Chebyshev Digital Filter Design Following an Analytical Approach

Ruben Johann van Staden

Module code:	EERI 414
Degree:	B.Eng Computer Electronic Engineering
Date:	2021/06/02

Student number: 30026792

TABLE OF CONTENTS

INTRODUCTION 1

1.1 Overview 1

1.2 Background 1

1.2.1 Filter Design Fundamentals 2

1.2.2 Chebyshev Filter 2

ANALYTICAL DESIGN 3

2.1 Analog Domain Edge Frequencies 3

2.2 Analog Lowpass Filter Design 4

2.2.1 Filter Order 5

2.2.2 Pole Calculation 5

2.2.3 Numerator Calculation 6

2.2.4 Transfer Function 7

2.3 Analog High-Pass Filter Design 7

2.4 Digital High-Pass Filter Design (Bilinear Transform) 8

2.5 Gray-Markel Realisation 8

2.5.1 Lattice Parameters 8

2.5.2 Feed-Forward Multipliers 10

SOFTWARE SOLUTION 12

3.1 Overview 12

3.2 Input Signal Generation 13

3.2.1 Linear Sweep 13

3.2.1.1 Time Domain Characteristics 13

3.2.1.2 Frequency Domain Characteristics 13

3.2.2 Log Sweep 14

3.2.2.1 Time Domain Characteristics 14

3.2.2.2 Frequency Components........ 15

3.2.3 Sine Sum........ 15

3.2.3.1 Time Domain Characteristics 16

3.2.3.2 Frequency Domain Characteristics 16

3.3 Analog Filter Design 17

3.3.1 Command Line Output........ 17

3.3.2 Spectrum Output 18

3.4 Digital Filter Design........ 20

3.5 Gray-Markel Realisation........ 22

CONCLUSION 26

REFERENCES........ 27

APPENDIX A - SOFTWARE SOURCE CODE 28

ChebyshevFilterGenerator.cpp........ 28

cSignalGenerator.h 39

cSignalGenerator.cpp........ 40

cFastFourierTransform.h 42

cFastFourierTransform.cpp 43

cFilterDesign.h 46

cFilterDesign.cpp 48

LIST OF TABLES

Table 1-1: Filter Design Specifications 1

Table 2-1: Analog Lowpass Filter Design Parameters 4

Table 2-2: Gray-Markel Lattice Parameters 10

Table 2-3: Gray-Markel Realization Parameters 11

LIST OF FIGURES

Figure 1-1: Normalized Magnitude Spectrum of a Digital Lowpass Filter 2

Figure 2-1: Gray-Markel Realization Lattice.. 9

Figure 3-1: Solution Overview - GUI 12

Figure 3-2: Linear Sweep (10Hz - 80kHz).. 13

Figure 3-3: Linear Sweep Frequency Components.. 14

Figure 3-4: Log Sweep (10Hz - 80kHz) 14

Figure 3-5: Log Sweep Frequency Components... 15

Figure 3-6: Sine Sum (10Hz - 80kHz).. 15

Figure 3-7: Sine Sum Frequency Components 16

Figure 3-8: CLI Output (Lowpass Filter Transfer Function) 17

Figure 3-9: Analog High Pass Filter Response 18

Figure 3-10: Analog Passband Edge Frequency 19

Figure 3-11: Analog Stopband Edge Frequency.. 19

Figure 3-12: Digital High Pass Filter Response 20

Figure 3-13: Digital Stopband Edge Frequency... 20

Figure 3-14: Digital Passband Edge Frequency 21

Figure 3-15: Filterer Comparison - Time Domain.. 23

Figure 3-16: Filter Comparison - Magnitude Spectrum 23

Figure 3-17: Filter Comparison - Phase Spectrum... 24

Figure 3-18: Realized Passband Edge Frequency... 24

Figure 3-19: Realized Stopband Edge Frequency 25

INTRODUCTION

This report entails a concise description of software-implemented digital filter design. For the purpose of this discussion, focus will fall on the steps taken to design a digital Chebyshev Type-1 High Pass filter that will adhere to a set of specifications. The realisation of this filter will also be analysed, by comparing signal characteristics in the frequency domain of an unfiltered and filtered signal.

1.1 Overview

Digital filter design follows a set of mathematical derivations to finally construct a filter configuration that would adhere to a certain set of specifications. An input signal with a maximum frequency of 80kHz, should be passed through a digital filter with a sample rate of 175 kHz. This specified sample rate adheres to the Nyquist limit stating that a sample rate of at least twice the highest frequency component contained in a signal, is needed to accurately reproduce an analog signal.

The stopband corner frequency of the high pass Chebyshev filter should be higher than 40kHz, where a minimum of 40dB stopband attenuation should occur. The passband of the filter should start at a frequency not exceeding 75kHz, with a maximum ripple of 0.3dB. As this value represents the upper-limit, a filter configuration will be designed to adhere to a 0.25dB ripple, the 0.3dB specification will still be adhered in the case of any discrepancies The chosen edge and corner frequencies, together with the complete set of specifications are shown in Table 1-1.

Table 1-1: Filter Design Specifications

Specification	**Value**	**Unit**
Sample Rate (F_T)	175 000	Hz
Passband Edge Frequency (F_P)	72 250	Hz
Stopband Corner Frequency (F_S)	42 500	Hz
Passband Ripple ($\alpha_{p,dB}$)	- 0.25	dB
Stopband Attenuation ($\alpha_{s,dB}$)	- 40	dB

1.2 Background

There are various approaches to digital filter design. This report entails an analytical approach as to derive a digital transfer function that closely relates to the desired filter configuration. As such, transfer function estimations based on the filter order and passband ripple through the use of design tables, will not be followed. The first step in designing the specified digital filter, will be to configure the filter for the correct digital edge frequencies by introducing the sampling rate.

Once these values are finalised, the appropriate analog edge frequencies will be calculated as an analog filter will first be designed to finally estimate the correct digital design.

1.2.1 Filter Design Fundamentals

Figure 1-1, shows the normalized magnitude spectrum of the basic analog lowpass filter, where the normalization refers to the unity gain in the passband. The maximum passband ripple is represented by the value $\frac{1}{\sqrt{1+\varepsilon^2}}$, where $\frac{1}{A}$ refers to the minimum stopband attenuation.

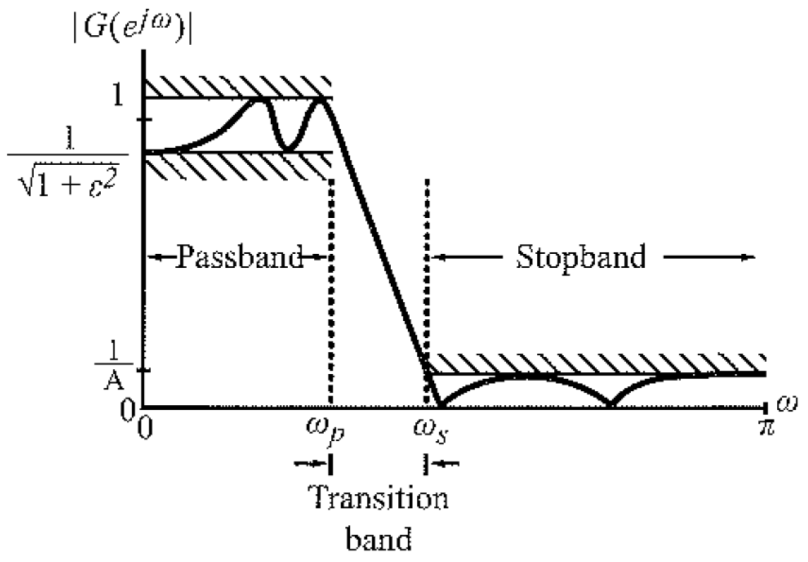


Figure 1-1: Normalized Magnitude Spectrum of a Digital Lowpass Filter

1.2.2 Chebyshev Filter

Digital Filters are cardinal components in any digital signal processing (DSP) application. They are used to eliminate unwanted frequencies such as noise, from signals. There exist two types of filter implementations, namely IIR and FIR digital filters. Infinite Impulse Response (IIR) digital filters are all characterised by having an Impulse Response which does not reach zero over time, as with Finite Impulse Response (FIR) filters [1].

Considering only IIR filters, two of the most widely used filters are Butterworth and Chebyshev filters [2]. IIR filters are commonly preferred over FIR filters due to much steeper roll off [3]. Two of the most widely used IIR digital filters are Butterworth and Chebyshev filters. The latter being widely used in medical applications to filter out redundant ECG data that gets added to the wanted signal due to noise [4]. Furthermore, they are also commonly used in image processing to extract rendering information as well as removing unwanted signal components [5].

ANALYTICAL DESIGN

In this section, the basic design steps followed to realise a digital high-pass Chebyshev filter, are laid out. The basic design process includes relating the digital cut-off frequencies to analog variants, designing an equivalent analog lowpass filter, transforming the designed filter to its high-pass variant, and finally relating the analog high-pass transfer function to the digital z-domain. The Gray-Markel realisation structure is also described.

2.1 Analog Domain Edge Frequencies

The basic process of any digital filter design, can be summed up in a few simple steps as characterised in [6]. The first step would be to define the stopband and passband region behaviour of the digital filter. Together with the sample rate, the normalized angular edge frequencies can be determined as all filter design techniques are developed in terms of these normalized frequencies.

$$\omega_p = \frac{2\pi F_P}{F_T} \tag{2-1}$$

$$\therefore \boldsymbol{\omega_p = 2.59406}$$

$$\omega_s = \frac{2\pi F_S}{F_T} \tag{2-2}$$

$$\therefore \boldsymbol{\omega_s = 1.52592}$$

It is important to note that these values relate to the angular frequencies in the digital domain. Thus, designing an analog filter for these specifications would render pointless, as the frequencies were not transformed to the analog domain. Such a transformation using the inverse bilinear transform, is shown below.

$$\widehat{\Omega}_p = \tan\frac{\omega_p}{2} \tag{2-3}$$

$$\therefore \boldsymbol{\widehat{\Omega}_p = 3.56102}$$

$$\widehat{\Omega}_s = \tan\frac{\omega_s}{2} \tag{2-4}$$

$$\therefore \boldsymbol{\widehat{\Omega}_s = 0.95610}$$

The specifications in section 1.1, specifies that a high-pass filter be designed. As such, the values $\widehat{\Omega}_p$ and $\widehat{\Omega}_s$ respectively refer to the analog high-pass passband and stopband edge frequencies.

The design process follows with an equivalent lowpass analog filter design. The edge frequencies need then to relate to the lowpass specifications.

$$\Omega_p = \widehat{\Omega}_s \tag{2-5}$$
$$\therefore \boldsymbol{\Omega_p = 0.95610\ \mathrm{rad/s}}$$

$$\Omega_s = \widehat{\Omega}_p \tag{2-6}$$
$$\therefore \boldsymbol{\Omega_s = 3.5610\ \mathrm{rad/s}}$$

The final conversion that needs to be made before the analog lowpass filter can be designed, is the conversion of the passband ripple and stopband attenuation from dB values to ratio values.

$$\alpha_{p,dB} = 20 \log \alpha_p \tag{2-7}$$
$$\alpha_p = 10^{-\frac{0.25}{20}}$$
$$\therefore \boldsymbol{\alpha_p = 0.97163}$$

$$\alpha_{s,dB} = 20 \log \alpha_s \tag{2-8}$$
$$\alpha_s = 10^{-\frac{40}{20}}$$
$$\therefore \boldsymbol{\alpha_s = 0.01}$$

The parameters needed for the equivalent analog lowpass filter design, is summarised in Table 2-1.

Table 2-1: Analog Lowpass Filter Design Parameters

Specification	**Value**	**Unit**
Passband Edge Frequency (Ω_p)	0.95610	rad/s
Stopband Corner Frequency (Ω_s)	3.5610	rad/s
Passband Ripple (α_p)	0.97163	
Stopband Attenuation (α_s)	0.01	

2.2 Analog Lowpass Filter Design

One approach to Analog filter design is through the use of filter order tables. In this report, the design follows an analytical approach to best estimate the digital transfer function needed for accurate filter realisation. As such, filter order tabled will not be used.

2.2.1 Filter Order

The first parameter to be calculated, is the order (N) of the Chebyshev Type-1 analog lowpass filter.

$$N \geq \frac{\cosh^{-1}\left(\sqrt{\frac{\frac{1}{{\alpha_s}^2}-1}{\frac{1}{{\alpha_p}^2}-1}}\right)}{\cosh^{-1}\left(\frac{\Omega_s}{\Omega_p}\right)} \qquad (2\text{-}9)$$

$$N \geq 3.37322$$

$$\therefore \boldsymbol{N \cong 4}$$

2.2.2 Pole Calculation

With the filter order known, we can calculate the analog lowpass filter transfer function. The poles of the transfer function can be represented by the equation below.

$$S_k = \mathrm{a}\cos(\phi_k) + jb\sin(\phi_k)\,, \qquad k = 0,1,..,N-1 \qquad (2\text{-}10)$$

The a and b values in the pole formula above, refers to the coefficients of the real and imaginary parts of the poles. These values, called the Minor Axis and Major Axis respectively, can be calculated once A and ε are known. The value ε can be obtained by assessing the maximum passband ripple as shown in Figure 1-1.

$$\alpha_p = \frac{1}{\sqrt{1+\varepsilon^2}} \qquad (2\text{-}11)$$

$$\varepsilon^2 = \frac{1}{{\alpha_p}^2} - 1$$

$$\therefore \boldsymbol{\varepsilon = 0.243421}$$

The value for A can be calculated next.

$$A = \varepsilon^{-1} + \sqrt{1+\varepsilon^{-2}} \qquad (2\text{-}12)$$

$$\therefore \boldsymbol{A = 8.33618}$$

The coefficients of the real and imaginary parts in the pole equation, can finally be calculated.

$$a = \Omega_p \frac{A^{1/N} - A^{-1/N}}{2} \qquad (2\text{-}13)$$

$$\therefore \boldsymbol{a = 0.530956}$$

$$b = \Omega_p \frac{A^{1/N} + A^{-1/N}}{2} \qquad (2\text{-}14)$$

$$\therefore \boldsymbol{b = 1.09364}$$

The ϕ_k relates to k values from 0 to N - 1.

$$\phi_k = \frac{\pi}{2} + \frac{(2k+1)\pi}{2N}, \qquad k = 0, 1, \ldots, N-1 \qquad (2\text{-}15)$$

$$\therefore \boldsymbol{\phi_0 = \frac{5}{8}\pi}$$

$$\therefore \boldsymbol{\phi_1 = \frac{7}{8}\pi}$$

$$\therefore \boldsymbol{\phi_2 = \frac{9}{8}\pi}$$

$$\therefore \boldsymbol{\phi_3 = \frac{11}{8}\pi}$$

The poles of the lowpass analog filter can finally be determined through the use of Equation 2-10.

$$S_k = \mathrm{a}\cos(\phi_k) + jb\sin(\phi_k), \qquad k = 0,1,..,N-1$$

$$\therefore \boldsymbol{S_0 = -0.203188 + j1.01039}$$

$$\therefore \boldsymbol{S_1 = -0.49054 + j0.418516}$$

$$\therefore \boldsymbol{S_2 = -0.203188 - j1.01039}$$

$$\therefore \boldsymbol{S_3 = -0.49054 - j0.418516}$$

Thus, the poles of the analog transfer function are found to be:

$$(s + 0.203188 \pm j1.01039)(s + 0.49054 \pm j0.418516) \qquad (2\text{-}16)$$

2.2.3 Numerator Calculation

To determination is dependent on the order of the filter. If the filter were to be of odd order, the numerator can be calculated by replacing the S value in the pole equation with 0. For even order filters, the 0-replacement value is then further divided by $\sqrt{1+\varepsilon^2}$. For the order 4 filter described above, this process will be followed.

$$Numerator = \frac{(0.203188 \pm j1.01039)(0.49054 \pm j0.418516)}{\sqrt{1+0.059254}} \qquad (2\text{-}17)$$

$$\boldsymbol{Numerator = 0.429103}$$

2.2.4 Transfer Function

Having calculated the denominator and the numerator, the final lowpass analog transfer function can be derived.

$$H_{LP}(s) = \frac{0.429103}{(s+0.203188 \pm j1.01039)(s+0.49054 \pm j0.418516)}$$

$$\therefore \boldsymbol{H_{LP}(s) = \frac{0.429103}{s^4 + 1.38746s^3 + 1.87664s^2 + 1.21104s + 0.441633}} \qquad (2\text{-}18)$$

2.3 Analog High-Pass Filter Design

The final analog lowpass transfer function described in 2-19, can be transformed into the relevant high pass equivalent, by replacing each occurrence of s in the transfer function, with the value 2-19.

$$s \Leftrightarrow \frac{\Omega_p \widehat{\Omega}_p}{s}$$

$$s \Leftrightarrow \frac{(0.95610)(3.5610)}{s}$$

$$s \Leftrightarrow \frac{3.404672}{s} \qquad (2\text{-}19)$$

After the transformation and some simplifying, the transfer function in 2-20 can be developed, describing the analog high pass Chebyshev Type-1 filter.

$$H_{HP}(s) = \frac{0.971628s^4}{s^4 + 9.336245s^3 + 49.257233s^2 + 123.989467s + 304.2563135} \qquad (2\text{-}20)$$

2.4 Digital High-Pass Filter Design (Bilinear Transform)

A simple bilinear transform can be performed to realize the high pass transfer function in 2-20, into the digital z-domain. The bilinear transform entails replacing each s-occurrence in the transfer function with the value in 2-21.

$$s \Leftrightarrow \frac{z-1}{z+1} \tag{2-21}$$

The transform delivers the final high pass digital Chebyshev Type-1 filter, adhering to the specifications set out in section 1.

$$H_{HP}(z) = \frac{0.0019917z^4 - 0.0079668z^3 + 0.011950z^2 - 0.0079668z + 0.0019917}{z^4 + 2.9565716z^3 + 3.5524476z^2 + 2.0164814z + 0.4534031} \tag{2-22}$$

2.5 Gray-Markel Realisation

The Gray-Markel technique was chosen to realize the designed filter transfer function in 2-22. As a fourth order digital filter is constructed, a fourth-order realization lattice needs to be used for realisation. A typical fourth-order Gray-Markel realization is shown in Figure 2-1.

