WEBER STATE UNIVERSITY
Engineering, Applied Science & Technology

DEPARTMENT OF
ELECTRICAL & COMPUTER
ENGINEERING

ECE 5210

Digital Signal Processing

Lab Manual

Author
Eric Gibbons

Version 4.0.0, Spring 2026

January 6, 2026

Contents

Acknowledgements

With this lab, we are standing on the shoulders of giants. I am not the strongest embedded systems programmer, so I want to acknowledge those who put out the online resources to make some sense of the audio I/O on this board.

I want to thank Steve Anderson at Bootladder Engineering for his publicly available codebase, clear YouTube tutorial, and willingness to engage over email to answer further questions.

Chapter 0

Introduction

In this lab, students derive algorithms from signal processing theory and map the algorithms to embedded software. They learn design flows from application theory, algorithm design, simulation in Python, and embedded software implementation in C. Students explore and quantify design trade-offs in signal quality versus implementation for various algorithms at various stages in the design flow.

0.1 Goals

The goal of the lab component of the course is threefold:

Implementing the algorithms We will implement DSP algorithms for tasks such as FIR and IIR filters, which are building blocks of many different systems.

Understanding the theory The DSP theory covered elsewhere in the course can be incredibly dense. Completing the design flow by implementing the algorithms provides another angle for comprehension.

Understanding the nuances We make assumptions in developing the theory that is often violated in practice. For example, floating point arithmetic introduces numerical errors. The lab exercises allow us to test and correct our assumptions when necessary.

0.2 Hardware and software

We will use the STM32F769 Discovery kit as a development kit. This is not an embedded systems course, so many of the technical details of what is happening "under the hood" will be largely ignored as the motivation for the lab is to explore the implementation of the algorithms presented in class. The user manual can be downloaded from the ST website. Ensure the board you check out is accompanied by a micro USB cable to connect to your laptop and two BNC to TS cables to connect with the oscilloscope and function generator.

0.2.1 Python environment

We will also use Python extensively in the course to prototype our algorithms. You are strongly encouraged to download Anaconda and use Python 3.12. Your Python routines will be tested

against Python 3.12 code (the same is true with your homework).

Windows users

Please download and install Ubuntu with the Windows Subsystem for Linux (WSL). You can easily download it through the Windows store. You will likely need to install basic development tools such as `GCC`, `make`, etc. Ensure that Python is using 3.12 (which is what comes standard with Ubuntu). You can set all of this up by using a virtual environment in a bash shell. One easy way is to set up a virtual environment in your Bash shell.

If you are a getting an error informing the shell cannot find Python, install `python-is-python3`, which creates a symbolic link to the Python 3 installation.

MacOS users

If you are using MacOS, you should have a UNIX terminal via the Terminal app. You will need to compile and test your code via the command line, and you can access a UNIX shell in that app. I would strongly suggest using the Anaconda distribution of Python to run your code. You can install Anaconda via a graphical interface as described here. Ensure that your PATH uses the Anaconda version of Python and that the Python version is 3.12. One easy way is to set up a virtual environment in your Bash shell. A comprehensive set of instructions to do this can be found here.

0.2.2 STM32CubeIDE installation

We will compile the code and flash the microprocessor using the Cube IDE software. You can download it free on the ST website here. This software is cross-platform so that you can use it natively on Windows, MacOS, and Linux computers.

When you install the software, please also install the accompanying drivers. The software is safe, so please permit it to perform the installation.

When you launch the program the first time, it will ask you to select a workspace. Please select a location that makes sense to you. This is where your projects for the lab assignments will live.

0.2.3 Starter code

This is not an embedded systems course, so you will not be expected to set up the board to perform the DSP routines. I will provide you with starter code each week, and you will need to edit only one source and one header file. If you are curious about the code base and firmware for the board, you are welcome to poke through the rest of the code, but I will strongly advise against editing anything in the chance that you mess something up.

You will download the starter code through GitHub. Each week, the starter code will roughly follow this directory structure:

```
ece5210_lab01
└── Core
    ├── Inc
    │   └── ece5210.h
    └── Src
```

```
    ├── ece5210.c
    └── Startup/
├── Drivers/
    ├── BSP/
    ├── CMSIS/
    └── STM32F7xx_HAL_Driver/
└── test/
    ├── Makefile
    ├── some_lab_python.py
    ├── sandbox.c
    └── test_utils/
        ├── python_files_for_testing.py
        ├── utest.h
        └── unittests.c
```

The signal processing routines will be implemented in the `ece5210.c` file. The `test/` folder includes all the unit tests to check your work. In this folder, the `sandbox.c` file is for debugging your code in `ece5210.c`. It is not used in the embedded processor and is a standalone executable. Feel free to modify as needed to send dummy signals through your DSP routines in a more controlled environment. The `Makefile` in this folder will build the `sandbox` and `test` executables. The `test` executable will run unit tests for both your C routines and your Python code to ensure both will run. I will use these tests to grade your work.

0.2.4 Basic Cube IDE usage

The starter code contains a project that you can import into the Cube IDE. Launch the Cube IDE and import the project by clicking `File->Import`. Select `Existing Projects into Workspace`. Use the `Select root directory` option and browse to the project directory. Select `Finish`. Your project should show up in the `Project Explorer`. You can browse through your code in the `Project Explorer` and use the IDE as a text editor[1].

Once the project is imported, select the project as the active project by double-clicking it. Connect the board to your computer using a micro USB cable. The board has two micro USB ports. Use the labeled `CN15 STLINK`. To compile and deploy the code, push the "play" icon at the top toolbar. This will both build the project and flash the board. The LEDs on the board will flash momentarily and then go solid. At this point, your project is running. The IDE has extensive debugging capabilities. Select the "bug" icon next to the "play" icon to debug your code. This will compile the code and flash the device. However, execution will be in "debug mode," allowing you to step through your code if you want.

0.2.5 C testing

Each of the programs you will need to write will come with a suite of unit tests to make sure you are on the right path (and you will be graded on whether you can pass these tests). In each of the project repositories there is a Makefile to build the tests. Make sure you can run these tests. If you are using Linux or MacOS, this should be trivial since GNU Make is installed on these systems

[1] You can also use an external text editor to write your code. I prefer emacs in a terminal or VS Code, but you are welcome to use whatever flavor of editing works for you. In VS Code there is an extension for STM32 boards, but I have not explored it. You are welcome to play around with it.

by default and is available in the terminal. If you are using Windows, you will need to install the Windows Subsystem for Linux (WSL) and install Ubuntu. You can then install GNU Make and run the tests in the Ubuntu terminal. You can also use the Windows version of GNU Make, but I have not tested it. You can download it here. There are other versions of Make that will work on Windows, but I am not as familiar with them.

0.2.6 Audio pass through

The starter code is set up to sample analog data only through the line-in headphone jack through the left channel. The data is sampled at 48 kHz and is provided as signed 16-bit integer data. This data is set through the `process_sample()` function in the `ece5210.c` file.

```
#include "ece5210.h"

#include <stdio.h>

int16_t process_sample(int16_t sample_in)
{
    int16_t sample_out = 0;
    float sample_in_f = (float)sample_in;

    // This is just simple pass through, if you want to
    // do any signal processing, start editing here...
    float sample_out_f = sample_in_f;

    // Convert back to int16_t
    sample_out = (int16_t)sample_out_f;
    return sample_out;
}
```

Unmodified, this function will simply convert the input signal from a 16-bit int to a 32-bit float and back to a 16-bit int. This default behavior of not modifying samples is called *pass through*. All signal processing routines must be implemented in this routine. You can use additional functions to be called in `process_sample()`, but the function declaration and return type cannot be modified.

For those curious, this function is called in `Src/myaudio.c` file in the `CopySampleBuffer()` function where the input buffer of 16-bit ints is copied to an output buffer of 16-bit ints.

```
static void CopySampleBuffer(int16_t *dst, int16_t *src,
                             uint32_t num_samples)
{
    for (uint32_t i = 0; i < num_samples; i++)
    {
        /*
          the process_sample() function is defined in
          ece5210.c

          the default is for a straight passthrough, but
          you can modify it to do some signal processing
```

```
        */
        dst[i] = process_sample(src[i]);
    }
}
```

The data is then converted to an output form (i.e., inserted into the output DMA buffer) and returned through the line-out output headphone jack. Data is returned through the headphone jack's left and right channels. The default behavior is to return processed data (i.e., data processed in the `process_data()` function) in the left channel. The right channel simply passes through data that has not been processed. You can see this behavior in the `Src/myaudio.c` file in the `ConvertSampleBufferToDMABuffer()` function.

```
static void ConvertSampleBufferToDMABuffer(
    int16_t *sampleInBuffer,
    int16_t *processOutBuffer,
    uint8_t *dmaBuffer,
    uint32_t num_samples)
{
    for (uint32_t i = 0; i < num_samples; i++)
    {
        // samples are spaced 8 bytes apart
        int16_t *p = (int16_t *) &dmaBuffer[i*8];
        *p = processOutBuffer[i]; // left channel

#ifdef PASSTHROUGH_RIGHT
        *(p+2) = sampleInBuffer[i]; // right channel
#else
        *(p+2) = processOutBuffer[i];
#endif
    }
}
```

In this function, a `#ifdef` feature flag can be turned on and off in the `Inc/ece5210.h`. If `PASSTHROUGH_RIGHT` is not defined, then the right channel will also return processed data like the left channel.

0.2.7 Hardware

We will use the function generators for much of the lab to provide us with an analog input signal. To get a signal from the generator to the board, connect a provided BNC to RCA adapter and connect the left RCA connector in a provided headphone/RCA cable. Plug the headphone terminal into the line-in jack on the board. For the output, connect the headphone/RCA cable to the board and connect BNC to RCA adapters to both the right and left RCA terminals on the cable. Connect both to the oscilloscope to visualize the signal.

0.2.8 Provided hardware at checkout

You will need to check out the equipment you will use throughout the semester. If you fail to return the equipment at the end of the semester, you will automatically receive an "E" grade. You

will be provided with the following:

- STM32F769I-DISCO kit
- micro USB cable
- 2× 3.5 mm TRS (stereo) to RCA breakout cables
- 3× RCA to BNC adapters
- TRS to TRS (stereo) cable

0.3 Lab techniques

0.3.1 Saving waveforms

We will be using the Keysight equipment in NB 104/112. Throughout the lab, we will need to record the data from the oscilloscope. You will be expected to make plots of the data you gather. You must save this data to a USB drive and plot it in Python. To do this, make sure your thumb drive is formatted as FAT. Plug it into the oscilloscope. The oscilloscope should recognize it by a spinning wheel at the top of the display and will eventually flash the drive's name. You can push the `Save/Recall` button. Select `Save`. You can change the save location to your drive by selecting the `Save to` option. You can also choose the channel source and the data format. When you have configured this, hit `Save`.

0.3.2 Frequency sweeps

We will repeatedly record our systems' frequency response in the lab. We could manually measure the phase and magnitude changes using a function generator and an oscilloscope, but that would be tedious. Our oscilloscope can automate this, saving you a lot of time. First, push the `Analyze` button on the oscilloscope. In the `Features` sub-menu, select `Frequency Response Analysis`. Select `Setup` to change the frequency sweep. Most of the time, we will be sweeping from some low frequency (say 10 Hz through Nyquist). Make sure you sample in frequency with enough points, which is also adjustable in the menu. The oscilloscope will sweep in a logarithmic fashion rather than a linear sweep. This works well enough for analog systems, but we tend to display the frequency linearly in digital systems. This should not be a major issue in this course, make sure that you sample with enough points to get a decent resolution in the system pass band.

Measuring the frequency response uses the oscilloscope's internal function generator. Ensure the system line input comes from the `Gen Out` terminal on your oscilloscope. The pass-through channel on your board should go into Input 1, and the processed signal should go into Input 2. You can change this behavior in the oscilloscope menus, but I recommend sticking with the traditional convention.

To save this data, insert your USB stick into the oscilloscope. In the `Save/Recall` menu, set the format to `Frequency Analysis Data (*.csv)`. Push the `Save to USB` button to save the data. You can read this `csv` data into Python and plot it using `matplotlib`.

0.3.3 Troubleshooting

Left and right channels

Typically, with RCA cables, the left is white, and red is the right. However, some of our cables are backward, with red as left and white as right. If you are in doubt, the tip of the TRS cable is the left channel. Check it against both RCAs using the lab multimeter to determine what you are working with. If your code does not match the expected behavior on the lab equipment, try swapping left/right channels. Once you have this sorted out, it might be a good idea to label your cables.

Managing multiple projects in the Cube IDE

Throughout the semester, you will have multiple projects in the `Project Explorer`. It is good practice to close the projects you are not actively working on. To do this, right-click on the project you are working on and select `Close unrelated projects`.

Chapter 1

Signal Distortion

Signal distortion is often considered bad, but it is often used in audio effects such as guitar distortion pedals and audio tube amplifiers. Historically, most of these effects come from malfunctioning or abused analog hardware. While you can buy various distortion pedals implemented in analog hardware today, it is much cheaper to implement comparable distortion using digital systems.

We are mostly interested in systems where some input $x[n]$ will equal the output $y[n]$. However, we might want to add nonlinear effects to make the signal—e.g., music—more interesting. In particular, in this exercise, we will implement soft-clipping, similar to an effect you might get from a tube amplifier. There are myriad versions of how to do this, but we will use one of the easier versions. The algorithm is pretty straightforward, where we just map some input sample $x[n]$ to some output sample $y[n]$ according to

$$y[n] = \begin{cases} -\frac{2}{3} & x[n] \leq -1 \\ x[n] - \frac{x[n]^3}{3} & -1 < x[n] < 1 \\ \frac{2}{3} & 1 < x[n] \end{cases}. \tag{1.1}$$

This input/output relationship is shown in Fig. 1.1.

Questions and tasks

- Is this system linear?
- Is this system time-invariant?
- Is this system causal?
- Is this system memoryless?
- Is this system stable?

