Solving the Power Flow Problem via the Newton-Raphson Method

By: Mike Mayhew and Michael Stickels

Overview

The power flowing around a system of interconnected buses, generators and loads can be determined by using the Power Flow equations. However, the complex power at each node within the system is dependent upon the values of power injected into the system, the system's topography and the physical properties of the system. This yields a complicated series of equations where the inputs and outputs of each node are completely dependent upon those of their connected nodes. As such, we cannot solve the power flow equation parameters explicitly.

By reviewing the given characteristics of the system and understanding the desired (or physically required) performance of the system, we can assign identities to each node. These designations break down the power flow equations at each node into two groups; the explicit equations and the implicit equations. Explicit equations are, as implied, simpler to solve for. The implicit equations are more complicated and cannot be solved directly.

To solve the implicit equations, we use systems of equations methods from linear algebra. The overall process is known as the "Newton-Raphson Method." This method works by building a matrix of the admittances between all nodes, building a matrix of all of the power flow equations at each node and another matrix with assumptions of values of the unknown variables. A Jacobian matrix of the power flow equations is built which includes four quadrants:

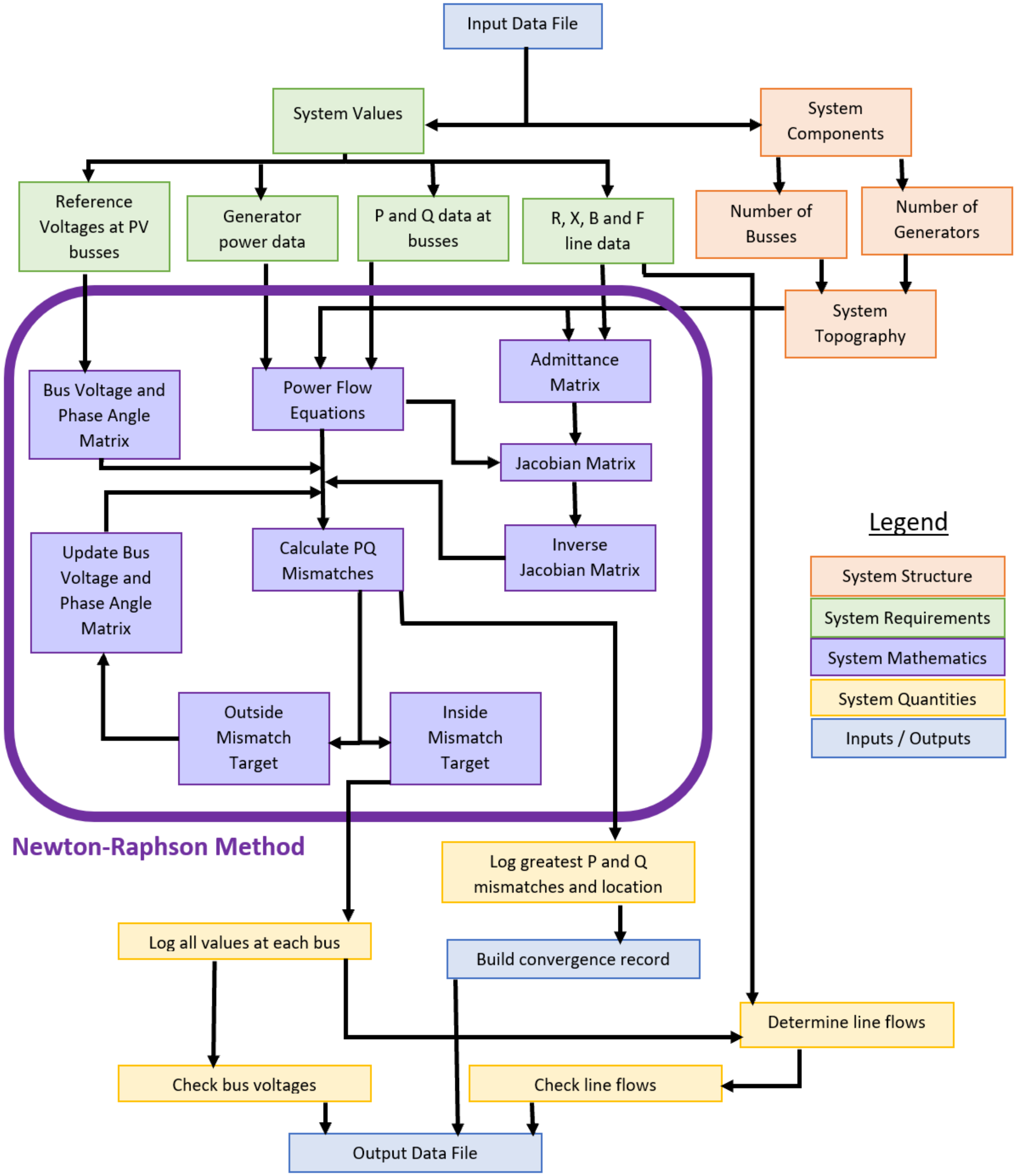
- H-quadrant (top left): Partial derivatives of P-equations with respect to angle θ.
- M-quadrant (top right): Partial derivatives of P-equations with respect to voltage.
- N-quadrant (bottom left): Partial derivatives of Q-equations with respect to angle θ.
- L-quadrant (bottom right): Partial derivatives of Q-equations with respect to voltage.

This Jacobian matrix is then inverted and it is multiplied by the negative matrix of the power flow equations. The assumed values of the unknown variables are plugged into this new "delta" matrix. It is named such because it is indicative of the changes of the unknown variables that have occurred. These suggested changes are added to the assumed values and the resulting matrix now takes the place of the previous version of the assumed value matrix.

These new assumed values for the unknown variables at each node are plugged into the power flow equation matrix which yields new values of the power flow equations at each node. The new values of P and Q are compared to the previous values. If the difference of these two values of P and Q at all nodes are within a designated tolerance, the power flow equations have been solved. If any of these values are outside of the tolerance, the associated unknown variable suggestions are returned to the top and the process is completed again with the new suggested values. This process continues until either the values of all unknowns yield P and Q values at all nodes are within the tolerance specified or, in rare circumstances, the values of P ad Q at all nodes diverge.

