Machine Problem: Applications of Digital Signal Processing

Alein Miguel P. Capañarihan, Khatelyn Jhuneryll E. Deang, and Jude Nazi D. Hugo
Department of Electronics Engineering, Faculty of Engineering, University of Santo Tomas, Manila, Philippines,
Contact: aleinmiguel.capanarihan.eng@ust.edu.ph, khatelynjhuneryll.deang.eng@ust.edu.ph, judenazi.hugo.eng@ust.edu.ph

***Abstract* — This paper presents solutions to various applications of Digital Signal Processing, namely system identification, filter design, digital oscillator design, telephony, and image processing through MATLAB. Multiple concepts in signal processing will be utilized, such as impulse response, convolution and deconvolution, discrete-time Fourier Transform, and poles and zeros.**

Index Terms — System Identification, Filter Design, Digital Oscillator, Telephony, Image Processing, MATLAB, DTMF

I. Introduction

Signal Processing is an essential discipline in modern engineering, forming the foundation for advancements in communication, control, imaging, and many other fields. This paper explores the application of theoretical concepts in signal processing through targeted experiments and problem-solving tasks. These tasks encompass a range of topics, including transmission line analysis, filter design, digital oscillator implementation, Discrete Fourier Transform (DFT) in telephony, and image deconvolution, showcasing the versatility and depth of the field.

This analysis integrates MATLAB® as a primary tool for simulation and verification, allowing for hands-on exploration of key concepts. Each application is designed to enhance understanding of topics such as system identification, pole-zero placement, spectral analysis, and noise management. By bridging theory and practice, the tasks encourage iterative experimentation, critical thinking, and the development of practical problem-solving skills necessary for real-world engineering challenges.

Through these multifaceted applications, this paper not only reinforces the theoretical foundations of signal processing but also demonstrates their relevance in addressing complex, real-world problems. By documenting the methodologies, results, and insights in a structured format, this paper provides a comprehensive overview of how signal processing principles can be applied to modern technological contexts, preparing students for future challenges in engineering and innovation.

II. Applications

Application 1: Transmission Lines

Implementation

A system is defined as an object in which different variables interact at all kinds of time and space scales and that produce observable signals. It typically has three system variables in place: the *input u*, *disturbance w*, and the *system state x* [1].

> *Input u*: observable, measurable and can be manipulated by the user
>
> *Disturbance w:* possibly observable signal and cannot be manipulated by the user. If the disturbance is not measurable, it is indicated as system noise
>
> *State x*: This summarizes the effects of the input u and the disturbance w; hence the behavior of this system is affected by the input signals, and the laws governing the system's internal workings.

The portion aims to identify the unknown model parameters of the system state x, or in the case of this paper, the characteristics of the transmission line via System Identification. System identification is defined as the methodology for building mathematical models of dynamic systems using measurements of the input and output signals of the system [2]. This paper will not be completing the process of system identification, albeit the first step in system identification will be tackled, which is to measure the input and output signals of the system in both the time and frequency domain.

The objective is to generate various input signals, input said signals into *transline.p*, generate the input and output

graphs, as well as the magnitude response of the given system. The graphs are generated using the following:

```
plot(t,input_sequence1) %Time Domain
title(sprintf('Time Domain, Pulse Width =
%d',pulse_duration(1)))
xlabel('Time')
ylabel('Amplitude')

plot(f,abs(in_fft1) %Frequency Domain
title(sprintf('Frequency Domain, Pulse Width =
%d',pulse_duration(1)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')

plot(f,mag_response1); %Magnitude Response
title(sprintf('Pulse Width - %d'
,pulse_duration(1)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
```

Six various input sequencing functions will be utilized for the inputs: impulse, rectangular, hamming, hanning, kaiser, and Blackman-harris. These functions are especially useful in system identification, as they have various characteristics that enable more accurate analysis of the system's characteristics and its effects on the inputs. The code for the impulse function and its graphs are shown below.

```
case 'Impulse'
  x_signal(1) = 1; %Impulse
```

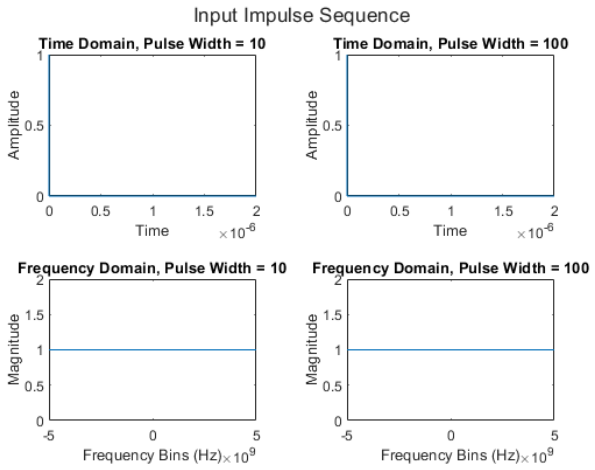

FIGURE 1.1. IMPULSE INPUT WITH VARYING PULSE WIDTH

It can be observed that varying the pulse-width of the impulse function has no effect, this is due to the fact that an impulse is just a singular spike in amplitude, and thus has no pulse width associated with it. The frequency response shown in Fig .1 also aligns with our understanding of the frequency spectrum of an impulse, which is the entire frequency spectrum.

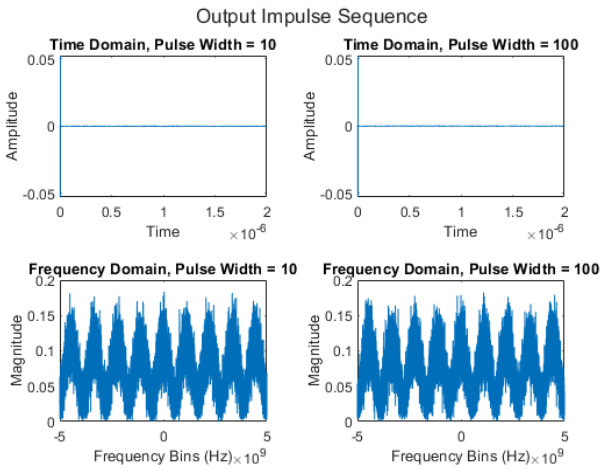

FIGURE 1.2. IMPULSE OUTPUT WITH VARYING PULSE WIDTHS

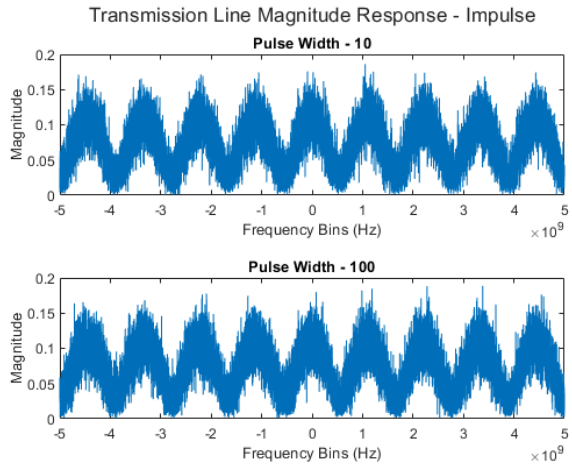

FIGURE 1.3. IMPULSE MAGNITUDE RESPONSE WITH VARYING PULSE WIDTH

Fig 1.2 -1.3 indicates that there is now noise all over the frequency spectrum, as the original horizontal line has now been converted to a sort of sinusoidal noise. This possibly indicates that the system is sinusoidal in nature and contains white noise.

The rectangular function is generated using the following code, and the graphs are shown below:

```
x_signal(1:pulse_duration(1)) = 1;
%Rectangular
```

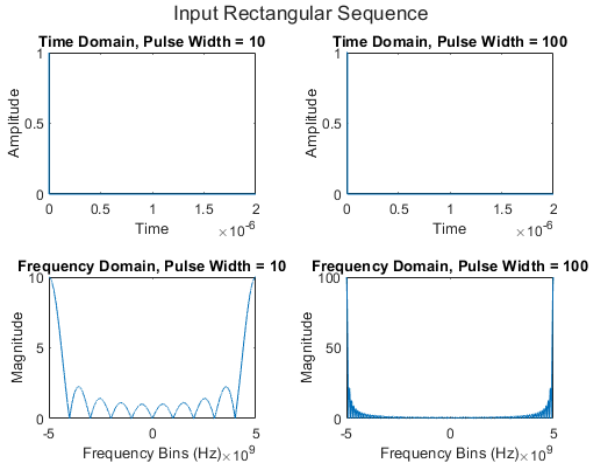

FIGURE 1.4. RECTANGULAR INPUT WITH VARYING PULSE WIDTH

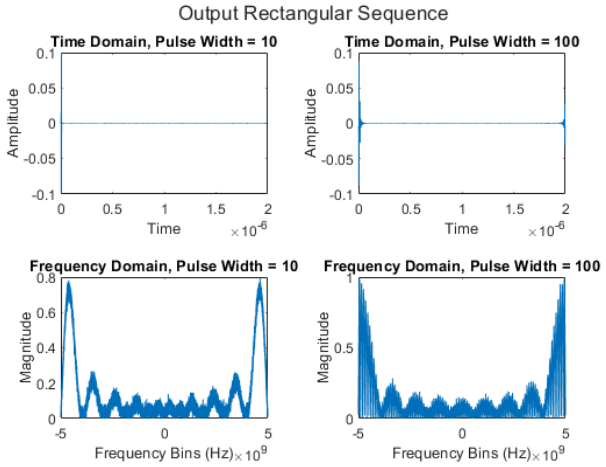

FIGURE 1.5. RECTANGULAR OUTPUT WITH VARYING PULSE WIDTH

Fig 1.4 now indicates that there is a change in the frequency domain when the pulse width is varied. It can be observed that as the pulse width increases, the frequency globe shifts away from the center, and there are less frequency globes overall. Meanwhile in Fig 1.5, we can see an increase in resolution of the signal i.e there are now more pronounced and thicker points on the frequency response of the signal. The same also holds true for Fig 1.6 as there are now more data peaks in between the frequency response, resembling the noisy sinusoidal signal in Fig 1.2.

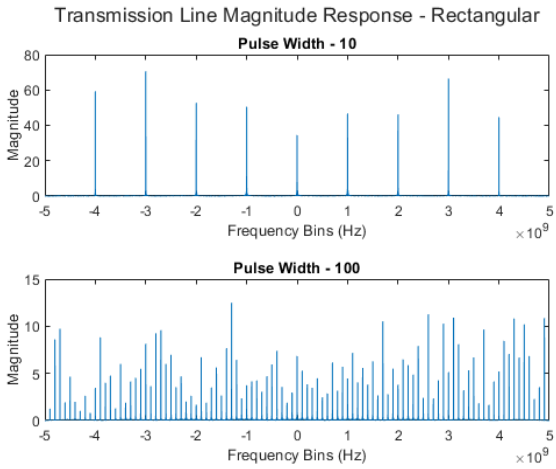

FIGURE 1.6. RECTANGULAR MAGNITUDE RESPONSE

The next type of input sequence are the hamming and the hanning window, both of which are highly similar so their codes and discussion will be discussed alongside one another, and the graphs for the hanning window shall be omitted, as the hamming window graphs are also highly representative of the time and frequency domain graphs of the hanning window.. The codes for these input sequences are shown, alongside the graphs.

```
x_signal(1:pulse_duration(1)) =
hamming(pulse_duration(1)); %Hamming
x_signal(1:pulse_duration(1)) =
hann(pulse_duration(1)); %Hanning
```

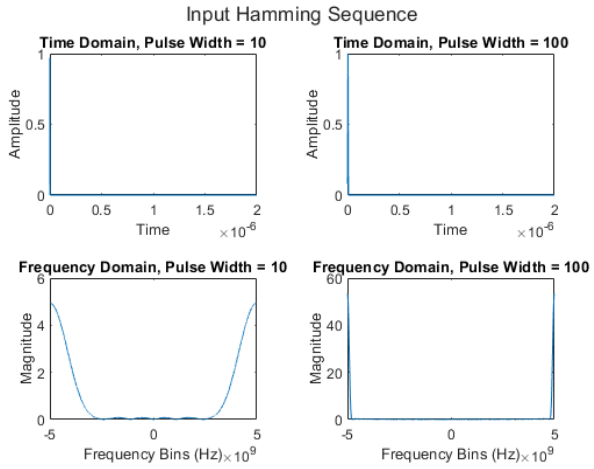

FIGURE 1.6. HAMMING INPUT WITH VARYING PULSE WIDTHS

It can be observed from Fig 1.6 that as the pulse width increases, the width of the sides decreases, there frequency globes on the hamming window are very minimal and can hardly be observed, especially at higher pulse widths

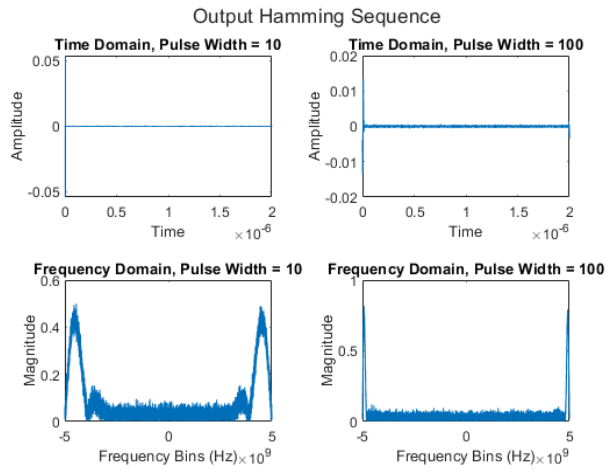

FIGURE 1.7 HAMMING OUTPUT WITH VARYING PULSE WIDTHS

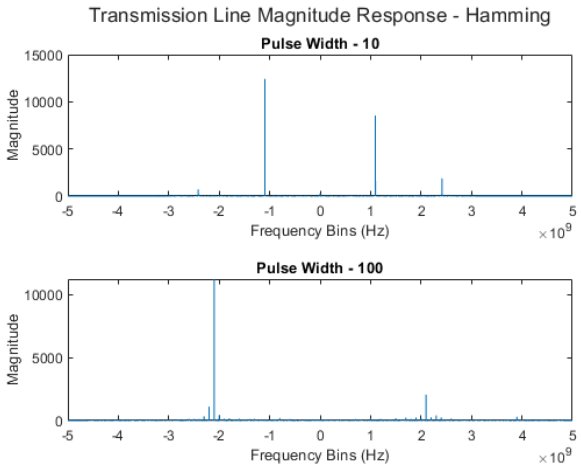

FIGURE 1.8 HAMMING MAGNITUDE RESPONSE

From analyzing Figure 1.7, the output frequency response closely mimics the input frequency response, although with greater noise all throughout the spectrum, most especially in between the two peaks at the ends of the graphs. Looking more closely at the frequency graphs of Fig 1.7-1.8. Changing the pulse width of the hamming window also effectively changed the frequency bins of the signal, thus the movement from 1Hz to 2Hz as seen in Figure 1.8.

