MA2451 - EE20084 Structured Programming Report

Program Description

The description is chronological, detailing each function in the order it runs. Imported libraries are numpy, math, sys, csv

Reading and storing node data:

The input file is read using sys.arg[1] which takes the 2nd argument of the input to the terminal and sets it as the file input. A single function, *'read_input'* is run three times with the input file as its argument, and with a secondary numerical input argument (from 1-3) that designates which delimiter block is to be focussed on. This function opens the file in read mode. A conditional check sets the start/end delimiters to two variables, so d1 and d2 may be "<CIRCUIT>" and "</CIRCUIT>" for example. The function then cycles through each line in the input file. If a line matches the first delimiter, the lines begin to be checked for comment designators by splitting the first word in the line and checking for a '#.' Any non-comment lines are stored in an array. If at any point the second delimiter is read, the array is returned and the function terminates. There will be three array outputs, each storing the information between the three delimiter sets which henceforth we will call data1, data2 and data3 for each of the delimiter blocks.

The information for the first section, <CIRCUIT>, is passed into a processing function *'process_circuit.'* A numpy array of size 4xN filled with zeroes is created where N is the length of data1, i.e. the number of impedances in the circuit. We then loop over every value in data1. Assuming no errors when reading, it is assumed there are three sections of data for each impedance - node1, node2 and the component value (a check is done with 'isalpha' to ensure these values are numerical). Therefore, the string input is split at its spaces leaving us with a 3x1 array. Each element in this array is split further at each = symbol. The second element for each is then stored in the 4xN array as shown in Fig.1. The component values will later be needed to calculate the impedance so it's important to differentiate between component types. I've done this by adding the fourth column to store a number that corresponds to type:

1 = Resistor (Ohms), 2 = Capacitor, 3 = Inductor, 4 = Resistor (Siemens)

The impedances must be sorted using the node information where any shunt resistor comes before a series resistor with the same value of node 1. This can be done easily with *np.argsort* which allows us to sort by moving entire rows - we sort the second column such that zeroes (shunts) appear at the top of the array. We then sort again by the first column so that the shunts and series are organised as described.

Node 1 Value	Node 2 Value	R / L / C	Impedance Type
1	2	220.0e-6	2
2	0	0.01	4
2	3	50	1

Fig.1. - An example of what my 4xN array might look like. Here, there are three impedances. The first is connected between node 1 and 2, has a component value of 220u and is a capacitor.

Storing source data:
The information in data2 is then processed in the function *'process_source.'* The input data will always be the voltage source / source resistance, load resistance and frequency information. These are split and stored as x, y and z respectively. A flag is used to detect if the frequency spacing should be logarithmic. 'isalpha' checks are done. RL is easily extracted as the second array element once split at the = symbol which is then turned into a float. The voltage source and source resistance are split twice again - once at the space, then both halves are split at their = symbols. The second values are checked to see if adjustment is needed from Siemens to Ohms, then stored. The frequencies are put into a list using the linspace or logspace functions that creates a defined number of spaced values from a defined starting value to an end value. The source information, RL and frequency list are returned. The first and second values in the source information are then stored as VS and ZS.

Storing output data:
The output information in data3 is fed into *'process_output.'* Three empty arrays are initialised to store their namesakes - 'values', 'units' and 'exponents.' For each line in output, the value and unit are split. The function then attempts to append the unit to the units array, but if the unit was not included, a library of default units is referenced against the corresponding value (i.e. Vin will automatically be given V as its unit). The values themselves are appended to the values array. Once this loop concludes, another begins for the same number of iterations. Each of the units are broken into a letter-by-letter list where the first letter is checked against another library. If this first letter is p,n,u,m,k,M,G or d (where d will represent decibels), an appropriate flag is added to the exponents array. All three arrays are then returned.

Values	Vin	Vout	Iin	Iout	Pin
Units	V	dBV	uA	A	W
Exponents	None	dB	u	None	None

Fig.2. - The three separate arrays. If Vin had no unit with it, V would be filled in automatically. Vout and Iin have modifiers which are stored in Exponents

ABCD Matrices & Data Formatting
I decided to use one large function, *'calculate_answers'* to correctly format the calculated data with a number of sub-functions within it performing the calculations. I could not figure out a way to vectorise the process. Therefore, calculation of the matrices and output values is done for each frequency which dramatically increases the runtime to a best case of n^2 operations and makes solving the large 100 - 400 impedance circuits unfeasible in a reasonable amount of time.

The function first initialises an array, AnswerMatrix, in which all of the finalised data will be stored for writing, then arrays value_list and unit_list which come preloaded with the "Freq" and "Hz" value and unit in them respectively. The rest of the function operates in a for loop of length (number of frequencies + 2) with the extra two iterations are tailored specifically to add value and unit information to AnswerMatrix.

Each value is appended twice, once for the real and once for the imaginary component designators. However, if the exponent for a given set of columns is given, the designators are changed to their magnitude and phase variations

Re(Vin)	Im(Vin)	Re(Iin)	Im(Iin)	|Pin|	/_Pin
V	V	mA	mA	dBW	dBW

Fig.3. - The first and second lines of the AnswerMatrix are represented as shown.

(Freq and Hz will always be in the first column, this is merely for illustrative purposes)

For the rest of the iterations, frequency is fetched from the earlier lin/logspace array. This frequency and the 4xN node data matrix are passed into the *'ABCD_matrices'* function. Angular frequency, represented in-code as *w*, is calculated as (2 * pi * Hertzian frequency). For each node, an ABCD matrix is created. Here, the impedance-type flags come into play. If a resistor is flagged, Z is set as its resistance in ohms.

If a capacitor or inductor are flagged, the impedance is instead:
$Z = -jX_C$ where X_C is the capacitor's reactance, 1 / (wC)
$Z = jX_L$ where X_L is the inductor's reactance, wL

Node 2 is checked. If a series component is detected (non-zero for node 2), the regular ABCD matrix is created where A = 1, B = Z, C = 0 and D = 1. If instead a zero is found, the matrix for a shunt resistor is created where B = 0 and C = 1/Z. The matrices for each component are then appended to a list in order.

$$\left[\ \overset{Z = -jX_C}{\begin{bmatrix} 1 & Z \\ 0 & 1 \end{bmatrix}},\ \overset{Z = R(\Omega)}{\begin{bmatrix} 1 & 0 \\ 1/Z & 1 \end{bmatrix}},\ \overset{Z = R(\Omega)}{\begin{bmatrix} 1 & Z \\ 0 & 1 \end{bmatrix}}\ \right]$$

Fig.4. - If we take the examples from Fig.1, the capacitor and two resistors, the matrix list will look like this. The impedance, Z, for the first matrix will be a complex number:
(-j / 2 * pi * F * 220e-06)

This list is returned and the function *np.linalg.multi_dot* is used to automatically perform matrix multiplication on every matrix in the list with no need for additional functions or loops at no cost to performance.

Calculations

Now we have a single matrix of An, Bn etc. we can calculate the circuit values. I used two unique functions to separate the equations that involve the ABCD matrix and those that do not, named *'impedances'* and *'parameters'* respectively.

The equations used in ‘impedances’. Where A, B, C and D represent An, Bn etc. are as follows:

Zin	=	(A * RL + B) / (C * RL + D)
Zout	=	(D * ZS + B) / (C * ZS + A)
Av	=	1 / (A + (B * (1 / RL)))
Ai	=	1 / (C * RL + D)

The equations used in ‘parameters’ are as follows:

Iin	=	VS / (ZS + Zin)	
Vin	=	Zin * Iin	
Vout	=	Av * Vin	
Iout	=	Vout / RL	
Pin	=	Vin * np.conj(Iin)	*(where the complex conjugate of input current is used)*
Pout	=	Vout * np.conj(Iout)	*(where the complex conjugate of output current is used)*

The real and imaginary parts of these values are then matched to their respective column headers in AnswerMatrix and given the %12.3e format (12 characters, 3d.p, scientific form).
If a value must be given in decibels, the calculations are as follows:

Magnitude	=	$20\log_{10}\left(\sqrt{re(X)^2 + \ im(X)^2}\right)$
Phase	=	$\arctan\left(im(X)\ /\ re(X)\right)$

If any exponents are present, all values of the column it’s in are multiplied by an adjustment value ranging from 10^12 for p to 10^-9 for G. Once all frequencies are used, every element in the matrix is stripped of whitespace using *.strip.* The first column is right justified to 10 characters and every other column is justified to 11. Each row is then written to sys.arg[2] opened in write mode.

There is a subsidiary function, *‘error_output’*, that is called if an error is detected in the input data. Ths function writes an empty csv file then terminates the program.

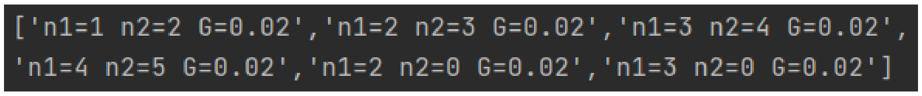
Function Testing

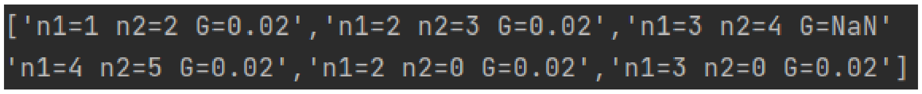
All function testing was done in isolated python files using modified test inputs.

