Rational Fitting of Transmission Line Functions

Assignment 5

Due: 2021/03/15

1 Introduction

In this assignment, we find an approximation for the frequency dependent characteristic impedance $Z_c(\omega)$ plotted in the previous assignment (the zero sequence) using Bode's asymptotic tracing technique. This produces a rational function approximation of the characteristic impedance which can in turn be converted into an equivalent circuit for use in the frequency dependent (FD) transmission line model. The original plots are superimposed with the resulting approximation for comparison and the differences are discussed.

2 Setup

As suggested in the assignment instructions, the original data was interpolated to create 10 points for every frequency decade, providing a total of 80 points from the original 41 points. This was done using numpy's `interp` function. During this stage, an ω value was calculated for every data point for use in the Python control system library `control`.

The Python `control` library allows one to easily generate Bode plots from a transfer function. It requires that the numerator and denominator be expanded into a polynomials of the form of $C_n s^n + C_{n-1}s^{n-1} + ... + C_2 s^2 + C_1 s^1 + C_0$. The coefficients C_n are placed in two lists, one for the numerator and another for the denominator and passed into the transfer function creator as follows:

```
H = K*control.tf(num_coeffs, den_coeffs)}
```

where `num_coeffs` is a list of the numerator's coefficients and `den_coeffs` is a list of the denominator's coefficients. H is the resulting control function object generated by the `tf` function. The transfer function can also be multiplied by a float constant K, but for this assignment it is kept at 1.0. Since every pole and zero will contribute to the starting magnitude of the transfer function, it was necessary to devise a way for the constant to adapt to changes in the poles and zeros as the tracing procedure took place. This process is explained further in this section. The constant K is defined after this process.

With the transfer function defined, the following call is made:

```
magnitude, phase, frequencies = control.bode_plot(H, interp_w_data)
```

where `magnitude` is a list of the magnitude data points, `phase` is a list of the phase data points, and `frequencies` is a list of the frequencies in Hz. H is the transfer function we created above and `interp_w_data` is a list of the ω values we calculated from the frequency data points, matching those of the original CSV interpreted values.

The coefficients for the transfer function above were obtained programmatically using the Python `SymPy` library, which performs symbolic computation. Each pole and zero was added manually as a frequency value in Hz in either the pole or zero list. The frequency pole and zero lists are then converted to their angular frequency equivalents. These lists are then used to generate the numerator and denominator polynomials. First, they are turned into the factored form of the polynomial in the form of $(s+p_n)(s+p_{n-1})...(s+p_1)(s+p_0)$. `SymPy` then converts this polynomial into its expanded form, for both the numerator and the denominator. From these polynomial, the coefficients are extracted.

$$Z_{eq}(s) = K\frac{(s+z_1)(s+z_2)(s+z_3)(s+z_4)}{(s+p_1)(s+p_2)(s+p_3)(s+p_4)} \tag{1}$$

$$Z_{eq}(s) = K\frac{N_4s^4 + N_3s^3 + N_2s^2 + N_1s^1 + N_0}{D_4s^4 + D_3s^3 + D_2s^2 + D_1s^1 + D_0} \tag{2}$$

$$\tag{3}$$

The numerator and denominator polynomial must also be converted into a series of simple fractions using partial fraction decomposition. This is also performed using `SymPy` using its `apart` function. These simple fraction are required to obtain an equivalent RC network representation, using the following relationships.

$$Z_{eq}(s) = K_0 + \frac{K_1}{s+p_1} + \frac{K_2}{s+p_2} + \frac{K_3}{s+p_3} + \frac{K_4}{s+p_4} \tag{4}$$

$$Z_{eq}(s) = R_0 + \frac{\frac{1}{C_1}}{s+R_1C_1} + \frac{\frac{1}{C_2}}{s+R_2C_2} + \frac{\frac{1}{C_3}}{s+R_3C_3} + \frac{\frac{1}{C_4}}{s+R_4C_4} \tag{5}$$