The following code was used to generate the kaiser function, alongside its corresponding graphs

```
x_signal(1:pulse_duration(1)) =
kaiser(pulse_duration(1),5); %Kaiser
```

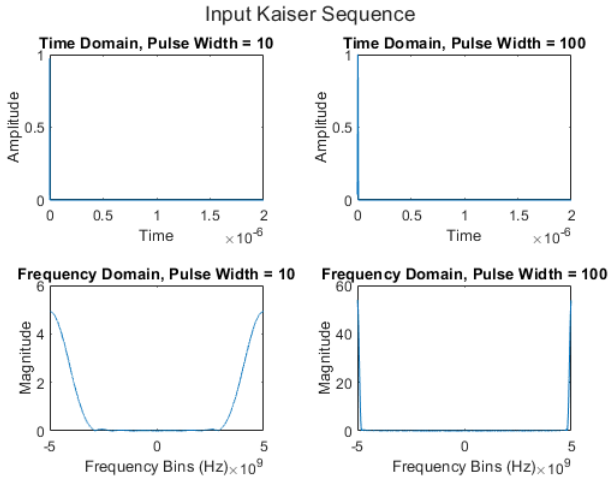

FIGURE 1.9. KAISER INPUT WITH VARYING PULSE WIDTHS

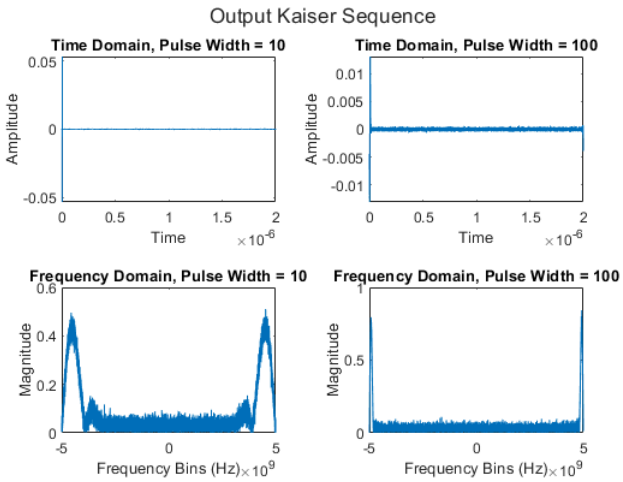

FIGURE 1.10. KAISER OUTPUT WITH VARYING PULSE WIDTHS

With the kaiser window, there is much more introduced noise in the output time domain sequence shown in Fig 1.10 as compared to the previous windowing techniques, but the same observations for the hamming and hanning window hold true for the kaiser window in the input portion as seen in Fig 1.9; increasing the pulse width reduces the thickness of the signal at the sides. The major changes occur at the magnitude response of the transmission line , at a pulse width of 10, the frequency response traces the positive half cycle generating the outline of a frequency globe. While pulse width at 100, it effectively spreads out the peaks. This is seen in Fig 1.11.

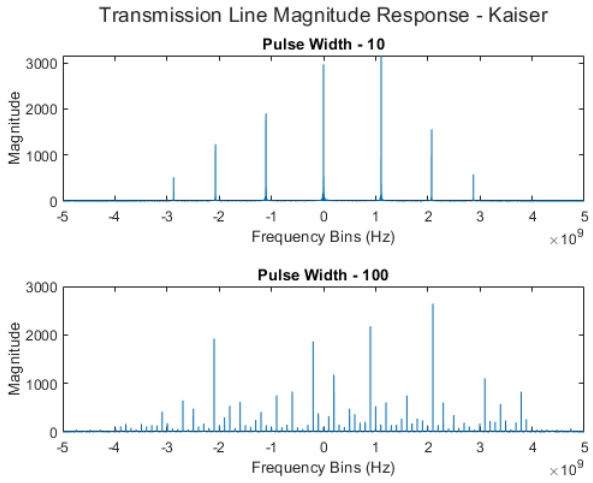

FIGURE 1.11 - KAISER MAGNITUDE RESPONSE

Lastly are the code and graphs for Blackmann-Harris, which are shown below

```
x_signal(1:pulse_duration(1)) =
blackmanharris(pulse_duration(1)); %BH
```

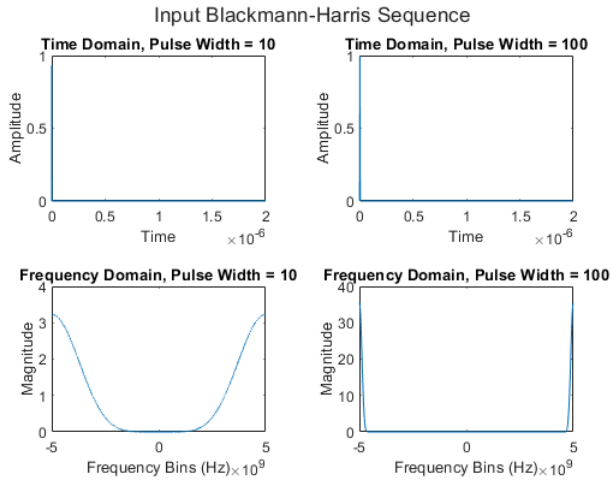

FIGURE 1.12. BLACKMAN-HARRIS INPUT WITH VARYING PULSE WIDTHS

The blackman-harris window is again quite similar to the kaiser window, with the variation that there is a secondary peak at the output frequency domain as shown in Fig 1.13. The biggest difference with the other windowing techniques lies in the magnitude response of the transmission line to the blackman-harris window. As the pulse width increases, there is a significant increase in the number of frequency peaks in the magnitude response of the transmission line as seen in Fig 1.14.

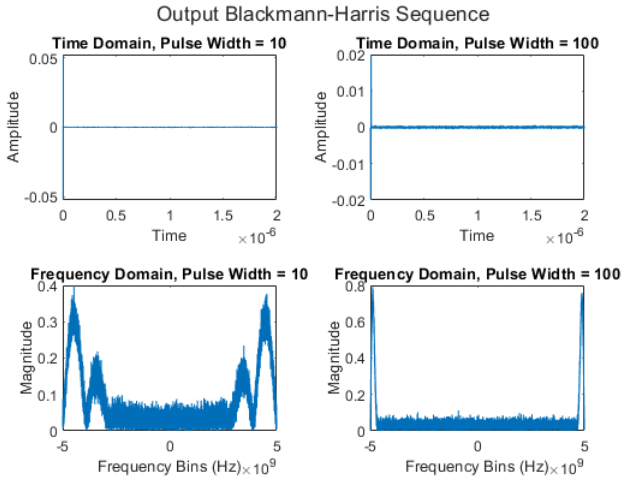

FIGURE 1.13. BLACKMAN-HARRIS OUTPUT WITH VARYING PULSE WIDTHS

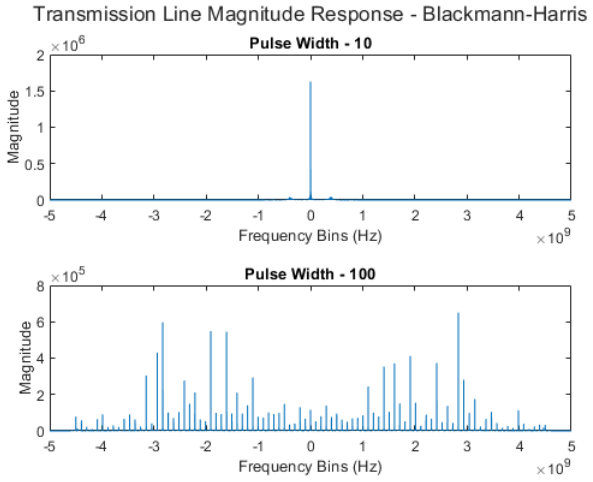

FIGURE 1.14. BLACKMAN-HARRIS OUTPUT WITH VARYING PULSE WIDTHS

ANALYSIS AND DISCUSSION OF RESULTS

This paper investigates the initial stage of system identification by examining the input and output signals of a transmission line in both time and frequency domains. Six input signals—impulse, rectangular, hamming, hanning, kaiser, and blackman-harris—were utilized to evaluate a system's response to different stimuli. Each input function demonstrated unique characteristics and interactions with the system, as outlined below.

The impulse function, characterized by mirroring the output frequency response and the magnitude response amplitude, showed no dependency on pulse width. The frequency response spanned the entire spectrum, proving property of an impulse signal spanning the entire frequency spectrum. However, the output showed that there was

sinusoidal noise across the frequency spectrum, suggesting the system's inherent sinusoidal nature and the presence of white noise.

The rectangular function showed a frequency response that varied with pulse width. As the pulse width increased, the central frequency shifted outward, and the number of frequency globes decreased. Additionally, the frequency resolution improved with pulse width, evidenced by thicker and more pronounced spectral peaks. This behavior highlights the system's sensitivity to changes in the duration of the rectangular input. The system's output also introduced sinusoidal noise, similar to the impulse response, further reinforcing the sinusoidal noise characteristics of the system.

The hamming and hanning windows, which are highly similar, displayed minimal observable frequency globes, especially at an increased pulse width. The frequency response of the output closely mirrored the input, but with the addition of significant noise between spectral peaks. Changes in pulse width shifted the frequency bins and reduced the width of side lobes in the response, demonstrating the impact of these windows on spectral leakage. These findings indicate that these windows are effective in reducing side lobes while preserving the main lobe characteristics.

The kaiser window introduced a higher level of noise in the output time-domain sequence compared to previous windowing techniques. As the pulse width increased, the thickness of the side lobes in the spectral response decreased, consistent with other window types. Notably, at higher pulse widths, the frequency response spread out the peaks, resulting in a more distributed spectrum.

The blackman-harris showed a secondary peak in the output response. Increasing the pulse width led to a significant increase in the number of frequency peaks within the magnitude response of the transmission line. This characteristic sets the blackman-harris window apart from other techniques, as it captures additional frequency components and provides higher spectral resolution.

In conclusion the distinct effects of various input signals on the dynamic behavior of the transmission line system were observed. Across all input types, the system consistently introduced sinusoidal noise into the output, suggesting an underlying sinusoidal nature and the presence of white noise. This behavior persisted regardless of the input signal type, indicating that the system's internal characteristics heavily influence its response.

The analysis of windowing functions revealed important insights into their role in shaping the system's frequency response. The impulse function provided a baseline,with the frequency response showing the full range of the output . The rectangular function demonstrated sensitivity to pulse width, with increased pulse widths enhancing frequency resolution and reducing central frequency concentration. The hamming and hanning windows effectively minimized spectral leakage, while the kaiser window exhibited greater flexibility in controlling the trade-off between spectral resolution and side lobe attenuation. Finally, the blackman-harris window stood out for its ability to reveal additional frequency components and provide superior spectral resolution.

These results highlight the importance of selecting appropriate input functions for system identification. By leveraging these observations, future work can focus on parameter estimation and developing a mathematical model that captures the transmission line's dynamic characteristics. The findings lay a strong foundation for advancing the system identification process and improving the understanding of the system's internal mechanisms.

Application 2: Filter Design

IMPLEMENTATION

Filter design is essential in signal processing for selectively passing or suppressing specific frequency components. The pole-zero placement method is a common approach for designing filters, where the locations of poles and zeros on the complex plane determine the system's frequency response. By adjusting these placements, the filter can approximate a desired magnitude response effectively. This method is particularly useful for Infinite Impulse Response (IIR) filters, which achieve sharp transitions with lower filter orders. The filter's performance is evaluated through its pole-zero plot, magnitude response, and phase response, ensuring it meets the design requirements (Proakis & Manolakis, 2007). [3]

The objective is to design an 8th-order filter using the pole-zero placement approach to approximate a given magnitude response. The magnitude response is shown in the plot, and the task involves adjusting the poles and zeros using trial and error to match this response. Moreover, the presentation of the pole-zero plot, magnitude response, and phase of the design are all displayed in this portion of the paper. Additionally, after passing a given input signal through the designed filter, the output was plotted and analyzed in both the time and frequency domains.

```
% Load the H.mat file
load('H.mat');

% Calculate magnitude and phase
magnitude = abs(h);  % Magnitude of the
frequency response

% Define parameters for plotting
N = length(h);
w_n = (0:N-1) / N;  % Normalized frequency

% Magnitude Response
figure;
plot(w_n, 20 * log10(magnitude));  % Plot
magnitude in dB
xlabel('Normalized Frequency');
ylabel('Magnitude (dB)');
```

```
title('Magnitude Response');
grid on;
```

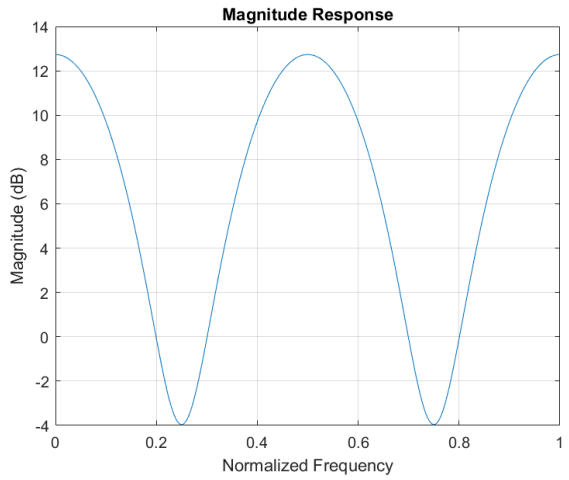

FIGURE 2.1. MAGNITUDE SPECTRUM OF THE H.MAT FILE

The code above provides the plot of the magnitude response of the system. Figure 2.1 will serve as the reference target, as the goal is to approximate a magnitude response closely resembling that of this plot.

```
% Approximated zeros (z) and poles (p)
p = [0.3492 + 0.4875i; -0.3492 + 0.4875i;
0.4875 + 0.3492i; -0.4845 + 0.3492i; ...
0.3492 - 0.4875i; -0.3492 - 0.4875i; 0.4875 -
0.3492i; -0.4845 - 0.3492i];
z = [0.8199 + 0.7900i; -0.8199 + 0.7900i;
0.7900 + 0.8199i; -0.7900 + 0.8199i; ...
0.8199 - 0.7900i; -0.8199 - 0.7900i; 0.7900 -
0.8199i; -0.7900 - 0.8199i];
```

The zeros (*z*) and poles (*p*) in this code represent approximated values determined through an iterative trial-and-error process to achieve the desired target magnitude response. Furthermore, the poles are positioned inside near the unit circle with complex conjugate pairs, ensuring system stability and shaping the filter's response. Similarly, the zeros are strategically placed outside the unit circle in complex conjugate pairs to cancel specific frequencies and refine the magnitude response. This careful adjustment of zeros and poles contributes to achieving the precise frequency characteristics required for the system.

```
% Generate the numerator (zeros) and
denominator (poles) coefficients
num = poly(z); % Numerator polynomial from the
zeros
denom = poly(p); % Denominator polynomial from
the poles

% Visualize the filter response
fvtool(num, denom); % Use FVTool for better
visualization of frequency response
```

This code generates the numerator (*N*) and denominator (*D*) coefficients of the transfer function by calculating polynomials from the provided zeros (*z*) and poles (*p*) using the *poly* function. These coefficients define the filter's transfer function. The *fvtool* function is then used to visualize the frequency response of the filter, providing a detailed and interactive view of its characteristics such as magnitude and phase responses.

