

430 Term Project Part II Report

1. Introduction

This project focuses on the development and implementation of a robust receiver for decoding Frequency Hopping Spread Spectrum (FHSS) signals categorized as ‘Category 2’ and ‘Category 3.’ The receiver includes a graphical user interface (GUI) capable of recording audio signals using the PC’s microphone, with adjustable sampling rate and recording duration. The recorded signal is analyzed using a spectrogram, with additional noise reduction techniques applied as needed, and both pre- and post-noise reduction spectrograms are displayed in the GUI. Synchronization between the transmitter and receiver is achieved using a pilot signal, allowing the receiver to identify the start of the transmission and track the hopping frequencies. Alternatively, frequency changes in the received signal are analyzed to determine the hop sequence without prior synchronization. The decoding process leverages time-frequency analysis to extract the hopping frequencies and demodulate the FSK-modulated signal, where the frequency deviation from the known hopping frequency corresponds to digital data. This data is subsequently decoded and displayed in the original text message. The receiver adheres to the specific requirements of ‘Category 2’ and ‘Category 3’ by accommodating the respective encoding-decoding rules and modulation orders, demonstrating successful recovery of transmitted messages over varying distances and signal conditions.

2. Noise Reduction

Bandpass Filtering eliminates the noise outside the defined frequency band, improving the signal-to-noise ratio (SNR) for the relevant frequencies and Spectral Thresholding enhances robustness against faint noise peaks by focusing only on significant spectral components.

2.1 Bandpass Filtering

Bandpass filtering is particularly effective for isolating the signal of interest in scenarios where the transmitted signal is surrounded by noise across a broader frequency spectrum. By selecting cutoff frequencies corresponding to the range of expected hop frequencies, the filter dynamically eliminates frequency components outside this range. The choice of a Butterworth filter ensures a maximally flat frequency response in the passband, minimizing distortion while attenuating unwanted signals. The 4th-order design strikes a balance between steep roll-off and computational efficiency, ensuring the filter can handle real-time processing without introducing excessive latency. However, it is important to note that while bandpass filtering significantly improves the signal-to-noise ratio (SNR), it does not address in-band noise or interference, which may require additional noise reduction techniques.

- Butterworth Filter

The Butterworth filter has the following properties:

The magnitude response $|H(e^{j\omega})|$ is maximally flat in the passband and decreases monotonically in the stopband. For a 4th-order Butterworth filter, the frequency response

$|H(f)|$ is given by:

$$|H(f)| = \frac{1}{\sqrt{1 + (\frac{f}{f_c})^2}} \quad (1)$$

A 4th-order Butterworth bandpass filter was applied to the received signal to focus on the frequency range of interest (e.g., [250 Hz - 11,000 Hz] for 8-FSK) as shown in Figure 1. The Nyquist frequency is half the sampling rate. It represents the highest frequency that can be accurately represented in the sampled data. The filter's cutoff frequencies are normalized with respect to this Nyquist frequency. The *butter* function designs a Butterworth filter, known for its maximally flat frequency response in the passband (no ripples). This specifies a 4th-order filter, which provides a steeper roll-off at the cutoff frequencies compared to the lower-order filters.

```
function filteredSignal = applyBandpassFilter(receivedSignal, samplingRate, lowFreq, highFreq)
    % Bandpass filter to focus on frequencies dynamically
    nyquist = samplingRate / 2;
    [b, a] = butter(4, [lowFreq, highFreq] / nyquist, 'bandpass');
    filteredSignal = filter(b, a, receivedSignal);
end
```

Figure 1: The Bandpass Filter Function.

This method removes components outside the specified frequency range and preserves the frequencies within the band, in other words reduces out-of-band noise while preserving the signal components essential for decoding.

2.2 Spectral Thresholding

Spectral thresholding is a sophisticated noise reduction technique applied in the frequency domain, widely used to enhance the quality of signals corrupted by noise. This method involves analyzing the spectrogram of the signal and selectively attenuating or discarding frequency components whose energy levels fall below a predefined threshold. The thresholding process effectively suppresses low-energy noise while retaining significant spectral components, thereby improving the signal-to-noise ratio (SNR). This technique is particularly effective in scenarios where the noise is broadband, spread across a wide frequency range, while the signal is characterized by concentrated energy peaks at specific

frequencies. By selectively focusing on these high-energy regions, the method preserves the fidelity of the desired signal components.

The method is implemented as depicted in Figure 2 :

```
function maxFreqs = getMaxEnergyFrequencies(filteredSignal, samplingRate, hopPeriod, pilotFreq, pilotTolerance)

    hopSamples = round(hopPeriod * samplingRate);
    numHops = ceil(length(filteredSignal) / hopSamples);
    maxFreqs = zeros(1, numHops);

    [S, F, T] = spectrogram(filteredSignal, 256, 200, 256, samplingRate);

    for i = 1:numHops
        startIdx = (i-1) * hopSamples + 1;
        endIdx = min(i * hopSamples, length(filteredSignal));
        timeRange = (T >= (startIdx - 1) / samplingRate) & (T <= endIdx / samplingRate);

        S_slice = S(:, timeRange);
        [~, maxIdx] = max(sum(abs(S_slice), 2));
        maxFreqs(i) = F(maxIdx);

        % Check for pilot signal
        if abs(maxFreqs(i) - pilotFreq) < pilotTolerance
            disp(['Pilot signal detected at frequency: ', num2str(maxFreqs(i)), ' Hz']);
        end
    end

    disp('Maximum Energy Frequencies for Each Hop:');
    disp(maxFreqs);
end
```

Figure 2: The Spectral Threshold Function.

Also, there are inherent trade-offs in this technique. Setting the threshold too high can inadvertently discard weak but meaningful signal components, causing loss of critical data. On the other hand, a threshold that is too low might fail to suppress noise adequately, thereby diminishing the enhancement of SNR. Modern implementations often integrate filtering techniques or machine learning-based classifiers to optimize the thresholding process, minimizing the risk of these pitfalls.

3. Frequency Hopping Synchronization

Frequency Hopping Spread Spectrum (FHSS) relies on the ability of both the transmitter and receiver to maintain synchronization. Synchronization ensures that the receiver can correctly follow the hopping pattern of the transmitter, accurately identifying the signal frequencies for each hop period. The process is as given below:


Firstly the transmitter sends a tone at a fixed frequency (e.g., 850 Hz) for a short duration before the first hop. Then the receiver listens for this tone using spectrogram analysis, identifying the presence of the pilot frequency within a predefined tolerance range (e.g., ± 50 Hz). Upon detecting the pilot signal, the receiver starts its internal clock to synchronize with the hop sequence.

3.1 Pilot Signal Detection


A predetermined pilot frequency, defined in both transmitter and receiver, (e.g., 850 Hz) was embedded at the start and end of each hop sequence and the spectrogram analysis was used to identify the pilot signal within a defined tolerance range (e.g., ± 50 Hz) as seen in Figure 3.

```
if abs(detectedFreq - pilotFrequency) < tolerance

    if isempty(startIdx)
        startIdx = startSampleIdx;
        disp('First pilot signal detected. Start processing.');
```



```
    elseif isempty(endIdx)
        endIdx = startSampleIdx;
        disp('Second pilot signal detected. Stop processing.');
```



```
        break;
    end
    continue;
end

% storing detected frequencies only between pilot signals
if ~isempty(startIdx) && isempty(endIdx)
    detectedFrequencies = [detectedFrequencies, detectedFreq];
end
end
```

Figure 3: The Pilot Frequency Detection.