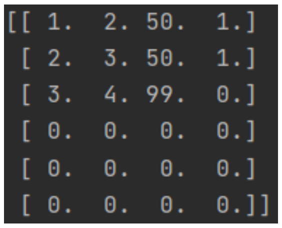
Read_input:
I tested this function using a variety of modified test nets adapted from the original model tests. These included every fail case I could think of: The delimiters not being present or wrong, the inclusion of comments, comments being improperly formatted with no space directly after, the file being empty, the delimiters being in the wrong order.

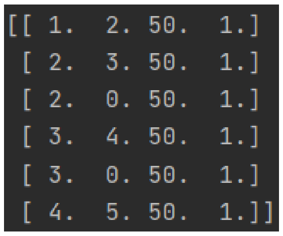
All tests were passed. The input must be formatted correctly for the function to not throw an error. Strictly speaking, this function doesn’t do much beyond saving the input contents so not much testing was required.

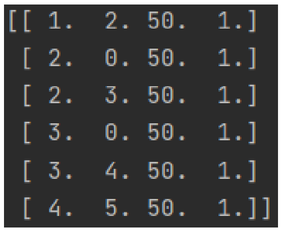
Process_circuit
Getting this function right was paramount. The node information had to be properly split and sorted, else none of the final output data would be correct. The procedure I used was to throw an arbitrary input of 6 0.02Siemen resistors through it, with two shunt resistors placed out of order at the end. This would test the ordering strategy as the shunts had to be placed before their respective series impedances at the corresponding node1. I also edited the input to contain non-numerical data, testing for error handling. (Images captured in Pycharm CE)


```
['n1=1 n2=2 G=0.02','n1=2 n2=3 G=0.02','n1=3 n2=4 G=0.02',
'n1=4 n2=5 G=0.02','n1=2 n2=0 G=0.02','n1=3 n2=0 G=0.02']
```
Test Data 1

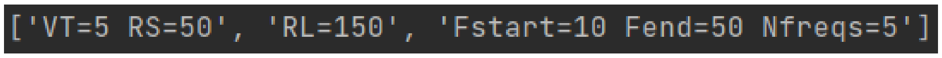

```
['n1=1 n2=2 G=0.02','n1=2 n2=3 G=0.02','n1=3 n2=4 G=NaN'
'n1=4 n2=5 G=0.02','n1=2 n2=0 G=0.02','n1=3 n2=0 G=0.02']
```
Test Data 2

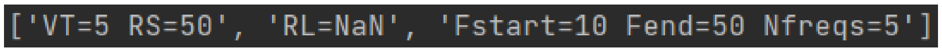

```
[[ 1.  2. 50.  1.]
 [ 2.  3. 50.  1.]
 [ 3.  4. 99.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```
I used the .isalpha function as error detection. If any value that was to be saved to the data array was not a number, an error would be thrown. In testing I represented this as a 99 but in the real case, the *error_output* function would be called instead. This test also proved that the automatic conversion from Siemens to Ohms worked as expected.


```
[[ 1.  2. 50.  1.]
 [ 2.  3. 50.  1.]
 [ 2.  0. 50.  1.]
 [ 3.  4. 50.  1.]
 [ 3.  0. 50.  1.]
 [ 4.  5. 50.  1.]]
```
In my first functionality tests, the shunt impedances were being placed ahead of the series impedances. From a circuit perspective this makes no sense and would affect the result of the matrix multiplication. I was using selection sort to shift the 2nd column and move every row value alongside. Evidently I had to adjust my strategy.

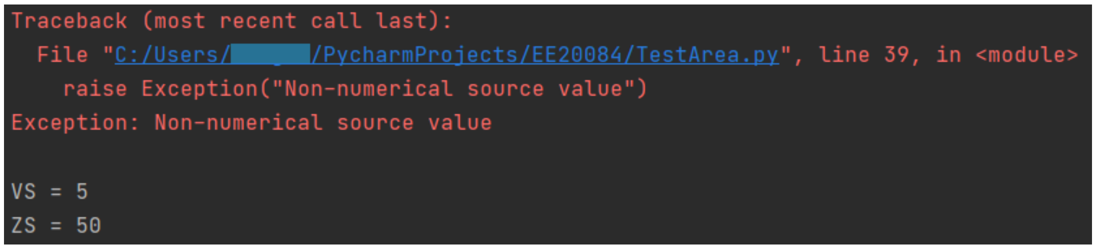

```
[[ 1.  2. 50.  1.]
 [ 2.  0. 50.  1.]
 [ 2.  3. 50.  1.]
 [ 3.  0. 50.  1.]
 [ 3.  4. 50.  1.]
 [ 4.  5. 50.  1.]]
```
It was at this point I had discovered argsort and put it to use by sorting the second column first. This would put all of the shunt resistances at the top of the array so that when argsort is again applied to the first column, every impedance is ordered as it should be, as this test example shows.

Process_source
Source processing is very similar to circuit processing. Therefore, I performed similar tests with a sample input that on occasion contained erroneous data. It is assumed that if any second halves of split() are non-numerical an error is present in the data and therefore the program terminates, outputting the empty csv file. In this case I raised an exception as a placeholder. If the source resistance is given in Siemens, it will be converted to Ohms.

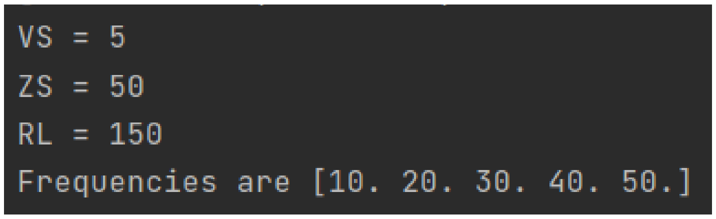

```
['VT=5 RS=50', 'RL=150', 'Fstart=10 Fend=50 Nfreqs=5']
```
Test Data 1


```
['VT=5 RS=50', 'RL=NaN', 'Fstart=10 Fend=50 Nfreqs=5']
```
Test Data 2

In the event that a non-numerical value is registered, an exception as shown below is raised. The non-numerical is RL so the program terminates at this point



```
Traceback (most recent call last):
  File "C:/Users/[redacted]/PycharmProjects/EE20084/TestArea.py", line 39, in <module>
    raise Exception("Non-numerical source value")
Exception: Non-numerical source value

VS = 5
ZS = 50
```

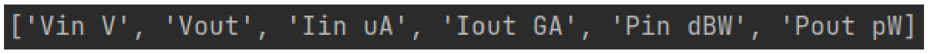


```
VS = 5
ZS = 50
RL = 150
Frequencies are [10. 20. 30. 40. 50.]
```

Provided that the input is formatted as expected, the function will always run smoothly and quickly create a list of frequencies with performance suffering when NFreqs is above ~100,000,000

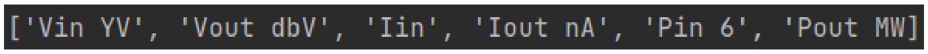
Process_output

Output processing has some unique quirks in that the units need to be automatically filled in if missing and prefixes need to be considered. I tested the function in much the same way as above but with information omitted and prefixes added - including prefixes that must not be considered.



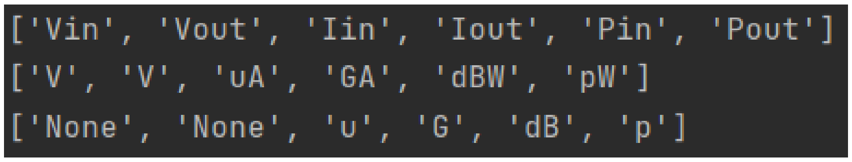
```
['Vin V', 'Vout', 'Iin uA', 'Iout GA', 'Pin dBW', 'Pout pW]
```

Test Data 1



```
['Vin YV', 'Vout dbV', 'Iin', 'Iout nA', 'Pin 6', 'Pout MW]
```

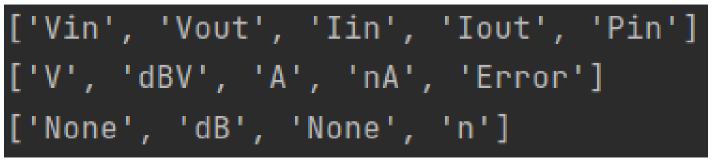
Test Data 2



```
['Vin', 'Vout', 'Iin', 'Iout', 'Pin', 'Pout']
['V', 'V', 'uA', 'GA', 'dBW', 'pW']
['None', 'None', 'u', 'G', 'dB', 'p']
```

From top to bottom, the values, units and exponents returned when processing Test Data 1

Any value in the test data without a unit is automatically assigned one. In this case, Vout was assigned V. As well as this, the exponents for the rest of the units were added to the corresponding index of the exponents array as well. Vin and Vout have no modifiers and the testing reflected this by leaving 'None' in the first two elements.



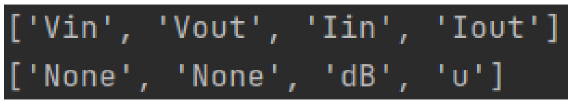
```
['Vin', 'Vout', 'Iin', 'Iout', 'Pin']
['V', 'dBV', 'A', 'nA', 'Error']
['None', 'dB', 'None', 'n']
```

The output for Test Data 2 was promising. It recognised the numeric value and threw an error

As before, the value with no unit and the values with modifiers were recognised. The function spotted the numeric unit and appended 'Error' to the array before printing and terminating the program. Naturally, the final element for values and units, as well as the final two for exponents were lost - this doesn't matter in the final program since it would simply print an empty csv file and terminate, making the output information useless.

Calculate_answers
As this function is relatively large, I broke it up into parts when testing:
1: Top row formatting   2: Matrix creation   3: Outdata formatting and justification

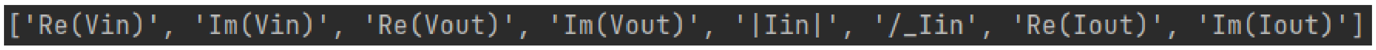
1) The top two rows stand apart from the rest of the AnswerMatrix. The units in row 2 can be directly imported from the Units array but row 1 must accommodate real, imaginary and dB.