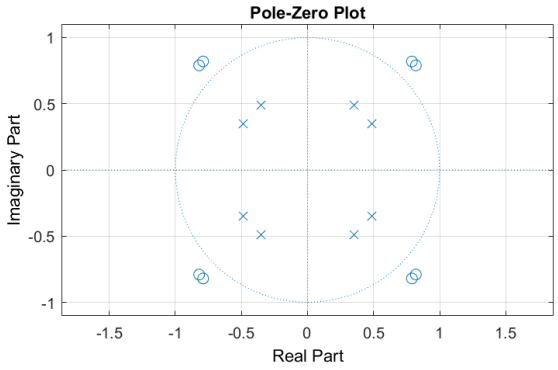

FIGURE 2.2. POLE-ZERO PLOT OF THE POLES AND ZEROS

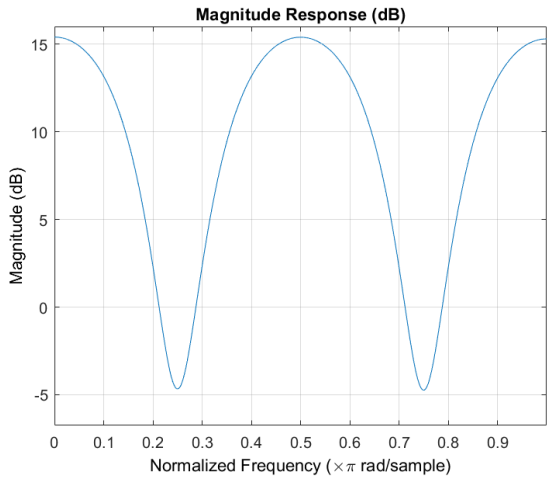

FIGURE 2.3. MAGNITUDE RESPONSE OF THE POLES AND ZEROS

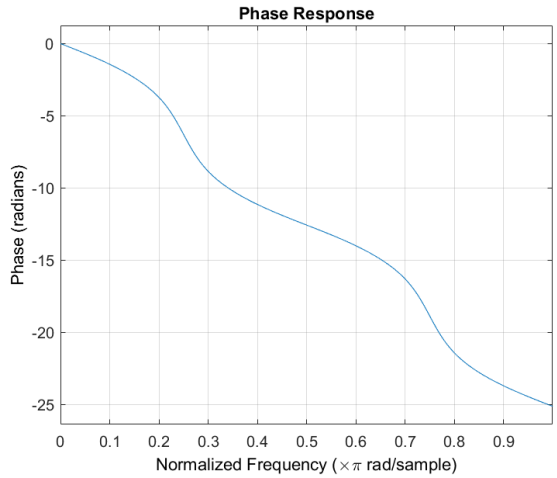

FIGURE 2.4. PHASE RESPONSE OF THE POLES AND ZEROS

The first plot, Figure 2.2, is a pole-zero plot, which visually depicts the system's poles and zeros on the complex plane. In line with this, the unit circle provides a reference, helping analyze system stability and frequency response, as poles inside the circle indicate stability. Moving to Figure 2.3, it is observed that the magnitude response in decibels (dB) shows how the system attenuates or amplifies signals at different frequencies. The repeating peaks and dips in the magnitude response highlight the system's filtering behavior, suggesting that it allows certain frequencies to pass while suppressing others. It is also observed that the approximation of poles and zeros for the magnitude response matches that of the provided *H.mat* file. Moreover, Figure 2.4 displays the phase response of the system, which shows how the phase of the output signal varies with normalized frequency. This gradual decrease in phase indicates the phase shift introduced by the system across different frequencies.

$$x(n) = \left[\cos\left(\frac{3\pi}{20}n\right) + \cos\left(\frac{\pi}{2}n\right) + \cos\left(\frac{3\pi}{4}n\right)\right]u(n)$$

EQUATION 2.1. PROVIDED INPUT SIGNAL

Upon completion of the designed system, it is now ready to process an input signal. The input signal, as described by the given equation above, should be passed through the system. Subsequently, the output of the system must be plotted and analyzed in both the time and frequency domains.

```
% Define the input signal
n = 0:499; % 500 samples
u = (n >= 0); % Unit step function u(n) using
logical indexing
x = (cos(3*pi*n/20) + cos(pi*n/2) +
cos(3*pi*n/4)) .* u; % Input signal with u(n)

% Apply the filter
y = filter(num, denom, x);
```

As for this section, the code defines an input signal *x(n)*, which is composed of a sum of three cosine components modulated by a unit step function *u(n)*, thereby ensuring the signal is causal. Subsequently, the signal is processed through a filter with numerator (*N*) and denominator (*D*) coefficients. Using the *filter* function, the output signal *y(n)* is computed, effectively applying the defined transfer function to the input.

```
% Frequency-domain analysis
X = abs(fft(x, 1024)); % FFT of input signal
Y = abs(fft(y, 1024)); % FFT of output signal
w = (0:511) / 512; % Normalized frequency axis
for half-spectrum (512 points)
```

This section performs a frequency-domain analysis of the input and output signals. The *fft* function is used to compute the magnitude spectra of the input signal (x) and the filtered output signal (y), with an FFT size of 1024 for improved frequency resolution. The variable *w* represents the normalized frequency axis, spanning 0 to 1, corresponding to the Nyquist range in normalized units.

```
% Time-domain plot
figure;
subplot(2, 1, 1);
plot(n, x, 'b', 'DisplayName', 'Input
Signal');
hold on;
plot(n, y, 'r', 'DisplayName', 'Filtered
Signal');
xlabel('Samples (n)');
ylabel('Amplitude');
title('Time Domain Signal');
legend;
grid on;

% Frequency-domain plot
subplot(2, 1, 2);
plot(w, (X(1:512)), 'b', 'DisplayName', 'Input
Spectrum'); % Corrected to match lengths
hold on;
plot(w, (Y(1:512)), 'r', 'DisplayName',
'Filtered Spectrum'); % Corrected to match
lengths
xlabel('Normalized Frequency');
ylabel('Magnitude (dB)');
title('Frequency Domain Signal');
legend;
grid on;
```

In this portion of the code, it visualizes the input and filtered signals in both time and frequency domains. The time-domain plot compares their amplitude over 500 samples, clearly distinguishing the signals with legends and grid lines. Moreover, the frequency-domain plot, using FFT, highlights their spectral content over the normalized range 0 to 1, showcasing the filtering effects on the signal's frequency components.

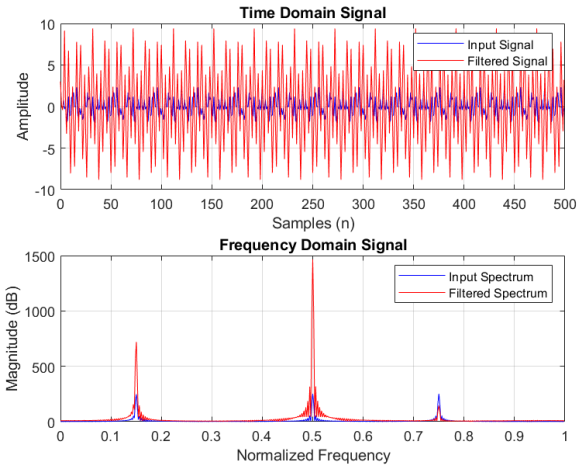

FIGURE 2.5. TIME AND FREQUENCY DOMAIN OF THE SIGNAL

Figure 2.5 illustrates the impact of filtering in both the time and frequency domains. In the time-domain subplot, the input signal (blue) and the filtered signal (red) reveal how the filter modifies the original signal by enhancing or attenuating specific components. Transitioning to the frequency-domain subplot, the magnitude spectra of the input and filtered signals highlight significant changes, with the filtered spectrum (red) showing suppression or amplification at certain frequency bands compared to the input spectrum (blue). This visualization effectively demonstrates the filter's influence on the signal, both in shaping its time-domain behavior and altering its frequency content.

ANALYSIS AND DISCUSSION OF RESULTS

The placement of poles and zeros, along with their approximation, is fundamental in shaping the filter's performance and behavior. The poles, positioned near the unit circle in complex conjugate pairs, ensure system stability while enabling sharp transitions in the frequency response. The zeros, carefully placed outside the unit circle, are also in complex conjugate pairs to effectively cancel specific unwanted frequency components and refine the filter's selectivity. These poles and zeros were approximated through an iterative trial-and-error process to closely match the target magnitude response. This strategic arrangement and fine-tuned approximation form the foundation of the filter's ability to isolate desired frequencies while suppressing others.

In terms of the results, the filter demonstrates clear success in modifying the signal's characteristics, both in the time and frequency domains. In the time-domain analysis, the filtered signal exhibits a noticeable increase in amplitude compared to the input signal. This increase suggests that the filter has amplified certain frequency components while suppressing others. Unlike what might be expected in terms of noise reduction, the filter appears to enhance the desired frequencies, as evidenced by the more pronounced peaks in the filtered signal. This outcome highlights the importance of the precise pole-zero approximation and placement in achieving the desired signal modification.

Furthermore, the frequency-domain analysis provides additional evidence of the filter's effectiveness. The magnitude spectrum of the filtered signal highlights substantial attenuation in unwanted frequency ranges, while the desired frequencies are allowed to pass through with minimal distortion. The alignment of the peaks in the filtered spectrum with the target frequency bands confirms the success of the pole-zero approximation and placement in shaping the magnitude response. By comparing the input and filtered spectra, it becomes evident that the filter achieves its intended goal of enhancing desired frequencies while attenuating irrelevant or undesired ones, showcasing the robustness of the design.

Application 3: Digital Oscillator Design

IMPLEMENTATION

Digital oscillators are essential components in signal processing, used to generate periodic waveforms such as sine, cosine, and more complex signals. Unlike analog oscillators, digital oscillators work in discrete time, making them suitable for digital systems like communication devices and audio processing. The design of a digital oscillator often involves using recursive difference equations or parallel components, where each component generates a specific frequency component of the desired signal. This approach ensures flexibility, efficiency, and accuracy. The system's performance is typically verified through simulations to ensure stability and precision, especially at high sampling frequencies, as in the case of a 100 kHz sampling rate (Proakis & Manolakis, 2016). [4]

The objective of this task is to design a digital oscillator that generates the signal *x(n)*, as specified, when excited by an impulse signal with a sampling frequency of 100 kHz. The desired signal is the sum of cosine functions, as outlined in the provided equation. The design should be computed, and the system simulated using MATLAB. Furthermore, the performance of the system must be verified. It is recommended that the system be represented using parallel components during both the design and simulation processes to ensure accurate implementation and functionality.

$$x(n) = \left[cos\left(\frac{3\pi}{20}n\right) + cos\left(\frac{\pi}{2}n\right) + cos\left(\frac{3\pi}{4}n\right)\right]u(n)$$

EQUATION 3.1. PROVIDED INPUT SIGNAL

In alignment with the previous application, the same input signal will now be applied to the design of the digital oscillator. Specifically, the system is designed to produce the signal *x(n)* when excited by an impulse signal, as outlined in the previous item. This input will be passed through the digital oscillator to evaluate its response and performance, ensuring

that the designed system generates the expected output. This step is essential for verifying the proper functionality of the digital oscillator and confirming that it meets the required specifications.

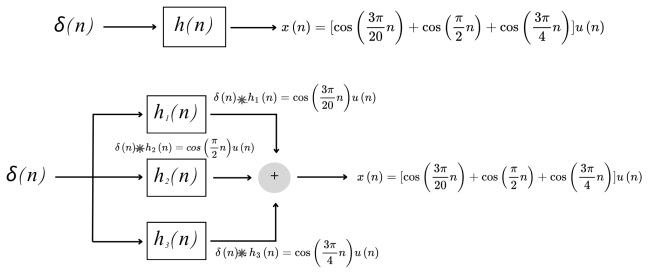

FIGURE 3.1. IMPLEMENTATION OF THE DIGITAL OSCILLATOR DESIGN

The figure illustrates the implementation of the digital oscillator design, which generates a signal by processing an impulse input *δ(n)*. The system consists of parallel components, where the impulse input is passed through individual filters, each corresponding to a specific frequency component. The outputs of these filters are then combined to produce the final signal, *x(n)*.