2.5.1 Lattice Parameters

The high pass transfer function shown in 2-22, can be simplified to be in the form as described in 2-23.

$$H_{HP}(z) = \frac{p_0 + p_1 z^{-1} + p_2 z^{-2} + p_3 z^{-3} + p_4 z^{-4}}{d_0 + d_1 z^{-1} + d_2 z^{-2} + d_3 z^{-3} + d_4 z^{-4}} \tag{2-23}$$

$$H_{HP}(z) = \frac{0.0019917 - 0.0079668z^{-1} + 0.011950z^{-2} - 0.0079668z^{-3} + 0.0019917z^{-4}}{1 + 2.9565716z^{-1} + 3.5524476z^{-2} + 2.0164814z^{-3} + 0.4534031z^{-4}} \tag{2-24}$$

The first step in realising the transfer function, is to determine the different Lattice Parameters $k_1 \rightarrow k_4$. The fourth-order Allpass transfer function can be constructed, and evaluated for $z = \infty$, to determine the first parameter.

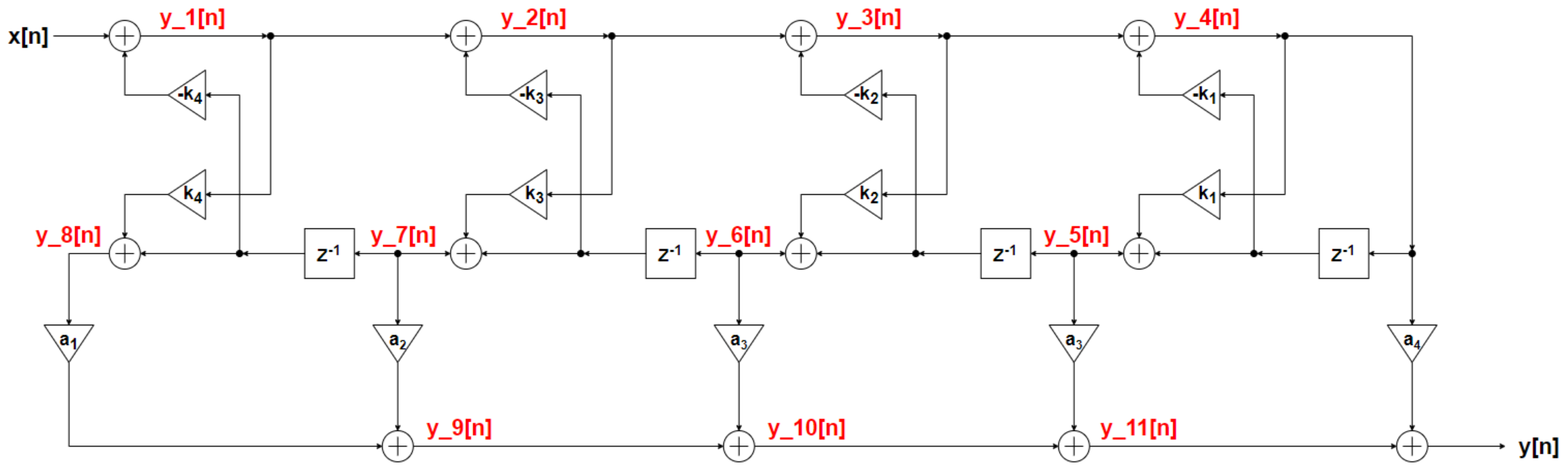


Figure 2-1: Gray-Markel Realization Lattice

$$A_4(z) = \frac{d_4 + d_3 z^{-1} + d_2 z^{-2} + d_1 z^{-3} + d_0 z^{-4}}{d_0 + d_1 z^{-1} + d_2 z^{-2} + d_3 z^{-3} + d_4 z^{-4}} \tag{2-25}$$

$$\therefore k_1 = A_4(\infty) = d_4 = 0.4534031 \tag{2-26}$$

To determine the second lattice parameter k_2, a third order Allpass transfer function can be determined as shown in 2-27.

$$A_3(z) = \frac{{d_3}' + {d_2}' z^{-1} + {d_1}' z^{-2} + {d_0}' z^{-3}}{{d_0}' + {d_1}' z^{-1} + {d_2}' z^{-2} + {d_3}' z^{-3}} \tag{2-27}$$

The coefficients ${d_0}' \rightarrow {d_3}'$, can be calculated following the Two-Pair Extraction Approach described below, where m relates to the order of the Allpass transfer function.

$${d_i}' = \frac{d_i - d_m d_{m-i}}{1 - {d_m}^2} \tag{2-28}$$

$$\therefore \boldsymbol{{d_1}' = 2.57077893}$$

$$\therefore \boldsymbol{{d_2}' = 2.444227354}$$

$$\therefore \boldsymbol{{d_3}' = 0.850882290}$$

Evaluating the Allpass transfer function in 2-27 for $z = \infty$, k_2 can be determined.

$$\therefore k_2 = A_3(\infty) = {d_3}' = 0.850882290$$

Repeating this process again for k_3 and k_4, we can derive these values as well. The Lattice Parameters relating to the realization of the high pass filter, is shown in Table 2-2.

Table 2-2: Gray-Markel Lattice Parameters

Lattice Parameters	Value
k_1	0.453403
k_2	0.850882
k_3	0.930427
k_4	0.921607

2.5.2 Feed-Forward Multipliers

With the Lattice Parameters known, the Feed-Forward Multipliers $\alpha_1 \rightarrow \alpha_5$, can be determined.

$$\alpha_1 = p_4 = 0.0019917 \quad (2\text{-}29)$$

$$\alpha_2 = p_3 - \alpha_1 d_1 = -0.0138554 \quad (2\text{-}30)$$

$$\alpha_3 = p_2 - \alpha_1 d_2 - \alpha_2 {d_1}' = 0.0404939 \quad (2\text{-}31)$$

$$\alpha_4 = p_1 - \alpha_1 d_3 - \alpha_2 {d_2}' - \alpha_3 {d_1}'' = -0.0501598 \quad (2\text{-}32)$$

$$\alpha_5 = p_0 - \alpha_1 d_4 - \alpha_2 {d_3}' - \alpha_3 {d_2}'' - \alpha_4 {d_1}''' = 0.021429 \quad (2\text{-}33)$$

The calculated Lattice Parameters and Feed-Forward Multipliers relating to the filter realization, is listed in Table 2-3.

Table 2-3: Gray-Markel Realization Parameters

Lattice Parameters	
k_1	0.453403
k_2	0.850882
k_3	0.930427
k_4	0.921607
Feed-Forward Multipliers	
α_1	0.0019917
α_2	- 0.0138554
α_3	0.0404939
α_4	- 0.0501598
α_5	0.021429

SOFTWARE SOLUTION

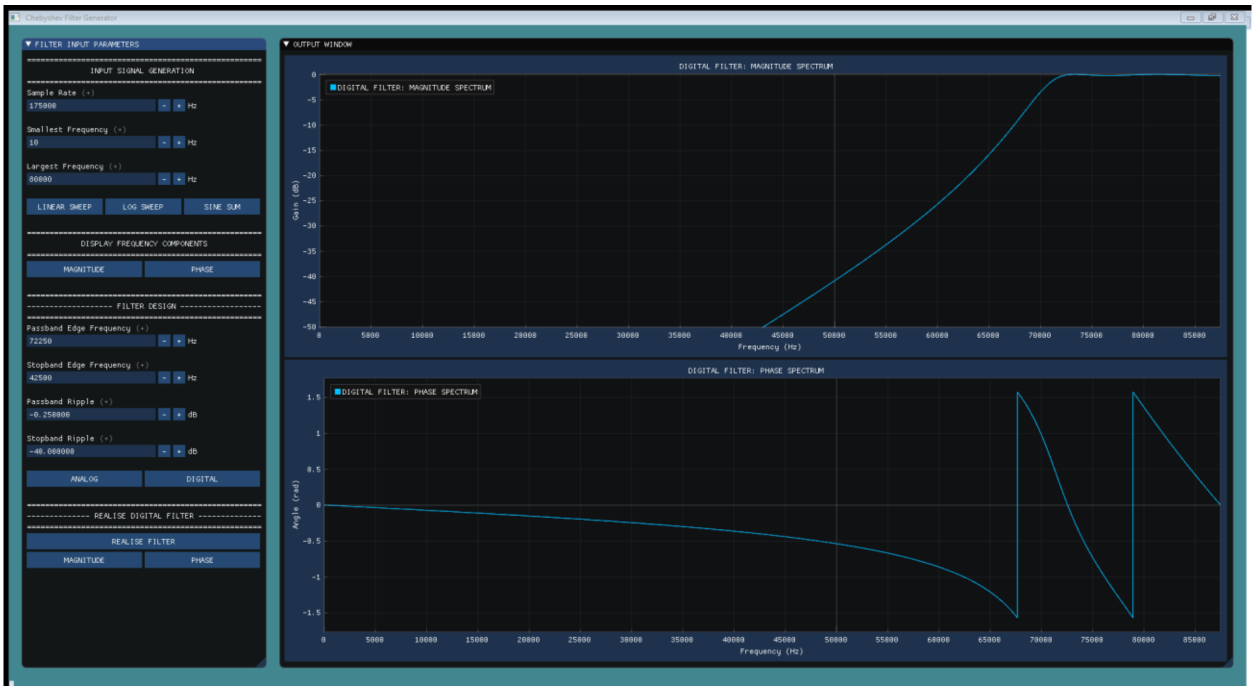
The purpose of this practical, is to realize a Chebyshev Type-1 High Pass filter based on a set of specifications. In this section, the software solution used for the filter design process, as well as the realization, is discussed. The solution will also be compared to the analytical calculations in section 2, as to determine whether the design was successfully implemented in code.

3.1 Overview

The software solution was written in Visual Studios C++. A graphical package called ImGui, was used to construct the GUI of the solution as shown in Figure 3-1. Furthermore, ImGui assisted with the plotting of the different graphs, through the use of the ImGui add-on, ImPlot.

The software was created with the intent to design a digital Chebyshev Type-1 filter based on any given input parameters. The filter design part of the software is thus not limited to the filter as specified in section 2. By varying the sample rate, edge frequencies, passband ripple and stopband attenuation, any filter configuration can be designed.

However, the realization is only based on the designed filter as discussed in this report. Bettering the program's ability to realize any given z-transfer function, could pave the way for the software to become a very useful Chebyshev Type-1 Filter generator.



Sample Rate (+) 175000 Hz

Smallest Frequency (+) 10 Hz

Largest Frequency (+) 80000 Hz

Passband Edge Frequency (+) 72250 Hz

Stopband Edge Frequency (+) 42500 Hz

Passband Ripple (+) -0.250000 dB

Stopband Ripple (+) -40.000000 dB

Figure 3-1: Solution Overview - GUI

3.2 Input Signal Generation

The software solution allows the user to generate three different signals as possible test inputs that can be used for filter realization. The signal characteristics is fully customizable as to enhance the user experience. Currently, the program only allows for a signal of 1 second duration to be generated.

3.2.1 Linear Sweep

The linear sweep option, allows the user to generate a signal with varying frequency components from the smallest to the largest frequency as specified by the user. It is important to note that the chosen sampling frequency plays a crucial part in the accuracy of the signal being generated. The Nyquist limit should always be followed as to ensure accurate signal generation.

3.2.1.1 Time Domain Characteristics

The time-domain representation of a linear sweep from 10Hz to 80kHz with sampling rate 175kHz is shown in Figure 3-2. It is interesting to note the small gaps in the signal. This is due to minimal aliasing introduced by the highest frequency nearing the sampling rate.

3.2.1.2 Frequency Domain Characteristics

A standard FFT algorithm was used to determine the magnitude and the phase components of the signal. For simplicity reasons, an FFT of length equal to the sampling rate was chosen, as to represent the different components. However, in practice the length of a transform is usually a factor of 2^x , with zeros padded to the input signal. This functionality is also possible through the use of the software package. The corresponding Magnitude and Phase components of the Linear Sweep is shown in Figure 3-3.

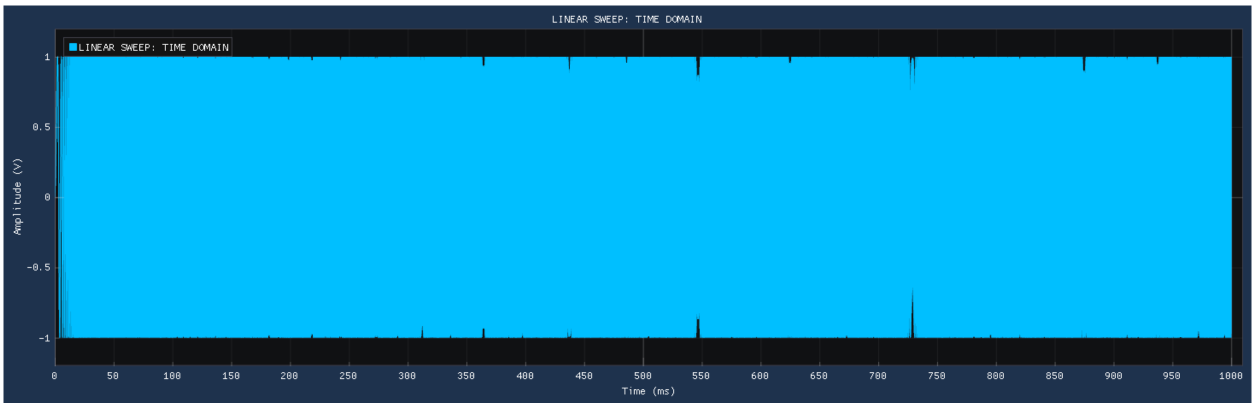


Figure 3-2: Linear Sweep (10Hz - 80kHz)

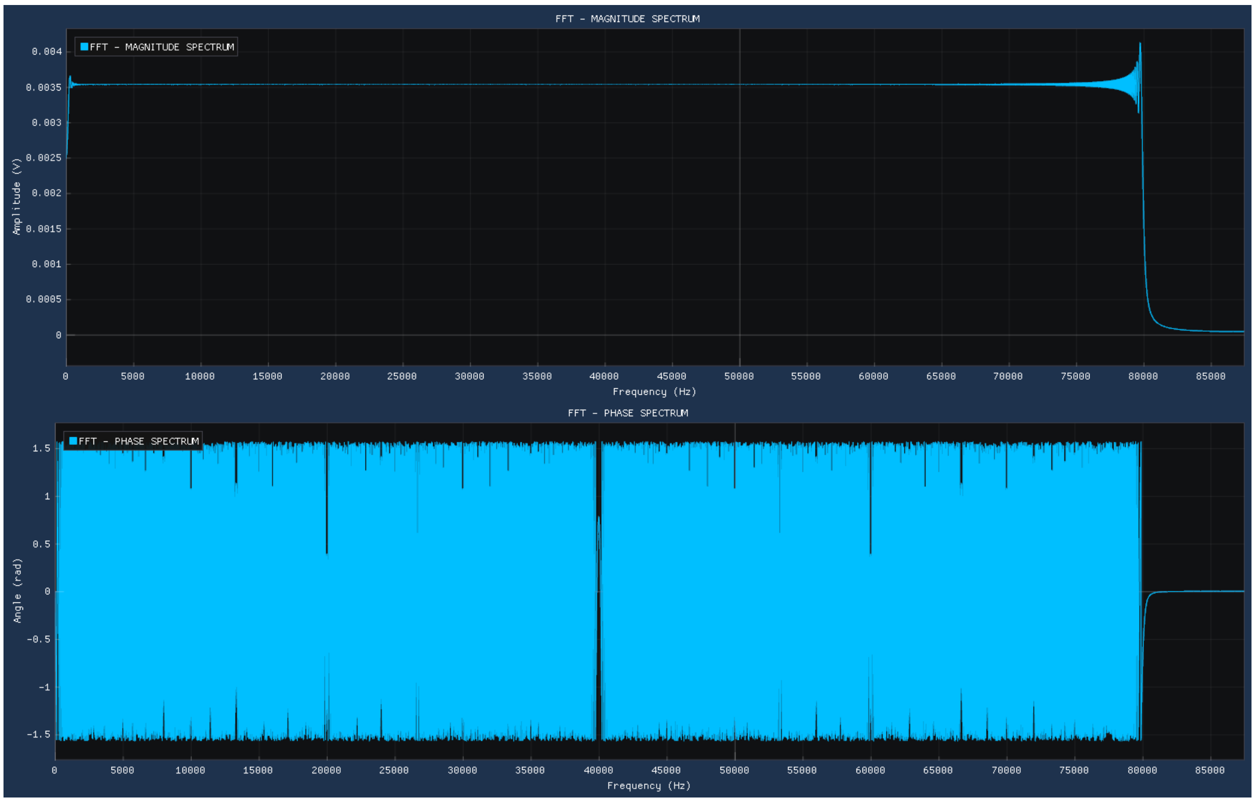


Figure 3-3: Linear Sweep Frequency Components

3.2.2 Log Sweep

The Log-Sweep option allows the user to sweep through the range of frequencies more effectively, by relating the frequency to a logarithmic ratio.

3.2.2.1 Time Domain Characteristics

For the same input parameters as was used to generate the linear sweep, the time-domain representation of a log-sweep is shown in Figure 3-4.

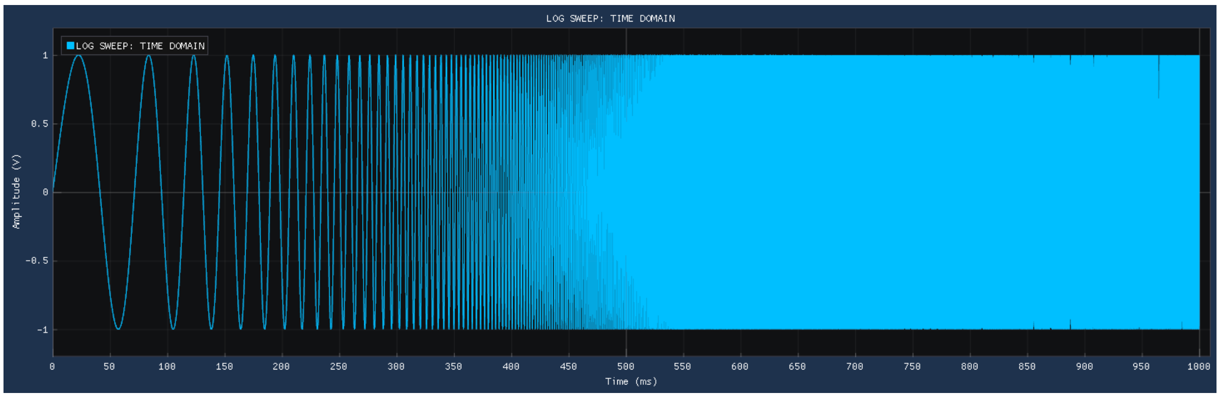


Figure 3-4: Log Sweep (10Hz - 80kHz)

3.2.2.2 Frequency Components

Passing the generated log-sweep through a standard FFT algorithm, delivers the magnitude and phase components as shown in Figure 3-5.

