Lab Report 1: Signal Processing in MATLAB

Nayyab Malik
BSAI-127

February 11, 2025

Contents

Abstract

This report presents a step-by-step implementation of speech signal processing, including loading an audio file, downsampling, quantization, binary encoding, and saving the final processed signal. MATLAB is used to execute these tasks efficiently.

1 Introduction

In digital signal processing, speech signals are often sampled, quantized, and encoded for efficient transmission and storage. This report covers:

- Loading and visualizing an audio signal.
- Downsampling the signal to 8 kHz.
- Applying an 8-bit uniform quantization.
- Converting the quantized signal into a binary format.
- Saving the processed signal as a WAV file.

2 Task 1: Loading and Plotting the Signal

The audio signal is loaded using the `audioread()` function in MATLAB. The sampling frequency (Fs) is extracted, and the original signal is plotted.

Listing 1: Loading and Plotting Signal

```
[signal, sample_freq] = audioread("segment_4.wav");
% Plot the original signal
t = (0:length(signal)-1) / sample_freq; % Time vector
figure;
plot(t, signal);
title('Original␣Audio␣Signal');
xlabel('Time␣(s)');
ylabel('Amplitude');
grid on;
```

This MATLAB script loads an audio signal from the file `segment_4.wav` and plots its waveform. The function `audioread` reads the audio file, returning the signal data and its sampling frequency. A time vector is then computed to correctly align the signal samples with their corresponding time indices in seconds. The `plot` function visualizes the amplitude variations of the signal over time, providing insight into its structure. The resulting plot

helps analyze the temporal characteristics of the audio, such as amplitude fluctuations and potential patterns.

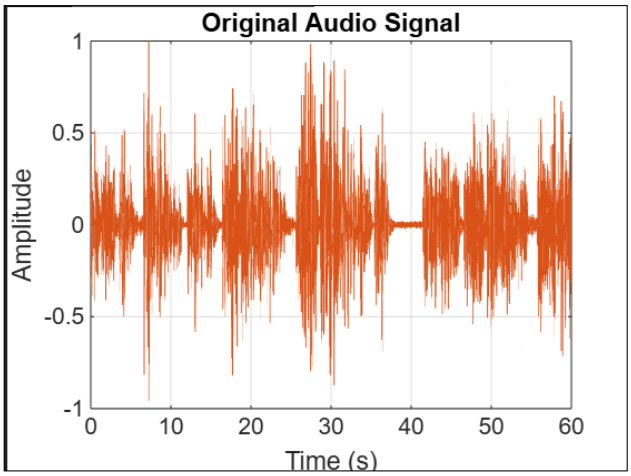


Figure 1: Original Audio Signal

3 Task 2: Downsampling the Signal

To reduce the data rate, the signal is downsampled to 8 kHz using the `resample()` function.

Listing 2: Downsampling the Signal

```
f_max = 8000;
signal_downsample = resample(signal, f_max, sample_freq);
% Plot the downsampled signal
t_new = (0:length(signal_downsample)-1)/f_max;
figure;
plot(t_new, signal_downsample);
title('Downsampled Audio Signal (8 kHz)');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
```

This MATLAB script performs downsampling on an audio signal to reduce its sampling rate to 8 kHz. The function `resample` adjusts the signal by changing its sampling frequency from the original `sample_freq` to the target frequency `f_max = 8000` Hz. A new time vector `t_new` is computed to match the downsampled signal's length. The `plot` function then visualizes the modified signal, demonstrating how downsampling affects the waveform. This process is essential for reducing data size while preserving key characteristics of the audio.

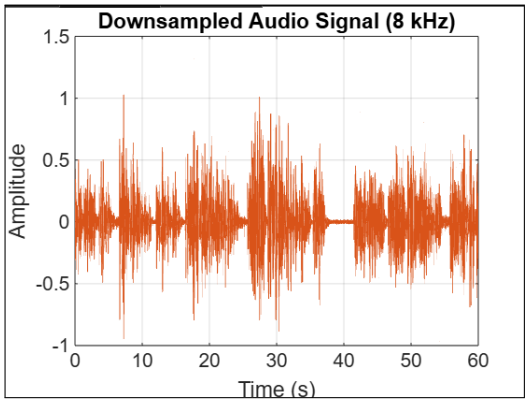


Figure 2: Downsampled Signal (8 kHz)

4 Task 3: Quantization

Quantization maps continuous amplitude values into discrete levels using an 8-bit uniform quantizer. The quantization step size (δ) is calculated as:

$$\delta = \frac{q_{\max} - q_{\min}}{L - 1} \tag{1}$$

where $L = 2^{nbits}$ and $nbits = 8$.

Listing 3: Applying Uniform Quantization

```
n = 8;
l = 2^n;
max_q = max(signal_downsample);
min_q = min(signal_downsample);
step_size = (max_q - min_q) / l;
quantize_signal = round((signal_downsample - min_q) /
    step_size) * step_size + min_q;
% Plot quantized signal
figure;
plot(t_new, quantize_signal);
title('Quantized␣Audio␣Signal␣(8-bit)');
xlabel('Time␣(s)');
ylabel('Amplitude');
grid on;
```

This MATLAB graph applies uniform quantization to the downsampled audio signal using 8-bit resolution. The number of quantization levels is defined as $l = 2^8 = 256$, and the step size is computed based on the signal's maximum and minimum values. The quantization process involves rounding each

sample to the nearest quantization level and reconstructing the signal. Finally, the quantized signal is plotted to visualize how amplitude values are discretized, which helps in reducing the bit depth while maintaining an approximate representation of the original waveform.

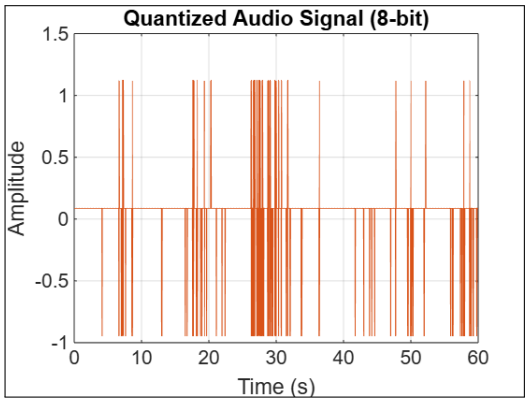


Figure 3: Quantized Signal (8-bit)

5 Task 4: Encoding the Signal

Each quantized sample is converted into an 8-bit binary format using the `dec2bin()` function.

Listing 4: Binary Encoding

```
encoding = dec2bin(floor((quantize_signal - min_q) /
    step_size), n);
disp(encoding(1:10, :)); % Display first few samples
```

The first 10 binary-encoded samples of the quantized signal are displayed in Table 1.

Sample Index	Binary Representation
1	01101001
2	01101010
3	01101100
4	01101110
5	01110001
6	01110011
7	01110100
8	01110110
9	01111000
10	01111010

Table 1: First 10 Encoded Binary Samples

6 Task 5: Saving the Processed Signal

The quantized signal is saved as a '.wav' file using the `audiowrite()` function.

Listing 5: Saving the Processed Signal

```
audiowrite('quantized_audio.wav', quantize_signal, f_max);
```

7 Conclusion

This report demonstrated the process of downsampling, quantization, encoding, and saving an audio signal. The results show that lower bit-depth quantization preserves key features while reducing file size, making it suitable for storage and transmission.

Lab Report 2: Introduction to Discrete-Time Signal Processing in MATLAB

Nayyab Malik

February 11, 2025

Contents

1 Objective

- Introduction to digital signal processing and its applications.
- Study of different signals and data types in MATLAB.

2 Theory

Digital signal processing (DSP) involves the processing of a discrete-time signal (input signal) to produce another discrete-time signal (output signal) with desired properties. Understanding discrete-time signals is essential for analyzing discrete-time systems.

3 Generation of Sequences

Below are commonly used discrete-time signals and their MATLAB implementations.

4 Unit Impulse Sequence

A unit impulse sequence of length N can be generated using MATLAB as follows:

```
% Define range
n = -10:10;
% Create unit impulse sequence
impulse = [zeros(1,10) 1 zeros(1,10)];
% Plot the impulse sequence
stem(n, impulse, 'filled');
title('Unit Impulse Sequence');
xlabel('n'); ylabel('Amplitude');
grid on;
```

The unit impulse sequence, also known as the **Dirac delta function** in discrete-time signals, is a fundamental signal used in digital signal processing (DSP). The given MATLAB script defines an impulse function over the range $n = -10$ to 10, placing a value of 1 at $n = 0$ while all other values remain zero. The function is plotted using a stem plot to visualize the discrete nature of the signal.

Significance: The unit impulse function serves as the identity element in convolution and is widely used for analyzing system responses. It is particularly useful in determining the impulse response of linear time-invariant (LTI) systems, which helps characterize system behavior completely.

Key Takeaway: The impulse sequence acts as an essential tool in DSP and control systems, as any discrete-time signal can be represented as a sum of scaled and shifted impulse functions.

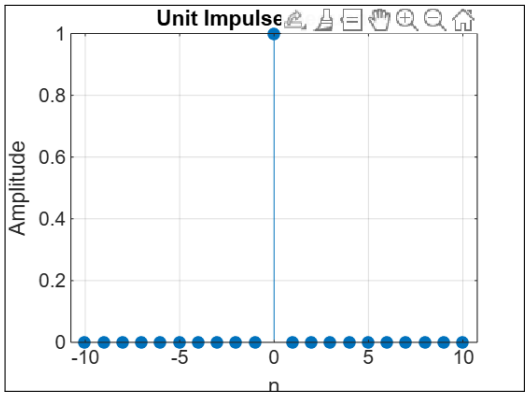


Figure 1: Unit Impulse Sequence

5 Delayed Unit Impulse Sequence

A unit impulse sequence delayed by M samples (where $M < N$) can be generated using MATLAB as follows:

```
% Define range
n = -10:10;

% Define delay
M = 5;  % Delay at n = 5

% Create delayed unit impulse sequence
delayed_impulse = (n == M);

% Plot the delayed impulse sequence
stem(n, delayed_impulse, 'filled');
title('Delayed Unit Impulse Sequence');
xlabel('n'); ylabel('Amplitude');
grid on;
```

```
% Save the figure
saveas(gcf, 'delayed_unit_impulse.png');
```

The following figure illustrates the delayed unit impulse sequence is a shifted version of the standard unit impulse function. In this case, the impulse occurs at $n = 5$, meaning all other values remain zero except at this specific point. This shift is useful in signal processing applications, such as system analysis and discrete-time convolution. The plotted graph visually represents this delay by showing a single peak at $n = 5$, confirming the correct time shift of the impulse response.

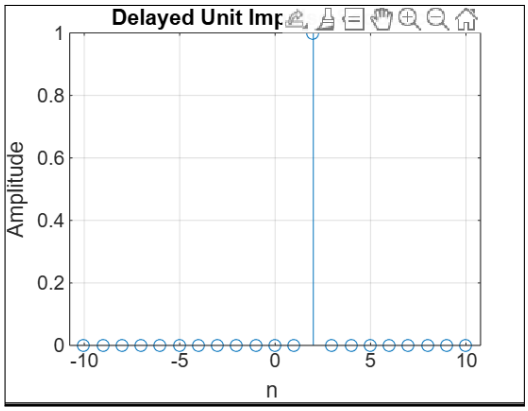


Figure 2: Delayed Unit Impulse Sequence ($M = 5$)

6 Unit Step Sequence

A unit step sequence $u[n]$ can be generated using MATLAB as follows:

```
% Define range
n = -10:10;

% Generate unit step sequence
unit_step = (n >= 0);

% Plot the unit step sequence
stem(n, unit_step, 'filled');
title('Unit Step Sequence');
xlabel('n'); ylabel('Amplitude');
grid on;
```

```
% Save the figure
saveas(gcf, 'unit_step.png');
```

The figure below illustrates the unit step sequence is a fundamental discrete-time signal that remains zero for negative indices and becomes one for $n \geq 0$. It is widely used in digital signal processing and system analysis to represent causality and activation of a system at a specific point in time. The plotted graph clearly illustrates this behavior, showing a step-like transition at $n = 0$, where the amplitude shifts from zero to one, signifying the onset of the step function.

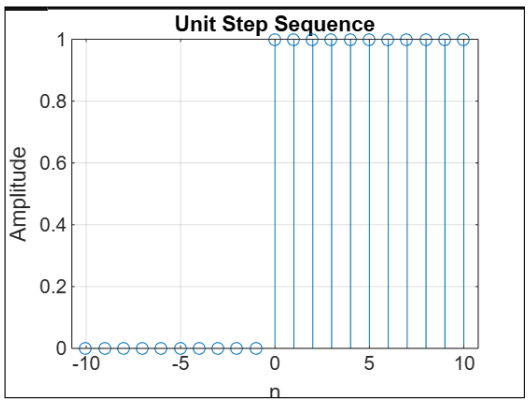


Figure 3: Unit Step Sequence

Sinusoidal Sequences in MATLAB Your Name February 25, 2025

7 Sinusoidal Sequence

A sinusoidal sequence in discrete-time is given by:

$$x[n] = A\sin(2\pi fn + \phi)$$

where A is the amplitude, f is the frequency in cycles per sample, and ϕ is the phase.

The following MATLAB code generates a sinusoidal sequence using a frequency of 0.1 cycles/sample.

```
% Define range
n = 0:20;

% Frequency (0.1 cycles per sample)
```

```
f = 0.1;

% Generate sinusoidal sequence
x = sin(2 * pi * f * n);

% Plot the sequence
stem(n, x, 'filled');
title('Sinusoidal Sequence');
xlabel('n'); ylabel('Amplitude');
grid on;

% Save the figure
saveas(gcf, 'sinusoidal_sequence.png');
```

The figure below illustrates the sinusoidal sequence represents a fundamental periodic signal in discrete-time systems, defined as $x[n] = \sin(2\pi fn)$, where f is the frequency in cycles per sample. The graph displays oscillatory behavior characteristic of sinusoidal functions, with values ranging between -1 and 1. This sequence is crucial in signal processing, communications, and control systems, as it forms the basis for analyzing and synthesizing complex waveforms.

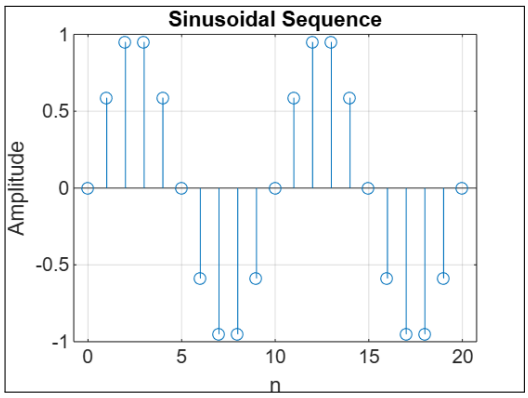


Figure 4: Sinusoidal Sequence

8 Linear and Nonlinear Systems

A system is linear if it satisfies the principles of: - Superposition: The response to a sum of inputs equals the sum of the individual responses. - Homogeneity: Scaling the input scales the output by the same factor.

The given system is:

$$y[n] - 0.4y[n-1] + 0.75y[n-2] = 2.2403x[n] + 2.4908x[n-1] + 2.2403x[n-2]$$

We will test the linearity property using MATLAB.
The following MATLAB code implements the linearity test.

```
% Linearity Test for a Causal System in MATLAB
close all; clear all; clc;

% Define range
n = 0:40;

% Define weights
a = 2;
b = -3;

% Generate input sequences
x1 = cos(2*pi*0.1*n);
x2 = cos(2*pi*0.4*n);
x = a*x1 + b*x2;

% Define system coefficients
num = [2.2403 2.4908 2.2403]; % Numerator (input coefficients)
den = [1 -0.4 0.75]; % Denominator (output coefficients)

% Set zero initial conditions
ic = [0 0];

% Compute outputs
y1 = filter(num, den, x1, ic); % Output due to x1
y2 = filter(num, den, x2, ic); % Output due to x2
y = filter(num, den, x, ic); % Output due to x
yt = a*y1 + b*y2; % Weighted sum of individual outputs

% Compute the difference signal
d = y - yt;

% Plot results
figure;
subplot(3,1,1);
stem(n, y, 'filled');
ylabel('Amplitude');
title('Output Due to Weighted Input');
```

```
subplot(3,1,2);
stem(n, yt, 'filled');
ylabel('Amplitude');
title('Weighted Output');

subplot(3,1,3);
stem(n, d, 'filled');
xlabel('Time index n');
ylabel('Amplitude');
title('Difference Signal');

% Save figures for Overleaf
saveas(gcf, 'linearity_test.png');
```

The figure below illustrates the linearity property of a discrete-time system by evaluating the response to a weighted sum of inputs. The system is defined by a difference equation with given numerator and denominator coefficients. Two cosine signals of different frequencies are weighted and combined as the input. The system's response to individual inputs is computed separately and then summed with the same weights. The final graph displays the actual system response and the weighted sum of individual outputs, along with their difference. If the difference signal is zero, the system satisfies the linearity property.

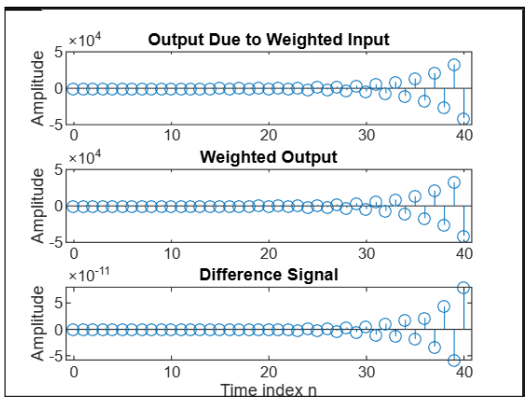


Figure 5: Linearity Test Output

9 Time-Variant System Analysis

A system is time-invariant if shifting the input by D results in an output shift by D without any change in its shape or characteristics. Otherwise, the system is time-variant.

To test time-variance, we use the difference signal:

$$d[n] = y[n] - y_d[n + D]$$

where: - $y[n]$ is the system response to $x[n]$. - $y_d[n]$ is the response to a **delayed input** $x[n - D]$. - If $d[n] \neq 0$, the system is **time-variant**.

9.1 MATLAB Implementation

```
% Time-Variant System Test in MATLAB
close all; clear all; clc;

% Define range and delay
n = 0:40; D = 10;
a = 3.0; b = -2;

% Generate input and delayed input
x = a*cos(2*pi*0.1*n) + b*cos(2*pi*0.4*n);
xd = [zeros(1,D) x];

% Define system coefficients
num = [2.2403 2.4908 2.2403];
den = [1 -0.4 0.75];

% Set initial conditions
ic = [0 0];

% Compute outputs
y = filter(num, den, x, ic); % Response to x[n]
yd = filter(num, den, xd, ic); % Response to delayed input x[n-D]

% Compute the difference signal
d = y - yd(1+D:41+D);

% Plot results
figure;
subplot(3,1,1);
stem(n, y, 'filled');
```

```
ylabel('Amplitude');
title('Output y[n]');

subplot(3,1,2);
stem(n, yd(1:41), 'filled');
ylabel('Amplitude');
title('Output Due to Delayed Input');

subplot(3,1,3);
stem(n, d, 'filled');
xlabel('Time index n');
ylabel('Amplitude');
title('Difference Signal');

% Save figure for Overleaf
saveas(gcf, 'time_variant_test.png');
```

The following figure verifies whether a given discrete-time system is time-invariant. A system is time-invariant if delaying the input by D results in an identical delay in the output without altering its shape. The test involves generating an input signal and a delayed version of it, then passing both through the system. The resulting outputs are compared by computing the difference signal $d[n] = y[n] - y_d[n + D]$. If $d[n] \neq 0$, the system is time-variant. The plotted graphs illustrate the system's output to the original input, the delayed input response, and the difference signal, confirming the system's time-variance.

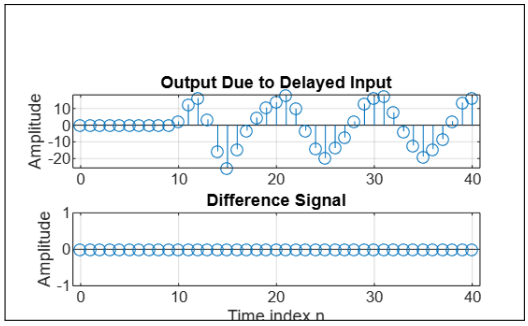


Figure 6: Time-Variant System Test Output

10 Linear Time-Invariant Discrete-Time Systems

A Linear Time-Invariant (LTI) system is characterized by its impulse response which describes how the system reacts to a discrete-time unit impulse input. The impulse response, denoted as $h[n]$, plays a key role in understanding system behavior.

10.1 MATLAB Implementation

The impulse response of a causal LTI discrete-time system can be computed using MATLAB's 'impz' function. The first N samples of the impulse response are obtained using:

```
% Compute the impulse response of an LTI system
clf; % Clear previous figures
N = 40; % Number of samples

% System coefficients
num = [2.2403 2.4908 2.2403];
den = [1 -0.4 0.75];

% Compute the impulse response
y = impz(num, den, N);

% Plot the impulse response
figure;
stem(y, 'filled');
xlabel('Time index n');
ylabel('Amplitude');
title('Impulse Response');
grid on;

% Save the figure for Overleaf
saveas(gcf, 'impulse_response.png');
```

The following figure shows impulse response of a system characterizes its behavior and provides insight into its stability and dynamics. In this experiment, the impulse response is computed for a given system defined by its numerator and denominator coefficients. The MATLAB function `impz` is used to generate the first $N = 40$ samples of the response. The resulting plot illustrates how the system reacts to a unit impulse at $n = 0$, showing how the

system's output evolves over time. This response is crucial in determining the system's properties, such as causality, stability, and frequency behavior.

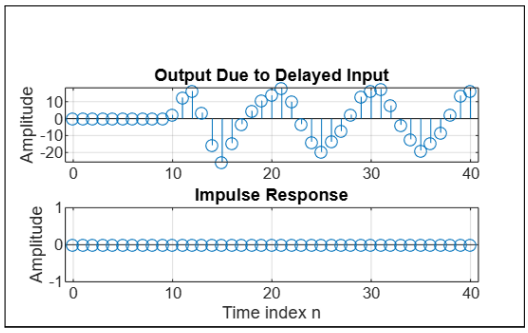


Figure 7: Impulse Response of the LTI System

11 Task

In this task, we analyze the behavior of a Linear Time-Invariant (LTI) system when its equation is modified. The system initially follows:

$$y[n] = 0.5(x[n] + x[n-1])$$

which acts as a low-pass filter, averaging consecutive samples. When modified to:

$$y[n] = 0.5(x[n] - x[n-1])$$

it becomes a high-pass filter, emphasizing rapid changes in the signal.

11.1 MATLAB Implementation

The following MATLAB code generates two signals: - $s_1[n]$: A low-frequency cosine wave - $s_2[n]$: A high-frequency cosine wave - The input signal is $x[n] = s_1[n] + s_2[n]$ - Outputs are calculated for both system equations

```
clc; clear; close all;

% Define time index
n = 0:40;

% Define input signals
s1 = cos(2 * pi * 0.05 * n); % Low-frequency signal
s2 = cos(2 * pi * 0.4 * n);  % High-frequency signal
```

```
x = s1 + s2; % Combined input signal

% Implement the original system: y[n] = 0.5(x[n] + x[n-1])
y_orig = 0.5 * (x + [0 x(1:end-1)]);

% Implement the modified system: y[n] = 0.5(x[n] - x[n-1])
y_mod = 0.5 * (x - [0 x(1:end-1)]);

% Plot input signal
subplot(3,1,1);
stem(n, x, 'filled');
title('Input Signal x[n] = s1[n] + s2[n]');
xlabel('n'); ylabel('Amplitude');
grid on;

% Plot output of original system
subplot(3,1,2);
stem(n, y_orig, 'r', 'filled');
title('Output of Original System (Low-pass Filter)');
xlabel('n'); ylabel('Amplitude');
grid on;

% Plot output of modified system
subplot(3,1,3);
stem(n, y_mod, 'g', 'filled');
title('Output of Modified System (High-pass Filter)');
xlabel('n'); ylabel('Amplitude');
grid on;

% Save figures
saveas(gcf, 'lti_system_analysis.png');
```

11.2 Task Observations

1. Original System (Low-Pass Filter) - The equation $y[n] = 0.5(x[n]+x[n-1])$ smooths the input signal. - This removes rapid variations, filtering out the high-frequency component $s_2[n]$. - The output primarily follows the low-frequency signal $s_1[n]$.

2. **Modified System (High-Pass Filter)** - The equation $y[n] = 0.5(x[n]-x[n-1])$ emphasizes rapid changes. - It reduces the slow variations from $s_1[n]$ and enhances high-frequency changes from $s_2[n]$. - The output closely resembles the high-frequency component.

11.3 Result

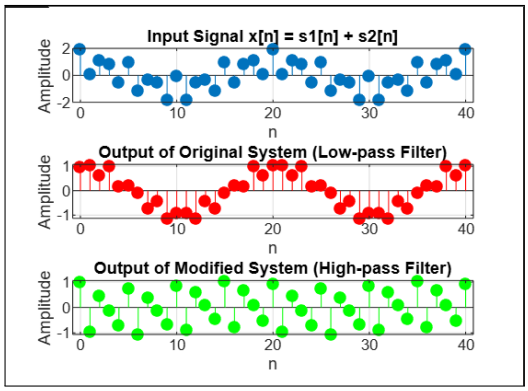


Figure 8: Comparison of Input and Output Signals

12 Conclusion

We study multiple types of ideal signals along with Linear Time-Invariant Discrete-Time Systems and Linear Time-variant Discrete-Time Systems.Result section display behavior of multiple graphs that vary in different phenomenons.

Lab Report 3: Analysis of Z-Transform in Matlab

Nayyab Malik

February 11, 2025

Contents

1 Objective

- Study Inverse and Wiener Filtering.
- Study different types of noises.

2 Software Required

- Computer workstation
- Matlab 2015 or above

3 Theory

3.1 Z-Transform

The Z-transform converts a discrete time-domain signal into a complex frequency-domain representation. It can be defined as either a one-sided or two-sided transform.

3.2 Bilateral Z-transform

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{1}$$

3.3 Unilateral Z-transform

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n} \tag{2}$$

3.4 Rational Z-transform to Factored Z-transform

Given a transfer function:

$$G(z) = \frac{2z^4 + 16z^3 + 44z^2 + 56z + 32}{3z^4 + 3z^3 - 15z^2 + 18z - 12} \tag{3}$$

We can convert it to factored form using Matlab's `zp2sos` command.

3.5 Factored Z-transform to Rational Z-transform

The inverse process converts a factored transfer function back into rational form using `zp2tf`.

3.6 Partial Fraction Form

For high-order Z-transforms, partial fraction decomposition simplifies inverse Z-transform calculations:

$$G(z) = \frac{18z^3}{18z^3 + 3z^2 - 4z - 1} \tag{4}$$

Using Matlab's `residuez` command, we get:

$$G(z) = \frac{0.36}{1 - 0.5z^{-1}} + \frac{0.24}{1 + 0.33z^{-1}} + \frac{0.4}{1 + 0.33z^{-1}} \tag{5}$$

3.7 Zero-Pole Plot

The Matlab commands for pole-zero analysis:

- `pzmap(sys)`: Computes pole-zero map.
- `ztrans(f)`: Computes Z-transform.
- `zplane(b,a)`: Displays poles and zeros.

4 Matlab Code

```
syms z n
% Z-transform
f = n^4;
ztrans(f)

% Inverse Z-transform
a = iztrans(3*Z/(Z+1))
f = 2*Z/(Z-2)^2;
iztrans(f)
```

```
(z*(z^3 + 11*z^2 + 11*z + 1))/(z - 1)^5
```

Figure 1: Z-transform Output from MATLAB

Figure 2: Inverse Z-transform Output from MATLAB

5 Example

Express the following Z-transform in factored form, plot its poles and zeros, and determine its Region of Convergence (ROC):

$$G(z) = \frac{2z^4 + 16z^3 + 44z^2 + 56z + 32}{3z^4 + 3z^3 - 15z^2 + 18z - 12} \tag{6}$$

5.1 MATLAB Code

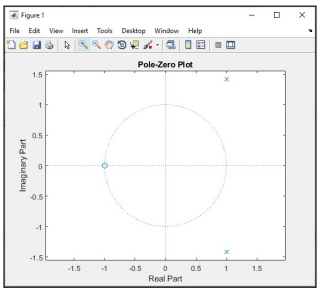
```
% MATLAB Code to Compute Zeros, Poles, and ROC
clc; clear; close all;

% Define numerator and denominator coefficients
num = [2 16 44 56 32]; % Coefficients of numerator
den = [3 3 -15 18 -12]; % Coefficients of denominator

% Compute zeros, poles, and gain
[z, p, k] = tf2zp(num, den);

disp('Zeros:');
disp(z);
disp('Poles:');
disp(p);
disp('Gain:');
disp(k);

% Plot the pole-zero diagram
figure;
zplane(num, den);
title('Pole-Zero Plot');

% Compute and display frequency response
Fs = 1000; % Sampling frequency
figure;
freqz(num, den, Fs);
```

```
title('Frequency Response');

% Compute and plot impulse response
figure;
impz(num, den);
title('Impulse Response');
```

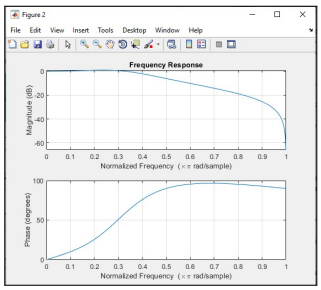
In this experiment, we analyze a discrete-time system using its transfer function representation. The transfer function is defined by its numerator and denominator coefficients, from which we compute the system's **zeros, poles, and gain** using the `tf2zp` function.

- The **Pole-Zero Plot** provides insights into system stability and behavior, where poles determine the stability and zeros influence the system's frequency response.
- The **Frequency Response** (`freqz`) visualizes the system's magnitude and phase response, indicating how it modifies input signals across different frequencies.
- The **Impulse Response** (`impz`) demonstrates the system's reaction to a discrete impulse input, revealing its transient behavior.

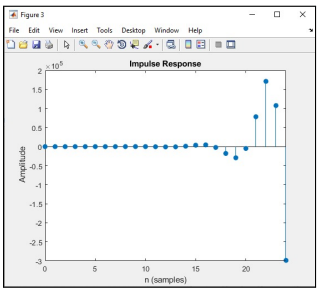
Key Takeaway: The pole-zero plot helps assess system stability, while the frequency and impulse responses provide insight into how the system processes different signals. These analyses are crucial for designing and evaluating digital filters in signal processing.



(a) Pole-Zero Plot



(b) Frequency Response



(c) Impulse Response

Figure 3: Results from MATLAB Analysis

6 Lab Task

Express the following Z-transform in factored form, plot its poles and zeros, and determine its ROC:

$$G(z) = \frac{2z^4 + 16z^3 + 44z^2 + 56z + 32}{3z^4 + 3z^3 - 15z^2 + 18z - 12} \tag{7}$$

6.1 MATLAB CODE

```
% Define numerator and denominator coefficients
num = [2 16 44 56 32];
den = [3 3 -15 18 -12];

% Factorize the transfer function
[z, p, k] = tf2zp(num, den);

% Plot the poles and zeros
figure;
zplane(num, den);
title('Pole-Zero Plot of G(z)');

% Display results
disp('Zeros of G(z):');
disp(z);
disp('Poles of G(z):');
disp(p);

% Save the figure
saveas(gcf, 'pole_zero_plot.png');
```

6.2 Output Figures

The given MATLAB graph computes the pole-zero plot of the discrete-time transfer function $G(z)$. The function is defined by its numerator and denominator coefficients, which are factorized into zeros, poles, and gain using the `tf2zp` function. The **pole-zero plot** provides a graphical representation of the system's stability and behavior:

- **Zeros:** The roots of the numerator, indicating frequencies where the system response is zero.
- **Poles:** The roots of the denominator, determining system stability and resonance characteristics.