Note that: X(Z) = *δ(Z) = 1*

$$H(Z) = \frac{Y(Z)}{X(Z)} = \frac{Y(Z)}{1} = Y(Z)$$

$$y_1(n) = cos(\frac{3\pi}{20}n)u(n)$$

$$H_1(Z) = Y_1(Z) = \frac{1-cos(\frac{3\pi}{20})z^{-1}}{1-2\,cos(\frac{3\pi}{20})z^{-1}+z^{-2}}$$

$$y_2(n) = cos(\frac{\pi}{2}n)u(n)$$

$$H_2(Z) = Y_2(Z) = \frac{1-cos(\frac{\pi}{2})z^{-1}}{1-2\,cos(\frac{\pi}{2})z^{-1}+z^{-2}}$$

$$y_3(n) = cos(\frac{3\pi}{4}n)u(n)$$

$$H_3(Z) = Y_3(Z) = \frac{1-cos(\frac{3\pi}{4})z^{-1}}{1-2\,cos(\frac{3\pi}{4})z^{-1}+z^{-2}}$$

EQUATION 3.2. COMPUTATION OF PARALLEL COMPONENTS

As outlined in Equation 3.2, the equation demonstrates the manual computation of the parallel components for the digital oscillator design. Initially, it is noted that the expression *X(Z) = δ(Z) = 1* signifies that the input signal in the Z-domain is represented by the delta function, which has a value of 1. This indicates that the system is excited by an impulse signal, and its Z-domain representation is simply 1. Each parallel component, $y_1(n)$, $y_2(n)$, and $y_3(n)$, is associated with a transfer function $H_1(Z)$, $H_2(Z)$, and $H_3(Z)$, respectively. These transfer functions are derived using cosine terms with different angular frequencies. Furthermore, the numerator and denominator coefficients of each transfer function are essential, as they are required for implementing the corresponding filters in the MATLAB code. These coefficients, determined by the system's frequency response, define the behavior of each filter in the Z-domain. Thus, the computation of these components is crucial for generating the desired output, as the individual filter responses are combined to form the complete oscillator signal.

```
% Sampling frequency
fs = 100e3;
% Angular frequencies
w1 = 3*pi/20;
w2 = pi/2;
w3 = 3*pi/4;

% Difference equation coefficients
b1 = [1, -cos(w1)]; % Numerator
b2 = [1, -cos(w2)]; % Numerator
b3 = [1, -cos(w3)]; % Numerator
a1 = [1, -2*cos(w1), 1]; % Denominator for w1
a2 = [1, -2*cos(w2), 1]; % Denominator for w2
a3 = [1, -2*cos(w3), 1]; % Denominator for w3
```

In this portion of the code, it sets up the foundation for implementing a digital oscillator by defining key parameters and coefficients for the difference equations. First, the sampling frequency (*fs*) is specified as 100 kHz, ensuring the system operates at the required resolution. Next, the angular frequencies (*w1, w2, w3*) corresponding to the desired frequency components of the output signal are calculated. Following this, the numerator coefficients (*b1, b2, b3*) for the difference equations are determined, which define the feedforward part of the filters. Similarly, the denominator coefficients (*a1, a2, a3*) are computed to specify the feedback portion of the filters. These coefficients are essential as they will be utilized in the MATLAB implementation to design filters for each frequency component and ensure the accurate synthesis of the digital oscillator's output.

```
% Input impulse signal
n = 0:249;
impulse = (n == 0);

% Filter responses
y1 = filter(b1, a1, impulse);
y2 = filter(b2, a2, impulse);
y3 = filter(b3, a3, impulse);
```

This section generates an impulse signal and calculates the filter responses for the digital oscillator. The impulse signal, with a length of 250 samples, is used to excite the system. Using the previously defined coefficients, the filter function computes the outputs (*y1, y2, and y3*) for each frequency component. These responses represent the individual building blocks of the digital oscillator's output.

```
% Combine outputs
x_combined = y1 + y2 + y3;

% Define the expected output
x_expected = (cos(3*pi/20 * n) + cos(pi/2 * n)
+ cos(3*pi/4 * n)) .* (n >= 0);
```

This MATLAB code combines the outputs of the individual filters to generate the complete signal for the digital oscillator. The outputs *y1, y2,* and *y3* are summed together to produce the combined output (*x_combined*), representing the final signal of the digital oscillator. Additionally, the expected output (*x_expected*) is defined using the cosine functions for the corresponding frequencies, and it is coded for comparison with the digital oscillator's output. This allows for a comparison between the designed oscillator output and the expected theoretical signal.

```
% Plot y1
figure;
subplot(3, 1, 1);
plot(n, y1);
xlabel('n');
ylabel('y_1(n)');
title('Impulse Response for w1');
grid on;

% Plot y2
subplot(3, 1, 2);
plot(n, y2);
xlabel('n');
ylabel('y_2(n)');
title('Impulse Response for w2');
grid on;

% Plot y3
subplot(3, 1, 3);
plot(n, y3);
xlabel('n');
ylabel('y_3(n)');
title('Impulse Response for w3');
grid on;
```

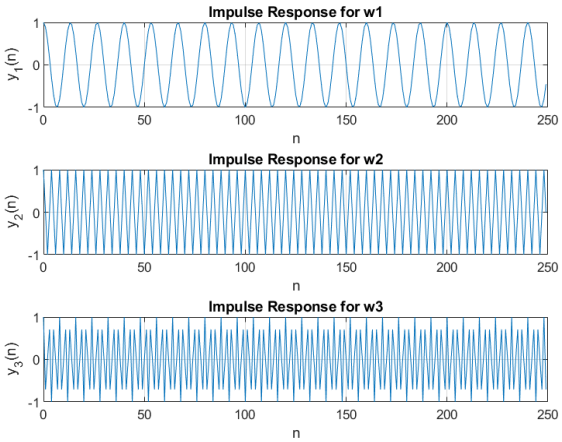

FIGURE 3.2. IMPULSE RESPONSES PLOT

The code above plots the impulse responses of the filters used in the digital oscillator (the generated plots are shown on Figure 3.2). First, the impulse response *y1* is plotted in the first subplot, representing the filter for angular frequency *w1*. Next, the impulse response *y2* is shown in the second subplot for the filter corresponding to *w2*. Finally, the third subplot displays the impulse response *y3* for the filter with angular frequency *w3*.

```
% Plot combined output
figure;
subplot(3, 1, 1);
plot(n, x_combined);
xlabel('n');
ylabel('x_{combined}(n)');
title('Combined Output');
grid on;

% Plot expected output
subplot(3, 1, 2);
plot(n, x_expected);
xlabel('n');
ylabel('x_{expected}(n)');
title('Expected Output');
grid on;

% Compare combined and expected outputs
subplot(3, 1, 3);
plot(n, x_combined, 'b', n, x_expected,
'r--');
legend('Combined', 'Expected');
xlabel('n');
ylabel('x(n)');
title('Comparison of Outputs');
grid on;
```

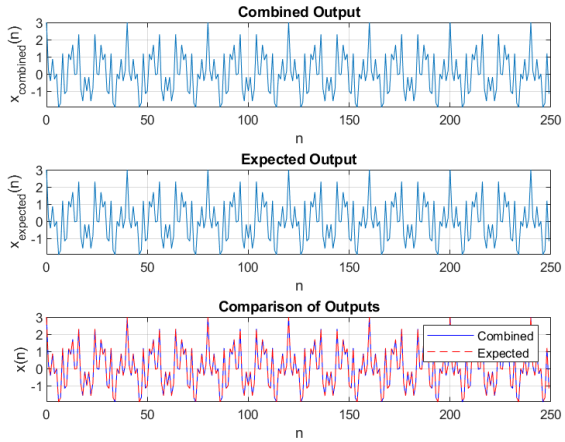

FIGURE 3.3. COMPARISON OF COMBINED AND EXPECTED OUTPUT

The generated subplots display the graphs for both the combined output of the digital oscillator and the expected output, along with a comparison of the two. Upon reviewing these plots, it is evident that both the combined output and the expected output yield identical results. This observation

confirms that the digital oscillator has been successfully designed and implemented, as its output aligns perfectly with the theoretically anticipated signal. The close match between the two signals further validates the accuracy of the oscillator's performance and its adherence to the desired specifications.

Analysis and Discussion of Results

The digital oscillator design effectively achieved its objective of generating the desired signal *x(n)*, composed of multiple cosine components, by employing parallel filtering components. Upon comparing the combined output of the oscillator with the expected theoretical signal, the results demonstrated a perfect match. This alignment indicates that the design and implementation were accurate, reflecting the precision of the derived mathematical models and their successful translation into the practical system.

The impulse response plots for each angular frequency (*w1, w2, w3*) provided further evidence of the system's correctness. Each filter accurately generated its respective frequency component, confirming that the recursive difference equations were implemented effectively. When combined, these components reconstructed the target signal, verifying the modularity and reliability of the parallel component design. This approach not only facilitated ease of debugging but also allowed each frequency component to be verified independently, enhancing the overall robustness of the system.

Moreover, the utilization of a high sampling frequency of 100kHz proved instrumental in ensuring accurate signal synthesis. This high sampling rate minimized aliasing effects, preserving the integrity of the generated frequency components. Additionally, the stability of the system was validated by the carefully calculated coefficients of the recursive difference equations. The placement of poles within the unit circle in the Z-plane guaranteed a bounded and predictable output, further solidifying the system's reliability.

The MATLAB implementation played a crucial role in verifying the system's performance. Using the *filter* function, the impulse input was accurately processed through each filter, yielding results that closely aligned with the theoretical predictions. The transfer functions derived for the individual cosine components were implemented correctly, as evidenced by the combined output precisely matching the expected signal. This outcome highlights the effectiveness of translating theoretical derivations into practical applications.

In conclusion, the digital oscillator design successfully fulfilled its requirements, demonstrating its effectiveness and accuracy. As mentioned, the precise alignment of the combined and expected signals underscores the reliability of the parallel filter approach. Furthermore, this design methodology offers scalability, allowing for future modifications, such as the inclusion of additional signal components or the introduction of modulation techniques. These results highlight the potential of the digital oscillator for applications in communication systems, audio processing, and other fields requiring stable and accurate signal generation.

Application 4: Discrete Fourier Transform in Telephony

Implementation

The Discrete Fourier Transform (DFT) is a crucial technique in telephony for analyzing touch-tone signals, also known as Dual-Tone Multi-Frequency (DTMF) signals. Each button press on a telephone keypad generates a unique combination of two sine wave frequencies—one from a low-frequency group and another from a high-frequency group. By applying the DFT to the sampled signal, these dominant frequency components can be efficiently extracted and analyzed, enabling accurate identification of the pressed key. This method is widely employed in digital communication systems for tasks such as call routing and interactive voice response, ensuring reliable and efficient signal processing (Ingle & Proakis, 2012). [5]

This application involves the use of the Discrete Fourier Transform (DFT) to analyze a digital telephone signal, where each button press corresponds to a combination of two sine wave frequencies. To begin, the application requires extracting a 256-sample segment from a signal sampled at 8 kHz, followed by the application of a windowing function and the computation of the 256-point DFT. The primary objective is to identify the indices in the resulting DFT vector that exhibit the highest amplitude, which corresponds to the pressed button at the time of recording. Specifically, in part (A), the analysis will focus on identifying the frequencies associated with button 9. Subsequently, in part (B), MATLAB code is to be written to process a recorded touch-tone signal from the file `*touchtone.wav*` and determine the sequence of button presses by analyzing the frequency pair patterns.

Part A. Calculation of Indices with Highest Amplitude in DFT of Windowed Signal

Manual Calculation of Indices

Sampling Frequency: $f_S = 8000\ Hz$
Frequencies for Button 9: $f_{LOW} = 852\ Hz, f_{HIGH} = 1477\ Hz$
DTF Size: $N = 256$

Indices Formula: $k = \frac{f}{fs}(N)$

$$k_{LOW} = \frac{852\ Hz}{8000\ Hz}(256) = 27.264 = \underline{27}$$

$$k_{HIGH} = \frac{1477\ Hz}{8000\ Hz}(256) = 47.264 = \underline{47}$$

Equation 4.1. Manual Calculation of Indices

The manual calculation of indices involves determining the frequency indices for given frequencies using the sampling

frequency and the DFT size. With a sampling frequency (f_S) of 8000 Hz, a low frequency (f_{LOW}) of 852 Hz, a high frequency (f_{HIGH}) of 1477 Hz, and a DFT size (N) of 256, the formula $k=\frac{f}{fs}(N)$ is applied. The calculated index for f_{LOW} is 27.264, and for f_{HIGH}, it is 47.264.

MATLAB Code Implementation

```
fs = 8000; % Sampling frequency in Hz
N = 256;   % DFT size (number of points)
frequencies = [852, 1477]; % Frequencies for
button 9 in Hz
```

The code snippet above sets up parameters for analyzing a telephone signal using the Discrete Fourier Transform (DFT). The sampling frequency (f_S) is defined as 8000 Hz, representing the rate at which the signal is sampled. The DFT size (N) is specified as 256 points, which determines the resolution of the frequency spectrum. The array *frequencies* contain the two specific frequencies, 852 Hz and 1477 Hz, associated with the button "9" on a telephone keypad based on the Dual-Tone Multi-Frequency (DTMF) signaling system.

```
% Calculate DFT indices for the given
frequencies
indices = round(frequencies / fs * N);
```

This code calculates the DFT indices corresponding to the specified frequencies in the *frequencies* array. By dividing each frequency by the sampling frequency (f_S) and multiplying by the DFT size (N), the code determines the normalized frequency positions in the DFT spectrum. The *round* function ensures the indices are integers, as required for indexing the DFT output. This allows mapping the given frequencies to their respective bins in the DFT.

```
% Display the indices
disp('DFT Indices for button 9');
fprintf('Frequency 852 Hz: %d\n', indices(1));
fprintf('Frequency 1477 Hz: %d\n',
indices(2));
```

This code displays the DFT indices corresponding to the frequencies for button "9" (852 Hz and 1477 Hz). The *disp* function outputs a label indicating the purpose of the indices, and the *fprintf* function prints the calculated indices for each frequency in a formatted manner. This provides a clear mapping of the frequencies to their respective positions in the DFT spectrum.

```
   >> DFT_Indices
DFT Indices for button 9
Frequency 852 Hz: 27
Frequency 1477 Hz: 47
```

The output displays the calculated DFT indices for the frequencies associated with button "9" on a telephone keypad. Specifically, the frequency 852 Hz maps to index 27, and 1477 Hz maps to index 47 in the DFT spectrum. These indices indicate where in the DFT the corresponding frequency components can be observed.

```
% Generate the signal corresponding to button
9 (two sine waves at 852 Hz and 1477 Hz)
signal = sin(2 * pi * frequencies(1) * n / fs)
+ sin(2 * pi * frequencies(2) * n / fs);

% Apply a Hanning window to reduce spectral
leakage
windowed_signal = signal .* hanning(N)';

% Perform the FFT on the windowed signal
x_fft = fft(windowed_signal, N);

% Plot the magnitude spectrum of the DFT
figure;
plot(n, abs(x_fft));  % Plot the magnitude of
the DFT coefficients
xlabel("Sample Index");
ylabel("Magnitude");
title("Magnitude Spectrum of Button 9
Signal");
grid on;

% Highlight the peaks at 852 Hz and 1477 Hz in
the frequency spectrum
hold on;
plot(indices(1), abs(x_fft(indices(1)+1)),
'o', 'MarkerSize', 10, 'Linewidth', 1,
'Color', [1 0 1]);  % Peak at 852 Hz
plot(indices(2), abs(x_fft(indices(2)+1)),
'o', 'MarkerSize', 10, 'Linewidth', 1,
'Color', [0 0 1]);  % Peak at 1477 Hz

% Add a legend to explain the plot
legend("Magnitude Spectrum", "Peak at 852 Hz",
"Peak at 1477 Hz");
hold off;
```

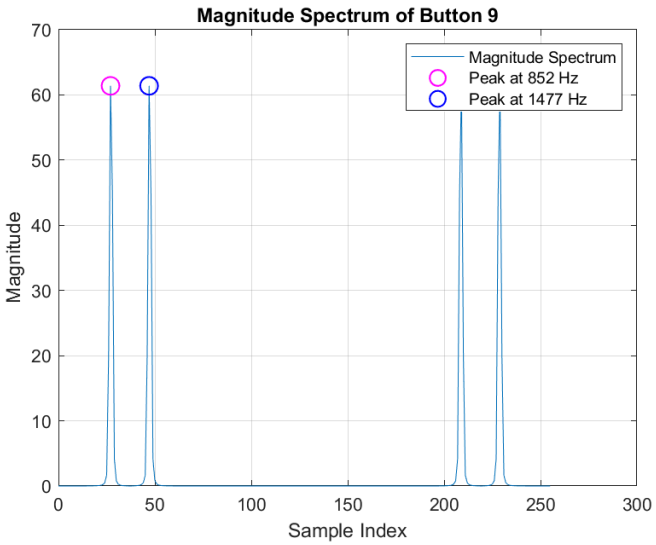

FIGURE 4.1. MAGNITUDE SPECTRUM OF BUTTON 9

This MATLAB code generates a signal for button 9 of a touch-tone keypad, which consists of two sine waves at 852

Hz and 1477 Hz. The signal is first generated using the *sin()* function, which creates the individual sine waves. A Hanning window is applied to the signal using the *hanning()* function to reduce spectral leakage before performing the Fast Fourier Transform (FFT) on the windowed signal using the *fft()* function. The resulting frequency spectrum is plotted using *plot()*, with the peaks at 852 Hz and 1477 Hz highlighted. The *legend()* function is used to add labels to the plot, clarifying the displayed information. Moreover, the plot above shows the magnitude spectrum of the button 9 signal, with peaks at sample indices 27 and 47 corresponding to the frequencies 852 Hz and 1477 Hz, respectively, indicating the DTMF tones for button 9.