Convergence of the P and Q values indicates that correct voltage and phase angle at all nodes has been correctly determined. At this point, other system parameters such as power flows between nodes can be determined and checked against the physical limitations of the system.

Visual Overview of Program to Solve Power Flow Equations



Details of Program to Solve Power Flow Equations

Python was used to develop a program to solve the power flow equations in a hypothetical power system. The power system's topography was provided along with data regarding all of the generators and busses. Some values of the variables at the busses were permanently established by the given data; others were to be solved for within the Python program.

Program front matter, imports and inputs:

As Numpy and Pandas were very helpful in the creation of this program, they were imported. Data about the system was imported from the provided .xlsx file. From this file, the system's bus parameters and topography could be determined directly.

Building the Admittance Matrix:

Two square matrices were initialized, one for each portion of the Y-matrix, both with the number of columns/rows equal to the number of busses. These two matrices, Real and Imaginary, were filled on the non-diagonal entries by having Python iterate over the input file, pull the R and X values associated with each bus and calculate the respective real or imaginary admittance value. For the diagonal entries of these matrices, the axis entries were summed and then negated. Finally, the shunt admittances were added to the diagonal entries.

Building the P-Q Mismatch Matrix and V-θ Matrix:

This matrix was determined to be single dimension as would be later required for matrix multiplication. It was determined that this matrix would first need to include the P-value at all busses with the exception of the slack bus. Python iterated over the input file to place the given P values into this matrix. The program then iterated over the Q-values of the non-generating busses. It appended these Q-values to the bottom of the mismatch matrix.

As required for the Newton-Rapshon method, another matrix of the same shape was initialized to house the V and θ values. While the θ values were all initialized at 0, the V values were set to those within the input file.

Building the P-Q Equation Matrix:

The P and Q equation matrices were built as functions within Python to allow the V-θ matrix to be passed in as parameter. The values of the V-θ matrix were inserted into the power flow equations at each node to find current values of P and Q. Both of the P and Q functions determined a temporary copy of their respective matrix that will be later passed into other functions within the Newton-Raphson method.

Building the Jacobian Matrix

The Jacobian matrix's shape was determined by finding the system topography via the input file. The four quadrants of the Jacobian matrix that were previously discussed in the Overview were built as individual matrices with shapes:

- H-quadrant: [# of busses – 1 (slack bus)] x [# of busses – 1 (slack bus)]
- M-quadrant: [# of busses – 1 (slack bus)] x [# of busses – # of generator busses]
- N-quadrant: [# of busses – # of generator busses] x [# of busses – 1 (slack bus)]
- L-quadrant: [# of busses – # of generator busses] x [# of busses – # of generator busses]

These four individual matrices had parameters passed in for indices that were to be used to place the values of the V-θ matrix. This converts all of the types within the four quadrant matrices from functions to floats. These four individual matrices are later assembled into a single full-size Jacobian matrix.

Functions for Updating the V-θ Matrix:

Later, during the Newton-Raphson method calculations, an ability to update the V-θ matrix will be needed to carry out the mismatch calculations. **Caution must be exercised while updating the V-θ matrix**. This stems from the fact that while all θ values will be updated, only busses without generators will have the V updated. A temporary matrix was constructed with two dimensions to prevent the Vset values from changing away from their declared values. The first dimension was used to house the index within the full V-θ matrix where the updates would occur. The second dimension contains the updates. This temporary matrix contains the same number of axes as the Jacobian and PQ matrices. This is critical as we are unable to multiply matrices with dimensional mismatches.

After the updates to the V-θ matrix were made to the correct corresponding index, the values of this matrix were passed into functions that updated the power flow equations at all busses with the new values for V and θ.

Setting Initial Values for the V-θ Matrix and Preparation for Newton-Raphson:

The V-θ matrix was initialized with V values provided by the input file and all θ values were set to the “flat start condition” of 0. In this same section, mismatch parameters were declared and a maximum number of iterations was added as a method to prevent the program from running eternally if the outputs of the program should result in divergence.

Newton-Raphson Algorithm

The Newton-Raphson method operates as a “while loop” as long as the mismatch of P and Q at any bus is larger than the acceptable mismatch of 0.1. The full-size Jacobian matrix from above is inverted, multiplied by the P and Q values at all busses and negated. The output of this operation is the change that needs to be made for each corresponding V and θ value in the system.

These changes are summed into the V-θ matrix and the new quantities in the V-θ matrix are sent into the P and Q equation functions to find the new values of P and Q at every bus. The new values of P and Q are compared to the assigned quantities of P and Q from the input file. If any of the values of P or Q vary by more than 0.1 from the assigned value at that bus, the new values of the V-θ matrix are sent back into the Newton-Raphson method.

If programmed correctly, the V-θ matrix should continue to be updated until the mismatch requirement is met. At that point, all variables have been solved for at all busses.

Output Information

Once all system parameters have been determined by the Newton-Raphson method, line flows can be calculated. All of this information is housed within a new matrix that defines all system characteristics. These values are compared to the ratings and requirements of the system, which are housed in another matrix. If system requirements are violated, the violation is flagged and printed with the output data.

System Testing

Throughout the creation of the program, print statements and execution halts were used to ensure completed portions of the program were acting as expected or provided correct data. These statements include spot checks throughout the program or full displays of large matrix values. In some cases, large matrices like the Jacobian were exported to a CSV file for easier readability. Indices were also checked via print statements. Determining correct indexing for the different functions was found to be one of the most demanding parts of writing the program. The large number of matrices used in the program have greatly varying shapes and some matrices like the PQ mismatch and Voltage θ matrix include two different variables for different sets of nodes.

Additional system testing would be used to ensure the outputs of the program meet the system's requirements and to verify that ratings within the system had not been violated.

Results Obtained

At the programs last state, our output continuously diverged, and we were unsuccessful in correctly calculating the power flow in the system. After extensive troubleshooting and debugging, the remaining problem is believed to lie within the calculations of the values of the Jacobian matrix and the associated indexing within these functions. Copies of the Jacobian were exported to .csv files to be inspected, and after many alterations and changes to the program they did not match the hand-calculated Jacobian, including cells which should have been empty that had data and vice versa. Because we were unsuccessful in correctly calculating the power flow, we cannot accurately check the parameters such as the line power limits. In our final run of the program, the calculated powers grew so large as to cause an overflow error, and so are obviously incorrect.

