = name
            self.index = Bus.counter

            # Increase Counter
            Bus.counter = Bus.counter + 1

    # Power Flow Setting Bus Data
    def setbusdata(self, type: str, real_P: float, Q_or_V: float):
        # If the bus is a Slack Bus, set necessary parameters
        if type == "Slack Bus":
            self.type = "Slack Bus"
            self.V = 1.0
            self.delta = 0.0
            self.P = 0.0
            self.Q = 0.0

        # If the bus is a Load Bus, set necessary parameters
        elif type == "Load Bus":
            self.type = "Load Bus"
            self.V = 0.0
            self.delta = 0.0
            self.P = real_P
            self.Q = Q_or_V

        # If the bus is a Voltage Controlled Bus, set necessary parameters
        elif type == "Voltage Controlled Bus":
            self.type = "Voltage Controlled Bus"
            self.V = Q_or_V
            self.delta = 0.0
            self.P = real_P
            self.Q = 0.0

        # If the bus is typed wrong or invalid, print Error Message
        else:
            print("Type not accepted. Enter Slack Bus, Load Bus, or Voltage Controlled Bus.")
            exit(-1)
```

Generator Class

This class creates a Generator object. Generators are used in power systems to control input voltage levels.

Syntax:

Defining a generator object:

```
Generator(name, bus1, nominalpower, x1gen, x2gen, x0gen,
            grounding_type, grounding_value)
```

```
Example:
newGenerator = Generator("G1", "Bus1", 100, 0.12, 0.14, 0.05, "Solid ground", 0)
```

Input Argument Name	Description	Value in example
Name	String name reference to generator	“G1”
Bus1	String reference to the node bus connection’s name	“Bus1”
Nominal power	Generator’s output power under standard test conditions	100 MVA
X1gen	Generator’s per unit positive sequence impedance	0.12
X2gen	Generator’s per unit negative sequence impedance	0.14
X0gen	Generator’s per unit zero sequence impedance	0.05
Grounding_type	Generator’s grounding connection type	“Solid Ground”
Grounding_value	Generator’s grounding connection value	0 Ω

Class Code: located in repository as Generator.py

```
# Generator Class
class Generator:

    # Generator Contains a name, bus, nominalpower (MVA), positive sequence impedance, negative sequence impedance, and the zero sequence impedance
    def __init__(self, name, bus1, nominalpower, x1gen, x2gen, x0gen, grounding_type, grounding_value):
        self.name = name
        self.bus1 = bus1
        self.nominalpower = nominalpower
        self.x1gen = x1gen
        self.x2gen = x2gen
        self.x0gen = x0gen
        self.grounding_type = grounding_type
        self.grounding_value = grounding_value
```

Transformer Class

This class creates a transformer object. Transformers are used in power systems to transfer energy from one circuit to another, often stepping up or down this voltage.

Syntax:

Defining a transformer object:

```
Transformer(name, bus1, bus2, apparentpower, v1rated, v2rated, impedance, xrratio,
            Sbase, Zt_connection1, Zt_grounding1, Zt_value1, Zt_connection2,
            Zt_grounding2, Zt_value2)

Example:
newTransformer = Transformer("T1", "Bus1", "Bus2", 125, 20, 230, 0.085, 10, 100,
                             "Delta", "N/A", 0, "Grounded Wye", "Resistor", 1)
```

Input Argument Name	Description	Value in example
Name	String name reference to transformer	“T1”
Bus1	String reference to the first node bus connection’s name	“Bus1”
Bus2	String reference to the second node bus connection’s name	“Bus2”
Apparentpowerrating	The transformers’ rated apparent power, in MVA	125 MVA
V1rated	The transformers’ primary side rated voltage, in kV	20 kV
V2rated	The transformers’ secondary side rated voltage, in kV	230 kV
Impedance	The transformers’ impedance, in pu	0.085
Xrratio	The transfromers’ reactance to resistance ratio	10
Sbase	The power systems base apparent power, in MVA	100 MVA
Zt_connection1	The primary side connection type, “Delta” or “Grounded Wye”	“Delta”
Zt_grounding1	Grounding connection on primary side	“N/A”
Zt_value1	Value of transformer’s grounding connection on primary side	0 Ω
Zt_connection2	The secondary side connection type, “Delta” or “Grounded Wye”	“Grounded Wye”
Zt_grounding2	Grounding connection on secondary side	“Resistor”
Zt_value2	Value of transformer’s grounding connection on secondary side	1 Ω

Other values stored in this object class:

Variable reference name	Description
`self.Rpu`	The calculated per unit resistance of the transformer
`self.Xpu`	The calculated per unit reactance of the transformer
`self.powerloss`	Power loss value is stored when store_power_loss function is referenced, used elsewhere in the simulator

Relevant Equations:

$$Z\_pu = Z * \frac{Z_{rated}}{Z_{base}} \qquad where\ Z_{rated} = \frac{{V_{2,rated}}^2}{S_{rated}} \quad and \quad Z_{base} = \frac{{V_{base}}^2}{S_{base}}$$

Class Code: located in repository as Transformer.py

```
# Transformer class
import numpy


class Transformer:

    # Transformer has base parameters name, bus1, bus2, apparentpower, v1rated, v2rated, impedance, xrratio, Sbase, Zt_connection1, Zt_grounding1, Zt_value1, Zt_connection2, Zt_grounding2, Zt_value2
    def __init__(self, name, bus1, bus2, apparentpowerrating, v1rated, v2rated, impedance, xrratio, Sbase, Zt_connection1, Zt_grounding1, Zt_value1, Zt_connection2, Zt_grounding2, Zt_value2):

        # Set base values
        self.name = name
        self.bus1 = bus1
        self.bus2 = bus2
        self.apparentpowerrating = apparentpowerrating
        self.v1rated = v1rated
        self.v2rated = v2rated
        self.impedance = impedance
        self.xrratio = xrratio
        self.powerloss = 0
        self.Zt_connection1 = Zt_connection1
        self.Zt_grounding1 = Zt_grounding1
        self.Zt_value1 = Zt_value1
        self.Zt_connection2 = Zt_connection2
        self.Zt_grounding2 = Zt_grounding2
        self.Zt_value2 = Zt_value2

        # Establish Sbase and Vbase
        self.Sbase = Sbase  # MVA
        Vbase = v2rated  # kV

        # Calculate Z Real and Z imaginary for the transformer
        self.Rpu = impedance * (v2rated * v2rated / apparentpowerrating)/(Vbase * Vbase/Sbase) * numpy.cos(numpy.arctan(xrratio))
        self.Xpu = impedance * (v2rated * v2rated / apparentpowerrating)/(Vbase * Vbase/Sbase) * numpy.sin(numpy.arctan(xrratio))

    def store_power_loss(self, powerloss):
        self.powerloss = powerloss
```

Transmission Line Class

This class creates a Transmission Line object. Transmission Lines carry power from one area/component of a circuit to another.

Defining a Transmission Line object:

```
    TransmissionLine((self, name: str, bus1: str, bus2: str, lengthmi: float,
                        axaxis: float, ayaxis: float, bxaxis: float, byaxis:
                        float, cxaxis: float, cyaxis: float,
                        codeword: str, numberofbundles: int, seperationdistance:
                        float, Vbase: float, Sbase: float):
Example:
    newLine = TransmissionLine ("L1", "Bus2", "Bus4", 10, 0, 0, 19.5, 0, 39, 0,
                                "Partridge", 2, 1.5)
```

Input Argument Name	Description	Value in example
Name	String name reference to transformer	“T1”
Bus1	String reference to the first node bus connection’s name	“Bus1”
Bus2	String reference to the second node bus connection’s name	“Bus2”
Lengthmi	Length of the line in miles, float value	10
Axaxis	Phase a’s spatial location, x coordinate	0
Ayaxis	Phase a’s spatial location, y coordinate	0
Bxaxis	Phase b’s spatial location, x coordinate	19.5
Byaxis	Phase b’s spatial location, y coordinate	0
Cxaxis	Phase c’s spatial location, x coordinate	39
Cyaxis	Phase c’s spatial location, y coordinate	0
Codeword	The string referring to the type of material the transmission line is composed of. Corresponds to specific physical characteristics associated with this material.	“Partridge”
Numberofbundles	The amount of conductors in the bundle, from 1 to 4 usually	2
separationdistance	The amount of space between the conductor bundles, in ft	1.5 ft

TransmissionLine is the parent to the TransmissionLineBundles class

This class assigns the appropriate Geometric mean radius, conductor radius r, and resistance depending on the codeword. For the sake of this simulator, the only codeword used is "Partridge" and it assigns the following values:

Variable	Value Assigned based on codeword "Partridge"
`Self.GMR`	`0.2604`
`Self.r`	`0.321`
`Self.resistancepermi`	`0.385`

TransmissionLineBundles calculates the following physical characteristics:

Variable reference in code	**Description**	**Equation used (for 2 bundles)**
`Self.DSL`	Equivalent inductance, depending on number of bundles	$DSL = \sqrt{GMR \cdot separationdistace}$
`Self.DSC`	Equivalent capacitance, depending on number of bundles	$DSC = \sqrt{R \cdot separationdistace}$
`Self.R`	Equivalent Resistance, depending on number of bundles	$R = \frac{resistacepermi}{numberofbundles}$

TransmissionLineBundles Code:

```
# TransmissionLineBundles class
# Used to find DSL and DSC given the number of bundles, distance between bundles in feet, and codeword of conductor

class TransmissionLineBundles:
    def __init__(self, numberofbundles: int, distance: float, codeword: str):

        # Change distance in feet to inches
        distance = distance * 12

        # If statement to set GMR, r, and resistance values for that specific codeword
        if codeword == "Partridge":
            self.GMR = 0.2604  # in inches
            self.r = 0.321  # in inches
            self.resistancepermi = 0.385 # in ohms per mile

        # If there is 1 bundle, set values
        if numberofbundles == 1:
            self.DSL = self.GMR
            self.DSC = self.r

        # If there are 2 bundles, set values
        elif numberofbundles == 2:
            self.DSL = (self.GMR * distance) ** (1 / 2)
            self.DSC = (self.r * distance) ** (1 / 2)

        # If there are 3 bundles, set values
        elif numberofbundles == 3:
            self.DSL = (self.GMR * distance * distance) ** (1 / 3)
            self.DSC = (self.r * distance * distance) ** (1 / 3)

        # If there are 4 bundles, set values
        elif numberofbundles == 4:
            self.DSL = (self.GMR * distance ** 3) ** (1 / 4)
            self.DSC = 1.0941 * (self.r * distance ** 3) ** (1 / 4)

        # Calculate R per mile
        self.R = self.resistancepermi / numberofbundles
```

Relevant Equations:

$$D_{eq} = \sqrt[3]{D_{ab}D_{bc}D_{ca}} \quad where \;\; D_{uv} = \sqrt{(U_x - V_x)^2 - (U_y - V_y)^2}$$

$$C(per\ mile) = \frac{2 * \pi * \varepsilon_0}{ln(\frac{D_{eq}}{D_{SC}})}$$

$$L(per\ mile) = 2 * 10^{-7} * 1609.355 * ln(\frac{D_{eq}}{D_{SL}})$$

Class Code: located in repository as TransmissionLine.py

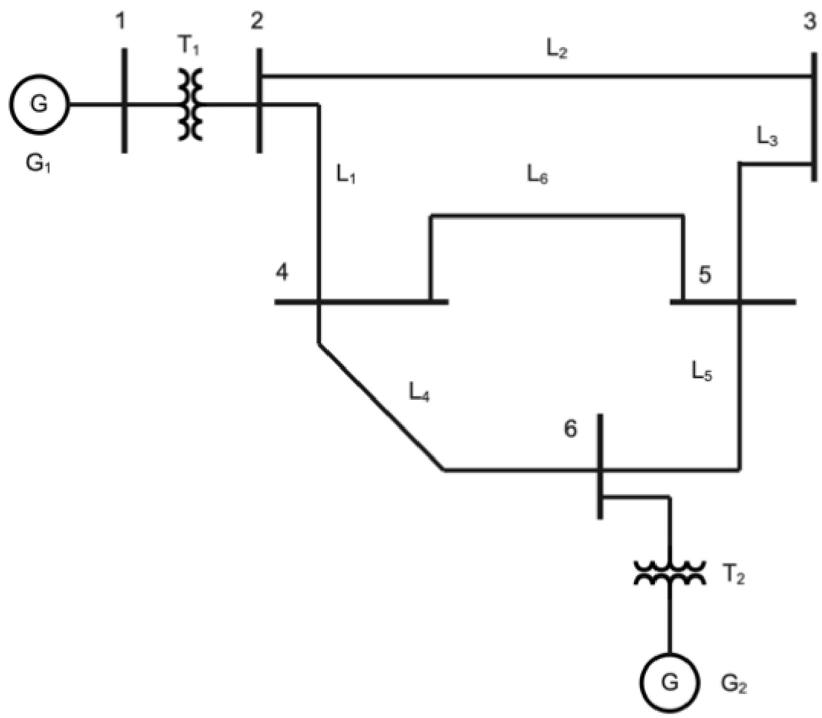
```
import math
import numpy

# TransmissionLineBundles contains codeword, DSC, DSL, and resistance per feet information
from TransmissionLineBundles import TransmissionLineBundles


class TransmissionLine:

    # By default, transmission line has a name, bus to bus location, length  (miles), coordinates for each phase,
    # A codeword for the conductor type, number of bundles per phase, separation of bundles per phase, Vbase, and Sbase
    def __init__(self, name: str, bus1: str, bus2: str, lengthmi: float,
                 axaxis: float, ayaxis: float, bxaxis: float, byaxis: float, cxaxis: float, cyaxis: float,
                 codeword: str, numberofbundles: int, seperationdistance: float, Vbase: float, Sbase: float):

        # Set name, buses, and length of line
        self.name = name
        self.bus1 = bus1
        self.bus2 = bus2
        self.lengthmi = lengthmi
        self.numberofbundles = numberofbundles
        self.powerloss = None

