ECE 4020

Lab Part 3 Report

Ryan Colon

02.19.22

Introduction

Contained in this report is the documentation of the design and evaluation process for a frequency shifter program intended to run on a general-purpose computer with sound card. To implement this program, a Hilbert filter was designed in MATLAB and implemented in the filter program discussed in [i] with slight modifications to implement frequency shifting capabilities. First the relevant theory will be discussed, followed by the design of the Hilbert filter and adjustments to the filter program, and then the testing process and subsequent conclusions will be detailed.

Theory

To create a frequency shifted signal, we start by combining a signal with its Hilbert transform to create a related analytic signal that has a single-sided spectrum. We can then frequency shift this analytic signal using the frequency shifting property of the discrete-time Fourier transform and take only the real part to reconstruct the original signal. Shown below is the equation that results from these steps:

Let $X(n)$ denote the original signal, $X_h(n)$ then is the Hilbert transform, and $Y(n)$ is the frequency shifted version of X(n).

$$Y(n) = X(n)\cos(\omega_0 n) - X_h(n)\sin(\omega_0 n)$$

Where ω_0 is the desired frequency shift.

Using this equation tells us what needs to be added to the original filter program as well as letting us know that we need to have a working Hilbert filter to implement it.

Hilbert Filter Design

To begin designing the filter, specifications must first be established. For this lab, these specifications were: a working input frequency range of 100 Hz to 14 kHz and an ability to shift from -16 kHz to + 16kHz. The sampling rate will be 32 kHz and we want a minimum of -50 dB rejection of the unwanted side band relative to the wanted sideband. Given these specifications, a working filter can now be designed.

MATLAB has a few different methods that can be used to design Hilbert filters. Multiple ways to do so were tested and it was found that the easiest method to work with was ‘firpm’. ‘firpm’ can be used to design a suite of filters, but in this case,

parameters were passed to it in the form of a bandpass filter plus one specifier that lets it know that a Hilbert filter is desired. The following snippet is the implementation of this for the filter used in this lab:

```
Fs = 32e3;
n = 1200;
amps = [0 1 1 1 1 0];
freqs = [0 49 59 15.7e3 15.701e3 16e3] / (Fs/2);
w = [0.1 1 0.2];


Coeffs = firpm(n, freqs, amps, w, 'hilbert');
```

Figure 1. MATLAB Filter Designer

The band amplitudes were set to be 1 because that is the behavior we expect from a Hilbert filter, and the: number of coefficients, weights, and frequency ranges were determined through testing a series of values and choosing the best performing ones. This took a few hundred iterations to determine. Using ‘freqz’ to plot the frequency response produces:

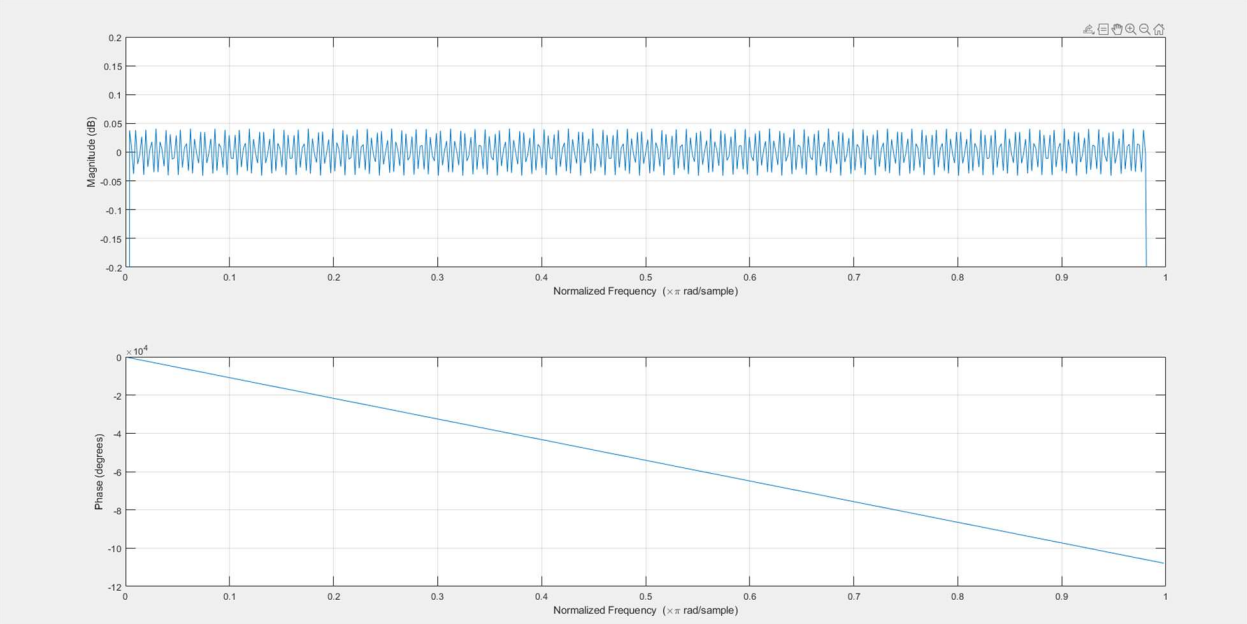

Figure 2. MATLAB Frequency Response

The response appears to be behaving correctly, so this filter can move on to being used in the program. The full MATLAB code for this section is supplied in Appendix A.

