Symbolic Analysis of a Twin-T Active Bandpass Filter with Python

Tiburonboy, juan1543cabrillo@sudomail.com

Last Update: 05 April 2025

Abstract: This report is an analysis of a band pass filter using Symbolic Modified Nodal Analysis implemented with Python in a JupyterLab Notebook then rendered into a PDF. The report relies on the work, *A Bandpass Twin-T Active Filter Used in the Buchla 200 Electric Music Box Synthesizer*, by Aaron D. Lanterman. The report continues with related analysis, thoughts and observations about the filter topology. The use of Python makes generating the node equations and obtaining analytic solutions for the node voltages almost effortless.
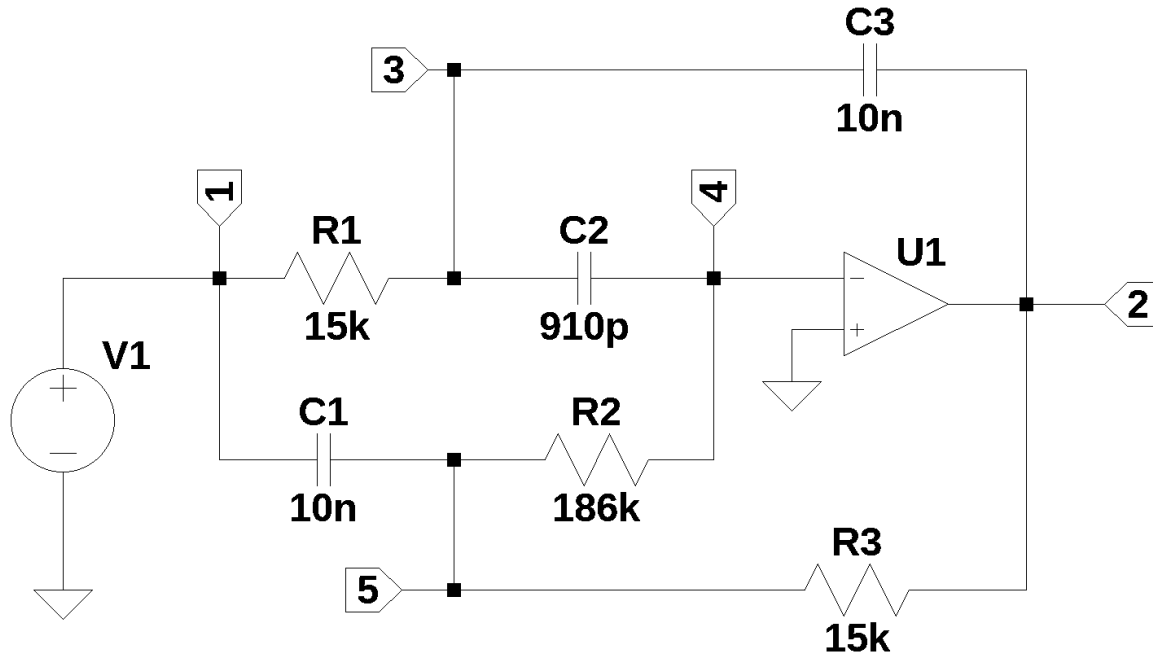
Figure 1: Schematic for Bandpass Twin-T Active Filter with components for the 1000 Hz band pass filter.

1

1 INTRODUCTION

This report walks through some of the analysis found in [1], which describes a third order bandpass filter (BPF) employed in the Buchla Model 295 10 Channel Comb Filter, a synthesizer module developed as part of the Buchla 200 Electric Music Box. The BPF described has a unique arrangement of elements not found in the typical Twin-T configuration, which makes this BPF a interesting candidate to study. I sometimes watch YouTube videos made by Lantertronics - Aaron Lanterman, the author of [1]. His video, Analysis of the Buchla 295 10 Channel Comb Filter - a Weird Twin-T Topology, describes this filter and some of its characteristics.

A procedure implemented in the Python programming language called modified nodal analysis (MNA) [2] is used to generate the network equations used in this analysis. The MNA procedure provides an algorithmic method for generating a system of independent equations for linear circuit analysis. Once the schematic is drawn and the net list is generated, the network equations and solutions for the node voltages are easily obtained as shown in this report. The code in the JupyterLab notebook can be used as a template to analyze almost any linear circuit. The schematic for the filter, Figure 1, was drawn using LTSpice and the netlist was exported for the analysis.

This report was written using a JupyterLab notebook and rendered to a PDF document using Quarto which is an open-source scientific and technical publishing system. The JupyterLab notebook which is the "source code" for this paper is available linked here and related material is linked here.

My motivation for writing this report is: a) my own educational purposes, b) to explore the ability of Python to solve circuit analysis problems, and c) to test and document the procedure for rendering a JupyterLab notebook into a PDF document.

1.1 Buchla History

Buchla Electronic Musical Instruments (BEMI) was founded by Don Buchla in 1963 under the name Buchla & Associates. The company manufactured synthesizers and MIDI controllers. Buchla's modular synthesizers, like the Buchla 100 and 200 series, were innovative in their design and sound creation approach, differing from Moog synthesizers by employing complex oscillators and waveshaping.

1.2 Related Work

The Python code employed in this report is somewhat unique since Python is open source, free and runs on a variety of platforms. The Python MNA code is made available under a public domain license and archived in a GitHub repository linked above.

There are other symbolic circuit analysis codes available and some of these are described here. Some of these codes are based on commercial software such as MATLAB, TINA and Maple.

SLiCAP is a symbolic linear analysis tool. SLiCAP is now a Python program, but originally it was written in MATLAB.

TINA is an acronym of Toolkit for Interactive Network Analysis. The TINA design suite is a circuit simulator and PCB design software package for analyzing, designing, and real time testing of analog, digital, HDL, MCU, and mixed electronic circuits and their PCB layouts. TINA has some symbolic analysis capability.

Maple is a mathematical package and there is an application note available describing its use in symbolic circuit analysis. The application note presents a method for evaluating, solving and designing a common, but not so simple pulse-mode high-gain transimpedance amplifier or TIA circuit.

Symbolic Circuit Analysis is a web page devoted to symbolic circuit analysis.

SAPWIN is a windows program package for symbolic and numerical simulation of analog circuits.

Lcapy is an experimental Python package for teaching linear circuit analysis. It uses SymPy for symbolic mathematics. In [3] there is an overview of Lcapy as well as a survey of symbolic circuit analysis packages.

1.3 Circuit Description

The schematic for the 1000 Hz section of the comb filter is shown in Figure 1. The schematic was drawn using LTSpice and the netlist was exported for this analysis. The schematic has component values for the 1000 Hz section of the comb filter. The input node is labeled node 1 and the output of the Op Amp is labeled node 2. The other nodes are sequentially labeled from 3 to 5. The filter circuit has three resistors, three capacitors and one Op Amp. The component values shown in the schematic, Figure 1, were chosen from [1, Table I], which is the 1000 Hz filter. In [4], the Op Amp is listed as N5556V, which was an operational amplifier manufactured by Signetics.

The Op Amp used in the analysis is an ideal Op Amp, which is defined in the Python MNA code as a circuit element of type O in the circuit netlist. The ideal Op Amp is a theoretical concept, used to simplify the analysis of circuits which contain one or more operational amplifiers and is a type of circuit element supported by the MNA code. The ideal Op Amp as modeled in the MNA code assumes that there is a DC negative feedback path in the circuit. No current flows into the input terminals of the Op Amp and the output impedance of the Op Amp is zero.

The Twin-T filter topology is made up of two "T" shaped networks connected together, which is why it's called "Twin-T". The arrangement of the resistors and capacitors in Figure 1 is a

bit different than the more common notch (band stop) Twin-T filter which shows up online when you search for "Twin-T". The filter has three capacitors which suggests that the circuit has a third order characteristic polynomial.

For the analysis that follows, the filter circuit is assumed to be a linear time-invariant (LTI) system. LTI systems have two key properties, First, the circuit obeys the superposition principle and second the circuit doesnt change over time, in other words, a time-shifted input results in a time-shifted output.

2 METHODS

The analysis of the circuit in Figure 1 uses both symbolic and numerical mathematics to solve a system of equations to obtain solutions of the node voltages. The circuit analysis relies on using Laplace transformed circuit elements, specifically, the capacitor values in Figure 1 are replace by $\frac{1}{sC_n}$, where s is the Laplace variable and C_n are the capacitor values. This means that the analysis is performed in the frequency domain where linear algebra and matrix inversions are used to solve for the unknown voltages and currents. This method avoids having to deal with differential equations. The circuit analysis is performed using a JupyterLab notebook running Python code.

LTSpice was used to draw the schematic of the filter and generate the netlist. This helps to ensure that the netlist is correct, which is helpful for circuits with a complicated topology or larger number of components. The netlist for Figure 1 could have been easily generated by hand, but since the schematic needed to be drawn out to include as a figure in this report, it is natural to just use LTSpice to generate the netlist. The component values in the netlist were manually edited to use scientific notation and the Op Amp entry was fixed. Later in this report the solutions obtained from LTSpice are compared to the solutions from the Python code.

The analysis begins with generating the network equations from the circuit's netlist by calling:

```
report, network_df, df2, A, X, Z = SymMNA.smna(example_net_list)
```

The function `SymMNA.smna()` is described in [5, chapter 4]. The SymPy function `solve` is capable of working through the algebra to solve the system of equations and can even find the roots of the numerator and denominator polynomials in some cases. Analytic expressions are easily generated for the filter's node voltages and the voltage transfer function.

The analysis continues with a reduced complexity version of the filter, as described in [1], followed by numerical solutions and various graphs and tables. Then three filter design examples are presented. The report ends with Results and Discussion sections.

This report was getting rather long, so related analysis topics were moved to separate Jupyter-Lab notebooks. Links to three related JupyterLab notebooks are provided. The first link is to an interesting alternate circuit analysis technique called design oriented analysis [6]. The second link is to a JupyterLab notebook that covers a sensitivity analysis of the filter. The third link looks at lowpass, highpass and band reject variations of the filter.

This report is directly rendered from a JupyterLab notebook and many of the code cells with output are displayed. The code cells are highlighted in gray and most of the code outputs are displayed by converting the equations to LaTex and then to markdown to make the fonts used by the PDF rendering engine consistent.

Since this report is about circuit analysis and Python coding, I felt that including the Python code in the PDF would illustrate this link. Some of the code cells have been suppressed in the PDF output by placing the mark down command, `#| echo: false`, in the first line of the code cell. This was done to improve readability since some of the code repeats. The `ipynb` notebook file in the GitHub repository can be examined to view all the notebook cells, even those hidden by using HTML comment tags.

Key techniques used in the analysis are described in the following sections:

2.1 Frequency Response Plot

A frequency response plot is a graphical representation of how a system responds to different input frequencies. It's a valuable tool in various fields, including engineering, audio, and signal processing. Typically, a frequency response plot shows two key characteristics of a system:

1. Magnitude Response: This indicates how much the system amplifies or attenuates different frequencies. It's often plotted in decibels (dB) on a logarithmic scale.
2. Phase Response: This shows the phase shift introduced by the system at different frequencies. It's usually plotted in degrees or radians.

By analyzing a frequency response plot, you can determine:

- Bandwidth: The range of frequencies a system can process effectively.
- Gain: The amount of amplification or attenuation at specific frequencies.
- Phase shift: The delay introduced by the system at different frequencies.
- Resonance: Peaks in the magnitude response indicating frequencies where the system amplifies strongly.
- Stability: Whether a system is stable or unstable based on the phase margin.

2.2 The Complex Frequency Plane

The complex frequency plane is a two-dimensional space where complex numbers represent frequency. Unlike the real-world concept of frequency, which is a scalar value representing the rate of oscillation, complex frequency adds a dimension of complexity. It's represented by the symbol s and is defined as:

$s = \sigma + j\omega$

Where:

- σ is the real part, representing exponential growth or decay.
- j is the imaginary unit $\sqrt{-1}$.
- ω is the imaginary part, representing the angular frequency (related to the frequency in Hertz).

The complex frequency plane is divided into four quadrants:

- Right half-plane (RHP): $\sigma > 0$, representing exponentially growing signals.
- Left half-plane (LHP): $\sigma < 0$, representing exponentially decaying signals.
- Imaginary axis: $\sigma = 0$, representing sinusoidal signals with constant amplitude.

By plotting poles and zeros of a system on this plane, you can gain insights into its behavior, stability, and frequency response.

2.3 Pole-Zero Diagram

A pole–zero plot is a graphical representation of a system's transfer function in the complex plane. It provides valuable insights into the system's behavior, stability, and frequency response. The diagram is plotted on the complex plane, with the real part on the horizontal axis (σ) and the imaginary part on the vertical axis ($j\omega$).

- Poles: These are the values of the complex variable 's' that make the transfer function infinite. They are represented by 'x' on the diagram.
- Zeros: These are the values of 's' that make the transfer function zero. They are represented by 'o' on the diagram.

The relative count of zeros and poles in a transfer function significantly impacts the shape of the frequency response.

- Poles:
    - Each pole introduces a downward slope of -20 dB/decade. Multiple poles at the same frequency amplify this effect.

– Each pole contributes a phase shift of -90 degrees. Multiple poles at the same frequency amplify this effect.

- Zeros:

  – Each zero introduces an upward slope of +20 dB/decade. Multiple zeros at the same frequency amplify this effect.
  – Each zero contributes a phase shift of +90 degrees. Multiple zeros at the same frequency amplify this effect.

- Overall Slope:

  – If the number of zeros exceeds the number of poles, the magnitude plot will have a net upward slope.
  – If the number of poles exceeds the number of zeros, the magnitude plot will have a net downward slope.
  – A balanced number of zeros and poles can lead to a flat magnitude plot over certain frequency ranges.

- High-Frequency Asymptote:

  – The high-frequency asymptote of the magnitude plot is determined by the difference between the number of zeros and poles.
  – A larger number of zeros compared to poles will result in a steeper upward slope at high frequencies.
  – A larger number of poles compared to zeros will result in a steeper downward slope at high frequencies.

