

PROJECT REPORT
DIGITAL SIGNAL PROCESSING (EC 301)



AUDIO STEGANOGRAPHY USING PHASE CODING

SUBMITTED TO:

Dr. Ravi Kumar Jatoth

Professor, Electronics and Communication Engineering

National Institute of Technology, Warangal

SUBMITTED BY:

Potluri Satya Sri Varsha -21ECB0B44

Sarvepalli Mahathi - 21ECB0B53

Vattam Sai Sharanya- 21ECB0B63

B.Tech in Electronics and Communication Engineering

National Institute of Technology, Warangal

ABSTRACT

Steganography, the art of covert communication, plays a crucial role in securing information by concealing it within seemingly innocuous carriers. This project explores audio steganography, focusing on the Phase Encoding Technique to embed secret messages within audio signals. Leveraging the principles of Hermitian symmetry and phase manipulation, the method ensures seamless integration of hidden information without perceptible distortions. The implementation involves dividing the audio signal into segments, encoding binary data in the phase spectrum, and maintaining the integrity of the original signal through inverse Fourier transformations. The proposed approach achieves a balance between robust concealment and minimal impact on audio quality. Experimental results demonstrate the effectiveness of the technique in securely embedding and extracting hidden messages, contributing to the realm of covert communication in audio domains.

1. INTRODUCTION

Audio steganography involves concealing a secret message within an audio file, known as the carrier audio, by manipulating the audio signal. This technique takes advantage of the limitations of the human auditory system, exploiting the challenge in distinguishing between low and loud sounds, where a louder sound can potentially overshadow a softer one.

In the realm of steganography, audio exhibits certain advantages when compared to image and video formats. Sound recordings are typically larger than image files, providing more space to accommodate a greater amount of encoded messages. On the other hand, video formats, while comparatively larger in size, suffer from reduced practicality as they lack the necessary algorithms to effectively manage and process their extensive content.

Windows Audio-Visual (WAV) is one of the most widely used audio file formats that can be used to conceal information. The storage environment and the digital representation of the intended signal are the two primary areas of change in a WAV file for data embedding.

The encoding techniques that are commonly employed to conceal data in this format include phase coding, spread spectrum, low-bit-encoding, and echo data hiding.

In this project we used the phase coding technique for embedding data in an audio signal.

2. METHODOLOGY

The phase coding method operates by replacing the phase of an initial audio segment with a reference phase that corresponds to the data. The phase of subsequent segments is adjusted to maintain the relative phase between segments. In terms of the signal-to-perceived noise ratio, phase coding is among the most effective coding techniques.

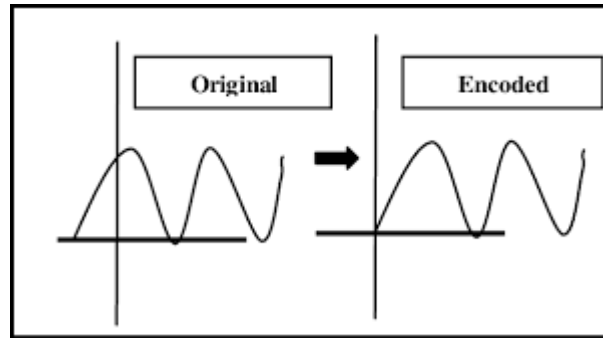
2.1 EMBEDDING

The embedding process is done by inserting the data bits into the first segment of the audio as a cover. The steps for embedding are as follows:

1. The audio signal is broken down into segments with a signal length that is sufficient to encode the secret message.
2. The Fast Fourier Transform (FFT) is applied to each segment to create a matrix of the phases and magnitudes.
3. The phase difference between adjacent matrices is calculated and stored.
4. The message is converted into its binary form (ascii values).
5. The secret message is inserted in the phase vector of the first signal segment as follows:

$$\begin{aligned}\text{Phase_new} &= \pi/2 \text{ if message bit} = 0 \\ &-\pi/2 \text{ if message bit} = 1\end{aligned}$$

6. A new phase matrix is created using the new phase of the first segment and the original phase differences.
7. Audio signal is generated using new phase matrix and original magnitude matrix by applying IFFT.
8. Concatenated the audio segment to the original audio file.



2.2 EXTRACTING:

The process of extraction is used to get the data that is previously incorporated into the audio. The steps for extraction are as follows:

1. Obtained the starting segment of the audio file.
2. Applied FFT on it and obtained phase values.
3. Phase values are observed and data bits are extracted as follows:

Data = 0 if phase > 0

1 if phase > 1

4. The obtained binary data is converted into characters using ascii to binary conversion.

2.3 FAST FOURIER TRANSFORM (FFT)

It is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of a sequence. It breaks down the DFT computation into smaller subproblems, significantly reducing the number of operations.

