

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	3
1.1	Objectives	3
2	Voice Signal Recording and Preprocessing	4
2.1	Recording Voice Signals	4
2.2	Saving Recorded Signals	4
3	Signal Filtering	5
3.1	Designing the Low-Pass Filter (LPF)	5
3.2	Filtered Signals	5
4	SSB Modulation	7
4.1	Carrier Frequency Selection	7
4.2	Implementation of SSB Modulation	7
4.2.1	Step 1: Double Sideband (DSB) Modulation	7
4.2.2	Step 2: High-Pass Filtering to Obtain the Upper Side- band	8
5	Frequency Division Multiplexing (FDM)	9
5.1	Multiplexing Process	9
6	SSB Demodulation and Signal Recovery	10
6.0.1	Step 1: Bandpass Filtering to Isolate Each Signal . . .	10
6.0.2	Step 2: Demodulation to Recover the Original Signal .	10

1 Introduction

This project explores the modulation of three speech signals using single side-band (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 selected sampling frequency is 44100 Hz. This choice adheres to the Nyquist theorem, which requires the sampling frequency to be at least twice the highest frequency present in the signal to avoid aliasing. Human speech generally ranges from 300 Hz to 3.4 kHz. By using a sampling frequency of 44.1 kHz, the signal is captured accurately, with no significant loss of high-frequency components.

2.2 Saving Recorded Signals

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

```
1 def record_audio(filename: str) -> None:
2     """
3     Record audio from the microphone and save it to a file.
4
5     Args:
6         filename (str): The path to save the audio file to.
7     """
8     input("Press Enter to start recording...")
9     print(f"Recording audio to {filename} for {DURATION}
10           seconds")
11     audio = sd.rec(int(DURATION * SAMPLE_RATE),
12                   samplerate=SAMPLE_RATE, channels=2, dtype=
13                     'int16')
14     sd.wait()
15     write(filename, SAMPLE_RATE, audio)
16     print(f"Audio saved to {filename}")
17     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:

```
1 # Design the low-pass filter
2 nyquist_rate = sample_rate / 2.0 # Nyquist frequency
3 normal_cutoff = cutoff_frequency / nyquist_rate # Normalize
   cutoff frequency
4
5 # Validate cutoff frequency
6 if not (0 < normal_cutoff < 1):
7     raise ValueError("Cutoff frequency must be between 0 and
   Nyquist frequency.")
8
9 # Design Butterworth filter
10 b, a = butter(order, normal_cutoff, btype='low', analog=False
   )
11
12 # Apply the filter to the signal
13 filtered_left = filtfilt(b, a, signal_left)
14 filtered_right = None
15 if signal_right is not None:
16     filtered_right = filtfilt(b, a, signal_right)
17
18 # Combine channels back (stereo if both channels exist)
19 if filtered_right is not None:
20     filtered_signal = np.vstack((filtered_left,
   filtered_right)).T
21 else:
22     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:

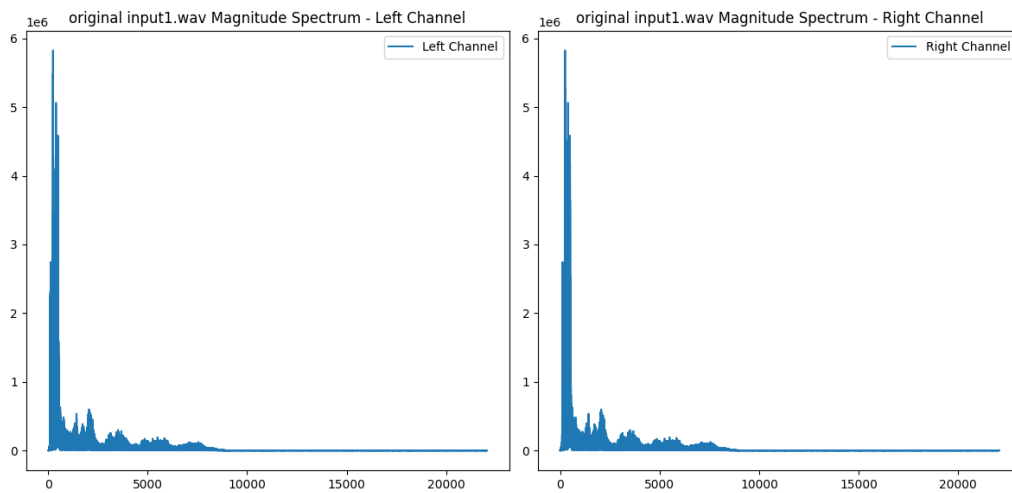


Figure 1: Magnitude Spectrum Before Filtering

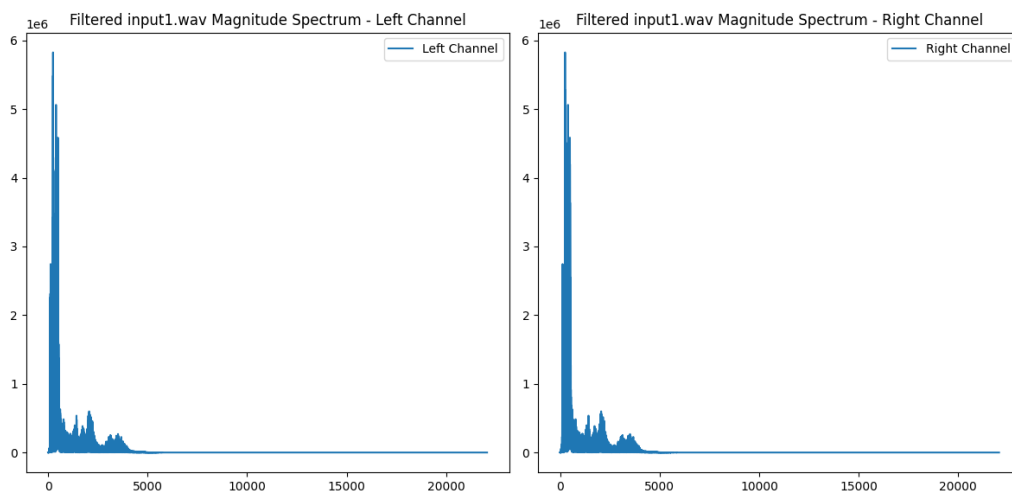


Figure 2: Magnitude Spectrum After Filtering

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:

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

Listing 3: DSB Modulation Code

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

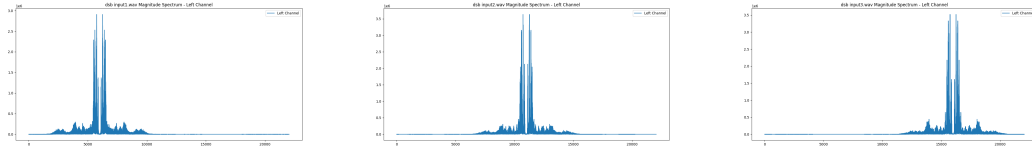


Figure 3: Magnitude Spectrum of DSB-Modulated Signals

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:

```

1 # Normalize the carrier frequency to the Nyquist frequency (
  half the sample rate)
2 nyquist_freq = sample_rate / 2.0
3 normalized_cutoff = carrier_freq / nyquist_freq
4
5 # Design a Butterworth high-pass filter
6 sos = butter(order, normalized_cutoff, btype='high', analog=
  False, output='sos')
7
8 # Apply the filter to the signal
9 filtered_signal = sosfilt(sos, signal)

```

Listing 4: High-Pass Filtering Code

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

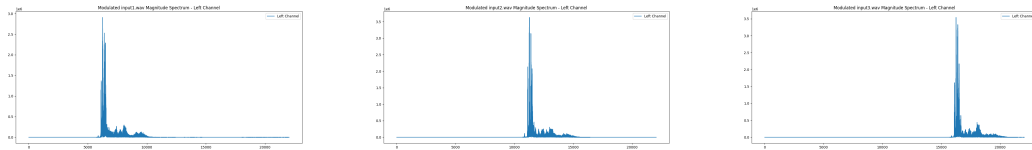


Figure 4: Magnitude Spectrum of SSB-Modulated Signals

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:

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

Listing 5: FDM Multiplexing Code

The magnitude spectrum of the multiplexed signal is shown below:

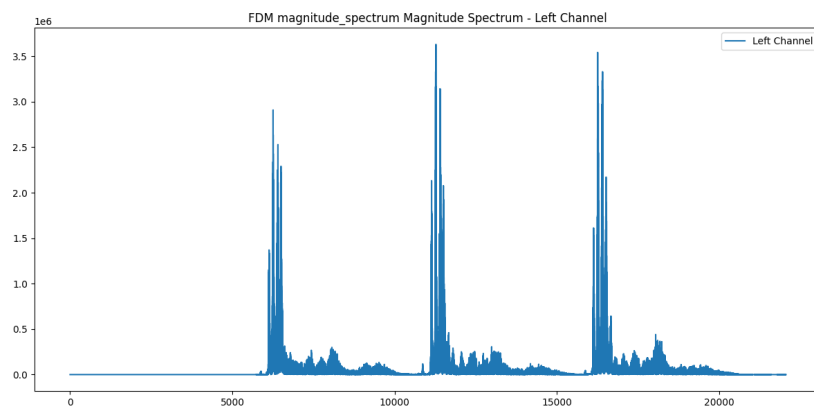


Figure 5: Magnitude Spectrum of Multiplexed Signals

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:

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

Listing 6: Bandpass Filtering Code

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

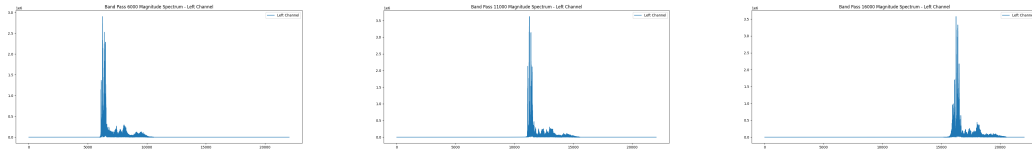


Figure 6: Magnitude Spectrum of Bandpass Filtered Signals

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.

Below is the code used for demodulation:

```

1 # Demodulate the signal using a cosine wave
2 sample_rate, signal = read(os.path.join(separated_audios_dir,
3     file))
4 t = np.linspace(0, DURATION, sample_rate * DURATION, endpoint
5     =False) # Time vector
6 carrier = np.cos(2 * np.pi * carrier_frequencies[i] * t)
7 demodulated_signal = AMPLIFYING_FACTOR * signal * carrier
8 demodulated_signal = Filterer.low_pass_filter(
9     demodulated_signal, sample_rate, LIMIT_FREQUENCY,
10    FilterType.LOW_PASS_BUTTERWORTH)

```

Listing 7: Demodulation Code

The demodulated signals for each input signal are shown below:

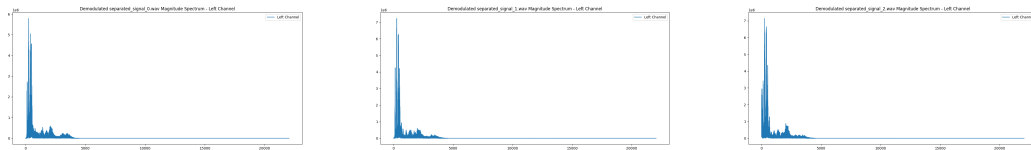


Figure 7: Magnitude Spectrum of Bandpass Filtered Signals

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 MATLAB code used in the project:

```
1  """
2  This file contains the common includes for the project.
3  """
4  from enum import Enum
5
6  DURATION = 10
7  SAMPLE_RATE = 44100
8  LIMIT_FREQUENCY = 4000
9  TOLERANCE_FREQUENCY = 500
10 AMPLIFYING_FACTOR = 4
11
12 class FilterType(Enum):
13     LOW_PASS_FOURIER = 1
14     LOW_PASS_BUTTERWORTH = 2
15     HIGH_PASS_IDEAL = 3
16     HIGH_PASS_BUTTERWORTH = 4
17     BAND_PASS_IDEAL = 5
18     BAND_PASS_BUTTERWORTH = 6
```

