Applications of Signal Processing Theory

Christian G. Cuaresma[1], Jerald Tyrone V. Tadena[2], Evitha Jovel E. Viola[3]

[1,2,3]*Department of Electronics Engineering, Faculty of Engineering, University of Santo Tomas, Manila, Philippines,*
[1]christian.cuaresma.eng@ust.edu.ph
[2]jeraldtyrone.tadena.eng@ust.edu.ph
[3]evithajovel.viola.eng@ust.edu.ph

***Abstract*— This paper presents the implementation and analysis of signal processing techniques applied to real-world problems. It involves five tasks that address practical scenarios in signal processing. The first task is simulating a guided transmission line by analyzing its magnitude response with various input signals and evaluating its power transmission efficiency. The second task focuses on designing an 8th-order filter to approximate a target magnitude response, highlighting the significance of filtering in isolating specific signal components. In the third task, a digital oscillator is designed to generate the signal from the previous task when excited by an impulse input, demonstrating the role of oscillators in producing periodic waveforms. The fourth task utilizes frequency analysis through the Discrete Fourier Transform (DFT) to determine which keypad button was pressed by interpreting a touch-tone audio signal. Finally, the fifth task addresses image deconvolution, aiming to restore a blurred image by reducing noise and improving image clarity. In conclusion, signal processing has a variety of applications that extend beyond the focus of this paper. Future research could dive deeper into other applications of signal processing theory.**

***Keywords*— Signal processing, Transmission line, Digital Oscillator, Filtering, Discrete Fourier Transform, Deconvolution**

I. Introduction

Signal processing focuses on the analysis, modification, and synthesis of signals. These signals can be in various forms, such as audio, visual, electrical, or other data types measured over time or in different spatial dimensions. Signal processing aims to extract information, eliminate noise, and convert the signal into a more appropriate format for analysis or transmission. Signal processing is applied in numerous fields, including telecommunications, audio engineering, medical, image processing, etc.

II. Applications

Application 1: Transmission Lines

In communication systems, transmission media carry signals between locations and can be classified as guided or unguided. The task is to simulate a guided transmission line using a coaxial cable. A well-designed transmission line, characterized by a uniform impedance, ensures maximum power transfer from the transmitting to the receiving end. However, physical damage to the line may lead to undesired power reflections.

The transmission line is modeled by the function [y] = translate (x), where x represents the input signal and y is the output signal. To evaluate the performance of the transmission line, the task involves designing an input signal to probe the system and approximate its magnitude response via system identification techniques, accounting for the presence of noise.

Input signals can be generated using various windowing methods, including impulse, rectangular, Hamming, Hanning, Kaiser, or Blackman-Harris windows. The pulse duration can be varied (e.g., 10 or 100), and zero-padding is applied to ensure appropriate sequence length.

For analysis, the following are plotted:

1. Input sequence in the time and frequency domains,
2. Output sequence in the time and frequency domains, and
3. The approximated magnitude response of the transmission line.

This approach facilitates identifying and characterizing the transmission line's behavior under noisy conditions.

Implementation

Sending a signal over a transmission line can be simulated through MATLAB code. An input signal needs to be generated; this will be the signal that will pass through the given system, acting as the transmission line. An impulse signal can be utilized for simplicity. The input signal is constructed as a simple impulse sequence. The pulse duration is set to 10 samples, and after padding with zeros, the sequence reaches a total length of 128 samples. It has a broad frequency spectrum, and it ensures it can act as a probing signal, which is ideal for system identification.

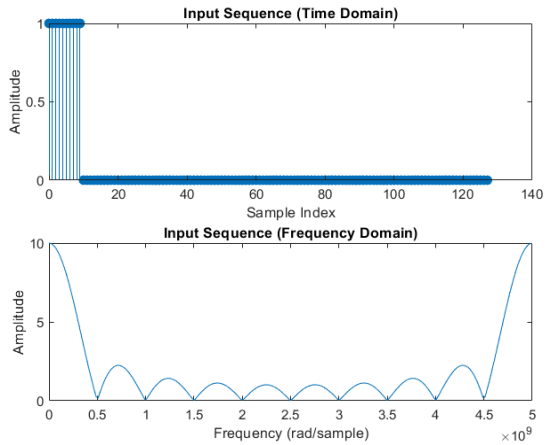

Figure 1.1. The input signal is in the time domain and frequency domain.

The given function, translate.p, which serves as the simulation's transmission line, convolves the input signal with the system's impulse response. Also, it introduces slight Gaussian noise to simulate a real-world scenario. The output reflects the combined effect of the transmission line's filtering properties and the additional noise.

