



**Department of Electrical, Computer
& Biomedical Engineering**
Faculty of Engineering & Architectural Science

BV04: Transactive Energy System in Python (Line-Wise)

by

Vineet Nischal, Mohammad Ikailan, Khalid Johar, Laeticia Osemeke

Computer/Electrical Engineering Capstone Design Project

Ryerson University, 2021

Acknowledgements

We would like to begin this report by acknowledging all the individuals and parties that made this project possible. First and foremost among them is our Faculty Lab Coordinator (FLC) and overseer Amr Mohamed. Throughout the planning, design and the implementation of the project Amr provided critical guidance and suggestions that expedited the project and smoothed any rough patches and troubles that were encountered. He was always present and active in answering any of our questions and concerns as well as provided valuable resources that made project much easier to implement. In fact, this project is deeply inspired by and could not have been possible without the program and the paper which he authored titled “Line-Wise Power Flow and Voltage Collapse”.

Our thanks are also due to the python development communities that created many of the libraries which this project depends on. They have made available extensive guides and documentations that smoothed the implementation of this project as well as clarified many issues and concerns in their vibrant and active forums. Specifically, we would like to thank the development teams behind NumPy and SciPy which allowed for the many matrix operations required for our calculations. Included in our acknowledgment are also the team behind the Streamlit package that made the construction of the website module simple and intuitive while also providing power tools that necessary for the purposes of our project.

Certification of Authorship

This is to attest that this report was authored and contributed to solely by the individuals Khalid Johar, Vineet Nischal, Mohammad Ikailan, Laeticia Osemeke. Any statement in this text or a method in our implementation that was not originally developed by us or was inspired by an external source was duly acknowledged and cited in the middle of the text as well as in the reference section. In addition, we also certify that the planning, design, implementation as well as the testing and quality check procedures were developed by the authors of this paper.

Table of Contents

Contents

Acknowledgements	2
Certification of Authorship	3
Table of Contents	4
Abstract:	5
Introduction & Background:.....	6
Objectives:	7
Theory and Design	8
Theory	8
Power flow equations derivation:	8
Newton Raphson estimation:	10
Determining buses most in danger of voltage collapse:	11
Design.....	12
Alternative Designs	14
Material/Component list	15
Measurement and Testing Procedures	16
Performance Measurement Results	19
Analysis of Performance	21
Conclusions	22
References	23
Appendices	24

Abstract:

In power system analysis, researchers' main goal is to incorporate the voltage stability with the optimal power flow (OPF). Having said that, voltage collapse is a common issue in power systems which typically happens when the reactive power demand of the load is not being met at one or more buses because of the shortage in reactive power production or transmission due to the voltage drop and heavy reactive power flow in the connected lines [1]. Bus-wise power balance equations analysis with Newton Raphson (NR) method are usually used in power system analysis to analyze the voltage collapse; however, it requires longer analysis to point out the most critical set of lines. Whereas line-wise method is twice faster than bus-wise balance equations as it does not require any additional analysis to point out the most critical set of lines. Therefore, in this project the aim is to export the algorithms provided in MATLAB for line-wise analysis using Newton Raphson and implement them in Python.

The first portion of the project was to simulate the data read and intake process which included the reading of bus data profiles retrieved from IEEE website.

The second part of the project involved a complete script for the line wise power flow equations and Jacobian matrix based on the data obtained from the data intake process file. This was followed by initialization of the variables and functions needed to construct the power flow and Jacobian matrix as seen in figure 3.1 below:

```
def LINEJAC():
    ## FUA
    LJAC[0*NT+0*NB+0:0*NT+0*NB+NT,0*NB+0*NT+0:0*NB+0*NT+NB] \
    = sparse.lil_matrix.multiply(
        MF.T, ((2*U[FBI-1,0] + 2*(Pa*RL + Qa*XL - U[TBI-1,0]/2))
              @ np.ones((1,NB))))
```

Figure 3.1: An excerpt of the function used to calculate and construct the sparse Jacobian matrix.

The third portion of the project involved completing the code for the Newton Raphson estimation and iterations in Python by cross checking the results with those obtained in the original MATLAB Code as seen in figure 3.2 below:

```
Number of Iterations: 4
Time taken: 795.6 ms
```

Figure 3.2: Newton Raphson iterations and estimation.

The last portion of the lab involved creating and presenting a final output integration of the code from previous stages to achieve the required final result, as well as creating a web interface for operating the software from heroku app server.

Introduction & Background:

In engineering, power flow analysis or the study of load flow is a powerful tool in many electrical power systems and it is usually used in power system planning and operations. The main purpose of power flow studies is to plan ahead for theoretical circumstances. For instance, if a transmission line is taken off or disconnected for support, will the remaining lines in the network handle the load without surpassing their rated values [1]. Thus, power system analysis is targeted to determine the currents, voltages at different busses, and real and reactive power flows in a network under a given load conditions. However, since operators and researchers' aim to achieve voltage stability with optimal power flow, voltage collapse is a main issue due to the voltage drop and the heavy reactive power flow in the connected lines which results in a lack of reactive power at one or more buses [1].

Having said that, bus-wise power balance equations using techniques like Newton Raphson method, the fast-decoupled methods and the Gauss-Seidel are widely used to analyse voltage collapse [2]. However, the bus-wise analysis using Newton Raphson method requires additional analysis to determine the most critical set of lines. Therefore, this project aims to use line-wise Newton Raphson method to analyze voltage collapse as it is reliable, and it determines the most critical set of lines without any additional computing as the voltage collapse index is shown for all lines in the Jacobian of the line-wise Newton Raphson method [2].

This project aims to develop a mechanism for Transactive Energy System Analysis in Python. Python (high-level programming language) is a widely used programming language due to its massive support in the area of artificial intelligence (AI), as it offers various libraries and packages that help programmers build a certain code without starting from scratch. Therefore, the intended design is to export the algorithms provided in MATLAB and implement it in Python to help analyze the voltage collapse using line-wise method, to save the operator's time and to be able point out the most critical set of lines without any additional analysis like the bus-wise method.

Objectives:

The objective of this project is to implement the line-wise method of power flow in Python. Provided with the algorithms in MATLAB, the Newton Raphson method will be used to export these algorithms and implement them in Python, as well as integrate additional input and output functionality. By implementing the line-wise analysis in Python, costs are reduced since Python is a free and open source programming language.

Theory and Design

This section explores motivating concerns for pursuing this project as well as the underlying theory on which the design will be based. A special focus will be made in this section on deriving the mathematical basis of the design; detailing all the calculations and equations which form the line-wise method for solving the power flow equations.

Theory

Preserving the stability of the system as well as the reliability of the supply are two of the most vital concerns operating and maintaining the power supply grid. A disruption in the power supply due to an instability in the system could cause the consumers, generators, as well as the utilities losses in magnitude of millions of dollars. Thus, it is paramount to ensure that the power grid is monitored for any signs of instability and to point out critical concerns so that appropriate action may be taken.

A particular phenomenon concerning power system stability which has recently become more frequent as a result of increased complexity and loading on the power grid is voltage collapse [2]. Voltage collapse is characterized by a decrease in voltage due to insufficient injection of reactive power at a particular bus. This phenomenon has been cited as a contributing factor to a number of large-scale blackouts in Europe and America including the infamous northeastern blackout of 2003 [3][4]. As a result, determining analytic methods of predicting and identifying lines and buses most in danger of voltage collapse is critical for maintaining stability of the power system.

An analytic method which readily provides relevant information concerning voltage stability is the line-wise power flow method. The line-wise method is an alternative solution technique for the analysis of power flow in a distribution network, similar to the more widespread and well-known bus wise method. The main advantage of the line-wise method is that it readily and more provides information concerning risk of voltage collapse compared to the bus-wise method without additional computation [1].

Power flow equations derivation:

The general procedure for the line-wise power flow method between any two buses, based on ref. [5], begin as follows:

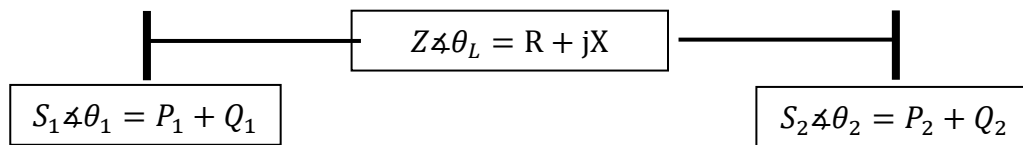


Figure 7.1.1: A representative model to illustrate the variables of interest for the power flow analysis.

Considering bus 1, the following is obtained:

$$V_1 \angle \theta_{v1} - V_2 \angle \theta_{v2} = \left(\frac{S_1 \angle \theta_1}{V_2 \angle \theta_{v2}} \right)^* Z \angle \theta_L \quad (1)$$

$$V_1 \angle \theta_{v1} (V_2 \angle -\theta_{v2}) - V_2^2 = (S_1 \angle -\theta_1) Z \angle \theta_L \quad (2)$$

By separating the real and imaginary components:

$$V_1 V_2 \cos(\theta_{v1} - \theta_{v2}) = P_2 R + Q_2 X + V_2^2 \quad (3)$$

$$V_1 V_2 \sin(\theta_{v1} - \theta_{v2}) = P_2 X - Q_2 R \quad (4)$$

Squaring and summing the above equations yields:

$$V_2^4 + 2V_2^2 \left(P_2 R + Q_2 X - \frac{V_1^2}{2} \right) + S_2^2 Z^2 = 0 \quad (5)$$

Which becomes can be converted to a quadratic by considering $U_2 = (V_2)^2$

$$PF_1 = U_2^2 + 2U_2 \left(P_2 R + Q_2 X - \frac{U_1}{2} \right) + S_2^2 Z^2 = 0 \quad (6)$$

Additionally, if the same equations squared and summed previously were divided:

$$PF_2 = (P_2 R + Q_2 X + V_2^2) \tan(\theta_{v1} - \theta_{v2}) - P_2 X + Q_2 R = 0 \quad (7)$$

Repeating the same steps when considering bus 2, the following is obtained:

$$PF_3 = U_1^2 + 2U_1 \left(P_1 R + Q_1 X - \frac{U_2}{2} \right) + S_1^2 Z^2 = 0 \quad (8)$$

$$PF_4 = (P_1 R + Q_1 X + V_1^2) \tan(\theta_{v2} - \theta_{v1}) - P_1 X + Q_1 R = 0 \quad (9)$$

