Chebyshev Digital Filter Design

**Ruben Johann van Staden**

**Module code:** EERI 414

**Degree:** BEng Computer Electronic Engineering

**Date:** 2021/06/02

**Student number:** 30026792

Abstract

Table of contents

[Abstract i](#_Toc72967580)

[Introduction 7](#_Toc72967581)

[1.1 Overview 7](#_Toc72967583)

[1.2 Background 7](#_Toc72967584)

[1.2.1 Filter Design Fundamentals 8](#_Toc72967585)

[1.2.2 Chebyshev Filter 8](#_Toc72967586)

[Analytical Design 10](#_Toc72967587)

[2.1 Analog Domain Edge Frequencies 10](#_Toc72967589)

[2.2 Analog Lowpass Filter Design 11](#_Toc72967590)

[2.2.1 Filter Order 12](#_Toc72967591)

[2.2.2 Pole Calculation 12](#_Toc72967592)

[2.2.3 Numerator Calculation 13](#_Toc72967593)

[2.2.4 Numerator Calculation 14](#_Toc72967594)

[2.3 Analog High-Pass Filter Design 15](#_Toc72967595)

[2.4 Digital High-Pass Filter Design (Bilinear Transform) 15](#_Toc72967596)

[2.5 Gray-Markel Realisation 15](#_Toc72967597)

[SOFTWARE SOLUTION 16](#_Toc72967598)

[3.1 Overview 16](#_Toc72967600)

[3.2 Analog Filter Design 16](#_Toc72967601)

[3.3 Digital Filter Design 16](#_Toc72967602)

[3.4 Input Signal Generation 16](#_Toc72967603)

[3.5 Gray-Markel Realisation 16](#_Toc72967604)

[CONCLUSION 17](#_Toc72967605)

[references 18](#_Toc72967607)

[appendiX A - Software source code 19](#_Toc72967609)

[ChebyshevFilterGenerator.cpp 19](#_Toc72967610)

[cSignalGenerator.h 30](#_Toc72967613)

[cSignalGenerator.cpp 31](#_Toc72967614)

[cFastFourierTransform.h 33](#_Toc72967615)

[cFastFourierTransform.cpp 34](#_Toc72967616)

[cFilterDesign.h 37](#_Toc72967617)

[cFilterDesign.cpp 39](#_Toc72967618)

List of Tables

[Table ‎1‑1: Filter Design Specifications 7](#_Toc72967619)

[Table ‎2‑1: Analog Lowpass Filter Design Parameters 11](#_Toc72967620)

List of Figures

[Figure ‎1‑1: Normalized Magnitude Spectrum of a Digital Lowpass Filter 8](file:///C:\NWU%20Pukke\BIng_RekenaarElektronies_4_2021\Semester%201\EERI%20414%20-%20Signal%20Theory%20III\EERI414_Practical_2021\Technical%20Report\EERI414_30026792_vanStaden_RJ_Practical.docx#_Toc72967621)

Nomenclature

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

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 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.

# Heading 1 won’t print. Don’t delete – doing so will lead to incorrect numbering.

## 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 accurate 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 ripple of 0.3dB. The chosen edge and corner frequencies, together with the complete set of specifications are shown in Table 1.

Table ‑: Filter Design Specifications

|  |  |  |
| --- | --- | --- |
| **Specification** | **Value** | **Unit** |
| Sample Rate () | 175 000 | Hz |
| Passband Edge Frequency () | 72 250 | Hz |
| Stopband Corner Frequency () | 42 500 | Hz |
| Passband Ripple () | - 0.25 | dB |
| Stopband Attenuation () | - 40 | dB |

## 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 will not be followed. The first step that will be followed 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.

