EE338
Digital Signal Processing
Evaluation Component 5

Autumn 2020

Group 18

Adway Girish 180070002
Andrews George Varghese 180070005
Nayan Khemraj Barhate 180070037

Contents

1 Abstract **1**

2 Digital Signal Processing principles involved **2**
2.1 DFT . . . 2
2.2 Convolution . . . 2
2.3 Cepstrum . . . 3

3 Audio watermarking **4**
3.1 Phase Coding . . . 4
3.2 Echo data hiding . . . 4

4 Our work **8**
4.1 Phase coding . . . 8
4.1.1 Encoding . . . 8
4.1.2 Decoding . . . 10
4.1.3 Auxiliary functions . . . 11
4.2 Echo data hiding . . . 12
4.2.1 Encoding . . . 12
4.2.2 Decoding . . . 14
4.3 Error analysis and effect of noise . . . 15
4.3.1 Phase coding . . . 15
4.3.2 Echo data hiding . . . 15

5 Conclusion **17**

1 Abstract

Here is our original abstract:

> "Watermarking is the process of embedding information into a signal. In particular, audio watermarking is the process of adding a distinctive sound pattern, imperceptible to the human ear, to an audio signal to allow a computer to identify it. This is done in such a way that it is difficult to remove. If the audio is copied, so is the watermark. This helps to identify files that have been illegally reproduced, and hence, watermarks are becoming widely popular in enabling copyright protection and ownership verification.
>
> But why is this important? As we move towards a 'self-reliant' and 'digital' India, a huge quantity of original content is being produced and it is becoming increasingly important to ensure the protection of intellectual property rights. Audio watermarking can help in this regard. Also, the presence of a watermark can be used to check for the authenticity of audio messages, which is a very valuable tool, especially in this age of deep fakes.
>
> We plan to do so by one of spread spectrum, least significant bit coding, phase coding and echo hiding. The aim will be to implement the watermarking while ensuring that
>
> 1. The watermark cannot be easily removed.
> 2. The quality of the audio signal is not compromised."

We have followed this without any deviations. We have implemented the process of audio watermarking using two methods - Phase coding and Echo data hiding. The organisation of this report is as follows: §2 introduces the relevant concepts from the field of Digital Signal Processing that are used in the implementation. §3 briefly explains the various steps involved in the process of watermarking are, for both the methods. §4 shows our investigations on this topic, including snippets of our code and the results obtained. §5 summarises and provides a comparison of the two methods used.

2 Digital Signal Processing principles involved

First, here are some of the concepts from Digital Signal Processing will be applied to implement our watermarking process (explained in §3.1 and §3.2).

2.1 DFT

The Discrete Fourier Transform converts a real discrete-time sequence $x[n]$ of length N into a complex discrete-frequency sequence $X[k]$ of the same length by

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn}$$

The DFT is a way of representing a time-domain signal in the frequency-domain. It is implemented using numerical algorithms such as the Fast Fourier Transform (FFT), which greatly increases the efficiency of computation. Since $X[k]$ is a complex quantity, we can talk of its magnitude and phase separately.

2.2 Convolution

The convolution is a mathematical operation on two functions to produce a third function which represents the "spreading" of one function over another. Given two discrete-time signals $x[n]$ and $y[n]$ of length N, we have the convolution $z[n]$ given by

$$z[n] = x[n] * y[n] = \sum_{l=0}^{N-1} \hat{x}[l]\, \hat{y}[n-l], \text{ for } 0 \leq n \leq N-1$$

where $\hat{x}$ and $\hat{y}$ are infinite-length periodic extensions of x and y respectively (this is only to account for cases when $n - l < 0$ or $n - l > N - 1$ since y is not defined there). Fig. 1 shows this operation in action.

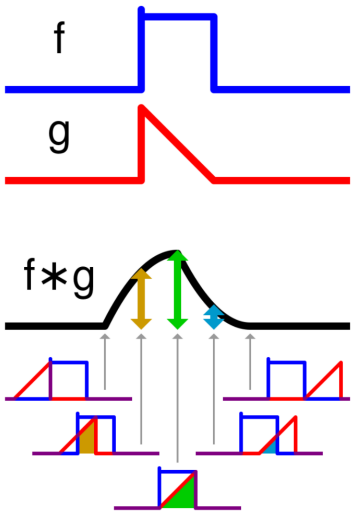


Figure 1: Visualisation of convolution as a measure of "spreading", taken from here