Considering the power balance equation:

$$PF_5 = PD - PG = [M] \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} - U \cdot GS \quad (10)$$

$$PF_6 = QD - QG = [M] \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} - U \cdot BS \quad (11)$$

Where GS and BS represent the shunt conductance and susceptance of the line losses in the pi-model. While M is a defined matrix based on the concept of bus incidence with the dimensions NB x 2NT, representing the number of buses and transmission branches, respectively. The matrix M has the following properties:

$$[M]_{x,l} = 1$$

If the bus x is either the first or second bus for any line l , otherwise, its values is zero. The power flow equations derived previously can be summarized as:

$$\begin{bmatrix} PF_1(W) \\ PF_2(W) \\ PF_3(W) \\ PF_4(W) \\ PF_5(W) \\ PF_6(W) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ PD - PG \\ QD - QG \end{bmatrix} \quad (12)$$

Or in the compact form below:

$$PF(W) = SV \quad (13)$$

Where W is the set of parameters $[U \ \theta \ P_1 \ P_2 \ Q_1 \ Q_2]$ and SV is the solution vector. The above equations are used to determine the unknown parameters of a particular bus. The following tables provides information concerning the known and unknown variables based on the type of the bus:

Table 7.1.1: Known and unknown parameters of the respective bus type

	Knowns	Unknowns
Slack Bus	$V = 1, \theta = 0$	$P = PG-PD,$ $Q = QG-QD$
Generator (PV) Bus	$V, P = PG-PD$	$\theta, Q = QG-QD$
Load Bus (PQ) BUS	$P = -PD, Q = -QD$	V, θ

Newton Raphson estimation:

It should be noted that nonlinearities exist in the set power flow equations as is clear from the equations derived previously, which makes it difficult to determine the unknown variables. However, this issue could be bypassed using the Newton-Raphson estimation technique which produces at the unknown parameters through a series of iterative estimations. The estimation process begins by making an initial guess about the parameters of the system and comparing the corresponding estimated results to the given and known parameters. The conventional initial estimation is to use the following parameters for all buses:

$$W_0: V = 1, \theta = 0 \quad (14)$$

The solution is considered accurate when corresponding estimated output based on the estimated parameters is within an accepted range of the actual output, usually less than 10^{-5} . Stated differently:

$$|SV - PF(W_n)| \leq \epsilon \quad (15)$$

If the above condition is not satisfied, a new set of parameters are generated based on factors including the previous estimated parameters, the discrepancy with the accepted values as well as the Jacobian matrix of the power flow equations. The Jacobian matrix is a matrix composed of the set first order partial derivatives of the power flow equations (PF) with respect to the set of parameters(W). Which can be shown using the compact form below:

$$\frac{\partial PF(W_n)}{\partial W_n} \quad (16)$$

Where the expanded form is:

$$\frac{\partial PF(W_n)}{\partial W_n} = \begin{bmatrix} \frac{\partial PF_1}{\partial U} \frac{\partial PF_1}{\partial \theta} \frac{\partial PF_1}{\partial P_1} \frac{\partial PF_1}{\partial P_2} \frac{\partial PF_1}{\partial Q_2} \frac{\partial PF_1}{\partial Q_2} \\ \frac{\partial PF_2}{\partial U} \frac{\partial PF_2}{\partial \theta} \frac{\partial PF_2}{\partial P_1} \frac{\partial PF_2}{\partial P_2} \frac{\partial PF_2}{\partial Q_2} \frac{\partial PF_2}{\partial Q_2} \\ \frac{\partial PF_3}{\partial U} \frac{\partial PF_3}{\partial \theta} \frac{\partial PF_3}{\partial P_1} \frac{\partial PF_3}{\partial P_2} \frac{\partial PF_3}{\partial Q_2} \frac{\partial PF_3}{\partial Q_2} \\ \frac{\partial PF_4}{\partial U} \frac{\partial PF_4}{\partial \theta} \frac{\partial PF_4}{\partial P_1} \frac{\partial PF_4}{\partial P_2} \frac{\partial PF_4}{\partial Q_2} \frac{\partial PF_4}{\partial Q_2} \\ \frac{\partial PF_5}{\partial U} \frac{\partial PF_5}{\partial \theta} \frac{\partial PF_5}{\partial P_1} \frac{\partial PF_5}{\partial P_2} \frac{\partial PF_5}{\partial Q_2} \frac{\partial PF_5}{\partial Q_2} \\ \frac{\partial PF_6}{\partial U} \frac{\partial PF_6}{\partial \theta} \frac{\partial PF_6}{\partial P_1} \frac{\partial PF_6}{\partial P_2} \frac{\partial PF_6}{\partial Q_2} \frac{\partial PF_6}{\partial Q_2} \end{bmatrix} \quad (17)$$

The above Jacobian matrix is then used to determine and obtain the updated parameter estimation as was stated previously based on the general form of iterative estimate in the NR method based as follows:

$$W_{n+1} = W_n - \frac{SV - PF(W_n)}{\left(\frac{\partial PF(W_n)}{\partial W_n}\right)} \quad (18)$$

Where W_n and W_{n+1} represents the old and updated estimated parameters, respectively.

Determining buses most in danger of voltage collapse:

The advantage that the line-wise method provides is that information pertaining voltage stability can be extracted without additional computation. The following elements of the Jacobian matrix indicate whether a line is at risk of voltage collapse or not:

$$VCI_1 = \frac{\partial PF_3}{\partial U_1} = 2U_1 + 2\left(P_1R + Q_1X - \frac{U_2}{2}\right)W_{n+1} = W_n - \frac{SV - PF(W_n)}{\left(\frac{\partial PF(W_n)}{\partial W_n}\right)} \quad (19)$$

$$VCI_2 = \frac{\partial PF_1}{\partial U_2} = 2U_2 + 2\left(P_1R + Q_1X - \frac{U_1}{2}\right) \quad (20)$$

For a given load, the buses that are most at risk of voltage collapse exhibit the lowest value for the voltage collapse index. In other words, the magnitude of the Jacobian elements above, will be smallest for the most susceptible bus compared to the other bus.

Design

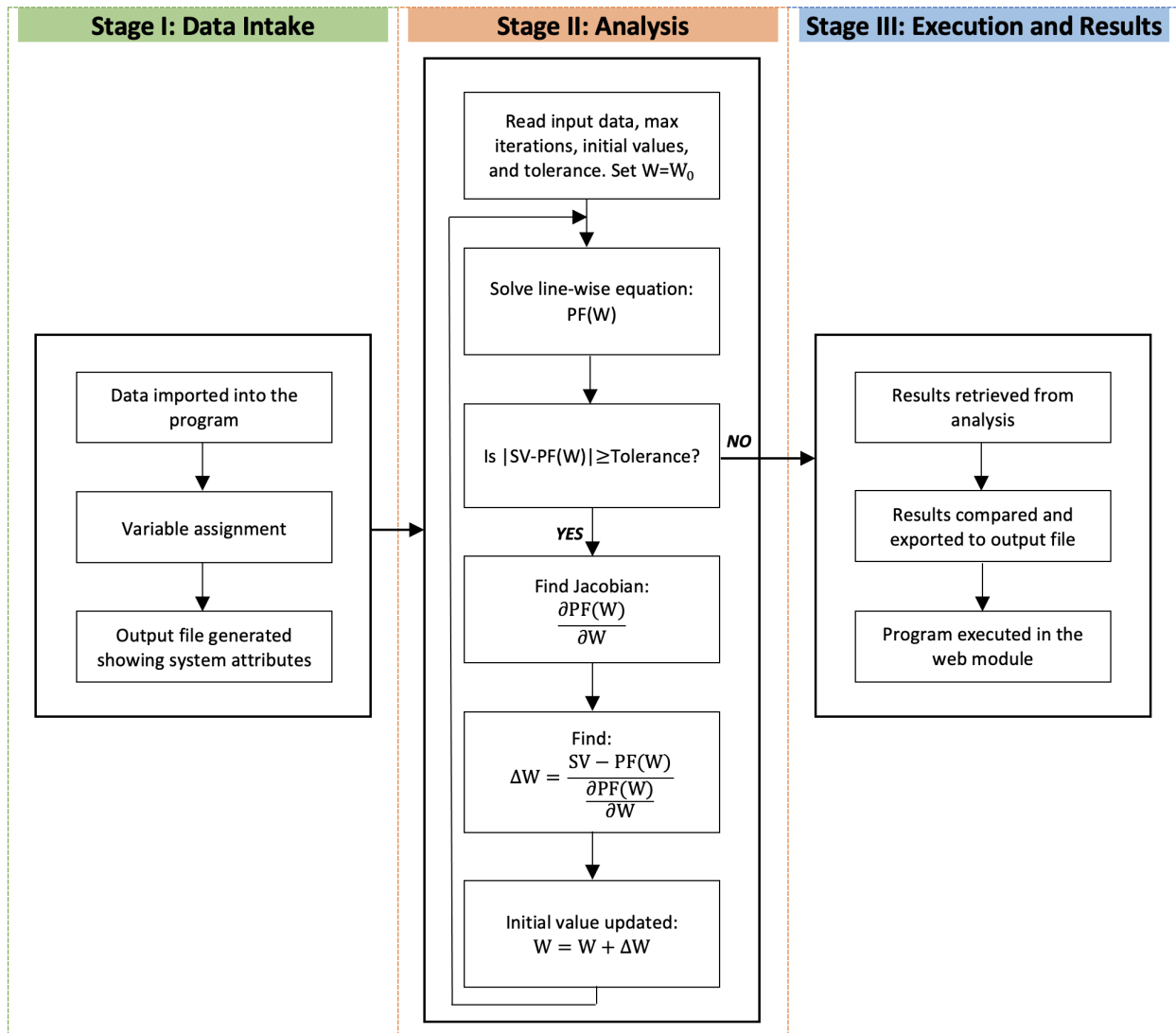


Figure 7.2.1: A representative model to illustrate the variables of interest for the power flow analysis. Complete block diagram of the project.

Stage 1: Data Intake

The first stage of the design is to intake data for analysis in Python. The data in our case is an n-bus system retrieved from IEEE database, where n is the number of busses. The code

for this section is executed in the `data_intake.py` file, which carries out the variable assignment. Variable assignment allows the system data to be used in the analysis section. Another aspect of this stage is to create an output file that portrays the raw data from IEEE in a more readable fashion. This is done in the same `data_intake.py` script.

