5월 2주차 보고서 (2024.05.06~2024.05.12)

하드웨어

💡 2024.05.07(화) : 18:00~22:30 (4.5h)
참여자: 김지윤, 정지원
오프라인 개발

💡 2024.05.08(수) 8:30~16:00(7.5h) / 18:00~19:00 (1h)
참여자: 김지윤, 정지원
오프라인 개발

💡 2024.05.09(목) : 15:00~21:00 (6h) / 23:00 ~ 2024.05.09(금) 6:00 (7h)
참여자: 김지윤, 정지원
오프라인 개발 및 중간 발표 준비

0507 화요일

MAX98357a 모듈과 라즈베리파이 다시 연결 후 다시 제어 시도

하드웨어 연결

1. **Raspberry Pi 핀아웃**:
   - GPIO 18: I2S BCLK
   - GPIO 19: I2S LRCK
   - GPIO 21: I2S DATA
   - GND: 접지
   - 5V 또는 3.3V: 전원
2. **MAX98357A 연결**:
   - BCLK (MAX98357A) → GPIO 18 (Raspberry Pi)
   - LRCK (MAX98357A) → GPIO 19 (Raspberry Pi)
   - DIN (MAX98357A) → GPIO 21(Raspberry Pi)
   - GND (MAX98357A) → GND (Raspberry Pi)
   - VCC (MAX98357A) → 5V (Raspberry Pi)
   - 스피커를 MAX98357A의 스피커 출력 단자에 연결

I2S 활성화

```
dtparam=i2s=on
```

WAV 파일 재생

```
sudo apt update
sudo apt install alsa-utils
aplay -D plughw:0,0 test1.wav
```

Python을 사용한 WAV 파일 재생:

```
import pygame
pygame.mixer.init()
sound = pygame.mixer.Sound("test1.wav")
sound.play()
while pygame.mixer.get_busy():
    pass
```

→ 결과 : 모듈과 연결된 스피커로 소리 자체가 출력이 되지 않음

프로토콜 문제 or 하드웨어 연결 문제

0508 수요일

1. 하드웨어 다시 연결 시도

2. Raspberry Pi I2S 설정

3. 소프트웨어 설치

```
sudo apt update
sudo apt install alsa-utils
```

4. WAV 파일 재생

```
aplay -D plughw:0,0 test1.wav
```

최종 확인

```
pi@raspberrypi:~ $ aplay -D hw:MAX98357A,0 test1.wav
Playing WAVE 'test1.wav' : Signed 16 bit Little Endian, Rate
```

→터미널에서 명령어를 통해 증폭 모듈 출력 성공 (wav 파일 재생)

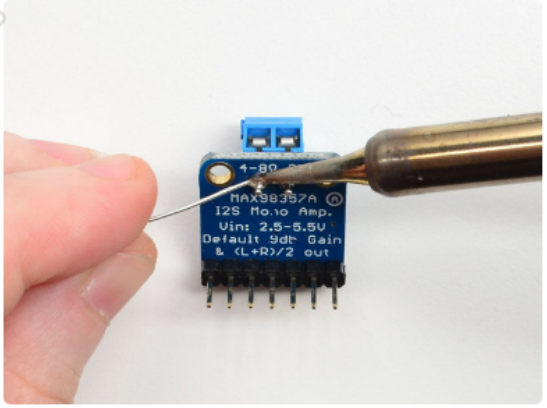
그러나 노이즈가 굉장히 심하고, 추측하건데 증폭자체가 너무 커서 소리가 깨지는 현상이 발생하는 거 같음

0509 목요일

목표: 증폭모듈을 통한 제대로 된 소리 출력

계속 소프트웨어적인 부분으로만 해결하다가 하드웨어적인 부분이 이상할 수도 있다는 생각에 납땜 다시 시도

→ 이전보다 훨씬 소리가 잘 뚜렷하게 들림



Solder in both pins with plenty of solder!

그러나 여전히 소리가 찢어지는 현상 발생

→ 100옴 저항을 꽂았더니 소리가 아예 들리 지 않음

가변 저항으로 저항의 세기를 최대한 약하게 했더니 소리의 찢어지는 소리 없이 제대로 된 오디오 파일 재생

중간발표 이후

중간발표 hw 관련 피드백

- 진행한 시간에 비해 캡스턴 진행도가 너무 느립니다. 만든 후 테스트까지 진행하여야 하는데 1달안에 진행할 수 있을지 의문이 듭니다.
- 증폭된 소리가 실제 사람 소리에 비해 기계음처럼 들립니다. 이를 보완할 방법이 필요해 보입니다.
- 출력 음성 퀄리티가 매우 낮아, 공연 현장에서 사용이 불가능한 수준임
- 입력 마이크 등의 장비가 공연 현장을 염두에 두고 설계된 것이 아님

소프트웨어

💡 2024.05.07(화)
11:00~16:00(5시간),
20:00~22:30(2시간30분)
참여자: 이다은
오프라인 개발

💡 2024.05.07(화)
14:30~16:30 (2시간)
18:00~22:30 (4시간 30분)
참여자: 박수진
오프라인 개발

💡 2024.05.08(수)
10:00~12:00(2시간),
20:00~22:30(2시간30분)
참여자: 이다은
오프라인 개발

💡 2024.05.08(수)
13:30~16:30 (3시간)
22:30~24:30 (2시간)
참여자: 박수진
오프라인 개발

💡 2024.05.09(목)
09:00~15:00(6시간),
18:30~20:00(1시간 30분)
참여자: 이다은
오프라인 개발

💡 2024.05.09(목)
17:00~21:30 (4시간 30분)
22:00~24:30 (2시간 30분)
참여자: 박수진
오프라인 개발

PiDTLN(노이즈 제거)

💡 기본 모델은 16kHz로 훈련 되어 있으며, 16kHz 샘플링 주파수만을 지원한다. 하지만, 우리가 사용하는 오디오 장치의 샘플링 주파수는 44.1kHz다. 따라서, 입력을 44.1kHz로 주고, 16kHz로 다운샘플링하여 모델에 적용한 후, 다시 44.1kHz로 업샘플링하여 출력해야 한다.

위의 방식대로 노이즈 제거를 계속 시도 하였으나, 원하는 결과를 얻을 수 없었다.

[출력 시 문제점]

1. **버퍼 사이즈:**

   데이터를 주고 받을 때, 리샘플링 과정에서 버퍼의 사이즈 등 맞추어야 할 요소가 많아서 오류가 지속적으로 발생하였다. 버퍼 사이즈가 너무 작거나 크면 데이터 관리 및 처리가 어려워진다. 입출력의 버퍼 사이즈 조정을 통해 데이터가 충분히 처리될 수 있도록 해야 한다.

2. **처리 속도:**

   데이터 전달 문제를 해결한 이후에도, 출력 결과 'input overflow output underflow'가 발생하였다. 이는 실시간으로 들어오는 데이터는 빠르고 많은데 이를 처리하지 못하고 그냥 넘어가다 보니 출력이 부족한 상태가 되어 노이즈 제거가 제대로 되지 않음을 의미한다. 예상한 바로는 리샘플링 과정을 거치는 동시에 실시간 처리가 이루어져야 하기 때문이다.

3. **노이즈 제거 X:**

실시간 오디오 스트림동안 녹음 파일로 저장하여 실행하였을 때, 기존의 심한 노이즈는 제거 되었으나 출력 음성 자체에 대한 오디오가 존재하였다. 또한, 테스트 용 마이크를 사용하여 마이크 성능에 따른 기계 잡음이 함께 출력되었다.

[수정 방향성]

1. **리샘플링 라이브러리:**

   44.1kHz를 16kHz로 다운샘플링하고, 출력 시에는 다시 44.1kHz로 업샘플링하는 과정에서 발생하는 처리 지연을 최소화하기 위해 리샘플링 라이브러리를 사용해야 한다. librosa 라이브러리와 scipy.signal 라이브러리를 사용하였으나, librosa 라이브러리에 중점을 두고 수정을 진행한다.