When we say that a signal x is "filtered" with another signal y, what we mean is that these signals are convolved with each other to obtain a new signal z. Interestingly, we have a simpler relation in the frequency domain when two signals are convolved:

$$\begin{aligned}
Z[k] &= \sum_{n=0}^{N-1} z[n]\, e^{-j\frac{2\pi}{N}kn} = \sum_{n=0}^{N-1} \left(\sum_{l=0}^{N-1} \hat{x}[l]\, \hat{y}[n-l]\, e^{-j\frac{2\pi}{N}kn} \right) \\
&= \sum_{n=0}^{N-1} \left(\sum_{l=0}^{N-1} \hat{x}[l]\, \hat{y}[n-l]\, e^{-j\frac{2\pi}{N}k(l+n-l)} \right) \\
&= \sum_{n=0}^{N-1} \left(\sum_{l=0}^{N-1} \hat{x}[l]\, \hat{y}[n-l]\, e^{-j\frac{2\pi}{N}k(l+n-l)} \right) \text{ put } m = n-l \\
&= \sum_{m=-l}^{N-1-l} \left(\sum_{l=0}^{N-1} \hat{x}[l]\, \hat{y}[m]\, e^{-j\frac{2\pi}{N}k(l+m)} \right) \\
&= \sum_{l=0}^{N-1} \hat{x}[l]\, e^{-j\frac{2\pi}{N}kl} \left(\sum_{m=-l}^{N-1-l} \hat{y}[m]\, e^{-j\frac{2\pi}{N}km} \right) = \sum_{l=0}^{N-1} \hat{x}[l]\, e^{-j\frac{2\pi}{N}kl}\, Y[k] \\
&= \left(\sum_{l=0}^{N-1} \hat{x}[l]\, e^{-j\frac{2\pi}{N}kl} \right) Y[k] = X[k]Y[k]
\end{aligned}$$

Hence we have a linear transformation which allows us to convert the rather complicated convolution in one domain into the much simpler multiplication operation in another domain.

2.3 Cepstrum

The cepstrum is a signal processing tool used widely, particularly when dealing with audio signals. It is defined as the signal whose Fourier transform is the logarithm of the Fourier transform of the original signal, i.e. the cepstrum of a signal $x[n]$ with DFT $X[k]$ is given by

$$C_x[n] = \mathcal{F}^{-1}\{\log|X[k]|\}$$

The cepstrum is useful when dealing with the deconvolution of signals. When two signals $x[n]$ and $y[n]$ are convolved to obtain $z[n]$, i.e $z[n] = x[n] * y[n]$, their DFTs are related as $Z[k] = X[k]Y[k]$. Then the cepstrum of $z[n]$ is given by

$$\begin{aligned}
C_z[n] &= \mathcal{F}^{-1}\{\log|Z[k]|\} = \mathcal{F}^{-1}\{\log|X[k]Y[k]|\} \\
&= \mathcal{F}^{-1}\{\log|X[k]| + \log|Y[k]|\} = \mathcal{F}^{-1}\{\log|X[k]|\} + \mathcal{F}^{-1}\{\log|Y[k]|\} \\
&= C_x[n] + C_y[n]
\end{aligned}$$

Hence we see that a cepstrum gives us a homomorphism from a system with convolution as the operation to one with addition, which is considerably easier to work with.

3 Audio watermarking

3.1 Phase Coding

In this method, the data to be stored (call it the message signal) is encoded in the *spectral phase* of the cover song (call it the base signal). This works on the principle that in the frequency domain, the average human ear is insensitive to small spectral phase changes (we keep the spectral magnitudes constant). The ear is instead most sensitive to the relative difference in phase between consecutive segments of the cover song.

In this algorithm, the message is converted to bits, and bit '0' is assigned a phase of $\phi_0 = \frac{\pi}{2}$ and bit '1' a phase of $-\phi_0 = -\frac{\pi}{2}$. We use these phase values as we want the phases to be as far apart from each other as possible, so as to make the encoding more robust to noise. The process of encoding these phases (representing bits) is briefly described below. Illustrations of the same can be found in Fig. 2.

1. The base signal is first divided into segments of required size.
2. We obtain the DFT of each segment, and preserve the amplitude spectrum of each. We work on the phase spectra.
3. We calculate the phase difference between adjacent segments, and save this.
4. We store the phases (representing the bits of the message) in the phase spectrum of the first segment (while maintaining hermitian symmetry), and then construct the phase spectra of the following segments using the phase differences and the first segment.
5. Finally, we concatenate the inverse DFTs of each segment and obtain the modified signal.

In order to retrieve the message when using phase encoding, the receiver needs to know how long the original message was, the length of each segment and what kind of message it was (for example, a .txt file). Given these, the decoding process is:

1. Obtain the first segment of the modified signal.
2. Calculate the DFT of this first segment, and obtain its phase spectrum.
3. Wherever the phase is $\frac{\pi}{2}$, the corresponding data bit is '0', and is '1' wherever the phase is $-\frac{\pi}{2}$

We thus obtain the bit stream corresponding to the original hidden message. Given the file type of the original message, we now obtain the final decoded message.