Stage 2: Analysis

The second stage consists of the algorithm for NR technique that is used to solve the system in question. The steps followed in a standard bus-wise Newton-Raphson Method for power flow analysis are very similar to that of a line-wise system. The only difference is the set of line-wise power balance equations to be solved and lack of the bus-wise admittance matrix. The block diagram shown above outlines the steps followed for the proposed technique, which are explained in detail in the theory section.

The known and unknowns for the problems are summarized in Table 1. Depending on the system in question and type of buses being used, the algorithm will generate results described below [4]:

- The squares of voltage magnitudes at load buses
- Voltage phase angles at all buses except the slack bus
- Real and reactive powers for sending and receiving ends

Stage 3: Execution

The execution stage consists of displaying the analysis results and using the results to analyse the effectiveness of our design. This will include the number of iterations, time spent on processing each iteration, and the roots, which in this case are the unknowns depending on the problem as discussed in stage 2. The results obtained are exported to an output file, the process of which is similar to one discussed in stage 1.



Figure 7.2.2: Flowchart of the software.

The IDE/Software used to program is Spyder. The primary reasons for this choice are the ease of access and the fact that Spyder is open-source. This will also prove as a selling factor for this project as it will attract potential users of this system. Figure 7.2.2 shows the tasks to be carried out using Python. Firstly, the instructions entered from the system data are interpreted by the algorithm. Instructions are interpreted as the known and unknown parameters for the NR algorithm, which are then analyzed by the program. After running the code, results are printed on the screen and exported to an output file. The final section of this project was to implement a web module. This requirement was fulfilled by using Streamlit Python library and Heroku cloud platform for creating web applications. As part of the web module, our program can be run via a website and generate results in a similar manner as it would when run locally using an IDE.

Alternative Designs

Alternative designs are key to the execution of every project and without one, a project can result in a failure or be drastically less efficient. An alternative design was prepared for each stage discussed in the design section above.

As noted above, stage 1 consisted primarily of the data intake process and used the system data, retrieved from IEEE database, directly into the program. An alternative to using this method was to convert the system data to an excel file first and then reading the data in `data_intake.py` script. In addition, a similar approach was used for the output file generated by the script, that is, to export the data as an excel file. Although this approach of creating an excel file was feasible, it did not serve much importance as our project requirement.

Analysis using the Newton Raphson technique was carried out in stage 2. The line wise power flow method provides relevant information concerning voltage stability. An alternative to this is the widespread and well-known bus-wise method. The major difference between these methods is the access to voltage collapse information. The line-wise method readily and more efficiently provides information concerning risk of voltage collapse compared to the bus wise method without additional computation.

Stage 3 included the execution of the program. The major task outlined in this section was to create a web module for the project. There are many tools available to achieve this and therefore, we had many options to choose from. As mentioned above, we used Heroku and Streamlit to create our final web module, however, an alternative to this was to use the Github Pages and TrackVia. TrackVia provided as many, if not more, functionalities for our project, however, it was a paid service and therefore, we decided to move forward with the Streamlit library. Additionally, Github pages are very similar to Heroku in terms of its functionality. Both of these tools are a cloud platform that are used to create web applications. Choosing either option would have sufficed our requirement, however, we used Heroku due to our previous experience with the platform.

Material/Component list

This project had no use for any physical materials or components, however, the software used in meeting the requirements are tabulated below.

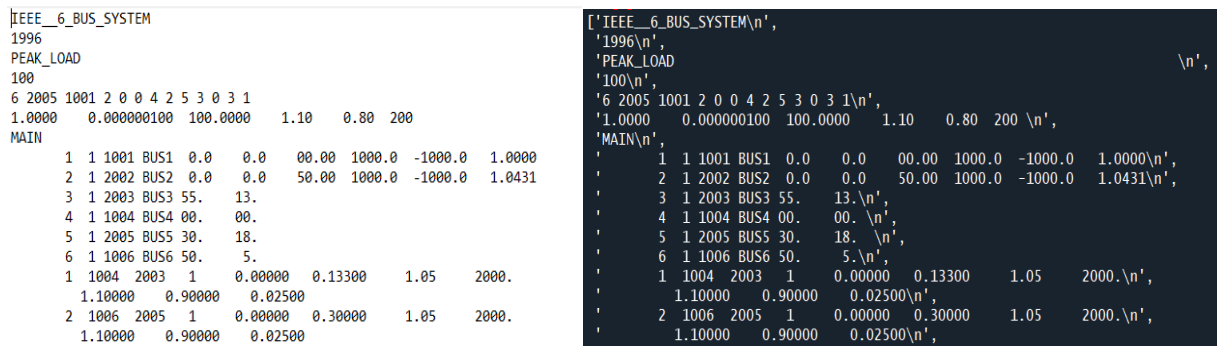
Table 9.1: Table of software used in the project, quantity and unit cost

Materials / Component	Quantity	Unit Cost (CAD)
Python programming software	N/A	0
Wix.com website builder	N/A	0
MATLAB	N/A	0
Website module hosted by Heroku	N/A	0

Measurement and Testing Procedures

To confirm the accuracy and precision of the resulting calculations of the python script developed, multiple methods of testing and measured were employed. To ensure that the intended process is being followed in Python with no errors at every stage, different testing procedures were used for each stage.

In the initial stage, where the input file is read and converted to a suitable format for the process, the first concern was confirming the input file is being in fact read. This was confirmed by iteratively going through the input and checking line by line to be compared with the original as can be seen below.



```

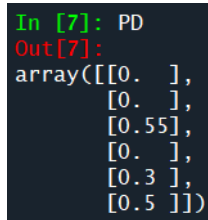
IEEE__6_BUS_SYSTEM
1996
PEAK_LOAD
100
6 2005 1001 2 0 0 4 2 5 3 0 3 1
1.0000 0.0000000100 100.0000 1.10 0.80 200
MAIN
1 1 1001 BUS1 0.0 0.0 00.00 1000.0 -1000.0 1.0000
2 1 2002 BUS2 0.0 0.0 50.00 1000.0 -1000.0 1.0431
3 1 2003 BUS3 55. 13.
4 1 1004 BUS4 00. 00.
5 1 2005 BUS5 30. 18.
6 1 1006 BUS6 50. 5.
1 1004 2003 1 0.00000 0.13300 1.05 2000.
1.10000 0.90000 0.02500
2 1006 2005 1 0.00000 0.30000 1.05 2000.
1.10000 0.90000 0.02500

['IEEE__6_BUS_SYSTEM\n',
 '1996\n',
 'PEAK_LOAD\n',
 '100\n',
 '6 2005 1001 2 0 0 4 2 5 3 0 3 1\n',
 '1.0000 0.0000000100 100.0000 1.10 0.80 200\n',
 'MAIN\n',
 '1 1 1001 BUS1 0.0 0.0 00.00 1000.0 -1000.0 1.0000\n',
 '2 1 2002 BUS2 0.0 0.0 50.00 1000.0 -1000.0 1.0431\n',
 '3 1 2003 BUS3 55. 13.\n',
 '4 1 1004 BUS4 00. 00.\n',
 '5 1 2005 BUS5 30. 18.\n',
 '6 1 1006 BUS6 50. 5.\n',
 '1 1004 2003 1 0.00000 0.13300 1.05 2000.\n',
 '1.10000 0.90000 0.02500\n',
 '2 1006 2005 1 0.00000 0.30000 1.05 2000.\n',
 '1.10000 0.90000 0.02500\n']

```

Figure 10.1: an excerpt of the input on the right compared to what is being read in python on the left. Note the inclusion of special character such as ‘\n’.

Once all anomalies have been pointed out, such as ‘\n’ above for example, and dealt with accordingly the contents of the input are copied to prepared variables to be used in the calculation in the next stages. To confirm that the intended values were copied correctly, the contents of the variables were manually checked as seen in the figures below.



```

In [7]: PD
Out[7]:
array([[0. ],
       [0. ],
       [0.55],
       [0. ],
       [0.3 ],
       [0.5 ]])

```

Figure 10.2: an example of verifying the contents of a variable. The contents of the load power variable are being verified in this case.

In the second stage which involved the calculation of line-wise power flow equations as well as the Jacobian matrix a different method was used to verify the accuracy of the program. Due to the sheer size of the Jacobian matrix and its solution vector, it is difficult to manually calculate the correct values and compare them with the observed results in the program. Thus, a different

method was employed. Since the python script which the team developed was based on an original MATLAB script, the result of the Jacobian matrix and the power flow equation of the two programs were compared.

```
>> full(LJAC)

ans =
Columns 1 through 13

      0      0 -1.0000    1.0000      0
      0      0      0      0 -1.0000
    1.0000      0      0      0      0
    1.0000      0      0 -1.0000      0
      0      0      0    1.0000      0
      0 -1.0000      0      0  0.9119
      0  1.1761 -1.0881      0      0
      0      0    1.0000 -1.0000      0

In [27]: LJAC.toarray()[:8,:5]
Out[27]:
array([[ 0.,      0.,     -1.,      1.,      0.],
       [ 0.,      0.,      0.,      0.,     -1.],
       [ 1.,      0.,      0.,      0.,      0.],
       [ 1.,      0.,      0.,     -1.,      0.],
       [ 0.,      0.,      0.,      1.,      0.],
       [ 0.,     -1.,      0.,      0.,  0.91194239],
       [ 0.,  1.17611522, -1.08805761,  0.,      0.],
       [ 0.,      0.,      1.,     -1.,      0.]])
```

Figure 10.3: verifying the results of the Jacobian matrix by comparing with the results from the original program.

Similar testing procedure was employed in the third stage since the Newton-Raphson method is an iterative estimation technique which depends on the results of the previous iteration to create the new estimate which requires tedious calculations if done manually. However, since both programs use the same NR technique, the same estimation error is expected to be observed for both programs. Thus, by comparing the error vectors as seen below, it is possible quickly and accurately confirm the results of the developed python script.

```
>> ERROR'

ans =
    0.5500
    0.0620
    0.0047
    0.0000
    0.0000

In [37]: ERROR.tolist()
Out[37]:
[0.55,
 0.06202772162194902,
 0.004663070591071683,
 4.647827039501451e-05,
 5.208591717653643e-09]
```

Figure 10.4: verifying the results of the error vector by comparing with the results from the original program.

Finally, to ensure that the intended variables and values are being presented in output of the program, key parameters of the network such as the bus voltage and power are manually checked as in with the intake process. The script was tested again using more input files and testing scenarios using the above verification procedure to ensure the accuracy and the precision of the code developed.