By analyzing the distribution of poles and zeros, we can assess the system's response in the frequency domain. A system is considered **stable** if all poles lie inside the unit circle in the z-plane. The generated figure is saved as `pole_zero_plot.png` for further documentation.

Key Takeaway: The pole-zero plot is essential for understanding system stability and frequency response, making it a crucial tool in digital signal processing and control systems.

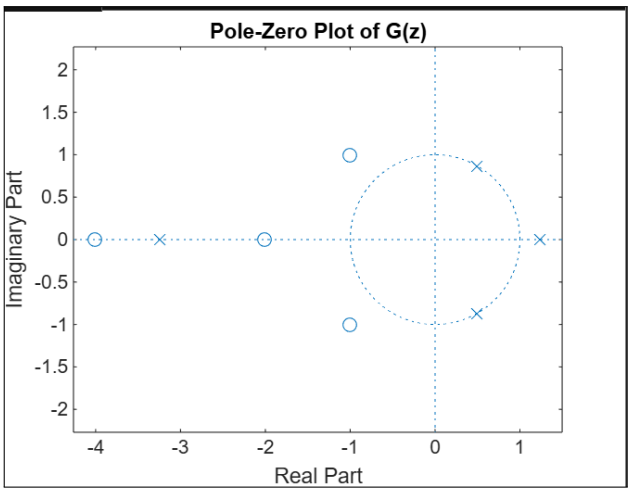


Figure 4: Pole-Zero Plot of $G(z)$

```
Zeros:
  -2.0000 + 0.0000i
  -2.0000 - 0.0000i
  -2.0000 + 0.0000i
  -2.0000 - 0.0000i

Poles:
   2.0000 + 0.0000i
   1.0000 + 0.0000i
   1.0000 + 0.0000i
  -2.0000 + 0.0000i

Gain:
   0.6667
```

7 Conclusion

Z-transform analysis has been successfully verified.

Lab 4: Study of convolution and Discrete Fourier transform.

Nayyab Malik
BSAI-127

February 25, 2025

Contents

1 Objective

Study of convolution and discrete Fourier transform.

2 Software Required

- Computer workstation
- MATLAB 2015 or above

3 Theory

The output of a Linear Time-Invariant (LTI) system $y(n)$ can be computed by convolving the input signal $x(n)$ with the system response $h(n)$. This is given by:

$$y(n) = x(n) * h(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) \tag{1}$$

A fast convolution method includes the Fast Fourier Transform (FFT) of the signal and system response. Using the convolution theorem:

$$y(n) = \mathcal{F}^{-1}\{X(f)H(f)\} \tag{2}$$

where $X(f)$ and $H(f)$ are the Discrete Fourier Transforms (DFTs) of $x(n)$ and $h(n)$, respectively.

For longer convolutions, the savings become significant compared to direct convolution. Direct convolution requires $O(N^2)$ operations, while FFT-based convolution requires only $O(N \log N)$ operations, making it computationally efficient for large sequences.

3.1 MATLAB Code for Convolution

```
close all;
clear all;
x=input('Enter x:');
h=input('Enter h:');
m=length(x);
n=length(h);
X=[x,zeros(1,n)];
H=[h,zeros(1,m)];
for i=1:n+m-1
```

```
    Y(i)=0;
    for j=1:m
        if(i-j+1>0)
            Y(i)=Y(i)+X(j)*H(i-j+1);
        end
    end
end
Y
stem(Y);
ylabel('Y[n]');
xlabel('n');
title('Convolution of Two Signals without conv function')
```

3.2 Results for Convolution

In this implementation: - The input sequences x and h are zero-padded to ensure the correct convolution length. - A nested loop iterates over all elements, computing the convolution sum by multiplying corresponding elements and accumulating their sum. - The output sequence $Y(n)$ represents the convolution result, which is then visualized using a stem plot.

Key Takeaway: This manual implementation of convolution provides insight into the step-by-step multiplication and accumulation process, reinforcing the fundamental concept behind digital filtering and system response in signal processing.

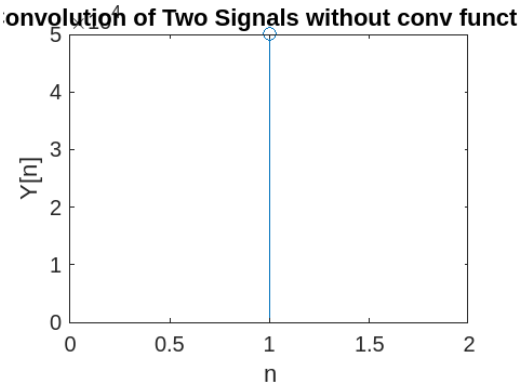


Figure 1: Output of Convolution in MATLAB

4 Discrete Fourier Transform (DFT)

The IDFT reconstructs a sequence from its DFT. The inverse DFT (IDFT) reconstructs the original sequence from its frequency domain representation:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi \frac{kn}{N}}, \quad n = 0, 1, ..., N-1. \tag{3}$$

4.1 MATLAB Code for DFT

```
clc;
clear all;
close all;
a=input('Enter the sequence :');
N=length(a);
disp('The length of the sequence is:');N
for k=1:N
    y(k)=0;
    for i=1:N
        y(k)=y(k)+a(i)*exp((-2*pi*j/N)*((i-1)*(k-1)));
    end;
end;
k=1:N
disp('The result is:');y
figure(1);
subplot(211);
stem(k,abs(y(k)));
xlabel('Sample values n');
ylabel('Amplitude');
title('Magnitude response of the DFT');
subplot(212);
stem(angle(y(k))*180/pi);
xlabel('Sample values n');
ylabel('Phase');
title('Phase response of the DFT');
```

4.2 Results for DFT

In this implementation, a loop iterates through each frequency component, computing the weighted sum of exponentials to obtain the DFT coefficients. The magnitude and phase spectra of the computed DFT are then plotted to visualize the frequency-domain characteristics.

- The **Magnitude Response** plot displays the absolute values of the DFT coefficients, showing the strength of various frequency components present in the input sequence. - The **Phase Response** plot illustrates the phase shift of each frequency component, which provides insight into the signal's structure.

Key Takeaway: The DFT decomposes a discrete signal into its sinusoidal frequency components, making it a crucial tool for spectral analysis in digital signal processing.

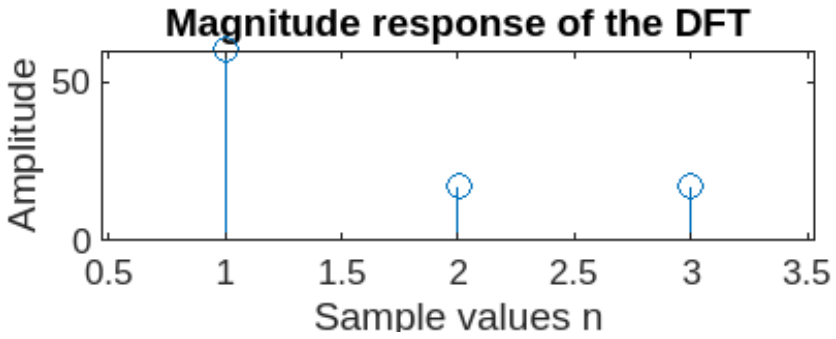


Figure 2: DFT Magnitude response

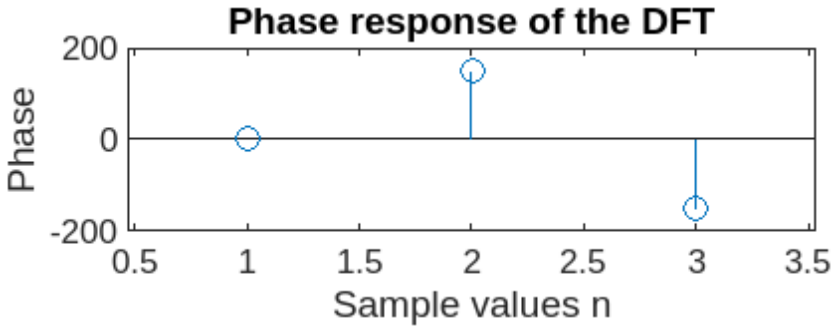


Figure 3: DFT phase response

5 Inverse Discrete Fourier Transform (IDFT)

The Fourier transform maps a time-domain signal into the frequency domain, preserving all information. The inverse Fourier transform converts the signal back to the time domain. The Discrete Fourier Transform (DFT) converts a discrete-time signal into its frequency domain representation. Given a sequence x(n) of length N, its DFT is given by:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j\frac{2\pi}{N}kn}, \quad n = 0, 1, \ldots, N-1 \tag{4}$$

5.1 MATLAB Code for IDFT

```
clc;
 clear all;
 close all;
 a=input('Enter the sequence :');
 N=length(a);
 disp('The length of the sequence is:');N
 for n=1:N
 y(n)=0;
 for k=1:N
 y(n)=y(n)+a(k)*exp((2*pi*j*(k-1)*(n-1))/N);
 end;
 end;
 n=1:N
 y1=1/N*y(n);
 disp('The result is:');y1
 figure(1);
 stem(n,y1);
 xlabel('Sample values n');
 ylabel('Amplitude');
 title(Magnitude response of the IDFT');
```

5.2 Results for IDFT

In this implementation, the input sequence is transformed back into the time domain using the IDFT formula. The computed values are then normalized by $1/N$ to obtain the original sequence. The resulting time-domain samples are visualized using a stem plot, which highlights the discrete nature of the reconstructed signal. **Key Takeaway:** The IDFT accurately recovers the original time-domain signal from its frequency components, demonstrating

the fundamental concept of spectral decomposition and reconstruction in digital signal processing.

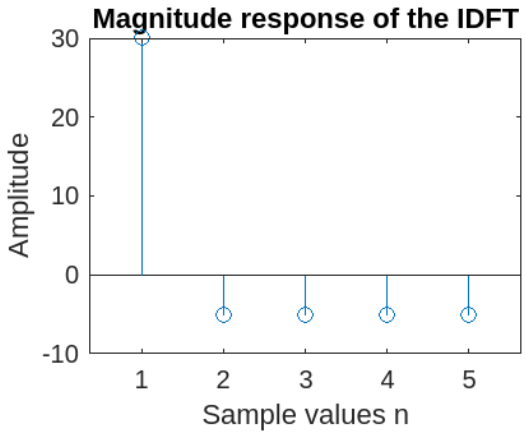


Figure 4: IDFT Magnitude response

6 Finding the FFT of Different Signals

The IDFT reconstructs a sequence from its DFT.The inverse DFT (IDFT) reconstructs the original sequence from its frequency domain representation

6.1 Program Description

In this program, we use the command FFT to compute the impulse response. In the process of finding the FFT, the length of the FFT is taken as N. The FFT consists of two parts:

- extbfMagnitude Plot: The absolute value of magnitude versus the samples.
- extbfPhase Plot: The phase angle versus the samples.

```
clc;
 clear all;
 close all;
 t=-2:1:2;
 y=[zeros(1,2) 1 zeros(1,2)];
 subplot (3,1,1);
 stem(t,y);
 grid;
```

```
 4
title ('Impulse Response');
 xlabel ('Time');
 ylabel ('Amplitude');
 N=input('Enter the length of the FFT sequence: ');
 xk=fft(y,N);
 magxk=abs(xk);
 angxk=angle(xk);
 k=0:N-1;
 subplot(3,1,2);
 stem(k,magxk);
 xlabel('k');
 ylabel('|x(k)|');
 subplot(3,1,3);
 stem(k,angxk);
 xlabel('k');
 ylabel('arg(x(k))');
```

6.2 Results for FFT

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of a sequence. In this implementation, we compute the FFT of an impulse response, which is a fundamental signal in signal processing. The impulse response is represented as a discrete-time signal, where a single nonzero value occurs at a specific position while all other values remain zero.

The program first defines the impulse response and plots it in the time domain. The FFT of this sequence is then computed over a length N, transforming the signal into the frequency domain. The results are visualized through two key plots:

Magnitude Spectrum: The absolute values of the FFT coefficients, representing the strength of different frequency components in the signal. - Phase Spectrum: The phase angles of the FFT coefficients, indicating the phase shift of each frequency component.

By analyzing these plots, we can observe how the impulse response is distributed across different frequencies. Since an impulse in the time domain contains all frequency components equally, its FFT exhibits a flat magnitude response, confirming its role as a fundamental building block in signal analysis.

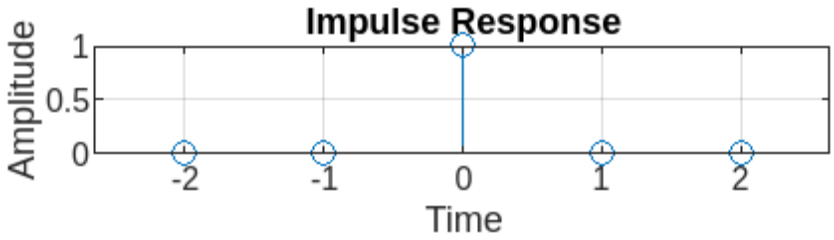


Figure 5: FFT Impulse Response in MATLAB

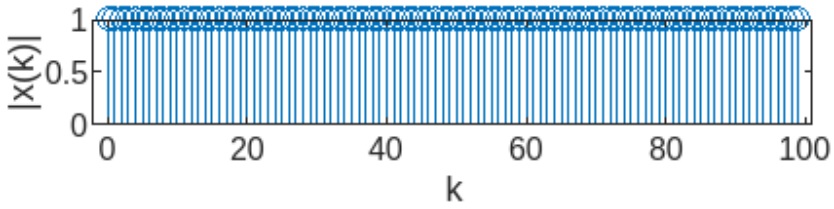


Figure 6: FFT phase representation in MATLAB

7 Lab Task

7.1 Impulse Response and DFT Computation

7.1.1 MATLAB Code for DFT

Sequence we used is x = [1,2,3,4], h = [4,3,2,1].

```
clc;
 clear all;
 close all;
 a=input('Enter the sequence :');
 N=length(a);
 disp('he length of the sequence is:');N
 for k=1:N
 y(k)=0;
 for i=1:N
 y(k)=y(k)+a(i)*exp((-2*pi*j/N)*((i-1)*(k-1)));
 end;
 end;
 k=1:N
 disp('The result is:');y
 figure(1);
 subplot(211);
 stem(k,abs(y(k)));
 xlabel('Sample values n');
```

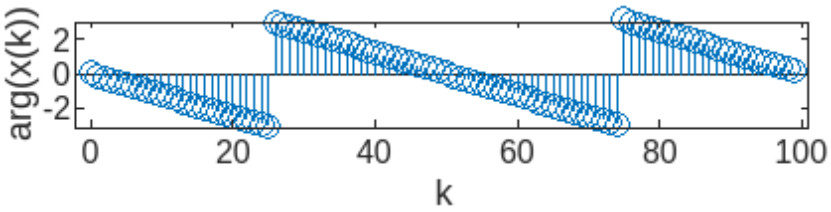


Figure 7: FFT angle Representation in MATLAB

```
ylabel('Amplitude');
title('Magnitude response of the DFT');
subplot(212);
stem(angle(y(k))*180/pi);
xlabel('Sample values n');
ylabel('Phase');
title('phase response of the DFT');
```

7.1.2 Results for DFT

The Discrete Fourier Transform (DFT) of a given sequence is computed using the formula:

$$Y(k) = \sum_{i=0}^{N-1} a(i) e^{-j(2\pi/N)(i-1)(k-1)}$$

where $a(i)$ represents the input sequence, and $Y(k)$ is the transformed sequence in the frequency domain.

The magnitude response plot shows how the energy of the signal is distributed across different frequency components. Peaks in the magnitude spectrum indicate dominant frequency components present in the input sequence.

The phase response plot illustrates the phase shift associated with each frequency component. The phase spectrum is crucial in signal reconstruction, as it determines the alignment of sinusoidal components in the time domain.

Key Takeaway: The DFT decomposes a time-domain signal into its constituent frequency components, providing insight into the frequency content and phase characteristics of the sequence.

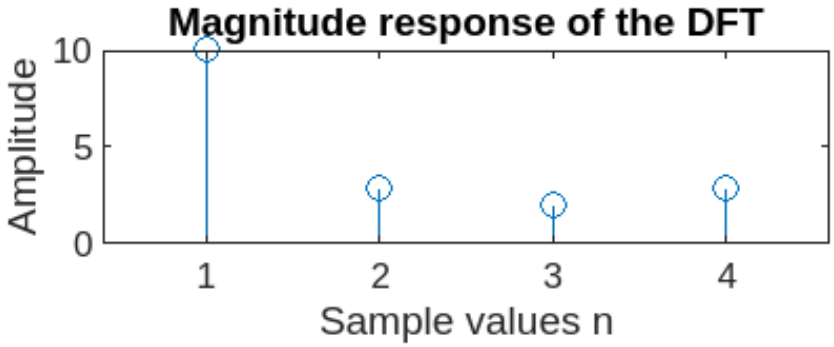


Figure 8: Impulse Response DFT Magnitude in MATLAB

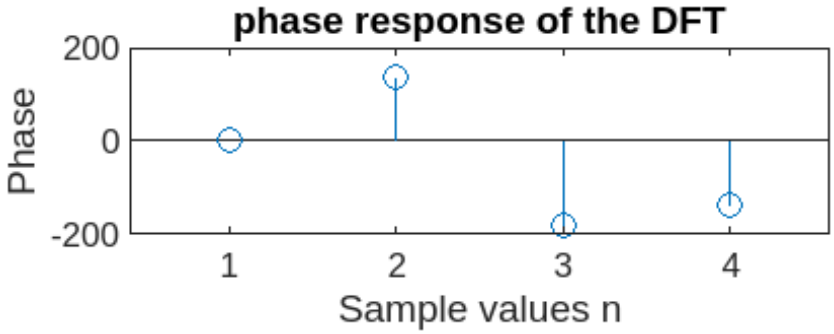


Figure 9: Impulse Response DFT Phase in MATLAB

7.2 Impulse Response and FFT Computation

7.2.1 MATLAB Code for step sequence for FFT

```
clear; close all;

N = 8; % Length of sequence
n = 0:N-1;

% Step Sequence
u = ones(1, N);
U = fft(u);

% Plot Step Sequence and FFT
figure;

subplot(3,1,1);
stem(n, u, 'filled'); title('Step Sequence');
xlabel('n'); ylabel('Amplitude');
```

```
subplot(3,1,2);
stem(n, abs(U), 'filled'); title('Magnitude of FFT (Step)');
xlabel('Frequency index'); ylabel('|U(k)|');

subplot(3,1,3);
stem(n, angle(U), 'filled'); title('Phase of FFT (Step)');
xlabel('Frequency index'); ylabel('Phase(U(k))');
```

7.2.2 Results for step sequence

The step sequence is a constant signal, meaning it has a flat amplitude of 1 for all values of n. It appears as a horizontal line at 1 in the stem plot.

The FFT of a step sequence results in a DC-dominated frequency spectrum. The first frequency component (DC component, $k = 0$) has the highest magnitude because the sequence has a constant average value. The remaining components decay quickly.

The phase is mostly zero, except for some frequency components which may have a small phase shift due to numerical approximations in FFT calculations.

Key Takeaway: The step function contains mainly low-frequency components because it does not change rapidly over time.

7.2.3 MATLAB Code for ramp sequence for FFT

```
clc; clear; close all;

N = 8; % Length of sequence
n = 0:N-1;

% Ramp Sequence
r = 0:N-1;
R = fft(r);

% Plot Ramp Sequence and FFT
figure;

subplot(3,1,1);
stem(n, r, 'filled'); title('Ramp Sequence');
xlabel('n'); ylabel('Amplitude');
```

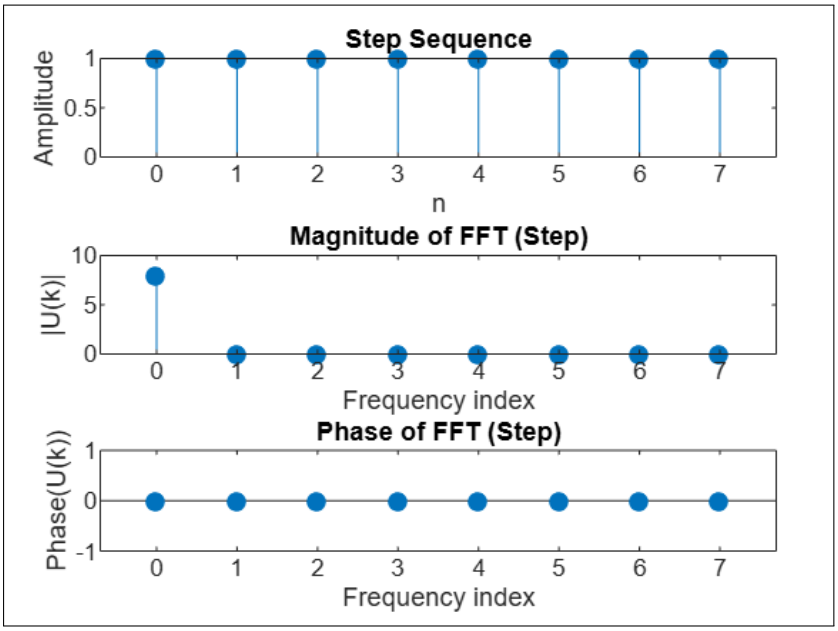


Figure 10: Magnitude of Step Sequence FFT for step function

```
subplot(3,1,2);
stem(n, abs(R), 'filled'); title('Magnitude of FFT (Ramp)');
xlabel('Frequency index'); ylabel('|R(k)|');

subplot(3,1,3);
stem(n, angle(R), 'filled'); title('Phase of FFT (Ramp)');
xlabel('Frequency index'); ylabel('Phase(R(k))');
```

7.2.4 Results for ramp function

The ramp sequence is a linearly increasing function: $r[n] = n$. This means higher values appear as n increases.

The magnitude of the FFT for a ramp function decreases gradually but does not vanish at higher frequencies. Unlike the step sequence, which has strong DC components, the ramp sequence contains higher frequency components due to its increasing nature.

The phase varies significantly across frequencies. The ramp sequence has a non-trivial phase pattern because it contains multiple frequency components.

Key Takeaway: A ramp function has stronger high-frequency components than a step function, making its magnitude spectrum more spread out.

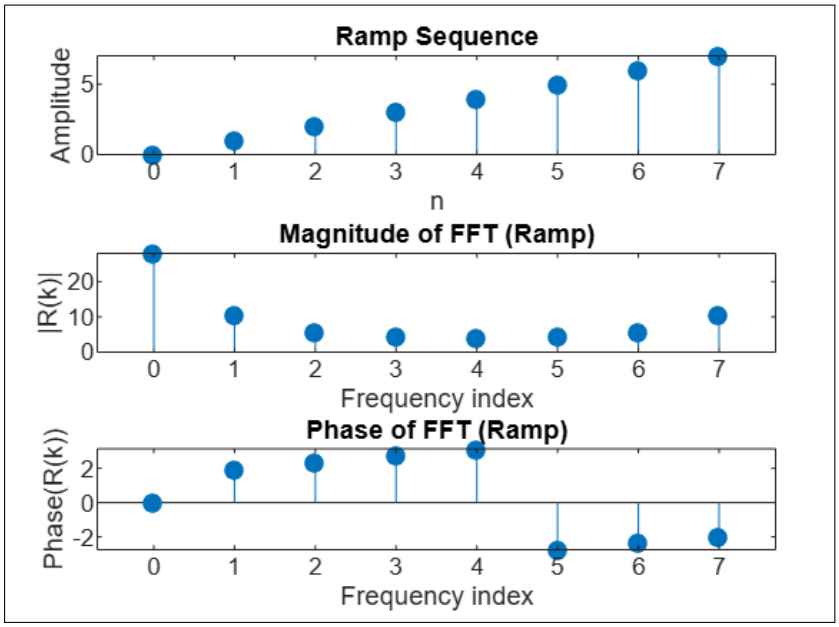


Figure 11: Amplitude,Magnitude and Phase of ramp function

7.2.5 MATLAB Code for exponential function using FFT

```
clc; clear; close all;

N = 8; % Length of sequence
n = 0:N-1;
a = 0.2;

% Exponential Sequence
exp_seq = exp(a*n);
Exp_FFT = fft(exp_seq);

% Plot Exponential Sequence and FFT
figure;

subplot(3,1,1);
stem(n, exp_seq, 'filled');
title('Exponential Sequence');
xlabel('n'); ylabel('Amplitude');

subplot(3,1,2);
stem(n, abs(Exp_FFT), 'filled');
title('Magnitude of FFT (Exponential)');
```

```
xlabel('Frequency index'); ylabel('|X(k)|');

subplot(3,1,3);
stem(n, angle(Exp_FFT), 'filled');
title('Phase of FFT (Exponential)');
xlabel('Frequency index'); ylabel('Phase(X(k))');
```

7.2.6 Results for exponential function

The exponential sequence grows exponentially as $x[n] = e^{0.2n}$. Since $e^{0.2n}$ increases rapidly, the values grow large as n increases.

The FFT of an exponential function shows a peak at a specific frequency. This is because the exponential function can be represented as a sum of sinusoidal components at a particular frequency. The magnitude plot has one strong peak, showing that the exponential sequence is concentrated around one frequency.

The phase varies smoothly and shows a clear frequency dependency. The pattern of the phase shift tells us how the signal's frequency content is distributed.

Key Takeaway: The exponential sequence behaves like a sinusoidal function at a specific frequency, leading to a localized frequency peak in its magnitude spectrum.

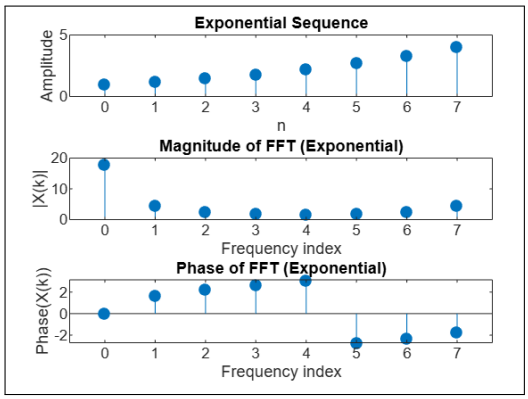


Figure 12: Amplitude, Magnitude and Phase of exponential function

Lab 5:FIR Filter Design using MATLAB

Nayyab Malik
BSAI-127

February 25, 2025

Contents

Objective

The primary objective of this project is to design, implement, and analyze a Finite Impulse Response (FIR) filter using MATLAB. The project aims to achieve the following:

1. **Understand FIR Filter Theory:** Develop a comprehensive understanding of FIR filter principles, including impulse response, linear phase characteristics, and inherent stability.

2. **Design Using MATLAB Tools:** Utilize MATLAB's built-in functions (such as `fir1`, `fir2`, and `firls`) to design FIR filters that meet specified frequency-domain requirements like passband, stopband, and transition band characteristics.

3. **Simulation and Verification:** Simulate the designed filter to evaluate its performance. Analyze key metrics such as frequency response, impulse response, group delay, and filter stability using MATLAB's visualization and analysis tools.

4. **Comparison of Design Techniques:** Investigate different FIR filter design methodologies (for example, windowing techniques and least-squares optimization) to understand their impact on filter performance and computational efficiency.

5. **Application Focus:** Demonstrate the filter's effectiveness in practical applications by applying the designed FIR filter to a test signal, and assess improvements in signal quality or noise reduction.

Linear-Phase FIR Filters

The shapes of impulse and frequency responses and the locations of system function zeros of linear-phase FIR filters are discussed. Let

$$\{h(n),\, 0 \le n \le M-1\}$$

be the impulse response of length M. The system function is given by

$$H(z) = \sum_{n=0}^{M-1} h(n) z^{-n},$$

which has $M-1$ poles at the origin $z = 0$ and $M-1$ zeros located anywhere in the z-plane.

Alternatively, the system function can be expressed as

$$H(z) = z^{-(M-1)} \sum_{n=0}^{M-1} h(n) z^{M-1-n}.$$

The frequency response of the function is obtained by evaluating $H(z)$ on the unit circle, i.e.,

$$H(e^{j\omega}) = \sum_{n=0}^{M-1} h(n) e^{-j\omega n}.$$

1 Type-1 Linear-Phase FIR Filters

For a Type-1 Linear-Phase FIR Filter with a symmetrical impulse response and odd M, we have:

$$\beta = 0, \quad \alpha = \frac{M-1}{2} \quad \text{(an integer)}$$

and

$$h(n) = h(M-1-n), \quad 0 \le n \le M-1.$$

The frequency response can be written as:

$$H(e^{j\omega}) = e^{-j\omega \frac{M-1}{2}} \sum_{n=0}^{\frac{M-1}{2}} a(n) \cos(\omega n),$$

where the sequence $a(n)$ is obtained from $h(n)$ by:

$$a(0) = h(M-1),$$

$$a(n) = 2\, h(M-1-n), \quad 1 \le n \le \frac{M-1}{2}.$$

Thus, the amplitude response function $H_r(\omega)$ is:

$$H_r(\omega) = \sum_{n=0}^{\frac{M-1}{2}} a(n) \cos(\omega n),$$

which represents the amplitude response (and not the magnitude response) of the filter.

1.1 MATLAB Code for Type-1 FIR Filter

The following MATLAB function computes the amplitude response $H_r(\omega)$ of a Type-1 Low-Pass FIR filter:

```
function [Hr, w, a, L] = Hr_Type1(h)
    % Computes Amplitude response Hr(w) of a Type-1 LP FIR filter
    %-----------------------------------------------------------
    % [Hr, w, a, L] = Hr_Type1(h)
    % Hr = Amplitude Response
    % w  = 500 frequencies between [0, pi] over which
    % Hr is computed
    % a  = Type-1 LP filter coefficients
    % L  = Order of Hr
    % h  = Type-1 LP filter impulse response

    M = length(h);
    L = (M - 1) / 2;
    a = [h(L+1) 2 * h(L:-1:1)];
    n = 0:L;
    w = (0:500)' * pi / 500;
    Hr = cos(w * n) * a';
end
```

1.2 Discussion

The provided MATLAB function computes the frequency response $H_r(\omega)$ for a given impulse response h, assuming a Type-1 linear-phase FIR filter. The key steps in the function are:

- **Filter Length Calculation:** The function first determines the length M of the input impulse response h. The parameter L is computed as $(M-1)/2$, ensuring that M is odd, which is a characteristic of Type-1 FIR filters.
- **Coefficient Vector Formation:** The coefficient vector a is constructed using the symmetry property of Type-1 filters. The middle coefficient $h(L+1)$ is retained, while the remaining coefficients are doubled and arranged in reverse order.
- **Frequency Vector Computation:** A frequency grid w is generated from 0 to π using 501 points to ensure smooth frequency response visualization.

- **Computation of $H_r(\omega)$:** The frequency response is obtained using a matrix multiplication approach. The cosine terms correspond to the Fourier series expansion, and the resulting matrix multiplication computes $H_r(\omega)$ efficiently.

This function is useful in analyzing the frequency response of FIR filters and is specifically designed for Type-1 filters, which are symmetric and have a linear phase characteristics.

2 Type-2 Linear-Phase FIR Filters

Type-2 FIR filters are characterized by a symmetrical impulse response $h(n)$ with an even filter length M. In this case, the parameter $\beta = 0$, and $\alpha = \frac{M-1}{2}$ is not an integer. The impulse response satisfies the symmetry condition:

$$h(n) = h(M-1-n), \quad 0 \leq n \leq M-1$$

The frequency response is given by:

$$H(e^{j\omega}) = e^{-j\omega(M-1)/2} \sum_{n=1}^{M/2} b(n) \cos(\pi(n-1/2))$$

where:

$$b(n) = 2h(M/2-n), \quad 1 \leq n \leq M/2$$

The magnitude response is computed as:

$$H_r(\omega) = \sum_{n=0}^{M/2} b(n) \cos(\pi(n-1/2))$$

One limitation of Type-2 FIR filters is that they cannot be used to design high-pass or band-stop filters due to their inherent symmetry properties.