The example usage of pilot frequencies in 500 Hz with 60 Hz tolerance is given below in Figure 4:

```
Detected Frequency: 468.75
First pilot signal detected. Start processing.
Detected Frequency: 5390.625
Detected Frequency: 9375
Detected Frequency: 2226.5625
Detected Frequency: 4218.75
Detected Frequency: 1054.6875
Detected Frequency: 1992.1875
Detected Frequency: 3750
Detected Frequency: 5039.0625
Detected Frequency: 5390.625
Detected Frequency: 7148.4375
Detected Frequency: 6210.9375
Detected Frequency: 468.75
Second pilot signal detected. Stop processing.
```

Figure 4: Example Usage of Pilot Frequency Detection.

By this method, we ensure that only the detected frequencies between pilot frequencies are used in frequency hopping.

3.2 Hop Timing Alignment

Once the pilot frequency was detected, subsequent hop periods were synchronized by dividing the incoming signal into segments corresponding to the hop period, which was calculated as the hop period * sampling rate. This ensures proper alignment of the signal with the expected hopping pattern. A random hop order is generated using the same seed as the transmitter $rng(1)$, ensuring synchronization with the transmitted frequency table derived from the student's unique ID.

4. Signal Demodulation

The frequency with the highest energy corresponds to the transmitted frequency for that hop since noise components typically have lower energy compared to the modulated signal. To extract the correct frequencies, we apply some techniques.

4.1 Maximum Energy Frequency Extraction

This process includes identifying the frequency with the highest energy during each hop period. This frequency corresponds to the modulated signal during that time segment.

Firstly, a spectrogram is generated for the filtered signal. The spectrogram represents the signal's energy distribution over time and frequency. For each hop period (time segment), a slice of the spectrogram corresponding to that period is isolated and the frequency with the maximum energy in this slice is identified as the dominant frequency for that hop.

The mathematical representation is as follows:

$$f_{max} = \operatorname{argmax} E(f, t) \quad (2)$$

The related code is shown in Figure 5:

```
[S, F, ~] = spectrogram(hopSegment, 256, 200, 256, samplingRate);  
[~, maxIdx] = max(abs(S), [], 1);  
detectedFreq = F(maxIdx(1)); % Extract detected frequency  
disp(['Detected Frequency: ', num2str(detectedFreq)]);
```

Figure 5: The Maximum Energy Frequency Extraction.

4.2 Frequency Deviation ($\Delta f m_k$) Calculation

The $\Delta f m_k$ calculation and modulation mapping ensure that the received signal is translated accurately into its corresponding binary representation. The process involves

extracting the frequency deviation and comparing it against pre-defined levels for each modulation order.

This process includes calculating the difference between the detected frequency and the known hop base frequency for the current hop. This deviation encodes the digital data. The obtained detected and hop base frequency pairs help to identify the message. The example of obtained detected-hop base frequency pairs from the student number “ 2575306 ” in the receiver screen of a 4-FSK modulation is shown in Figure 6:

```
Detected Frequency: 1171.875, Hop Frequency: 900
Matched 11
Detected Frequency: 3203.125, Hop Frequency: 2900
Matched 11
Detected Frequency: 1718.75, Hop Frequency: 1900
Matched 01
Detected Frequency: 1015.625, Hop Frequency: 900
Matched 10
Detected Frequency: 1562.5, Hop Frequency: 1900
Matched 00
Detected Frequency: 3203.125, Hop Frequency: 2900
Matched 11
Detected Frequency: 3750, Hop Frequency: 3900
Matched 01
Detected Frequency: 5078.125, Hop Frequency: 4900
Matched 10
Detected Frequency: 3593.75, Hop Frequency: 3900
Matched 00
Detected Frequency: 5781.25, Hop Frequency: 5900
Matched 01
Detected Frequency: 1718.75, Hop Frequency: 1900
Matched 01
Detected Frequency: 8046.875, Hop Frequency: 7900
Matched 10
Detected Frequency: 1015.625, Hop Frequency: 900
Matched 10
Detected Frequency: 6171.875, Hop Frequency: 5900
Matched 11
```

Figure 6: Obtained Detected-Hop Base Frequency Pairs in a 4-FSK.

The deviation corresponds to the modulation index or offset introduced by the digital data. This relationship allows the receiver to map $\Delta f m_k$ to binary values, reconstructing the

transmitted message. Firstly the transmitted signal hops between pre-determined frequencies (base hop frequencies) from a frequency table. The base hop frequency for a hop is the frequency selected from this table for the current time interval. Then, using spectrogram analysis, the frequency with the maximum energy during the hop period is identified as the detected frequency, and the frequency deviation $\Delta f m_k$ is calculated as the difference

between the detected frequency $f_{detected}$ and the base frequency f_{base} :

$$\Delta f m_k = f_{detected} - f_{base}$$

Since the deviation corresponds to the modulation index or offset introduced by the digital data, this relationship allows the receiver to map $\Delta f m_k$ to binary values, reconstructing the transmitted message.

The modulation order determines how $\Delta f m_k$ is interpreted and mapped to binary data.

4.3 Modulation Mapping

- **2-FSK**

Two possible frequencies represent binary values 0 and 1:

$$1) \quad f_{detected} - f_{base} = -\Delta f, m_k = -1 \Rightarrow \text{binary data} = 0$$

$$2) \quad f_{detected} - f_{base} = +\Delta f, m_k = +1 \Rightarrow \text{binary data} = 1$$

The code implementation of 4-FSK demodulation is as follows in Figure 7:

```
if modulationOrder == 2 % 2-FSK
    % Map frequency difference to binary 0 or 1
    if abs(detectedFreqs - (hopFreq - deltaF)) < abs(detectedFreqs - (hopFreq + deltaF))
        binaryDatas = [binaryDatas, 0];
    else
        binaryDatas = [binaryDatas, 1];
    end
end
```

Figure 7: The 2-FSK Demodulation.

- **4-FSK**

Four possible frequencies represent binary values 00, 01, 10, and 11:

$$1) \quad f_{detected} - f_{base} = -2\Delta f, m_k = -2 \Rightarrow \text{binary data} = 00$$

$$2) \quad f_{detected} - f_{base} = -\Delta f, m_k = -1 \Rightarrow \text{binary data} = 01$$

$$3) \quad f_{detected} - f_{base} = +\Delta f, m_k = +1 \Rightarrow \text{binary data} = 10$$

$$4) \quad f_{detected} - f_{base} = +2\Delta f, m_k = +2 \Rightarrow \text{binary data} = 11$$

The code implementation of 4-FSK demodulation is as follows in Figure 8:

```

elseif modulationOrder == 4 % 4-FSK
    % Map frequency difference to 2-bit binary (00, 01, 10, 11)
    if abs(detectedFreqs - (hopFreq - 2 * deltaF)) < tolerance
        disp('Matched 00');
        binaryDatas = [binaryDatas, 0, 0];
    elseif abs(detectedFreqs - (hopFreq - deltaF)) < tolerance
        disp('Matched 01');
        binaryDatas = [binaryDatas, 0, 1];
    elseif abs(detectedFreqs - (hopFreq + deltaF)) < tolerance
        disp('Matched 10');
        binaryDatas = [binaryDatas, 1, 0];
    elseif abs(detectedFreqs - (hopFreq + 2 * deltaF)) < tolerance
        disp('Matched 11');
        binaryDatas = [binaryDatas, 1, 1];
    end

```

Figure 8: The 4-FSK Demodulation.

So we ensure the binary data matching by defining a tolerance to suppress minimal frequency shiftings.