1.1 STM32F769I implementation

In the `ece5210.c` file, implement this system. You will need to write a function with the following declaration:

```
/**
```

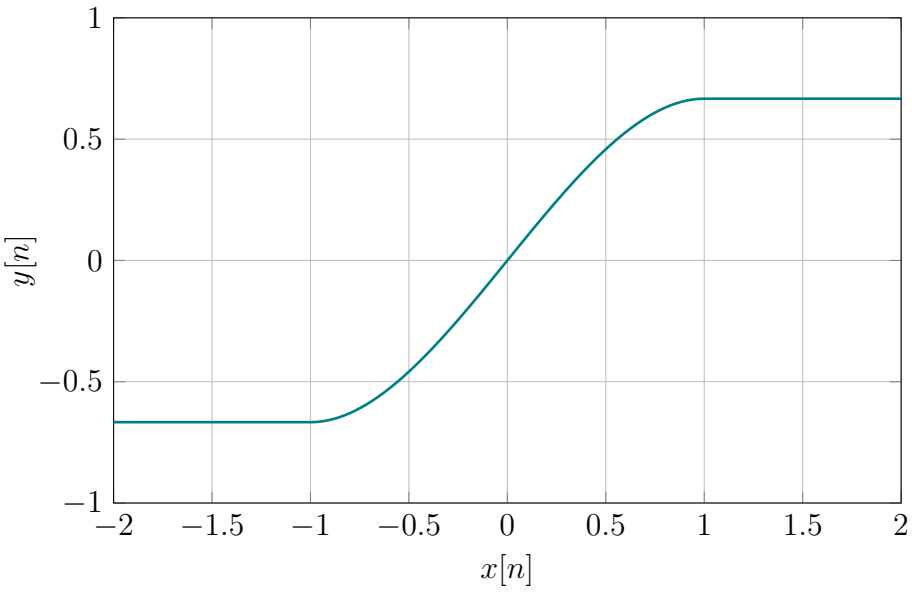

Figure 1.1: Soft-clipping as defined in Eq. (1.1).

```
* @brief Implementation of soft-clipping as defined in the lab
   manual
*
* @param[in] insample The input sample x[n]
* @param[in] limit The clippling limit
* @return The output sample y[n]
*/
float distortion(float insample, float limit);
```

The `insample` argument is the input sample $x[n]$. The `limit` argument sets the clipping limit. In Eq. (1.1) this is just 1. However, in our systems, we will want something more flexible, so we will set up the function variable `limit`, which means you must modify Eq. (1.1) a bit to include this flexibility. (Don't overthink this!)

You will call this function in the `process_sample()` function in `ece5210.c`. This should look like:

```
float sample_out_f = distortion(sample_in_f, LIMIT);
```

In your `ece5210.c` file, I would define a constant `LIMIT`, so it is easily changeable rather than just hard-coding it when you call the `distortion()` function. You can play with this value to see what looks decent on an oscilloscope, but 4500 is a reasonable place to start.

1.1.1 Compiling

You have a couple of options to compile your code. The first approach is to compile your code via the STM32CubeIDE. This will allow you to compile your code and flash the board.

The second option is to compile your code with the supplied `Makefile` in the `test/` directory. In `test/` there is a `sandbox.c` file, which contains a driver `main()` function that you can use to test your code using simple cases to see if your code in `ece5210.c` is doing what you think it is.

Compiling this code will look something like this:

```
$ make
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3 -I
    ../Core/Inc
gcc -o sandbox sandbox.o ece5210.o -lm
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../Core
   /Inc
gcc -o test unittests.o ece5210.o -lm
```

The code in `sandbox.c` is entirely independent of the STM32F769 code and reduces your work to a problem similar to what you saw in ECE 1400 or ECE 3210. This is a good place to start if you need to spend time debugging your code. Once you are comfortable with your work, I recommend compiling your code for the board via STM32CubeIDE.

1.1.2 Testing

You are given a unit test to ensure that your `distortion()`functionsg as it should. When you run your `Makefile`, you create an executable `test`. If you run this executable, it will check that your function works as you hope it will.

```
$ ./test
[==========] Running 1 test cases.
[ RUN      ] ece5210_lab01.distort
[       OK ] ece5210_lab01.distort (32363710ns)
[==========] 1 test cases ran.
[  PASSED  ] 1 tests.
```

1.1.3 Validation

Set the clipping limit to 4500 in your code. Compile the code for the STM32 board and flash it. Attach your board to the function generator and attach the output channels to the oscilloscope. Generate a 100 mV (peak-to-peak), 1 kHz sinusoidal wave as an input. Measure the peak-to-peak amplitudes for the processed and pass-through channels. Increase the input voltage by 100 mV (peak-to-peak) increments and re-record the processed and pass-through voltages. Repeat until the output is at 500 mV. Plot this data with the pass-through voltage on the x-axis and the processed voltage values on the y-axis. Hopefully, these align well with what we roughly observe in Fig. 1.1.

Questions and tasks

- Plot the peak-to-peak voltage of the processed voltages against the pass-through voltages.
- Demonstrate the clipping on the oscilloscope to the instructor.

1.1.4 Listening test

Once you know your `distortion()` function is working correctly, compile your code for the STM32 board and flash it. Hook up the 3.5mm stereo cable to the input 3.5mm on the board and into

the headphone jack on your computer. Hook up a pair of headphones into your board. Fire up whatever audio streaming service of your choice. Choose some music and listen to the distortion. I would suggest something acoustic and straightforward to get the "electric guitar" effect. You might want to control the level of distortion in your signal by playing with the `LIMIT` constant value.

Questions and tasks

- What music did you use for this test?
- What value for `LIMIT` did you use?
- How did changing the value of `LIMIT` change what you heard?

1.2 Deliverables

d

1.2.1 Submission

Please fill out the Jupyter notebook `test/distortion.ipynb` answer each of the questions. Make sure you include your data files for your plots in your GitHub repository!

1.2.2 Grading

The lab assignment will have the following grade breakdown.

- `distortion()`: 60pts.
- Clipping plot: 25pts.
- Clipping demonstration: 20pts.
- Correct submission of Jupyter notebook: 5pts.

Chapter 2

Moving Average

The moving average filter is a basic discrete-time system we discussed in class. This system can be described by

$$y[n] = \frac{1}{M_1 + M_2 + 1} \sum_{k=-M_1}^{M_2} x[n-k]. \tag{2.1}$$

In practice, we will be working with causal systems, and we can also assume that the system starts averaging at the first sample. In other words, we can simplify this expression to

$$y[n] = \frac{1}{M_2 + 1} \sum_{k=0}^{M_2} x[n-k]. \tag{2.2}$$

In this exercise, we will implement this system on the STM32F769I boards. We will measure and benchmark its frequency response against the theoretical frequency response shown by the DTFT. We will use a 7-point moving average, so we will set $M_2 = 6$ which gives

$$y[n] = \frac{1}{7} \sum_{k=0}^{6} x[n-k]. \tag{2.3}$$

2.1 STM32F769I implementation

You can find the starter code for this lab in the provided link on Canvas. Import this code into the Cube IDE using the procedure described in Sec. 0.2.4. Before modifying the code, please try to build and run the project first. This should provide simple audio pass-through functionality.

In the `ece5210.c` file, implement a 7-sample moving average system. This system will require knowledge of prior samples, so we must store those each time the function is called. In C, if we declare an array *inside* the `process_sample()` function to store these previous samples, it would be stored on the stack and erased with each new function call. This would be problematic since we hope to use these values for each sample. To get around this, we can declare our variables as static global variables, which will exist during the entire program runtime. To do this, you will need to add the `static` identifier to the variable declaration, and the variables will need to be declared outside of the scope of any functions.

Once you have modified `process_sample()`, check that it works using the unit tests. To do this, compile your code using `Makefile` in the `test/` folder.

```
$ make
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../Core
   /Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3 -I
    ../Core/Inc
gcc -o test unittests.o ece5210.o -lm
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/Inc
gcc -o sandbox sandbox.o ece5210.o -lm
```

We can run our unit tests.

```
$ ./test
[==========] Running 2 test cases.
[ RUN      ] ece5210_lab01.moving_average
[       OK ] ece5210_lab01.moving_average (578667112ns)
...
```

Once you have passed your unit test, please build your project and flash the board in the Cube IDE. Generate a 300 Hz square wave as an input to the board. Record the pass-through and processed signals on the oscilloscope on a USB drive. Generate a plot of both channels in Python to include in your report. To save the data from the oscilloscope, see Sec. 0.3.1.

Questions and tasks

- Generate a plot of the oscilloscope's processed and pass-through signals.
- If we are sampling at 48 kHz, what is the Nyquist frequency?

2.2 Frequency analysis

In class, we discussed the Discrete-Time Fourier Transform (DTFT). This transformed is defined as

$$X\left(e^{j\omega}\right) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}.$$

Remember that the output is 2π-periodic, so we usually only need to compute this transform over a single period.

2.2.1 Phase shifts

As a side note, you might notice a phase shift between the right and left channels as you perform various frequency sweeps. This is because the samples are interleaved in the output buffer, so we see a slight delay in the right channel relative to the left channel. Perform a frequency sweep of a pass-through without any processing from $f = [500\,\text{Hz}, 23\,\text{kHz}]$. Record these phase shifts at each frequency. Make a plot of this phase relative to the frequency. Is this phase shift linear? Recall that a sinusoid with a phase shift can be represented as

$$\begin{aligned} x[n] &= \cos(\omega n + \phi) \\ &= \cos(\omega(n + n_d)). \end{aligned}$$

Therefore, we can determine the phase shift as

$$\phi = \omega n_d$$
$$\implies n_d = \frac{\phi}{\omega}.$$

The number of delayed samples is the slope of the phase line!

2.2.2 Python DTFT implementation

Implement the DTFT in Python in a file called `test/moving_average.py`. Your function should have the following definition.

```
def dtft(x, omega, n=None):
    """
    Compute the discrete-time Fourier transform (DTFT)
    according to

    X = \sum^{-\infty}_{n=-\infty} x[n] e^{-j*\omega*n}

    This is NOT a DFT.  The output should mimic a continuous
    function according to the input omega you use.

    Parameters
    ----------
    x : ndarray
        signal to compute the DTFT in the time domain
    omega : ndarray
        frequencies to compute the DTFT over
    n : ndarray (default None)
        associated index values for x if needed, if ommitted, n
        should be set to n = 0, 1, 2, 3,...,N-1

    Returns
    -------
    X : ndarray
        DTFT of the input signal with each element corresponding
        to a frequency as defined in omega

    """
```

Most of the time, we will deal with signals that are going to be causal, so the `n` array will default to $\{0, 1, 2, \ldots, N-1\}$ where N is the length of the signal x (or the array representation `x`). Unless `n` is specifically indicated, then it should default to this. There is a unit test for this function. You can call this unit test using the `test` executable. Of you can call the Python script (directly.

2.2.3 Moving average impulse response

We will want to determine the moving average system's DTFT, but we will need the impulse response $h[n]$. Remember that the impulse response to an LTI system is $h[n] = T\{\delta[n]\}$.

Questions and tasks

- What is this system's impulse response $h[n]$?

2.2.4 Plotting the DTFT

Now that we know $h[n]$, determine the system's DTFT $H\left(e^{j\omega}\right)$ analytically. Include this in your report. Now plot this expression. Plot the magnitude and phase on two separate plots from $\omega \in [0, \pi]$. Use your `dtft()` function to numerically compute the DTFT and plot the magnitude and phase on the same plots. Ideally, the lines should overlap completely.

Next, using your board and function generator, do a frequency sweep from $f \in [50\,\text{Hz}, 23.5\,\text{kHz}]$. Plot this data as scatter plots on your existing DTFT plots.

You should notice that the measured magnitude response matches the theoretical very closely. However, the measured phase might have a linear offset relative to the theoretical phase response. This is because the pass-through channel might be delayed relative to the processed channel and messes up the phase response computation. This is easy to correct if you want. Otherwise, please make a note of any phase discrepancies in your report and justify your reasoning.

Questions and tasks

- Make a plot of the system's frequency response. Plot the magnitude response and phase in two separate plots. Your plots should include the derived DTFT, the DTFT from your `dtft()` function, and the data recorded from the board. Make sure your two plots have legends and labels.
- How close do each of the plot align?
- Is there is a linear difference between the phase of the theoretical and measured data? If so, what is the slope of this line? What is the delay in samples between the two channels?

2.3 Deliverables

2.3.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots. Your code should be submitted to GitHub. Additionally, you must demonstrate your functioning board at the beginning of the lab section.

2.3.2 Grading

The lab assignment will have the following grade breakdown.

- `process_sample()`: 30pts.
- `dtft()`: 15pts.

- Frequency response plot: 20pts.
- Oscilloscope output plot: 10pts.
- Demo: 10pts.
- Write-up: 15pts.

Chapter 3

FIR system implementation

In this exercise, we will implement a Finite Impulse Response filter, a foundational DSP algorithm. In class, we learned that an FIR system will take the form

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M]. \tag{3.1}$$

This LTI system can be represented as a convolutional system

$$y[n] = h[n] * x[n]$$

where $h[n]$ is a finite impulse response as defined by the coefficients b_m as defined in Eq. (3.1)

$$h[n] = b_0 \delta[n] + b_1 \delta[n-1] + \cdots + b_M \delta[n-M].$$

We can take the z-transform

$$H(z) = b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}.$$

From this, we sketch the block diagram of the system $H(z)$ in Fig. 3.1. In this diagram, the z^{-1} blocks indicate a unit delay in the sample. According to the z-transform properties discussed in class

$$x[n-1] \Longleftrightarrow X(z) z^{-1}.$$

Notice that if there is a cascade of unit delay blocks, that would imply that the signal would be delayed by the number of delay blocks it passes through. In other words

$$\begin{aligned} x[n-M] \Longleftrightarrow & X(z) \left(z^{-1}\right)^M \\ = & X(z) z^{-M}. \end{aligned}$$

The system has memory, as we will need to use prior samples of the input $x[n]$ to compute the output $y[n]$.

3.1 C implementation

3.1.1 Generating filter coefficients

Typically, we will design and generate filter coefficients in Python and copy them to a global array in the embedded C code. However, we will want to be able to adjust the filter size quickly; we will

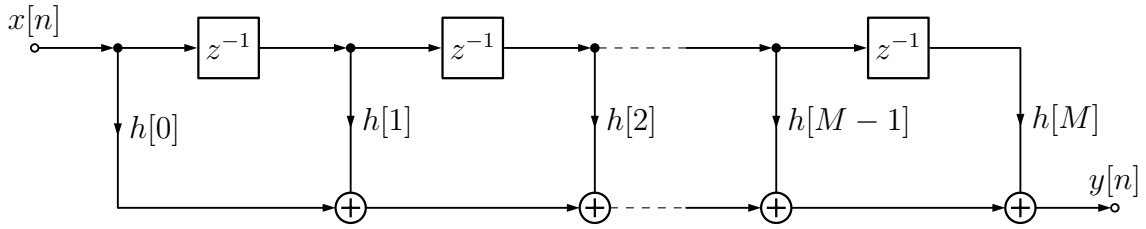

Figure 3.1: Block diagram of an FIR system. In this system, the z^{-1} block indicates a unit shift in the signal.

write a function to generate and populate a global array declared in `ece5210.c` with the computed filter coefficients. We will define our filter according to

$$h[n] = \frac{(0.8)^n(u[n] - u[n - M - 1])}{\sum_{n=-\infty}^{\infty}(0.8)^n(u[n] - u[n - M - 1])}$$

where M is the filter length. Note that this is normalized such that its energy is one. Write a function with the following function declaration that populates a global array throughout the program. I would first define the filter length near the top of `ece5210.c` and then declare the global array using this definition. We will vary the filter length, so changing the definition of the filter length will be more convenient than changing hard-coded lengths throughout the code. For example, in `ece5210.c` We could include the following code:

```
#define NUM_TAPS 15
...
float h[NUM_TAPS] = {0.f};
...
```

We can then populate `h[NUM_TAPS]` in a function with the following function declaration.

```
/**
Populates a global array with filter coefficients as
defined as

h[n] = 0.8^n (u[n] - u[n-M-1]) / (sum (0.8^n (u[n] - u[n-M-1])))

   * @param void
   * @return void
*/
void init_filter(void);
```

We will call this function outside `ece5210.c` inside the `myaudio.c` file. This will initialize the filter in a global array that you can use in `process_sample()` as an input for `fir()`. You will need to insert this at the top of the `audio_init()` function similar to the following:

```
void audio_init(void)
{
    /* intialize filter here */
    init_filter();
    ...
```

```
}
```

3.1.2 The FIR system

You will need to implement an FIR filter system in the `ece5210.c` file. The FIR filter declaration should look like the following:

```
/**
Implementation of an FIR filter system.

   * @param sample_in the current sample x[n] to be processed
   * @param *b array of filter coefficients b_0, b_1,...,b_M
   * @param len_b the length of b
   * @return the output sample y[n]
 */
float fir(float sample_in, float *b, uint16_t len_b);
```

One key detail with a real-time FIR system is how to store the state variables (i.e., the delayed samples of $x[n-1]$, $x[n-2]$, $x[n-M]$). The naive way is to set up a linear array and recopy the samples to the delayed positions in the buffer for each function call of `fir()`. This would also require that this array be declared a global array, so the values will not change for each function call (i.e., don't make this array an auto variable!). This approach will not work well on lower-powered microprocessors because these copies would be very onerous and take too much time. Traditionally, these DSP algorithms use circular buffers to avoid recopying the data. However, the STM32F769 microprocessors have sufficient cache onboard, and the compiler is quite good at optimizing your code. Using a linear buffer in this exercise, you should avoid significant penalties, which will be easier to implement. You are welcome to implement a circular buffer if you choose so long as it can outperform a basic linear buffer. The circular buffer, if implemented efficiently, will outperform the linear buffer in the long run.