3.2 Echo data hiding

The basic idea behind echo hiding is to artificially introduce an echo for each bit of data to be embedded. By controlling parameters such as the amplitude of the echo and the delay of the echoes representing the "one" and "zero" bits, we can ensure the inaudibility of the embedded signal as well as the robustness of the hiding.

The details of the implementation vary greatly, since there are different ways of introducing the echo. The most straightforward is the single echo kernel method, which only involves one echo apart from the original signal. We can obtain a more natural sound by using a time-spread echo kernel, which better represents the natural echoing due to reflections in a room. Here we will use the former method to simply illustrate the process.

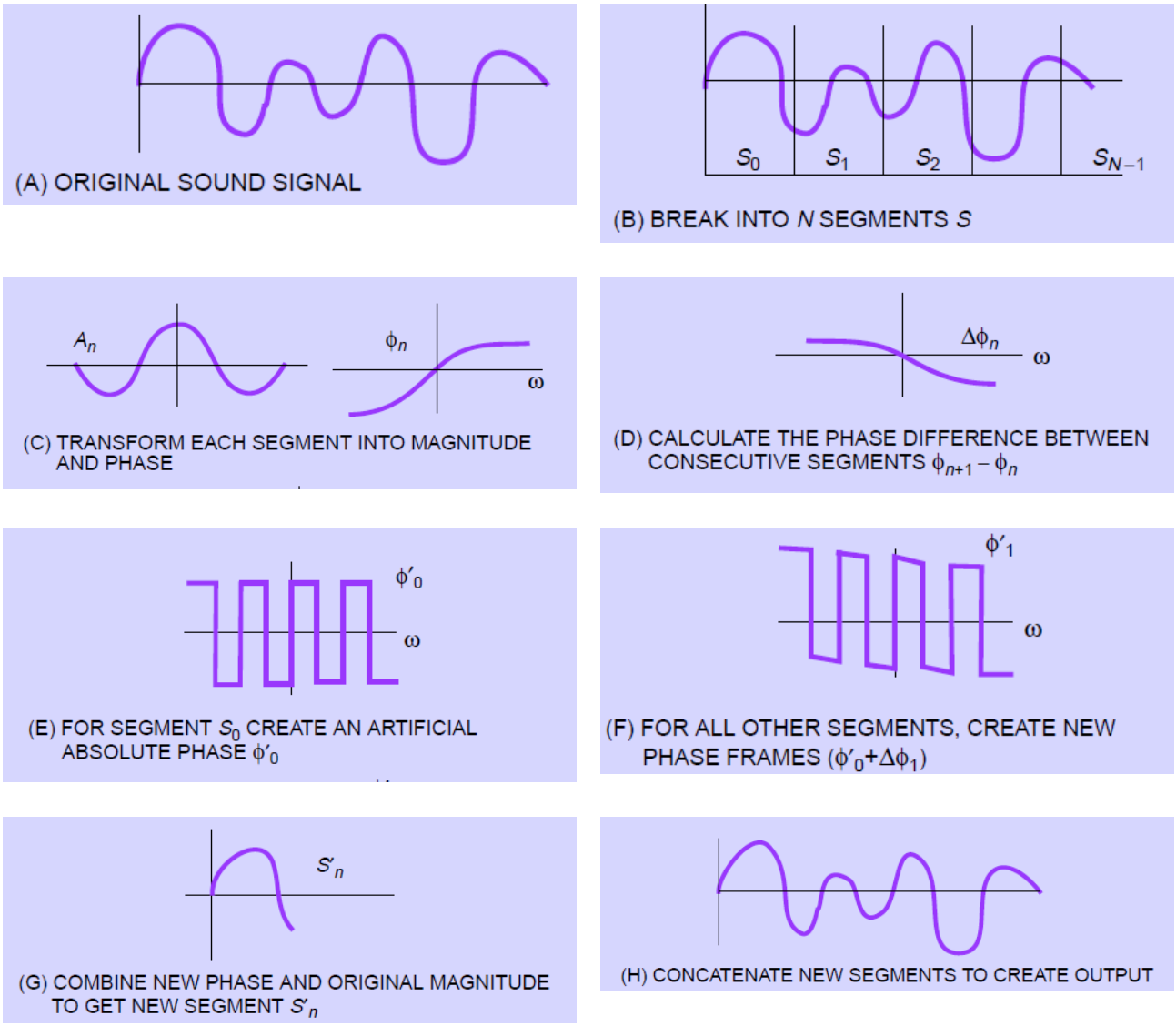


Figure 2: Illustration of the steps involved in phase encoding (Figures taken from [1])

