Technical University of Denmark

46705 Power grid analysis and protection

Assignment 1: Power Flow in Python for Grid Analysis

Group 2

By

Maxwell Guerne-Kieferndorf, s226411

Carlos Hermana Rivera, s220258

Alina Adriana Feldmeier, s222582

Gianmarco Petracchin, s230143

March 28, 2023

Contents

Contribution

Task #		Alina	Carlos	Maxwell	Gianmarco
Coding	Task 1 - PowerFlowNewton		X		X
	Task 2 - LoadNetworkData	X		X	
	Task 3 - Display results	X	X	X	
Task 4 - Analysis of the Kundur system		X	X	X	X
Report writing		X	X	X	X

Table 1: Table of contributions

1 Introduction & Learning objectives

This assignment's goals are to create Python software that analyzes the power flow in a particular network and examines various scenarios to determine whether voltage or generator limits have been exceeded [1].

The learning objectives for this assignment are:

- Understand the theory and reasoning behind a power flow analysis: the inputs and outputs, the iterative calculations, etc.
- Implement a power flow analysis on Python.
- Be able to build a network model from the data given.
- Adapt the script to different power flow scenarios.

2 Theoretical basis

To develop correctly a power flow analysis, first, it is important to understand the physical, mathematical, and electrical principles behind each power system. In this section theory behind an electrical power flow analysis will be briefly explained.

For a power system to work properly, certain conditions must be met:

- Generation must be equal to consumption plus losses to achieve a constant frequency. In a real grid, energy consumption varies through time so generation must also vary to compensate. Therefore, frequency is always fluctuating inside a certain range.
- Bus voltage magnitude must also fluctuate inside a safe range.
- Generators must be within their operating limits.
- Transformers and lines must not be overloaded.

A power flow analysis consists of computing voltages magnitudes and angles at every bus in a power system during a steady state condition [2].

A common grid configuration is divided into buses connected by transmission lines. Each bus is characterized by P,Q, V, and θ. We can set three types of buses:

- **PV buses:** where P and V are known. They represent generators.
- **PQ buses:** where P and Q are known. They represent loads.
- **Reference bus:** where V and θ are known. Only one bus in the system can be this type.

To carry out the power flow analysis we will compute certain network equations:

$$\mathbf{I} = \mathbf{Y}_{\text{bus}} \cdot \mathbf{V} \tag{1}$$

This equation is applied in a matrix way to characterize all buses.

To relate the current and voltage in each bus we have to consider the following:

$$\mathbf{V} \cdot \mathbf{I}^* = P + jQ \tag{2}$$

The boundary conditions imposed by the different buses make the problem nonlinear. So the power flow equations are solved using iterative methods such as the Newton-Raphson, which is the one used in this study.

Basically, the Newton-Raphson method consists in inferring the value of the unknown variables (e.g. θ and V for PQ buses) in each bus to calculate the other variables (e.g. P and Q for PQ buses) and then calculate the error with respect to the known variables (e.g. P and Q for PQ buses) as follows:

$$\Delta f = P_{known} - P(\theta, V)_{inferred} \tag{3}$$

The above equation only applies to PQ buses. After calculating the error, a correction must be made in each inferred value in each iteration until the error is low enough.

3 Task 1: `PowerFlowNewton()`

Our goal is to understand whether the grid under analysis is working properly depending on the data provided at each bus. To do this it will be necessary to apply an iterative method to figure out whether the values obtained are close to the predetermined ones and that therefore their difference is as close to zero as possible. For this purpose, the function Power Flow Newton

will have to be implemented. Power flow Newton is a function based on the Newton-Raphson method, an iterative method for solving n irrational equations in n unknowns, i.e., the power flow equations, by iteratively updating the voltage magnitudes and angles at each node until the power balance is achieved. The method requires an initial guess for the voltage magnitudes and angles, and then it iteratively refines the solution until convergence is achieved. Convergence is achieved when the difference between the calculated and actual values of the power flow variables is below a specified tolerance. The function considered is itself composed of other functions that allow the Newton-Raphson method to be solved and thus verify convergence, that is, whether the power is balanced at each node of the network under analysis.

The functions are as follows:

- Calculate F
- Check tolerance
- Generate derivatives
- Generate Jacobian
- Update voltages

3.1 `calculate_F()`

This function allows us to calculate the mismatch vector F, a vector composed by the mismatch between specified values of active and reactive power and the corresponding calculated ones. The inputs needed are the bus admittance matrix Ybus ($N \times N$ matrix based on the layout of the considered grid), the specified apparent power injection vector Sbus ($N \times 1$), the complex bus voltage vector V ($N \times 1$), and the indices for PV-busses and PQ-busses.