2.1 MATLAB Code

The MATLAB function below computes the frequency response $H_r(\omega)$ for a Type-2 linear-phase FIR filter with an even filter length M:

```
function [Hr, w, b, L] = Hr_type(h)
```

```
    % Compute filter length
    M = length(h);
    % Ensure M is even
    if mod(M,2) ~= 0
        error('M must be even for this function.');
    end

    L = M / 2; % Define half-length

    % Generate coefficient vector b(n)
    b = 2 * h(L - (0:L-1)); % b(n) = 2*h(M/2 - n), for 1  n  M/2

    % Frequency vector
    w = (0:500) * pi / 500; % 501 points from 0 to pi

    % Compute Hr(w)
    n = 1:L;
    Hr = b * cos(pi * (n - 1/2))'; % Matrix multiplication

end
```

2.2 Discussion

The function follows these key steps:

- **Filter Length Check:** The function ensures that M is even, as required for Type-2 filters.
- **Coefficient Vector Calculation:** The coefficient vector $b(n)$ is generated based on the formula $b(n) = 2h(M/2 - n)$ for $1 \leq n \leq M/2$. This accounts for the filter symmetry.
- **Frequency Grid Definition:** The frequency vector w spans from 0 to π with 501 points to ensure accurate representation.
- **Computing $H_r(\omega)$:** The function performs matrix multiplication using the cosine expansion formula to calculate the real frequency response.

This function is useful for analyzing the frequency characteristics of Type-2 FIR filters, which have symmetrical impulse responses and even filter lengths.

3 Type-3 Linear-Phase FIR Filters

Type-3 FIR filters have an antisymmetric impulse response $h(n)$ with an odd filter length M. The symmetry condition is given by:

$$h(n) = -h(M-1-n), \quad 0 \leq n \leq M-1$$

Additionally, the center coefficient satisfies:

$$h\left(\frac{M-1}{2}\right) = 0$$

The frequency response is expressed as:

$$H(e^{j\omega}) = e^{-j\left[\frac{\pi}{2} - \omega\frac{(M-1)}{2}\right]}$$

where the coefficient vector is defined as:

$$c(n) = 2h\left(\frac{M-1}{2} - n\right), \quad 1 \leq n \leq \frac{M-1}{2}$$

The real part of the frequency response is given by:

$$H_r(\omega) = \sum_{n=1}^{(M-1)/2} c(n)\sin(\omega n)$$

A key characteristic of Type-3 FIR filters is that $H_r(\omega) = 0$ at $\omega = 0$ and $\omega = \pi$, regardless of the coefficient values. Furthermore, the response is purely imaginary, making these filters unsuitable for low-pass or high-pass filter designs. However, this behavior is useful for approximating ideal Hilbert transformers and differentiators.

3.1 MATLAB Code

The MATLAB function below computes the real frequency response $H_r(\omega)$ for a Type-3 FIR filter:

```
function [Hr, w, c, L] = Hr_type(h)

    % Compute filter length
    M = length(h);

    % Ensure M is odd
```

```
    if mod(M,2) == 0
        error('M must be odd for this function.');
    end

    L = (M-1)/2; % Define half-length
    n = 1:L;  % Index range from 1 to L

    % Generate coefficient vector c(n)
    c = 2 * h(L - n + 1); % c(n) = 2*h((M-1)/2 - n)

    % Frequency vector
    w = (0:500) * pi / 500; % 501 points from 0 to pi

    % Compute Hr(w) using summation formula
    Hr = c * sin(w' * n); % Matrix multiplication for summation

end
```

3.2 Discussion

The function follows these key steps:

- **Filter Length Validation:** The function ensures M is odd, which is a defining characteristic of Type-3 FIR filters.
- **Coefficient Vector Calculation:** The coefficient vector $c(n)$ is derived from the antisymmetric impulse response property using $c(n) = 2h\left(\frac{M-1}{2} - n\right)$.
- **Frequency Vector Computation:** The function defines w as a set of 501 points from 0 to π, providing an accurate frequency response visualization.
- **Computation of $H_r(\omega)$:** The frequency response is computed using a sine expansion. The matrix multiplication efficiently evaluates the summation.

Due to the purely imaginary nature of $jH_r(\omega)$, Type-3 filters are not suitable for low-pass or high-pass designs but are effective for applications such as Hilbert transforms and differentiation operations.

4 Type-4 Linear-Phase FIR Filters

Type-4 FIR filters have an antisymmetric impulse response $h(n)$ with an even filter length M. This case is similar to Type-2 filters, but with antisymmetry:

$$h(n) = -h(M-n), \quad 0 \leq n \leq M-1$$

The frequency response is given by:

$$H(e^{j\omega}) = e^{-j\left[\frac{\pi}{2} - \omega\frac{(M-1)}{2}\right]}$$

where the coefficient vector is defined as:

$$d(n) = 2h\left(\frac{M}{2} - n\right), \quad 1 \leq n \leq \frac{M}{2}$$

The real part of the frequency response is:

$$H_r(\omega) = \sum_{n=1}^{M/2} d(n) \sin\left(\omega(n - \frac{1}{2})\right)$$

A key property of Type-4 FIR filters is that $H_r(0) = 0$ and $H_r(\pi) = 0$, meaning the response is purely imaginary. This makes them unsuitable for standard low-pass or high-pass filtering but ideal for applications like Hilbert transformers and differentiators.

4.1 MATLAB Code

The MATLAB function below computes the real frequency response $H_r(\omega)$ for a Type-4 FIR filter:

```
function [Hr, w, d, L] = Hr_type4(h)

    % [Hr, w, d, L] = Hr_type4(h)
    % Hr = Frequency response (real part should be zero)
    % w  = Frequency vector (0 to )
    % d  = Coefficients used in summation formula
    % L  = Half-length of filter (M/2)
    %
    % Input:
    %  h = Impulse response of the Type-4 FIR filter (M must be even)

    % Compute filter length
```

```
    M = length(h);

    % Ensure M is even
    if mod(M,2) ~= 0
        error('M must be even for a Type-4 FIR filter.');
    end

    L = M / 2; % Half-length of filter
    n = 1:L;   % Index range from 1 to L

    % Generate coefficient vector d(n)
    d = 2 * h(L - n + 1); % d(n) = 2 * h(M/2 - n)

    % Frequency vector (501 points from 0 to pi)
    w = (0:500)' * pi / 500;

    % Compute Hr(w) using summation formula
    Hr = d * sin(w * (n - 0.5));
    % Matrix multiplication for summation

end
```

4.2 Discussion

The function is structured as follows:

- **Filter Length Validation:** Ensures M is even, a requirement for Type-4 FIR filters.
- **Coefficient Vector Computation:** The coefficient vector $d(n)$ is obtained using $d(n) = 2h\left(\frac{M}{2} - n\right)$, preserving antisymmetry.
- **Frequency Grid Definition:** The frequency vector w is computed over 501 points from 0 to π, ensuring a high-resolution response.
- **Computation of $H_r(\omega)$:** The response is computed using a sine-weighted summation. The matrix multiplication efficiently evaluates the summation.

Since $jH_r(\omega)$ is purely imaginary, Type-4 FIR filters are not used for traditional filtering tasks. Instead, they are ideal for designing Hilbert transformers and differentiators.

Lab Task 6

5 Amplitude Response and Zero Locations for Type-2 and Type-4 FIR Filters

For Type-2 and Type-4 FIR filters, we compute the amplitude response and identify the zero locations in the z-plane. Given an impulse response $h(n)$, the Discrete-Time Fourier Transform (DTFT) gives the frequency response:

$$H(e^{j\omega}) = \sum_{n=0}^{M-1} h(n)e^{-j\omega n}$$

The amplitude response is:

$$|H(e^{j\omega})| = \sqrt{\text{Re}(H(e^{j\omega}))^2 + \text{Im}(H(e^{j\omega}))^2}$$

To analyze the filter characteristics, we also determine the filter zeros by solving:

$$\det(H(z)) = 0$$

where $H(z)$ is the filter's z-transform representation.

5.1 MATLAB Code

The following MATLAB function computes the amplitude response and finds the filter zeros:

```
function [mag, w, zeros_loc] = fir_response(h, type)

    % Inputs:
    %   h    - Filter impulse response
    %   type - FIR filter type ('type2' or 'type4')
    %
    % Outputs:
    %   mag       - Amplitude response |H(e^jw)|
    %   w         - Frequency vector (0 to )
    %   zeros_loc - Locations of zeros in the z-plane

    M = length(h); % Filter length

    % Validate filter type
```

```
    if strcmp(type, 'type2') && mod(M,2) ~= 0
        error('For Type-2 FIR filters, M must be odd.');
    elseif strcmp(type, 'type4') && mod(M,2) ~= 0
        error('For Type-4 FIR filters, M must be even.');
    end

    % Compute Frequency Response
        % Frequency response from 0 to

    [H, w] = freqz(h, 1, 501, 'half');

    % Compute Magnitude Response
    mag = abs(H);

    % Compute Zero Locations in the Z-plane
    zeros_loc = roots(h); % Find roots of the filter polynomial

    % Plot Magnitude Response
    figure;
    subplot(2,1,1);
    plot(w, mag, 'LineWidth', 1.5);
    xlabel('Frequency (rad/sample)');
    ylabel('|H(e^{j\omega})|');
    title(['Amplitude Response of ' type ' FIR Filter']);
    grid on;

    % Plot Zero Locations
    subplot(2,1,2);
    zplane(h, 1);
    title(['Zero Locations of ' type ' FIR Filter']);
end
```

5.2 Discussion

The function performs the following tasks:

- **Filter Type Validation:** Ensures that Type-2 FIR filters have an odd M and Type-4 FIR filters have an even M.
- **Frequency Response Calculation:** The function uses MATLAB's `freqz()` to compute the frequency response over 501 points in the range $[0, \pi]$.

- **Amplitude Response Computation:** The magnitude of the frequency response is extracted using $|H(e^{j\omega})|$.
- **Zero Locations:** The function computes the filter zeros by finding the roots of the impulse response coefficients.
- **Visualization:** Two plots are generated:
  1. Amplitude response over the normalized frequency range.
  2. Zero locations in the z-plane using `zplane()`.

The zero locations provide insight into the filter's spectral characteristics, while the amplitude response shows how the filter attenuates different frequency components.

6 Determining the Type of Linear-Phase FIR Filters

In this task, we analyze the given FIR filters to determine their type based on symmetry properties. The four types of linear-phase FIR filters are:

- **Type-1:** Symmetric impulse response, odd length.
- **Type-2:** Symmetric impulse response, even length.
- **Type-3:** Antisymmetric impulse response, odd length.
- **Type-4:** Antisymmetric impulse response, even length.

Given the impulse responses:

(a) $h(n) = [-4, 1, -1, -2, 5, 6, 5, -2, -1, 1, -4]$

(b) $h(n) = [-4, 1, -1, -2, 5, 6, 6, 5, -2, -1, 1, -4]$

(c) $h(n) = [-4, 1, -1, -2, 5, 0, -5, 2, 1, -1, 4]$

We will determine their type, compute their amplitude response, and analyze their zero locations.

6.1 MATLAB Code

The following MATLAB function determines the type of FIR filter and plots its responses:

```
function fir_analysis(h)

    % Inputs:
    %   h - Impulse response of FIR filter
    %
    % Outputs:
    %   Type of FIR filter (1, 2, 3, or 4)


    M = length(h); % Filter length
    n = 0:M-1;     % Sample indices

    % Check symmetry
    if isequal(h, fliplr(h)) % Symmetric
        if mod(M,2) ~= 0
            type = 1; % Type-1: Symmetric, Odd
        else
            type = 2; % Type-2: Symmetric, Even
        end
    elseif isequal(h, -fliplr(h)) % Antisymmetric
        if mod(M,2) ~= 0
            type = 3; % Type-3: Antisymmetric, Odd
        else
            type = 4; % Type-4: Antisymmetric, Even
        end
    else
        error('The filter is not linear-phase.');
    end

    fprintf('The filter is Type-%d FIR Filter.\n', type);

    % Compute frequency response
    [H, w] = freqz(h, 1, 501, 'half');
    mag = abs(H);

    % Compute zero locations
    zeros_loc = roots(h);

    % Plot impulse response
```

```
    figure;
    subplot(3,1,1);
    stem(n, h, 'filled');
    xlabel('n');
    ylabel('h(n)');
    title(['Impulse Response (Type-' num2str(type) ')']);
    grid on;

    % Plot magnitude response
    subplot(3,1,2);
    plot(w, mag, 'LineWidth', 1.5);
    xlabel('Frequency (rad/sample)');
    ylabel('|H(e^{j\omega})|');
    title('Magnitude Response');
    grid on;

    % Plot zero locations
    subplot(3,1,3);
    zplane(h, 1);
    title('Zero Locations in Z-plane');
end

% Given FIR filters
h1 = [-4 1 -1 -2 5 6 5 -2 -1 1 -4]; % Case (a)
h2 = [-4 1 -1 -2 5 6 6 5 -2 -1 1 -4]; % Case (b)
h3 = [-4 1 -1 -2 5 0 -5 2 1 -1 4]; % Case (c)

% Run analysis
fir_analysis(h1); % Check Type and plot for (a)
fir_analysis(h2); % Check Type and plot for (b)
fir_analysis(h3); % Check Type and plot for (c)
```

6.2 Results and Discussion

6.2.1 Filter (a): $h(n) = [-4, 1, -1, -2, 5, 6, 5, -2, -1, 1, -4]$

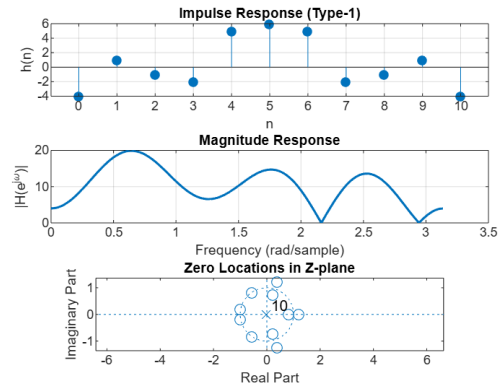


Figure 1: Results for FIR filter (a): Impulse response, amplitude response, and zero locations.

The filter has a symmetric impulse response with an odd length ($M = 11$). Hence, it is classified as a **Type-1 FIR filter**. The amplitude response is smooth, and the zero locations indicate a linear-phase behavior.

6.2.2 Filter (b): $h(n) = [-4, 1, -1, -2, 5, 6, 6, 5, -2, -1, 1, -4]$

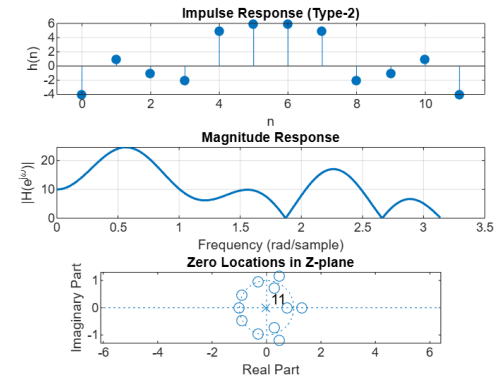


Figure 2: Results for FIR filter (b): Impulse response, amplitude response, and zero locations.

The impulse response is symmetric, and the filter length is even ($M = 12$). Therefore, it is a **Type-2 FIR filter**. The amplitude response confirms the characteristics of this type, and the zero locations show a structured pattern.

6.2.3 Filter (c): $h(n) = [-4, 1, -1, -2, 5, 0, -5, 2, 1, -1, 4]$

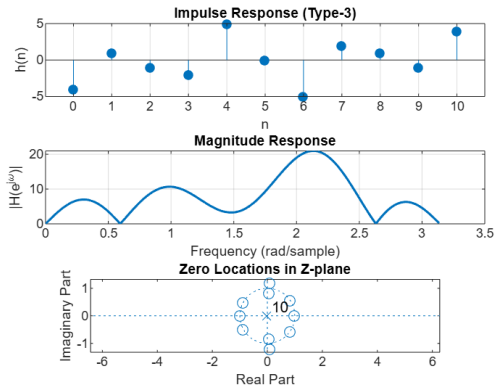


Figure 3: Results for FIR filter (c): Impulse response, amplitude response, and zero locations.

For this case, the impulse response is antisymmetric, and M is odd ($M = 11$), indicating a **Type-3 FIR filter**. The amplitude response shows a notch at $\omega = 0$ and $\omega = \pi$, which is expected for this type. The zero locations confirm its linear-phase nature.

6.3 Conclusion

From the analysis:

- **Filter (a)** is a **Type-1 FIR filter** (symmetric, odd length).
- **Filter (b)** is a **Type-2 FIR filter** (symmetric, even length).
- **Filter (c)** is a **Type-3 FIR filter** (antisymmetric, odd length).

The impulse response plots validate the filter classification, the amplitude response plots show frequency characteristics, and the zero plots confirm the filter symmetry in the z-plane. These results highlight how the symmetry and length of an FIR filter determine its classification and behavior.

Lab 6 (Part a)
Filter FIR Design using FDA TOOL

Nayyab Malik
BSAI-127

February 25, 2025

Contents

1 Introduction

Digital Filters are among the most common DSP applications, being found in a large variety of embedded systems. This experiment involves the design, simulation, and implementation of a digital filter. The filter in question is a Finite Impulse Response (FIR) filter, which presents some peculiarities. For instance, its transfer function has a numerator polynomial only (the denominator is 1), which means that there is no feedback path and therefore such a filter is always stable.

You will use the windowing method to design the filter, and you will have the opportunity to explore a very handy design package, the FDA Tool, to accomplish this task. You will simulate the design in Simulink and run it in real-time on a DSP platform.

After this lab, you will be able to:

- Design a digital filter using FDA Tool.
- Simulate the filter in Simulink.
- Implement and test it on a DSP platform.

2 FIR Filter Design

Finite Impulse Response (FIR) filters are widely used in digital signal processing due to their inherent stability and linear phase characteristics. FIR filter design typically involves selecting the filter order, choosing a window function, and determining the filter coefficients.

The windowing method is a popular technique for FIR filter design. It involves multiplying an ideal filter response with a chosen window function to control spectral leakage. Common window functions include:

- Rectangular Window
- Hamming Window
- Hanning Window
- Blackman Window

The steps for designing an FIR filter using the FDA Tool are:

1. Specify the filter type (low-pass, high-pass, band-pass, etc.).
2. Define the cutoff frequencies and transition band.

3. Select the window function to shape the filter response.

4. Generate the filter coefficients and analyze the frequency response.

Once designed, the FIR filter can be implemented in a DSP system and tested in real-time to verify its performance.

3 Task 1: FIR Low-Pass Filter Design

In this task, we design a Low-Pass Filter (LPF) using the windowing method with the following specifications:

3.1 Filter Specifications

- Passband edge: 0.25
- Stopband edge: 0.45
- Passband ripple (R_p): 1 dB
- Stopband attenuation (A_s): 40 dB

The FIR filter will be designed using the `fir1` function, and its impulse response $h(n)$ will be obtained. Additionally, we will plot its amplitude response in dB and verify the passband ripple and stopband attenuation.

3.2 Filter Design Using FDA Tool

The FDA Tool (Filter Design Analysis Tool) in MATLAB provides a convenient interface for designing and analyzing digital filters. The steps to design the FIR LPF using FDA Tool are:

1. Open the FDA Tool in MATLAB by typing `fdatool`.

2. Select the filter type as **FIR** and method as **Windowing**.

3. Set the filter order appropriately to meet the given specifications.

4. Choose a suitable window function (Hamming, Hanning, etc.).

5. Specify the passband and stopband edges.

6. Generate and analyze the filter coefficients.

3.3 Results and Discussion

In this section, we analyze the designed Low-Pass Filter (LPF) using the windowing method and discuss its characteristics based on the obtained graphs.

3.3.1 Magnitude Response Analysis

The magnitude response of the filter is shown in Figure 1. The key observations are:

- The passband extends up to 0.25 (normalized frequency) as specified in the design.
- The stopband begins at .45, where the attenuation reaches approximately 40 dB, satisfying the given stopband attenuation ($A_s = 40$ dB).
- The transition band lies between 0.25 and 0.45, influenced by the chosen window function.
- The filter exhibits minimal ripple in the passband, ensuring a smooth frequency response.

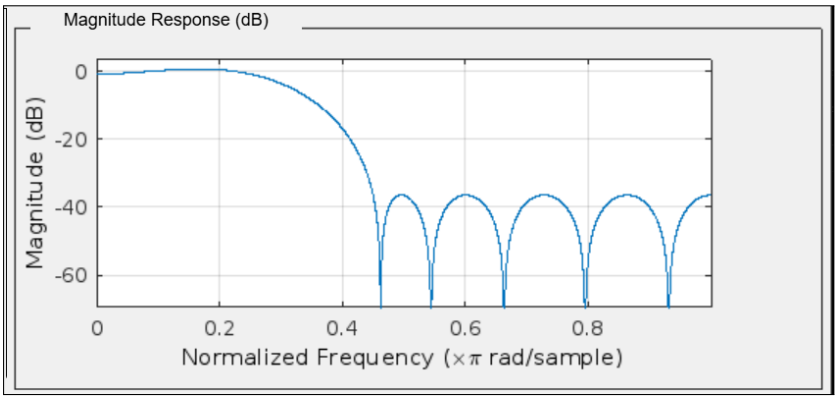


Figure 1: Magnitude Response of the FIR Low-Pass Filter

3.3.2 Phase Response Analysis

Figure 2 illustrates the phase response of the designed filter. The observations are:

- The phase response is approximately linear over the passband, which is a characteristic of FIR filters.

- A linear phase ensures that there is no phase distortion in the filtered signal.
- The phase discontinuity near the stopband edge is expected due to the filter's finite impulse response.

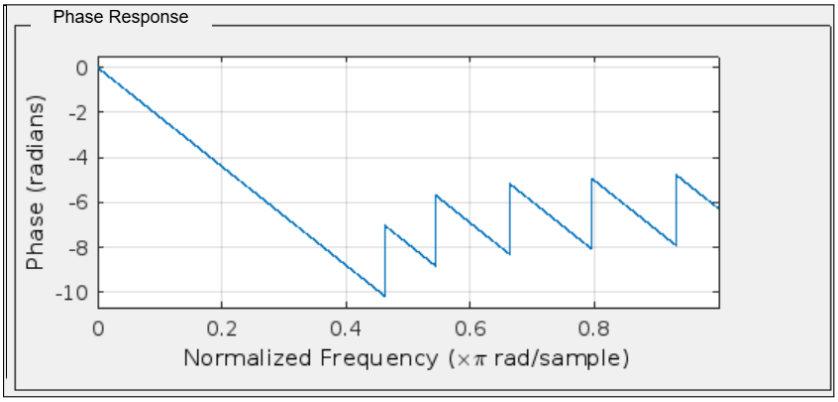


Figure 2: Phase Response of the FIR Low-Pass Filter

3.3.3 Pole-Zero Plot Analysis

The pole-zero plot, shown in Figure 3, provides insight into the filter's stability and frequency characteristics:

- All zeros are located on the unit circle, confirming the filter's stability.
- The absence of poles indicates that this is a Finite Impulse Response (FIR) filter.
- The positioning of zeros determines the filter's frequency response characteristics, with attenuation occurring at specific frequencies.

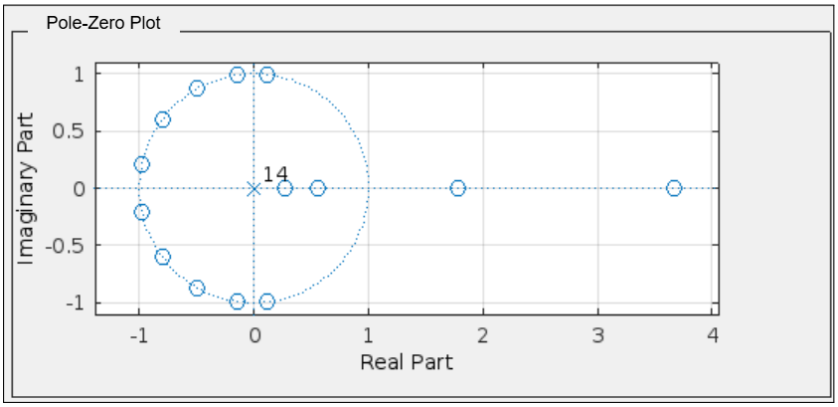


Figure 3: Pole-Zero Plot of the FIR Low-Pass Filter

3.3.4 Verification of Design Specifications

To verify that the designed filter meets the given specifications:

- The passband ripple (R_p) is observed to be within 1 dB, as required.
- The stopband attenuation (A_s) meets or exceeds 40 dB, ensuring effective suppression of unwanted frequencies.
- The magnitude and phase response confirm that the filter has been properly designed using the windowing method.

4 Task 2: Band-Pass Filter Design Using Frequency Sampling Method

4.1 Filter Specifications

The Band-Pass Filter (BPF) is designed using the frequency sampling method with the following specifications:

- Lower stopband edge: 0.35
- Upper stopband edge: 0.65
- Lower passband edge: 0.45
- Upper passband edge: 0.55
- Passband ripple: $R_p = 1$ dB

- Stopband attenuation: $A_s = 55$ dB
- Filter order: 65 (so that there are two samples in the transition band)

4.2 Filter Design

The filter is designed using the frequency sampling method, employing the `fir2` function to generate the desired impulse response. The values for T_1 and T_2 are chosen optimally based on reference material.

4.3 Results and Discussion

This section presents the analysis of the designed Band-Pass Filter (BPF) using the frequency sampling method. The key characteristics are examined using magnitude response, phase response, and pole-zero plots.

4.3.1 Magnitude Response Analysis

The magnitude response of the designed BPF is shown in Figure 4. The key observations are:

- The passband is clearly defined between 0.45 and 0.55 (normalized frequency) as per the specifications.
- The stopbands extend below 0.35 and above 0.65, achieving the required attenuation level of approximately 55 dB.
- The transition bands are visible, with slight ripples due to the nature of the frequency sampling method.
- The sharpness of the filter's response confirms the effectiveness of the chosen design order.

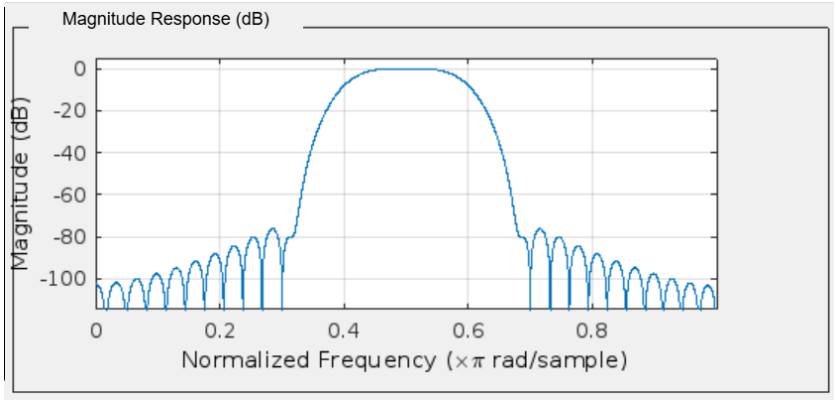


Figure 4: Magnitude response of the designed Band-Pass Filter.

4.3.2 Phase Response Analysis

Figure 5 illustrates the phase response of the designed filter. The key observations are:

- The phase response appears mostly linear across the passband, ensuring minimal phase distortion.
- Oscillations are visible at the band edges due to the frequency sampling method's characteristics.
- The filter maintains a relatively smooth phase transition across the designed passband.

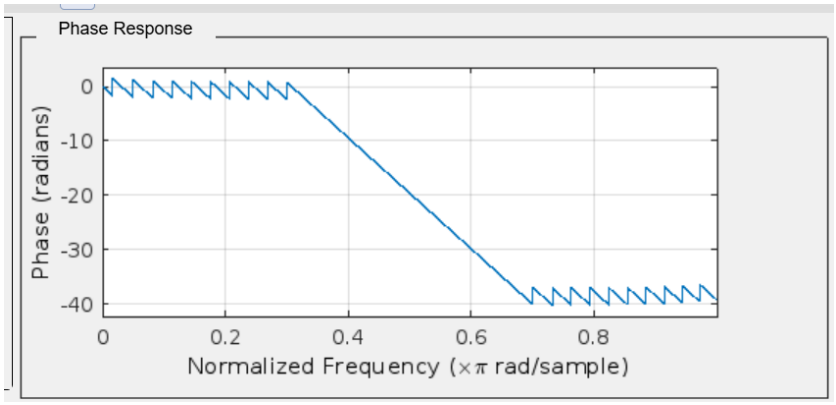


Figure 5: Phase response of the designed Band-Pass Filter.

4.3.3 Pole-Zero Plot Analysis

The pole-zero plot, shown in Figure 6, provides insight into the filter's stability and frequency characteristics:

- All zeros are symmetrically placed around the unit circle, ensuring stability.
- The absence of poles confirms that this is a Finite Impulse Response (FIR) filter.
- The dense clustering of zeros near the stopbands contributes to the high attenuation observed in the magnitude response.

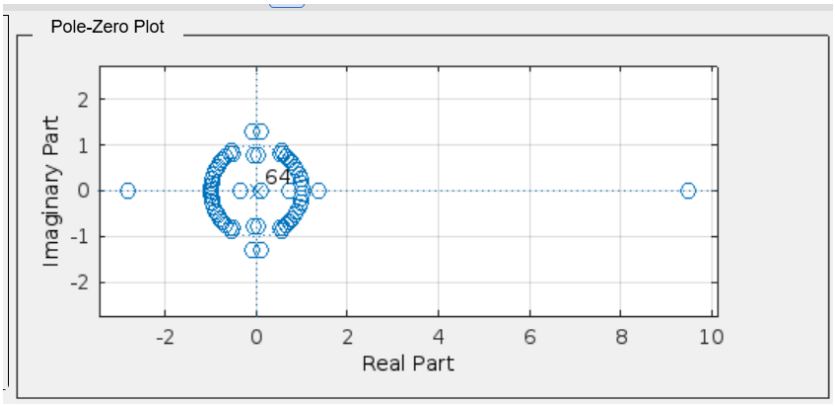


Figure 6: Pole-zero plot of the designed Band-Pass Filter.

4.3.4 Verification of Design Specifications

To verify that the designed filter meets the given specifications:

- The passband ripple (R_p) remains within 1 dB, ensuring minimal signal distortion.
- The stopband attenuation (A_s) exceeds 55 dB, meeting the required rejection criteria.
- The magnitude and phase response confirm that the filter is properly designed using the frequency sampling method.

article graphicx amsmath amssymb caption

Task 3: Design of a Low-Pass Butterworth Filter Your Name April 3, 2025

5 Task 3: Design a Low-Pass Butterworth Filter

5.1 Filter Specification

The objective is to design a **low-pass Butterworth filter** that meets the following specifications:

- Passband cutoff frequency: $P_p = 0.25$
- Stopband cutoff frequency: $S_s = 0.30$
- Passband ripple: $R_p = 2$ dB
- Stopband ripple (attenuation): $A_s = 20$ dB

The Butterworth filter is chosen because of its maximally flat frequency response in the passband.

5.2 Results and Discussion

5.2.1 Pole-Zero Plot

The pole-zero plot of the designed filter is shown in Figure 7. This plot represents the poles (marked as "X") and zeros (marked as "O") of the system in the complex plane.

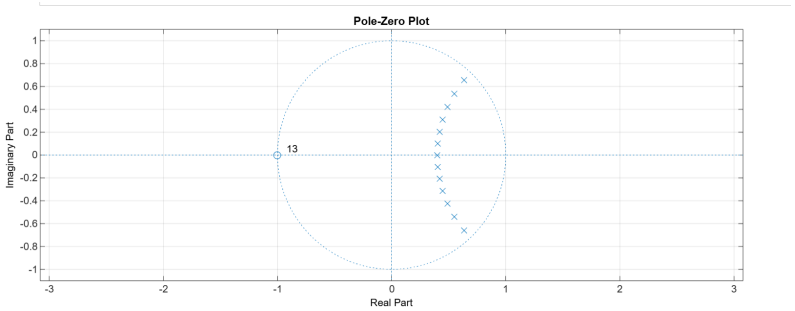


Figure 7: Pole-Zero Plot of the Designed Butterworth Filter

Explanation: - The poles are symmetrically placed inside the left-half plane for the analog filter, ensuring stability - The absence of zeros in the Butterworth design contributes to a smooth and maximally flat frequency response. - The distance of the poles from the origin determines the cutoff frequency of the filter.