Your `fir()` function should be called within the `process_sample()` function in `ece5210.c`. You will be provided a unit test to check the accuracy of your routine.

3.1.3 Testing our implementation

Like the exercise in Chapter 2, you will be provided tools in the `test/` folder to test your implementation. In this folder, there is a `sandbox.c` file contains a `main()` driver function for you to test simple cases on your implementation before you try testing or deploying to the STM32F769. You do not need to work with the `sandbox.c` file, but it exists for your convenience.

There are also unit tests to check both of your functions. To run the unit tests, use the following in a bash shell:

```
$ cd test/

$ make clean
rm -f sandbox.o debug.o unittests.o ece5210.o sandbox test *~* *.
   mtx *.dSYM *.out
```

```
$ make
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../Core
   /Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3 -I
    ../Core/Inc
gcc -o test unittests.o ece5210.o -lm
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/Inc
gcc -o sandbox sandbox.o ece5210.o -lm

$ ./test
[==========] Running 2 test cases.
[ RUN      ] ece5210_lpf_ez.init_filter
[       OK ] ece5210_lpf_ez.init_filter (334ns)
[ RUN      ] ece5210_lpf_ez.fir_accuracy
[       OK ] ece5210_lpf_ez.fir_accuracy (594386548ns)
[==========] 2 test cases ran.
[  PASSED  ] 2 tests.
```

To pass the tests, you must let `NUM_TAPS` be defined as 35.

3.2 STM32F769I deployment

3.2.1 Determining maximum FIR length

Once you pass both tests, you add the initialization code to the `myaudio.c` file, compile your code in CubeMX, and flash the STM32F769I-DISCO board. Hook your board to the function generator and oscilloscope as described in Sec. 0.2.7. Set the function generator to a 200 Hz square wave. You should see a normal pass-through square wave on the right channel. The square wave should be filtered on the left channel with rounded corners. Now, in your `ece5210.c` file, adjust the value of `NUM_TAPS`. Increase that number until the board cannot produce a reliable waveform on the output. Record the maximum number of taps you can use before the output breaks. Here, the board cannot keep up with the computational demand at each sample at 48 kHz and is missing output blocks. This exercise helps us get an intuition for the upper limits of the STM32F769 processor[1].

Questions and tasks

- What is the upper limit of the number of taps you can use in your FIR implementation?

3.2.2 Measuring the frequency response

Using the longest FIR filter you can reliably use on the board, change the function generator output to a sinusoidal with an amplitude of 1 Vpp. Run a frequency sweep from $f \in [50\,\text{Hz}, 23.5\,\text{kHz}]$ similar to what you did in Sec. 2.2.4. Plot this in Python—magnitude and phase separately—as

[1] A quick aside: ST does provide specially tuned code for the processor's architecture that will execute much faster than what we can do here. Using that code would allow us to use much longer FIR filters. Additionally, we are using floating point arithmetic, which can be slower than fixed point. If we used integer data types instead, the processing might be enbable longer filter lengths. We will discuss this in detail in Chapter 7

well as a numerical computation of the DTFT of your FIR filter. They should match. Plot the measured magnitude and phase responses using scatter plots. You will likely need to make a phase correction to the data to account for the shift between the two channels. What type of filter is this? Does it have linear phase? This filter is very inelegant, but it will suffice for this lab. In the future, we will learn how to design proper digital filters.

Questions and tasks

- Generate plots of this system's magnitude and phase data using the system's analytical frequency response, the numerically computed frequency response, and the measured data. Include proper labels.
- What type of filter is this?
- Does it have linear phase?

3.2.3 Noise removal

The function generators in the lab combine two signals of interest. On the function generator's channel one, generate a 1 Vpp square wave with a frequency of 200 Hz. On channel two, create a noise signal with an amplitude of 1.5 Vpp. Combine both channels into a single output on channel one by hitting `Setup`, then `Dual Channel`, and make sure that `Combine` is selected.

Your DSP should remove some of the noise (but not all of it). Gather this data from the oscilloscope using a USB drive and plot the data in Python.

Questions and tasks

- Generate a plot of the pass-through noisy and processed data. Do you see an improvement?

3.3 Deliverables

3.3.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots.

Your code should be submitted to GitHub.

Demonstration

Before the lab due date, show the noise removal on the oscilloscope to the instructor.

3.3.2 Grading

The lab assignment will have the following grade breakdown.

- `filter_init()`: 10pts.

- `fir()`: 25pts.
- Frequency response plot: 20pts.
- Noise removal plot: 10pts.
- Write-up: 20pts.
- Demo: 15pts.

Chapter 4

Decimation and 60 Hz noise

Interference from 60 Hz power sources is common in sensitive instrumentation recordings. One major issue in echocardiogram recordings is the appearance of unwanted 60 Hz interference in the output[2]. A synthetically generated example of this is seen in Fig. 4.1. This lab aims to use downsampling, FIR filtering, and upsampling to knock out this interference.

4.1 FIR notch filtering

In the homework, we designed a three-tap FIR filter to remove a certain frequency in our signal. This filter will normally take the form

$$h[n] = \delta[n] + a\delta[n-1] + b\delta[n-2]. \tag{4.1}$$

Determine the values for a and b to remove all 60 Hz frequencies from our signal of interest. Assume we are sampling at 48 kHz. Plot the DTFT (magnitude and phase) of your FIR filter. What is the issue with using this type of filter?

Questions and tasks

- What are the filter coefficients for a notch filter at 60 Hz using a 48 kHz sampling rate?
- Plot the DTFT of your FIR filter.
- What is the issue with this particular filter?

4.1.1 Undersampled approach

The issue with the approach used in Sec. **??** is that because we are sampling so fast and trying to notch out such a low frequency, we see that $\omega_0 \to 0$ and as a consequence, we get $a \approx -2$, which leads to suboptimal frequency response. To get around this, we can resample the signal discussed in the textbook. We can adjust our processing pipeline to include a downsampling factor. This is seen in Fig. 4.2

Decimation changes the digital frequency. For example, a digital frequency might be ω_0 before decimation, but this becomes ω_0/M after decimation. This has a couple of implications for us. The first is that the Nyquist frequency is a factor of M lower than the originally sampled signal. If the

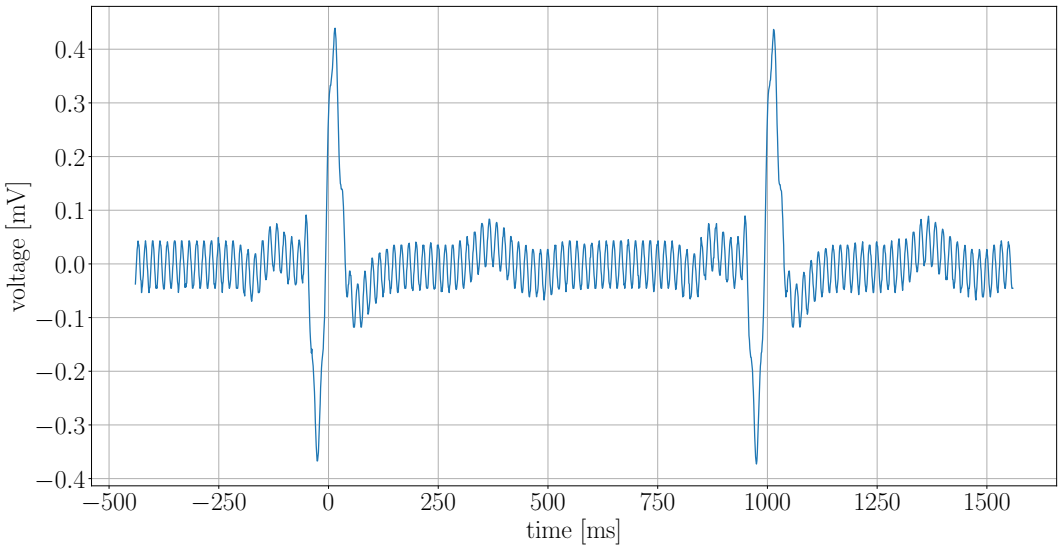

Figure 4.1: Unwanted 60 Hz in ECG recording

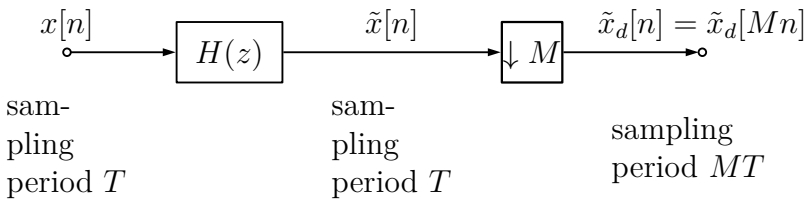

Figure 4.2: Schematic of a typical downsampling process. First, we will apply a digital anti-aliasing filter with $\omega_c = \pi/M$ and a unity passband gain. Then we will downsample by a factor of M.

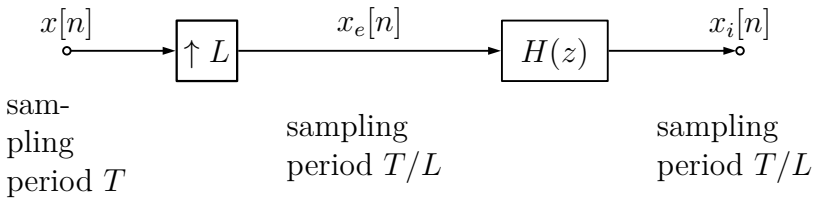

Figure 4.3: Upsampling and interpolation pipeline. First, we expand the signal by inserting zeros between known samples to $x_e[n]$, then we will use linear interpolation.

original signal had a sampling frequency f_s, then the new sampling frequency is $\frac{f_s}{M}$. Consequently, any frequency above $f_s/2M$ will become aliased. We can first apply an anti-aliasing low pass filter with a cutoff of π/M to avoid undesired aliasing. After the anti-aliasing filtering we can downsample the signal to $\tilde{x}[n]$ as seen in Fig. 4.2.

However, we will skip this step in this lab for two reasons. First, implementing this in a computationally efficient manner is beyond the scope of this lab, and we have not discussed the algorithm to do this yet. Second, the ECG signal is reasonably bandlimited, so we can avoid using an anti-aliasing filter in this one instance. We will explore these filters and efficient implementations in Chapter 5.

We also will need to recompute our notch filter

$$h[n] = \delta[n] - 2\cos(M\omega_0)\,\delta[n-1] + \delta[n-2].$$

If we were to downsample aggressively with $M = 100$, we would be able to get a much more desirable filter. Please compute the DTFT of this filter and plot its phase and magnitude. Does this filter better match what we had hoped with our notch filter?

We can then process the downsampled signal with this new FIR filter. After filtering, we must upsample our signal to match the $f_s = 48\,\text{kHz}$ on the STM32's DAC using a sampling rate expander. This essentially inserts zeros between known zeros. Mathematically, this is described as

$$x_e[n] = \begin{cases} x[n/L] & n = L \\ 0 & \text{else} \end{cases}$$

where L is the upsampling factor. For this exercise, we will use $M = L = 100$. We need to reconstruct this signal to interpolate between the known samples. There are several ways to do this, but here we will use linear interpolation. This pipeline is seen in Fig. 4.3.

Questions and tasks

- What is the recomputed $h[n]$ after downsampling by a factor of $M = 100$?
- Make a plot of the magnitude and phase of the DTFT of $h[n]$.

4.2 STM32F769I implementation

We now need to implement this system on the STM32F7 board. We will significantly depart from the pipeline described in Fig. 4.2 by omitting the anti-aliasing filter. Usually, this part of the system would be critical. However, the signal is bandlimited below the Nyquist frequency after

downsampling. Indeed, the lower bound of sampling needed for ECG signals has been an active area of research, with some suggesting that the signal would have a bandwidth of 150 Hz−250 Hz[3], which should work nicely for our downsampling scheme. The more significant reason we would want to omit the anti-aliasing filter is that it would require a very long FIR filter to effectively filter with a cutoff at $\omega_c = \pi/M$ with $M = 100$ in this case. As you learned in Chapter 3, the STM32F7 boards have an upper bound for FIR filtering, and such a filter would easily exceed those capabilities.

4.2.1 Downsampling

You will need to downsample the signal by a factor of $M = 100$ in the `process_sample()` function found in `ece5210.c`. This can be as simple as just ignoring 99 out of the 100 samples coming into this function and keeping just one sample to work with. Keep the first sample and discard the subsequent 99 samples and so forth.

4.2.2 Notch filtering

Filter the samples you did keep with the notch filter. You are welcome to use the `fir()` function you wrote in Chapter 3. You must precompute the filter coefficients for $h[n]$ and hard code them into your code as a global variable.

4.2.3 Upsampling and interpolation

You will need to upsample and filter the result with an interpolation filter. We could use various types of interpolation filters, but for this lab, we will use linear interpolation between two adjacent samples. The easiest implementation for linear interpolation is

$$y = y_1 + (x - x_1)\frac{y_2 - y_1}{x_2 - x_1}. \tag{4.2}$$

You are also welcome to think of linear interpolation in terms of convolution, similar to what we discussed in the lecture, but that approach tends to be more complicated than the more closed-form approach.