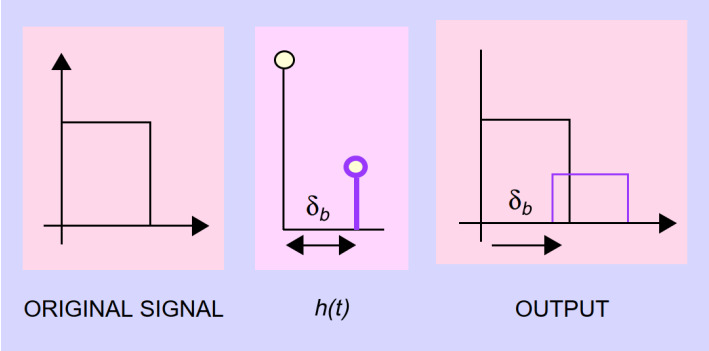


Figure 3: Introducing an echo using a single echo kernel. We use different δ_b to differentiate between "zero" and "one" bits. (Figure taken from [1])

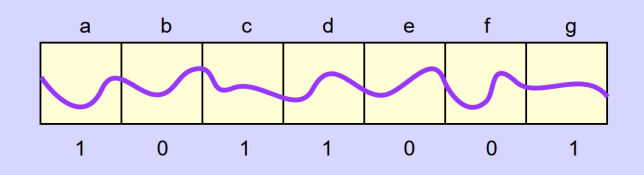
We ensure inaudibility by choosing delays small enough that the original sound and the echo blend together as one (this is helped by the average human ear being unable to resolve sounds that are less than a microsecond apart). Giving the echo an amplitude considerably smaller than the original also helps mask it. The process of encoding can be summarised as:

1. First the entire signal is divided into segments of some fixed size to represent each bit.
2. Each segment is filtered using the appropriate kernel ("zero" or "one", depending on the bit that segment is to carry). The implementation for this is carried out in a more efficient manner using a mixer.
3. The segments are put together to obtain the watermarked signal with the encoding.

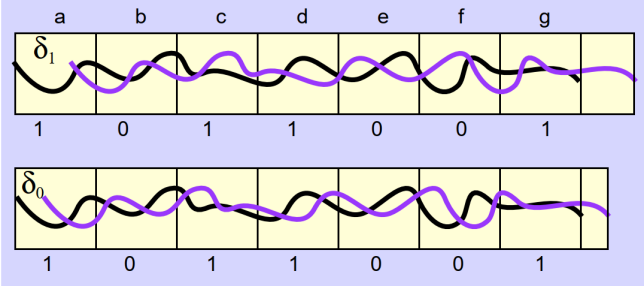
It is also important to be able to extract the embedded information from the encoded signal. The philosophy here is to detect the spacing between the echoes of each segment, since that is how we embedded the bits into the signal. We convolve a kernel with the signal to generate the echo which contains the hidden data. To decode the hidden message, we need to look at the echo in each segment and classify the kernel used to generate that echo. The cesptrum turns out to be very handy for this. The cepstrum of the encoded signal will be the sum of the cepstrum of the kernel and that of the original signal. While non-linear, the cepstrum is still homomorphic, so the value of the cepstrum of any signal at a certain time is likely to be greater than the value at another time if the signal itself also follows the same order. Hence, comparing the values of the cepstra at the positions of the impulses in the "zero" and "one" kernels, we can make an estimate for which kernel was used to generate the echo, and hence predict the encoded bit. To decode the encoded signal and obtain the hidden bits:

1. The length of the segment needed to represent each bit is known and this is used to identify the segments representing each bit.
2. The cesptrum of each segment is found, and the magnitudes of the real cepstrum are compared at the delays corresponding to the "zero" bit and the "one" bit.
3. The one with the higher magnitude is assigned as the bit value corresponding to that segment.

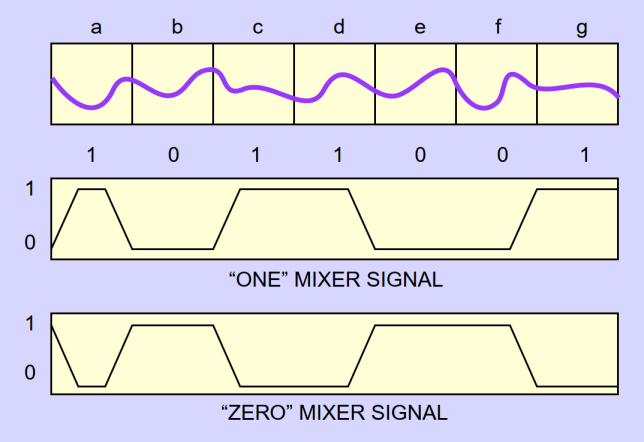
Thus we can extract the entire sequence of bits embedded into the original signal. A summary of the encoding process is shown in Fig. 4