```
['Vin', 'Vout', 'Iin', 'Iout']
['None', 'None', 'dB', 'u']
```

As this function comes after the processing stage it can be assumed all values will be correct

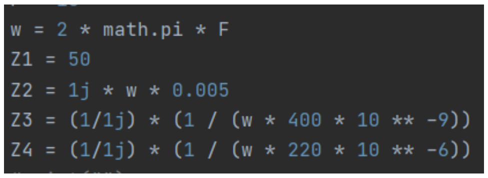
Passing the above values and exponents into the function yielded the following:



```
['Re(Vin)', 'Im(Vin)', 'Re(Vout)', 'Im(Vout)', '|Iin|', '/_Iin', 'Re(Iout)', 'Im(Iout)']
```

The real and imaginary column headers were accurately placed and the only header changed by the exponents array was Iin which instead had the magnitude / phase designators.

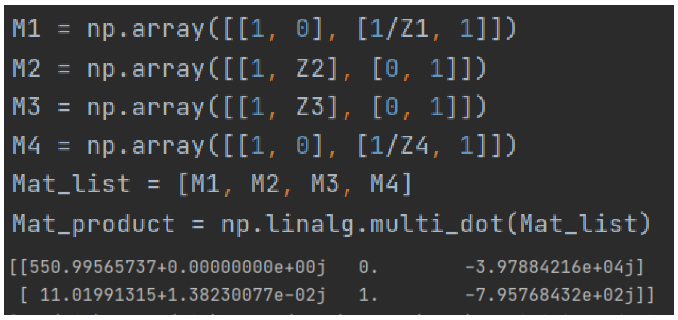
2) I solved a number of sample tests for the ABCD matrix multiplication by hand and with online tools across a range of input frequencies. I was sure before starting that the impedance values were correct, having thoroughly checked them with online impedance calculators.



```
w = 2 * math.pi * F
Z1 = 50
Z2 = 1j * w * 0.005
Z3 = (1/1j) * (1 / (w * 400 * 10 ** -9))
Z4 = (1/1j) * (1 / (w * 220 * 10 ** -6))
```

I used a combination of resistors, capacitors and inductors to ensure that frequency was being factored in and complex values were handled, as well as both shunt and series resistors to cover both matrix configurations.

I worked with many methods to perform matrix multiplication. Numpy's matmul, dot and vdot were promising but could only be used with two matrices at a time meaning I would have to slowly iterate through every impedance, matmul-ing the previous matrix with the current until only one remained. Instead, I found and opted for linalg.multi_dot.



```
M1 = np.array([[1, 0], [1/Z1, 1]])
M2 = np.array([[1, Z2], [0, 1]])
M3 = np.array([[1, Z3], [0, 1]])
M4 = np.array([[1, 0], [1/Z4, 1]])
Mat_list = [M1, M2, M3, M4]
Mat_product = np.linalg.multi_dot(Mat_list)
[[550.99565737+0.00000000e+00j   0.         -3.97884216e+04j]
 [ 11.01991315+1.38230077e-02j   1.         -7.95768432e+02j]]
```

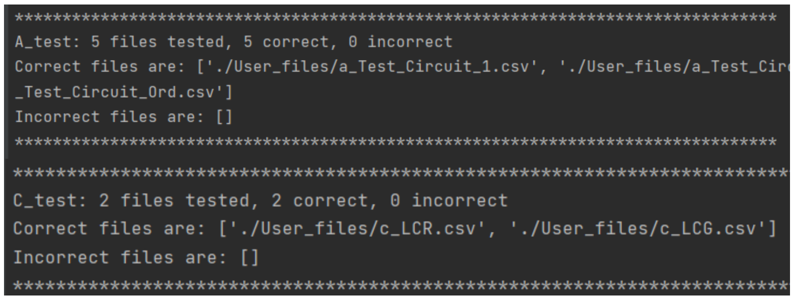
The lines used here for the array creation are exactly as they appear in my code. They were instead predefined as shunt/series. The output matrix was determined to be correct by the Reshish matrix calculator. The matrices have been accurate to ~10^-26 in my testing.

Because the nodes are always ordered correctly, the matrix multiplication will be correct for any input net - assuming the net is formatted correctly.

3) Testing the output formatting took very little effort. By trial and error looking at the model test files I figured out that the first column had to be justified such that it was 10 characters long and every other column such that they were 11 long. Column headers are matched easily with their values using a library so little testing was necessary here.

Full Program Tests

Most of my full program testing was done using version 7 of the AutoTest script with an absolute and relative tolerance of 1e-15.

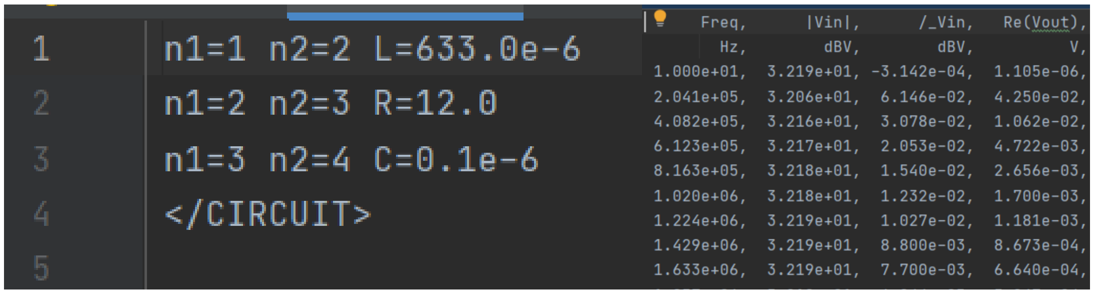


```
********************************************************************************
A_test: 5 files tested, 5 correct, 0 incorrect
Correct files are: ['./User_files/a_Test_Circuit_1.csv', './User_files/a_Test_Cir
_Test_Circuit_Ord.csv']
Incorrect files are: []
********************************************************************************
********************************************************************************
C_test: 2 files tested, 2 correct, 0 incorrect
Correct files are: ['./User_files/c_LCR.csv', './User_files/c_LCG.csv']
Incorrect files are: []
********************************************************************************
```

The program passed tests A through D with flying colors. It cannot pass test E within a reasonable period of time. Though, if we were to extrapolate the results here, I would assume it could complete them given infinite time.

However, I didn't want to take the efficacy of these tests at face value as they did not cover every possible error - or at least the additional errors I could think of. So, I modified them to have missing delimiters, missing values, strings instead of numerical values & decibel inputs.

Even if a delimiter is missing, the corresponding data will still be added to the correct array:

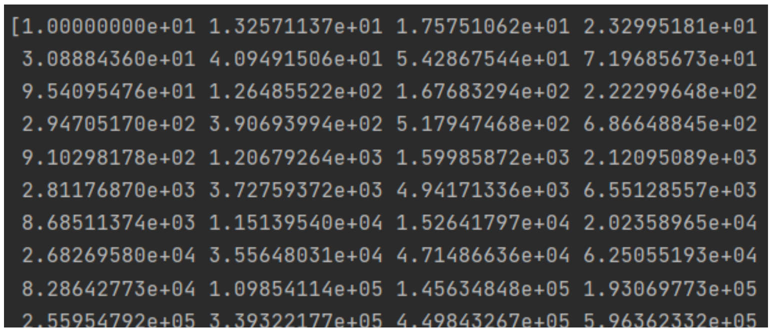