3.2 `CheckTolerance()`

This part is implemented to check if the method is converging or not. This function verifies if the mismatch vector F is smaller than a specified tolerance. The inputs needed are the mismatch vector F, the iteration counter n (useful to know how many iterations are needed for convergence) and the specified error tolerance, now set at 1×10^{-4}. If the difference is less than the tolerance then the function will return the "success" flag.

3.3 `generate_Derivatives()`

The Jacobian matrix that we should create to solve the system of power flow equations is composed of derivatives of active power and reactive power, so it will be necessary to find these components. The Generate Derivatives function has this task. It calculates the derivatives of apparent power S with respect to voltage magnitude and to voltage angles. In order to complete the assignment the admittance matrix and the complex vector of voltages are needed. The output of Generate Derivatives are two $N \times N$ matrices of the derivatives of S.

3.4 `generate_Jacobian()`

The Jacobian matrix is composed of the derivatives of active and reactive power with respect to voltage magnitude and its angle, so it will be necessary to divide the apparent power components found in the previous step into the respective P and Q. With these components, the function will create sub-matrices that will form the desired Jacobian matrix. An important step to observe is the use of bus indices, so the Jacobian can be assembled to match the structure of the vector F, with which it will be multiplied

3.5 `Update_Voltages()`

This function is used to update the complex voltage vector after an iteration is finished. To do so the inputs have to be the incremented vector dx, the previous complex bus voltage vector, and the indices to the PV and PQ busses.

4 Task 2: `LoadNetworkData()`

The purpose of this function is to construct the different matrices needed to compute the power flows in the system. The data has been read out in the function `ReadNetworkData()`, which was alreadly provided to us.
The bus data, which is a data structure generated by the `ReadNetworkData()` function, is a list of N lists which contain the bus number, the bus label, the nominal voltage at the bus and the bus code of the respective buses. The required data from this data structure are the bus labels and the bus codes. The bus labels are simply to identify which bus number corresponds to which real bus. The bus codes are used to classify the bus indices (which are basically the bus numbers minus 1 to account for the fact that python indices start from 0) into the categories reference (bus code 3), PV (bus code 2) and PQ buses (bus code 1). The lists of indices `pq_index`, `pv_index` and `ref` are needed for the solution of the power flow problem.
The vector `Sbus` is of length N and contains the total apparent power which is injected into each bus. The values for the apparent power are obtained from the generator and load data which are extracted with `ReadNetworkData()`. The load power is counted negatively in this vector, while generator power is positive. Where there is no load or generator connected, the entries are 0. The vector `S_LD` is also of length N and contains only the load power drawn at each node. In both vectors, the apparent power is given in the per unit system, i.e. referred to the base power provided in `ReadNetworkData()`.
Finally, the matrices $\mathbf{Y}_{\text{bus}}$ and $\mathbf{Y}_{\text{from}}$ and $\mathbf{Y}_{\text{to}}$ are constructed. $\mathbf{Y}_{\text{bus}}$ is an $N \times N$ matrix which is built up as follows:

$$Y_{\text{bus},kk} = \sum_i y_{k,i} \quad \text{where } k = 1, \ldots, N \ . \tag{4}$$

Here, $y_{k,i}$ denotes the admittance (including shunt susceptance) of the i^{th} object (line or transformer) connected to bus k. When calculating $y_{k,i}$, for transmission lines, we must consider the transmission line parameters for the pi model of the line, r_i, x_i and b_i, which are obtained from `ReadNetworkData()` again. This means that

$$y_{k,i} = \frac{1}{r_i + jx_i} + jb_i/2 \ . \tag{5}$$

For transformers, the admittance is calculated based on the phase shifter equation for generality as follows:

$$y_{k,i} = \frac{y_{\text{e},i}}{|\tilde{n}|^2} \ . \tag{6}$$

Here, $y_{\text{e},i}$ is the equivalent impedance (series resistance, leakage reactance) of the transformer and $\tilde{n} = n\angle\underline{\alpha}$ is the complex turns ratio, all given by `ReadNetworkData()`. The off-diagonal elements are simply minus the total admittance of the lines connecting the buses k and m:

$$Y_{\text{bus},km} = -\sum_{j=0}^{M} y_j, \tag{7}$$

where y_j is the admittance of each line and M is the total number of lines connected in parallel between buses k and m.
The matrix $\mathbf{Y}_{\text{to}}$ is constructed so that the rows represent a connection (line or transformer) in the network between two buses and the columns represent the buses, therefore is has the dimensions $K \times N$ (K is the number of connections/branches, N the number of buses). Each row has two

nonzero entries, one at the column representing the starting point of the connection ("direction" is defined in the input text file but is arbitrary), and one representing the end point. Consider a general admittance matrix for the connection:

$$\mathbf{Y} = \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} \tag{8}$$

In the $\mathbf{Y}_{\text{to}}$ matrix, the entry representing the end point of the connection contains y_{22} and the entry representing the starting point contains y_{12}. The matrix $\mathbf{Y}_{\text{from}}$ is built up analogously, but here, the connection starting point entry contains y_{11} and the end point contains y_{21}. These two matrices are then used to compute the branch flows, however this computation takes place in the `DisplayResults()` function (cf. (9), (10)).