        # Set S base, frequency, and calculate Z base
        Zbase = Vbase**2/Sbase
        frequency = 60

        # Calculate Equivalent Distance between lines (in feet)
        Dab = ((axaxis - bxaxis) ** 2 - (ayaxis - byaxis) ** 2) ** (1/2)
        Dbc = ((bxaxis - cxaxis) ** 2 - (byaxis - cyaxis) ** 2) ** (1/2)
        Dca = ((cxaxis - axaxis) ** 2 - (cyaxis - ayaxis) ** 2) ** (1/2)
        dequivalent = (Dab * Dbc * Dca) ** (1/3)

        # Create a Bundles object that contains the desired values from the TranmissionLineBundles class
        Bundles = TransmissionLineBundles(numberofbundles, seperationdistance, codeword)

        # Store the variables from that class
        DSL = Bundles.DSL
        DSC = Bundles.DSC
        Rpermi = Bundles.R

        # Calculate Capacitance values
        CFpermi = ((2 * numpy.pi * 8.8541878128 * (10 ** (-12))) / (math.log(dequivalent * 12 / DSC))) * 1609.344 # F/mi
        Ctotal = CFpermi * lengthmi # Farads

        # Calculate Inductance Values
        LFpermi = (2 * (10 ** (-7)) * math.log(dequivalent * 12 / DSL)) * 1609.344 # F/mi
        Ltotal = LFpermi * lengthmi # Farads

        # Calculate total resistance of line
        self.Rtotal = Rpermi * lengthmi

        # Use R to get R per unit
        self.Rpu = self.Rtotal / Zbase

        # Use L to get X per unit
        self.Xpu = Ltotal * 2 * numpy.pi * frequency / Zbase

        # Use C to get B per unit
        self.Bpu = Ctotal * 2 * numpy.pi * frequency * Zbase

    def store_power_loss(self, powerloss):
        self.powerloss = powerloss
```

Section 2: Simulation Classes

Grid Class

This class physically lays out the power grid that is being analyzed in the simulation. For this specific simulation, the following grid was used

6-Node Looped Transmission System



Variable reference in code	Description
self.Ybus	Empty 7x7 admittance matrix that is populated using the below equations in Calculate_Ybus function
self.Rpu	Per unit resistance imported from either the Transformer or Transmission Line class, depending on bus connections
self.Xpu	Per unit reactance imported from either the Transformer or Transmission Line class, depending on bus connections
self.Bpu	Per unit susceptance imported from the Transmission Line class

Relevant Equations:

Non-diagonal elements:

$$Ybus_{kn} = Ybus_{nk} = -\sum \frac{1}{Z_{kn}} = \sum \frac{1}{Rpu_{kn} + j \cdot Xpu_{kn}}$$

Diagonal elements:

$Ybus_{kk} = -\sum \frac{1}{Rpu_{kn}+j \cdot Xpu_{kn}} + \sum_{i=1}^{I} \frac{j \cdot Bpu_{ki}}{2}$, where i is the i^{th} transmission line from k to n.

Class Code: located in repository as Grid.py

```
# import Dictionaries, List, and numpy
from typing import Dict, List
import numpy
import sys

# import files the Grid contains
from Bus import Bus
from Generator import Generator
from Transformer import Transformer
from TransmissionLine import TransmissionLine

# Create grid class to contain all buses, generators, transformers, and transmission lines
class Grid:

    # Pass the name of the Grid upon initialization
    def __init__(self, name: str):

        # Set name of Grid
        self.name: str = name

        # Establish bus orders and a dictionary for the buses
        self.buses_order: List[str] = list()
        self.buses: Dict[str, Bus] = dict()

        # Create a dictionary for generators, transformers, and transmissionlines, with the key being their name
        self.generators: Dict[str, Generator] = dict()
        self.transformers: Dict[str, Transformer] = dict()
        self.transmissionline: Dict[str, TransmissionLine] = dict()

        # Parameters for all Files
        self.Sbase = 100 # In units of MVA
        self.Vbase = 230 # In units of kV
        self.convergencevalue = 0.0001 #pu
        self.Q_k_limit = 175000000 #VA

    # Function to add a bus, by first making sure the bus does not already exist
    def __add_bus(self, bus):
        if bus not in self.buses.keys():
            self.buses[bus] = Bus(bus)
            self.buses_order.append(bus)

    # Function to add a transformer to the grid. It takes a name, initial bus, final bus, apparentpower (MVA),
    # low side voltage rating (kV), high side voltage rating (kV), impedance (pu), xrratio, and grounding information
    def add_transformer(self, name: str, bus1: str, bus2: str, apparentpower: float,
                        v1rated: float, v2rated: float, impedance: float, xrratio: float,
                        Zt_connection1: str, Zt_grounding1: str, Zt_value1: float, Zt_connection2: str, Zt_grounding2: str, Zt_value2: float):

        # Check for errors before adding transformer, Sequence Network data will only be checked if user needs a fault calculation
        self.error_check_transformer(bus1, bus2, apparentpower, v1rated, v2rated, impedance, xrratio)

        # Add transformer to dictionary with all of its values
        self.transformers[name] = Transformer(name, bus1, bus2, apparentpower, v1rated, v2rated, impedance, xrratio, self.Sbase, Zt_connection1, Zt_grounding1, Zt_value1, Zt_connection2, Zt_grounding2, Zt_value2)

        # Add the buses it is connected to
        self.__add_bus(bus1)
        self.__add_bus(bus2)

    # Function to add a transmission line to the grid.
    # It takes parameters name, initial bus, final bus, length (miles), line coordinates (ft.(inches/12)),
    # Codeword of wire, number of bundles, and spacing between bundles (ft.(inches/12))
    # This function will throw an error if two phases are in the same exact location,
    # the number of bundles is not an integer 1-4, or if the codeword entered does not have data stored
    def add_transmissionline(self, name: str, bus1: str, bus2: str, lengthmi: float,
                             axaxis: float, ayaxis: float, bxaxis: float, byaxis: float, cxaxis: float, cyaxis: float,
```

```
    # Function to add a transmission line to the grid.
    # It takes parameters name, initial bus, final bus, length (miles), line coordinates (ft.(inches/12)),
    # Codeword of wire, number of bundles, and spacing between bundles (ft.(inches/12))
    # This function will throw an error if two phases are in the same exact location,
    # the number of bundles is not an integer 1-4, or if the codeword entered does not have data stored
    def add_transmissionline(self, name: str, bus1: str, bus2: str, lengthmi: float,
                             axaxis: float, ayaxis: float, bxaxis: float, byaxis: float, cxaxis: float, cyaxis: float,
                             codeword: str, numberofbundles: int, seperationdistance: float):

        # Check for errors before adding line
        self.error_check_transmission_line(bus1, bus2, lengthmi, axaxis, ayaxis, bxaxis, byaxis, cxaxis, cyaxis, codeword, numberofbundles, seperationdistance)

        # Add Vbase as the high voltage from the first transformer entered
        Vbase = self.transformers[list(self.transformers.keys())[0]].v2rated

        # Add transmission line to dictionary
        self.transmissionline[name] = TransmissionLine(name, bus1, bus2, lengthmi, axaxis, ayaxis, bxaxis, byaxis, cxaxis, cyaxis, codeword, numberofbundles, seperationdistance, Vbase, self.Sbase)

        # Add the buses it is connected to
        self.__add_bus(bus1)
        self.__add_bus(bus2)

    # Function to add a generator. Paramters are a name, bus, nominalpower (MVA), positive sequence impedance, negative sequence impedance, and the zero sequence impedance (per unit impedances)
    def add_generator(self, name, bus1, nominalpower, x1gen, x2gen, x0gen, grounding_type, grounding_value):

        # Check for errors before adding generator
        self.error_check_generator(nominalpower)

        # Add generator to dictionary
        self.generators[name] = Generator(name, bus1, nominalpower, x1gen, x2gen, x0gen, grounding_type, grounding_value)
        self.__add_bus(bus1)

    # Function to calculate the Ybus. There are no input parameters, and it outputs the Ybus matrix.
    # This function only works for a 7 bus network with the specified grid given in class.
    def calculate_Ybus(self):

        # Create a matrix with all zeros depending on how many buses are in the system of complex variables
        self.Ybus = numpy.zeros((len(self.buses_order), len(self.buses_order)), dtype=complex)

        # Assign Values to each component

        # Set Non-diagonals just using -1/Z
        self.Ybus[0][1] = -1 / (self.transformers["T1"].Rpu + 1j * self.transformers["T1"].Xpu) # T1
        self.Ybus[1][0] = self.Ybus[0][1] # T1
        self.Ybus[1][2] = -1 / (self.transmissionline["L2"].Rpu + 1j * self.transmissionline["L2"].Xpu) # L2
        self.Ybus[2][1] = self.Ybus[1][2] # L2
        self.Ybus[3][1] = -1 / (self.transmissionline["L1"].Rpu + 1j * self.transmissionline["L1"].Xpu) # L1
        self.Ybus[1][3] = self.Ybus[3][1] # L1
        self.Ybus[4][2] = -1 / (self.transmissionline["L3"].Rpu + 1j * self.transmissionline["L3"].Xpu) # L3
        self.Ybus[2][4] = self.Ybus[4][2] # L3
        self.Ybus[4][3] = -1 / (self.transmissionline["L6"].Rpu + 1j * self.transmissionline["L6"].Xpu) # L6
        self.Ybus[3][4] = self.Ybus[4][3] # L6
        self.Ybus[5][3] = -1 / (self.transmissionline["L4"].Rpu + 1j * self.transmissionline["L4"].Xpu) # L4
        self.Ybus[3][5] = self.Ybus[5][3] # L4
        self.Ybus[5][4] = -1 / (self.transmissionline["L5"].Rpu + 1j * self.transmissionline["L5"].Xpu) # L5
        self.Ybus[4][5] = self.Ybus[5][4] # L5
        self.Ybus[6][5] = -1 / (self.transformers["T2"].Rpu + 1j * self.transformers["T2"].Xpu) # T2
        self.Ybus[5][6] = self.Ybus[6][5] # T2

        # Set diagonals, do not include generators, include capacitances here
        # Use previous matrix values plus shunt charging values for necessary transmission lines
        self.Ybus[0][0] = -self.Ybus[0][1]  # G1, T1
        self.Ybus[1][1] = -self.Ybus[0][1] - self.Ybus[1][3] - self.Ybus[1][2] + (1j * self.transmissionline["L1"].Bpu / 2) + 1j * self.transmissionline["L2"].Bpu / 2# T1, L1, L2
        self.Ybus[2][2] = -self.Ybus[1][2] - self.Ybus[2][4] + (1j * self.transmissionline["L2"].Bpu / 2) + 1j * self.transmissionline["L3"].Bpu / 2           # L2, L3
        self.Ybus[3][3] = -self.Ybus[1][3] - self.Ybus[3][5] - self.Ybus[3][4] + (1j * self.transmissionline["L1"].Bpu / 2) + 1j * self.transmissionline["L4"].Bpu / 2 + 1j * self.transmissionline["L6"].Bpu / 2  # L1, L4, L6
        self.Ybus[4][4] = -self.Ybus[2][4] - self.Ybus[4][5] - self.Ybus[3][4] + (1j * self.transmissionline["L3"].Bpu / 2) + 1j * self.transmissionline["L5"].Bpu / 2 + 1j * self.transmissionline["L6"].Bpu / 2  # L3, L5, L6
        self.Ybus[5][5] = -self.Ybus[3][5] - self.Ybus[4][5] - self.Ybus[5][6] + (1j * self.transmissionline["L4"].Bpu / 2) + 1j * self.transmissionline["L5"].Bpu / 2  # L4, L5, T2
        self.Ybus[6][6] = -self.Ybus[5][6]  # G2, T2

    # Store bus data from main
    def setBusData(self, bus: str, bustype: str,  real_P: float, Q_or_V: float):
        self.buses[bus].setbusdata(bustype, real_P, Q_or_V)

    # Store power loss calculated from Power Flow class
    def store_power_loss(self, name: str, powerloss):
        self.transmissionline[name].store_power_loss(powerloss)

    def store_power_loss_transformer(self, name:str, powerloss):
        self.transformers[name].store_power_loss(powerloss)

    # Function to check errors in transmission line
    def error_check_transmission_line(self, bus1, bus2, lengthmi, axaxis, ayaxis, bxaxis, byaxis, cxaxis, cyaxis, codeword, numberofbundles, seperationdistance):
```

```python
    # List of acceptable codewords
    codewordlist = ["Partridge"]

    # If two phases are in the same location, throw an error
    if (axaxis == bxaxis and ayaxis == byaxis) or (bxaxis == cxaxis and byaxis == cyaxis) or (
            axaxis == cxaxis and ayaxis == cyaxis):
        sys.exit("Error. Phases can not be in the same location. Enter phases at different coordinates.")

    # If number of bundles was less than 1 or more than 4, throw an error
    if numberofbundles < 1 or numberofbundles > 4:
        sys.exit("Error. Number of Bundles not accepted. Enter a value 1-4")

    # If codeword does not have data stored, throw an error
    if codeword not in codewordlist:
        sys.exit("Error: Codeword not accepted. Enter a different conductor type.")

    # If transmission line begins and ends at same bus, throw error
    if bus1 == bus2:
        sys.exit("Error: Cannot end a transmission line to the same bus it begins. Enter different buses.")

    # If transmission line length is negative or 0, throw an error
    if lengthmi <= 0:
        sys.exit("Error: Cannot have a negative or zero transmission line length. Enter a positive value.")

    # If separation distance is negative, throw an error
    if seperationdistance < 0:
        sys.exit("Error: Cannot have a negative bundle separation distance. Enter a positive value.")

# Function to check errors in generator
def error_check_generator(self, nominalpower):

    # If nominal power is negative, throw an error
    if nominalpower < 0:
        sys.exit("Error: Cannot have a negative generator power. Enter a positive value.")

# Function to check errors in generator
def error_check_transformer(self, bus1, bus2, apparentpower, v1rated, v2rated, impedance, xrratio):

    # If transmission line begins and ends at same bus, throw error
    if bus1 == bus2:
        sys.exit("Error: Cannot end a transformer at the same bus it begins. Enter different buses.")

    # If apparent power is less than zero, throw error
    if apparentpower < 0:
        sys.exit("Error: Cannot have a negative apparent power in the transformer. Enter a positive value.")

    # If v1rated is negative, throw an error
    if v1rated < 0:
        sys.exit("Error: Cannot have a negative v1 rated voltage. Enter a positive value.")