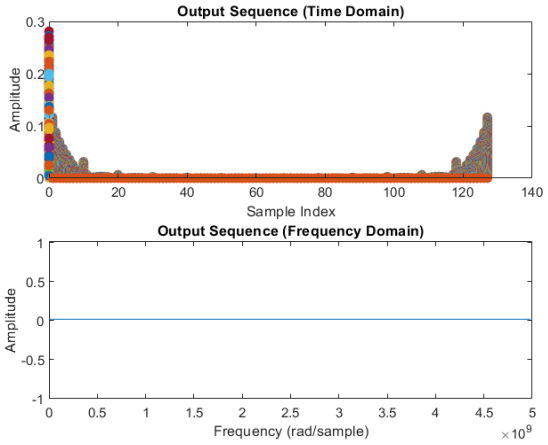

Figure 1.2. The output signal in the time domain and frequency domain.

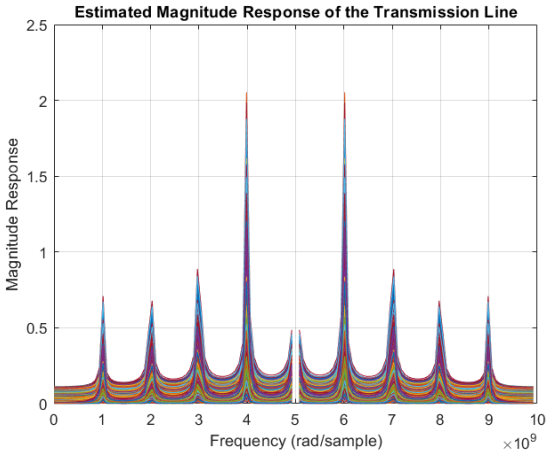

Figure 1.3. The approximated/estimated magnitude response of the transmission line.

After generating the input signal and the output signal after the given system has processed it, the magnitude response can be evaluated. The input and output signals are converted to the frequency domain using the Fast Fourier Transform (FFT), and the magnitude response can be obtained from the ratio of the magnitude of the FFT of the output signal and the magnitude of the FFT of the input signal.

Analysis and Discussion of Results

It can be observed that the input signal has ripples due to the constant amplitude found in impulse signals. Using the function representing the transmission line, a resulting signal with evident additional noise is displayed, as the function aims to simulate. As the magnitude response is evaluated, many amplitude peaks can be observed, showing that the transmitted signal is subjected to artificial interference.

Conclusion

For a signal to be received as clearly as possible, the transmission process must amplify the relevant aspects of the signal and must be able to cut off unwanted noise so the signal would still be of use at the other end.

Application 2: Filter Design

Another application for signal processing is filter design. An 8th-order filter with the magnitude response is to be designed with an impulse response, as shown in Figure 2.1.

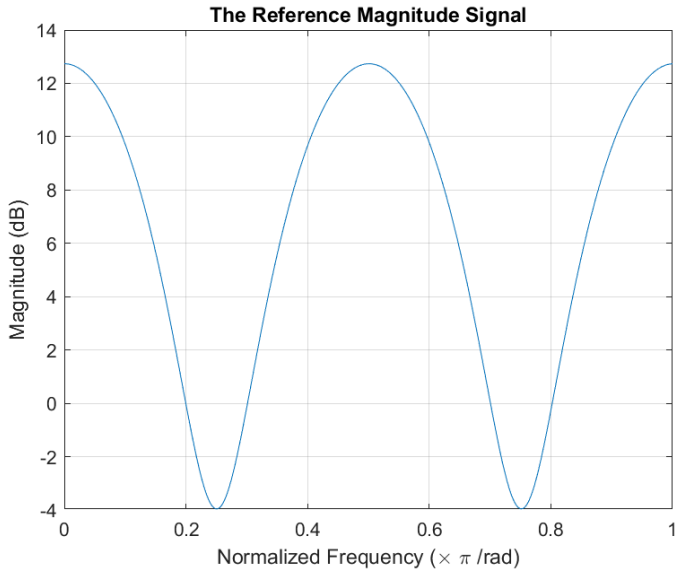

Figure 2.1. The expected magnitude response.

An input signal x(n) would then be passed in the designed filter. The output signal would then be plotted in the time and frequency domain.

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

IMPLEMENTATION

In designing the filter, the reference magnitude signal was thoroughly analyzed designing the filter d to see the placement of poles and zeros. The peaks in the magnitude signal mean that there are poles in their respective frequencies. Therefore, a pole is at frequencies 0, π/2, and π. On the other hand, The troughs in the wave mean that there are zeros in those frequencies. Likewise, there is a zero at frequencies π/4 and 3π/4. The peak of approximately 12 dB, having a more significant effect than the attenuation of approximately -4 dB, suggests that there would be more poles than zeros inside the unit circle. Additionally, the oscillatory behavior of the magnitude response indicates that there would be conjugate pairs in the pole-zero plot.

For ease in the pole-zero placement approach, a function `rad2im` was devised where the input is an angle in radians, and the output is the imaginary number in polar form. To quickly see the placement of poles and zeros and how it relates to the magnitude response, a function `pole_zero` was made where the inputs are poles and zeros, and the output gives the filter coefficients, the pole-zero plot, magnitude response, and phase response. To adjust how pronounced the poles and zeros would affect the response, the poles and zeros are multiplied by a scale factor in the input of the function `pole_zero`.