```
n1=1 n2=2 L=633.0e-6
n1=2 n2=3 R=12.0
n1=3 n2=4 C=0.1e-6
</CIRCUIT>
```

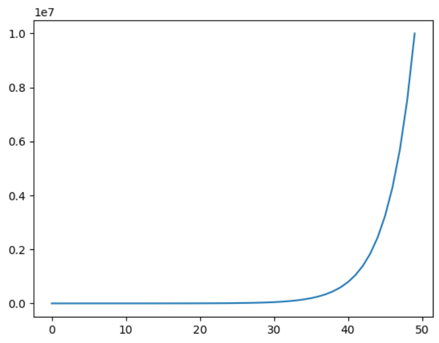
```
      Freq,       |Vin|,       /_Vin,   Re(Vout),
        Hz,         dBV,         dBV,          V,
 1.000e+01,  3.219e+01, -3.142e-04,  1.105e-06,
 2.041e+05,  3.206e+01,  6.146e-02,  4.250e-02,
 4.082e+05,  3.216e+01,  3.078e-02,  1.062e-02,
 6.123e+05,  3.217e+01,  2.053e-02,  4.722e-03,
 8.163e+05,  3.218e+01,  1.540e-02,  2.656e-03,
 1.020e+06,  3.218e+01,  1.232e-02,  1.700e-03,
 1.224e+06,  3.219e+01,  1.027e-02,  1.181e-03,
 1.429e+06,  3.219e+01,  8.800e-03,  8.673e-04,
 1.633e+06,  3.219e+01,  7.700e-03,  6.640e-04,
```

Here, this modified version of c_LCR lacks the <CIRCUIT> delimiter and expects decibels for Vin. Both exceptions are handled and the data is presented regardless.

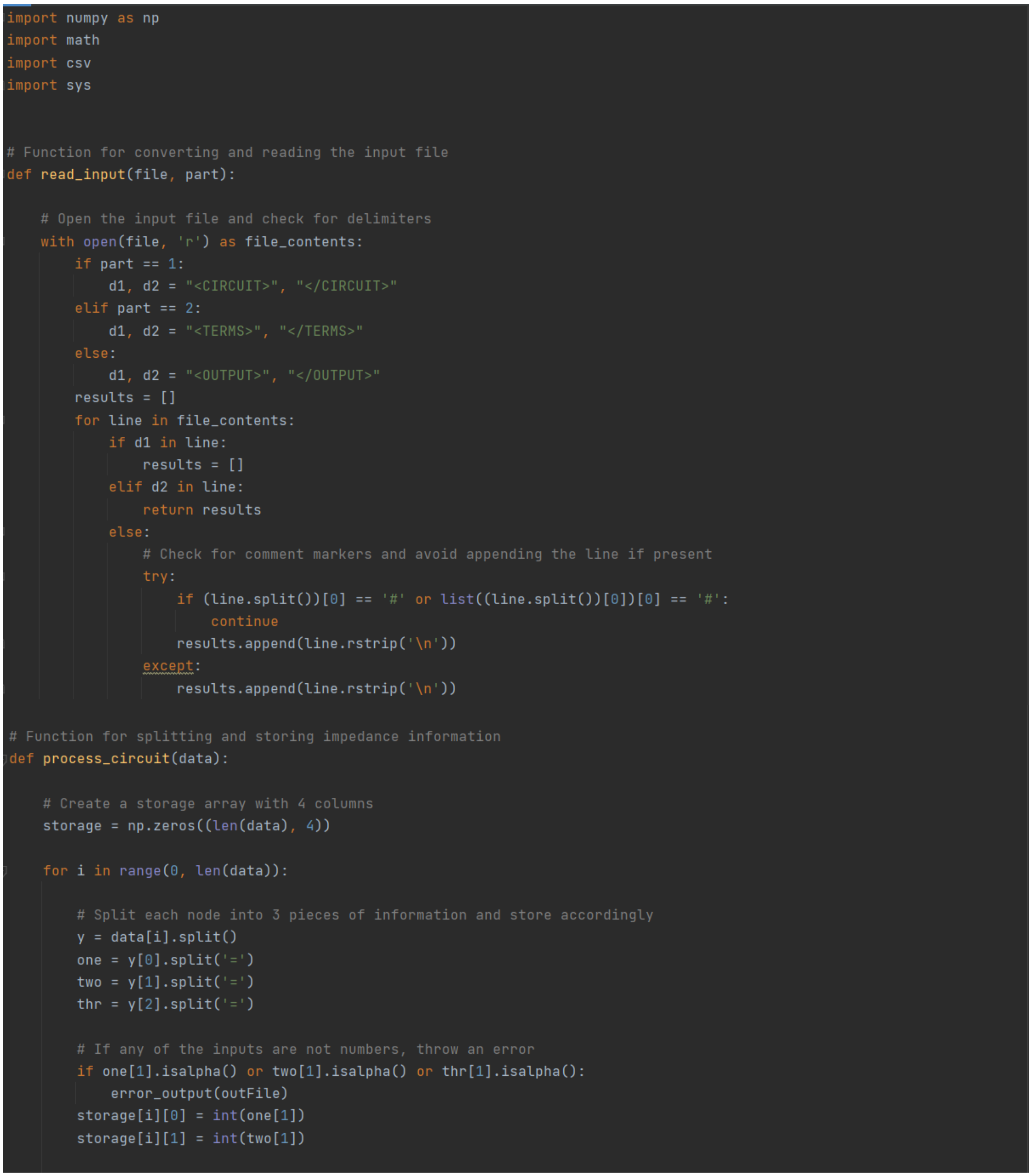
The logarithmic frequency was calculated with *(math.floor(math.log10(abs(X))))* where X is Fstart and Fend. These values were fed into np.logspace with the number