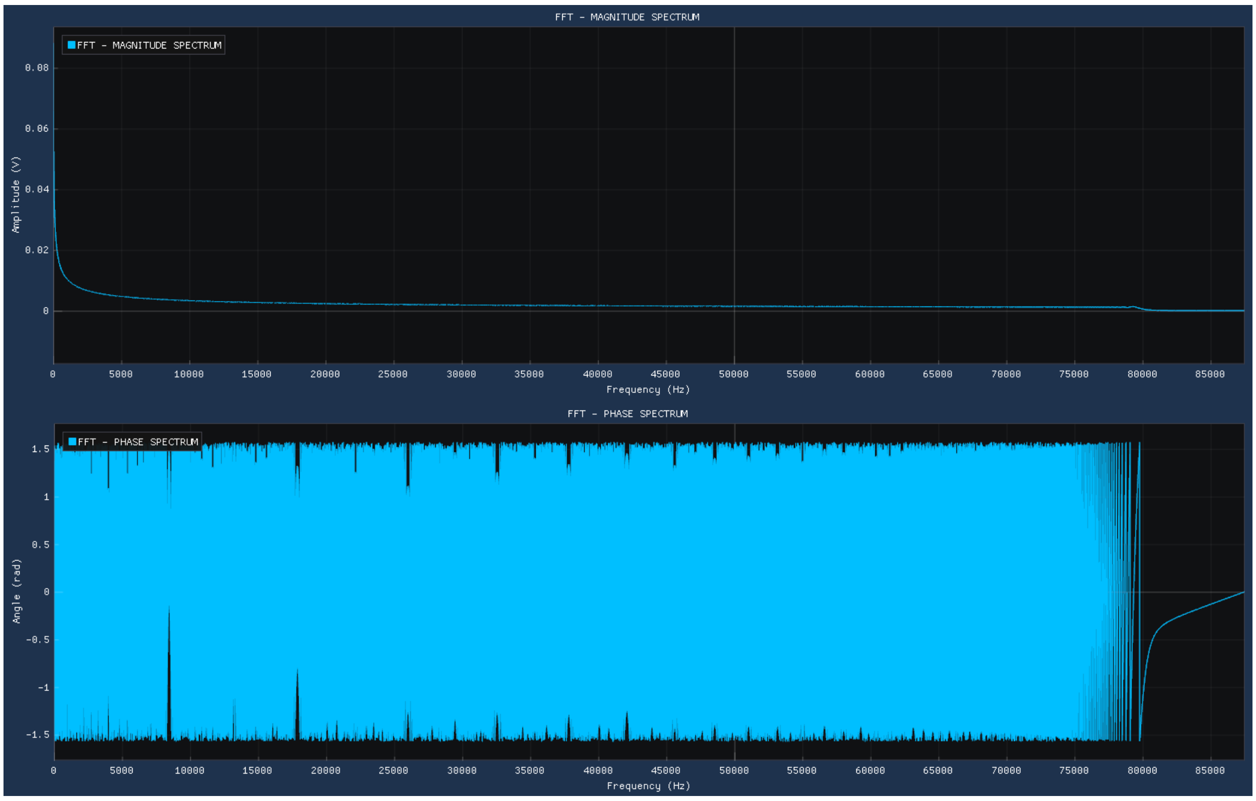


Figure 3-5: Log Sweep Frequency Components

3.2.3 Sine Sum

The Sine Summation generator, generates a signal with 500 equally distanced frequencies, having all the same unity magnitude. This generation technique delivers the best representation to assess whether the filter realization has been correctly implemented.

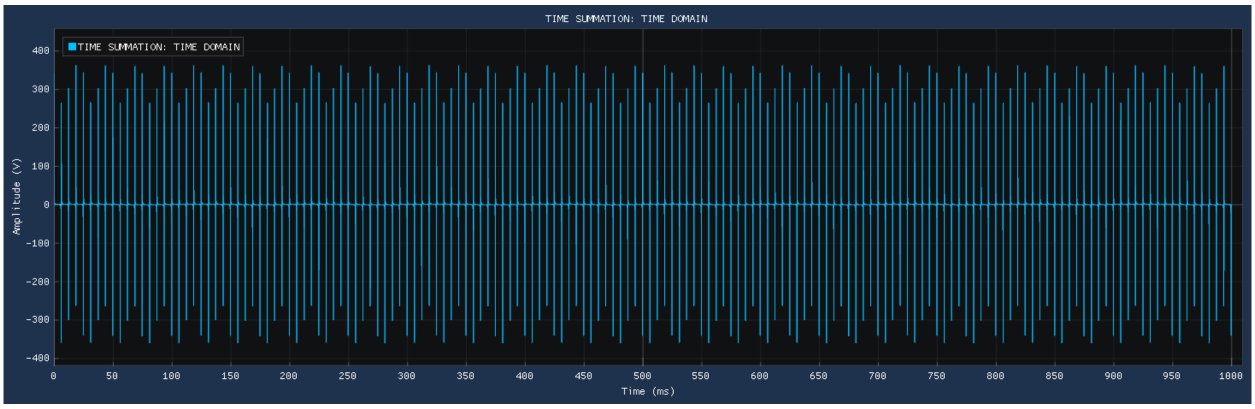


Figure 3-6: Sine Sum (10Hz - 80kHz)

3.2.3.1 Time Domain Characteristics

A signal generated for the specifications listed above, is shown in Figure 3-6. The shown signal comprises of 500 different frequency components as made clearer when observing the Magnitude Spectrum.

3.2.3.2 Frequency Domain Characteristics

Figure 3-7 shows the magnitude and phase spectrum of the generated signal in Figure 3-6. It is clear that this signal is the most desirable to test the filter realization. The magnitude spectrum reveals 500 equally distanced frequencies all having a magnitude of 1. The hypothesis then, is that when the digital filter is realized, the form of the magnitude spectrum will mimic the form of the digital filter, with maximum ripple in the passband of 0.25dB.

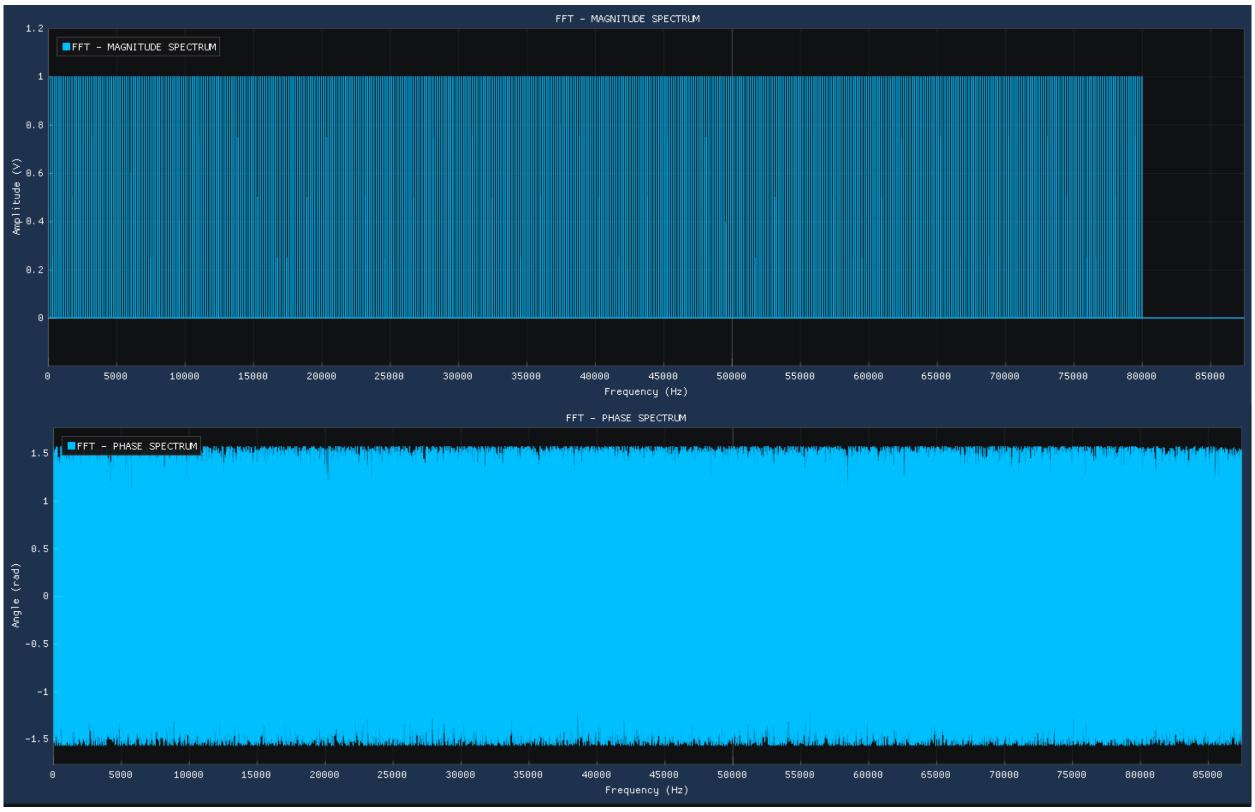


Figure 3-7: Sine Sum Frequency Components

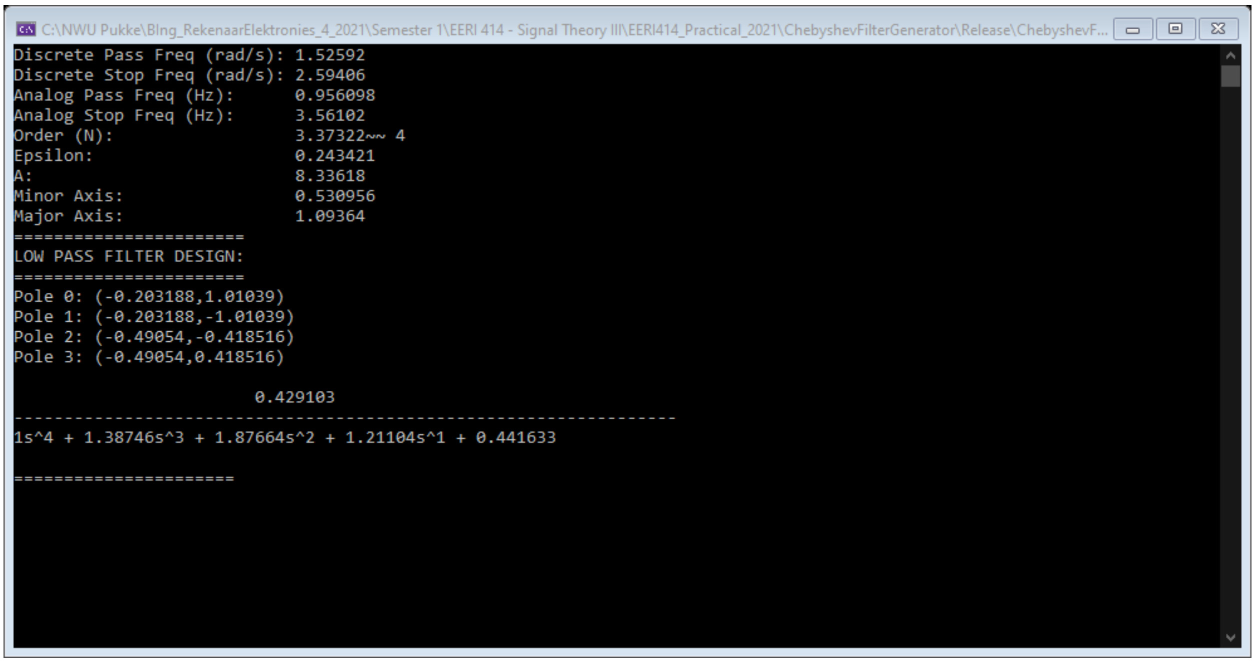
3.3 Analog Filter Design

The analog filter design process described in section 1, was followed very carefully when implementing the calculations in a software environment. The complete design of the filter, is realised in the cFilterDesign.h and cFilterDesign.cpp files attached as part of Appendix A. The software solution allows the user to implement any configuration filter, it is not limited to the one discussed in section 2. In this section, the results will be verified by comparing the designed analog filter to the analytical calculations.

3.3.1 Command Line Output

The analog filter parameters are calculated and shown to the user in the Command Line Interface (CLI). In addition, the analog filter's transfer function is also shown. By inputting the design specifications, as discussed in section 1, the user can click on the 'ANALOG' button in the 'FILTER DESIGN' section of the GUI, to display the magnitude and phase spectrum of the Analog Filter. The command line output is shown in Figure 3-8.

Only the lowpass equivalent filter's transfer function is shown in the CLI. The software gathers the appropriate high pass analog and digital output, by transforming the lowpass filter's spectrum directly. As such, the transfer functions are not shown. This can be included as a further improvement of the software. The analog design variables and the associated transfer function, corresponds with the calculated values as shown in section 1.


C:\NWU Pukke\BIng_RekenaarElektronies_4_2021\Semester 1\EERI 414 - Signal Theory III\EERI414_Practical_2021\ChebyshevFilterGenerator\Release\ChebyshevF...

```
Discrete Pass Freq (rad/s): 1.52592
Discrete Stop Freq (rad/s): 2.59406
Analog Pass Freq (Hz):      0.956098
Analog Stop Freq (Hz):      3.56102
Order (N):                  3.37322~~ 4
Epsilon:                    0.243421
A:                          8.33618
Minor Axis:                 0.530956
Major Axis:                 1.09364
=======================
LOW PASS FILTER DESIGN:
=======================
Pole 0: (-0.203188,1.01039)
Pole 1: (-0.203188,-1.01039)
Pole 2: (-0.49054,-0.418516)
Pole 3: (-0.49054,0.418516)

                        0.429103
------------------------------------------------------------------
1s^4 + 1.38746s^3 + 1.87664s^2 + 1.21104s^1 + 0.441633

======================
```

Figure 3-8: CLI Output (Lowpass Filter Transfer Function)

3.3.2 Spectrum Output

The magnitude and phase spectrum of the high pass analog filter, is shown below in Figure 3-9. As stated in the CLI output, as well as the analytical calculations in section 2, the maximum passband ripple of -0.25dB, should be noted at a normalized angular frequency of $\widehat{\Omega}_p = 3.56102$. This value can be transformed back into Hz, to compare the displayed results.

$$\widehat{\Omega}_p = 3.56102$$

$$\widehat{\mathrm{F}}_p = \frac{\widehat{\Omega}_p F_T}{2\pi}$$

$$\therefore \widehat{\mathbf{F}}_{\boldsymbol{p}} = \mathbf{99.182}\boldsymbol{kHz} \qquad (3\text{-}1)$$

By enlarging the displayed Analog magnitude response, the value at which the passband ripple occurs, is precisely to specification of 99.182kHz, as can be seen in Figure 3-10. The stopband edge frequency of $\widehat{\Omega}_s = 0.95610$, can similarly be transformed to a value in Hz. This value is calculated to be 26.629kHz. The minimum stopband attenuation at this point, should be 40dB. In reality, the design performs within specification, as the attenuation at the edge frequency was found to be much higher than the specified 40dB. Figure 3-11 shows this value to be 50.83dB, much higher than the minimum 40dB.