Listing 8: Common Includes

```
1  """
2  This module records audio from the microphone and saves it to
3  a file.
4  """
5  import sounddevice as sd
6  from scipy.io.wavfile import write
7  import sys
8  import os
9  from common_includes import *
10
11 def record_audio(filename: str) -> None:
12     """
13     Record audio from the microphone and save it to a file.
14
15     Args:
16         filename (str): The path to save the audio file to.
17     """
18     input("Press Enter to start recording...")
19     print(f"Recording audio to {filename} for {DURATION} seconds")
20     audio = sd.rec(int(DURATION * SAMPLE_RATE),
21                    samplerate=SAMPLE_RATE, channels=2, dtype=
22                        'int16')
```

```

22     sd.wait()
23     write(filename, SAMPLE_RATE, audio)
24     print(f"Audio saved to {filename}")
25     print()
26
27
28 if __name__ == "__main__":
29     is_test = "--test" in sys.argv
30     save_dir = os.path.join(os.path.dirname(
31         os.path.abspath(__file__)), "..", "data", "input")
32     if not os.path.exists(save_dir):
33         os.makedirs(save_dir)
34     if is_test:
35         from plot_signal import Plotter
36         test_filename = os.path.join(save_dir, "test_input.
37             wav")
38         record_audio(test_filename)
39         Plotter.plot_wavfile(
40             test_filename, "Audio Waveform", "Time (Seconds)",
41             "Amplitude")
42         Plotter.plot_wavfile_magnitude_spectrum(
43             test_filename, "Audio Magnitude Spectrum", "
44             Frequency (Hz)", "Magnitude")
45     else:
46         record_audio(os.path.join(save_dir, "input1.wav"))
47         record_audio(os.path.join(save_dir, "input2.wav"))
48         record_audio(os.path.join(save_dir, "input3.wav"))

```

Listing 9: Record Audio

```

1  import numpy as np
2  from scipy.signal import butter, filtfilt, sosfilt
3  from scipy.io.wavfile import write, read
4  import os
5  from common_includes import *
6  from plot_signal import Plotter
7
8
9  class Filterer:
10     @classmethod
11     def low_pass_filter(cls, signal: np.ndarray, sample_rate:
12         int, cutoff_frequency: float, mode: FilterType=
13         FilterType.LOW_PASS_BUTTERWORTH) -> np.ndarray:
14         """
15         Apply a low-pass filter to a signal.
16
17         Args:
18             signal (np.ndarray): The signal to filter.
19             cutoff_frequency (float): The cutoff frequency of

```

```

18         the filter.
19         mode (FilterType): The type of filter to apply.
20
21     Returns:
22         np.ndarray: The filtered signal.
23     """
24     if mode == FilterType.LOW_PASS_FOURIER:
25         return cls._low_pass_fourier(signal, sample_rate,
26                                     cutoff_frequency)
27     elif mode == FilterType.LOW_PASS_BUTTERWORTH:
28         return cls._low_pass_butterworth(signal,
29                                         sample_rate, cutoff_frequency)
30     else:
31         raise ValueError(f"Invalid filter mode: {mode}")
32
33 @classmethod
34 def suppress_lower_sideband(cls, signal: np.ndarray,
35                             sample_rate: int, carrier_freq: float, mode:
36                             FilterType=FilterType.HIGH_PASS_BUTTERWORTH) -> np.
37                             ndarray:
38     """
39     Suppress all frequencies below the carrier frequency
40     using a high-pass filter.
41     This converts the signal to Single Sideband (SSB) by
42     removing the lower sideband.
43
44     Args:
45     - signal (np.ndarray): The signal to filter.
46     - sample_rate (int): The sampling frequency of the
47       signal.
48     - carrier_freq (float): The carrier frequency around
49       which modulation happens.
50     - mode (FilterType): The type of high-pass filter to
51       apply.
52
53     Returns:
54     - np.ndarray: The filtered signal with the lower
55       sideband suppressed.
56     """
57     if mode == FilterType.HIGH_PASS_IDEAL:
58         return cls._suppress_lower_sideband_ideal(signal,
59                                                   sample_rate, carrier_freq)
60     elif mode == FilterType.HIGH_PASS_BUTTERWORTH:
61         return cls._suppress_lower_sideband_butterworth(
62             signal, sample_rate, carrier_freq)
63     else:
64         raise ValueError(f"Invalid filter mode: {mode}")
65
66 @classmethod

