Communications Project: Frequency Division Multiplexing (FDM) using SSB Modulation

Fall 2024

Presented to:
Michael Melek

Presented by:

Name	ID
Moamen Moahmmed	9220886
Mina Hany	9220895

Contents

1 Introduction

This project explores the modulation of three speech signals using single sideband (SSB) modulation within a frequency-division multiplexing (FDM) system. The goal is to implement signal processing techniques such as low-pass filtering, modulation, and demodulation, ensuring high-quality transmission and recovery of signals.

1.1 Objectives

- Record and process three voice signals.
- Filter the signals using a low-pass filter.
- Modulate the filtered signals using SSB modulation.
- Multiplex the modulated signals using FDM.
- Demodulate the multiplexed signal to recover the original signals.

2 Voice Signal Recording and Preprocessing

2.1 Recording Voice Signals

The chosen sampling frequency is `44100 Hz`. This complies with the Nyquist theorem, which states that the sampling frequency must be at least twice the highest frequency component in the signal to avoid aliasing. Human speech primarily occupies the frequency range of 300 Hz to 3.4 kHz. By selecting a sampling rate of 44.1 kHz, the signal is accurately captured with no significant loss of high-frequency details. Furthermore, this sampling frequency ensures sufficient bandwidth to prevent distortion and aliasing during the modulation process, particularly as the signal's frequency content is shifted to higher frequencies during single-sideband (SSB) modulation. This choice supports the accurate transmission and recovery of the signals in the Frequency Division Multiplexing (FDM) system.

2.2 Saving Recorded Signals

The signals were saved as `input1.wav`, `input2.wav`, and `input3.wav`. Below is the code used for recording:

```
def record_audio(filename: str) -> None:
    """
    Record audio from the microphone and save it to a file.

    Args:
        filename (str): The path to save the audio file to.
    """
    input("Press Enter to start recording...")
    print(f"Recording audio to {filename} for {DURATION}
       seconds")
    audio = sd.rec(int(DURATION * SAMPLE_RATE),
                   samplerate=SAMPLE_RATE, channels=2, dtype=
                      'int16')
    sd.wait()
    write(filename, SAMPLE_RATE, audio)
    print(f"Audio saved to {filename}")
    print()
```

Listing 1: Voice Recording Code

3 Signal Filtering

3.1 Designing the Low-Pass Filter (LPF)

The low-pass filter (LPF) was designed to limit the maximum frequency of the signals to 4000 Hz. A Butterworth filter was chosen for its smooth frequency response in the passband.

The cutoff frequency ensures that the desired signal components are retained while unwanted high-frequency noise is attenuated. Below is the code used to implement the filter:

```
# Design the low-pass filter
nyquist_rate = sample_rate / 2.0  # Nyquist frequency
normal_cutoff = cutoff_frequency / nyquist_rate  # Normalize
    cutoff frequency

# Validate cutoff frequency
if not (0 < normal_cutoff < 1):
    raise ValueError("Cutoff frequency must be between 0 and
        Nyquist frequency.")

# Design Butterworth filter
b, a = butter(order, normal_cutoff, btype='low', analog=False
    )

# Apply the filter to the signal
filtered_left = filtfilt(b, a, signal_left)
filtered_right = None
if signal_right is not None:
    filtered_right = filtfilt(b, a, signal_right)

# Combine channels back (stereo if both channels exist)
if filtered_right is not None:
    filtered_signal = np.vstack((filtered_left,
        filtered_right)).T
else:
    filtered_signal = filtered_left
```

Listing 2: Butterworth LPF Code

3.2 Filtered Signals

The magnitude spectrum of the signals before and after filtering is shown below:

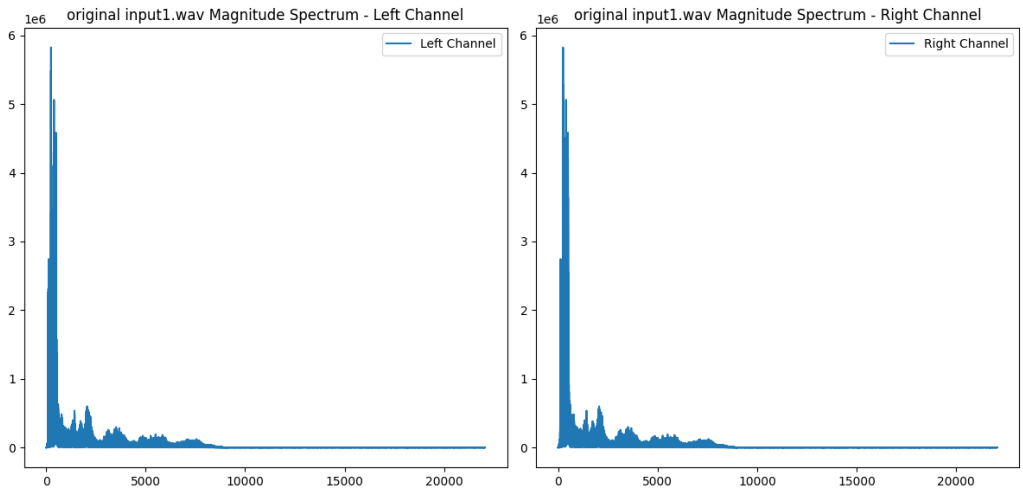

Figure 1: Magnitude Spectrum Before Filtering

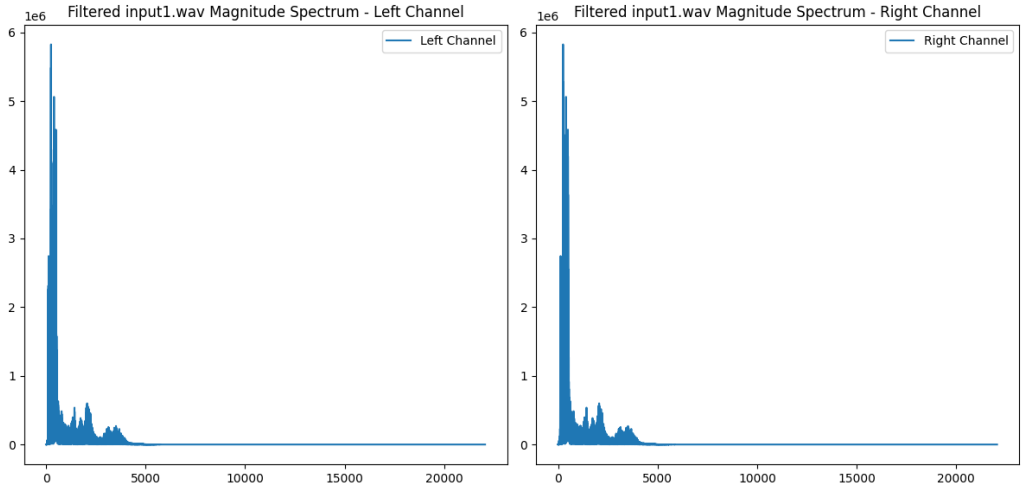

Figure 2: Magnitude Spectrum After Filtering

4 SSB Modulation

4.1 Carrier Frequency Selection

Carrier frequencies were chosen as `6K`, `11K`, and `16K` Hz. These frequencies ensure that the modulated signals do not overlap in the frequency domain, which is critical for avoiding interference during demodulation. Due to not using an ideal filter, a gap of 5 kHz between the carrier frequencies, which exceeds the filter's cutoff frequency of 4 kHz, provides sufficient separation between the signals.

4.2 Implementation of SSB Modulation

In this section, we outline the implementation of Single Sideband (SSB) Modulation. The process involves two key steps: Double Sideband (DSB) Modulation and filtering to isolate the desired sideband. Below, we detail each step with placeholders for code snippets and graphical representations of the results.

4.2.1 Step 1: Double Sideband (DSB) Modulation

To perform DSB Modulation, the input signal is multiplied with a carrier signal. Different carrier frequencies are chosen for each signal to ensure proper modulation. This can be expressed mathematically as:

$$s_{DSB}(t) = m(t) \cdot \cos(2\pi f_c t),$$

where $m(t)$ is the input signal and f_c is the carrier frequency.

Below is the code used for DSB Modulation:

```
# Apply DSB modulation to the left channel of the signal
sample_rate, signal = read(os.path.join(filtered_dir, file))
left_channel = signal[:, 0]
t = np.linspace(0, DURATION, sample_rate * DURATION, endpoint
    =False)
carrier = np.cos(2 * np.pi * carrier_frequencies[i] * t)
modulated_left = left_channel * carrier
```

Listing 3: DSB Modulation Code

Below is the graphical representation of the DSB-modulated signals for each input signal:

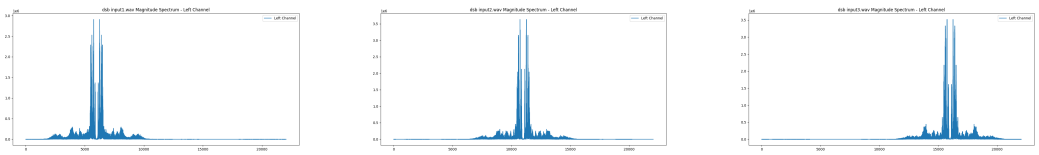

Figure 3: Magnitude Spectrum of DSB-Modulated Signals

4.2.2 Step 2: High-Pass Filtering to Obtain the Upper Sideband

After DSB Modulation, a high-pass filter is applied to extract the upper sideband (USB) of the signal. The filtering operation removes the lower sideband (LSB), leaving only the USB component. The high-pass filter is designed with a cutoff frequency just above the carrier frequency.

Below is the code used for high-pass filtering to isolate the upper sideband:

```
# Normalize the carrier frequency to the Nyquist frequency (
    half the sample rate)
nyquist_freq = sample_rate / 2.0
normalized_cutoff = carrier_freq / nyquist_freq

# Design a Butterworth high-pass filter
sos = butter(order, normalized_cutoff, btype='high', analog=
    False, output='sos')

# Apply the filter to the signal
filtered_signal = sosfilt(sos, signal)
```

Listing 4: High-Pass Filtering Code

The magnitude spectrum of the SSB-modulated signals is shown below:

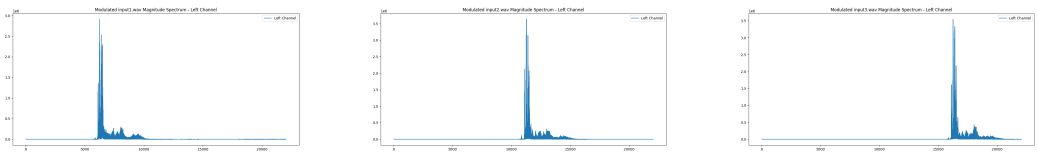

Figure 4: Magnitude Spectrum of SSB-Modulated Signals

5 Frequency Division Multiplexing (FDM)

5.1 Multiplexing Process

The multiplexing process involves combining the SSB-modulated signals into a single composite signal.