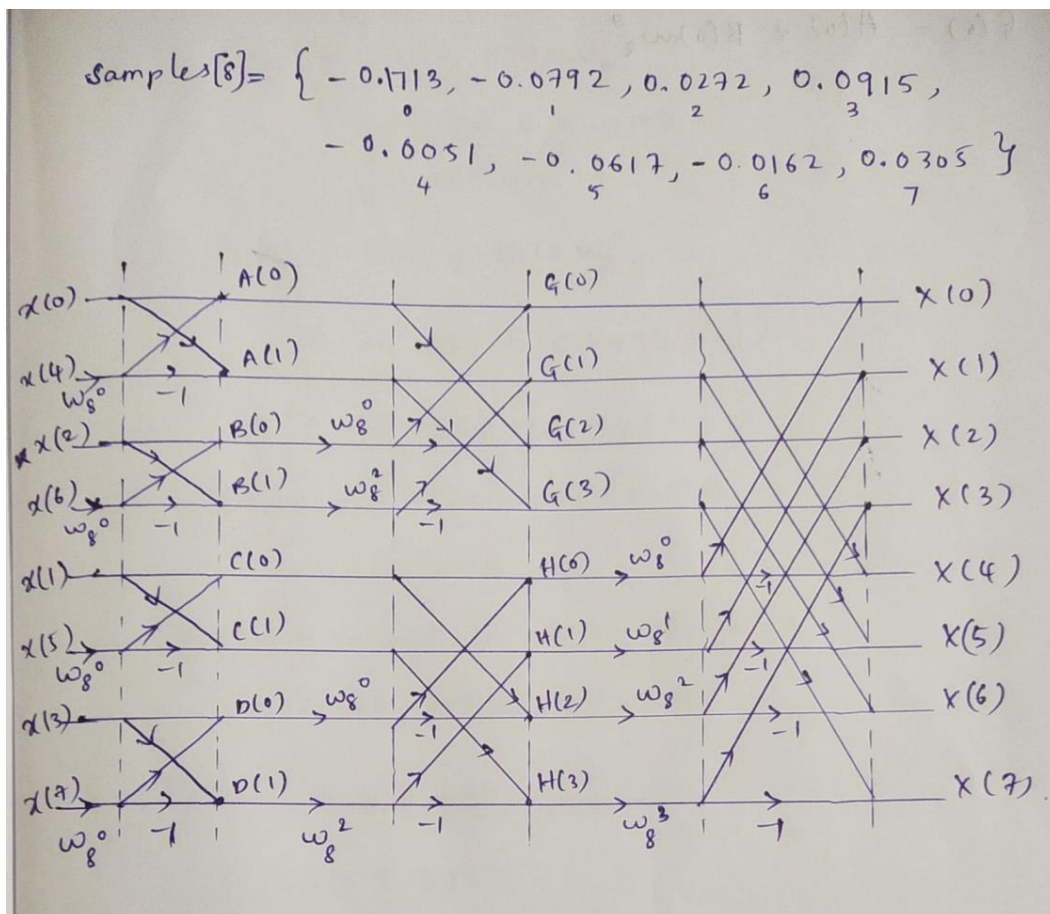
The direct computation of the N-point DFT involves pairwise multiplications and additions for each element in the sequence with the total number of operations (complex multiplications and additions) is proportional to N^2 . The FFT algorithm, on the other hand, significantly reduces the number of operations compared to the direct DFT computation. For a Cooley-Tukey Radix-2 FFT, the number of operations is approximately

proportional to $N \log N$. This represents a substantial reduction in computational complexity, especially for large N .

Radix-2 FFT is a variant of the Fast Fourier Transform (FFT) designed for sequences with sizes that are powers of 2. It employs a divide-and-conquer strategy, recursively breaking down the Discrete Fourier Transform (DFT) into smaller subproblems. The algorithm's efficiency is particularly notable for power-of-two input sizes, utilizing butterfly operations to combine DFT values.