$$R_0 = K_0 \tag{6}$$

$$R_i = \frac{K_i}{p_i} \tag{7}$$

$$C_i = \frac{1}{K_i} \tag{8}$$

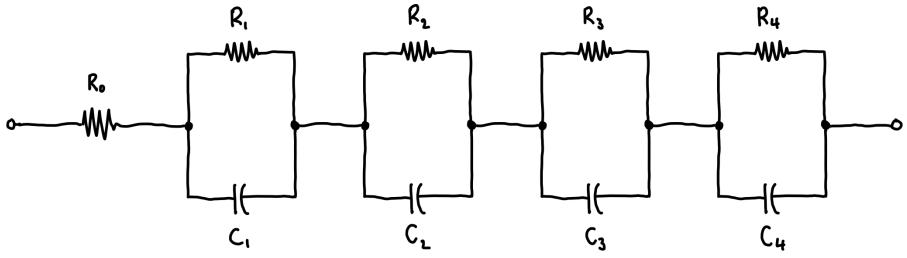

Figure 1: Equivalent R and C network for $Z_c(\omega)$

To account for the varying starting magnitude as poles and zeros were added to the transfer function, the resulting starting magnitude value from the `bode_plot` function was used to determine how much the entire transfer function should be scaled by, by comparing it to the starting magnitude of the original $Z_c(\omega)$ function. Using this scaling factor, the entire Bode plot `magnitude` list was multiplied by this scaling factor to scale it up to the original $Z_c(\omega)$ function. This has the effect of defining the transfer function's K constant. This automatic scaling meant that the starting magnitude of the transfer function was automatically generated to match that of the original function.

With the program now dynamically generating Bode plots superimposed onto the original data using a list of pole and zero frequencies, the curve fitting process took place. The Bode plot was constructed one pole and one zero at a time, checking the resulting plot at every step. This required a fair bit of trial and error until all of the values were obtained. The resulting poles and zeros are outlined in the following section.

3 Results

After some trial and error, the following zeros and poles were chosen for the rational approximations.

Zeros (Hz)	1.8	107	3.5e3	2.27e5
Poles (Hz)	1.5	90	3e3	2e5

Table 1: Poles and zeros chosen for rational approximation

As outlined previously, these were converted into their angular frequency equivalents and used to create the factored form of the the numerator and denominator polynomials used in the transfer function. The following shows the resulting transfer function used for the rational approximation and the resulting R_i and C_i component values for the equivalent network circuit.

$$Z_{eq}(s) = \frac{470.88s^4 + 6.8229 \cdot 10^8 s^3 + 1.5235 \cdot 10^{13} s^2 + 1.0093 \cdot 10^{16} s + 1.0917 \cdot 10^{17}}{s^4 + 1.2761 \cdot 10^6 s^3 + 2.4418 \cdot 10^{10} s^2 + 1.3602 \cdot 10^{13} s + 1.2045 \cdot 10^{14}} \tag{9}$$

$$Z_{eq}(s) = 470.88 + \frac{165.24}{0.11111s + 1.0} + \frac{118.19}{0.0017699s + 1.0} + \frac{88.719}{5.3053 \cdot 10^{-5} s + 1.0} + \frac{63.402}{7.9578 \cdot 10^{-7} s + 1.0} \tag{10}$$

$$Z_{eq}(s) = 470.88 + \frac{1.48717 \cdot 10^3}{s + 9.00009} + \frac{66.7778 \cdot 10^3}{s + 565.004} + \frac{1.67227 \cdot 10^6}{s + 18.8491 \cdot 10^3} + \frac{79.6728 \cdot 10^6}{s + 1.25663 \cdot 10^6} \tag{11}$$

$$Z_{eq}(s) = K_0 + \frac{K_1}{s + p_1} + \frac{K_2}{s + p_2} + \frac{K_3}{s + p_3} + \frac{K_4}{s + p_4} \tag{12}$$

$$Z_{eq}(s) = R_0 + \frac{\frac{1}{C_1}}{s + R_1 C_1} + \frac{\frac{1}{C_2}}{s + R_2 C_2} + \frac{\frac{1}{C_3}}{s + R_3 C_3} + \frac{\frac{1}{C_4}}{s + R_4 C_4} \tag{13}$$

$$R_0 = K_0 \tag{14}$$

$$R_i = \frac{K_i}{p_i} \tag{15}$$

$$C_i = \frac{1}{K_i} \tag{16}$$

i	0	1	2	3	4
R_i	470.88 $k\Omega/km$	165.24 $k\Omega/km$	118.19 $k\Omega/km$	88.72 $k\Omega/km$	63.40 $k\Omega/km$
C_i	-	672.42 $\mu F/km$	14.98 $\mu F/km$	598.00 nF/km	12.55 nF/km

Table 2: R_i and C_i network values derived from rational approximation

The Bode plot of the rational approximation function was superimposed on the $Z_c(\omega)$ and is plotted in Figure 2. The zeros and poles chosen for the rational approximation are also shown as vertical lines to help illustrate how they form the rational approximation.