Below is the code used for FDM Multiplexing:

```
# Combine the modulated signals
modulated_signals = [modulated_input1, modulated_input2,
    modulated_input3]
fdm_signal = np.sum(modulated_signals, axis=0)
```

Listing 5: FDM Multiplexing Code

The magnitude spectrum of the multiplexed signal is shown below:

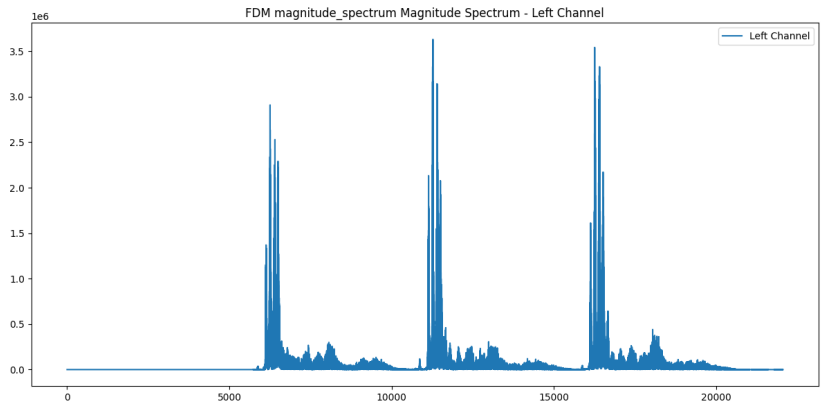

Figure 5: Magnitude Spectrum of Multiplexed Signals

6 SSB Demodulation and Signal Recovery

In this section, we describe the implementation of Single Sideband (SSB) demodulation and signal recovery. The process involves applying a bandpass filter to isolate each signal, followed by demodulation using multiplication with a cosine wave. Finally, the signal is scaled to its original amplitude.

6.0.1 Step 1: Bandpass Filtering to Isolate Each Signal

To recover individual signals, a bandpass filter is applied to extract each frequency band corresponding to the desired signal. The bandpass filter is designed with a passband centered around the carrier frequency for each signal. This ensures that only the desired signal is retained for further processing. Mathematically, this step isolates:

$$s_{BPF}(t) = H_{BP}(f) \cdot s_{SSB}(t),$$

where $H_{BP}(f)$ is the transfer function of the bandpass filter.

Below is the code used for bandpass filtering:

```
# Apply bandpass filtering to isolate each signal
nyquist_freq = sample_rate / 2.0
low = low_cutoff / nyquist_freq
high = high_cutoff / nyquist_freq
sos = butter(order, [low, high], btype='band', analog=False,
    output='sos')
filtered_signal = sosfilt(sos, signal)
```

Listing 6: Bandpass Filtering Code

The magnitude spectrum of the bandpass-filtered signals is shown below:

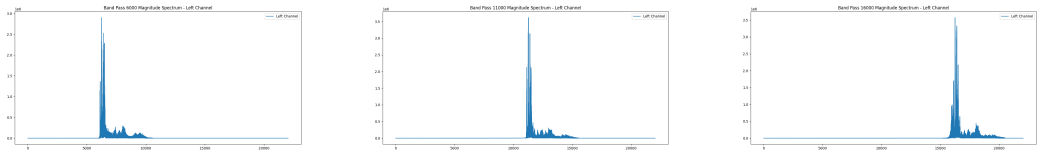

Figure 6: Magnitude Spectrum of Bandpass Filtered Signals

6.0.2 Step 2: Demodulation to Recover the Original Signal

After isolating the desired signal, it is multiplied with a cosine wave of the corresponding carrier frequency to demodulate it back to its baseband form. This operation effectively reverses the modulation process:

$$s_{demod}(t) = s_{BPF}(t) \cdot \cos(2\pi f_c t).$$

To restore the signal's amplitude, we multiply it by a factor of 4, as both modulation and demodulation reduce the amplitude by half each.

Below is the code used for demodulation:

```
# Demodulate the signal using a cosine wave
sample_rate, signal = read(os.path.join(separated_audios_dir,
    file))
t = np.linspace(0, DURATION, sample_rate * DURATION, endpoint
   =False)  # Time vector
carrier = np.cos(2 * np.pi * carrier_frequencies[i] * t)
demodulated_signal = AMPLIFING_FACTOR * signal * carrier
demodulated_signal = Filterer.low_pass_filter(
   demodulated_signal, sample_rate, LIMIT_FREQUENCY,
   FilterType.LOW_PASS_BUTTERWORTH)
```

Listing 7: Demodulation Code

The demodulated signals for each input signal are shown below:

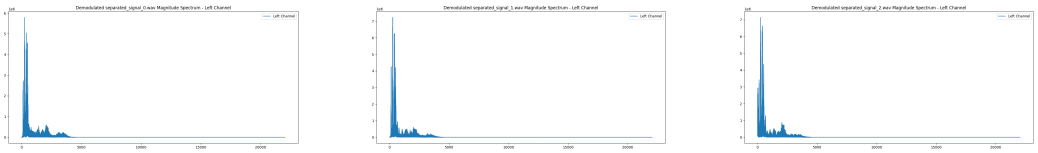

Figure 7: Magnitude Spectrum of Bandpass Filtered Signals

By following the above steps, we successfully recover the original signal from its SSB-modulated form, restoring it to its initial amplitude and frequency domain characteristics.