Program

The goal of this program is to implement the frequency shifting algorithm. For this lab we would also like to be able to adjust the frequency we shift by as the program is running. From the previous lab [i], there is already a very good foundation for this program that can be modified to meet the given goal.

In the original program, the function process samples was used to implement the filter. We can add a few extra lines to implement the frequency shift:

```
//Function is called whenever audio samples become available from the soundcard
int process_samples (jack_nframes_t nframes, void *arg)
{
    jack_default_audio_sample_t* in;
    jack_default_audio_sample_t* out1;

    static float pipeline[maxCoeff] = {0.0};

    float hilbTrans;

    in = jack_port_get_buffer (inPort, nframes);
    out1 = jack_port_get_buffer (outPort, nframes);

    for (int i=0; i < nframes ;i++) {

        for(int j = numCoeff - 1; j > 0; j--)
            pipeline[j] = pipeline[j-1];

        pipeline[0] = in[i];
        hilbTrans = 0;

        for(int j = 0; j < numCoeff; j++)
            hilbTrans += pipeline[j]*coeffArray[j];


        thisTheta = (thisTheta > 2*M_PI) ? (thisTheta - (2*M_PI)) : thisTheta;
        thisTheta = (thisTheta < (-2*M_PI)) ? (thisTheta + (2*M_PI)) : thisTheta;

        out1[i] = ((pipeline[(int)floor(numCoeff/2)]*cos(thisTheta)) - (hilbTrans*sin(thisTheta)));
        thisTheta += delTheta;
    }

    return 0;
}
```

Figure 3. Modified Process Samples

To break down the process: the function receives a series of inputs from the soundcard, applies the Hilbert filter, finds the remainder of the current theta divided by 2π, runs and outputs the frequency shift equation, and then increments theta. Theta in this program is an accumulator implementation of the $\omega_0 n$ portion of the equation, and it was implemented in this way to prevent many issues associated with the values of n getting large.

The only other thing left to modify is the main loop to give the user the opportunity to input their desired frequency shift:

```
fprintf(stderr,"Running ... press CTRL-C to exit ...  \n");
while (1) {
    printf("Enter desired frequency shift in Hz: ");

    if(scanf("%f", &frequencyShiftInput));
    delTheta = 2*M_PI*frequencyShiftInput/sample_rate;
}
```

Figure 4. Modified Main Loop

Since there is no second output for this program, the option of displaying it for the user was removed and replaced by a prompt asking the user for the desired frequency shift. After the user inputs a frequency shift, which is expected to be in Hz, it is converted to radians and then divided by the sample rate to get the 'deltaTheta' value, which is the value that theta gets incremented by every sample. With these modifications the program is ready to run for this lab. The full C program is supplied in Appendix B.

Test Procedure

List of equipment used:

- Arbitrary Function Generator
- Oscilloscope
- gnu-Linux PC with soundcard and speaker
- Spectrum Analyzer

The evaluation of the frequency shifter began with a simple test to check to make sure everything was operating as it should be. For this test, the function generator output was connected to the mic in of the pc and the line out of the pc was hooked up to the channel 1 input of the oscilloscope as well as a speaker. A block diagram of the connections is show on the next page:

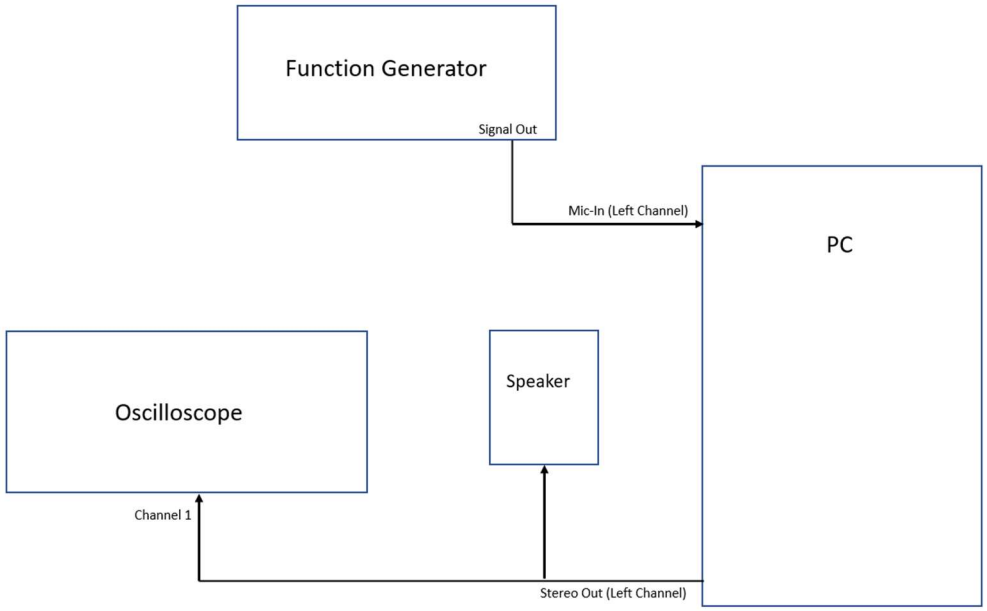

Figure 5. Basic Test Connection Diagram

To test basic operation, a sinusoidal signal of frequency 1 kHz was provided to the PC. A series of frequency shifts: -500 Hz, + 1000 Hz, and – 2000 Hz was then applied to the signal and monitored on the oscilloscope. The speaker was also turned up for the duration of the test as any issues would be easy to hear. If the program and filter were not working as intended the output signal would appear as though it was being amplitude modulated on the oscilloscope. The oscilloscope output of these tests are shown below:

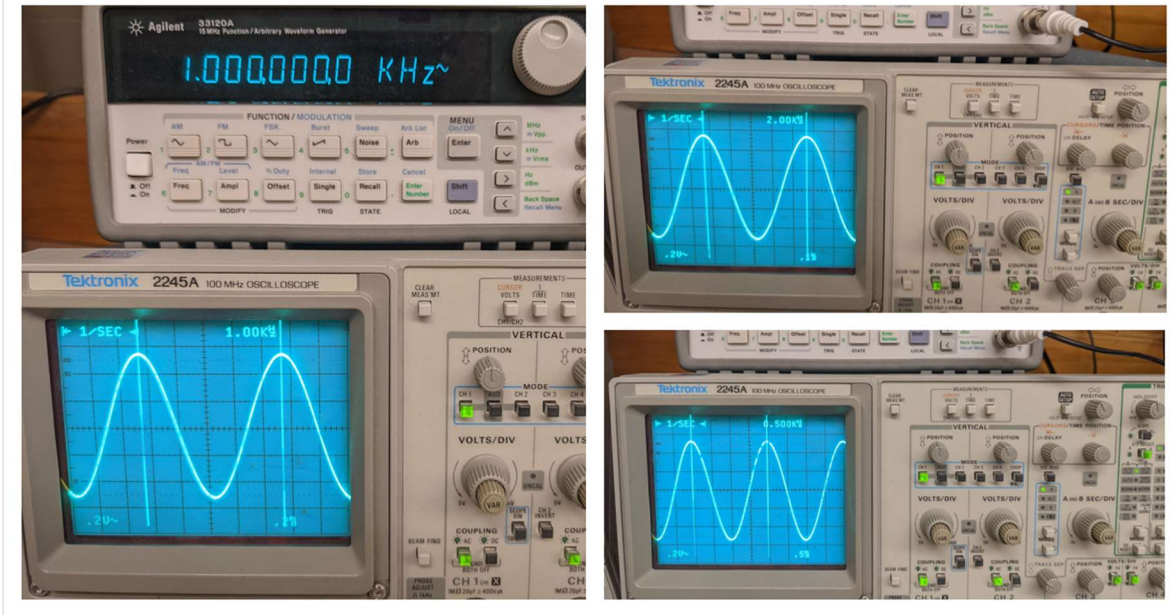

Figure 6. Basic Functionality Test (Left: -500 Hz, Top Right: +1 kHz, Bottom Right: -2 kHz)

By listening to the output of the speaker and observing the oscilloscope it is obvious the shifter is working as intended. So now it's performance can be thoroughly scrutinized on the Spectrum Analyzer.

For this test, the function generator was left connected the same way it was before, but the PC line out left channel was plugged into R1 on the spectrum analyzer. A block diagram of this configuration is shown below:

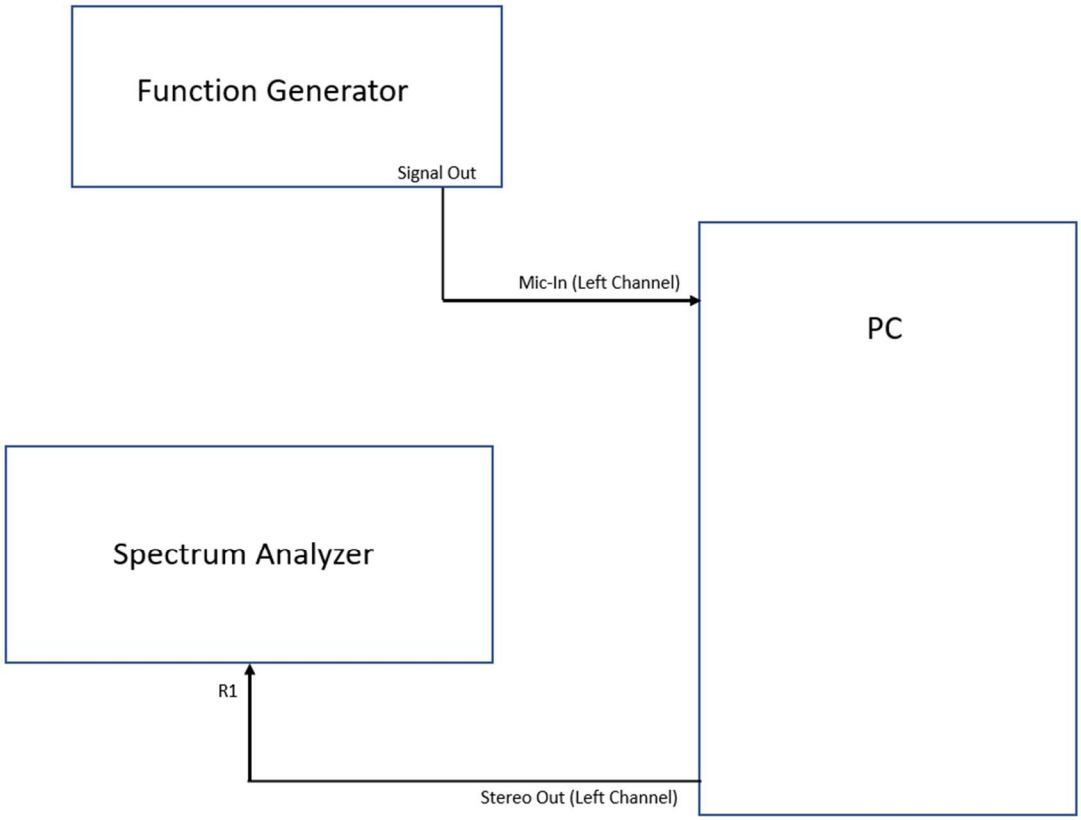

Figure 7. Performance Test Connection Diagram

The purpose of this test is to make sure the filter meets specifications such that the frequency shifter works as intended. In total, 4 sets of frequencies and shifts were tested across the spectrum. In order they are: 12.1 kHz + 1.2 kHz, 3.6 kHz + 1.3 kHz, 100 Hz + 400 Hz, 14 kHz + 1.2 kHz. They are imaged below in order:

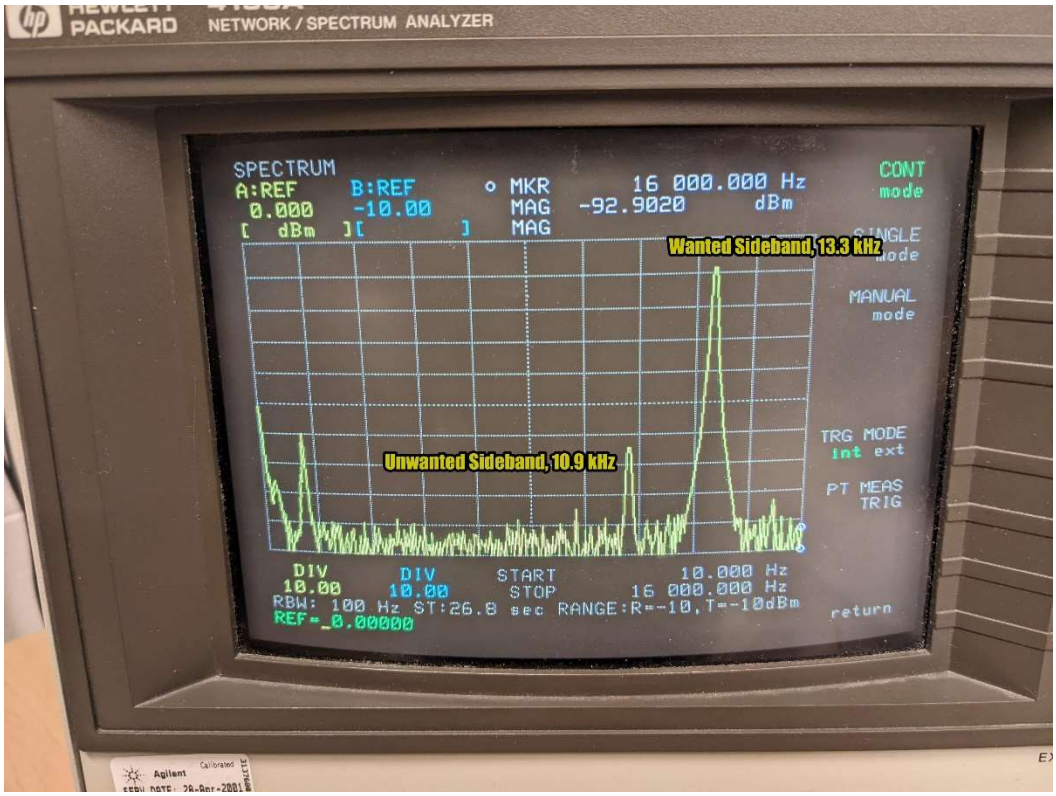

Figure 8. 12.1 kHz + 1.2 kHz

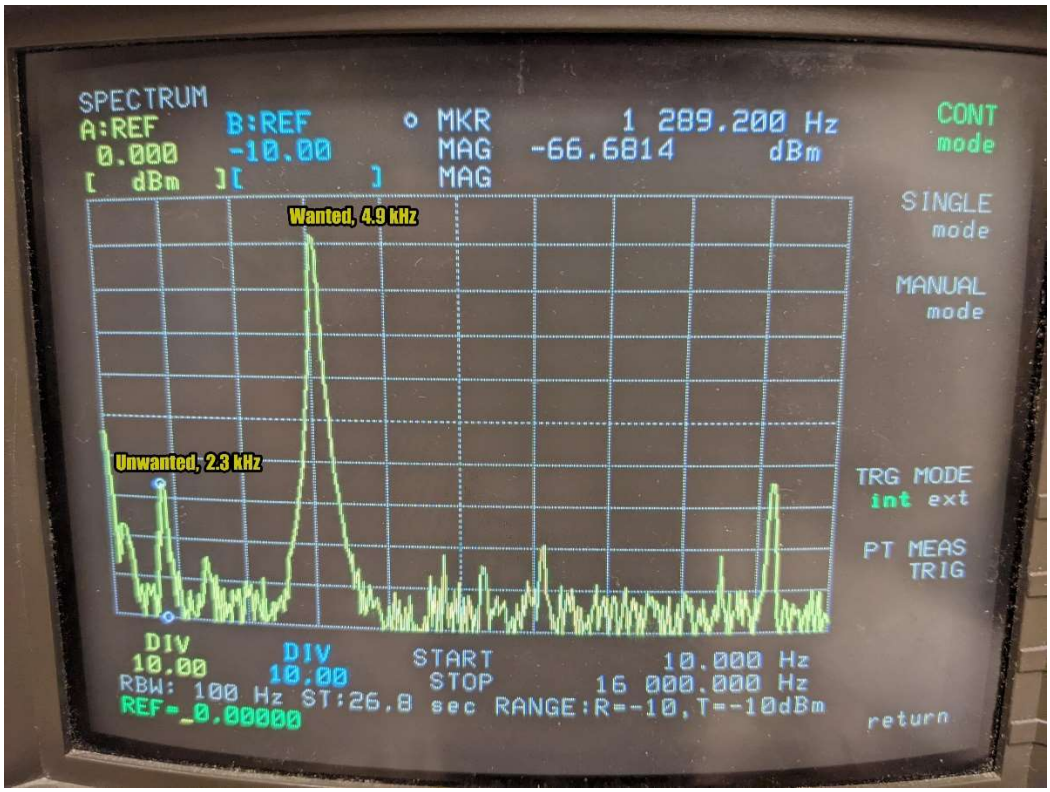

Figure 9. 3.6 kHz + 1.3 kHz

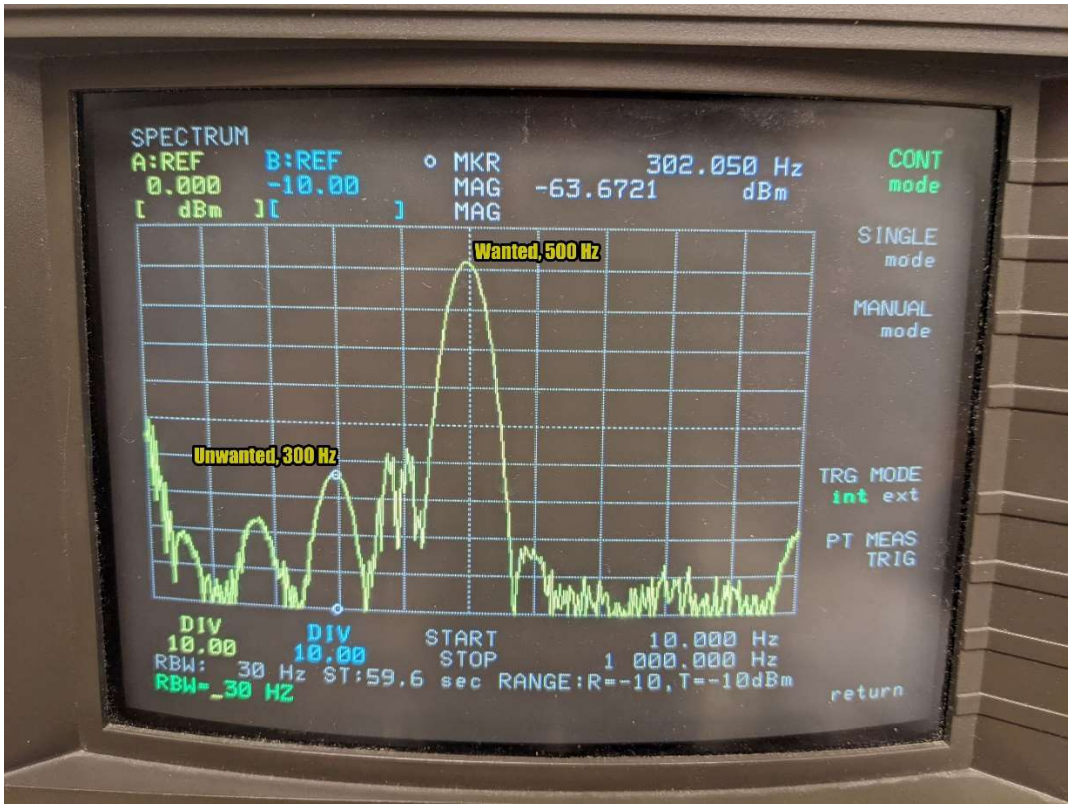

Figure 10. 100 Hz + 400 Hz

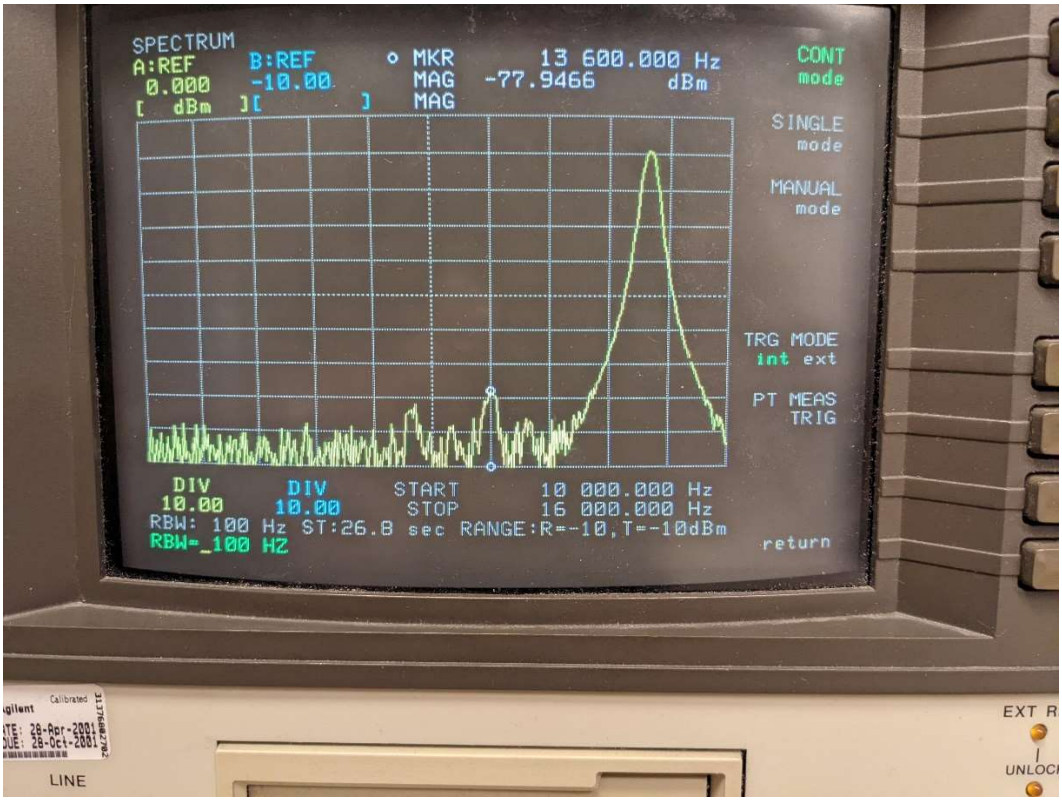

Figure 11. 14 kHz + 1.2 kHz

To repeat, the goal is -50 dB suppression of the unwanted sideband across the entire spectrum. To make it easy to visualize, the vertical divisions were set to 10 dB/div. As can be seen in all the above images, the filter has provided well over 5 divisions suppression of the unwanted sideband, thus meeting the specified criteria.