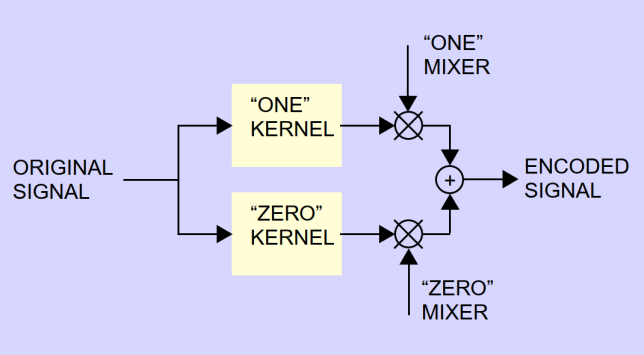
(a) Divide the original signal into segments for each bit



(b) "One" signal and "zero" signal generated by filtering original signal with "One" and "Zero" kernels respectively



(c) "One" mixer and "Zero" mixer with smoothed transitions



(d) Obtaining the final encoded signal by combining the signals with the respective mixers

Figure 4: The process of encoding using echo hiding (Figures taken from [1])

4 Our work

We have carried out the process of watermarking on the audio file given in this in drive[1] using two methods: echo data hiding and phase coding. We have analysed the accuracy of the methods using the Bit Error Rate and also looked at the deterioration in performance in the presence of noise.

4.1 Phase coding

Here is the detailed algorithm for the method given in §3.1, along with code snippets of the implementation in MATLAB to make understanding it easier.

4.1.1 Encoding

1. Store the base song signal onto which encoding is performed in the variable `signal`, and message text in the variable `text`

2. Initialise the following variables (`getBits` gets the bit sequence equivalent to the message in text, as defined in §4.1.3)

```
base = signal(:,1);
msg = getBits(text);
I = length(base);
m = length(msg);      % Length of the bit sequence to be hidden
```

3. Obtain the length of each segment and the number of segments (we have used powers of 2 for lengths of the segments, as taking FFT of such segments is the most efficient, and this provided better results as far as audio quality is concerned)

```
L = min(2^nextpow2(2*m), I)% length of each segment
if L == 2*m
    L = L + 2;
    if L >= I
        disp(['Text message too long - find a longer base signal']);
        out = 0;
        return;
    end
end
halfL = floor(L/2);
N = floor(I/L);% Number of points per segment
```

4. Segment the signal, and obtain the DFTs of each segment. Calculate the phase and amplitude spectra for each segment

[1]The complete code and audio files used for all parts can be found here - `https://drive.google.com/drive/folders/1BvoXn1Ue_pdNI7mw48RtNcORmlJTw1gd?usp=sharing`

```
% Segment the signal
s = reshape(base(1:N*L,1), L, N);

% obtain L-point DFT of each signal, and the phase and amplitudes
w = fft(s);
Phi = angle(w);         % Phases - N columns, each of L rows
A = abs(w);             % Amplitudes
```

5. Obtain the phase differences between consecutive segments

```
% Calculating phase differences between adjacent segments
DeltaPhi = zeros(L,N);
for i=2:N
    DeltaPhi(:,i)=Phi(:,i)-Phi(:,i-1);
end
```

6. Convert the message data bit sequence into phases

```
% Convert the data bits into phase {'0' : pi/2, '1' : -pi/2}
PhiData = zeros(1, m);
phi0 = pi/2;
for i=1:m
    if msg(i) == '0'
        PhiData(i) =phi0;
    else
        PhiData(i) = -phi0;
    end
end
```

7. Store the message in the phase of the first segment

```
Phi_new = zeros(size(DeltaPhi));

% Save message in column 1 of the phase matrix
% remember to maintain hermitian symmetry
Phi_new(:,1) = Phi(:,1);
Phi_new(halfL-m+1:halfL,1) = PhiData;
Phi_new(halfL+1+1:halfL+1+m,1) = -flip(PhiData); % Hermitian symmetry
```

8. Obtain the phases for the remaining segments (by maintaining the phase differences between consecutive segments)

```
% Constructing phase matrix using phase differences DeltaPhi
for i=2:N
    Phi_new(:,i) = Phi_new(:,i-1) + DeltaPhi(:,i);
end
```

9. Finally, take the inverse DFT of each segment, and concatenate the segments in time.

```
% Reconstructing the sound signal by taking IFFT
z = real(ifft(A .* exp(1i*Phi_new)));
out = reshape(z, N*L, 1);                % concatenating
if N*L + 1 <= I
    % Adding rest of signal - I/L may not be an integer
    out  = [out; base(N*L+1:I)];
end
```

4.1.2 Decoding

Here is the detailed decoding algorithm, along with code snippets of the implementation in MATLAB to make understanding it easier:

1. Assume the encoded signal is stored in the variable *modifiedSignal* and the length of the message signal is in variable *L_msg*.