Summarize the outcomes of the project, key findings, and possible improvements for future work.

Appendix: Code

Below is the full Python code used in the project:

```
"""
This file contains the common includes for the project.
"""
from enum import Enum

DURATION = 10
SAMPLE_RATE = 44100
LIMIT_FREQUENCY = 4000
TOLERANCE_FREQUENCY = 500
AMPLIFING_FACTOR = 4

class FilterType(Enum):
    LOW_PASS_FOURIER = 1
    LOW_PASS_BUTTERWORTH = 2
    HIGH_PASS_IDEAL = 3
    HIGH_PASS_BUTTERWORTH = 4
    BAND_PASS_IDEAL = 5
    BAND_PASS_BUTTERWORTH = 6
```

Listing 8: Common Includes

```
"""
This module records audio from the microphone and saves it to
    a file.
"""
import sounddevice as sd
from scipy.io.wavfile import write
import sys
import os
from common_includes import *


def record_audio(filename: str) -> None:
    """
    Record audio from the microphone and save it to a file.

    Args:
        filename (str): The path to save the audio file to.
    """
    input("Press Enter to start recording...")
    print(f"Recording audio to {filename} for {DURATION}
       seconds")
    audio = sd.rec(int(DURATION * SAMPLE_RATE),
                   samplerate=SAMPLE_RATE, channels=2, dtype=
                      'int16')
```

```
    sd.wait()
    write(filename, SAMPLE_RATE, audio)
    print(f"Audio saved to {filename}")
    print()


if __name__ == "__main__":
    is_test = "--test" in sys.argv
    save_dir = os.path.join(os.path.dirname(
        os.path.abspath(__file__)), "..", "data", "input")
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    if is_test:
        from plot_signal import Plotter
        test_filename = os.path.join(save_dir, "test_input.
            wav")
        record_audio(test_filename)
        Plotter.plot_wavfile(
            test_filename, "Audio Waveform", "Time (Seconds)"
                , "Amplitude")
        Plotter.plot_wavfile_magnitude_spectrum(
            test_filename, "Audio Magnitude Spectrum", "
                Frequency (Hz)", "Magnitude")
    else:
        record_audio(os.path.join(save_dir, "input1.wav"))
        record_audio(os.path.join(save_dir, "input2.wav"))
        record_audio(os.path.join(save_dir, "input3.wav"))
```