5 Task 3: `DisplayResults()`

The function has two main inputs and three optional parameters:

- `V`: An array of the complex voltages in the power system.
- `lnd`: Contains information from the `LoadNetworkData()` function about the power system, including the admittance matrix, bus and branch data, etc.
- `rnd`: Specifies the number of decimal places to round the results to. The default value is 3.
- `filenameBus`: this variable is used to set the filename for exporting the bus results to a CSV file.
- `filenameBranch`: this is a parameter to establish the name of the CSV file to export the branch data.

The function first initializes some variables by extracting data from the lnd object. It then computes the apparent power injection at each bus by multiplying the voltage vector with the complex conjugate of the admittance matrix multiplied by the voltage vector.

After that, a dictionary-based data structure called `bus_results` is created. The purpose of it is to store the power flow results at each bus. These include the bus number, bus label, voltage magnitude, and angle, generator real and reactive power injection, and load real and reactive power injection. The script loops over each bus and appends the power flow results to their corresponding category. The dictionary is then converted into a pandas data frame called `busResDF`.

Next, the function computes the complex power flow on each branch by multiplying the complex voltage on the "from" end and "to" end of the branch with the complex conjugate of the respective branch currents. The following matrix multiplications are implemented in python to compute the branch flows (see also section 7.3, lines 57-58):

$$\mathbf{S}_{\text{from}} = \mathbf{V}_{\text{from}}\mathbf{I}^*_{\text{from}} \quad \text{, where} \quad \mathbf{I}_{\text{from}} = \mathbf{Y}_{\text{from}}\mathbf{V} \tag{9}$$

$$\mathbf{S}_{\text{to}} = \mathbf{V}_{\text{to}}\mathbf{I}^*_{\text{to}} \quad \text{, where} \quad \mathbf{I}_{\text{to}} = \mathbf{Y}_{\text{to}}\mathbf{V} \tag{10}$$

In both equations, $\mathbf{V}$ is the vector of bus voltages and $\mathbf{V}_{\text{from}}$ and $\mathbf{V}_{\text{to}}$ are vectors containing the voltages at the beginnings/ends of the lines. The function then creates a structure called `branch_results` to store the branch flow results. The branch flow results include the branch number, "from" and "to" bus numbers, real and reactive power injections, etc. This data list is filled using the same method as before and is converted into `branchResDF` at the end.

Finally, the function prints the two data frames and exports them to CSV files using the filenames specified in the input parameters. The function returns the two data frames just in case the information needs to be used later.

```
The maximum mismatch is  15.75 in the iteration  0
The maximum mismatch is  1.9065233284563412 in the iteration  1
The maximum mismatch is  0.08263445156830507 in the iteration  2
The maximum mismatch is  0.0007086528368844824 in the iteration  3
The maximum mismatch is  3.5427582645297 66e-08 in the iteration  4
The Newton Rapson Power Flow Converged in 4 iterations!
================================== Bus results ==================================
    Bus Nr. Bus Label  Voltage Mag. (pu)  Voltage Ang. (deg) Gen. P (pu) Gen. Q (pu) Load P (pu) Load Q (pu)
0       1      BUS1             1.000               0.00       7.268       1.095           -           -
1       2      BUS2             1.000             -11.02         7.0        2.28           -           -
2       3     BUS12             1.000             -21.46         7.0       2.324           -           -
3       4     BUS11             1.000             -11.03         7.0       1.061           -           -
4       5    BUS101             0.983              -5.02           -           -           -           -
5       6    BUS102             0.969             -15.85           -           -           -           -
6       7      BUS3             0.956             -24.51           -       0.735       11.59           -
7       8     BUS13             0.954             -34.80           -       0.899       15.75           -
8       9    BUS112             0.969             -26.29           -           -           -           -
9      10    BUS111             0.984             -15.87           -           -           -           -

================================ Branch flows ================================
    Branch Nr.  From Bus  To Bus  From Bus P inj. (pu)  From Bus Q inj. (pu)  To Bus P inj. (pu)  To Bus Q inj. (pu)
0            1         5       6                 3.607                 0.224              -3.540              0.381
1            2         5       6                 3.607                 0.223              -3.539              0.382
2            3         6       7                 7.014                 0.435              -6.909              0.589
3            4         6       7                 7.010                 0.432              -6.905              0.592
4            5         7       8                 0.741                -0.149              -0.728             -0.020
5            6         7       8                 0.741                -0.149              -0.728             -0.020
6            7         7       8                 0.741                -0.149              -0.728             -0.020
7            8         8       9                -6.785                 0.478               6.887              0.511
8            9         8       9                -6.781                 0.481               6.883              0.507
9           10         9      10                -3.413                 0.326               3.475              0.230
10          11         9      10                -3.412                 0.327               3.475              0.229
11          12         1       5                 7.268                 1.095              -7.214             -0.446
12          13         2       6                 7.000                 2.280              -6.946             -1.630
13          14         3       9                 7.000                 2.324              -6.946             -1.671
14          15         4      10                 7.000                 1.061              -6.950             -0.459
```