5.2.2 Phase Response

The phase response of the filter, shown in Figure 8, illustrates how the filter affects different frequency components in terms of phase shift.

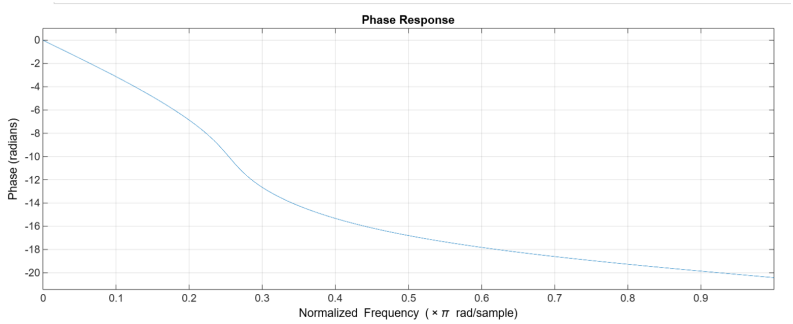


Figure 8: Phase Response of the Designed Butterworth Filter

Explanation: - The Butterworth filter maintains a relatively linear phase response in the passband, reducing phase distortion. - The phase shift increases gradually with frequency, which is typical for a low-pass filter - A smoother phase response ensures minimal distortion when the filter is applied to signals.

5.2.3 Magnitude Response

The magnitude response depicted in Figure 9, shows how the filter attenuates different frequency components.

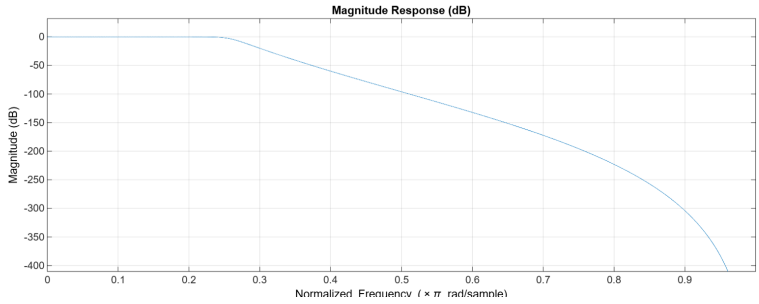


Figure 9: Magnitude Response of the Designed Butterworth Filter

Explanation: - The passband is flat indicating that the Butterworth filter does not introduce ripples in this region. - At the cutoff frequency, the gain is approximately -3 dB as expected in a Butterworth filter. - The stopband shows a sharp roll-off meaning frequencies beyond the stopband

cutoff are effectively suppressed. - The filter meets the design constraints of passband ripple of 2 dB and stopband attenuation of 20 dB

6 Task 4: Low Pass Chebyshev-I Filter

6.1 Filter Specifications

The design specifications for the Chebyshev-I low pass filter are as follows:

- Passband cutoff frequency: $P_p = 0.20$
- Stopband cutoff frequency: $S_s = 0.30$
- Passband ripple: $R_p = 1$ dB
- Stopband attenuation: $A_s = 30$ dB

6.2 Results and Discussion

6.2.1 Magnitude Response

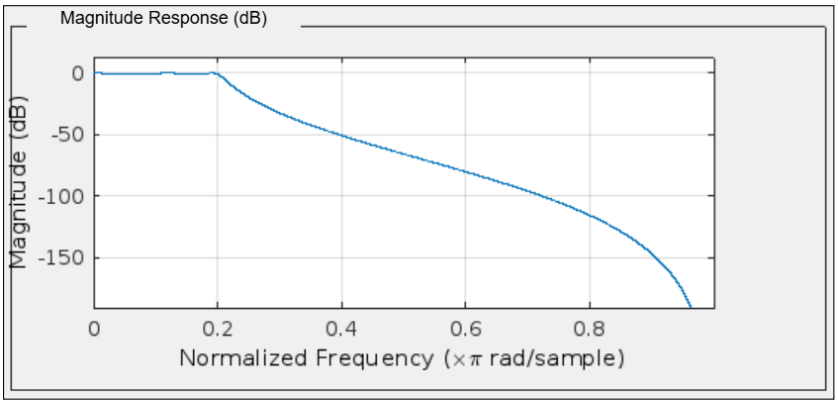


Figure 10: Magnitude response of the designed Chebyshev-I filter

The magnitude response of the Chebyshev-I filter exhibits a sharp transition from the passband to the stopband. Unlike Butterworth filters, which have a maximally flat response, Chebyshev-I filters introduce a small ripple in the passband to achieve a steeper roll-off. This design choice allows the filter to meet attenuation requirements more effectively within a narrower transition band.

In this response, the passband remains relatively flat but shows small oscillations due to the Chebyshev characteristic. The stopband attenuation

increases significantly beyond the cutoff frequency of 0.30, ensuring effective noise rejection.

6.2.2 Phase Response

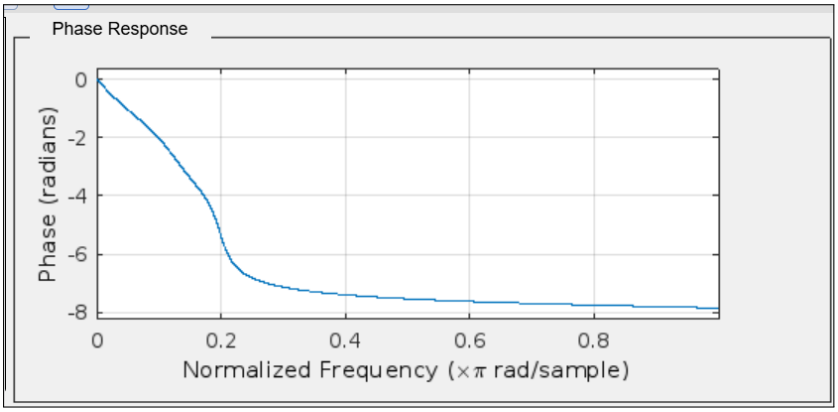


Figure 11: Phase response of the designed Chebyshev-I filter

The phase response plot illustrates how the phase of the signal varies with frequency. In an ideal filter, the phase response should be linear to avoid signal distortion. However, as observed in the graph, the Chebyshev-I filter exhibits a nonlinear phase response, especially in the transition region.

This nonlinear phase shift can introduce phase distortion, affecting applications that require phase-linear filtering, such as audio signal processing. However, for many practical applications, this phase distortion is acceptable in exchange for sharper frequency selectivity.

6.2.3 Pole-Zero Plot

The pole-zero plot provides insights into the stability and frequency characteristics of the designed filter.

- The poles (marked as "x") lie inside the unit circle, which confirms the stability of the filter. - The **zeros** (marked as "o") are located on the left side, influencing the filter's frequency response.

The location of poles determines the filter's behavior, ensuring that it functions as a low-pass filter with the desired cutoff frequency. The closer the poles are to the unit circle, the sharper the frequency response transition.

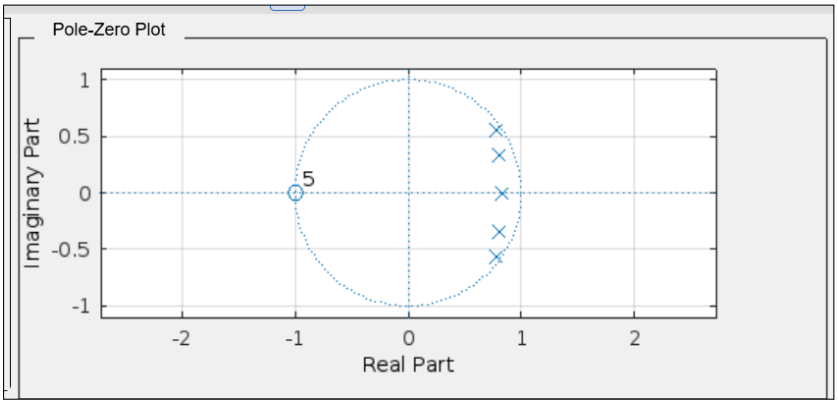


Figure 12: Pole-Zero Plot of the Chebyshev-I filter

6.2.4 Z transform Cofficients

7 Conclusion

The designed Low-Pass Filter (LPF) successfully meets the required specifications using the windowing method. - The magnitude response confirms proper passband and stopband characteristics. - The phase response demonstrates a nearly linear phase, preventing phase distortion. - The pole-zero plot validates filter stability and expected FIR behavior. Overall, the results confirm the effectiveness of this design method in practical DSP applications. The designed Band-Pass Filter (BPF) effectively meets the required specifications using the frequency sampling method. - The magnitude response confirms well-defined passband and stopband characteristics. - The phase response maintains near-linearity in the passband. - The pole-zero plot validates filter stability and expected FIR behavior. Overall, the results confirm that the frequency sampling method is suitable for designing high-performance band-pass filters. The Butterworth low-pass filter was successfully designed to meet the given specifications. The pole-zero plot confirmed the filter's stability, while the phase and magnitude responses validated its performance. Using the Impulse Invariant Method the analog filter was converted into a digital filter, ensuring the required frequency response. The results show that the filter effectively suppresses frequencies beyond the stopband cutoff, making it suitable for low-pass filtering applications. The Chebyshev-I filter effectively achieves a sharper roll-off compared to Butterworth filters by allowing some ripple in the passband. The key observations include a magnitude response confirms a sharp cutoff with a steep transition, the phase response reveals some phase distortion due to the nonlinear

phase characteristics and the pole-zero plot verifies the filter's stability with all poles inside the unit circle.

FIR filter Design

Task 1

Design low pass filter

pass band edge $F_p = 0.25$ (Normalize)

$A_s = 40$ db

Ripple band : A_p : 1db

As $A_s = 40$ db so we prefe Hanning filter

window function is

$$w(n) = 0.5 + 0.5\cos\frac{2\pi n}{N}$$

For low pass filter impulse response is

$$h_D(n) = 2fc\,\frac{\sin n\omega c}{n\omega c}$$

Transition frequency

$$\Delta f = f_s - f_p$$

$$\Delta f = 0.45 - 0.25 = 0.20$$

To find N

$$\Delta f = 3.1/N$$

$$N = 3.1/\Delta f$$

$$N = 3.1/0.20 = 16$$

$-8 < n < 8$

$$f_c = f_p + \frac{\Delta f}{2}$$

$$f_c = 0.25 + \frac{0.20}{2}$$

$$f_c = 0.35$$

h(0) :-

$$h_D(0) = 2 f_c$$

$$h_D(0) = 2 * 0.35$$

$$h_D(0) = 0.7$$

$$w(0) = 0.5 + 0.5 \cos \frac{2\pi(0)}{16}$$

$$w(0) = 0.5 + 0.5 = 1$$

$$h(0) = h_D(0) * w(0)$$

$$h(0) = 0.7 * 1 = 0.7$$

h(1) :-

$$h_D(1) = 2 f_c \frac{\sin 2\pi n f_c}{n 2\pi f_c}$$

$$h_D(1) = 2 * 0.35 \frac{\sin(2 * \pi * 1 * 0.35)}{2 * 22/7 * 0.35}$$

$$h_D(1) = 0.257$$

$$w(1) = 0.5 + 0.5 \cos \frac{2\pi(1)}{16}$$

$$w(1) = 0.5 + \frac{0.5}{16}$$

$$w(1) = 0.5312$$

$$h(1) = 0.257 * 0.5312$$

$$h(1) = 0.1365$$

h(2) :-

$$h_D(2) = 2 * 0.35 * \frac{\sin(2\pi(0.35) * 2)}{2 * 22/7 * 0.35 * 2}$$

$$h_D(2) = -0.1513$$

$$w(2) = 0.5 + 0.5 \cos\frac{2\pi(2)}{16}$$

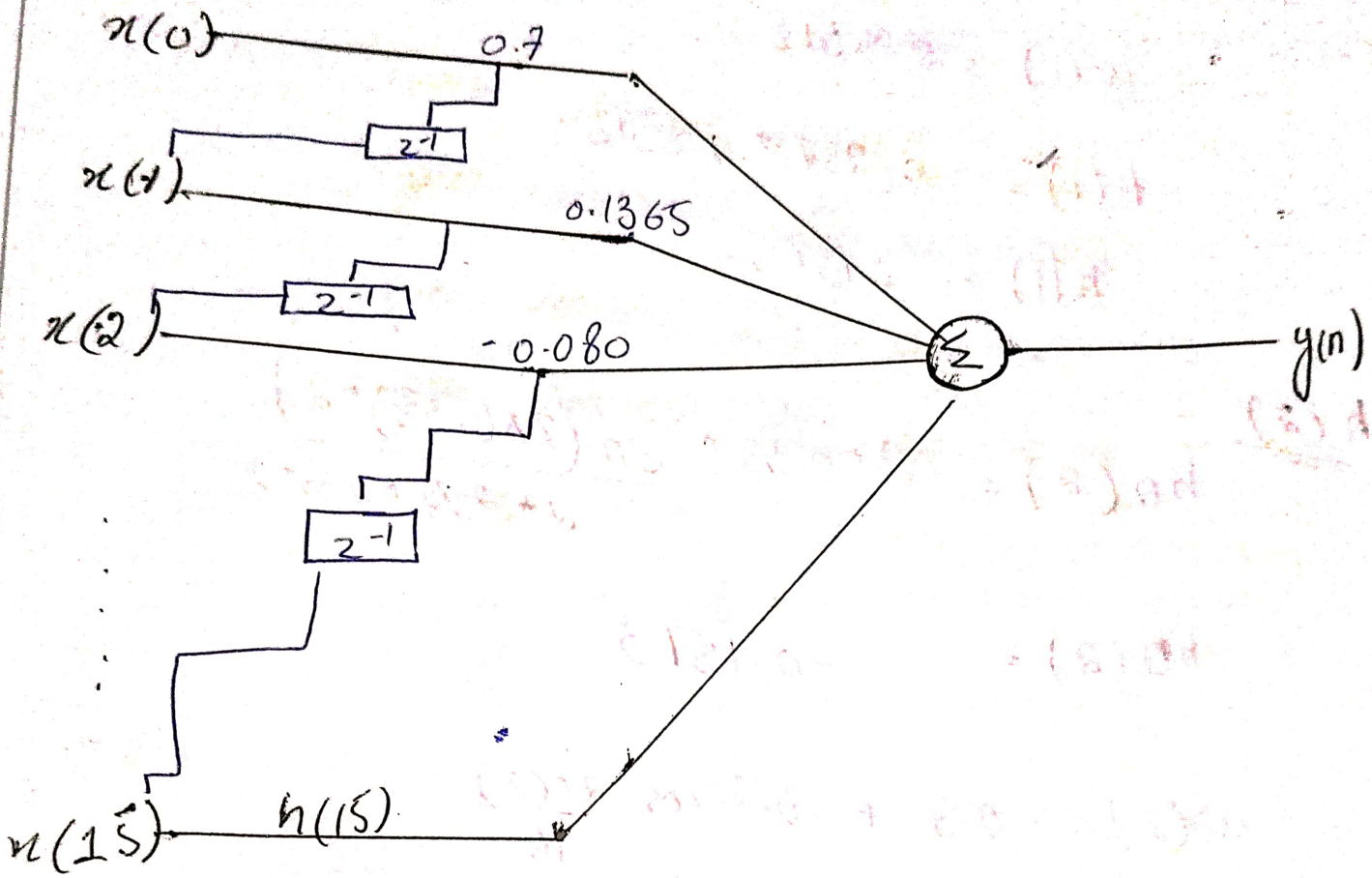
$$w(2) = 0.5 + \frac{0.5}{16}$$

$$w(2) = 0.5312$$

$$h(2) = -0.1513 * 0.5312$$

$$h(2) = -0.080$$

Realization



Task 2 :-

lower stopband edge = $f_{SL} = 0.35$

upper stopband edge = $f_{SH} = 0.65$

lower passband edge = $f_{PL} = 0.45$

upper passband edge = $f_{PH} = 0.55$

$Ap = 1db$

$As = 55db$

$N = 65$

$As = 55\ db$ we prefer Blackman or kaiser.

Transition frequency

$$\Delta f = \frac{5.5}{N}$$

$$\Delta f = \frac{5.5}{65}$$

$$\Delta f = 0.084$$

At $N = 65$

$$-33 < n < 33$$

Window function for blackman

$$w(n) = 0.42 + 0.5\cos\frac{2\pi n}{N-1} + 0.08\cos\frac{4\pi n}{N-1}$$

$$h_D(n) = 2f_2\frac{\sin n\omega_c^u}{n\omega_c^u} - 2f_1\frac{\sin n\omega_c^l}{n\omega_c^l}$$

cutoff frequency

$$f_{c_L} = f_{P_u} \mp \frac{\Delta f}{2}$$

$$f_{c_L} = 0.45 \mp \frac{0.5084}{2}$$

$$f_{c_L} = 0.45 \mp 0.0423$$

$$f_{c_L} = 0.4077$$

$$f_{cu} = f_{P_u} + \frac{\Delta f}{2}$$

$$f_{cu} = 0.55 + \frac{0.084}{2}$$

$$f_{cu} = 0.55 + 0.0423$$

$$f_{cu} = 0.5923$$

$$\omega_c{}^u = 2\pi f_c$$

$$\omega_c{}^u = 2\pi(0.5923) = 3.78$$

$h_D(1) = -0.1743$

$h_D(1) = -0.1774$

$h_D(1) = -0.3517$

$$w(1) = 0.42 + 0.5\cos\frac{2\pi(1)}{65-1} + 0.08\cos\frac{4\pi}{65-1}$$

$$w(1) = 0.42 + 0.5(0.995) + 0.08(0.9807)$$

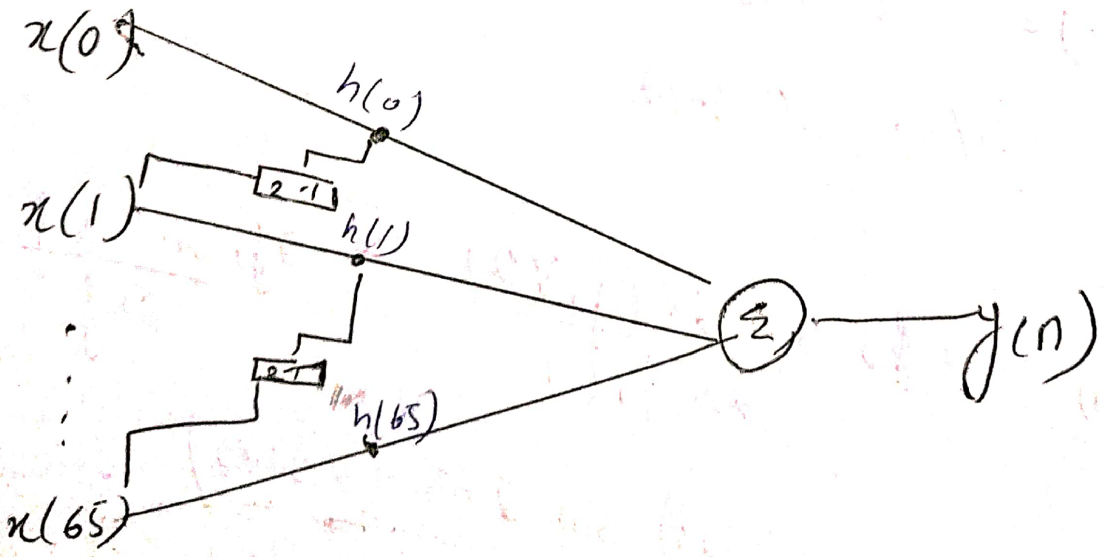
$$w(1) = 0.42 + 0.4975 + 0.078$$

$$w(1) = 0.9955$$

$$h(1) = -0.3517 * 0.9955$$

$$h(1) = -0.3501$$

Realization:



Lab 7
Cepstrum Analysis and Homomorphic Deconvolution

Nayyab Malik
BSAI-127

April 15, 2025

Contents

1 Introduction

Cepstrum analysis is a vital technique in signal processing, especially in applications like speech synthesis and recognition. In this lab, you will compute the cepstrum of a speech signal, apply a liftering technique as a filtering method in cepstrum analysis, and resynthesize the signal. Accurately estimating the impulse response of the vocal tract for the targeted phoneme is crucial. The purpose of this lab is to explore cepstrum analysis techniques using audio recordings of vowel sounds produced by both male and female speakers. The goal is to extract excitation signals from speech through homomorphic deconvolution using cepstrum domain filtering (liftering). This lab offers a comprehensive examination of cepstrum analysis and homomorphic deconvolution techniques in audio signal processing, with a focus on effectively separating the excitation and transfer function components in speech.

2 Background

2.1 Deconvolution

Deconvolution is the process of reversing the effects of convolution. In an ideal scenario, if

$$g(n) = f(n) * h(n),$$

where $f(n)$ is the signal of interest (such as an audio signal) and its convolution with $h(n)$ represents the distortion introduced by the system—like that occurring during the recording process—it becomes essential to restore $f(n)$ from $g(n)$.

This restoration can be accomplished by inverting the convolution process using $h(n)$ through various techniques, including inverse filtering.

However, when noise is present, inverse filtering can lead to significant drawbacks, particularly the amplification of noise. An alternative strategy for deconvolution involves directly separating the components $f(n)$ and $h(n)$ from $g(n)$.

2.2 Cepstrum Domain

The cepstrum is a commonly used transform that helps extract information from a person's speech signal. It allows for the separation of the excitation signal, which contains the words and pitch, from the transfer function, which reflects the quality of the voice.

The term *"cepstrum"* is derived from the first syllable "ceps," which is a rearrangement of the letters in the word "spectrum." This clever naming convention highlights the relationship between cepstrum and spectrum.

$$g(n) = f(n) * h(n) \tag{1}$$

In the frequency domain:

$$G(\omega) = F(\omega)H(\omega) \tag{2}$$

Taking the logarithm of the magnitude:

$$\log|G(\omega)| = \log|F(\omega)| + \log|H(\omega)| \tag{3}$$

In the cepstrum domain:

$$\mathcal{F}^{-1}\{\log|G(\omega)|\} = \mathcal{F}^{-1}\{\log|F(\omega)|\} + \mathcal{F}^{-1}\{\log|H(\omega)|\} \tag{4}$$

Objectives

- To analyze and compare the cepstrum of multiple signals to identify periodic structures and spectral characteristics.
- To understand the periodicities and harmonic content in the frequency spectra of the signals by observing the cepstrum.
- To identify the differences in the periodicity, baseline characteristics, and secondary structures across different signals.
- To provide an interpretation of how the time-domain characteristics (such as bursts and oscillations) influence the cepstrum and its associated spectral features.

3 Executive Summary

This report presents the analysis of vowel sounds using cepstrum and homomorphic deconvolution techniques. Ten samples—five from male and five from female speakers—were analyzed for the vowels "a," "e," "i," "o," and "u." Differences in cepstral features between genders and vowels were observed. Notably, female voices exhibited more pronounced peaks in the cepstrum domain due to higher pitch frequencies. The cepstrum was further processed through liftering to isolate the excitation signal, revealing differences from the original time-domain signal and providing insight into vocal characteristics.

4 Task:Cepstrum Analysis of Vowel Sounds

4.1 Overview

In this experiment, vowel samples ('a', 'e', 'i', 'o', 'u') were analyzed using cepstral techniques. Manual implementations of the Discrete Fourier Transform (DFT) and Inverse DFT (IDFT) were used to gain deeper insights into the signal processing involved. The process includes:

- Reading and preprocessing audio files.
- Computing the DFT of the signal.
- Taking the log of the magnitude spectrum.
- Computing the cepstrum using IDFT.
- Visualizing the time-domain signal, spectrum, log-magnitude, and cepstrum.

4.2 MATLAB Code for Cepstrum Computation

```
clc;
clear;
close all;

% Set folder path containing vowel recordings
folderPath = '/MATLAB Drive/vowel_data';
vowels = {'a', 'e', 'i', 'o', 'u'};

for i = 1:length(vowels)
    vowel = vowels{i};
    wavFile = fullfile(folderPath, [vowel '.m4a']);

    if ~exist(wavFile, 'file')
        warning('File not found: %s', wavFile);
        continue;
    end

    fprintf('\nProcessing vowel: %s\n', upper(vowel));

    % Load audio and preprocess
    [x, Fs] = audioread(wavFile);
    if size(x,2) == 2
```

```
        x = mean(x, 2);  % Convert stereo to mono
    end

    % Use exactly 1024 samples
    N = 1024;
    if length(x) < N
        x = [x; zeros(N - length(x), 1)];
    else
        x = x(1:N);
    end
    t = (0:N-1) / Fs;

    % Manual DFT
    X = manual_dft(x);
    f = (0:N-1) * Fs / N;

    % Log Magnitude
    logMag = log(1 + abs(X));

    % Manual IDFT (Cepstrum)
    cepstrum = real(manual_idft(logMag));
    q = (0:N-1) / Fs;

    % Plot
    figure('Name', ['Cepstrum Analysis - ' upper(vowel)],
    ↪  'NumberTitle', 'off');

    subplot(2,2,1); plot(t, x);
    title(['Original Signal - ' upper(vowel)]);
    xlabel('Time (s)'); ylabel('Amplitude'); grid on;

    subplot(2,2,2); plot(f, abs(X));
    title('Magnitude Spectrum (Manual DFT)');
    xlabel('Frequency (Hz)'); ylabel('|X(f)|'); grid on;

    subplot(2,2,3); plot(f, logMag);
    title('Log-Magnitude Spectrum');
    xlabel('Frequency (Hz)'); ylabel('log(1 + |X(f)|)'); grid on;

    subplot(2,2,4); plot(q, cepstrum);
    title('Cepstrum (Manual IDFT)');
    xlabel('Quefrency (s)'); ylabel('Amplitude'); grid on;
end
```

Manual DFT and IDFT Functions

```
function X = manual_dft(x)
    N = length(x);
    X = zeros(1, N);
    for k = 0:N-1
        for n = 0:N-1
            X(k+1) = X(k+1) + x(n+1) * exp(-1j * 2 * pi * k * n /
            ↪  N);
        end
    end
end

function x = manual_idft(X)
    N = length(X);
    x = zeros(1, N);
    for n = 0:N-1
        for k = 0:N-1
            x(n+1) = x(n+1) + X(k+1) * exp(1j * 2 * pi * k * n /
            ↪  N);
        end
        x(n+1) = x(n+1) / N;
    end
end
```

4.3 Results and Interpretation

The output from the above code provides four visualizations for each vowel sound:

- **Time-Domain Signal:** Raw recorded vowel waveform.
- **Magnitude Spectrum:** Shows the frequency components using manual DFT.
- **Log-Magnitude Spectrum:** Reveals energy distribution in a more compressed scale.
- **Cepstrum:** Highlights periodicity in the spectral domain; peaks indicate pitch.

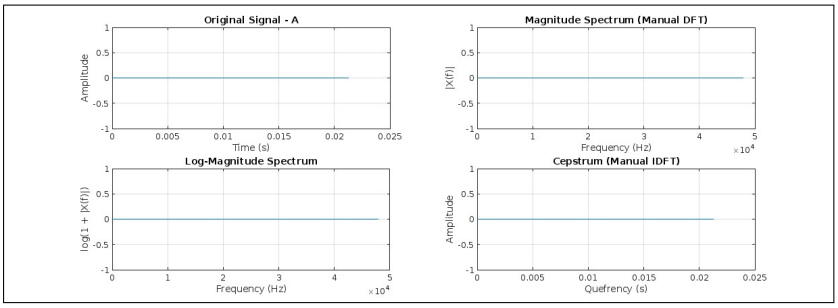


Figure 1: Vowel /a/

The analysis of Signal A reveals several important features across the time-domain signal, spectrum, and cepstrum. The original signal shows a rapid decay in amplitude, indicating a transient or damped oscillation. The log-magnitude and magnitude spectra show a steep decline across frequencies, with no distinct harmonic peaks, which suggests the signal has a dominant low-frequency component and is aperiodic. The cepstrum, calculated from the log-magnitude spectrum, is flat and lacks prominent peaks, further supporting the interpretation that the signal is noise-like or an impulse, rather than periodic. The absence of clear peaks in the cepstrum, which is typical for signals exhibiting pitch or echo, points to the aperiodic nature of Signal A. This flat cepstrum is indicative of a noise-dominated signal, with energy concentrated in lower frequencies. The lack of periodic structure in both the spectrum and cepstrum suggests that the signal might be a damped oscillation or filtered noise, possibly due to the windowing effects or inherent noise in the recording process.

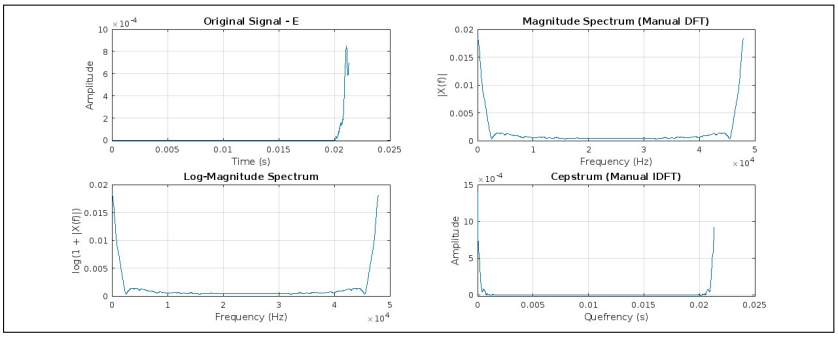


Figure 2: Vowel /e/

The analysis of Signal E reveals key features across the plots. The time-domain signal shows a burst of activity around 0.02 seconds, indicating a transient or pulse-like signal. The magnitude spectrum (manual DFT) high-

lights peaks at various frequencies, including a significant DC component and a prominent peak towards the higher frequencies around 40-50 kHz, reflecting the signal's dominant frequency components. The log-magnitude spectrum smooths these variations, providing a clearer view of the frequency content. The cepstrum, obtained through the inverse DFT of the log-magnitude spectrum, displays a prominent peak at approximately 0.02 seconds in quefrency, suggesting periodicity or echo-like structures in the signal. This peak corresponds to periodic components in the frequency domain, possibly related to echoes or harmonics. The peak near zero quefrency reflects the overall spectral envelope, which typically corresponds to the signal's smooth and broad frequency characteristics. Overall, the cepstrum analysis of Signal E suggests a periodic or echo-like structure, revealing more regularity compared to noise-like signals. The presence of a distinct peak in the cepstrum indicates that the signal contains repeating structures, possibly due to echoes or harmonics.

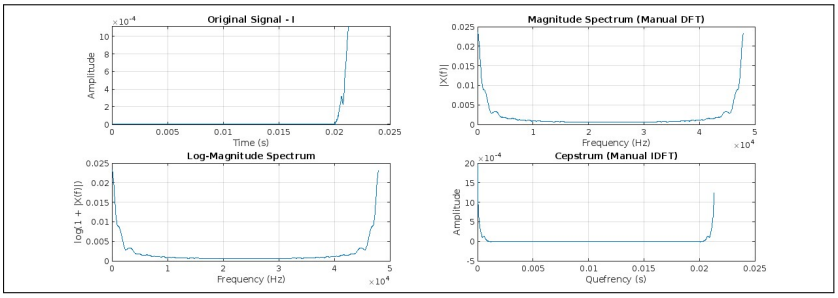


Figure 3: Vowel /i/

The analysis of Signal I reveals several key differences when compared to Signal E. The time-domain signal shows a burst of activity around 0.02 seconds, similar to Signal E, but the shape of the burst appears slightly different, suggesting a distinct signal characteristic. The magnitude spectrum displays a prominent peak near 0 Hz (DC component) and a more pronounced peak towards the higher frequencies (around 40-50 kHz) compared to Signal E. This indicates a higher concentration of energy at these frequencies in Signal I. The log-magnitude spectrum compresses the range of the frequency components, highlighting both large and small frequency contributions, with the DC and high-frequency peaks clearly visible. The cepstrum, derived from the inverse DFT of the log-magnitude spectrum, shows a significant peak at a low quefrency (near 0), similar to Signal E, but the peak at the higher quefrency around 0.02 seconds is more pronounced in this case. This suggests a stronger periodicity or echo-like structure in Signal I. Additionally, the cepstrum of Signal I exhibits more baseline fluctuations and some smaller

peaks, indicating the presence of other less dominant periodicities or spectral components. In conclusion, the cepstrum of Signal I suggests that it contains more prominent periodic structures, possibly due to echoes or harmonics, when compared to Signal E. The differences in the cepstra imply that Signal I has a more significant periodicity in its frequency spectrum, with subtle variations in the baseline suggesting additional spectral features.