Performance Measurement Results

Using the measurement procedure outlined above for the 6-bus network case, the following results were obtained. Note that the computation time was also included in the measurement of the performance to provide insight on the quality of the process used.

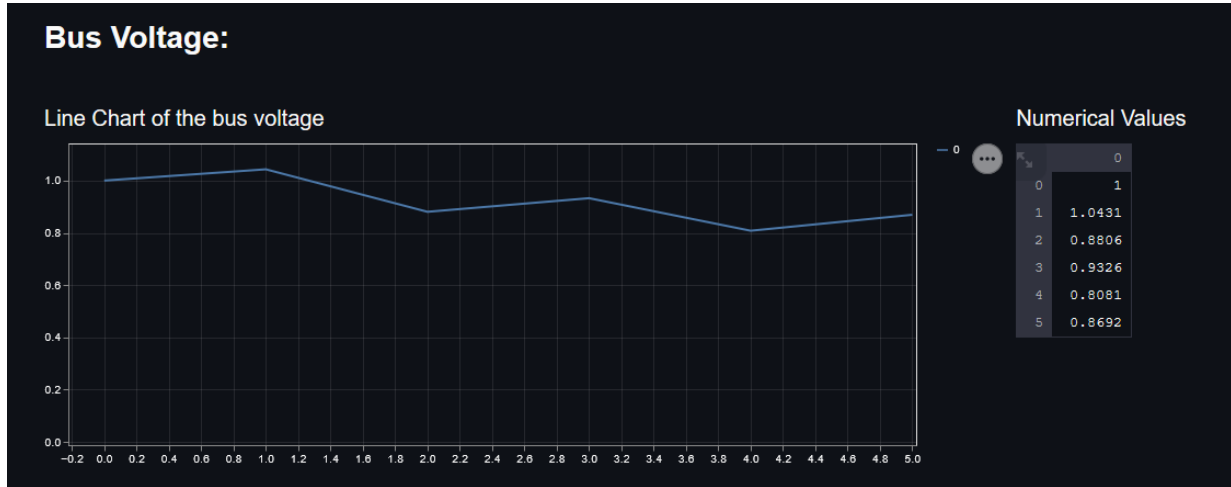


Figure 11.1: Bus voltage of the 6-bus network case.

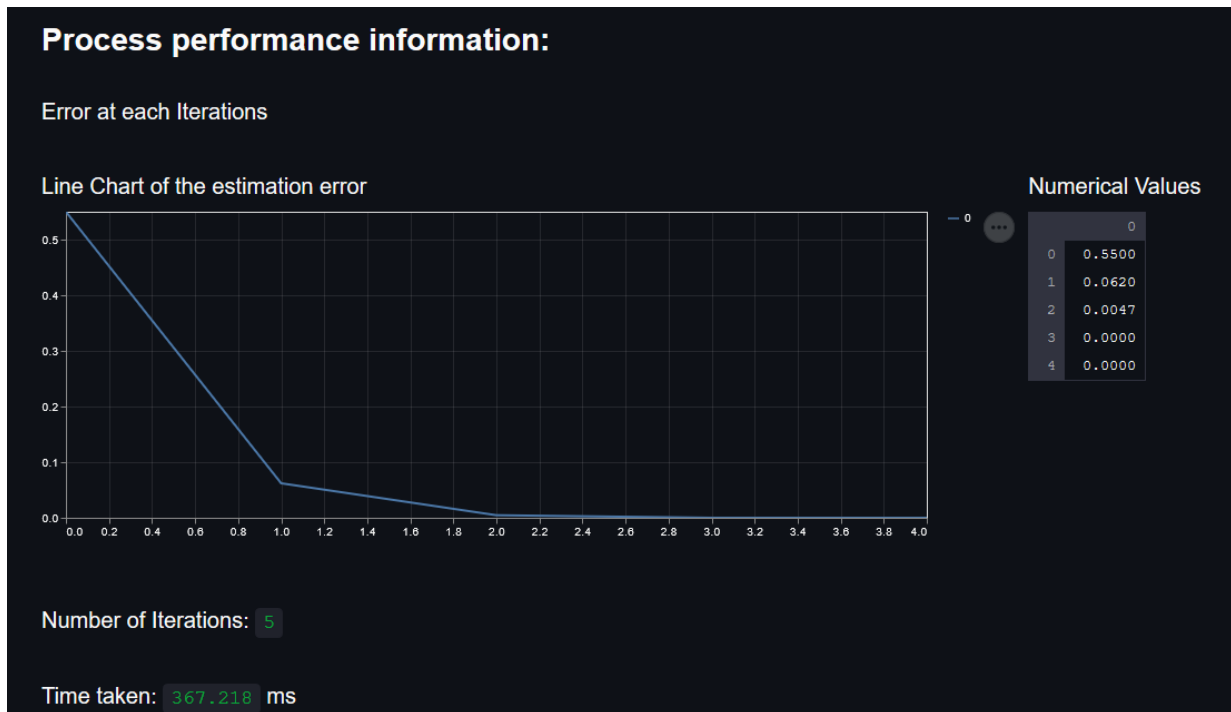


Figure 11.2: plot of the estimation error for the 6-bus network case.

```
SUMMARY OF THE RESULTS
"-----
"
-----
TOTAL REAL GENERATION      =      147.25953
TOTAL REAL LOAD            =      135.00000
TOTAL REAL LOSS            =       12.25953
-----
TOTAL REACTIVE GENERATION =       62.11281
TOTAL REACTIVE LOAD       =       36.00000
TOTAL REACTIVE LOSS       =       40.44338
-----
TRANSMISSION LOSS         =       12.25953
-----
```

Figure 11.3: Summary of the results for the 6-bus case using the Line-wise PF method.

The above results are generally consistent with the results observed in the original MATLAB code. In fact, all values were observed to be identical which confirms that python script developed accurately calculates the parameters of interest.

```
SUMMARY OF THE RESULTS
-----
-----
TOTAL REAL GENERATION      =      147.25953
TOTAL REAL LOAD            =      135.00000
TOTAL REAL LOSS            =       12.25953
-----
TOTAL REACTIVE GENERATION =       62.11281
TOTAL REACTIVE LOAD       =       36.00000
TOTAL REACTIVE LOSS       =       40.44338
-----
TRANSMISSION LOSS         =       12.25953
-----
,
```

Figure 11.4: Original Summary of the results for the 6-bus case from the MATLAB code.

Analysis of Performance

Figure 11.1 illustrates the bus voltage of the 6-bus network case. In order to test the validity of obtained results for this case, different tests were performed with different cases and all the results obtained were consistent with the results observed in the original MATLAB code. This confirms the accuracy of the developed Python scripts and the overall project.

In figure 11.2, the estimation error of this system is illustrated for five iterations. The purpose of this error estimation is to demonstrate the efficiency of the line-wise method. While examining the figure, it can be observed that this method takes four milliseconds to perform five iterations. While this is the same result as the traditional bus-wise method in the case of the 6-bus network, more tests were performed with bigger buses and it was observed that as the number of busses increased, the line-wise method took less time to execute the same, if not more iterations than the traditional bus-wise method. This, among other reasons, is why the line-wise method is preferred over the traditional bus-wise method.

Conclusions

The objective of this project was to develop a mechanism for Transactive Energy System Analysis in Python. The design solution was to export the provided algorithms from MATLAB and implement it in Python to analyze voltage collapse using the line-wise Newton Raphson method. The project was implemented in four stages. The first stage was to intake data, which was an n-bus system retrieved from IEEE database for analysis in Python, where n is the number of busses. The code for this stage was implemented in the `data_intake.py` file, which carries out the variable assignment, allowing for the raw data to be useful in the analysis section. An output file was also created to portray the raw data from IEEE to be more understandable, and this is done in the `data_intake.py` file as well. The second stage was to implement the algorithm for NR technique used to solve the system. The third stage was to show the results of the algorithm and use the results to analyze the accuracy of the design. The final stage was to export the results and analysis of the design. This code for this stage was implemented in the `output.py` file, which displays the results of the design in a comprehensible form.

There are no discrepancies between the initial objectives and what was accomplished. In addition to meeting the objectives, a website interface was created for operating the software from the cloud, demonstrating the key components of the project. This was executed by first researching different methods for writing the code for the interface. Streamlit was the preferred choice due to its use of Python libraries and its efficiency in turning data scripts into web applications. The Heroku cloud platform was also used when creating the interface. A ‘dummy’ website was then created to test the functionalities and limitations of the libraries. After tests had been run, the design layout was finalized and the code was compiled into a single file and uploaded to the website. The website was also designed to let the user generate and download the output file.

This project was focused on handling 6-bus systems in Python. In light of its success, it can be taken even further to handle much bigger systems. During testing procedures, systems of sizes 33, 181 and 191 were used, and more in depth research has shown that the line-wise method is capable of handling up to 9241-bus systems. Taking this into account alongside its speed, cost effectiveness and accuracy, the line-wise method can be adopted to handle even much bigger systems. This could mean a town, a district or even a city.

Not only is the line-wise NR method fast for analysis of power flow, it can also be used as an online tool to assess the stability of voltage across power lines. Other benefits of adopting this method include its ability to provide voltage collapse index (VCI) values without further computation, no complexity when setting its Jacobian, accurate solutions and ability to handle much larger systems [1].

References

- [1] Dobson, H. Glavitsch, C. -. Liu, Y. Tamura and K. Vu, "Voltage collapse in power systems," in IEEE Circuits and Devices Magazine, vol. 8, no. 3, pp. 40-45, May 1992, doi: 10.1109/101.136788.
- [2] A. Adel Mohamed and B. Venkatesh, "Line-Wise Power Flow and Voltage Collapse," in IEEE Transactions on Power Systems, vol. 33, no. 4, pp. 3768-3778, July 2018, doi: 10.1109/TPWRS.2017.2787752.
- [3] Simpson-Porco, J., Dörfler, F. & Bullo, F. Voltage collapse in complex power grids. Nat Commun 7, 10790 (2016). <https://doi.org/10.1038/ncomms10790>
- [4] Machowski, Jan Bialek, Janusz W. Bumby, James R.. (2008). Power System Dynamics - Stability and Control (2nd Edition). (pp. 326). Wiley - IEEE Press. Retrieved from <https://app.knovel.com/hotlink/toc/id:kpPSDSCE01/power-system-dynamics/powersys-tem-dynamics>

Appendices

Nomenclature:

PF – Power Flow

NR – Newton Raphson SLP –
Service Location Protocol

IDE – Integrated Development Environment

OPF – Optimal Power Flow

List of Tables:

Table 6.1.1: Known and unknown parameters of the respective bus type

Table 9.1: Table of software used in the project, quantity and unit cost

List of Figures:

Figure 7.2.1: An excerpt of the function used to calculate and construct the sparse Jacobian matrix.