- A Category 2 Example:

Let's define parameters as:

- ❖ Student Number: 2575306
- ❖ $\Delta f = 150$ Hz

By the rule of Category 2:

'row 2' = [1500 + bS, 2500 + bS, 3500 + bS, 4500 + bS, 5500 + bS, 6500 + bS, 7500 + bS, 8500 + bS] where b = 1 if N5 is even, b = -1 otherwise. S = 100 · N7.

The N5=3 which is odd, b = -1 and N7 = 6 \Rightarrow bS = - 600

'row 2' becomes:

'row 2' = [1500 - 600, 2500 - 600, 3500 - 600, 4500 - 600, 5500 - 600, 6500 - 600, 7500 - 600, 8500 - 600] = [900, 1900, 2900, 3900, 4900, 5900, 6900, 7900]

Hence,

$f_{base} = [900, 1900, 2900, 3900, 4900, 5900, 6900, 7900]$

The $f_{detected}$ frequencies obtained in the receiver are shown in Figure 9:


```

Detected Frequency: 1171.875, Hop Frequency: 900
Matched 11
Detected Frequency: 3203.125, Hop Frequency: 2900
Matched 11
Detected Frequency: 1718.75, Hop Frequency: 1900
Matched 01
Detected Frequency: 1015.625, Hop Frequency: 900
Matched 10

```

Figure 9: Category 2 Example Frequency Pairs.

According to the hop order, a part of the hop frequencies and detected frequencies are given in the hop order :

$$f_{base} = [900, 2900, 1900, 900]$$

$$f_{detected} = [1171.875, 3203.125, 1718.75, 1015.625]$$

Demodulation occurs:

$$1) f_{detected} - f_{base} = -2\Delta f, m_k = -2 \Rightarrow \text{binary data} = 00$$

$$2) f_{detected} - f_{base} = -\Delta f, m_k = -1 \Rightarrow \text{binary data} = 01$$

$$3) f_{detected} - f_{base} = +\Delta f, m_k = +1 \Rightarrow \text{binary data} = 10$$

$$4) f_{detected} - f_{base} = +2\Delta f, m_k = +2 \Rightarrow \text{binary data} = 11$$

$$\Rightarrow 1) 1171.875 - 900 = 271.875 \simeq +2\Delta f = 300 \Rightarrow \text{binary data} = 11$$

$$2) 3203.125 - 2900 = 303.125 \simeq +2\Delta f = 300 \Rightarrow \text{binary data} = 11$$

$$3) 1718.75 - 1900 = -181.25 \simeq -\Delta f = -150 \Rightarrow \text{binary data} = 01$$

$$4) 1015.625 - 900 = 115.625 \simeq +\Delta f = 150 \Rightarrow \text{binary data} = 10$$

● 8-FSK

Eight possible frequencies represent binary values 000, 001, 010, 011, 100, 101, 110, and 111:

$$1) f_{detected} - f_{base} = -4\Delta f, m_k = -4 \Rightarrow \text{binary data} = 000$$

$$2) f_{detected} - f_{base} = -3\Delta f, m_k = -3 \Rightarrow \text{binary data} = 001$$

$$3) f_{detected} - f_{base} = -2\Delta f, m_k = -2 \Rightarrow \text{binary data} = 010$$

$$4) f_{detected} - f_{base} = -\Delta f, m_k = -1 \Rightarrow \text{binary data} = 011$$

- 5) $f_{detected} - f_{base} = + \Delta f, m_k = + 1 \Rightarrow \text{binary data} = 100$
- 6) $f_{detected} - f_{base} = + 2\Delta f, m_k = + 2 \Rightarrow \text{binary data} = 101$
- 7) $f_{detected} - f_{base} = + 3\Delta f, m_k = + 3 \Rightarrow \text{binary data} = 110$
- 8) $f_{detected} - f_{base} = + 4\Delta f, m_k = + 4 \Rightarrow \text{binary data} = 111$

The code implementation of 8-FSK demodulation is as follows in Figure 10:

```
elseif modulationOrder == 8 % 8-FSK
% Map frequency difference to 3-bit binary (000, 001, ..., 111)
if abs(detectedFreqs - (hopFreq - 4 * deltaF)) < tolerance
    disp('Matched 000');
    binaryDats = [binaryDats, 0, 0, 0];
elseif abs(detectedFreqs - (hopFreq - 3 * deltaF)) < tolerance
    disp('Matched 001');
    binaryDats = [binaryDats, 0, 0, 1];
elseif abs(detectedFreqs - (hopFreq - 2 * deltaF)) < tolerance
    disp('Matched 010');
    binaryDats = [binaryDats, 0, 1, 0];
elseif abs(detectedFreqs - (hopFreq - deltaF)) < tolerance
    disp('Matched 011');
    binaryDats = [binaryDats, 0, 1, 1];
elseif abs(detectedFreqs - (hopFreq + deltaF)) < tolerance
    disp('Matched 100');
    binaryDats = [binaryDats, 1, 0, 0];
elseif abs(detectedFreqs - (hopFreq + 2 * deltaF)) < tolerance
    disp('Matched 101');
    binaryDats = [binaryDats, 1, 0, 1];
elseif abs(detectedFreqs - (hopFreq + 3 * deltaF)) < tolerance
    disp('Matched 110');
    binaryDats = [binaryDats, 1, 1, 0];
elseif abs(detectedFreqs - (hopFreq + 4 * deltaF)) < tolerance
    disp('Matched 111');
    binaryDats = [binaryDats, 1, 1, 1];
else
    disp('Frequency does not match any expected value.');
```

Figure 10: The 8-FSK Demodulation.

5. Results: Decoding and Validation

5.1 Category 1 Signals

The spectrogram of the transmitter and receiver signals showed accurate synchronization and recovery of hop frequencies. The transmitter GUI is set for 2 -FSK (Category 1), and the parameters and sampling rate are set accordingly, as shown in Figure 11:

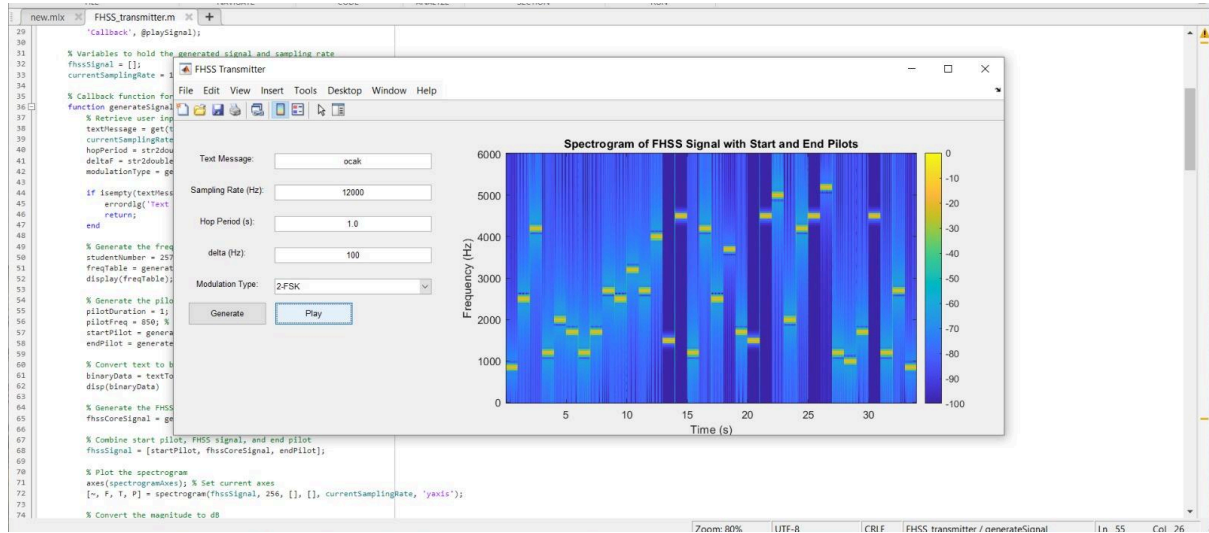


Figure 11: The Transmitter GUI set to Category 1.