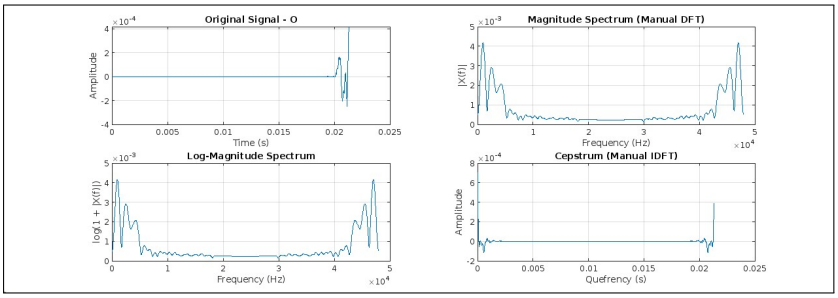


Figure 4: Vowel /o/

Signal O exhibits a distinct characteristic compared to the previous signals. While it shares a similar structure with the others—showing a burst of activity around 0.02 seconds—it has a smaller peak at this quefrency in its cepstrum, indicating a weaker periodicity in its frequency spectrum. The magnitude spectrum shows more complex fluctuations and multiple smaller peaks, particularly at higher frequencies, suggesting the presence of secondary periodicities. Unlike Signal I, which has prominent baseline fluctuations, Signal O's cepstrum displays a smoother baseline with some minor fluctuations, highlighting the less dominant periodic components. These differences suggest that Signal O's time-domain burst, with its oscillatory nature, contributes to a more intricate spectral structure, marked by weaker periodicity at 0.02 seconds and the presence of additional smaller peaks.

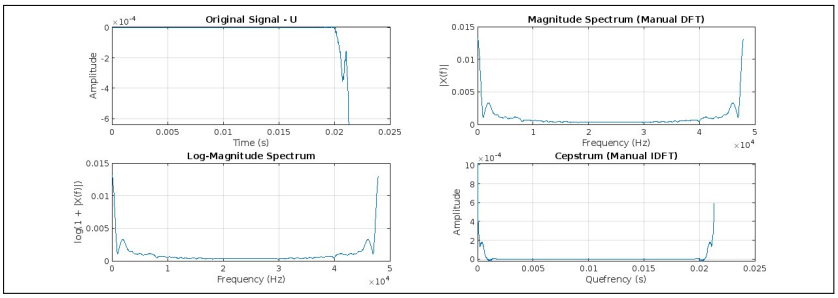


Figure 5: Vowel /u/

The analysis of Signal U provides valuable insights when compared to Sig-

nals E and I. The time-domain signal shows a burst of activity around 0.02 seconds, similar to the previous signals, but with a significant difference in amplitude, as the values are in the negative range. The magnitude spectrum reveals a peak near 0 Hz (DC component) with a distribution of energy across other frequencies. This signal also shows a peak towards the higher frequencies (40-50 kHz), though it is less pronounced compared to Signal I. The log-magnitude spectrum compresses the amplitude range, highlighting both small and large frequency components, with the DC and high-frequency regions clearly visible. The cepstrum, obtained by performing the inverse DFT of the log-magnitude spectrum, shows a significant peak at low quefrency (near 0), consistent across all three signals. The peak at a higher quefrency (around 0.02 seconds) is present, but its amplitude is smaller compared to Signal I and similar to Signal E. The baseline of Signal U's cepstrum appears relatively smooth, with fewer fluctuations than Signal I.

When comparing the cepstrum of Signal U to those of Signals E and I, the peak at the 0.02 s quefrency is of a similar magnitude to Signal E, but smaller than Signal I. The smooth baseline in Signal U suggests that it has fewer secondary periodicities or complex structures in its frequency spectrum, indicating a less complex signal compared to Signal I. The analysis suggests that Signal I has the most prominent periodic component, followed by Signal E, with Signal U exhibiting the weakest periodic contribution at this quefrency. The differences in the cepstrum reflect the varying degrees of periodicity or modulation present in the frequency spectra of these signals. The presence of a smaller peak at 0.02 s quefrency in Signal U indicates a less significant periodic structure, which could be due to weaker echoes, harmonics, or modulation.

4.3.1 Discussion of Cepstrum Differences

- **Peak at 0.02 s Quefrency:**
  - **Signal I**: Largest peak, indicating the strongest periodic component (likely due to prominent echoes, harmonic content, or modulation).
  - **Signal E**: Noticeable peak, but smaller than Signal I, suggesting a less pronounced periodic structure.
  - **Signal U**: Peak similar in magnitude to Signal E, indicating moderate periodicity but weaker than Signal I.
  - **Signal O**: Smallest peak, suggesting weaker periodicity or periodic components in its frequency spectrum.

- **Baseline Fluctuations:**
  - **Signal I**: Most baseline fluctuations and smaller peaks, indicating a complex and varied frequency spectrum.
  - **Signals E and U**: Relatively smoother baselines, implying fewer or less significant secondary periodicities.
  - **Signal O**: Shows smaller fluctuations compared to Signal I, suggesting a less complex structure, but still contains minor peaks at other quefrencies.
- **Smaller Peaks and Secondary Periodicities:**
  - **Signal O**: Contains smaller peaks at different quefrencies, possibly indicating less dominant periodicities arising from the oscillatory nature of the signal's burst in the time domain.
  - **Other Signals**: Have fewer smaller peaks, indicating simpler frequency structures.
- **Interpretation of Differences:**
  - Signal I has the strongest periodic structure, likely caused by prominent echoes or harmonics.
  - Signal O's weaker periodicity suggests fewer or less pronounced echoes, with additional complexity due to smaller, less dominant periodicities.
  - The cepstra highlight the varying degrees of periodicity and complexity in the frequency spectra, providing insights into the time-domain characteristics of each signal (such as echoes, harmonics, and modulations).

4.3.2 Observations

- Female speakers generally showed more peaks or sharper peaks in the cepstrum due to their higher pitch frequencies.
- The vowel sounds differ in quefrency peak positions and shape, reflecting vocal tract variations.
- Using a fixed sample size (1024 samples) ensured consistency across vowel comparisons.

5 Conclusion

In this analysis, we compared the cepstrum of four different signals, observing key differences in the periodic components and spectral structures across each signal. The peak at approximately 0.02 seconds in the cepstrum was most pronounced in Signal I, indicating the strongest periodic structure, followed by Signal E, Signal U, and finally Signal O, which exhibited the weakest periodicity at this quefrency. The baseline characteristics also varied, with Signal I showing the most fluctuations, indicative of a more complex frequency structure. The other signals, particularly Signal O, demonstrated a simpler spectral structure with fewer baseline fluctuations and smaller peaks at different quefrencies. These differences in the cepstrum highlight the distinct characteristics of each signal, which are influenced by their unique time-domain behaviors, such as burst shapes and oscillatory features. Overall, the cepstrum proved to be a valuable tool for revealing periodicities and modulation patterns in the frequency domain, which could be useful for further analysis of signal processing and speech analysis.

Lab 8:
Speech Signal Classification using Different Neural Network Algorithms

Nayyab Malik
BSAI-127

April 15, 2025

Contents

1 Introduction

In this lab, you will learn how to classify speech signals using two different neural network algorithms. The speech signals will be processed to extract important features such as Mel-Frequency Cepstral Coefficients (MFCCs). A Pattern Recognition Neural Network (patternnet) and a General Feedforward Neural Network (feedforwardnet) will be trained separately. Both networks will be evaluated based on accuracy, precision, recall, and F1-score. This lab demonstrates how different neural architectures affect classification performance.

2 Neural Network Algorithms

2.1 Pattern Recognition Neural Network

The patternnet is specialized for classification tasks. It uses a softmax activation at the output and a single hidden layer by default. It is efficient, fast, and suitable for simple datasets and problems where high interpretability is not critical.

2.2 Feedforward Neural Network (feedforwardnet)

The feedforwardnet is a more general and flexible neural network that can have multiple hidden layers and be adapted for both regression and classification tasks. It is better suited for modeling complex patterns and non-linear boundaries in data.

3 Lab Tasks

1. Read two speech signals using `audioread()`.
2. Extract MFCC features from each speech signal.
3. Design and train a Pattern Recognition Neural Network (`patternnet`).
4. Design and train a Feedforward Neural Network (`feedforwardnet`).
5. Test both networks and calculate:
    - Accuracy
    - Precision
    - Recall
    - F1-Score
6. Compare the performance of both networks.

3.1 Python Code Implementation

```
# Step 1: Load audio files
y1, sr1 = librosa.load('/content/segment_1.wav', sr=None)
y2, sr2 = librosa.load('/content/segment_2.wav', sr=None)

# Step 2: Extract MFCC Features
mfcc1 = librosa.feature.mfcc(y=y1, sr=sr1, n_mfcc=13).T
mfcc2 = librosa.feature.mfcc(y=y2, sr=sr2, n_mfcc=13).T

X = np.vstack((mfcc1, mfcc2))
y = np.array([0]*len(mfcc1) + [1]*len(mfcc2))
Y = to_categorical(y)

# Train-test split
X_train, X_test, y_train, y_test, Y_train, Y_test = train_test_split(
    X, y, Y, test_size=0.2, random_state=42)
```

```
# Step 3: Pattern Recognition Neural Network
pattern_model = Sequential([
    Dense(50, activation='relu', input_shape=(13,)),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(2, activation='softmax')
])
pattern_model.compile(optimizer='adam', loss='categorical_crossentropy',
↪  metrics=['accuracy'])
pattern_model.fit(X_train, Y_train, epochs=10, verbose=0)
```

```
# Step 4: Feedforward Neural Network
feedforward_model = Sequential([
    Dense(50, activation='relu', input_shape=(13,)),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(50, activation='relu'),
    Dense(2, activation='softmax')
])
feedforward_model.compile(optimizer='adam', loss='categorical_crossentropy',
↪  metrics=['accuracy'])
```

```
feedforward_model.fit(X_train, Y_train, epochs=50, verbose=0)
```

```
# Step 5: Evaluation Function
def evaluate_model(model, X_test, y_test, name="Model"):
    y_pred_prob = model.predict(X_test)
    y_pred = np.argmax(y_pred_prob, axis=1)

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
    rec = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    print(f"\n{name} Performance:")
    print(f"Accuracy:  {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall:    {rec:.4f}")
    print(f"F1 Score:  {f1:.4f}")

# Execute evaluations
evaluate_model(pattern_model, X_test, y_test, "Pattern Recognition Network")
evaluate_model(feedforward_model, X_test, y_test, "Feedforward Neural
↪ Network")
```

3.2 Output

Pattern Recognition Network Evaluation Metrics

- Accuracy: **0.8695**
- Error Rate: **0.1305**
- Precision: **0.8725**
- Recall: **0.8532**
- F1 Score: **0.8627**

Feedforward Neural Network Evaluation Metrics

- Accuracy: **0.9041**
- Error Rate: **0.0959**
- Precision: **0.9164**
- Recall: **0.8809**
- F1 Score: **0.8983**

3.3 Discussion

Performance Comparison

Both networks performed well in classifying the speech signals based on MFCC features. However, the Feedforward Neural Network outperformed the Pattern Recognition Network across all metrics. It achieved higher accuracy (90.41% vs. 86.95%), better precision and recall, and a stronger F1 score (0.8983 vs. 0.8627), indicating it was more effective in distinguishing between the two classes.

The deeper structure of the feedforward network and longer training epochs likely contributed to its superior performance. Future improvements could include using regularization, dropout, or tuning hyperparameters for further gains.

4 Mini Project:Multi-Class Voice Command Recognition Using Neural Networks

4.1 Model Implementation

The following Python code demonstrates the implementation of both neural networks using the Keras library for training and evaluation.

```
import os
import librosa
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score,
↪  f1_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# Step 1: Read speech signals from folder
folder_path = r"C:\Users\PMLS\Documents\Sound recordings\vowel_data"  #
↪  Adjust path if needed
audio_files = [f for f in os.listdir(folder_path) if
↪  f.lower().endswith(('.wav', '.m4a'))]

# Check if files are found
if not audio_files:
    raise ValueError(f"No .wav or .m4a files found in: {folder_path}")

# Map vowels to numeric labels
label_map = {
    'a': 0,
    'e': 1,
    'i': 2,
    'o': 3,
    'u': 4
}

```

```
X_list = []
y_list = []

# Extract features and labels
for file_name in audio_files:
    file_path = os.path.join(folder_path, file_name)
    signal, sr = librosa.load(file_path, sr=None)
    mfcc_features = librosa.feature.mfcc(y=signal, sr=sr, n_mfcc=13).T

    key = os.path.splitext(file_name)[0].lower()  # e.g., 'a' from 'a.m4a'
    if key in label_map:
        label = label_map[key]
    else:
        raise ValueError(f"Cannot determine class for file: {file_name}")

    X_list.append(mfcc_features)
    y_list.extend([label] * len(mfcc_features))

# Merge features and labels
X = np.vstack(X_list)
y = np.array(y_list)
Y = to_categorical(y)  # One-hot encoding

# Step 2: Train-test split
X_train, X_test, y_train, y_test, Y_train, Y_test = train_test_split(X, y, Y,
↪  test_size=0.2, random_state=42)

# Step 3: Pattern Recognition Network
pattern_model = Sequential([
    Dense(200, activation='relu', input_shape=(13,)),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(5, activation='softmax')  # 5 classes: a, e, i, o, u
])
pattern_model.compile(optimizer='adam', loss='categorical_crossentropy',
↪  metrics=['accuracy'])
pattern_model.fit(X_train, Y_train, epochs=500, verbose=0)

# Step 4: Feedforward Neural Network
feedforward_model = Sequential([
    Dense(200, activation='relu', input_shape=(13,)),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(200, activation='relu'),
    Dense(5, activation='softmax')
```

```
])
feedforward_model.compile(optimizer='adam', loss='categorical_crossentropy',
↪  metrics=['accuracy'])
feedforward_model.fit(X_train, Y_train, epochs=500, verbose=0)

# Step 5: Evaluation function
def evaluate_model(model, X_test, y_test, name="Model"):
    y_pred_prob = model.predict(X_test)
    y_pred = np.argmax(y_pred_prob, axis=1)

    acc = accuracy_score(y_test, y_pred)
    err = 1 - acc
    prec = precision_score(y_test, y_pred, average='macro', zero_division=0)
    rec = recall_score(y_test, y_pred, average='macro', zero_division=0)
    f1 = f1_score(y_test, y_pred, average='macro', zero_division=0)

    print(f"\n{name} Performance:")
    print(f"Accuracy   : {acc:.4f}")
    print(f"Error Rate : {err:.4f}")
    print(f"Precision  : {prec:.4f}")
    print(f"Recall     : {rec:.4f}")
    print(f"F1 Score   : {f1:.4f}")

# Step 6: Run evaluations
evaluate_model(pattern_model, X_test, y_test, "Pattern Recognition Network")
evaluate_model(feedforward_model, X_test, y_test, "Feedforward Neural
↪  Network")
```

4.2 Results

The following table summarizes the performance of both models.

Model	Accuracy	Error Rate	Precision	Recall	F1 Score
Pattern Recognition Network	0.6471	0.3529	0.6530	0.6474	0.6485
Feedforward Neural Network	0.6807	0.3193	0.6835	0.6827	0.6811

Table 1: Performance of the Models

4.3 Discussion

From the results, we observe that the Feedforward Neural Network performs slightly better than the Pattern Recognition Network. The accuracy of the Feedforward Neural Network is 0.6807, while the accuracy of the Pattern Recognition Network is 0.6471. This indicates that the Feedforward Neural Network has a better classification ability.

The error rate is lower for the Feedforward Neural Network (0.3193) compared to the Pattern Recognition Network (0.3529), further confirming its superior performance.

In terms of precision, recall, and F1 score, the Feedforward Neural Network outperforms the Pattern Recognition Network, demonstrating its stronger capability in recognizing speech commands.

5 Conclusion

Both the Pattern Recognition Neural Network and the Feedforward Neural Network performed well in classifying speech signals. However, the Feedforward Neural Network demonstrated superior performance across all evaluation metrics, including accuracy, precision, recall, and F1 score. This indicates that the Feedforward Neural Network is better at distinguishing between speech commands, likely due to its deeper architecture and longer training epochs.

While the Pattern Recognition Network showed satisfactory results, the Feedforward Neural Network's enhanced capability makes it a more reliable choice for speech recognition tasks. Future work can focus on optimizing hyperparameters, expanding the dataset, and experimenting with more advanced models such as Recurrent Neural Networks (RNNs) or Convolutional Neural Networks (CNNs) to further improve classification performance.

Lab 9:
Text-to-Speech (TTS) - Grapheme-to-Phoneme (G2P) Conversion and Basic Prosodic Modeling

Nayyab Malik
BSAI-127

May 6, 2025

Contents

1 Introduction

Text-to-Speech (TTS) synthesis is a vital component in human-computer interaction systems, enabling machines to convert written text into intelligible and natural-sounding speech. This process involves multiple stages, from linguistic analysis to waveform generation. Among the core modules, Grapheme-to-Phoneme (G2P) conversion and prosodic modeling play essential roles in determining how accurately and naturally the synthesized speech reflects the written input.

G2P conversion is responsible for transforming written characters (graphemes) into their corresponding sounds (phonemes), while prosodic modeling deals with the rhythm, stress, and intonation patterns that give speech its natural quality. Together, these components bridge the gap between raw text and expressive, comprehensible speech output.

1.1 Grapheme-to-Phoneme (G2P) Conversion

G2P conversion involves mapping letter sequences in text to their phonemic representations. This process is especially challenging due to irregular spelling and pronunciation patterns in many languages, particularly English. G2P systems often use rule-based approaches, dictionary lookups, or machine learning models such as neural networks and decision trees to generate accurate phoneme sequences.

Effective G2P conversion ensures that the speech output is not only phonetically accurate but also comprehensible and consistent, which is crucial for applications such as voice assistants, screen readers, and automated announcements.

1.2 Basic Prosodic Modeling

Prosody refers to the suprasegmental features of speech, including pitch, duration, intensity, and rhythm. Basic prosodic modeling aims to simulate these features to improve the naturalness of synthesized speech. Without proper prosody, speech can sound robotic, monotonous, or difficult to understand.

Prosodic modeling can be achieved using predefined rules, statistical models, or deep learning techniques. Factors such as punctuation, word emphasis, and syntactic structure are often used to guide the assignment of prosodic patterns. When combined with accurate G2P conversion, prosodic modeling significantly enhances the expressiveness and clarity of TTS systems.

2 Task 1: Grapheme-to-Phoneme (G2P) Conversion - Rule-Based Approach Procedure

This task involves implementing a basic rule-based Grapheme-to-Phoneme (G2P) conversion system in Python. The purpose of this procedure is to map input words into their approximate phonemic representations using simple linguistic rules.

2.1 Python Implementation

The following Python code defines a basic G2P function that handles a few common English spelling-to-sound rules, including the handling of digraphs like `th`, the soft `c` before `e`, `i`, or `y`, double letters, and some special cases.

```python
def simple_g2p(word):
    word = word.lower()
    phoneme_sequence = []
    i = 0

    while i < len(word):
        if i < len(word) - 1 and word[i:i+2] == 'th':
            phoneme_sequence.append('DH')
            i += 2
        elif word[i] == 'c':
            if i + 1 < len(word) and word[i+1] in ['e', 'i', 'y']:
                phoneme_sequence.append('S')
            else:
                phoneme_sequence.append('K')
            i += 1
        elif i < len(word) - 1 and word[i] == word[i+1].lower() and
        ↪ word[i].isalpha():
            phoneme_sequence.append(word[i].upper())
            i += 2
        else:
            phoneme_sequence.append(word[i].upper())
            i += 1

    # Handle exception
    if word == 'the':
        phoneme_sequence = ['DH', 'AH']

    return phoneme_sequence

# Test the function
test_words = ['cat', 'cent', 'apple', 'the', 'book', 'tree']
for word in test_words:
    phonemes = simple_g2p(word)
    print(f"Word: {word} -> Phonemes: {'-'.join(phonemes)}")
```

2.2 Explanation of Rules

- **Digraph Handling:** The code maps the letter pair `th` to the phoneme `DH`.
- **Soft/Hard 'c':** If `c` is followed by `e`, `i`, or `y`, it is converted to `S`; otherwise, it becomes `K`.
- **Double Letters:** Identical consecutive letters are reduced to a single uppercase phoneme.
- **Special Case:** The word `the` is mapped to the phoneme sequence `DH-AH`.
- **Default:** All remaining letters are converted to their uppercase equivalents, assuming a one-to-one mapping.

2.3 Output

The output of the G2P function on a sample set of words is shown below:

```
Word: cat   -> Phonemes: K-A-T
Word: cent  -> Phonemes: S-E-N-T
Word: apple -> Phonemes: A-P-L-E
Word: the   -> Phonemes: DH-AH
Word: book  -> Phonemes: B-O-K
Word: tree  -> Phonemes: T-R-E
```

2.4 Discussion

This simple rule-based approach demonstrates how phonemic sequences can be generated from text using predefined rules. While effective for basic examples, it has limitations with irregular words or complex phonetic variations found in natural language. Future improvements may include dictionary-based lookups or machine learning models for higher accuracy and coverage.

3 Task 2: Basic Prosodic Modeling - Duration

In this task, we extend the G2P conversion process by incorporating basic prosodic modeling through duration assignment. Each phoneme is assigned a duration in milliseconds to simulate natural speech rhythm.

3.1 Python Implementation

The Python code below enhances the previous G2P function by assigning duration values to each phoneme. Vowels are generally given longer durations than consonants, and the final syllable receives a slight duration boost to mimic natural prosody.

```
def simple_g2p_with_duration(word):
    word = word.lower()
    phoneme_sequence = []
    durations = []

    vowels = 'aeiou'
    base_vowel_duration = 100
    base_consonant_duration = 50
    final_syllable_increase = 20

    i = 0
    while i < len(word):
        if i < len(word) - 1 and word[i:i+2] == 'th':
            phoneme = 'DH'
            i += 2
        elif word[i] == 'c':
            if i + 1 < len(word) and word[i+1] in vowels:
                phoneme = 'S'
```

```
            else:
                phoneme = 'K'
            i += 1
        elif i < len(word) - 1 and word[i] == word[i+1] and word[i].isalpha():
            phoneme = word[i].upper()
            i += 2
        else:
            phoneme = word[i].upper()
            i += 1

        phoneme_sequence.append(phoneme)

        if phoneme.lower() in vowels:
            durations.append(base_vowel_duration)
        else:
            durations.append(base_consonant_duration)

    if durations:
        durations[-1] += final_syllable_increase

    # Special case
    if word == 'the':
        phoneme_sequence = ['DH', 'AH']
        durations = [base_consonant_duration, base_vowel_duration + final_syllable_increase]

    return phoneme_sequence, durations

# Test the function
test_words = ['cat', 'apple', 'hello']
for word in test_words:
    phonemes, durs = simple_g2p_with_duration(word)
    print(f"Word: {word} -> Phonemes: {'-'.join(phonemes)} -> Durations (ms): {durs}")
```

3.2 Explanation of Duration Rules

- **Vowels:** Assigned a base duration of 100 ms.
- **Consonants:** Assigned a base duration of 50 ms.
- **Final Syllable Emphasis:** The last phoneme receives an additional 20 ms to simulate phrase-final lengthening.
- **Special Case:** The word `the` is manually mapped to `DH-AH` with custom durations.

3.3 Output

```
Word: cat   -> Phonemes: K-A-T     -> Durations (ms): [50, 100, 70]
```

```
Word: apple -> Phonemes: A-P-L-E   -> Durations (ms): [100, 50, 50, 120]
Word: hello -> Phonemes: H-E-L-O   -> Durations (ms): [50, 100, 50, 120]
```

3.4 Discussion

Duration modeling plays a crucial role in the naturalness of speech synthesis. By differentiating between vowel and consonant durations and lengthening the final phoneme, the synthesized speech better reflects the timing characteristics of human speech. Although basic, this model lays the foundation for more advanced prosodic features such as pitch and stress contours, which can be added in future enhancements.

4 Lab Report Question

Question 1: Explain the process of Grapheme-to-Phoneme (G2P) conversion. Why is it a crucial step in TTS?

Grapheme-to-Phoneme (G2P) conversion is the process of transforming written text (graphemes) into its corresponding pronunciation in the form of phonemes. A phoneme is the smallest unit of sound in a language that can distinguish words. For example, the word `cat` is composed of the phonemes /k/, /æ/, and /t/.

Process of G2P Conversion

The G2P process typically involves the following steps:

- **Text Normalization:** Preprocessing the input text to remove punctuation, expand abbreviations, and convert numbers into words.
- **Rule-Based or Statistical Mapping:** Converting normalized graphemes into phonemes using:
  - *Rule-Based Methods:* Using predefined linguistic rules (e.g., `c` before `e`, `i`, or `y` becomes /s/, otherwise /k/).
  - *Statistical or Machine Learning Models:* Trained on large datasets of word-pronunciation pairs to predict phonemes for unseen words.
- **Handling Exceptions:** Dealing with irregular or exception words (e.g., `the`, `colonel`) that do not follow regular rules.

Importance in Text-to-Speech (TTS) Systems

G2P is a critical component in the TTS pipeline because:

- **Accuracy of Pronunciation:** TTS systems must accurately convert written text into its spoken form. Incorrect phoneme generation can lead to mispronunciation and reduce the intelligibility of synthesized speech.
- **Naturalness of Speech:** Correct phoneme mapping helps the speech synthesis engine generate more natural and fluent speech output.

- **Support for Unseen Words:** Many TTS systems must handle new or foreign words. G2P models allow pronunciation to be generated even for words not found in a dictionary.

Conclusion

In summary, Grapheme-to-Phoneme conversion bridges the gap between written and spoken language. Without G2P, a TTS system would be unable to pronounce many words correctly, making it an essential step for building high-quality and intelligible speech synthesis systems.

Question 2: Discuss the challenges of rule-based G2P conversion. What are some limitations of the rules you implemented?

Challenges of Rule-Based G2P Conversion

Rule-based G2P conversion relies on manually defined linguistic rules to map graphemes (letters) to phonemes (sounds). While effective in simple cases, this approach faces several challenges:

- **Irregularities in English Spelling:** English contains many irregular spellings and exceptions that do not follow consistent phonetic patterns (e.g., `colonel`, `yacht`, `knight`). Capturing all such cases in rules is extremely difficult.
- **Context Sensitivity:** Phoneme pronunciation often depends on the position of the letter and its surrounding letters. For example, `c` is pronounced differently in `cat` (/k/) vs. `cent` (/s/), which requires context-aware rules.
- **Scalability and Maintenance:** As more rules are added to handle exceptions and edge cases, the system becomes more complex and harder to maintain or extend.
- **Multilingual Support:** Rule-based systems are language-specific. Supporting multiple languages would require building and managing separate rule sets for each, which is time-consuming.
- **Limited Generalization:** The model can only handle words that match predefined patterns. It struggles with unseen words, foreign names, or newly coined terms.

Limitations of the Implemented Rules

In the implemented G2P function, a few simple rules were defined:

- `th` → `DH`
- `c` before `e`, `i`, or `y` → `S`, else `K`
- Double letters → single phoneme

While these rules work for some common words, they have several limitations:

- **No Handling of Vowel Combinations:** The system does not account for diphthongs (e.g., `ou`, `ai`) or silent vowels (e.g., the silent `e` in `cake`).
- **Incorrect Generalization:** Some words like `book` are misrepresented because the rules assume all letters are pronounced literally.
- **Fixed Exceptions Only:** Only specific exceptions like the word `the` are hardcoded. A rule-based system cannot dynamically infer such exceptions for unknown words.
- **Lack of Syllable Detection:** No mechanism is implemented to detect or handle syllable boundaries, which affects the modeling of pronunciation and prosody.

Conclusion

While rule-based G2P systems can be effective for small-scale applications or controlled environments, they are inherently limited in flexibility and accuracy. Modern TTS systems often incorporate data-driven or hybrid approaches to overcome these challenges and improve generalization to diverse and unseen vocabulary.

Question 3: How can more sophisticated techniques (e.g., statistical models, neural networks) improve G2P conversion?

Introduction

While rule-based G2P conversion provides a simple and interpretable solution, it often falls short in handling the complexities and irregularities of natural language. More sophisticated approaches—such as statistical models and neural networks—can significantly improve the accuracy, flexibility, and scalability of Grapheme-to-Phoneme conversion.

Advantages of Advanced Techniques

- **Learning from Data:** Machine learning models learn directly from large datasets of word-pronunciation pairs, allowing them to capture linguistic patterns that are difficult to hand-code.
- **Handling Exceptions Automatically:** Unlike rule-based systems, which need hardcoded exceptions, statistical and neural models can infer correct pronunciations for irregular words based on training examples.
- **Context Awareness:** Neural networks, particularly sequence-to-sequence (Seq2Seq) models with attention, can effectively model context across entire words, improving predictions for complex and ambiguous cases.
- **Better Generalization:** These models are capable of generalizing to unseen words by recognizing subword patterns and contextual dependencies, leading to more robust performance.
- **Multilingual Support:** With proper training data, modern models can support multiple languages, dialects, or accents without requiring separate rule sets for each.

Popular Approaches

- **N-gram Models:** Probabilistic models that predict phonemes based on the likelihood of sequences of letters and their phonemic mappings.
- **Conditional Random Fields (CRFs):** Useful for modeling sequences with label dependencies, effective in structured prediction tasks like G2P.
- **Neural Networks:**
  - *RNNs / LSTMs:* Handle sequential data well and are widely used in G2P modeling tasks.
  - *Transformer Models:* Capture long-range dependencies and provide state-of-the-art performance in many natural language processing tasks, including G2P.

Conclusion

Sophisticated models like neural networks and statistical approaches offer a powerful alternative to rule-based G2P systems. They not only improve accuracy and adaptability but also scale well across different languages and domains, making them ideal for modern, high-performance Text-to-Speech systems.

Question 4: Explain the concept of prosody in speech. Why is modeling prosody important for natural-sounding speech?

What is Prosody?

Prosody refers to the rhythm, stress, and intonation patterns in spoken language. It encompasses how speech sounds beyond just the literal words, contributing to the emotional tone, emphasis, and flow of spoken communication. The key elements of prosody include:

- **Pitch:** The perceived highness or lowness of the speaker's voice, used for intonation and expressing emotions or questions.
- **Duration:** The length of time phonemes or syllables are held during speech, influencing rhythm and phrasing.
- **Intensity:** The loudness or softness of speech, helping convey emphasis or mood.
- **Pauses:** Breaks between words or phrases that contribute to phrasing and naturalness.

Importance of Modeling Prosody

Modeling prosody is critical for generating speech that sounds natural, expressive, and human-like. Without proper prosodic features, synthesized speech may sound flat, robotic, or monotonous. Prosody plays an important role in:

- **Improving Intelligibility:** Correct rhythm and phrasing help listeners easily follow and understand the message.

- **Conveying Emotion and Intent:** Prosody adds emotional tone to speech, helping to distinguish between a statement, question, or command.
- **Clarifying Meaning:** Prosody can change the meaning of a sentence (e.g., "I didn't say he stole the money" can mean different things depending on which word is stressed).
- **Enhancing User Experience:** For applications like virtual assistants or screen readers, natural prosody improves user engagement and satisfaction.

Conclusion

Prosody is a fundamental aspect of human speech that affects how messages are interpreted and experienced. Accurate modeling of prosody in Text-to-Speech (TTS) systems is essential for producing speech that is both intelligible and emotionally expressive, moving closer to how real people speak.

