ECE 4020

Lab Part 2 Report

Ryan Colon

02.06.22

Introduction

Contained in this report is the documentation of a general-purpose computer implementation for an FIR low pass filter designed and evaluated in MATLAB in a previous report [i]. To implement this filter on a personal computer, a program to implement filters generally was written in C using the jack audio connection library to take a hardware input signal and filter it to a hardware output. A gnu-Linux pc was used to run the program and various lab equipment was used to evaluate the performance. In the future, this program and filter will be implemented on a dedicated digital signal processing chip.

Program Description

For the program to work, a finite maximum number of coefficients needs to be specified in the code before it is compiled, in this case it was set to 200. When the program is run it expects a command line argument for the name of the coefficients file in the "Coefficients" sub directory of the executable. It expects the file to be of the format:

Line 1. String

Line 2. Coefficient 1

Line 3. Coefficient 2

Where the first line is an arbitrary string that will be printed to the screen when the file is read, followed by a list of newline-delimited floating-point numbers that represent the coefficients of the filter in order. If the max number of coefficients condition is met, the file exists in the sub directory, and it is of the correct format then the program will start up JACK and run its process sample loop. Imaged on the next page is a snippet of the code that reads the file and a screenshot of the program starting up proper.

```
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[56];
if(fgets(firstLine, 56, 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);
```

Figure 1. File Read Code Snippet

```
racolon42@PRSC437-D06:~/Documents$ ls
Coefficients  Filt.exe  Source
racolon42@PRSC437-D06:~/Documents$ ls Coefficients/
filter_test_HUGE_h.txt  filter_test_TTU_morse_code_h.txt  RyanColon_FIR_Coefficients_Rev4.txt
racolon42@PRSC437-D06:~/Documents$ ./Filt.exe RyanColon_FIR_Coefficients_Rev4.txt
Ryan Colon, FIR Filter, Attempt 4 (Optimizing)

Coefficient 0:          0.002355
Coefficient 1:          0.002206
Coefficient 2:          0.002801
Coefficient 3:          0.003122
Coefficient 4:          0.003043
Coefficient 5:          0.002503
Coefficient 6:          0.001548
Coefficient 7:          0.000321
Coefficient 8:          -0.000961
Coefficient 9:          -0.002034
Coefficient 10:         -0.002667
Coefficient 11:         -0.002704
Coefficient 12:         -0.002129
Coefficient 13:         -0.001066
Coefficient 14:         0.000229
Coefficient 15:         0.001429
Coefficient 16:         0.002211
Coefficient 17:         0.002339
Coefficient 18:         0.001736
Coefficient 19:         0.000513
Coefficient 20:         -0.001046
Coefficient 21:         -0.002542
Coefficient 22:         -0.003565
Coefficient 23:         -0.003801
Coefficient 24:         -0.003122
Coefficient 25:         -0.001639
Coefficient 26:         0.000313
Coefficient 27:         0.002243
Coefficient 28:         0.003629
Coefficient 29:         0.004053
Coefficient 30:         0.003322
Coefficient 31:         0.001543
Coefficient 32:         -0.000885
Coefficient 33:         -0.003356
Coefficient 34:         -0.005203
Coefficient 35:         -0.005875
Coefficient 36:         -0.005095
Coefficient 37:         -0.002954
Coefficient 38:         0.000078
Coefficient 39:         0.003254
Coefficient 40:         0.005724
Coefficient 41:         0.006762
Coefficient 42:         0.005969
Coefficient 43:         0.003407
Coefficient 44:         -0.000375
Coefficient 45:         -0.004454
Coefficient 46:         -0.007748
Coefficient 47:         -0.009294
Coefficient 48:         -0.008520
Coefficient 49:         -0.005430
```

Figure 2. Program Beginning Properly

In the process sample loop, it will take a sample from the hardware input and place it in its correct spot in history in a sample buffer. Then the program will calculate the convolution of the sample buffer and coefficients list and will output that to the hardware output. If enabled, the program will also output a passthrough of the input signal to another hardware output (this can be disabled/enabled while the program is operating). Imaged below is the code implementation of this algorithm:

```
int process_samples (jack_nframes_t nframes, void *arg)
{
    jack_default_audio_sample_t *in;
    jack_default_audio_sample_t* out1;
    jack_default_audio_sample_t* out2;

    static float pipeline[maxCoeff] = {0.0};

    float y;
    float x;

/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/

    in = jack_port_get_buffer (input_port, nframes);
    out1 = jack_port_get_buffer (outputPort1, nframes);
    out2 = jack_port_get_buffer (outputPort2, nframes);

    for (int i=0; i < nframes ;i++) {
/***** REPLACE NEXT LINE WITH YOUR ALGORITHM *****/
/***** REPLACE NEXT LINE WITH YOUR ALGORITHM *****/
/***** REPLACE NEXT LINE WITH YOUR ALGORITHM *****/
        x = in[i];

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

        pipeline[0] = x;
        y = 0;

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

        out1[i] = y;
        out2[i] = (choise == 'y' || choise == 'Y') ? x : 0;
    }

    return 0;
}
```