ANALYSIS AND DISCUSSION OF RESULTS

After a thorough analysis of the reference signal, the designed filter has a pole-zero plot as follows:

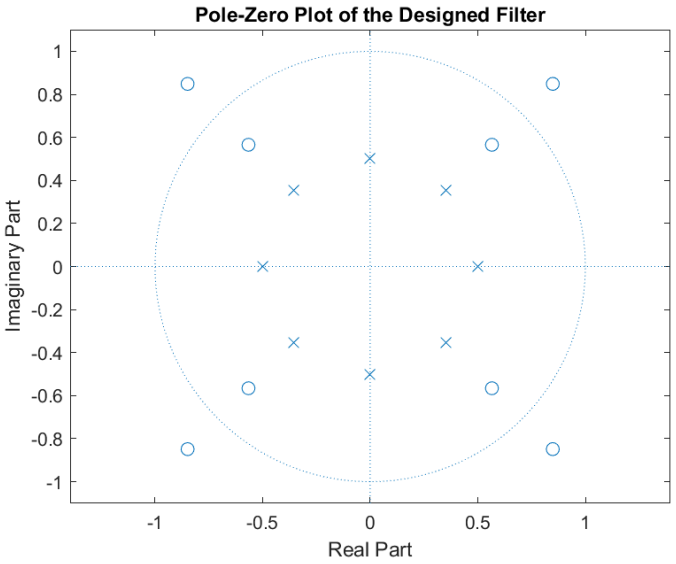

Figure 2.2. The pole-zero plot of the designed filter.

Figure 2.2 shows the pole-zero plot. As mentioned, there is a pole at frequencies 0, π/2, and π, and there is a zero at frequencies π/4 and 3π/4. Since the problem requires an eighth-order filter, equidistant points are added so that the response is not affected by the addition of poles and zeros.

The magnitude and phase response of the designed filter is seen in Figure 2.3.

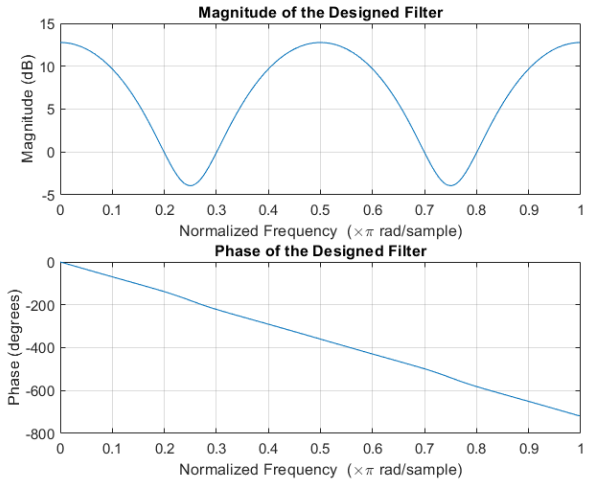

Figure 2.3. The magnitude and phase response of the filter.

The phase response shows how the output signal has a phase shift in the corresponding frequency. The phase response resembles a linear graph with a negative slope.

From the pole-zero plot and the frequency response plot, the properties of the system are observed:

a. The system is stable as all the poles are inside the unit circle. This means that there is a bounded output for a bounded input signal.
b. The energy of the system is bounded.
c. The resonant frequencies are the frequencies corresponding to the angular frequency of the poles. In this case, the resonant frequencies are 0, π/2, π, and 3π/2. These are the angular frequencies without corresponding zeros.
d. The oscillation in the magnitude response is sustained since the poles and zeros are equidistant from the unit circle. The oscillatory behavior is also due to the conjugate pairs of poles and zeros.
e. The linear phase response suggests as the frequency increases, the phase shift of the output decreases. This also means that the frequency components are delayed by the same time.

Afterward, the input signal x(n) is passed through the designed filter and viewed in the time and frequency domain.

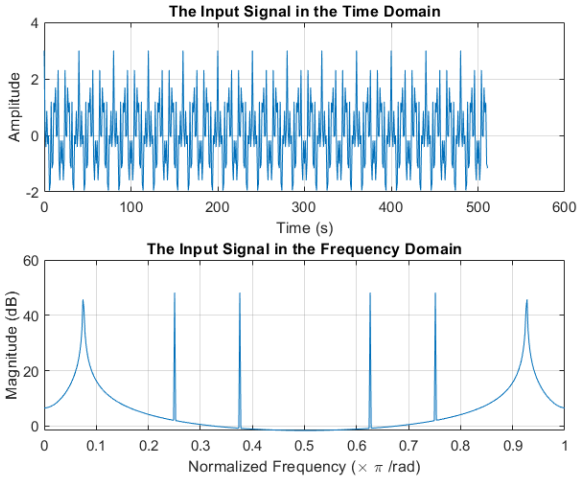

Figure 2.4. The input signal in the time and frequency domain.