Figure 6.2.2: Newton Raphson iterations and estimation.

Figure 7.2.1: A representative model to illustrate the variables of interest for the power flow analysis. Complete block diagram of the project.

Figure 7.2.2: Flowchart of the software.

Figure 10.1: An excerpt of the input on the right compared to what is being read in python on the left. Note the inclusion of special characters such as ‘\n’.

Figure 10.2: An example of verifying the contents of a variable. The contents of the load power variable are being verified in this case.

Figure 10.3: Verifying the results of the Jacobian matrix by comparing with the results from the original program.

Figure 10.4: Verifying the results of the error vector by comparing with the results from the original program.

Figure 11.1: Bus voltage of the 6-bus network case.

Figure 11.2: Plot of the estimation error for the 6-bus network case.

Figure 11.3: Summary of the results for the 6-bus case using the Line-wise PF method.

Figure 11.4: Original Summary of the results for the 6-bus case from the MATLAB code.

Program Script:

```
import streamlit as st
import pandas as pd
import numpy as np
from math import cos,sin, pi
import time
import base64
from scipy import sparse
from scipy.sparse.linalg import spsolve

def input_process(infile):

    global NB, NBB, NS, NG, NLB, NTR, NTRL, NT, NSHC, NSVS, NSHR, NSH, NREG
    global VSLACK, TOLER, PBASE, VLMAX, VLMIN, ITMAX
    global BIND, BSN, BNAM, PG, QG, PD, QD, V, DEL, QGMAX, QGMIN, VSH
    global FB, TB, NCKT, YL, ZL, BL, TAP, RAT, KV, LEN, TAPMAX, TAPMIN, TAPSTP
    global SNO, SUS, SUSMAX, SUSMIN, SUSSTP
    global YB
    global LAM
    global PGMAX, PGMIN, VGMAX, VGMIN, ap, bp, cp
    global raw_text, lines, elements, output

    raw_text = str(input_file.read(),"utf-8")
    lines = raw_text.split("\n");
    lines = list(filter(lambda x: not x.isspace(), lines));
    lines = list(filter(None, lines))

    line0 = lines[0];
    line1 = lines[1];
    line2 = lines[2];
    line3 = lines[3];
    line4 = lines[4];
    line5 = lines[5];

    ttt1 = format(line4.strip());
    ttt2 = ttt1.split();
    ttt = list(map(int, ttt2));

    NB = ttt[0];
```

```
NBB = ttt[1];
NS = ttt[2];
NG = ttt[3];
temp = ttt[4];
temp = ttt[5];
NLB = ttt[6];
NTR = ttt[7];
NTRL = ttt[8];
NSHC = ttt[9];
NSVS = ttt[10];
NSHR = ttt[11];
NREG = ttt[12];

NT = NTR + NTRL;
NSH = NSHC + NSVS + NSHR;

ttt1 = format(line5.strip());
ttt2 = ttt1.split();
ttt = list(map(float, ttt2));

VSLACK = ttt[0];
TOLER = ttt[1];
PBASE = ttt[2];
VLMAX = ttt[3];
VLMIN = ttt[4];
ITMAX = int(ttt[5]);

BSN = np.zeros((NB,1),dtype=int);
BNAM = list(map(str,np.arange(NB)));
PD = np.zeros((NB,1),dtype=float);
QD = np.zeros((NB,1),dtype=float);
PG = np.zeros((NB,1),dtype=float);
QG = np.zeros((NB,1),dtype=float);
QGMAX = np.zeros((NB,1),dtype=float);
QGMIN = np.zeros((NB,1),dtype=float);
VSH = np.zeros((NB,1),dtype=float);

V = np.ones((NB,1));
DEL = np.zeros((NB,1));

BSNMAX = 0;

st.write("### **Case information:**")
st.write("### Input File Name:", infile.name)
st.write("SYSTEM : ", format(line0.strip()))
st.write("YEAR : ", format(line1.strip()))
```

```

st.write("""CASE : """,format(line2.strip()));
st.write("""NUMBER : """,format(line3.strip()));
st.write("""NUMBER OF BUSES : """, str(NB));
st.write("""SLACK BUS NUMBER : """, str(NS));
st.write("""NUMBER OF GENERATORS : """, str(NG));
st.write("""NUMBER OF LOAD BUSES : """, str(NLB));
st.write("""NUMBER OF TRANSFORMERS : """, str(NTR));
st.write("""NUMBER OF TRANSMISSION LINES : """, str(NTRL));
st.write("""NUMBER OF SHUNT CAPACITORS : """, str(NSHC));
st.write("""NUMBER OF SWITCHABLE CAPACITORS : """, str(NSVS));
st.write("""NUMBER OF SHUNT REACTORS : """, str(NSHR));

```

for the output file

```

output = ["INPUT FILE NAME: " + infile.name]
output.append("OUTPUT FILE NAME: ")
output.append("SYSTEM: {}".format(line0.strip()))
output.append("YEAR: {}".format(line1.strip()))
output.append("CASE: {}".format(line2.strip()));
output.append("NUMBER: {}".format(line3.strip()));
output.append("NUMBER OF BUSES : " + str(NB));
output.append("SLACK BUS NUMBER : " + str(NS));
output.append("NUMBER OF GENERATORS : " + str(NG));
output.append("NUMBER OF LOAD BUSES : " + str(NLB));
output.append("NUMBER OF TRANSFORMERS : " + str(NTR));
output.append("NUMBER OF TRANSMISSION LINES : " + str(NTRL));
output.append("NUMBER OF SHUNT CAPACITORS : " + str(NSHC));
output.append("NUMBER OF SWITCHABLE CAPACITORS : " + str(NSVS));
output.append("NUMBER OF SHUNT REACTORS : " + str(NSHR));
output.append("SLACK BUS VOLATGE : {:.4f}".format(VSLACK));
output.append("TOLERANCE (MW) : {:.4f}".format(TOLER*PBASE));
output.append("BASE (MVA) 60 : {:.4f}".format(PBASE));
output.append("MINIMUM LOAD BUS VOLTAGE : {:.4f}".format(VLMIN));
output.append("MAXIMUM LOAD BUS VOLTAGE : {:.4f}".format(VLMAX));
output.append("MAXIMUM NUMBER OF ITERATIONS : " + str(ITMAX));
output.append("DETAILED OUTPUT IN PER UNIT ");
output.append("GENERATOR BUS DATA");
output.append("SNO R.NO B.NO BUS NAME ---PG--- ---PD--- ---QD--- --QGMX--- ---
QGMN--- --V SH--");

```

```

for k in range(0, NG):
    line = list(map(str, (format(lines[7+k])).split()));
    BSN[k] = int(line[2]);

```

```

BNAM[k] = line[3];
PD[k] = float(line[4]);
QD[k] = float(line[5]);
PG[k] = float(line[6]);
QG[k] = 0;
QGMAX[k] = float(line[7]);
QGMIN[k] = float(line[8]);
VSH[k] = float(line[9]);
V[k] = VSH[k];

if BSNMAX < BSN[k]:
    BSNMAX = BSN[k];

output.append(" " + str(k+1)+" " + str(k+1)+" " + format(BSN[k])+" "+\
    str(BNAM[k])+str("%8.4f" %PG[k])+str("%8.4f" %PD[k])\
    +str("%8.4f" %QD[k])+str("%8.4f" %QGMAX[k])+ \
    str("%8.4f" %QGMIN[k]) +str("%8.4f" %VSH[k]));

PD[k] = PD[k]/PBASE;
QD[k] = QD[k]/PBASE;
PG[k] = PG[k]/PBASE;
QG[k] = QG[k]/PBASE;
QGMAX[k] = QGMAX[k]/PBASE;
QGMIN[k] = QGMIN[k]/PBASE;

output.append("LOAD BUS DATA");
output.append("SNO R.NO  B.NO BUS NAME ---PD--- ---QD---");

for k in range(NG, NB):
    line = list(map(str, (format(lines[7+k])).split()));
    BSN[k] = int(line[2]);
    BNAM[k] = line[3];
    PD[k] = float(line[4]);
    QD[k] = float(line[5]);

    if BSNMAX < BSN[k]:
        BSNMAX = BSN[k];

    output.append(" " + str(k+1)+" " + str(k+1)+" " + format(BSN[k])+" "+\
        str(BNAM[k])+str("%8.4f" %PD[k])+str("%8.4f" %QD[k]))

    PD[k] = PD[k]/PBASE;
    QD[k] = QD[k]/PBASE;

BIND = np.zeros((int(BSNMAX),1));

