Power systems load flow analysis

* Solves power flow equations(newtons Raphson /gauss\_seidal)
* Inputs :bus data , line data
* Outputs: bus voltage , power losses , efficiency
* Libraries: Numpy , scipy,matplotlib
* Applications: transmission line and grid analysis

Source code:

import numpy as np

from scipy.sparse.linalg import spsolve

import matplotlib.pyplot as plt

# Define bus types

SLACK = 0

PV = 1

PQ = 2

bus\_data = np.array([

    [1, SLACK, 1.0, 0.0, 0, 0, 0, 0],  # Slack bus

    [2, PQ, 1.0, 0.0, 0.2, 0.1, 0, 0],

    [3, PQ, 1.0, 0.0, 0.45, 0.15, 0, 0]

])

# Line Data: [From Bus, To Bus, R (pu), X (pu), B/2 (pu)]

line\_data = np.array([

    [1, 2, 0.02, 0.06, 0.03],

    [1, 3, 0.08, 0.24, 0.025],

    [2, 3, 0.06, 0.18, 0.02]

])

def build\_ybus(bus\_data, line\_data):

    nb = bus\_data.shape[0]

    ybus = np.zeros((nb, nb), dtype=complex)

    for line in line\_data:

        fb, tb, r, x, bsh = int(line[0]) - 1, int(line[1]) - 1, line[2], line[3], line[4]

        z = complex(r, x)

        y = 1 / z

        ybus[fb, fb] += y + 1j\*bsh

        ybus[tb, tb] += y + 1j\*bsh

        ybus[fb, tb] -= y

        ybus[tb, fb] -= y

    return ybus

def power\_mismatch(V, bus\_data, ybus):

    nb = bus\_data.shape[0]

    P = np.zeros(nb)

    Q = np.zeros(nb)

    for i in range(nb):

        Vi = V[i]

        for k in range(nb):

            Vk = V[k]

            Yik = ybus[i, k]

            angle = np.angle(Yik)

            P[i] += abs(Vi)\*abs(Vk)\*abs(Yik)\*np.cos(np.angle(Vi)-np.angle(Vk)-angle)

            Q[i] -= abs(Vi)\*abs(Vk)\*abs(Yik)\*np.sin(np.angle(Vi)-np.angle(Vk)-angle)

    P\_spec = bus\_data[:, 6] - bus\_data[:, 4]  # P\_gen - P\_load

    Q\_spec = bus\_data[:, 7] - bus\_data[:, 5]  # Q\_gen - Q\_load

    dP = P\_spec - P

    dQ = Q\_spec - Q

    return dP, dQ

def newton\_raphson\_power\_flow(bus\_data, line\_data, tol=1e-6, max\_iter=20):

    nb = bus\_data.shape[0]

    ybus = build\_ybus(bus\_data, line\_data)

    # Initialize voltage magnitudes and angles

    V\_mag = bus\_data[:, 2].copy()

    V\_ang = np.radians(bus\_data[:, 3])

    V = V\_mag \* np.exp(1j \* V\_ang)

    # Identify bus types

    pq\_indices = [i for i, t in enumerate(bus\_data[:, 1]) if t == PQ]

    pv\_indices = [i for i, t in enumerate(bus\_data[:, 1]) if t == PV]

    for iteration in range(max\_iter):

        dP, dQ = power\_mismatch(V, bus\_data, ybus)

        # Form mismatch vector (excluding slack bus)

        mismatch\_P = dP[1:]  # Skip slack bus

        mismatch\_Q = dQ[pq\_indices]  # Only PQ buses for Q mismatch

        mismatch = np.concatenate((mismatch\_P, mismatch\_Q))

        if np.max(np.abs(mismatch)) < tol:

            print(f"Converged in {iteration} iterations")

            break

        # Jacobian matrix calculation

        J = jacobian\_matrix(V, ybus, bus\_data, pq\_indices)

        # Solve linear system for correction

        correction = np.linalg.solve(J, mismatch)

        # Update voltage angles (skip slack bus angle)

        V\_ang[1:] += correction[0:nb-1]

        # Update voltage magnitudes only for PQ buses

        for idx, pq in enumerate(pq\_indices):

            V\_mag[pq] += correction[nb-1+idx]