Figure 1: Results displayed from the studied grid

Below, a screenshot of the obtained results for the Kundur power system can be seen.

Some comments to better understand the results:

- The number of iterations and the error magnitude (calculated in `checkTolerance`) in each iteration can be found in the top.
- Bus 1 is the reference bus.
- In the displayed results for each bus, regarding power, just positive values are shown as a distinction between generated and consumed power is made (negative generated power = positive consumed power).
- Active power injected (positive) by a bus is always higher than the active power received (negative) by the connected bus. This is due to the losses through the transmission line.
- The reactive power injected and absorbed by the connected buses also changes because it is generated or absorbed throughout lines.
- There are 15 branches, which correspond to 11 transmission lines and 4 transformers.

6 Task 4: Analysis of the Kundur's two area test system

The implemented code was used to analyze the *kundur two area system* regarding its voltage stability and the operational limits of the generators for different operating conditions. The results for the voltage magnitude in p.u. and the loadings of the generators regarding their active power injection and their injection of apparent power in percent for the base case scenario of the grid can be seen in table 2. The buses 5 to 10 of the kundur system are not connected to generators, hence there is no power injection on those buses.

bus nr.	voltage magnitude (pu)	Generator loading (P inj.)	Generator loading (S inj.)
1	1	97.44%	81.67%
2	1	100.00%	81.80%
3	1	100.00%	81.95%
4	1	100.00%	78.67%
5	0.983	-	-
6	0.969	-	-
7	0.956	-	-
8	0.954	-	-
9	0.969	-	-
10	0.983	-	-

Table 2: Voltage magnitues and generator loadings in the kundur system, base case.

It can be seen that for each bus the voltage magnitude stays within an acceptable deviation of ± 0.05 pu and that the injection of apparent and active power of the generators causes no violation of operational limits. In addition, three out of the four generators in the grid operate at their operational limit of 100% regarding the injection of active power. Since the power injections of the generators are known values in the power flow equations, as explained in chapter 2, loadings of 100% for the generators on bus 2, 3 and 4 show, that the given power injections of those generators were met. For the generator on bus 1, which functions as slack machine, the maximum loading of 100% is not met. This means that in order to solve the equations of the Newton-Rapson method the maximum utilization of the slack machine does not have to be reached.
Consequentely, the modeling of the kondur system is sufficient to not violate any security constraints in the base case scenario.
In table 3 the same results for the kundur system as in table 2 are shown for an outage scenario, where one line between bus 6 and bus 7 is outaged.

bus nr.	voltage magnitude (pu)	Generator loading (P inj.)	Generator loading (S inj.)
1	1	101.79%	86.72%
2	1	100.00%	91.00%
3	1	100.00%	84.01%
4	1	100.00%	78.98%
5	0.975	-	-
6	0.945	-	-
7	0.899	-	-
8	0.940	-	-
9	0.962	-	-
10	0.982	-	-

Table 3: Voltage magnitues and generator loadings in the kundur system, post-contingency case.

In the post-contingency scenario the buses 6, 7 and 8 show voltages with values smaller than the required 0.95 pu. Generally, the grid stability requirements of some countries require the bus voltages to stay within a range of ± 0.1 pu, in which case the bus voltages in the kundur system in post-contingency case would not represent a violation of security constraints (apart from a very slight violation by 0.001 pu at bus 7) [3]. However, in this case the condition for the bus voltages was to stay within a limit of ± 0.05 pu, which is not satisfied for the analyzed post-contingency state of the kundur system [1].
In addition, the generator on bus 1 is not working within its operational limits regarding the injection of active power. The overloading generator is the slack machine, which means that in order to find a solution for the equations of the Newton-Raphson method, the injection of active power of the slack machine has to be higher than the operational limit. A generator exceeding its maximum load is clearly not admissible due to the risk of damage [3].