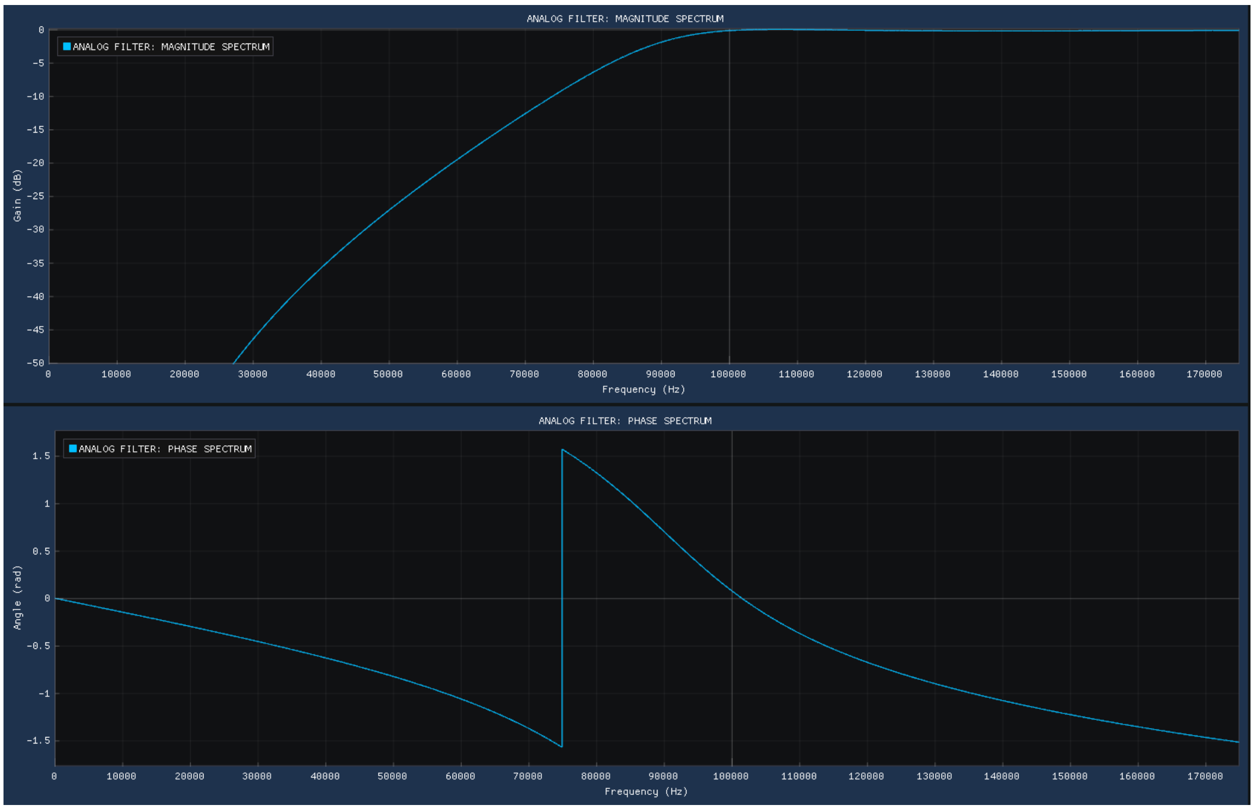


Figure 3-9: Analog High Pass Filter Response

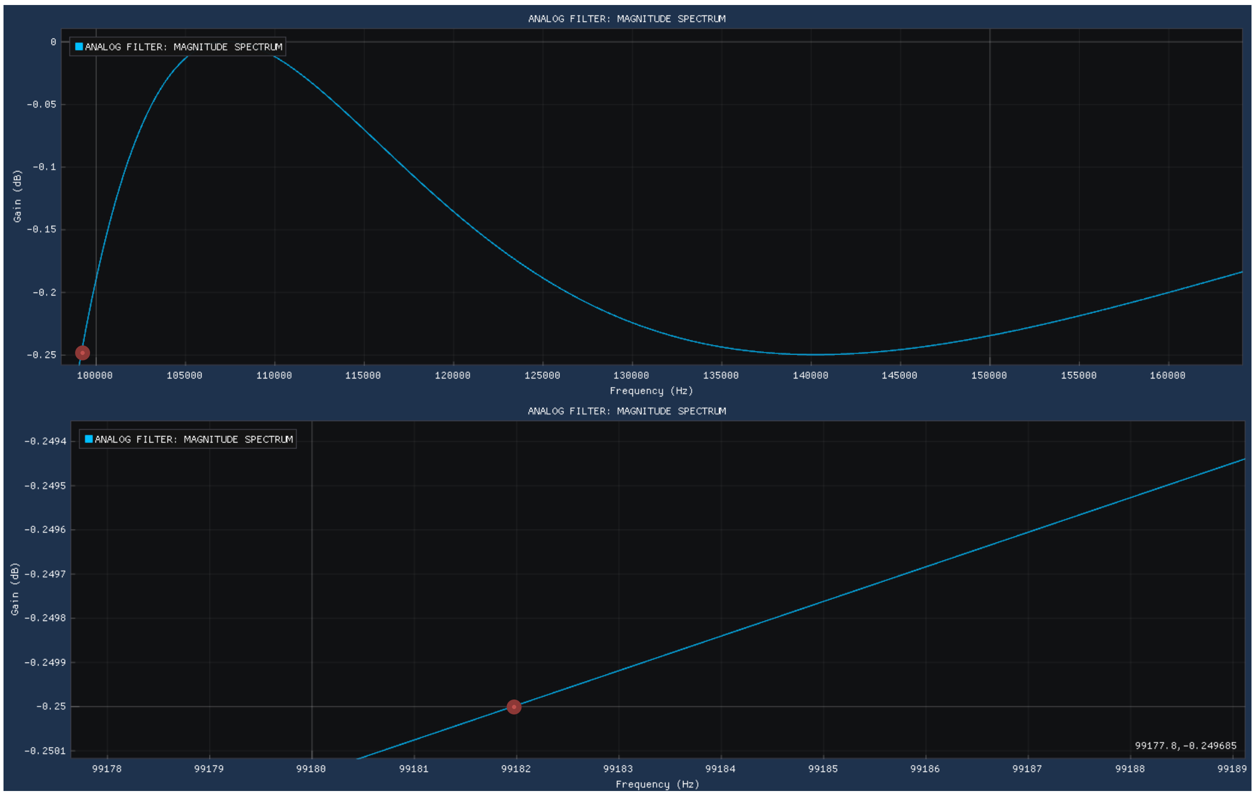


Figure 3-10: Analog Passband Edge Frequency

From the designed Analog High Pass filter, it is clear that the transformation to the digital domain, would deem successful, seen as it adheres to the set-out specifications.

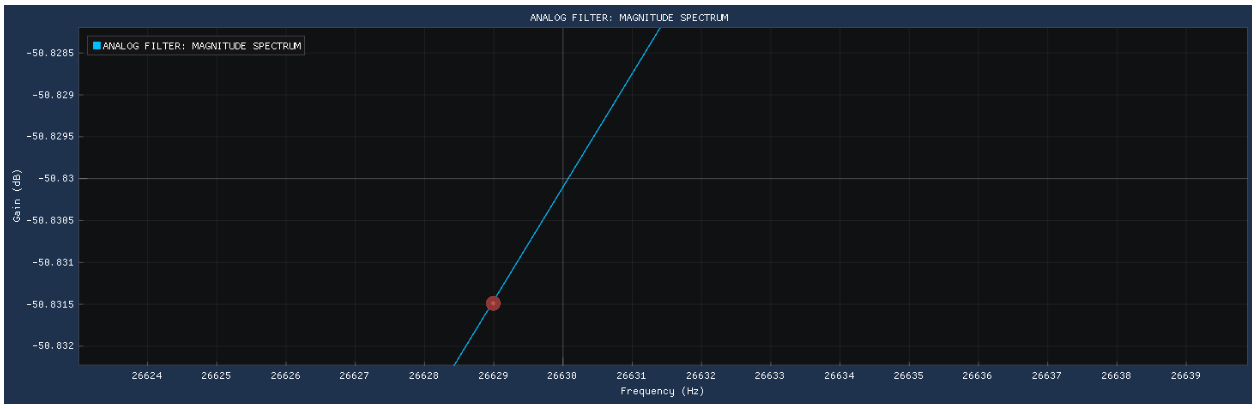


Figure 3-11: Analog Stopband Edge Frequency

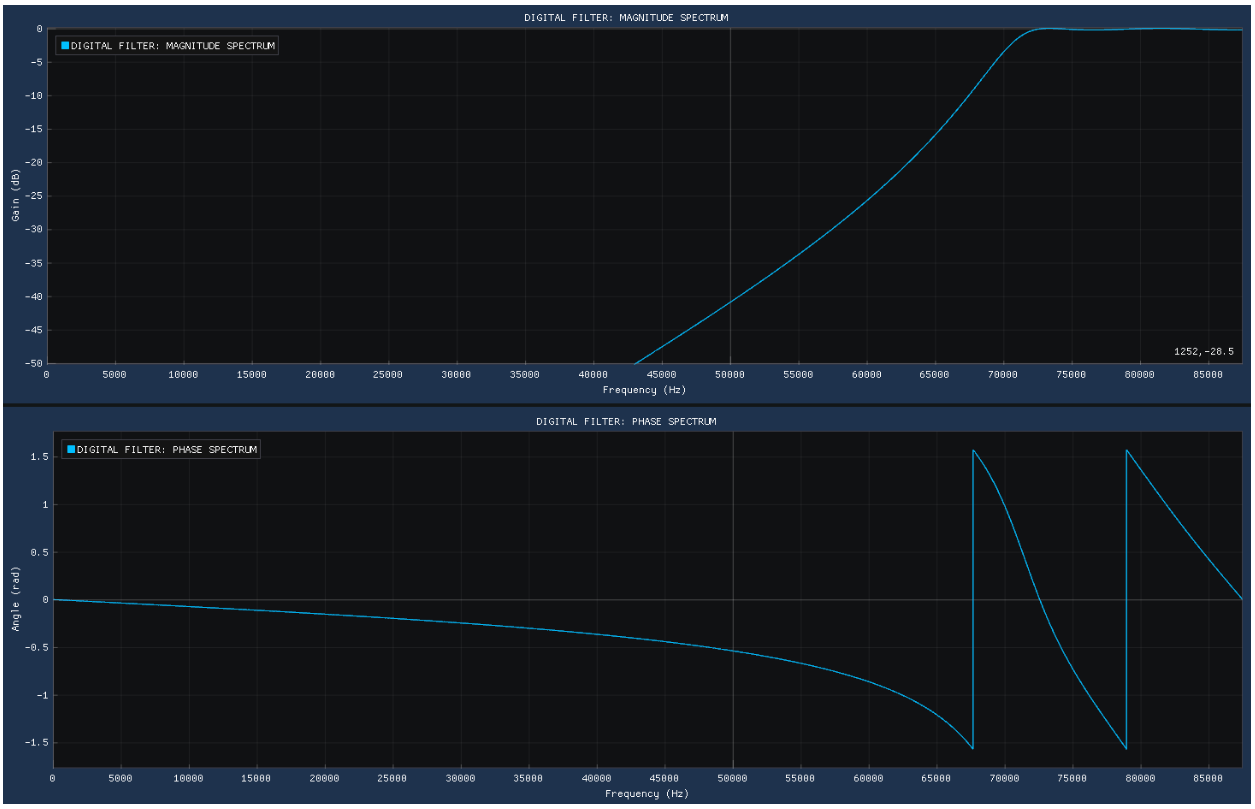


Figure 3-12: Digital High Pass Filter Response

3.4 Digital Filter Design

The bilinear transform is implemented programmatically to the magnitude and phase spectrum of the analog filter. This was done, as it revealed to be quite tricky to generate a transfer function form in the software. The algorithmic development of these functions is included in the setDigitalMagnitude() and setDigitalPhase() functions within the cFilterDesign class. The magnitude and phase spectrum of the digital filter is shown in Figure 3-12.

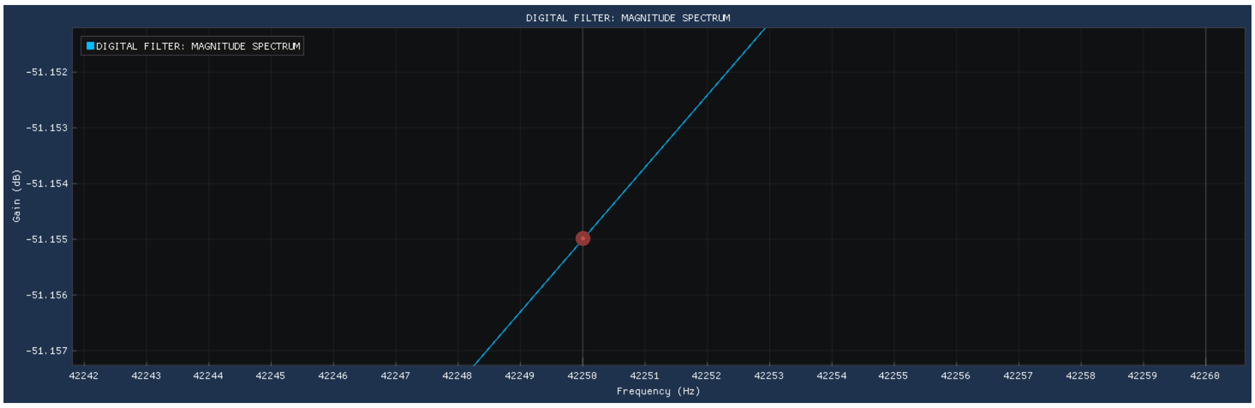


Figure 3-13: Digital Stopband Edge Frequency

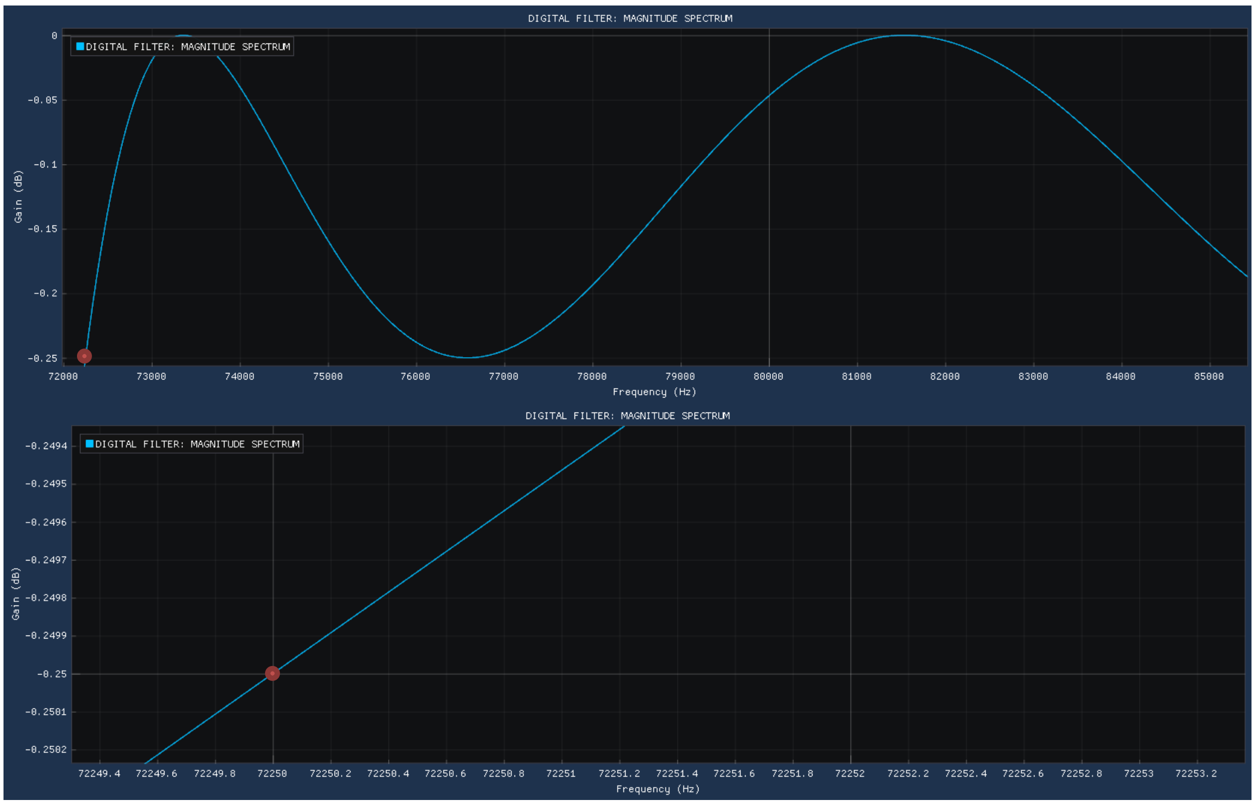


Figure 3-14: Digital Passband Edge Frequency

For the digital filter to function as required, it must adhere to the stopband and passband specifications as listed in section 1. At the specified passband edge frequency of 72,25kHz, the maximum passband ripple should occur. Figure 3-14 shows that this is in fact the case.

The attenuation at the stopband edge frequency of 42.25kHz, should minimally be -40dB. Figure 3-13 clearly shows this value to be much higher at -51.155dB. Adhering to this figure, as well as the unity gain, it is very clear that the designed fourth order Digital High Pass Type-1 Chebyshev filter would function as required.

3.5 Gray-Markel Realisation

Referring to the Gray-Markle Lattice configuration in Figure 2-1, the time-domain realization can be simplified by looking at various points in the lattice. More specifically eleven functions have been defined representing the realisation at a different point in the lattice. These eleven functions are defined below. Note that the integrator z^{-1} refers to a one sample time-delay in the time domain.

$$y_1[n] = x[n] - k_4 y_7[n-1]$$

$$y_2[n] = y_1[n] - k_3 y_6[n-1]$$

$$y_3[n] = y_2[n] - k_2 y_5[n-1]$$

$$y_4[n] = y_3[n] - k_1 y_4[n-1]$$

$$y_5[n] = k_1 y_4[n] + y_4[n-1]$$

$$y_6[n] = k_2 y_3[n] + y_5[n-1]$$

$$y_7[n] = k_3 y_2[n] + y_6[n-1]$$

$$y_8[n] = k_4 y_1[n] + y_5[n-1]$$

$$y_9[n] = \alpha_1 y_8[n] + \alpha_2 y_7[n-1]$$

$$y_{10}[n] = \alpha_3 y_6[n] + y_9[n]$$

$$y_{11}[n] = \alpha_4 y_5[n] + y_{10}[n]$$

The final filtered signal output $y[n]$, is given below.

$$y[n] = \alpha_5 y_4[n] + y_{11}[n] \tag{3-2}$$

Implementing this in the software package was done quite easily. For the generated signal described in section 3.2.3, the filter realization delivered promising results, as can be seen in the time-domain comparison of the filtered and unfiltered signals in Figure 3-15. The bottom filtered signal has a much lower peak voltage, seen as many frequency components have been removed.

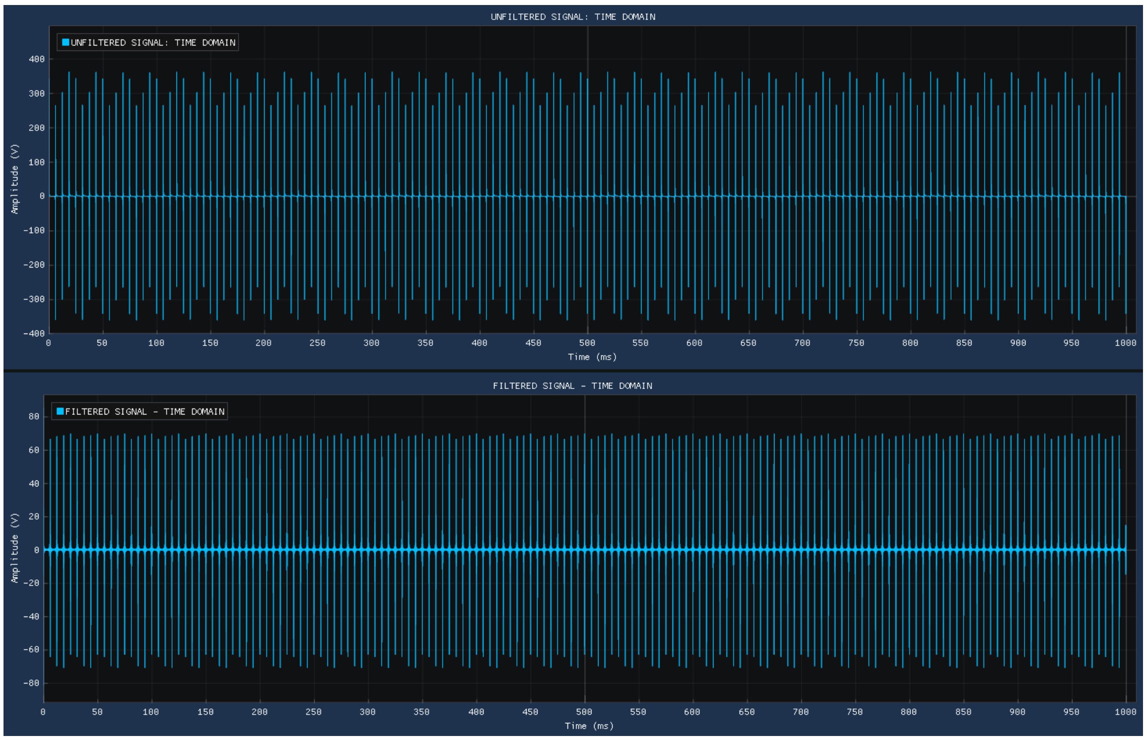


Figure 3-15: Filterer Comparison - Time Domain

The effectiveness of the filter can be clearly seen in the Magnitude and Phase spectrum comparison. The Magnitude Spectrum comparison is shown in Figure 3-16, while the Phase Spectrum comparison is shown in Figure 3-17. The Phase Spectrum of the Realized filter more accurately represents the phase response of the Digital filter, than it does the unfiltered signal.