The transmitted word is determined as 'ocak'. It is transmitted by $8 \times 4 = 32$ binaries (32 hop frequencies). The receiver GUI set for 2 -FSK (Category 1) is shown in Figure 12:

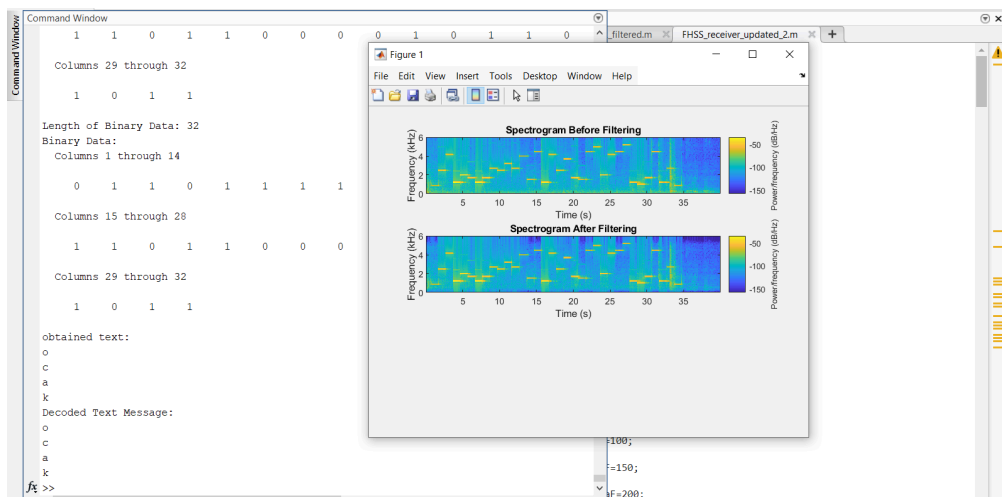


Figure 12: The Receiver GUI set to Category 1.

The output can be seen on the left-hand side of the screen. The transmission is successful.

5.2 Category 2 Signals

The spectrogram of the transmitter and receiver signals showed accurate synchronization and recovery of hop frequencies. The transmitter GUI is set for 4 -FSK (Category 2), and the parameters and sampling rate are set accordingly, as shown in Figure 13:

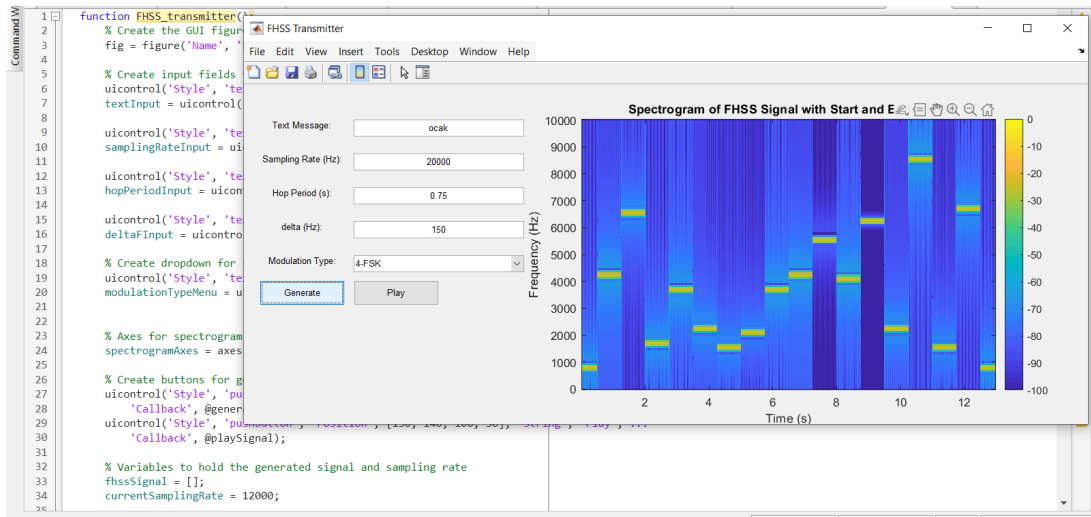


Figure 13: The Transmitter GUI set to Category 2.

The transmitted word is determined as 'ocak'. It is transmitted by $8 \times 4 = 32$ binaries (16 hop frequencies). The receiver GUI set for 4 -FSK (Category 2) is shown in Figure 14:

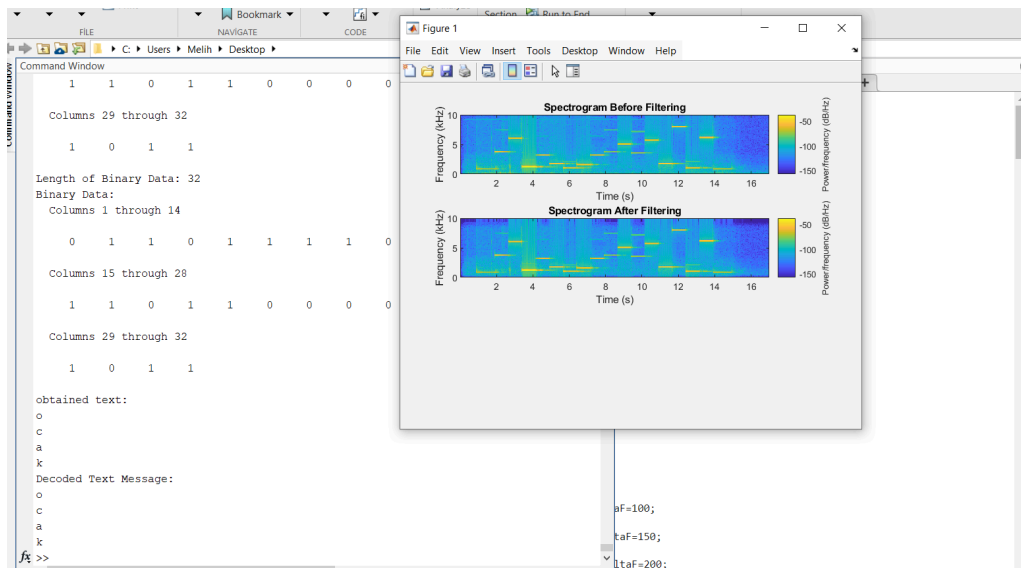


Figure 14: The Receiver GUI set to Category 2.

The output can be seen on the left-hand side of the screen. The transmission is successful.

5.3 Category 3 Signals

5.3.1 The Problem with 8-FSK

In the 8-FSK, each hop corresponds to a fixed time interval, and the transmitted signal is divided into frames. To ensure synchronization between the transmitter and receiver, the

length of each frame must be consistent. However, the data does not perfectly fit the required frame length when the total number of bits is not a multiple of 3 in 8-FSK. To handle this, in the transmitter zero padding is added to fill the remaining space. This ensures that every frame has the same duration, avoiding errors caused by mismatched frame lengths.

The implemented code for zero padding is depicted in Figure 15:

```
bitsPerSymbol = log2(modulationType * 2); % log2(2), log2(4), log2(8)

if mod(length(data), bitsPerSymbol) ~= 0
    paddingBits = bitsPerSymbol - mod(length(data), bitsPerSymbol);
    data = [data, zeros(1, paddingBits)]; % Add zero-padding
end
```

Figure 15: The Zero Padding in 8-FSK.