2. **스레드 수 조정:**

   처리 스레드의 수를 조정하여, 병렬 처리로 진행한다.

---

1. **resampy**
   - 고품질 리샘플링을 위해 사인 곱셈 방식을 사용하며, 오디오 전용 작업에 특화되어 있다.

     → 에일리어싱을 최소화하면서도 주파수 응답을 잘 보존한다.

     ```
     import resampy
     # 44.1kHz에서 16kHz로 다운샘플링
     audio_downsampled = resampy.resample(audio_original, 44100, 16000)
     # 처리 후, 16kHz에서 44.1kHz로 업샘플링
     audio_upsampled = resampy.resample(audio_processed, 16000, 44100)
     ```

2. **librosa**
   - 오디오 분석 및 다른 처리 기능을 제공하기 때문에 더 복잡한 오디오 처리 파이프라인을 구성하고자 할 때 유용하다.

     → resampy를 내부적으로 사용하여 리샘플링을 수행한다.

```
import librosa
# 44.1kHz에서 16kHz로 다운샘플링
audio_downsampled = librosa.resample(audio_original,
44100, 16000)
# 처리 후, 16kHz에서 44.1kHz로 업샘플링
audio_upsampled = librosa.resample(audio_processed, 1
6000, 44100)
```

3. **scipy.signal**
   - FFT 기반의 리샘플링을 제공하며, 계산 속도가 매우 빠르다는 장점이 있다.

     → 품질이 조금 더 낮을 수 있지만, 처리 속도가 매우 중요한 경우 유리하다.

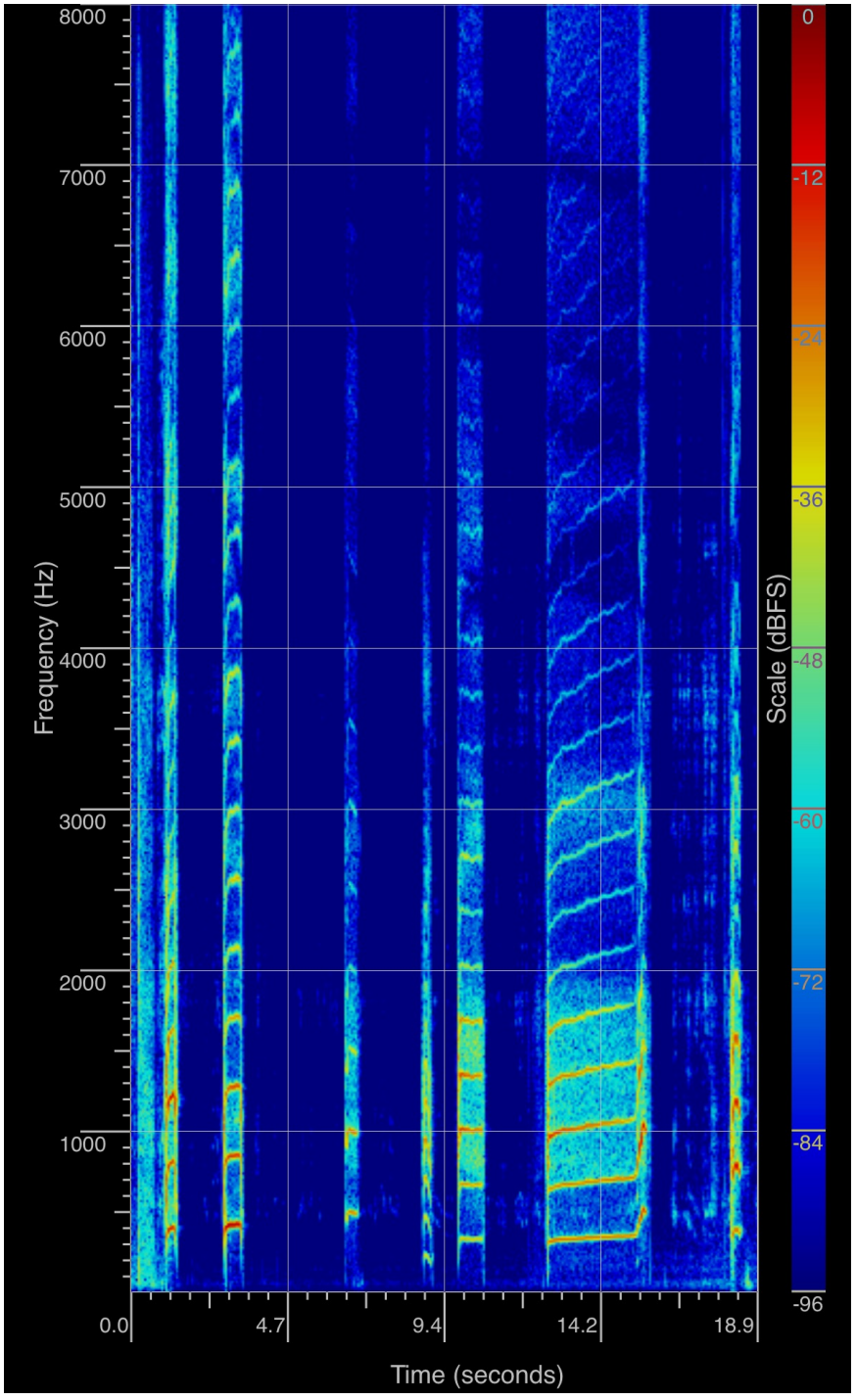
```
from scipy.signal import resample
# 44.1kHz에서 16kHz로 다운샘플링
audio_downsampled = resample(audio_original, int(len
(audio_original) * (16000 / 44100)))
# 처리 후, 16kHz에서 44.1kHz로 업샘플링
audio_upsampled = resample(audio_processed, int(len(a
udio_processed) * (44100 / 1
```

> 현재 개발에서는, 실시간 처리와 오디오 품질 모두 중요하다. 그렇기에 오디오 처리에 필요한 기능들을 제공하면서도, 내부적으로 resampy를 사용하여 고성능 리샘플링을 하는 librosa 라이브러리를 사용한다.

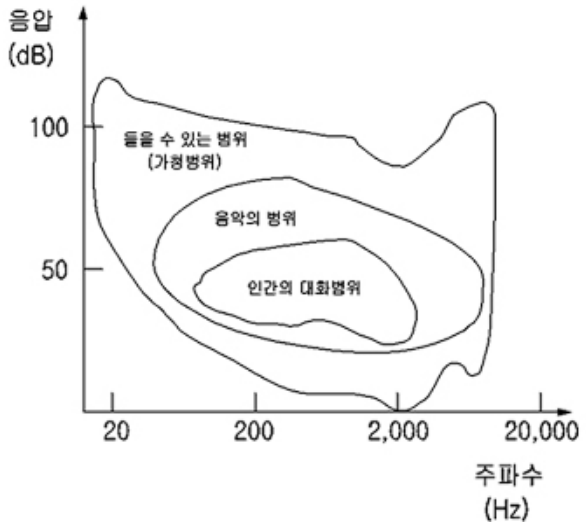
[새로운 노이즈 제거 방법 모색]

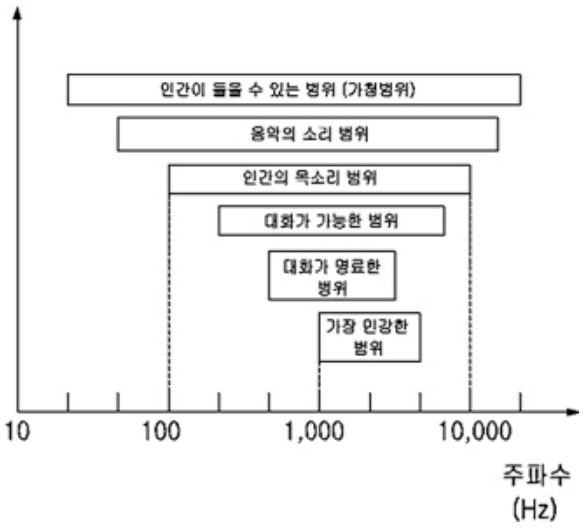
- 노이즈 제거는 RMS 계산과 음량 증폭을 위한 사전 단계이다.
- 노이즈가 많을 시, 음성과 함께 증폭되어 오디오의 품질이 떨어지게 된다.
- 그렇기에 음성 주파수보다 높은 주파수는 우선적으로 제거하고, 음성 주파수만을 뽑아서 증폭 시킨다. → **Low-pass filter**를 통해 해결하고자 한다.
- 사람이 낼 수 있는 음성의 주파수에는 한계가 존재한다. 이를 기준으로 높은 주파수를 제거한다.