But before declaring the filter to meet specifications, one more test was conducted to measure the ripple of the filter and find its worst-case suppression. For this test, the spectrum analyzer was used again, and all the wires were left connected the way they were in the previous test. The input signal for this test started at 3.6 kHz and was shifted by 1.3 kHz. The spectrum analyzer was then set to manual mode and a measurement of the wanted signal was taken, ~ -8.7 dB. The wanted signal will shift very little for this test, so this measurement was used as the reference. Next, measurements of the unwanted sideband were taken ~-64.4 dB.

Leaving the spectrum analyzer to measure just the unwanted sideband, the function generator input signal was shifted in 1 Hz steps from 3.4 kHz to 3.8 kHz. The unwanted sideband was measured at every Hz in this range to see where the worst-case suppression occurred and how much it was. In this case, worst case suppression occurred at roughly 3.55 kHz and was roughly -53 dB suppression, still within specifications. With this test concluded, the filter has now been proven to meet specifications completely. There was no good way to image this test occurring, so no image has been provided.

Concluding Remarks

In this report, a frequency shifter program was created and thoroughly criticized to ensure it met the desired specifications. The theory for a frequency shifter application was discussed and followed up by application to the design of a Hilbert filter and C program implementation. This program was then subject to a variety of tests that conclusively showed the shifter met specifications.

Though this shifter surpassed the specifications, there is room for improvement. The design process for the Hilbert filter was suboptimal, and most likely resulted in the MATLAB ‘firpm’ specifications being well over-constrained and thus the number of coefficients being way larger than necessary (1200). If computation time/power is a concern, one might consider going back to the MATLAB Hilbert filter design and optimizing it to reduce the number of coefficients.

Citations

i. Colon, R., 2022. *ECE 4020 Lab Part 2 Report*.

Appendix

A. MATLAB Hilbert Filter Designer Code

```
%DSP_HILBApprox_1.m
%Ryan Colon
%02.12.22
%Purpose is to create a causal approximation of a Hilbert Transform Filter
%meeting following specifications

%Specifications:
%Sample rate: 32 kHz
%Input Signal Range: 100 Hz to 14 kHz
%Frequency shift range: -16 kHz to +16 kHz
%Attenuation of unwanted component: >-50 dB
%Frequency Shift needs to be able to change mid program

clear
clc

Fs = 32e3;
n = 1200;
amps = [0 1 1 1 1 0];
freqs = [0 49 59 15.7e3 15.701e3 16e3] / (Fs/2);
w = [0.1 1 0.2];


Coeffs = firpm(n, freqs, amps, w, 'hilbert');
freqz(Coeffs)
ylim([-0.2 0.2])

%Write the coefficients to a file for submission
FID = fopen('RyanColon_HilCoeff17.txt','w');

if FID > 0
    fprintf(FID, "Ryan Colon, Hilbert Transform Filter, Attempt 17
(Guess)\n");

    for i = 1:(length(Coeffs)-1)
        fprintf(FID, '%.14f\n', Coeffs(i));
    end
    fprintf(FID, '%.14f', Coeffs(length(Coeffs)));
    fclose(FID);
end
```