Part B. MATLAB Code for Identifying Button Sequences from Touch-Tone Signals

```
% Loading the Given Audio File
[audio_data, sampling_rate] =
    audioread("touchtone.wav");  % Read the
    audio file
num_samples = length(audio_data);  % Get the
    number of samples in the audio data
```

This section of the code loads the audio file *touchtone.wav* using the *audioread* function, extracting the audio signal into the variable *audio_data* and the sampling rate into *sampling_rate*. The total number of samples in the audio signal is calculated and stored in *num_samples* using the *length* function, which provides the signal's size and prepares it for further analysis.

```
% FFT Full
freq_data = fftshift(fft(audio_data .*
hanning(num_samples)));  % Apply Hanning
window and FFT
plot(((-num_samples / 2 : (num_samples / 2) -
1) / num_samples) * sampling_rate,
abs(freq_data));  % Plot the frequency
spectrum
title('Frequency Spectrum of Audio Data');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
grid on;
```

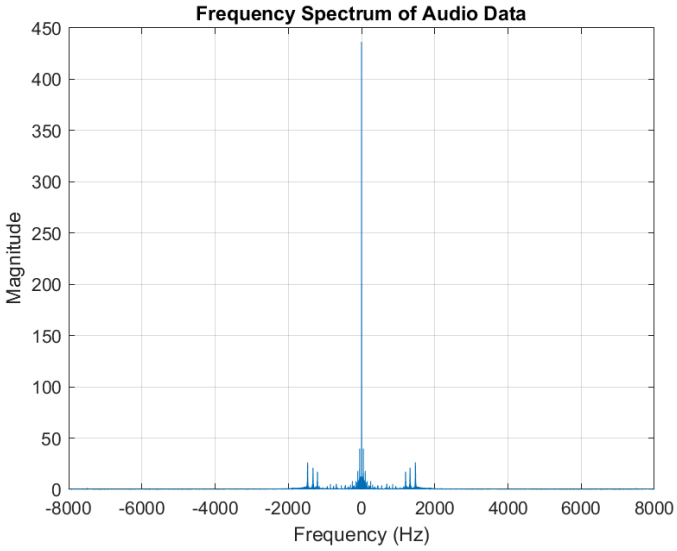

FIGURE 4.2. FREQUENCY SPECTRUM PLOT

This portion of the MATLAB code calculates and visualizes the full frequency spectrum of the audio signal. In line with this , the Fast Fourier Transform (FFT) is applied to the signal after multiplying it by a Hanning window (*hanning(num_samples)*) to reduce spectral leakage. Then, the *fftshift* function rearranges the FFT output so that the zero frequency is centered. The frequency spectrum is plotted with frequencies on the x-axis and magnitude on the y-axis, featuring a title, axis labels, and a grid for better readability. Moreover, Figure 4.2 reveals the distribution of magnitudes across the signal's frequency components, with the prominent peak at 0 Hz likely caused by static at the start of the recording, while smaller peaks correspond to the various frequencies present in the signal

```
% Pad audio_data to be divisible by 4096
padded_audio = [audio_data' zeros(1,
(ceil(num_samples / 4096) * 4096) -
num_samples)];  % Pad the audio data
```

The *audio_data* signal is padded with zeros to make its length divisible by 4096. The value 4096 was chosen because it is a power of 2, which allows for more efficient computation of the Fast Fourier Transform (FFT), optimizing processing time and performance. Using the *ceil* function, the smallest integer greater than or equal to the ratio of *num_samples* to 4096 is calculated, ensuring the total length is a multiple of 4096. The result is stored in the *padded_audio* variable, where the original audio data remains intact, and zeros are added to meet the required size. This padding is often necessary for processing the signal in fixed-size chunks.

```
% Generate frames with window function
frame_length = 512; overlap_length = 256;  %
Set frame length and overlap length
frame_shift_length = frame_length -
overlap_length;  % Calculate frame shift
length
frame_count = floor((length(padded_audio) -
```

```
frame_length) / frame_shift_length) + 1;  % Calculate number of frames
audio_frames = zeros(frame_length, frame_count);  % Initialize the matrix for frames

for i = 1 : frame_count  % Loop through each frame
   audio_frames(:, i) = padded_audio((i - 1) * frame_shift_length + 1 : (i - 1) * frame_shift_length + frame_length);  % Extract each frame
end

audio_frames = audio_frames .* repmat(hanning(frame_length), 1, frame_count);  % Apply Hanning window to each frame
```

In this section, the audio signal is divided into overlapping frames for further analysis. The frame length is set to 512 samples, with a 256-sample overlap between consecutive frames. The total number of frames is calculated based on the signal length and frame shift length. The *audio_frames* matrix is initialized to store each frame, and a loop extracts each frame from the padded audio signal. A Hanning window is applied to each frame to reduce spectral leakage, with the windowing function being multiplied by each frame using the *repmat* function.

```
% Perform FFT
fft_result = fft(audio_frames);  % Apply FFT to each frame
fft_magnitude = abs(fft_result);  % Get the magnitude of the FFT result
```

In this portion of the code, the *fft* function is applied to each audio frame, converting the signal from the time domain to the frequency domain. Furthermore, the *abs* function is used to calculate the magnitude of the FFT result by taking the absolute value of the complex FFT output, which allows for the analysis of the strength of each frequency component present in the audio signal. With that in mind, this transformation enables a clearer understanding of the frequency distribution across the frames.

```
% Detection Algorithm
frequency_matrix = zeros(2, size(fft_magnitude, 2));  % Initialize frequency matrix

for k = 1 : size(fft_magnitude, 2)  % Loop through each frame
   [~, low_peaks] = findpeaks(fft_magnitude(10 : 1000 / (sampling_rate / frame_length), k));  % Find low frequency peaks
   [~, high_peaks] = findpeaks(fft_magnitude(1000 / (sampling_rate / frame_length) : frame_length / 2, k));  % Find high frequency peaks
   [~, sorted_low] = sort(fft_magnitude(10 : 1000 / (sampling_rate / frame_length), k), 'descend');  % Sort low peaks
   [~, sorted_high] = sort(fft_magnitude(1000 / (sampling_rate / frame_length) : frame_length / 2, k), 'descend');  % Sort high peaks

  peak_indices = [sorted_low(1) + 9, sorted_high(1) + (1000 / (sampling_rate / frame_length)) - 1];  % Identify peak indices
  peak_indices(fft_magnitude(peak_indices(1), k) < 0.1) = 0;  % Set low peak indices below threshold to 0
  peak_indices(fft_magnitude(peak_indices(2), k) < 0.1) = 0;  % Set high peak indices below threshold to 0

   peak_frequencies = peak_indices .* (sampling_rate / frame_length);  % Convert indices to frequencies
   [low_freq, high_freq] = classify_tones(peak_frequencies(1), peak_frequencies(2));  % Classify the tone frequencies
   frequency_matrix(:, k) = [low_freq; high_freq];  % Store the classified frequencies in the matrix
end
```

This section of the code implements a detection algorithm for identifying the frequencies in each frame of the audio signal. It initializes a frequency matrix to store the detected frequencies for each frame. Within the loop, the *findpeaks* function is used to identify low and high-frequency peaks in the magnitude of the FFT result, with the results being sorted in descending order. The peak indices are then extracted, and any peaks below a threshold are discarded. These indices are converted to actual frequency values based on the sampling rate and frame length. The *classify_tones* function, which is a user-made function, is then used to classify the detected low and high frequencies into their corresponding tones. The classified frequencies are stored in the frequency matrix for further analysis, helping to identify the tones associated with the pressed buttons in the touch-tone signal.

```
function [low_freq, high_freq] = classify_tones(low_tone, high_tone)
   % CLASSIFY_TONES Classifies tones into standard DTMF frequencies.
   if low_tone == 0 || high_tone == 0
       low_freq = 0; high_freq = 0; return;
   end

   % Low frequency classification
   dtmf_rows = [697, 770, 852, 941];
   [~, row_idx] = min(abs(dtmf_rows - low_tone));
   low_freq = dtmf_rows(row_idx);

   % High frequency classification
   dtmf_cols = [1209, 1336, 1477, 1633];
   [~, col_idx] = min(abs(dtmf_cols - high_tone));
   high_freq = dtmf_cols(col_idx);
end
```

The *classify_tones* function classifies the detected low and high frequencies into standard Dual-Tone Multi-Frequency

(DTMF) tones. It first checks if either frequency is zero, in which case it returns zeros. Then, the function classifies the low frequency by finding the closest match from the predefined DTMF row frequencies (697, 770, 852, 941 Hz) and assigns the corresponding value. Similarly, it classifies the high frequency by finding the closest match from the DTMF column frequencies (1209, 1336, 1477, 1633 Hz). The function returns the classified low and high frequencies, which are used to identify the pressed button in the DTMF signal.

```
% Sequence Grouping
grouped_frequencies =
group_frequency_ranges(frequency_matrix(1, :),
frequency_matrix(2, :));  % Group frequencies
by sequence
```

In this section, the *group_frequency_ranges* function (a user-made function) is used to group the detected frequencies into sequences based on the input frequency matrix. The function takes the low and high frequencies from the frequency matrix and organizes them into groups corresponding to the sequence of button presses. This grouping helps in identifying the series of DTMF tones, which can then be used to determine the sequence of buttons pressed on the touch-tone pad.

```
function [freq_grouped] =
group_frequency_ranges(low_freq, high_freq)
   % GROUP_FREQUENCY_RANGES Groups consecutive
frequency ranges.
   non_zero_indices = find(low_freq); % Find
non-zero low frequencies
   gap_indices = find(diff(non_zero_indices) >
1); % Identify gaps between indices

   group_start = [1, gap_indices + 1]; % Start
of each group
   group_end = [gap_indices,
numel(non_zero_indices)]; % End of each group

   freq_grouped = cell(numel(group_start), 1);

   for i = 1:numel(group_start)
       low_freq_group =
low_freq(non_zero_indices(group_start(i)):non_
zero_indices(group_end(i)));
       high_freq_group =
high_freq(non_zero_indices(group_start(i)):non
_zero_indices(group_end(i)));
       freq_grouped{i} = [low_freq_group;
high_freq_group];

   end

end
```

The *group_frequency_ranges* function groups consecutive frequency ranges based on the detected low and high frequencies. It first identifies the non-zero low frequencies and finds any gaps between consecutive frequencies using the *diff* function. The start and end indices for each group are determined by the positions of these gaps. The function then creates a cell array to store each group, where each group consists of the low and high frequencies that fall within consecutive ranges. This grouped data helps in identifying the sequence of button presses corresponding to the detected tones.

```
% Coded Cases
% Define the DTMF key codes
key_1 = [697; 1209]; key_2 = [697; 1336];
key_3 = [697; 1477]; key_A = [697; 1633];
key_4 = [770; 1209]; key_5 = [770; 1336];
key_6 = [770; 1477]; key_B = [770; 1633];
key_7 = [852; 1209]; key_8 = [852; 1336];
key_9 = [852; 1477]; key_C = [852; 1633];
key_Asterisk = [941; 1209]; key_0 = [941;
1336]; key_Hash = [941; 1477]; key_D = [941;
1633];

key_codes = {key_1, key_2, key_3, key_A,
key_4, key_5, key_6, key_B, key_7, key_8,
key_9, key_C, key_Asterisk, key_0, key_Hash,
key_D};  % Store all key codes

key_labels = '123A456B789C*0#D';  % Key labels
corresponding to the codes
```

This section defines the DTMF key codes, each consisting of a pair of frequencies representing the low and high frequency components of a specific button on the touch-tone pad. The key codes are stored in a cell array, where each cell corresponds to a particular button, such as key 1, key 2, or key A, and so on. The key labels are also defined as a string of characters ('123A456B789C*0#D') to represent the buttons on the keypad. This setup is used to map the detected frequencies to the corresponding buttons pressed on the DTMF pad.

```
% Remove Unwanted Inputs
dial_sequence = char(zeros(1,
length(grouped_frequencies)));  % Initialize
the dial sequence

for q = 1 : length(grouped_frequencies)  %
Loop through each grouped frequency
   group_matrix =
cell2mat(grouped_frequencies(q));  % Convert
group to matrix
   low_group = group_matrix(1, :); high_group
= group_matrix(2, :);  % Separate low and high
frequency groups
   if length(group_matrix) > 10  % If there
are more than 10 elements, eliminate outliers
   group_matrix = [low_group(low_group ~=
mode(low_group)); high_group(low_group ~=
mode(low_group))];
   end
   detected_key = [mode(group_matrix(1, :));
mode(group_matrix(2, :))];  % Find the most
frequent low and high frequency



   % Find matching key
   for i = 1:length(key_codes)  % Loop through
all key codes
       if isequal(detected_key, key_codes{i})
% Check if the detected key matches a key code
           dial_sequence(1, q) =
```

```
key_labels(i);  % Store the corresponding key
label
            break;
        end
    end
end
```

This section of the code processes the grouped frequencies to identify and remove unwanted inputs, ensuring a clean dial sequence. It initializes a dial sequence and loops through each group of frequencies. For each group, the frequencies are separated into low and high frequency components, and outliers are removed if a group contains more than 10 elements by eliminating values that differ from the most frequent (mode) low frequency, using the *mode* function. The most common low and high frequencies are then identified, and the code compares the detected frequencies with predefined DTMF key codes using the *isequal* function. If a match is found, the corresponding key label is stored in the dial sequence. This process helps refine the detection and eliminate erroneous inputs, resulting in a sequence of valid button presses.

```
% Spectrogram
figure;
imagesc(0 : frame_shift_length / sampling_rate
: (length(audio_data) - frame_length) /
sampling_rate, ...
        0 : sampling_rate / (size(fft_result,
1) - 1) : sampling_rate / 2, abs(fft_result(1
: size(fft_result, 1) / 2, :)));  % Display
the spectrogram
axis xy;  % Adjust axis orientation
xlabel('Time (s)');
ylabel('Frequency (Hz)');
title('Spectrogram');
colorbar;
```