```

```

53 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:
54     """
55     Apply a band-pass filter to a signal.
56
57     Args:
58     - signal (np.ndarray): The signal to filter.
59     - sample_rate (int): The sampling frequency of the
        signal.
60     - low_cutoff (float): The lower bound of the passband
        .
61     - high_cutoff (float): The upper bound of the
        passband.
62     - mode (FilterType): The type of filter to apply.
63
64     Returns:
65     - np.ndarray: The filtered signal
66     """
67     if mode == FilterType.BAND_PASS_IDEAL:
68         return cls._band_pass_filter_ideal(signal,
            sample_rate, low_cutoff, high_cutoff)
69     elif mode == FilterType.BAND_PASS_BUTTERWORTH:
70         return cls._band_pass_filter_butterworth(signal,
            sample_rate, low_cutoff, high_cutoff)
71     else:
72         raise ValueError(f"Invalid filter mode: {mode}")
73
74 @classmethod
75 def _suppress_lower_sideband_ideal(cls, signal: np.
ndarray, sample_rate: int, carrier_freq: float) -> np.
ndarray:
76     """
77     Suppress all frequencies below the carrier frequency
78     using an ideal high-pass filter.
79     This converts the signal to Single Sideband (SSB) by
80     removing the lower sideband.
81
82     Parameters:
83     - signal: Input signal (1D numpy array).
84     - sample_rate: Sampling frequency (Hz).
85     - carrier_freq: Carrier frequency around which
86     modulation happens (Hz).
87
88     Returns:
89     - Filtered signal with the lower sideband suppressed
90     (SSB).
91     """

```

```

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,
132     sample_rate: int, low_cutoff: float, high_cutoff:
133     float) -> np.ndarray:
134     """
135     Apply an ideal band-pass filter to a signal, passing
136     only the frequencies between low_cutoff and
137     high_cutoff.
138
139     Parameters:
140     - signal: Input signal (1D numpy array).
141     - sample_rate: Sampling frequency (Hz).
142     - low_cutoff: Lower bound of the passband (Hz).
143     - high_cutoff: Upper bound of the passband (Hz).
144
145     Returns:
146     - Filtered signal with only frequencies between
147       low_cutoff and high_cutoff passed.
148     """
149     # Perform Fourier Transform (FFT) of the signal
150     spectrum = np.fft.fft(signal)
151     freqs = np.fft.fftfreq(len(signal), d=1/sample_rate)
152
153     # Create an ideal band-pass filter: 1 for frequencies
154     # in range, 0 otherwise
155     band_pass_filter = (np.abs(freqs) >= low_cutoff) & (
156         np.abs(freqs) <= high_cutoff)
157
158     # Apply the band-pass filter by zeroing out
159     # frequencies outside the passband
160     spectrum[~band_pass_filter] = 0
161
162     # Perform Inverse Fourier Transform (IFFT) to get the
163     # filtered signal in the time domain
164     filtered_signal = np.fft.ifft(spectrum).real
165
166     return filtered_signal
167
168 @classmethod
169 def _band_pass_filter_butterworth(cls, signal: np.ndarray
170     , sample_rate: int, low_cutoff: float, high_cutoff:
171     float, order: int = 100) -> np.ndarray:
172     """

```

```

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

163
164     Parameters:
165     - signal: Input signal (1D numpy array).
166     - sample_rate: Sampling frequency (Hz).
167     - low_cutoff: Lower bound of the passband (Hz).
168     - high_cutoff: Upper bound of the passband (Hz).
169     - order: Order of the Butterworth filter (default: 5)
170
171     Returns:
172     - Filtered signal with only frequencies between
       low_cutoff and high_cutoff passed.
173     """
174     # Normalize the cutoff frequencies to the Nyquist
       frequency (half the sample rate)
175     nyquist_freq = sample_rate / 2.0
176     low = low_cutoff / nyquist_freq
177     high = high_cutoff / nyquist_freq
178
179     # Design a Butterworth band-pass filter as second-
       order sections (SOS)
180     sos = butter(order, [low, high], btype='band', analog
       =False, output='sos')
181
182     # Apply the filter to the signal
183     filtered_signal = sosfilt(sos, signal)
184
185     return filtered_signal
186
187     @classmethod
188     def _low_pass_fourier(cls, signal: np.ndarray,
       sample_rate: int, cutoff_frequency: float) -> np.
       ndarray:
189         """
190         Apply a low-pass Fourier filter to a signal.
191
192         Args:
193             signal (np.ndarray): The signal to filter.
194             cutoff_frequency (float): The cutoff frequency of
               the filter.
195
196         Returns:
197             np.ndarray: The filtered signal.
198         """
199         print("Applying Fourier filter")
200         nyquist = 0.5 * sample_rate