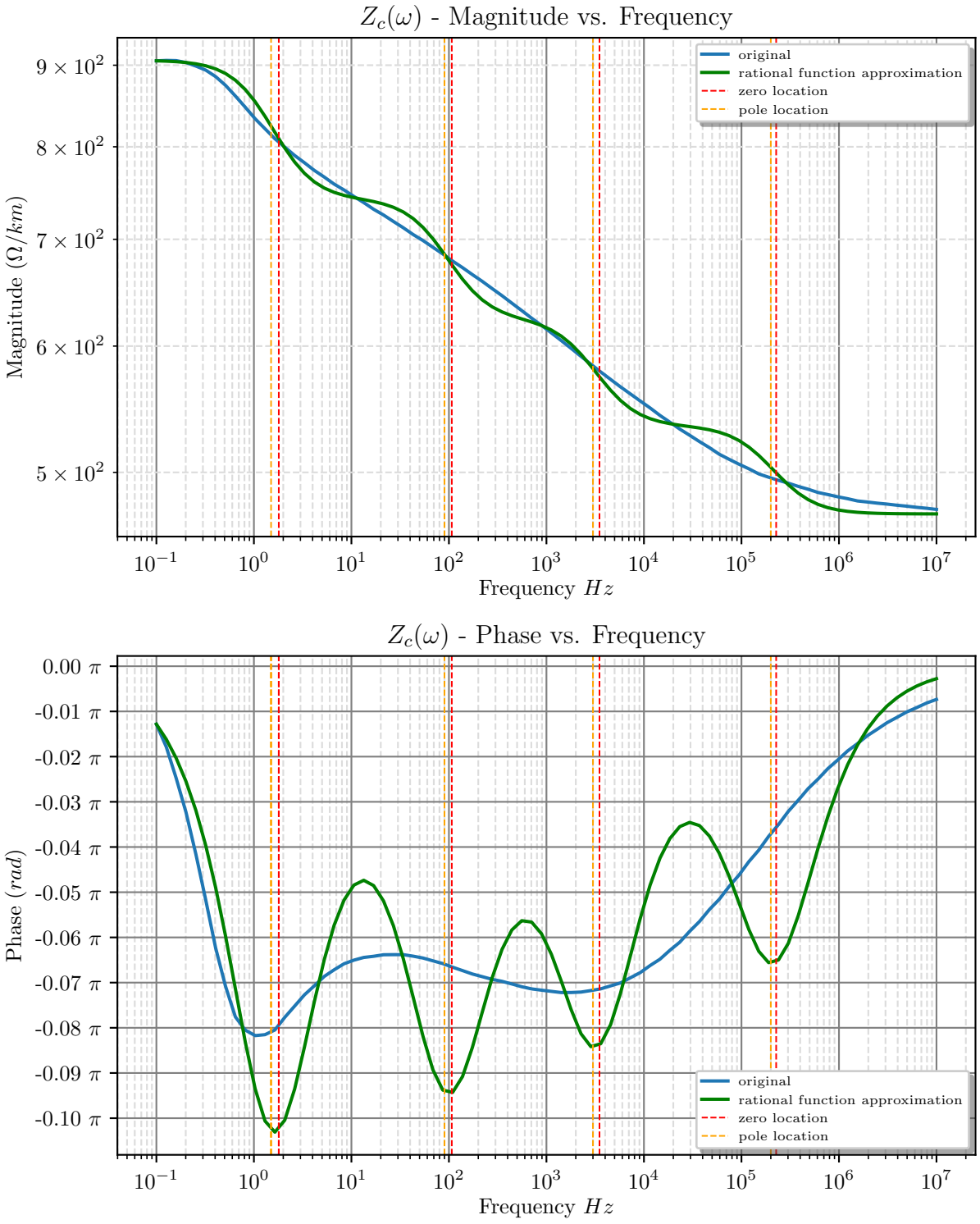

Figure 2: $Z_c(\omega)$ Original and Approximation Magnitude and Phase vs. Frequency Plots

4 Conclusion

The rational approximation generated a fairly reasonable approximation of the frequency dependent characteristic impedance $Z_c(\omega)$ in terms of both magnitude and phase.