FFT of first 8 samples:

First 8 samples of audio signal are obtained and radix 2 FFT algorithm is used to compute discrete fourier transform.



$$A(0) = x(0) + x(4) \times \omega_8^0 = -0.1713 + (-0.0051) \times 1$$

$$= -0.1764$$

$$A(1) = -0.1713 + 0.0051 = -0.1662$$

$$B(0) = x(2) + x(6) \times \omega_8^0 = 0.0272 + (-0.0162)$$

$$= 0.0110$$

$$B(1) = x(2) - x(6) \times \omega_8^0 = 0.0272 + 0.0162$$

$$= 0.0434$$

$$C(0) = -0.0792 - 0.0617 = -0.1409$$

$$C(1) = -0.0792 + 0.0617 = -0.0175$$

$$D(0) = 0.0915 + 0.0305 = 0.1220$$

$$D(1) = 0.0915 - 0.0305 = 0.0610$$

$$H(0) = -0.1409 + 0.1220$$

$$= -0.0189$$

$$H(1) = -0.0175 + (-j) \times 0.0610$$

$$= -0.0175 - j0.0610$$

$$H(2) = -0.1409 - 0.1220$$

$$= -0.2629$$

$$H(3) = -0.0175 + (-1) \times (-j) \times 0.0610$$

$$= -0.0175 + j0.0610$$

$$G(0) = A(0) + B(0) \times \omega_8^0$$

$$= -0.1764 + 0.0110$$

$$= -0.1654$$

$$G(1) = A(1) + B(1) \times \omega_8^2$$

$$= -0.1662 + (0.0434)(-j)$$

$$= -0.1662 - j0.0434$$

$$G(2) = -0.1764 - 1 \times 1 \times 0.0110$$

$$= -0.1874$$

$$G(3) = -0.1662 - 1 \times -j \times 0.0434$$

$$= -0.1662 + j0.0434$$

$$X(0) = G(0) + \omega_8^0 \times H(0)$$

$$= -0.1654 + (-0.0189)$$

$$= -0.1843$$

$$X(1) = G(1) + \omega_8^1 H(1)$$

$$= -0.1662 - j0.0434 + (0.707 - j0.707) \times (-0.0175 - j0.0610)$$

$$= -0.2217 - 0.0742j$$

$$X(2) = -0.1874 + (-j) \times (-0.2629)$$

$$= -0.1874 + j0.2629$$

$$X(3) = -0.1662 + j0.0434 + (-0.707 - 0.707j) \times (-0.0189)$$

$$= -0.1528 + 0.0568j$$

$$X(4) = -0.1654 + 0.0189 =$$

$$X(5) = -0.1662 - j0.0434 - (0.707 - j0.707) \times (-0.0175 - j0.0610)$$

$$= -0.1107 - 0.0126j$$

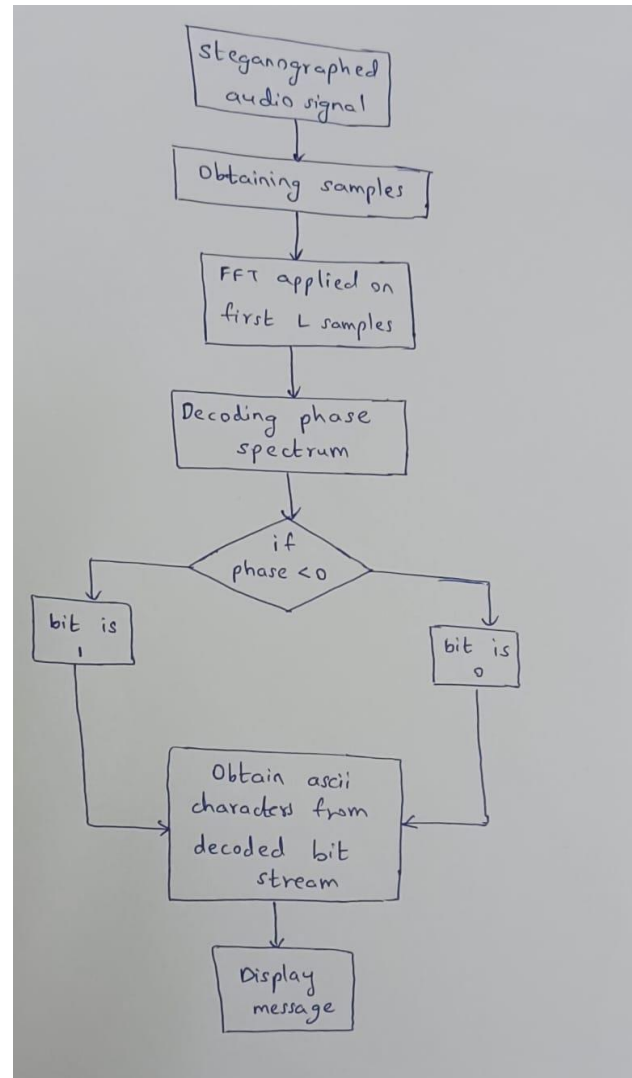
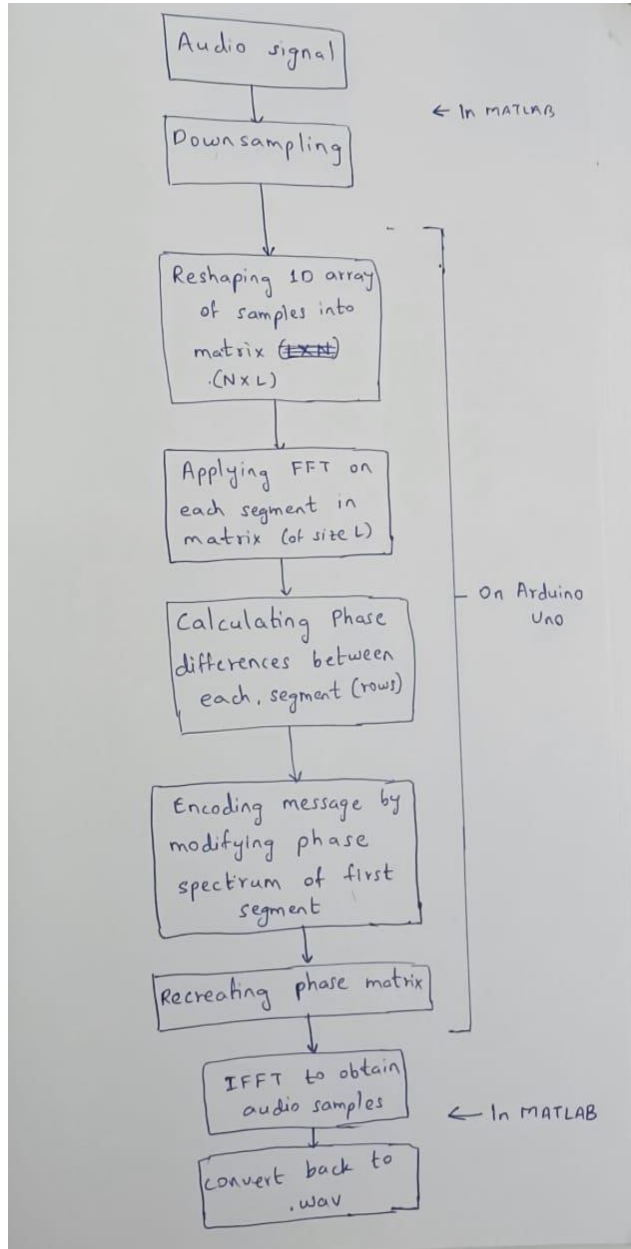
$$X(6) = -0.1874 - (-j) \times (-0.2629)$$

$$= -0.1874 - j(0.2629)$$

$$X(7) = -0.1662 + j0.0434 - (-0.707 - 0.707j) \times (-0.0175 + j0.0610)$$

$$= -0.2217 + 0.0741j$$

3. IMPLEMENTATION



3.1 PREPROCESSING OF AUDIO SIGNAL