    # If v2rated is negative, throw an error
    if v2rated < 0:
        sys.exit("Error: Cannot have a negative v2 rated voltage. Enter a positive value.")

    # If impedance is negative, throw an error
    if impedance < 0:
        sys.exit("Error: Cannot have a negative transformer impedance. Enter a positive value.")

    # If xrratio is negative, throw an error
    if xrratio < 0:
        sys.exit("Error: Cannot have a negative transformer xrratio. Enter a positive value.")
```

Newton_Raphson_Power_Flow Class

This class implements the Newton-Raphson Power Flow Solver, which solves the power flow equations for an AC power system using the Newton-Raphson method. This method calculates the Jacobian and power mismatches through multiple iterations until the maximum power mismatch fits the convergence criteria. This class also calculates the maximum reactive power and adds a capacitor bank of the limit is exceeded. Further, this class calculates current in each line and ensures no values exceed the max ampacity. It also calculates the power flowing through each line, power losses in each line, and the total power losses of the system. Finally, this class outputs the real and reactive power injected into the slack bus and the reactive power injected into the PV bus.

Variable reference in code	Description
P_mismatch	Real power mismatches
Q_mismatch	Reactive power mismatches
mismatchPQ	Combined array of Real and Reactive power mismatches
J11	Top left quadrant of Jacobian matrix
J12	Top right quadrant of Jacobian matrix
J21	Bottom left quadrant of Jacobian matrix
J22	Bottom right quadrant of Jacobian matrix
self.V_complex	Array for the complex voltage values
self.I_values	Array of the current values in each line
self.P_inj_slack	Real power injected into slack bus
self.Q_inj_slack	Reactive power injected into slack bus
self.Q_inj_PV	Reactive power injected into PV bus
Self.Vbase	Base voltage for the system, given by v2rated of 1st transformer
powerloss	Total power loss of the system, calculated using every element within of transformer and transmission line dictionaries

Variable Name	Description	Value
self.convergencevalue	Maximum mismatchPQ value at which Newton-Raphson converges	0.0001
V	Initial voltage array for a flat start	[1 1 1 1 1 1 1]
delta	Initial phase angle array for a flat start	[0 0 0 0 0 0 0]
self.Q_k_limit	The maximum allowable reactive power in the system	175 MVAR

Relevant Equations:

Power Mismatches:

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

Jacobian Off Diagonal Elements

$$J11_{kn} = V_k Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn})$$
$$J12_{kn} = V_k Y_{kn} \cos(\delta_k - \delta_n - \theta_{kn})$$
$$J21_{kn} = -V_k Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn})$$
$$J22_{kn} = V_k Y_{kn} \sin(\delta_k - \delta_n - \theta_{kn})$$

Jacobian Diagonal Elements

$$J11_{kk} = -V_k \sum_{n=1,\, n\neq k}^{N} Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn})$$
$$J12_{kk} = V_k Y_{kk} cos\theta_{kk} + \sum_{n=1}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn})$$
$$J21_{kk} = V_k \sum_{n=1,\, n\neq k}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn})$$
$$J22_{kk} = -V_k Y_{kk} \sin\theta_{kk} + \sum_{n=1}^{N} Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn})$$

$$J = \left[\frac{J11\ J12}{J21\ J22}\right] \qquad \left[\frac{J11(i)\ J12(i)}{J21(i)\ J22(i)}\right]\left[\frac{\Delta\delta(i)}{\Delta V(i)}\right] = \left[\frac{\Delta P(i)}{\Delta Q(i)}\right]$$

$$x = \left[\frac{\delta}{V}\right] \qquad x(i+1) = \left[\frac{\delta(i+1)}{V(i+1)}\right] = \left[\frac{\delta(i)}{V(i)}\right] + \left[\frac{\Delta\delta(i)}{\Delta V(i)}\right]$$ **where i is the ith iteration**

Current Calculations

$$V\_complex_i = V_i \cdot \cos(\delta_i) + j \cdot V_i \cdot \sin(\delta_i)$$
$$I\_values = V\_complex \cdot Ybus$$

Power Losses

$$powerloss = I\_values \cdot Rpu^2$$

Calculated for each element and summed within code

Power Injections

$$P\_inj\_slack = \left| Ybus[slack\_bus,\ i] \cdot \frac{V\_complex[i]}{Vbase} \right| \cdot \cos\big(angle(V\_complex[slack\_bus])\big) - \\ - angle(V\_complex[i]) - angle(Ybus[slack\_bus,\ i])$$

$$Q\_inj\_slack = \left| Ybus[slack\_bus,\ i] \cdot \frac{V\_complex[i]}{Vbase} \right| \cdot \sin\big(angle(V\_complex[slack\_bus])\big) - \\ - angle(V\_complex[i]) - angle(Ybus[slack\_bus,\ i])$$

$$Q\_inj\_PV = \left| Ybus[VC\_bus,\ i] \cdot \frac{V\_complex[i]}{Vbase} \right| \cdot \sin\big(angle(V\_complex[VC\_bus])\big) - \\ - angle(V\_complex[i]) - angle(Ybus[VC\_bus,\ i])$$

where VC bus is voltage-controlled bus

Class Code: located in repository as Newton_Raphson_Power_Flow.py

```
# Newton_Rhapson
import numpy as np
import pandas as pd

from Grid import Grid

# Clean up comment
class NewtonRhapson:

    # Power flow
    def __init__(self, Grid, capacitor_bank):
        # add_cap is set so that 1 will add capacitor bank to lower generator MVAR, 0 to solve exceeded limit function
        # from user input
        self.add_cap = capacitor_bank

        # B will be the capactitor bank added
        self.B = 0
        # Start on iteration 1
        iteration = 1

        # Set to 0, if becomes 1 means an adjustment was recently made with the capacitor
        # bank and the program should reiterate
        self.capacitor_bank_adjustment = 0

        # Initialize paramters and blank arrays
        self.length = len(Grid.buses)
        self.P_loss = np.zeros((self.length, self.length))
        self.system_loss = 0
        self.P_inj_slack = 0
        self.Q_inj_slack = 0
        self.Q_inj_PV = 0
        self.Q_k_limit = Grid.Q_k_limit # In units of VA
        self.Q_limit_passed = 0
        self.S_values = np.zeros((self.length, self.length), dtype=complex)
        self.Q_values = np.zeros((self.length, self.length))
        self.P_values = np.zeros((self.length, self.length))
        self.I_values = np.zeros((self.length, self.length), dtype=complex)
        self.V_complex = np.zeros(self.length, dtype=complex)
        J11 = np.zeros((self.length - 1, self.length - 1))
        J12 = np.zeros((self.length - 1, self.length - 1))
        J21 = np.zeros((self.length - 1, self.length - 1))
        J22 = np.zeros((self.length - 1, self.length - 1))
        Parr = np.zeros(self.length)
        Qarr = np.zeros(self.length)
        self.convergencemet = 0
        self.convergencevalue = Grid.convergencevalue

        # Set base values
        self.Vbase = Grid.transformers[list(Grid.transformers.keys())[0]].v2rated * 1000  # In Volts
        self.Sbase = Grid.Sbase * 1000000  # VA
        self.Vbase_slack1 = Grid.transformers[list(Grid.transformers.keys())[0]].v1rated * 1000  # In Volts
        self.Vbase_slack2 = Grid.transformers[list(Grid.transformers.keys())[1]].v1rated * 1000  # In Volts

        # set blank array for given values
        P_given = np.zeros(self.length)
        Q_given = np.zeros(self.length)
        P_mismatch = np.zeros(self.length-1)
        Q_mismatch = np.zeros(self.length-2)
```

```
# find slack bus
self.slack_bus = None
for i in range(self.length):
    if Grid.buses["Bus" + str(i + 1)].type == "Slack Bus":
        self.slack_bus = i
    if Grid.buses["Bus" + str(i + 1)].type == "Voltage Controlled Bus":
        self.voltage_controlled_bus = i

# set given values
for i in range(self.length):
    if Grid.buses["Bus" + str(i + 1)].type == "Load Bus":
        P_given[i] = -Grid.buses["Bus" + str(i + 1)].P / 100
        Q_given[i] = -Grid.buses["Bus" + str(i + 1)].Q / 100
    else:
        P_given[i] = Grid.buses["Bus" + str(i + 1)].P / 100
        Q_given[i] = Grid.buses["Bus" + str(i + 1)].Q / 100

# Set self.Q_given so that Q_given can be used in other functions
self.Q_given = Q_given

# set intial guess
V = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
delta = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# Make sure convergence has not been met and that a capacitor bank was not just added, or that the max
# Iterations has not been exceeded
while (self.convergencemet == 0 and self.capacitor_bank_adjustment == 1) or iteration < 30:

    # Reset the calculated values every time through
    for i in range(self.length):
        Parr[i] = 0
        Qarr[i] = 0

    # calculate mismatch, ignoring the slack bus
    for i in range(self.length):
        # if slack bus, skip
```

```
    if i == self.slack_bus:
        continue

    for j in range(self.length):
        Parr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.cos(delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
        if i == self.voltage_controlled_bus:
            continue
        Qarr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.sin(delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

# P does not include slack bus, Q does not include Voltage controlled bus or slack bus
if self.slack_bus == 0:
    P_mismatch = P_given[1:] - Parr[1:]
    Q_mismatch = Q_given[1:6] - Qarr[1:6]
if self.slack_bus == 6:
    P_mismatch = P_given[0:6] - Parr[0:6]
    Q_mismatch = Q_given[1:6] - Qarr[1:6]

# Set the convergence met variable to 1
self.convergencemet = 1

# Check Power Mismatch for Convergence
mismatchPQ = np.concatenate((P_mismatch, Q_mismatch))
for i in range(len(mismatchPQ)):
    # If any value in the mismatched is greater than the convergence requirement, set convergence parameter
    # to not met
    if mismatchPQ[i] > self.convergencevalue:
        self.convergencemet = 0
        break
# If convergence has been met and there was no capacitor adjustment, notify user it converged
if self.convergencemet == 1 and self.capacitor_bank_adjustment == 0:
    break

# Calculate Jacobian Matrix
skipterm = 0
for i in range(self.length):
    # if slack bus skip
    if i == self.slack_bus:
        skipterm = 1
        continue
    for j in range(self.length):
        if j == self.slack_bus:
            continue

        # Diagonals have their own formula
        if i == j:
            for z in range(self.length):
                J12[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.cos(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))
                J22[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

                if z == i:
                    continue

                J11[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))
                J21[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.cos(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

            # Non-Summation Modification
            J11[i - skipterm, j - skipterm] = -J11[i - skipterm, j - skipterm] * V[i]
            J12[i - skipterm, j - skipterm] = J12[i - skipterm, j - skipterm] + (
                        V[i] * abs(Grid.Ybus[i, j]) * np.cos(np.angle(Grid.Ybus[i, j])))
            J21[i - skipterm, j - skipterm] = (J21[i - skipterm, j - skipterm] * V[i])
            J22[i - skipterm, j - skipterm] = J22[i - skipterm, j - skipterm] - (
                        V[i] * abs(Grid.Ybus[i, j]) * np.sin(np.angle(Grid.Ybus[i, j])))

        # If not a diagonal of the jacobian
        else:
            J11[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * V[j] * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J12[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J21[i - skipterm, j - skipterm] = -V[i] * abs(Grid.Ybus[i, j]) * V[j] * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J22[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

# Combine Jacobian
J = np.block([[J11, J12], [J21, J22]])

# Delete Voltage Controlled bus 1 or 7, here if it is 7
if self.slack_bus == 0:
    J_temp = np.delete(J, 11, 1)
    J = J_temp
    J_temp = np.delete(J, 11, 0)
    J = J_temp

# Here VCB is 1
if self.slack_bus == 6:
    J_temp = np.delete(J, 6, 1)
    J = J_temp
    J_temp = np.delete(J, 6, 0)
    J = J_temp
```

```python
            # Calculate inverse of the Jacobian
            J_inv = np.linalg.inv(J)

            # combine mismatches
            mismatch = np.concatenate((P_mismatch, Q_mismatch))

            # Find correction
            correction = np.dot(J_inv, mismatch)

            # Seperate the angle correction and the voltage correction
            delta_correction = correction[:6]
            V_correction = correction[6:]

            # do not update the angle of the slack bus
            delta_correction = np.concatenate((delta_correction[:self.slack_bus], [0], delta_correction[self.slack_bus:]),
axis=0)

            # do not update the voltage of the slack bus or voltage controlled bus
            V_correction = np.concatenate((V_correction[:6], [0], V_correction[6:]), axis=0)
            V_correction = np.concatenate((V_correction[:0], [0], V_correction[0:]), axis=0)

            # Update the angle and voltage
            delta += delta_correction
            V += V_correction

            # Increase iteration
            iteration += 1

            # Parameter to store Q from Voltage controlled bus
            self.Q_k = 0

            # Calculate Q_k
            for i in range(self.length):
                self.Q_k += abs(Grid.Ybus[self.voltage_controlled_bus, i]) * V[i] * np.sin(delta[self.voltage_controlled_bus] -
delta[i] - np.angle(Grid.Ybus[self.voltage_controlled_bus,i]))
            self.Q_k *= V[self.voltage_controlled_bus]
            self.Q_k *= self.Sbase

            # If Q from VCB has exceeded the limit, and a capacitor bank is not wanted
            if self.Q_k > self.Q_k_limit and self.add_cap == 0:
                print("VAR LIMIT EXCEEDED: Generator bus will no longer be a Voltage Controlled Bus")
                self.solve_exceeded_var_power_flow(Grid)
                self.Q_limit_passed = 1
                break

            # Reset the term to 0 to see if another adjustment needs made
            self.capacitor_bank_adjustment = 0

            # If Q from VCB has exceeded the limit, and a capacitor bank is wanted
            if self.Q_k > self.Q_k_limit and self.add_cap == 1:
                # Adjustment Needed
                print("LIMIT EXCEEDED, INCREASING CAPACITOR BANK")
                self.capacitor_bank_adjustment = 1
                # Add capacitor bank to highest MVAR load
                j = 0
                for i in range(self.length):
                    if self.Q_given[j] >= self.Q_given[i]:
                        j = i
                self.B += -1j * self.Q_given[j] * 100000000 / (self.Vbase ** 2) #* self.Sbase
                Grid.Ybus[j, j] += self.B