For the injection of apparent power, the generators do not exceed their operational limits.
In conclusion, the kundur two area system is not meeting the required security conditions in the analyzed post-contingency case and is therefore not modelled adequately. Adjustments in the grid modeling would have to be made in order to meet the required security constraints.

7 Conclusions

In this assignment we learned what it means to analyze the power flow of a small power grid composed of 10 buses, 4 generators, 4 transformers and 11 lines. This network has been analyzed under stationary and post-contingency conditions, this has allowed us to understand how the power injection and voltage behave in these situations. To do so it was necessary to apply the iterative method of Newton-Raphson, with which we were able to verify whether the network limits were respected or not.
We used Python to build up the network simulation with the data provided and to implement the analysis. For the Newton-Raphson method, we used the function `PowerFlowNewton()`, for building the necessary data structures, the function `LoadNetworkData()` and to process the results the function `DisplayResults()`.
Another fundamental aspect covered by this assignment is the ability to use the same Python code for the analysis of any network, in fact in our case, the first analysis was done on a simple test network to verify the functioning of the code, Later, when the code was completed, we applied it to the Kundur's two area test system. As a result, changing only the text file providing the network parameters allows for the analysis of any network.

References

[1] Hjörtur Jóhannsson. *46705 Assignment 1: Power Flow in Python for Grid Analysis.*

[2] Hjörtur Jóhannsson. *Lecture 2 notes from 46705 Power grid analysis and protection.*

[3] Mattia Marinelli. *Lecture notes from 46745 Introduction to wind power in the power system.*

Appendix

7.1 Code for `PowerFlowNewton()`

```
"""
46705 - Power Grid Analysis
This file contains the definitions of the functions needed to
carry out Power Flow calculations in python.

How to carry out Power Flow in a new *.py file?
See the example in table 1 in the assignment text
"""

import numpy as np
import pandas as pd
import os
import csv


# 1. the PowerFlowNewton() function
def PowerFlowNewton(Ybus,Sbus,V0,pv_index,pq_index,max_iter,err_tol):
    # initialize
    success = 0 # Initialization of flag, counter and voltage
    n = 0       # number of iteration (index of the data calculated)
    V = V0      # initial guess

    # Determine mismatch between initial guess and and specified value for P and Q
    F = calculate_F(Ybus,Sbus,V,pv_index,pq_index)
    success = CheckTolerance(F,n,err_tol) # Check if the desired tolerance is
    reached

    while (not success) and (n < max_iter): # Start Newton iterations
        # Update counter
        n += 1
        # Generate the Jacobian matrix
        J_dS_dVm,J_dS_dTheta = generate_Derivatives(Ybus,V)
        J = generate_Jacobian(J_dS_dVm,J_dS_dTheta,pv_index,pq_index)
        # Compute step
        dx = np.linalg.solve(J,F)
        # Update voltages and check if tolerance is now reached