```

```
for k in range(0,NB):
    BIND[BSN[k]-1] = k+1;

ZL = (0+0j) * np.zeros((NT,1));
YL = (0+0j) * np.zeros((NT,1));
BL = (0+0j) * np.zeros((NT,1));

FB = np.zeros((NT,1), dtype=int);
TB = np.zeros((NT,1), dtype=int);
NCKT = np.zeros((NT,1), dtype=int);
TAP = np.ones((NT,1));
RAT = np.zeros((NT,1));
KV = np.zeros((NT,1));
LEN = np.zeros((NT,1));

TAPMAX = np.ones((NT,1));
TAPMIN = np.ones((NT,1));
TAPSTP = np.ones((NT,1));

output.append("TRANSFORMER DATA FOR TOTAL NOS OF CIRCUITS");
output.append("SNO F.NO   T.NO NCKT --R PU-- --X PU-- ---AN--- --RAT---");

m = 0;
l = 1;

for k in range(0, NTR):
    line = list(map(float, (format(lines[7+NB+m])).split()));
    m = 2+m;

    FB[k] = int(line[1]);
    TB[k] = int(line[2]);
    NCKT[k] = int(line[3]);
    ZL[k] = complex(line[4],line[5]);
    TAP[k] = float(line[6]);
    BL[k] = complex(0,0);
    RAT[k] = float(line[7])/PBASE;
    LEN[k] = 0;
    KV[k] = 0;

    BL[k] = BL[k]*NCKT[k];
    ZL[k] = ZL[k]/NCKT[k];
```

```

YL[k] = 1/ZL[k];

lineTAP = list(map(float, (format(lines[7+NB+l]).split())));
l = 2+l;

TAPMAX[k] = float(lineTAP[0]);
TAPMIN[k] = float(lineTAP[1]);
TAPSTP[k] = float(lineTAP[2]);
output.append(""+ str(k+1)+ str(FB[k])+ str(TB[k])+ str(NCKT[k])+ \
    str("%8.4f" %ZL[k].real)+ str("%8.4f" %ZL[k].imag)+ \
    str("%8.4f" %TAP[k])+ str("%8.4f" %RAT[k]));

output.append("LINE DATA FOR TOTAL NOS OF CIRCUITS");
output.append("SNO F.NO   T.NO NCKT --R PU-- --X PU-- --HLC")

m = 0;

for k in range(NTR, NT):
    line = list(map(float, (format(lines[7+NB+NTR+NTR+m]).split())));
    m = m+1;

    FB[k] = line[1];
    TB[k] = line[2];
    NCKT[k] = line[3];
    ZL[k] = complex(line[4],line[5]);
    BL[k] = complex(0,line[6]);
    RAT[k] = line[7]/PBASE;
    LEN[k] = line[8];
    KV[k] = line[9];

    BL[k] = BL[k] * NCKT[k];
    ZL[k] = ZL[k]/NCKT[k];

    if LEN[k]>0 and KV[k]>0:
        ZBASE = KV[k] * KV[k] / PBASE;
        ZL[k] = ZL[k] * LEN[k] / ZBASE;
        BL[k] = BL[k] * LEN[k] * ZBASE;

    YL[k] = 1/ZL[k];

    output.append(""+ str(k+1)+ str(FB[k])+ str(TB[k])+ str(NCKT[k])+ \
        str("%8.4f" %ZL[k].real)+ str("%8.4f" %ZL[k].imag)+ \
        str("%8.4f" %BL[k].real)+ str("%8.4f" %RAT[k])+ \
        str("%8.4f" %TAP[k]));

```

```

SNO = np.zeros((NSH,1), dtype=int);
SUS = np.zeros((NSH,1));
SUSMAX = np.zeros((NSH,1));
SUSMIN = np.zeros((NSH,1));
SUSSTP = np.zeros((NSH,1));

output.append("SHUNT CAPACITOR DATA");
output.append("SNO B.NO -MVAR-pu-");

for k in range(0, NSHC):
    line = list(map(float, (format(lines[7+NB+NTR+NTR+NTRL+k])).split()));
    SNO[k] = line[1];
    SUS[k] = line[2]/PBASE;

output.append("SWITCHABLE CAPACITOR DATA");
output.append("SNO B.NO --MAX--- --MIN--- --STEP-- -ACTUAL-");
output.append("      --MVAR-- --MVAR-- --MVAR-- --MVAR--");

for k in range(NSHC, (NSHC+NSVS)):
    line = list(map(float, (format(lines[7+NB+NTR+NTR+NTRL+NSHC+(k-10)]).split())));

    SNO[k] = line[1];
    SUSMAX[k] = line[2] / PBASE;
    SUSMIN[k] = line[3] / PBASE;
    SUSSTP[k] = line[4] / PBASE;
    SUS[k] = line[5] / PBASE;
    output.append("'" + str(k+1) + str(SNO[k]) + str("%8.4f" %SUSMAX[k]) + \
        str("%8.4f" %SUSMIN[k]) + str("%8.4f" %SUSSTP[k]) + \
        str("%8.4f" %SUS[k]));

output.append("SHUNT REACTOR DATA");
output.append("SNO B.NO --MVAR--");
m = 0;
for k in range((NSHC+NSVS), NSH):
    line = list(map(float, (format(lines[7+NB+NTR+NTR+NTRL+NSHC+NSVS+m])).split()));
    mat(lines[7+NB+NTR+NTR+NTRL+NSHC+NSVS+m]);

    m = m+1;

    SNO[k] = line[1];
    SUS[k] = -line[2]/PBASE;
    output.append("'" + str(k+1) + str(SNO[k]) + str("%8.4f" %SUS[k]));

```

```

def PFE_initialization():
    # global NB, NBB, NS, NG, NLB, NTR, NTRL, NT, NSHC, NSVS, NSHR, NSH, NREG
    # global VSLACK, TOLER, PBASE, VLMAX, VLMIN, ITMAX
    # global BIND, BSN, BNAM, PG, QG, PD, QD, V, DEL, QGMAX, QGMIN, VSH
    # global FB, TB, NCKT, YL, ZL, BL, TAP, RAT, KV, LEN, TAPMAX, TAPMIN, TAPSTP
    # global SNO, SUS, SUSMAX, SUSMIN, SUSSTP
    # global YB, BP, BPP, LBP, PBP, UBP, LBPP, PBPP, UBPP
    global LAM, LRI, LCI
    # global PGMAX, PGMIN, VGMAX, VGMIN, ap, bp, cp
    # global ofp, ifp
    # global JBF, JB
    # global DELP, DELQ, lfep, lfeq
    # global SLFLOW, SBUS, SLOSS, NSI
    global LNSI, LNVI, FBI, TBI, PAI, PBI, QAI, QBI, RL, XL, ZLM2, MF, MT, MS, YS
    global LMIS, LJAC, U, Pa, Pb, Qa, Qb, ZL, XO

    NSI = int(BIND[NS-1,0])
    LNSI = np.concatenate((np.arange(1,NSI) , np.arange(NSI+1,NB+1)))
    LNVI = np.array(np.arange(NG+1,NB+1))
    LRI = np.concatenate(((np.arange(1,4*NT+1)), (4*NT+LNSI), (4*NT+NB+LNVI)))
    # LRI = LRI[np.newaxis,: ]
    LCI = np.concatenate((LNVI, (LNSI + NB), (np.arange(1,4*NT+1)) + 2*NB))
    # LCI = LCI[np.newaxis,: ]

    FBI = BIND[FB-1,0].astype(int)
    TBI = BIND[TB-1,0].astype(int)
    PAI = 2*NB+0*NT+np.arange(1,NT);
    QAI = 2*NB+1*NT+np.arange(1,NT);
    PBI = 2*NB+2*NT+np.arange(1,NT);
    QBI = 2*NB+3*NT+np.arange(1,NT);

    MF = sparse.lil_matrix(((np.ones((NT,1)) * np.arange(1,NB+1))
                           == (FBI @ np.ones((1,NB))))*1).T
    MT = sparse.lil_matrix(((np.ones((NT,1)) * np.arange(1,NB+1))
                           == (TBI @ np.ones((1,NB))))*1).T
    MS = sparse.lil_matrix(((np.ones((NSH,1)) * np.arange(1,NB+1))
                           == (BIND[SNO-1,0] @ np.ones((1,NB))))*1).T

    YS = MF@BL + MT@BL + MS@SUS*1j + MF@((1/ZL)*(1-TAP)/TAP**2) + \
        MT @((1/ZL)*(TAP-1)/(TAP))

    ZL = ZL * TAP
    RL = ZL.real
    XL = ZL.imag