- An Example:

The Binary Data to be Transmitted: [01010010]

Length of the Binary Data: 8

To modulate this in 8-FSK we need to divide it to the length of 3 bits. (010+100+10?)

But since 8 is not a multiple of 3, it is not possible. In order to succeed in this operation, it is logical to add a zero at the end of the Binary Data and do the transmission that way. So, the new Binary Data to be Transmitted is: [01010010**0**] which can be divided into 3 as [010+100+100] and transmitted in this way.

Hence, padding the data with zeros ensures that the number of bits is always divisible by 3, enabling seamless modulation into 8-FSK. Without padding, the final group of bits might lead to ambiguous or incorrect frequency mappings.

5.3.2 8-FSK Implementation

The transmitter GUI is set for 8 -FSK (Category 3), and the parameters and sampling rate are set accordingly, as shown in Figure 16:

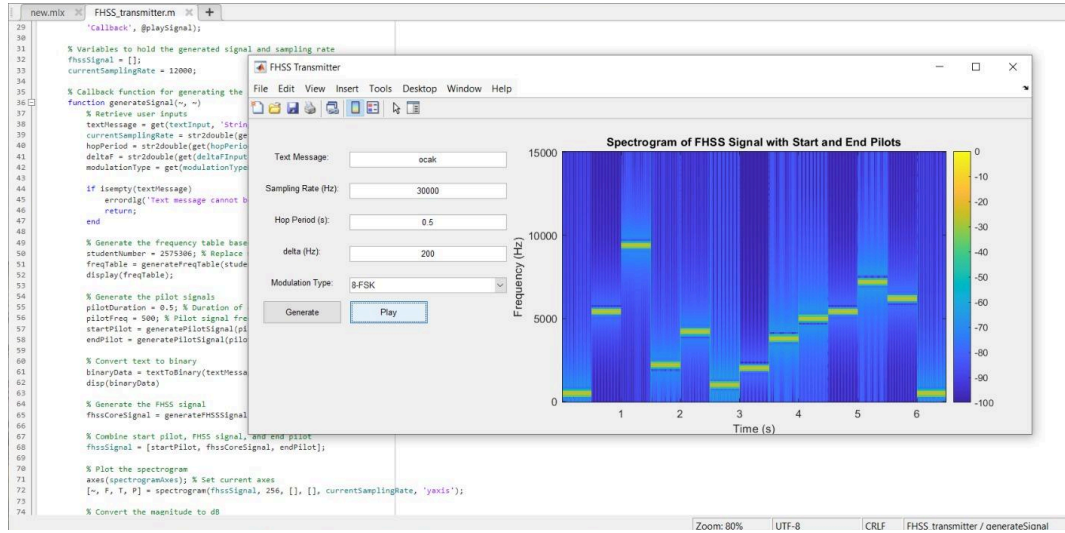


Figure 16: The Transmitter GUI set to Category 3.

The transmitted word is determined as 'ocak'. It is transmitted by $32+1(\text{zero padding}) = 33$ binaries (11 hop frequencies).

Since the transmitted binaries are zero-padded, the extra zeros must be truncated as in the code shown in Figure 17:

```
nearestMultiple = floor(length(binaryDatas) / 8) * 8;  
if nearestMultiple > 0  
    binaryDatas = binaryDatas(1:nearestMultiple);  
else  
    binaryDatas = []; % If no multiple of 8, set to empty  
end
```

Figure 17: The Truncation Process.

After obtaining the truncated binary data, the demodulation starts according to the rules of Category 3.

The receiver GUI set for 8 - FSK (Category 3) is shown in Figure 18:

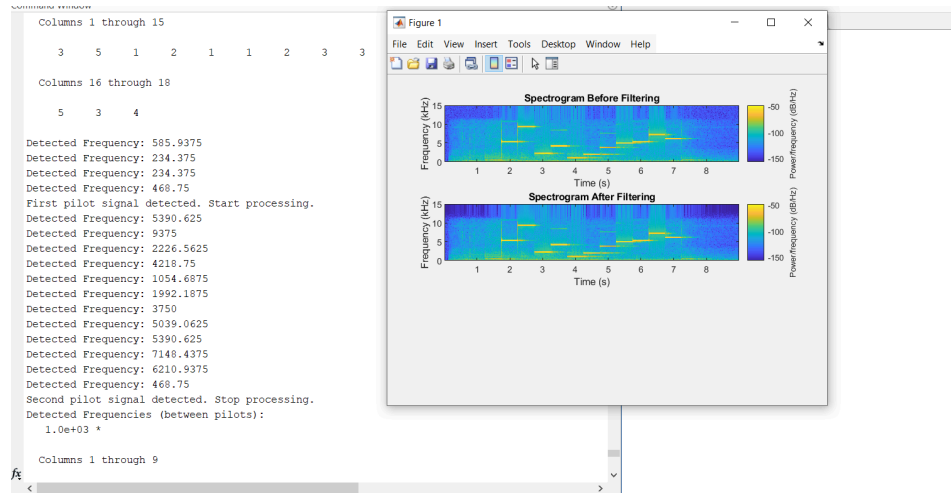


Figure 18: The Receiver GUI set to Category 3.

The code also shows the detected-base hop frequency pairs, matched binary data for all of them, and the received text as depicted in Figure 19a, 19b, 19c:

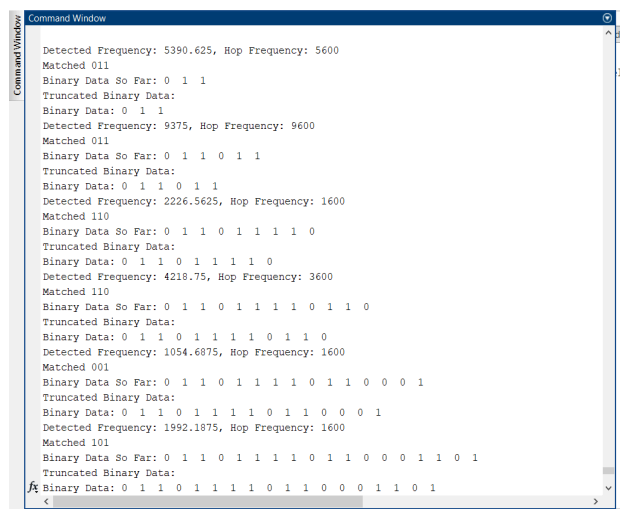


Figure 19a: The 8-FSK Demodulation.

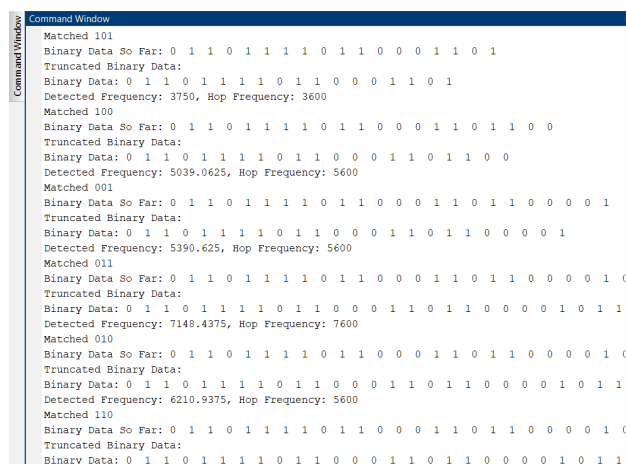


Figure 19a: The 8-FSK Demodulation.



Figure 19c: The 8-FSK Result.

The output can be seen on the left-hand side of the screen in Figure 19c. The transmission is completed successfully.