of frequencies which produced the following results, clearly showing logarithmic growth in frequency. This can be interchanged with the linspace function and the program does not suffer any problems:



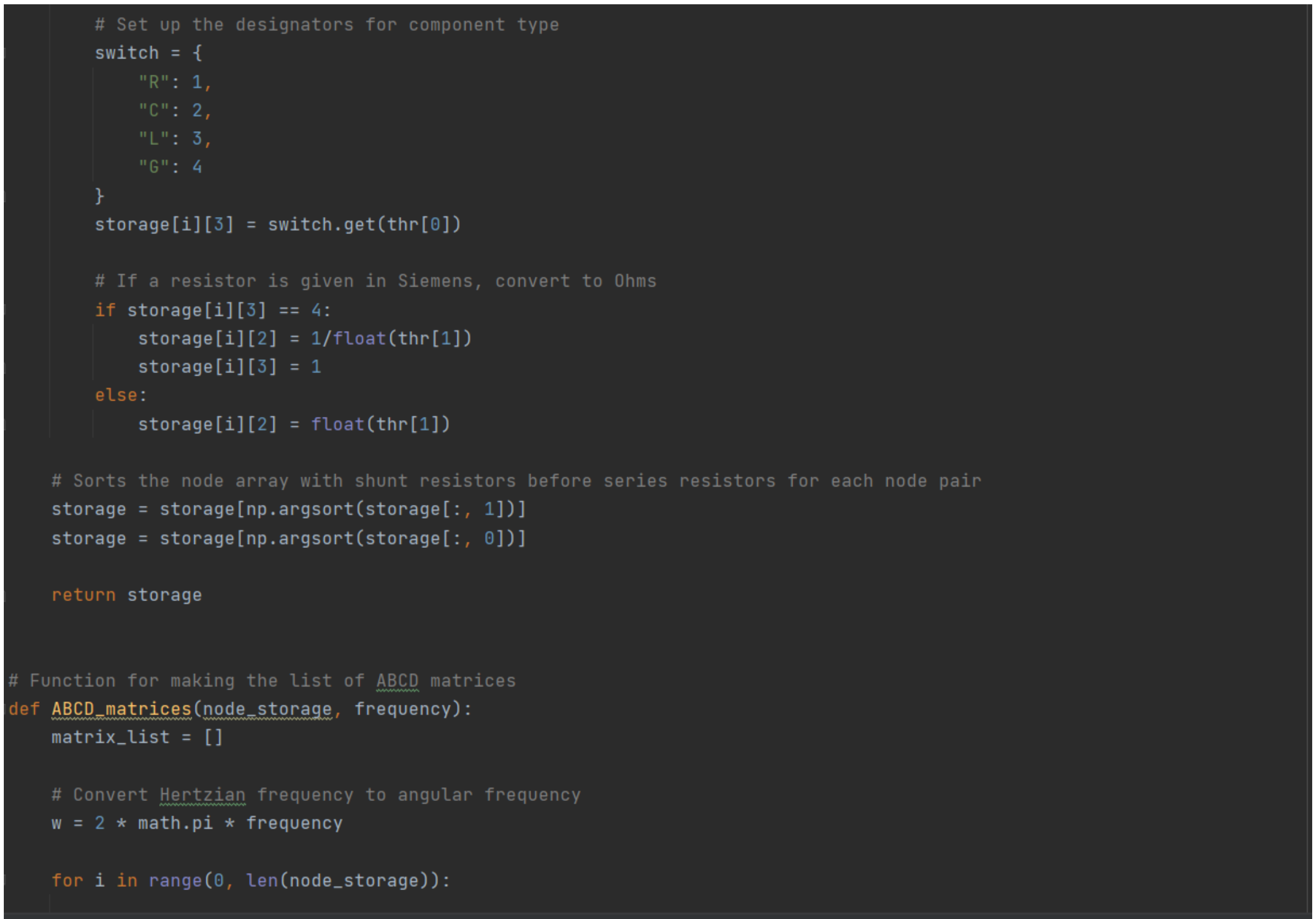
```
[1.00000000e+01 1.32571137e+01 1.75751062e+01 2.32995181e+01
 3.08884360e+01 4.09491506e+01 5.42867544e+01 7.19685673e+01
 9.54095476e+01 1.26485522e+02 1.67683294e+02 2.22299648e+02
 2.94705170e+02 3.90693994e+02 5.17947468e+02 6.86648845e+02
 9.10298178e+02 1.20679264e+03 1.59985872e+03 2.12095089e+03
 2.81176870e+03 3.72759372e+03 4.94171336e+03 6.55128557e+03
 8.68511374e+03 1.15139540e+04 1.52641797e+04 2.02358965e+04
 2.68269580e+04 3.55648031e+04 4.71486636e+04 6.25055193e+04
 8.28642773e+04 1.09854114e+05 1.45634848e+05 1.93069773e+05
 2.55954792e+05 3.39322177e+05 4.49843267e+05 5.96362332e+05
```



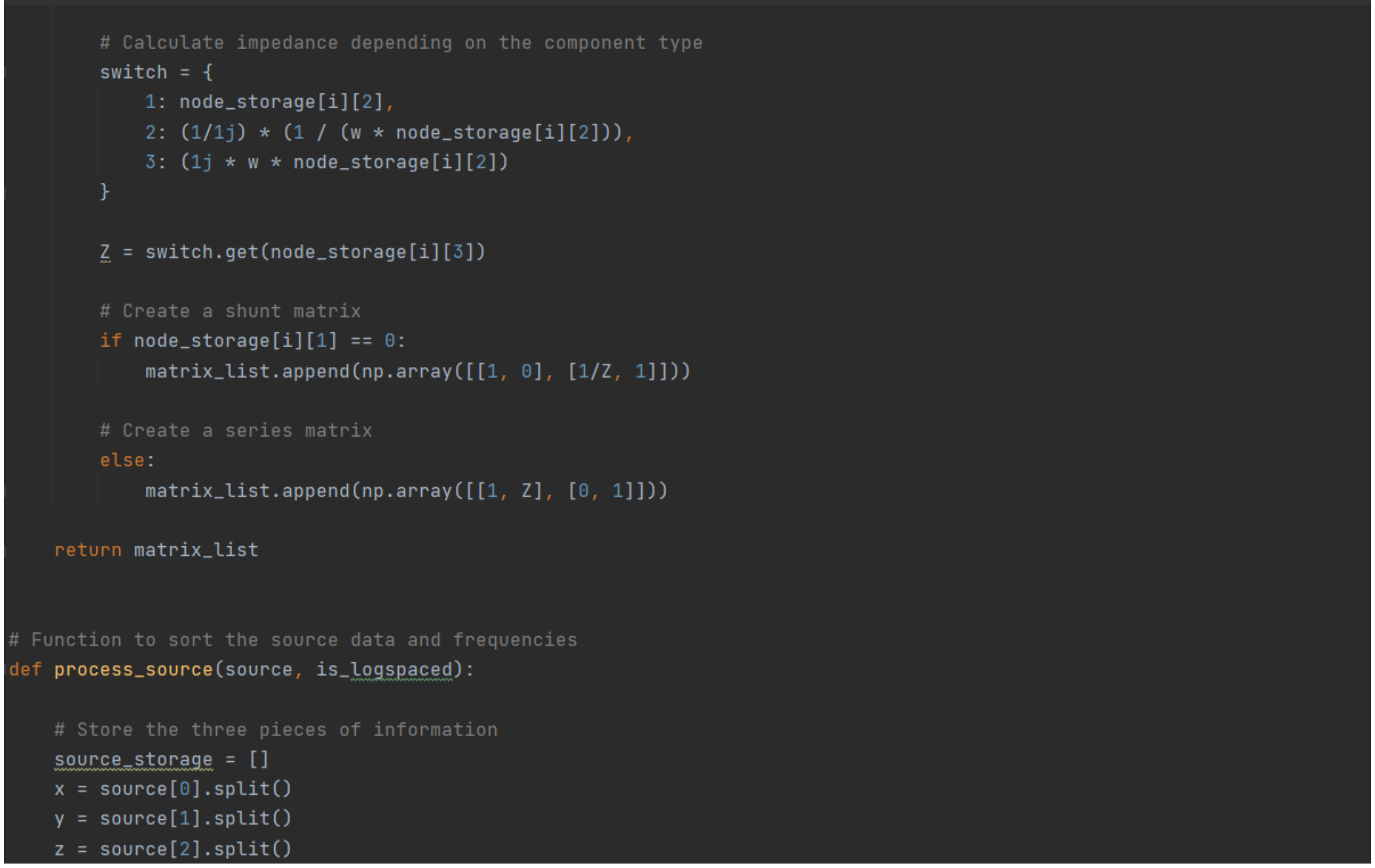
Appendix



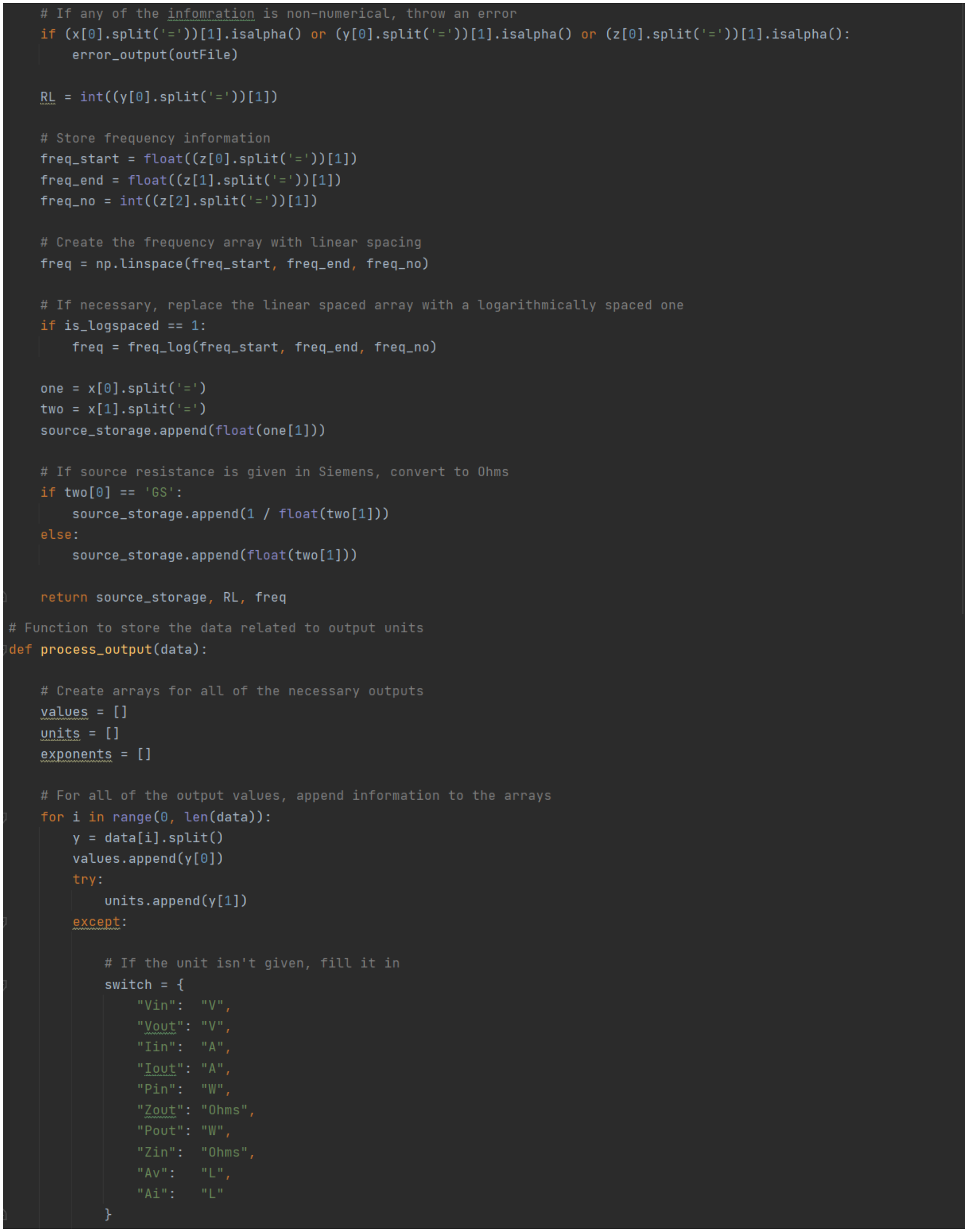
```python
import numpy as np
import math
import csv
import sys