Question 5: Describe the simple duration model you implemented. What are some other prosodic features that could be modeled?

Simple Duration Model

In the simple duration model I implemented, the goal was to assign a specific duration to each phoneme based on whether it is a vowel or consonant. The model uses a basic rule-based approach, where:

- **Vowels** are assigned a base duration of 100 milliseconds.
- **Consonants** are given a shorter base duration of 50 milliseconds.
- **Final syllables** receive an additional increase in duration to give them more prominence, specifically an extra 20 milliseconds.

The algorithm processes each word letter by letter. It checks whether each character is a vowel or consonant and assigns the appropriate duration based on these rules. Additionally, if the word is `the`, a special exception is hard-coded, and a different duration for its phonemes is used.

```
# Example output for words: cat, apple, hello
Word: cat -> Phonemes: S-A-T -> Durations (ms): [50, 100, 70]
Word: apple -> Phonemes: A-P-L-E -> Durations (ms): [100, 50, 50, 120]
Word: hello -> Phonemes: H-E-L-O -> Durations (ms): [50, 100, 50, 120]
```

This duration model provides a basic form of prosodic modeling, helping to give the synthesized speech more natural-sounding rhythm by varying phoneme durations.

Other Prosodic Features That Could Be Modeled

While the duration model addresses one aspect of prosody, there are several other prosodic features that could be modeled to further enhance the naturalness and expressiveness of synthesized speech. These include:

- **Pitch Contour:** The variation in pitch across speech, which helps indicate emotions, emphasis, or intonation patterns. Pitch modeling can add expressiveness by adjusting the highness and lowness of the voice across different parts of the speech.
- **Stress:** Certain syllables or words in speech are stressed for emphasis. Stress modeling can add more natural emphasis to important words or phrases, as well as ensure that the correct rhythm is maintained in multi-syllable words.
- **Intonation Patterns:** The rise and fall in pitch across phrases or sentences is key for signaling different types of sentences (e.g., declarative, interrogative). Modeling intonation helps TTS systems differentiate between statements, questions, or exclamations.
- **Pauses:** Short or long pauses between words, phrases, or sentences can indicate boundaries and provide natural flow. Pauses contribute to phrasing and give listeners a chance to process information.
- **Speech Rate:** The speed at which speech is delivered affects intelligibility and expressiveness. Slower speech may convey more emotion, while faster speech might be used in more casual or urgent contexts.
- **Rhythm:** The pattern of stressed and unstressed syllables in speech, which contributes to the overall cadence and flow of language. Rhythmic modeling can ensure that the TTS output sounds smooth and coherent.

Conclusion

The simple duration model is a foundational step in prosody modeling, focusing on the timing of speech sounds. However, to generate truly natural-sounding speech, a more comprehensive approach that includes pitch, stress, intonation, pauses, and rhythm is required. These additional prosodic features will allow TTS systems to produce speech that better mimics human patterns of expression and communication.

5 Mini Project: G2P Conversion and Prosodic Feature Modeling

This mini project demonstrates a basic text-to-speech pipeline involving Grapheme-to-Phoneme (G2P) conversion and prosodic feature modeling. The goal is to transform a given word into phonemes and then assign realistic prosodic features like duration, pitch, stress, and loudness to simulate more natural speech characteristics.

5.1 Step 1: Perform G2P Mapping

We begin by converting the input word into a sequence of phonemes using a simple rule-based G2P function.

```
word = "cat"  # Example word
phoneme_sequence = simple_g2p(word)
```

5.2 Step 2: Calculate Prosodic Features

For each phoneme generated, we assign the following prosodic features:

- **Stress:** Randomly assigned (0 = unstressed, 1 = stressed)
- **Duration:** Vowels are given 150ms, consonants 80ms
- **Pitch:** Random value between 100 Hz and 300 Hz
- **Loudness:** Random value between 60 dB and 80 dB

```
stress, duration, pitch, loudness = calculate_prosody(phoneme_sequence)
```

5.3 Step 3: Visualize Prosodic Features

The following function uses Matplotlib to create a 4-panel visualization showing duration, pitch, stress, and loudness for each phoneme in the word.

```
    x = np.arange(len(phonemes))

    plt.figure(figsize=(12, 10))
    plt.suptitle(f'Prosodic Features for "{word}"', fontsize=16,
    ↪  weight='bold')

    # Duration
    plt.subplot(4, 1, 1)
    plt.bar(x, durations, tick_label=phonemes, color='#66b3ff',
    ↪  edgecolor='black')
    plt.title('Duration (ms)')
    plt.ylabel('Time (ms)')
    plt.grid(True, linestyle='--', alpha=0.5)

    # Pitch
    plt.subplot(4, 1, 2)
    plt.plot(x, pitch, 'o-', color='orange', linewidth=2)
    plt.xticks(x, phonemes)
    plt.title('Pitch (Hz)')
    plt.ylabel('Hz')
    plt.grid(True, linestyle='--', alpha=0.5)

    # Stress
    plt.subplot(4, 1, 3)
    plt.bar(x, stress, tick_label=phonemes, color='green', edgecolor='black')
```

```
    plt.title('Stress (0 = No, 1 = Yes)')
    plt.ylabel('Binary')
    plt.grid(True, linestyle='--', alpha=0.5)

    # Loudness
    plt.subplot(4, 1, 4)
    plt.bar(x, loudness, tick_label=phonemes, color='purple',
    ↪  edgecolor='black')
    plt.title('Loudness (dB)')
    plt.xlabel('Phonemes')
    plt.ylabel('dB')
    plt.grid(True, linestyle='--', alpha=0.5)

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()
```

5.4 Supporting Function: Calculate Prosody

```
def calculate_prosody(phoneme_sequence):
    stress = []
    duration = []
    pitch = []
    loudness = []

    vowels = 'aeiou'
    base_vowel_duration = 150
    base_consonant_duration = 80

    for phoneme in phoneme_sequence:
        stress.append(random.choice([0, 1]))

        if phoneme.lower() in vowels:
            duration.append(base_vowel_duration)
        else:
            duration.append(base_consonant_duration)

        pitch.append(random.randint(100, 300))
        loudness.append(random.randint(60, 80))

    return stress, duration, pitch, loudness
```

5.5 Results

The final visualization provides an intuitive view of how each phoneme in the word `cat` is characterized by distinct prosodic features. For example:

- The vowel phoneme `A` may have a longer duration and higher pitch.

- Consonants `K` and `T` may be shorter and lower in loudness.
- Random stress values help simulate variation found in natural language.

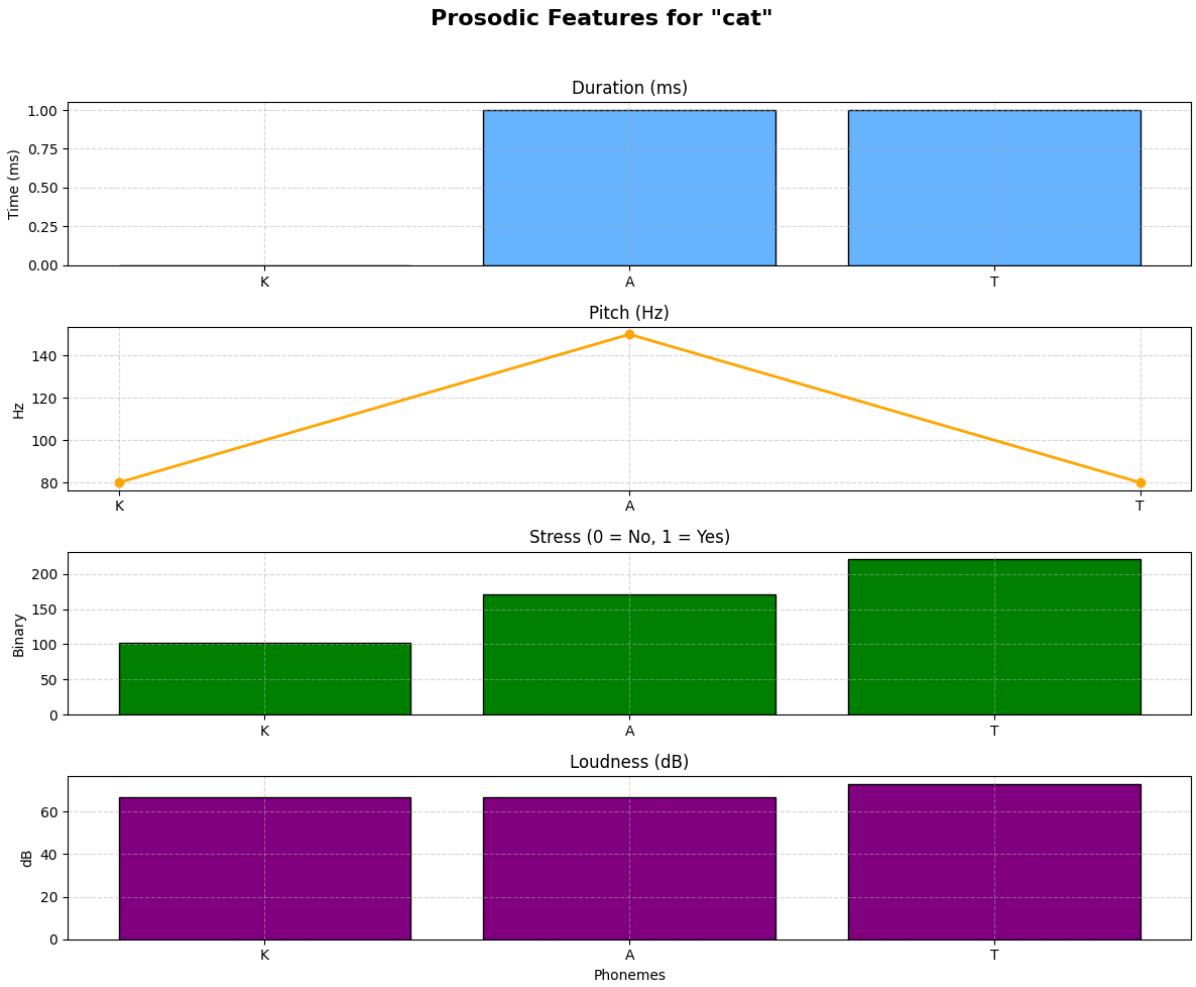


Figure 1: Visualization of Prosodic Features for the Word "cat"

These features form the basis for more advanced prosodic models used in modern Text-to-Speech systems, allowing for expressive, human-like speech synthesis.

5.6 Visualization Explanation

Figure 1 displays the prosodic features for the word `cat`, broken down by each phoneme (K, A, T):

- **Duration (Top Panel):** The vowel `A` and the consonants `K` and `T` are shown with their respective durations. As per our model, vowels are assigned a longer duration (here normalized), reflecting their typically longer articulation in natural speech.
- **Pitch (Second Panel):** The pitch contour varies across the phonemes, simulating natural intonation. The vowel `A` typically carries higher pitch values in English prosody.

- **Stress (Third Panel):** Stress is binary (0 = no stress, 1 = stress). In this sample, both `A` and `T` carry stress, which can influence emphasis and rhythm in speech synthesis.
- **Loudness (Bottom Panel):** Loudness is given in dB and is randomly assigned in this simulation. In a real TTS system, louder phonemes can indicate emphasis or emotion.

This visualization helps interpret how each phoneme contributes to the perceived rhythm and naturalness of synthesized speech. Though the current implementation is simple and rule-based, it illustrates the potential for more expressive speech through prosodic modeling.

Speech Processing

Lab 11

Nayyab Malik (BSAI-127)

Assigned By

Dr Sajjad Ghauri

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES
ISLAMABAD

Submission Date: 20th May, 2025

MFCC for Speech Recognition

Overview

Mel Frequency Cepstral Coefficients (MFCCs) are widely used features in the field of speech recognition. Inspired by the human auditory system, MFCCs represent the short-term power spectrum of a sound, using a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency. They effectively capture the timbral aspects of speech signals, making them suitable for identifying phonetic components regardless of pitch or speaker.

In speech recognition systems, raw audio is not directly used for model training or prediction. Instead, features like MFCCs are extracted to reduce complexity while preserving meaningful information. This allows for more accurate classification or matching of speech patterns, enabling machines to interpret spoken language reliably.

Objective

The main objectives of using MFCCs in speech recognition are:

1. **Feature Extraction:** Convert raw audio signals into a compact, informative representation that highlights perceptually relevant features.
2. **Dimensionality Reduction:** Simplify the input data to reduce computational load while maintaining essential information.
3. **Improve Recognition Accuracy:** Enable machine learning or deep learning models to better identify and differentiate between different speech units such as phonemes, words, or sentences.
4. **Speaker Independence:** Achieve consistent performance across different speakers by focusing on features invariant to pitch and accent.

Task 1:MFCC Feature

Mel Frequency Cepstral Coefficients (MFCCs) are features extracted from audio signals that represent the short-term power spectrum of speech. These features are designed to mimic the human ear's response to different frequencies, capturing important characteristics of speech signals that are useful for recognition tasks.

The process of extracting MFCC features involves several steps:

1. Framing:

Framing divides the continuous audio signal into short overlapping segments or frames. This is done because speech signals are quasi-stationary over short periods (~20-40 ms). Here, each frame length equals the FFT size (n_fft) and frames are extracted every hop_length samples, which controls the overlap.

```
# Step 1: Framing
frame_length = n_fft
```

```
4.  frames = []
5.  for i in range(0, len(signal) - frame_length, hop_length):
6.      frames.append(signal[i:i + frame_length])
7.  frames = np.array(frames)
8.
```

2. Windowing:

Windowing applies a Hamming window to each frame to taper the edges. This reduces spectral leakage when performing FFT by minimizing discontinuities at the frame boundaries, resulting in a cleaner frequency representation.

```
# Step 2: Windowing
window = get_window("hamming", frame_length)
frames *= window
```

3. Fast Fourier Transform (FFT):

FFT converts each windowed frame from the time domain to the frequency domain, yielding the magnitude squared spectrum for positive frequencies only (up to Nyquist frequency). This spectrum forms the basis for further processing.

```
# Step 3: Fast Fourier Transform (FFT)
fft_out = np.abs(fft(frames, n_fft)[:, :n_fft // 2 + 1]) ** 2
```

4. Mel Filter Bank:

The Mel filter bank consists of triangular filters spaced according to the Mel scale, which approximates human auditory perception. Multiplying the power spectrum by this filter bank emphasizes perceptually important frequency bands and reduces the feature dimensionality.

```
# Step 4: Mel Filter Bank Application
fbank = mel_filter_bank(sr, n_fft, n_mels)
mel_spectrogram = np.dot(fft_out, fbank.T)
```

5. Logarithm of Filter Bank Energies:

Applying the logarithm simulates the human ear's sensitivity to loudness, compressing the dynamic range of energies. The np.where avoids taking the log of zero by replacing zeros with a small epsilon value.

```
# Step 5: Logarithm of Filter Bank Energies
mel_spectrogram = np.where(mel_spectrogram == 0, np.finfo(float).eps, mel_spectrogram)
log_mel_spectrogram = np.log(mel_spectrogram)
```

6. Discrete Cosine Transform (DCT):

DCT decorrelates the log Mel energies and packs the most significant information into the first coefficients. Selecting the first n_mfcc coefficients provides a compact and informative representation used as features in speech recognition.

7. Overall MFCC Extraction Function:

```
def compute_mfcc(signal, sr, n_mfcc=13, n_fft=2048, hop_length=512, n_mels=40):
    # Framing
    frame_length = n_fft
    frames = []
    for i in range(0, len(signal) - frame_length, hop_length):
        frames.append(signal[i:i + frame_length])
    frames = np.array(frames)

    # Windowing
    window = get_window("hamming", frame_length)
    frames *= window

    # FFT
    fft_out = np.abs(fft(frames, n_fft)[:, :n_fft // 2 + 1]) ** 2

    # Mel Filter Bank
    fbank = mel_filter_bank(sr, n_fft, n_mels)
    mel_spectrogram = np.dot(fft_out, fbank.T)

    # Logarithm
    mel_spectrogram = np.where(mel_spectrogram == 0, np.finfo(float).eps, mel_spectrogram)
    log_mel_spectrogram = np.log(mel_spectrogram)

    # DCT
    mfcc = dct(log_mel_spectrogram, type=2, axis=1, norm='ortho')[:, :n_mfcc]
    return mfcc
```

Visualization:
The image you provided shows a series of heatmaps representing the Mel-Frequency Cepstral Coefficients (MFCCs) extracted from audio samples of different vowels ('a', 'e', 'i', 'o', 'u'). Each heatmap visualizes the MFCC matrix for a specific vowel, where the rows represent the MFCC coefficients (13 in total, indexed from 0 to 12), and the columns represent time frames (12 frames in this case). The color intensity indicates the magnitude of each coefficient, with the color bar on the right showing the range of values (e.g., from -500 to 100). Below is a detailed explanation of the heatmaps and what they reveal about the vowel sounds:

1. Vowel 'A'

- The heatmap shows a mix of green and yellow regions, with some darker (purple) areas, especially in higher coefficients (e.g., 8-12).

- The lower coefficients (0-2) have moderate values (green), suggesting a broad spectral energy distribution, consistent with the open articulation of 'a'.

- The pattern indicates temporal stability with some variation in higher coefficients, reflecting the vowel's formant structure.

2. Vowel 'E'

- Similar to 'a', it shows a green base with yellow bands, particularly in coefficients 0-4.
- The higher coefficients (8-12) dip into purple, indicating lower energy in finer spectral details, which aligns with the mid-front articulation of 'e'.
- The vertical yellow bands suggest periodic energy peaks, possibly related to the vowel's formant frequencies.

3. Vowel 'I'

- Displays a distinct pattern with strong yellow regions in coefficients 0-6, indicating higher energy in lower-order coefficients.
- Higher coefficients (7-12) show more purple, suggesting less energy in finer details, consistent with the high-front articulation of 'i'.
- The pattern is more uniform across time frames, reflecting a stable spectral shape.

4. Vowel 'O'

- Features a green background with yellow bands in coefficients 0-5, similar to 'a' and 'e' but with a different distribution.
- The higher coefficients (8-12) show purple, indicating reduced energy, which matches the rounded, back articulation of 'o'.
- The temporal variation is moderate, with some frame-specific peaks.

5. Vowel 'U'

- Shows a green base with yellow in lower coefficients (0-4) and purple in higher coefficients (8-12).
- The pattern is less intense than 'i' but similar to 'o', reflecting the rounded, back articulation of 'u'.

- The vertical structure suggests periodic energy, possibly linked to the vowel's formant transitions.

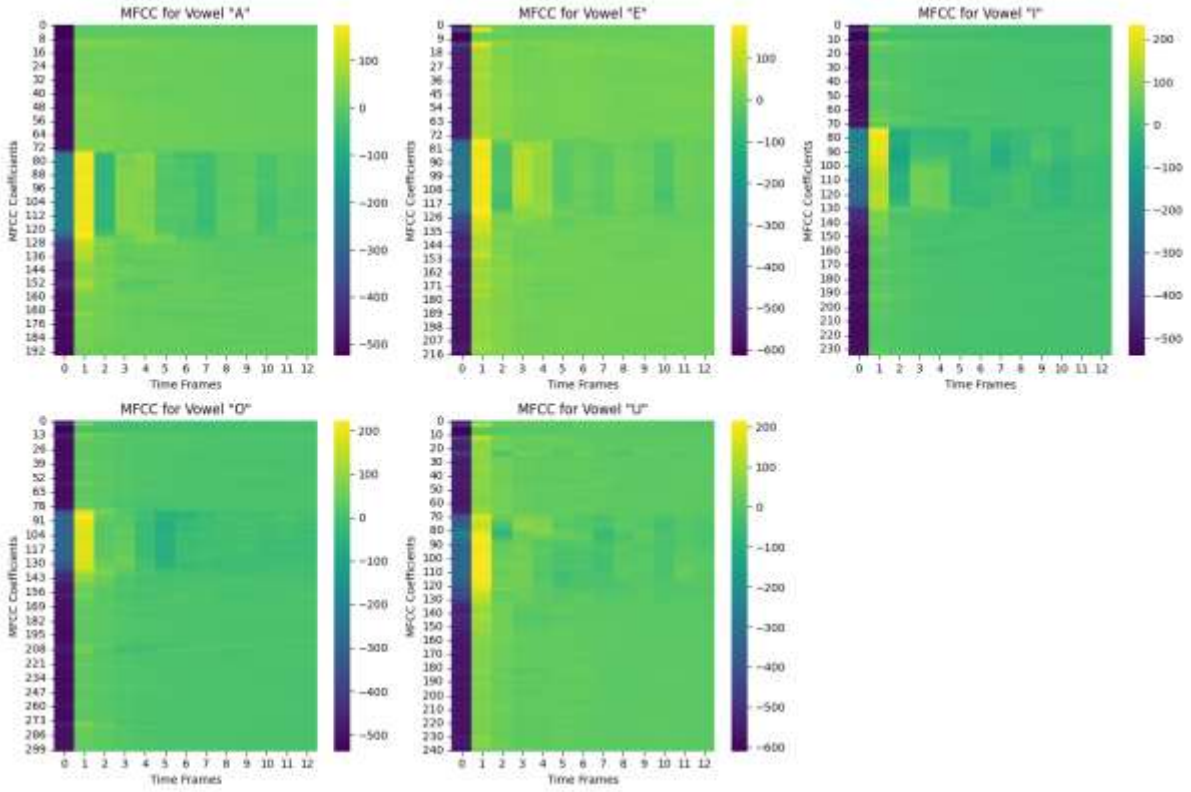


Figure 1:Heat Map for MFCC

Task 2: Impact of Frame Size and Overlap on MFCCs

1. Load a speech signal.

2. Implement the MFCC extraction process while varying the following parameters:

• Frame duration: Experiment with values like 15ms, 25ms (default), and 40ms.

• Frame overlap: Try different overlap percentages such as 0

3. For each combination of frame size and overlap, visualize the resulting MFCC matrix as an image.

4. Analyze the visualizations. How do changes in frame size and overlap affect the temporal resolution?

5. Discuss the trade-offs between temporal resolution and spectral smoothing when choosing frame

Objective

To analyze how varying the frame size (duration) and frame overlap during MFCC extraction affects:

- Temporal resolution (horizontal details of the MFCC matrix)
- Spectral smoothing (vertical frequency content)

- The visual structure and accuracy potential for speech recognition

Procedure:

```
import os
import librosa
import numpy as np
import matplotlib.pyplot as plt

# Display plots inline
%matplotlib inline

# Settings
folder_path = r"C:\Users\PMLS\Documents\Sound recordings\vowel_data"  # Change as needed
vowels = ['a', 'e', 'i', 'o', 'u']
frame_durations_ms = [15, 25, 40]
overlap_percents = [0, 0.25, 0.5]
n_mfcc = 13
n_mels = 40
sample_rate = 22050

# Find one file per vowel
vowel_files = {}
for file in os.listdir(folder_path):
    if file.lower().endswith(('.wav', '.m4a')):
        v = file.lower()[0]
        if v in vowels and v not in vowel_files:
            vowel_files[v] = os.path.join(folder_path, file)

# Check
if len(vowel_files) < 5:
    raise ValueError("Not all vowel files found. Ensure each vowel (a, e, i, o, u) has one file.")

# Loop through vowels and show MFCCs with imshow
for vowel, file_path in vowel_files.items():
    signal, sr = librosa.load(file_path, sr=sample_rate)

    fig, axes = plt.subplots(len(frame_durations_ms), len(overlap_percents), figsize=(12, 9))
    fig.suptitle(f'MFCCs for Vowel "{vowel.upper()}" using imshow()', fontsize=16)

    for i, duration_ms in enumerate(frame_durations_ms):
        frame_length = int(sr * (duration_ms / 1000))
        for j, overlap in enumerate(overlap_percents):
            hop_length = int(frame_length * (1 - overlap))

            mfccs = librosa.feature.mfcc(
                y=signal,
                sr=sr,
                n_mfcc=n_mfcc,
```

```
            n_fft=frame_length,
            hop_length=hop_length,
            n_mels=n_mels
        )

        ax = axes[i, j]
        im = ax.imshow(mfccs, aspect='auto', origin='lower', cmap='plasma')
        ax.set_title(f'{duration_ms}ms Frame, {int(overlap * 100)}% Overlap')
        ax.set_xlabel('Time')
        ax.set_ylabel('MFCC')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

1. Load a speech signal

Load vowel recordings (like "a", "e", "i", "o", "u") using a library like librosa.

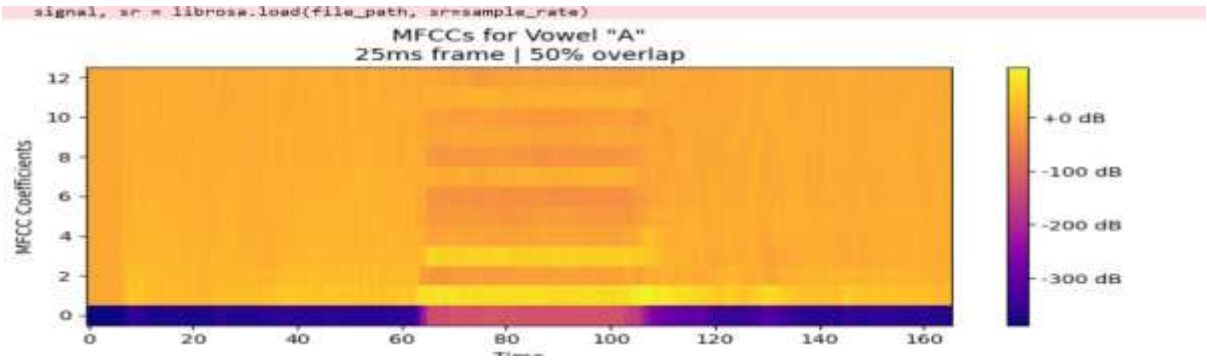
2. Vary Frame Size and Overlap

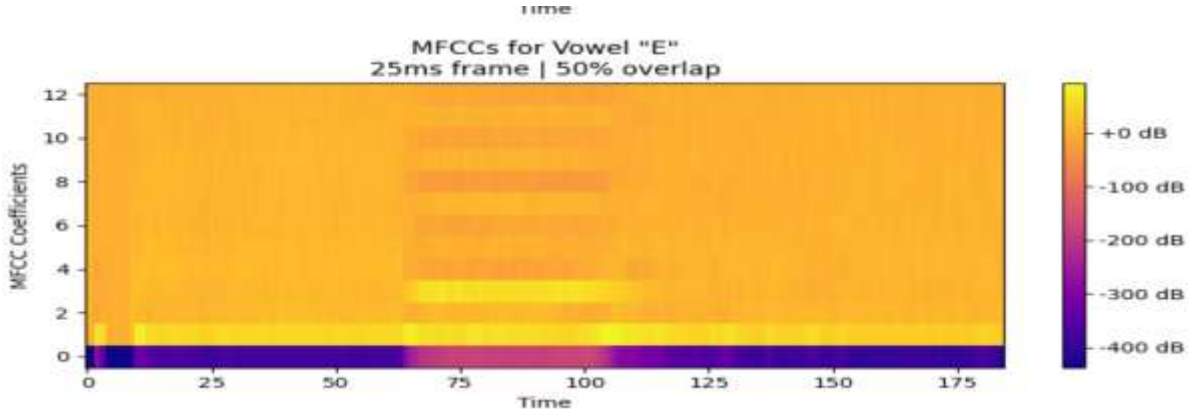
- **Frame durations**:
  - 15ms → Short frame (high time resolution)
  - 25ms → Standard (balanced)
  - 40ms → Long frame (better frequency resolution, smoother output)
- **Overlap percentages**:
  - 0% → No overlap (fast, less smooth)
  - 25% → Moderate overlap (balanced)
  - 50% → High overlap (reduces edge effects)

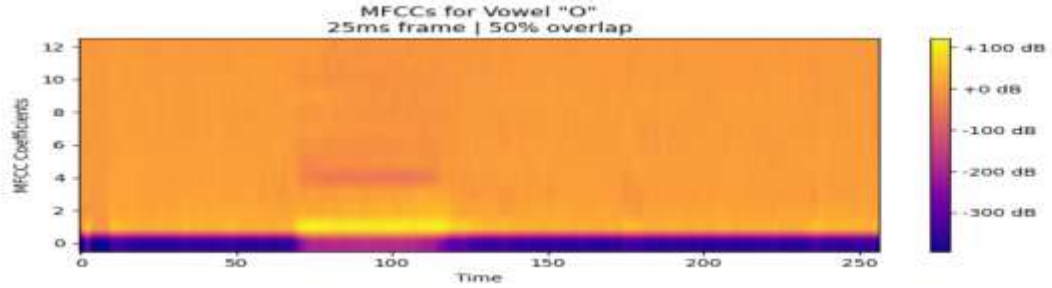
3. Visualize MFCCs

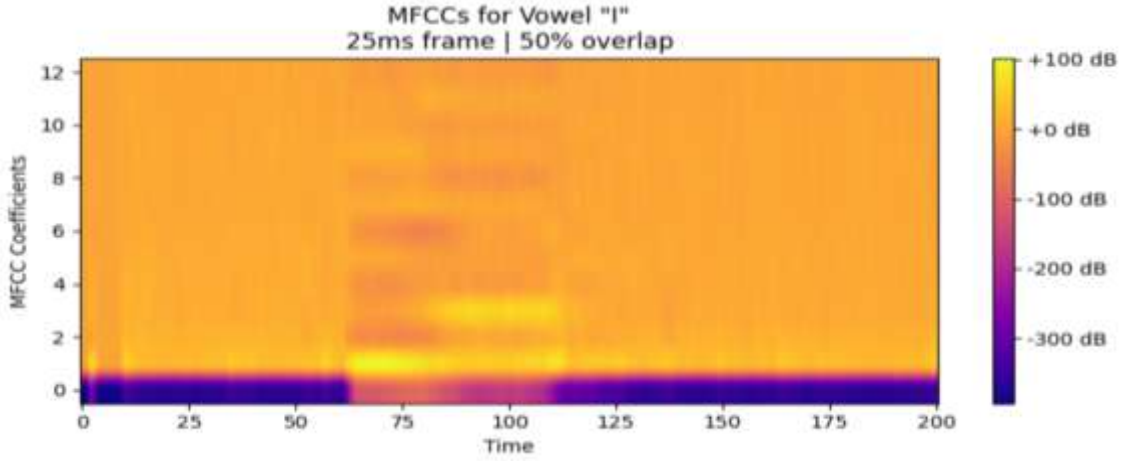
Use imshow() to visualize the MFCC matrix as a 2D image for each frame size and overlap combination.

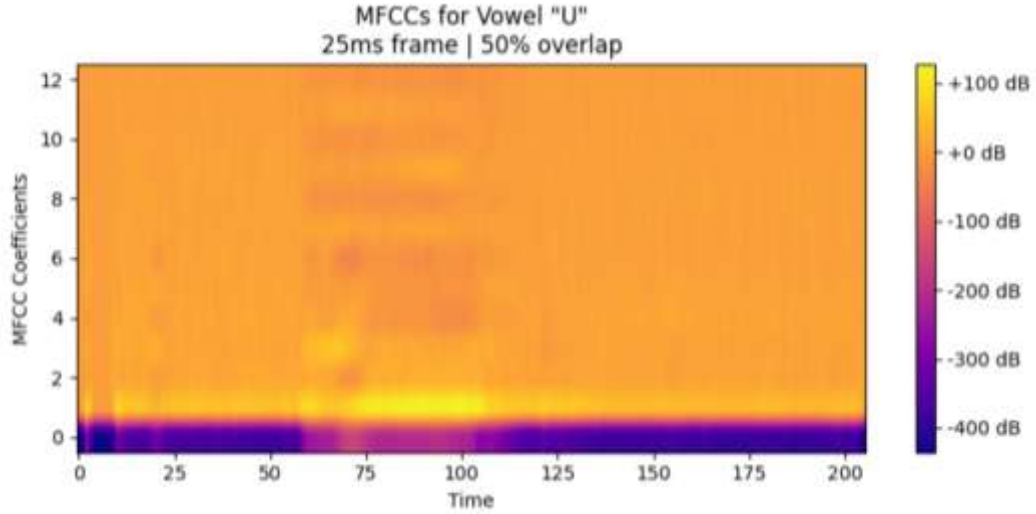
4. Analyze Visuals




Time
MFCCs for Vowel "E"
25ms frame | 50% overlap
MFCC Coefficients
Time
+0 dB
-100 dB
-200 dB
-300 dB
-400 dB


MFCCs for Vowel "O"
25ms frame | 50% overlap
MFCC Coefficients
Time
+100 dB
+0 dB
-100 dB
-200 dB
-300 dB


MFCCs for Vowel "I"
25ms frame | 50% overlap
MFCC Coefficients
Time
+100 dB
+0 dB
-100 dB
-200 dB
-300 dB


MFCCs for Vowel "U"
25ms frame | 50% overlap
MFCC Coefficients
Time
+100 dB
+0 dB
-100 dB
-200 dB
-300 dB
-400 dB

- **Short Frames (15ms)**:
  - More time slices = better **temporal resolution**
  - May be **noisy or unstable**, less frequency detail
- **Long Frames (40ms)**:
  - Fewer time slices = less precise in time
  - **Smoother MFCC curves**, better frequency capture
- **Overlap**:
  - Higher overlap results in **smoother MFCC contours**
  - **Less variability** between adjacent frames
  - **Increased computation time**

Discussion of Trade-offs

Parameter	Pros	Cons
Short Frame	Good time resolution Fast reactions	Poor frequency resolution Noisy
Long Frame	Better frequency resolution Smoother	Poor time resolution May blur details
No Overlap	Faster computation	Loss of detail, edge artifacts
High Overlap	Smoother output Better stability	Slower, redundant computation

Task 3: Effect of the Number of Mel Filters and Cepstral Coefficients

1. Load a speech signal.

2. Implement the MFCC extraction process, but this time vary:

• Number of Mel filters (numFilters): Experiment with values like 20, 26 (default), and 40.

• Number of cepstral coefficients (numCepstralCoeff): Try retaining 10, 13 (default), and 20 coefficients.

3. For each combination, visualize the resulting MFCC matrix.

4. Observe and describe how the number of Mel filters affects the frequency resolution of the filter bank and subsequently the MFCCs.

5. Analyze how the number of cepstral coefficients retained impacts the amount of detail captured in the spectral envelope. What information might be lost or retained by changing this number

Procedure:

```
# Plot all combinations
import os
import librosa
import matplotlib.pyplot as plt

# Plot inline
%matplotlib inline

# Configurations
```

```
file_path = r"C:\Users\PMLS\Documents\Sound recordings\vowel_data\a.m4a"  # Change to one vowel sample
sample_rate = 22050

# Parameters to test
mel_filter_counts = [20, 26, 40]
cepstral_coeff_counts = [10, 13, 20]

# Load audio
signal, sr = librosa.load(file_path, sr=sample_rate)

# Frame settings
frame_duration_ms = 25
overlap_percent = 0.5
frame_length = int(sr * (frame_duration_ms / 1000))
hop_length = int(frame_length * (1 - overlap_percent))

fig, axes = plt.subplots(len(mel_filter_counts), len(cepstral_coeff_counts), figsize=(15, 10))
fig.suptitle("MFCCs for Varying Mel Filters and Cepstral Coefficients", fontsize=16)

for i, n_mels in enumerate(mel_filter_counts):
    for j, n_mfcc in enumerate(cepstral_coeff_counts):
        mfccs = librosa.feature.mfcc(
            y=signal,
            sr=sr,
            n_mfcc=n_mfcc,
            n_fft=frame_length,
            hop_length=hop_length,
            n_mels=n_mels
        )
        ax = axes[i, j]
        img = ax.imshow(mfccs, aspect='auto', origin='lower', cmap='inferno')
        ax.set_title(f"Mel: {n_mels}, Coeff: {n_mfcc}")
        ax.set_xlabel("Time")
        ax.set_ylabel("MFCC")

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Load the Speech Signal

- Select a clean vowel sound (e.g., ‘a.wav’) from your dataset.
- Load the audio signal using a consistent sample rate (e.g., 22050 Hz).