- Phase Shift:

  – The total phase shift at any frequency is the sum of the phase shifts contributed by each pole and zero.
  – A larger number of zeros compared to poles will result in a larger positive phase shift.
  – A larger number of poles compared to zeros will result in a larger negative phase shift.

The location of poles and zeros on the complex plane provides information about:

- Stability: If all poles are in the left half of the plane (LHP), the system is stable. If any pole is in the right half of the plane (RHP), the system is unstable.
- Frequency response: The distance of poles and zeros from the imaginary axis affects the system's gain and phase response at different frequencies.
- Time-domain behavior: The location of poles and zeros can influence the system's transient response and steady-state behavior.

2.4 Impulse Response

The impulse response of a system is its output when presented with a brief input signal called an impulse. An impulse is an idealized signal that has infinite amplitude and infinitely short duration, but with a total area of one. The impulse response can be obtained from the transfer function by using the SciPy function impulse.

However, for the analysis of Figure 1, I didn't find the impulse response of the transfer function to be that informative. Using SciPy, the impulse response is easy to calculate and plot, but I couldn't relate the shape of the impulse response back to the circuit in an intuitive way. Additionally, using the impulse response along with the convolution function did not produce correct results since the SciPy convolve and Numpy convolve functions seem to treat the signal and circuit as a discrete time system. For this reason, the impulse of the transfer functions are not included in the analysis.

2.5 Step Response

The step response of a system is its output when the input is a sudden change from zero to a constant value. This constant value is often referred to as a "step" input. The system's behavior as it transitions from its initial state to a new steady-state condition is captured by the step response.

Several parameters are used to characterize a step response:

- Rise time: The time it takes for the output to go from 10% to 90% of its final value.
- Overshoot: The maximum amount by which the output exceeds the final value.
- Settling time: The time it takes for the output to settle within a specified percentage (e.g., 1%) of the final value.
- Steady-state error: The difference between the final value of the output and the desired output.

The step response provides valuable insights into a system's behavior:

- Stability: A stable system will eventually reach a steady-state value.
- Speed of response: The rise time indicates how quickly the system reacts to changes.
- Damping: Overshoot and settling time reveal information about the system's damping characteristics.

The step response is calculated by using the SciPy function lsim, which can be used to simulate output of a LTI system. The function, lsim(), allows us to evaluate the performance characteristics of the circuit with a square wave or arbitrary input signal. SciPy has step function in the library, but in this report, several cycles of a squarewave are used to examine the step response.

2.6 Group Delay

Group delay is a measure of the time delay experienced by a group of frequencies as they pass through a system. It's essentially the rate of change of the phase response with respect to frequency. Group Delay is important for the following reasons:

- Signal distortion: Variations in group delay across different frequencies can cause signal distortion, affecting the quality of audio, video, and data transmission.
- System design: Understanding group delay is crucial for designing systems with linear phase characteristics, which minimize distortion.
- Pulse propagation: In fields like optics and telecommunications, group delay affects the shape and timing of pulses.

Group delay, τ_g, is calculated as the negative derivative of the phase response, ϕ, with respect to angular frequency, ω:

$\tau_g(\omega) = -\frac{d\phi(\omega)}{d\omega}$

A system with a constant group delay is called a linear phase system. These systems introduce a pure time delay to all frequency components without altering their relative phase relationships, preserving the original waveform shape. According to Audibility of Group-Delay Equalization, the threshold is 2 ms. The abstract for the paper states:

> The audibility thresholds for group-delay variation from several previous related studies are shown in Fig. 1. If not otherwise stated, these studies have been conducted using headphones. Green applied Huffman sequences, or truncated impulse responses of second-order allpass filters, to study the audibility of phase distortion. He found a threshold value for the peak group delay of about 2 ms for center frequencies of 625 Hz, 1875 Hz, and 4062 Hz.

The group delay for Figure 1, was only plotted for one case since this result was not used in the evaluation of the filter.

2.7 Reproducibility

The analysis is based on Python modules consisting of SymPy, NumPy, SciPy and Pandas. Code to automatically generate network equations from circuit netlist using Modified-Nodal-Analysis [2]. The function, `SymMNA()`, is described in [5, chapter 4].

The results presented in this report should be reproducible if the package versions listed in Table 1 are used.

Table 1: Package or Library versions

Package	version
Python	3.10.9
JupyterLab	3.5.3
IPython	8.10.0
NumPy	1.23.5
SymPy	1.11.1
SciPy	1.10.0
Pandas	1.5.3
Matplotlib	3.7.0
Tabulate	0.8.10
LTSpice	17.1.8
Quarto	1.4.553

The JupyterLab notebook used to generate this report was run on a laptop with an Intel i3-8130U CPU @ 2.2 GHz, illustrating that a high performance machine is not required to perform the analysis described in this document.

The following Python modules are used in this notebook.

```
from sympy import *
import numpy as np
from tabulate import tabulate
from scipy import signal
import matplotlib.pyplot as plt
import SymMNA
from IPython.display import display, Markdown, Math, Latex
from tabulate import tabulate
init_printing()
```

3 RESULTS

In this section the analysis methods described above are applied to the netlist of Figure 1. The Python code presented below can be used as a template to analyze almost any LTI circuit, where the circuit elements are those described in [5, Chapter 2].

3.1 Symbolic Analysis

Symbolic analysis presented in this report employs the MNA technique to generate network equations from the filter's netlist where the circuit's element values are represented by symbols. SymPy is used to solve the system of equations in symbolic form to obtain analytic expressions for the filter's voltage transfer function. The symbolic expressions can in some cases provide a deeper understanding of how each component contributes to the circuit's operation that can complement numerical simulations.

3.1.1 Circuit Netlist

The netlist generated by LTSpice from the schematic is shown below. Some edits were made to fix up the formatting of the component values and the Op Amp declaration. The nodes were labeled in the schematic, otherwise LTSpice will use default labels such as N001, N002 etc. and the `SymMNA.smna` function wants integer values for the node numbers and these need to be consecutively ordered with no gaps in the numbering.

```
* Bandpass-Twin-T-Active-Filter.asc
R1 3 1 15k
R2 4 5 186k
R3 2 5 15k
C1 5 1 10n
C2 4 3 910p
C3 2 3 10n
XU1 4 0 2 opamp Aol=100K GBW=10Meg
V1 1 0 1 AC 1
.lib opamp.sub
.ac dec 1000 100 10k
.backanno
.end
```

Generation of the netlist from a schematic capture program is convenient and less error prone than generating the netlist by hand. A visual inspection of the schematic ensures that the circuit to be analyzed is correct and it follows that the netlist is also correct. This is especially true for larger, more complicated schematics.

The cleaned up netlist was copied to the cell below. The triple quotes allow text strings to span multiple lines since the line breaks are preserved.

```
example_net_list = '''
* Bandpass-Twin-T-Active-Filter.asc
R1 3 1 15e3
R2 4 5 186e3
R3 2 5 15e3
C1 5 1 10e-9
C2 4 3 910e-12
C3 2 3 10e-9
O1 4 0 2
V1 1 0 1
'''
```

3.1.2 Generate Network Equations

The function `SymMNA.smna(example_net_list)` implements the MNA method on the filter's netlist. Stamps which are templates for modifying the B, C and D matrices are used to facilitate the construction of the matrices. The stamps used in this implementation of the MNA follow the stamps of [6]. The function is divided in the following sections.

- The preprocessor reads in the netlist text file and removes comments, extra spaces and blank lines. The first letter of the element type is capitalized to make subsequent parsing of the file easier. The number of lines are counted and the number of entries on each line are checked to make sure the count is consistent with the element type.
- The parser code loads the preprocessed netlist into a data frame. A report is generated which consists of a count of the element types in the netlist.
- Matrix formulation: Each of the matrices and vectors are generated.
- Circuit equation generation: The circuit equations are generated in a `for` loop. Sympy automatically does some simplification according to its default settings. The Laplace variable s is used when inductors and capacitors are included in the circuit.

There is a limited amount of error checking performed. The number of items on each line in the netlist is checked to make sure the count is correct depending on the element type. The node numbering needs to be consecutive with no skipped numbers, otherwise empty rows or columns will be included in the matrix. Unknown element types create an error.

The function takes one argument which is a text string that is the circuit's netlist. The format of the netlist is very similar to the standard spice netlist and any of the required changes can be easily made with a text editor. The function returns six items.

1. report - a text string, which is the netlist report.
2. df - a Pandas data frame, which the circuit's net list info loaded into a data frame.
3. df2 - a Pandas data frame, which contains the branches with unknown currents.

4. *A* - a SymPy matrix, which is $(m + n)$ by $(m + n)$, where n is the number of nodes and m is the number of current unknowns. *A* is the combination of four smaller matrices, G, B, C, and D. These matrices are described in the GitHub repository.
5. *X* - a list that holds the unknown node voltages and the currents through the independent voltage sources.
6. *Z* - a list that holds the independent voltage and current sources

The netlist can be generated by hand or exported from a schematic capture program and pasted into the JupyterLab notebook. The following call is made to generate the NMA matrices and report.

```
report, network_df, df2, A, X, Z = SymMNA.smna(example_net_list)
```

The report lists some of the relevant statistics about the circuits netlist.

```
print(report)
```

```
Net list report
number of lines in netlist: 8
number of branches: 7
number of nodes: 5
number of unknown currents: 2
number of RLC (passive components): 6
number of resistors: 3
number of capacitors: 3
number of inductors: 0
number of independent voltage sources: 1
number of independent current sources: 0
number of Op Amps: 1
number of E - VCVS: 0
number of G - VCCS: 0
number of F - CCCS: 0
number of H - CCVS: 0
number of K - Coupled inductors: 0
```

The returned parameters, *X* and *Z*, which are of type, `list`, and are the unknown node voltages and currents as well as the independent voltage and current sources need to be turned into SymPy matrices by the following code:

```
X = Matrix(X)
Z = Matrix(Z)
```

Use the SymPy function `Eq` to formulate the network equations.

```
NE_sym = Eq(A*X,Z)
```

The equations, `NE_sym`, define the mathematical relationship between voltages and currents in the filter circuit in terms of node voltages, v_1, v_2 etc., components R_1, C_1 etc., the Laplace variable s and the independent voltage source V_1. The equations are shown below:

```
# display the equations
temp = ''
for i in range(shape(NE_sym.lhs)[0]):
    temp += '<p>${:s} = {:s}$</p>'.format(latex(NE_sym.lhs[i]),
        latex(NE_sym.rhs[i]))

Markdown(temp)
```

$-C_1 s v_5 + I_{V1} + v_1 \left(C_1 s + \frac{1}{R_1} \right) - \frac{v_3}{R_1} = 0$

$-C_3 s v_3 + I_{O1} + v_2 \left(C_3 s + \frac{1}{R_3} \right) - \frac{v_5}{R_3} = 0$

$-C_2 s v_4 - C_3 s v_2 + v_3 \left(C_2 s + C_3 s + \frac{1}{R_1} \right) - \frac{v_1}{R_1} = 0$

$-C_2 s v_3 + v_4 \left(C_2 s + \frac{1}{R_2} \right) - \frac{v_5}{R_2} = 0$

$-C_1 s v_1 + v_5 \left(C_1 s + \frac{1}{R_3} + \frac{1}{R_2} \right) - \frac{v_2}{R_3} - \frac{v_4}{R_2} = 0$

$v_1 = V_1$

$v_4 = 0$

The two lines of Python code below turn the free symbols from the `NE_sym` equations into SymPy variables and the element values contained in the netlist are put into a Python dictionary data structure to be used in the numerical solutions later in this report.

```
var(str(NE_sym.free_symbols).replace('{','').replace('}',''))
element_values = SymMNA.get_part_values(network_df)
```

3.1.3 Solve the Network Equations

The SymPy function `solve` is used to obtain the node voltages and independent voltage source currents.

```
U_sym = solve(NE_sym,X)
```

The solution to the network equations are the node voltages expressed in terms of the Laplace variable s are displayed below.

```
temp = ''
for i in U_sym.keys():
    if str(i)[0] == 'v': # only display the node voltages
        temp += '<p>${:s} = {:s}$</p>'.format(latex(i),latex(U_sym[i]))

Markdown(temp)
```

$v_1 = V_1$

$v_2 = \frac{-C_1 C_2 R_1 R_3 V_1 s^2 - C_1 C_2 R_2 R_3 V_1 s^2 - C_1 C_3 R_1 R_3 V_1 s^2 - C_1 R_3 V_1 s - C_2 R_2 V_1 s - C_2 R_3 V_1 s}{C_1 C_2 C_3 R_1 R_2 R_3 s^3 + C_2 C_3 R_1 R_2 s^2 + C_2 C_3 R_1 R_3 s^2 + C_2 R_1 s + C_3 R_1 s + 1}$

$v_3 = \frac{-C_1 C_3 R_1 R_3 V_1 s^2 + V_1}{C_1 C_2 C_3 R_1 R_2 R_3 s^3 + C_2 C_3 R_1 R_2 s^2 + C_2 C_3 R_1 R_3 s^2 + C_2 R_1 s + C_3 R_1 s + 1}$

$v_4 = 0$

$v_5 = \frac{C_1 C_2 C_3 R_1 R_2 R_3 V_1 s^3 - C_2 R_2 V_1 s}{C_1 C_2 C_3 R_1 R_2 R_3 s^3 + C_2 C_3 R_1 R_2 s^2 + C_2 C_3 R_1 R_3 s^2 + C_2 R_1 s + C_3 R_1 s + 1}$

3.1.4 Voltage Transfer Function

The voltage transfer function,

$H(s) = \frac{v_2(s)}{v_1(s)}$

for the filter is generated and displayed by the following code:

```
H_sym = cancel(U_sym[v2]/U_sym[v1],s)
Markdown('$H(s)={:s}$'.format(latex(H_sym)))
```

$H(s) = \frac{s^2(-C_1 C_2 R_1 R_3 - C_1 C_2 R_2 R_3 - C_1 C_3 R_1 R_3) + s(-C_1 R_3 - C_2 R_2 - C_2 R_3)}{C_1 C_2 C_3 R_1 R_2 R_3 s^3 + s^2(C_2 C_3 R_1 R_2 + C_2 C_3 R_1 R_3) + s(C_2 R_1 + C_3 R_1) + 1}$

This expression agrees with [1 eq. (1)]. The numerator is a second order polynomial and the denominator is a third order polynomial in terms of s. Generally, the order of the dominator is equal to the number of reactive elements in the circuit. The roots of the numerator polynomial are called the zeros of the transfer function and the roots of the denominator are called the poles of the transfer function.

```
H_sym_num, H_sym_denom = fraction(H_sym,s) #returns numerator and denominator
```

3.1.5 Numerator Polynomial

The numerator polynomial is:

```
Markdown('$N(s)={:s}$'.format(latex(H_sym_num)))
```

$$N(s) = s^2 \left(-C_1 C_2 R_1 R_3 - C_1 C_2 R_2 R_3 - C_1 C_3 R_1 R_3\right) + s \left(-C_1 R_3 - C_2 R_2 - C_2 R_3\right)$$

The coefficients of each Laplace terms can be equated to the variables b_2, b_1 and b_0 in the expression:

$$b_2 s^2 + b_1 s + b_0$$

where b_2, b_1 and b_0 are:

```
b2 = H_sym_num.coeff(s**2)
b1 = H_sym_num.coeff(s**1)
b0 = H_sym_num - b1*s*1 - b2*s**2

Markdown('<p>$b_2={:s}$</p><p>$b_1={:s}$\
    </p><p>$b_0={:s}$</p>'.format(latex(b2),latex(b1),latex(b0)))
```

$$b_2 = -C_1 C_2 R_1 R_3 - C_1 C_2 R_2 R_3 - C_1 C_3 R_1 R_3$$

$$b_1 = -C_1 R_3 - C_2 R_2 - C_2 R_3$$

$$b_0 = 0$$

Notice that the terms b_2 and b_1 are negative. This follows from the Op Amp being configured as an inverting amplifier. R2 and R3 provide a DC path from the Op Amp's output to the inverting terminal.