# Function for converting and reading the input file
def read_input(file, part):

    # Open the input file and check for delimiters
    with open(file, 'r') as file_contents:
        if part == 1:
            d1, d2 = "<CIRCUIT>", "</CIRCUIT>"
        elif part == 2:
            d1, d2 = "<TERMS>", "</TERMS>"
        else:
            d1, d2 = "<OUTPUT>", "</OUTPUT>"
        results = []
        for line in file_contents:
            if d1 in line:
                results = []
            elif d2 in line:
                return results
            else:
                # Check for comment markers and avoid appending the line if present
                try:
                    if (line.split())[0] == '#' or list((line.split())[0])[0] == '#':
                        continue
                    results.append(line.rstrip('\n'))
                except:
                    results.append(line.rstrip('\n'))

# Function for splitting and storing impedance information
def process_circuit(data):

    # Create a storage array with 4 columns
    storage = np.zeros((len(data), 4))

    for i in range(0, len(data)):

        # Split each node into 3 pieces of information and store accordingly
        y = data[i].split()
        one = y[0].split('=')
        two = y[1].split('=')
        thr = y[2].split('=')

        # If any of the inputs are not numbers, throw an error
        if one[1].isalpha() or two[1].isalpha() or thr[1].isalpha():
            error_output(outFile)
        storage[i][0] = int(one[1])
        storage[i][1] = int(two[1])
```



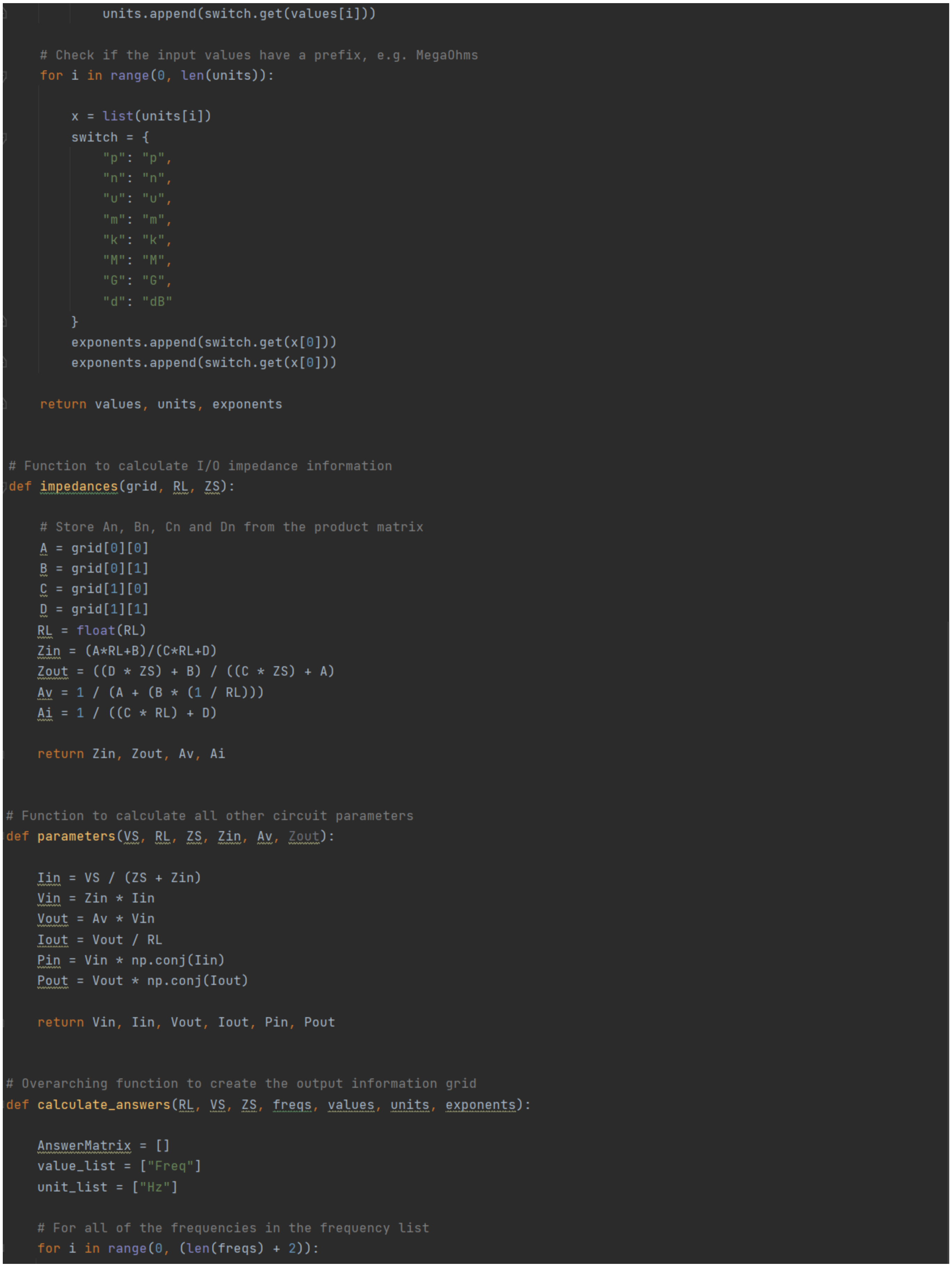
```python
        # Set up the designators for component type
        switch = {
            "R": 1,
            "C": 2,
            "L": 3,
            "G": 4
        }
        storage[i][3] = switch.get(thr[0])

        # If a resistor is given in Siemens, convert to Ohms
        if storage[i][3] == 4:
            storage[i][2] = 1/float(thr[1])
            storage[i][3] = 1
        else:
            storage[i][2] = float(thr[1])

    # Sorts the node array with shunt resistors before series resistors for each node pair
    storage = storage[np.argsort(storage[:, 1])]
    storage = storage[np.argsort(storage[:, 0])]

    return storage


# Function for making the list of ABCD matrices
def ABCD_matrices(node_storage, frequency):
    matrix_list = []

    # Convert Hertzian frequency to angular frequency
    w = 2 * math.pi * frequency

    for i in range(0, len(node_storage)):
```



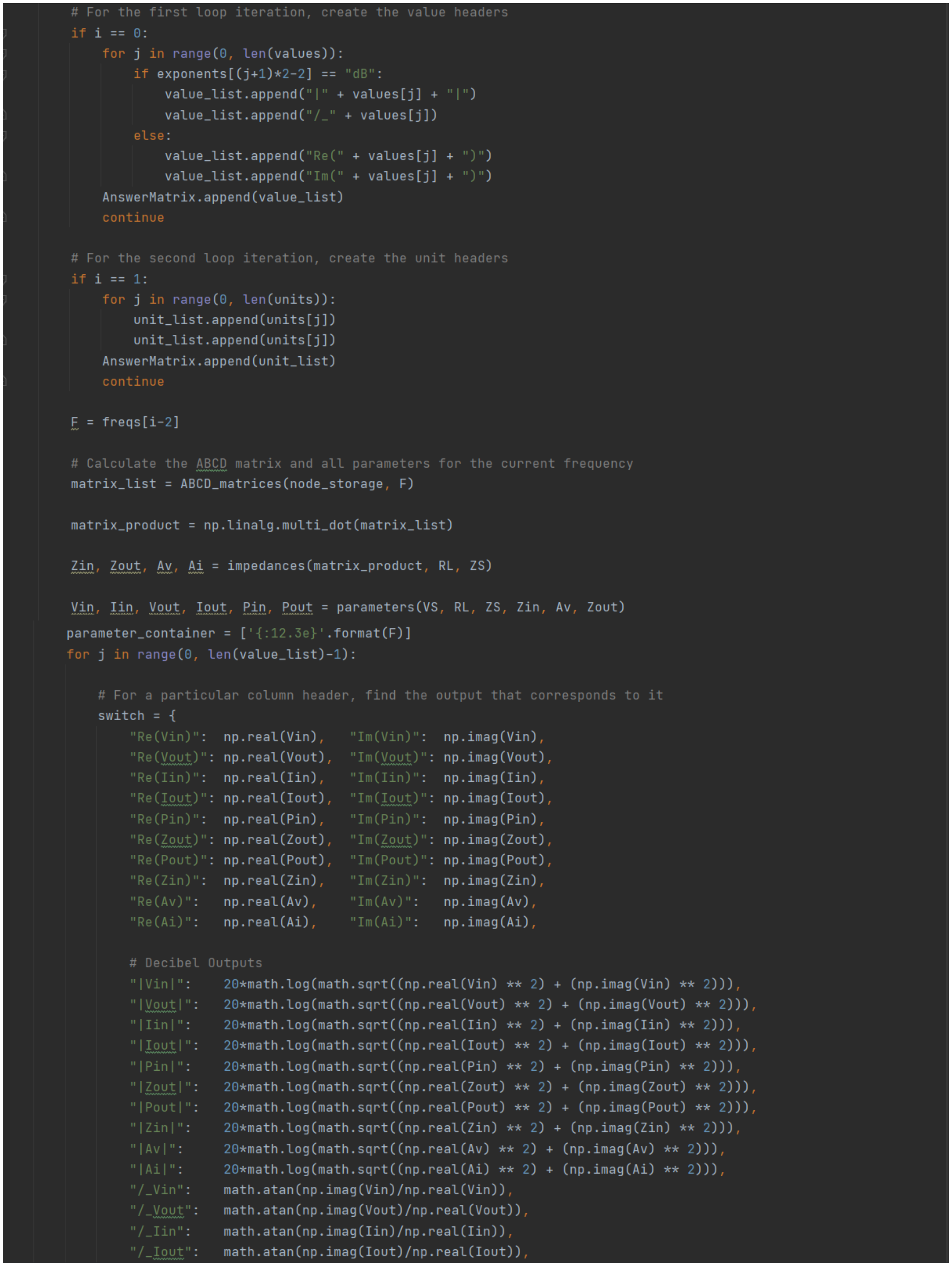
```python
        # Calculate impedance depending on the component type
        switch = {
            1: node_storage[i][2],
            2: (1/1j) * (1 / (w * node_storage[i][2])),
            3: (1j * w * node_storage[i][2])
        }