                # Reset iteration so that it does not exceed the iteration limit while adding banks
                iteration = 0
        # Check to make sure V_complex was not set up by Var limit solver
        if self.Q_limit_passed == 0:
            # Calculate the complex voltage
            for i in range(self.length):
                self.V_complex[i] = V[i] * np.cos(delta[i]) + 1j * V[i] * np.sin(delta[i])

        # Function to solve power flow
        self.solve_power_flow(Grid)

    def solve_power_flow(self, Grid):

        # Calculate per unit current value
        I_values_per_unit = np.zeros((self.length, self.length), dtype=complex)
        for i in range(self.length):
            for j in range(self.length):
                I_values_per_unit[i, j] = (self.V_complex[j] - self.V_complex[i]) * (Grid.Ybus[i, j])

        # Calculate actual current values
        self.I_values = (self.Sbase / (self.Vbase * np.sqrt(3))) * I_values_per_unit
        self.I_values[0, 1] *= self.Vbase / self.Vbase_slack1 # -> Because from Bus1 to 2
        self.I_values[6, 5] *= self.Vbase / self.Vbase_slack2 # -> Because from Bus7 to 6

        # Check ampacity limit
        self.check_ampacity()

        # Take voltage values out of per unit
        self.V_complex *= self.Vbase
        I_conjugate = np.conjugate(self.I_values)

        # Set up to find P and Q
        # Find P, S = P + jQ, and S = I_conjugate * V * sqrt(3)
        # Find S first
```

```
        for i in range(self.length):
            for j in range(self.length):
                self.S_values[i, j] = self.V_complex[i] * I_conjugate[i, j] * np.sqrt(3)  # because 3 phase
                # Transformer has different base
                if i==0:
                    self.S_values[i, j] = self.S_values[i, j]/self.V_complex[i] * self.Vbase_slack1
                # Transformer has different base
                if i==6:
                    self.S_values[i, j] = self.S_values[i, j]/self.V_complex[i] * self.Vbase_slack2

        # P is the real part of S, Q is the imaginary part of S
        self.P_values = np.real(self.S_values)
        self.Q_values = np.imag(self.S_values)

        self.system_loss = 0

        # Loop over every item in the transmission line dictionary, i is the name and value is the object
        for i1 in range(self.length):
            for i2 in range(self.length):
                for i, value in Grid.transmissionline.items():
                    if value.bus1 == ("Bus" + str(i1+1)) and value.bus2 == ("Bus" + str(i2+1)):
                        powerloss = abs(self.I_values[i1, i2]) ** 2 * value.Rtotal * 3 # because 3 phase
                        # Store the power loss, so it can be accessed through the grid class
                        Grid.store_power_loss(i, powerloss)
                        self.system_loss += powerloss

        # Same for transformers
        for i1 in range(self.length):
            for i2 in range(self.length):
                for i, value in Grid.transformers.items():
                    if value.bus1 == ("Bus" + str(i1+1)) and value.bus2 == ("Bus" + str(i2+1)) or value.bus1 == ("Bus" +
str(i2+1)) and value.bus2 == ("Bus" + str(i1+1)):
                        if abs(self.I_values[i1, i2]) == 0:
                            continue
                        powerloss = abs(self.I_values[i1, i2]) ** 2 * value.Rpu * self.Vbase ** 2/self.Sbase * 3 # because 3
phase
                        # Transformers have a different base
                        if value.name == "T1":
                            powerloss = powerloss / (self.Vbase ** 2) * self.Vbase_slack1 ** 2
                        if value.name == "T2":
                            powerloss = powerloss / (self.Vbase **2) * self.Vbase_slack2 ** 2
                        # Store the power loss in the transformer in the Grid class also
                        Grid.store_power_loss_transformer(i, powerloss)
                        # Update system power loss
                        self.system_loss += powerloss

        # POWER AND Q INJECTIONS FOR SLACK + PV
        for i in range(self.length):
            # Iterate through each Bus to sum up these values
            self.Q_inj_slack += abs(Grid.Ybus[self.slack_bus, i]) * abs(self.V_complex[i])/self.Vbase *
 np.sin(np.angle(self.V_complex[self.slack_bus]) - np.angle(self.V_complex[i]) - np.angle(Grid.Ybus[self.slack_bus, i]))
            self.P_inj_slack += abs(Grid.Ybus[self.slack_bus, i]) * abs(self.V_complex[i])/self.Vbase *
 np.cos(np.angle(self.V_complex[self.slack_bus]) - np.angle(self.V_complex[i]) - np.angle(Grid.Ybus[self.slack_bus, i]))
            self.Q_inj_PV += abs(Grid.Ybus[self.voltage_controlled_bus, i]) * abs(self.V_complex[i])/self.Vbase *
 np.sin(np.angle(self.V_complex[self.voltage_controlled_bus]) - np.angle(self.V_complex[i]) -
 np.angle(Grid.Ybus[self.voltage_controlled_bus, i]))

        # Convert to the values to their actual values instead of per unit
        self.P_inj_slack *= abs(self.V_complex[self.slack_bus])/self.Vbase * self.Sbase
        self.Q_inj_slack *= abs(self.V_complex[self.slack_bus])/self.Vbase * self.Sbase
        self.Q_inj_PV *= abs(self.V_complex[self.voltage_controlled_bus])/self.Vbase * self.Sbase

        # Print these values in "Mega" units
        print("self.P_inj_slack:", self.P_inj_slack/1000000, "MW")
        print("self.Q_inj_slack:", self.Q_inj_slack/1000000, "MVAR")
        print("self.Q_inj_PV:", self.Q_inj_PV/1000000, "MVAR")

        # Print I Values
        print("\nI_Values")
        for i in range(self.length):
            for j in range(self.length):
                # If the value is 0, do not print, if the value is negative, current is flowing the other way
                # So it will print out at a later iteration in the loop
                if abs(self.I_values[i, j]) == 0 or self.I_values[i, j] < 0:
                    continue
                # Only print Line currents, although transformer currents also matched with powerworld
                if i == 0 or j == 0 or i == 6 or j == 6:
                    continue
                print("B" + str(i + 1) + " to B" + str(j + 1) + " ", abs(self.I_values[i, j]))

        # Print the P values from the Sending End, meaning they are positive
        print("\nP_values Sending End")
        for i in range(self.length):
            for j in range(self.length):
                if abs(self.P_values[i, j]) == 0 or self.P_values[i, j] < 0:
                    continue
                print("Bus" + str(i + 1) + " to Bus" + str(j + 1) + " ", self.P_values[i, j] / 1000000, "MW")

        # Print the P values from the Receiving End, meaning they are negative
        print("\nP_values Receiving End")
        for i in range(self.length):
            for j in range(self.length):
                if abs(self.P_values[i, j]) == 0 or self.P_values[i, j] > 0:
                    continue
                print("Bus" + str(i + 1) + " from Bus" + str(j + 1) + " ", self.P_values[i, j] / 1000000, "MW")
```

```python
    # Print the Q values from the Sending End, meaning they are positive
    print("\nQ_values Sending End")
    for i in range(self.length):
        for j in range(self.length):
            if abs(self.Q_values[i, j]) == 0 or self.Q_values[i, j] < 0:
                continue
            print("Bus" + str(i + 1) + " to Bus" + str(j + 1) + " ", self.Q_values[i, j] / 1000000, "MVAR")

    # Print the Q values from the Sending End, meaning they are negative
    print("\nQ_values Receiving End")
    for i in range(self.length):
        for j in range(self.length):
            if abs(self.Q_values[i, j]) == 0 or self.Q_values[i, j] > 0:
                continue
            print("Bus" + str(i + 1) + " from Bus" + str(j + 1) + " ", self.Q_values[i, j] / 1000000, "MVAR")

    # Print Power Loss Per Line
    print("\nPower Loss Per Line")
    for i, value in Grid.transmissionline.items():
        print(i, value.powerloss/1000000, "MW")

    # Print Power Loss in the Transformers
    print("\nPower Loss in Transformers")
    for i, value in Grid.transformers.items():
        print(i, value.powerloss/1000000, "MW")

    # Print the total system power loss
    print("\nTotal System Power Loss:", self.system_loss/1000000, "MW")

# If the QVAR limit was exceeded previously, this function will be used if no capacitor bank is added.
# It has the same functionality as before, however there is no longer a Voltage controlled bus
def solve_exceeded_var_power_flow(self, Grid):
    # set blank array for given values
    P_given = np.zeros(self.length)
    Q_given = np.zeros(self.length)
    P_mismatch = np.zeros(self.length-1)
    Q_mismatch = np.zeros(self.length-1)
    # find slack bus
    for i in range(self.length):
        if Grid.buses["Bus" + str(i + 1)].type == "Slack Bus":
            self.slack_bus = i
        if Grid.buses["Bus" + str(i + 1)].type == "Voltage Controlled Bus":
            self.voltage_controlled_bus = i

    # set given values
    for i in range(self.length):
        if Grid.buses["Bus" + str(i + 1)].type == "Load Bus":
            P_given[i] = -Grid.buses["Bus" + str(i + 1)].P / 100
            Q_given[i] = -Grid.buses["Bus" + str(i + 1)].Q / 100
        else:
            P_given[i] = Grid.buses["Bus" + str(i + 1)].P / 100
            Q_given[i] = Grid.buses["Bus" + str(i + 1)].Q / 100
    Q_given[self.voltage_controlled_bus] = self.Q_k_limit / self.Sbase

    # set intial guess
    V = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
    delta = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# establish array for calculated value, without slack bus
Parr = np.zeros(self.length)
Qarr = np.zeros(self.length)
iteration = 1

while self.convergencemet == 0 and iteration < 30:
    for i in range(self.length):
        Parr[i] = 0
        Qarr[i] = 0
    # calculate mismatch, ignoring the slack bus
    for i in range(self.length):
        # if slack bus, skip
        if i == self.slack_bus:
            continue

        for j in range(self.length):
            Parr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            Qarr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

    # P does not include slack bus
    if self.slack_bus == 0:
        P_mismatch = P_given[1:7] - Parr[1:7]
        Q_mismatch = Q_given[1:7] - Qarr[1:7]
    if self.slack_bus == 6:
        P_mismatch = P_given[0:6] - Parr[0:6]
        Q_mismatch = Q_given[0:6] - Qarr[0:6]

    self.convergencemet = 1

    # Check Power Mismatch for Convergence
    mismatchPQ = np.concatenate((P_mismatch, Q_mismatch))
    for i in range(len(mismatchPQ)):
        if mismatchPQ[i] > self.convergencevalue:
            self.convergencemet = 0
```

```
        break

if self.convergencemet == 1:
    break

# Calculate Jacobian Matrix
J11 = np.zeros((self.length - 1, self.length - 1))
J12 = np.zeros((self.length - 1, self.length - 1))
J21 = np.zeros((self.length - 1, self.length - 1))
J22 = np.zeros((self.length - 1, self.length - 1))

skipterm = 0

for i in range(self.length):
    # if slack bus skip
    if i == self.slack_bus:
        skipterm = 1
        continue
    for j in range(self.length):
        if j == self.slack_bus:
            continue
        if i == j:
            for z in range(self.length):
                J12[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.cos(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))
                J22[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

                if z == i:
                    continue

                J11[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))
                J21[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.cos(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

            # Non-Summation Modification
            J11[i - skipterm, j - skipterm] = -J11[i - skipterm, j - skipterm] * V[i]
            J12[i - skipterm, j - skipterm] = J12[i - skipterm, j - skipterm] + (
                    V[i] * abs(Grid.Ybus[i, j]) * np.cos(np.angle(Grid.Ybus[i, j])))
            J21[i - skipterm, j - skipterm] = (J21[i - skipterm, j - skipterm] * V[i])
            J22[i - skipterm, j - skipterm] = J22[i - skipterm, j - skipterm] - (
                    V[i] * abs(Grid.Ybus[i, j]) * np.sin(np.angle(Grid.Ybus[i, j])))

        else:
            J11[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * V[j] * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J12[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J21[i - skipterm, j - skipterm] = -V[i] * abs(Grid.Ybus[i, j]) * V[j] * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J22[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

            # Combine Jacobian
            J = np.block([[J11, J12], [J21, J22]])

            # calculate change in voltage and phase angle
            J_inv = np.linalg.inv(J)

            # combine mismatches
            mismatch = np.concatenate((P_mismatch, Q_mismatch))
            correction = np.dot(J_inv, mismatch)

            delta_correction = correction[:6]
            V_correction = correction[6:]

            # do not update the angle of the slack bus
            delta_correction = np.concatenate((delta_correction[:self.slack_bus], [0], delta_correction[self.slack_bus:]),
axis=0)

            # do not update the voltage of the slack bus
            V_correction = np.concatenate((V_correction[:self.slack_bus], [0], V_correction[self.slack_bus:]), axis=0)

            # Correction
            delta += delta_correction
            V += V_correction
            iteration += 1


    def check_ampacity(self):
        i = 1
        j = 1
        while i < 6:
            while j < 6:
                if abs(self.I_values[i, j]) == 0:
                    j += 1
                    continue
                if self.I_values[i][j] >= 475:
                    print("Error: Ampacity limit exceeded")
                    exit(-1)
                j += 1
            j = 0
            i += 1
```

DC_Power_Flow_Solver Class

This class solves the power flow equations for a DC power system.

Class Code: located in repository as DC_Power_Flow_Solver.py

```
import numpy as np
from Grid import Grid
class DCPowerFlow:

    # Power flow
    def __init__(self, Grid):
        # DC Power Flow Solver does not work properly
        # Initial parameters that will be used later
        self.Q_inj_PV = 0
        self.P_inj_slack = 0
        self.Q_inj_slack = 0
        self.length = len(Grid.buses)
        self.P_loss = 0
        self.system_loss = 0
        self.slack_bus = None
        self.voltage_controlled_bus = None

        # Establish Empty Arrays
        P_given = np.zeros(self.length)
        self.voltage_pu = np.ones(self.length)
        self.S_values = np.zeros((self.length, self.length), dtype=complex)
        self.Q_values = np.zeros((self.length, self.length))
        self.P_values = np.zeros((self.length, self.length))
        self.I_values = np.zeros((self.length, self.length), dtype=complex)
        self.V_complex = np.zeros(self.length-1, dtype=complex)
        self.Bbus = np.zeros((self.length, self.length))
        self.Bbus_inv = np.zeros((self.length, self.length))