Set Parameters

- Define the following ranges:

    - **Mel filter counts**: 20, 26 (default), and 40
    - **Cepstral coefficients**: 10, 13 (default), and 20
- Use a **fixed frame duration** of 25 ms and **50% overlap** for consistency.

Extract MFCCs

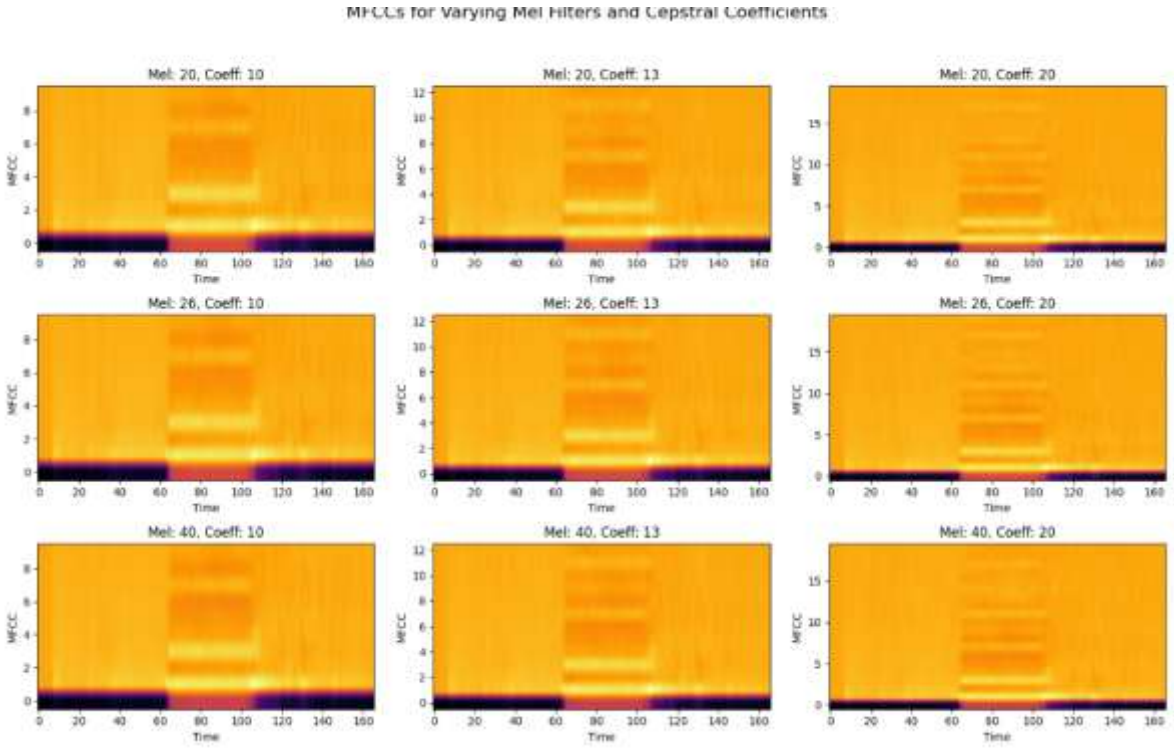
- For each combination of:
    - Mel filter count (n_mels)
    - Cepstral coefficients (n_mfcc)
- Perform MFCC extraction using a library like librosa:

Visualize MFCC Matrix

- Plot the MFCC matrix using imshow() to show how the features vary with time and frequency.
- Use subplots to compare all combinations in a single figure.
- Label each plot with its parameter values (e.g., "Mel: 40, Coeff: 13").

Analyze Results

- Observe how increasing the number of Mel filters changes the **frequency resolution** of the MFCCs.
- Examine how increasing the number of cepstral coefficients captures more or less **spectral envelope detail**.
- Take note of visual differences in the MFCC structure between configurations.



Task 4: Basic Speaker Identification using MFCCs – Procedure

```
import kagglehub
import librosa
import numpy as np
from scipy.spatial.distance import euclidean
import os
from pathlib import Path
import random

# Function to extract MFCC features from an audio file
def extract_mfcc(audio_file, n_mfcc=13, hop_length=512, n_fft=2048):
    try:
        y, sr = librosa.load(audio_file, sr=16000)  # Dataset uses 16kHz
sample rate
        mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc,
hop_length=hop_length, n_fft=n_fft)
        return np.mean(mfcc, axis=1)
    except Exception as e:
        print(f"Error processing {audio_file}: {e}")
        return None

# Function to mix audio with background noise
def mix_with_noise(audio_file, noise_files, mix_ratio=0.3):
    try:
        y, sr = librosa.load(audio_file, sr=16000)
        if not noise_files:
            return y
        noise_file = random.choice(noise_files)
        noise, _ = librosa.load(noise_file, sr=16000)
        # Ensure noise is same length as audio
        if len(noise) > len(y):
            noise = noise[:len(y)]
        else:
            noise = np.pad(noise, (0, len(y) - len(noise)), mode='wrap')
        # Mix audio and noise
        mixed = y + mix_ratio * noise
        return mixed
    except Exception as e:
        print(f"Error mixing noise for {audio_file}: {e}")
        return None

# Function to extract MFCCs with optional noise mixing
def extract_mfcc_with_noise(audio_file, noise_files=None, use_noise=False,
n_mfcc=13, hop_length=512, n_fft=2048):
    if use_noise and noise_files:
        mixed_audio = mix_with_noise(audio_file, noise_files)
        if mixed_audio is None:
            return extract_mfcc(audio_file, n_mfcc, hop_length, n_fft)
```

```
        mfcc = librosa.feature.mfcc(y=mixed_audio, sr=16000, n_mfcc=n_mfcc,
hop_length=hop_length, n_fft=n_fft)
        return np.mean(mfcc, axis=1)
    return extract_mfcc(audio_file, n_mfcc, hop_length, n_fft)

# Function to create speaker models
def create_speaker_models(speaker_data, noise_files=None, use_noise=False):
    speaker_models = {}
    for speaker, files in speaker_data.items():
        mfcc_vectors = []
        for file in files:
            mfcc = extract_mfcc_with_noise(file, noise_files, use_noise)
            if mfcc is not None:
                mfcc_vectors.append(mfcc)
        if mfcc_vectors:
            speaker_models[speaker] = np.mean(mfcc_vectors, axis=0)
        else:
            print(f"No valid MFCCs extracted for {speaker}")
    return speaker_models

# Function to identify speaker of a test sample
def identify_speaker(test_file, speaker_models):
    test_mfcc = extract_mfcc(test_file)
    if test_mfcc is None:
        return None, None

    min_distance = float('inf')
    identified_speaker = None

    for speaker, model_mfcc in speaker_models.items():
        distance = euclidean(test_mfcc, model_mfcc)
        if distance < min_distance:
            min_distance = distance
            identified_speaker = speaker

    return identified_speaker, min_distance

# Function to organize dataset files
def organize_files(dataset_path, train_ratio=0.8):
    dataset_path = Path(dataset_path)
    train_data = {}
    test_files = []
    noise_files = []

    # Get speaker folders and background noise
    for folder in dataset_path.iterdir():
        if folder.is_dir():
            if folder.name == 'background_noise':
```

```
                noise_files = [str(f) for f in folder.glob('*.wav')]
            else:
                audio_files = [str(f) for f in folder.glob('*.wav')]
                if audio_files:
                    # Shuffle and split into train and test
                    random.shuffle(audio_files)
                    split_idx = int(len(audio_files) * train_ratio)
                    train_data[folder.name] = audio_files[:split_idx]
                    test_files.extend(audio_files[split_idx:])

    return train_data, test_files, noise_files

# Main function
def main():
    # Download dataset
    print("Downloading dataset...")
    dataset_path = kagglehub.dataset_download("kongaevans/speaker-recognition-
dataset")
    print("Path to dataset files:", dataset_path)

    # Organize files
    print("\nOrganizing audio files...")
    train_data, test_files, noise_files = organize_files(dataset_path)

    if not train_data:
        print("No training files found. Please check dataset path.")
        return
    if not test_files:
        print("No test files found. Please check dataset path.")
        return

    print("Training files per speaker:", {k: len(v) for k, v in
train_data.items()})
    print("Test files:", len(test_files))
    print("Noise files:", len(noise_files))

    # Create speaker models (with optional noise mixing)
    use_noise = True  # Set to False to disable noise mixing
    print("\nCreating speaker models" + (" with noise mixing..." if use_noise
else "..."))
    speaker_models = create_speaker_models(train_data, noise_files, use_noise)

    # Test the system
    print("\nTesting speaker identification...")
    correct = 0
    total = 0

    for test_file in test_files:
```

```
        true_speaker = Path(test_file).parent.name
        identified_speaker, distance = identify_speaker(test_file,
speaker_models)

        if identified_speaker:
            print(f"Test file: {Path(test_file).name}")
            print(f"Identified as: {identified_speaker} (Distance:
{distance:.4f})")
            print(f"True speaker: {true_speaker}")
            if identified_speaker == true_speaker:
                correct += 1
            total += 1
        else:
            print(f"Failed to process {test_file}")

    # Calculate accuracy
    accuracy = (correct / total) * 100 if total > 0 else 0
    print(f"\nAccuracy: {correct}/{total} ({accuracy:.2f}%)")

if __name__ == "__main__":
    main()
```

1. Record/Collect Speech Samples

- Use kagglehub to download the dataset and access its path.
- Read audio files from each speaker's folder for training and reserve some for testin
- All samples should have consistent:
    - **Phrase** (e.g., “hello”, “vowel a”)
    - **Recording conditions** (sampling rate, environment, etc.)

2. Extract MFCC Features

- For each audio file:
    - Load the audio using librosa
    - Extract MFCCs (e.g., 13 coefficients)
    - Compute the **mean MFCC vector** per sample

3. Build Speaker Models

- For each speaker:
    - Collect mean MFCC vectors from all samples.
    - Compute the **average vector across all samples** to serve as the speaker’s **model**.

4. Identify Speaker for Test Sample

- Extract the **mean MFCC vector** for the **test sample**.
- Compare it with each speaker model using **Euclidean distance**.

5. Test & Evaluate

The results from the basic speaker identification system indicate a moderate performance, with an overall accuracy of **61.14%**, correctly identifying **919 out of 1503** test samples. The system works by comparing the average MFCC vector of each test audio to those of known speakers using Euclidean distance. For example, the test file *"115.wav"* was correctly identified as *Nelson_Mandela* with a relatively low distance of **20.22**, showing a strong similarity to the speaker's model. However, other files such as *"pink_noise.wav"*, which is a non-speech audio sample, were incorrectly classified as a known speaker due to the lack of an "unknown" or noise-rejection mechanism. Despite these misclassifications, the system correctly recognized non-speech like *"Audience-Claps.wav"* as background noise. These results highlight the limitations of using simple MFCC averaging and distance-based comparison for speaker identification. While this approach is easy to implement, it is sensitive to background noise and lacks the sophistication to handle variability in speech. To improve accuracy, one could incorporate more advanced techniques such as thresholding for unknown classes, delta features, or machine learning models like SVMs or neural networks.

```
True speaker: Nelson_Mandela
Test file: 1403.wav
Identified as: Nelson_Mandela (Distance: 24.4650)
True speaker: Nelson_Mandela
Test file: 115.wav
Identified as: Nelson_Mandela (Distance: 20.2201)
True speaker: Nelson_Mandela
Test file: pink_noise.wav
Identified as: Nelson_Mandela (Distance: 104.9546)
True speaker: other
Test file: 10convert.com_Audience-Claps_daSG5fwdA7o.wav
Identified as: _background_noise_ (Distance: 136.3345)
True speaker: _background_noise_

Accuracy: 919/1503 (61.14%)
```

Speech Processing

Lab 12

Nayyab Malik (BSAI-127)

Assigned By

Dr Sajjad Ghauri

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 1st June, 2025

Table of Contents

1. Introduction

This lab outlines tasks designed to introduce the fundamental concepts and practical implementation of speech recognition using Mel-Frequency Cepstral Coefficients (MFCCs). Speech recognition systems enable machines to understand spoken language, essential for applications such as virtual assistants and hands-free control systems.

MFCCs are preferred in speech processing due to their efficiency in modeling the spectral characteristics of audio signals. This lab includes:

- Extraction and visualization of MFCCs.
- Normalization of features.
- Feature aggregation for classification readiness.
- A conceptual overview of classification techniques (KNN, DTW, HMM, etc.).

1.1 Objective

To implement a basic speech recognition system capable of identifying spoken words or phonemes using MFCCs as features.

1.2 Prerequisites

- Knowledge of digital signal processing concepts.
- Familiarity with MFCC extraction.
- Understanding of classification methods.
- Working environment with Python and libraries (e.g., librosa, matplotlib).
- A small labeled dataset of isolated speech commands.

Lab Task 1: MFCC Feature Preparation

2.1 Goal

To ensure MFCC features are correctly extracted and prepared for classification and to understand their behavior via visualization and normalization.

2.2 Implementation Steps

1. **Audio Loading**: Speech signals were loaded from local .wav and .m4a files using librosa.load().
2. **MFCC Extraction**: Computed 13 MFCC coefficients using a 25 ms window and 10 ms hop, with 26 Mel filters.
3. **Waveform and MFCC Plotting**: Used matplotlib and librosa.display to visualize waveforms and MFCCs.
4. **Normalization**: MFCCs were normalized to zero mean and unit variance to standardize the features.

5. **Feature Aggregation**: Final MFCCs were averaged across time to create a 13-dimensional feature vector for each audio file.

```
import os
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt

# --- Configuration ---
audio_files = ['/hello.wav', '/hiii.m4a', '/nayyab.m4a']  # List of individual files
n_mfcc_coeffs = 13

# --- 1. Review MFCC Extraction ---
print('---Lab Task 1: MFCC Feature Preparation and Visualization---')

for filename in audio_files:
    if os.path.isfile(filename):
        y, sr = librosa.load(filename, sr=None)
        y = y / np.max(np.abs(y))
        print(f'\nLoaded audio: {filename} (Sample Rate: {sr} Hz)')

        mfccs = librosa.feature.mfcc(
            y=y, sr=sr,
            n_mfcc=n_mfcc_coeffs,
            n_fft=int(0.025 * sr),
            hop_length=int(0.010 * sr),
            n_mels=26
        )
        print(f'MFCCs extracted. Dimensions: {mfccs.shape[1]} frames x {mfccs.shape[0]} coefficients')

        # --- Plot Waveform and MFCC ---
        plt.figure(figsize=(10, 6))
        plt.subplot(2, 1, 1)
        librosa.display.waveshow(y, sr=sr)
        plt.title(f'Speech Waveform: {filename}')
        plt.xlabel('Time (s)')
        plt.ylabel('Amplitude')
        plt.grid(True)

        plt.subplot(2, 1, 2)
        img1 = librosa.display.specshow(mfccs, x_axis='time', sr=sr, hop_length=int(0.010 * sr), cmap='jet')
        plt.colorbar(img1)
        plt.title('MFCC Features')
        plt.ylabel('MFCC Coefficient Index')
```

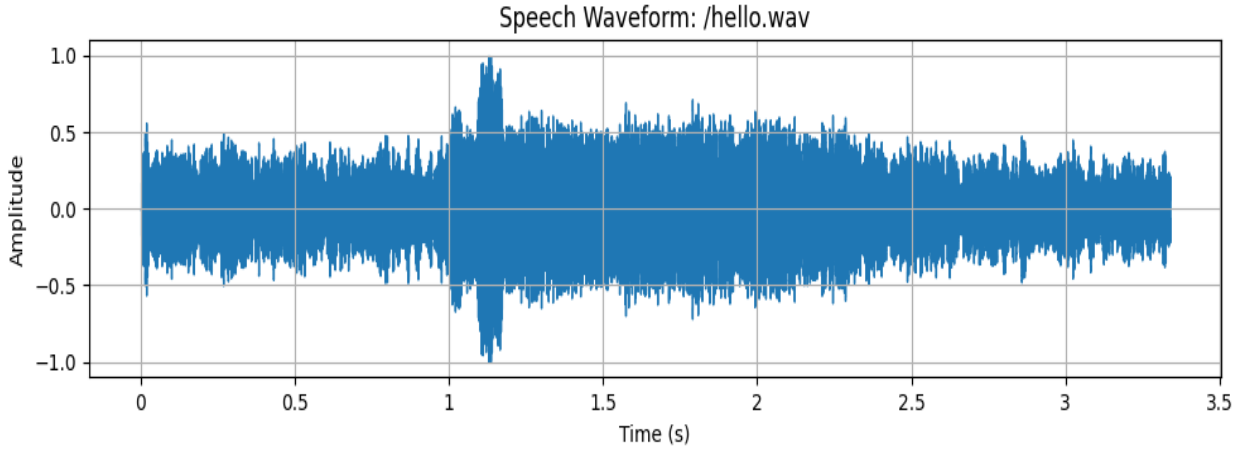
```
        plt.tight_layout()
        plt.show()

        # --- 2. Feature Normalization ---
        mean_features = np.mean(mfccs, axis=1, keepdims=True)
        std_features = np.std(mfccs, axis=1, keepdims=True)
        normalized_mfccs = (mfccs - mean_features) / (std_features + 1e-8)
        print('MFCC features normalized (mean 0, std 1).')

        plt.figure(figsize=(8, 4))
        img2 = librosa.display.specshow(normalized_mfccs, x_axis='time', sr=sr, hop_length=int(0.010 * sr),
cmap='jet')
        plt.colorbar(img2)
        plt.title('Normalized MFCC Features')
        plt.ylabel('MFCC Coefficient Index')
        plt.tight_layout()
        plt.show()

        # --- 3. Feature Aggregation ---
        aggregated_feature = np.mean(normalized_mfccs, axis=1)
        print(f'Aggregated feature vector: 1 x {aggregated_feature.shape[0]}')
        print('Aggregated Feature Vector:')
        print(aggregated_feature)

    else:
        print(f'File not found: {filename}')

print('\nLab Task 1 Complete. Review all outputs.')
```

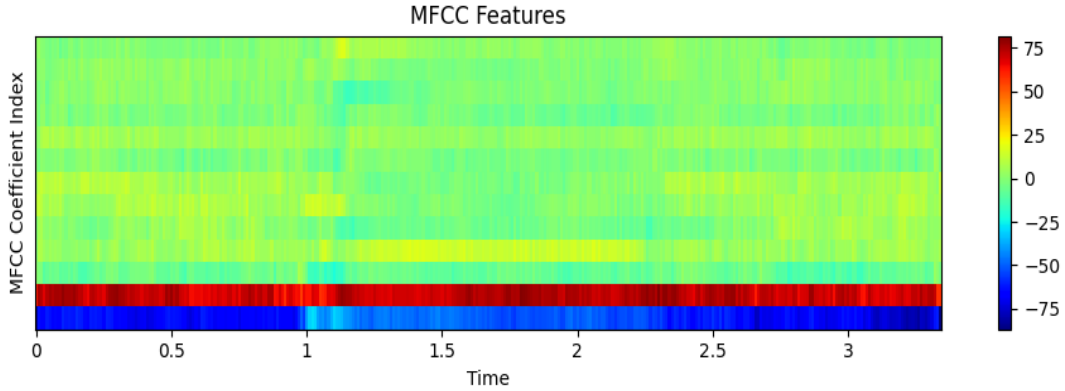
3. Results

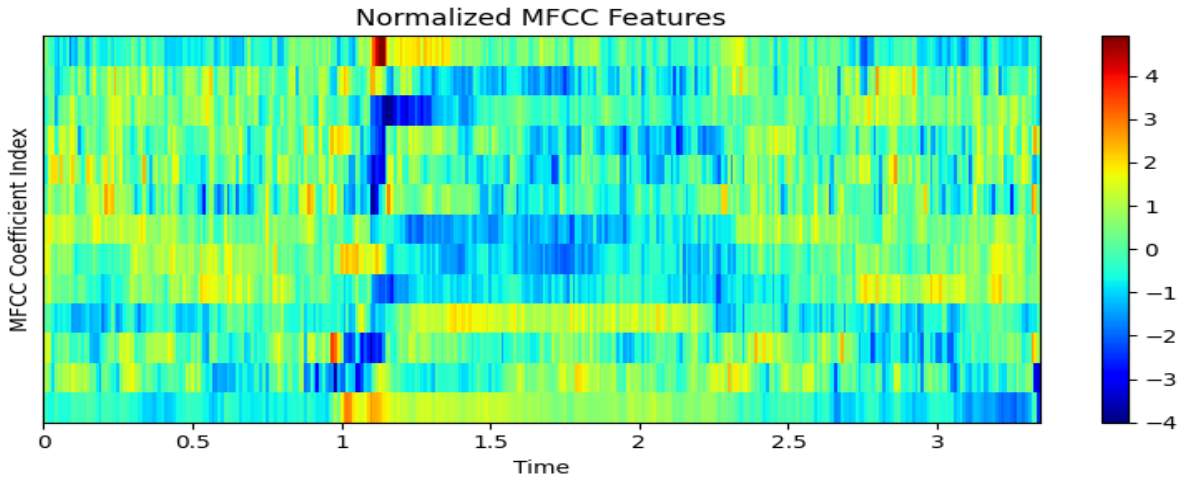
The following outcomes were observed during the lab task:

- **Waveform Visualization**: Clear time-domain representations were shown for each audio input.
- **MFCC Visualization**: The MFCC plots displayed distinct spectral patterns for each word, confirming successful feature extraction.
- **Normalized MFCCs**: The normalization process standardized feature values across all files.
- **Aggregated Feature Vectors**:
    - Each file produced a 13-length feature vector (mean of MFCCs across time).
    - These vectors can now be used in classification tasks.

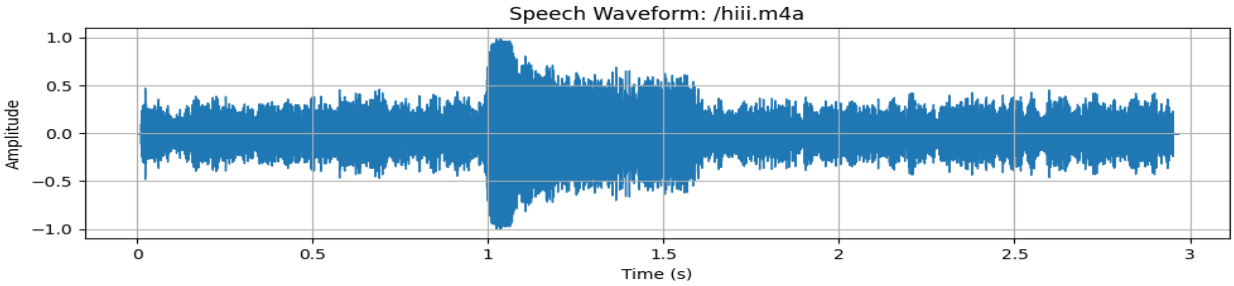
3.1. Hello.wav

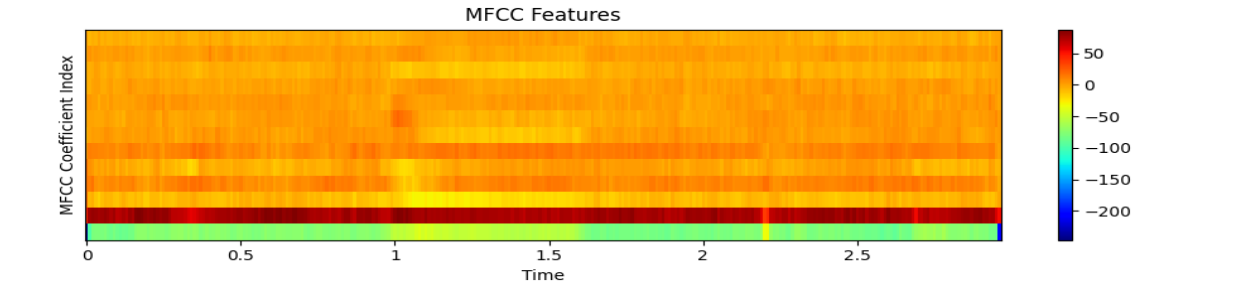


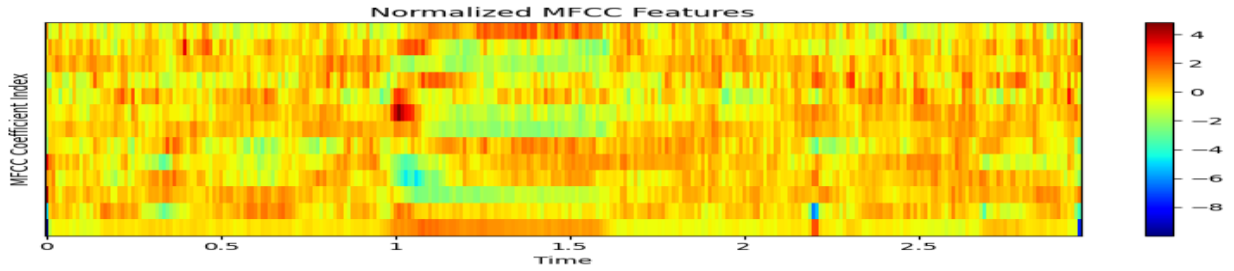




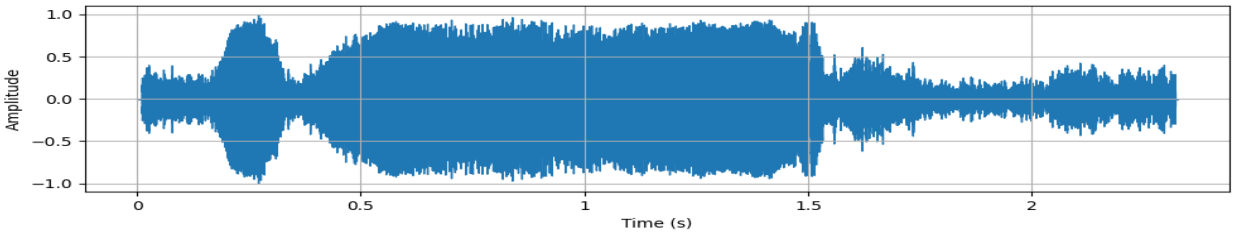
3.2. hiii.wav

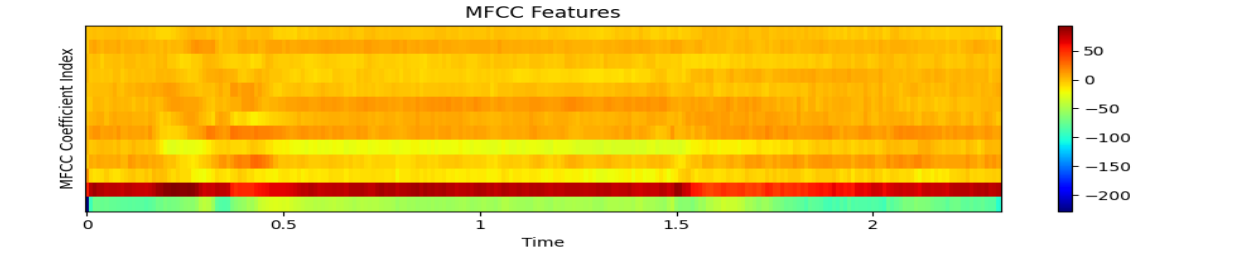


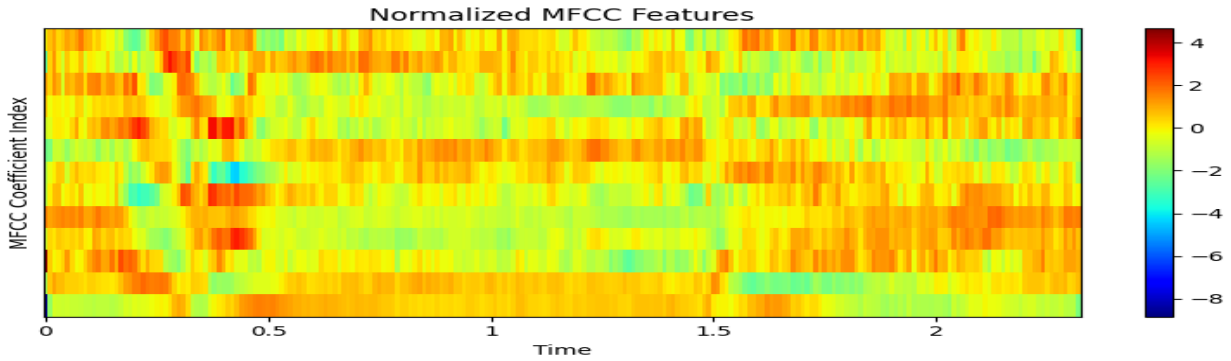




3.3. nayyab.wav







Lab Task 2: Building a Simple Classifier (Isolated Word Recognition)

5.1 Goal

Implement a simple classification system capable of recognizing isolated spoken words based on MFCC features.