2. Initialise the various constants

```
% Length of bit sequence (assuming 8bits per character)
m_decode = 8*L_msg;
I_decode = length(modifiedSignal);
% length of each segment
L_decode = min(2^nextpow2(2*m_decode), I_decode);
if L_decode == 2*m_decode
    L_decode = L_decode + 2;
end
halfL_decode = floor(L_decode/2);
```

3. Obtain the first segment of *modifiedSignal* and obtain its phase spectrum

```
x = modifiedSignal(1:L_decode,1);        % First segment
Phi_decode = angle(fft(x));        % Phase angles of first segment
```

4. Retrieve the data stored in the phase of the first segment.

```
% Retrieving data back from phases stored in first segment
data_decode = char(zeros(1,m_decode));
for k=1:m_decode
        if Phi_decode(halfL_decode - m_decode + k)>=0
            data_decode(k)='0';
    else
        data_decode(k)='1';
        end
end
```

5. Process the data obtained and convert it into characters to be printed

```
bin_decode = reshape(data_decode(1:m_decode), 8, m_decode/8)';
out_decode = char(bin2dec(bin_decode))';
```

6. Obtain final error rates (`BER` calculates the bit error rate, as defined in §4.1.3)

```
ratio = BER(out_decode, text); % Bit error rate
disp(['******** RESULTS ********']);
fprintf('Original Text: %s\n', text);
fprintf('Decoded Text: %s\n', out_decode);
fprintf('BER : %d\n', ratio);
```

4.1.3 Auxiliary functions

The `getBits` function takes a string (each element is of data type `char`) and returns the bit sequence corresponding to that string.

```
function bit_seq = getBits(text)
    % we are assuming that we store txt - char data type
    matrix  = dec2bin(uint8(text),8);
    bit_seq = reshape(matrix', 1, 8*length(text));
end
```

The `BER` function calculates the bit error rate between 2 strings - a source string and a target string.

```
function out = BER(a, b)
    %BER Bit Error Rate
    a_bits = getBits(a);
    b_bits = getBits(b);
    len_a = length(a_bits);
    len_b = length(b_bits);
    len = min(len_a, len_b);
    ber = 0;
    for i=1:len
        ber = ber +  (a_bits(i) ~= b_bits(i));
    end
    ber = ber + abs(len_b - len_a);
    out = 100*(ber/len);
end
```

4.2 Echo data hiding

Here is the detailed algorithm for the method given in §3.2, along with code snippets of the implementation in MATLAB:

4.2.1 Encoding

1. Store the original signal onto which data is to be embedded in `signal` and the message text in `text` and initialise these related variables- `msg` stores the text as bits.

```
matrix  = dec2bin(uint8(text),8);
msg = reshape(matrix', 1, 8*length(text)); % Converts text into bits
I = length(signal);
m = length(msg);     % Length of the bit sequence to be hidden
```

2. Set these as the parameters for echo hiding.

```
delay0 = 150;      % Delay rate for bit0
delay1 = 200;      % Delay rate for bit1
amp = 0.5;         % Echo amplitude
L = 8*1024;        % Length of segment to encode each bit
nframe = floor(I/L);
N = nframe - mod(nframe,8); % Number of frames needed for all bits
```

3. Now we define the echo kernels for the "zero" and "one" bits, and generate the "zero" and "one" signals by filtering the original signal with the respective kernels - these are just the original signals with a delay and scaled by `amp`. Check Figs. 4b and 5a.

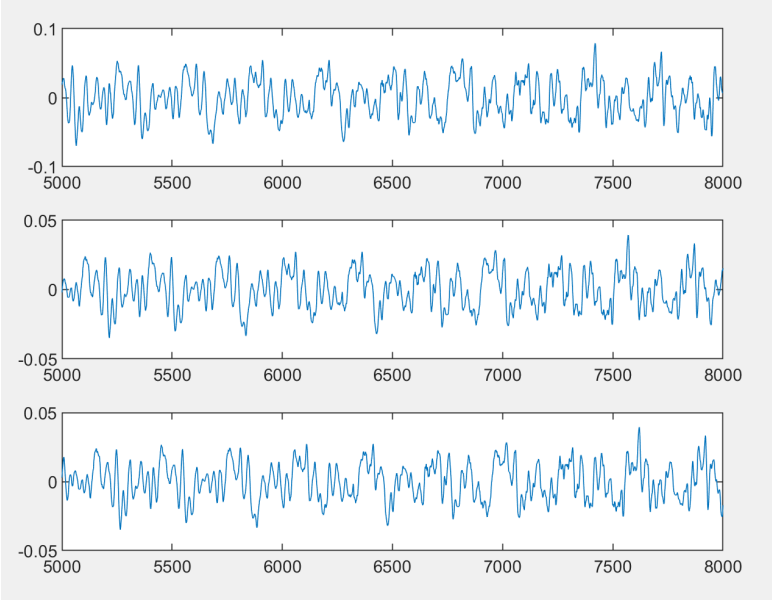
```
ker0 = [zeros(delay0, 1); 1]*amp;         % "zero" kernel
ker1 = [zeros(delay1, 1); 1]*amp;         % "one" kernel
echo_zero = filter(ker0, 1, signal);      % "zero" signal
echo_one = filter(ker1, 1, signal);       % "one" signal
```