        V = Update_Voltages(dx,V,pv_index,pq_index)
        F = calculate_F(Ybus,Sbus,V,pv_index,pq_index)
        success = CheckTolerance(F,n,err_tol)

        print('iteration | maximum P & Q mismatch (pu)')
        print('----------|-------------------------\n',
              f'   {n}    |     {np.max(F):.3f}      \n')

    if success:
        print('The Newton Raphson Power Flow Converged in %d iterations!' % (n,))
    else:
        print('No Convergence!!\n Stopped after %d iterations without solution...'
     % (n,))

    return V,success,n

# 2. the calculate_F() function
def calculate_F(Ybus,Sbus,V,pv_index,pq_index):

    Delta_S = Sbus - V * (Ybus.dot(V)).conj() #check theory and ask professor,
    shouldn't it be diag (V)
    Delta_P = Delta_S.real
    Delta_Q = Delta_S.imag

    F = np.concatenate((Delta_P[pv_index],Delta_P[pq_index],Delta_Q[pq_index]),
    axis=0)

    return F
```

```
61
62
63 # 3. the CheckTolerance() function
64 def CheckTolerance(F,n,err_tol):
65
66     #Compare n value with n-1 and see if the error is higher than err_tol
67     #if err < err_tol ---> success = 1
68     #else ---> success = 0
69
70     normF = np.linalg.norm(F,np.inf) #Returns the greatest absolute value of the
       column vector F
71
72     print("The maximum mismatch is ", normF, "in the iteration ", n)
73     if normF < err_tol:
74         success = True
75     else:
76         success = False
77
78     return success
79
80
81 # 4. the generate_Derivatives() function
82 def generate_Derivatives(Ybus,V):
83
84     J_ds_dVm=np.diag(V / np.absolute(V)).dot(np.diag((Ybus.dot(V)).conj())) + \
85              np.diag(V).dot(Ybus.dot(np.diag(V / np.absolute(V))).conj())
86     J_dS_dTheta = 1j * np.diag(V).dot((np.diag(Ybus.dot(V)) - Ybus.dot(np.diag(V))
       ).conj())
87
88     return J_ds_dVm,J_dS_dTheta
89
90
91 # 5. the generate_Jacobian() function
92 def generate_Jacobian(J_dS_dVm,J_dS_dTheta,pv_index,pq_index):
93
94     # Append PV and PQ indices for convenience
95     pvpq_ind = np.append(pv_index, pq_index)
96
97     # Create the sub-matrices
98     J_11 = np.real(J_dS_dTheta[np.ix_(pvpq_ind, pvpq_ind)])
99     J_12 = np.real(J_dS_dVm[np.ix_(pvpq_ind, pq_index)])
100    J_21 = np.imag(J_dS_dTheta[np.ix_(pq_index, pvpq_ind)])
101    J_22 = np.imag(J_dS_dVm[np.ix_(pq_index, pq_index)])
102
103    # Merge the sub-matrices to create the Jacobian matrix
104    J = np.block([[J_11,J_12],[J_21,J_22]])
105
106    return J
107
108
109 # 6. the Update_Voltages() function
110 def Update_Voltages(dx,V,pv_index,pq_index):
111
112    # Note: difference between Python and Matlab when using indices
113    N1 = 0; N2 = len(pv_index) # dx[N1:N2]-ang. on the pv buses
114    N3 = N2; N4 = N3 + len(pq_index) # dx[N3:N4]-ang. on the pq buses
115    N5 = N4; N6 = N5 + len(pq_index) # dx[N5:N6]-mag. on the pq buses
116
117    Theta = np.angle(V); Vm = np.absolute(V)
118    if len(pv_index)>0:
119        Theta[pv_index] += dx[N1:N2]
120
121    if len(pq_index)>0:
122        Theta[pq_index] += dx[N3:N4]
123        Vm[pq_index] += dx[N5:N6]
124    V = Vm * np.exp(1j * Theta)
125
126    return V
```