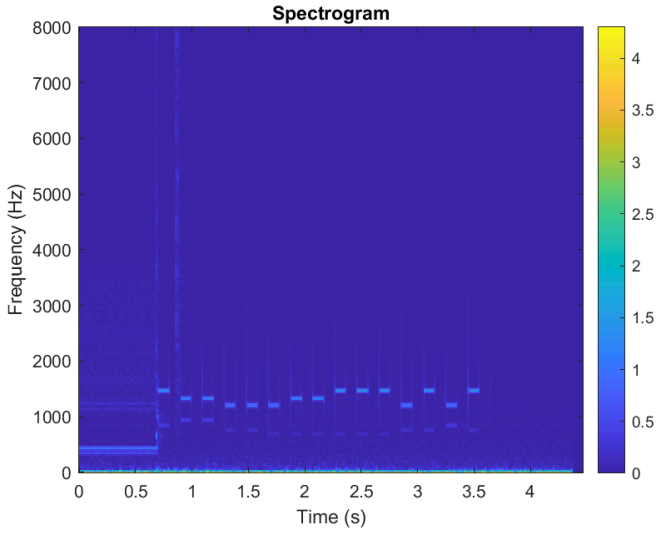

FIGURE 4.3. SPECTROGRAM OF THE AUDIO SIGNAL

This section generates a spectrogram of the audio signal using the *imagesc* function, where time is on the x-axis, frequency on the y-axis, and color intensity represents the magnitude of frequency components. The *axis xy* command ensures proper axis orientation, and a color bar is added to indicate the magnitude scale. This provides a visual representation of how the signal's frequency content varies over time. The spectrogram plot shows how the frequency components of the audio signal change over time. It highlights the prominent frequencies corresponding to the DTMF tones, with vertical bands indicating periods of key presses and transitions between different tones.

```
% Display Detected Sequence
disp('Detected Button Sequence:');  % Display
the detected dial sequence
disp(dial_sequence);  % Print the dial
sequence
```

This section displays the detected button sequence by printing the dial sequence to the console. The *disp* function is used to output the text "Detected Button Sequence" followed by the actual sequence of button presses identified from the audio signal.

```
Detected Button Sequence:
900441223334676
```

The output shows the detected button sequence "900441223334676," which represents the series of button presses identified from the audio signal. Each digit corresponds to a specific key pressed on the DTMF keypad, and this sequence reflects the result of analyzing the audio data using the detection algorithm.

ANALYSIS AND DISCUSSION OF RESULTS

The analysis of the DFT indices for the frequencies associated with button "9" (852 Hz and 1477 Hz) confirms their alignment with the expected values. Specifically, the calculated indices, 27 for 852 Hz and 47 for 1477 Hz, match the theoretical expectations based on a sampling frequency of 8000 Hz and a DFT size of 256. Furthermore, the plotted magnitude spectrum validates this result, as clear peaks are observed at sample indices 27 and 47, corresponding to 852 Hz and 1477 Hz. These peaks, which are highlighted in the graph, further reinforce the accuracy of both the manual calculations and MATLAB results.

In addition, the consistency between the manual calculation, MATLAB computation, and the plotted graph underscores the reliability of the frequency resolution in identifying DTMF tones. Although slight rounding of fractional values in the manual calculation (27.264 and 47.264) occurs, it does not affect the detection process, as the peaks in the graph are well-aligned with the expected frequencies.

As for Part B, the MATLAB code effectively processes the touch-tone audio signal to detect the button sequence based on the frequencies associated with the Dual-Tone Multi-Frequency (DTMF) signaling system. Initially, the Fast Fourier Transform (FFT) is applied to the audio signal,

followed by the detection of prominent frequency peaks. This allows the frequencies to be mapped to their corresponding DTMF button codes. The user-created functions *classify_tones* and *group_frequency_ranges* play a key role in accurately identifying DTMF signals. The function c*lassify_tones* maps detected frequency peaks to corresponding DTMF keypad digits, ensuring precise interpretation of the signal, while *group_frequency_ranges* organizes frequencies into valid tone pairs, filtering out noise or interference. Together, these functions enhance the robustness of the detection process and ensure reliable mapping of tones to button presses.

Additionally, the frequency spectrum and spectrogram provide valuable visual insight into how the signal's frequency components evolve over time, making it easier to identify the distinct tones corresponding to different button presses. The method of grouping the frequencies into sequences ensures accurate recognition of successive button presses, while the removal of outliers eliminates noise or erroneous detections, resulting in a cleaner and more reliable sequence.

To further elaborate, the generated spectrogram plot reveals how the frequency content of the audio signal changes over time, showing distinct patterns where the peaks in color intensity correspond to the prominent frequencies at specific moments. Moreover, the presence of vertical bands in the spectrogram indicates consistent frequency components, which are characteristic of the DTMF tones. These bands represent the time periods when certain keys were pressed, with their frequencies clearly visible in the plot. The transitions between different tones are also observable, as it reflects the changing frequencies as each button is pressed in the sequence.

Consequently, the detected button sequence, "900441223334676," corresponds to a series of button presses on a standard touch-tone keypad. This output highlights the system's effectiveness in recognizing the correct DTMF tones and accurately mapping them to the appropriate digits. By using a combination of FFT analysis, peak detection, and classification, the algorithm achieves a high level of accuracy in identifying the intended button sequence. Furthermore, the use of a Hanning window and zero-padding contributes to the accuracy by minimizing spectral leakage and ensuring efficient FFT computation. However, it is important to note that the accuracy of the sequence depends on several factors, including the quality of the input signal and the detection thresholds for the peaks. Finally, as a way to validate the results, online DTMF generators were employed as a supplementary verification method. The findings consistently produced the same output sequence corresponding to the buttons pressed, thereby confirming the accuracy of the analysis.

Application 5: Deconvolution in Image Processing

Implementation

In digital image processing, deblurring is a crucial technique used to recover a sharp image from a blurred one. Blurring in images can result from various factors, such as motion, defocus, or optical imperfections, which degrade image quality. One of the most common approaches for image restoration is **deconvolution**, which aims to reverse the effects of blurring by using knowledge of the system's point spread function (PSF)[6]. This application focuses on the deblurring of a stars image, where the blurred image and its corresponding PSF are provided as input. The **Lucy-Richardson algorithm**, a popular iterative deconvolution method, is employed to recover the original image by using the PSF and the blurred image[7]. To mitigate the effects of noise and distortions, especially near the image boundaries, **windowing techniques**, specifically the **Tukey window**, are applied. Windowing helps to reduce edge artifacts and enhance the overall deblurring performance. The effectiveness of different windowing strategies in controlling noise and distortions will be explored in this study, providing insight into their impact on the quality of the restored image. [8]

To deblur the given blurred image of stars, we implemented the Lucy-Richardson deconvolution algorithm in MATLAB. The impulse response of the imaging system, provided as a Point Spread Function (PSF) image, was used for this process. The deblurring was performed with noise and distortion control achieved through the application of a Tukey window.

```
% Load the blurred image and PSF
blurredImage = imread('stars-blurred.png');
psfImage = imread('stars-psf.png');

% Convert images to grayscale and normalize to
[0, 1]
blurredImage = mat2gray(blurredImage);
psfImage = mat2gray(psfImage);

% Normalize the PSF to ensure it sums to 1
psfNormalized = psfImage / sum(psfImage(:));

% Get the size of the blurred image
[rows, cols] = size(blurredImage);
```

In the **Loading and Pre-Processing** section, the code loads the blurred image *(stars-blurred.png)* and the Point Spread Function (PSF) image *(stars-psf.png)* using the *imread* function. After loading, both images are converted to grayscale and normalized to the [0, 1] range using *mat2gray*. This step ensures that the pixel values are within a standard range, which is important for most image processing tasks. The PSF is then normalized by dividing it by the sum of its pixel values *(psfNormalized = psfImage / sum(psfImage(:)))*. This ensures that the PSF has a total sum of 1, which is important for preventing the deconvolution process from exaggerating or diminishing the blur effect.

```
% Apply different windowing methods
tukeyRow = tukeywin(rows, 0.3); % Tukey window
for rows
tukeyCol = tukeywin(cols, 0.3); % Tukey window
for columns
tukeyWindow = tukeyRow * tukeyCol'; % Outer
product to form 2D Tukey window
blurredTukey = blurredImage .* tukeyWindow;
```

In the **Windowing** section, the code applies a **Tukey window** to the blurred image to reduce edge artifacts. A Tukey window is created for the rows of the image using *tukeywin(rows, 0.3)* and for the columns using *tukeywin(cols, 0.3)*. The 0.3 parameter specifies the tapering factor, which controls the amount of tapering at the edges. The window for rows (tukeyRow) and columns (tukeyCol) are then combined using the outer product *(tukeyWindow = tukeyRow * tukeyCol')*, creating a 2D Tukey window. This window is applied to the blurred image through element-wise multiplication *(blurredTukey = blurredImage .* tukeyWindow)*. This process reduces edge artifacts that can arise during transformations like the Fourier transform and helps improve the stability and quality of the deconvolution process.[8]

```
% Perform deconvolution using Lucy-Richardson
algorithm
numIterations = 30; % Adjust iterations based
on image quality
restoredTukey = deconvlucy(blurredTukey,
psfNormalized, numIterations);
```

In the **Deconvolution** section, the code performs the actual **Lucy-Richardson deconvolution** to restore the blurred image. The *deconvlucy* function is used, which iteratively refines the estimate of the original image. The number of iterations is set to 30 *(numIterations = 30)*, which can be adjusted based on the desired level of restoration. The input to the function is the windowed blurred image *(blurredTukey)* and the normalized PSF *(psfNormalized)*. The algorithm works by using the PSF to model the blur and gradually improves the image estimate with each iteration, restoring fine details and reducing the effects of blur. [7]

```
% Display results
figure;
subplot(1, 2, 1);
imshow(blurredTukey, []);
title('Tukey Windowed Blurred Image');


subplot(1, 2, 2);
imshow(restoredTukey, []);
title('Restored Image (Tukey Window)');
```

In the **Visualization** section, the code displays the results of the image processing. The first subplot shows the blurred image after the Tukey window has been applied *(blurredTukey)*, giving a visual representation of the image with reduced edge artifacts. The second subplot displays the restored image *(restoredTukey)*, which shows the outcome of the Lucy-Richardson deconvolution. The *imshow* function is used to display both images, and the *[]* option ensures that the images are scaled appropriately for visualization. This side-by-side comparison allows for an easy evaluation of the effectiveness of the deconvolution process.

```
% Frequency domain plot for the original image
F_original = fftshift(fft2(blurredImage)); %
2D FFT and shift zero frequency to center
mag_original = abs(F_original); % Magnitude of
the Fourier transform


% Frequency domain plot for the image with
Tukey window
F_tukey = fftshift(fft2(blurredTukey)); % 2D
FFT of the Tukey windowed image
mag_tukey = abs(F_tukey); % Magnitude of the
Fourier transform


% Frequency domain plot for the restored image
after deconvolution
F_restored = fftshift(fft2(restoredTukey)); %
2D FFT of the restored image
mag_restored = abs(F_restored); % Magnitude of
the Fourier transform


figure;
% Show the magnitude spectrum of the original
image
subplot(1, 3, 1);
imshow(log(1 + mag_original), []); % Use log
scale for better visualization
title('Magnitude Spectrum of Original Image');


% Show the magnitude spectrum of the image
with Tukey window
subplot(1, 3, 2);
imshow(log(1 + mag_tukey), []); % Use log
scale for better visualization
title('Magnitude Spectrum of Image with Tukey
Window');


% Show the magnitude spectrum of the restored
image after deconvolution
subplot(1, 3, 3);
imshow(log(1 + mag_restored), []); % Use log
scale for better visualization
title('Magnitude Spectrum of Restored Image
(Deconvolution)');
```

To visualize the deconvolution process in the frequency domain, we begin by calculating the 2D Fourier transform (FFT) of the restored image after deconvolution using the *fft2* function. The Fourier transform shifts the zero-frequency component to the center of the spectrum with *fftshift*, facilitating the analysis of frequency content. The magnitude of the FFT is computed using *abs()*, and to enhance visibility, a logarithmic scale *(log(1 + magnitude))* is applied to compress the dynamic range and emphasize low-frequency components. This process is similarly applied to the original blurred image and the Tukey windowed image. By comparing

the magnitude spectra of these images in a side-by-side display, we can clearly visualize the frequency differences and the effects of windowing and deconvolution on the image's frequency characteristics.

Analysis and Discussion of Results

The group is provided with two PNG files named *stars-blurred.png* and *stars-psf.png*, where *blurred.png* represents the blurred image of stars and *stars-psf.png* is the corresponding point spread function (PSF) that models the blur. In this analysis, we aim to recover the sharp image from the blurred version by applying deconvolution techniques, with the Lucy-Richardson algorithm as our primary method.

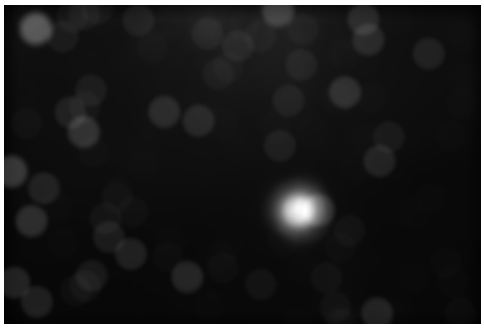

Figure 5.1: stars-blurred.png

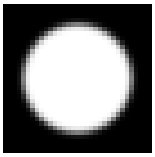

Figure 5.2: stars-psf.png

Building upon the initial analysis, we now focus on the differences observed in the frequency domain between the original image, the application of the Tukey window, and the deconvolution results. The magnitude spectrum provides an insightful representation of the frequency content of each image, revealing how each processing step impacts the image's clarity and sharpness.

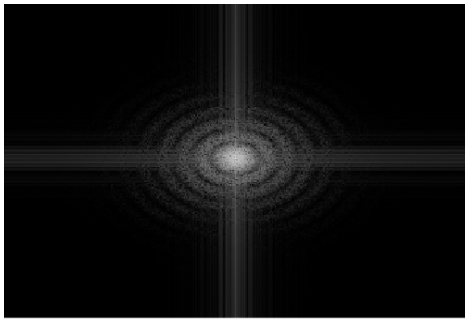

Figure 5.3: Magnitude Spectrum of the Original Image

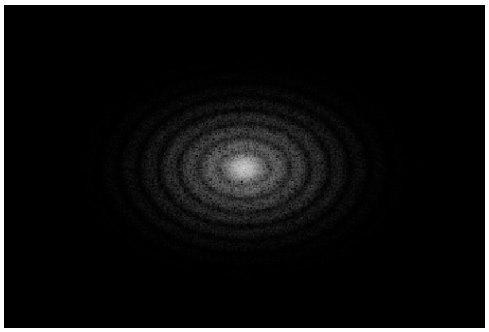

Figure 5.4: Magnitude Spectrum of Image with Tukey Windowing

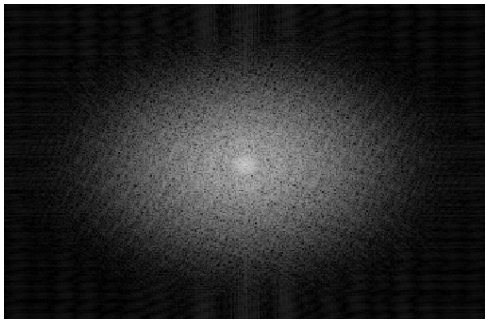

Figure 5.5: Magnitude Spectrum of Restored Image(Deconvolution)

In *Figure 5.3*, The magnitude spectrum of the original blurred image reveals a central peak surrounded by a cross-shaped pattern. This characteristic pattern is indicative of a point source of light that has undergone a blurring process. The central peak represents the low-frequency components of the image, while the cross-shaped pattern corresponds to the high-frequency components.