The roots of the numerator polynomial can easily be found with SymPy.

This filter has two transmission zeros

```
num_root_sym = solve(H_sym_num,s)
```

There are two solutions, $z_1 = 0$ and the other root is:

```
Markdown('$z_2={:s}$'.format(latex(num_root_sym[1])))
```

$z_2 = \frac{-C_1 R_3 - C_2 R_2 - C_2 R_3}{C_1 R_3 (C_2 R_1 + C_2 R_2 + C_3 R_1)}$

3.1.6 Denominator Polynomial

The denominator polynomial of the transfer function is called the characteristic polynomial. The roots of the denominator, also called poles of the system, determine the system's stability. If any of these roots have a positive real part, the system is unstable, meaning its output will grow unbounded. The roots also influence how the system responds to changes in input (the transient response). They affect things like how quickly the system settles to a new state, whether it oscillates, and the damping of those oscillations. Each root of the characteristic polynomial corresponds to a natural mode of the system.

The denominator polynomial is:

$D(s) = C_1 C_2 C_3 R_1 R_2 R_3 s^3 + s^2 \left(C_2 C_3 R_1 R_2 + C_2 C_3 R_1 R_3 \right) + s \left(C_2 R_1 + C_3 R_1 \right) + 1$

The coefficients of each Laplace terms can be equated to the variables a_3, a_2, a_1 and a_0 in the expression:

$a_3 s^3 + a_2 s^2 + a_1 s + a_0$

where a_3, a_2, a_1 and a_0 are:

$a_3 = C_1 C_2 C_3 R_1 R_2 R_3$

$a_2 = C_2 C_3 R_1 R_2 + C_2 C_3 R_1 R_3$

$a_1 = C_2 R_1 + C_3 R_1$

$a_0 = 1$

The roots of the denominator polynomial can be found with SymPy. The filter circuit being analyzed in this report has a denominator polynomial that SymPy can quickly solve. This is not always the case. The expressions are long and do not render well when the JupyterLab notebook is converted to PDF, so the expressions were converted to LaTex strings and then to PNG images with latex2png for this report.

```
denom_root_sym = solve(H_sym_denom,s)
```

17

$$-\frac{\sqrt[3]{\dfrac{\sqrt{-4\left(-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}\right)^3+\left(\dfrac{27}{C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{C_1^2C_2C_3R_2^2R_3^2}+\dfrac{2(R_2+R_3)^3}{C_1^3R_2^3R_3^3}\right)^2}}{2}+\dfrac{27}{2C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{2C_1^2C_2C_3R_2^2R_3^2}+\dfrac{(R_2+R_3)^3}{C_1^3R_2^3R_3^3}}}{3}$$
$$-\frac{-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}}{3\sqrt[3]{\dfrac{\sqrt{-4\left(-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}\right)^3+\left(\dfrac{27}{C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{C_1^2C_2C_3R_2^2R_3^2}+\dfrac{2(R_2+R_3)^3}{C_1^3R_2^3R_3^3}\right)^2}}{2}+\dfrac{27}{2C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{2C_1^2C_2C_3R_2^2R_3^2}+\dfrac{(R_2+R_3)^3}{C_1^3R_2^3R_3^3}}}$$
$$-\frac{R_2+R_3}{3C_1R_2R_3}$$

Figure 2: Terms for the first pole.

$$-\frac{\left(-\dfrac{1}{2}-\dfrac{\sqrt{3}i}{2}\right)\sqrt[3]{\dfrac{\sqrt{-4\left(-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}\right)^3+\left(\dfrac{27}{C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{C_1^2C_2C_3R_2^2R_3^2}+\dfrac{2(R_2+R_3)^3}{C_1^3R_2^3R_3^3}\right)^2}}{2}+\dfrac{27}{2C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{2C_1^2C_2C_3R_2^2R_3^2}+\dfrac{(R_2+R_3)^3}{C_1^3R_2^3R_3^3}}}{3}$$
$$-\frac{-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}}{3\left(-\dfrac{1}{2}-\dfrac{\sqrt{3}i}{2}\right)\sqrt[3]{\dfrac{\sqrt{-4\left(-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}\right)^3+\left(\dfrac{27}{C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{C_1^2C_2C_3R_2^2R_3^2}+\dfrac{2(R_2+R_3)^3}{C_1^3R_2^3R_3^3}\right)^2}}{2}+\dfrac{27}{2C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{2C_1^2C_2C_3R_2^2R_3^2}+\dfrac{(R_2+R_3)^3}{C_1^3R_2^3R_3^3}}}$$
$$-\frac{R_2+R_3}{3C_1R_2R_3}$$

Figure 3: Terms for the second pole.

$$-\frac{\left(-\dfrac{1}{2}+\dfrac{\sqrt{3}i}{2}\right)\sqrt[3]{\dfrac{\sqrt{-4\left(-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}\right)^3+\left(\dfrac{27}{C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{C_1^2C_2C_3R_2^2R_3^2}+\dfrac{2(R_2+R_3)^3}{C_1^3R_2^3R_3^3}\right)^2}}{2}+\dfrac{27}{2C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{2C_1^2C_2C_3R_2^2R_3^2}+\dfrac{(R_2+R_3)^3}{C_1^3R_2^3R_3^3}}}{3}$$
$$-\frac{-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}}{3\left(-\dfrac{1}{2}+\dfrac{\sqrt{3}i}{2}\right)\sqrt[3]{\dfrac{\sqrt{-4\left(-\dfrac{3(C_2+C_3)}{C_1C_2C_3R_2R_3}+\dfrac{(R_2+R_3)^2}{C_1^2R_2^2R_3^2}\right)^3+\left(\dfrac{27}{C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{C_1^2C_2C_3R_2^2R_3^2}+\dfrac{2(R_2+R_3)^3}{C_1^3R_2^3R_3^3}\right)^2}}{2}+\dfrac{27}{2C_1C_2C_3R_1R_2R_3}-\dfrac{9(C_2+C_3)(R_2+R_3)}{2C_1^2C_2C_3R_2^2R_3^2}+\dfrac{(R_2+R_3)^3}{C_1^3R_2^3R_3^3}}}$$
$$-\frac{R_2+R_3}{3C_1R_2R_3}$$

Figure 4: Terms for the third pole.

As is evident in the analytic expressions for the poles of the transfer function shown in the figures above, these expressions are not very intuitive or useful unless some simplification is performed. With the help of the Python code and SymPy, the procedure outlined above can be considered a "brute force" analysis, where analytic expressions can be easily obtained from a circuit's netlist with little effort. If the circuit is small, that is having only a few components or nodes, the symbolic expressions can be very useful.

3.2 Reduced Complexity

As described in [1], the transfer function simplifies under the case of $R_1 = R_3$ and $C_1 = C_3$ to a second order characteristic polynomial. This simplification works because a zero cancels with a pole in the voltage transfer function. The numerator polynomial reduces to the following when the Laplace term, s, is collected and factored.

```
C, R = symbols('C R')
Markdown('$N_r(s)={:s}$'.format(latex(H_sym_num.subs({R1:R, R3:R, C1:C,
    C3:C}).collect(s).factor())))
```

$$N_r(s) = -s\,(CRs+1)\,(CR + C_2R + C_2R_2)$$

In a similar fashion, the denominator polynomial is:

```
Markdown('$D_r(s)={:s}$'.format(latex(H_sym_denom.subs({R1:R, R3:R, C1:C,
    C3:C}).collect(s).factor())))
```

$$D_r(s) = (CRs+1)\,(CC_2RR_2s^2 + C_2Rs + 1)$$

You can see the $(CRs+1)$ terms will cancel in the numerator and denominator.

$$\frac{N_r(s)}{D_r(s)} = \frac{-s(CRs+1)(CR+C_2R+C_2R_2)}{(CRs+1)(CC_2RR_2s^2+C_2Rs+1)} = \frac{-s(CR+C_2R+C_2R_2)}{(CC_2RR_2s^2+C_2Rs+1)}$$

3.2.1 Symbolic Analysis for $R1 = R3 = R$ and $C1 = C3 = C$

The following Python code makes the substitution, $R_1 = R_3 = R$ and $C_1 = C_3 = C$, into the network equations.

```
NE_sym_reduced_complexity = NE_sym.subs({R1:R,R3:R,C1:C,C3:C})
```

This produces the following network equations:

$$0 = -Csv_5 + I_{V1} + v_1\left(Cs + \frac{1}{R}\right) - \frac{v_3}{R}$$

$$0 = -Csv_3 + I_{O1} + v_2\left(Cs + \frac{1}{R}\right) - \frac{v_5}{R}$$

$$0 = -Csv_2 - C_2sv_4 + v_3\left(Cs + C_2s + \frac{1}{R}\right) - \frac{v_1}{R}$$

$$0 = -C_2sv_3 + v_4\left(C_2s + \frac{1}{R_2}\right) - \frac{v_5}{R_2}$$

$$0 = -Csv_1 + v_5\left(Cs + \frac{1}{R_2} + \frac{1}{R}\right) - \frac{v_4}{R_2} - \frac{v_2}{R}$$

$$V_1 = v_1$$

$$0 = v_4$$

The unknown node voltages and current from V_1 can be found by using the SymPy `solve` function.

This produces the following expressions for the node voltages.

$$v_1 = V_1$$

$$v_2 = \frac{-CRV_1 s - C_2 RV_1 s - C_2 R_2 V_1 s}{CC_2 RR_2 s^2 + C_2 Rs + 1}$$

$$v_3 = \frac{-CRV_1 s + V_1}{CC_2 RR_2 s^2 + C_2 Rs + 1}$$

$$v_4 = 0$$

$$v_5 = \frac{CC_2 RR_2 V_1 s^2 - C_2 R_2 V_1 s}{CC_2 RR_2 s^2 + C_2 Rs + 1}$$

3.2.2 Transfer Function

The voltage transfer function

$$H(s) = \frac{v_2(s)}{v_1(s)}$$

for the reduced complexity network is:

$$H(s) = \frac{s(-CR - C_2 R - C_2 R_2)}{CC_2 RR_2 s^2 + C_2 Rs + 1}$$

The equation above is different than [1, eq (8)], which is missing the s term in the numerator.

3.2.2.1 Characteristic Polynomial

The characteristic polynomial for $R_1 = R_3 = R$ and $C_1 = C_3 = C$, is shown below.

$$CC_2 RR_2 s^2 + C_2 Rs + 1$$

The coefficients of the Laplace variable, s, can be represented by a, b, and c in the quadratic equation:

$$as^2 + bs + c = 0$$

where:

$$a = CC_2 RR_2$$

$$b = C_2 R$$

$$c = 1$$

The poles of the transfer function are:

$$\left[\frac{-C_2 R - \sqrt{C_2 R(-4CR_2 + C_2 R)}}{2CC_2 RR_2}, \ \frac{-C_2 R + \sqrt{C_2 R(-4CR_2 + C_2 R)}}{2CC_2 RR_2} \right]$$

The pole are complex if, $C_2 R < 4CR_2$.

3.2.3 Natural Frequency, Q Factor and Damping Ratio

As described in Q factor, a two pole filter has the following characteristic polynomial:

$$s^2 + \frac{\omega_n}{Q}s + \omega_n^2$$

By equating the coefficients, ω_n, Q Factor and Damping Ratio can be expressed in terms of R_2, R, C_2 and C.

$$\omega_n = \sqrt{\frac{1}{CC_2 RR_2}}$$

$$Q = CR_2 \sqrt{\frac{1}{CC_2 RR_2}}$$

The expression for Q in [1, eq. (12)] is different from the SymPy derived expression for Q above. It seems that SymPy does not want to fully simplify expressions containing radicals. Rather than complicating the code in an attempt to get SymPy to reduce the expression for Q to its simplest form, we will just use it as is.