        # Establish Base Parameters
        self.Vbase = Grid.transformers[list(Grid.transformers.keys())[0]].v2rated * 1000  # In Volts
        self.Vbase_slack1 = Grid.transformers[list(Grid.transformers.keys())[0]].v1rated * 1000  # In Volts
        self.Sbase = Grid.Sbase * 1000000  # VA

        # Find slack bus and voltage controlled bus by looping through Buses Dictionary in Grid Class
        for i in range(self.length):
            if Grid.buses["Bus" + str(i + 1)].type == "Slack Bus":
                self.slack_bus = i
            if Grid.buses["Bus" + str(i + 1)].type == "Voltage Controlled Bus":
                self.voltage_controlled_bus = i

        # Set given values by looping through Buses Dictionary in Grid Class, Loads are negative
        for i in range(self.length):
            if Grid.buses["Bus" + str(i + 1)].type == "Load Bus":
                P_given[i] = -Grid.buses["Bus" + str(i + 1)].P / 100
            else:
                P_given[i] = Grid.buses["Bus" + str(i + 1)].P / 100
        self.P_given = P_given

        # Calculate Bbus and delete slack row and column
        self.Bbus = np.imag(Grid.Ybus)
        self.Bbus_whole = self.Bbus
        self.Bbus = np.delete(self.Bbus, self.slack_bus, 0)
        self.Bbus = np.delete(self.Bbus, self.slack_bus, 1)

        # Calculate Bbus and delete slack row and column
        self.Bbus = np.imag(Grid.Ybus)
        self.Bbus_whole = self.Bbus
        self.Bbus = np.delete(self.Bbus, self.slack_bus, 0)
        self.Bbus = np.delete(self.Bbus, self.slack_bus, 1)

        # Calculate the Bbus inverse
        self.Bbus_inv = np.linalg.inv(self.Bbus)
        temp_P_given = P_given

        # Get rid of slack bus for the P_given array
        if self.slack_bus == 0:
            temp_P_given = temp_P_given[1:7]

        if self.slack_bus == 6:
            temp_P_given = temp_P_given[0:6]

        # Calculate the angle values by Bbus_inv * P_given
        self.delta = -1 * np.dot(self.Bbus_inv, temp_P_given)

        # Print angle and voltage information
        #for i in range(self.length-1):
        #    print("Bus:", i+2, "delta:", self.delta[i]*180/np.pi, "voltage:", self.voltage_pu[i], "P_given:", P_given[i+1])

        # Calculate the complex voltage
        for i in range(self.length-1):
            self.V_complex[i] = self.voltage_pu[i] * np.cos(self.delta[i]) + 1j * self.voltage_pu[i] * np.sin(self.delta[i])

        self.V_complex = np.insert(self.V_complex, self.slack_bus, 1)
        #for i in range(self.length):
        #    print("V_complex for bus", i+1, ":", self.V_complex[i])

        # Solve for Powers, Currents, and other important values and print them out
        self.solve_power_flow(Grid)

    def solve_power_flow(self, Grid):

        # Calculate per unit current value
        I_values_per_unit = np.zeros((self.length, self.length), dtype=complex)

        # Going from the smaller bus to the larger bus
        for i in range(self.length):
            for j in range(self.length):
                I_values_per_unit[i, j] = -(self.V_complex[j] - self.V_complex[i]) * (self.Bbus_whole[i, j])

        # Calculate actual current values
```

```python
    # Calculate actual current values
    self.I_values = (self.Sbase / (self.Vbase * np.sqrt(3))) * I_values_per_unit
    self.I_values[0, 1] *= self.Vbase / self.Vbase_slack1  # -> Because from Bus1 to 2

    # Check ampacity limit
    self.check_ampacity()

    # Take voltage values out of per unit
    self.V_complex *= self.Vbase

    # Set up to find P
    self.P_inj_slack = 0
    for i in range(self.length):
        self.P_inj_slack += self.P_given[i]
    self.P_inj_slack = -self.P_inj_slack * 100
    print("self.P_inj_slack:", self.P_inj_slack, "MW")

    I_conjugate = np.conjugate(self.I_values)
    for i in range(self.length):
        for j in range(self.length):
            self.S_values[i, j] = self.V_complex[i] * I_conjugate[i, j] * np.sqrt(3)  # because 3 phase
            # Transformer has different base
            if i==0:
                self.S_values[i, j] = self.S_values[i, j]/self.V_complex[i] * self.Vbase_slack1

    # Print Values
    P = abs(self.S_values)
    P[0, 1] = self.P_inj_slack*1000000
    P[1, 0] = self.P_inj_slack*1000000

    print("\nI_Values")
    for i in range(self.length):
        for j in range(self.length):
            if abs(self.I_values[i, j]) == 0 or self.I_values[i, j] < 0:
                continue
            # Only print Line currents
            if i == 0 or j == 0 or i == 6 or j == 6:
                continue
            print("B" + str(i + 1) + " to B" + str(j + 1) + " ", abs(self.I_values[i, j]))


    # Print the P values from the Sending End, meaning they are positive
    print("\nP_values Sending End")
    for i in range(self.length):
        for j in range(self.length):
            if abs(P[i, j]) == 0 or np.imag(self.S_values[i,j]) > 0:
                continue
            print("Bus" + str(i + 1) + " to Bus" + str(j + 1) + " ", P[i, j] / 1000000, "MW")

    # Print the P values from the Receiving End, meaning they are negative
    print("\nP_values Receiving End")
    for i in range(self.length):
        for j in range(self.length):
            if abs(P[i, j]) == 0 or np.imag(self.S_values[i,j]) < 0:
                continue
            print("Bus" + str(i + 1) + " from Bus" + str(j + 1) + " ", P[i, j] / 1000000, "MW")

def check_ampacity(self):
    i = 1
    j = 1
    while i < 6:
        while j < 6:
            if abs(self.I_values[i, j]) == 0:
                j += 1
                continue
            if self.I_values[i][j] >= 475:
                print("Error: Ampacity limit exceeded")
                exit(-1)
            j += 1
        j = 0
        i += 1
```

<u>Fast_Decoupled_Solver Class</u>

This class solves the power flow equations for an AC power system using the fast decoupled method.

Relevant Equations:

$$\boldsymbol{J_1 \Delta\partial = \Delta P \qquad \rightarrow \qquad \Delta\partial = {J_1}^{-1} \Delta P}$$
$$\boldsymbol{J_4 \Delta V = \Delta Q \qquad \rightarrow \qquad \Delta V = {J_1}^{-1} \Delta Q}$$
$$\boldsymbol{\partial = B^{-1} P \qquad where\ B = B_{bus} = Im\{Y_{bus}\}}$$

For additional reference, the J indexes refer to the Jacobian matrix output for the Power system in question. Visually:

$$\begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix}$$

Class Code: located in repository as Fast_Decoupled_Solver.py

```
# Fast_Decoupled_Solver
import numpy as np
import pandas as pd

from Grid import Grid


class FastDecoupled:

    # Fast Decoupled
    def __init__(self, Grid, capacitor_bank):

        # Initial parameters that will be used later
        self.length = len(Grid.buses)
        self.system_loss = 0
        self.P_inj_slack = 0
        self.Q_inj_slack = 0
        self.Q_inj_PV = 0
        self.Q_limit_passed = 0
        self.convergencemet = 0
        iteration = 1

        # Set to 0, if becomes 1 means an adjustment was recently made with the capacitor
        # bank and the program should reiterate
        self.capacitor_bank_adjustment = 0

        # B is added capacitance if VAR limit is exceeded
        self.B = 0

        # 1 is to use capacitor bank correction on an exceeded Var limit, 0 is to get rid
        # of "PV" status of the generator's bus
        self.add_cap = capacitor_bank

        # Take the Q limit and convergence value from the Grid file
        self.Q_k_limit = Grid.Q_k_limit # In units of VA
        self.convergencevalue = Grid.convergencevalue


        # Base Unit Parameters
        self.Vbase = Grid.transformers[list(Grid.transformers.keys())[0]].v2rated * 1000  # In Volts
        self.Sbase = Grid.Sbase * 1000000  # VA
        self.Vbase_slack1 = Grid.transformers[list(Grid.transformers.keys())[0]].v1rated * 1000  # In Volts
        self.Vbase_slack2 = Grid.transformers[list(Grid.transformers.keys())[1]].v1rated * 1000  # In Volts
```

```
# Base Unit Parameters
self.Vbase = Grid.transformers[list(Grid.transformers.keys())[0]].v2rated * 1000  # In Volts
self.Sbase = Grid.Sbase * 1000000  # VA
self.Vbase_slack1 = Grid.transformers[list(Grid.transformers.keys())[0]].v1rated * 1000  # In Volts
self.Vbase_slack2 = Grid.transformers[list(Grid.transformers.keys())[1]].v1rated * 1000  # In Volts

# Initialize Empty Arrays/ Matrices
self.S_values = np.zeros((self.length, self.length), dtype=complex)
self.Q_values = np.zeros((self.length, self.length))
self.P_values = np.zeros((self.length, self.length))
self.I_values = np.zeros((self.length, self.length), dtype=complex)
self.V_complex = np.zeros(self.length, dtype=complex)
self.Parr = []
self.Qarr = []
self.P_loss = np.zeros((self.length, self.length))
P_given = np.zeros(self.length)
Q_given = np.zeros(self.length)
P_mismatch = np.zeros(self.length-1)
Q_mismatch = np.zeros(self.length-2)

# Array for calculated P, Q, and Jacobian values
Parr = np.zeros(self.length)
Qarr = np.zeros(self.length)
J11 = np.zeros((self.length - 1, self.length - 1))
J22 = np.zeros((self.length - 1, self.length - 1))

# Find slack bus and voltage controlled bus by looping through Buses Dictionary in Grid Class
self.slack_bus = None
for i in range(self.length):
    if Grid.buses["Bus" + str(i + 1)].type == "Slack Bus":
        self.slack_bus = i
    if Grid.buses["Bus" + str(i + 1)].type == "Voltage Controlled Bus":
        self.voltage_controlled_bus = i

# Set given values by looping through Buses Dictionary in Grid Class, Loads are negative
for i in range(self.length):
    if Grid.buses["Bus" + str(i + 1)].type == "Load Bus":
        P_given[i] = -Grid.buses["Bus" + str(i + 1)].P / 100
        Q_given[i] = -Grid.buses["Bus" + str(i + 1)].Q / 100
    else:
        P_given[i] = Grid.buses["Bus" + str(i + 1)].P / 100
        Q_given[i] = Grid.buses["Bus" + str(i + 1)].Q / 100

# Add a statement to turn Q_given into self. so that it can be used in other functions
self.Q_given = Q_given

# Set flat start
V = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
delta = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# Make sure convergence has not been met and that a capacitor bank was not just added, or that the max
# Iterations has not been exceeded
while (self.convergencemet == 0 and self.capacitor_bank_adjustment == 1) or iteration < 30:
    J11 = np.zeros((self.length - 1, self.length - 1))
    J22 = np.zeros((self.length - 1, self.length - 1))
    # Reset the calculated values every time through
    for i in range(self.length):
        Parr[i] = 0
        Qarr[i] = 0

    # calculate mismatch, ignoring the slack bus
    for i in range(self.length):
        # if slack bus, skip
        if i == self.slack_bus:
            continue

        for j in range(self.length):
            Parr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            if i == self.voltage_controlled_bus:
                continue
            Qarr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

    # P does not include slack bus, Q does not include Voltage controlled bus or slack bus
    if self.slack_bus == 0:
        P_mismatch = P_given[1:] - Parr[1:]
        Q_mismatch = Q_given[1:6] - Qarr[1:6]
    if self.slack_bus == 6:
        P_mismatch = P_given[0:6] - Parr[0:6]
        Q_mismatch = Q_given[1:6] - Qarr[1:6]

    # Set the convergence met variable to 1
    self.convergencemet = 1

    # Check Power Mismatch for Convergence
    mismatchPQ = np.concatenate((P_mismatch, Q_mismatch))
    for i in range(len(mismatchPQ)):
        # If any value in the mismatched is greater than the convergence requirement, set convergence parameter
        # to not met
        if mismatchPQ[i] > self.convergencevalue:
            self.convergencemet = 0
            break
```

```
# If convergence has been met and there was no capacitor adjustment, notify user it converged
if self.convergencemet == 1 and self.capacitor_bank_adjustment == 0:
    break

# Calculate Jacobian Matrix
skipterm = 0
for i in range(self.length):
    # if slack bus skip
    if i == self.slack_bus:
        skipterm = 1
        continue
    for j in range(self.length):
        # if slack bus skip
        if j == self.slack_bus:
            continue

        # Diagonals have their own formula
        if i == j:
            for z in range(self.length):
                J22[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

                if z == i:
                    continue

                J11[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

            # Non-Summation Modification
            J11[i - skipterm, j - skipterm] = -J11[i - skipterm, j - skipterm] * V[i]
            J22[i - skipterm, j - skipterm] = J22[i - skipterm, j - skipterm] - (
                        V[i] * abs(Grid.Ybus[i, j]) * np.sin(np.angle(Grid.Ybus[i, j])))

        # If not a diagonal of the jacobian
        else:
            J11[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * V[j] * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J22[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * np.sin(delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

# Delete Voltage Controlled bus 1 or 7, here if it is 7
if self.slack_bus == 0:
    J_temp22 = np.delete(J22, 5, 1)
    J22 = J_temp22
    J_temp22 = np.delete(J22, 5, 0)
    J22 = J_temp22

            # Here VCB is 1
            if self.slack_bus == 6:
                J_temp22 = np.delete(J22, 0, 1)
                J22 = J_temp22
                J_temp22 = np.delete(J22, 0, 0)
                J22 = J_temp22

            # calculate change in voltage and phase angle
            delta_correction = np.dot(np.linalg.inv(J11), P_mismatch)
            V_correction = np.dot(np.linalg.inv(J22), Q_mismatch)

            # do not update the angle of the slack bus
            delta_correction = np.concatenate((delta_correction[:self.slack_bus], [0], delta_correction[self.slack_bus:]), axis=0)