Once your code is complete, a unit test will check your work.

```
$ make
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../Core
   /Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3 -I
    ../Core/Inc
gcc -o test unittests.o ece5210.o -lm
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/Inc
gcc -o sandbox sandbox.o ece5210.o -lm

$ ./test
[==========] Running 1 test cases.
[ RUN      ] ece5210_60hz.process_sample
[       OK ] ece5210_60hz.process_sample (19154151ns)
[==========] 1 test cases ran.
[  PASSED  ] 1 tests.
```

4.2.4 Verification

To check this works on the board, we will do a frequency sweep within the frequencies of the downsampled signal. Recall from class after we downsample that the effective sampling frequency will be scaled by $f_s \to {}^{f_s}/_M$, so we will only need to check from $f \in [10\,\text{Hz}, 240\,\text{Hz}]$. Record the magnitude and phase response. Plot this overlaid the DTFT of the notch filter. Is this the same? How is it different? Why is it different?

Next, generate an ECG signal corrupted by 60 Hz noise. We can set up an arbitrary waveform on channel 1 to do this. Make sure that channel 1 is selected and select `Select Waveform`. Then select `Arb` on the side menu, then `Select Arb`, then `Select Internal`, then scroll to `Cardiac` which will provide a cardiac waveform. Hit the `Parameter` button on the function generator and change the sample rate to $450\,\text{S}\,\text{s}^{-1}$. This corresponds to a heart rate of 60 bpm, which is reasonable. Adjust the amplitude to 600 mVpp.

Next, we will corrupt the cardiac signal with interference. On channel 2, set up a 60 Hz sine wave with an amplitude of 100 mVpp. To combine the channels, go back to channel 1. Hit `Setup`, then `Dual Channel`, and make sure that `Combine` is selected.

Record the filtered and pass-through data on a USB drive and plot the data. Is the 60 Hz interference removed?

Questions and tasks

- Generate a plot of the frequency response (magnitude and phase) of the system between $f \in [10\,\text{Hz}, 240\,\text{Hz}]$ in 10 Hz steps. Overlay this data over the DTFT of the notch filter. How is it the same? How is it different? If it is different, why?
- Plot the pass-through and filtered data on the same plot. Was the filter effective?

4.3 Deliverables

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots. Your code should be submitted to GitHub. Additionally, you will need to demonstrate your functioning board at the beginning of the lab section.

4.3.1 Grading

The lab assignment will have the following grade breakdown.

- `process_sample()`: 35pts.
- Frequency response plot: 20pts.
- Interference removal plot: 10pts.
- Write-up: 20pts.
- Demo: 15pts.

Chapter 5

Polyphase decimation and interpolation

In the previous lab, we assumed that the bandwidth of the ECG signal would be below 240 Hz. We used this *a priori* knowledge to avoid using a digital anti-aliasing filter before decimation. While this worked, we cannot always make this assumption. We must efficiently apply a low pass filter to remove higher frequency signal components to ameliorate the effects of aliasing post-decimation. This lab focuses on implementing polyphase decimation and interpolation.

5.1 Polyphase decimation

Recall that we can represent some discrete-time system $h[n]$ in its polyphase components

$$e_k[n] = h[nM + k]$$

with a z-transform

$$H(z) = \sum_{k=0}^{M-1} E_k(z^M) z^{-1}.$$

Given some input $x[n]$, the system response is therefore

$$\begin{aligned} Y(z) &= H(z)X(z) \\ &= \sum_{k=0}^{M-1} E_k(z^M) X(z) z^{-1}. \end{aligned}$$

Remember that we are often interested in this implementation in the context of decimation, so we can swap the decimation and filter such that we downsample first *and then* filter. This avoids computing the filtering first and discarding the samples during downsampling. A block diagram of this process is seen in Fig. 5.1.

We can apply this same reasoning to interpolation, which you likely have implemented in the exercises in Chapter 4. A block diagram of polyphase interpolation is seen in Fig. 5.2. In this exercise, we will implement polyphase interpolation and decimation to determine how long of a low-pass filter we can get away with on this board.

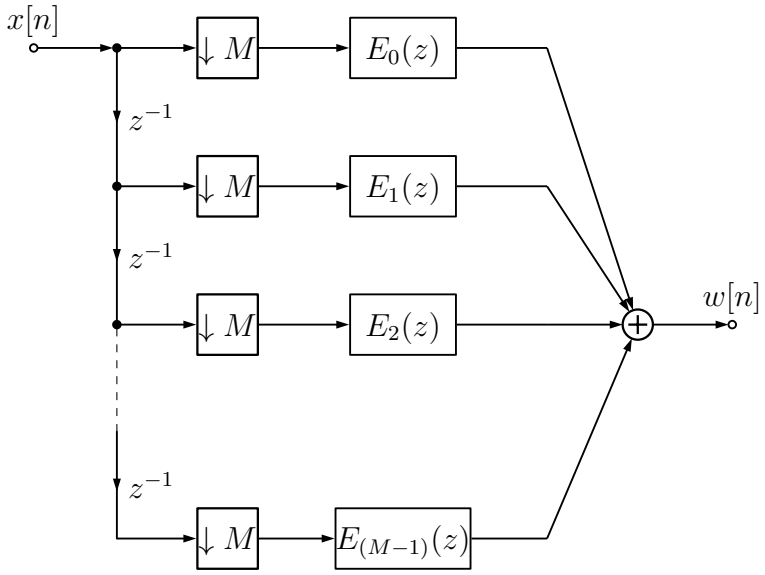

Figure 5.1: Implementation of decimation filter using polyphase decimation decomposition.

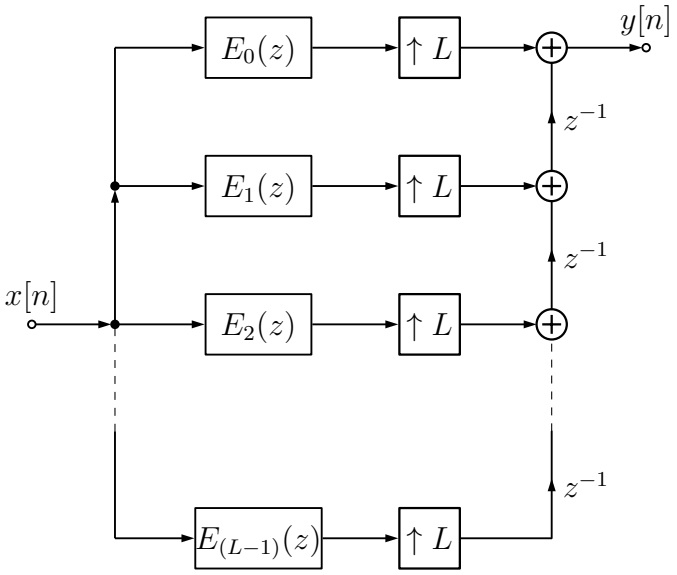

Figure 5.2: Implementation of interpolation filter using polyphase interpolation decomposition.

5.2 STM32F769I implementation

5.2.1 Filter design

This exercise aims to implement a fast decimation and interpolation scheme. To do this, we will need some low-pass filters. To keep things simple, we will let $L = M$ throughout the lab to use the same low-pass filter for both anti-aliasing and interpolation. The filter will have a unity gain (0 dB) in the passband and a cutoff at $\omega_c = \pi/M$. In the `ece5210.c` file you will be provided with a function `init_firwin()` that populates a global array `h_aa[NUM_TAPS]` with these filter coefficients. This filter is not particularly sophisticated as it is just a sinc with a Hamming window. This simple design allows us to design the filter on the fly with variable lengths. Change the definition of `NUM_TAPS` to change the filter length or the definition of `W_C` to change the frequency cutoff. This

function will be called in `myaudio.c`, similar to what we did in Chapter 3.

The `init_firwin()` sets up a 1D array where the filter coefficients are laid out in contiguous memory. If we were to constantly extract the polyphase component filters from `h_aa`, we would be striding through the array continually, which would cause cache misses and potentially bog down your program. Instead, you should set up a new global 1D array called `float h_poly[NUM_TAPS]`, where the polyphase components are stored sequentially instead of interleaved. In other words, if we have a filter with 100 coefficients decimating by a factor of 10, we will set up `h_poly` such that the ten coefficients for $e_0[n]$ are the first ten elements of that array. The following ten elements of `h_poly` are the ten coefficients of $e_1[n]$. And so forth. In the `init_firwin()` function, you should populate a `h_poly` array with the polyphase components at the end of the function. You can check your work with a unit test. Ensure that for all unit tests, $M = 100$ and the number of taps is 900.

```
$ ./test
[==========] Running 6 test cases.
[ RUN      ] ece5210_polyphase.polyphase_components
[       OK ] ece5210_polyphase.polyphase_components (11226723ns)
...
[==========] 6 test cases ran.
[  PASSED  ] 6 tests.
```

5.2.2 Decimation

First, we will need to implement a fast polyphase decimation routine. Remember, the polyphase decomposition aims to gain numerical efficiencies to deploy them on a lower-powered microprocessor. Let's start by writing a function with the following declaration.

```
/**
Implementation of polyphase decimation with an efficient
FIR filter.

* @param x incoming sample x[n]
* @param *h coefficients of the entire FIR LPF
* @param M decimation factor (must be integer)
* @return output is either the decimation value or a zero
*/
float polyphase_decimation(float x, float *h, uint16_t M)
```

Because this C function will need to return *something* for every input $x[n]$, the easiest way to handle this is to return a zero where the signal should be downsampled. You will likely need some internal counter inside the function to track which samples you will keep and which samples you will discard (i.e., set to zero).

You should test this function at $M = 100$ with a filter length of 900 taps. There is a unit test to check your work. Your function is tested for both accuracy and speed. Successful unit testing should look like the following:

```
$ ./test
[==========] Running 6 test cases.
...
```

```
[ RUN      ] ece5210_polyphase.decimate_accuracy
your code is being tested with M=100 and NUM_TAPS=900
[       OK ] ece5210_polyphase.decimate_accuracy (1468759425ns)
[ RUN      ] ece5210_polyphase.decimate_time
[       OK ] ece5210_polyphase.decimate_time (213671549ns)
...
[==========] 6 test cases ran.
[  PASSED  ] 6 tests.
```

The speed test is against a naive implementation where the signal is filtered using a generic FIR system followed by decimation. Your code must be ten times faster than this implementation. On the board, your code will be much faster than the generic FIR system, but on your laptop, there is a lot of gcc compilation magic that will narrow the gap between the two considerably.

5.2.3 Interpolation

You will also need to implement an efficient interpolation routine. You likely have done this already if you completed the tasks in Chapter 4, so you are welcome to adapt your code for usage here. You will need to write a function to perform the interpolation using the following function declaration:

```
/**
Implementation of polyphase decimation with an efficient
FIR filter.

* @param x incoming sample x[n], if no sample is needed
this should be taken as a zero
* @param *h coefficients of the entire FIR LPF (with
unity gain at DC)
* @param L interpolation factor (must be integer)
* @return interpolated sample y[n]
*/
float polyphase_interpolation(float x, float *h,
uint16_t L)
```

This function will take input samples $x[n]$ where the "missing" samples are passed in as zeros. You should ignore those terms in your interpolation. Once again, you will be tested with $L = 100$ and a 900-tap low-pass filter. Your function will need to be both accurate and efficient and will be tested for both.

```
$ ./test
[==========] Running 6 test cases.
...
[ RUN      ] ece5210_polyphase.interpolate_accuracy
your code is being tested with L=100 and NUM_TAPS=900
[       OK ] ece5210_polyphase.interpolate_accuracy (7773902ns)
[ RUN      ] ece5210_polyphase.interpolate_time
[       OK ] ece5210_polyphase.interpolate_time (1318702ns)
...
```

```
[       OK ] ece5210_polyphase.total_system (8107596ns)
[==========] 6 test cases ran.
[  PASSED  ] 6 tests.
```

Putting it all together

You now have all the correct pieces. You need to modify `process_sample()` to take an input sample, decimate it, filter the downsampled signal similar to what we did in Chapter 4, and then interpolate it with your polyphase interpolation. To test if this works, set the decimation and interpolation factors to 100 and then set the anti-aliasing/imaging filter length to 900 taps.

You can test this entire system.

```
$ ./test
[==========] Running 6 test cases.
...
[ RUN      ] ece5210_polyphase.total_system
your code is being tested with M=100 and NUM_TAPS=900
[       OK ] ece5210_polyphase.total_system (8107596ns)
[==========] 6 test cases ran.
[  PASSED  ] 6 tests.
```

5.2.4 Speed

You might struggle to get the most out of your algorithms if you are trying to use linear buffers for your filters. Linear buffers worked very well for Chapter 2 exercises. In this instance, we have multiple linear buffers the cache does not seem to keep up as much. A circular buffer (also known as a ring buffer) could be very helpful since it avoids constantly moving array elements in memory. I would strongly suggest reading the Wikipedia entry on circular buffers as a quick primer on how these work. The goal of this exercise is not to implement the fastest circular buffer known to man, so in the interest of your time, here is some code that might help make this work:

```
#define BUFF_SIZE (128U)
#define BUFF_SIZE_MASK (BUFF_SIZE-1U)

typedef struct buffer {
    float data[BUFF_SIZE];
    uint16_t write_buff_index;
} buffer;


/**
Write to circular buffer.

   * @param *buff pointer to the buffer
   * @param value the value to be inserted
   * @return void
*/
void write_buff_val(buffer *buff, float value)
```

```
{
    buff->data[(++buff->write_buff_index) & BUFF_SIZE_MASK] =
        value;
}


/**
Read value from circular buffer.

   * @param *buff pointer to the buffer
   * @param idx_in index you want to read from (as if it is a
                   linear buffer
   * @return value from index of interest
*/
float read_buff_val(buffer *buff, uint16_t idx_in)
{
    return buff->data[(buff->write_buff_index - idx_in) &
                      BUFF_SIZE_MASK];
}
```

If you end up using this instead of a traditional linear buffer, please read through this code to ensure you understand what is going on with the mechanics. Knowing how these work is a good idea because they pop up a lot in DSP algorithms.

5.2.5 Verification

The task of this exercise is to efficiently implement decimation with anti-aliasing filtering and upsampling with an interpolation filter. In your `process_sample()` function in `ece5210.c` simply downsample the signal and upsample the signal. Compile your code for the STM32F769I and flash the board. Run a 500 mVpp sine at 100 Hz to make sure that you get a sine wave with the same frequency and amplitude on the output (there will be, of course, a phase shift between the processed and pass through signals). We are downsampling and upsampling at $M = L = 100$, but we vary the number of taps starting at 900 and increase it by increments of 100 to see where the limit is. Where does it break? Why do you increase the number of taps in the filter by increments of 100?

Once you have found the limit of the number of taps you can use, perform a frequency sweep from 10 Hz to 320 Hz in increments of 10 Hz, record the change in magnitude and phase and plot them.