Figure 3. Sample Processing Code Snippet

Test Procedure

List of equipment used:

- Arbitrary Function Generator
- Oscilloscope
- gnu-Linux PC
- Network Analyzer
- Resistive Power Splitter
- 6 dB Attenuator

To test the overall program was working, a Linux PC's microphone input was hooked up to an arbitrary function generator, and its stereo output was hooked up to two channels of an oscilloscope. This setup is imaged in the block diagram below:

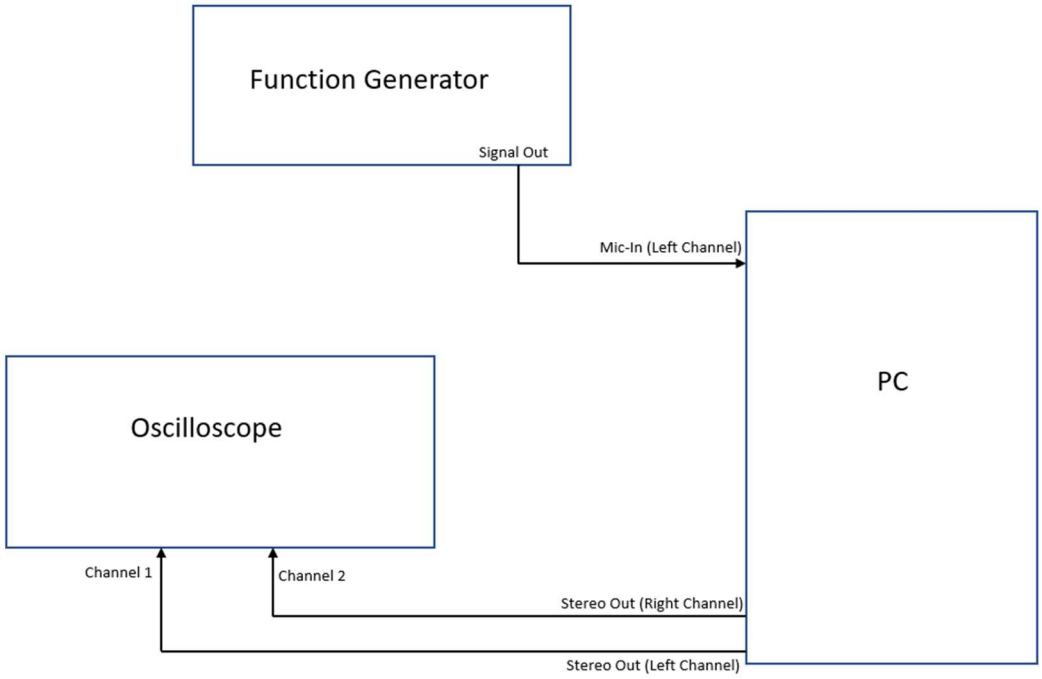

Figure 4. Function Generator - Oscilloscope Test Setup

The function generator was set to a 4 kHz sine wave so the filter and passthrough channels could be easily differentiated on the o-scope, and then the program was run with the filter designed in [i]. Imaged on the next page was the oscilloscope output as the program was running:

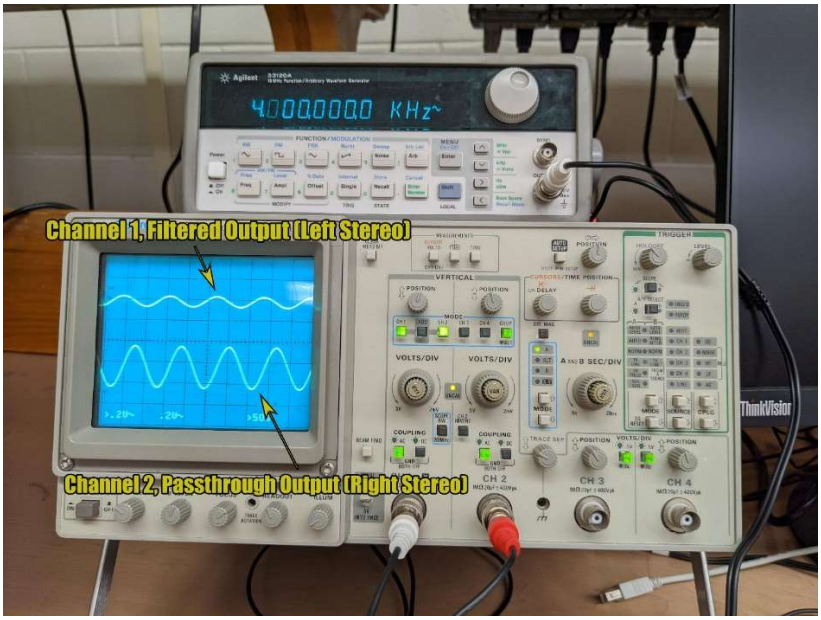

Figure 5. Working Program Measurement

Since the channel 1 signal was being damped at a frequency inside the transition band, it indicated that the program was working as intended. To test the generality of the program at implementing filters, this same test was conducted using a second filter file and checking to make sure the output was correct for an impulse-like input.

To generate the impulse-like input, the function generator was set to produce a 20% duty cycle square wave at 1 kHz, with the burst mode toggled on. A filter file, “filter_test_TTU_morse_code_h.txt”, was created to have an impulse response shaped like TTU’s morse code signal and was used as the filter for this test. Running the program under these conditions produced the following output:

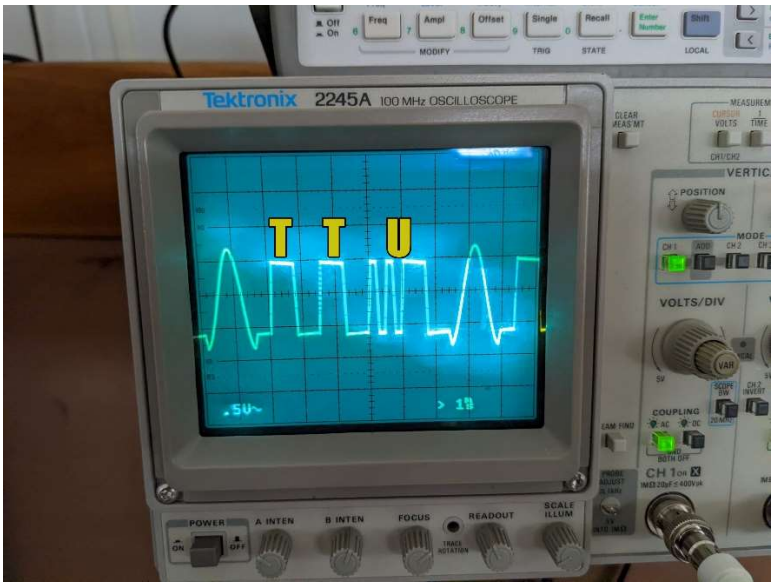

Figure 6. TTU Morse Code Test Filter Output

The output is as expected so the filter program could implement any filter. However, there is a limit in the code for how many coefficients that will be allowed by the program. If exceeded, the program will exit gracefully informing the user of the issue. To check this feature was working, a coefficient file, “filter_test_HUGE_h.txt”, was created and passed through the program. Imaged below is the program output:

```
racolon42@PRSC437-D06:~/Documents$ ls Coefficients/
filter_test_HUGE_h.txt  filter_test_TTU_morse_code_h.txt  RyanColon_FIR_Coefficients_Rev4.txt
racolon42@PRSC437-D06:~/Documents$ ./Filt.exe filter_test_HUGE_h.txt
HUGE test impulse response for FIR filter

Too many coefficients, max number of coefficients:       200
Exitting Program.
racolon42@PRSC437-D06:~/Documents$
```

Figure 7. Output of Program When Overloaded With Coefficients

These tests combined conclude the program is working as intended. Now that program is sure to be working properly, the filter in [i] can be tested to see if it meets spec. The specifications as listed are: 1-1.5 dB gain in the passband (0 to 3.7 kHz) and -50 dB gain in the stopband (> 4.3 kHz). To show the filter meets these specifications, a network analyzer, hooked up to a 3-way resistive power splitter and 6 dB attenuator, was calibrated and hooked up to PC. A block diagram of the connection setup is shown below:

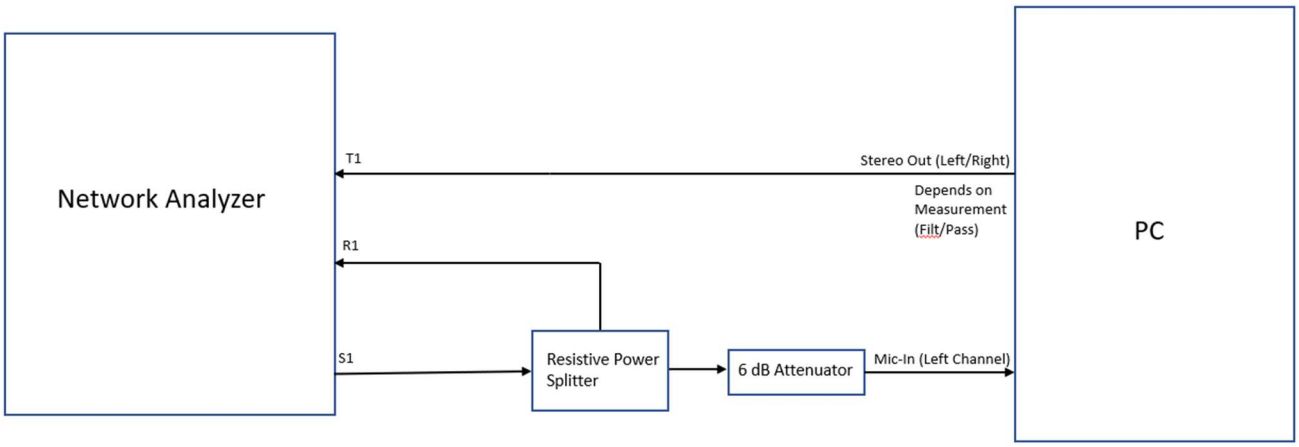

Figure 8. Network Analyzer Test Setup

To make sure the scope was calibrated properly, a measurement of the pass-through line up to the Nyquist frequency (24 kHz) was taken, and it produced the following plot (see next page):

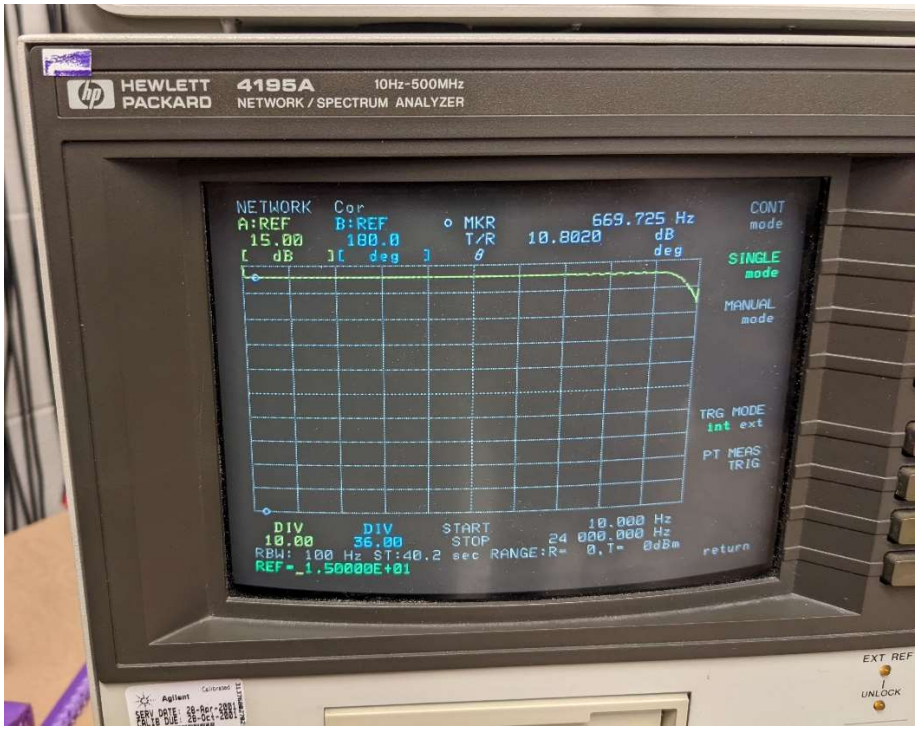

Figure 9. Measurement of Passthrough from 10 Hz to 24 kHz

This is what we expect to see if the analyzer is calibrated properly, therefore it is working. There is some odd behavior at the very low and high ends, but this is because of the network analyzer setup and characteristics of the wires being used. Taking a measurement of the filter over this same range produces the following plot:

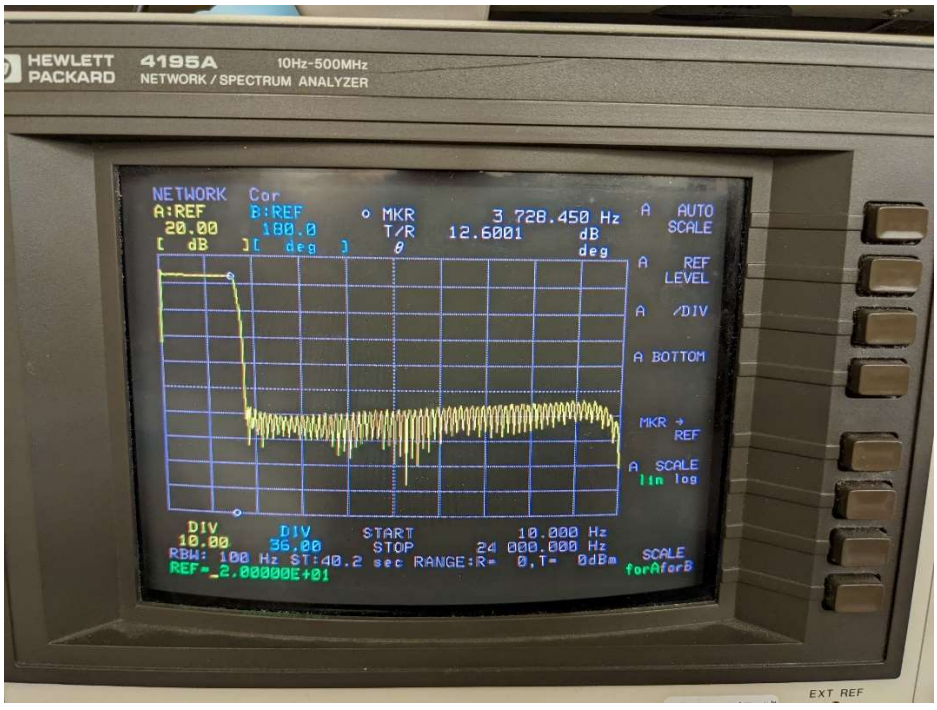

Figure 10. Measurement of Filter from 10 Hz to 24 kHz

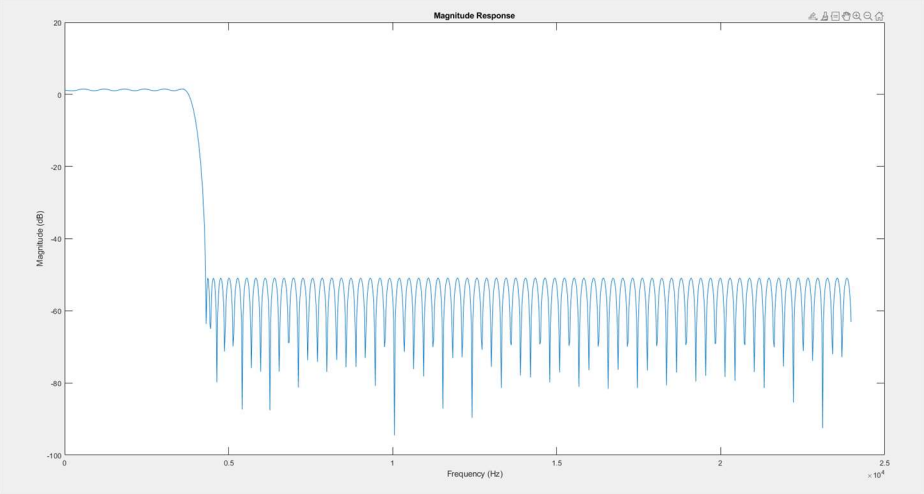

Figure 11. Magnitude-Frequency Response Plot in Matlab

When compared to the frequency response plot generated in Matlab in [i] it can be seen that the filter closely matches. However, as we go up in frequency there is an upward drift in the frequency response plot not present in the Matlab plot. This is again likely due to the setup of the soundcard and characteristics of the network analyzer setup. Regardless, it can be seen by the number of divisions in between the pass and stop bands that the filter still meets spec in the stopband. However, a closer measurement of the passband needs to be taken to ensure it still meets spec. The range on the network analyzer was adjusted to 10 Hz-3.8 kHz and a measurement of the passthrough was taken again to ensure calibration correction was still working:

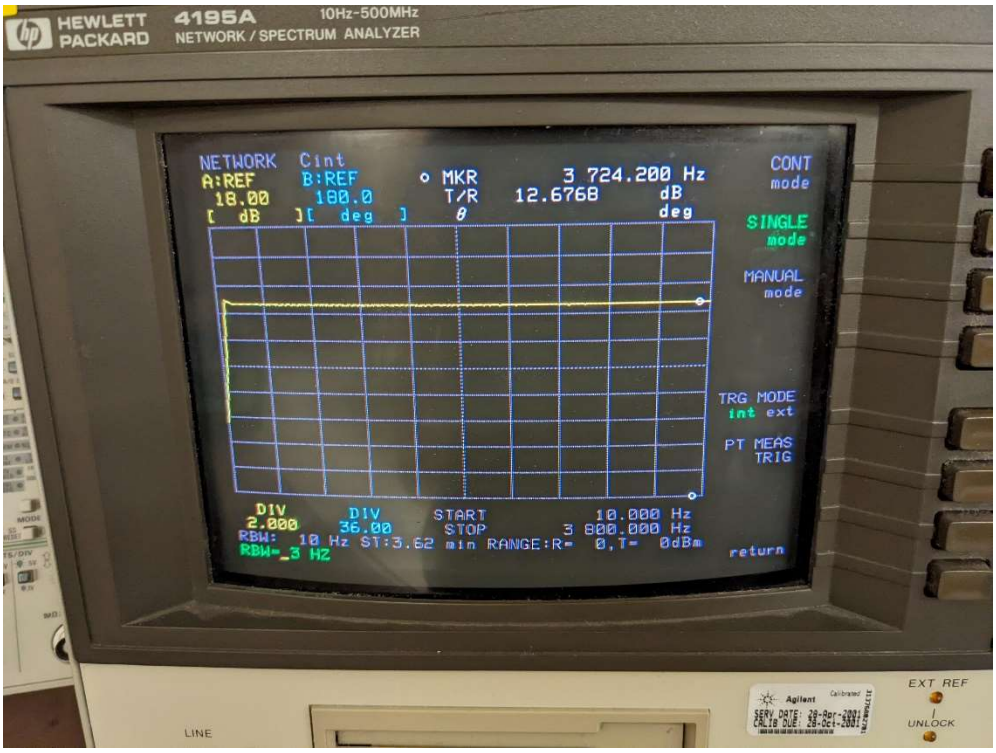

Figure 12. Measurement of Passthrough from 10 Hz to 3.8 kHz

The passthrough measurement appeared as it should, so a measurement of the filter was taken:

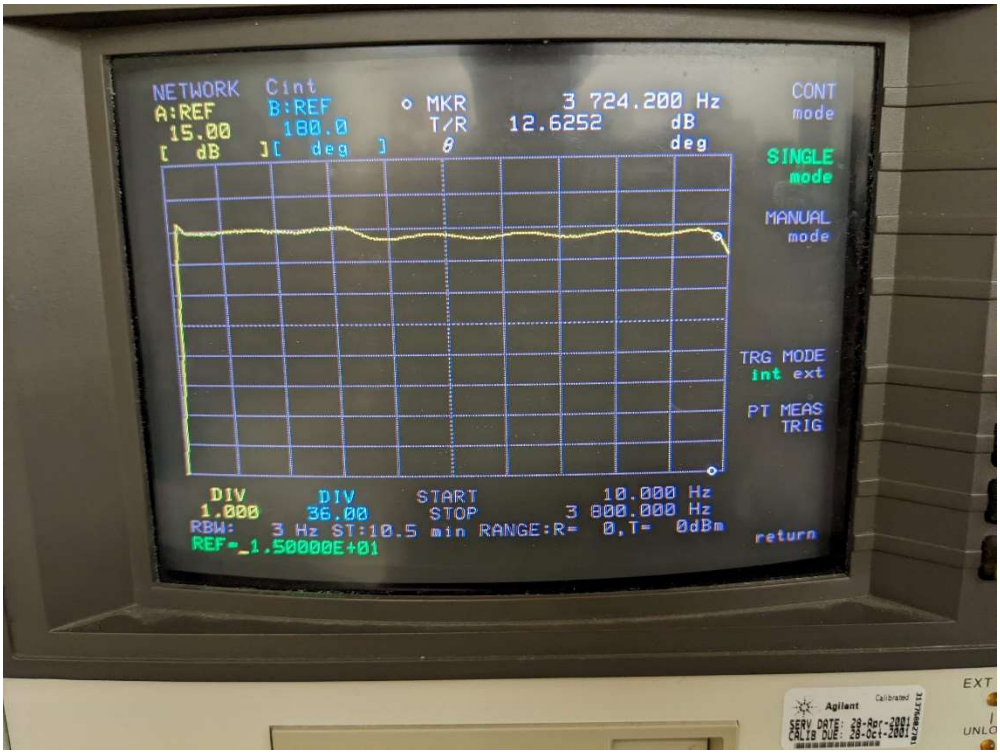

Figure 13. Measurement of Filter from 10 Hz to 3.8 kHz

When compared to figure 11, it is not a very close match, however when compared to the divisions around it in the network analyzer it can be seen that the ripple is within specifications. The center is at ~13 dB instead of 1.25 dB but this is because the system was not calibrated with the PC involved and so the volume slider on the soundcard shifted the entire system upward. The filter is still within relative specifications.

Conclusion

In this report an FIR filter designed and evaluated in Matlab in a previous report [i] was implemented onto a gnu-Linux PC using C programming and the JACK audio library. The program and filter went through several measurements to ensure proper working order and that specifications were met. It was found that the filter met specifications and the program worked as intended.

Citations

i. Colon, R., 2022. *ECE 4020 Lab Part 1 Report*.

Appendix

A. Complete program source code:

```
/* Note: this is an example program and is heavily commented - you should remove
or shorten some of the comments */

/** @file simple_client.c
 *
 * @brief This simple client demonstrates the most basic features of JACK
 * as they would be used by many applications.
 *
 * This is a modified version of the original simple_client.c program.
 * The original was downloaded on 2012-01-14 from:
 *   http://trac.jackaudio.org/wiki/WalkThrough/Dev/SimpleAudioClient
 *   http://trac.jackaudio.org/browser/trunk/jack/example-clients/simple_client.c
 *   (previous versions/downloads: Oct. 2009)
 *
 * This program initializes the sound card and copies the audio samples from the
input to the output.
 * It can be modified to create a filter or other DSP application which uses the
sound card.
 * For most applications the only changes needed are in the marked areas; the
rest of the program
 * should not need to be modified.
 * This version is monophonic. The input is on the left channel and output is on
either the
 * left channel or both channels, depending on the sound card driver
configuration.
 *
 * On Linux using gcc compile with the '-ljack' option to link with the jack
library.
 * If you use math functions, e.g. cos, the '-lm' option is also needed to link
with the math library.
 *   E.g., gcc -Wall myprog.c -ljack -lm -o myprog
 * The -Wall enables most warnings and the -o specifies the executable file name.
 *
 * Updated: Jan. 2022
 */

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

```c
#include <jack/jack.h>

#define maxCoeff 200

jack_port_t *input_port;
jack_port_t* outputPort1;
jack_port_t* outputPort2;
jack_client_t* client;

/***** INSERT GLOBAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT GLOBAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT GLOBAL VARIABLE DECLARATIONS HERE *****/
int numCoeff = 0;
float coeffArray[maxCoeff];
char choise;

/**
 * The process callback for this JACK application is called in a
 * special realtime thread once for each audio cycle.
 *
 * This client does nothing more than copy data from its input
 * port to its output port. It will exit when stopped by
 * the user (e.g. using Ctrl-C on a unix-ish operating system)
 */
int process_samples (jack_nframes_t nframes, void *arg)
{
    jack_default_audio_sample_t *in;
    jack_default_audio_sample_t* out1;
    jack_default_audio_sample_t* out2;

    static float pipeline[maxCoeff] = {0.0};

    float y;
    float x;

/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/

    in = jack_port_get_buffer (input_port, nframes);
    out1 = jack_port_get_buffer (outputPort1, nframes);
    out2 = jack_port_get_buffer (outputPort2, nframes);

    for (int i=0; i < nframes ;i++) {
/***** REPLACE NEXT LINE WITH YOUR ALGORITHM *****/
```