6. Discussion and Challenges

Implementing a robust receiver for 8-FSK transmission, particularly for Category 2 and Category 3 signals, posed several challenges. Synchronizing the receiver with the transmitter was critical, especially in the presence of noise and varying environmental conditions. The use of pilot signals for synchronization proved effective, but detecting these signals in high-noise environments required precise thresholding and frequency analysis. Another challenge was maintaining accurate decoding despite fast hopping rates in Category 3 signals, which demanded efficient time-frequency analysis and real-time processing. Noise reduction techniques, such as bandpass filtering and spectral thresholding, significantly improved signal clarity but introduced limitations, such as potential attenuation of weak signal components or reliance on optimal filter parameters. Additionally, ensuring seamless demodulation of the received signal into binary data required precise alignment of detected frequencies with expected modulation levels, particularly for 8-FSK, where small errors in frequency detection could lead to incorrect decoding. These challenges highlight the complexity of implementing reliable FHSS systems and underscore the importance of robust algorithms and adaptive techniques to handle dynamic signal conditions.

7. Conclusion

The implementation of the receiver for Frequency Hopping Spread Spectrum (FHSS) signals, particularly for Category 2 and Category 3, demonstrates the feasibility of decoding rapid and complex signals in dynamic environments. Key techniques such as pilot signal synchronization, bandpass filtering, and precise time-frequency analysis were employed to accurately track the hopping frequencies, extract the transmitted signal, and reconstruct the original message. The integration of noise reduction methods, including Butterworth bandpass filtering and spectral thresholding, significantly improved the signal-to-noise ratio (SNR), ensuring robust operation even under noisy conditions. Advanced demodulation

techniques allowed for precise mapping of detected frequencies to binary data, accommodating varying modulation orders, including the demanding 8-FSK scheme. This implementation addressed critical challenges, such as achieving precise synchronization, handling high hopping rates, and balancing noise reduction without compromising signal fidelity. Pilot signal synchronization was pivotal in aligning the receiver with the transmitter's hopping sequence, while efficient segmentation of the signal into hop periods and real-time spectrogram analysis facilitated accurate frequency detection. Noise reduction techniques, such as adaptive filtering and spectral thresholding, were carefully designed to preserve signal integrity while suppressing noise. Frequency deviation mapping further ensured reliable decoding by systematically associating detected frequencies with binary data. Innovative approaches contributed to the receiver's success. The use of zero padding to align data frames ensured seamless processing across varying signal lengths, while adaptive filtering and thresholding enhanced the receiver's resilience to changing noise environments. The spectrogram-based frequency analysis offered a reliable mechanism for extracting hopping frequencies, even in the presence of overlapping spectral components. These advancements collectively enabled the receiver to operate effectively in challenging conditions, meeting the requirements of both Category 2 and Category 3 signals.

This work provides a comprehensive framework for implementing FHSS receivers, with practical applications in secure and interference-resistant communication systems. Future directions for enhancing the system could include incorporating error correction coding to mitigate data loss, dynamic thresholding to optimize performance in variable noise conditions, and machine learning-based adaptive filtering for real-time parameter tuning. The successful decoding of Category 2 and Category 3 signals demonstrates the practicality of the approach, contributing valuable insights to the field of digital and secure communication.

```

function FHSS_transmitter();

    fig = figure('Name', 'FHSS Transmitter', 'NumberTitle', 'off',
'Position', [100, 100, 600, 400]);

    uicontrol('Style', 'text', 'Position', [20, 340, 100, 20], 'String',
'Text Message:');
    textInput = uicontrol('Style', 'edit', 'Position', [130, 340, 200, 20]);

    uicontrol('Style', 'text', 'Position', [20, 300, 100, 20], 'String',
'Sampling Rate (Hz):');
    samplingRateInput = uicontrol('Style', 'edit', 'Position', [130, 300,
200, 20], 'String', '12000');

    uicontrol('Style', 'text', 'Position', [20, 260, 100, 20], 'String',
'Hop Period (s):');
    hopPeriodInput = uicontrol('Style', 'edit', 'Position', [130, 260, 200,
20], 'String', '1.0');

    uicontrol('Style', 'text', 'Position', [20, 220, 100, 20], 'String',
'delta (Hz):');
    deltaFInput = uicontrol('Style', 'edit', 'Position', [130, 220, 200,
20], 'String', '100');

    uicontrol('Style', 'text', 'Position', [20, 180, 100, 20], 'String',
'Modulation Type:');
    modulationTypeMenu = uicontrol('Style', 'popupmenu', 'Position', [130,
180, 200, 20], ...
                                'String', {'2-FSK', '4-FSK', 'null', '8-
FSK'});

    spectrogramAxes = axes('Parent', fig, 'Position', [0.4, 0.1, 0.55, 0.8]);

    uicontrol('Style', 'pushbutton', 'Position', [20, 140, 100, 30],
'String', 'Generate', ...
            'Callback', @generateSignal);
    uicontrol('Style', 'pushbutton', 'Position', [130, 140, 100, 30],
'String', 'Play', ...
            'Callback', @playSignal);

    fhssSignal = [];
    currentSamplingRate = 12000;

    function generateSignal(~, ~)
        % Retrieve user inputs
        textMessage = get(textInput, 'String');
        currentSamplingRate = str2double(get(samplingRateInput, 'String'));
        hopPeriod = str2double(get(hopPeriodInput, 'String'));
        deltaF = str2double(get(deltaFInput, 'String'));
        modulationType = get(modulationTypeMenu, 'Value');
        if isempty(textMessage)
            error('Text message cannot be empty.', 'Input Error');

```

```

        return;
    end

    studentNumber = 2575306;
    freqTable = generateFreqTable(studentNumber, modulationType);
    display(freqTable);

    pilotDuration = 0.5;
    pilotFreq = 850;
    startPilot = generatePilotSignal(pilotFreq, pilotDuration,
currentSamplingRate);
    endPilot = generatePilotSignal(pilotFreq, pilotDuration,
currentSamplingRate);

    binaryData = textToBinary(textMessage);
    disp(binaryData)

    fhssCoreSignal = generateFHSSSignal(binaryData, freqTable, deltaF,
hopPeriod, currentSamplingRate, modulationType);

    fhssSignal = [startPilot, fhssCoreSignal, endPilot];

    axes(spectrogramAxes); % Set current axes
    [~, F, T, P] = spectrogram(fhssSignal, 256, [], [],
currentSamplingRate, 'yaxis');

    P_dB = 10 * log10(abs(P));

    imagesc(T, F, P_dB);
    axis xy;
    xlabel('Time (s)');
    ylabel('Frequency (Hz)');
    title('Spectrogram of FHSS Signal with Start and End Pilots');
    colorbar;
    clim([-100 0]);
end

function playSignal(~, ~)
    if isempty(fhssSignal)
        errordlg('Generate a signal first.', 'Playback Error');
        return;
    end
    sound(fhssSignal, currentSamplingRate);
end

function pilotSignal = generatePilotSignal(freq, duration, samplingRate)
    t = 0:1/samplingRate:duration - 1/samplingRate;
    pilotSignal = sin(2 * pi * freq * t);
end

function freqTable = generateFreqTable(studentNumber, modulationType)
    digits = num2str(studentNumber) - '0';
    N5 = digits(5);
    N6 = digits(6);