            # do not update the voltage of the slack bus or voltage controlled bus
            V_correction = np.concatenate((V_correction[:6], [0], V_correction[6:]), axis=0)
            V_correction = np.concatenate((V_correction[:0], [0], V_correction[0:]), axis=0)

            # Update the voltage and angle and increase the iteration
            delta += delta_correction
            V += V_correction
            iteration += 1

            # Parameter to store Q from Voltage controlled bus
            self.Q_k = 0

            # Calculate Q_k
            for i in range(self.length):
                self.Q_k += abs(Grid.Ybus[self.voltage_controlled_bus, i]) * V[i] * np.sin(delta[self.voltage_controlled_bus] - delta[i] -
np.angle(Grid.Ybus[self.voltage_controlled_bus,i]))
            self.Q_k *= V[self.voltage_controlled_bus]
            self.Q_k *= self.Sbase

            # If Q from VCB has exceeded the limit, and a capacitor bank is not wanted
            if self.Q_k > self.Q_k_limit and self.add_cap == 0:
                print("VAR LIMIT EXCEEDED: Generator bus will no longer be a Voltage Controlled Bus")
                self.solve_exceeded_var_power_flow(Grid)
                self.Q_limit_passed = 1
                break

            # Reset the term to 0 to see if another adjustment needs made
            self.capacitor_bank_adjustment = 0
```

```
        # If Q from VCB has exceeded the limit, and a capacitor bank is wanted
        if self.Q_k > self.Q_k_limit and self.add_cap == 1:
            # Adjustment Needed
            print("LIMIT EXCEEDED, INCREASING CAPACITOR BANK")
            self.capacitor_bank_adjustment = 1
            # Add capacitor bank to highest MVAR load
            j = 0
            for i in range(self.length):
                if self.Q_given[j] >= self.Q_given[i]:
                    j = i
            self.B += -1j * self.Q_given[j] * 100000000 / (self.Vbase ** 2) #* self.Sbase
            Grid.Ybus[j, j] += self.B

            # Reset iteration so that it does not exceed the iteration limit while adding banks
            iteration = 0

    # Check to make sure V_complex was not set up by Var limit solver
    if self.Q_limit_passed == 0:
        # Calculate the complex voltage
        for i in range(self.length):
            self.V_complex[i] = V[i] * np.cos(delta[i]) + 1j * V[i] * np.sin(delta[i])

    # Function to solve power flow
    self.solve_power_flow(Grid)

def solve_power_flow(self, Grid):

    # Calculate per unit current value
    I_values_per_unit = np.zeros((self.length, self.length), dtype=complex)
    for i in range(self.length):
        for j in range(self.length):
            I_values_per_unit[i, j] = (self.V_complex[j] - self.V_complex[i]) * (Grid.Ybus[i, j])

    # Calculate actual current values
    self.I_values = (self.Sbase / (self.Vbase * np.sqrt(3))) * I_values_per_unit
    self.I_values[0, 1] *= self.Vbase / self.Vbase_slack1 # -> Because from Bus 1 to 2, transformer
    self.I_values[6, 5] *= self.Vbase / self.Vbase_slack2 # -> Becasue from Bus 7 to 6, transformer

    # Check ampacity limit on lines
    self.check_ampacity()

    # Take voltage values out of per unit
    self.V_complex *= self.Vbase

    # Calcualte conjugate of current values
    I_conjugate = np.conjugate(self.I_values)
```

```
        # Set up to find P and Q
        # Find P, S = P + jQ, and S = I_conjugate * V * sqrt(3)
        # Find S first
        for i in range(self.length):
            for j in range(self.length):
                self.S_values[i, j] = self.V_complex[i] * I_conjugate[i, j] * np.sqrt(3)  # because 3 phase
                # Transformer has different base
                if i==0:
                    self.S_values[i, j] = self.S_values[i, j]/self.V_complex[i] * self.Vbase_slack1
                # Transformer has different base
                if i==6:
                    self.S_values[i, j] = self.S_values[i, j]/self.V_complex[i] * self.Vbase_slack2

        # P is the real part of S, Q is the imaginary part of S
        self.P_values = np.real(self.S_values)
        self.Q_values = np.imag(self.S_values)

        # Loop over every item in the transmission line dictionary, i is the name and value is the object
        for i1 in range(self.length):
            for i2 in range(self.length):
                for i, value in Grid.transmissionline.items():
                    if value.bus1 == ("Bus" + str(i1+1)) and value.bus2 == ("Bus" + str(i2+1)):
                        powerloss = abs(self.I_values[i1, i2]) ** 2 * value.Rtotal * 3 # because 3 phase
                        # Store the power loss, so it can be accessed through the grid class
                        Grid.store_power_loss(i, powerloss)
                        # Update the system loss
                        self.system_loss += powerloss

        # Same for transformers
        for i1 in range(self.length):
            for i2 in range(self.length):
                for i, value in Grid.transformers.items():
                    if value.bus1 == ("Bus" + str(i1+1)) and value.bus2 == ("Bus" + str(i2+1)) or value.bus1 == ("Bus" + str(i2+1)) and value.bus2 == ("Bus" +
str(i1+1)):
                        if abs(self.I_values[i1, i2]) == 0:
                            continue
                        powerloss = abs(self.I_values[i1, i2]) ** 2 * value.Rpu * self.Vbase ** 2/self.Sbase * 3 # because 3 phase
                        # Transformers have a different base
                        if value.name == "T1":
                            powerloss = powerloss / (self.Vbase ** 2) * self.Vbase_slack1 ** 2
                        if value.name == "T2":
                            powerloss = powerloss / (self.Vbase ** 2) * self.Vbase_slack2 ** 2
                        # Store the power loss in the transformer in the Grid class also
                        Grid.store_power_loss_transformer(i, powerloss)
                        # Update system power loss
                        self.system_loss += powerloss
```

```
        # POWER AND Q INJECTIONS FOR SLACK + PV
        for i in range(self.length):
            # Iterate through each Bus to sum up these values
            self.Q_inj_slack += abs(Grid.Ybus[self.slack_bus, i]) * abs(self.V_complex[i])/self.Vbase * np.sin(np.angle(self.V_complex[self.slack_bus]) -
np.angle(self.V_complex[i]) - np.angle(Grid.Ybus[self.slack_bus, i]))
            self.P_inj_slack += abs(Grid.Ybus[self.slack_bus, i]) * abs(self.V_complex[i])/self.Vbase * np.cos(np.angle(self.V_complex[self.slack_bus]) -
np.angle(self.V_complex[i]) - np.angle(Grid.Ybus[self.slack_bus, i]))
            self.Q_inj_PV += abs(Grid.Ybus[self.voltage_controlled_bus, i]) * abs(self.V_complex[i])/self.Vbase *
np.sin(np.angle(self.V_complex[self.voltage_controlled_bus]) - np.angle(self.V_complex[i]) - np.angle(Grid.Ybus[self.voltage_controlled_bus, i]))

        # Convert to the values to their actual values instead of per unit
        self.P_inj_slack *= abs(self.V_complex[self.slack_bus])/self.Vbase * self.Sbase
        self.Q_inj_slack *= abs(self.V_complex[self.slack_bus])/self.Vbase * self.Sbase
        self.Q_inj_PV *= abs(self.V_complex[self.voltage_controlled_bus])/self.Vbase * self.Sbase

        # Print these values in "Mega" units
        print("self.P_inj_slack:", self.P_inj_slack/1000000, "MW")
        print("self.Q_inj_slack:", self.Q_inj_slack/1000000, "MVAR")
        print("self.Q_inj_PV:", self.Q_inj_PV/1000000, "MVAR")

        # Print I Values
        print("\nI_Values")
        for i in range(self.length):
            for j in range(self.length):
                # If the value is 0, do not print, if the value is negative, current is flowing the other way
                # So it will print out at a later iteration in the loop
                if abs(self.I_values[i, j]) == 0 or self.I_values[i, j] < 0:
                    continue
                # Only print Line currents, although transformer currents also matched with powerworld
                if i == 0 or j == 0 or i == 6 or j == 6:
                    continue
                print("B" + str(i + 1) + " to B" + str(j + 1) + " ", abs(self.I_values[i, j]))

        # Print the P values from the Sending End, meaning they are positive
        print("\nP_values Sending End")
        for i in range(self.length):
            for j in range(self.length):
                if abs(self.P_values[i, j]) == 0 or self.P_values[i, j] < 0:
                    continue
                print("Bus" + str(i + 1) + " to Bus" + str(j + 1) + " ", self.P_values[i, j] / 1000000, "MW")

        # Print the P values from the Receiving End, meaning they are negative
        print("\nP_values Receiving End")
        for i in range(self.length):
            for j in range(self.length):
                if abs(self.P_values[i, j]) == 0 or self.P_values[i, j] > 0:
                    continue
                print("Bus" + str(i + 1) + " from Bus" + str(j + 1) + " ", self.P_values[i, j] / 1000000, "MW")
```

```
    # Print the Q values from the Sending End, meaning they are positive
    print("\nQ_values Sending End")
    for i in range(self.length):
        for j in range(self.length):
            if abs(self.Q_values[i, j]) == 0 or self.Q_values[i, j] < 0:
                continue
            print("Bus" + str(i + 1) + " to Bus" + str(j + 1) + " ", self.Q_values[i, j] / 1000000, "MVAR")

    # Print the Q values from the Sending End, meaning they are negative
    print("\nQ_values Receiving End")
    for i in range(self.length):
        for j in range(self.length):
            if abs(self.Q_values[i, j]) == 0 or self.Q_values[i, j] > 0:
                continue
            print("Bus" + str(i + 1) + " from Bus" + str(j + 1) + " ", self.Q_values[i, j] / 1000000, "MVAR")

    # Print Power Loss Per Line
    print("\nPower Loss Per Line")
    for i, value in Grid.transmissionline.items():
        print(i, value.powerloss/1000000, "MW")

    # Print Power Loss in the Transformers
    print("\nPower Loss in Transformers")
    for i, value in Grid.transformers.items():
        print(i, value.powerloss/1000000, "MW")

    # Print the total system power loss
    print("\nTotal System Power Loss:", self.system_loss/1000000, "MW")

# If the QVAR limit was exceeded previously, this function will be used if no capacitor bank is added.
# It has the same functionality as before, however there is no longer a Voltage controlled bus
def solve_exceeded_var_power_flow(self, Grid):
    # set blank array for given values
    P_given = np.zeros(self.length)
    Q_given = np.zeros(self.length)
    P_mismatch = np.zeros(self.length-1)
    Q_mismatch = np.zeros(self.length-1)
    # find slack bus
    for i in range(self.length):
        if Grid.buses["Bus" + str(i + 1)].type == "Slack Bus":
            self.slack_bus = i
        if Grid.buses["Bus" + str(i + 1)].type == "Voltage Controlled Bus":
            self.voltage_controlled_bus = i
```

```
# set given values
for i in range(self.length):
    if Grid.buses["Bus" + str(i + 1)].type == "Load Bus":
        P_given[i] = -Grid.buses["Bus" + str(i + 1)].P / 100
        Q_given[i] = -Grid.buses["Bus" + str(i + 1)].Q / 100
    else:
        P_given[i] = Grid.buses["Bus" + str(i + 1)].P / 100
        Q_given[i] = Grid.buses["Bus" + str(i + 1)].Q / 100
Q_given[self.voltage_controlled_bus] = self.Q_k_limit / self.Sbase

# set intial guess
V = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
delta = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# establish array for calculated value, without slack bus
Parr = np.zeros(self.length)
Qarr = np.zeros(self.length)
iteration = 1

while self.convergencemet == 0 and iteration < 30:
    for i in range(self.length):
        Parr[i] = 0
        Qarr[i] = 0
    # calculate mismatch, ignoring the slack bus
    for i in range(self.length):
        # if slack bus, skip
        if i == self.slack_bus:
            continue

        for j in range(self.length):
            Parr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.cos(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            Qarr[i] += V[i] * V[j] * abs(Grid.Ybus[i, j]) * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))

    # P does not include slack bus
    if self.slack_bus == 0:
        P_mismatch = P_given[1:7] - Parr[1:7]
        Q_mismatch = Q_given[1:7] - Qarr[1:7]
    if self.slack_bus == 6:
        P_mismatch = P_given[0:6] - Parr[0:6]
        Q_mismatch = Q_given[0:6] - Qarr[0:6]

    self.convergencemet = 1

# Check Power Mismatch for Convergence
mismatchPQ = np.concatenate((P_mismatch, Q_mismatch))
for i in range(len(mismatchPQ)):
    if mismatchPQ[i] > self.convergencevalue:
        self.convergencemet = 0
        break

if self.convergencemet == 1:
    break

# Calculate Jacobian Matrix
J11 = np.zeros((self.length - 1, self.length - 1))
J22 = np.zeros((self.length - 1, self.length - 1))

skipterm = 0

for i in range(self.length):
    # if slack bus skip
    if i == self.slack_bus:
        skipterm = 1
        continue
    for j in range(self.length):
        if j == self.slack_bus:
            continue
        if i == j:
            for z in range(self.length):
                J22[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))
                if z == i:
                    continue
                J11[i - skipterm, j - skipterm] += abs(Grid.Ybus[i, z]) * V[z] * np.sin(
                    delta[i] - delta[z] - np.angle(Grid.Ybus[i, z]))

            # Non-Summation Modification
            J11[i - skipterm, j - skipterm] = -J11[i - skipterm, j - skipterm] * V[i]
            J22[i - skipterm, j - skipterm] = J22[i - skipterm, j - skipterm] - (
                    V[i] * abs(Grid.Ybus[i, j]) * np.sin(np.angle(Grid.Ybus[i, j])))

        else:
            J11[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * V[j] * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
            J22[i - skipterm, j - skipterm] = V[i] * abs(Grid.Ybus[i, j]) * np.sin(
                delta[i] - delta[j] - np.angle(Grid.Ybus[i, j]))
```

```
        # Combine Jacobian
        J_inv11 = np.linalg.inv(J11)
        J_inv22 = np.linalg.inv(J22)

        # calculate change in voltage and phase angle
        delta_correction = np.dot(J_inv11, P_mismatch)
        V_correction = np.dot(J_inv22, Q_mismatch)

        # do not update the angle of the slack bus
        delta_correction = np.concatenate((delta_correction[:self.slack_bus], [0], delta_correction[self.slack_bus:]), axis=0)