From Q factor:

$$\frac{\omega_n}{Q} = 2\zeta\omega_n = 2\alpha$$

Solving for the damping ratio, ζ:

$$\zeta = \frac{1}{2CR_2 \sqrt{\frac{1}{CC_2 RR_2}}}$$

The attenuation parameter, α, represents the rate of decay of the oscillations of the system.

$$\alpha = \frac{1}{2CR_2}$$

3.3 Numerical Analysis

After substituting the element values from the netlist into the equations with the following code:

```
equ_N = NE_sym.subs(element_values)
```

we get the following numerical equations:

$$0 = I_{V1} - 1.0 \cdot 10^{-8} s v_5 + v_1 \cdot (1.0 \cdot 10^{-8} s + 6.6667 \cdot 10^{-5}) - 6.6667 \cdot 10^{-5} v_3$$

$$0 = I_{O1} - 1.0 \cdot 10^{-8} s v_3 + v_2 \cdot (1.0 \cdot 10^{-8} s + 6.6667 \cdot 10^{-5}) - 6.6667 \cdot 10^{-5} v_5$$

$$0 = -1.0 \cdot 10^{-8} s v_2 - 9.0 \cdot 10^{-10} s v_4 - 6.6667 \cdot 10^{-5} v_1 + v_3 \cdot (1.1 \cdot 10^{-8} s + 6.6667 \cdot 10^{-5})$$

$$0 = -9.0 \cdot 10^{-10} s v_3 + v_4 \cdot (1.0 \cdot 10^{-9} s + 5.376 \cdot 10^{-6}) - 5.376 \cdot 10^{-6} v_5$$

$$0 = -1.0 \cdot 10^{-8} s v_1 - 6.6667 \cdot 10^{-5} v_2 - 5.376 \cdot 10^{-6} v_4 + v_5 \cdot (1.0 \cdot 10^{-8} s + 7.2043 \cdot 10^{-5})$$

$$1.0 = v_1$$

$$0 = v_4$$

SymPy can solve for unknown node voltages and the unknown currents from the V_1 and the Op Amp. The unknown currents, I_{V1} and I_{O1}, are not displayed. The node voltages are:

$$v_1 = 1.0$$

$$v_2 = \frac{-1.53414746543779 \cdot 10^{18} s^2 - 1.02276497695853 \cdot 10^{22} s}{117000000000000.0 s^3 + 8.42903225806452 \cdot 10^{17} s^2 + 5.02764976958526 \cdot 10^{21} s + 3.07219662058372 \cdot 10^{25}}$$

$$v_3 = \frac{1.53609831029186 \cdot 10^{24} - 3.45622119815668 \cdot 10^{16} s^2}{5850000000000.0 s^3 + 4.21451612903226 \cdot 10^{16} s^2 + 2.51382488479263 \cdot 10^{20} s + 1.53609831029186 \cdot 10^{24}}$$

$$v_4 = 0.0$$

$$v_5 = \frac{5850000000000.0 s^3 - 2.6 \cdot 10^{20} s}{5850000000000.0 s^3 + 4.21451612903226 \cdot 10^{16} s^2 + 2.51382488479263 \cdot 10^{20} s + 1.53609831029186 \cdot 10^{24}}$$

The network transfer function, $H(s) = \frac{v_2(s)}{v_1(s)}$, is:

$$H(s) = \frac{-1.53414746543779 \cdot 10^{18} s^2 - 1.02276497695853 \cdot 10^{22} s}{117000000000000.0 s^3 + 8.42903225806452 \cdot 10^{17} s^2 + 5.02764976958526 \cdot 10^{21} s + 3.07219662058372 \cdot 10^{25}}$$

The SciPy function, TransferFunction, is used to represent the system as the continuous-time transfer function.

```
H_num, H_denom = fraction(H) #returns numerator and denominator

# convert symbolic to numpy polynomial
a = np.array(Poly(H_num, s).all_coeffs(), dtype=float)
b = np.array(Poly(H_denom, s).all_coeffs(), dtype=float)

sys_tf = signal.TransferFunction(a,b)
```

3.3.1 Pole Zero Plot

The poles and zeros of the transfer function can easily be obtained with the following code:

```
sys_zeros = np.roots(sys_tf.num)
sys_poles = np.roots(sys_tf.den)
```

The poles and zeros of the transfer function are plotted on the complex frequency plane.

```
plt.plot(np.real(sys_zeros/(2*np.pi)), np.imag(sys_zeros/(2*np.pi)), 'ob',
    markerfacecolor='none')
plt.plot(np.real(sys_poles/(2*np.pi)), np.imag(sys_poles/(2*np.pi)), 'xr')
plt.legend(['Zeros', 'Poles'], loc=0)
plt.title('Pole / Zero Plot')
plt.xlabel('real part, \u03B1')
plt.ylabel('imaginary part, j\u03C9')
plt.grid()
plt.show()
```


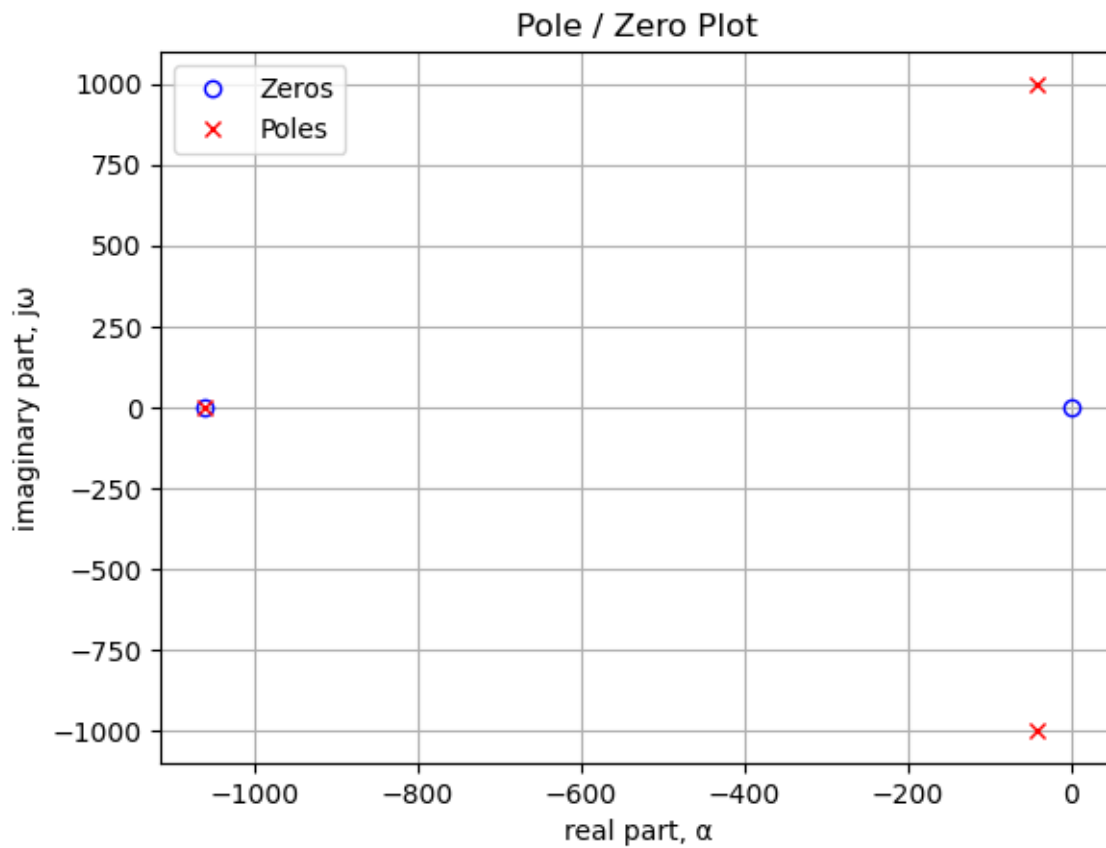Figure 5: A plot of the pole and zero locations for circuit in Figure 1.

The code below generates a table that lists the values of the pole and zero locations.

```
table_header = ['Zeros, Hz', 'Poles, Hz']
num_table_rows = max(len(sys_zeros),len(sys_poles))
```

23

```python
table_row = []

for i in range(num_table_rows):
    if i < len(sys_zeros):
        z = '{:.2f}'.format(sys_zeros[i]/(2*np.pi))
    else:
        z = ''
    if i < len(sys_poles):
        p = '{:.2f}'.format(sys_poles[i]/(2*np.pi))
    else:
        p = ''
    table_row.append([z,p])

Markdown(tabulate(table_row, headers=table_header,
    colalign = ('left','left'), tablefmt="simple"))
```

Table 2: Table of poles and zeros for circuit in Figure 1.

Zeros, Hz	Poles, Hz
-1061.03	-1061.03+0.00j
0.00	-42.78+997.93j
	-42.78-997.93j

The zero at -1061.033 Hz and the pole at -1061.033+0.000j Hz cancel.

3.3.2 Magnitude and Phase Response

The plot of the filter's magnitude and phase response for the transfer function $H(s)$ is shown below.

```python
x = np.logspace(1.9, 4.1, 1000, endpoint=True)*2*np.pi # x axis data values
w, mag, phase = signal.bode(sys_tf, w=x) # returns: rad/s, mag in dB and
                                         # phase in deg

fig, ax1 = plt.subplots()
ax1.set_ylabel('magnitude, dB')
ax1.set_xlabel('frequency, Hz')

plt.semilogx(w/(2*np.pi), mag,'-b')     # magnitude plot
```

```
ax1.tick_params(axis='y')
plt.grid()

# instantiate a second y-axes that shares the same x-axis
ax2 = ax1.twinx()
color = 'tab:red'

plt.semilogx(w/(2*np.pi), phase,':',color='tab:red')  # phase plot

ax2.set_ylabel('phase, deg',color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.title('Magnitude and Phase')
plt.show()
```
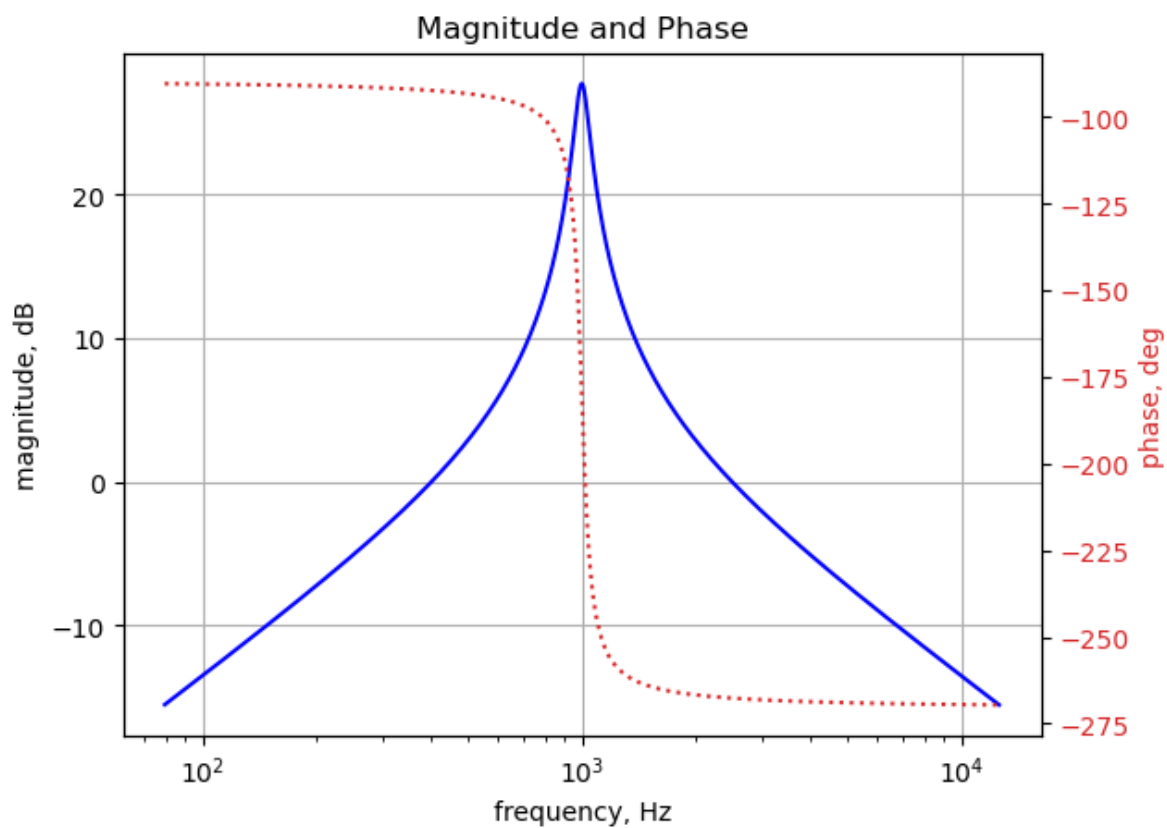

Figure 6: Magnitude and phase plot for the circuit in Figure 1.

The filter has a maximum gain of 27.7 dB at 997.5 Hz and the -3dB frequency is 1044.0 Hz. The attenuation to the right of the peak is 41.3 dB per decade and the filter's quality factor, Q, is 12.32.

The filter Q factor can also be calculated from the element values.

```
Markdown('Calculated Q factor: {:.3f}'.format(N(Q_factor.subs({R:15000,
    R2:186000, C:1e-8, C2:9.1e-10})))))
```

Calculated Q factor: 11.673

The calculated Q factor is a bit different than the Q measured from the frequency response plot.

The envelope of oscillation decays proportional to $e^{-\alpha t}$, where α is the attenuation parameter calculated above.

```
alpha = atten_param.subs({R:15000,R2:186000,C:1e-8,C2:9.1e-10})
Markdown('$\\alpha$ = {:.3f}'.format(alpha))
```

$\alpha = 268.817$

3.3.3 Step Response

The step response of the is calculated using the SciPy function lsim, which can be used to simulate output of a continuous-time linear system from the continuous-time linear time invariant system base class. The function `lsim` allows us to evaluate the performance characteristics of the circuit to square wave input.