### 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 , where refers to the minimum stopband attenuation.

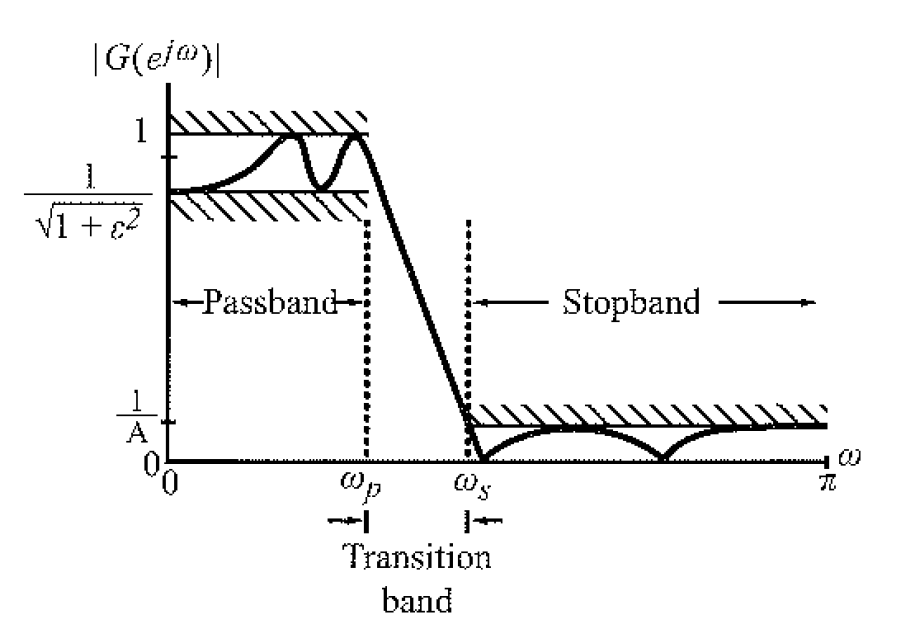


Figure ‑: Normalized Magnitude Spectrum of a Digital Lowpass Filter

### 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].

There exist two types of Chebyshev digital filters. The first type is characterised by a ripple in the passband, while a ripple in the stopband is unique to Type-2 Chebyshev filters [1].

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.

# Heading 1 won’t print. Don’t delete – doing so will lead to incorrect numbering.

## 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.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-1) |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-2) |

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.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-3) |

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-4) |

The specifications in section 1.1, specifies that a high-pass filter be designed. As such, the values and 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.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-5) |

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-6) |

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.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-7) |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-8) |
|  |  |  |
|  |  |  |

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

Table ‑: Analog Lowpass Filter Design Parameters

|  |  |  |
| --- | --- | --- |
| **Specification** | **Value** | **Unit** |
| Passband Edge Frequency () | 72 250 | rad/s |
| Stopband Corner Frequency () | 42 500 | rad/s |
| Passband Ripple () | 0.97163 |  |
| Stopband Attenuation () | 0.01 |  |

## 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.

### Filter Order

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

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-9) |
|  |  |  |
|  |  |  |

### 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.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-10) |

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 and are known. The value can be obtained by assessing the maximum passband ripple as shown in Figure ‎1‑1.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-11) |
|  |  |  |
|  |  |  |

The value for can be calculated next.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-12) |
|  |  |  |

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

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-13) |
|  |  |  |
|  |  | (‎2-14) |
|  |  |  |

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

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-15) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-16) |

### 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 value in the pole equation with 0. For even order filters, the 0-replacement value is then further divided by . For the order 4 filter described above, this process will be followed.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-17) |
|  |  |  |

### Transfer Function

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

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  | (‎2-18) |

## Analog High-Pass Filter Design

The final analog lowpass transfer function described by Equation 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.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  | (‎2-19) |

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

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-20) |

## 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.

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-16) |

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

|  |  |  |
| --- | --- | --- |
|  |  | (‎2-20) |

## Gray-Markel Realisation

SOFTWARE SOLUTION

# Heading 1 won’t print. Don’t delete – doing so will lead to incorrect numbering.

## Overview

## Analog Filter Design

## Digital Filter Design

## Input Signal Generation

## Gray-Markel Realisation

CONCLUSION

# Heading 1 won’t print. Don’t delete – doing so will lead to incorrect numbering.

references

# Bibliography

|  |  |
| --- | --- |
| [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;

// ---------- 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;

}