```



```
ZLM2 = abs(ZL)**2
```

```
U = np.concatenate((VSH[0:NG], np.ones((NLB,1))))**2
```

```
# DEL = np.zeros((NB,1))
```

```
Pa = np.zeros((NT,1))
```

```
Qa = np.zeros((NT,1))
```

```
Pb = np.zeros((NT,1))
```

```
Qb = np.zeros((NT,1))
```

```
LJAC = sparse.lil_matrix((2*Nb+3*NT+NT,2*Nb+3*NT+NT))
```

```
LMIS = sparse.lil_matrix((4*NT+NB-1+NLB,1))
```

```
def LINEJAC():
```

```
## FUA
```

```
LJAC[0*NT+0*Nb+0:0*NT+0*Nb+NT,0*Nb+0*NT+0:0*Nb+0*NT+NB] \
```

```
= sparse.lil_matrix.multiply(
```

```
    MF.T, ((2*U[FBI-1,0] + 2*(Pa*RL + Qa*XL - U[TBI-1,0]/2))
```

```
    @ np.ones((1,NB))))
```

```
LJAC[0*NT+0*Nb+0:0*NT+0*Nb+NT,0*Nb+0*NT+0:0*Nb+0*NT+NB] \
```

```
= LJAC[0*NT+0*Nb+0:0*NT+0*Nb+NT,0*Nb+0*NT+0:0*Nb+0*NT+NB] + \
```

```
sparse.lil_matrix.multiply(MT.T,(-U[FBI-1,0] @ np.ones((1,NB))))
```

```
LJAC[0*NT+0*Nb+0:0*NT+0*Nb+NT,2*Nb+0*NT+0:2*Nb+0*NT+NT] \
```

```
= sparse.spdiags((2*U[FBI-1,0]*RL+2*Pa*ZLM2).T,0,NT,NT)
```

```
LJAC[0*NT+0*Nb+0:0*NT+0*Nb+NT,2*Nb+1*NT+0:2*Nb+1*NT+NT] \
```

```
= sparse.spdiags((2*U[FBI-1,0]*XL+2*Qa*ZLM2).T,0,NT,NT)
```

```
## FUB
```

```
LJAC[1*NT+0*Nb+0:1*NT+0*Nb+NT,0*Nb+0*NT+0:0*Nb+0*NT+NB] \
```

```
= sparse.lil_matrix.multiply(
```

```
    MT.T, ((2*U[TBI-1,0] + 2*(Pb*RL + Qb*XL - U[FBI-1,0]/2))
```

```
    @ np.ones((1,NB))))
```

```
LJAC[1*NT+0*Nb+0:1*NT+0*Nb+NT,0*Nb+0*NT+0:0*Nb+0*NT+NB] \
```

```
= LJAC[1*NT+0*Nb+0:1*NT+0*Nb+NT,0*Nb+0*NT+0:0*Nb+0*NT+NB] + \
```

```
sparse.lil_matrix.multiply(MF.T,(-U[TBI-1,0] @ np.ones((1,NB))))
```

```
LJAC[1*NT+0*Nb+0:1*NT+0*Nb+NT,2*Nb+2*NT+0:2*Nb+2*NT+NT] \
```

```
= sparse.spdiags((2*U[TBI-1,0]*RL+2*Pb*ZLM2).T,0,NT,NT)
```

```
LJAC[1*NT+0*Nb+0:1*NT+0*Nb+NT,2*Nb+3*NT+0:2*Nb+3*NT+NT] \
```

```
= sparse.spdiags((2*U[TBI-1,0]*XL+2*Qb*ZLM2).T,0,NT,NT)
```

```

## FDA
LJAC[2*NT+0*NB+0:2*NT+0*NB+NT,0*NB+0*NT+0:0*NB+0*NT+NB] \
= sparse.lil_matrix.multiply(
    MF.T, np.tan(DEL[TBI-1,0]-DEL[FBI-1,0])*np.ones((1,NB)))

LJAC[2*NT+0*NB+0:2*NT+0*NB+NT,1*NB+0*NT+0:1*NB+0*NT+NB] \
= sparse.lil_matrix.multiply(
    -MF.T, (((U[FBI-1,0]+Pa*RL+Qa*XL)/np.cos(DEL[TBI-1,0]-DEL[FBI-1,0])**2)
            *np.ones((1,NB))))

LJAC[2*NT+0*NB+0:2*NT+0*NB+NT,1*NB+0*NT+0:1*NB+0*NT+NB] \
= LJAC[2*NT+0*NB+0:2*NT+0*NB+NT,1*NB+0*NT+0:1*NB+0*NT+NB] + \
    sparse.lil_matrix.multiply(
        MT.T, (((U[FBI-1,0]+Pa*RL+Qa*XL)\
                /np.cos(DEL[TBI-1,0]-DEL[FBI-1,0])**2)*np.ones((1,NB))))

LJAC[2*NT+0*NB+0:2*NT+0*NB+NT,2*NB+0*NT+0:2*NB+0*NT+NT] \
= sparse.spdiags((RL*np.tan(DEL[TBI-1,0]-DEL[FBI-1,0])-XL).T,0,NT,NT)

LJAC[2*NT+0*NB+0:2*NT+0*NB+NT,2*NB+1*NT+0:2*NB+1*NT+NT] \
= sparse.spdiags((XL*np.tan(DEL[TBI-1,0]-DEL[FBI-1,0])+RL).T,0,NT,NT)

## FDB
LJAC[3*NT+0*NB+0:3*NT+0*NB+NT,0*NB+0*NT+0:0*NB+0*NT+NB] \
= sparse.lil_matrix.multiply(
    MT.T, np.tan(DEL[FBI-1,0]-DEL[TBI-1,0])*np.ones((1,NB)))

LJAC[3*NT+0*NB+0:3*NT+0*NB+NT,1*NB+0*NT+0:1*NB+0*NT+NB] \
= sparse.lil_matrix.multiply(
    MF.T, (((U[TBI-1,0]+Pb*RL+Qb*XL)/np.cos(DEL[FBI-1,0]-DEL[TBI-1,0])**2)
            *np.ones((1,NB))))

LJAC[3*NT+0*NB+0:3*NT+0*NB+NT,1*NB+0*NT+0:1*NB+0*NT+NB] \
= LJAC[3*NT+0*NB+0:3*NT+0*NB+NT,1*NB+0*NT+0:1*NB+0*NT+NB] - \
    sparse.lil_matrix.multiply(
        MT.T, (((U[TBI-1,0]+Pb*RL+Qb*XL)\
                /np.cos(DEL[FBI-1,0]-DEL[TBI-1,0])**2)*np.ones((1,NB))))

LJAC[3*NT+0*NB+0:3*NT+0*NB+NT,2*NB+2*NT+0:2*NB+2*NT+NT] \
= sparse.spdiags((RL*np.tan(DEL[FBI-1,0]-DEL[TBI-1,0])-XL).T,0,NT,NT)

LJAC[3*NT+0*NB+0:3*NT+0*NB+NT,2*NB+3*NT+0:2*NB+3*NT+NT] \
= sparse.spdiags((XL*np.tan(DEL[FBI-1,0]-DEL[TBI-1,0])+RL).T,0,NT,NT)

```

```
## FP
```

```
LJAC[4*NT+0*Nb+0:4*NT+0*Nb+Nb,0*Nb+0*NT+0:0*Nb+0*NT+Nb] \
= sparse.spdiags((-YS.real).T,0,Nb,Nb)
```

```
LJAC[4*NT+0*Nb+0:4*NT+0*Nb+Nb,2*Nb+0*NT+0:2*Nb+0*NT+NT] \
= MF
```

```
LJAC[4*NT+0*Nb+0:4*NT+0*Nb+Nb,2*Nb+2*NT+0:2*Nb+2*NT+NT] \
= MT
```

```
## FQ
```

```
LJAC[4*NT+1*Nb+0:4*NT+1*Nb+Nb,0*Nb+0*NT+0:0*Nb+0*NT+Nb] \
= sparse.spdiags((YS.imag).T,0,Nb,Nb)
```

```
LJAC[4*NT+1*Nb+0:4*NT+1*Nb+Nb,2*Nb+1*NT+0:2*Nb+1*NT+NT] \
= MF
```

```
LJAC[4*NT+1*Nb+0:4*NT+1*Nb+Nb,2*Nb+3*NT+0:2*Nb+3*NT+NT] \
= MT
```

```
def LINEMIS():
```

```
LMIS[np.arange(0,(1*NT))] = U[FBI-1,0]**2 + \
2*U[FBI-1,0]*(Pa*RL+Qa*XL-U[TBI-1,0]/2) + (Pa**2+Qa**2)*ZLM2
```

```
LMIS[np.arange(NT,(2*NT))] = U[TBI-1,0]**2 + \
2*U[TBI-1,0]*(Pb*RL+Qb*XL-U[FBI-1,0]/2) + (Pb**2+Qb**2)*ZLM2
```

```
LMIS[np.arange(2*NT,(3*NT))] = \
(U[FBI-1,0]+Pa*RL+Qa*XL)*np.tan(DEL[TBI-1,0]-DEL[FBI-1,0]) - (Pa*XL-Qa*RL)
```

```
LMIS[np.arange(3*NT,(4*NT))] = \
(U[TBI-1,0]+Pb*RL+Qb*XL)*np.tan(DEL[FBI-1,0]-DEL[TBI-1,0]) - (Pb*XL-Qb*RL)
```

```
LMIS[np.arange(4*NT,4*NT+1*(Nb-1))] = \
PG[LNSI-1] - PD[LNSI-1] + MF[LNSI-1,:]*Pa + MT[LNSI-1,:]*Pb - \
(U[LNSI-1]*YS[LNSI-1]).real
```

```
LMIS[np.arange(4*NT+(Nb-1),4*NT+(Nb-1)+(Nb-NG))] = \
QG[LNVI-1] - QD[LNVI-1] + MF[LNVI-1,:]*Qa + MT[LNVI-1,:]*Qb + \
(U[LNVI-1]*YS[LNVI-1]).imag
```

```

def NR_est():
    global LMIS, LJAC, U, Pa, Pb, Qa, Qb, ZL, ERROR, runT, V

    IT= 0
    ERROR = np.zeros((1,1))
    DXX = (spsolve(-LJAC[np.ix_(LRI-1,LCI-1)],LMIS) \
            [np.newaxis,:]).T
    ERROR[IT] = max(abs(LMIS)).toarray()[0,0]
    # ERROR = np.append(ERROR, max(abs(LMIS)).toarray()[0,0])
    t1 = time.time()

    while (ERROR[IT]>TOLER and IT <= ITMAX):
        LINEJAC()

        DX = (spsolve(-LJAC[np.ix_(LRI-1,LCI-1)],LMIS) \
                [np.newaxis,:]).T
        #DX = (sparse.inv(-LJAC[np.ix_(LRI-1,LCI-1)])) @ LMIS

        if (IT == 0):
            DXX = DX
        else:
            DXX = np.append(DXX, DX, axis=1)

        U[LNVI-1] = U[LNVI-1] + DX[0:NB-NG]
        DEL[LNSI-1] = DEL[LNSI-1] + DX[(NB-NG)+0:(NB-NG)+NB-1]
        Pa = Pa + DX[(NB-NG+NB-1)+0*NT:(NB-NG+NB-1)+1*NT]
        Qa = Qa + DX[(NB-NG+NB-1)+1*NT:(NB-NG+NB-1)+2*NT]
        Pb = Pb + DX[(NB-NG+NB-1)+2*NT:(NB-NG+NB-1)+3*NT]
        Qb = Qb + DX[(NB-NG+NB-1)+3*NT:(NB-NG+NB-1)+4*NT]

        LINEMIS()
        IT = IT + 1;
        ERROR = np.append(ERROR, max(abs(LMIS)).toarray()[0,0])

    t2 = time.time()

    runT = (t2-t1)*1000

    V = np.sqrt(U);
    st.write("""
    #
    ## **Bus Voltage:**
    """)

```

```
col1, col2 = st.beta_columns([3, 1])
col1.subheader("Line Chart of the bus voltage")
col1.line_chart(V)

col2.subheader("Numerical Values")
col2.write(V)

st.write("""
#
#
## **Process performance information:**
### Error at each Iterations
""")

col3, col4 = st.beta_columns([3, 1])
col3.subheader("Line Chart of the estimation error")
col3.line_chart(ERROR)

col4.subheader("Numerical Values")
col4.write(ERROR)

st.write("""### Number of Iterations: """, IT+1)
st.write("""### Time taken: """, round(runT, 3), ""ms""")

def Final_results():
    PI = 3.1415927
    PG_TOT = 0
    QG_TOT = 0
    PD_TOT = 0
    QD_TOT = 0
    BSNMAX = 0

    SLOSS = complex(0, 0)

    SLFLOW = (0 + 0j) * np.zeros((NT, 2), dtype=int)
    SBUS = (0 + 0j) * np.zeros((NB, 1), dtype=int)

    #BIND = np.zeros((int(BSNMAX), 1))

    for k in range(0, NT):
        i1 = int(BIND[FB[k]-1])-1
        j1 = int(BIND[TB[k]-1])-1
        vi = V[i1] * complex(cos(DEL[i1]), sin(DEL[i1]))
```

```

vj = V[j1] * complex(cos(DEL[j1]), sin(DEL[j1]))
SLFLOW[k, 0] = vi * np.conj((vi - vj) * YL[k] / TAP[k])
SLFLOW[k, 0] = SLFLOW[k, 0] + vi * np.conj(vi * BL[k])
SLFLOW[k, 0] = SLFLOW[k, 0] + vi * np.conj(vi * YL[k] * (1 / (TAP[k] * TAP[k]) - 1 /
TAP[k]))