A square wave with a frequency of 25 Hz and an amplitude of one volt peak to peak is used as the input to the filter. The code below generates the plot.

```
# define the time interval and create a square wave step waveform.
t = np.linspace(0, .06, 500, endpoint=False)
sqr_signal = signal.square(2*np.pi*25*t, duty=0.5)

# call lsim to generate the response signal
t_step, y_step, x_step = signal.lsim(sys_tf, U=sqr_signal, T=t)

plt.plot(t_step*1e3, sqr_signal, 'r', alpha = 1.0, linewidth=1, label='input')
plt.plot(t_step*1e3, y_step,'b', linewidth = 1.0, label='output')

plt.plot((t_step[0:166]+0.02)*1e3, 3.8*np.exp(-float(alpha)*t_step[0:166]),
```

```
    'k', linewidth = 1.5, label='e^(- t)')

plt.title('Square Wave Response')
plt.ylabel('volts')
plt.xlabel('time, msec')
plt.grid()
plt.legend(loc='best')
# show plot
plt.show()
```
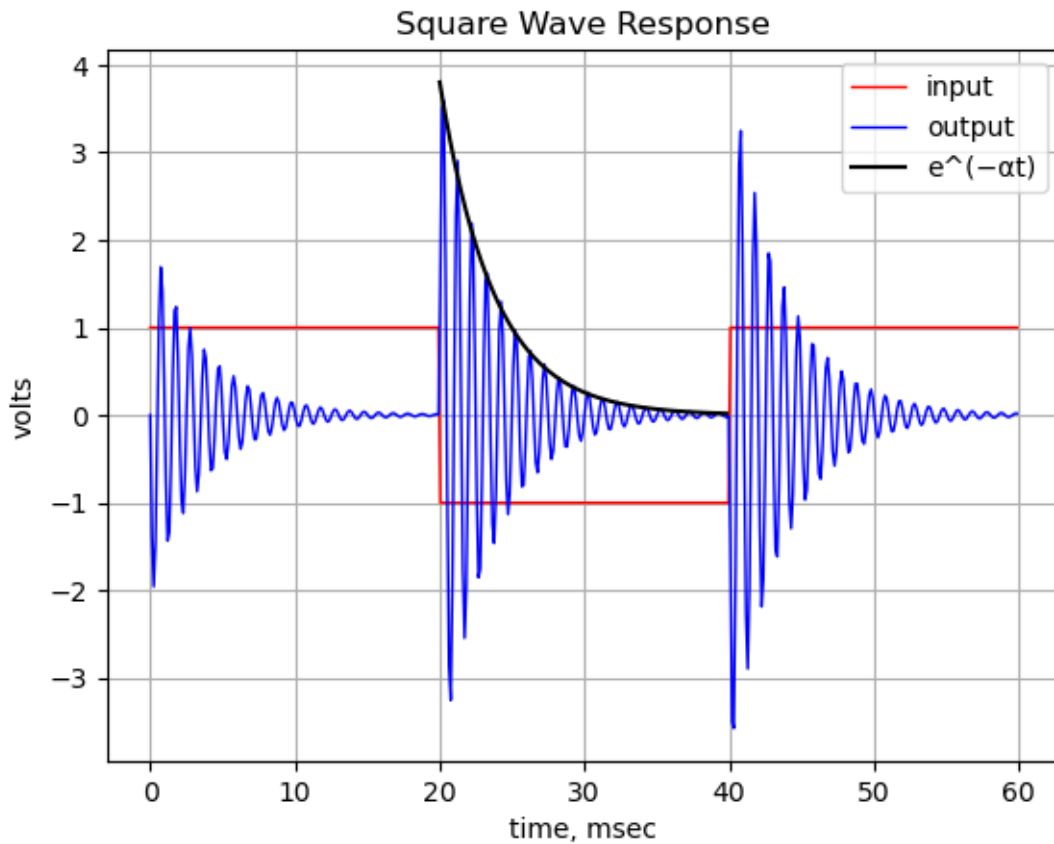

Figure 7: Square wave response of circuit in Figure 1 with decay plotted.

The filter's response to a square wave input shows a damped oscillation that occurs at each edge of the square wave. The frequency of the oscillation is about 10 cycles over 10 ms, which is 1000 Hz and corresponds to $\omega_n = 1\text{kHz}$. The decay time constant is about 10 ms.

3.3.4 Group Delay

Group delay is a measure of the time delay experienced by a group of frequencies as they pass through a system. It's essentially the rate of change of the phase response with respect to frequency. Group Delay is important for the following reasons:

- Signal distortion: Variations in group delay across different frequencies can cause signal distortion, affecting the quality of audio, video, and data transmission.
- System design: Understanding group delay is crucial for designing systems with linear phase characteristics, which minimize distortion.
- Pulse propagation: In fields like optics and telecommunications, group delay affects the shape and timing of pulses.

Group delay (τ_g) is calculated as the negative derivative of the phase response (ϕ) with respect to angular frequency (ω):

$$\tau_g(\omega) = -\frac{d\phi(\omega)}{d\omega}$$

A system with a constant group delay is called a linear phase system. These systems introduce a pure time delay to all frequency components without altering their relative phase relationships, preserving the original waveform shape. According to Audibility of Group-Delay Equalization, the threshold is 2 ms. The abstract for the paper states:

> The audibility thresholds for group-delay variation from several previous related studies are shown in Fig. 1. If not otherwise stated, these studies have been conducted using headphones. Green applied Huffman sequences, or truncated impulse responses of second-order allpass filters, to study the audibility of phase distortion. He found a threshold value for the peak group delay of about 2 ms for center frequencies of 625 Hz, 1875 Hz, and 4062 Hz.

```
plt.title('Group Delay')
plt.semilogx(w/(2*np.pi), -np.gradient(phase*np.pi/180)/np.gradient(w)*1000,
    '-',label='group delay')

plt.ylim((0,4))

plt.ylabel('Group delay, msec')
plt.xlabel('Frequency, Hz')
plt.grid()
plt.show()
```
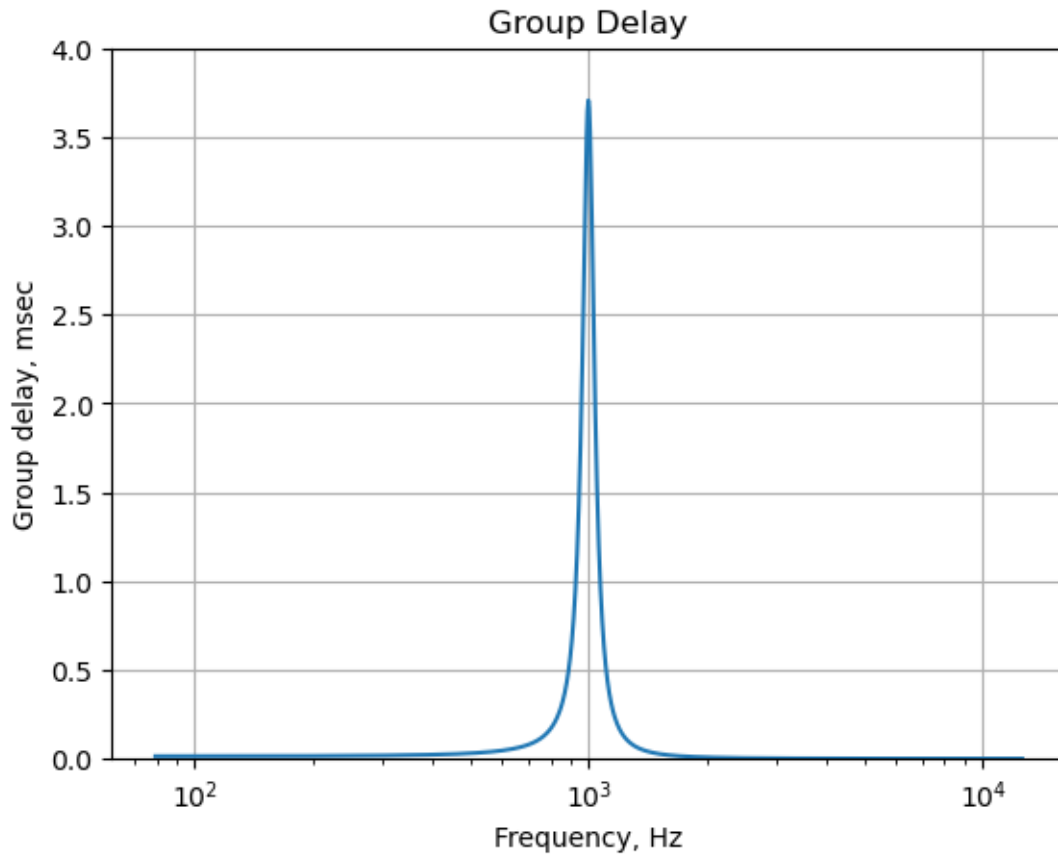
Figure 8: Group delay plot of the circuit in Figure 1.

3.4 Comparison with LTSpice

The transient and frequency response of the filter were simulated in LTSpice and the results are a close match with the Python results as shown in the plots below.
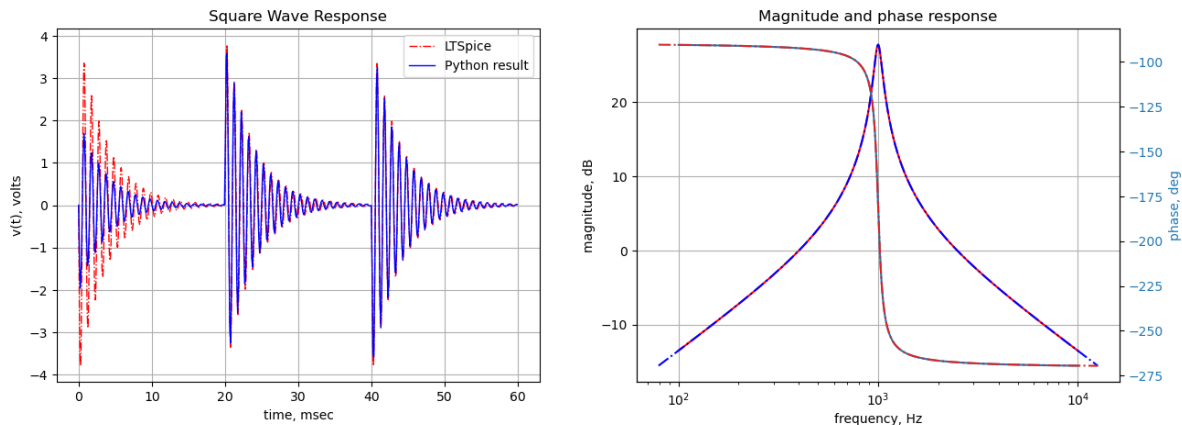
Figure 9: LTSpice and Python results for Figure 1 plotted for comparision.

In the plot on the left, the initial transient is different since the LTSpice input starts with a value -1 and the Python code has the initial value of the square wave equal to zero. Other than that difference, there is a one to one correspondence.

3.5 Filter Design Examples

This section will walk through the design steps to implement and analyze the filter.

3.5.1 Example 1, where $R_1 = R_3 = R$ and $C_1 = C_3 = C$

In this example, we want to design a filter with a center frequency of 1.4kHz and a filter Q of approximately 3. The filter in Figure 1 is simplified by using $R_1 = R_3 = R$ and $C_1 = C_3 = C$. Usually it's more convenient to pick capacitor values from standard values and solve for resistors. The standard values for resistors and capacitors follow the E-Series of numbers. The table below can be used to select values for C and R to put the filter response in the ball park for the desired center frequency.

Table 3: Suggested C and R values versus frequency.

Frequency Range	C	R
10 to 500 Hz	1 μF	10 kΩ
500 to 1 kHz	100 nF	10 kΩ
1 to 5 kHz	10 nF	10 kΩ
5 to 10 kHz	1 nF	5 kΩ

The code below will plot the range of R_2, C_2 and filter Q for a given center frequency, R and C. Note some of the lines of code were to long to display across the page, so I broke a few of the longer lines at commas and periods in the code to control where the lines are wrapped.

```python
E6_cap_nF_list = np.array([2.2, 3.3, 4.7, 6.8, 10, 15, 22, 33, 47])*1e-9

C2_plot = E6_cap_nF_list

R2_plot = np.zeros(len(C2_plot))
Q_plot = np.zeros(len(C2_plot))

freq = 1.4 # kHz
C_value = 10e-9
R_value = 10e3

fig, ax1 = plt.subplots()

# solve for R2 for each C2 in the capacitor list
for i in range(len(C2_plot)):
    R2_plot[i] = solve(omega_n.subs({R:R_value, C2:C2_plot[i],
        C:C_value})-2*np.pi*freq*1e3,R2)[0]
    Q_plot[i] = Q_factor.subs({R:R_value, R2:R2_plot[i],
        C:C_value, C2:C2_plot[i]})
    plt.annotate('{:.1f}nF'.format(C2_plot[i]*1e9), xy=(C2_plot[i]*1e9,
        R2_plot[i]/1e3), xytext=(5,2), textcoords='offset points')

ax1.set_ylabel('R2, k ohms')
ax1.set_xlabel('C2, nF')

# set the color to white to hide the plot
plt.plot(C2_plot*1e9,R2_plot/1e3,'x-w')
plt.xlim((0,55))

ax1.tick_params(axis='y')
plt.grid()

# instantiate a second y-axes that shares the same x-axis
ax2 = ax1.twinx()
color = 'k'

plt.plot(C2_plot*1e9,Q_plot,'x-')

ax2.set_ylabel('Q Factor',color=color)
```

```
ax2.tick_params(axis='y', labelcolor=color)

plt.title('C2 vs R2 for \u03C9={:.1f} kHz, R={:.0f}k, C={:,.0f} nF'.
    format(freq, R_value/1e3,C_value*1e9))
plt.show()
```
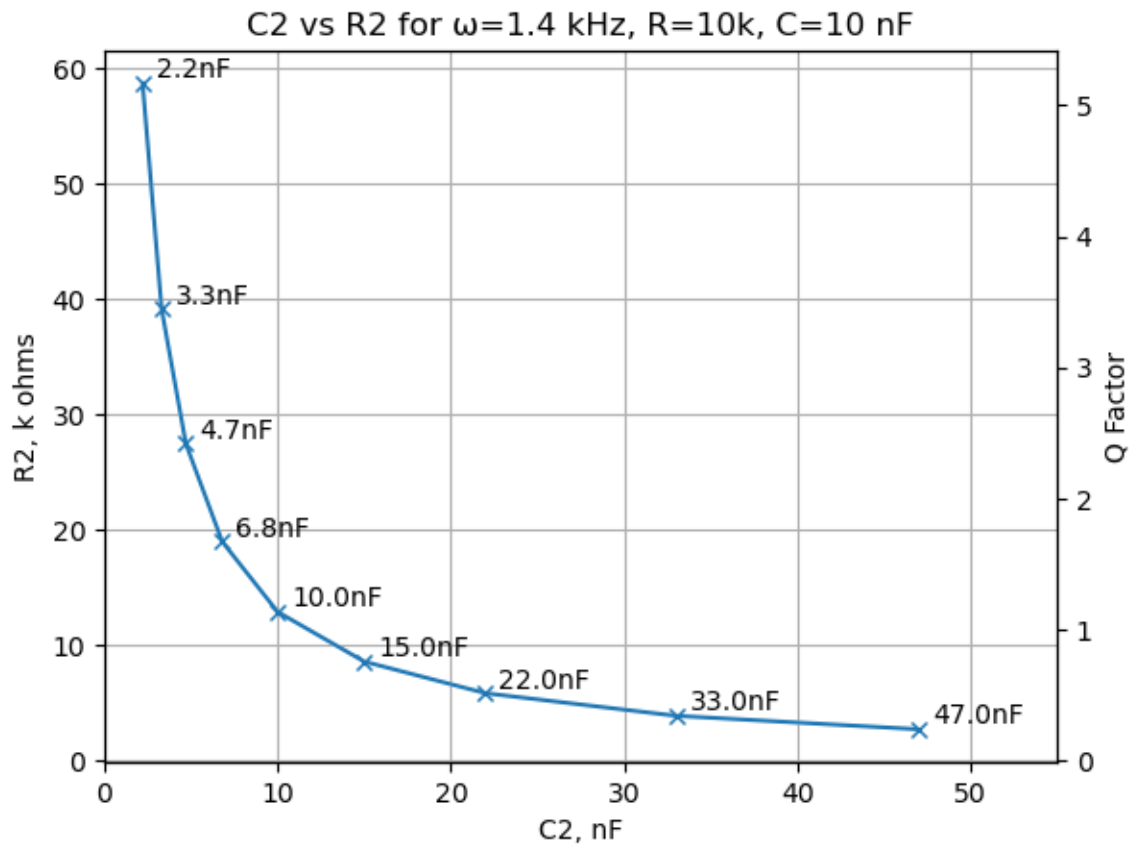