        Z = switch.get(node_storage[i][3])

        # Create a shunt matrix
        if node_storage[i][1] == 0:
            matrix_list.append(np.array([[1, 0], [1/Z, 1]]))

        # Create a series matrix
        else:
            matrix_list.append(np.array([[1, Z], [0, 1]]))

    return matrix_list


# Function to sort the source data and frequencies
def process_source(source, is_logspaced):

    # Store the three pieces of information
    source_storage = []
    x = source[0].split()
    y = source[1].split()
    z = source[2].split()
```



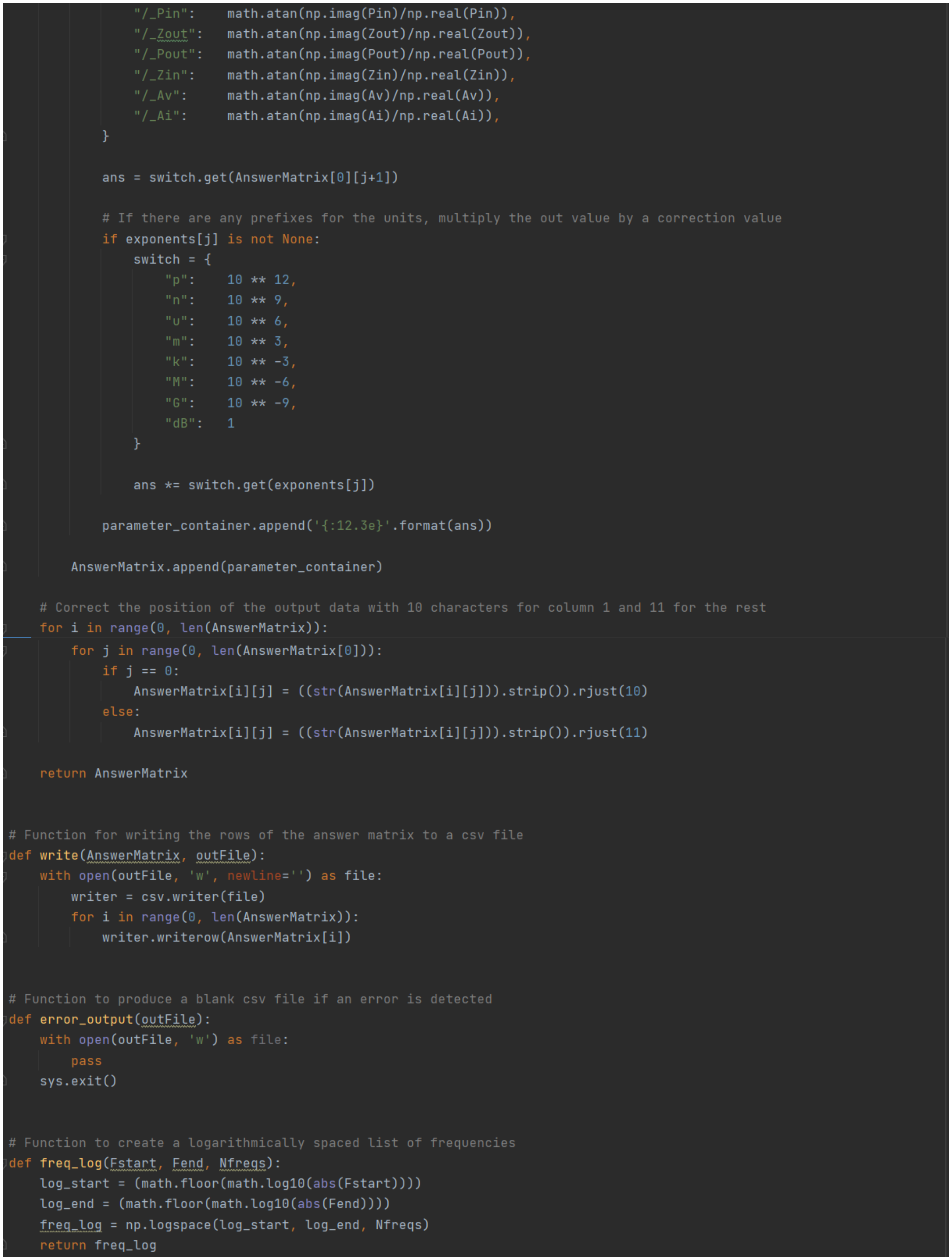
```
    # If any of the infomration is non-numerical, throw an error
    if (x[0].split('='))[1].isalpha() or (y[0].split('='))[1].isalpha() or (z[0].split('='))[1].isalpha():
        error_output(outFile)

    RL = int((y[0].split('='))[1])

    # Store frequency information
    freq_start = float((z[0].split('='))[1])
    freq_end = float((z[1].split('='))[1])
    freq_no = int((z[2].split('='))[1])

    # Create the frequency array with linear spacing
    freq = np.linspace(freq_start, freq_end, freq_no)

    # If necessary, replace the linear spaced array with a logarithmically spaced one
    if is_logspaced == 1:
        freq = freq_log(freq_start, freq_end, freq_no)

    one = x[0].split('=')
    two = x[1].split('=')
    source_storage.append(float(one[1]))

    # If source resistance is given in Siemens, convert to Ohms
    if two[0] == 'GS':
        source_storage.append(1 / float(two[1]))
    else:
        source_storage.append(float(two[1]))

    return source_storage, RL, freq

# Function to store the data related to output units
def process_output(data):

    # Create arrays for all of the necessary outputs
    values = []
    units = []
    exponents = []

    # For all of the output values, append information to the arrays
    for i in range(0, len(data)):
        y = data[i].split()
        values.append(y[0])
        try:
            units.append(y[1])
        except:

            # If the unit isn't given, fill it in
            switch = {
                "Vin":  "V",
                "Vout": "V",
                "Iin":  "A",
                "Iout": "A",
                "Pin":  "W",
                "Zout": "Ohms",
                "Pout": "W",
                "Zin":  "Ohms",
                "Av":   "L",
                "Ai":   "L"
            }
```



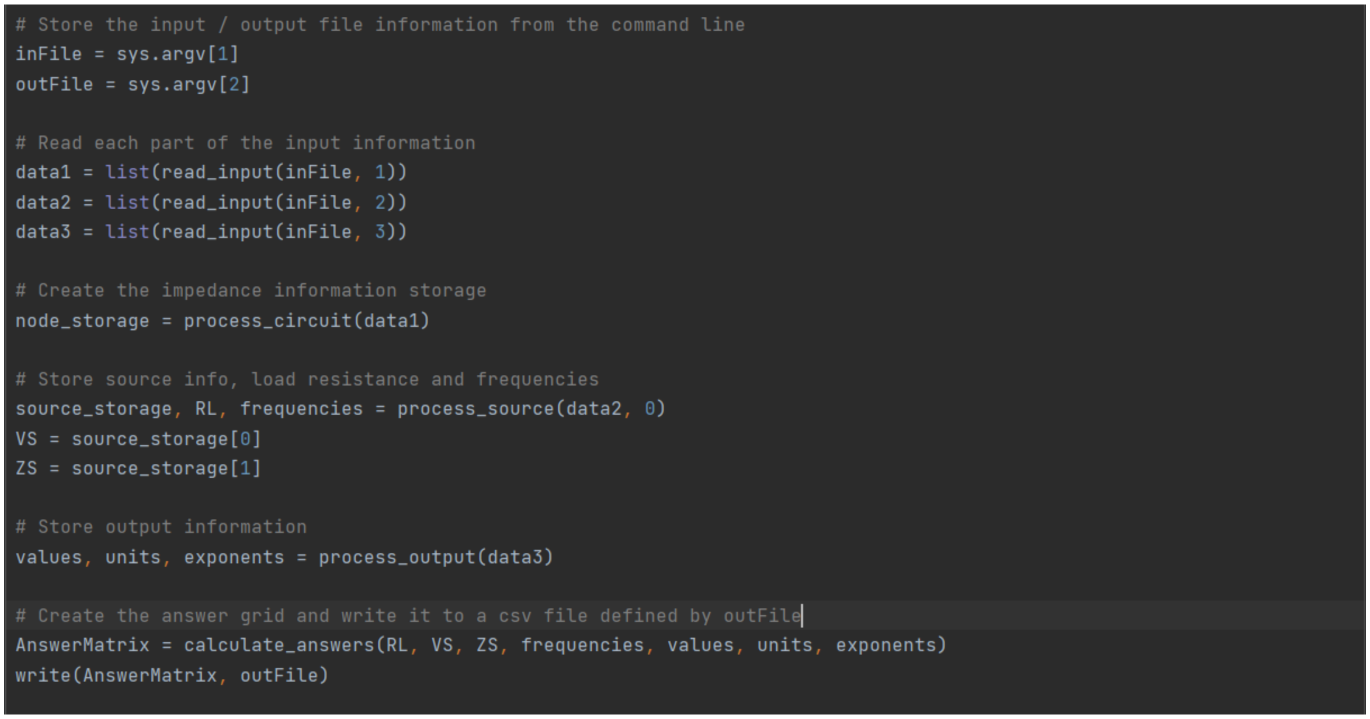
```
            units.append(switch.get(values[i]))