Step for Further Troubleshooting

We are confident that the major cause for divergence lies inside the calculation of the Jacobian. Unfortunately, we ran out of time to troubleshoot. However, if we had time to proceed our next step would be to add a 0th column to our PQ and Vθ matrix that includes the buss number. This would significantly increase the complexity of our matrix arithmetic, but it would allow us to much more easily troubleshoot our indexing errors and ensure the correct values are being used in calculations.

Convergence Records

Convergence record functions were coded into the program and do output a CSV file, however the values are obviously incorrect as the calculation diverges.

Contributions of Team Members

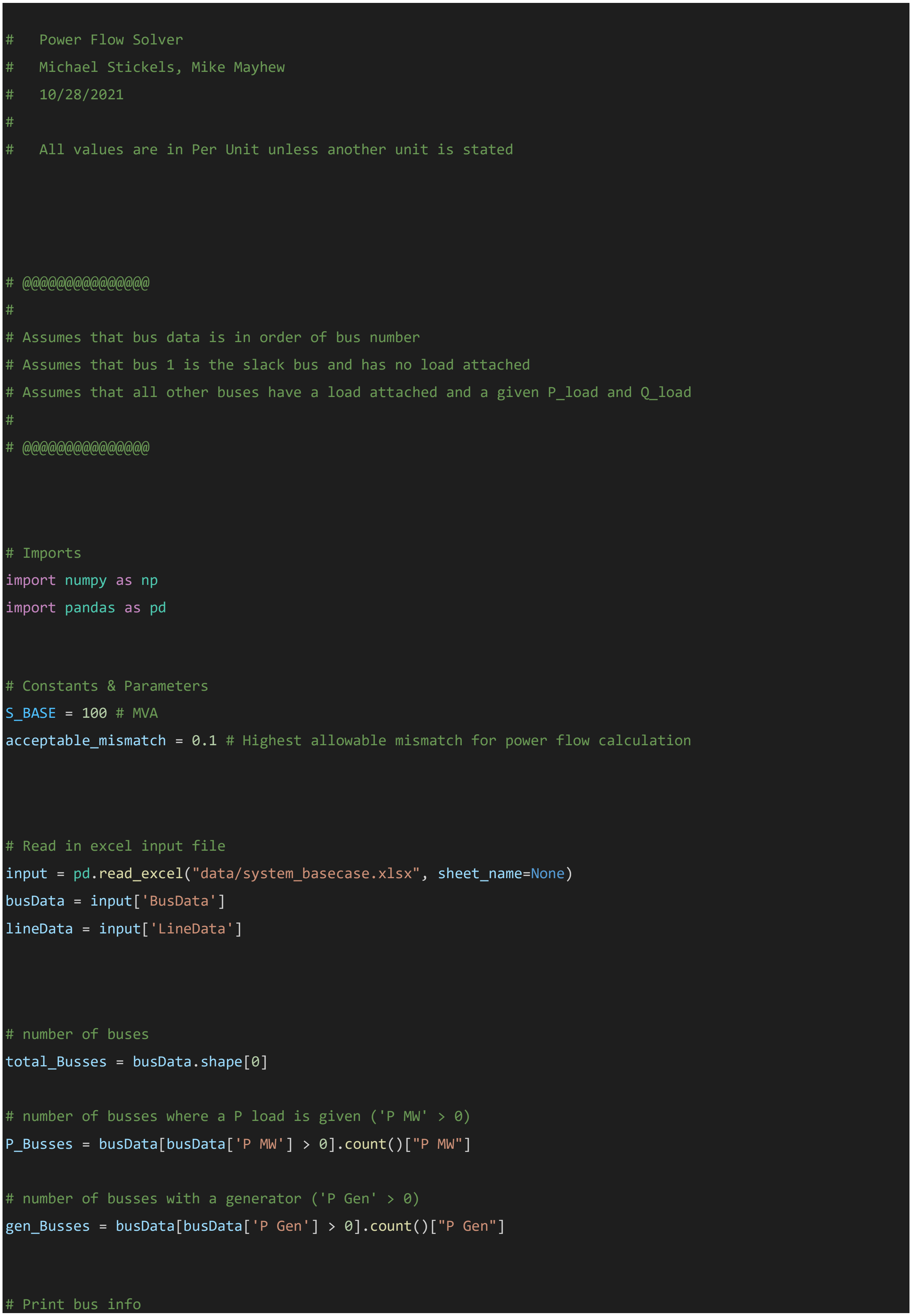
Mike Mayhew- Completed calculations for power flow equations, Jacobian matrix and conversions of input data by hand to ensure accuracy and for testing. Developed pseudocode, some troubleshooting. Completed research and gathered information. Completed report and flow chart.

Michael Stickels- Completed all coding of the project into Python and completed troubleshooting. Completed research and gathered information. Proofreading and review of report and flow chart.

Concluding Thoughts

This project really helped us gain a good understanding of this method of solving network power flow, and the application of the method to such a large network as well as our efforts to make the program function for any general system of any size forced us to pick apart the equations and learn a lot about the algorithm that we hadn't gained from the simple two bus examples and homework problems. We had obviously hoped to have a functioning program, but just getting as far as we did helped us learn a lot about this topic, and we are certain that the cause for the divergence is a lack of understanding and experience in Python rather than a lack of understanding of the solving method.