Questions and tasks

- What is the limit of the number of taps you could get before the processor could not keep up?
- Why do you increase the number of taps by increments of 100?
- Plot the magnitude and phase response of this system.

5.3 Deliverables

5.3.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots. Your code should be submitted to GitHub. Additionally, you must demonstrate your functioning board at the beginning of the lab section.

5.3.2 Grading

The lab assignment will have the following grade breakdown.

- `h_poly`: 5pts.
- `polyphase_decimation()` accuracy: 20pts.
- `polyphase_decimation()` speed: 10pts.
- `polyphase_interpolation()` accuracy: 20pts.
- `polyphase_interpolation()` speed: 10pts.
- `process_sample()`: 15pts.
- Frequency response plot: 10pts.
- Write-up: 5pts.
- Demo: 5pts.

Chapter 6

All-pass systems

Any LTI system can be decomposed into a minimum phase, and all-pass components

$$H(z) = H_{\text{ap}}(z)H_{\text{min}}(z).$$

A typical all-pass system will take the form

$$H_{\text{ap}}(z) = A\prod_{k=1}^{M_r}\frac{z^{-1}-d_k}{1-d_kz^{-1}}\prod_{k=1}^{M_c}\frac{(z^{-1}-e_k^*)(z^{-1}-e_k)}{(1-e_kz^{-1})(1-e_k^*z^{-1})}$$

where d_k are the real zeros outside the unit circle while e_k and e_k^* are the zeros that are complex conjugate pairs outside the unit circle. This system will have unity gain throughout the entire frequency spectrum and some arbitrary phase response.

A typical minimum phase system $H_{\text{min}}(z)$ will consist of the zeros and poles from $H(z)$ that are inside the unit circle and any additional zeros to compensate for the poles in $H_{\text{ap}}(z)$. A minimum phase system will have the same magnitude response as $H(z)$, though the phase response will have minimum phase lag. Taken together, we should expect

$$\begin{aligned}
&\left|H_{\text{ap}}\left(e^{j\omega}\right)\right| = 1 \\
&\left|H\left(e^{j\omega}\right)\right| = \left|H_{\text{min}}\left(e^{j\omega}\right)\right| \\
&\angle H\left(e^{j\omega}\right) = \angle H_{\text{min}}\left(e^{j\omega}\right) + \angle H_{\text{ap}}\left(e^{j\omega}\right).
\end{aligned} \tag{6.1}$$

In class, we also discussed the types of generalized linear phase FIR filters and where to place zeros to ensure a system maintains a linear phase response. In this exercise, we will design a generalized linear phase system given some requirements for a couple of placements of zeros. We will also decompose this system into its minimum phase and all-pass forms and implement them on the STM32F769I board to verify this behavior is maintained.

6.1 System design

6.1.1 Designing a generalized phase FIR system

Your first task is to design a Type II FIR system. This system should have at least one zero at $z = 0.8e^{j\pi/4}$ and another zero at $z = 0.4e^{j4\pi/5}$. You must add additional zeros to match the

Type II system form. What is $H(z)$? You may report this as a product of zeros rather than a polynomial of z^{-k} terms. Use a gain of 0.1 such that $H(z) = 0.1(1 - c_0 z^{-1})(1 - c_1 z^{-1}) \ldots$. Make a plot of the zeros and poles of $H(z)$, and make sure this is properly labeled with the unit circle marked. Next, we will want to determine the filter coefficients. To do this, we must multiply these zero terms, which can be onerous. You may use SciPy's `signal.zpk2tf()` function to do this work for you. Ensure all Python work is done in a file called `test/allpass.py` in the project repository. Please refer to the online documentation for how this works. Make a stem plot for $h[n]$, the impulse response of $H(z)$. On separate plots, plot the frequency response $|H\left(e^{j\omega}\right)|$ and $\angle H\left(e^{j\omega}\right)$ on separate plots.

Questions and tasks

- What is $H(z)$?
- Make a labeled plot marking the zeros and poles of $H(z)$. Indicate where the unit circle is.
- On separate plots, plot the system's magnitude response $|H\left(e^{j\omega}\right)|$ and the phase response $\angle H\left(e^{j\omega}\right)$. Make sure that the magnitude plots are in dB.
- Make a stem plot of $h[n]$ coefficients.

6.1.2 System decomposition

Decompose $H(z)$ into an all-pass system $H_{\text{ap}}(z)$ and a minimum phase system $H_{\min}(z)$. I know SciPy has tools for this, but please do this part by hand (there is a good chance you will be asked to do this later in the course without computation aids, so this is good practice).

On the same plots where you plotted $|H\left(e^{j\omega}\right)|$ and $\angle H\left(e^{j\omega}\right)$, now add plots for the magnitude and phase response for $H_{\text{ap}}\left(e^{j\omega}\right)$ and $H_{\min}\left(e^{j\omega}\right)$. As you can notice, $H_{\min}(z)$ is an FIR system, and we have computed frequency responses for these systems all system. However, $H_{\text{ap}}(z)$ is an IIR system, so your DTFT function will not work here. In this instance, you are welcome to use SciPy's `signal.freqz()` function to determine the frequency responses of these systems. Please read the documentation on how this function works. Do the frequency responses follow the behavior that you would expect based on Eq. (6.1)?

Lastly, please compute $h_{\min}[n]$ and plot it on the same stem plot you created for $h[n]$. Even those impulse responses will generate the same frequency responses; what are the main differences between the two impulse responses in the time domain?

Questions and tasks

- What is $H_{\text{ap}}(z)$ and $H_{\min}(z)$? You may leave these in factored form.
- Add the frequency responses to the magnitude and phase plots you generated in Sec. 6.1.1. These must be clearly labeled. Do these match your expected behavior based on Eq. (6.1)?
- Determine $h_{\min}[n]$ and add this to the stem plot for $h[n]$ you created in Sec. 6.1.1. How do these impulse responses compare?

6.2 STM32F769I implementation

6.2.1 Generalized linear phase system

We will first implement $H(z)$ on our STM32F769I boards. This is just a simple FIR system, which you have done several times in this course. Write a function in `ece5210.c` that will perform FIR filtering with a Type II filter. You must save the filter coefficients in a floating-point array global variable called `b_linear`. You will need to manually copy/paste the coefficients from the filter you designed in Python. Ensure you copy over eight decimal places of precision (`%.8f` if you are writing to file, which I recommend). The function declaration (which will also be needed in `ece5210.h` to pass the unit tests) will look like the following:

```
/**
FIR filtering with a Type II filter
   * @param x floating-point input sample
   * @param *b floating-point filter coefficients
   * @param len_b uint16_t length of the filter
   * @param floating-point output sample
*/
float H_linear(float x, float *b, uint16_t len_b);
```

Deploy this function to the board and call it in the `process_sample()` function to process the signal. Hook up your board to the function generator and oscilloscope and perform a frequency sweep from 500 Hz to 23.5 kHz in 500 Hz. Record the phase and amplitude changes. Plot this data as scatter plot points on the theoretical frequency response plots you created in Sec. 6.1.1. Make sure the data is properly color-coded and it is labeled. As you can see, these plots will be pretty large, so you might increase the plot size before you save it (you can adjust this in the `plt.figure()` call).

Questions and tasks

- Collect and plot the frequency response data from 500 Hz to 23.5 kHz.
- Does this data agree with the theoretical data?

6.2.2 Minimum phase system

Next, we will implement $H_{\min}(z)$ on our STM32F769I boards. This is very redundant to what we did in Sec. 6.2.1 other than we will be using different filter coefficients. Once again, copy the coefficients from Python and paste them into a floating-point array global variable called `b_min`. Make sure you maintain eight decimal places of precision. The function declaration (which will also be needed in `ece5210.h` to pass the unit tests) will look like the following:

```
/**
FIR filtering with a minimum phase FIR filter
   * @param x floating-point input sample
   * @param *b floating-point filter coefficients
   * @param len_b uint16_t length of the filter
   * @param floating-point output sample
*/
float H_min(float x, float *b, uint16_t len_b);
```

Deploy this function to the board and call it in the `process_sample()` function to process the signal. You must comment on the previous call to `H_linear()`. Hook up your board to the function generator and oscilloscope and perform a frequency sweep from 500 Hz to 23.5 kHz in 500 Hz. Record the phase and amplitude changes. Plot this data as scatter plot points on the theoretical frequency response plots you created in Sec. 6.1.1. Make sure the data is properly color-coded and it is labeled.

Questions and tasks

- Collect and plot the frequency response data from 500 Hz to 23.5 kHz.
- Does this data agree with the theoretical data?

6.2.3 All-pass system

Lastly, we will implement $H_{\text{ap}}(z)$ on our STM32F769I boards. This is an IIR filter, which we have not done before. There are a few ways to approach this, and we will not concern ourselves with having either the most efficient or the most numerically stable implementation. The IIR difference function is usually[1] written as

$$y[n] + \sum_{k=1}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

which would have the corresponding transfer function

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}}. \tag{6.2}$$

There are typically two implementations of an IIR filter. The first is a direct form I structure that is a bit simpler to understand, as seen in Fig. 6.1. In this implementation, the signal goes through the left-hand side of the system, which is effectively an FIR system to output $v[n]$. After this, scaled and delayed copies of $y[n]$ are added to $v[n]$ to provide negative feedback. These figures are similar to the Oppenheim text which assumes that the coefficients a_k are going to be negated (i.e., multiplied by -1) to provide negative feedback to satisfy Eq. (6.2).

The second implementation—called the direct form II—is a bit less intuitive, though it is more efficient and ultimately simpler to implement. It is seen in Fig. 6.2. In this form, the feedback and feedforward connections are reversed to reduce the number of delays by sharing a common $w[n]$. Once again, the a_k coefficients are assumed to be negated to satisfy Eq. (6.2).

You can choose either the direct form I or II to implement the all-pass system. Once again, you must copy the filter coefficients from Python and paste them into a floating-point array global variable called `b_ap` and `a_ap`. Make sure you maintain eight decimal places of precision. The function declaration (which will also be needed in `ece5210.h` to pass the unit tests) will look like the following:

[1] This is a deviation from the book's notation, which negates the a_k coefficients. This is generally inconsistent with how most people write this difference equation. To be compatible with the Python/SciPy IIR filter design functions, we will stick with the more popular convention in Eq. (6.2).

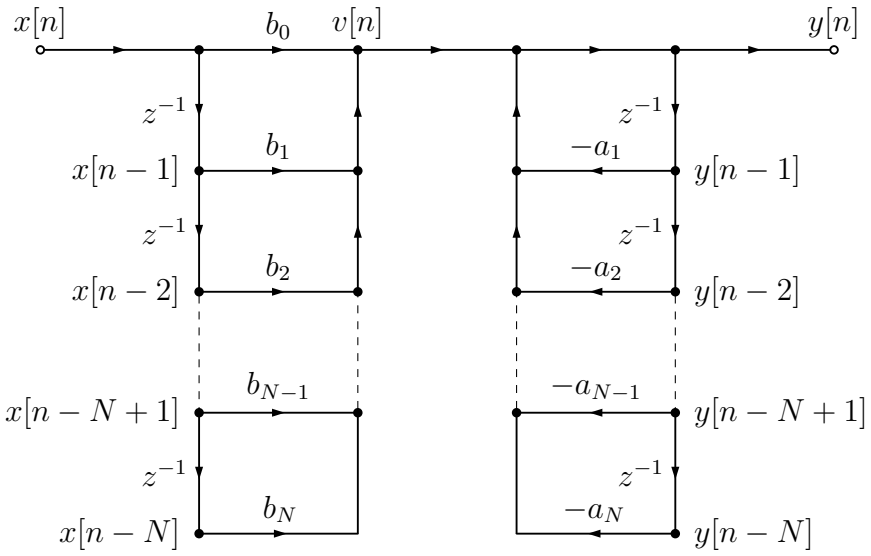

Figure 6.1: Signal flow graph of direct form I structure for an N^{th}-order system.

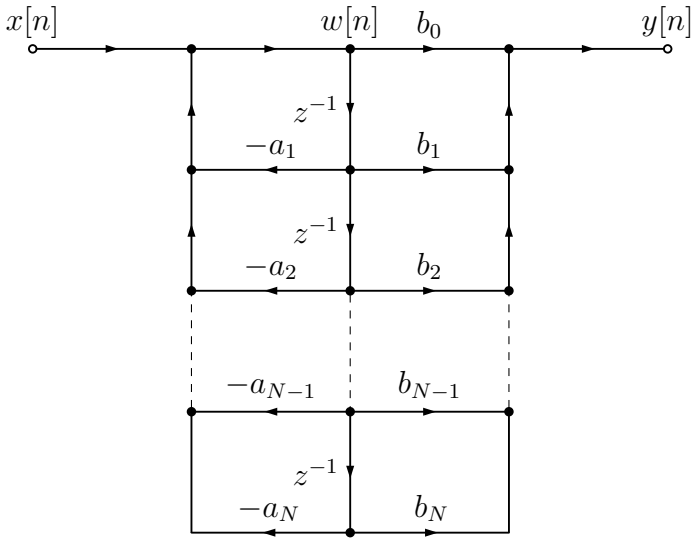

Figure 6.2: Signal flow graph of direct form II structure for an N^{th}-order system.

```
/**
IIR filtering with an all-pass filter
   * @param x floating-point input sample
   * @param *b floating-point numerator filter coefficients
   * @param *a floating-point denominator filter coefficients
   * @param len_b uint16_t length of the filter
   * @return floating-point output sample
*/
float H_ap(float x, float *b, float *a, uint16_t len_b);
```

Deploy this function to the board and call it in the `process_sample()` function to process the signal. You must comment out the previous call to `H_linear()`. Hook up your board to the function generator and oscilloscope and perform a frequency sweep from 500 Hz to 23.5 kHz in 500 Hz increments. Record the phase and magnitude responses. Plot this data and the theoretical frequency response plots you created in Sec. 6.1.1 on the same figure. Make sure the data is properly color-coded and it is labeled.

Questions and tasks

- Collect and plot the frequency response data from 500 Hz to 23.5 kHz in 500 Hz increments.
- Does this data agree with the theoretical data?

6.2.4 Testing your code

You will have unit tests to check your functions against the SciPy implementation to ensure they are correct. Successful passage of tests looks like the following:

```
$ make
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../Core
   /Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3 -I
    ../Core/Inc
gcc -o test unittests.o ece5210.o -lm
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/Inc
gcc -o sandbox sandbox.o ece5210.o -lm

$ ./test
[==========] Running 3 test cases.
[ RUN      ] ece5210_allpass.linear
[       OK ] ece5210_allpass.linear (13629677ns)
[ RUN      ] ece5210_allpass.min
[       OK ] ece5210_allpass.min (7560136ns)
[ RUN      ] ece5210_allpass.all_pass
[       OK ] ece5210_allpass.all_pass (8365096ns)
[==========] 3 test cases ran.
[  PASSED  ] 3 tests.
```

6.2.5 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

Please include the requested plots in the appendices. Submit your code to GitHub. Additionally, you must demonstrate your functioning board at the beginning of the lab section.