7.2 Code for `LoadNetworkData()`

```
import numpy as np
import ReadNetworkData as rd


def LoadNetworkData(filename):
    global Ybus, Sbus, V0, buscode, pq_index, pv_index, ref, Y_fr, Y_to, br_from,
    \
        br_to, S_LD, ind_to_bus, bus_to_ind, MVA_base, bus_labels, bus_kv
    # read in the data from the file...
    bus_data, load_data, gen_data, line_data, tran_data, mva_base, bus_to_ind, \
        ind_to_bus = rd.read_network_data_from_file(filename)

    #
    ##############################################################################################

    # Construct the required matrices from the data obtained from "
    read_network_data..."#
    #
    ##############################################################################################

    MVA_base = mva_base

    N = len(bus_data)  # Number of buses
    buscode = []
    bus_labels = []
    bus_kv = []
    # read out important parameters for each bus
    for row in bus_data:
        b_nr, label, kv, code = row
        buscode.append(code)
        bus_labels.append(label)
        # bus_kv.append(kv)

    # bus_kv = np.array(bus_kv)
    buscode = np.array(buscode)

    pq_index = np.where(buscode == 1)[0] # Find indices for all PQ-busses
    pv_index = np.where(buscode == 2)[0] # Find indices for all PV-busses
    ref = np.where(buscode == 3)[0]      # Find index for ref bus

    S_LD = np.zeros(N, dtype=complex)   # complex load at each bus
    Sbus = np.zeros(N, dtype=complex)   # power injection at each bus
    V0 = np.ones(N, dtype=complex)      # voltage at each bus

    for line in load_data:
        bus_nr, PLD, QLD = line # extract load data
        ix = bus_to_ind[bus_nr]
        S_LD[ix] = (PLD + 1j * QLD) / MVA_base
        Sbus[ix] -= S_LD[ix]

    for line in gen_data:
        bus_nr, MVA, MW_gen = line
        ix = bus_to_ind[bus_nr]
        Sbus[ix] += MW_gen / MVA_base


    N_lines = len(line_data)
    N_trans = len(tran_data)
    N_branches = N_lines + N_trans
    Ybus = np.zeros((N, N), dtype=complex)

    # Create the two branch admittance matrices
    Y_fr = np.zeros((N_branches, N), dtype=np.complex)
    Y_to = np.zeros((N_branches, N), dtype=np.complex)

    br_from = np.zeros(N_branches, dtype=np.int8) # The from busses
    br_to = np.zeros(N_branches, dtype=np.int8) # The to busses
```

```
62
63      # Fill the two branch admittance matrices and the bus admittance matrix with
        line data first
64      for line, ii in zip(line_data, range(N_lines)):
65          bus_fr, bus_to, id_, R, X, B_2 = line      # row of line data
66          br_from[ii] = bus_to_ind[bus_fr]  # from node is the first (zeroth)
        element
67          br_to[ii] = bus_to_ind[bus_to]  # to node is the second element
68
69          Z_br = R + 1j * X   # branch impedance
70          B_br = 1j * B_2 / 2    # branch shunt susceptance (divide by 2 needed?)
71          Y_fr[ii, br_from[ii]] = 1 / Z_br + B_br
72          Y_fr[ii, br_to[ii]] = -1 / Z_br
73          Y_to[ii, br_from[ii]] = -1 / Z_br
74          Y_to[ii, br_to[ii]] = 1 / Z_br + B_br
75
76          Ybus[br_from[ii], br_from[ii]] += 1 / Z_br + B_br
77          Ybus[br_from[ii], br_to[ii]] += -1 / Z_br
78          Ybus[br_to[ii], br_from[ii]] += -1 / Z_br
79          Ybus[br_to[ii], br_to[ii]] += 1 / Z_br + B_br
80
81      # Fill the two branch admittance matrices and the bus admittance matrix with
        transformer data
82      for tran, ii in zip(tran_data, range(N_lines, N_branches)):
83          bus_fr, bus_to, id_, R, X, n, ang1 = tran      # row of tran data
84          br_from[ii] = bus_to_ind[bus_fr]   # from node is the first (zeroth)
        element, subtract 1 for correct indexing
85          br_to[ii] = bus_to_ind[bus_to]   # to node is the second element
86
87          Z_e = R + 1j * X                # equivalent impedance
88          n_comp = n * np.exp(1j * ang1 / 180 * np.pi)  # complex turns ratio
89          Y_fr[ii, br_from[ii]] = (1 / Z_e) * (1 / abs(n_comp)**2)
90          Y_fr[ii, br_to[ii]] = -(1 / Z_e) * (1 / n_comp)
91          Y_to[ii, br_from[ii]] = -(1 / Z_e) * (1 / n_comp.conj())
92          Y_to[ii, br_to[ii]] = 1 / Z_e
93
94          Ybus[br_from[ii], br_from[ii]] += (1 / Z_e) * (1 / abs(n_comp)**2)
95          Ybus[br_from[ii], br_to[ii]] += -(1 / Z_e) * (1 / n_comp)
96          Ybus[br_to[ii], br_from[ii]] += -(1 / Z_e) * (1 / n_comp.conj())
97          Ybus[br_to[ii], br_to[ii]] += 1 / Z_e
```