Documented code



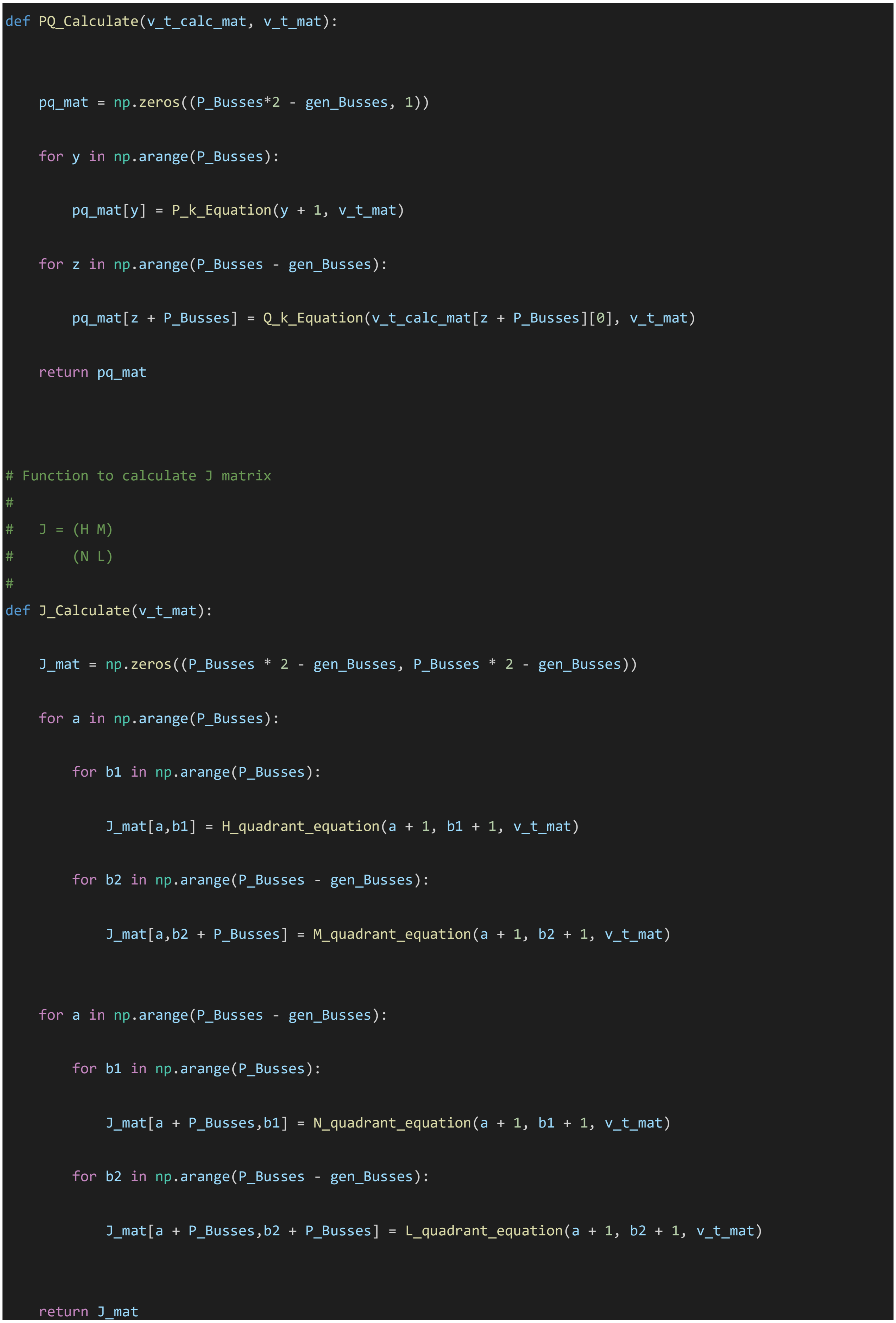
```
#   Power Flow Solver
#   Michael Stickels, Mike Mayhew
#   10/28/2021
#
#   All values are in Per Unit unless another unit is stated




# @@@@@@@@@@@@@@@
#
# Assumes that bus data is in order of bus number
# Assumes that bus 1 is the slack bus and has no load attached
# Assumes that all other buses have a load attached and a given P_load and Q_load
#
# @@@@@@@@@@@@@@@



# Imports
import numpy as np
import pandas as pd


# Constants & Parameters
S_BASE = 100 # MVA
acceptable_mismatch = 0.1 # Highest allowable mismatch for power flow calculation



# Read in excel input file
input = pd.read_excel("data/system_basecase.xlsx", sheet_name=None)
busData = input['BusData']
lineData = input['LineData']




# number of buses
total_Busses = busData.shape[0]

# number of busses where a P load is given ('P MW' > 0)
P_Busses = busData[busData['P MW'] > 0].count()["P MW"]

# number of busses with a generator ('P Gen' > 0)
gen_Busses = busData[busData['P Gen'] > 0].count()["P Gen"]



# Print bus info
```

```
print('Total Busses:', total_Busses)
print('Active Load Busses:', P_Busses)
print('Generator Busses (not including slack bus):', gen_Busses)



# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Build Y Matrix
# Y = G + jB

# Initialize empty admittance matrix
matrix_Y_real = np.zeros((total_Busses,total_Busses))
matrix_Y_imaginary = np.zeros((total_Busses,total_Busses))

# Input line admittances (off-diagonal values) from input data
for ind in lineData.index:

    i = lineData['From'][ind] - 1
    j = lineData['To'][ind] - 1

    # Real part -> G(i,j) = (-R)/(R^2+X^2)
    matrix_Y_real[i,j] = (-lineData['Rtotal, p.u.'][ind])/(lineData['Rtotal, p.u.'][ind]**2 +
lineData['Xtotal, p.u.'][ind]**2)
    matrix_Y_real[j,i] = (-lineData['Rtotal, p.u.'][ind])/(lineData['Rtotal, p.u.'][ind]**2 +
lineData['Xtotal, p.u.'][ind]**2)

    # Imaginary part -> B(i,j) = (X)/(R^2+X^2)
    matrix_Y_imaginary[i,j] = (lineData['Xtotal, p.u.'][ind])/(lineData['Rtotal, p.u.'][ind]**2 +
lineData['Xtotal, p.u.'][ind]**2)
    matrix_Y_imaginary[j,i] = (lineData['Xtotal, p.u.'][ind])/(lineData['Rtotal, p.u.'][ind]**2 +
lineData['Xtotal, p.u.'][ind]**2)

# Sum rows/cols for imtermidiate bus admittance
for i in np.arange(0,matrix_Y_real.shape[0]):

    # Real part -> ?
    matrix_Y_real[i,i] = matrix_Y_real.sum(axis=1)[i] * -1

    # Imaginary part -> ?
    matrix_Y_imaginary[i,i] = matrix_Y_imaginary.sum(axis=1)[i] * -1

# Add shunt admittances to diagonals (bus admittances)
for ind in lineData.index:

    i = lineData['From'][ind] - 1
    j = lineData['To'][ind] - 1

    # Imaginary part -> B(i,i) = G(i,i) + B_i/2
    matrix_Y_imaginary[i,i] = matrix_Y_imaginary[i,i] + lineData['Btotal, p.u.'][ind]/2
    matrix_Y_imaginary[j,j] = matrix_Y_imaginary[j,j] + lineData['Btotal, p.u.'][ind]/2
```