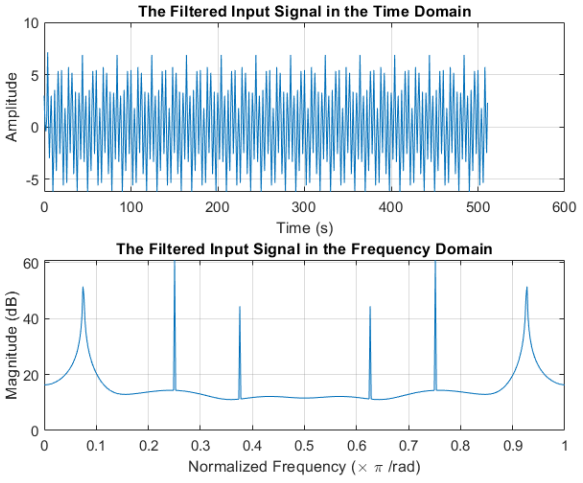

Figure 2.5. The filtered signal in the time and frequency domain.

The output plot of the filtered input signal matches the magnitude of the system's response. It is observed that the lower frequencies will be delayed more than higher frequencies, but this delay is uniform across all components of the signal. The effect is that all frequencies are delayed by the same relative amount, which means the waveform shape is preserved (no distortion).

Conclusion

In summary, the properties of a system are observed/. It is seen that the stability of the filter and the frequency response are affected by the position of poles and zeros. The magnitude and frequency response show how the system acts across different frequencies and if there would be distortion in the output. The filtered signal reflects the magnitude and phase response of the signal in which the filter's characteristics are applied.

Application 3: Digital Oscillator Design

The third application of digital signal processing is designing a digital oscillator with a response x(n) when excited with an impulse signal □(n).

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

Implementation

By inspection, it can be seen that the signal is a composite signal that is the sum of three cosine components with different frequencies. This means that the overall impulse response can be found by finding the contribution of each element to the overall system. Additionally, after passing through individual subsystems, the sum of signals results in a parallel configuration. The process is shown in the block diagram in Figure 3.1.

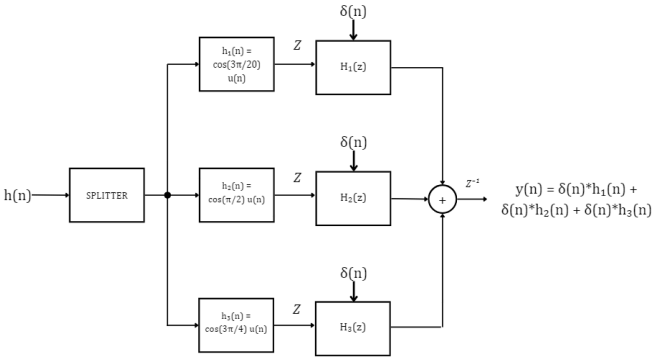

Figure 3.1. The block diagram for the digital oscillator design.

For ease, each subsystem was z-transformed, and the impz() function was used to find the impulse response in the code implementation. The resulting plot was plotted and compared to the original signal.

Analysis and Discussion of Results

The first 100 samples of the signal (both the original and the oscillator) were retrieved. The comparison of the results is plotted in Figure 3.2.

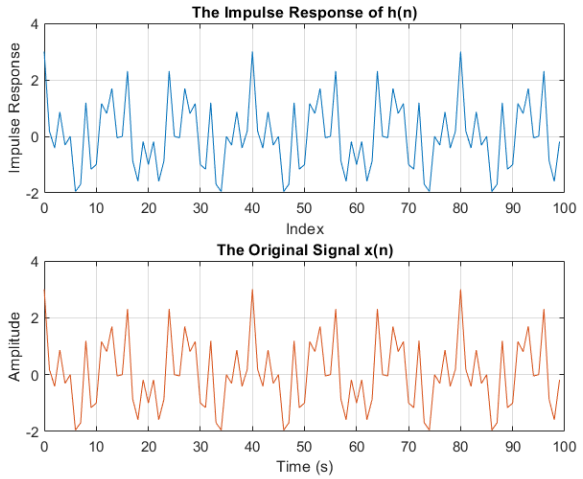

Figure 3.2. A comparison of the plots of the impulse response of the designed oscillator and the original signal.

In the figure above, it is observed that the original signal x(n) is accurately represented by the impulse response of the system h(n). The z-transform of each component was taken and multiplied by the corresponding system transfer function (or the impulse response. The result is the sum of the individual outputs. If it were inverse z-transformed, the original signal would be x(n) because the Z-transform has a linearity property. Thus, the original signal is reconstructed.

Conclusion

In conclusion, the impulse response of a composite signal is a parallel system since each subsystem operates independently. When the composite signal is broken down into its components, the impulse response of each subsystem is applied and then added back together, and the original signal is essentially reconstructed.

Application 4: Discrete Fourier Transform in Telephony

DTMF (Dual-Tone Multi-Frequency) is the general term for the push-button telephone signaling systems, which is the same as the Touch-Tone system used within the Bell System. DTMF is also commonly used in electronic mail systems and telephone banking, where users can choose options from a menu by sending DTMF signals via telephone. In a DTMF signaling system, a specific digit or the characters * and # are represented by a combination of a high-frequency tone and a low-frequency tone; it uses a combination of two sine tones to signal which button is currently being pressed.