- 아래 이미지에서 빨간색 계열이 실제 음성을 입력했을 경우다. 이 각각의 음성 주파수를 뽑아 내어 증폭하고자 한다.



SpectrumView 어플 사용





Low-pass filter

```
import numpy as np
import sounddevice as sd
from scipy.signal import butter, lfilter, lfilter_zi

def butter_lowpass(cutoff, fs, order=1):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='low', analog=F
    return b, a

def butter_lowpass_filter(data, b, a, zi):
    y, zo = lfilter(b, a, data, zi=zi)
    return y, zo

# 사용자 입력을 통한 오디오 장치 선택
input_device_id = 5
output_device_id = 5

# 샘플 레이트, Hz
fs = 44100
# 음성 주파수 범위의 상한, Hz
cutoff = 3400

# 필터 계수 및 초기 조건 설정
```

```
b, a = butter_lowpass(cutoff, fs, order=1)
zi = lfilter_zi(b, a)

def callback(indata, outdata, frames, time, status):
    if status:
        print(status)
    global zi
    # 필터 적용
    y, zo = butter_lowpass_filter(indata[:, 0], b, a, zi)
    zi = zo
    outdata[:, 0] = y

# 스트림 생성 및 실행, 지정된 입력 및 출력 장치 사용
with sd.Stream(callback=callback, samplerate=fs, channels=1, 
               input_device=input_device_id, output_device=ou
    print("Streaming started with low-pass filter. Press Ctrl
    input("Press Enter to exit...")  # Ctrl+C 또는 Enter 누르면
```

- 신호 처리 전문성:

  scipy.signal은 넓은 범위의 신호 처리 기능을 제공한다. 이는 필터 설계, 신호 분석, 시스템 응답 등 광범위한 신호 처리 작업을 지원한다.

- 실시간 처리 적합성:

  필터링과 같은 실시간 신호 처리에 필요한 기능들이 잘 최적화되어 있어, 시스템 성능에 민감한 실시간 응용 프로그램에 적합하다.

[조교님의 피드백]

> 특정 주파수 이상만 노이즈가 아닌, 신호와 비슷한 주파수의 nosie도 생길 것이다. 그렇기에 low-pass filter보다는 adaptive filter를 통해 실시간으로 제거하는게 더 맞는 방향성 같다.

Adaptive Filter (적응형 필터)

- 입력 신호의 통계적 특성이나 외부 환경이 변할 때 이에 적응하여 자동으로 조정되는 필터다.

- 주로 신호 예측, 잡음 제거, 에코 캔슬레이션 등에 사용된다.
- 자체적으로 최적의 필터링 성능을 유지하기 위해 필터 매개변수를 실시간으로 업데이트 한다.
- 가장 흔히 사용되는 알고리즘은 최소 평균 제곱(LMS, Least Mean Squares)과 재귀적 최소 제곱(RLS, Recursive Least Squares)이다. 이 알고리즘들은 입력 신호와 원하는 출력 신호 간의 오차를 최소화하는 방향으로 필터 계수를 지속적으로 조정한다.

Low-pass filter vs. Adaptive filter

Low-pass filter	Adaptive filter
고주파 노이즈를 제거하는 데 주로 사용	변화하는 환경에 맞춰 신호를 처리하는 데 사용
고정된 주파수 응답을 가짐	실시간으로 환경 변화에 따라 특성 변화

Adaptive filter

```
import numpy as np
import sounddevice as sd

class LMSFilter:
    def __init__(self, filter_order, mu):
        self.filter_order = filter_order
        self.mu = mu
        self.weights = np.zeros(filter_order)
        self.input_vector = np.zeros(filter_order)

    def process(self, input_signal, desired_signal):
        # 입력 신호 업데이트
        self.input_vector[:-1] = self.input_vector[1:]
        self.input_vector[-1] = input_signal

        # 필터 출력 계산
        output_signal = np.dot(self.weights, self.input_vecto

        # 오차 계산
        error_signal = desired_signal - output_signal

        # 가중치 업데이트
```

```
        self.weights += 2 * self.mu * error_signal * self.inp

        return output_signal

def audio_callback(indata, outdata, frames, time, status):
    global lms_filter
    if status:
        print(status)

    # 입력 데이터는 모노입니다.
    mono_data = indata[:, 0]  # 모노 데이터 접근

    filtered_signal = np.zeros_like(mono_data)
    for i in range(len(mono_data)):
        # 목표 신호를 입력 신호로 사용합니다.
        filtered_signal[i] = lms_filter.process(mono_data[i],

    # 스테레오 출력 설정
    outdata[:, 0] = filtered_signal  # 왼쪽 채널
    outdata[:, 1] = filtered_signal  # 오른쪽 채널

# 필터 매개변수 설정
filter_order = 10
mu = 0.01  # 학습률
lms_filter = LMSFilter(filter_order, mu)

# 오디오 스트림 설정: 입력은 모노, 출력은 스테레오
with sd.Stream(channels=(1, 2), callback=audio_callback):
    print("Press Ctrl+C to stop")
    try:
        while True:
            sd.sleep(500)  # 반복적으로 적은 시간 동안 대기하며, Ctr
    except KeyboardInterrupt:
        print("Stopped by user")
```

> FFT를 Single FFT로 output을 낼텐데, 실제 신호의 주기를 FFT sample window와 잘 맞춰야될 것이다. 맞지 않을 경우 실제보다 주파수 성분이 넓게 분포되는 형상이 발생할 것이기 때문이다. 그렇기에 Single FFT output을 바로 사용하고 않고 여러 기법을 추가해야 한다.

- *FFT를 사용할 때 입력 신호의 주기가 FFT 샘플 윈도우와 정확히 맞지 않으면* ***Spectral leakage****이 발생할 수 있다.*
- *신호가 FFT 윈도우의 길이로 완벽하게 나누어 떨어지지 않는 경우, 변환된 주파수 스펙트럼이 실제보다 넓게 퍼질 수 있다.*
- *이로 인해 실제로는 존재하지 않는 주파수 성분이 나타나거나, 실제 주파수 성분이 더 넓게 퍼져 보이는 현상이 발생한다.*

Spectral leakage(스펙트럼 누설)

- 신호를 스펙트럼 분석 했을 때, 원래의 신호에는 포함되어 있지 않은 주파수 성분이 관측되는 현상을 말한다.
- 특히 FFT(Fast Fourier Transform)를 사용할 때 발생하는 현상으로, 실제 신호의 주파수 구성 요소가 실제 주파수 대역 외부로 누설되어 나타나는 문제다.
- 신호의 주파수 성분이 불명확하게 보이게 하고, 주파수 해석의 정확도를 저하시킨다.

Spectral leakage의 발생 원인

1. 비주기적 신호:

   FFT는 신호가 주기적이라는 가정 하에 설계되었다. 즉, FFT가 처리하는 데이터 블록은 그 끝이 다시 처음과 연결되어 있을 것으로 가정한다. 만약 신호가 이러한 주기성을 만족하지 않을 경우, 신호가 FFT의 윈도우 크기에 정확히 맞지 않을 경우, 신호의 시작과 끝에서 불연속이 생기며 이는 스펙트럴 누설을 일으킨다.