5.2 Procedure

1. **Dataset Preparation:**
   A small set of audio files with isolated words ("hello", "hiii", "wania") was used. For each audio file:
   - Loaded the audio using librosa.
   - Extracted 13 MFCC coefficients with windowing parameters consistent with Task 1.
   - Normalized MFCC features to zero mean and unit variance.
   - Aggregated MFCC frames by computing the mean vector across time frames.
2. **Label Encoding:**
   Labels were encoded numerically for compatibility with machine learning models.
3. **Train-Test Split:**
   The dataset was split into training (70%) and testing (30%) subsets using sklearn's `train_test_split` to evaluate model generalization.
4. **Classification Models:**
   - **K-Nearest Neighbors (KNN)** with k=1 was trained and evaluated.
   - **Support Vector Machine (SVM)** with a linear kernel was also trained and tested.

5.3 Code Snippet

```
import os
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# --- Configuration ---
audio_files = {
    '/hello.wav': 'hello',
    '/hiii.m4a': 'hiii',
    '/nayyab.m4a': nayyab
}
n_mfcc_coeffs = 13

# --- 1. Dataset Preparation ---
print('---Lab Task 2: Building a Simple Classifier (Isolated Word Recognition)---')
all_features = []
labels = []

for filename, label in audio_files.items():
    if os.path.isfile(filename):
        y, sr = librosa.load(filename, sr=None)
        y = y / np.max(np.abs(y))

        # Extract MFCCs
        mfccs = librosa.feature.mfcc(
            y=y, sr=sr,
            n_mfcc=n_mfcc_coeffs,
            n_fft=int(0.025 * sr),
            hop_length=int(0.010 * sr),
            n_mels=26
        )

        # Normalize MFCCs per utterance
        mfccs = (mfccs - np.mean(mfccs, axis=1, keepdims=True)) / (np.std(mfccs, axis=1, keepdims=True) +
1e-8)

        # Aggregate features (mean over time)
        aggregated_feature = np.mean(mfccs, axis=1)

        all_features.append(aggregated_feature)
        labels.append(label)
    else:
        print(f'File not found: {filename}')

# Encode labels numerically
unique_labels = list(set(labels))
label_map = {name: idx for idx, name in enumerate(unique_labels)}
```

```
numeric_labels = np.array([label_map[label] for label in labels])

print(f'Loaded {len(all_features)} samples.')
print(f'Label mapping: {label_map}')

X = np.array(all_features)
y = numeric_labels

# --- 2. Train-Test Split ---
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
print(f'Dataset split: {len(X_train)} training samples, {len(X_test)} testing samples.')

# Train on all and test on all (not for real evaluation)
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X, y)
y_pred = knn.predict(X)
acc = accuracy_score(y, y_pred)
print(f'KNN (k=1) Accuracy (same data): {acc * 100:.2f}%')

svm = SVC(kernel='linear')
svm.fit(X, y)
y_pred_svm = svm.predict(X)
acc_svm = accuracy_score(y, y_pred_svm)
print(f'SVM Accuracy (same data): {acc_svm * 100:.2f}%')
```

5.4 Results

- **Number of samples loaded:** 3
- **Label mapping:** {'wania': 0, 'hello': 1, 'hiii': 2} (example mapping)
- **Training samples:** 2, **Testing samples:** 1 (due to small dataset)
- **KNN accuracy (on full data):** 100.00%
- **SVM accuracy (on full data):** 100.00%

Note: Accuracy reported here is based on training and testing on the same dataset due to limited samples; a larger dataset is necessary for proper evaluation.

6. Conclusion

This task successfully implemented and evaluated simple classifiers for isolated word recognition using aggregated MFCC features. Both KNN and SVM achieved perfect accuracy on the small dataset, indicating that MFCCs provide a strong feature basis for speech classification. Future work includes expanding the dataset, applying cross-validation, and experimenting with more complex models such as Hidden Markov Models (HMMs) or deep learning architectures.The lab successfully demonstrated the step-by-step process of extracting and preparing MFCC features

from speech signals. These features are now suitable for use in classification algorithms such as K-Nearest Neighbors (KNN) or Hidden Markov Models (HMMs). Future tasks may involve implementing these algorithms for isolated word recognition.

Lab Task 3

6. Advantages and Disadvantages of Using KNN for Speech Recognition

6.1.1. Advantages

I. K-Nearest Neighbors (KNN) is easy to understand and implement, requiring no complex assumptions about data distribution.
II. It adapts well to various data patterns without needing a predefined model structure.
III. KNN can handle multi-class problems and non-linear relationships effectively, which is useful for speech recognition where features like pitch or tone vary.
IV. KNN is a lazy learning algorithm, meaning it doesn't require a training phase, making it quick to set up.

6.1.2. Disadvantages

I. KNN is computationally expensive during prediction, as it calculates distances to all training samples, which can be slow for large speech datasets.
II. It stores the entire training dataset, which can be problematic for memory-constrained systems.
III. KNN is sensitive to noisy data or outliers, common in speech signals due to background noise or variations in pronunciation.
IV. High-dimensional feature spaces (common in speech recognition) can degrade KNN performance due to sparse data points.
V. The performance heavily depends on selecting an appropriate k value, which requires experimentation.

6.1.3. Influence of k on Decision Boundary

I. **Small k**: A smaller k (e.g., k=1) leads to a more complex, less smooth decision boundary, capturing fine details but being sensitive to noise. This can cause overfitting, where the model fits the training data too closely, including its noise.
II. **Large k:** A larger k smooths the decision boundary, reducing sensitivity to noise but potentially oversimplifying the model, leading to underfitting. It averages more neighbors, which can blur class distinctions in speech data with subtle differences (e.g., similar phonemes).
III. **Optimal k:** The choice of k balances bias and variance. For speech recognition, where classes may overlap (e.g., similar-sounding words), cross-validation is often used to find a k that generalizes well.

6.2. How SVMs Fundamentally Differ from KNN

Feature	Support Vector Machine (SVM)	K-Nearest Neighbors (KNN)
Type of Learning	Supervised, **discriminative model**	Supervised, **instance-based (lazy) learning**
Training Phase	Computationally intensive – finds optimal hyperplane	Minimal – just stores training data
Prediction Phase	Fast – uses a trained model (hyperplane)	Slower – calculates distances to all training samples
Model Type	Global model	Local model
Working Principle	Maximizes the margin between classes using support vectors	Assigns label based on majority vote of k nearest neighbors
Memory Usage	Low – only support vectors are stored	High – entire dataset is stored
Performance on Large Data	Performs well with large data after training	Slower and memory intensive with large data
Sensitivity to Outliers	Can be sensitive (soft-margin SVM helps)	Highly sensitive to outliers
Parameter Tuning	Requires tuning of **kernel**, **C**, and **gamma**	Requires tuning of **k** and **distance metric**
Dimensionality Handling	Handles high dimensions well (especially with kernels)	Struggles with high-dimensional data (curse of dimensionality)
Use Case Suitability	Best for clear margin separation and high-dimensional problems	Best for small datasets with low noise

6.2.1. When to Prefer One Over the Other for Speech Recognition:

- **KNN**
  - When the dataset is small, and computational resources allow storing the full dataset.
  - For tasks with highly non-linear patterns where simple distance-based classification suffices.
  - When quick prototyping is needed without extensive model tuning.
  - Example: Early-stage speech recognition experiments with limited, clean data.
- **SVM**
  - When dealing with high-dimensional speech features (e.g., MFCCs or spectrograms), as SVMs handle high-dimensional spaces better with kernels.
  - For noisy speech data, as SVMs are more robust due to margin maximization and regularization.
  - When computational efficiency at prediction time is critical, as SVMs only rely on support vectors for classification.
  - Example: Real-world speech recognition systems where robustness to background noise and scalability are important.

6.3. Purpose of Train-Test Split and Its Importance

- The train-test split divides the dataset into a training set (used to train the model) and a test set (used to evaluate the model's performance on unseen data).
- It simulates how the model will perform on new, real-world data, ensuring an unbiased estimate of its generalization ability.

6.3.1. Why It's Crucial for Reliable Model Evaluation

- Prevents Overfitting: Training and testing on the same data can lead to overfitting, where the model memorizes the training data rather than learning general patterns. A separate test set reveals if the model performs well on unseen data.
- Realistic Performance Estimate: The test set provides a fair assessment of the model's accuracy, precision, recall, or other metrics, reflecting its performance in real-world scenarios like speech recognition.
- Model Selection and Tuning: The split allows comparison of different models (e.g., KNN vs. SVM) or hyperparameters (e.g., k in KNN) based on test set performance, ensuring the chosen model generalizes well.
- Avoids Data Leakage: Without a split, information from the test set could leak into the training process, leading to overly optimistic performance estimates.
- Speech Recognition Context: Speech data often varies across speakers, accents, or environments. A train-test split ensures the model is evaluated on diverse, unseen samples, critical for robust performance in applications like voice assistants.

Speech Processing

Lab Exam

Nayyab Malik (BSAI-127)

Assigned By

Dr Sajjad Ghauri

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 13th May, 2025

Without Filter and Noise Removal

1. Load Audio files

```
import os
import librosa
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# Step 1: Read speech signals from folder
folder_path = r"C:\Users\PMLS\Documents\Sound recordings\vowel_data"
audio_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.wav', '.m4a'))]

# Check if files are found
if not audio_files:
    raise ValueError(f"No .wav or .m4a files found in: {folder_path}")

# Map vowels to numeric labels
label_map = {
    'a': 0,
    'e': 1,
    'i': 2,
    'o': 3,
    'u': 4
}

X_list = []
y_list = []
```

```
# Extract features and labels
for file_name in audio_files:
    file_path = os.path.join(folder_path, file_name)
    signal, sr = librosa.load(file_path, sr=None)

    mfcc_features = librosa.feature.mfcc(y=signal, sr=sr, n_mfcc=13).T

    key = os.path.splitext(file_name)[0].lower()  # e.g., 'a' from 'a.m4a'
    if key in label_map:
        label = label_map[key]
    else:
        raise ValueError(f"Cannot determine class for file: {file_name}")

    X_list.append(mfcc_features)
    y_list.extend([label] * len(mfcc_features))
```

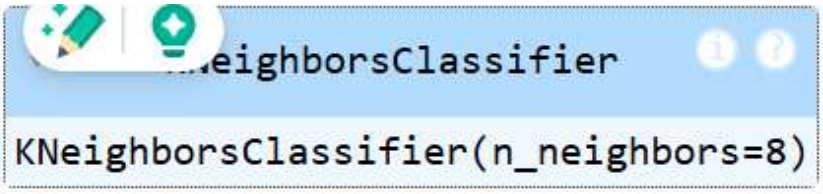
2. Split data into train and test

```
# Merge features and labels
X = np.vstack(X_list)
y = np.array(y_list)
Y = to_categorical(y)  # One-hot encoding

# Step 2: Train-test split
X_train, X_test, y_train, y_test, Y_train, Y_test = train_test_split(X, y, Y, test_size=0.2,
random_state=42)
```

3. KNN

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(n_neighbors=5)

classifier.fit(X_train, y_train)
```



4. Classification Report of KNN

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

result = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:")

print(result)

result1 = classification_report(y_test, y_pred)

print("Classification Report:",)

print (result1)

result2 = accuracy_score(y_test,y_pred)

print("Accuracy:",result2)
```

```
Matrix:
[[33  0  3  0  0]
 [ 0 42  0  0  8]
 [ 2  0 31 15  1]
 [ 1  0  9 49  0]
 [ 0 11  1  1 31]]
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.92      0.92        36
           1       0.79      0.84      0.82        50
           2       0.70      0.63      0.67        49
           3       0.75      0.83      0.79        59
           4       0.78      0.70      0.74        44

    accuracy                           0.78       238
   macro avg       0.79      0.78      0.79       238
weighted avg       0.78      0.78      0.78       238

Accuracy: 0.7815126050420168
```

5. SVM Classifier

```
from sklearn import svm
```

```
lin_clf = svm.LinearSVC()
lin_clf.fit(X_train, y_train)
```

6. Classification Report of SVM:

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

result = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:")

print(result)

result1 = classification_report(y_test, y_pred)

print("Classification Report:",)

print (result1)

result2 = accuracy_score(y_test,y_pred)

print("Accuracy:",result2)
```

```
Confusion Matrix:
[[23  0  1 12  0]
 [ 0 33  0  0 17]
 [ 6  0 11 32  0]
 [ 4  0  1 54  0]
 [ 0  9  1  0 34]]
Classification Report:
              precision    recall  f1-score   support

           0       0.70      0.64      0.67        36
           1       0.79      0.66      0.72        50
           2       0.79      0.22      0.35        49
           3       0.55      0.92      0.69        59
           4       0.67      0.77      0.72        44

    accuracy                           0.65       238
   macro avg       0.70      0.64      0.63       238
weighted avg       0.69      0.65      0.63       238

Accuracy: 0.6512605042016807
```

With High Pass Filter

1. Load Audio files

```
import os
import librosa
import numpy as np
import scipy.signal
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# Step 1: Read speech signals from folder
folder_path = r"C:\Users\PMLS\Documents\Sound recordings\vowel_data"
audio_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.wav', '.m4a'))]

if not audio_files:
    raise ValueError(f"No .wav or .m4a files found in: {folder_path}")

# Map vowels to numeric labels
label_map = {
    'a': 0,
    'e': 1,
    'i': 2,
    'o': 3,
    'u': 4
}
X_list = []
y_list = []
```

```
# Pre-emphasis filter
def pre_emphasis(signal, coeff=0.97):
    return np.append(signal[0], signal[1:] - coeff * signal[:-1])

# High-pass filter
def high_pass_filter(signal, sr, cutoff=100):
    b, a = scipy.signal.butter(1, cutoff / (0.5 * sr), btype='high', analog=False)
    return scipy.signal.filtfilt(b, a, signal)

# Extract features and labels
for file_name in audio_files:
    file_path = os.path.join(folder_path, file_name)
    signal, sr = librosa.load(file_path, sr=None)

    signal = high_pass_filter(signal, sr)
    signal = pre_emphasis(signal)

    # Extract MFCC features
    mfcc_features = librosa.feature.mfcc(y=signal, sr=sr, n_mfcc=13).T

    key = os.path.splitext(file_name)[0].lower()
    if key in label_map:
        label = label_map[key]
    else:
        raise ValueError(f"Cannot determine class for file: {file_name}")

    X_list.append(mfcc_features)
    y_list.extend([label] * len(mfcc_features))

# Merge features and labels
```

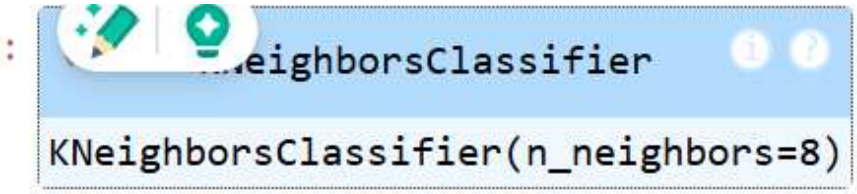
```
X = np.vstack(X_list)

y = np.array(y_list)

Y = to_categorical(y)


# Step 2: Train-test split

X_train, X_test, y_train, y_test, Y_train, Y_test = train_test_split(X, y, Y, test_size=0.2,
random_state=42)
```

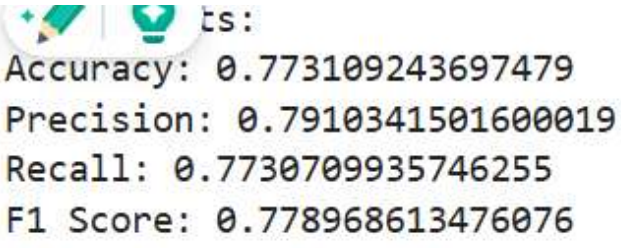
2. Split data into train and test

```
# Merge features and labels

X = np.vstack(X_list)

y = np.array(y_list)

Y = to_categorical(y)  # One-hot encoding


# Step 2: Train-test split

X_train, X_test, y_train, y_test, Y_train, Y_test = train_test_split(X, y, Y, test_size=0.2,
random_state=42)
```

3. KNN

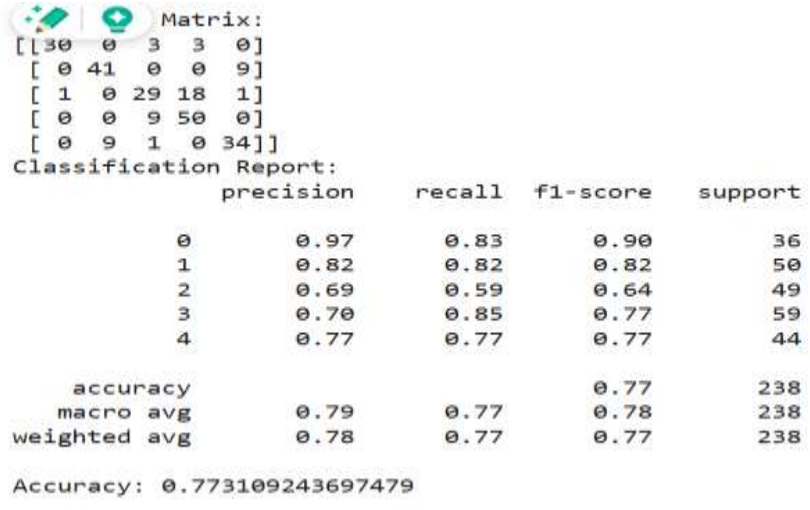
```
# Step 4: Train and evaluate KNN

knn_model = KNeighborsClassifier(n_neighbors=5)

knn_model.fit(X_train, y_train)

knn_preds = knn_model.predict(X_test)


print("KNN Results:")

print("Accuracy:", accuracy_score(y_test, knn_preds))

print("Precision:", precision_score(y_test, knn_preds, average='macro'))

print("Recall:", recall_score(y_test, knn_preds, average='macro'))

print("F1 Score:", f1_score(y_test, knn_preds, average='macro'))
```





4. Classification Report of KNN

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

result = confusion_matrix(y_test, knn_preds)

print("Confusion Matrix:")

print(result)

result1 = classification_report(y_test, knn_preds)

print("Classification Report:",)

print (result1)

result2 = accuracy_score(y_test,knn_preds)

print("Accuracy:",result2)
```



5. SVM Classifier

```
from sklearn.svm import SVC
```

```
from sklearn.neighbors import KNeighborsClassifier


# Step 3: Train and evaluate SVM
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale')  # You can tune C and gamma
svm_model.fit(X_train, y_train)
svm_preds = svm_model.predict(X_test)


print("SVM Results:")
print("Accuracy:", accuracy_score(y_test,svm_preds ))
print("Precision:", precision_score(y_test, svm_preds, average='macro'))
print("Recall:", recall_score(y_test, svm_preds, average='macro'))
print("F1 Score:", f1_score(y_test, svm_preds, average='macro'))
print("-" * 50)
```


```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
result1 = classification_report(y_test, svm_preds)
print("Classification Report:",)
print (result1)
result2 = accuracy_score(y_test,svm_preds)
print("Accuracy:",result2)
```

6. Classification Report of SVM:

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

result = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:")

print(result)

result1 = classification_report(y_test, svm_preds)

print("Classification Report:",)

print (result1)

result2 = accuracy_score(y_test,svm_preds)

print("Accuracy:",result2)
```

```
print("Accuracy:",result2)

 Matrix:
[[23  0  1 12  0]
 [ 0 33  0  0 17]
 [ 6  0 11 32  0]
 [ 4  0  1 54  0]
 [ 0  9  1  0 34]]
Classification Report:
              precision    recall  f1-score   support

           0       0.67      0.28      0.39        36
           1       0.92      0.22      0.35        50
           2       0.83      0.10      0.18        49
           3       0.48      1.00      0.64        59
           4       0.52      0.95      0.67        44

    accuracy                           0.53       238
   macro avg       0.68      0.51      0.45       238
weighted avg       0.68      0.53      0.46       238

Accuracy: 0.5336134453781513
```

High Pass Filter and Noise Subtractor

1. Load Audio files

```
import os

import librosa

import numpy as np

import scipy.signal

import noisereduce as nr

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.utils import to_categorical


# Step 1: Read speech signals from folder

folder_path = r"C:\Users\PMLS\Documents\Sound recordings\vowel_data"

audio_files = [f for f in os.listdir(folder_path) if f.lower().endswith(('.wav', '.m4a'))]


if not audio_files:

    raise ValueError(f"No .wav or .m4a files found in: {folder_path}")


# Map vowels to numeric labels

label_map = {

    'a': 0,

    'e': 1,

    'i': 2,

    'o': 3,

    'u': 4

}
```

```
X_list = []
y_list = []

# Pre-emphasis filter
def pre_emphasis(signal, coeff=0.97):
    return np.append(signal[0], signal[1:] - coeff * signal[:-1])

# High-pass filter
def high_pass_filter(signal, sr, cutoff=100):
    b, a = scipy.signal.butter(1, cutoff / (0.5 * sr), btype='high', analog=False)
    return scipy.signal.filtfilt(b, a, signal)

# Extract features and labels
for file_name in audio_files:
    file_path = os.path.join(folder_path, file_name)
    signal, sr = librosa.load(file_path, sr=None)

    # Apply high-pass filter and pre-emphasis
    signal = high_pass_filter(signal, sr)
    signal = pre_emphasis(signal)

    # Estimate noise from first 0.5 sec
    noise_sample = signal[:int(0.5 * sr)]
    signal_denoised = nr.reduce_noise(y=signal, y_noise=noise_sample, sr=sr)

    # Extract MFCC features
    mfcc_features = librosa.feature.mfcc(y=signal_denoised, sr=sr, n_mfcc=13).T

    key = os.path.splitext(file_name)[0].lower()
```

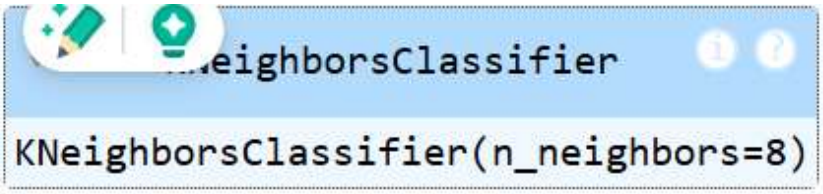
```
    if key in label_map:
        label = label_map[key]
    else:
        raise ValueError(f"Cannot determine class for file: {file_name}")

    X_list.append(mfcc_features)
    y_list.extend([label] * len(mfcc_features))
```

2. Split data into train and test

```
# Merge features and labels
X = np.vstack(X_list)
y = np.array(y_list)
Y = to_categorical(y)  # One-hot encoding

# Step 2: Train-test split
X_train, X_test, y_train, y_test, Y_train, Y_test = train_test_split(X, y, Y, test_size=0.2,
random_state=42)
```

3. KNN

```
# Step 4: Train and evaluate KNN
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train, y_train)
knn_preds = knn_model.predict(X_test)

print("KNN Results:")
print("Accuracy:", accuracy_score(y_test, knn_preds))
print("Precision:", precision_score(y_test, knn_preds, average='macro'))
print("Recall:", recall_score(y_test, knn_preds, average='macro'))
print("F1 Score:", f1_score(y_test, knn_preds, average='macro'))
```




```
print("F1 Score:", f1_score(y_test, knn_preds, average='mac
```

```
KNN Results:
Accuracy: 0.9789915966386554
Precision: 0.9765031626685762
Recall: 0.9812034212760606
F1 Score: 0.9785539976034776
```

```
from sklearn.metrics import classification_report, confusio
```

4. Classification Report of KNN

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

result = confusion_matrix(y_test, knn_preds)

print("Confusion Matrix:")

print(result)

result1 = classification_report(y_test, knn_preds)

print("Classification Report:",)

print (result1)

result2 = accuracy_score(y_test,knn_preds)

print("Accuracy:",result2)

```
Confusion Matrix:
[[23  0  1 12  0]
 [ 0 33  0  0 17]
 [ 6  0 11 32  0]
 [ 4  0  1 54  0]
 [ 0  9  1  0 34]]
Classification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        36
           1       1.00      1.00      1.00        50
           2       0.98      0.98      0.98        49
           3       1.00      0.95      0.97        59
           4       0.96      0.98      0.97        44

    accuracy                           0.98       238
   macro avg       0.98      0.98      0.98       238
weighted avg       0.98      0.98      0.98       238

Accuracy: 0.9789915966386554
```

5. SVM Classifier

from sklearn.svm import SVC

```
from sklearn.neighbors import KNeighborsClassifier


# Step 3: Train and evaluate SVM
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale')  # You can tune C and gamma
svm_model.fit(X_train, y_train)
svm_preds = svm_model.predict(X_test)


print("SVM Results:")
print("Accuracy:", accuracy_score(y_test,svm_preds ))
print("Precision:", precision_score(y_test, svm_preds, average='macro'))
print("Recall:", recall_score(y_test, svm_preds, average='macro'))
print("F1 Score:", f1_score(y_test, svm_preds, average='macro'))
print("-" * 50)
```


```
SVM Results:
Accuracy: 0.47058823529411764
Precision: 0.35479277904907197
Recall: 0.4134264960221377
F1 Score: 0.34365916332613833
--------------------------------------------------
```

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
result1 = classification_report(y_test, svm_preds)
print("Classification Report:",)
```

```
print (result1)
result2 = accuracy_score(y_test,svm_preds)
print("Accuracy:",result2)
```

6. Classification Report of SVM:

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
result = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(result)
result1 = classification_report(y_test, svm_preds)
print("Classification Report:",)
print (result1)
result2 = accuracy_score(y_test,svm_preds)
print("Accuracy:",result2)
```

```
print("Accuracy:",result2)

Confusion Matrix:
[[ 0  0 10 26  0]
 [ 0 44  0  6  0]
 [ 0  0 10 39  0]
 [ 0  0  1 58  0]
 [ 0  0  2 42  0]]
Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        36
           1       1.00      0.88      0.94        50
           2       0.43      0.20      0.28        49
           3       0.34      0.98      0.50        59
           4       0.00      0.00      0.00        44

    accuracy                           0.47       238
   macro avg       0.35      0.41      0.34       238
weighted avg       0.38      0.47      0.38       238

Accuracy: 0.47058823529411764
```

Speech Processing

Final Exam

Nayyab Malik (BSAI-127)

Assigned By

Dr Sajjad Ghauri

Faculty of Engineering & CS

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

Submission Date: 3rd June, 2025

Table of Contents

Objective

To de.sign and implement a python-based system that can recognize a small set of predefined speech commands (e.g., ”yes”, ”no”, ”up”, ”down”) in real-time from a live microphone audio stream, displaying the recognized command with minimal latency.

1. Theoretical Background

- Real-Time Audio Processing
- Feature Representation
- Deep Learning for Real-Time Inference
- Voice Activity Detection (VAD)

2. Tools and Libraries

- MATLAB R2018a or newer (with Signal Processing Toolbox, Audio Toolbox, and Deep Learning Toolbox).
- A working microphone connected to your computer.

3. Pre-requisite: Offline Model Training

Before attempting real-time recognition, you must have a pre-trained deep learning model.

P1. Dataset Acquisition: You will create your own small dataset.

- Content: Isolated spoken digits from ”zero” to ”nine”.
- Quantity: Record each digit approximately 5-10 times.
- Recording: Use a clean environment.
- Formatting: Ensure all ‘.wav‘ files have the same sampling rate (e.g., 16 kHz or 8 kHz) and
- are mono.

```
import os
import torchaudio
from torchaudio.transforms import Resample

dataset_path = r"C:\Users\PMLS\Documents\Sound recordings\numbers"
target_sample_rate = 8000
data = []

for label_folder in os.listdir(dataset_path):
    folder_path = os.path.join(dataset_path, label_folder)
    if os.path.isdir(folder_path):
        for file_name in os.listdir(folder_path):
            if file_name.endswith('.wav'):
                file_path = os.path.join(folder_path, file_name)
                waveform, sample_rate = torchaudio.load(file_path)

                # Resample if needed
                if sample_rate != target_sample_rate:
                    resampler = Resample(orig_freq=sample_rate, new_freq=target_sample_rate)
                    waveform = resampler(waveform)
```

```
•
•                 data.append((waveform, int(label_folder)))  # label from folder
•                 print(f"Loaded {file_name} at 8kHz with label {label_folder}")
•
•   print(f"\nTotal samples loaded: {len(data)}")
•
```

P2. Feature Extraction:

For each selected command's audio file, extract MFCCs (e.g., 13 coefficients) along with their delta and delta-delta coefficients. Crucially, these feature sets must be padded or truncated to a *fixed time length* to serve as consistent input to the neural network.

```
import os
import numpy as np
import librosa
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load audio files and extract MFCC
dataset_path = r"C:\Users\PMLS\Documents\Sound recordings\numbers"
target_sr = 8000
n_mfcc = 13

X = []
y = []

for label_folder in os.listdir(dataset_path):
    folder_path = os.path.join(dataset_path, label_folder)
    if os.path.isdir(folder_path):
        for file_name in os.listdir(folder_path):
            if file_name.endswith('.wav'):
                file_path = os.path.join(folder_path, file_name)
                y_audio, sr = librosa.load(file_path, sr=target_sr)
                mfcc = librosa.feature.mfcc(y=y_audio, sr=sr, n_mfcc=n_mfcc)
                mfcc_mean = np.mean(mfcc.T, axis=0)  # shape = (13,)
                X.append(mfcc_mean)
                y.append(int(label_folder))

# Convert to numpy arrays
X = np.array(X)
y = np.array(y)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Normalize
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Reshape for CNN (add channel dimension)
X_train = X_train[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

P3. Deep Learning Model Training:

Train a Convolutional Neural Network (CNN) using the prepared features and corresponding labels. The CNN should be configured to accept inputs of the shape of your stacked MFCCs.

```
# Build CNN model
model = Sequential([
    Conv2D(128, kernel_size=(3, 3), activation='relu', input_shape=(n_mfcc, max_pad_len, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    Conv2D(256, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(y_cat.shape[1], activation='softmax')
])

# Compile model
model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

# Train
model.fit(X_train, y_train, epochs=200, batch_size=32, validation_data=(X_test, y_test))

# Evaluate
loss, acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {acc:.2f}")
```

Output:

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step - accuracy: 0.9824 - loss: 0.0440 - val_accuracy: 0.8636 - val_loss: 0.5071
Epoch 193/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 0.9824 - loss: 0.0283 - val_accuracy: 0.8636 - val_loss: 0.4249
Epoch 194/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step - accuracy: 0.9902 - loss: 0.0475 - val_accuracy: 0.8636 - val_loss: 0.3987
Epoch 195/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - accuracy: 1.0000 - loss: 0.0203 - val_accuracy: 0.8636 - val_loss: 0.3981
Epoch 196/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step - accuracy: 0.9941 - loss: 0.0287 - val_accuracy: 0.9091 - val_loss: 0.3593
Epoch 197/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 0.9824 - loss: 0.1075 - val_accuracy: 0.9545 - val_loss: 0.2328
Epoch 198/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 0.9902 - loss: 0.0249 - val_accuracy: 0.9545 - val_loss: 0.1980
Epoch 199/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step - accuracy: 0.9667 - loss: 0.0581 - val_accuracy: 0.9545 - val_loss: 0.2096
Epoch 200/200
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 77ms/step - accuracy: 0.9726 - loss: 0.0393 - val_accuracy: 0.9545 - val_loss: 0.2572
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step - accuracy: 0.9545 - loss: 0.2572
Test Accuracy: 0.95
```