6.2.6 Grading

The lab assignment will have the following grade breakdown.

- Pole/zero plot: 5pts.
- Filter stem plots: 5pts.
- Magnitude response plot: 25pts.
- Phase response plot: 25pts.
- `H_linear()`: 5pts.
- `H_min()`:5pts.
- `H_ap()`: 10pts.
- Write-up: 15pts.
- Check-off: 5pts.

Chapter 7

Fixed-point DSP

DSP chips are generally divided into fixed-point and floating-point types. Both types can process floating-point data, but the fixed-point processors use software libraries to simulate floating-point computation integer arithmetic. This is inherently very slow. The processor on the STM32F769I-DISCO board comes with a floating-point unit (FPU), which enables rapid floating-point arithmetic. However, not every processor you will encounter in your career will have a decent FPU because these electronics are more complicated and expensive. In this lab, we will implement an FIR system using fixed-point representation and arithmetic and analyze the numerical effects.

7.1 Fixed-point systems

7.1.1 Fixed-point representation

Single-precision floating-point representation usually has 6-9 significant digits of precision with the smallest positive value of 1.4×10^{-38} and the largest positive value is 3.4×10^{38}. With a huge dynamic range and particular values and having unique values for $\pm\infty$ and `NaN`, there is a very low likelihood of overflow. Python generally uses double precision for most computations. However, this is considerably slower than single-precision on a 64-bit system (e.g., your laptop), let alone on a 32-bit system like your STM32F769I-DISCO.

Fixed-point representation forces decimal values to predefined quantized levels. Using fixed-point, we can perform basic arithmetic (i.e., addition, subtraction, multiplication, and division). Because this representation has a much smaller dynamic range, we will need to exercise extra care to avoid overflow.

Suppose we have a $B+1$ bit datatype with bits $\{a_0, a_1, \ldots, a_B\}$ in two's complement fraction form $a_{0\diamond}a_1a_2 \ldots a_B$ where $\diamond$ is the binary point and q is the number of bits right of $\diamond$. We can represent decimal values according to

$$\text{decimal value} = 2^{-q}\left(-a_0 2^B + \sum_{i=1}^{B} a_i 2^{B-i}\right).$$

For example, if we had a binary representation $10_\diamond 01$ we would have $B = 3$ and $q = 2$. From this,

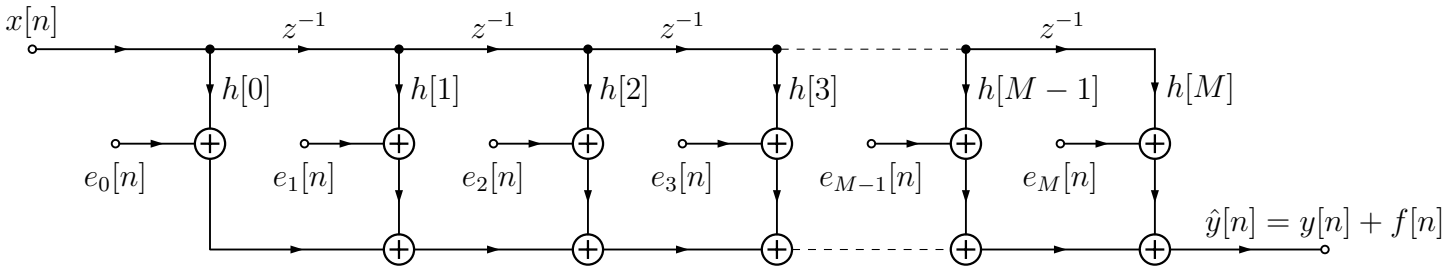

Figure 7.1: Linear-noise model of direct-form of an FIR system.

we can compute

$$\begin{aligned} 10_{\diamond}01 &= 2^{-2}\left(-1\cdot 2^3 + \left(0\cdot 2^{3-1} + 0\cdot 2^{3-2} + 1\cdot 2^{3-3}\right)\right) \\ &= \frac{1}{4}(-8+0+2+1) \\ &= \frac{1}{4}(-5) \\ &= -1.25. \end{aligned}$$

Converting floating-point to fixed-point is just a matter of bit-wise operations. The C language does not natively support fixed point arithmetic, but we can approximate it using integer-type data. This is extensively covered online, and ARM has prepared a document with relevant C-code to approach it found here. While fixed-point addition and multiplication are relatively trivial (indeed, ARM suggests just handling it with simple C macros), there is a risk of overflow because you lose the dynamic range in your number representation. You need to be aware of this and perhaps add checks for overflow and set the operation results to some saturation maximum or minimum values.

7.1.2 Fixed-point FIR systems

We have talked extensively in this course (and ECE 3430) about what happens when you quantize a signal, but what happens when you quantize a system? Consider the FIR system in Fig. 7.1. When you quantize the filter coefficients, we are effectively adding quantization noise $e_k[n]$ to every filter coefficient $h[n]$. When we quantize the filter coefficients, we will use the same number of fractional bits for all the coefficients. We hope to minimize the quantization errors while maximizing the number of fractional bits and avoiding overflow. You will generally run into issues when you have very large and very small coefficients.

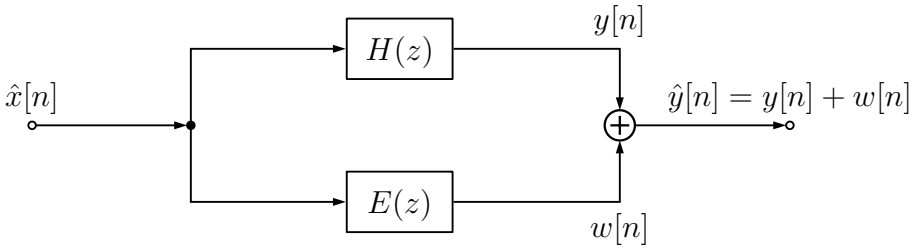

Figure 7.2: Parallel representation of the quantized FIR system.

If we quantize the filter coefficients $h[n]$ to $\hat{h}[n]$ such that

$$\begin{aligned}\hat{H}(z) &= \sum_{n=0}^{N-1} \hat{h}[n] z^{-n} \\ &= \sum_{n=0}^{N-1} (h[n] + e_n) z^{-n} \\ &= \sum_{n=0}^{N-1} h[n] z^{-n} + \sum_{n=0}^{N-1} e_n z^{-n} \\ &= H(z) + E(z).\end{aligned}$$

This leaves us with an effective parallel system where we have the quantized coefficients $\hat{H}(z)$ as well as $E(z)$ which is an FIR system of noise terms that come from quantizing $H(z)$. We can see this in Fig. 7.2 where $E(z)$ is represented in parallel by the unquantized $H(z)$ system.

A causal FIR system $E(z)$ with $N = M+1$ coefficients with a region of convergence (ROC) $|z| > 0$ will have a frequency response

$$E\left(e^{j\omega}\right) = \sum_{n=0}^{N-1} e_n e^{-j\omega n}.$$

If you employ rounding quantization, each e_n will be bounded between $-\Delta/2$ and $\Delta/2$, assuming no saturation. It follows

$$\begin{aligned}\left|E\left(e^{j\omega}\right)\right| &= \left|\sum_{n=0}^{N-1} e_n e^{-j\omega n}\right| \\ &\leq \sum_{n=0}^{N-1} \left|e_n e^{-j\omega n}\right| \\ &\leq \sum_{n=0}^{N-1} \frac{\Delta}{2} \\ &= \frac{N\Delta}{2}.\end{aligned}$$

From this, we see that

$$\left(\left|H\left(e^{j\omega}\right)\right| - \frac{N\Delta}{2}\right)^{+} \leq \left|H\left(e^{j\omega}\right)\right| \leq \left|H\left(e^{j\omega}\right)\right| + \frac{N\Delta}{2}.$$

This means that when we quantize our filter coefficients, there will be an upper and lower bound to the frequency response.

In terms of implementation, unlike IIR filters, we usually use a direct form realization of an FIR filter because they provide acceptable performance. Quantization will not impact filter symmetry (particularly Type I and Type II filters), so you should preserve the linear phase using a direct form realization. You could use a cascade or parallel form, as they provide better performance by reducing the dynamic range of the coefficients. However, most FIR filters will still use a direct form implementation unless you have zeros in $H(z)$ that are clustered together tightly. If you use a cascade form, you should be thoughtful in your implementation to preserve the linear phase.

7.2 STM32F769I implementation

7.2.1 Quantizing the filter coefficients

Once again, you are provided a function `init_firwin()` in the `ece5210.c` file. This will build a Hamming-windowed sinc FIR filter with length `NUM_TAPS` that you can control via a macro. The cutoff frequency can also be controlled by macro, but we will use $\omega_c = \pi/3$ for this exercise. This function will populate a global variable `h_float`, an array of 32-bit floating point numbers. You must add lines to populate a second global variable `h_fixed`, an array of 32-bit integers that will use fixed-point representation. Use 12 fractional bits here. Additionally, while you can mix and match the number of fractional digits in your fixed-point arithmetic, this becomes cumbersome, and, in practice, most people will usually use the same number of fractional bits throughout the system. Once you have this conversion, use `NUM_TAPS=12` and create several plots of the coefficients using both fixed-point and floating-point representation. First, make a stem plot with both filter coefficients (obviously with a label and legend so we know which is which). You must convert your integer-valued coefficients back to decimals for these plots. They should be quantized. I recommend using the `sandbox.c` program to write these coefficients to a text file you can read into Python (a `fprintf()` in a `for` loop should make this happen). Second, a plot of the poles and zeros of both filters is created. Remember the filter coefficients will correspond to

$$H(z) = h[0] + h[1]z^{-1} + h[2]z^{-2} + \cdots + h[M]z^{-M}.$$

We will need to factor this to

$$H(z) = \prod_{k=0}^{M} \left(1 - c_k z^{-1}\right)$$

to find the zeros. Remember that an FIR system also has a pole at the origin. Lastly, make a plot of the frequency response for both $H(z)$ and $\hat{H}(z)$. Plot the magnitude response in dB and $ARG[\cdot]$ for the phase response.

Questions and tasks

- On the same figure, create stem plots for both $h[n]$ and $\hat{h}[n]$.
- On the same figure, plot the poles and zeros for both $H(z)$ and $\hat{H}(z)$.
- On the same figure, plot the magnitude response of both $H(e^{j\omega})$ and $\hat{H}(e^{j\omega})$.
- On the same figure, plot the phase response of both $H(e^{j\omega})$ and $\hat{H}(e^{j\omega})$.

7.2.2 FIR filter implementation

We will implement an FIR system using a fixed-point representation. The function definition for the fixed-point FIR system will look like the following:

```
/**
Implementation of an FIR filter system.

* @param sample_in the current sample x[n] to be processed
* @param len_h the length of the filter
* @return the output sample y[n]
*/
int16_t fir_fixed(int16_t sample_in, uint16_t len_h);
```

We will benchmark the accuracy of this system against the floating-point implementation, assuming that the floating-point implementation is an appropriate reference standard. You can adjust the number of fractional bits to achieve better accuracy. Your implementation may not use any floating-point number. You will have two unit tests to check your implementation against a standard floating-point implementation. The first test checks against a random input. The second test checks against a constant input close to the saturation point of a steady 16-bit integer[1]. Successful testing will look like this:

```
$ ./test
[==========] Running 2 test cases.
[ RUN      ] ece5210_fixed.random_input
[       OK ] ece5210_fixed.random_input (221619ns)
[ RUN      ] ece5210_fixed.dc_input
[       OK ] ece5210_fixed.dc_input (125472ns)
[==========] 2 test cases ran.
[  PASSED  ] 2 tests.
```

Questions and tasks

- How many fractional bits did you use?

7.2.3 Board deployment

Compile your code in STM32CubeIDE and flash your board. Use a 200 Hz square wave with 1.2 Vpp. Now adjust the number of taps in your filter until the board cannot keep up with the computation. How many taps were you able to use? What other "tricks" did you use in your system to make your implementation more efficient? Did you use linear or circular buffers?

Please perform a frequency sweep to record the frequency response for this system from 500 Hz to 23.5 kHz in 500 Hz increments. Plot this against the theoretical response. You can generate the theoretical response from the SciPy `scipy.signal.firwin()` function. To generate the filter coefficients, use

```
import numpy as np
```

[1]The ADC on our boards is 12-bit, so this test is on the extreme end, but it is often good engineering practice to shoot for more conservative tolerances.

```
from scipy import signal

h_theoretical = signal.firwin(NUM_TAPS, np.pi/3)
```

Use dB for the magnitude response and degrees for the phase response.

Questions and tasks

- How many taps were you able to use with your implementation?
- How did you optimize this implementation?
- Plot the theoretical frequency response against the measured frequency response. How do they differ?

7.2.4 A final note

You might have noticed that you could use a similarly long or even longer FIR filter in the exercises in Chapter 3. This might feel backward because fixed-point arithmetic is supposed to be more efficient than floating-point. However, a lot of effort has gone into optimizing floating-point arithmetic. Our board does come with an FPU, which would make the operations rather zippy. The Oppenheim book states

> Nowadays, digital filtering of multi-media signals is often implemented on personal computers or workstations that have very accurate floating point numerical representations and high-speed arithmetic units. In such cases, the quantization issues...are generally of little or no concern. However, in high-volume systems, fixed point arithmetic is generally required to achieve low cost.

Despite the promise of floating-point arithmetic, as an engineer, you will not be able to assume the hardware you will work with will have a decent FPU, so exploring these implementations is worth our while!

7.3 Deliverables

7.3.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots. Your code should be submitted to GitHub. Additionally, you must demonstrate your functioning board at the beginning of the lab section.

7.3.2 Grading

The lab assignment will have the following grade breakdown.

- Filter stem plots: 10pts.
- Pole/zero plot: 10pts.

- Magnitude response plot: 5pts.
- Phase response plot: 5pts.
- `fir_fixed()`: 25pts.
- System efficiency: 10pts.
- Measured frequency response: 15pts.
- Write-up: 20pts.

Chapter 8

IIR filtering and sound restoration

Audio digital signal processing has a rich history in the state of Utah. In the late 1960's and throughout the 1970's, Thomas Stockham pioneered many advances in audio DSP. At the time, he was a faculty member in the Department of Electrical Engineering at the University of Utah. Still, he founded Soundstream Inc., one of the first digital audio recording companies. Some of his notable work includes his restoration of Enrico Caruso recordings and his work on the deleted 18 minutes from the Nixon tapes during Watergate.

The cause of the buzzing audio can vary depending on your signal path. The most likely culprit lies with cabling, such as grounding issues, electromagnetic interference, improper shielding, poor wire connections, etc. Long cable runs with the audio signal and power cables intertwined can also cause audible interference, especially with unbalanced cables. The recording medium can also lead to high-frequency noise corruption if the recording media (e.g., vinyl, acetate, tape) degrades. Of course, there are other instances where audio can be purposely corrupted for political purposes. This exercise gives you an audio file `noisy_audio.wav` corrupted with high-frequency noise. You must remove this high-frequency buzz to restore the original audio to a listenable form.