2. 윈도우 크기와 신호의 주파수:

   특정 주파수의 신호 성분이 FFT 윈도우의 크기와 정확히 맞지 않을 때, 해당 주파수 성분은 여러 주파수 빈(bin)에 걸쳐 나타나게 된다. 이는 FFT가 계산하는 주파수 빈이 신호의 실제 주파수를 정확히 표현하지 못하는 경우에 발생하게 된다.

Spectral leakage의 영향

1. 주파수 해상도 저하:

   실제 주파수 성분이 여러 주파수 빈(bin)에 걸쳐 나타나므로, 주파수 성분들이 서로 겹쳐 보일 수 있다. 이로 인해 신호의 주파수 해상도가 저하되며, 정확한 주파수를 식별하기 어려워진다.

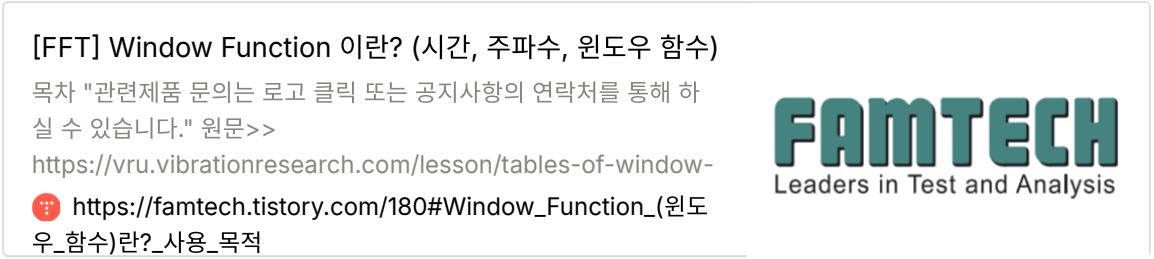
2. 신호의 강도 왜곡:

   원래 신호의 강도가 실제보다 낮게 나타나거나, 없는 주파수 성분이 나타나는 것처럼 보일 수 있다.

Spectral leakage 해결 방법

1. 윈도우 함수 사용:

   윈도우 함수는 신호의 양 끝을 서서히 0으로 감소시켜, 신호가 주기적이지 않아도 스펙트럼이 갑자기 끊기는 것을 방지한다.



2. 제로 패딩(Zero Padding):

   신호의 끝에 0을 추가하여 FFT의 샘플 수를 늘리는 방법이다. 이는 주파수 해상도를 개선하고 스펙트럴 누설을 줄이는 데 도움을 줄 수 있다.

3. 오버랩(Overlap):

   FFT를 적용할 때, 각 윈도우가 일부 겹치도록 설정한다.

Hamming window function

```
# ...

import librosa

# ...
```

```
args = parser.parse_args(remaining)

# Set some parameters
fs_original = 44100  # Original sampling frequency
fs_target = 16000    # Target sampling frequency for processi
block_len_ms = 32
block_shift_ms = 8

....

# FFT setup
if g_use_fftw:
    fft_buf = pyfftw.empty_aligned(512, dtype='float32')
    rfft = pyfftw.builders.rfft(fft_buf)
    ifft_buf = pyfftw.empty_aligned(257, dtype='complex64')
    irfft = pyfftw.builders.irfft(ifft_buf)

# Hamming window
hamming_window = np.hamming(block_len)  # 추가된 창 함수

t_ring = collections.deque(maxlen=100)

# Callback function updated with resampling and windowing
def callback(indata, outdata, frames, buf_time, status):
    global in_buffer, out_buffer, states_1, states_2, t_ring,

    if args.measure:
        start_time = time.time()

    if status:
        print(status)

    # Resample input from 44.1kHz to 16kHz
    indata_resampled = librosa.resample(indata.T, orig_sr=fs_

    # Process resampled input
    if args.channel is not None:
        indata_resampled = indata_resampled[:, [args.channel]
```

```
    if args.no_denoise:
        outdata[:] = indata_resampled
        if args.measure:
            t_ring.append(time.time() - start_time)
        return

    # Write to buffer
    in_buffer[:-block_shift] = in_buffer[block_shift:]
    in_buffer[-block_shift:] = np.squeeze(indata_resampled)

    # Apply Hamming window to buffer before FFT
    # 창 함수 적용 부분
    windowed_buffer = in_buffer * hamming_window

    # Calculate FFT of input block
    if g_use_fftw:
        fft_buf[:] = windowed_buffer
        in_block_fft = rfft()
    else:
        in_block_fft = np.fft.rfft(windowed_buffer)

    in_mag = np.abs(in_block_fft)
    in_phase = np.angle(in_block_fft)

    # Reshape magnitude to input dimensions
    in_mag = np.reshape(in_mag, (1,1,-1)).astype('float32')

    # Set tensors to the first model
    interpreter_1.set_tensor(input_details_1[1]['index'], sta
    interpreter_1.set_tensor(input_details_1[0]['index'], in_
    # Run calculation
    interpreter_1.invoke()

    # Get the output of the first block
    out_mask = interpreter_1.get_tensor(output_details_1[0]['
    states_1 = interpreter_1.get_tensor(output_details_1[1]['
```

```
    # Calculate the IFFT
    estimated_complex = in_mag * out_mask * np.exp(1j * in_ph
    if g_use_fftw:
        ifft_buf[:] = estimated_complex
        estimated_block = irfft()
    else:
        estimated_block = np.fft.irfft(estimated_complex)

    # Reshape the time domain block
    estimated_block = np.reshape(estimated_block, (1,1,-1)).a

    # Set tensors to the second block
    interpreter_2.set_tensor(input_details_2[1]['index'], sta
    interpreter_2.set_tensor(input_details_2[0]['index'], est

    # Run calculation
    interpreter_2.invoke()

    # Get output tensors
    out_block = interpreter_2.get_tensor(output_details_2[0][
    states_2 = interpreter_2.get_tensor(output_details_2[1]['

    # Write to buffer
    out_buffer[:-block_shift] = out_buffer[block_shift:]
    out_buffer[-block_shift:] = np.zeros((block_shift))
    out_buffer  += np.squeeze(out_block)

    # Resample output from 16kHz back to 44.1kHz
    outdata_resampled = librosa.resample(out_buffer[:block_sh

    # Output to soundcard
    outdata[:] = np.expand_dims(outdata_resampled, axis=-1)

    if args.measure:
        t_ring.append(time.time() - start_time)

# ...
```

TODO

- [ ] PiDTLN을 사용할 것인지 VS. 자체 개발을 진행할 것인지
- [ ] Adaptive filter 사용하여 시도
- [ ] Spectral leakage 해결을 위해 window function과 zero padding 사용하여 시도

PiDTLN에 적용도 해보고, 자체적으로 개발도 진행해보자.