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Figure 4.1. Push-button Telephone system with its Corresponding Frequency

A. Indices of Highest Amplitude When Button 9 is Pressed

A DTMF signal is generated by pressing a button on a 4x4 keypad, producing two sinusoidal frequencies: one from the row and one from the column of the button. For button "9", the corresponding frequencies are 852 Hz (row) and 1477 Hz (column). The signal is digitized with a sampling frequency of 8kHz.

To analyze the signal:

1. A 256-sample segment is extracted from the signal.

2. The segment is multiplied by a hamming window to reduce spectral leakage.

3. A 256-point DFT is computed to determine the frequency components.

The objective is to find the DFT indices corresponding to the highest amplitudes, representing the frequencies generated by the button "9".

Implementation

Given a sampling frequency *fs* = 8kHz and a DFT size of $N = 256$, the frequency resolution is $f_{res} = \frac{f_s}{N} = 31.25\ Hz$. The indices of the DFT corresponding to the frequencies of the interest are calculated as:

$$k = \frac{fN}{f_s}$$

For $f_{low} = 852\ Hz$:

$$k_{low} = \frac{(852)(256)}{(8000)} = 27.264 = \sim \mathbf{27}$$

For $f_{high} = 1477\ Hz$:

$$k_{high} = \frac{(1477)(256)}{(8000)} = 47.264 = \sim \mathbf{47}$$

Thus, the highest amplitudes in the DFT spectrum are expected at indices 27 and 47.

This procedure has been adapted into MATLAB code. The following parameters are initialized: given sampling frequency (fs = 8000), given DFT size (N = 256), and the given row and column frequencies that represent button "9" (f_low = 852, f_high = 1477).

To generate the DTMF signal for button "9", the 256 DT samples are converted into CT time vector (t = nT). This will generate the signal representing the button "9" being pressed by adding the sinusoidal signals containing both low and high frequencies (852 Hz & 1477 Hz).

A windowing function, specifically the Hamming window, is then applied to the signal. Afterwards, the 256-point DFT signal is computed by using the Fast Fourier Transform. The amplitude spectrum is also computed for observation.

The peak indices were then pinpointed, and the results were visualized for further inspection.

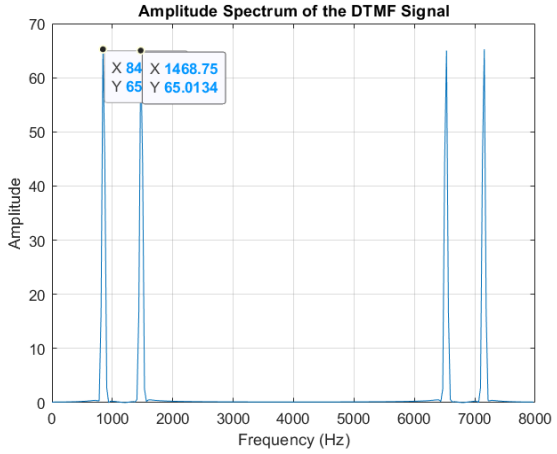

Figure 4.1.1 The plot of the amplitude spectrum that shows prominent peaks at around 852 Hz and 1477 Hz.

Analysis and Discussion of Results

As can be seen from the results of both the manual calculations and resultant plot generated by the MATLAB code, the peak amplitudes at around both 852 Hz and 1477 Hz, which represent button "9" being pressed, are approximately found around the indices of **27 and 47** in the resultant vector.

B. Determine Button Sequence in touchtone.wav

Implementation

The goal is to determine which specific button is pressed in the given DTMF audio. This can be achieved by iteratively identifying which frequency corresponds to the keypad for each audio frame.

The process begins by inputting and playing the DTMF audio file ("touchtone.wav") while initializing the telephone keypad with its buttons and the corresponding high and low frequencies associated with each button.

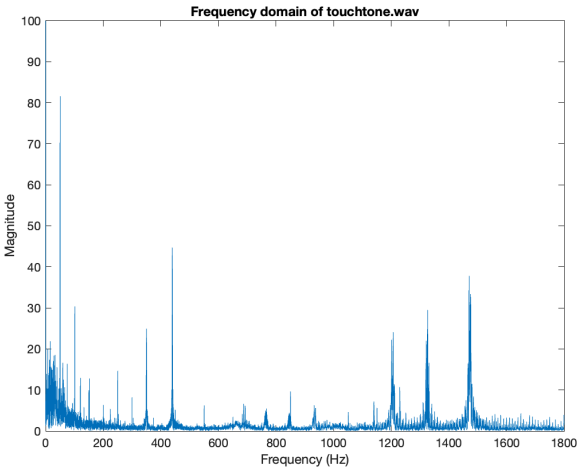

Figure 4.2.1. Frequency domain of DTMF audio

In Figure 4.2.1, the "touchtone.wav" audio is plotted in its frequency domain to visualize its frequency components. Additionally, the built-in MATLAB command *floor()* is applied to the length of the audio. An error was encountered due to a "non-integer operand," which occurs when a colon expression has a non-integer value during plotting. To resolve this, the *floor()* function rounds the value down to its nearest integer, ensuring that only integer values are used for indexing. Limits were also applied to both the x and y axes to provide a clear plot view.