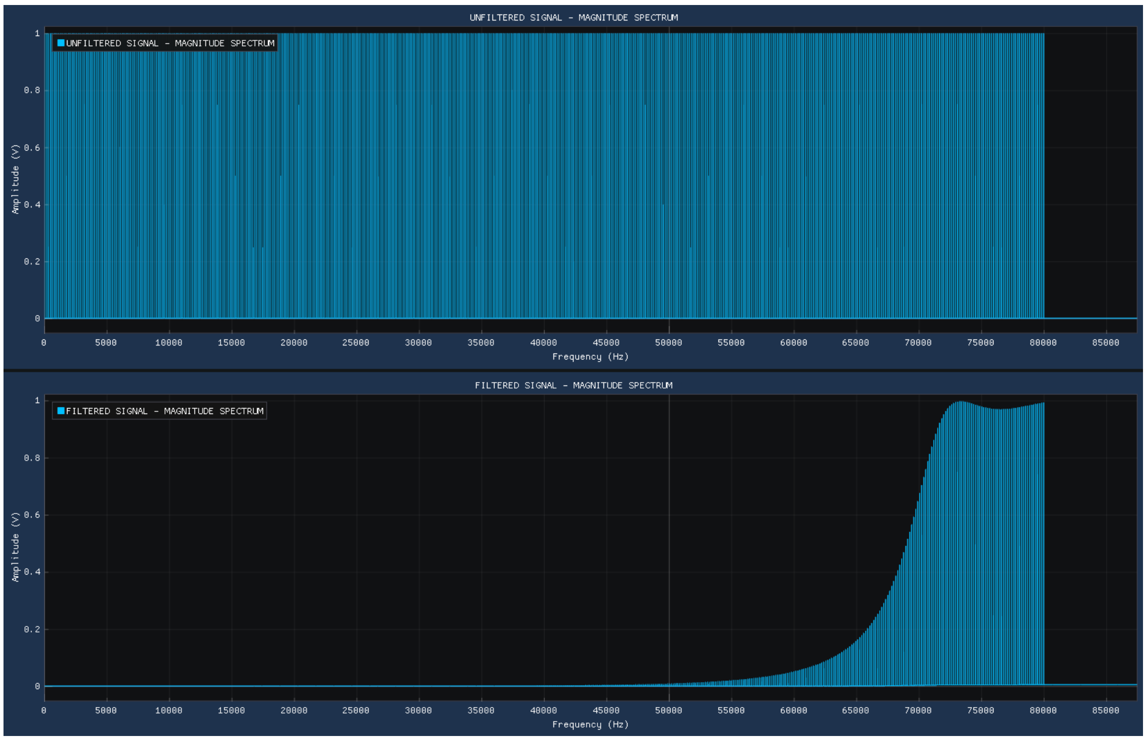


Figure 3-16: Filter Comparison - Magnitude Spectrum

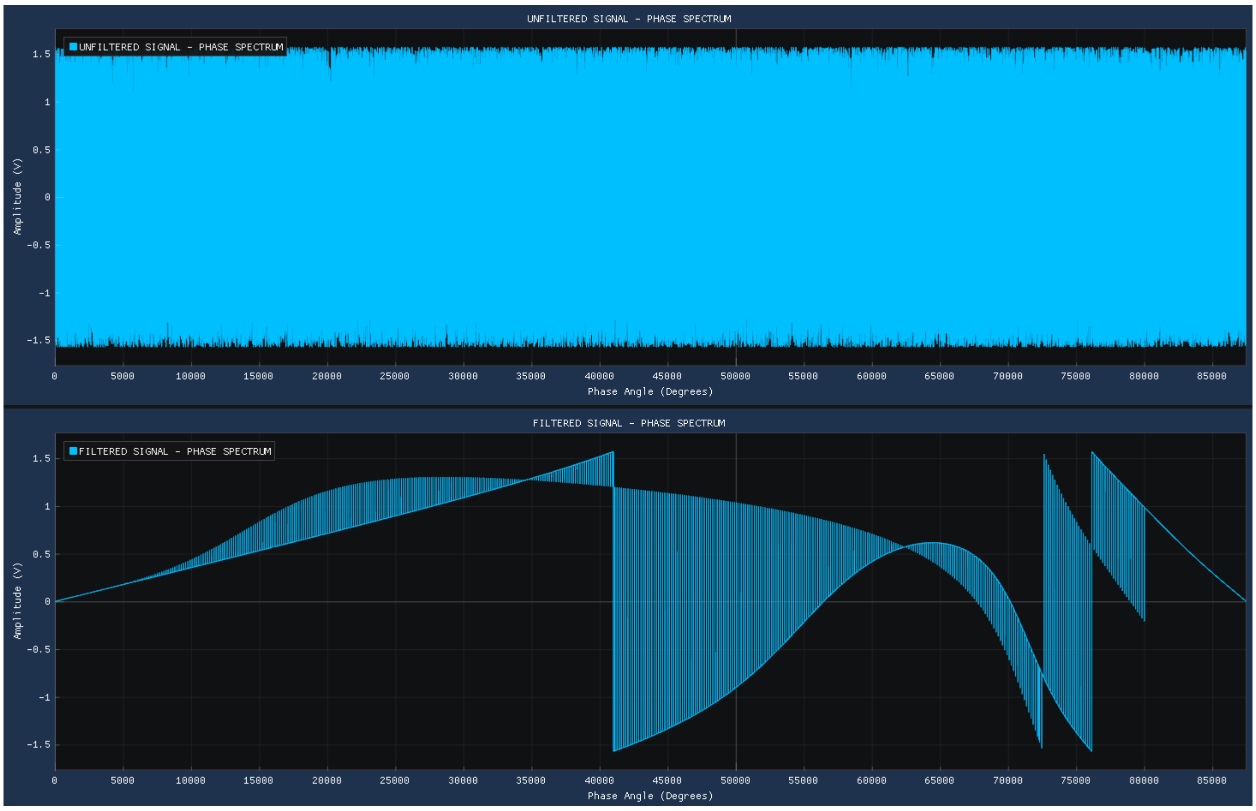


Figure 3-17: Filter Comparison - Phase Spectrum

Note the filtered signal mimics the behaviour of the digital filter in the magnitude spectrum. When looking at the desired passband edge frequency in Figure 3-18, a voltage value of 0.9685 V can be seen. This is the desired -0.25dB ripple point.

$$\alpha_{p,dB(realized)} = 20\log(0.9685) \tag{3-3}$$

$$\boldsymbol{\alpha_{p,dB(realized)} = -0.278\ dB}$$

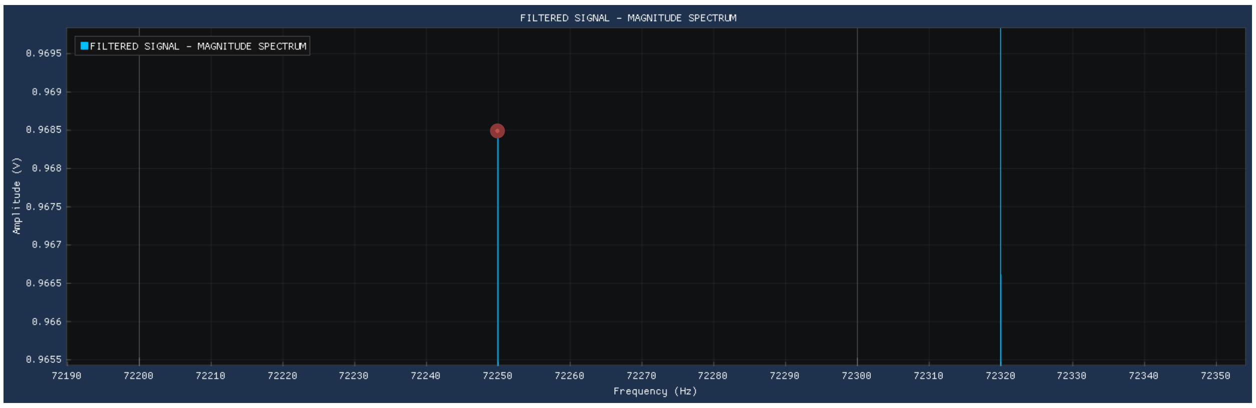


Figure 3-18: Realized Passband Edge Frequency

It is clear that some discrepancy exists, but this can purely be because of rounding errors made during the calculation of the Gray-Markel lattice parameters. However, the initial design specification required a maximum passband ripple of -0.3 dB, which the realized filter still adheres to. The same can be seen when comparing the realized stopband attenuation with the Analog design stopband attenuation. Figure 3-19 shows a voltage level of 0.0028V at the stopband edge frequency of 42.25kHz.

$$\alpha_{s,dB(realized)} = 20\log(0.0028) \tag{3-4}$$

$$\boldsymbol{\alpha_{s,dB(realized)} = -51.0568\ dB}$$

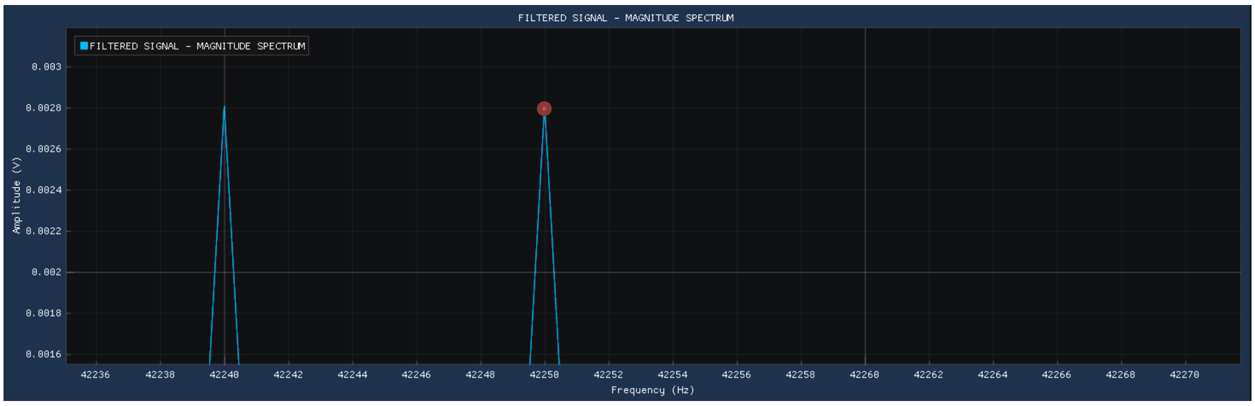


Figure 3-19: Realized Stopband Edge Frequency

CONCLUSION

This report briefly described the design process behind developing a Digital High Pass Chebyshev Type-1 filter to be realized using the Gray-Markel approach. The steps followed analytically was implemented well within a software package that could cleanly display graphical results. The realized filter yielded results that adhered to the set-out design specifications, however slight changes to the rounding of floating-point values within the software, could make the designed filter even more accurate.

A sine summation input signal was filtered using a digital filter with a desired stopband edge frequency of 42.25kHz and passband edge frequency of 72.25khz. The desired passband ripple should be less than 0.3dB, while a stopband attenuation of more than 40dB would suffice. Referring to the maximum passband ripple, a filter with desired ripple of 0.25dB was chosen, as any discrepancies would still adhere to the 0.3dB threshold.

The final designed filter performed extremely well when realized. When used on the sine summation signal, a maximum passband ripple of 0.278dB was found, which is a bit more than what was designed for, however the reasoning behind choosing 0.25dB as the designed parameter sufficed, as it still adheres to the threshold value. The minimum stopband attenuation of 51.056 dB is also well within specification.

The choice of filter all depends on what the filter will be used for. A Chebyshev Type-1 filter attenuates frequencies in the stopband without any ripple, while having a non-linear ripple in the passband. If a more linear passband region were desired, then a Type-2 filter would have sufficed.

REFERENCES

[1] P. Wilson, “Chapter 9 - Digital Filters,” in *Design Recipes for FPGAs*, Newnes, 2016, pp. 117-134.

[2] S. Bakshi, I. Javid, M. Sahni and S. Naz, “DESIGNAND COMPARISON BETWEEN IIR BUTTERWOTH AND CHEBYSHEV DIGITAL FILTERS USING MATLAB,” in *International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, Greater Noida, 2019.

[3] N. Agrawal, A. Kumar and G. Singh, “Design of Bandpass and Bandstop Infinite Impulse Response,” *IEEE transactions on,* 2018.

[4] K. Kumar, B. Yazdanpanah and P. Kumar, ““Removal of Noise from Electrocardiogram Using Digital FIR,” *IEEE ICCSP,* 2018.

[5] V. Musoko, “BIOMEDICAL SIGNAL AND IMAGE PROCESSING,” Institute of Chemical Technology, Prague, Department of Computing and Control Engineering, Prague, 2005.

[6] S. K. Mitra, “Chapter 9: IIR Digital Filter Design,” in *Digital Signal Processing: A Computer-Based Approach*, 4th ed., New York, McGraw-Hill, 2011, pp. 489-526.