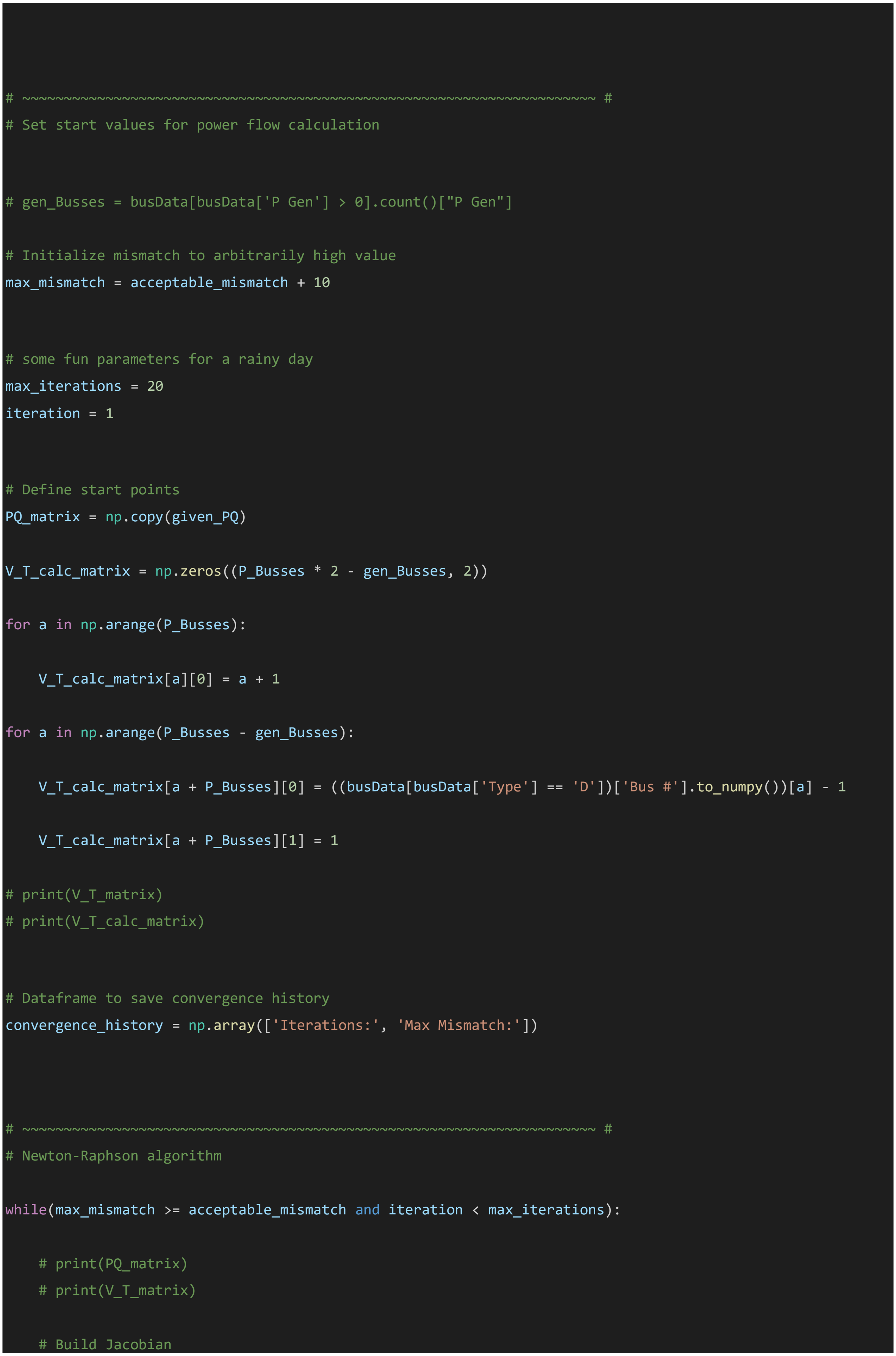
```
# Export matrix Y to csv files for troubleshooting
pd.DataFrame(matrix_Y_real).to_csv("output/matrix_Y_real.csv")
pd.DataFrame(matrix_Y_imaginary).to_csv("output/matrix_Y_imaginary.csv")



# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #

# Build matrix of given active and reactive power for mismatch calculation
given_PQ = np.zeros((P_Busses * 2 - gen_Busses, 1))

for x in np.arange(P_Busses):

    # given active power (P)
    given_PQ[x] = ((busData[busData['Type'] != 'S'])['P MW'].to_numpy())[x] -
((busData.loc[busData['Type'] != 'S'])['P Gen'].to_numpy())[x]


for x in np.arange(P_Busses - gen_Busses):

    # given reactive power (Q)
    given_PQ[x + P_Busses] = ((busData[busData['Type'] == 'D'])['Q MVAr'].to_numpy())[x]


# Initialize starting point for VT Matrix
#   All Thetas set to 0
#   Bus voltage set to given value from input
V_T_matrix = np.zeros((total_Busses * 2, 1), dtype=float)

for y in np.arange(total_Busses):

    V_T_matrix[y + P_Busses + 1][0] = busData['V Set'][y]




# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Determine P and Q Equations at Buses

# Calculate P_k
def P_k_Equation(k, v_t_mat):
    p_temp = 0
    k = int(k)

    for i in np.arange(P_Busses):

        p_temp += v_t_mat[int(k + total_Busses)] * v_t_mat[int(i + total_Busses)] * (matrix_Y_real[k][i] *
np.cos(v_t_mat[k] - v_t_mat[i]) + matrix_Y_imaginary[k][i] * np.sin(v_t_mat[k] - v_t_mat[i]))
```

```
    p_temp += busData['P MW'][k] - busData['P Gen'][k]

    return p_temp

# Calculate Q_k
def Q_k_Equation(k, v_t_mat):
    q_temp = 0
    k = int(k)

    for i in np.arange(total_Busses):

        q_temp += v_t_mat[int(k + total_Busses)] * v_t_mat[int(i + total_Busses)] * (matrix_Y_real[k][i] * np.sin(v_t_mat[k] - v_t_mat[i]) - matrix_Y_imaginary[k][i] * np.cos(v_t_mat[k] - v_t_mat[i]))

    q_temp += busData['Q MVAr'][k]

    return q_temp



# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #




# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Define functions for calculating Jacobian values
#
#   J = (H M)
#       (N L)
#

def H_quadrant_equation(k, i, v_t_mat):

    H_temp = 0

    if(k == i):

        for x in np.arange(total_Busses):
            if(x != k):

                H_temp += v_t_mat[k + P_Busses] * v_t_mat[x + P_Busses + 1] * (-matrix_Y_real[k,x] * np.sin(v_t_mat[k] - v_t_mat[x]) + matrix_Y_imaginary[k,x] * np.cos(v_t_mat[k] - v_t_mat[x]))


    else:

        H_temp = v_t_mat[k + P_Busses] * v_t_mat[i + P_Busses + 1] * (matrix_Y_real[k,i] * np.sin(v_t_mat[k] - v_t_mat[i]) - matrix_Y_imaginary[k,i] * np.cos(v_t_mat[k] - v_t_mat[i]))

    print(v_t_mat[k + P_Busses])
```