The Goertzel algorithm is used to detect specific frequency components within a signal, making it ideal for decoding Dual-Tone Multi-Frequency (DTMF) signals. A DTMF tone consists of two distinct frequencies: one from a low-frequency group and one from a high-frequency group.

Variable *N* defines the frame size, where *fs* is the sampling rate of the *"touchtone.wav"* audio. The value that would be multiplied by fs is the scaling factor that determines how much of the audio signal is processed in each frame relative to the sampling rate; varying the scaling factor is proportional to the number of samples per frame that it reads, setting 0.21 as the scaling factor. The frequencies are normalized based on the number of samples and the sampling rate. The *"coefficient"* variable computes the coefficients for the Goertzel algorithm. A *for-loop* is utilized in the analysis of the frequencies of the DTMF audio; the "for i" loop iterates through the audio signal frame by frame, and the loop *"for freq_idx"* iterates through each frequency of the tones in the

keypad. Furthermore, data in prev_value and prev_value2 are accumulated to calculate the magnitude of each frequency in the current frame being processed. The magnitude(freq_idx) stores the resulting magnitude for each frequency., and the "for sample" loop iterates over each sample within the audio frame. The function "magnitude(freq_idx)" calculates the magnitude of the detected frequency in the signal. By evaluating the magnitudes of all frequencies in the frame, the highest magnitudes for each frequency row and column identify the pressed key. The key that matches the frequency is then appended.

Analysis and discussion of results

The Goertzel algorithm is a signal processing method used in DTMF decoding to efficiently detect specific frequencies within an audio signal. This algorithm significantly reduces the computational complexity compared to a full Discrete Fourier Transform. Focusing only on the frequencies corresponding to the tones generated by pressing a keypad on a telephone.

The "touchtone.wav" audio was processed using the Goertzel algorithm. Each key on the keypad corresponds to a unique combination of high and low frequencies, and the algorithm was applied frame by frame to identify these frequencies.

The code proved that the Goertzel algorithm successfully detected the frequencies corresponding to each key pressed. By evaluating the magnitude of each frequency in the signal, the algorithm identified the key by comparing the magnitudes of the frequencies in the audio frame. The highest magnitudes for each row and column of frequencies identified the specific key pressed, and the key that matched will be appended to the output.

The detected keypad sequence is **999900441223394676879.** Using online tools like a DTMF audio generator, the tones generated from the sequence "999900441223394676879" matched the decoded tones from the "touchtone.wav" DTMF audio.

Conclusion

Overall, the MATLAB code effectively decoded the DTMF audio from *"touchtone.wav"* and identified the keys that were pressed. The use of the Goertzel algorithm, normalization, and frequency analysis enabled accurate detection of the DTMF frequencies. Verification with an online DTMF audio sequence generator confirmed that the decoded sequence matched the input audio from "touchtone.wav" in the MATLAB code. This process showed the effectiveness of using algorithms such as Goertzel for decoding frequencies.

Application 5: Deconvolution in Image Processing

Blurred or unclear images can significantly affect someone's daily life, affecting communication, decision-making, and various aspects, especially in medical imaging where image precision is crucial. Image degradation typically arises from motion, defocus, or other distortions and is mathematically modeled as a convolution with a Point Spread Function (PSF). This results in the loss of high-frequency details necessary for maintaining sharpness and clarity. Image deblurring is a fundamental aspect of image processing that seeks to restore sharp and clear visuals from blurred images. MATLAB provides a way to address this by reversing the effects of image blur through both blind and non-blind deblurring methods.

Implementation

The goal is to utilize MATLAB's image deconvolution function to restore a given blurred image, *"stars-blurred.png,"* as shown in Figure 5.1. and its corresponding point spread function (PSF) *"stars-psf.png"* as shown in Figure 5.2.

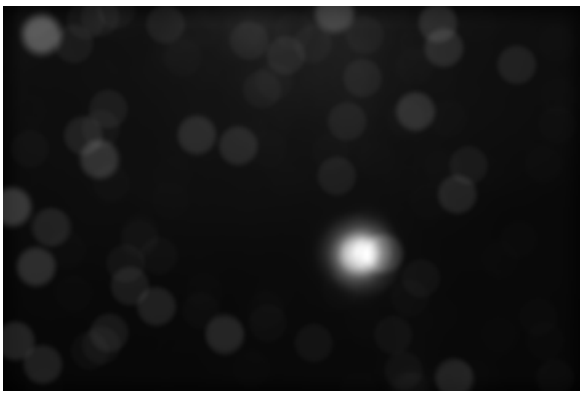

Figure 5.1 Blurred Image ("stars-blurred.png")