8.1 General IIR filtering on STM32F769I

8.1.1 Implementing the IIR filter

Before we do any IIR filter design, we need to understand the limits of our STM32F769I boards. We must write a function that performs IIR filtering in second-order stages, known as "biquad" stages. In Sec. 6.2.3 we implemented an IIR filter, but it was likely not in the second-order stages format. In Chapter 7, we learned about the impact of numerical errors and will want to avoid those as much as possible. Consequently, when implementing IIR filters, we will want to use a cascade of second-order stages (parallel implementations will also work, but we will use cascade here).

Remember from lecture that we will group real factors and complex conjugate pairs into 2^{nd}-order factors such that

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}. \tag{8.1}$$

Fig. 8.1 shows a 4^{th}-order IIR filter as a cascade of a pair of 2^{nd}-order subsystems. Implement an

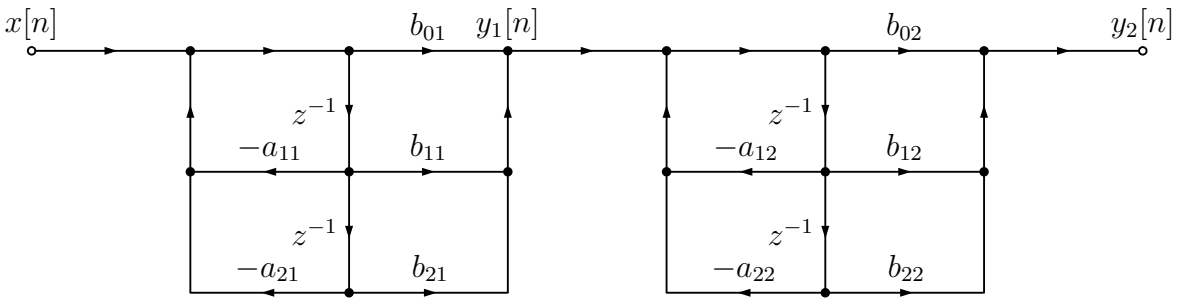

Figure 8.1: Cascade for a 4th-order system with a direct form II realization of each 2nd-order subsystem.

IIR filtering routine called `iir_sos()`, which a cascade of N_s biquad stages, in C. This function should have the following function declaration:

```
/**
 * @brief IIR filtering using cascaded biquad stages
 *
 * @param x input sample
 * @param b array of b coefficients as 1D array in groups of 3
 * @param a array of a coefficients as 1D array in groups of 3
 * @param num_stages number of biquad stages
 * @return float processed sample
 */
float iir_sos(float x, float *b, float *a, int num_stages)
```

This implementation is based on Eq. (8.1). Similar to our previous implementation of an IIR filter in Sec. 6.2.3, we will deviate from the Oppenheim book's notation and not automatically negate the a_k coefficients. This will be important later in the lab because we will use SciPy to generate our filter coefficients, and SciPy uses this convention.

Your filter should be general in the sense that it should be able to handle any arbitrary number of biquad stages (N_s). This number is defined for you as `NS`. You will have a unit test that has $N_s = 3$ biquad stages with predefined filter coefficients (`b_sos_test` and `a_sos_test` in `test/unittests.c`). Build your code and run the test.

```
$ ./test
[==========] Running 2 test cases.
...
[ RUN      ] ece5210_iir.filt_sos
[       OK ] ece5210_iir.filt_sos (17814294ns)
[==========] 2 test cases ran.
[  PASSED  ] 2 tests.
```

8.1.2 Board deployment

Deploy your code on your STM32F769I boards. You will need to know how many biquad stages you can use before you design your filter in Python. You will have many IIR filter coefficients in your code that you can test in a predefined global variable `b_len_test` and `a_len_test`. Adjust

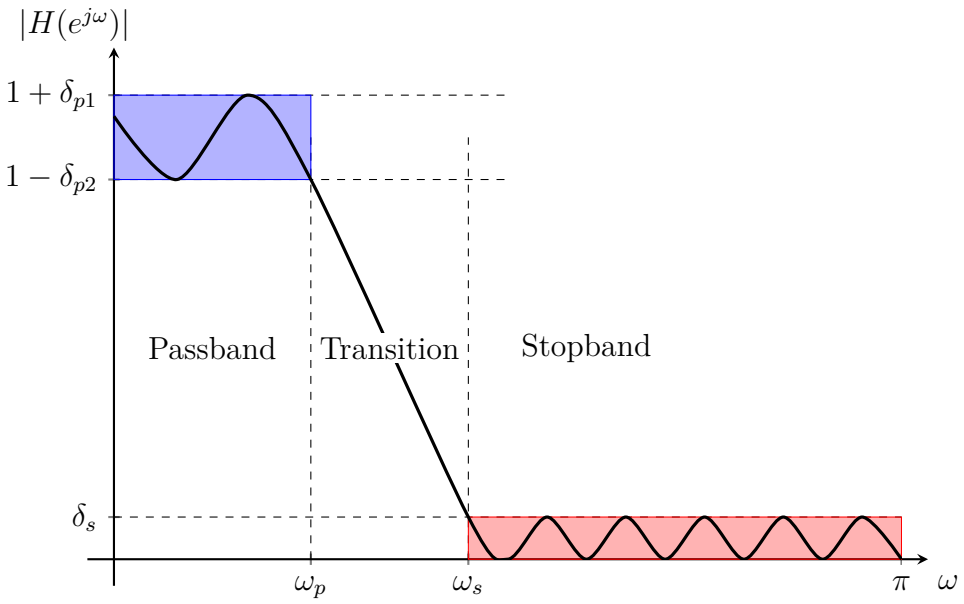

Figure 8.2: Low-pass filter tolerance scheme.

the value of N_S until the board can't handle the number of filter coefficients. How many biquad stages are you able to use? As a practical consideration, you may use linear or circular buffers. I found that linear buffer was faster in my implementation, but perhaps you will have different results.

Questions and tasks

- How many biquad stages can you use based on your implementation?

8.2 Filter design

In this lab, you will filter a speech from the 1960s corrupted with high-frequency noise. You will need to design a filter that is up to the task.

8.2.1 Filter design parameters

Human speech is generally confined to the spectra below 10 kHz. Researchers at Bell Labs showed that decreasing the cutoff frequency of a lowpass filter from 7 kHz to 2.85 kHz decreased the percentage of correctly identified syllables presented in quiet from 98% to 82% [1]. We will design a low-pass filter with the following specifications using these results as our guide. See Fig. 8.2 as a reference for the specification terms.

- $\delta_{p1} = \delta_{p2} = 0.1$ with $f_p = 2.4$ kHz (you may need to convert this frequency to the digital frequency depending on how you use the SciPy tools)
- $\delta_s = 0.001$ with $f_s = 3$ kHz

You may choose any IIR filter type you wish (e.g., Butterworth, Chebyshev I, Chebyshev II, Elliptic) so long as you can hit the filter specs. You are welcome to use the `scipy.signal` tools to design your filter. Make sure to read the API documentation before you start designing!

Plot the frequency response of your filter. Because we are working with an IIR filter, you must use the `signal.freqz()` function. Plot the magnitude in dB rather than just $|H(e^{j\omega})|$. In the second figure, plot the principal argument of the frequency response phase. On both plots, we are not as concerned with the stop-band, so please plot from $\omega \in [0, 5\,\text{kHz}]$. Make sure you are using grids, and you are labeling your plots. Does the magnitude response match the given specifications? How non-linear is the phase in the passband? How about the stopband? Is that an issue? Lastly, please make a plot of the zeros and poles. Please include the unit circle and appropriate labels on this plot.

The functions in the `signal` library can generate the filter coefficients directly rather than first generating the a_k and b_k coefficients. In my experience, this tends to be somewhat buggy. If you design your filter using the `output="sos"` option, please check each stage to ensure the coefficients make sense. I find that designing with `output="ba"` and then converting to second-order-stage coefficients with `signal.tf2sos()` seems to work well.

Save the filter coefficients as global variables in your `ece5210.c` file. Name these arrays `a_sos` and `b_sos`. Save the array length as a constant, defined as `LEN_SOS_FILT` in the `ece5210.h` file. You will need these values to pass the filter specifications unit test.

There is a unit test checking to make sure your filter design meets the design specifications.

```
$ ./test
[==========] Running 2 test cases.
[ RUN      ] ece5210_iir.filt_specs
Passband max: 1.000203
Passband min: 0.909732
Stopband max: 0.003964
[       OK ] ece5210_iir.filt_specs (7734ns)
...
[==========] 2 test cases ran.
[  PASSED  ] 2 tests.
```

Questions and tasks

- Plot the frequency response of $H(e^{j\omega})$ from $\omega \in [0, 5\,\text{kHz}]$. Make sure your graphs are properly labeled.
- Does the magnitude response match the given specifications?
- How non-linear is the phase in the passband?
- How about the stopband, and is that an issue?
- Plot the filter poles and zeros with labels and the unit circle.

8.2.2 Filtering the audio

Copy the second-order filter coefficients to your C code in `ece5210.c` to use in your filter. Make sure you don't mix up the coefficients! Do a frequency sweep from $[150\,\text{Hz}, 5\,\text{kHz}]$ in $150\,\text{Hz}$ increments. Record the magnitude and phase and plot this overlaid your plot from Sec. 8.2.1 to verify the actual performance matches your designed filter.

If your filter is performing as you think it should be, connect the supplied 3.5mm auxiliary cable to the board's line-in headphone jack to your computer's headphone output. Connect a set of headphones to the board's headphone output jack. Play the downloaded `noisy_audio.wav` file[1]. The right channel should have a lot of high-frequency noise. You may or may not be able to discern the underlying speech. The left channel should be filtered. There should be a huge difference! If you cannot tolerate the noisy audio, comment out the `#define PASSTHROUGH_RIGHT` macro in `ece5210.h`. This will copy the processed signal to both the right and left channels of the audio output. Please identify who the speaker is, where the speech was given, and the date[2]

Questions and tasks

- Record the filter's frequency response from $[150\,\text{Hz}, 5\,\text{kHz}]$ in $150\,\text{Hz}$ increments. Overlay this data over your previous plot.
- Who is the speaker in the audio? Where was the address given, and what was the date?

8.3 Deliverables

8.3.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots. Your code should be submitted to GitHub. Additionally, you must demonstrate your functioning board at the beginning of the lab section.

8.3.2 Grading

The lab assignment will have the following grade breakdown.

- `iir_sos()`: 30pts.
- `filt_specs()`: 10pts.
- Number of biquad stages you can use: 5pts.
- Plot $H(e^{j\omega})$ in Python: 10pts.
- Pole/zero plot: 5pts.
- Measured frequency response: 15pts.
- Properly filtering the audio: 15pts.
- Write-up: 10pts.

[1] Some laptops—e.g., Macs—will auto-detect the impedance of the load and will auto-adjust the output voltage. As a result, you will not be able to get any output from them. If this happens, use the lab computer's audio output.

[2] It is a very famous speech. Just Google the text, and you should figure it out quickly.

Chapter 9

FIR design and digital crossovers

An audio crossover is a device or a circuit separating an audio signal into two or more frequency bands, sending each band to a specific speaker driver, such as a woofer, tweeter, or midrange driver. The purpose of an audio crossover is to ensure that each speaker driver only reproduces the frequencies it is best suited for, resulting in a more accurate and balanced sound reproduction.

Historically, there have been two types of audio crossovers: passive and active. Passive crossovers are simple circuits that use inductors and capacitors to filter the audio signal and direct it to the appropriate speaker driver. They are typically used in speaker systems with two or three drivers. On the other hand, active crossovers are implemented with analog circuitry, typically using op-amps to realize specific types of circuit topologies. Switches or plugin modules select different crossover frequencies. This type of active crossover is limited because each filter must be realized with a physical circuit. For example, making the crossover slope steeper would require additional analog circuitry—which is not easy once a unit is in the field.

With modern DSP technology, crossovers can be implemented entirely with digital computation. This means the audio processing can be changed quickly without hardware changes. The amount of audio processing is limited only by the DSP power available. Digital crossovers also support direct input from a digital source, such as a computer. Fig. 9.1 shows a diagram of a typical system configuration, where volume control can be done digitally either in the source or in the crossover itself.

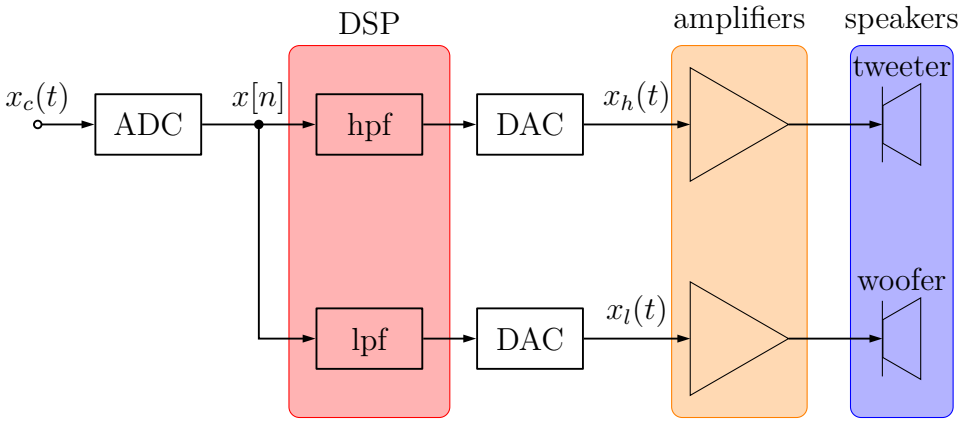

Figure 9.1: System block diagram of a digital crossover.

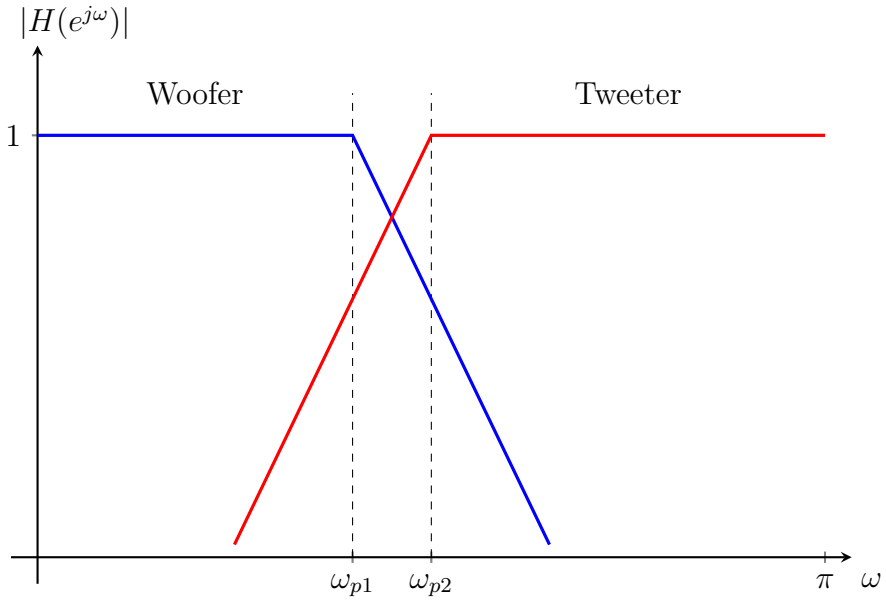

Figure 9.2: Sketch of frequency response for simple two-way crossover.