Listing 9: Record Audio

```
import numpy as np
from scipy.signal import butter, filtfilt, sosfilt
from scipy.io.wavfile import write, read
import os
from common_includes import *
from plot_signal import Plotter


class Filterer:
    @classmethod
    def low_pass_filter(cls, signal: np.ndarray, sample_rate:
        int, cutoff_frequency: float, mode: FilterType=
       FilterType.LOW_PASS_BUTTERWORTH) -> np.ndarray:
        """
        Apply a low-pass filter to a signal.

        Args:
            signal (np.ndarray): The signal to filter.
            cutoff_frequency (float): The cutoff frequency of
```

```
                the filter.
            mode (FilterType): The type of filter to apply.

        Returns:
            np.ndarray: The filtered signal.
        """
        if mode == FilterType.LOW_PASS_FOURIER:
            return cls._low_pass_fourier(signal, sample_rate,
                cutoff_frequency)
        elif mode == FilterType.LOW_PASS_BUTTERWORTH:
            return cls._low_pass_butterworth(signal,
                sample_rate, cutoff_frequency)
        else:
            raise ValueError(f"Invalid filter mode: {mode}")

    @classmethod
    def suppress_lower_sideband(cls, signal: np.ndarray,
        sample_rate: int, carrier_freq: float, mode:
        FilterType=FilterType.HIGH_PASS_BUTTERWORTH) -> np.
        ndarray:
        """
        Suppress all frequencies below the carrier frequency
            using a high-pass filter.
        This converts the signal to Single Sideband (SSB) by
            removing the lower sideband.

        Args:
        - signal (np.ndarray): The signal to filter.
        - sample_rate (int): The sampling frequency of the
            signal.
        - carrier_freq (float): The carrier frequency around
            which modulation happens.
        - mode (FilterType): The type of high-pass filter to
            apply.

        Returns:
        - np.ndarray: The filtered signal with the lower
            sideband suppressed.
        """
        if mode == FilterType.HIGH_PASS_IDEAL:
            return cls._suppress_lower_sideband_ideal(signal,
                sample_rate, carrier_freq)
        elif mode == FilterType.HIGH_PASS_BUTTERWORTH:
            return cls._suppress_lower_sideband_butterworth(
                signal, sample_rate, carrier_freq)
        else:
            raise ValueError(f"Invalid filter mode: {mode}")

    @classmethod
```

```
    def band_pass_filter(cls, signal: np.ndarray, sample_rate: int, low_cutoff: float, high_cutoff: float, mode: FilterType=FilterType.BAND_PASS_BUTTERWORTH) -> np.ndarray:
        """
        Apply a band-pass filter to a signal.

        Args:
        - signal (np.ndarray): The signal to filter.
        - sample_rate (int): The sampling frequency of the signal.
        - low_cutoff (float): The lower bound of the passband.
        - high_cutoff (float): The upper bound of the passband.
        - mode (FilterType): The type of filter to apply.

        Returns:
        - np.ndarray: The filtered signal
        """
        if mode == FilterType.BAND_PASS_IDEAL:
            return cls._band_pass_filter_ideal(signal, sample_rate, low_cutoff, high_cutoff)
        elif mode == FilterType.BAND_PASS_BUTTERWORTH:
            return cls._band_pass_filter_butterworth(signal, sample_rate, low_cutoff, high_cutoff)
        else:
            raise ValueError(f"Invalid filter mode: {mode}")

    @classmethod
    def _suppress_lower_sideband_ideal(cls, signal: np.ndarray, sample_rate: int, carrier_freq: float) -> np.ndarray:
        """
        Suppress all frequencies below the carrier frequency using an ideal high-pass filter.
        This converts the signal to Single Sideband (SSB) by removing the lower sideband.

        Parameters:
        - signal: Input signal (1D numpy array).
        - sample_rate: Sampling frequency (Hz).
        - carrier_freq: Carrier frequency around which modulation happens (Hz).

        Returns:
        - Filtered signal with the lower sideband suppressed (SSB).
        """
```

```
88          # Perform Fourier Transform to convert signal to
                frequency domain
89          spectrum = np.fft.fft(signal)
90          freqs = np.fft.fftfreq(len(signal), d=1/sample_rate)
91
92          # Create an ideal high-pass filter (1 for frequencies
                above the carrier, 0 for below)
93          high_pass_filter = np.abs(freqs) >= carrier_freq
94
95          # Apply the high-pass filter (remove lower sideband)
96          spectrum[~high_pass_filter] = 0  # Set the lower
                sideband components to 0
97
98          # Perform Inverse Fourier Transform to get the
                filtered signal in time domain
99          filtered_signal = np.fft.ifft(spectrum).real
100
101         return filtered_signal
102
103     @staticmethod
104     def _suppress_lower_sideband_butterworth(signal: np.
            ndarray, sample_rate: int, carrier_freq: float, order:
             int = 150) -> np.ndarray:
105         """
106         Suppress all frequencies below the carrier frequency
                using a Butterworth high-pass filter.
107         This converts the signal to Single Sideband (SSB) by
                removing the lower sideband.
108
109         Parameters:
110         - signal: Input signal (1D numpy array).
111         - sample_rate: Sampling frequency (Hz).
112         - carrier_freq: Carrier frequency around which
                modulation happens (Hz).
113         - order: Order of the Butterworth filter (default: 5)
                .
114
115         Returns:
116         - Filtered signal with the lower sideband suppressed
                (SSB).
117         """
118         # Normalize the carrier frequency to the Nyquist
                frequency (half the sample rate)
119         nyquist_freq = sample_rate / 2.0
120         normalized_cutoff = carrier_freq / nyquist_freq
121
122         # Design a Butterworth high-pass filter
123         sos = butter(order, normalized_cutoff, btype='high',
                analog=False, output='sos')
```

```
124
125             # Apply the filter to the signal
126             filtered_signal = sosfilt(sos, signal)
127
128             return filtered_signal
129
130         @classmethod
131         def _band_pass_filter_ideal(cls, signal: np.ndarray,
                sample_rate: int, low_cutoff: float, high_cutoff:
                float) -> np.ndarray:
132             """
133             Apply an ideal band-pass filter to a signal, passing
                    only the frequencies between low_cutoff and
                    high_cutoff.
134
135             Parameters:
136             - signal: Input signal (1D numpy array).
137             - sample_rate: Sampling frequency (Hz).
138             - low_cutoff: Lower bound of the passband (Hz).
139             - high_cutoff: Upper bound of the passband (Hz).
140
141             Returns:
142             - Filtered signal with only frequencies between
                    low_cutoff and high_cutoff passed.
143             """
144             # Perform Fourier Transform (FFT) of the signal
145             spectrum = np.fft.fft(signal)
146             freqs = np.fft.fftfreq(len(signal), d=1/sample_rate)
147
148             # Create an ideal band-pass filter: 1 for frequencies
                    in range, 0 otherwise
149             band_pass_filter = (np.abs(freqs) >= low_cutoff) & (
                    np.abs(freqs) <= high_cutoff)
150
151             # Apply the band-pass filter by zeroing out
                    frequencies outside the passband
152             spectrum[~band_pass_filter] = 0
153
154             # Perform Inverse Fourier Transform (IFFT) to get the
                    filtered signal in the time domain
155             filtered_signal = np.fft.ifft(spectrum).real
156
157             return filtered_signal
158
159         @classmethod
160         def _band_pass_filter_butterworth(cls, signal: np.ndarray
                , sample_rate: int, low_cutoff: float, high_cutoff:
                float, order: int = 100) -> np.ndarray:
161             """
```

```
        Apply a Butterworth band-pass filter to a signal,
            passing only the frequencies between low_cutoff
            and high_cutoff.

        Parameters:
        - signal: Input signal (1D numpy array).
        - sample_rate: Sampling frequency (Hz).
        - low_cutoff: Lower bound of the passband (Hz).
        - high_cutoff: Upper bound of the passband (Hz).
        - order: Order of the Butterworth filter (default: 5)
            .

        Returns:
        - Filtered signal with only frequencies between
            low_cutoff and high_cutoff passed.
        """
        # Normalize the cutoff frequencies to the Nyquist
            frequency (half the sample rate)
        nyquist_freq = sample_rate / 2.0
        low = low_cutoff / nyquist_freq
        high = high_cutoff / nyquist_freq

        # Design a Butterworth band-pass filter as second-
            order sections (SOS)
        sos = butter(order, [low, high], btype='band', analog
            =False, output='sos')

        # Apply the filter to the signal
        filtered_signal = sosfilt(sos, signal)

        return filtered_signal

    @classmethod
    def _low_pass_fourier(cls, signal: np.ndarray,
        sample_rate: int, cutoff_frequency: float) -> np.
        ndarray:
        """
        Apply a low-pass Fourier filter to a signal.

        Args:
            signal (np.ndarray): The signal to filter.
            cutoff_frequency (float): The cutoff frequency of
                the filter.

        Returns:
            np.ndarray: The filtered signal.
        """
        print("Applying Fourier filter")
        nyquist = 0.5 * sample_rate
```

```
201         cutoff_idx = int(cutoff_frequency / nyquist * (len(
                signal) // 2))
202         signal_fft = np.abs(np.fft.rfft(signal, axis=0))
203         freqs = np.fft.rfftfreq(len(signal), d=1/sample_rate)
204         signal_fft[cutoff_idx:] = 0
205         filtered_signal = np.fft.irfft(signal_fft, n=len(
                signal), axis=0)
206         return filtered_signal
207 
208     @classmethod
209     def _low_pass_butterworth(cls, signal: np.ndarray,
            sample_rate: int, cutoff_frequency: float, order: int
            =8) -> np.ndarray:
210         """
211         Apply a low-pass Butterworth filter to a signal.
212 
213         Args:
214             signal (np.ndarray): The signal to filter.
215             cutoff_frequency (float): The cutoff frequency of
                    the filter.
216 
217         Returns:
218             np.ndarray: The filtered signal.
219         """
220         if len(signal.shape) > 1:
221             signal_left = signal[:, 0]
222             signal_right = signal[:, 1]
223         else:
224             signal_left = signal
225             signal_right = None
226 
227         # Design the low-pass filter
228         nyquist_rate = sample_rate / 2.0  # Nyquist frequency
229         normal_cutoff = cutoff_frequency / nyquist_rate  #
                Normalize cutoff frequency
230 
231         # Validate cutoff frequency
232         if not (0 < normal_cutoff < 1):
233             raise ValueError("Cutoff frequency must be
                    between 0 and Nyquist frequency.")
234 
235         # Design Butterworth filter
236         b, a = butter(order, normal_cutoff, btype='low',
                analog=False)
237 
238         # Apply the filter to the signal
239         filtered_left = filtfilt(b, a, signal_left)
240         filtered_right = None
241         if signal_right is not None:
```

```
242             filtered_right = filtfilt(b, a, signal_right)
243
244         # Combine channels back (stereo if both channels
                exist)
245         if filtered_right is not None:
246             filtered_signal = np.vstack((filtered_left,
                    filtered_right)).T
247         else:
248             filtered_signal = filtered_left
249
250         return filtered_signal
251
252
253 if __name__ == "__main__":
254     input_dir = os.path.join(os.path.dirname(os.path.abspath(
            __file__)), "..", "data", "input")
255     input_magnitude_spectrum_dir = os.path.join(input_dir, "
            magnitude_spectrum")
256     save_dir = os.path.join(os.path.dirname(os.path.abspath(
            __file__)), "..", "data", "filtered")
257     save_magnitude_spectrum_dir = os.path.join(save_dir, "
            magnitude_spectrum")
258
259     if not os.path.exists(save_dir):
260         os.makedirs(save_dir)
261
262     # loop through all the files in the data directory
263     for file in os.listdir(input_dir):
264         if file.endswith(".wav"):
265             # read the file
266             sample_rate, signal = read(os.path.join(input_dir
                    , file))
267             # filter the signal
268             filtered_signal = Filterer.low_pass_filter(signal
                    , sample_rate, LIMIT_FREQUENCY, FilterType.
                    LOW_PASS_BUTTERWORTH)
269             # save the filtered signal
270             write(os.path.join(save_dir, file), sample_rate,
                    filtered_signal.astype(np.int16))
271             # plot the magnitude spectrum of the original
                    signal
272             Plotter.plot_magnitude_spectrum(signal,
                    sample_rate, title=f"original {file} Magnitude
                     Spectrum", save_dir=
                    input_magnitude_spectrum_dir, file_name=f"
                    original_{file}_magnitude_spectrum.png")
273             # plot the magnitude spectrum of the filtered
                    signal
274             Plotter.plot_magnitude_spectrum(filtered_signal,
```

```
                sample_rate, title=f"Filtered {file} Magnitude
                 Spectrum", save_dir=
                save_magnitude_spectrum_dir, file_name=f"
                filtered_{file}_magnitude_spectrum.png")
```