Initially we started by taking 8 bit mono .wav file in MATLAB and converted it into .txt format using MATLAB. To reduce computations, we downsampled our audio signal from 44100 to 2940 i.e. 15 times downsampled using the below code.


```

Editor - C:\Users\Varsha\Desktop\5th sem notes\DSP\phase_coding\phase_coding\downsampling_audio.m
1      % Read the audio file
2      [FileName, PathName] = uigetfile('*.wav', 'Select audio file. ');
3      [~, audio.name] = fileparts(FileName);
4      [audio.data, audio.fs] = audioread([PathName FileName]);
5
6      % Set the desired decimation factor
7      decimationFactor = 15; % Change this to your desired factor
8      % Downsample by selecting every decimationFactor-th sample
9      downsampledData = audio.data(1:decimationFactor:end);
10
11     % Write the downsampled data to a text file
12     outputFileText = 'decimated_signal.txt';
13     dlmwrite(outputFileText, downsampledData, 'precision', 6, 'delimiter', '\t');
14     disp(['Downsampled audio data written to ' outputFileText]);
15     % Write the downsampled data to a new .wav file
16     outputFileWav = 'decimated_signal.wav';
17     audiowrite(outputFileWav, downsampledData, audio.fs / decimationFactor);
18     disp(['Downsampled audio data written to ' outputFileWav]);

```

The text file “audio_samples.txt” created consists of downsampled signal values.

```

Editor - C:\Users\Varsha\Desktop\5th sem notes\DSP\phase_coding\phase_coding\audio_to_text.m
1      % Read the audio file
2      [FileName, PathName] = uigetfile('*.wav', 'Select audio file. ');
3      [~, audio.name] = fileparts(FileName);
4      [audio.data, audio.fs] = audioread([PathName FileName]);
5
6      outputFile = 'audio_samples.txt';
7      dlmwrite(outputFile, audio.data, 'precision', 16, 'delimiter', '\t');

```

3.2 PHASE ENCODING

The data bits of the message text are encoded in the phase spectrum of the audio signal. This is performed by first converting each character in the message text into an 8 bit binary number, and embedding each character into one segment of the fourier transform of the signal.