```

```

SLFLOW[k, 1] = vj * np.conj((vj - vi) * YL[k] / TAP[k])
SLFLOW[k, 1] = SLFLOW[k, 1] + vj * np.conj(vj * BL[k])
SLFLOW[k, 1] = SLFLOW[k, 1] + vj * np.conj(vj * YL[k] * (1 - 1 / TAP[k]))

```

```

SBUS[i1] = SBUS[i1] + SLFLOW[k, 0]
SBUS[j1] = SBUS[j1] + SLFLOW[k, 1]

```

```

SLOSS = SLOSS + SLFLOW[k, 0] + SLFLOW[k, 1]

```

```

i1 = int(BIND[NS-1])-1
PG[i1] = np.real(SBUS[i1])

```

```

QG[0:NG] = np.imag(SBUS[0:NG])

```

```

output.append("\n\n\n\n\n")
output.append("OUTPUT OF THE LOAD FLOW FDLF METHOD PROGRAM ")
output.append("-----\n\n")
output.append("                GENERATOR STATUS")
output.append("-----\n")
output.append("-----")
output.append("SNO B.NO BUS NAME ---PD--- ---QD--- ---PG--- ---QG--- --QGMX-- --")
QGMN-- --V SH-- ---DEL---")
output.append("SNO B.NO BUS NAME  (MW)  (MVAR)  (MW)  (MVAR)  (MVAR)  (MVAR)  (PU)  (DEGREES)")
output.append("-----")

```

```

for k in range(0, NG):

```

```

    output.append(" " + str(k) + " " + str(BSN[k]) + " " + str(BNAM[k]) + \
        str("%8.2f" % (PD[k] * PBASE)) + " " + str("%8.2f" % (QD[k] * PBASE)) + \
        str("%8.2f" % (PG[k] * PBASE)) + " " + str("%8.2f" % (QG[k] * PBASE)) + " " + \
        str("%8.2f" % (QGMAX[k] * PBASE)) + " " + str("%8.2f" % (QGMIN[k] * PBASE)) + \
        str("%8.4f" % (V[k])) + " " + str("%8.4f" % (DEL[k] * (180.0 / pi))))
    PG_TOT = PG_TOT + PG[k]
    QG_TOT = QG_TOT + QG[k]
    PD_TOT = PD_TOT + PD[k]
    QD_TOT = QD_TOT + QD[k]

```

```

output.append("-----\n\n\n")

output.append("          LOAD BUS STATUS")
output.append("          -----\n")
output.append("-----")
output.append("SNO B.NO BUS NAME ---PD--- ---QD--- --V SH-- --DEL---")
output.append("          (MW) (MVAR) (PU) (DEGREE)")

for k in range(NG , NB ):
    output.append(" "+ str(k)+ " "+ str(BSN[k])+ " "+ str(BNAM[k])+ \
        str("%8.2f" % (PD[k] * PBASE))+ " "+ str("%8.2f" % (QD[k] * PBASE))+ \
        " "+ str("%8.4f" % (V[k]))+ " "+ str("%8.4f" % (DEL[k] * (180.0 / pi)))
    PD_TOT = PD_TOT + PD[k]
    QD_TOT = QD_TOT + QD[k]

output.append("-----\n\n")

output.append("TRANSMISSION LINE FLOW --- FLOWS ARE PER CIRCUIT ")
output.append("----- \n")
output.append("-----")
output.append("SNo FBNO --NAME-- TBNO --NAME-- NCKT   FROM FLOW   TO
FLOW   MVA ")
output.append("          ----- FLOW ")
output.append("          REAL REACTIVE REAL REACTIVE   PER
")
output.append("          (MW) (MVAR) (MW) (MVAR)   CKT ")
output.append("-----")

for k in range(0, NT):
    k1 = int(BIND[FB[k]-1])-1
    k2 = int(BIND[TB[k]-1])-1
    output.append(" "+ str(k)+ " "+ str(BSN[k1])+ " "+ str(BNAM[k1])+ " "+ \
        str(BSN[k2])+ " "+ str(BNAM[k2])+ " "+ " "+ str(NCKT[k])+ \
        str("%8.2f" % (np.real(SLFLOW[k, 0]) * PBASE))+ \
        str("%8.2f" % (np.imag(SLFLOW[k, 0]) * PBASE))+ \
        str("%8.2f" % (np.real(SLFLOW[k, 1]) * PBASE))+ \
        str("%8.2f" % (np.imag(SLFLOW[k, 1]) * PBASE))+ \
        str("%8.2f" % (abs(SLFLOW[k, 0]) * PBASE))+ " ")

    if abs(SLFLOW[k, 0]) < (0.1 * RAT[k]):
        output[len(output)-1] = output[len(output)-1] + \
            str("UNDER - %4.2f" % (100 * abs(SLFLOW[k, 0]) / (RAT[k])))

```

```

if abs(SLFLOW[k, 1]) > 1.1 * RAT[k]:
    output[len(output)-1] = output[len(output)-1] + \
        (str("OVER - %4.2f" % (100 * abs(SLFLOW[k, 0]) / (RAT[k]))))

if abs(SLFLOW[k, 0]) > 0.1 * RAT[k] and abs(SLFLOW[k, 0]) < 1.1 * RAT[k]:
    output.append(" ")

output.append("-----\n\n")

output.append("SUMMARY OF THE RESULTS")
output.append("-----\n")

output.append("-----")
output.append("TOTAL REAL GENERATION    = %15.5f" % (PG_TOT * PBASE))
output.append("TOTAL REAL LOAD            = %15.5f" % (PD_TOT * PBASE))
output.append("TOTAL REAL LOSS             = %15.5f" % (np.real(SLOSS) * PBASE))
output.append("-----")
output.append("TOTAL REACTIVE GENERATION = %15.5f" % (QG_TOT * PBASE))
output.append("TOTAL REACTIVE LOAD      = %15.5f" % (QD_TOT * PBASE))
output.append("TOTAL REACTIVE LOSS      = %15.5f" % (np.imag(SLOSS) * PBASE))
# output.append("TOTAL SHUNT GENERATION = %15.5f" % (sh_gen * PBASE));
output.append("-----")
output.append("TRANSMISSION LOSS        = %15.5f" % (np.real(SLOSS) * PBASE))
output.append("-----")

def filedownload(df):
    # Source
    # https://discuss.streamlit.io/t/how-to-download-file-in-streamlit/1806

    df = pd.Series(df)
    df.str.replace("", " ")
    # st.write(df)

    st.write("""
    #
    #
    ## **Save the output file.**
    """)

    file_name = input_file.name[:-2]
    csv = df.to_csv(index=False)
    b64 = base64.b64encode(csv[1:].encode()).decode() # strings <-> bytes conversions
    href = f'<a href="data:file/csv;base64,{b64}" download="{file_name}-LW-results.txt">link
to save the file</a>'

```



```
return href
```

```
st.set_page_config(layout="wide")
st.sidebar.header("Pages")
page = ["Home Page", "Operation Module", "About Us", "Contact Us"]
selectP = st.sidebar.selectbox("", page)
```

```
if selectP == "Home Page":
    st.title("TRANSACTIVE ENERGY SYSTEM ANALYSIS IN PYTHON (LINE-WISE)")
    col1, col2 = st.beta_columns([3, 1])
    col1.subheader("")
    col1.write("""
    # About Our Project:
```

```
    ## In engineering, power flow analysis or the study of load flow is a powerful tool in many
    electrical power systems and it is usually used in power system planning and operations. The
    main purpose of power flow studies is to plan a head for theoretical circumstances. For instance,
    if a transmission line is taken off or disconnected for support, will the remaining lines in the
    network handle the load without surpassing their rated values. Thus, Power system analysis is
    targeted to determine the currents, voltages at different busses, and real and reactive power flows
    in a network under a given load conditions. However, since operators and researchers' aim to
    achieve voltage stability with optimal power flow, voltage collapse is a main issue due to the
    voltage drop and the heavy reactive power flow in the connected lines which results in a lack of
    reactive power at one or more buses.
```

```
    ## Bus-wise power balance equations using techniques like Newton Raphson method, the
    fast-decoupled methods and the Gauss-Seidel are widely used to analyze voltage collapse. How-
    ever, the Bus-wise analysis using Newton Raphson method requires additional analysis to deter-
    mine the most critical set of lines. Therefore, this project aims to use Line-wise Newton Raphson
    method to analyze voltage collapse as it is reliable, and it determines the most critical set of lines
    without any additional computing as the voltage collapse index is shown for all lines in the Jaco-
    bian of the Line-wise Newton Raphson method.
```

```
    #
    """)
```

```
col3, col4, col5 = st.beta_columns([1, 1, 1])
with col3:
    col3.write("""
    ## Test the module
    """)
    if st.button("Operation Module"):
        selectP = "Operation Module"
with col4:
    col4.write("""
    ## learn about the team
```

```
        """)
    if st.button("About Us"):
        selectP = "About Us"
with col5:
    col5.write("""
    ## Contact information
    """)
    if st.button("Contact Us"):
        selectP = "Contact Us"

if selectP == "Operation Module":
    st.title("Linewise Method for Power Flow and Voltage Collapse Calculations")
    st.write("""
    ## Please enter the data file below

    """)
    input_file = st.file_uploader("", type=["txt", "csv", "i"])
    if (input_file is not None):
        input_process(input_file)
        PFE_initialization()
        LINEJAC()
        LINEMIS()
        NR_est()
        Final_results()
        st.markdown(filedownload(output), unsafe_allow_html=True)

if selectP == "About Us":
    st.title("About Us")
    st.write("""
    ## Team BV04:
    ### Mohammad Ikailan, Vineet Nischal, Khalid Johar and Laeticia Osemeke are fourth
    year electrical engineering students at Ryerson university known as Team BV04 . We are pas-
    sionate about power flow analysis. We work together to design, create and produce work that we
    are proud of.
    """)

if selectP == "Contact Us":
    st.title("Contact Us")
    st.write("""
    #
    ## **FLC**:
    ### Amr Mohammed: amr.adel@ryerson.ca
    """)
```

```
#  
## **Team members**:  
###  Mohammad Ikailan: mikailan@ryerson.ca  
###  Vineet Nischal: vineet.nischal@ryerson.ca  
###  Khalid Johar: khald.muhammed@ryerson.ca  
###  Laeticia Osemeke: losemeke@ryerson.ca  
""")
```