Figure 10: Various values of C2 versus R2 for 1.4 kHz and R = 10k and C = 10 nF.

In the code above the center frequency is set by the assignment: `freq = 1.4`, which has units of kHz. The desired Q is about 3, so $C_2 = 4.7$ nF is chosen.
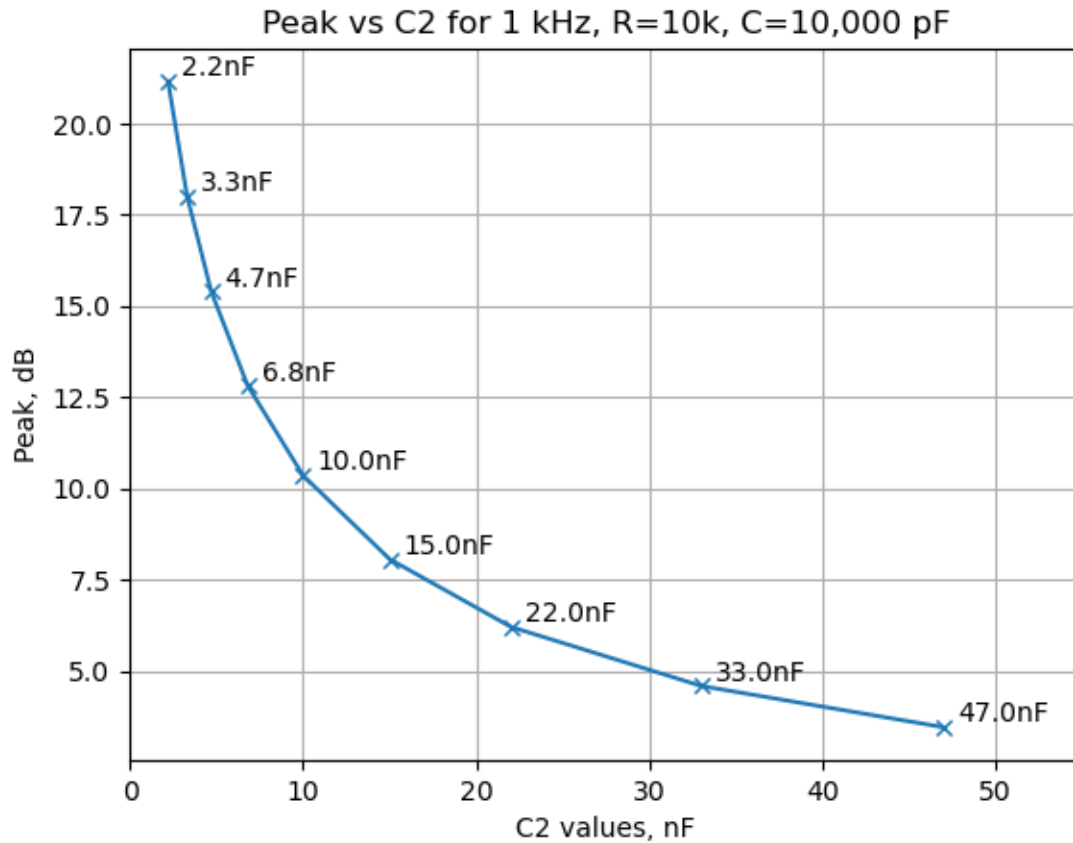
The plot below shows the filter's peak amplitude versus the value of C_2.

Figure 11: Peak filter amplitude versus various values of C2.

We can choose the value of C2, which is $C_2 = 4.7$nF and solve for R_2.

```
C2_value = 4.7e-9

R2_value = solve(omega_n.subs({R:R_value, C2:C2_value,
    C:C_value})-2*np.pi*freq*1e3,R2)[0]
Markdown('$R_2={:.3f}k$'.format(R2_value/1e3))
```

$R_2 = 27.497k$

The value happens to be very close to a standard value 1% metal film resistor. Setting R_2 equal to 27,500.

```
R2_value = 27500
```

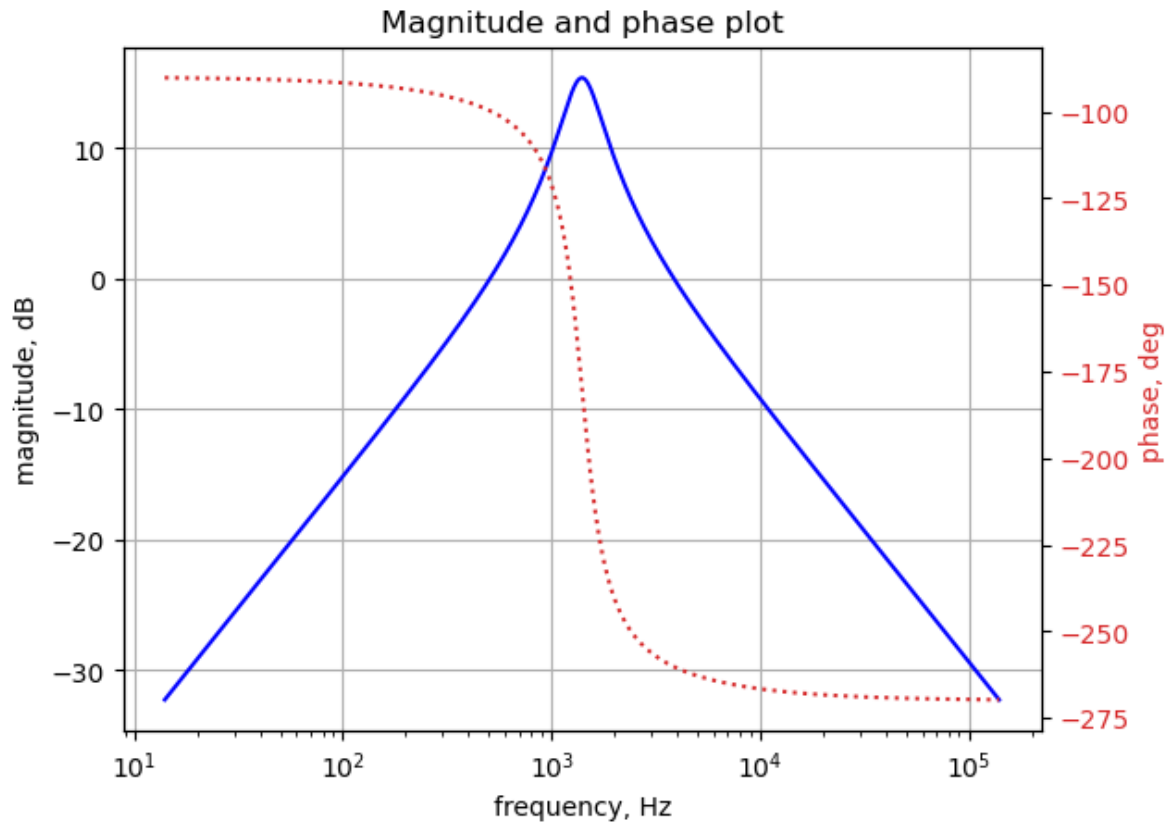The plot of the magnitude and phase are shown below.

Figure 12: Magnitude and phase response for example 1 filter.

The filter has a maximum gain of 15.4 dB at 1393.6 Hz and the -3dB frequency is 1722.7 Hz. The attenuation to the right of the peak is 27.6 dB per decade and the filter's quality factor, Q, is 2.49.

The filter Q factor can also be calculated from the element values.

```
Markdown('Calculated Q factor: {:.3f}'.format(N(Q_factor.subs({R:10e3,
    R2:27500, C:10e-9, C2:4.7e-9})))))
```

Calculated Q factor: 2.419
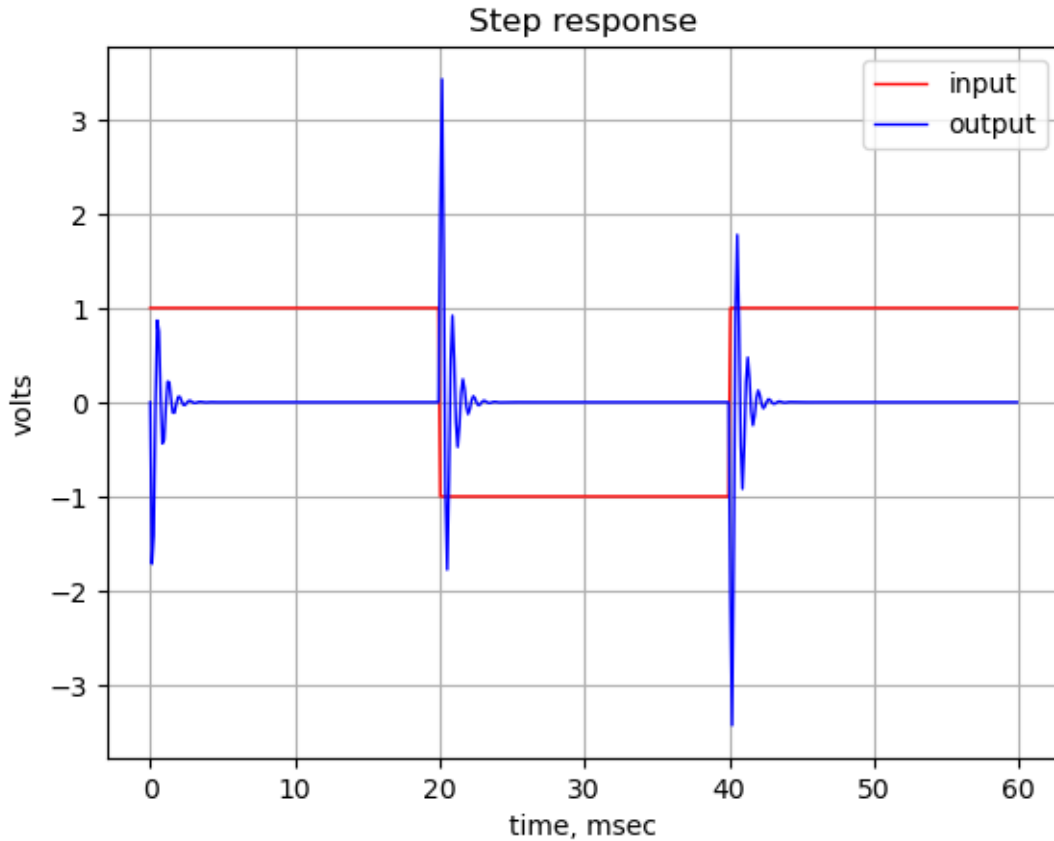
The square wave response is shown below.

Figure 13: Square wave response for example 1 filter.

3.5.2 Example 2, Design from Pole Locations

In this example given capacitor values and the pole locations, the values of the resistors are obtained with the help of SymPy's solver function. Three variables p_1, p_2 and p_3 are declared, which are the three poles of the characteristic polynomial.

I'll write the characteristic polynomial in the form, $(s + p_1)(s + p_2)(s + p_3)$. The code below expands and collects the terms of the equation.

```
d_equ = expand((s+p1)*(s+p2)*(s+p3)).collect(s)
Markdown('${:s}$'.format(latex(d_equ)))
```

$p_1 p_2 p_3 + s^3 + s^2 (p_1 + p_2 + p_3) + s (p_1 p_2 + p_1 p_3 + p_2 p_3)$

The coefficients of the variable s are:

$d_3 = 1$

$d_2 = p_1 + p_2 + p_3$

$d_1 = p_1 p_2 + p_1 p_3 + p_2 p_3$

$d_0 = p_1 p_2 p_3$

Dividing each term by d_0, we get the coefficients in canonical form.