```

```

201         cutoff_idx = int(cutoff_frequency / nyquist * (len(
202             signal) // 2))
203         signal_fft = np.abs(np.fft.rfft(signal, axis=0))
204         freqs = np.fft.rfftfreq(len(signal), d=1/sample_rate)
205         signal_fft[cutoff_idx:] = 0
206         filtered_signal = np.fft.irfft(signal_fft, n=len(
207             signal), axis=0)
208         return filtered_signal
209
210 @classmethod
211 def _low_pass_butterworth(cls, signal: np.ndarray,
212     sample_rate: int, cutoff_frequency: float, order: int
213     =8) -> np.ndarray:
214     """
215     Apply a low-pass Butterworth filter to a signal.
216
217     Args:
218         signal (np.ndarray): The signal to filter.
219         cutoff_frequency (float): The cutoff frequency of
220             the filter.
221
222     Returns:
223         np.ndarray: The filtered signal.
224     """
225     if len(signal.shape) > 1:
226         signal_left = signal[:, 0]
227         signal_right = signal[:, 1]
228     else:
229         signal_left = signal
230         signal_right = None
231
232     # Design the low-pass filter
233     nyquist_rate = sample_rate / 2.0 # Nyquist frequency
234     normal_cutoff = cutoff_frequency / nyquist_rate #
235         Normalize cutoff frequency
236
237     # Validate cutoff frequency
238     if not (0 < normal_cutoff < 1):
239         raise ValueError("Cutoff frequency must be
240             between 0 and Nyquist frequency.")
241
242     # Design Butterworth filter
243     b, a = butter(order, normal_cutoff, btype='low',
244         analog=False)
245
246     # Apply the filter to the signal
247     filtered_left = filtfilt(b, a, signal_left)
248     filtered_right = None
249     if signal_right is not None:

```

```

242         filtered_right = filtfilt(b, a, signal_right)
243
244     # Combine channels back (stereo if both channels
245     # exist)
246     if filtered_right is not None:
247         filtered_signal = np.vstack((filtered_left,
248                                     filtered_right)).T
249     else:
250         filtered_signal = filtered_left
251
252     return filtered_signal
253
254 if __name__ == "__main__":
255     input_dir = os.path.join(os.path.dirname(os.path.abspath(
256         __file__)), "..", "data", "input")
257     input_magnitude_spectrum_dir = os.path.join(input_dir, "
258         magnitude_spectrum")
259     save_dir = os.path.join(os.path.dirname(os.path.abspath(
260         __file__)), "..", "data", "filtered")
261     save_magnitude_spectrum_dir = os.path.join(save_dir, "
262         magnitude_spectrum")
263
264     if not os.path.exists(save_dir):
265         os.makedirs(save_dir)
266
267     # loop through all the files in the data directory
268     for file in os.listdir(input_dir):
269         if file.endswith(".wav"):
270             # read the file
271             sample_rate, signal = read(os.path.join(input_dir,
272             , file))
273             # filter the signal
274             filtered_signal = Filterer.low_pass_filter(signal
275             , sample_rate, LIMIT_FREQUENCY, FilterType.
276             LOW_PASS_BUTTERWORTH)
277             # save the filtered signal
278             write(os.path.join(save_dir, file), sample_rate,
279             filtered_signal.astype(np.int16))
280             # plot the magnitude spectrum of the original
281             signal
282             Plotter.plot_magnitude_spectrum(signal,
283             sample_rate, title=f"original {file} Magnitude
284             Spectrum", save_dir=
285             input_magnitude_spectrum_dir, file_name=f"
286             original_{file}_magnitude_spectrum.png")
287             # plot the magnitude spectrum of the filtered
288             signal
289             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