```

```

    N7 = digits(7);

    if modulationType == 1
        b = 1;
        if mod(N6, 2) ~= 0
            b = -1;
        end
        S = 100 * mod(N7, 5);
        baseFreqs = [1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500,
5000];
        freqTable = baseFreqs + b * S;

    elseif modulationType == 2
        b = 1;
        if mod(N5, 2) ~= 0
            b = -1;
        end
        S = 100 * N7;
        baseFreqs = [1500, 2500, 3500, 4500, 5500, 6500, 7500, 8500];
        freqTable = baseFreqs + b * S;

    elseif modulationType == 4
        S = 100 * N7;
        baseFreqs = [1000, 3000, 5000, 7000, 9000, 11000];
        freqTable = baseFreqs + S;
    end
end

function binaryData = textToBinary(text)
    binaryData = reshape(dec2bin(text, 8)' - '0', 1, []);
end

function signal = generateFHSSSignal(data, freqTable, deltaF, hopPeriod,
samplingRate, modulationType)
    t = 0:1/samplingRate:hopPeriod - 1/samplingRate;
    signal = [];

    bitsPerSymbol = log2(modulationType * 2); % log2(2), log2(4), log2(8)

    if mod(length(data), bitsPerSymbol) ~= 0
        paddingBits = bitsPerSymbol - mod(length(data), bitsPerSymbol);
        data = [data, zeros(1, paddingBits)]; % adding zero-padding
    end

    numSymbols = length(data) / bitsPerSymbol;

    symbols = zeros(1, numSymbols);

    for i = 1:numSymbols
        startIdx = (i-1) * bitsPerSymbol + 1;
        endIdx = i * bitsPerSymbol;
        symbolBits = data(startIdx:endIdx);
        display( symbolBits)
    end
end

```

```

    if modulationType == 1
        symbols(i) = 2 * symbolBits - 1; % Map 0 to -1, 1 to +1

    elseif modulationType == 2
        if isequal(symbolBits, [0 0])
            symbols(i) = -2;
        elseif isequal(symbolBits, [0 1])
            symbols(i) = -1;
        elseif isequal(symbolBits, [1 0])
            symbols(i) = +1;
        elseif isequal(symbolBits, [1 1])
            symbols(i) = +2;
        end

    elseif modulationType == 4
        if isequal(symbolBits, [0 0 0])
            symbols(i) = -4;
        elseif isequal(symbolBits, [0 0 1])
            symbols(i) = -3;
        elseif isequal(symbolBits, [0 1 0])
            symbols(i) = -2;
        elseif isequal(symbolBits, [0 1 1])
            symbols(i) = -1;
        elseif isequal(symbolBits, [1 0 0])
            symbols(i) = +1;
        elseif isequal(symbolBits, [1 0 1])
            symbols(i) = +2;
        elseif isequal(symbolBits, [1 1 0])
            symbols(i) = +3;
        elseif isequal(symbolBits, [1 1 1])
            symbols(i) = +4;
        end
    end
end

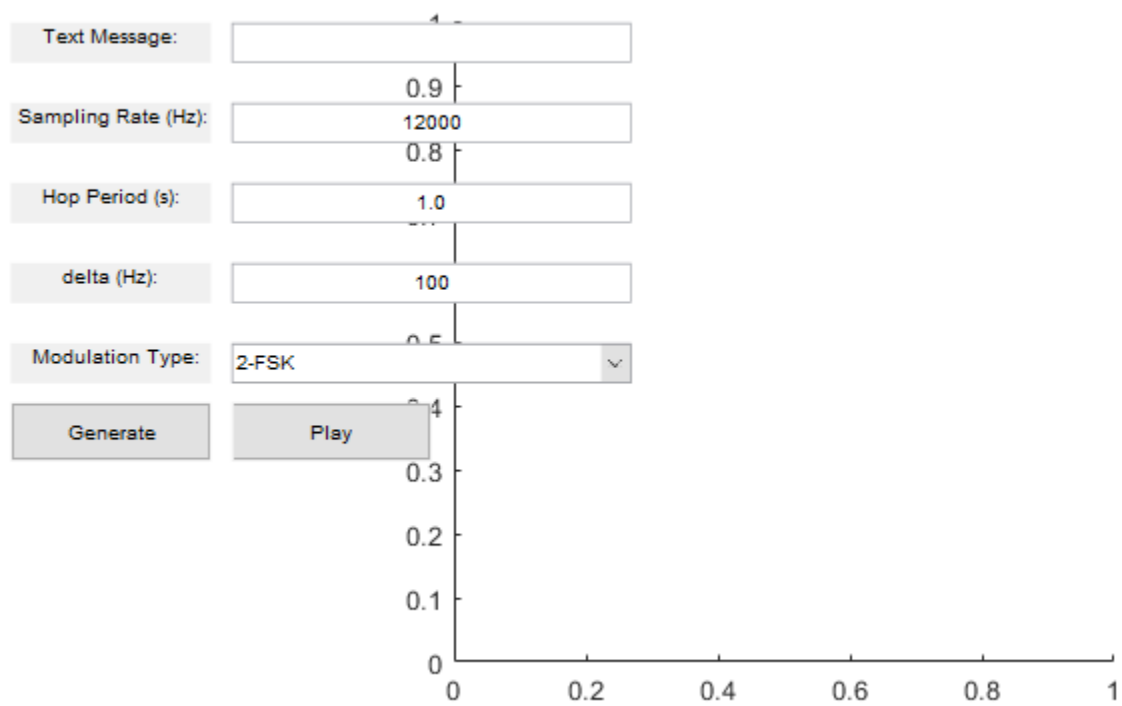
rng(1);
hopOrder = randi(length(freqTable), 1, numSymbols);
display(hopOrder)

for i = 1:numSymbols
    hopFreq = freqTable(hopOrder(i)); % frequency for current hop

    modFreq = hopFreq + symbols(i) * deltaF;
    display(modFreq)
    signal = [signal, sin(2 * pi * modFreq * t)]; % appends sine wave
end
end

end

```



Published with MATLAB® R2024b

```

function FHSS_receiver()

    fig = uifigure('Name', 'FHSS Receiver', 'Position', [100, 100, 500,
600]);

    uilabel(fig, 'Position', [20, 550, 120, 22], 'Text', 'Sampling Rate
(Hz):');
    samplingRateField = ueditfield(fig, 'numeric', 'Position', [150, 550,
300, 22]);

    uilabel(fig, 'Position', [20, 510, 120, 22], 'Text', 'Recording Duration
(s):');
    durationField = ueditfield(fig, 'numeric', 'Position', [150, 510, 300,
22]);

    uilabel(fig, 'Position', [20, 470, 120, 22], 'Text', 'FSK Type:');
    fskTypeDropdown = uidropdown(fig, 'Position', [150, 470, 300, 22], ...
    'Items', {'2-FSK', '4-FSK', '8-FSK'}, 'Value', '2-FSK');

    uibutton(fig, 'Position', [150, 400, 200, 30], 'Text', 'Record and
Decode', ...
    'ButtonPushedFcn', @(btn, event) recordAndDecode());

    function recordAndDecode()

        samplingRate = samplingRateField.Value;
        duration = durationField.Value;
        fskType = fskTypeDropdown.Value;

        switch fskType
            case '2-FSK'
                lowFreq = 500; highFreq = 5000; hopPeriod = 1.0;
modulationOrder = 2; deltaF=100;
            case '4-FSK'
                lowFreq = 500; highFreq = 8500; hopPeriod = 0.75;
modulationOrder = 4; deltaF=150;
            case '8-FSK'
                lowFreq = 200; highFreq = 11000; hopPeriod = 0.50;
modulationOrder = 8; deltaF=200;
            end

        recorder = audiorecorder(samplingRate, 16, 1);
        disp('Recording...');
        recordblocking(recorder, duration);
        receivedSignal = getaudiodata(recorder);
        disp('Recording completed.');
```

```

        subplot(3, 1, 1);
        spectrogram(receivedSignal, 256, 200, 256, samplingRate, 'yaxis');
        title('Spectrogram Before Filtering');
```

```

        filteredSignal = applyBandpassFilter(receivedSignal, samplingRate,
lowFreq, highFreq);

        subplot(3, 1, 2);
        spectrogram(filteredSignal, 256, 200, 256, samplingRate, 'yaxis');
        title('Spectrogram After Filtering');

        binaryDatas = decodeFHSS(filteredSignal, samplingRate,
modulationOrder, hopPeriod, deltaF);
        disp('Binary Data:');
        disp(binaryDatas);

        decodedText = binarytoText(binaryDatas);
        disp('Decoded Text Message:');
        disp(decodedText);
    end

    function filteredSignal = applyBandpassFilter(receivedSignal,
samplingRate, lowFreq, highFreq)

        nyquist = samplingRate / 2;
        [b, a] = butter(4, [lowFreq, highFreq] / nyquist, 'bandpass');
        filteredSignal = filter(b, a, receivedSignal);
    end

    function binaryDatas = decodeFHSS(filteredSignal, samplingRate,
modulationOrder, hopPeriod, deltaF)

        studentNumber = 2575306;
        freqTable = generateFreqTable(studentNumber, modulationOrder);

        hopSamples = round(hopPeriod * samplingRate);
        numHops = floor(length(filteredSignal) / hopSamples);

        pilotFrequency = 850;
        tolerance = 60;

        rng(1);
        hopOrder = randi(length(freqTable), 1, numHops);
        display(hopOrder);

        startIdx = []; % start index after first pilot
        endIdx = []; % end index after second pilot
        detectedFrequencies = [];
        binaryDatas = [];

        for i = 1:numHops
            % extract segment for current hop
            startSampleIdx = (i-1) * hopSamples + 1;
            endSampleIdx = min(startSampleIdx + hopSamples - 1,
length(filteredSignal));
            hopSegment = filteredSignal(startSampleIdx:endSampleIdx);