- The magnitude fit was completely within the control of the zeros and poles; therefore, its fit was very close to that of the original characteristic impedance. As described in the earlier sections, the starting magnitude was automatically adjusted when the zeros and poles were added and adjusted, so it matches exactly the original characteristic impedance at DC. The final zero was chosen to try to make the final magnitude of the rational fit match as closely as possible to the original characteristic impedance, and this was successful as well.
- The phase fit was surprisingly close to the phase of the original characteristic impedance. At approximately 250 Hz we see the largest difference of approximately 0.07π which is quite small. The plot makes these differences look larger than they actually are. It was also observed that the phase matches very well at either extremity of the phase plot. If more zeros and poles were used, it is predicted that the phase match would be even closer, although we can see that four poles is largely sufficient.
- As described in earlier sections, the Bode plot was plotted every time a new zero or pole was added to the plot. This was immensely helpful in quickly choosing through trial and error the correct values to shape the resulting rational approximation. It was easiest to choose the zeros and poles as a pair, with the zero at a slightly higher frequency than the pole. The values were then tweaked to make the resulting Bode plot of the rational approximation match the original characteristic as closely as possible. This was repeated four times in total for each zero/pole pair. The automatic scaling of the constant K which multiplied the entire transfer function was also immensely helpful in this task.
- The process of trial error to choose the different zeros and poles for the rational approximation was likely not the most efficient way of creating it. It may in fact be possible to have the program sweep through different combinations of zeros and poles to obtain an even better match. The program could work a zero/pole pair at a time and attempt to minimize the magnitude difference within a working window. This window would be approximately a decade wide for each pair. It was observed that the delta between the zero/pole frequency pair was proportional to the slope of the curve it was attempting to fit. The steeper the slope, the wider the zero/pole pair frequency bandwidth. Understanding this relationship would help choose an appropriate zero/pole pair to begin the fitment at a given frequency. Equally spacing out the zero/pole pairs across the entire fitting spectrum also allows for the greatest fitting flexibility and the program would also try to maintain this spacing (given a characteristic impedance like the one used in this assignment).
- An interesting exercise would be to take the equivalent circuit this circuit generates and use it model the frequency dependent transmission line and compare it to a model that PSCAD supports.
- Another interesting exercise would be to create additional rational approximations with less and more poles and try to determine what the most ideal and/or practical number of poles is.
- An open question is: how well does this model work for radio frequencies? Are there other high frequency effects, such as the skin effect, that would affect the utility of a model like this or is the frequency dependent model still effective at characterizing radio frequencies?