RMS

- **그래프 출력 함수 구현**
  - [x] ~~입력된 음성을 그래프로 어떻게 보여줄건지(x축, y축 등 기준값 선정)~~
  - [x] ~~input data를 wav파일로 저장하여 그래프로 변환 (2개의 audio 동시 입력)~~
  - [ ] normalized data를 wav파일로 저장하여 그래프로 변환 (2개의 audio 동시 입력)
  - [x] ~~실시간으로 input data의 rms를 그래프로 출력 (2개의 audio 동시 입력)~~

실시간 파형 출력

- **input data를 wav파일로 저장하여 그래프로 변환**

```
# 필요한 라이브러리 임포트
import sounddevice as sd
import soundfile as sf
import threading
import time
import numpy as np

# 마이크 및 스피커 장치 설정
mic1_device = 9  # 첫 번째 마이크 (예시)
```

```
mic2_device = 9  # 두 번째 마이크 (예시)

# 각 마이크의 채널 수
mic_channels = 1  # 모노 입력

# 스피커 장치 설정
speaker1_device = 8  # 첫 번째 스피커 (스테레오)
speaker2_device = 8  # 두 번째 스피커 (스테레오)

# 스피커의 채널 수
speaker_channels = 2  # 스테레오 출력

# RMS를 저장하기 위한 클래스
class RMSRecorder:
    def __init__(self):
        self.rms_values = []

    def add_rms(self, rms):
        self.rms_values.append(rms)

    def average_rms(self):
        return np.mean(self.rms_values) if self.rms_values

    def all_rms(self):
        return self.rms_values


import matplotlib.pyplot as plt
import soundfile as sf

def save_waveform(file_path, image_path):
    # 오디오 파일 읽기
    data, samplerate = sf.read(file_path)

    # 파형을 플로팅
    plt.figure(figsize=(10, 4))
    plt.plot(data)
    plt.title('Audio Waveform')
```

```
    plt.xlabel('Sample Index')
    plt.ylabel('Amplitude')
    plt.grid(True)

    # 이미지로 저장
    plt.savefig(image_path)
    plt.close()



# 첫 번째 스트리밍 및 녹음 함수
def stream_and_record_initial(input_device, output_device,
    with sf.SoundFile(output_file, mode='w', samplerate=44
        start_time = time.time()

        def callback(indata, outdata, frames, time, status
            if status:
                print(f"Status: {status}")

            # 입력 데이터를 WAV 파일로 녹음
            file.write(indata)

            # 입력 데이터를 그대로 출력으로 복사
            outdata[:] = indata

            # RMS 계산 및 저장
            rms = np.sqrt(np.mean(indata**2))
            rms_recorder.add_rms(rms)

        # 스트리밍 지속
        with sd.Stream(device=(input_device, output_device
                               channels=(input_channels, output_ch
                               callback=callback):
            while time.time() - start_time < duration:
                time.sleep(0.1)

# 두 번째 스트리밍 및 녹음 함수
def stream_and_record_with_adjustment(input_device, output_
```

```
    with sf.SoundFile(output_file, mode='w', samplerate=44
        def callback(indata, outdata, frames, time, status
            if status:
                print(f"Status: {status}")

            file.write(indata)

            current_rms = np.sqrt(np.mean(indata**2))

            if current_rms > 0:
                scaling_factor = overall_average_rms / cur
                adjusted_data = indata * scaling_factor
            else:
                adjusted_data = indata

            outdata[:] = adjusted_data

        with sd.Stream(device=(input_device, output_device
                       channels=(input_channels, output_ch
                       callback=callback):
            while not stop_event.is_set():
                time.sleep(0.1)

# RMSRecorder 객체 생성
rms_recorder1 = RMSRecorder()
rms_recorder2 = RMSRecorder()

# 스트리밍 종료를 위한 이벤트 생성
stop_event = threading.Event()

# 첫 번째 스트리밍을 통해 RMS 기록
thread1 = threading.Thread(target=stream_and_record_initia
                           args=(mic1_device, speaker1_dev
thread2 = threading.Thread(target=stream_and_record_initia
                           args=(mic2_device, speaker2_dev

# 쓰레드 시작
thread1.start()
```

```
thread2.start()

# 첫 번째 스트리밍이 종료될 때까지 대기
thread1.join()
thread2.join()

# 각 마이크의 평균 RMS 값 계산 및 출력
average_rms1 = rms_recorder1.average_rms()
average_rms2 = rms_recorder2.average_rms()
combined_rms = rms_recorder1.all_rms() + rms_recorder2.all
overall_average_rms = np.mean(combined_rms)

print(f"Average RMS for mic1: {average_rms1:.5f}")
print(f"Average RMS for mic2: {average_rms2:.5f}")
print(f"Overall Average RMS: {overall_average_rms:.5f}")

# 파형 이미지 저장
save_waveform("mic1_recording.wav", "mic1_waveform.png")
save_waveform("mic2_recording.wav", "mic2_waveform.png")

# 두 번째 스트리밍을 통해 평균 RMS에 맞게 조정
# 스트리밍 및 녹음 함수 호출
thread1 = threading.Thread(target=stream_and_record_with_a
                           args=(mic1_device, speaker1_dev
thread2 = threading.Thread(target=stream_and_record_with_a
                           args=(mic2_device, speaker2_dev



# 두 번째 스트리밍 시작
thread1.start()
thread2.start()

# 사용자 입력을 기다림
input("Press 'Enter' to stop streaming and save waveform i

# 스트리밍 종료
stop_event.set()
```

```
# 스트리밍 종료 대기
thread1.join()
thread2.join()

# 조정된 녹음 파일들의 파형 이미지 저장
save_waveform("mic1_recording_adjusted.wav", "mic1_wavefor
save_waveform("mic2_recording_adjusted.wav", "mic2_wavefor

print("모든 작업이 완료되었습니다.")
```

- 조정 전후 차이를 확인하기 위해 wav파일로 저장하여 파형을 출력하도록 함
- 두 마이크의 조정 전 소리는 잘 저장되고 출력되나, 조정된 녹음 파일이 저장이 안 되는 문제 발생

  ⇒ ctl+c 를 사용하여 종료해서 코드 진행 시간을 주지 않은 것이 문제.

  ⇒ 종료 버튼을 생성하여 문제 해결 시도.

- **실시간 RMS 오디오 파형 출력**

```
import sounddevice as sd  # 오디오 입출력 관련 라이브러리
import numpy as np  # 배열 및 수학 함수 관련 라이브러리
import threading  # 스레드 관련 라이브러리
import matplotlib.pyplot as plt  # 데이터 시각화 관련 라이브러리
from matplotlib.animation import FuncAnimation  # 애니메이션 생성

# RMS를 저장하고 관리하기 위한 클래스
class RMSRecorder:
    def __init__(self, max_length=200):
        """
        RMSRecorder 클래스 초기화 함수

        Parameters:
            max_length (int): RMS 값을 저장할 최대 길이
        """
        self.rms_values = []  # RMS 값을 저장할 리스트
        self.max_length = max_length  # 최대 저장 길이
```

```
    def add_rms(self, rms):
        """
        RMS 값을 리스트에 추가하는 함수

        Parameters:
            rms (float): 추가할 RMS 값
        """
        if len(self.rms_values) >= self.max_length:
            self.rms_values.pop(0)  # 최대 길이를 초과하면 가장 오
        self.rms_values.append(rms)

# 오디오 스트림의 콜백 함수
def audio_callback(indata, outdata, frames, time, status, rms
    """
    오디오 스트림의 콜백 함수
    입력 오디오를 그대로 출력하고 RMS 값을 계산하여 저장함

    Parameters:
        indata (ndarray): 입력 오디오 데이터
        outdata (ndarray): 출력 오디오 데이터
        frames (int): 프레임 수
        time (CData): 현재 시간
        status (str): 상태 정보
        rms_recorder (RMSRecorder): RMS 값을 저장하기 위한 RMSRec
    """
    if status:
        print(f"Status: {status}")
    outdata[:] = indata  # 입력 오디오를 그대로 출력
    rms = np.sqrt(np.mean(indata**2))  # RMS 계산
    rms_recorder.add_rms(rms)  # RMS 값 저장

# 그래프를 업데이트하는 함수
def update_line(i, line, rms_recorder):
    """
    그래프를 업데이트하는 함수
    RMSRecorder 객체에서 가져온 RMS 값을 이용하여 그래프를 업데이트함

    Parameters:
```

```
        i (int): 애니메이션 프레임 인덱스
        line (Line2D): 업데이트할 선 객체
        rms_recorder (RMSRecorder): RMS 값을 저장하기 위한 RMSRec

    Returns:
        tuple: 업데이트된 선 객체
    """
    line.set_ydata(rms_recorder.rms_values)  # 그래프 데이터 업데
    line.set_xdata(range(len(rms_recorder.rms_values)))  # X축
    return line,

# 오디오 스트리밍 실행 함수
def run_stream(input_device, output_device, rms_recorder):
    """
    오디오 스트리밍 실행 함수
    지정된 입력 및 출력 장치에서 오디오를 스트리밍하여 RMS 값을 계산하고 

    Parameters:
        input_device (int): 입력 장치 ID
        output_device (int): 출력 장치 ID
        rms_recorder (RMSRecorder): RMS 값을 저장하기 위한 RMSRec
    """
    with sd.Stream(device=(input_device, output_device),
                   samplerate=44100, channels=1,
                   callback=lambda indata, outdata, frames, t
        input("Press 'Enter' to stop recording...\n")  # 사용자

# 메인 함수 설정
def main():
    input_device = 9  # 마이크 입력 장치 ID
    output_device = 8  # 스피커 출력 장치 ID
    rms_recorder = RMSRecorder(max_length=200)  # RMSRecorder

    # 그래프 초기화 및 설정
    fig, ax = plt.subplots()
    line, = ax.plot([], [], lw=2)  # 빈 그래프 선 객체 생성
    ax.set_xlim(0, 200)  # X축 범위 설정
    ax.set_ylim(0, 0.03)  # Y축 범위 설정
```

```
    ax.set_xlabel('Time')  # X축 레이블 설정
    ax.set_ylabel('RMS Value')  # Y축 레이블 설정
    ax.set_title('Real-Time RMS Plot')  # 그래프 제목 설정

    # FuncAnimation을 사용하여 실시간 그래프 업데이트
    ani = FuncAnimation(fig, update_line, fargs=(line, rms_re

    # 오디오 스트리밍 쓰레드 시작
    thread = threading.Thread(target=run_stream, args=(input_

    thread.start()

    plt.show()  # 그래프 표시

    thread.join()  # 오디오 스트리밍 쓰레드 종료 대기

if __name__ == "__main__":
    main()
```