```

```

[S, F, ~] = spectrogram(hopSegment, 256, 200, 256, samplingRate);
[~, maxIdx] = max(abs(S), [], 1);
detectedFreq = F(maxIdx(1)); % extract detected frequency
disp(['Detected Frequency: ', num2str(detectedFreq)]);

if abs(detectedFreq - pilotFrequency) < tolerance

    if isempty(startIdx)
        startIdx = startSampleIdx;
        disp('First pilot signal detected. Start processing.');
```

```

    elseif isempty(endIdx)
        endIdx = startSampleIdx;
        disp('Second pilot signal detected. Stop processing.');
```

```

        break;
    end
    continue;
end

% storing detected frequencies between pilot signals
if ~isempty(startIdx) && isempty(endIdx)
    detectedFrequencies = [detectedFrequencies, detectedFreq];
end
end

disp('Detected Frequencies (between pilots):');
disp(detectedFrequencies);

numDetected = length(detectedFrequencies);
numHopFreqs = length(hopOrder);

for j = 1:min(numDetected, numHopFreqs)

    hopFreq = freqTable(hopOrder(j));
    detectedFreqs = detectedFrequencies(j);

    disp(['Detected Frequency: ', num2str(detectedFreqs), ', Hop
Frequency: ', num2str(hopFreq)]);

    if modulationOrder == 2

        if abs(detectedFreqs - (hopFreq - deltaF)) < abs(detectedFreqs -
(hopFreq + deltaF))
            binaryDatas = [binaryDatas, 0];
        else
            binaryDatas = [binaryDatas, 1];
        end

    elseif modulationOrder == 4

        if abs(detectedFreqs - (hopFreq - 2 * deltaF)) < tolerance

```

```

        disp('Matched 00');
        binaryDats = [binaryDats, 0, 0];
    elseif abs(detectedFreqs - (hopFreq - deltaF)) < tolerance
        disp('Matched 01');
        binaryDats = [binaryDats, 0, 1];
    elseif abs(detectedFreqs - (hopFreq + deltaF)) < tolerance
        disp('Matched 10');
        binaryDats = [binaryDats, 1, 0];
    elseif abs(detectedFreqs - (hopFreq + 2 * deltaF)) < tolerance
        disp('Matched 11');
        binaryDats = [binaryDats, 1, 1];
    end

elseif modulationOrder == 8

    if abs(detectedFreqs - (hopFreq - 4 * deltaF)) < tolerance
        disp('Matched 000');
        binaryDats = [binaryDats, 0, 0, 0];
    elseif abs(detectedFreqs - (hopFreq - 3 * deltaF)) < tolerance
        disp('Matched 001');
        binaryDats = [binaryDats, 0, 0, 1];
    elseif abs(detectedFreqs - (hopFreq - 2 * deltaF)) < tolerance
        disp('Matched 010');
        binaryDats = [binaryDats, 0, 1, 0];
    elseif abs(detectedFreqs - (hopFreq - deltaF)) < tolerance
        disp('Matched 011');
        binaryDats = [binaryDats, 0, 1, 1];
    elseif abs(detectedFreqs - (hopFreq + deltaF)) < tolerance
        disp('Matched 100');
        binaryDats = [binaryDats, 1, 0, 0];
    elseif abs(detectedFreqs - (hopFreq + 2 * deltaF)) < tolerance
        disp('Matched 101');
        binaryDats = [binaryDats, 1, 0, 1];
    elseif abs(detectedFreqs - (hopFreq + 3 * deltaF)) < tolerance
        disp('Matched 110');
        binaryDats = [binaryDats, 1, 1, 0];
    elseif abs(detectedFreqs - (hopFreq + 4 * deltaF)) < tolerance
        disp('Matched 111');
        binaryDats = [binaryDats, 1, 1, 1];
    else
        disp('Frequency does not match any expected value.');
```

end

```

disp(['Binary Data So Far: ', num2str(binaryDats)]);

binaryDats = binaryDats(:)';

disp('Truncated Binary Data:');
disp(['Binary Data: ', num2str(binaryDats)]);

end

end
```

```

nearestMultiple = floor(length(binaryDatas) / 8) * 8;
if nearestMultiple > 0
    binaryDatas = binaryDatas(1:nearestMultiple);
else
    binaryDatas = [];
end

disp('Decoded Binary Data:');
display(binaryDatas)
binaryDatas = binaryDatas(:).'; % Convert to row vector
disp(['Length of Binary Data: ', num2str(length(binaryDatas))]);
end

function freqTable = generateFreqTable(studentNumber, modulationOrder)

    digits = num2str(studentNumber) - '0';
    N5 = digits(5);
    N6 = digits(6);
    N7 = digits(7);

    if modulationOrder == 2
        b = 1; if mod(N6, 2) ~= 0, b = -1; end
        S = 100 * mod(N7, 5);
        baseFreqs = [1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500,
5000];
        freqTable = baseFreqs + b * S;
    elseif modulationOrder == 4
        b = 1; if mod(N5, 2) ~= 0, b = -1; end
        S = 100 * N7;
        baseFreqs = [1500, 2500, 3500, 4500, 5500, 6500, 7500, 8500];
        freqTable = baseFreqs + b * S;
    elseif modulationOrder == 8
        S = 100 * N7;
        freqTable = [1000, 3000, 5000, 7000, 9000, 11000] + S;
    else
        error('Unsupported modulation order. Use 2, 4, or 8.');
```

end

```

end

function text = binarytoText(binaryDatas)

    binaryMatrix = reshape(binaryDatas, 8, []).';
    text = char(bin2dec(num2str(binaryMatrix)));
    disp('obtained text:');
    disp(text);
end

```

Sampling Rate (Hz):

Recording Duration ...

FSK Type:

Published with MATLAB® R2024b