Listing 10: Filterer

```
import numpy as np
from scipy.signal import butter, filtfilt
from scipy.io.wavfile import write, read
import os
from common_includes import *
from plot_signal import Plotter
from filter_signal import Filterer


carrier_frequencies = [6000, 11000, 16000]  # Carrier
    frequencies for FDM


if __name__ == "__main__":
    # path filtered signals
    filtered_dir = os.path.join(os.path.dirname(os.path.
        abspath(__file__)), "..", "data", "filtered")
    # path modulated signals
    modulated_dir = os.path.join(os.path.dirname(os.path.
        abspath(__file__)), "..", "data", "modulated")
    modulated_magnitude_spectrum_dir = os.path.join(
        modulated_dir, "magnitude_spectrum")
    fdm_magnitude_spectrum_dir = os.path.join(modulated_dir,
        "fdm_spectrum")
    dsb_magnitude_spectrum_dir = os.path.join(modulated_dir,
        "dsb_spectrum")


    if not os.path.exists(modulated_dir):
        os.makedirs(modulated_dir)
    if not os.path.exists(modulated_magnitude_spectrum_dir):
        os.makedirs(modulated_magnitude_spectrum_dir)
    if not os.path.exists(fdm_magnitude_spectrum_dir):
        os.makedirs(fdm_magnitude_spectrum_dir)

    # Modulation and FDM
    modulated_signals = []
    for i, file in enumerate(os.listdir(filtered_dir)):
        if file.endswith(".wav"):
            sample_rate, signal = read(os.path.join(
                filtered_dir, file))
            left_channel = signal[:, 0]
```

```
            #right_channel = signal[:, 1]
            # Generate carrier signal

            t = np.linspace(0, DURATION, sample_rate *
                DURATION, endpoint=False)  # Time vector
            carrier = np.cos(2 * np.pi * carrier_frequencies[
                i] * t)


            modulated_left = left_channel * carrier
            #modulated_right = right_channel * carrier


            # Combine the modulated channels back into a
                stereo signal
            #modulated_stereo_signal = np.column_stack((
                modulated_left, modulated_right))
            Plotter.plot_magnitude_spectrum(modulated_left,
                sample_rate, mono=True, title=f"dsb {file}
                Magnitude Spectrum", save_dir=
                dsb_magnitude_spectrum_dir, file_name=f"dsb_{
                file}_magnitude_spectrum.png")
            # Suppress lower sideband
            ssb_signal = Filterer.suppress_lower_sideband(
                modulated_left, sample_rate,
                carrier_frequencies[i])
            # plot the magnitude spectrum of the filtered
                signal
            Plotter.plot_magnitude_spectrum(ssb_signal,
                sample_rate, mono=True, title=f"Modulated {
                file} Magnitude Spectrum", save_dir=
                modulated_magnitude_spectrum_dir, file_name=f"
                modulated_{file}_magnitude_spectrum.png")
            modulated_signals.append(ssb_signal)

        # Sum all modulated signals to form the FDM signal
    fdm_signal = np.sum(modulated_signals, axis=0)

    Plotter.plot_magnitude_spectrum(fdm_signal, SAMPLE_RATE,
        mono=True, title=f"FDM {file} Magnitude Spectrum",
        save_dir=fdm_magnitude_spectrum_dir, file_name=f"FDM_{
        file}_magnitude_spectrum.png")

    # Save the FDM signal
    write(os.path.join(modulated_dir, "fdm_signal.wav"),
        sample_rate, fdm_signal.astype(np.int16))
```