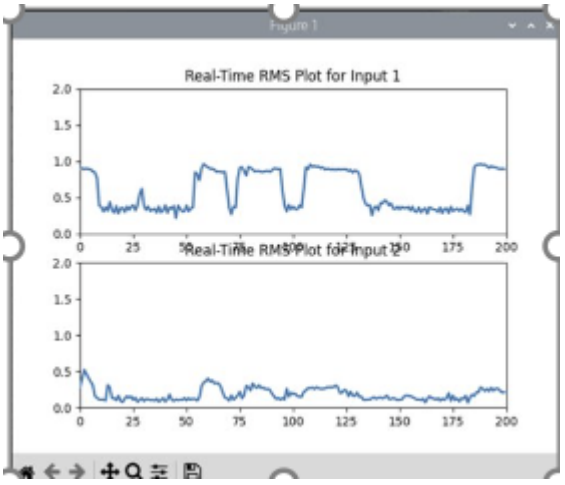
- **다중 input을 받아 각 마이크의 파형을 출력하도록 구현**

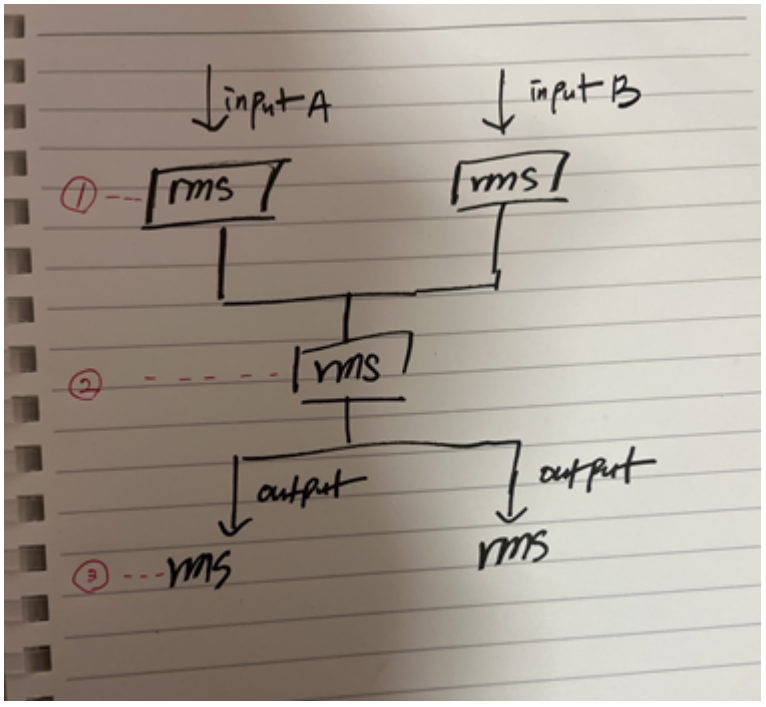
```
import sounddevice as sd
import numpy as np
import threading
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# RMS values management class
class RMSRecorder:
    def __init__(self, max_length=200):
        self.rms_values = []
        self.max_length = max_length

    def add_rms(self, rms):
        if len(self.rms_values) >= self.max_length:
            self.rms_values.pop(0)
        self.rms_values.append(rms)
```

```
# Audio stream callback function
def audio_callback(indata, outdata, frames, time, status, rms
    if status:
        print(f"Status: {status}")
    outdata[:] = indata  # Echo input to output
    rms = np.sqrt(np.mean(indata**2))
    rms_recorder.add_rms(rms)

# Update graph for real-time plotting
def update_line(i, line, rms_recorder):
    line.set_ydata(rms_recorder.rms_values)
    line.set_xdata(range(len(rms_recorder.rms_values)))
    return line,

# Function to handle audio streaming
def run_stream(input_device, output_device, rms_recorder):
    with sd.Stream(device=(input_device, output_device),
                   samplerate=44100, channels=1,
                   callback=lambda indata, outdata, frames, t
        input("Press 'Enter' to stop recording...\n")

# Main function setup
def main():
    # Device IDs for inputs and outputs
    input_device1, output_device1 = 3, 3
    input_device2, output_device2 = 5, 5

    rms_recorder1 = RMSRecorder(max_length=200)
    rms_recorder2 = RMSRecorder(max_length=200)

    # Initialize graphs
    fig, (ax1, ax2) = plt.subplots(2, 1)
    line1, = ax1.plot([], [], lw=2)
    line2, = ax2.plot([], [], lw=2)

    ax1.set_xlim(0, 200)
    ax1.set_ylim(0, 2)
    ax1.set_title('Real-Time RMS Plot for Input 1')
```

```
    ax2.set_xlim(0, 200)
    ax2.set_ylim(0, 2)
    ax2.set_title('Real-Time RMS Plot for Input 2')

    # Use FuncAnimation to update real-time graphs
    ani1 = FuncAnimation(fig, update_line, fargs=(line1, rms_
    ani2 = FuncAnimation(fig, update_line, fargs=(line2, rms_

    # Start audio streaming threads
    thread1 = threading.Thread(target=run_stream, args=(input
    thread2 = threading.Thread(target=run_stream, args=(input
    thread1.start()
    thread2.start()

    plt.show()  # Display graphs

    thread1.join()
    thread2.join()

if __name__ == "__main__":
    main()
```