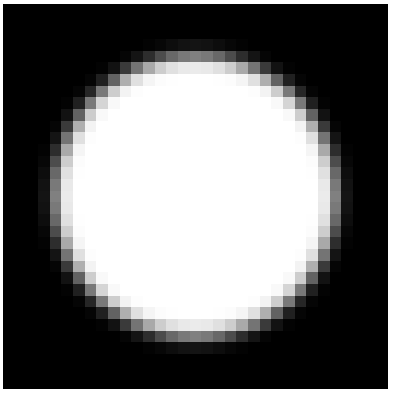

Figure 5.2 Image Point-spread Function ("stars-psf.png")

MATLAB's built-in command imread(), loads an inputted image and returns it as a matrix. In this part, the group loaded the image to be deconvolved, including the point spread function (PSF), representing the imaging system's blurring effect. Furthermore, it describes the impulse response of a focused optical imaging system to a point source or point object. *imshow()* is a built-in command in Matlab that displays an image.

The function *im2double()* converts the image I to double precision. The images that will be deconvolved underwent this process. This process is crucial for image deconvolution as it converts the image data to double-precision floating-point numbers. Consequently, it enhances the precision and reliability of deconvolution by providing a more robust and accurate representation of the image data.

Both images are normalized since this is crucial for deconvolution, especially when reducing blur from images. This ensures that it sums to 1, maintains energy conservation, improves stability, and achieves consistent scaling, leading to more accurate results.

Deconvolution reverses the blurring process by estimating the original, sharp image from the blurred image and its PSF. The Richardson-Lucy deconvolution algorithm was applied in the image deblurring process using the built-in MATLAB function 'deconvlucy'. It is an iterative algorithm that is used to restore an image that has been blurred by convolution with PSF. The command iteratively refines the initial estimate and attempts to recover the original, unblurred image. The more iterations, the better the image quality becomes. In this case, 150 iterations were applied to maximize the deblurring process and enable clearer deconvolved image output. This deconvolution algorithm showed the best result among other available deconvolution commands.

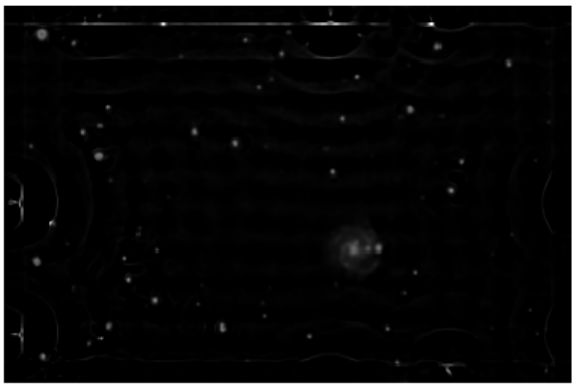

Figure 5.3. Deconvolved Image Without Windowing

Focusing on Figure 5.3. It can be seen that there is some distortion on the edges of the deconvolved image. The group applied windowing to the blurred star image, which controls the noise and distortions on the margins of the image, without the windowing process. Furthermore, the *Tukeywin* or the Tukey (tapered cosine) window can be utilized for windowing. It provided the best output among other windowing commands.

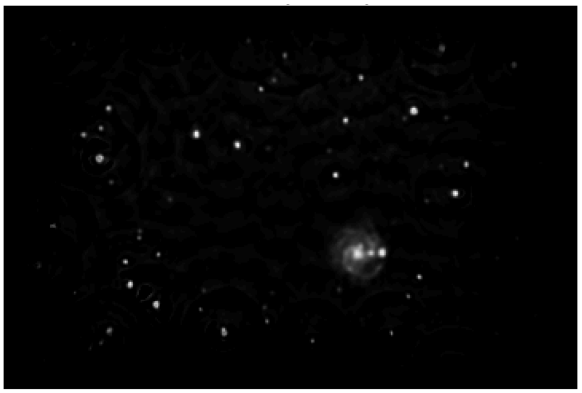

Figure 5.4. Deconvolved Image with Windowing

Focusing on Figure 5.4, the deconvolved image has noticeably large pixels. To address this issue, MATLAB's built-in command *imresize()* and the 'bilinear' method, effectively reduce the size of the pixels generated from the deconvolved image and by utilizing a scaling factor of 5 since it already produced a good-quality image.