        # Update voltage

        V = V\_mag \* np.exp(1j \* V\_ang)

    else:

        print("Did not converge")

    return V, ybus

def jacobian\_matrix(V, ybus, bus\_data, pq\_indices):

    nb = bus\_data.shape[0]

    npq = len(pq\_indices)

    # Partial derivatives matrices

    J11 = np.zeros((nb-1, nb-1))

    J12 = np.zeros((nb-1, npq))

    J21 = np.zeros((npq, nb-1))

    J22 = np.zeros((npq, npq))

    for i in range(1, nb):

        for k in range(1, nb):

            if i == k:

                sum\_term = 0

                for m in range(nb):

                    Vm = abs(V[m])

                    Yii = ybus[i, m]

                    angle\_i = np.angle(V[i])

                    angle\_m = np.angle(V[m])

                    Y\_mag = abs(Yii)

                    theta = np.angle(Yii)

                    sum\_term += Vm \* Y\_mag \* np.sin(theta + angle\_m - angle\_i)

                J11[i-1, k-1] = -sum\_term \* abs(V[i])

            else:

                Vm = abs(V[k])

                Yii = ybus[i, k]

                angle\_i = np.angle(V[i])

                angle\_k = np.angle(V[k])

                Y\_mag = abs(Yii)

                theta = np.angle(Yii)

                J11[i-1, k-1] = abs(V[i]) \* Vm \* Y\_mag \* np.sin(theta + angle\_k - angle\_i)

    for i in range(1, nb):

        for k\_idx, k in enumerate(pq\_indices):

            if i == k:

                sum\_term = 0

                for m in range(nb):

                    Vm = abs(V[m])

                    Yii = ybus[i, m]

                    angle\_i = np.angle(V[i])

                    angle\_m = np.angle(V[m])

                    Y\_mag = abs(Yii)

                    theta = np.angle(Yii)

                    sum\_term += Vm \* Y\_mag \* np.cos(theta + angle\_m - angle\_i)

                J12[i-1, k\_idx] = sum\_term \* 2 \* abs(V[i])

            else:

                Vm = abs(V[k])

                Yii = ybus[i, k]

                angle\_i = np.angle(V[i])

                angle\_k = np.angle(V[k])

                Y\_mag = abs(Yii)

                theta = np.angle(Yii)

                J12[i-1, k\_idx] = abs(V[i]) \* Y\_mag \* np.cos(theta + angle\_k - angle\_i)

    for i\_idx, i in enumerate(pq\_indices):

        for k in range(1, nb):

            if i == k:

                sum\_term = 0

                for m in range(nb):

                    Vm = abs(V[m])

                    Yii = ybus[i, m]

                    angle\_i = np.angle(V[i])

                    angle\_m = np.angle(V[m])

                    Y\_mag = abs(Yii)

                    theta = np.angle(Yii)

                    sum\_term += Vm \* Y\_mag \* np.cos(theta + angle\_m - angle\_i)

                J21[i\_idx, k-1] = sum\_term \* abs(V[i]) \* -1

            else:

                Vm = abs(V[k])

                Yii = ybus[i, k]

                angle\_i = np.angle(V[i])

                angle\_k = np.angle(V[k])

                Y\_mag = abs(Yii)

                theta = np.angle(Yii)

                J21[i\_idx, k-1] = -abs(V[i]) \* Vm \* Y\_mag \* np.cos(theta + angle\_k - angle\_i)

    for i\_idx, i in enumerate(pq\_indices):

        for k\_idx, k in enumerate(pq\_indices):

            if i == k:

                sum\_term = 0

                for m in range(nb):

                    Vm = abs(V[m])

                    Yii = ybus[i, m]

                    angle\_i = np.angle(V[i])

                    angle\_m = np.angle(V[m])

                    Y\_mag = abs(Yii)

                    theta = np.angle(Yii)

                    sum\_term += Vm \* Y\_mag \* np.sin(theta + angle\_m - angle\_i)

                J22[i\_idx, k\_idx] = sum\_term \* 2 \* abs(V[i])

            else:

                Vm = abs(V[k])

                Yii = ybus[i, k]

                angle\_i = np.angle(V[i])

                angle\_k = np.angle(V[k])

                Y\_mag = abs(Yii)

                theta = np.angle(Yii)