5 Code Listings and Data

5.1 Python Code Listing

The following is the code written in Python to perform the calculations derived for this homework assignment as well as generate the plots used in this report. The transmission line parameters were converted to a CSV file which is read in by this Python script.

```
import argparse
import csv
import matplotlib
import matplotlib.ticker as tck
import matplotlib.pyplot as plt
import numpy as np
import control
import sympy

 # Matplotlib export settings
matplotlib.use('pgf')
import matplotlib.pyplot as plt
matplotlib.rcParams.update({
    'pgf.texsystem': 'pdflatex',
    'font.size': 10,
    'font.family': 'serif',  # use serif/main font for text elements
    'text.usetex': True,     # use inline math for ticks
    'pgf.rcfonts': False     # don't setup fonts from rc parameters
})

# Main function
def main(args):

    # Circuit Constants
    C_zero = 7.5240e-03 * 1e-6 # Farads/km
    G_zero = 2.0000e-08 # Mhos/km

    # List indices
    # CSV data
    FREQ_INDEX = 0
    R_ZERO_INDEX = 1
    L_ZERO_INDEX = 2
    # Calculated impedance
    MAGNITUDE_INDEX = 0
    PHASE_INDEX = 1

    # ZEROES (in Hz)
    zeros = [1.8,107,3.5e3,2.27e5]
    # POLES (in Hz)
    poles = [1.5,90,3e3,2e5]

    # prepopulate data with a list of five empty lists
    data = [[] for i in range(3)]

    # Read in PSCAD .CSV data
    print('*** Opening assignment 4 CSV data file...')
    with open('data_assign04.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        line_count = 0
        # Read in row data
        for row in csv_reader:
            if line_count == 0:
                print('Column names are: ' + ', '.join(row))
            else:
                data[FREQ_INDEX].append(float(row[0]))
                data[R_ZERO_INDEX].append(float(row[1])) # Ohms/km
                data[L_ZERO_INDEX].append(float(row[2]) * 1e-3) # Henries/km
```

```
            line_count += 1
        # Figure out when break switched
        print('Processed ' + str(line_count) + ' data points.')

    # Interpolate additional data
    # - We have a total of eight decades, and we want 10 points per decade = 80 datapoints
    # - Create an array of X values to interpolate for, from 1e-1 to 1e-7, base 10
        logarithmic increase
    interp_f_data = np.logspace(-1, 7, base=10, num=80)
    # Obtain omega values from frequencies
    interp_w_data = 2*np.pi*interp_f_data
    # Use numpy's interp function to interpolate from existing data
    interp_r_data = np.interp(interp_f_data, data[FREQ_INDEX], data[R_ZERO_INDEX])
    interp_l_data = np.interp(interp_f_data, data[FREQ_INDEX], data[L_ZERO_INDEX])

    num_interp_data_points = len(interp_f_data)

    # Prepare values for interpreted Z(w) magnitude and phase
    interp_impedance_zero = [[],[]]
    for index in range(num_interp_data_points):
        omega = 2*np.pi*interp_f_data[index]
        interp_impedance_zero_val = np.sqrt((interp_r_data[index] + (1j*omega*interp_l_data[
            index]))/(G_zero + (1j*omega*C_zero)))
        interp_impedance_zero[MAGNITUDE_INDEX].append(np.absolute(interp_impedance_zero_val)
            )
        # print(interp_impedance_zero[MAGNITUDE_INDEX][-1])
        interp_impedance_zero[PHASE_INDEX].append(np.angle(interp_impedance_zero_val))
        # print(interp_impedance_zero[PHASE_INDEX][-1])

    #### Generate Bode plot ####

    # Covert poles and zeros into omega values
    poles_w = [2*np.pi*f for f in poles]
    zeros_w = [2*np.pi*f for f in zeros]

    # Python control library requires the numerator and denominator be expanded polynomials.
    # The following code will prepare the poles and zeros for expansion and extraction of
        the coeffecients.
    # Create the numerator string
    numerator = ''.join(['(s + %d)'%zero if i is 0 else '*(s + %d)'%zero for i, zero in
        enumerate(zeros_w)])
    # Create the denominator string
    denominator = ''.join(['(s + %d)'%pole if i is 0 else '*(s + %d)'%pole for i, pole in
        enumerate(poles_w)])

    # Tell sympy we're using s as a symbol in our equations
    s = sympy.symbols('s')

    # Change numerator and denominator strings into sympy symbolic expressions