실행결과 아래와 같은 결과를 출력 (실시간 파형 캡쳐 이미지)




input A
input B
①
rms
rms
②
rms
output
output
③
rms
rms

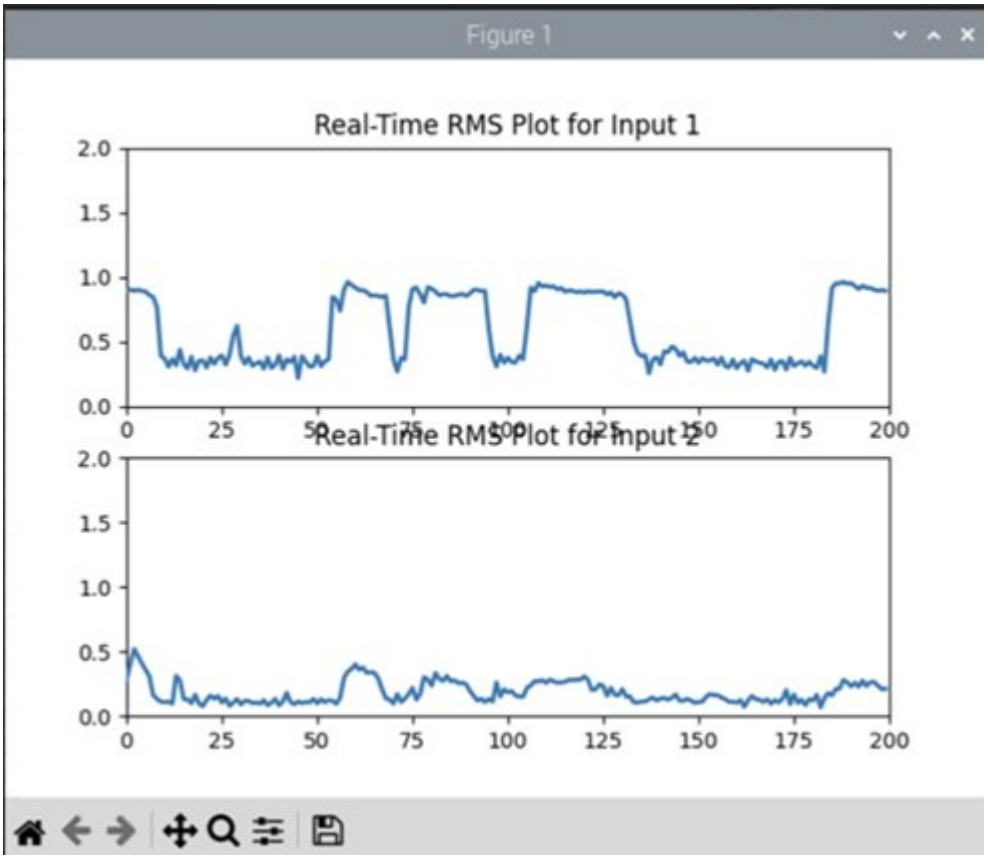


Figure 1
Real-Time RMS Plot for Input 1
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200
Real-Time RMS Plot for Input 2
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200

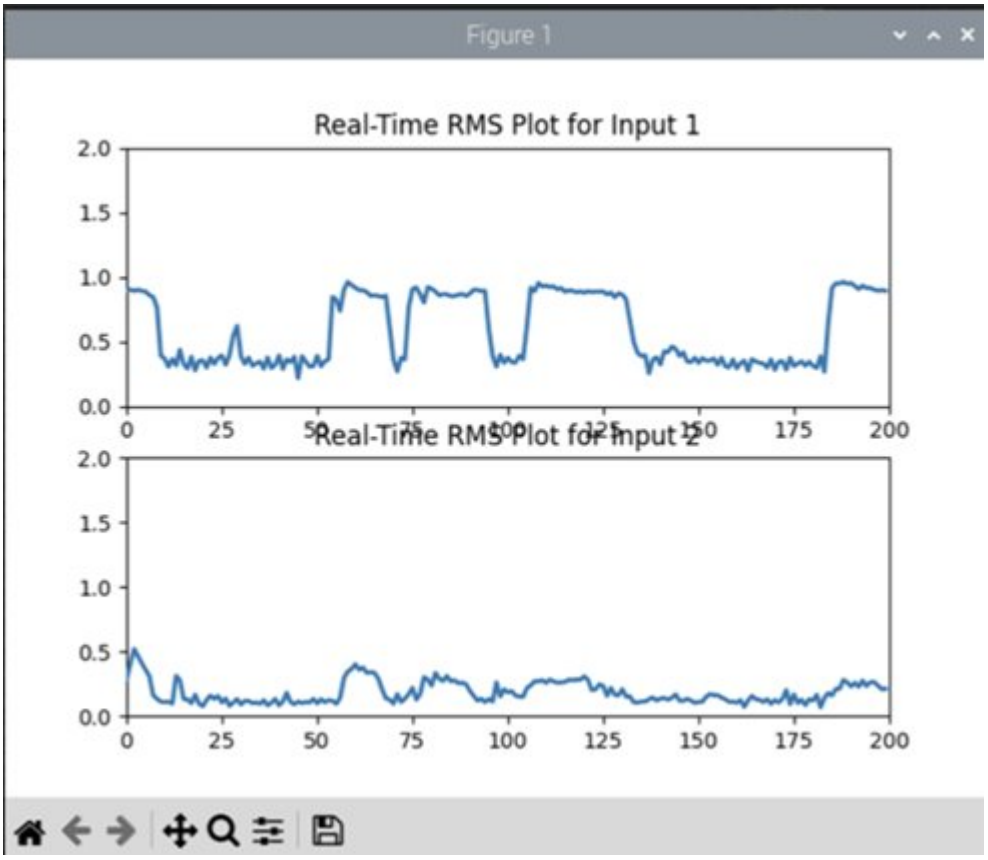


Figure 1
Real-Time RMS Plot for Input 1
Real-Time RMS Plot for Input 2
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200

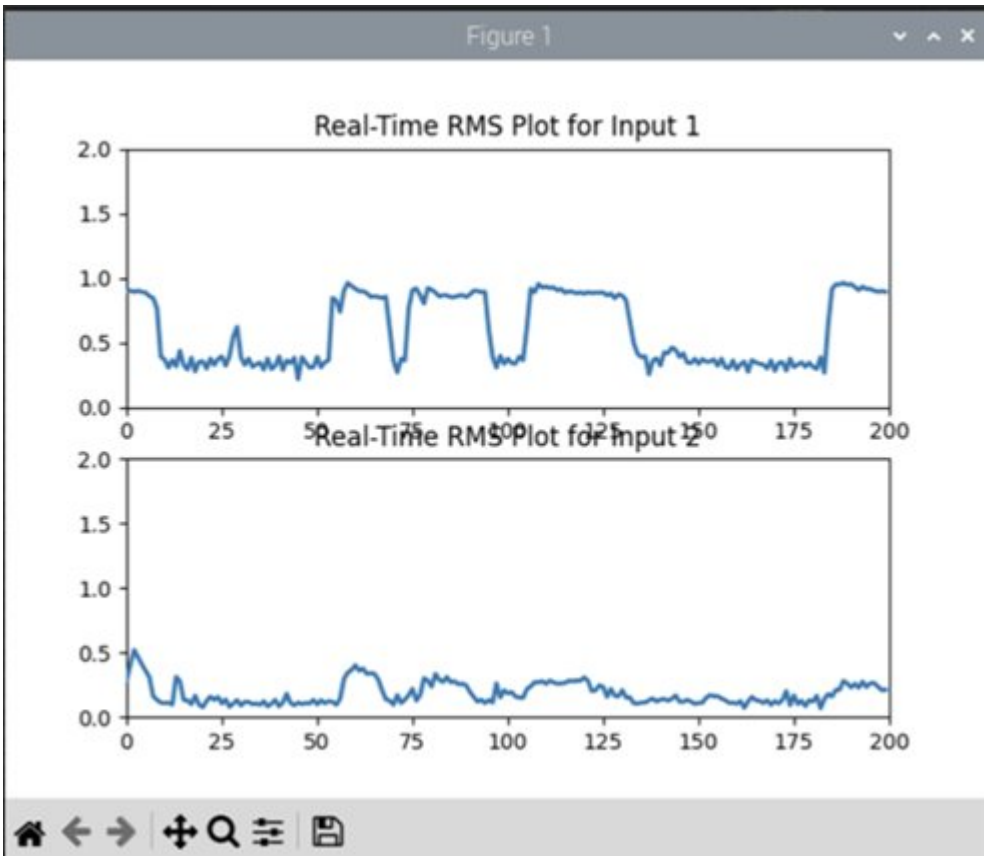


Figure 1
Real-Time RMS Plot for Input 1
Real-Time RMS Plot for Input 2
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200