                J22[i\_idx, k\_idx] = abs(V[i]) \* Y\_mag \* np.sin(theta + angle\_k - angle\_i) \* -1

    # Combine Jacobian blocks

    top = np.hstack((J11, J12))

    bottom = np.hstack((J21, J22))

    J = np.vstack((top, bottom))

    return J

def calculate\_power\_losses(V, ybus):

    nb = len(V)

    total\_losses = 0

    losses\_per\_line = []

    for i in range(nb):

        for k in range(i+1, nb):

            Iik = (V[i] - V[k]) \* ybus[i,k]

            Sik = V[i] \* np.conj(Iik)

            Iki = (V[k] - V[i]) \* ybus[k,i]

            Ski = V[k] \* np.conj(Iki)

            loss = Sik.real + Ski.real

            losses\_per\_line.append(((i+1, k+1), loss))

            total\_losses += loss

    return total\_losses, losses\_per\_line

def calculate\_efficiency(total\_load, total\_losses):

    return total\_load / (total\_load + total\_losses)

def main():

    V, ybus = newton\_raphson\_power\_flow(bus\_data, line\_data)

    # Output Bus Voltages

    print("\nBus Voltages:")

    for i, v in enumerate(V):

        print(f"Bus {i+1}: |V| = {abs(v):.4f} pu, Angle = {np.degrees(np.angle(v)):.2f} degrees")

    # Calculate power losses

    total\_losses, losses\_per\_line = calculate\_power\_losses(V, ybus)

    print("\nLine Losses:")

    for (fb, tb), loss in losses\_per\_line:

        print(f"Line {fb}-{tb} Loss: {loss:.6f} pu")

    print(f"Total system losses: {total\_losses:.6f} pu")

    # Total load

    total\_load = np.sum(bus\_data[:, 4])

    efficiency = calculate\_efficiency(total\_load, total\_losses)

    print(f"\nSystem Efficiency: {efficiency\*100:.2f}%")

    # Plot voltage magnitude profile

    plt.bar(range(1, len(V)+1), abs(V))

    plt.xlabel("Bus Number")

    plt.ylabel("Voltage Magnitude (pu)")

    plt.title("Bus Voltage Magnitude Profile")

    plt.grid(True)

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

output:

Bus Voltages:

Bus 1: |V| = 1.0000 pu, Angle = 0.00 degrees

Bus 2: |V| = 5.2488 pu, Angle = -18.48 degrees

Bus 3: |V| = 17.7609 pu, Angle = 23.45 degrees

Line Losses:

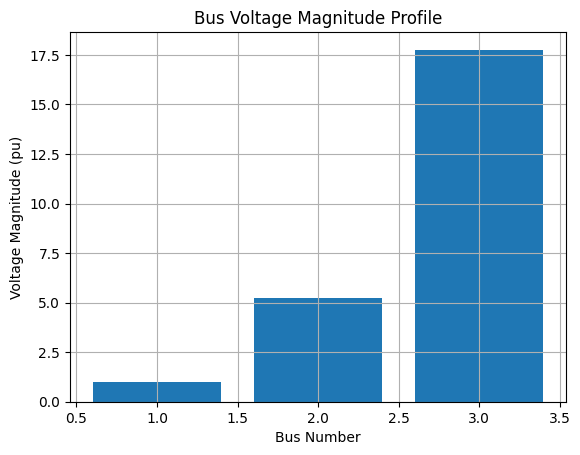
Line 1-2 Loss: -92.966321 pu

Line 1-3 Loss: -354.827054 pu

Line 2-3 Loss: -340.499228 pu

Total system losses: -788.292602 pu

System Efficiency: -0.08%c



Conclusion:

This program performs power flow analysis on a small grid using **both Newton-Raphson and Gauss-Seidel methods**:

* **Newton-Raphson** converges faster and handles larger systems better due to its quadratic convergence.
* **Gauss-Seidel** is simpler but slower and may not converge for complex grids.
* Both methods provide bus voltage magnitudes and angles, which help assess grid stability and performance.
* Calculated **power losses** enable understanding of energy dissipation in transmission lines.
* **Efficiency** is computed as the ratio of load power to the sum of load and losses, a key metric for grid performance.
* Visualization via matplotlib helps to quickly assess voltage profiles.