        # do not update the voltage of the slack bus
        V_correction = np.concatenate((V_correction[:self.slack_bus], [0], V_correction[self.slack_bus:]), axis=0)

        # Correction
        delta += delta_correction
        V += V_correction
        iteration += 1

    # Setup V_complex
    for i in range(self.length):
        self.V_complex[i] = V[i] * np.cos(delta[i]) + 1j * V[i] * np.sin(delta[i])

def check_ampacity(self):
    i = 1
    j = 1
    while i < 6:
        while j < 6:
            if abs(self.I_values[i, j]) == 0:
                j += 1
                continue
            if self.I_values[i][j] >= 475:
                print("Error: Ampacity limit exceeded")
                exit(-1)
            j += 1
        j = 0
        i += 1
```

Sequence_Networks Class

This class calculates the positive, negative, and zero sequence impedances in the power system. The positive sequence and negative bus admittance matrices are practically the same as the Ybus calculated in the grid class, but each add their own generator impedance values. The zero sequence matrices depend on both the generator and transformer groundings, and our code allows for user input of any grounding type and still calculates the zero sequence networks.

Variable reference in code	Description
self.Zbus0	Zero sequence bus impedance matrix
self.Zbus1	Positive sequence bus impedance matrix
self.Zbus2	Negative sequence bus impedance matrix
self.Ybus0	Zero sequence bus admittance matrix
self.Ybus1	Positive sequence bus admittance matrix
self.Ybus2	Negative sequence bus admittance matrix
self.x1generators_newpu1	The per unit impedance value for generator one in the positive sequence network
self.x1generators_newpu2	The per unit impedance value for generator two in the positive sequence network
self.x2generators_newpu1	The per unit impedance value for generator one in the negative sequence network
self.x2generators_newpu2	The per unit impedance value for generator two in the negative sequence network
self.x0generators_newpu1	The per unit impedance value for generator one in the zero sequence network
self.x0generators_newpu2	The per unit impedance value for generator two in the zero sequence network

Input Argument Name	Description	Value in example
self.Zg1	Generator 1 grounding value	0
self.Zg2	Generator 2 grounding value	1 Ω
self.Zt1_value1	Transformer 1 side 1 grounding value	None
self.Zt1_value2	Transformer 1 side 2 grounding value	1 Ω
self.Zt2_value1	Transformer 2 side 1 grounding value	None
self.Zt2_value2	Transformer 2 side 2 grounding value	inf

Relevant Equations:

Positive Sequence

$$Ybus1 = Grid.Ybus$$

$$Ybus1[0][0] = \frac{1}{Grid.transformers[\text{T1}].Rpu + x1generators\_newpu1 + j \cdot Grid.transformers[\text{T1}].Xpu}$$

$$Ybus1[6][6] = \frac{1}{Grid.transformers[\text{T2}].Rpu + x1generators\_newpu2 + j \cdot Grid.transformers[\text{T2}].Xpu}$$

$$Zbus1 = Ybus1^{-1}$$

Negative Sequence

$$Ybus2 = Grid.Ybus$$

$$Ybus2[0][0] = \frac{1}{Grid.transformers[\text{T1}].Rpu + x2generators\_newpu2 + j \cdot Grid.transformers[\text{T1}].Xpu}$$

$$Ybus2[6][6] = \frac{1}{Grid.transformers[\text{T2}].Rpu + x2generators\_newpu2 + j \cdot Grid.transformers[\text{T2}].Xpu}$$

$$Zbus2 = Ybus2^{-1}$$

Zero Sequence Impedances

Lines: $z_0 = 3z_1$

Generators: $z_0 = 3z_g + x_0$

$$z_g = 0 \; solid\ ground$$

$$z_g = \infty \; ungrounded$$

Transformers:

Grounded Wye/Grounded Wye

$z_{T0} = 3(Zt1\_value1 + Zt1\_value2) + Grid.transformers[T1].Rpu + j \cdot Grid.transformers[T1].Xpu$

Delta / Grounded Wye

$$z_{T0} = 3Zt1\_value2 + Grid.transformers[T1].Rpu + j \cdot Grid.transformers[T1].Xpu$$

Ungrounded Wye / Grounded Wye

$$z_{T0} = 0$$

Sum additional impedances connected to each bus, and invert for zero admittance matrices or vice versa.

Class code: located in repository as Sequence_Networks.py

```
# File to develop the positive, negative, and zero sequence bus impedance matrices
import numpy as np
class SequenceNet:

    # Power flow
    def __init__(self, Grid):
        # Values for generator
        self.x1generators_oldpu1 = Grid.generators["G1"].x1gen
        self.x2generators_oldpu1 = Grid.generators["G1"].x2gen
        self.x0generators_oldpu1 = Grid.generators["G1"].x0gen
        self.x1generators_oldpu2 = Grid.generators["G2"].x1gen
        self.x2generators_oldpu2 = Grid.generators["G2"].x2gen
        self.x0generators_oldpu2 = Grid.generators["G2"].x0gen
        Zg1_grounding = Grid.generators["G1"].grounding_type
        Zg1_value = Grid.generators["G1"].grounding_value
        Zg2_grounding = Grid.generators["G2"].grounding_type
        Zg2_value = Grid.generators["G2"].grounding_value

        # Transformer values
        Zt1_connection1 = Grid.transformers["T1"].Zt_connection1
        Zt1_grounding1 = Grid.transformers["T1"].Zt_grounding1
        Zt1_value1 = Grid.transformers["T1"].Zt_value1
        Zt1_connection2 = Grid.transformers["T1"].Zt_connection2
        Zt1_grounding2 = Grid.transformers["T1"].Zt_grounding2
        Zt1_value2 = Grid.transformers["T1"].Zt_value2
        Zt2_connection1 = Grid.transformers["T2"].Zt_connection1
        Zt2_grounding1 = Grid.transformers["T2"].Zt_grounding1
        Zt2_value1 = Grid.transformers["T2"].Zt_value1
        Zt2_connection2 = Grid.transformers["T2"].Zt_connection2
        Zt2_grounding2 = Grid.transformers["T2"].Zt_grounding2
        Zt2_value2 = Grid.transformers["T2"].Zt_value2

        # Establish Which Sequence Networks you want to create
        Generate_Zbus_0 = 1
        Generate_Zbus_1 = 1
        Generate_Zbus_2 = 1

        # Establish Base Values
        self.Sbase = Grid.Sbase * 1000000 # In VA
        self.Vbase_main = Grid.Vbase * 1000
        self.Vbase_bus1 = Grid.transformers[list(Grid.transformers.keys())[0]].v1rated * 1000
        self.Vbase_bus7 = Grid.transformers[list(Grid.transformers.keys())[1]].v1rated * 1000
        nominalpower1 = Grid.generators[list(Grid.generators.keys())[0]].nominalpower * 1000000
        nominalpower2 = Grid.generators[list(Grid.generators.keys())[1]].nominalpower * 1000000
        self.Zbasemain = self.Vbase_main ** 2 / self.Sbase
        self.Zbase1 = self.Vbase_bus1 ** 2 / nominalpower1
        self.Zbase7 = self.Vbase_bus7 ** 2 / nominalpower2
        # Establish Other Parameters -> [Row][Column]
        self.length = len(Grid.buses)
        self.Zbus0 = np.zeros((self.length, self.length), dtype=complex)
        self.Zbus1 = np.zeros((self.length, self.length), dtype=complex)
        self.Zbus2 = np.zeros((self.length, self.length), dtype=complex)
        self.Ybus0 = np.zeros((self.length, self.length), dtype=complex)
        self.Ybus1 = np.zeros((self.length, self.length), dtype=complex)
        self.Ybus2 = np.zeros((self.length, self.length), dtype=complex)

        # Transformer Grounding ->transformer1 through 1 Ohm on Y side, transformer2 Y ungrounded

        # Generate Zbus1 -> Positive Sequence
        if Generate_Zbus_1 == 1:

            # Switch Generators to System Per Unit Values instead of individual
            self.x1generators_newpu1 = self.x1generators_oldpu1 * self.Sbase / nominalpower1
            self.x1generators_newpu2 = self.x1generators_oldpu2 * self.Sbase / nominalpower2

            # Ybu1 is slightly modified Ybus from before
            self.Ybus1 = Grid.Ybus

            # Update Bus 1 with generator information
            self.Ybus1[0][0] = 1 / (Grid.transformers["T1"].Rpu + 1j * self.x1generators_newpu1 + 1j * Grid.transformers["T1"].Xpu)

            # Update Bus 7 with generator information
            self.Ybus1[6][6] = 1 / (Grid.transformers["T2"].Rpu + 1j * self.x1generators_newpu2 + 1j * Grid.transformers["T2"].Xpu)

            # Zbus1 is inverse of Ybus1
            self.Zbus1 = np.linalg.inv(self.Ybus1)
            #print("Ybus1")
            #self.printmatrix(self.Ybus1)
            #print("Zbus1")
            #self.printmatrix(self.Zbus1)

        # Generate Zbus2 -> Negative Sequence
        if Generate_Zbus_2 == 1:
            # Switch Generators to System Per Unit Values instead of individual
            self.x2generators_newpu1 = self.x2generators_oldpu1 * self.Sbase / nominalpower1
            self.x2generators_newpu2 = self.x2generators_oldpu2 * self.Sbase / nominalpower2

            # Ybu2 is slightly modified Ybus from before
            self.Ybus2 = Grid.Ybus

            # Update Bus 1 with generator information
            self.Ybus2[0][0] = 1 / (Grid.transformers["T1"].Rpu + 1j * self.x2generators_newpu1 + 1j * Grid.transformers["T1"].Xpu)
```

```python
    # Update Bus 7 with generator information
    self.Ybus2[6][6] = 1 / (Grid.transformers["T2"].Rpu + 1j * self.x2generators_newpu2 + 1j * Grid.transformers["T2"].Xpu)

    # Zbus2 is inverse of Ybus2
    self.Zbus2 = np.linalg.inv(self.Ybus2)
    #print("Ybus2")
    #self.printmatrix(self.Ybus2)
    #print("Zbus2")
    #self.printmatrix(self.Zbus2)

# Generate Zbus0 -> 0 Sequence, Need Help
if Generate_Zbus_0 == 1:

    # Generator 1 Grounding Information
    if Zg1_grounding == "Solid ground":
        self.Zg1 = 0
    elif Zg1_grounding == "Ungrounded":
        self.Zg1 = float('inf')
    elif Zg1_grounding == "Resistor":
        self.Zg1 = Zg1_value
    else:
        print("Invalid Grounding Information For Generator 1")
        exit(-1)

    # Generator 2 Grounding Information
    if Zg2_grounding == "Solid ground":
        self.Zg2 = 0
    elif Zg2_grounding == "Ungrounded":
        self.Zg2 = float('inf')
    elif Zg2_grounding == "Resistor":
        self.Zg2 = Zg2_value
    else:
        print("Invalid Grounding Information For Generator 2")
        exit(-1)

    # Establish Generators, takes into account 3Zg as well -> Include in Ybus0 calculation
    self.totalx0generators_newpu1 = (1j * self.x0generators_oldpu1 + 3 * self.Zg1 / self.Zbase1) * (self.Sbase / nominalpower1)
    self.totalx0generators_newpu2 = (1j * self.x0generators_oldpu2 + 3 * self.Zg2 / self.Zbase7) * (self.Sbase / nominalpower2)

# Transformer 1 Grounding Information, Side 1
if Zt1_grounding1 == "Solid ground":
    self.Zt1_value1 = 0
elif Zt1_grounding1 == "Ungrounded":
    self.Zt1_value1 = float('inf')
elif Zt1_grounding1 == "Resistor":
    self.Zt1_value1 = Zt1_value1
elif Zt1_grounding1 == "N/A":
    self.Zt1_value1 = None
else:
    print("Invalid Grounding Information For Transformer 1 Side 1")
    exit(-1)

# Transformer 1 Grounding Information, Side 2
if Zt1_grounding2 == "Solid ground":
    self.Zt1_value2 = 0
elif Zt1_grounding2 == "Ungrounded":
    self.Zt1_value2 = float('inf')
elif Zt1_grounding2 == "Resistor":
    self.Zt1_value2 = Zt1_value2
elif Zt1_grounding2 == "N/A":
    self.Zt1_value2 = None
else:
    print("Invalid Grounding Information For Transformer 1 Side 2")
    exit(-1)

# Transformer 2 Grounding Information, Side 1
if Zt2_grounding1 == "Solid ground":
    self.Zt2_value1 = 0
elif Zt2_grounding1 == "Ungrounded":
    self.Zt2_value1 = float('inf')
elif Zt2_grounding1 == "Resistor":
    self.Zt2_value1 = Zt2_value1
elif Zt2_grounding1 == "N/A":
    self.Zt2_value1 = None
else:
    print("Invalid Grounding Information For Transformer 2 Side 1")
    exit(-1)

# Transformer 2 Grounding Information, Side 2
if Zt2_grounding2 == "Solid ground":
    self.Zt2_value2 = 0
elif Zt2_grounding2 == "Ungrounded":
    self.Zt2_value2 = float('inf')
elif Zt2_grounding2 == "Resistor":
    self.Zt2_value2 = Zt2_value2
elif Zt2_grounding2 == "N/A":
    self.Zt2_value2 = None
else:
    print("Invalid Grounding Information For Transformer 2 Side 2")
    exit(-1)

# This is Transformer Information -> Depends on grounding, initial remains here for reference only while I develop the proper code
#self.Ybus0[6][5] = -1 / (Grid.transformers["T2"].Rpu + 1j * Grid.transformers["T2"].Xpu)  # T2
#self.Ybus0[5][6] = self.Ybus0[6][5]  # T2
#self.Ybus0[0][1] = -1 / (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu)  # T1
#self.Ybus0[1][0] = self.Ybus0[0][1]  # T1