APPENDIX A - SOFTWARE SOURCE CODE

ChebyshevFilterGenerator.cpp

```
// ---------------------------------------------------------------------------
// NAME & SURNAME       : RJ VAN STADEN
// STUDENTNUMBER        : 30026792
// ---------------------------------------------------------------------------
// DATE                 : 2021/05/05
// REVISION             : rev01
// ---------------------------------------------------------------------------
// PROJECT NAME         : ChebyshevFilterGenerator
// ---------------------------------------------------------------------------

#define _USE_MATH_DEFINES

// ---------------------------------------------------------------------------
// SYSTEM INCLUDE FILES
#include <algorithm>
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <sstream>
#include <iterator>
#include <tchar.h>
#include <cmath>
#include <windows.h>
#include <string>
#include <sstream>

// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES
#include <GL/gl3w.h>
#include <GLFW/glfw3.h>

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES
#include "imgui.h"
#include "implot.h"
#include "imgui_impl_glfw.h"
#include "imgui_impl_opengl3.h"
#include "cSignalGenerator.h"
#include "cFastFourierTransform.h"
#include "cFilterDesign.h"

static void glfw_error_callback(int error, const char* description) {
    fprintf(stderr, "GLFW Error: %d: %s\n", error, description);
}

// GLOBAL FUNCTION DECLERATIONS
//void test_PlotData(std::vector<double>& vsPlotData_top, std::vector<double>& vsPlotData_bottom,
ImGuiCond& plotCondition);
void plotTopGraph(std::vector<double> &vsPlotY_top, std::vector<double> &vsPlotX_top, std::string
strTitle, std::string strX, std::string strY, ImGuiCond& plotCondition);
void plotBottomGraph(std::vector<double> &vsPlotY_bottom, std::vector<double> &vsPlotX_bottom, std::string
strTitle, std::string strX, std::string strY, ImGuiCond& plotCondition);
static void HelpMarker(const char* desc, bool same_line = true);
std::vector<double> plotFFT(bool bComponent, std::vector<double> &vfTime);

// GLOBAL VARIABLE DECLERATIONS
std::shared_ptr<cSignalGenerator> pLocalSignal = std::make_shared<cSignalGenerator>();
std::shared_ptr<cFastFourierTransform> pLocalFFT = std::make_shared<cFastFourierTransform>();

ImVec4 foreground_color = ImVec4(0.258, 0.529, 0.561, 1);

int main(int argc, char* argv[])
{
    // ---------- GLFW ERROR STATE ----------
    glfwSetErrorCallback(glfw_error_callback);
    if (!glfwInit())
        return 1;
```

```cpp
// ---------- GLSL FOR WINDOWS ----------
const char* glsl_version = "#version 130";
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
GLFWwindow* window = glfwCreateWindow(1920, 1080, "Chebyshev Filter Generator", NULL, NULL);
if (window == NULL)
    return 1;
glfwMakeContextCurrent(window);
glfwSwapInterval(1);
gl3wInit();

// ---------- SETUP IMGUI ----------
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImPlot::CreateContext();
ImGuiIO &io = ImGui::GetIO();
(void)io;

ImGui::StyleColorsDark();

ImGui_ImplGlfw_InitForOpenGL(window, true);
ImGui_ImplOpenGL3_Init(glsl_version);

std::vector<double> vfTime_Y;
std::vector<double> vfTime_X;
std::vector<double> vfFreq_Y;
std::vector<double> vfFreq_X;

std::vector<double> vfTime_Original;
std::vector<double> vfTime_Realisation;
std::vector<double> vfMag_Original;
std::vector<double> vfMag_Realisation;
std::vector<double> vfPhase_Original;
std::vector<double> vfPhase_Realisation;

bool bMag = true;

// RESET INPUT VARIABLES
int iSampleRate_Hz = 175000;
int iSmallestFreq_Hz = 10;
int iLargestFreq_Hz = 80000;
int iSignalLength_ms = 1000;

int iOmegaPass_Hz = 42500;
int iOmegaStop_Hz = 72250;
double iRipplePass = -0.3;
double iRippleStop = -40;

std::string strTop_Title;
std::string strTop_X;
std::string strTop_Y;
std::string strBottom_Title;
std::string strBottom_X;
std::string strBottom_Y;

int iSweepVariable = 2;

// ---------- MAIN PROGRAM LOOP ----------
while (!glfwWindowShouldClose(window))
{
    // Poll Events one-by-one
    glfwPollEvents();

    // Start the new Frame
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();

    // =======================
    // Enter Program Code Here
    // =======================
        // DISPLAY INPUT PANEL
        ImGui::SetNextWindowSize(ImVec2(380, 977));
        ImGui::SetNextWindowPos(ImVec2(20, 20));
        ImGui::Begin("FILTER INPUT PARAMETERS");
        ImGuiWindowFlags window_flags = 0;
        window_flags |= ImGuiWindowFlags_NoTitleBar;

        ImGuiCond cond_Plot = ImGuiCond_Once;

        ImGui::Text("====================================================");
        ImGui::Text("              INPUT SIGNAL GENERATION               ");
        ImGui::Text("====================================================");

        // INPUT SAMPLE RATE
        ImGui::Text("Sample Rate");
        HelpMarker("Set the sample rate in hertz (Hz)");
        ImGui::InputInt("Hz##SampleRate", &iSampleRate_Hz, 1, 10);
        ImGui::Text("\n");

        // INPUT SMALLEST FREQUENCY
        ImGui::Text("Smallest Frequency");
        HelpMarker("The smallest frequency in the generated signal (Hz)");
        ImGui::InputInt("Hz##SmallestFreq", &iSmallestFreq_Hz, 1, 10);
        ImGui::Text("\n");

        // INPUT LARGEST FREQUENCY
        ImGui::Text("Largest Frequency");
        HelpMarker("The largest frequency in the generated signal (Hz)");
        ImGui::InputInt("Hz##LargestFreq", &iLargestFreq_Hz, 1, 10);
        ImGui::Text("\n");

        // IF BUTTON PRESSED CALCULATE DATA AND THEN DISPLAY
        std::vector<double> vfTop_Y;
        std::vector<double> vfTop_X;
        std::vector<double> vfBottom_Y;
        std::vector<double> vfBottom_X;

        std::shared_ptr<cSignalGenerator> pLocalSignal =
std::make_shared<cSignalGenerator>(iSampleRate_Hz, iSignalLength_ms, iSmallestFreq_Hz, iLargestFreq_Hz);
        if (ImGui::Button("LINEAR SWEEP", ImVec2(118, 25))) {
            // PLOT DATA BASED ON INPUT DATA
            pLocalSignal->generateSignal(0);
            iSweepVariable = 0;

            vfTime_Y = pLocalSignal->getSignal_Time();
            vfTime_X.clear();
            for (int i = 0; i < vfTime_Y.size(); i++) {
                vfTime_X.push_back(((double)i / (double) iSampleRate_Hz) * (double)iSignalLength_ms);
            }

            strTop_Title = "LINEAR SWEEP: TIME DOMAIN";
            strTop_X = "Time (ms)";
            strTop_Y = "Amplitude (V)";

            cond_Plot = ImGuiCond_Always;
        }
        ImGui::SetCursorPosY(ImGui::GetCursorPosY() - 29);
        ImGui::SetCursorPosX(ImGui::GetCursorPosX() + 122);
        if (ImGui::Button("LOG SWEEP", ImVec2(118, 25))) {
            // PLOT DATA BASED ON INPUT DATA
            pLocalSignal->generateSignal(1);
            iSweepVariable = 1;

            vfTime_Y = pLocalSignal->getSignal_Time();
            vfTime_X.clear();
            for (int i = 0; i < vfTime_Y.size(); i++) {
                vfTime_X.push_back(((double)i / (double)iSampleRate_Hz) * (double)iSignalLength_ms);
            }

            strTop_Title = "LOG SWEEP: TIME DOMAIN";
            strTop_X = "Time (ms)";
            strTop_Y = "Amplitude (V)";
```

```
    cond_Plot = ImGuiCond_Always;

}
ImGui::SetCursorPosY(ImGui::GetCursorPosY() - 29);
ImGui::SetCursorPosX(ImGui::GetCursorPosX() + 122 + 122);
if (ImGui::Button("SINE SUM", ImVec2(118, 25))) {
    // PLOT DATA BASED ON INPUT DATA
    pLocalSignal->generateSignal(2);
    iSweepVariable = 2;

    vfTime_Y = pLocalSignal->getSignal_Time();
    vfTime_X.clear();
    for (int i = 0; i < vfTime_Y.size(); i++) {
        vfTime_X.push_back(((double)i / (double)iSampleRate_Hz) * (double)iSignalLength_ms);
    }

    strTop_Title = "TIME SUMMATION: TIME DOMAIN";
    strTop_X = "Time (ms)";
    strTop_Y = "Amplitude (V)";

    cond_Plot = ImGuiCond_Always;
}

ImGui::Text("\n");
ImGui::Text("====================================================");
ImGui::Text("            DISPLAY FREQUENCY COMPONENTS            ");
ImGui::Text("====================================================");

if (ImGui::Button("MAGNITUDE", ImVec2(179, 25))) {
    // PLOT DATA BASED ON INPUT DATA
    vfFreq_Y = plotFFT(true, vfTime_Y);
    vfFreq_X.clear();
    for (int i = 0; i < vfFreq_Y.size(); i++) {
        vfFreq_X.push_back(i);
    }

    strBottom_Title = "FFT - MAGNITUDE SPECTRUM";
    strBottom_X = "Frequency (Hz)";
    strBottom_Y = "Amplitude (V)";


     cond_Plot = ImGuiCond_Always;


}
ImGui::SetCursorPosY(ImGui::GetCursorPosY() - 29);
ImGui::SetCursorPosX(ImGui::GetCursorPosX() + 183);
if (ImGui::Button("PHASE", ImVec2(179, 25))) {
    // PLOT DATA BASED ON INPUT DATA
    vfFreq_Y = plotFFT(false, vfTime_Y);
    vfFreq_X.clear();
    for (int i = 0; i < vfFreq_Y.size(); i++) {
        vfFreq_X.push_back(i);
    }

    strBottom_Title = "FFT - PHASE SPECTRUM";
    strBottom_X = "Frequency (Hz)";
    strBottom_Y = "Angle (rad)";

    cond_Plot = ImGuiCond_Always;
}


ImGui::Text("\n");
ImGui::Text("====================================================");
ImGui::Text("------------------- FILTER DESIGN ------------------");
ImGui::Text("====================================================");

// INPUT PASSBAND EDGE FREQUENCY
ImGui::Text("Passband Edge Frequency");
HelpMarker("Set the Passband Edge Frequency (Hz)");
ImGui::InputInt("Hz##PassbandEdge", &iOmegaStop_Hz, 1, 10);
ImGui::Text("\n");
```

```
        // INPUT STOPBAND EDGE FREQUENCY
        ImGui::Text("Stopband Edge Frequency");
        HelpMarker("Set the Stopband Edge Frequency (Hz)");
        ImGui::InputInt("Hz##StopbandEdge", &iOmegaPass_Hz, 1, 10);
        ImGui::Text("\n");

        // INPUT PASSBAND RIPPLE
        ImGui::Text("Passband Ripple");
        HelpMarker("Set the Passband Ripple (dB)");
        ImGui::InputDouble("dB##PassbandRipple", &iRipplePass, 1, 10);
        ImGui::Text("\n");

        // INPUT STOPBAND RIPPLE
        ImGui::Text("Stopband Ripple");
        HelpMarker("Set the Stopband Ripple (dB)");
        ImGui::InputDouble("dB##StopbandRipple", &iRippleStop, 1, 10);
        ImGui::Text("\n");

        std::shared_ptr<cFilterDesign> pLocalFilter = std::make_shared<cFilterDesign>(iOmegaPass_Hz,
iOmegaStop_Hz, iRipplePass, iRippleStop, iSampleRate_Hz);
        if (ImGui::Button("ANALOG", ImVec2(179, 25))) {
            pLocalFilter->setAnalogFilterTF();
            vfTime_Y = pLocalFilter->getYAxis(true, true);
            vfFreq_Y = pLocalFilter->getYAxis(true, false);
            vfTime_X.clear();
            vfFreq_X.clear();
            for (int i = 0; i < vfFreq_Y.size(); i++) {
                vfTime_X.push_back(i);
                vfFreq_X.push_back(i);
            }

            strTop_Title = "ANALOG FILTER: MAGNITUDE SPECTRUM";
            strTop_X = "Frequency (Hz)";
            strTop_Y = "Gain (dB)";
            strBottom_Title = "ANALOG FILTER: PHASE SPECTRUM";
            strBottom_X = "Frequency (Hz)";
            strBottom_Y = "Angle (rad)";

            cond_Plot = ImGuiCond_Always;
        }
        ImGui::SetCursorPosY(ImGui::GetCursorPosY() - 29);
        ImGui::SetCursorPosX(ImGui::GetCursorPosX() + 183);
        if (ImGui::Button("DIGITAL", ImVec2(179, 25))) {
            pLocalFilter->setAnalogFilterTF();
            vfTime_Y = pLocalFilter->getYAxis(false, true);
            vfFreq_Y = pLocalFilter->getYAxis(false, false);
            vfTime_X.clear();
            vfFreq_X.clear();
            for (int i = 0; i < vfFreq_Y.size(); i++) {
                vfTime_X.push_back(i);
                vfFreq_X.push_back(i);
            }

            strTop_Title = "DIGITAL FILTER: MAGNITUDE SPECTRUM";
            strTop_X = "Frequency (Hz)";
            strTop_Y = "Gain (dB)";
            strBottom_Title = "DIGITAL FILTER: PHASE SPECTRUM";
            strBottom_X = "Frequency (Hz)";
            strBottom_Y = "Angle (rad)";

            cond_Plot = ImGuiCond_Always;
        }

        ImGui::Text("\n");
        ImGui::Text("====================================================");
        ImGui::Text("-------------- REALISE DIGITAL FILTER --------------");
        ImGui::Text("====================================================");

        if (ImGui::Button("REALISE FILTER", ImVec2(362, 25))) {
            // INITIALIZE THE TRANSFER FUNCTION OF H(Z)
            std::vector<double> num = { 0.00184647, -0.007385884, 0.01107883, -0.0073858844, 0.0018464711
};
```

```
std::vector<double> den = { 1, 2.990801, 3.62682596, 2.07580297, 0.470359711 };
//std::vector<double> num = { 0, 0.44, 0.36, 0.02 };
//std::vector<double> den = { 1, 0.4, 0.18, -0.2 };
pLocalFilter->setGrayMarkel(num, den);

// GET LATTICE AND FEED_FORWARD GAINS
std::vector<double> vfFeedForward_a = pLocalFilter->getFeedForward();
std::vector<double> vfLattice_k = pLocalFilter->getLattice();
std::cout << "Lattice: ";
for (int i = 0; i < vfLattice_k.size(); i++) {
    std::cout << vfLattice_k.at(i) << "  ";
}
std::cout << std::endl;
std::cout << "Feed-Forward: ";
for (int i = 0; i < vfFeedForward_a.size(); i++) {
    std::cout << vfFeedForward_a.at(i) << "  ";
}
std::cout << std::endl;

// GET SIGNAL IN TIME DOMAIN
pLocalSignal->generateSignal(iSweepVariable);
std::vector<double> vfX = pLocalSignal->getSignal_Time();
std::cout << vfX.size() << " " << iSweepVariable << std::endl;

// CREATE CASCADE PARAMETERS
std::vector<double> vfY_1;
std::vector<double> vfY_2;
std::vector<double> vfY_3;
std::vector<double> vfY_4;
std::vector<double> vfY_5;
std::vector<double> vfY_6;
std::vector<double> vfY_7;
std::vector<double> vfY_8;
std::vector<double> vfY_9;
std::vector<double> vfY_10;
std::vector<double> vfY_11;
std::vector<double> vfY_FINAL;

// CREATE VECTORS FILLED WITH ZEROS
for (int i = 0; i < vfX.size(); i++) {
    vfY_1.push_back(0);
    vfY_2.push_back(0);
    vfY_3.push_back(0);
    vfY_4.push_back(0);
    vfY_5.push_back(0);
    vfY_6.push_back(0);
    vfY_7.push_back(0);
    vfY_8.push_back(0);
    vfY_9.push_back(0);
    vfY_10.push_back(0);
    vfY_11.push_back(0);
    vfY_FINAL.push_back(0);
}

for (int i = 1; i < 2; i++) {
    for (int n = 0; n < vfX.size(); n++) {
        vfY_1[n] = (vfX[n] - vfLattice_k[3] * vfY_7[n-1]);

        vfY_2[n] = (vfY_1[n] - vfLattice_k[2] * vfY_6[n-1]);

        vfY_3[n] = (vfY_2[n] - vfLattice_k[1] * vfY_5[n - 1]);

        vfY_4[n] = (vfY_3[n] - vfLattice_k[0] * vfY_4[n - 1]);

        vfY_5[n] = (vfLattice_k[0] * vfY_4[n] + vfY_4[n - 1]);

        vfY_6[n] = (vfLattice_k[1] * vfY_3[n] + vfY_5[n - 1]);

        vfY_7[n] = (vfLattice_k[2] * vfY_2[n] + vfY_6[n - 1]);

        vfY_8[n] = (vfLattice_k[3] * vfY_1[n] + vfY_7[n - 1]);

        vfY_9[n] = (vfFeedForward_a[0] * vfY_8[n] + vfFeedForward_a[1] * vfY_7[n]);
```

```
                vfY_10[n] = (vfFeedForward_a[2] * vfY_6[n] + vfY_9[n]);

                vfY_11[n] = (vfFeedForward_a[3] * vfY_5[n] + vfY_10[n]);

                vfY_FINAL[n] = (vfFeedForward_a[4] * vfY_4[n] + vfY_11[n]);
            }
        }

        vfTime_Realisation = vfY_FINAL;
        vfTime_Original = vfX;

        vfTime_Y = vfTime_Original;
        vfTime_X.clear();
        for (int i = 0; i < vfTime_Y.size(); i++) {
            vfTime_X.push_back(((double)i / (double)iSampleRate_Hz) * (double)iSignalLength_ms);
        }

        strTop_Title = "UNFILTERED SIGNAL: TIME DOMAIN";
        strTop_X = "Time (ms)";
        strTop_Y = "Amplitude (V)";

        vfFreq_Y = vfTime_Realisation;
        vfFreq_X.clear();
        for (int i = 0; i < vfFreq_Y.size(); i++) {
            vfFreq_X.push_back(((double)i / (double)iSampleRate_Hz) * (double)iSignalLength_ms);
        }

        strBottom_Title = "FILTERED SIGNAL - TIME DOMAIN";
        strBottom_X = "Time (ms)";
        strBottom_Y = "Amplitude (V)";

        cond_Plot = ImGuiCond_Always;
    }
    if (ImGui::Button("MAGNITUDE##Mag_Realisation", ImVec2(179, 25))) {
        // PLOT DATA BASED ON INPUT DATA

        vfTime_Y = plotFFT(true, vfTime_Original);
        vfTime_X.clear();
        for (int i = 0; i < vfTime_Y.size(); i++) {
            vfTime_X.push_back(i);
        }

        strTop_Title = "UNFILTERED SIGNAL - MAGNITUDE SPECTRUM";
        strTop_X = "Frequency (Hz)";
        strTop_Y = "Amplitude (V)";

        vfFreq_Y = plotFFT(true, vfTime_Realisation);
        vfFreq_X.clear();
        for (int i = 0; i < vfFreq_Y.size(); i++) {
            vfFreq_X.push_back(i);
        }

        strBottom_Title = "FILTERED SIGNAL - MAGNITUDE SPECTRUM";
        strBottom_X = "Frequency (Hz)";
        strBottom_Y = "Amplitude (V)";

        cond_Plot = ImGuiCond_Always;
    }
    ImGui::SetCursorPosY(ImGui::GetCursorPosY() - 29);
    ImGui::SetCursorPosX(ImGui::GetCursorPosX() + 183);
    if (ImGui::Button("PHASE##Phase_Realisation", ImVec2(179, 25))) {
        // PLOT DATA BASED ON INPUT DATA
        vfTime_Y = plotFFT(false, vfTime_Original);
        vfTime_X.clear();
        for (int i = 0; i < vfTime_Y.size(); i++) {
            vfTime_X.push_back(i);
        }

        strTop_Title = "UNFILTERED SIGNAL - PHASE SPECTRUM";
        strTop_X = "Phase Angle (Degrees)";
        strTop_Y = "Amplitude (V)";
```

```
             vfFreq_Y = plotFFT(false, vfTime_Realisation);
            vfFreq_X.clear();
            for (int i = 0; i < vfFreq_Y.size(); i++) {
                vfFreq_X.push_back(i);
            }

            strBottom_Title = "FILTERED SIGNAL - PHASE SPECTRUM";
            strBottom_X = "Phase Angle (Degrees)";
            strBottom_Y = "Amplitude (V)";

            cond_Plot = ImGuiCond_Always;
        }

        vfTop_Y = vfTime_Y;
        vfTop_X = vfTime_X;
        vfBottom_Y = vfFreq_Y;
        vfBottom_X = vfFreq_X;

        ImGui::SetNextWindowPos(ImVec2(420, 20));
        ImGui::SetNextWindowSize(ImVec2(1480, 977));
        ImGui::Begin("OUTPUT WINDOW");

        plotTopGraph(vfTop_Y, vfTop_X, strTop_Title, strTop_X, strTop_Y, cond_Plot);
        plotBottomGraph(vfBottom_Y, vfBottom_X, strBottom_Title, strBottom_X, strBottom_Y, cond_Plot);

        ImGui::End();

        vfTop_Y.clear();
        vfTop_X.clear();
        vfBottom_Y.clear();
        vfBottom_X.clear();

        ImGui::End();

        // Final GUI Rendering
        ImGui::Render();
        int display_w, display_h;
        glfwGetFramebufferSize(window, &display_w, &display_h);
        glViewport(0, 0, display_w, display_h);
        glClearColor(foreground_color.x, foreground_color.y, foreground_color.z, foreground_color.w);
        glClear(GL_COLOR_BUFFER_BIT);
        ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

        glfwSwapBuffers(window);

    }

    // ---------- IMGUI & OPENGL CLEANUP ----------
    ImGui_ImplOpenGL3_Shutdown();
    ImGui_ImplGlfw_Shutdown();
    ImGui::DestroyContext();
    ImPlot::DestroyContext();

    glfwDestroyWindow(window);
    glfwTerminate();

    return 0;
}

// Plot Function to plot the Top Graph
void plotTopGraph(std::vector<double>& vsPlotY_top, std::vector<double>& vsPlotX_top, std::string
strTitle, std::string strX, std::string strY, ImGuiCond& plotCondition)
{
    ImGuiCond condPlot = plotCondition;

    // PLOT FOR TOP GRAPH
    if (vsPlotY_top.size() != 0) {
        // START PLOT INSTANCE
        //ImGui::SetCursorPosY(50);
        double max = -999999;
        double min = 999999;
        for (int i = 1; i < vsPlotX_top.size(); i++) {
            if (vsPlotX_top.at(i) > max)
```

```
                max = vsPlotX_top.at(i);
        }
        ImPlot::SetNextPlotLimitsX(0, max + 10, plotCondition);

        max = -999999;
        for (int i = 1; i < vsPlotY_top.size(); i++) {
            if (vsPlotY_top.at(i) > max)
                max = vsPlotY_top.at(i);
            if (vsPlotY_top.at(i) < min && vsPlotY_top.at(i) > -50)
                min = vsPlotY_top.at(i);
        }
        ImPlot::SetNextPlotLimitsY(min - 0.2, max + 0.2, plotCondition);
        ImPlot::BeginPlot(strTitle.c_str(), strX.c_str(), strY.c_str(), ImVec2(ImGui::GetWindowWidth() -
18, ImGui::GetWindowHeight() / 2 - 20));

        double* dat_y = new double[vsPlotY_top.size()];
        double* dat_x = new double[vsPlotY_top.size()];
        for (int i = 0; i < vsPlotY_top.size(); i++) {
            dat_y[i] = vsPlotY_top[i];
            dat_x[i] = vsPlotX_top[i];

        }
        ImPlot::PlotLine(strTitle.c_str(), dat_x, dat_y, vsPlotY_top.size());
        ImPlot::EndPlot();

        delete dat_y;
        delete dat_x;
    }
}

// Plot Function to plot the Bottom Graph
void plotBottomGraph(std::vector<double>& vsPlotY_bottom, std::vector<double>& vsPlotX_bottom, std::string
strTitle, std::string strX, std::string strY, ImGuiCond& plotCondition)
{
    ImGuiCond condPlot = plotCondition;

    // PLOT FOR TOP GRAPH
    if (vsPlotY_bottom.size() != 0) {
        // START PLOT INSTANCE
        //ImGui::SetCursorPosY(50);
        double max = -999999;
        double min = 999999;
        for (int i = 1; i < vsPlotX_bottom.size(); i++) {
            if (vsPlotX_bottom.at(i) > max)
                max = vsPlotX_bottom.at(i);
        }
        ImPlot::SetNextPlotLimitsX(0, max + 10, plotCondition);

        max = -999999;
        for (int i = 1; i < vsPlotY_bottom.size(); i++) {
            if (vsPlotY_bottom.at(i) > max)
                max = vsPlotY_bottom.at(i);
            if (vsPlotY_bottom.at(i) < min && vsPlotY_bottom.at(i) > -50)
                min = vsPlotY_bottom.at(i);
        }
        ImPlot::SetNextPlotLimitsY(min - 0.2, max + 0.2, plotCondition);
        ImPlot::BeginPlot(strTitle.c_str(), strX.c_str(), strY.c_str(), ImVec2(ImGui::GetWindowWidth() -
18, ImGui::GetWindowHeight() / 2 - 20));

        double* dat_y = new double[vsPlotY_bottom.size()];
        double* dat_x = new double[vsPlotY_bottom.size()];
        for (int i = 0; i < vsPlotY_bottom.size(); i++) {
            dat_y[i] = vsPlotY_bottom[i];
            dat_x[i] = vsPlotX_bottom[i];

        }
        ImPlot::PlotLine(strTitle.c_str(), dat_x, dat_y, vsPlotY_bottom.size());
        ImPlot::EndPlot();

        delete dat_y;
        delete dat_x;
    }
}
```

```
void test_PlotData(std::vector<double> &vsPlotData_top, std::vector<double> &vsPlotData_bottom, ImGuiCond
&plotCondition)
{
    ImGuiCond condPlot = plotCondition;
    ImGui::SetNextWindowPos(ImVec2(420, 20));
    ImGui::SetNextWindowSize(ImVec2(1480, 977));
    ImGui::Begin("##TOP GRAPH");

    // PLOT FOR TOP GRAPH
    if (vsPlotData_top.size() != 0) {
        // START PLOT INSTANCE
        //ImGui::SetCursorPosY(50);
        ImPlot::SetNextPlotLimitsX(0, vsPlotData_top.size() + 10, plotCondition);
        double min = 999999;
        double max = -999999;
        for (int i = 1; i < vsPlotData_top.size(); i++) {
            if (vsPlotData_top.at(i) > max)
                max = vsPlotData_top.at(i);
            if (vsPlotData_top.at(i) < min && vsPlotData_top.at(i) > -50)
                min = vsPlotData_top.at(i);
        }
        ImPlot::SetNextPlotLimitsY(min - 0.5, max + 0.5, plotCondition);
        ImPlot::BeginPlot("PLOT OF SINE WAVE", "Y AXIS", "X AXIS", ImVec2(ImGui::GetWindowWidth() - 18,
ImGui::GetWindowHeight()/2 - 10));

        double* dat_y = new double[vsPlotData_top.size()];
        double* dat_x = new double[vsPlotData_top.size()];
        for (int i = 0; i < vsPlotData_top.size(); i++) {
            dat_y[i] = vsPlotData_top[i];
            dat_x[i] = i;
        }
        ImPlot::PlotLine("Test Plot", dat_x, dat_y, vsPlotData_top.size());
        ImPlot::EndPlot();

        delete dat_y;
        delete dat_x;
    }

    // PLOT FOR BOTTOM GRAPH
    if (vsPlotData_bottom.size() != 0) {
        // START PLOT INSTANCE
        //ImGui::SetCursorPosY(50);
        ImPlot::SetNextPlotLimitsX(0, vsPlotData_bottom.size() + 10, plotCondition);
        double min = 999999;
        double max = -999999;
        for (int i = 1; i < vsPlotData_bottom.size(); i++) {
            if (vsPlotData_bottom.at(i) > max)
                max = vsPlotData_bottom.at(i);
            if (vsPlotData_bottom.at(i) < min)
                min = vsPlotData_bottom.at(i);
        }
        ImPlot::SetNextPlotLimitsY(min - 0.5, max + 0.5, plotCondition);
        ImPlot::BeginPlot("PLOT OF FREQUENCY COMPONENTS", "Y AXIS", "X AXIS",
ImVec2(ImGui::GetWindowWidth() - 18, ImGui::GetWindowHeight() / 2 - 10));

        double* dat_y = new double[vsPlotData_bottom.size()];
        double* dat_x = new double[vsPlotData_bottom.size()];
        for (int i = 0; i < vsPlotData_bottom.size(); i++) {
            dat_y[i] = vsPlotData_bottom[i];
            dat_x[i] = i;
        }
        ImPlot::PlotLine("Test Plot", dat_x, dat_y, vsPlotData_bottom.size());
        ImPlot::EndPlot();

        delete dat_y;
        delete dat_x;
    }

    ImGui::End();
}
```

```
// Helper to display a little (?) mark which shows a tooltip when hovered.
static void HelpMarker(const char* desc, bool same_line)
{
    if (same_line)
        ImGui::SameLine();

    ImGui::TextDisabled("(+)");
    if (ImGui::IsItemHovered())
    {
        ImGui::BeginTooltip();
        ImGui::PushTextWrapPos(ImGui::GetFontSize() * 35.0f);
        ImGui::TextUnformatted(desc);
        ImGui::PopTextWrapPos();
        ImGui::EndTooltip();
    }
}

// 0 - MAGNITUDE      1 - PHASE
std::vector<double> plotFFT(bool bComponent, std::vector<double> &vfTime)
{
    pLocalFFT->getFFT(vfTime);
    if (bComponent == true) {
        return pLocalFFT->getMagnitude();
    }
    else {
        return pLocalFFT->getPhase();
    }
}
```