```

1  import numpy as np
2  from scipy.signal import butter, filtfilt
3  from scipy.io.wavfile import write, read
4  import os
5  from common_includes import *
6  from plot_signal import Plotter
7  from filter_signal import Filterer
8
9
10 carrier_frequencies = [6000, 11000, 16000] # Carrier
    frequencies for FDM
11
12
13 if __name__ == "__main__":
14     # path filtered signals
15     filtered_dir = os.path.join(os.path.dirname(os.path.
        abspath(__file__)), "..", "data", "filtered")
16     # path modulated signals
17     modulated_dir = os.path.join(os.path.dirname(os.path.
        abspath(__file__)), "..", "data", "modulated")
18     modulated_magnitude_spectrum_dir = os.path.join(
        modulated_dir, "magnitude_spectrum")
19     fdm_magnitude_spectrum_dir = os.path.join(modulated_dir,
        "fdm_spectrum")
20     dsb_magnitude_spectrum_dir = os.path.join(modulated_dir,
        "dsb_spectrum")
21
22
23     if not os.path.exists(modulated_dir):
24         os.makedirs(modulated_dir)
25     if not os.path.exists(modulated_magnitude_spectrum_dir):
26         os.makedirs(modulated_magnitude_spectrum_dir)
27     if not os.path.exists(fdm_magnitude_spectrum_dir):
28         os.makedirs(fdm_magnitude_spectrum_dir)
29
30     # Modulation and FDM
31     modulated_signals = []
32     for i, file in enumerate(os.listdir(filtered_dir)):
33         if file.endswith(".wav"):
34             sample_rate, signal = read(os.path.join(
                filtered_dir, file))
35             left_channel = signal[:, 0]

```

```

36     #right_channel = signal[:, 1]
37     # Generate carrier signal
38
39     t = np.linspace(0, DURATION, sample_rate *
40                     DURATION, endpoint=False) # Time vector
41     carrier = np.cos(2 * np.pi * carrier_frequencies[
42                     i] * t)
43
44     modulated_left = left_channel * carrier
45     #modulated_right = right_channel * carrier
46
47     # Combine the modulated channels back into a
48     # stereo signal
49     #modulated_stereo_signal = np.column_stack((
50     #    modulated_left, modulated_right))
51     Plotter.plot_magnitude_spectrum(modulated_left,
52     sample_rate, mono=True, title=f"dsb {file}
53     Magnitude Spectrum", save_dir=
54     dsb_magnitude_spectrum_dir, file_name=f"dsb_{
55     file}_magnitude_spectrum.png")
56     # Suppress lower sideband
57     ssb_signal = Filterer.suppress_lower_sideband(
58     modulated_left, sample_rate,
59     carrier_frequencies[i])
60     # plot the magnitude spectrum of the filtered
61     # signal
62     Plotter.plot_magnitude_spectrum(ssb_signal,
63     sample_rate, mono=True, title=f"Modulated {
64     file} Magnitude Spectrum", save_dir=
65     modulated_magnitude_spectrum_dir, file_name=f"
66     modulated_{file}_magnitude_spectrum.png")
67     modulated_signals.append(ssb_signal)
68
69     # Sum all modulated signals to form the FDM signal
70     fdm_signal = np.sum(modulated_signals, axis=0)
71
72     Plotter.plot_magnitude_spectrum(fdm_signal, SAMPLE_RATE,
73     mono=True, title=f"FDM {file} Magnitude Spectrum",
74     save_dir=fdm_magnitude_spectrum_dir, file_name=f"FDM_{
75     file}_magnitude_spectrum.png")
76
77     # Save the FDM signal
78     write(os.path.join(modulated_dir, "fdm_signal.wav"),
79     sample_rate, fdm_signal.astype(np.int16))

```

Listing 11: Modulator

```

1 import numpy as np
2 from scipy.signal import butter, filtfilt
3 from scipy.io.wavfile import write, read
4 import os
5 from common_includes import *
6 from plot_signal import Plotter
7 from filter_signal import Filterer
8
9
10 carrier_frequencies = [6000, 11000, 16000] # Carrier
    frequencies for FDM
11
12
13 if __name__ == "__main__":
14     # path modulated signals
15     modulated_dir = os.path.join(os.path.dirname(os.path.
        abspath(__file__)), "..", "data", "modulated")
16     # path demodulated signals
17     demodulated_dir = os.path.join(os.path.dirname(os.path.
        abspath(__file__)), "..", "data", "demodulated")
18     separated_spectrums_dir = os.path.join(demodulated_dir, "
        separated_spectrums")
19     separated_audios_dir = os.path.join(demodulated_dir, "
        separated_audios")
20     signals_after_demodulation_spectrum_dir = os.path.join(
        demodulated_dir, "signals_after_demodulation_spectrum"
        )
21     signals_after_demodulation_audios_dir = os.path.join(
        demodulated_dir, "signals_after_demodulation_audios")
22
23     if not os.path.exists(demodulated_dir):
24         os.makedirs(demodulated_dir)
25
26     if not os.path.exists(separated_spectrums_dir):
27         os.makedirs(separated_spectrums_dir)
28
29     if not os.path.exists(separated_audios_dir):
30         os.makedirs(separated_audios_dir)
31
32     if not os.path.exists(
        signals_after_demodulation_spectrum_dir):
33         os.makedirs(signals_after_demodulation_spectrum_dir)
34
35     if not os.path.exists(
        signals_after_demodulation_audios_dir):
36         os.makedirs(signals_after_demodulation_audios_dir)
37
38
39     for file in os.listdir(modulated_dir):

```