In *Figure 5.4*, the magnitude spectrum of the blurred image with the Tukey window applied shows a similar pattern to the original image, but with some key differences. The central peak is more pronounced, and the cross-shaped pattern is less intense, indicating that the Tukey window has reduced the high-frequency components. This is beneficial for deconvolution, as it helps to minimize noise and artifacts. The Tukey window attenuates high-frequency content, making it a useful tool for improving the deconvolution process by reducing unwanted noise.

Furthermore, in *Figure 5.5*, The magnitude spectrum of the restored image showcases a more concentrated central peak with significantly reduced surrounding noise compared to the original blurred image and the Tukey-windowed image. This indicates that the deconvolution process, specifically the Lucy-Richardson algorithm, has successfully recovered a substantial portion of the high-frequency information lost during the blurring process. The sharp central peak signifies a well-defined point source of light, suggesting that the deconvolution has effectively mitigated the blurring effect.

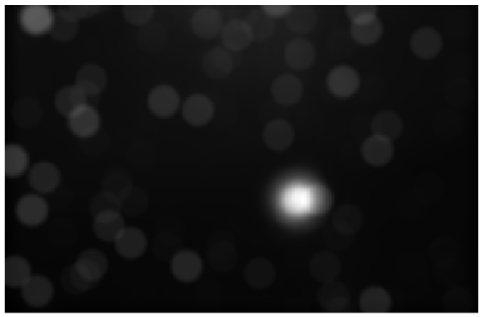

FIGURE 5.6: ORIGINAL BLURRED IMAGE

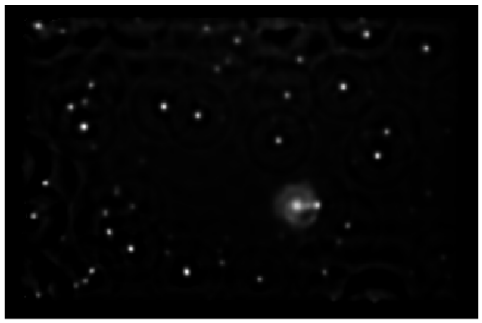

Figure 5.7: Restored Image (After Deconvolution)

The comparison between Figure 5.6: Original Blurred Image and Figure 5.7: Restored Image highlights the effectiveness of the deblurring process. The original blurred image shows a lack of clarity, with diffused light sources and diminished sharpness, making it difficult to discern finer details. This blurriness is a result of convolution, where the point sources of light are spread out due to the imaging system's limitations, as characterized by the Point Spread Function (PSF).

In contrast, the restored image demonstrates significant improvements in sharpness and detail. The application of the Tukey window before deconvolution reduced edge noise and artifacts, while the Lucy-Richardson algorithm, leveraging the PSF, successfully recovered much of the high-frequency information lost during the blurring process. This restoration not only enhances the visibility of individual light sources but also brings out finer structures that were obscured in the blurred image.

Through this comparison, the effectiveness of combining PSF-based deconvolution, windowing techniques, and iterative algorithms is clearly demonstrated in producing a more accurate and visually appealing image.

III. CONCLUSION AND RECOMMENDATION

This paper explored the applications of Digital Signal Processing (DSP) in five key areas: system identification, filter design, digital oscillator implementation, telephony, and image deconvolution. Each application demonstrated the power and versatility of DSP in solving engineering problems and highlighted the importance of MATLAB as a tool for simulation and analysis. The methodologies employed ranged from signal analysis and mathematical modeling to practical implementations, offering a comprehensive approach to understanding DSP principles.

In the system identification experiment, input signals were used to characterize a transmission line's behavior, showcasing the impact of signal type and parameters such as pulse width on frequency responses. The filter design application utilized pole-zero placement to approximate a desired magnitude response, emphasizing the critical role of stability and precision in achieving optimal filtering. For digital oscillator design, parallel components were implemented to synthesize multi-frequency signals accurately, with the results closely aligning with theoretical expectations.

The telephony application demonstrated the use of the Discrete Fourier Transform (DFT) to analyze and decode DTMF tones, providing a reliable method for identifying keypad inputs in telecommunication systems. Finally, the image deconvolution task employed the Lucy-Richardson algorithm and windowing techniques to restore a blurred image, illustrating the potential of DSP in image processing and enhancement.

Overall, the results validated theoretical concepts and showcased their practical relevance in solving real-world challenges. These applications highlight the critical role of DSP in modern engineering fields such as communications, audio processing, and imaging. Future work could explore advanced DSP techniques, such as adaptive filtering and machine learning integration, to further enhance system performance and expand the scope of DSP applications.

IV. References

[1] Keesman, K. J. (2011). *System Identification: An Introduction*. Springer Publishing.

[2] *System Identification Overview*. www.mathworks.com. Retrieved December 15, 2024, https://www.mathworks.com/help/ident/gs/about-system-identification.html

[3] Vinay K. Ingle and John G. Proakis. *Digital Signal Processing using MATLAB,* 3rd Edition. Cengage Learning, 2012.

[4] Proakis, J. G., & Manolakis, D. G. (2007). *Digital Signal Processing: Principles, Algorithms, and Applications*. Pearson Prentice Hall.

[5] John G. Proakis and Dimitris G. Manolakis. Digital Signal Processing: Principles, Algorithms, and Application. Prentice-Hall International, 2016.

[6] V. Krishnamurthi, Y.-H. Liu, T. J. Holmes, B. Roysam, and J. N. Turner, "Blind deconvolution of 2-D and 3-D fluorescent micrographs," Proceedings of SPIE, the International Society for Optical Engineering/Proceedings of SPIE, vol. 1660, pp. 95–102, Jun. 1992, doi: 10.1117/12.59544.

[7] D. A. Fish, J. G. Walker, A. M. Brinicombe, and E. R. Pike, "Blind deconvolution by means of the Richardson–Lucy algorithm," Journal of the Optical Society of America A, vol. 12, no. 1, p. 58, Jan. 1995, doi: 10.1364/josaa.12.000058.

[8] A. Bovik, "Streaking in median filtered images," IEEE Transactions on Acoustics Speech and Signal Processing, vol. 35, no. 4, pp. 493–503, Apr. 1987, doi: 10.1109/tassp.1987.1165153.

V. Distribution of Tasks

Group Members	Tasks
Capañarihan, Alein Miguel P.	➢ Application 1 ➢ Conclusion
Deang, Khatelyn Jhuneryll E.	➢ Application 2 ➢ Application 3 ➢ Application 4
Hugo, Jude Nazi D.	➢ Application 5 ➢ Introduction

VI. Appendix

Application 1: Transmission Lines

```
max = 5e9; % maximum frequency of the
signal
fs = 2*fmax; % sampling frequency
T = 1/fs; % % period
L = 20001; % total number of samples
t = (0:L-1)*T; % converting samples to
time domain
deltaf = fs/(L-1);
f = -fs/2:deltaf:fs/2;
sequence_length = 20001; %Sequence length
pulse_duration = [10 100]; %pulse
durations
%Input sequence with zero-padding
[x_signal,x_signal2] =
deal(zeros(1,sequence_length));
%sequence_type = input('Input the
sequence type:','s'); %Request for input
sequence_type = 'Impulse';
switch sequence_type
   case 'Impulse'
       x_signal(1) = 1; %Impulse
       x_signal2(1) = 1;
   case 'Rectangular'
       x_signal(1:pulse_duration(1)) = 1;
%Rectangular
```

```
        x_signal2(1:pulse_duration(2)) =
1;
    case 'Hamming'
        x_signal(1:pulse_duration(1)) =
hamming(pulse_duration(1)); %Hamming
        x_signal2(1:pulse_duration(2)) =
hamming(pulse_duration(2));f
    case 'Hanning'
        x_signal(1:pulse_duration(1)) =
hann(pulse_duration(1)); %Hanning
        x_signal2(1:pulse_duration(2)) =
hann(pulse_duration(2));
    case 'Kaiser'
        x_signal(1:pulse_duration(1)) =
kaiser(pulse_duration(1),5); %Kaiser
        x_signal2(1:pulse_duration(2)) =
kaiser(pulse_duration(2),5);
    case 'Blackmann-Harris'
        x_signal(1:pulse_duration(1)) =
blackmanharris(pulse_duration(1)); %BH
        x_signal2(1:pulse_duration(2)) =
blackmanharris(pulse_duration(2));
end
input_sequence1 = x_signal; %Pulse width
= 10
input_sequence2 = x_signal2; %Pulse width
= 100
% Input signal passes through the system
output_sequence1 =
transline(input_sequence1);
output_sequence2 =
transline(input_sequence2);
%Sequences with Pulse Width of 10
in_fft1 = fft(input_sequence1);
out_fft1 = fft(output_sequence1);
mag_response1 = abs(out_fft1) ./
(abs(in_fft1));
%Sequences with Pulse Width of 100
in_fft2 = fft(input_sequence2);
out_fft2 = fft(output_sequence2);
mag_response2 = abs(out_fft2) ./
(abs(in_fft2));
%Input Sequence
figure(1)
s = tiledlayout(2,2);
title(s,sprintf('Input %s
Sequence',sequence_type))
nexttile
plot(t,input_sequence1) %Time Domain
title(sprintf('Time Domain, Pulse Width =
%d',pulse_duration(1)))
xlabel('Time')
ylabel('Amplitude')
nexttile
plot(t,input_sequence2) %Time Domain
title(sprintf('Time Domain, Pulse Width =
%d',pulse_duration(2)))
xlabel('Time')
ylabel('Amplitude')
nexttile
plot(f,abs(in_fft1))
title(sprintf('Frequency Domain, Pulse
Width = %d',pulse_duration(1)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
nexttile
plot(f,abs(in_fft2))
title(sprintf('Frequency Domain, Pulse
Width = %d',pulse_duration(2)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
%Output Sequence
figure(2)
p = tiledlayout(2,2);
title(p,sprintf('Output %s
Sequence',sequence_type))
nexttile
plot(t,output_sequence1) %Time Domain
title(sprintf('Time Domain, Pulse Width =
%d',pulse_duration(1)))
xlabel('Time')
ylabel('Amplitude')
nexttile
plot(t,output_sequence2) %Time Domain
title(sprintf('Time Domain, Pulse Width =
%d',pulse_duration(2)))
xlabel('Time')
ylabel('Amplitude')
nexttile
plot(f,abs(out_fft1))
title(sprintf('Frequency Domain, Pulse
Width = %d',pulse_duration(1)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
nexttile
plot(f,abs(out_fft2))
title(sprintf('Frequency Domain, Pulse
Width = %d',pulse_duration(2)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
figure(3)
d = tiledlayout('vertical');
title(d,sprintf('Transmission Line
Magnitude Response - %s',sequence_type))
nexttile
plot(f,mag_response1);
title(sprintf('Pulse Width - %d'
,pulse_duration(1)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
nexttile
plot(f,mag_response2);
title(sprintf('Pulse Width - %d'
,pulse_duration(2)))
xlabel('Frequency Bins (Hz)')
ylabel('Magnitude')
```

Application 2: Filter Design

```
% Approximated zeros (z) and poles (p)
p = [0.3492 + 0.4875i; -0.3492 + 0.4875i;
0.4875 + 0.3492i; -0.4845 + 0.3492i; ...
    0.3492 - 0.4875i; -0.3492 - 0.4875i;
0.4875 - 0.3492i; -0.4845 - 0.3492i];
z = [0.8199 + 0.7900i; -0.8199 + 0.7900i;
0.7900 + 0.8199i; -0.7900 + 0.8199i; ...
    0.8199 - 0.7900i; -0.8199 - 0.7900i;
0.7900 - 0.8199i; -0.7900 - 0.8199i];

% Generate the numerator (zeros) and
denominator (poles) coefficients
num = poly(z); % Numerator polynomial
from the zeros
denom = poly(p); % Denominator polynomial
from the poles

% Visualize the filter response
fvtool(num, denom); % Use FVTool for
better visualization of frequency
response

% Define the input signal
n = 0:499; % 500 samples
u = (n >= 0); % Unit step function u(n)
using logical indexing
x = (cos(3*pi*n/20) + cos(pi*n/2) +
cos(3*pi*n/4)) .* u; % Input signal with
u(n)

% Apply the filter
y = filter(num, denom, x);

% Frequency-domain analysis
X = abs(fft(x, 1024)); % FFT of input
signal
Y = abs(fft(y, 1024)); % FFT of output
signal
w = (0:511) / 512; % Normalized frequency
axis for half-spectrum (512 points)

% Time-domain plot
figure;
subplot(2, 1, 1);
plot(n, x, 'b', 'DisplayName', 'Input
Signal');
hold on;
plot(n, y, 'r', 'DisplayName', 'Filtered
Signal');
xlabel('Samples (n)');
ylabel('Amplitude');
title('Time Domain Signal');
legend;
grid on;

% Frequency-domain plot
subplot(2, 1, 2);
plot(w, (X(1:512)), 'b', 'DisplayName',
'Input Spectrum');
hold on;
plot(w, (Y(1:512)), 'r', 'DisplayName',
'Filtered Spectrum');
xlabel('Normalized Frequency');
ylabel('Magnitude (dB)');
title('Frequency Domain Signal');
legend;
grid on;
```