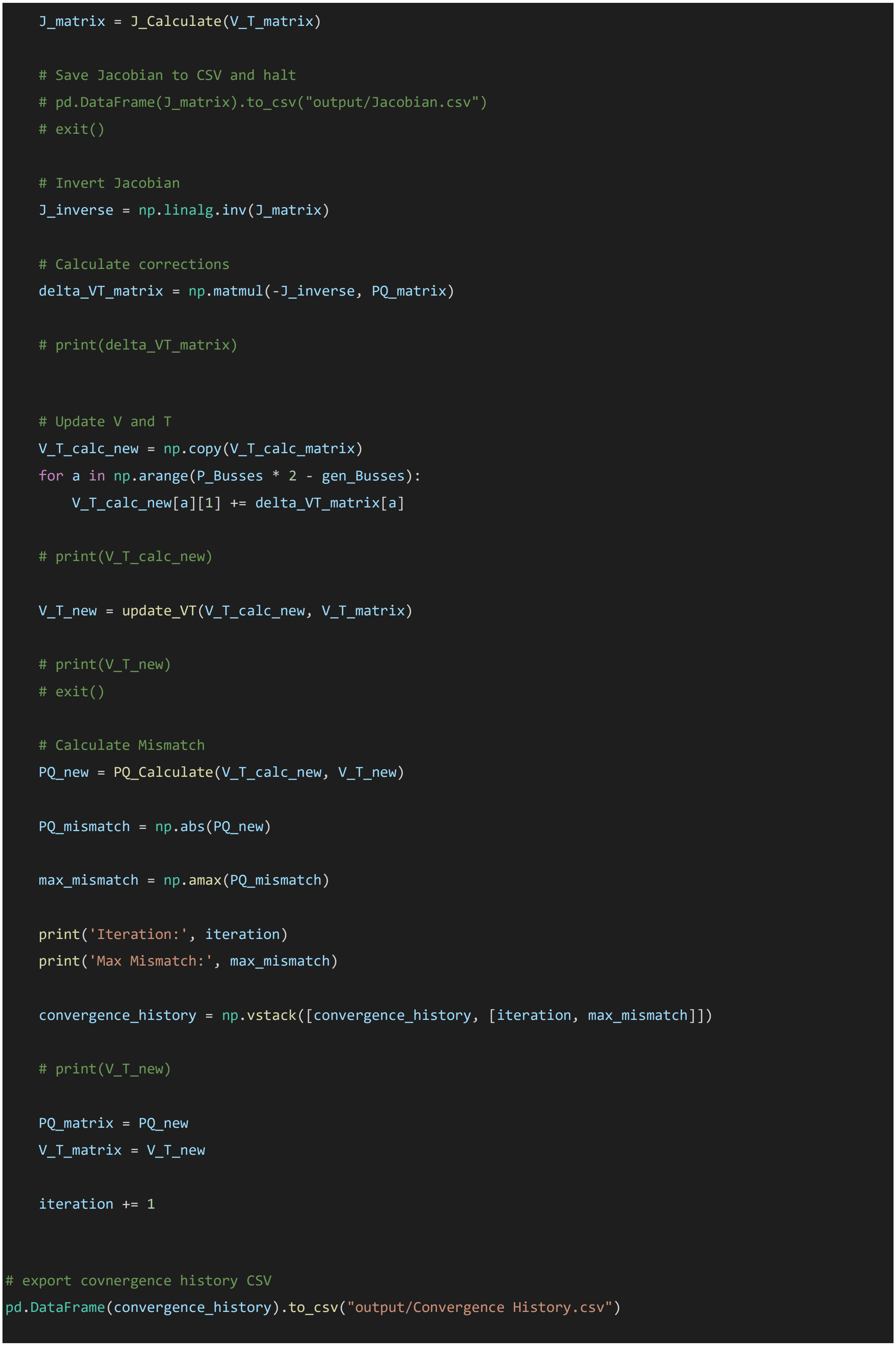
```
    return H_temp


def M_quadrant_equation(k, i, v_t_mat):

    M_temp = 0

    if(k == i):

        for x in np.arange(total_Busses):
            if(x != k):

                M_temp += v_t_mat[x + P_Busses + 1] * (matrix_Y_real[k,x] * np.cos(v_t_mat[k] - v_t_mat[x]) + matrix_Y_imaginary[k,x] * np.sin(v_t_mat[k] - v_t_mat[x]))

        M_temp +=  2 * matrix_Y_real[k,k] * v_t_mat[k + P_Busses]

    else:

        M_temp = v_t_mat[k + P_Busses] * (matrix_Y_real[k,i] * np.cos(v_t_mat[k] - v_t_mat[i]) + matrix_Y_imaginary[k,i] * np.sin(v_t_mat[k] - v_t_mat[i]))

    return M_temp


def N_quadrant_equation(k, i, v_t_mat):

    N_temp = 0

    if(k == i):

        for x in np.arange(total_Busses):
            if(x != k):

                N_temp += v_t_mat[k + P_Busses] * v_t_mat[x + P_Busses + 1] * (-matrix_Y_real[k,x] * np.cos(v_t_mat[k] - v_t_mat[x]) - matrix_Y_imaginary[k,x] * np.sin(v_t_mat[k] - v_t_mat[x]))

    else:

        N_temp = v_t_mat[k + P_Busses] * v_t_mat[i + P_Busses + 1] * (-matrix_Y_real[k,i] * np.cos(v_t_mat[k] - v_t_mat[i]) - matrix_Y_imaginary[k,i] * np.sin(v_t_mat[k] - v_t_mat[i]))

    return N_temp


def L_quadrant_equation(k, i, v_t_mat):

    L_temp = 0

    if(k == i):
        for x in np.arange(total_Busses):
            if(x != k):

                L_temp += v_t_mat[x + P_Busses + 1] * (matrix_Y_real[k,x] * np.sin(v_t_mat[k] -
v_t_mat[x]) - matrix_Y_imaginary[k,x] * np.cos(v_t_mat[k] - v_t_mat[x]))

        L_temp += -2 * matrix_Y_imaginary[k,k] * v_t_mat[k + P_Busses]

    else:

        L_temp = v_t_mat[k + P_Busses] * (matrix_Y_real[k,i] * np.sin(v_t_mat[k] - v_t_mat[i]) -
matrix_Y_imaginary[k,i] * np.cos(v_t_mat[k] - v_t_mat[i]))

    return L_temp



# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #


# Function to add delta V and T to VT Matrix
def update_VT(v_t_calc_mat, v_t):

    # print(v_t_calc_mat)
    # print(v_t)

    # Put changes into full VT matrix
    for x in np.arange(P_Busses):

        v_t[int(v_t_calc_mat[x][0])] = v_t_calc_mat[x][1]


    # print(v_t)

    for x in np.arange(P_Busses - gen_Busses):

        # print(x)
        # print(int(v_t_calc_mat[x + P_Busses][0]))
        # print(v_t_calc_mat[x + P_Busses][1])
        # print(x + P_Busses)
        # print(v_t[int(v_t_calc_mat[x + P_Busses][0])])

        v_t[int(v_t_calc_mat[x + P_Busses][0] + P_Busses + 1)] = v_t_calc_mat[x + P_Busses][1]

    # print(v_t)

    return v_t



# Function to calculate PQ matrix
```