B. Frequency Shifter C Program

```
/*  FreqShift.C
    Ryan Colon
    02.14.22
    Program takes an input signal from mic (left channel) and a user input frequency shift to shift the frequency
of the input
    signal and outputs it to stereo (left channel). Original signal is output stereo (channel right) */

#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include <math.h>

#include <jack/jack.h>

#define maxCoeff 2000

jack_port_t* inPort;
jack_port_t* outPort;
jack_client_t* client;

int numCoeff = 0;
float coeffArray[maxCoeff];
float frequencyShiftInput = 0;
float delTheta = 0;
float thisTheta = 0;

//Function is called whenever audio samples become available from the soundcard
int process_samples (jack_nframes_t nframes, void *arg)
{
    jack_default_audio_sample_t* in;
    jack_default_audio_sample_t* out1;

    static float pipeline[maxCoeff] = {0.0};

    float hilbTrans;

    in = jack_port_get_buffer (inPort, nframes);
    out1 = jack_port_get_buffer (outPort, nframes);

    for (int i=0; i < nframes ;i++) {
```

```
        for(int j = numCoeff - 1; j > 0; j--)
            pipeline[j] = pipeline[j-1];

        pipeline[0] = in[i];
        hilbTrans = 0;

        for(int j = 0; j < numCoeff; j++)
            hilbTrans += pipeline[j]*coeffArray[j];


        thisTheta = (thisTheta > 2*M_PI) ? (thisTheta - (2*M_PI)) : thisTheta;
        thisTheta = (thisTheta < (-2*M_PI)) ? (thisTheta + (2*M_PI)) : thisTheta;

        out1[i] = ((pipeline[(int)floor(numCoeff/2)]*cos(thisTheta)) - (hilbTrans*sin(thisTheta)));
        thisTheta += delTheta;
    }

    return 0;
}

/**
 * JACK calls this shutdown_callback if the server ever shuts down or
 * decides to disconnect the client.
 */
void jack_shutdown (void *arg)
{
    exit (EXIT_FAILURE);
}

int main (int argc, char *argv[])
{
    const char **ports;
    const char *client_name = "Frequency-Shift-Client";   /***** you can change the client name *****/
    const char *server_name = NULL;
    jack_options_t options = JackNullOption;
    jack_status_t status;

    int sample_rate;

    char* preppend = "./Coefficients/";

    char fileAddress[strlen(preppend) + strlen(argv[1])];

    int count = 0;
```

```
while(preppend[count] != '\0'){
    fileAddress[count] = preppend[count];
    ++count;
}

int currCount = count;
count = 0;
while(argv[1][count] != '\0'){
    fileAddress[currCount] = argv[1][count];
    ++count;
    ++currCount;
}

fileAddress[currCount] = '\0';

FILE* coeffFile = fopen(fileAddress, "r");

if(NULL == coeffFile){
    printf("Couldn't open file, filename:\t%s\n",argv[1]);
    return(0);
}

char firstLine[200];
if(fgets(firstLine, 200, coeffFile) == NULL)
    return(0);

printf("%s\n", firstLine);

float fileVal = 0;
while(fscanf(coeffFile, "%f", &fileVal) == 1){
    if(numCoeff >= maxCoeff){
        printf("Too many coefficients, max number of coefficients:\t %d\nExitting Program.\n", maxCoeff);
        return 0;
    }
    coeffArray[numCoeff] = fileVal;
    numCoeff++;
}

for(int i = 0; i < numCoeff; i++)
    printf("Coefficient %d:    \t%f\n", i, coeffArray[i]);

fclose(coeffFile);

/* open a client connection to the JACK server */
```

```
client = jack_client_open (client_name, options, &status, server_name);
if (client == NULL) {
    fprintf (stderr, "jack_client_open() failed, "
         "status = 0x%2.0x\n", status);
    if (status & JackServerFailed) {
        fprintf (stderr, "Unable to connect to JACK server\n");
    }
    exit (EXIT_FAILURE);
}
if (status & JackServerStarted) {
    fprintf (stderr, "JACK server started\n");
}
if (status & JackNameNotUnique) {
    client_name = jack_get_client_name(client);
    fprintf (stderr, "unique name `%s' assigned\n", client_name);
}

jack_set_process_callback (client, process_samples, 0);

jack_on_shutdown (client, jack_shutdown, 0);

/* get current sample rate */

    sample_rate = jack_get_sample_rate (client);

    printf ("engine sample rate: %d\n", sample_rate);

/* create two ports */

inPort = jack_port_register (client, "input",
                 JACK_DEFAULT_AUDIO_TYPE,
                 JackPortIsInput, 0);
outPort = jack_port_register (client, "output1",
                  JACK_DEFAULT_AUDIO_TYPE,
                  JackPortIsOutput, 0);


if ((inPort == NULL) || (outPort == NULL)) {
    fprintf(stderr, "no more JACK ports available\n");
    exit (EXIT_FAILURE);
}

/* Tell the JACK server that we are ready to begin.
 * Our process_samples() callback will start running now.
 */
```

```
    if (jack_activate (client)) {
        fprintf (stderr, "cannot activate client");
        exit (EXIT_FAILURE);
    }

    ports = jack_get_ports (client, NULL, NULL,
                JackPortIsPhysical|JackPortIsOutput);
    if (ports == NULL) {
        fprintf(stderr, "no physical capture ports\n");
        exit (EXIT_FAILURE);
    }

    if (jack_connect (client, ports[0], jack_port_name (inPort))) {
        fprintf (stderr, "cannot connect input ports\n");
    }

    free (ports);

    ports = jack_get_ports (client, NULL, NULL,
                JackPortIsPhysical|JackPortIsInput);
    if (ports == NULL) {
        fprintf(stderr, "no physical playback ports\n");
        exit (EXIT_FAILURE);
    }

    if (jack_connect (client, jack_port_name (outPort), ports[0])) {
        fprintf (stderr, "cannot connect output port 1\n");
    }

    free (ports);

    sleep(1);     /* delay is to let other messages be output before ours */
    fprintf(stderr,"Running ... press CTRL-C to exit ...  \n");
    while (1) {
        printf("Enter desired frequency shift in Hz: ");

        if(scanf("%f", &frequencyShiftInput));
        delTheta = 2*M_PI*frequencyShiftInput/sample_rate;
    }

    jack_client_close (client);
    exit (EXIT_SUCCESS);
}
```