    num_poly = sympy.poly(sympy.core.sympify(numerator))
    den_poly = sympy.poly(sympy.core.sympify(denominator))

    # Get polynomial coefficients
    num_coeffs = [float(num) for num in num_poly.coeffs()]
    den_coeffs = [float(den) for den in den_poly.coeffs()]

    # Create our control function from the coefficients
    H = control.tf(num_coeffs, den_coeffs)

    h_mag, h_phase, h_frequencies = control.bode_plot(H, interp_w_data)
    h_mag_scaled = np.array(h_mag) * interp_impedance_zero[MAGNITUDE_INDEX][0]/h_mag[0]
    h_scale_factor = interp_impedance_zero[MAGNITUDE_INDEX][0]/h_mag[0]
    print("Scaled Bode plot by a factor of: " + str(h_scale_factor))

    # Perform partial fraction expansion to obtain our equivalent circuit components
    symbolic_transfer_function = sympy.core.sympify(h_scale_factor)*num_poly/den_poly
    print("Transfer function factor form: \r\n" + str(sympy.latex(sympy.N(
        symbolic_transfer_function,5))))
```

```
    print("Simple␣fraction␣expansion␣of␣transfer␣function:␣\r\n" + str(sympy.latex(sympy.N(
        sympy.apart(symbolic_transfer_function),5))))

    # Plots for publication
    legend_font_size = 6

    # Plot Z(w) magnitude and phase
    fig, ax = plt.subplots(2)

    ax[0].plot(interp_f_data, interp_impedance_zero[MAGNITUDE_INDEX], label='original')
    ax[0].plot(interp_f_data, h_mag_scaled, color='g', label='rational␣function␣
        approximation')
    ax[0].set(xlabel='Frequency␣$Hz$', ylabel='Magnitude␣($\Omega/km$)', title='$Z_c(\omega
        {})$␣-␣Magnitude␣vs.␣Frequency')
    ax[0].grid(b=True, which='major', color='gray', linestyle='-')
    ax[0].grid(b=True, which='minor', color='gainsboro', linestyle='--')
    ax[0].set_yscale('log')
    ax[0].set_xscale('log')
    # Plot zero locaitons
    for i in range(len(zeros)):
        ax[0].axvline(x=zeros[i], linestyle='dashed', c='red', linewidth='0.75')
    # Plot pole locations
    for i in range(len(poles)):
        ax[0].axvline(x=poles[i], linestyle='dashed', c='orange', linewidth='0.75')
    handles, labels = ax[0].get_legend_handles_labels()
    handles.append(plt.axvline(x=zeros[0], linestyle='dashed', c='red', linewidth='0.75'))
    labels.append('zero␣location')
    handles.append(plt.axvline(x=poles[0], linestyle='dashed', c='orange', linewidth='0.75')
        )
    labels.append('pole␣location')
    ax[0].legend(handles=handles, labels=labels, loc='upper␣right', prop={'size':
        legend_font_size}, fancybox=True, shadow=True)

    ax[1].plot(interp_f_data, interp_impedance_zero[PHASE_INDEX], label='original')
    ax[1].plot(interp_f_data, h_phase, color='g', label='rational␣function␣approximation')
    ax[1].yaxis.set_major_formatter(tck.FormatStrFormatter('%1.2f␣$\pi$'))
    ax[1].yaxis.set_major_locator(tck.MultipleLocator(base=1/100))
    ax[1].set(xlabel='Frequency␣$Hz$', ylabel='Phase␣($rad$)', title='$Z_c(\omega{})$␣-␣
        Phase␣vs.␣Frequency')
    ax[1].grid(b=True, which='major', color='gray', linestyle='-')
    ax[1].grid(b=True, which='minor', color='gainsboro', linestyle='--')
    ax[1].set_xscale('log')
    # Plot zero locaitons
    for i in range(len(zeros)):
        ax[1].axvline(x=zeros[i], linestyle='dashed', c='red', linewidth='0.75')
    # Plot pole locations
    for i in range(len(poles)):
        ax[1].axvline(x=poles[i], linestyle='dashed', c='orange', linewidth='0.75')
    handles, labels = ax[1].get_legend_handles_labels()
    handles.append(plt.axvline(x=zeros[0], linestyle='dashed', c='red', linewidth='0.75'))
    labels.append('zero␣location')
    handles.append(plt.axvline(x=poles[0], linestyle='dashed', c='orange', linewidth='0.75')
        )
    labels.append('pole␣location')
    ax[1].legend(handles=handles, labels=labels, loc='lower␣right', prop={'size':
        legend_font_size}, fancybox=True, shadow=True)
    fig.set_size_inches(6.5,8)
    fig.tight_layout()
    fig.savefig('zc_magnitude_and_phase_plot.pgf')
    fig.savefig('zc_magnitude_and_phase_plot.png')

if __name__ == '__main__':
    # the following sets up the argument parser for the program
    parser = argparse.ArgumentParser(description='Assignment␣5␣solution␣generator')

    args = parser.parse_args()

    main(args)
```