cSignalGenerator.h

```
#pragma once
// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES
#include <iostream>
#include <string>
#include <vector>
#include <cmath>

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES


class cSignalGenerator
{
public:
	// Constructors
	cSignalGenerator();
	cSignalGenerator(int &iSampleFreq_Hz, int &iSignalLength_ms, int &iLowestFreq_Hz, int 
&iHighestFreq_Hz);

	// Public Methods
	std::vector<double> getSignal_Time();
	std::vector<double> getSignal_Freq();
	void generateSignal(int iSweepType);

	// Public Attributes

private:
	// Private Methods

	// Private Attributes
	std::vector<double> m_vfSignal_Time;
	std::vector<double> m_vfSignal_Freq;
	int m_iSampleFreq_Hz;
	int m_iLowestFreq_Hz;
	int m_iHighestFreq_Hz;
	int m_iSignalLength_ms;
};
```

cSignalGenerator.cpp

```
// ---------------------------------------------------------------------------
// NAME & SURNAME        : RJ VAN STADEN
// STUDENTNUMBER         : 30026792
// ---------------------------------------------------------------------------
// DATE                  : 2021/05/05
// REVISION              : rev01
// ---------------------------------------------------------------------------
// PROJECT NAME          : ChebyshevFilterGenerator
// ---------------------------------------------------------------------------

#define _USE_MATH_DEFINES

// ---------------------------------------------------------------------------
// SYSTEM INCLUDE FILES

// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES
#include <complex>

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES
#include "cSignalGenerator.h"

cSignalGenerator::cSignalGenerator(void) :
        m_iHighestFreq_Hz(0),
        m_iLowestFreq_Hz(0),
        m_iSampleFreq_Hz(0),
        m_iSignalLength_ms(0)
{
        // DEFAULT CONSTRUCTOR
}

cSignalGenerator::cSignalGenerator(int &iSampleFreq_Hz, int &iSignalLength_ms, int &iLowestFreq_Hz, int
&iHighestFreq_Hz)
{
        m_iSampleFreq_Hz = iSampleFreq_Hz;                       // Get Sample Rate from main
        m_iLowestFreq_Hz = iLowestFreq_Hz;                       // Get smallest Frequency in range for
generation
        m_iHighestFreq_Hz = iHighestFreq_Hz;             // Get largest frequency in range for generation
        m_iSignalLength_ms = iSignalLength_ms;           // Get length of signal in seconds
}

// ---------- GENERATE TIME-DOMAIN SIGNAL ----------
std::vector<double> cSignalGenerator::getSignal_Time()
{
        return m_vfSignal_Time;
}

std::vector<double> cSignalGenerator::getSignal_Freq()
{
        return m_vfSignal_Freq;
}

void cSignalGenerator::generateSignal(int iSweepType)
{
        /*
        Generate a 5 second Signal with frequencies ranging from 0 30 kHz
        - Sweep for 0 khz- 30kHz in first 2.5 seconds
        - Sweep from 30kHz - 0kHz in last 2.5 seconds
        */
        m_vfSignal_Time = { };
        double T = (double)m_iSignalLength_ms / (double)1000;
        double fNrSamples = T * m_iSampleFreq_Hz;

        for (int i = 0; i < fNrSamples; i++) {
                // Translate Nr of Samples to Times
                double t = i * (T / fNrSamples);

                switch (iSweepType) {
                // Calculate the Linear Sweep from Lowest to Highest Frequency
                case 0:
                        m_vfSignal_Time.push_back(1 * sin(2 * M_PI * ((m_iLowestFreq_Hz * t) + 
(((m_iHighestFreq_Hz - m_iLowestFreq_Hz) / (2 * T)) * pow(t, 2)))));
                        break;

                // Calculate the Logarithmic Sweep from Lowest to Highest Frequency
                case 1:
                        m_vfSignal_Time.push_back(1 * sin(2 * M_PI * m_iLowestFreq_Hz * T * 
((pow(((m_iHighestFreq_Hz) / (m_iLowestFreq_Hz)), (t / T)) - 1) / (log((m_iHighestFreq_Hz) / 
(m_iLowestFreq_Hz))))));
                        break;

                // Calculate the summation of 5 different frequencies, equally distanced
                case 2:
                        double d = sin(2 * M_PI * m_iLowestFreq_Hz * t);
                        for (int j = 1; j <= 5; j++) {
                                double f = ((double)m_iHighestFreq_Hz / (double)5) * j;
                                d = d + sin(2 * M_PI * f * t);
                        }
                        m_vfSignal_Time.push_back(d);
                        break;
                }
        }
}
```

cFastFourierTransform.h

```
#pragma once
// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <math.h>
#include <complex>

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES
#include "fftw3.h"

class cFastFourierTransform
{
public:
        // Constructors
        cFastFourierTransform();

        // Public Methods
        void getFFT(std::vector<double>& vfSignal_Time);
        std::vector<double> getMagnitude();
        std::vector<double> getPhase();

        // Public Attributes

private:
        // Private Methods
        void fft(fftw_complex* in, fftw_complex* out);
        void ifft(fftw_complex* in, fftw_complex* out);
        void displayComplex(fftw_complex* y);
        void displayReal(fftw_complex* x);
        void setMagnitude(std::vector<std::complex<double>>& vfFFT_complex);
        void setPhase(std::vector<std::complex<double>>& vfFFT_complex);
        //std::vector<double> getMagnitude();
        //std::vector<double> getPhase();

        // Private Attributes
        int m_iFFTSize;
        std::vector<std::complex<double>> m_vfFFT_complex;
        std::vector<double> m_vfMagnitude;
        std::vector<double> m_vfPhase;
        fftw_complex* x;
        fftw_complex* y;
};
```

cFastFourierTransform.cpp

```
// ---------------------------------------------------------------------------
// NAME & SURNAME        : RJ VAN STADEN
// STUDENTNUMBER         : 30026792
// ---------------------------------------------------------------------------
// DATE                  : 2021/05/05
// REVISION              : rev01
// ---------------------------------------------------------------------------
// PROJECT NAME          : ChebyshevFilterGenerator
// ---------------------------------------------------------------------------

#define _USE_MATH_DEFINES
#define REAL 0
#define IMAG 1
//#define N 524288
#define N 175000

// ---------------------------------------------------------------------------
// SYSTEM INCLUDE FILES

// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES
#include "cFastFourierTransform.h"

// ---------- CONSTRUCTOR CALLS ----------
cFastFourierTransform::cFastFourierTransform():
        m_iFFTSize(0)
{
        // DEFAULT CONSTRUCTOR
}

// ---------- RETURN FFT IN VECTOR FORMAT ----------
void cFastFourierTransform::getFFT(std::vector<double>& vfSignal_Time)
{
        // Set FFT array size
        m_iFFTSize = vfSignal_Time.size();

        // Input array
        x = new fftw_complex[N];

        // Output array
        y = new fftw_complex[N];

        int u = 0;

        // Populate the Array with values for sampling rate
        for (int i = 0; i < m_iFFTSize; i++) {
                x[i][REAL] = vfSignal_Time.at(i);
                x[i][IMAG] = 0;
                u++;
        }

        // Zero Padding for efficiency
        for (int i = m_iFFTSize; i < N; i++) {
                x[i][REAL] = 0;
                x[i][IMAG] = 0;
                u++;
        }

        // Compute FFT
        this->fft(x, y);

        // Display the results
        //std::cout << "FFT:" << std::endl;
        //this->displayComplex(y);

        // Compute IFFT
        //this->ifft(y, x);
        // Display the results
        //std::cout << "\nIFFT:" << std::endl;
        //this->displayReal(x);

        m_vfFFT_complex.clear();
        for (int i = 0; i < N; i++) {
                std::complex<double> mycomplex(y[i][REAL], y[i][IMAG]);
                m_vfFFT_complex.push_back(mycomplex);
                //std::cout << "\n" << y[i][REAL] << " " << y[i][IMAG] << "i\n" << std::endl;
        }

        // Calculate Magnitude and Phase Spectrums
        this->setMagnitude(m_vfFFT_complex);
        this->setPhase(m_vfFFT_complex);
}

// ---------- RETURN IFFT IN VECTOR FORMAT ----------

// ---------- RETURN MAGNITUDE RESPONSE ----------
std::vector<double> cFastFourierTransform::getMagnitude()
{
        return m_vfMagnitude;
}

// ---------- RETURN PHASE RESPONSE ----------
std::vector<double> cFastFourierTransform::getPhase()
{
        return m_vfPhase;
}

// ---------- COMPUTE MAGNITUDE SPECTRUM ----------
void cFastFourierTransform::setMagnitude(std::vector<std::complex<double>>& vfFFT_complex)
{
        m_vfMagnitude.clear();
        for (int i = 0; i < vfFFT_complex.size() / 2; i++) {
                double mag = abs(vfFFT_complex.at(i)) / (double)N;
                m_vfMagnitude.push_back(2*mag);
        }

        //m_vfMagnitude = 2 * m_vfMagnitude;
}

// ---------- COMPUTE PHASE SPECTRUM ----------
void cFastFourierTransform::setPhase(std::vector<std::complex<double>>& vfFFT_complex)
{
        m_vfPhase.clear();
        for (int i = 0; i < vfFFT_complex.size() / 2; i++) {
                double real = vfFFT_complex.at(i).real();
                double imag = vfFFT_complex.at(i).imag();
                double phase = atan(imag / real);
                m_vfPhase.push_back(phase);
        }
}

// ---------- COMPUTE 1-D FAST FOURIER TRANSFORM ----------
void cFastFourierTransform::fft(fftw_complex* in, fftw_complex* out)
{
        // Create a DFT plan
        fftw_plan plan = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);

        // Execute the plan
        fftw_execute(plan);

        // Cleanup
        fftw_destroy_plan(plan);
        fftw_cleanup();
}

// ---------- COMPUTE 1-D INVERSE FAST FOURIER TRANSFORM ----------
void cFastFourierTransform::ifft(fftw_complex* in, fftw_complex* out)
{
        // Create a DFT plan
        fftw_plan plan = fftw_plan_dft_1d(N, in, out, FFTW_BACKWARD, FFTW_ESTIMATE);

        // Execute the plane
        fftw_execute(plan);

        // Cleanup
        fftw_destroy_plan(plan);
        fftw_cleanup();

        // Scale the output to obtain exact inverse transform
        for (int i = 0; i < N; i++) {
                out[i][REAL] /= N;
                out[i][IMAG] /= N;
        }
}

// ---------- DISPLAY COMPLEX NUMBERS IN THE FORM a +/- bi ----------
void cFastFourierTransform::displayComplex(fftw_complex* y)
{
        for (int i = 0; i < N; i++) {
                if (y[i][IMAG] < 0) {
                        std::cout << y[i][REAL] << " - " << abs(y[i][IMAG]) << "i" << std::endl;
                }
                else {
                        std::cout << y[i][REAL] << " + " << y[i][IMAG] << "i" << std::endl;
                }
        }
}

// ---------- DISPLAY REAL PARTS OF COMPLEX NUMBERS ----------
void cFastFourierTransform::displayReal(fftw_complex* x)
{
        for (int i = 0; i < N; i++) {
                std::cout << x[i][REAL] << std::endl;
        }
}
```