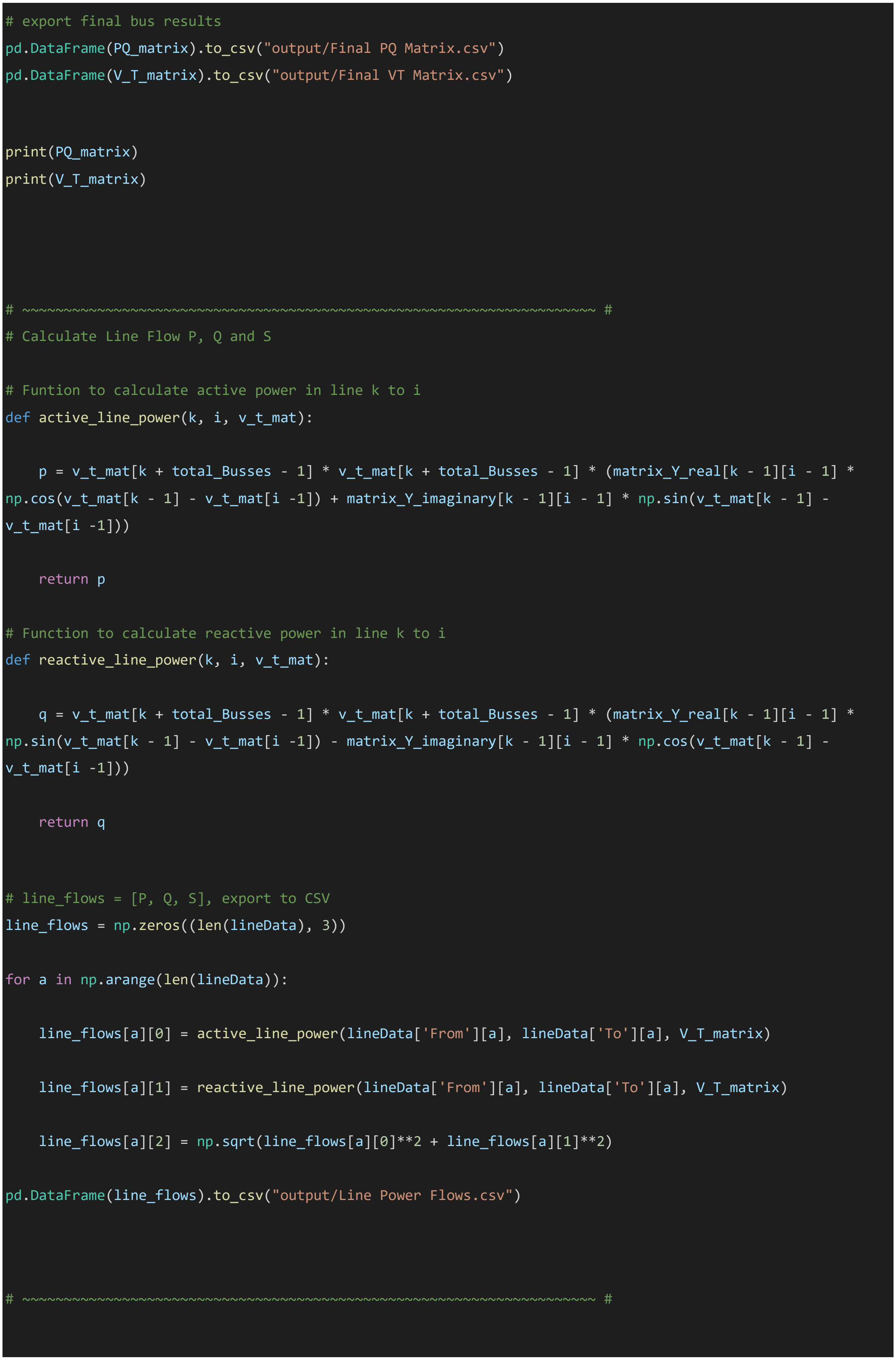
```python
def PQ_Calculate(v_t_calc_mat, v_t_mat):


    pq_mat = np.zeros((P_Busses*2 - gen_Busses, 1))

    for y in np.arange(P_Busses):

        pq_mat[y] = P_k_Equation(y + 1, v_t_mat)

    for z in np.arange(P_Busses - gen_Busses):

        pq_mat[z + P_Busses] = Q_k_Equation(v_t_calc_mat[z + P_Busses][0], v_t_mat)

    return pq_mat



# Function to calculate J matrix
#
#   J = (H M)
#       (N L)
#
def J_Calculate(v_t_mat):

    J_mat = np.zeros((P_Busses * 2 - gen_Busses, P_Busses * 2 - gen_Busses))

    for a in np.arange(P_Busses):

        for b1 in np.arange(P_Busses):

            J_mat[a,b1] = H_quadrant_equation(a + 1, b1 + 1, v_t_mat)

        for b2 in np.arange(P_Busses - gen_Busses):

            J_mat[a,b2 + P_Busses] = M_quadrant_equation(a + 1, b2 + 1, v_t_mat)


    for a in np.arange(P_Busses - gen_Busses):

        for b1 in np.arange(P_Busses):

            J_mat[a + P_Busses,b1] = N_quadrant_equation(a + 1, b1 + 1, v_t_mat)

        for b2 in np.arange(P_Busses - gen_Busses):

            J_mat[a + P_Busses,b2 + P_Busses] = L_quadrant_equation(a + 1, b2 + 1, v_t_mat)


    return J_mat
```



```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Set start values for power flow calculation


# gen_Busses = busData[busData['P Gen'] > 0].count()["P Gen"]

# Initialize mismatch to arbitrarily high value
max_mismatch = acceptable_mismatch + 10


# some fun parameters for a rainy day
max_iterations = 20
iteration = 1


# Define start points
PQ_matrix = np.copy(given_PQ)

V_T_calc_matrix = np.zeros((P_Busses * 2 - gen_Busses, 2))

for a in np.arange(P_Busses):

    V_T_calc_matrix[a][0] = a + 1

for a in np.arange(P_Busses - gen_Busses):

    V_T_calc_matrix[a + P_Busses][0] = ((busData[busData['Type'] == 'D'])['Bus #'].to_numpy())[a] - 1

    V_T_calc_matrix[a + P_Busses][1] = 1

# print(V_T_matrix)
# print(V_T_calc_matrix)


# Dataframe to save convergence history
convergence_history = np.array(['Iterations:', 'Max Mismatch:'])




# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Newton-Raphson algorithm

while(max_mismatch >= acceptable_mismatch and iteration < max_iterations):

    # print(PQ_matrix)
    # print(V_T_matrix)

    # Build Jacobian
```



```
    J_matrix = J_Calculate(V_T_matrix)

    # Save Jacobian to CSV and halt
    # pd.DataFrame(J_matrix).to_csv("output/Jacobian.csv")
    # exit()

    # Invert Jacobian
    J_inverse = np.linalg.inv(J_matrix)

    # Calculate corrections
    delta_VT_matrix = np.matmul(-J_inverse, PQ_matrix)