$d_3 = \frac{1}{p_1 p_2 p_3}$

$d_2 = \frac{p_1 + p_2 + p_3}{p_1 p_2 p_3}$

$d_1 = \frac{p_1 p_2 + p_1 p_3 + p_2 p_3}{p_1 p_2 p_3}$

$d_0 = 1$

The coefficients of the filter's characteristic equation were obtained above and are:

$a_3 = C_1 C_2 C_3 R_1 R_2 R_3$

$a_2 = C_2 C_3 R_1 R_2 + C_2 C_3 R_1 R_3$

$a_1 = C_2 R_1 + C_3 R_1$

$a_0 = 1$

The `scipy.signal` module provides a set of functions for designing and analyzing analog filters. The SciPy functions to design classic analog filters are:

- `scipy.signal.buttap`: Butterworth filter
- `scipy.signal.cheb1ap`: Chebyshev type I filter
- `scipy.signal.cheb2ap`: Chebyshev type II filter
- `scipy.signal.besselap`: Bessel filter

These functions return (z,p,k) for an analog prototype filter; the second parameter, p, are the poles of the filter. Commenting out the other filter types, we can use `cheb2ap` to calculate the pole locations for this type of filter to use in this example.

```
N = 3 # filter order
Wn = 1 # normalised frequency

#filter_param = signal.besselap(N)
#filter_param = signal.buttap(N)
rp = 0.8
rs = 5.0
#filter_param = signal.cheb1ap(N,rp)
filter_param = signal.cheb2ap(N,rs)
```

The pole locations are tabulated below.

Table 4: table of poles and zeros

Pole #	Value, rad/s
1	-0.22-1.02j
2	-2.48-0.00j
3	-0.22+1.02j

In the code below the normalized values of C_1, C_2 and C_3 were chosen arbitrarily. It might be necessary to adjust these values if the resistor values don't end up being positive real numbers. The poles can be calculated from SciPy filter functions or chosen to be either three positive real numbers or one positive real number and a pair of complex conjugate numbers.

The values for the poles are entered into the code as positive numbers since we defined the characteristic polynomial in terms of $(s + p_1)(s + p_2)(s + p_3)$.

```
C1_value = 0.7
C2_value = 0.2
C3_value = 0.3
p1_value = -filter_param[1][0]
p2_value = -filter_param[1][1]
p3_value = -filter_param[1][2]

sub_values = {C1:C1_value, C2:C2_value, C3:C3_value, p1:p1_value,
    p2:p2_value, p3:p3_value}
```

Equating the coefficients of s we get:

$d_1 = a_1$

$d_2 = a_2$

$d_3 = a_3$

In the expanded version of these equations, we see that there are three equations and three unknowns, which are R_1, R_2 and R_3, since the values for the capacitors and poles have been assigned.

$d_3 = \frac{1}{p_1 p_2 p_3} = a_3 = C_1 C_2 C_3 R_1 R_2 R_3$

$d_2 = \frac{p_1 + p_2 + p_3}{p_1 p_2 p_3} = a_2 = C_2 C_3 R_1 R_2 + C_2 C_3 R_1 R_3$

$d_1 = \frac{p_1 p_2 + p_1 p_3 + p_2 p_3}{p_1 p_2 p_3} = a_1 = C_2 R_1 + C_3 R_1$

Using the SymPy function `solve`, to solve for the resistor values, the solution is:

```
sol_values = solve(((d3-a3).subs(sub_values), (d2-a2).subs(sub_values),
    (d1-a1).subs(sub_values)),(R1,R2,R3))
Markdown('${:s}$'.format(latex(sol_values[0])))
```

$(1.61172938442011, \ 0.51227330881633, \ 10.6011406578226)$

There are two solutions, so we can just arbitrarily choose the first one. Using the second solution means that the value for R_2 and R_3 are swapped.

```
R1_value = sol_values[0][0]
R2_value = sol_values[0][1]
R3_value = sol_values[0][2]
```

The values for the resistors and capacitors can be put into the network equations, which are displayed below.

$$0 = I_{V1} - 0.7sv_5 + v_1 \cdot (0.7s + 0.620451553) - 0.620451553v_3$$

$$0 = I_{O1} - 0.3sv_3 + v_2 \cdot (0.3s + 0.094329472) - 0.094329472v_5$$

$$0 = -0.3sv_2 - 0.2sv_4 - 0.620451553v_1 + v_3 \cdot (0.5s + 0.620451553)$$

$$0 = -0.2sv_3 + v_4 \cdot (0.2s + 1.952082966) - 1.952082966v_5$$

$$0 = -0.7sv_1 - 0.094329472v_2 - 1.952082966v_4 + v_5 \cdot (0.7s + 2.046412438)$$

$$V_1 = v_1$$

$$0 = v_4$$

Solving for voltages and currents in terms of Laplace variable s, we get:

$$v_1 = V_1$$

$$v_2 = \frac{-1.54018451100814 \cdot 10^{44} V_1 s^2 - 2.20352198180951 \cdot 10^{44} V_1 s}{8.4 \cdot 10^{42} s^3 + 2.45569492532383 \cdot 10^{43} s^2 + 1.84138955312286 \cdot 10^{43} s + 2.28498601678766 \cdot 10^{43}}$$

$$v_3 = \frac{-8.19874845657021 \cdot 10^{43} V_1 s^2 + 2.28498601678766 \cdot 10^{43} V_1}{8.4 \cdot 10^{42} s^3 + 2.45569492532383 \cdot 10^{43} s^2 + 1.84138955312286 \cdot 10^{43} s + 2.28498601678766 \cdot 10^{43}}$$

$$v_4 = 0.0$$

$$v_5 = \frac{8.4 \cdot 10^{42} V_1 s^3 - 2.34107469483772 \cdot 10^{42} V_1 s}{8.4 \cdot 10^{42} s^3 + 2.45569492532383 \cdot 10^{43} s^2 + 1.84138955312286 \cdot 10^{43} s + 2.28498601678766 \cdot 10^{43}}$$

The network transfer function, $H(s) = \frac{v_2(s)}{v_1(s)}$ is:

$$H(s) = \frac{-1.54018451100814 \cdot 10^{44} s^2 - 2.20352198180951 \cdot 10^{44} s}{8.4 \cdot 10^{42} s^3 + 2.45569492532383 \cdot 10^{43} s^2 + 1.84138955312286 \cdot 10^{43} s + 2.28498601678766 \cdot 10^{43}}$$

The poles and zeros of the transfer function are plotted below:
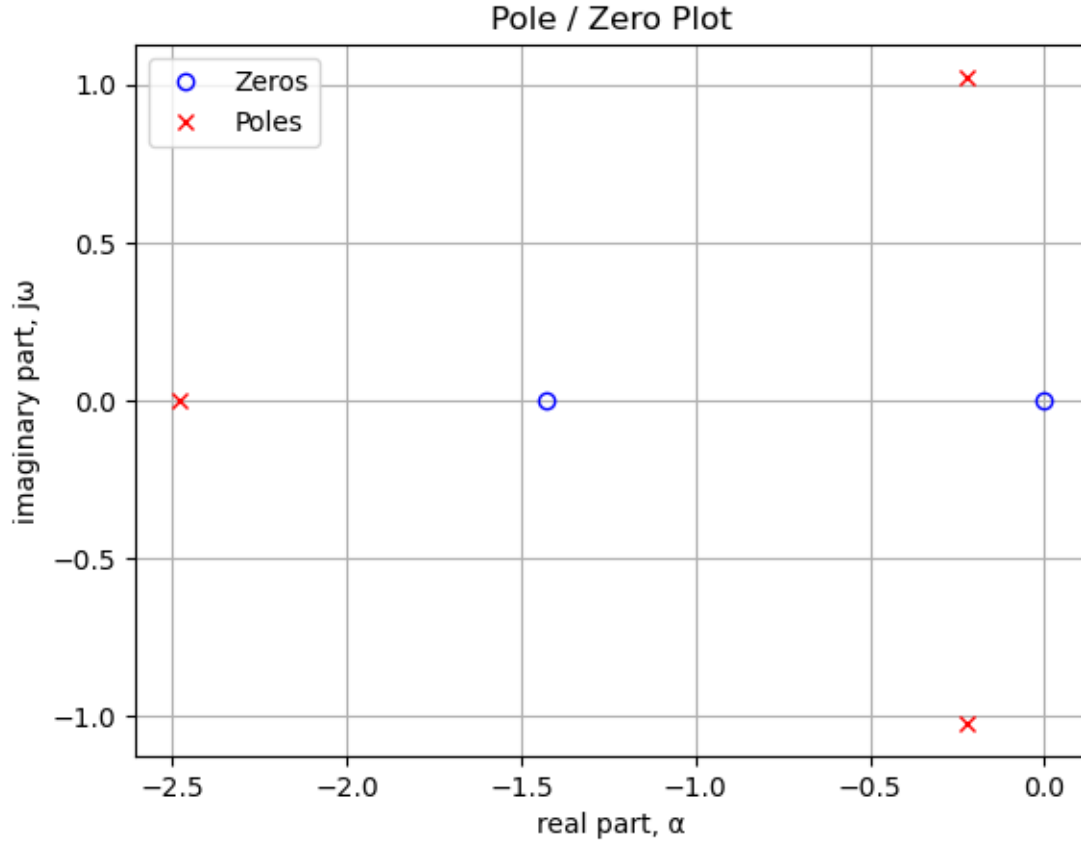
Figure 14: Pole and zero locations for example 2 filter.

The units of the poles and zeros are in radian frequency and are tabulated below.

Table 5: Pole and zero values for example 2 filter.

Zeros, rad/s	Poles, rad/s
-1.43	-2.48+0.00j
0.00	-0.22+1.02j
	-0.22-1.02j

The values of the poles network are the same as chosen above, thus validating the solution.

The magnitude and phase of the filter's transfer function is plotted below.
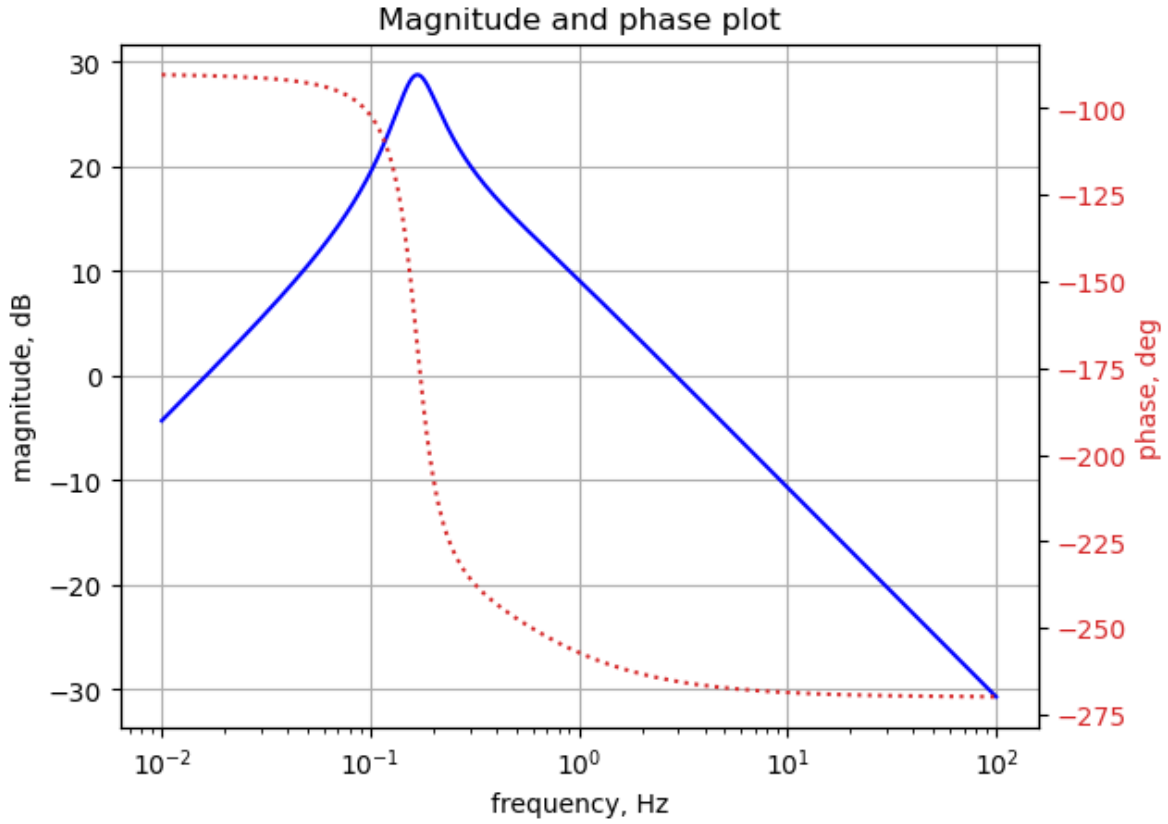
Figure 15: Magnitude and phase response plot for example 2 filter.