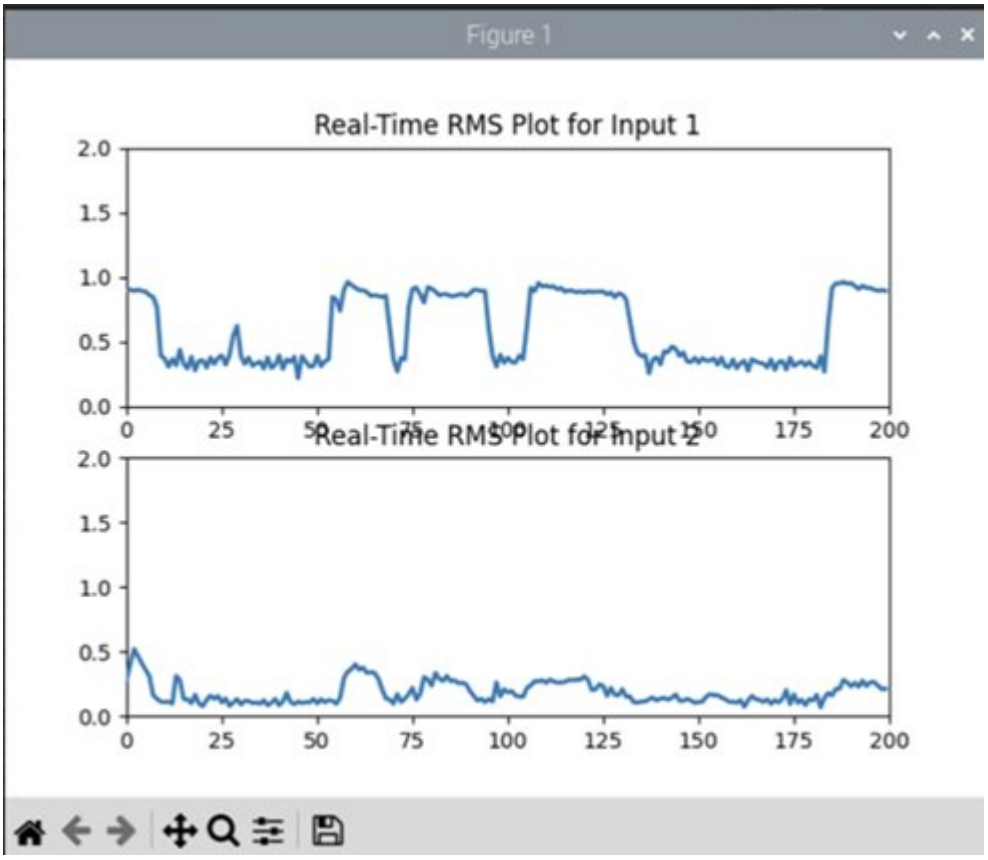


Figure 1
Real-Time RMS Plot for Input 1
Real-Time RMS Plot for Input 2
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200

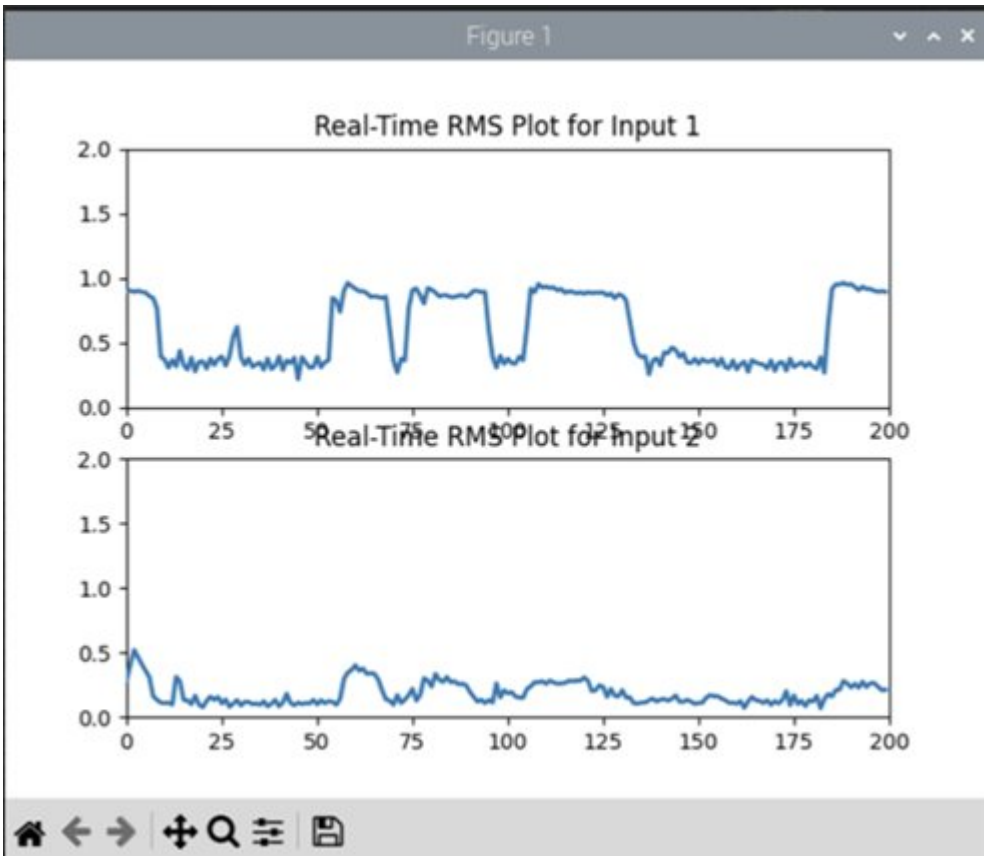


Figure 1
Real-Time RMS Plot for Input 1
Real-Time RMS Plot for Input 2
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
200

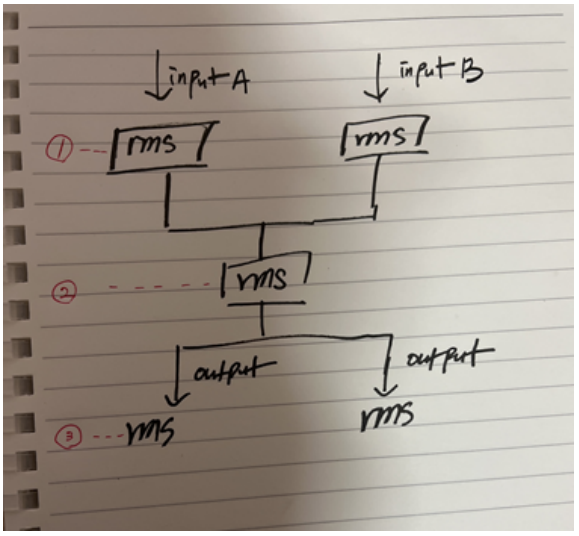
💡 **조교님 피드백**

RMS로 평균적인 세기를 구한 정확한 이유가 있을지, 신호의 Hz의 피크가 어느 정도일지는 알 지 못하나 실험을 통해 실험구성원분들은 알 것입니다.
따라서 예를들어 피크가 많은 크기차이가 없다면 RMS(E)가 아닌 MS(E)가 더 맞지않을까 싶습니다 (더 정밀하게 조정가능)

TODO

💡 평균 rms기반으로 조정된 값이 유의미한지 검증이 필요함

step. 3를 구현하고 파형을 출력할 수 있도록 개발



- **피드백 검토**
  - ☐ RMS(E), MS(E) 공부
  - ☐ RMS와 MS 중 적합한 데이터로 코드변경
- **데이터 검증**
  - ☐ wav파일의 rms를 구하는 코드를 만들어 normalized 전후 데이터 돌려보가
- **step. 3 개발**
  - ☐ t세팅타임을 주어 평균 rms 계산
  - ☐ 계산한 평균 rms를 이용하여 input data scaling
  - ☐ 평균 rms를 이용항 output data scaling