4. Generate the mixer signal of length `N*L` which is zero for the intervals when `msg` is zero and one when `msg` is one, smoothen it using a Hanning window, and normalize it. Check Figs. 4c and 5b.

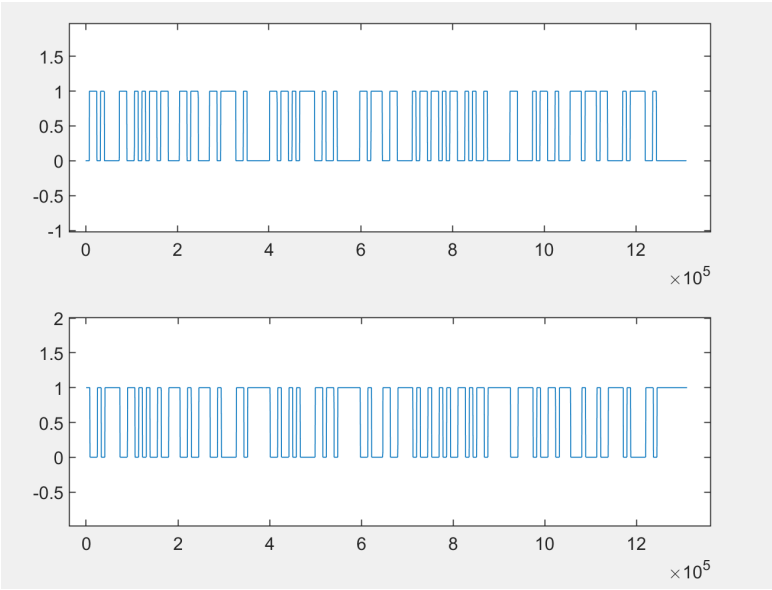
```
encbit = str2num(reshape(bits, N, 1))';
% Creates a matrix of all the bits
m_sig  = reshape(ones(L,1)*encbit, N*L, 1);
% Converts the matrix into one vector - a rudimentary mixer signal

K = 256; % length of Hanning window
c      = conv(m_sig, hanning(K));     % Windowing to smoothen
mix  = c(K/2+1:end-K/2+1) / max(abs(c)); % Normalization - final mixer
```

5. Now all that's left is to combine the mixer with the kernel to get the signal with the encoding.

```
encoded = signal(1:N*L, :) + echo_zero(1:N*L, :) .* abs(mix-1) ...
        + echo_one(1:N*L, :) .* mix;     % Including the bits
encoded = [encoded; signal(N*L+1:I, :)];   % Rest of the signal
```



(a) The topmost one is a segment of the original signal, the middle one is the same segment of the "zero" signal, and the bottom one is the same segment of the "one" signal (compare with Fig. 4b). Observe the different delays for the signals generated using the kernels.



(b) The upper one is the "one" mixer for some text and the lower one is the "zero" mixer for the same text to be hidden (compare with Fig. 4c).

Figure 5: Examples of mixer and "zero" and "one" signals from MATLAB simulations

4.2.2 Decoding

1. Assume the encoded signal is stored in the variable *encoded* and the length of each segment is L (just as for encoding).

2. Initialise some variables

```
N = floor(length(encoded)/L);    % Number of segments (and hence bits)
segs = reshape(encoded(1:N*L,1), L, N); % Dividing signal into segments
data = char.empty(N, 0);    % To store decoded message
```

3. Now for each segment, calculate the real spectrum and compare magnitudes at the delay values of the "zero" and "one" kernels. Assign one to this bit if the value is greater for the "one" kernel, and vice-versa.

```
for k=1:N
    rceps = ifft(log(abs(fft(segs(:,k)))));  % Real cepstrum
    if (rceps(d0+1) >= rceps(d1+1)) % Assign 0 if delay of "zero" kernel
        data(k) = '0';              % has higher cepstrum magnitude
    else                    % Assign 1 if delay of "one" kernel
        data(k) = '1';      % has higher cepstrum magnitude
    end
end
```

4. Finally, convert the binary sequence into text.

```
m   = floor(N/8);
bin = reshape(data(1:8*m), 8, m)';    % Message in binary
msg = char(bin2dec(bin))';            % Converting back to text
```

4.3 Error analysis and effect of noise

To test the quality of encoding and decoding, we have used a parameter called the Bit Error Rate (BER) defined as

$$\text{BER} = \frac{\text{number of bits incorrectly estimated}}{\text{total number of bits estimated}}$$

We also add Additive White Gaussian Noise of different Signal-to-Noise ratios to observe the performance of this algorithm under adverse conditions.