    # print(delta_VT_matrix)


    # Update V and T
    V_T_calc_new = np.copy(V_T_calc_matrix)
    for a in np.arange(P_Busses * 2 - gen_Busses):
        V_T_calc_new[a][1] += delta_VT_matrix[a]

    # print(V_T_calc_new)

    V_T_new = update_VT(V_T_calc_new, V_T_matrix)

    # print(V_T_new)
    # exit()

    # Calculate Mismatch
    PQ_new = PQ_Calculate(V_T_calc_new, V_T_new)

    PQ_mismatch = np.abs(PQ_new)

    max_mismatch = np.amax(PQ_mismatch)

    print('Iteration:', iteration)
    print('Max Mismatch:', max_mismatch)

    convergence_history = np.vstack([convergence_history, [iteration, max_mismatch]])

    # print(V_T_new)

    PQ_matrix = PQ_new
    V_T_matrix = V_T_new

    iteration += 1


# export covnergence history CSV
pd.DataFrame(convergence_history).to_csv("output/Convergence History.csv")
```



```
# export final bus results
pd.DataFrame(PQ_matrix).to_csv("output/Final PQ Matrix.csv")
pd.DataFrame(V_T_matrix).to_csv("output/Final VT Matrix.csv")


print(PQ_matrix)
print(V_T_matrix)




# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Calculate Line Flow P, Q and S

# Funtion to calculate active power in line k to i
def active_line_power(k, i, v_t_mat):

    p = v_t_mat[k + total_Busses - 1] * v_t_mat[k + total_Busses - 1] * (matrix_Y_real[k - 1][i - 1] * np.cos(v_t_mat[k - 1] - v_t_mat[i -1]) + matrix_Y_imaginary[k - 1][i - 1] * np.sin(v_t_mat[k - 1] - v_t_mat[i -1]))

    return p

# Function to calculate reactive power in line k to i
def reactive_line_power(k, i, v_t_mat):

    q = v_t_mat[k + total_Busses - 1] * v_t_mat[k + total_Busses - 1] * (matrix_Y_real[k - 1][i - 1] * np.sin(v_t_mat[k - 1] - v_t_mat[i -1]) - matrix_Y_imaginary[k - 1][i - 1] * np.cos(v_t_mat[k - 1] - v_t_mat[i -1]))

    return q


# line_flows = [P, Q, S], export to CSV
line_flows = np.zeros((len(lineData), 3))

for a in np.arange(len(lineData)):

    line_flows[a][0] = active_line_power(lineData['From'][a], lineData['To'][a], V_T_matrix)

    line_flows[a][1] = reactive_line_power(lineData['From'][a], lineData['To'][a], V_T_matrix)

    line_flows[a][2] = np.sqrt(line_flows[a][0]**2 + line_flows[a][1]**2)

pd.DataFrame(line_flows).to_csv("output/Line Power Flows.csv")



# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
```

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
# Build Ratings Matrix [Use Common Structure For All Matrices]



# Check Node Voltages Against Ratings



# Check Line Power Flows Against Ratings



# Flag Violations of Operating Limits / Settings



# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
```