cFilterDesign.h

```
#pragma once
// ---------------------------------------------------------------------------
// SYSTEM INCLUDE FILES
#include <vector>
#include <complex>

// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES

class cFilterDesign
{
public:
        // Constructors
        cFilterDesign();
        cFilterDesign(int &iOmegaPass_Hz, int &iOmegaStop_Hz, double &iRipplePass_dB, double
&iRippleStop_dB, int &iSampleRate_Hz);

        // Public Methods
        void setAnalogFilterTF();
        void setGrayMarkel(std::vector<double>& vfNumerator, std::vector<double>& vfDenominator);
        std::vector<std::complex<double>> getTransferFunction(bool bAnalog);
        std::vector<double> getYAxis(bool bAnalog, bool bMag);
        std::vector<double> getXAxis(bool bAnalog);
        std::vector<double> getLattice();
        std::vector<double> getFeedForward();

        // Public Attributes

private:
        // Private Methods
        void setAnalogMagnitude();
        void setAnalogPhase();
        void setDigitalMagnitude();
        void setDigitalPhase();
        void displayFilterParameters();

        std::vector<std::complex<double>> polyMul(std::vector<std::complex<double>> A,
std::vector<std::complex<double>> B);
        void printPoly(std::vector<double>& vfPolynomial);
        double t_n(double& fFreq);

        // Private Attributes
        double m_fDiscretePass_rads;
                // Discrete Edge Frequency (rad/s)
        double m_fDiscreteStop_rads;
                // Discrete Corner Frequency (rad/s)
        double m_fOmegaPass_rads;
                        // Analog Edge Frequency (rad/s)
        double m_fOmegaStop_rads;
                        // Analog Corner Frequency (rad/s)
        double m_fRipplePass_ratio;
                        // Analog Ripple Passband (ratio)
        double m_fRippleStop_ratio;
                        // Analog Ripple Stopband (ratio)
        double fEpsilon;
                        // Epsilon for Analog Filter Design
        double m_fNumerator_s;
                        // Numerator of Analog Transfer Function

        int m_iSampleRate_Hz;
                        // Sample Rate (Hz)
        int m_iOrder_N;
                                // Order of designed filter

        std::vector<std::complex<double>> m_vfTransferFunction_s;                // Transfer Function
vector of Analog Filter
        std::vector<double> m_vfDenominator_s;
        // Denominator of Analog Transfer Function
        std::vector<double> m_vfMagnitude_s;
        // Analog Filter Magnitude Response
        std::vector<double> m_vfMagnitude_z;
        // Digital Filter Magnitude Response
        std::vector<double> m_vfPhase_s;
        // Analog Filter Phase Response
        std::vector<double> m_vfPhase_z;
        // Digital Filter Phase Response
        std::vector<double> m_vfX_s;
                // X-axis of the Analog Magnitude
        std::vector<double> m_vfX_z;
                // X-axis of the Digital Magnitude
        std::vector<double> m_vfLattice_k;
                // Gray-Markel Realisation Lattice Vector
        std::vector<double> m_vfFeedForward_a;
        // Gray-Markel Realisation Feedforward Vector

};
```

cFilterDesign.cpp

```
// ---------------------------------------------------------------------------
// NAME & SURNAME       : RJ VAN STADEN
// STUDENTNUMBER        : 30026792
// ---------------------------------------------------------------------------
// DATE                 : 2021/05/05
// REVISION             : rev01
// ---------------------------------------------------------------------------
// PROJECT NAME         : ChebyshevFilterGenerator
// ---------------------------------------------------------------------------

#define _USE_MATH_DEFINES

// ---------------------------------------------------------------------------
// SYSTEM INCLUDE FILES

// ---------------------------------------------------------------------------
// LIBRARY INCLUDE FILES
#include <cmath>
#include <math.h>
#include <iostream>

// ---------------------------------------------------------------------------
// LOCAL INCLUDE FILES
#include "cFilterDesign.h"


// ---------- CONSTRUCTOR CALLS ----------
cFilterDesign::cFilterDesign() :
        m_fOmegaPass_rads(0.0),
        m_fOmegaStop_rads(0.0),
        m_fRipplePass_ratio(0.0),
        m_fRippleStop_ratio(0.0)
{
        // DEFAULT CONSTRUCTOR
}

cFilterDesign::cFilterDesign(int& iOmegaPass_Hz, int& iOmegaStop_Hz, double& iRipplePass_dB, double&
iRippleStop_dB, int& iSampleRate_Hz)
{
        // Set filter design paramters based on Input variables
        m_iSampleRate_Hz = iSampleRate_Hz;
        m_fDiscretePass_rads = (double)(iOmegaPass_Hz * 2 * M_PI) / (double)(m_iSampleRate_Hz);
        m_fDiscreteStop_rads = (double)(iOmegaStop_Hz * 2 * M_PI) / (double)(m_iSampleRate_Hz);
        m_fRipplePass_ratio = pow(10, iRipplePass_dB / (double)20);
        m_fRippleStop_ratio = pow(10, iRippleStop_dB / (double)20);

        // Transform the Discrete Cut Off Frequencies to the Analog Domain
        m_fOmegaPass_rads = (tan(m_fDiscretePass_rads / (double)2));
        m_fOmegaStop_rads = (tan(m_fDiscreteStop_rads / (double)2));
}

// ---------- DISPLAY FILTER PARAMETERS ----------
void cFilterDesign::displayFilterParameters()
{
        std::cout << "Discrete Pass Freq (rad/s): " << m_fDiscretePass_rads << std::endl;
        std::cout << "Discrete Stop Freq (rad/s): " << m_fDiscreteStop_rads << std::endl;
        std::cout << "Analog Pass Freq (Hz):      " << m_fOmegaPass_rads << std::endl;
        std::cout << "Analog Stop Freq (Hz):      " << m_fOmegaStop_rads << std::endl;
}

// ---------- CALCULATE ANALOG FILTER CHARACTERISTICS ----------
void cFilterDesign::setAnalogFilterTF()
{
        // Display Filter Parameters
        this->displayFilterParameters();

        // Set Order N
        double fOrder = acosh(sqrt(((1 / pow(m_fRippleStop_ratio, 2)) - 1) / ((1 /
pow(m_fRipplePass_ratio, 2)) - 1))) / acosh(m_fOmegaStop_rads / m_fOmegaPass_rads);
        m_iOrder_N = ceil(fOrder);
        std::cout << "Order (N):                  " << fOrder << "~~ " << m_iOrder_N <<std::endl;
```

```
	// Calculate the Minor and Major Axis
	fEpsilon = sqrt(1 / pow(m_fRipplePass_ratio, 2) - 1);
	std::cout << "Epsilon:                    " << fEpsilon << std::endl;
	double fUnit = pow(fEpsilon, -1) + sqrt(1 + pow(fEpsilon, -2));
	std::cout << "A:                          " << fUnit << std::endl;
	double fMinorAxis = m_fOmegaPass_rads * ((pow(fUnit, (double)1 / (double)m_iOrder_N) - pow(fUnit, 
(double)-1 / (double)m_iOrder_N)) / (double)2);
	std::cout << "Minor Axis:                 " << fMinorAxis << std::endl;
	double fMajorAxis = m_fOmegaPass_rads * ((pow(fUnit, (double)1 / (double)m_iOrder_N) + pow(fUnit, 
(double)-1 / (double)m_iOrder_N)) / (double)2);
	std::cout << "Major Axis:                 " << fMajorAxis << std::endl;

	// Determine the Denominator of the Transfer Function
	std::vector<std::complex<double>> vfDenominator;
	for (int i = 0; i < m_iOrder_N; i++) {

		double fPhi = (M_PI / (double)2) + ((((2*i) + 1)*M_PI) / (2*(double)m_iOrder_N));
		std::complex<double> fPole((fMinorAxis * cos(fPhi)), (fMajorAxis * sin(fPhi)));
		vfDenominator.push_back(fPole);
		if (m_iOrder_N % 2 == 0) {
			fPole = std::complex<double>(fPole.real(), fPole.imag() * -1);
			vfDenominator.push_back(fPole);
			i++;
		}
	}
	std::cout << "======================" << std::endl;
	std::cout << "LOW PASS FILTER DESIGN:" << std::endl;
	std::cout << "======================" << std::endl;
	for (int i = 0; i < vfDenominator.size(); i++) {
		std::cout << "Pole " << i << ": " << vfDenominator.at(i) << std::endl;
	}
	std::vector<std::complex<double>> vfA{ 1 };
	for (int i = 0; i < vfDenominator.size(); i++) {
		vfDenominator[i] = std::complex<double>(vfDenominator[i].real() * -1, 
vfDenominator[i].imag());
		std::vector<std::complex<double>> vfB{ vfDenominator.at(i), 1 };
		vfA = polyMul(vfA, vfB);
	}
	std::cout << std::endl;
	for (int i = 0; i < vfA.size(); i++) {
		m_vfDenominator_s.push_back(vfA[vfA.size() - 1 - i].real());
	}

	// Determine the Numerator of the Transfer Function
	std::complex<double> num(1,0);
	for (int i = 0; i < vfDenominator.size(); i++) {
		num = num * vfDenominator.at(i);
	}
	if (m_iOrder_N % 2 == 0) {
		num = num / sqrt(1 + pow(fEpsilon, 2));			// If Even order, divide by sqrt 
(1 + epsilon^2)
	}
	m_fNumerator_s = num.real();

	// Display LP Transfer Function
	std::cout << "\t\t\t" << m_fNumerator_s << std::endl;
	std::cout << "------------------------------------------------------------------" << std::endl;
	this->printPoly(m_vfDenominator_s);

	std::cout << "\n=====================" << std::endl << std::endl;

	// Continue to Rest of Filter Design
	this->setAnalogMagnitude();
	this->setAnalogPhase();
	this->setDigitalMagnitude();
	this->setDigitalPhase();
}

// ---------- Multiply Factors of Polynomial ----------
std::vector<std::complex<double>> cFilterDesign::polyMul(std::vector<std::complex<double>> A, 
std::vector<std::complex<double>> B)
{
        std::vector<std::complex<double>> vfProd;
        int m = A.size();
        int n = B.size();
        // Initialize the porduct polynomial
        for (int i = 0; i < m + n - 1; i++)
                vfProd.push_back((0, 0));

        // Multiply two polynomials term by term

        // Take ever term of first polynomial
        for (int i = 0; i < m; i++)
        {
                // Multiply the current term of first polynomial
                // with every term of second polynomial.
                for (int j = 0; j < n; j++)
                        vfProd.at(i + j) += A.at(i) * B.at(j);
        }

        return vfProd;
}

// ---------- Print a Multiplied Polynomial ----------
void cFilterDesign::printPoly(std::vector<double>& vfPolynomial)
{
        int n = vfPolynomial.size();
        for (int i = 0; i < n; i++)
        {
                std::cout << vfPolynomial.at(i);
                if (i != n - 1) {
                        std::cout << "s^" << (n - 1 - i);
                        std::cout << " + ";
                }
        }
        std::cout << std::endl;
}

// ---------- Analog Design: T(N) Function ----------
double cFilterDesign::t_n(double& fFreq)
{
        if (abs(fFreq) <= 1) {
                return cos(m_iOrder_N * acos(fFreq));
        }
        else {
                return cosh(m_iOrder_N * acosh(fFreq));
        }
}

// ---------- Determine Analog Magnitude ----------
void cFilterDesign::setAnalogMagnitude()
{
        m_vfMagnitude_s = { };
        //m_vfPhase_s = { };
        m_vfX_s = { };
        for (int i = 0; i < m_iSampleRate_Hz; i++) {
                // Sweeping through possible frequencies
                std::complex<double> num(m_fNumerator_s, 0);
                std::complex<double> den(0, 0);

                //double jw = i * 2 * M_PI;
                double jw = (i * 2 * M_PI) / (double)m_iSampleRate_Hz;
                for (int j = 0; j < m_vfDenominator_s.size(); j++) {
                        //std::complex<double> val = (m_fOmegaPass_rads * m_fOmegaStop_rads) /
std::complex<double>(0, jw);

                        std::complex<double> s = std::complex<double>(0, jw);
                        std::complex<double> val = std::complex<double>(m_fOmegaStop_rads *
m_fOmegaPass_rads, 0) / s;

                        //den = den + (m_vfDenominator_s.at(j) * pow(std::complex<double>(0, jw),
m_vfDenominator_s.size() - 1 - j));
                        den = den + (m_vfDenominator_s.at(j) * pow(val, m_vfDenominator_s.size() - 1 -
j));
                }
                std::complex<double> val = num / den;
                m_vfMagnitude_s.push_back(20 * log10(abs(val)));
                m_vfX_s.push_back(jw);
        }
}

// ---------- Determine Digital Magnitude ----------
void cFilterDesign::setDigitalMagnitude()
{
        m_vfMagnitude_z = { };
        //m_vfPhase_s = { };
        m_vfX_z = { };
        for (int i = 0; i < m_iSampleRate_Hz / 2; i++) {
                // Sweeping through possible frequencies
                std::complex<double> num(m_fNumerator_s, 0);
                std::complex<double> den(0, 0);

                //double jw = i * 2 * M_PI;
                double jw = (i * 2 * M_PI) / (double)m_iSampleRate_Hz;
                for (int j = 0; j < m_vfDenominator_s.size(); j++) {

                        std::complex<double> ejw = exp(std::complex<double>(0, jw));
                        std::complex<double> s = (ejw - std::complex<double>(1, 0)) / (ejw +
std::complex<double>(1, 0));
                        std::complex<double> val = std::complex<double>(m_fOmegaStop_rads *
m_fOmegaPass_rads, 0) / s;

                        //den = den + (m_vfDenominator_s.at(j) * pow(s, m_vfDenominator_s.size() - 1 -
j));
                        den = den + (m_vfDenominator_s.at(j) * pow(val, m_vfDenominator_s.size() - 1 -
j));
                }

                std::complex<double> val = num / den;
                double mag = sqrt(pow(val.real(), 2) + pow(val.imag(), 2));
                m_vfMagnitude_z.push_back(20 * log10(mag));
                m_vfX_z.push_back(jw);
        }
}

// ---------- Determine Analog Phase ----------
void cFilterDesign::setAnalogPhase()
{
        m_vfPhase_s = { };
        m_vfTransferFunction_s = { };
        for (int i = 0; i < m_iSampleRate_Hz; i++) {
                // Sweeping through possible frequencies
                std::complex<double> num(m_fNumerator_s, 0);
                std::complex<double> den(0, 0);

                //double jw = i * 2 * M_PI;
                double jw = (i * 2 * M_PI) / (double)m_iSampleRate_Hz;
                for (int j = 0; j < m_vfDenominator_s.size(); j++) {

                        std::complex<double> s = std::complex<double>(0, jw);
                        std::complex<double> val = std::complex<double>(m_fOmegaStop_rads *
m_fOmegaPass_rads, 0) / s;

                        //den = den + (m_vfDenominator_s.at(j) * pow(std::complex<double>(0, jw),
m_vfDenominator_s.size() - 1 - j));
                        den = den + (m_vfDenominator_s.at(j) * pow(val, m_vfDenominator_s.size() - 1 -
j));
                }

                std::complex<double> val = num / den;
                m_vfTransferFunction_s.push_back(val);
                m_vfPhase_s.push_back(atan(val.imag() / val.real()));
        }
}

// ---------- Determine Digital Phase ----------
void cFilterDesign::setDigitalPhase()
```

```
{
	m_vfPhase_z = { };
	m_vfTransferFunction_s = { };
	for (int i = 0; i < m_iSampleRate_Hz / 2; i++) {

		// Sweeping through possible frequencies
		std::complex<double> num(m_fNumerator_s, 0);
		std::complex<double> den(0, 0);

		//double jw = i * 2 * M_PI;
		double jw = (i * 2 * M_PI) / (double)m_iSampleRate_Hz;
		for (int j = 0; j < m_vfDenominator_s.size(); j++) {
			//std::complex<double> val = (m_fOmegaPass_rads * m_fOmegaStop_rads) /
std::complex<double>(0, jw);

			std::complex<double> ejw = exp(std::complex<double>(0, jw));
			std::complex<double> s = (ejw - std::complex<double>(1, 0)) / (ejw +
std::complex<double>(1, 0));
			std::complex<double> val = std::complex<double>(m_fOmegaStop_rads *
m_fOmegaPass_rads, 0) / s;

			den = den + (m_vfDenominator_s.at(j) * pow(val, m_vfDenominator_s.size() - 1 -
j));
		}

		std::complex<double> val = num / den;
		m_vfTransferFunction_s.push_back(val);
		m_vfPhase_z.push_back(atan(val.imag() / val.real()));
	}
}

// ---------- Determine Analog Phase ----------
std::vector<std::complex<double>> cFilterDesign::getTransferFunction(bool bAnalog)
{
	if (bAnalog == true) {
		// Return Analog Transfer Function
		return m_vfTransferFunction_s;
	}
	else {
		// Return Digital Transfer Function
		//return m_vfTransferFunction_z;
	}
}

// ---------- Determine Analog Phase ----------
std::vector<double> cFilterDesign::getYAxis(bool bAnalog, bool bMag)
{
	if (bAnalog == true) {
		// Return Analog Magnitude / Phase
		if (bMag == true) {
			// Return Magnitude Spectrum
			return m_vfMagnitude_s;
		}
		else {
			return m_vfPhase_s;
		}
	}
	else {
		// Return Digital Magnitude / Phase
		if (bMag == true) {
			// Return Magnitude Spectrum
			return m_vfMagnitude_z;
		}
		else {
			return m_vfPhase_z;
		}
	}
}

// ---------- Gray-Markel Realisation
void cFilterDesign::setGrayMarkel(std::vector<double>& vfNumerator, std::vector<double>& vfDenominator)
{
	// Clear Lattice and Feedback Vectors
	m_vfLattice_k = { };
	m_vfFeedForward_a = { };

	// Get Degree of Polynomial
	int N = vfDenominator.size() - 1;

	std::vector<double> a_1;
	std::vector<double> del;
	for (int i = 0; i < N;  i++) {
		// Initialise Lattice Vector with ones
		m_vfLattice_k.push_back(1);
	}

	// Determine a_1
	for (int i = 0; i < N + 1; i++) {
		a_1.push_back(vfDenominator.at(i) / vfDenominator.at(0));
		del.push_back(vfDenominator.at(i) / vfDenominator.at(0));
	}

	// Initialise Feed-Forward vector
	for (int i = N; i >= 0; i--) {
		m_vfFeedForward_a.push_back(vfNumerator.at(i) / vfDenominator.at(0));
	}

	// Gray-Markel Realisation
	for (int i = N - 1; i >= 0; i--) {
		// Set Feed-Forward
		int t = 1;
		for (int j = N - i; j <= N; j++) {
			m_vfFeedForward_a[j] = m_vfFeedForward_a[j] - (m_vfFeedForward_a[N - i - 1] *
a_1[t]);
			t++;
		}

		// Set Lattice
		m_vfLattice_k[i] = a_1[i+1];

		// Set a_1
		t = i+1;
		for (int j = 0; j <= i+1; j++) {
			del[j] = (a_1[j] - (m_vfLattice_k[i] * a_1[t])) / ((double)1 - (m_vfLattice_k[i]
* m_vfLattice_k[i]));
			t--;
		}
		a_1 = del;
	}
}

// ---------- Determine Analog Phase ----------
std::vector<double> cFilterDesign::getXAxis(bool bAnalog)
{
	if (bAnalog == true) {
		// Return Analog x-axis
		return m_vfX_s;
	}
	else {
		// Return Digital x-axis
		return m_vfX_z;
	}
}

// ---------- Return Lattice Vector (Gray-Markel) ----------
std::vector<double> cFilterDesign::getLattice()
{
	return m_vfLattice_k;
}

// ---------- Return Feed-Forward Vector (Gray-Markel) ----------
std::vector<double> cFilterDesign::getFeedForward()
{
	return m_vfFeedForward_a;
}
```