In this exercise, we will implement a simple two-way digital crossover in our STM32F769I boards with FIR filters, which you will design. You will test this implementation with our audio source—i.e., your laptop headphone output using the audio source of your choice.

9.1 Optimized FIR filtering on the STM32F769I

9.1.1 Optimized linear phase FIR implementation

In this exercise, we will work with Type I filters where M is even and the number of taps is odd. Recall that this class of FIR filters is symmetric such that $h[n] = h[M-n]$. We can exploit this to reduce our system's floating point multiplications. A typical FIR implementation can be decomposed into

$$\begin{aligned} y[n] &= \sum_{k=0}^{M} h[k]x[n-k] \\ &= \sum_{k=0}^{M/2} h[k]x[n-k] + h[M/2]x[n-M/2] + \sum_{k=M/2+1}^{M} h[k]x[n-k] \\ &= \sum_{k=0}^{M/2-1} h[k]x[n-k] + h[M/2]x[n-M/2] + \sum_{k=0}^{M/2-1} h[M-k]x[n-M+k]. \end{aligned}$$

Finally, because $h[k] = h[M-k]$ we can combine

$$y[n] = \sum_{k=0}^{M/2-1} h[k](x[n-k] + x[n-M+k]) + h[M/2]x[n-M/2] \tag{9.1}$$

which nearly halves the number of floating point multiplications needed for FIR filtering! A signal flow diagram of this filter is seen in Fig. 9.3 which shows how the delayed input samples are added *prior* to multiplication with the filter taps.

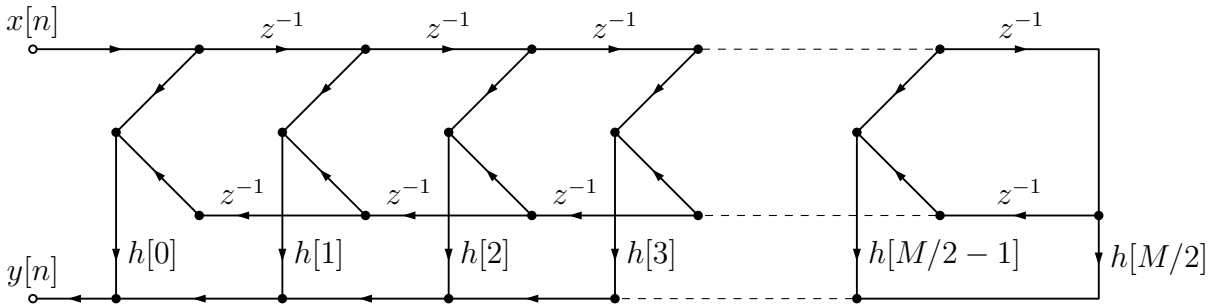

Figure 9.3: Direct form structure for an FIR linear-phase system when M is an even integer.

You must implement this form of FIR in the `Core/Src/ece5210.c` file. We will need to implement this twice: once for a low-pass filter and once for a high-pass filter. The function declaration should look like the following:

```
/**
FIR implementation of a low-pass filter using an efficient
low-pass filter structure.

   * @param x input sample
   * @param *h array of filter coefficients
   * @param M filter length
   * @return output sample
*/
float lpf(float x, float *h, uint8_t M);

/**
FIR implementation of a high-pass filter using an efficient
low-pass filter structure.

   * @param x input sample
   * @param *h array of filter coefficients
   * @param M filter length
   * @return output sample
*/
float hpf(float x, float *h, uint8_t M);
```

You can choose which data structure you want to represent the delay line. You will have unit tests to test the fidelity of your filter using FIR filters with random coefficients. Successful testing will look like the following:

```
$ make
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../
 Core/Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3
 -I ../Core/Inc
gcc -o test unittests.o ece5210.o -lm
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/
 Inc
```

```
gcc -o sandbox sandbox.o ece5210.o -lm

$ ./test
[==========] Running 2 test cases.
[ RUN      ] ece5210_fir.fir_half
[       OK ] ece5210_fir.fir_half (32749ns)
[ RUN      ] ece5210_fir.fir_least_squares

Testing to make sure no firls()
.
Testing fir_least_squares() function
.

 ----------------------------------------------------------------------

Ran 2 tests in 0.006s
```

9.1.2 Determining system capability

Unlike previous exercises, we will be processing both channels simultaneously. In `ece5210.c` you will have two functions: `process_function_left()` where we will low-pass filter the input samples, and `process_sample_right()` where we will high-pass filter the input samples. The `process_sample_left()` function will always be on. However, you can toggle the right channel behavior between a pass-through or signal processing by defining `PASSTHROUGH_RIGHT` in `Core/Inc/ece5210.h`.

We will be The `ece5210.c` file has a function `void init_firwin(void)` which will populate `h_lp[NUM_TAPS]` and `h_hp[NUM_TAPS]` which are simple Hamming-windowed sinc low- and high-pass filters with cutoff frequency $\omega_c = \pi/2$. Put your `lpf()` function (with `h_lp` as the filter coefficients) in `process_sample_left()` and your `hpf()` function in `process_sample_right()` (with `h_hp` as the filter coefficients). Ensure you comment out `#define PASSTHROUGH_RIGHT` to ensure you get dual-channel behavior and then hook your system up to the function generator and oscilloscope. Use a sine wave as an input, adjust the frequency from some low-frequency to Nyquist, and observe what happens to both sine waves. You should see one shrink as the grows as you pass the cutoff frequency. Adjust `NUM_TAPS` in `ece5210.c` to determine the maximum filter length you can get away with on this board and your particular implementation.

Questions and tasks

- What is the longest filter length you can use if you are simultaneously processing the right and left channels of the output?

9.2 Audio processing and filter design

9.2.1 Least-squares filter design

In class, we discussed the least-squares linear phase FIR filter design method. This method allows us to specify a frequency grid ω_k and define a specific frequency response for some non-causal

$\hat{h}[n]$

$$H(\omega_k) = \sum_{n=-M/2}^{M/2} h[n]e^{-j\omega_k n},\ k = 0, 1, 2, \ldots, L-1,\ L \gg M.$$

We do not worry about the causality about $\hat{h}[n]$ for now because we will correct it later on. If $\hat{h}[n]$ is a Type I FIR filter, then it will have even symmetry and we see that the frequency response reduces to a sum of cosines

$$H(e^{j\omega_k}) = \hat{h}[M/2] + \sum_{n=1}^{M/2} 2\hat{h}[n]\cos(\omega_k n).$$

This allows us to specify a frequency response at various frequencies and cast it as a system of equations. You will generally ignore the transition bands and focus on frequencies you care about. In other words, if some ω_k is in the transition band, do not use that frequency in your system of equations. We can cast this in linear algebra form

$$\underbrace{\begin{bmatrix} 1 & 2\cos(\omega_0) & 2\cos(2\omega_0) & \ldots & 2\cos((M/2)\omega_0) \\ 1 & 2\cos(\omega_1) & 2\cos(2\omega_1) & \ldots & 2\cos((M/2)\omega_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2\cos(\omega_{L-1}) & 2\cos(2\omega_{L-1}) & \ldots & 2\cos((M/2)\omega_{L-1}) \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \hat{h}[0] \\ \hat{h}[1] \\ \hat{h}[2] \\ \vdots \\ \hat{h}[M/2] \end{bmatrix}}_{\mathbf{h}} = \underbrace{\begin{bmatrix} H(e^{j\omega_0}) \\ H(e^{j\omega_1}) \\ H(e^{j\omega_2}) \\ \vdots \\ H(e^{j\omega_{M/2}}) \end{bmatrix}}_{\mathbf{d}}$$

We notice that matrix $\mathbf{A}$ will be a "skinny" matrix because it will have more rows than columns. There won't be an exact solution to this system of equations, but we can approximate it by solving the optimization problem

$$\mathbf{h}^\star = \underset{\mathbf{h}}{\operatorname{argmin}} \|\mathbf{A}\mathbf{h} - \mathbf{d}\|_2^2.$$

There are numerous ways to solve this. One of the most common approaches is to solve for the pseudo-inverse which is

$$\mathbf{h} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{d}.$$

The matrix product $\mathbf{A}^T\mathbf{A}$ will be square, semi-positive definite, and will be able to solve an inverse relatively easily. Using this approach, we can use `np.linalg.solve()` to solve for $\mathbf{h}$. Alternatively, you can skip the pseudo-inverse approach and solve directly using `np.linalg.lstsq()`. This approach solves the least squares problem using QR decomposition followed by back substitution. Regardless of how you solve for $\mathbf{h}$, this provides only half of the filter coefficients

$$\mathbf{h} = \begin{bmatrix} \hat{h}[0] & \hat{h}[1] & \hat{h}[2] & \ldots & \hat{h}[M/2] \end{bmatrix}^T.$$

We can construct the full FIR filter

$$h[n] = \{\hat{h}[M/2], \hat{h}[(M/2)-1], \ldots, \hat{h}[2], \hat{h}[1], \hat{h}[0], \hat{h}[1], \hat{h}[2], \ldots, \hat{h}[M/2]\},\ n = 0, \ldots, M$$

In a Python script `test/fir_filters.py` write a Python function to compute FIR filter coefficients using the least-squares approach. The function definition should look like the following:

```
def fir_least_squares(n_taps, bands, desired):
    """
    FIR filter design using least-squares error minimization.
    Calculate the filter coefficients for the linear-phase finite
    impulse response (FIR) filter, which has the best approximation
    to the desired frequency response described by `bands` and
    `desired` in the least squares sense (i.e., the integral of the
    weighted mean-squared error within the specified bands is
    minimized).

    Parameters
    ----------
    n_taps : int
        The number of taps in the FIR filter. `n_taps` must be odd.
    bands : array_like
        A monotonic nondecreasing sequence containing the band
   edges in
        Hz. All elements must be non-negative and be representative
    of the
        digital frequencies. The bands must be specified as an nx2
   sized
        2D array or a list of lists, where n is the number of bands
   ,
        e.g, `np.array([[0, 1], [2, 3], [4, 5]])`.
    desired : array_like
        A sequence the same size as `n` above contains the desired
   gain
        of each band.

    Returns
    -------
    coeffs : ndarray
        Coefficients of the optimal (in a least squares sense) FIR
   filter.

    """
```

You will have a unit test to check the accuracy of your function against the SciPy implementation. Please do not just have your function call the SciPy API; there is a unit test to check for this. Successful testing will look like the following:

```
$ make
gcc -c -o unittests.o unittests.c -Wall -Wconversion -O3 -I ../Core
   /Inc
gcc -c -o ece5210.o ../Core/Src/ece5210.c -Wall -Wconversion -O3 -I
    ../Core/Inc
gcc -o test unittests.o ece5210.o -lm
gcc -c -o sandbox.o sandbox.c -Wall -Wconversion -O3 -I ../Core/Inc
gcc -o sandbox sandbox.o ece5210.o -lm
```

```
$ ./test
[==========] Running 2 test cases.
[ RUN      ] ece5210_lab08.fir_half
[       OK ] ece5210_lab08.fir_half (995785ns)
[ RUN      ] ece5210_lab08.fir_least_squares

Testing to make sure no firls()
.
Testing fir_least_squares() function
.
-------------------------------------------------------------------------------

Ran 2 tests in 2.312s

OK
[       OK ] ece5210_lab08.fir_least_squares (4112125007ns)
[==========] 2 test cases ran.
[  PASSED  ] 2 tests.
```

9.2.2 Audio crossover design and implementation

We are now going to design and deploy a simple two-way crossover. First, we are going to use our `fir_least_squares()` function to design a low-pass filter with the following specifications:

- Passband defined from 0 Hz to 4.5 kHz with a gain of 1.
- Stopband defined from 5 kHz to Nyquist with a gain of 0.
- Use the maximum number of taps possible based on your implementation of `lpf()`.

Next, design a high-pass filter with the following specifications:

- Passband defined from 4.5 kHz to Nyquist with a gain of 1.
- Stopband defined from 0 Hz to 4 kHz with a gain of 0.
- Use the maximum number of taps possible based on your implementation of `hpf()`.

Plot the frequency response (magnitude and phase) for both filters. Please plot the magnitude response in dB.

Copy the filter coefficients as global arrays in `ece5210.c` and use these coefficients as inputs in your filter functions. Keep the right channel in pass-through mode and record the frequency response for both filters using the oscilloscope. You must change and compile your code several times to use the left channel to record the response for both the high- and low-pass filters.

After you have recorded the frequency response, put the `lpf()` function and associated filter coefficients into `process_sample_left()` and the `hpf()` and associated filter coefficients into `process_sample_right()`. Use your laptop as an audio source and use the 3.5 mm aux cable as an input into the board. Use a pair of headphones in the headphone jack and play some music. What do you hear?

You can also check your implementation using the pure tone generator here.

Questions and tasks

- Plot the theoretical frequency response for both filters. Please record the magnitude response in dB and the principal argument of the phase response.
- Perform a frequency sweep from 100 Hz to Nyquist and plot this data overlaid with the theoretical response. Do they match?
- How well does the crossover separate the high and low-frequency audio?

9.3 Deliverables

9.3.1 Demonstration and check-off

Please demonstrate how your code works by using any audio source with the left channel playing the low frequencies and the right channel playing the high-frequencies.

9.3.2 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise. Make sure to address the specific questions raised throughout the chapter.

In the appendices, please include the requested plots. Your code should be submitted to GitHub. Additionally, you must demonstrate your functioning board at the beginning of the lab section.

9.3.3 Grading

The lab assignment will have the following grade breakdown.

- `lpf()`: 20pts.
- Longest filter length: 10pts.
- `fir_least_squares()`: 20pts.
- Theoretical frequency response: 15pts.
- Measured frequency response: 15pts.
- Write-up: 20pts.

Bibliography

[1] Norman R French and John C Steinberg. “Factors governing the intelligibility of speech sounds”. In: *The Journal of the Acoustical Society of America* 19.1 (1947), pp. 90–119.

[2] James C Huhta and John G Webster. “60-Hz interference in electrocardiography”. In: *IEEE Transactions on Biomedical Engineering* 2 (1973), pp. 91–101.

[3] Peter R Rijnbeek, Jan A Kors, and Maarten Witsenburg. “Minimum bandwidth requirements for recording of pediatric electrocardiograms”. In: *Circulation* 104.25 (2001), pp. 3087–3090.