```

40     if file.endswith(".wav"):
41         sample_rate, signal = read(os.path.join(
42             modulated_dir, file))
43         # iterate through the carrier frequencies and do
44         # band pass filter from filter_signal.py and add
45         # them to signalsarray
46         for i, carrier_frequency in enumerate(
47             carrier_frequencies):
48             band_pass_signal = Filterer.band_pass_filter(
49                 signal, sample_rate, carrier_frequency,
50                 carrier_frequency + LIMIT_FREQUENCY +
51                 TOLERANCE_FREQUENCY)
52             # plot the magnitude spectrum of the filtered
53             # signal
54             Plotter.plot_magnitude_spectrum(
55                 band_pass_signal, sample_rate, mono=True,
56                 title=f"Band Pass {carrier_frequency}
57                     Magnitude Spectrum", save_dir=
58                     separated_spectrums_dir, file_name=f"
59                     band_pass_{carrier_frequency}
60                     _magnitude_spectrum.png")
61             # save the filtered signal
62             write(os.path.join(separated_audios_dir, f"
63                 separated_signal_{i}.wav"), sample_rate,
64                 band_pass_signal.astype(np.int16))
65
66     for i, file in enumerate(os.listdir(separated_audios_dir)
67 ):
68         if file.endswith(".wav"):
69             sample_rate, signal = read(os.path.join(
70                 separated_audios_dir, file))
71
72             t = np.linspace(0, DURATION, sample_rate *
73                 DURATION, endpoint=False) # Time vector
74             carrier = np.cos(2 * np.pi * carrier_frequencies[
75                 i] * t)
76
77             demodulated_signal = AMPLIFYING_FACTOR * signal *
78                 carrier
79
80             demodulated_signal = Filterer.low_pass_filter(
81                 demodulated_signal, sample_rate,
82                 LIMIT_FREQUENCY, FilterType.
83                 LOW_PASS_BUTTERWORTH)
84             # plot the magnitude spectrum of the filtered
85             # signal
86             Plotter.plot_magnitude_spectrum(
87                 demodulated_signal, sample_rate, mono=True,

```



```

63         title=f"Demodulated {file} Magnitude Spectrum"
64         , save_dir=
        signals_after_demodulation_spectrum_dir,
        file_name=f"out{i}.png")

        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,
10             y_label=None):
11         plt.title(title)
12         plt.plot(time, signal)
13         plt.xlabel(x_label)
14         plt.ylabel(y_label)
15         plt.show()
16
17     @classmethod
18     def plot_wavfile(cls, filename, title=None, x_label=None,
19                    y_label=None):
20         sample_rate, signal = wavfile.read(filename)
21         if len(signal.shape) == 2:
22             signal = signal[:, 0]
23         time = np.arange(len(signal)) / sample_rate
24         cls.plot(time, signal, title, x_label, y_label)
25
26     @classmethod
27     def plot_wavfile_magnitude_spectrum(cls, filename, title=
28         None, x_label=None, y_label=None):
29         sample_rate, signal = wavfile.read(filename)
30
31         if len(signal.shape) == 1: # Mono
32             cls.plot_magnitude_spectrum(signal, sample_rate,
33                 title, x_label, y_label, mono=True)
34         elif len(signal.shape) == 2 and signal.shape[1] == 2:
35             # Stereo
36             cls.plot_magnitude_spectrum(signal, sample_rate,
37                 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