In a given segment, each bit is encoded into one phase value by changing the phase value. This is done as follows:

$$\text{Phase}_{\text{new}} = \pi/2 \text{ if message bit} = 0$$

$$-\pi/2 \text{ if message bit} = 1$$

Let us consider i as signalLength, m as textLenth*8 and N as i/L . Where L is the length of each segment(Here 32). N represents the number of segments(Here 1). First $N \times L$ samples

from input signal are reshaped into matrix of size $N \times L$ i.e 1×32 . Note that the size of input signal is taken only 32 due to size constraint of Arduino Uno.

```

149 void phaseCoding(float* signal, int signalLength, char* text, int textLength, float* output) {
150     int i = signalLength; // total signal length
151     int m = textLength*8; // number of bits in message
152     int N = i / L; // number of segments
153     float s[N][L];
154     float DeltaPhi[N][L]; // to store phase differences of adjacent segments
155     float PhiData[m]; // to store phase representation of data
156     float Phin[N][L]; // new phase matrix
157     char* bin_seq = getBits(text); // text to binary
158 }

```

Now we calculated fast fourier transform of each segment obtained. The output of FFT function is an array of real and imaginary values. Using these, phase and magnitude are calculated and stored in $Phin[N][L]$ and $s[N][L]$.

```

160 // Dividing audio file into segments
161 for (int i = 0; i < N; ++i) {
162     float real[L];
163     float imag[L];
164     for (int j = 0; j < L; ++j) {
165         real[j] = signal[i * L + j];
166         imag[j] = 0.0;
167     }
168     // Extract each segment and do fft
169     fft(real, imag, L, L);
170     // extract the phase from FFT outputs
171     for (int j = 0; j < L; ++j) {
172         if ((real[j] < 0) && (imag[j] > 0))
173             Phin[i][j] = PI + atan(imag[j]/real[j]);
174         else if ((real[j] < 0) && (imag[j] < 0))
175             Phin[i][j] = -(PI - atan(imag[j]/real[j]));
176         else
177             Phin[i][j] = atan(imag[j]/real[j]);
178         s[i][j] = sqrt(real[j] * real[j] + imag[j] * imag[j]); // finding magnitude
179     }
180 }

```

The phase values are calculated in radians and range from -180 to +180 degrees. Next step is to calculate phase difference of adjacent segments as the phase shift between consecutive segments is easily detected. We only change the values of phase but not the relative difference between them and store them in $DeltaPhi[N][L]$. We converted our text into binary sequence of length $8 \times \text{textlength}$. An array of size $PhiData[m]$ stores $\pi/2$ if corresponding i th bit is 0 else it stores $-\pi/2$ in radians.

```

182 // Calculating phase differences of adjacent segments
183 for (int k = 1; k < N; ++k) {
184     for (int i = 0; i < L; ++i) {
185         DeltaPhi[k][i] = Phin[k][i] - Phin[k-1][i];
186     }
187 }
188 // Binary data is represented as {-pi/2, pi/2} and stored in PhiData
189 for (int k = 0; k < m; ++k) {
190     PhiData[k] = (bin_seq[k] == '0') ? PI / 2.0 : -PI / 2.0;
191 }

```

Hermitian symmetry, also known as conjugate symmetry, refers to a property where the values of a function or signal have a specific relationship with their complex conjugates. It is being used to ensure that the modified phase data maintains a symmetric pattern when applied to the Fourier-transformed segments of the audio signal. The purpose of maintaining Hermitian symmetry in this context is likely related to the fact that the input signal is real-valued. For real-valued signals, the magnitude spectrum is symmetric, and the phase spectrum exhibits Hermitian symmetry. When modifying the phase information, it's important to ensure that the resulting signal remains a valid, real-valued signal after the inverse Fourier transform.. So add Phidata to the first segment of Phin from $L/2-m$ to $L/2-1$ and from $L/2+1$ to $L/2+1+m$. Next were created phase matrices by adding the phase differences matrix and Phin matrix.

```

192 // embedding message in matrix
193 for (int k = L/2-m; k <= L/2-1; ++k) {
194     Phin[0][k] = PhiData[k-(L/2-m)];
195 }
196 //Maintaining Hermitian symmetry
197 for (int k = 0; k < m; ++k) {
198     Phin[0][L/2+1+k] = -PhiData[m-k-1];
199 }
200 // Re-creating phase matrices using phase differences
201 for (int k = 1; k < N; ++k) {
202     for (int i = 0; i < L; ++i) {
203         Phin[k][i] = Phin[k-1][i] + DeltaPhi[k][i];
204     }
205 }