Listing 11: Modulator

```python
import numpy as np
from scipy.signal import butter, filtfilt
from scipy.io.wavfile import write, read
import os
from common_includes import *
from plot_signal import Plotter
from filter_signal import Filterer


carrier_frequencies = [6000, 11000, 16000]  # Carrier frequencies for FDM


if __name__ == "__main__":
    # path modulated signals
    modulated_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data", "modulated")
    # path demodulated signals
    demodulated_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), "..", "data", "demodulated")
    separated_spectrums_dir = os.path.join(demodulated_dir, "separated_spectrums")
    separated_audios_dir = os.path.join(demodulated_dir, "separated_audios")
    signals_after_demodulation_spectrum_dir = os.path.join(demodulated_dir, "signals_after_demodulation_spectrum")
    signals_after_demodulation_audios_dir = os.path.join(demodulated_dir, "signals_after_demodulation_audios")

    if not os.path.exists(demodulated_dir):
        os.makedirs(demodulated_dir)

    if not os.path.exists(separated_spectrums_dir):
        os.makedirs(separated_spectrums_dir)

    if not os.path.exists(separated_audios_dir):
        os.makedirs(separated_audios_dir)

    if not os.path.exists(signals_after_demodulation_spectrum_dir):
        os.makedirs(signals_after_demodulation_spectrum_dir)

    if not os.path.exists(signals_after_demodulation_audios_dir):
        os.makedirs(signals_after_demodulation_audios_dir)


    for file in os.listdir(modulated_dir):
```

```
40          if file.endswith(".wav"):
41              sample_rate, signal = read(os.path.join(
                    modulated_dir, file))
42              # iterate through the carrier frequencies and do
                    band pass filter from filter_signal.py and add
                     them to signalsarray
43              for i , carrier_frequency in enumerate(
                    carrier_frequencies):
44                  band_pass_signal = Filterer.band_pass_filter(
                        signal, sample_rate, carrier_frequency,
                        carrier_frequency + LIMIT_FREQUENCY +
                        TOLERANCE_FREQUENCY)
45                  # plot the magnitude spectrum of the filtered
                         signal
46                  Plotter.plot_magnitude_spectrum(
                        band_pass_signal, sample_rate, mono=True,
                        title=f"Band Pass {carrier_frequency}
                        Magnitude Spectrum", save_dir=
                        separated_spectrums_dir, file_name=f"
                        band_pass_{carrier_frequency}
                        _magnitude_spectrum.png")
47                  # save the filtered signal
48                  write(os.path.join(separated_audios_dir, f"
                        separated_signal_{i}.wav"), sample_rate,
                        band_pass_signal.astype(np.int16))
49
50      for i, file in enumerate(os.listdir(separated_audios_dir)
            ):
51          if file.endswith(".wav"):
52              sample_rate, signal = read(os.path.join(
                    separated_audios_dir, file))
53
54              t = np.linspace(0, DURATION, sample_rate *
                    DURATION, endpoint=False)  # Time vector
55              carrier = np.cos(2 * np.pi * carrier_frequencies[
                    i] * t)
56
57
58              demodulated_signal = AMPLIFING_FACTOR * signal *
                    carrier
59
60              demodulated_signal = Filterer.low_pass_filter(
                    demodulated_signal, sample_rate,
                    LIMIT_FREQUENCY, FilterType.
                    LOW_PASS_BUTTERWORTH)
61              # plot the magnitude spectrum of the filtered
                    signal
62              Plotter.plot_magnitude_spectrum(
                    demodulated_signal, sample_rate, mono=True,
```

```
                title=f"Demodulated {file} Magnitude Spectrum"
                , save_dir=
                signals_after_demodulation_spectrum_dir,
                file_name=f"out{i}.png")
63
64          write(os.path.join(
                signals_after_demodulation_audios_dir, f"out{i
                +1}.wav"), sample_rate, demodulated_signal.
                astype(np.int16))
```