Application 3: Digital Oscillator Design

```
% Sampling frequency
fs = 100e3;

% Angular frequencies
w1 = 3*pi/20;
w2 = pi/2;
w3 = 3*pi/4;

% Difference equation coefficients
b1 = [1, -cos(w1)]; % Numerator
b2 = [1, -cos(w2)]; % Numerator
b3 = [1, -cos(w3)]; % Numerator
a1 = [1, -2*cos(w1), 1]; % Denominator
for w1
a2 = [1, -2*cos(w2), 1]; % Denominator
for w2
a3 = [1, -2*cos(w3), 1]; % Denominator
for w3

% Input impulse signal
n = 0:249;
impulse = (n == 0);

% Filter responses
y1 = filter(b1, a1, impulse);
y2 = filter(b2, a2, impulse);
y3 = filter(b3, a3, impulse);

% Combine outputs
x_combined = y1 + y2 + y3;

% Define the expected output
x_expected = (cos(3*pi/20 * n) + cos(pi/2
* n) + cos(3*pi/4 * n)) .* (n >= 0);

% Plot y1
figure;
subplot(3, 1, 1);
plot(n, y1);
xlabel('n');
ylabel('y_1(n)');
title('Impulse Response for w1');
grid on;

% Plot y2
subplot(3, 1, 2);
plot(n, y2);
xlabel('n');
ylabel('y_2(n)');
title('Impulse Response for w2');
```

```
grid on;

% Plot y3
subplot(3, 1, 3);
plot(n, y3);
xlabel('n');
ylabel('y_3(n)');
title('Impulse Response for w3');
grid on;

% Plot combined output
figure;
subplot(3, 1, 1);
plot(n, x_combined);
xlabel('n');
ylabel('x_{combined}(n)');
title('Combined Output');
grid on;

% Plot expected output
subplot(3, 1, 2);
plot(n, x_expected);
xlabel('n');
ylabel('x_{expected}(n)');
title('Expected Output');
grid on;

% Compare combined and expected outputs
subplot(3, 1, 3);
plot(n, x_combined, 'b', n, x_expected,
'r--');
legend('Combined', 'Expected');
xlabel('n');
ylabel('x(n)');
title('Comparison of Outputs');
grid on;
```

Application 4: Discrete Fourier Transform in Telephony

Part A

```
% Parameters
fs = 8000;  % Sampling frequency in Hz
N = 256;    % DFT size (number of points)
frequencies = [852, 1477];  % Frequencies
for button 9 in Hz
n = 0:N-1;

% Calculate DFT indices for the given
frequencies
indices = round(frequencies / fs * N);

% Display the indices for the button 9
frequencies
disp('DFT Indices for button 9');
fprintf('Frequency 852 Hz: %d\n',
indices(1));
fprintf('Frequency 1477 Hz: %d\n',
indices(2));

% Generate the signal corresponding to
button 9 (two sine waves at 852 Hz and
1477 Hz)
signal = sin(2 * pi * frequencies(1) * n
/ fs) + sin(2 * pi * frequencies(2) * n /
fs);

% Apply a Hanning window to reduce
spectral leakage
windowed_signal = signal .* hanning(N)';

% Perform the FFT on the windowed signal
x_fft = fft(windowed_signal, N);

% Plot the magnitude spectrum of the DFT
figure;
plot(n, abs(x_fft));  % Plot the
magnitude of the DFT coefficients
xlabel("Sample Index");
ylabel("Magnitude");
title("Magnitude Spectrum of Button 9");
grid on;

% Highlight the peaks at 852 Hz and 1477
Hz in the frequency spectrum
hold on;
plot(indices(1),
abs(x_fft(indices(1)+1)), 'o',
'MarkerSize', 10, 'Linewidth', 1,
'Color', [1 0 1]);  % Peak at 852 Hz
plot(indices(2),
abs(x_fft(indices(2)+1)), 'o',
'MarkerSize', 10, 'Linewidth', 1,
'Color', [0 0 1]);  % Peak at 1477 Hz

% Add a legend to explain the plot
legend("Magnitude Spectrum", "Peak at 852
Hz", "Peak at 1477 Hz");
hold off;
```

Part B

```
% Loading the Given Audio File
[audio_data, sampling_rate] =
audioread("touchtone.wav");  % Read the
audio file
num_samples = length(audio_data);  % Get
the number of samples in the audio data

% FFT Full
freq_data = fftshift(fft(audio_data .*
hanning(num_samples)));  % Apply Hanning
window and FFT
plot(((-num_samples / 2 : (num_samples /
2) - 1) / num_samples) * sampling_rate,
abs(freq_data));  % Plot the frequency
spectrum
title('Frequency Spectrum of Audio
Data');
```

```
xlabel('Frequency (Hz)');
ylabel('Magnitude');
grid on;

% Pad audio_data to be divisible by 4096
padded_audio = [audio_data' zeros(1,
(ceil(num_samples / 4096) * 4096) -
num_samples)];  % Pad the audio data

% Generate frames with window function
frame_length = 512; overlap_length = 256;
% Set frame length and overlap length
frame_shift_length = frame_length -
overlap_length;  % Calculate frame shift
length
frame_count = floor((length(padded_audio)
- frame_length) / frame_shift_length) +
1;

% Calculate number of frames
audio_frames = zeros(frame_length,
frame_count);  % Initialize the matrix
for frames
for i = 1 : frame_count  % Loop through
each frame
   audio_frames(:, i) = padded_audio((i -
1) * frame_shift_length + 1 : (i - 1) *
frame_shift_length + frame_length);  %
Extract each frame
end
audio_frames = audio_frames .*
repmat(hanning(frame_length), 1,
frame_count);  % Apply Hanning window to
each frame

% Perform FFT
fft_result = fft(audio_frames);  % Apply
FFT to each frame
fft_magnitude = abs(fft_result);  % Get
the magnitude of the FFT result
% Detection Algorithm
frequency_matrix = zeros(2,
size(fft_magnitude, 2));  % Initialize
frequency matrix
for k = 1 : size(fft_magnitude, 2)  %
Loop through each frame
   [~, low_peaks] =
findpeaks(fft_magnitude(10 : 1000 /
(sampling_rate / frame_length), k));  %
Find low frequency peaks
   [~, high_peaks] =
findpeaks(fft_magnitude(1000 /
(sampling_rate / frame_length) :
frame_length / 2, k));  % Find high
frequency peaks
   [~, sorted_low] =
sort(fft_magnitude(10 : 1000 /
(sampling_rate / frame_length), k),
'descend');  % Sort low peaks
   [~, sorted_high] =
sort(fft_magnitude(1000 / (sampling_rate
/ frame_length) : frame_length / 2, k),
'descend');  % Sort high peaks

   peak_indices = [sorted_low(1) + 9,
sorted_high(1) + (1000 / (sampling_rate /
frame_length)) - 1];  % Identify peak
indices

peak_indices(fft_magnitude(peak_indices(1
), k) < 0.1) = 0;  % Set low peak indices
below threshold to 0

peak_indices(fft_magnitude(peak_indices(2
), k) < 0.1) = 0;  % Set high peak
indices below threshold to 0

   peak_frequencies = peak_indices .*
(sampling_rate / frame_length);  %
Convert indices to frequencies
   [low_freq, high_freq] =
classify_tones(peak_frequencies(1),
peak_frequencies(2));  % Classify the
tone frequencies
   frequency_matrix(:, k) = [low_freq;
high_freq];  % Store the classified
frequencies in the matrix
end

% Sequence Grouping
grouped_frequencies =
group_frequency_ranges(frequency_matrix(1
, :), frequency_matrix(2, :));  % Group
frequencies by sequence

% Coded Cases
% Define the DTMF key codes
key_1 = [697; 1209]; key_2 = [697; 1336];
key_3 = [697; 1477]; key_A = [697; 1633];
key_4 = [770; 1209]; key_5 = [770; 1336];
key_6 = [770; 1477]; key_B = [770; 1633];
key_7 = [852; 1209]; key_8 = [852; 1336];
key_9 = [852; 1477]; key_C = [852; 1633];
key_Asterisk = [941; 1209]; key_0 = [941;
1336]; key_Hash = [941; 1477]; key_D =
[941; 1633];
key_codes = {key_1, key_2, key_3, key_A,
key_4, key_5, key_6, key_B, key_7, key_8,
key_9, key_C, key_Asterisk, key_0,
key_Hash, key_D};  % Store all key codes
key_labels = '123A456B789C*0#D';  % Key
labels corresponding to the codes

% Remove Unwanted Inputs
dial_sequence = char(zeros(1,
length(grouped_frequencies)));  %
Initialize the dial sequence
for q = 1 : length(grouped_frequencies)

% Loop through each grouped frequency
```

```
    group_matrix =
cell2mat(grouped_frequencies(q));  %
Convert group to matrix
    low_group = group_matrix(1, :);
high_group = group_matrix(2, :);  %
Separate low and high frequency groups

    if length(group_matrix) > 10  % If
there are more than 10 elements,
eliminate outliers
        group_matrix =
[low_group(low_group ~= mode(low_group));
high_group(low_group ~=
mode(low_group))];
    end

    detected_key = [mode(group_matrix(1,
:)); mode(group_matrix(2, :))];  % Find
the most frequent low and high frequency

    % Find matching key
    for i = 1:length(key_codes)  % Loop
through all key codes
        if isequal(detected_key,
key_codes{i})  % Check if the detected
key matches a key code
            dial_sequence(1, q) =
key_labels(i);  % Store the corresponding
key label
            break;
        end
    end
end

% Spectrogram
figure;
imagesc(0 : frame_shift_length /
sampling_rate : (length(audio_data) -
frame_length) / sampling_rate, ...
        0 : sampling_rate /
(size(fft_result, 1) - 1) : sampling_rate
/ 2, abs(fft_result(1 : size(fft_result,
1) / 2, :)));  % Display the spectrogram
axis xy;  % Adjust axis orientation
xlabel('Time (s)');
ylabel('Frequency (Hz)');
title('Spectrogram');
colorbar;

% Display Detected Sequence
disp('Detected Button Sequence:');  %
Display the detected dial sequence
disp(dial_sequence);  % Print the dial
sequence
```

User-made Function Files

```
function [low_freq, high_freq] =
classify_tones(low_tone, high_tone)
    % CLASSIFY_TONES Classifies tones into
standard DTMF frequencies.
    %
    % Inputs:
    %   low_tone  - Measured lower
frequency value.
    %   high_tone - Measured higher
frequency value.
    %
    % Outputs:
    %   low_freq  - Classified lower
frequency from DTMF standards.
    %   high_freq - Classified higher
frequency from DTMF standards.
    %
    % Description:
    %   Maps the input `low_tone` and
`high_tone` values to the closest
standard
    %   DTMF (Dual-Tone Multi-Frequency)
frequencies used in telecommunication
systems.
    if low_tone == 0 || high_tone == 0
        low_freq = 0; high_freq = 0;
return;
    end
    % Low frequency classification
    dtmf_rows = [697, 770, 852, 941];
    [~, row_idx] = min(abs(dtmf_rows -
low_tone));
    low_freq = dtmf_rows(row_idx);
    % High frequency classification
    dtmf_cols = [1209, 1336, 1477, 1633];
    [~, col_idx] = min(abs(dtmf_cols -
high_tone));
    high_freq = dtmf_cols(col_idx);
end

function [freq_grouped] =
group_frequency_ranges(low_freq,
high_freq)
    % GROUP_FREQUENCY_RANGES Groups
consecutive frequency ranges.
    %
    % Inputs:
    %   low_freq  - Array of lower
frequency values (non-zero entries are
grouped).
    %   high_freq - Array of higher
frequency values corresponding to
low_freq.
    %
    % Outputs:
    %   freq_grouped - Cell array where
each cell contains a 2-row matrix:
    %                  [low_freq_group;
high_freq_group].
    %
    % Description:
    %   Groups consecutive non-zero values
in `low_freq` and their corresponding
```

```
    %   `high_freq` values into separate
arrays, based on gaps in indices.
    non_zero_indices = find(low_freq); %
Find non-zero low frequencies
    gap_indices =
find(diff(non_zero_indices) > 1); %
Identify gaps between indices

    group_start = [1, gap_indices + 1]; %
Start of each group
    group_end = [gap_indices,
numel(non_zero_indices)]; % End of each
group

    freq_grouped =
cell(numel(group_start), 1);

    for i = 1:numel(group_start)
        low_freq_group =
low_freq(non_zero_indices(group_start(i))
:non_zero_indices(group_end(i)));
        high_freq_group =
high_freq(non_zero_indices(group_start(i)
):non_zero_indices(group_end(i)));
        freq_grouped{i} = [low_freq_group;
high_freq_group];
    end
end
```

Application 5: Deconvolution in Image Processing

```
% Load the blurred image and PSF
blurredImage =
imread('stars-blurred.png');
psfImage = imread('stars-psf.png');

% Convert images to grayscale and
normalize to [0, 1]
blurredImage = mat2gray(blurredImage);
psfImage = mat2gray(psfImage);

% Normalize the PSF to ensure it sums to
1
psfNormalized = psfImage /
sum(psfImage(:));
% Get the size of the blurred image
[rows, cols] = size(blurredImage);

% Apply different windowing methods
tukeyRow = tukeywin(rows, 0.3); % Tukey
window for rows
tukeyCol = tukeywin(cols, 0.3); % Tukey
window for columns
tukeyWindow = tukeyRow * tukeyCol'; %
Outer product to form 2D Tukey window
blurredTukey = blurredImage .*
tukeyWindow;

% Perform deconvolution using
Lucy-Richardson algorithm
numIterations = 30; % Adjust iterations
based on image quality
restoredTukey = deconvlucy(blurredTukey,
psfNormalized, numIterations);

%% Display results
figure;
% Show the original blurred image (before
Tukey windowing)
subplot(1, 3, 1);
imshow(blurredImage, []);
title('Original Blurred Image');

% Show the blurred image after Tukey
windowing
subplot(1, 3, 2);
imshow(blurredTukey, []);
title('Blurred Image with Tukey Window');

% Show the restored image after
deconvolution (using Tukey windowed
image)
subplot(1, 3, 3);
imshow(restoredTukey, []);
title('Restored Image (after
Deconvolution)');

%% Frequency domain plot for the original
image
F_original =
fftshift(fft2(blurredImage)); % 2D FFT
and shift zero frequency to center
mag_original = abs(F_original); %
Magnitude of the Fourier transform

% Frequency domain plot for the image
with Tukey window
F_tukey = fftshift(fft2(blurredTukey)); %
2D FFT of the Tukey windowed image
mag_tukey = abs(F_tukey); % Magnitude of
the Fourier transform

% Frequency domain plot for the restored
image after deconvolution
F_restored =
fftshift(fft2(restoredTukey)); % 2D FFT
of the restored image
mag_restored = abs(F_restored); %
Magnitude of the Fourier transform

% Display results in the frequency domain
figure;

% Show the magnitude spectrum of the
original image
subplot(1, 3, 1);
imshow(log(1 + mag_original), []); % Use
log scale for better visualization
title('Magnitude Spectrum of Original
Image');
```

```
% Show the magnitude spectrum of the
image with Tukey window
subplot(1, 3, 2);
imshow(log(1 + mag_tukey), []); % Use log
scale for better visualization
title('Magnitude Spectrum of Image with
Tukey Window');

% Show the magnitude spectrum of the
restored image after deconvolution
subplot(1, 3, 3);
imshow(log(1 + mag_restored), []); % Use
log scale for better visualization
title('Magnitude Spectrum of Restored
Image (Deconvolution)');
```