```

We now have our new phase matrix. Next we re-created our audio signal by doing IDFT using new phase matrix and original amplitude matrix. We only took $N \times L$ values for encoding and now we combined these values to the rest of the signal and stored in output. This gives us our steganographed audio signal.

```

207     for (int i = 0; i < N; ++i)
208     {
209         float real[L];
210         float imag[L];
211         for (int j = 0; j < L; ++j) {
212             real[j] = s[i][j] * cos(Phin[i][j]);
213             imag[j] = s[i][j] * sin(Phin[i][j]);
214         }
215         ifft(real, imag, L, L); // regenerating audio samples
216
217         for (int j = 0; j < L; ++j){
218             output[j+L*i]=real[j];
219         }
220     }
221     for (int j = N*L; j < signalLength; ++j)
222         output[j] = signal[j];
223 }

```

FFT of segment

$$\text{fft}[8] = \{-0.12, -0.17 - 0.06i, -0.41 - 0.48i, 0.27 + 0.28i, \\ 0.19, -0.18 + 0.05i, -0.95 + 0.70i, -0.36\}$$

Phase spectrum of fft

$$\text{Phin}[8] = \{0, 0.339, 0.863, 0.803, 0, -0.271, -0.635, 0\}$$

Amplitude spectrum of fft

$$s[8] = \{0, -0.06, -0.48, 0.28, -0, 0.05, 0.70, 0\}$$

Binary sequence of coded text

$$\text{bin-seq} = [0, 1, 0, 0, 1, 0, 0, 0]$$

Encoded phase spectrum

$$\text{phiData}[8] = \{1.57, -1.57, 1.57, 1.57, -1.57, 1.57, 1.57, 1.57\}$$

Now we convert .txt file which contain values of steganographed signal back into ,wav audio file using the following Matlab code.

```
Editor - C:\Users\Varsha\Desktop\5th sem notes\DSP\phase_coding\phase_coding\text_to_audio.m
1      % Load the data from the text file
2      file = 'stego_file.txt';
3      data = load(file);
4
5      % Specify the sampling frequency (replace with your actual sampling frequency)
6      fs = 2940;
7
8      % Create a .wav file
9      output_file = 'stego_file.wav';
10
11     % Write the data to the .wav file
12     audiowrite(output_file, data, fs);
13
14     disp('WAV file created successfully.');
```

3.3 PHASE DECODING

In the above sample, the bit sequence 01001000 was encoded in the segment and the phase values of the spectrum were modified accordingly.

To decode the spectrum and extract the bit sequence, we first perform the fourier transform of the steganographed audio signal.

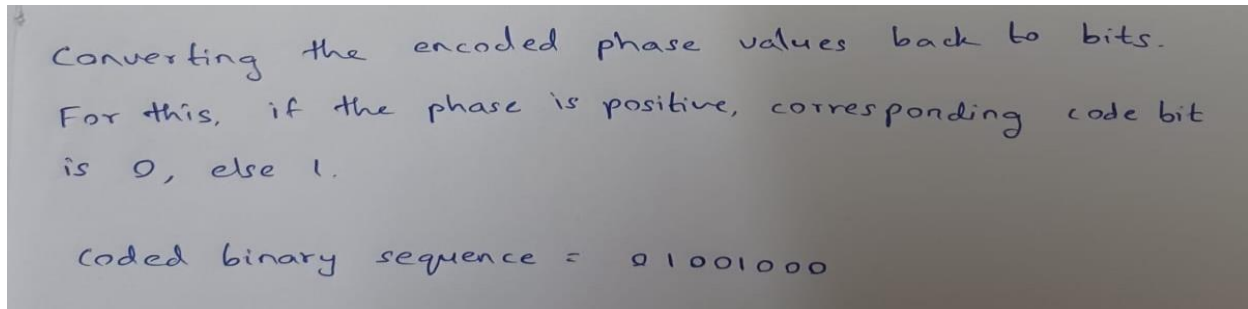
```
char Decoding(float *signal, int sig_lines, int L_msg) {
    int m = 8 * L_msg;
    char bin_data[m];
    float real[L];
    float imag[L];
    for (int i = 0; i < L; ++i) {
        real[i] = signal[i];
        imag[i]=0;
    }
}
```

To extract the encoded bits,we use the below method:

Binary bit = (0, if phase > 0

1, if phase < 0)

I.e., if the phase for a particular frequency is negative, the bit corresponding to that position is 1, else 0.