4.3.1 Phase coding

In the algorithm mentioned in §3.1, we used a phase of $\phi_0 = \pm\frac{\pi}{2}$ to encode bits `'0'` and `'1'`. This was to ensure maximum separation between the two bits, so as to make the encoding more robust to noise. Here we tabulate the bit error rates at varying levels of AWGN noise, for different values of ϕ_0.

ϕ_0	SNR (in dB)	BER (in %)
$\pi/10$	-	0
$\pi/10$	80	1.0
$\pi/10$	60	3.3
$\pi/10$	40	25.9
$\pi/5$	-	0
$\pi/5$	80	0.46
$\pi/5$	60	2.5
$\pi/5$	40	15.9
$\pi/2$	-	0
$\pi/2$	80	0.21
$\pi/2$	70	1.1
$\pi/2$	60	1.9
$\pi/2$	40	9.6

Table 1: Table showing data obtained from MATLAB simulations

4.3.2 Echo data hiding

Note that in the code for echo data hiding in §4.2, we took the length of the segment of the audio needed to code one bit, `L = 8*1024`. Taking a smaller value of `L` lets us encode more bits of data into the audio signal, but comes at the cost of increased errors while decoding. The value we used was arrived at after comparing the performances at different lengths under different noise conditions.

L	SNR (in dB)	BER (in %)
1024	-	0.78125
1024	80	9.7222
2*1024	-	0
2*1024	80	3.3333
2*1024	60	4.1667
4*1024	-	0
4*1024	80	2.1875
4*1024	60	3.75
4*1024	40	6.5625
8*1024	-	0
8*1024	80	0
8*1024	60	0.625
8*1024	40	1.875
8*1024	30	5.625

Table 2: Table showing data obtained from MATLAB simulations

Tables 1 and 2 show the results obtained. A BER greater than 10% makes the original data practically unrecognizable, and BER less than 2% usually gives a fairly decent decoding. These results are for one particularly text message encoded in one particular audio file only, but it was ensured by comparing with other files that this file was not an outlier in terms of BER.

5 Conclusion

We have seen two methods of hiding data in an audio file. Phase coding codes the entire signal into the fist segment, while echo data hiding requires the entire signal to transmit all the bits. Phase coding makes use of the phase to hide the data while echo data hiding uses the entire time domain sequence. Since variations in phase are difficult to detect audibly, it is possible to create larger differences with respect to the original while ensuring inaudibility. Hence we can encode a larger number of bits using phase coding as compared to echo hiding, where we require a large segment of the original signal to store each bit since the variation between the kernels used to represent the bits is only a very small delay.

We have only looked at an elementary implementation of watermarking. In practice there are many improvements made. The error rates can be reduced by using error correcting codes to detect and rectify errors in a few bits.

While these methods of data hiding have innumerable applications, here we consider their use in audio watermarking. These techniques can be used to encode some key that cannot be easily reproduced into some audio file such as an original recording of a song while it is produced. Since the encoding is embedded within the signal, it cannot be removed by any kind of filtering. To detect plagiarised copies of this original, the suspected audio file is checked for the presence of encoding and subjected to decoding. If the file has the original audio in a quality such that it is usable, the encoding can also be retrieved. This allows us to check for illegal copies. Further, a unique ID for the buyer can be encoded in the audio, and any suspected plagiarised copies can be decoded to obtain this information, and hence identify the guilty party.

References

[1] W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35(3.4):313–336, 1996.

[2] L. Boney, A. H. Tewfik, and K. N. Hamdy. Digital watermarks for audio signals. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems*, pages 473–480, 1996.

[3] Darko Kirovski and Henrique Malvar. Spread-spectrum watermarking of audio. *Signal Processing, IEEE Transactions on*, 51:1020 – 1033, 05 2003.

[4] Ryouichi Nishimura, Yôiti Suzuki, and B.-S Ko. *Advanced Audio Watermarking Based on Echo Hiding: Time-Spread Echo Hiding*, pages 123–151. 01 2007.

[5] Kadir Tekeli. Audio steganography algorithms. `https://github.com/ktekeli/audio-steganography-algorithms`.

[6] Lihao Yang Tianruo Sun and Zimo Cheng. Mp3 audio watermarking. `http://www2.ece.rochester.edu/~zduan/teaching/ece472/projects/2020/TianruoSun_LihaoYang_ZimoCheng_MP3Watermarking_ProjectReport.pdf`.