    # Check if the input values have a prefix, e.g. MegaOhms
    for i in range(0, len(units)):

        x = list(units[i])
        switch = {
            "p": "p",
            "n": "n",
            "u": "u",
            "m": "m",
            "k": "k",
            "M": "M",
            "G": "G",
            "d": "dB"
        }
        exponents.append(switch.get(x[0]))
        exponents.append(switch.get(x[0]))

    return values, units, exponents


# Function to calculate I/O impedance information
def impedances(grid, RL, ZS):

    # Store An, Bn, Cn and Dn from the product matrix
    A = grid[0][0]
    B = grid[0][1]
    C = grid[1][0]
    D = grid[1][1]
    RL = float(RL)
    Zin = (A*RL+B)/(C*RL+D)
    Zout = ((D * ZS) + B) / ((C * ZS) + A)
    Av = 1 / (A + (B * (1 / RL)))
    Ai = 1 / ((C * RL) + D)

    return Zin, Zout, Av, Ai


# Function to calculate all other circuit parameters
def parameters(VS, RL, ZS, Zin, Av, Zout):

    Iin = VS / (ZS + Zin)
    Vin = Zin * Iin
    Vout = Av * Vin
    Iout = Vout / RL
    Pin = Vin * np.conj(Iin)
    Pout = Vout * np.conj(Iout)

    return Vin, Iin, Vout, Iout, Pin, Pout


# Overarching function to create the output information grid
def calculate_answers(RL, VS, ZS, freqs, values, units, exponents):

    AnswerMatrix = []
    value_list = ["Freq"]
    unit_list = ["Hz"]

    # For all of the frequencies in the frequency list
    for i in range(0, (len(freqs) + 2)):
```



```python
# For the first loop iteration, create the value headers
if i == 0:
    for j in range(0, len(values)):
        if exponents[(j+1)*2-2] == "dB":
            value_list.append("|" + values[j] + "|")
            value_list.append("/_" + values[j])
        else:
            value_list.append("Re(" + values[j] + ")")
            value_list.append("Im(" + values[j] + ")")
    AnswerMatrix.append(value_list)
    continue

# For the second loop iteration, create the unit headers
if i == 1:
    for j in range(0, len(units)):
        unit_list.append(units[j])
        unit_list.append(units[j])
    AnswerMatrix.append(unit_list)
    continue

F = freqs[i-2]

# Calculate the ABCD matrix and all parameters for the current frequency
matrix_list = ABCD_matrices(node_storage, F)

matrix_product = np.linalg.multi_dot(matrix_list)

Zin, Zout, Av, Ai = impedances(matrix_product, RL, ZS)

Vin, Iin, Vout, Iout, Pin, Pout = parameters(VS, RL, ZS, Zin, Av, Zout)
parameter_container = ['{:12.3e}'.format(F)]
for j in range(0, len(value_list)-1):

    # For a particular column header, find the output that corresponds to it
    switch = {
        "Re(Vin)":  np.real(Vin),   "Im(Vin)":  np.imag(Vin),
        "Re(Vout)": np.real(Vout),  "Im(Vout)": np.imag(Vout),
        "Re(Iin)":  np.real(Iin),   "Im(Iin)":  np.imag(Iin),
        "Re(Iout)": np.real(Iout),  "Im(Iout)": np.imag(Iout),
        "Re(Pin)":  np.real(Pin),   "Im(Pin)":  np.imag(Pin),
        "Re(Zout)": np.real(Zout),  "Im(Zout)": np.imag(Zout),
        "Re(Pout)": np.real(Pout),  "Im(Pout)": np.imag(Pout),
        "Re(Zin)":  np.real(Zin),   "Im(Zin)":  np.imag(Zin),
        "Re(Av)":   np.real(Av),    "Im(Av)":   np.imag(Av),
        "Re(Ai)":   np.real(Ai),    "Im(Ai)":   np.imag(Ai),

        # Decibel Outputs
        "|Vin|":    20*math.log(math.sqrt((np.real(Vin) ** 2) + (np.imag(Vin) ** 2))),
        "|Vout|":   20*math.log(math.sqrt((np.real(Vout) ** 2) + (np.imag(Vout) ** 2))),
        "|Iin|":    20*math.log(math.sqrt((np.real(Iin) ** 2) + (np.imag(Iin) ** 2))),
        "|Iout|":   20*math.log(math.sqrt((np.real(Iout) ** 2) + (np.imag(Iout) ** 2))),
        "|Pin|":    20*math.log(math.sqrt((np.real(Pin) ** 2) + (np.imag(Pin) ** 2))),
        "|Zout|":   20*math.log(math.sqrt((np.real(Zout) ** 2) + (np.imag(Zout) ** 2))),
        "|Pout|":   20*math.log(math.sqrt((np.real(Pout) ** 2) + (np.imag(Pout) ** 2))),
        "|Zin|":    20*math.log(math.sqrt((np.real(Zin) ** 2) + (np.imag(Zin) ** 2))),
        "|Av|":     20*math.log(math.sqrt((np.real(Av) ** 2) + (np.imag(Av) ** 2))),
        "|Ai|":     20*math.log(math.sqrt((np.real(Ai) ** 2) + (np.imag(Ai) ** 2))),
        "/_Vin":    math.atan(np.imag(Vin)/np.real(Vin)),
        "/_Vout":   math.atan(np.imag(Vout)/np.real(Vout)),
        "/_Iin":    math.atan(np.imag(Iin)/np.real(Iin)),
        "/_Iout":   math.atan(np.imag(Iout)/np.real(Iout)),
```



```python
                "/_Pin":    math.atan(np.imag(Pin)/np.real(Pin)),
                "/_Zout":   math.atan(np.imag(Zout)/np.real(Zout)),
                "/_Pout":   math.atan(np.imag(Pout)/np.real(Pout)),
                "/_Zin":    math.atan(np.imag(Zin)/np.real(Zin)),
                "/_Av":     math.atan(np.imag(Av)/np.real(Av)),
                "/_Ai":     math.atan(np.imag(Ai)/np.real(Ai)),
            }

            ans = switch.get(AnswerMatrix[0][j+1])

            # If there are any prefixes for the units, multiply the out value by a correction value
            if exponents[j] is not None:
                switch = {
                    "p":    10 ** 12,
                    "n":    10 ** 9,
                    "u":    10 ** 6,
                    "m":    10 ** 3,
                    "k":    10 ** -3,
                    "M":    10 ** -6,
                    "G":    10 ** -9,
                    "dB":   1
                }

                ans *= switch.get(exponents[j])

            parameter_container.append('{:12.3e}'.format(ans))

        AnswerMatrix.append(parameter_container)

    # Correct the position of the output data with 10 characters for column 1 and 11 for the rest
    for i in range(0, len(AnswerMatrix)):
        for j in range(0, len(AnswerMatrix[0])):
            if j == 0:
                AnswerMatrix[i][j] = ((str(AnswerMatrix[i][j])).strip()).rjust(10)
            else:
                AnswerMatrix[i][j] = ((str(AnswerMatrix[i][j])).strip()).rjust(11)

    return AnswerMatrix


# Function for writing the rows of the answer matrix to a csv file
def write(AnswerMatrix, outFile):
    with open(outFile, 'w', newline='') as file:
        writer = csv.writer(file)
        for i in range(0, len(AnswerMatrix)):
            writer.writerow(AnswerMatrix[i])


# Function to produce a blank csv file if an error is detected
def error_output(outFile):
    with open(outFile, 'w') as file:
        pass
    sys.exit()


# Function to create a logarithmically spaced list of frequencies
def freq_log(Fstart, Fend, Nfreqs):
    log_start = (math.floor(math.log10(abs(Fstart))))
    log_end = (math.floor(math.log10(abs(Fend))))
    freq_log = np.logspace(log_start, log_end, Nfreqs)
    return freq_log
```



```python
# Store the input / output file information from the command line
inFile = sys.argv[1]
outFile = sys.argv[2]

# Read each part of the input information
data1 = list(read_input(inFile, 1))
data2 = list(read_input(inFile, 2))
data3 = list(read_input(inFile, 3))

# Create the impedance information storage
node_storage = process_circuit(data1)

# Store source info, load resistance and frequencies
source_storage, RL, frequencies = process_source(data2, 0)
VS = source_storage[0]
ZS = source_storage[1]

# Store output information
values, units, exponents = process_output(data3)

# Create the answer grid and write it to a csv file defined by outFile
AnswerMatrix = calculate_answers(RL, VS, ZS, frequencies, values, units, exponents)
write(AnswerMatrix, outFile)
```