# Updating Ybus Depending on Transformer 1 Connection
self.Ybus0 = Grid.Ybus

# Grounded Wye can go to Grounded Wye or Delta to not be 0
# Grounded Wye to Grounded Wye on Sequence Networks-> Set Ohms for each side using each side's base
if Zt1_connection1 == "Grounded Wye":
    if Zt1_connection2 == "Grounded Wye":
        self.Zt11 = Zt1_value1 * 3 / self.Zbase1
        self.Zt12 = Zt1_value2 * 3 / self.Zbasemain
        self.Ybus0[0][1] = -1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt11 + self.Zt12)  # T1
        self.Ybus0[1][0] = self.Ybus0[0][1]  # T1
        self.Ybus0[0][0] = 1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt11 + self.Zt12 + 1j * self.totalx0generators_newpu1)

    elif Zt1_connection2 == "Delta":
        self.Zt11 = Zt1_value1 * 3 / self.Zbase1
        self.Ybus0[0][1] = -1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt11)  # T1 Number
        self.Ybus0[1][0] = 0  # T1
        self.Ybus0[0][0] = 1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt11 + 1j * self.totalx0generators_newpu1)

    elif Zt1_connection2 == "Ungrounded Wye":
        self.Ybus0[0][1] = 0  # T1
        self.Ybus0[1][0] = 0  # T1
        self.Ybus0[0][0] = 1 / (1j * self.totalx0generators_newpu1)

    else:
        print("Invalid Information For Transformer 1 Connection 2")
        exit(-1)
```

```
# Delta to Grounded Wye Only Non-Zero Combination
elif Zt1_connection1 == "Delta":
    if Zt1_connection2 == "Grounded Wye":
        #print("Transformer 1: Delta to Grounded Wye")
        self.Zt12 = Zt1_value2 * 3 / self.Zbasemain
        self.Ybus0[0][1] = 0  # T1
        self.Ybus0[1][0] = -1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt12)  # T1  # T1
        self.Ybus0[0][0] = 1 / (1j * self.totalx0generators_newpu1)
        #print("self.totalx0generators_newpu1", self.totalx0generators_newpu1)
    elif Zt1_connection2 == "Delta" or Zt1_connection2 == "Ungrounded Wye":
        self.Ybus0[0][1] = 0  # T1
        self.Ybus0[1][0] = 0  # T1
        self.Ybus0[0][0] = 1 / (1j * self.totalx0generators_newpu1)

    else:
        print("Invalid Information For Transformer 1 Connection 2")
        exit(-1)
# All Ungrounded Wye have 0
elif Zt1_connection1 == "Ungrounded Wye":
    self.Ybus0[0][1] = 0  # T1
    self.Ybus0[1][0] = 0  # T1
    self.Ybus0[0][0] = 1 / (1j * self.totalx0generators_newpu1)

else:
    print("Invalid Information For Transformer 1 Connection 1")
    exit(-1)

# Updating Ybus Depending on Transformer 1 Connection
# Grounded Wye can go to Grounded Wye or Delta to not be 0
if Zt2_connection1 == "Grounded Wye":
    if Zt2_connection2 == "Grounded Wye":
        self.Zt21 = Zt2_value1 * 3 / self.Zbase7
        self.Zt22 = Zt2_value2 * 3 / self.Zbasemain
        self.Ybus0[6][5] = -1 / (3 * (Grid.transformers["T2"].Rpu + 1j * Grid.transformers["T2"].Xpu) + self.Zt21 + self.Zt22)  # T2
        self.Ybus0[5][6] = self.Ybus0[6][5]
        self.Ybus0[6][6] = 1 / (3 * (Grid.transformers["T2"].Rpu + 1j * Grid.transformers["T2"].Xpu) + self.Zt21 + self.Zt22 + 1j * self.totalx0generators_newpu2)

    elif Zt2_connection2 == "Delta":
        self.Zt21 = Zt2_value1 * 3 / self.Zbase7
        self.Ybus0[6][5] = -1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt21)  # T2
        self.Ybus0[5][6] = 0  # T2
        self.Ybus0[6][6] = 1 / (3 * (Grid.transformers["T1"].Rpu + 1j * Grid.transformers["T1"].Xpu) + self.Zt21 + 1j * self.totalx0generators_newpu2)

    elif Zt2_connection2 == "Ungrounded Wye":
        self.Ybus0[6][5] = 0  # T2
        self.Ybus0[5][6] = 0  # T2
        self.Ybus0[6][6] = 1 / (1j * self.totalx0generators_newpu2)

    else:
        print("Invalid Information For Transformer 2 Connection 2")
        exit(-1)

# Delta to Grounded Wye Only Non-Zero Combination
elif Zt2_connection1 == "Delta":
    if Zt2_connection2 == "Grounded Wye":
        #print("Transformer 2: Delta to Grounded Wye")
        self.Zt22 = Zt2_value2 * 3 / self.Zbasemain
        self.Ybus0[6][5] = 0  # T2
        self.Ybus0[5][6] = -1 / (3 * (Grid.transformers["T2"].Rpu + 1j * Grid.transformers["T2"].Xpu) + self.Zt22)  # T2
        self.Ybus0[6][6] = 1 / (1j * self.totalx0generators_newpu2)

    elif Zt2_connection2 == "Delta" or Zt2_connection2 == "Ungrounded Wye":
        self.Ybus0[6][5] = 0  # T2
        self.Ybus0[5][6] = 0  # T2
        self.Ybus0[6][6] = 1 / (1j * self.totalx0generators_newpu2)

    else:
        print("Invalid Information For Transformer 2 Connection 2")
        exit(-1)

# All Ungrounded Wye have 0
elif Zt2_connection1 == "Ungrounded Wye":
    self.Ybus0[6][5] = 0  # T2
    self.Ybus0[5][6] = 0  # T2
    self.Ybus0[6][6] = 1 / (1j * self.totalx0generators_newpu2)

else:
    print("Invalid Information For Transformer 2 Connection 1")
    exit(-1)

        # Establish Lines -> Z0 = 3Z1
        # Set Non-diagonals just using -1/Z, Diagonals are same as original Ybus, or updated for generators in if segment

        self.Ybus0[1][2] = -1 / (3 * (Grid.transmissionline["L2"].Rpu + 1j * Grid.transmissionline["L2"].Xpu))  # L2
        self.Ybus0[2][1] = self.Ybus0[1][2]  # L2
        self.Ybus0[3][1] = -1 / (3 * (Grid.transmissionline["L1"].Rpu + 1j * Grid.transmissionline["L1"].Xpu))  # L1
        self.Ybus0[1][3] = self.Ybus0[3][1]  # L1
        self.Ybus0[4][2] = -1 / (3 * (Grid.transmissionline["L3"].Rpu + 1j * Grid.transmissionline["L3"].Xpu))  # L3
        self.Ybus0[2][4] = self.Ybus0[4][2]  # L3
        self.Ybus0[4][3] = -1 / (3 * (Grid.transmissionline["L6"].Rpu + 1j * Grid.transmissionline["L6"].Xpu)) # L6
        self.Ybus0[3][4] = self.Ybus0[4][3]  # L6
        self.Ybus0[5][3] = -1 / (3 * (Grid.transmissionline["L4"].Rpu + 1j * Grid.transmissionline["L4"].Xpu))  # L4
        self.Ybus0[3][5] = self.Ybus0[5][3]  # L4
        self.Ybus0[5][4] = -1 / (3 * (Grid.transmissionline["L5"].Rpu + 1j * Grid.transmissionline["L5"].Xpu))  # L5
        self.Ybus0[4][5] = self.Ybus0[5][4]  # L5

        # Set Diagonals, That do not have a generator attached
        self.Ybus0[1][1] /= 3
        self.Ybus0[2][2] /= 3
        self.Ybus0[3][3] /= 3
        self.Ybus0[4][4] /= 3
        self.Ybus0[5][5] /= 3

        # Print the Z0-bus matrix
        self.Zbus0 = np.linalg.inv(self.Ybus0)
        #print("Ybus0 Matrix")
        #self.printmatrix(self.Ybus0)
        #print("Z0-bus Matrix")
        #self.printmatrix(self.Zbus0)

# Function to easily print Matrices
def printmatrix(self, Matrix):
    i = 0
    while i < 7:
        j = 0
        print("\nRow " + str(i + 1))
        while j < 7:
            print(Matrix[i][j])
            j += 1
        i = i + 1
    print("\n")
```

Fault_Calculation Class

This class calculates fault currents and voltages in the power system. When the main function is run, the user is asked if they would like to perform a fault calculation. If they respond with YES, the follow up question is regarding the type of fault analysis they would like performed. The options for the fault analysis include Symmetrical Fault, Single Line to Ground Fault, Line to Line Fault, Double Line to Ground Fault. The user then gets to choose the bus for the fault location and enter a faulting impedance if they so desire.

Variable reference in code	Description
`faulttype`	`Type of fault input by user`
`faultlocation`	`Bus number input by user where fault is located`
`faulting_impedance`	`Optional user input for faulting impedance value`
`self.VF`	`Pre fault voltage`
`self.Zf`	`Faulting impedance (zero for bolted fault)`
`self.In_1`	`Positive sequence fault current`
`self.In_2`	`Negative sequence fault current`
`self.In_0`	`Zero sequence fault current`
`self.V_1`	`Positive sequence fault voltage`
`self.V_2`	`Negative sequence fault voltage`
`self.V_0`	`Zero sequence fault voltage`
`self.I_012`	`Array of all sequence fault currents`
`self.I_abc`	`Three phase current component vector`
`self.V_012`	`Array of all sequence fault voltages`
`self.V_abc`	`Three phase voltage component vector`
`self.Znn_1`	`Positive sequence fault impedance`
`self.Znn_2`	`Negative sequence fault impedance`
`self.Znn_0`	`Zero sequence fault impedance`
`self.A`	`Symmetrical Component Transform Matrix`

Relevant Equations:

<u>Example:</u> Single Line to Ground Fault

$$I_{n-0} = I_{n-1} = I_{n-2} = \frac{VF}{Z_{nn-0} + Z_{nn-1} + Z_{nn-2} + 3Zf}$$

Different I_n equations exist for each fault type, but the following generalized equations can be used to find $I_{abc}\ and\ V_{abc}$ for any fault type.

For any bus k:

$$\begin{bmatrix} V_{k-0} \\ V_{k-1} \\ V_{k-2} \end{bmatrix} = \begin{bmatrix} 0 \\ VF \\ 0 \end{bmatrix} - \begin{bmatrix} Z_{kn-0} & 0 & 0 \\ 0 & Z_{kn-1} & 0 \\ 0 & 0 & Z_{kn-2} \end{bmatrix} \begin{bmatrix} I_{n-0} \\ I_{n-1} \\ I_{n-2} \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & a^2 & a \\ 1 & a & a^2 \end{bmatrix}, where\ a = 1\angle 120°,\ a^2 = 1\angle 240°$$

$$V_{abc} = A\, V_{012}$$
$$I_{abc} = A\, I_{012}$$

Class Code: located in repository as Fault_Calculation.py

```
# File to Perform Fault Calculations
import numpy as np
class FaultCalculation:
    # Draw Zero Sequence for submission
    # Fault Calculation
    def __init__(self, Grid, SeqNet, faulttype: str, faultlocation: int, faulting_impedance: float):
        self.length = len(Grid.buses)
        self.V_0 = np.zeros(self.length, dtype=complex)
        self.V_1 = np.zeros(self.length, dtype=complex)
        self.V_2 = np.zeros(self.length, dtype=complex)
        self.V_012 = np.zeros((self.length, 3), dtype=complex)
        self.V_abc = np.zeros((self.length, 3), dtype=complex)
        self.In_1 = None
        self.In_2 = None
        self.In_0 = None
        self.I_012 = None
        self.I_abc = None

        self.Znn_1 = SeqNet.Zbus1[faultlocation-1][faultlocation-1]
        self.Znn_2 = SeqNet.Zbus2[faultlocation-1][faultlocation-1]
        self.Znn_0 = SeqNet.Zbus0[faultlocation-1][faultlocation-1]
        self.a1 = -0.5 + 1j * 0.8660254
        self.a2 = -0.5 - 1j * 0.8660254
        self.A = [[1, 1, 1], [1, self.a2, self.a1], [1, self.a1, self.a2]]
        # Vf will always be 1 per unit because we are neglecting pre fault currents
        self.VF = 1

        # A bolted fault will have a Zf of 0
        self.Zf = faulting_impedance
```

```
# Using the Fault Type Information, Calculate the Currents
if faulttype == "Symmetrical Fault":
    self.In_1 = self.VF/self.Znn_1
    self.In_2 = 0
    self.In_0 = 0

elif faulttype == "Single Line to Ground Fault":
    self.In_0 = self.VF/((self.Znn_0 +self.Znn_1 + self.Znn_2) + 3 * self.Zf)
    self.In_1 = self.In_0
    self.In_2 = self.In_0

elif faulttype == "Line to Line Fault":
    self.In_0 = 0
    self.In_1 = self.VF/(self.Znn_1 + self.Znn_2 + self.Zf)
    self.In_2 = -self.In_1

elif faulttype == "Double Line to Ground Fault":
    self.In_1 = self.VF/(self.Znn_1+(self.Znn_2 * (self.Znn_0 + 3 * self.Zf))/(self.Znn_2 + self.Znn_0+3 * self.Zf))
    self.In_2 = -self.In_1 * (self.Znn_0 + 3 * self.Zf) / (self.Znn_0 + 3 * self.Zf + self.Znn_2)
    self.In_0 = -self.In_1 * self.Znn_2 / (self.Znn_0 + 3 * self.Zf + self.Znn_2)

else:
    print("Invalid Fault Type")
    exit(-1)

# Set I_012
self.I_012 = np.array([[self.In_0], [self.In_1], [self.In_2]])

# Solve for all V_012 Values and V-012
for k in range(self.length):
    self.V_0[k] = -(SeqNet.Zbus0[k][faultlocation-1] * self.In_0)
    self.V_1[k] = self.VF - (SeqNet.Zbus1[k][faultlocation-1] * self.In_1)
    self.V_2[k] = -(SeqNet.Zbus2[k][faultlocation-1] * self.In_2)
    self.V_012[k] = np.array([[self.V_0[k]], [self.V_1[k]], [self.V_2[k]]]).flatten()

# Calculate I_abc and V_abc
self.I_abc = np.dot(self.A, self.I_012)
for k in range(self.length):
    self.V_abc[k] = np.dot(self.A, self.V_012[k])
print("\nI_abc")
for i in range(len(self.I_abc)):
    print(self.I_abc[i])
print("\nAll V_abc")
for k in range(self.length):
    print("V_abc[", k, "] = ", self.V_abc[k])
```