Converting the encoded phase values back to bits.
For this, if the phase is positive, corresponding code bit is 0, else 1.

coded binary sequence = 01001000

The follow code segment achieves the above calculations and extracts the message signal from the coded phase spectrum

```
// PERFORMING FFT ON THE ENCODED SIGNAL AND OBTAINING THE PHASE SPECTRUM
fft(real, imag,L,L);
float Phin[L];
for (int j = 0; j < L; ++j) {
    if((real[j]<0)&&(imag[j]>0))
    {
        Phin[j]=PI+atan(imag[j]/real[j]);
    }
    else if((real[j]<0)&&(imag[j]<0))
    {
        Phin[j]=-(PI-atan(imag[j]/real[j]));
    }
    else
    {
        Phin[j]=atan(imag[j]/real[j]);
    }
}
```

4. RESULTS

An audio signal of frequency 44100 Hz was considered for steganographing a message of ascii characters. The audio signal was downsampled to 2980 Hz, to reduce the number of samples while maintaining the same quality of signal. Matlab was used for reading the samples of audio from the mono.wave input signal and efficiently reducing the number of samples.

The samples were then stored on Arduino uno (ATMega328P controller) to perform offline phase coding on the spectrum of the audio samples.

```
Output  Serial Monitor x
Message (Enter to send message to 'Arduino Uno' on 'COM5')

reconstructed stego signal
-0.09,
-0.10,
0.04,
0.07,
0.06,
-0.08,
-0.02,
0.03,
0.15,
0.09,
0.00,
-0.02,
-0.04,
-0.08,
0.02,
0.00,
-0.04,
```

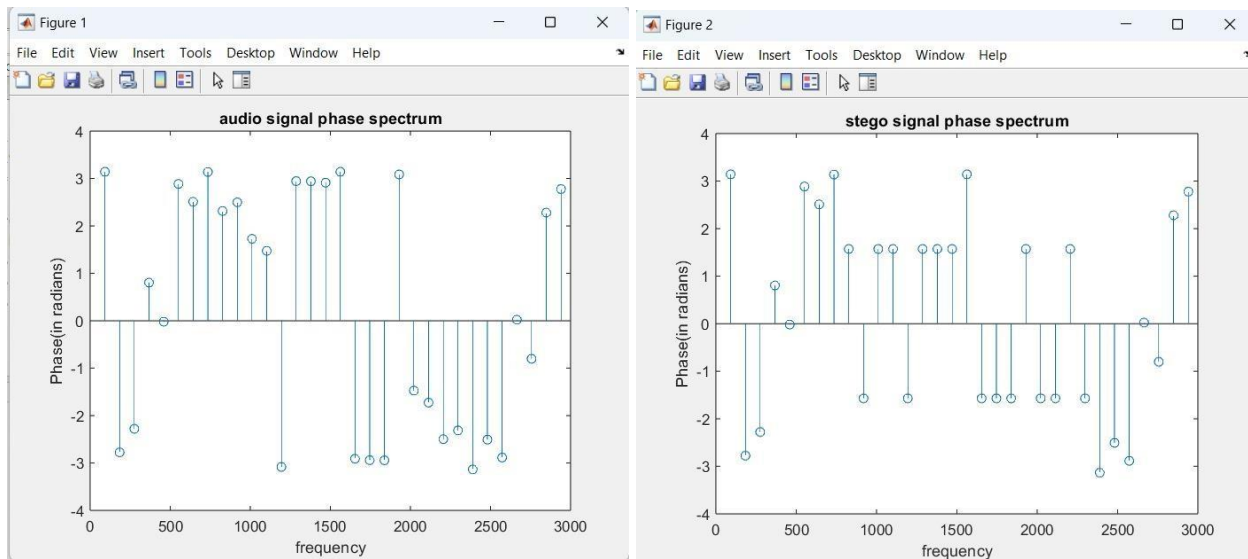
These audio samples were then converted back into mono.wave format through matlab.

Another code then processes the encoded audio samples on the arduino and extracts back the message signal through the decoding algorithm of phase-coding steganography.