```
/***** REPLACE NEXT LINE WITH YOUR ALGORITHM *****/
/***** REPLACE NEXT LINE WITH YOUR ALGORITHM *****/
        x = in[i];

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

        pipeline[0] = x;
        y = 0;

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

        out1[i] = y;
        out2[i] = (choise == 'y' || choise == 'Y') ? x : 0;
    }

    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 = "simple-client";   /***** you can change the client
name *****/
    const char *server_name = NULL;
    jack_options_t options = JackNullOption;
    jack_status_t status;
```

```
    int sample_rate;

/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/
/***** INSERT LOCAL VARIABLE DECLARATIONS HERE *****/

    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[56];
    if(fgets(firstLine, 56, 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);
}

/* tell the JACK server to call `process_samples()' whenever
 * there is work to be done.
 */

jack_set_process_callback (client, process_samples, 0);

/* tell the JACK server to call `jack_shutdown()' if
 * it ever shuts down, either entirely, or if it
 * just decides to stop calling us.
 */

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 */

    input_port = jack_port_register (client, "input",
                     JACK_DEFAULT_AUDIO_TYPE,
                     JackPortIsInput, 0);
    outputPort1 = jack_port_register (client, "output1",
                      JACK_DEFAULT_AUDIO_TYPE,
                      JackPortIsOutput, 0);
    outputPort2 = jack_port_register (client, "output2",
                      JACK_DEFAULT_AUDIO_TYPE,
                      JackPortIsOutput, 0);

    if ((input_port == NULL) || (outputPort1 == NULL) || (outputPort2 == NULL)) {
        fprintf(stderr, "no more JACK ports available\n");
        exit (EXIT_FAILURE);
    }

/***** INSERT INITIALIZATION CODE HERE *****/
/***** INSERT INITIALIZATION CODE HERE *****/
/***** INSERT INITIALIZATION CODE HERE *****/

    /* 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);
    }

    /* Connect the ports.  You can't do this before the client is
     * activated, because we can't make connections to clients
     * that aren't running.  Note the confusing (but necessary)
     * orientation of the driver backend ports: playback ports are
     * "input" to the backend, and capture ports are "output" from
     * it.
     */

    ports = jack_get_ports (client, NULL, NULL,
                JackPortIsPhysical|JackPortIsOutput);
    if (ports == NULL) {
```

```
    fprintf(stderr, "no physical capture ports\n");
    exit (EXIT_FAILURE);
}

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

/* Connect the ports.  You can't do this before the client is
 * activated, because we can't make connections to clients
 * that aren't running.  Note the confusing (but necessary)
 * orientation of the driver backend ports: playback ports are
 * "input" to the backend, and capture ports are "output" from
 * it.
 */

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 (input_port))) {
    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 (outputPort1), ports[0])) {
    fprintf (stderr, "cannot connect output port 1\n");
}

if (jack_connect (client, jack_port_name (outputPort2), ports[1])) {
fprintf (stderr, "cannot connect output port 2\n");
}
```

```
    free (ports);

    /* it is now running....
         * do whatever needs to be done (if anything) while it is running.
         */

/***** (OPTIONAL) ADD TO OR REPLACE THE LINES BELOW WITH OTHER ACTIONS TO DO
WHILE ALGORITHM IS RUNNING *****/
/***** (OPTIONAL) ADD TO OR REPLACE THE LINES BELOW WITH OTHER ACTIONS TO DO
WHILE ALGORITHM IS RUNNING *****/
/***** (OPTIONAL) ADD TO OR REPLACE THE LINES BELOW WITH OTHER ACTIONS TO DO
WHILE ALGORITHM IS RUNNING *****/

    /* Nothing to do in main program ... wait until stopped by the user.
     * Make a little spinning thing to show the program is running.
     * Output is to stderr, avoiding line buffering, to make this work.
     */

    sleep(1);     /* delay is to let other messages be output before ours */
    fprintf(stderr,"Running ... press CTRL-C to exit ...  \n");
    while (1) {
        /*fprintf(stderr,"\b\\");  \b is the backspace character
        sleep(1);
        fprintf(stderr,"\b|");
        sleep(1);
        fprintf(stderr,"\b/");
        sleep(1);
        fprintf(stderr,"\b-");
        sleep(1);*/

        printf("Enter y/n to enable to disable passthrough.\n");

        if(scanf("%s", &choise));

    }

    /* This is may or may not be reached, depending on what is directly before
it,
     * but if the program had some other way to exit besides being killed,
     * they would be important to call.
     */

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