7.3 Code for `DisplayResults()`

```
1  def DisplayResults(V, lnd, rnd=3, filenameBus='test1', filenameBranch='test2'):
2  '''
3  Display results in the terminal
4  Args:
5      V (np.array):         Voltage magnitudes
6      lnd ():               Load Network data object
7      filenameBus (str):    filename to save bus data in (as .csv)
8      filenameBranch (str): filename to save branch data in (as .csv)
9
10 Returns:
11     busResDF (dataFrame):    power flow results
12     branchResDF (dataFrame): branch flow results
13 '''
14
15 # initialization
16 Ybus = lnd.Ybus
17 Y_from = lnd.Y_fr
18 Y_to = lnd.Y_to
19 br_f = lnd.br_from
20 br_t = lnd.br_to
21 # buscode = lnd.buscode
22 SLD = lnd.S_LD
23 ind_to_bus = lnd.ind_to_bus
24 # bus_to_ind = lnd.bus_to_ind
```

```
# MVA_base = lnd.MVA_base
bus_labels = lnd.bus_labels

# apparent power injection at each bus
S_inj = V * (Ybus.dot(V)).conj()
# data structure for power flow results at each bus
bus_results = { 'Bus Nr.': [],
                'Bus Label': [],
                'Voltage Mag. (pu)': [],
                'Voltage Ang. (deg)': [],
                'Gen. P (pu)': [],
                'Gen. Q (pu)': [],
                'Load P (pu)': [],
                'Load Q (pu)': []
              }
# loading data into structure
for key in iter(ind_to_bus):
    bus_results['Bus Nr.'].append(ind_to_bus[key])
    bus_results['Bus Label'].append(bus_labels[key])
    bus_results['Voltage Mag. (pu)'].append(np.abs(V[key]))
    bus_results['Voltage Ang. (deg)'].append(np.angle(V[key]) * 180 / np.pi)
    # only print real (i.e. positive) generation
    bus_results['Gen. P (pu)'].append(np.real(S_inj[key]))
    bus_results['Gen. Q (pu)'].append(np.imag(S_inj[key]))
    # print(np.imag(S_inj[key]) ,np.imag(S_inj[key]) > 1e-3)
    # only print real loads
    bus_results['Load P (pu)'].append(np.real(SLD[key]))
    bus_results['Load Q (pu)'].append(np.imag(SLD[key]))

# Data frame with bus voltages/power injections
busResDF = pd.DataFrame(bus_results)

S_from = V[br_f] * (Y_from.dot(V)).conj()
S_to = V[br_t] * (Y_to.dot(V)).conj()
# data structure for branch flow results
branch_results = {  'Branch Nr.': [],
                    'From Bus': [],
                    'To Bus': [],
                    'From Bus P inj. (pu)': [],
                    'From Bus Q inj. (pu)': [],
                    'To Bus P inj. (pu)': [],
                    'To Bus Q inj. (pu)': []
                  }
# loading data into structure
for ix in range(len(br_f)):
    branch_results['Branch Nr.'].append(ix + 1)
    branch_results['From Bus'] .append(ind_to_bus[br_f[ix]])
    branch_results['To Bus'].append(ind_to_bus[br_t[ix]])
    branch_results['From Bus P inj. (pu)'].append(np.real(S_from[ix]))
    branch_results['From Bus Q inj. (pu)'].append(np.imag(S_from[ix]))
    branch_results['To Bus P inj. (pu)'].append(np.real(S_to[ix]))
    branch_results['To Bus Q inj. (pu)'].append(np.imag(S_to[ix]))

# data frame containing the branch flow information
branchResDF = pd.DataFrame(branch_results)

# prepare export
this_dir = os.getcwd()
rel_path = "outputs/"
filenameBus = f'{filenameBus}.csv'
filename1 = os.path.join(this_dir, rel_path, filenameBus)
busResDF.to_csv(filename1, index=False)

filenameBranch = f'{filenameBranch}.csv'
filename2 = os.path.join(this_dir, rel_path, filenameBranch)
branchResDF.to_csv(filename2, index=False)

# print results
print('================================================================ Bus results
```

```
    ================================================================\n')
with open(filename1) as csv1:
    busres = csv.reader(csv1, delimiter=',')
    ix = 0
    for row in busres:
        for j in range(len(row)):
            if ix == 0:
                print(f'{row[j]}  ', end='\t')
            else:
                if j == 0:
                    print(f'{int(float(row[j]))}\t', end='\t')
                elif j == 1:
                    print(f'{row[j]}\t', end='\t')
                elif j == 2:
                    print(f'{float(row[j]):.{rnd}f}   ', '\t', end='\t')
                elif j == 3:
                    print(f'{float(row[j]):.{rnd-1}f}   ', '\t', end='\t')
                else:
                    if float(row[j]) <= 1e-3:
                        print('-', '\t', end='\t')
                    else:
                        print(f'{float(row[j]):.{rnd}f}', '\t', end='\t')
        print('')
        ix = ix + 1

print('\n================================================================ Branch
    flows ================================================================\n')
with open(filename2) as csv2:
    busres = csv.reader(csv2, delimiter=',')
    ix = 0
    for row in busres:
        for j in range(len(row)):
            if ix == 0:
                print(f'{row[j]}  ', end='\t')
            else:
                if j == 0 or j == 1 or j == 2:
                    print(f'{int(float(row[j]))}', '\t', end='\t')
                else:
                    print(f'{float(row[j]):.{rnd}f}', 2 * '\t', end='\t')
        print('')
        ix = ix + 1

return busResDF, branchResDF
```

7.4 Code for `PowerFlowImplementation()`

```
import PowerFlow_46705 as pf    # import Power Flow functions
import LoadNetworkData as lnd   # load the network data to global variables
import os

max_iter = 30   # Iteration settings
err_tol = 1e-4
# Load the Network data ...
this_dir = os.getcwd()
rel_path = "text/Kundur_two_area_system.txt"
filename = os.path.join(this_dir, rel_path)
lnd.LoadNetworkData(filename)  # makes Ybus available as lnd.Ybus and etc.

# Carry out the power flow analysis ...
V, success, n = pf.PowerFlowNewton(
    lnd.Ybus, lnd.Sbus, lnd.V0, lnd.pv_index, lnd.pq_index, max_iter, err_tol)

# Display results if the power flow analysis converged
if success:
    buses, branches = pf.DisplayResults(V, lnd)  # Now we are just passing the lnd
     module as input
```