The filter has a maximum gain of 28.7 dB at 0.17 Hz and the -3dB frequency is 0.2 Hz. The attenuation to the right of the peak is 24.0 dB per decade and the filter's quality factor, Q, is 2.43.
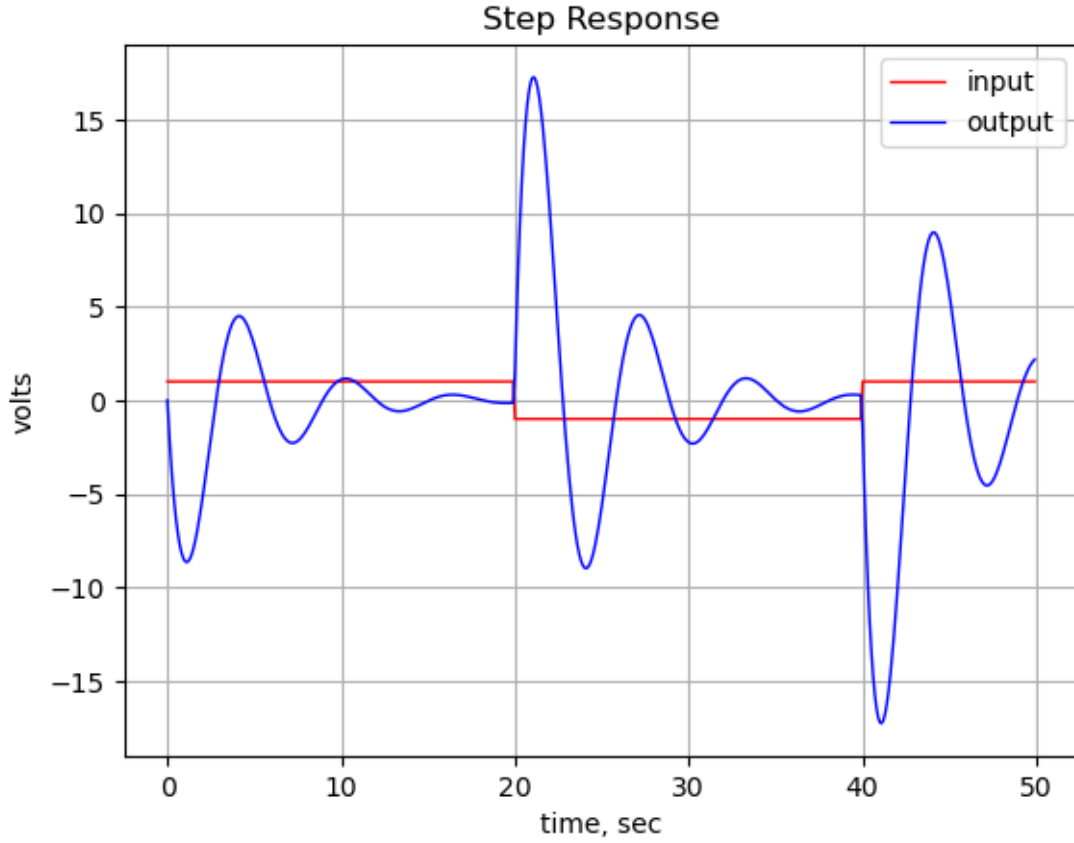
The step response is plotted below.

Figure 16: Square wave response for example 2 filter.

The values of the resistors and capacitors can be frequency scaled to put the center frequency of the filter at a different location in the frequency domain.

3.5.3 Example 3, Time Domain Solution

In this example, the use of SymPy's inverse Laplace transform function is examined.

Using $U_{sym}[v_2]$ from above to get the expression for the node 2's voltage.

$$v_2(t) = \frac{-C_1 C_2 R_1 R_3 V_1 s^2 - C_1 C_2 R_2 R_3 V_1 s^2 - C_1 C_3 R_1 R_3 V_1 s^2 - C_1 R_3 V_1 s - C_2 R_2 V_1 s - C_2 R_3 V_1 s}{C_1 C_2 C_3 R_1 R_2 R_3 s^3 + C_2 C_3 R_1 R_2 s^2 + C_2 C_3 R_1 R_3 s^2 + C_2 R_1 s + C_3 R_1 s + 1}$$

Substituting in the values for the components,

```
C2 = 4.7e-9
R2 = 27500
C1 = C3 = 10e-9
```

41

```
R1 = R3 = 10e3
```

and setting $V_1 = \frac{1}{s}$, which is the Laplace function for a unit step, we get:

```
v2_ex3_s = U_sym[v2].subs({V1:1/s, R1:10e3, R2:27500, R3:10e3,
    C1:10e-9, C2:4.7e-9, C3:10e-9}).nsimplify().simplify().expand().together()
Markdown('$v_2(s)={:s}$'.format(latex(v2_ex3_s)))
```

$$v_2(s) = -\frac{11050000}{517s^2 + 1880000s + 40000000000}$$

The expression for v_2 is simplified with the chain of operators applied to the expression; nsimplify(), simplify(), expand() and together(). This helps SymPy solve the inverse Laplace transform.

The inverse Laplace transform can be calculated with the following code:

```
v2_t = inverse_laplace_transform(v2_ex3_s, s, t)

Markdown('$v_2(t)={:s}$'.format(latex(v2_t)))
```

$$v_2(t) = -\frac{85\sqrt{611}e^{-\frac{20000t}{11}}\sin\left(\frac{180000\sqrt{611}t}{517}\right)}{846}$$

The SymPy inverse Laplace transform seems to only work on simple expressions, like the 2nd order one above. It took SymPy about 20 seconds to solve the inversere on my laptop.

3.5.3.1 The Inverse Transform, Alternate Method

If we let $R_3 = 5k$, the expression for $v_2(s)$ does not simplify and the denominator is a 3rd order polynomial.

```
v2_ex3_s_alt = U_sym[v2].subs({V1:1/s, R1:10e3, R2:27500, R3:5e3,
    C1:10e-9, C2:4.7e-9, C3:10e-9}).nsimplify().simplify().expand().together()
Markdown('$v_2(s)={:s}$'.format(latex(v2_ex3_s_alt)))
```

$$v_2(s) = \frac{50000(-221s - 3244000)}{517s^3 + 12220000s^2 + 117600000000s + 800000000000000}$$

Using SciPy's residue function to compute the partial-fraction expansion, we get r, which corresponds to the residues of the poles and p, which are the poles ordered by magnitude in ascending order. These values are tabulated below.

```
n, d = fraction(v2_ex3_s_alt)
cn = Poly(n, s).all_coeffs()
cd = Poly(d, s).all_coeffs()
r, p, k = signal.residue(cn, cd, tol=0.001, rtype='avg')
```

Table 6: table residues and poles

residues	poles
-0.04+1.12j	-4123.53+9140.36j
-0.04-1.12j	-4123.53-9140.36j
0.07+0.00j	-15389.30+0.00j

The following code builds a symbolic expression for each of the residues and finds the inverse Laplace of each one and appends the expression to the result from the other poles.

```
v2_ex3_alt = 0
for i in range(len(r)):
    m = (r[i]/(s-p[i]))
    v2_ex3_alt += inverse_laplace_transform(m, s, t)

Markdown('$v_2(t)={:s}$'.format(latex(round_expr(v2_ex3_alt,3))))
```

$$v_2(t) \quad = \quad -\left(0.036 + 1.125i\right)e^{-4123.53t - 9140.357it} - \left(0.036 - 1.125i\right)e^{-4123.53t + 9140.357it} + 0.072e^{-15389.303t}$$

The expression for the inverse Laplace transform does look a bit unusual, but I think it's correct. We can use the SymPy function lambdify to transform the SymPy expressions to a lambda function which can be used to calculate the numerical values.

```
x = np.linspace(0, 0.002, 200, endpoint=True)
v2_alt = np.zeros(len(x),dtype = complex)

v2_alt = lambdify(t, v2_ex3_alt)(x)
```
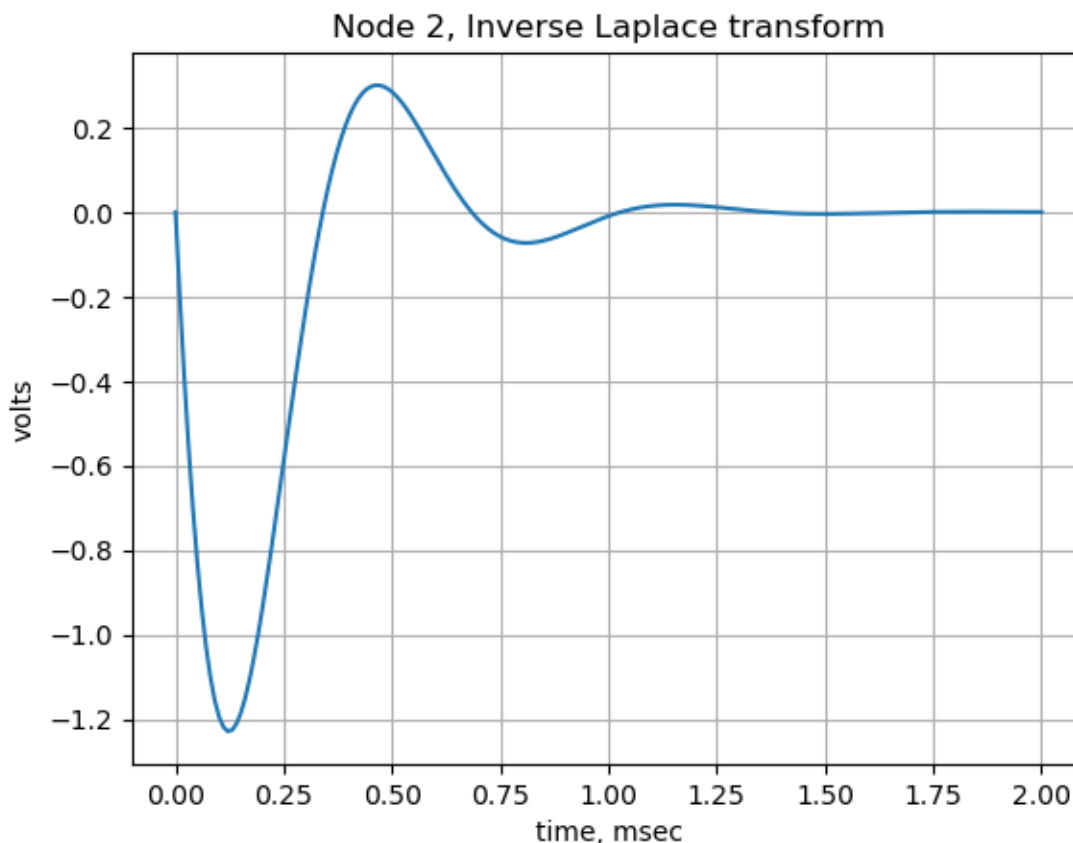
The plot of the inverse transform is shown below.

Figure 17: Inverse Laplace transform for node 2 of example 3.

4 DISCUSSION

Network equations for the filter were generated by the MNA procedure and solved symbolically. The resulting analytic expressions were rather long and didn't provide any insight into the circuit's operation. Symbolic solutions for the filter's poles and zeros also were not very insightful. After simplifying the filter by letting $R_1 = R_3$ and $C_1 = C_3$, the filter's characteristic polynomial is reduced to a second order polynomial. Then the traditional expressions for natural frequency and quality factor can be related to component values. A concept called Design Oriented Analysis was briefly mentioned and additional calculations are contained in a linked notebook.

The main advantage of the filter is that it uses only one Op Amp, whereas the traditional active BPF filters typically use two Op Amps. The amplitude response of the filter can't independently be controlled. This means that when deploying the filter in an equalizer circuit, variable resistors or voltage dividers are needed to set the relative peak gain for each band.

The sensitivity of the filter was calculated in a linked JupyterLab notebook, but the relative sensitivity was not compared to other active BPF's. This report could be used as a template to analyze the Sallen Key, State Variable, BiQuadratic and Dual Amplifier Bandpass filter topologies.

The validity of the Python based analysis was checked with LTSpice and there is a one to one correspondence.

In [1], there were references to two prior works that examined the Twin-T topology in a bandpass configuration that could possibly implement a lead-lag network. This idea was explored a bit in example 4, where an all-pass topology was implemented with the help of a second Op Amp. The location of the poles and zeros didn't end up being symmetrical about the origin of the complex frequency plane.

With some minor adjustments to the filter's topology, lowpass, highpass and band reject filters can be implemented. This idea was explored in example 4. This feature might be useful in that a common PCB layout could be used to instantiate various types of filters by including jumpers and extra pads to accommodate either resistors or capacitors in the circuit's branches.

4.1 Conclusion

The filter's transfer function naturally has two zeros and three poles which can be reduced to a second order characteristic polynomial when $R_1 = R_3$ and $C_1 = C_3$. The simplified case was analyzed with symbolic coefficients. Analytical expressions were derived for network equations. SymPy was used to solve for the node voltages along with obtaining equations for the poles and zeros. The expressions for the poles in symbolic form are rather long and do not provide much insight into the operation of the filter.

This report has shown that with the help of Python and SymPy, electric LTI circuits can be easily analyzed using a mixture of symbolic and numerical methods when following the template presented in this report. A JupyterLab notebook can be used as the "source code" for publishing reports or documents with the help of Quarto.

Obtaining analytic expressions for this filter do not appear to be that useful, but are easily obtained with the Python code presented in this report. Most scholarly publications include some analytic expressions and the code presented here can easily generate some of those expressions.

The use of JupyterLab, Python and Quarto provide a convenient workflow for analyzing electric circuits and presenting the results in a PDF. However, if particular formatting is required, as for example many journal publications require, additional work would be required. Some online advice is only to use the JupyterLab notebook and Quarto for first and second drafts and then to export the report material to LaTex for publication.

References

[1] Aaron D. Lanterman, "A Bandpass Twin-T Active Filter Used in the Buchla 200 Electric Music Box Synthesizer", https://arxiv.org/abs/2411.11358, last accessed Jan 20, 2025.

[2] C. Ho and A. Ruehli and P. Brennan, "The modified nodal approach to network analysis", IEEE Transactions on Circuits and Systems, 1975

[3] M. Hayes, Lcapy: symbolic linear circuit analysis with Python, PeerJ. Computer science, 2022, https://doi.org/10.7717/peerj-cs.875

[4] Buchla Electronic Musical Instruments, Schematic, 10 Chan. Comb Filter Model 295(A), http://fluxmonkey.com/historicBuchla/295-10chanfilt.htm, last accessed Feburary, 24, 2025

[5] Symbolic Modified Nodal Analysis using Python, https://tiburonboy.github.io/Symbolic-Modified-Nodal-Analysis-using-Python/, last accessed February, 24, 2025

[6] O. Palusinski, Automatic Formulation of Circuit Equations, https://www2.engr.arizona.edu/~ece570/session3.pdf, last accessed January 11, 2024

[7] R. D. Middlebrook, Low-entropy expressions: the key to design-oriented analysis, Proceedings Frontiers in Education Twenty-First Annual Conference. Engineering Education in a New World Order, 1991

[8] Vatché Vorpérian, Fast Analytical Techniques for Electrical and Electronic Circuits, Cambridge University Press, 2002