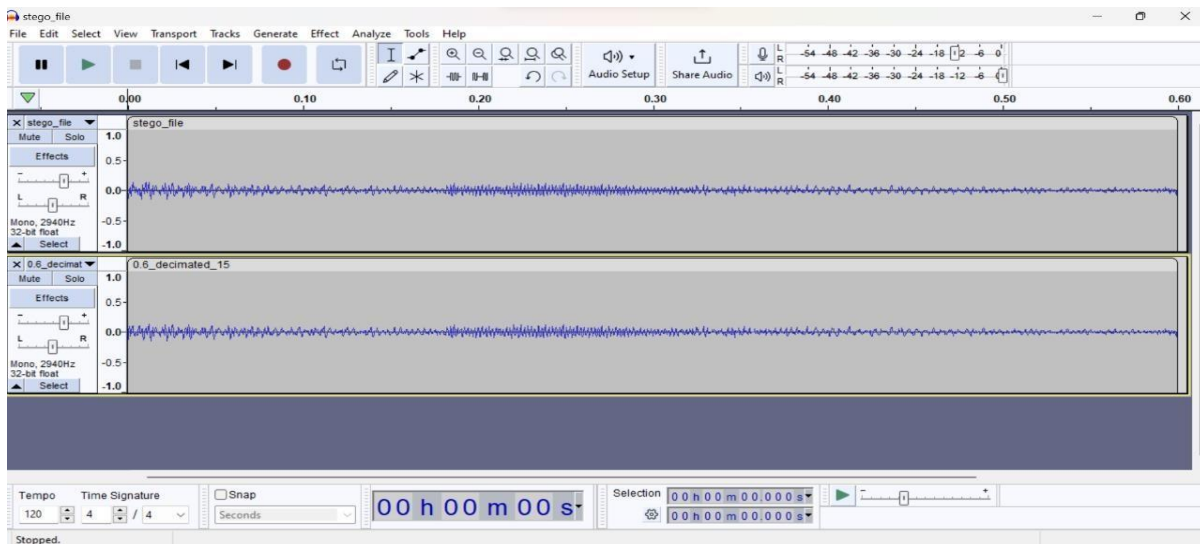
```
Output  Serial Monitor x
Message (Enter to send message to 'Arduino Uno' on 'COM5')

extracted message
H
```


Phase spectrum before and after Phase Encoding: (In MATLAB)



The following are respectively the waveforms of steganographed and original audio signals in the time domain.



There is an insignificant distortion in the waveforms of the original and encoded signals.

Phase coding is a steganography algorithm which encodes the message into the frequency domain. Compared to other techniques, it offers more encoding capacity, reduced susceptibility to steganalysis and improved perpetual transparency as the human ear is not less sensitive to phase shifts in the audio signal.

5. FUTURE SCOPE

- Due to limited memory of Arduino UNO we are using very less samples of audio signal to encode the text. The size of text to be encoded is currently limited to one character. Our future plan is to use SD card for storing large amount of audio data and phase code it.
- The algorithm can also be implemented on controllers such as Arduino Mega which have larger sizes of RAM.
- A combination of Phase coding and another steganography algorithm can be used to make it more secure. Message signal can also be encrypted prior to encoding for more secure transmission.

6. CONCLUSION

In conclusion, the implementation of phase coding steganography on an audio signal has proven to be effective in concealing a message within the frequency domain. The decision to downsample the audio signal to 2980 Hz while maintaining signal quality demonstrates a thoughtful trade-off between reducing computational load and preserving the essential information for steganographic purposes.

The integration of Matlab and an Arduino Uno, specifically utilizing the ATmega328P controller, showcases a practical approach to embedding and extracting hidden information in an offline setting. The inconspicuous distortion observed in the waveforms of the original and encoded signals in the time domain indicates a successful implementation, highlighting the potential for maintaining audio quality during the steganographic process.

While this report provides a successful implementation of phase coding steganography, it is essential to acknowledge that the effectiveness of any steganographic technique is contingent upon various factors, including the sophistication of potential adversaries and the specific requirements of the application. Future work could explore the robustness of the proposed method against more advanced steganalysis techniques and evaluate its performance in diverse audio environments. Nonetheless, the presented system stands as a promising example of leveraging phase coding for covert communication within audio signals.

REFERENCES

<https://ieeexplore.ieee.org/document/7423319>

https://doi.org/10.2991/978-94-6463-084-8_8

<https://github.com/ktekeli/audio-steganography-algorithms/tree/master/04-Phase-Coding>