Listing 12: Demodulator

```
1  import numpy as np
2  import os
3  import matplotlib.pyplot as plt
4  from scipy.io import wavfile
5  from common_includes import *
6
7  class Plotter:
8      @classmethod
9      def plot(cls, time, signal, title=None, x_label=None,
           y_label=None):
10         plt.title(title)
11         plt.plot(time, signal)
12         plt.xlabel(x_label)
13         plt.ylabel(y_label)
14         plt.show()
15
16     @classmethod
17     def plot_wavfile(cls, filename, title=None, x_label=None,
           y_label=None):
18         sample_rate, signal = wavfile.read(filename)
19         if len(signal.shape) == 2:
20             signal = signal[:, 0]
21         time = np.arange(len(signal)) / sample_rate
22         cls.plot(time, signal, title, x_label, y_label)
23
24     @classmethod
25     def plot_wavfile_magnitude_spectrum(cls, filename, title=
           None, x_label=None, y_label=None):
26         sample_rate, signal = wavfile.read(filename)
27
28         if len(signal.shape) == 1:  # Mono
29             cls.plot_magnitude_spectrum(signal, sample_rate,
                   title, x_label, y_label, mono=True)
30         elif len(signal.shape) == 2 and signal.shape[1] == 2:
               # Stereo
31             cls.plot_magnitude_spectrum(signal, sample_rate,
                   title, x_label, y_label, mono=False)
```

```
32          else:
33              raise ValueError("Unsupported audio format")
34
35      @classmethod
36      def plot_magnitude_spectrum(cls, signal, sample_rate=
            SAMPLE_RATE, title=None, x_label=None, y_label=None,
            mono=False, save_dir=None, file_name="
            magnitude_spectrum.png"):
37          # If mono, work with the single channel; if stereo,
                split the channels
38          if mono:
39              data_left = signal
40              data_right = None
41          else:
42              data_left = signal[:, 0]
43              data_right = signal[:, 1]
44
45          # Calculate frequency bins using FFT
46          frequencies = np.fft.rfftfreq(len(signal), d=1 /
                sample_rate)
47
48          # Compute the magnitude spectrum for left and right
                channels
49          magnitude_left = np.abs(np.fft.rfft(data_left))
50          magnitude_right = np.abs(np.fft.rfft(data_right)) if
                data_right is not None else None
51
52          # Plotting the frequency domain
53          plt.figure(figsize=(12, 6))
54
55          # Plot for left channel
56          plt.subplot(1, 2 if not mono else 1, 1)
57          plt.plot(frequencies, magnitude_left, label='Left
                Channel')
58          plt.title(f'{title} - Left Channel')
59          plt.xlabel(x_label)
60          plt.ylabel(y_label)
61          plt.legend()
62
63          # Plot for right channel if stereo
64          if not mono and magnitude_right is not None:
65              plt.subplot(1, 2, 2)
66              plt.plot(frequencies, magnitude_right, label='
                    Right Channel')
67              plt.title(f'{title} - Right Channel')
68              plt.xlabel(x_label)
69              plt.ylabel(y_label)
70              plt.legend()
71
```

```
72          # Adjust the x-axis range to zoom in on the higher
                frequencies (e.g., 0-50 kHz)
73          #plt.xlim(0, 60000)  # Adjust this range as needed to
                include carrier frequencies
74
75          plt.tight_layout()
76
77           # Save the plot to the specified directory
78          if save_dir:
79              os.makedirs(save_dir, exist_ok=True)  # Create
                    the directory if it doesn't exist
80              save_path = os.path.join(save_dir, file_name)
81              plt.savefig(save_path)
82
83          #plt.show()
```

Listing 13: Plot Signal