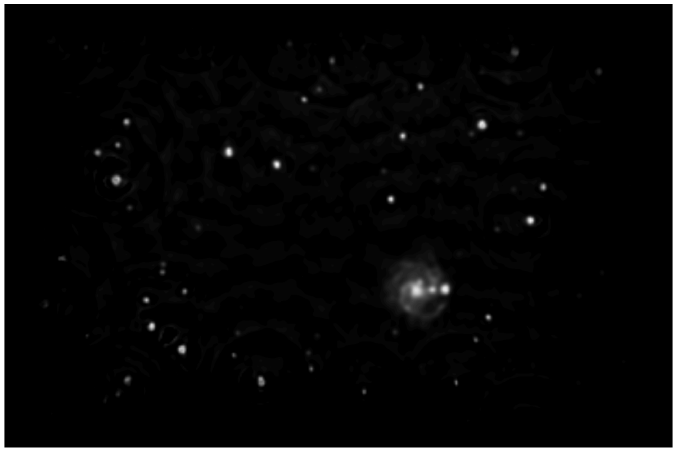

Figure 5.5. Scaled Deconvolved Image with Windowing

In comparison to the original blurred image in Figure 5.1, the stars in the scaled deconvolved windowed image shown in Figure 5.5 appear more distinct, as the blurring effect has been minimized. The deconvolution and windowing techniques helped to reduce noise and artifacts, enhancing the overall clarity.

Analysis and Discussion of Results

The Richardson-Lucy deconvolution algorithm effectively deblurred the blurred image by iteratively refining the initial estimate to recover the unblurred image. It works by estimating the original, sharp image from the blurred image and its PSF. Both images were normalized to maintain consistent energy levels during the processing. By using 150 iterations, the iterative Richardson-Lucy algorithm successfully recovered high-frequency details, resulting in an improved version of the blurred image. However, some margin distortions and noise remained. To mitigate these distortions, a Tukey (tapered cosine) window was applied, which significantly reduced the margin artifacts, although it also led to some data loss at the edges. Additionally, scaling the windowed image helped reduce pixel size.

Conclusion

The MATLAB code implements image deblurring or deconvolution in MATLAB. The code begins by loading the blurred image and its corresponding Point Spread Function (PSF) using *imread()*. Initially, convert the image to double-precision for improved accuracy, normalize the PSF for stability, and apply windowing to minimize margin distortions. Applying the *deconvlucy()* MATLAB command with 150 iterations to effectively restore the image. Finally, the deconvolved image is resized using *imresize()* with a bilinear method and a scaling factor of 5 to reduce pixel size and enhance visual quality. This approach effectively addresses image blurring through MATLAB's built-in command and selecting the best parameters to achieve the best results.

References

[1] R. M. Gonzalez and R. E. Woods, *Digital Signal Processing Using MATLAB*, 3rd ed. [Online]. Available: https://research.iaun.ac.ir/pdfs/UploadFile_6417. [Accessed: Dec. 15, 2024].

[2] S. K. Mitra, *Digital Signal Processing: Principles, Algorithms, and Applications*, 3rd ed. New York, NY, USA: McGraw-Hill, 2006. [Online]. Available: https://uvceee.wordpress.com/wp-content/uploads/2016/09/digital_signal_processing_principles_algorithms_and_applications_third_edition.pdf. [Accessed: Dec. 15, 2024].

[3] OnlineSound, "DTMF Generator - Online Sound Tools," [Online]. Available: https://onlinesound.net/dtmf-generator. [Accessed: 15-Dec-2024].

Distribution of Task

Application 1	Tadena, Jerald Tyrone V.
Application 2	Viola, Evitha Jovel E,
Application 3	Viola, Evitha Jovel E.
Application 4	Cuaresma, Christian G. Tadena, Jerald Tyrone V.
Application 5	Cuaresma, Christian G.

Appendix

A. Code for Application 1

```
%% Machine Problem
%{
ECE 21113
AY 24-25
%}
clc % reset Command Window
clear % reset Workspace
close all % close figure windows
%% Initialization
fmax = 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); % change in frequency
f = -fs/2:deltaf:fs/2; % range of
frequencies
%% Input signal x
% Task 1: Design an input sequence x
x = zeros(1,128); % placeholder only
% ---- Insert your code here
x(1:10) = 1; % generate impulse signal
Sx = fft(x,256); % Fourier Transform for
frequency domain
w = ((0:255)/256)*(fs/2); % frequency axis
figure(1); % plot figure
subplot(2,1,1) % subplot for input signal
in time domain
stem(0:length(x)-1,x,'filled') % create
plot
title('Input Sequence (Time Domain)') % add
title
xlabel('Sample Index') % add x-axis label
ylabel('Amplitude') % add y-axis label
```

```
subplot(2,1,2) % subplot for input signal
in frequency domain
plot(w,abs(Sx(1:256))); % create figure
title('Input Sequence (Frequency Domain)')
% add title
xlabel('Frequency (rad/sample)') % add
x-axis label
ylabel('Amplitude') % add y-axis label
x = transpose(x); % transpose input signal
before passing through system
%% Input signal passes through the system
y = transline(x); % given system
representing transmission line
%% System Identification
% Task 2: Approximate the magnitude
response of the system
% ---- Insert your code here
Sy = fft(y,256); % Fourier Transform for
frequency domain
w = ((0:255)/256)*(fs/2); % frequency axis
figure(2); % plot figure
subplot(2,1,1) % subplot for output in time
domain
stem(0:min(size(y))-1,abs(y),'filled') %
create plot
title('Output Sequence (Time Domain)') %
add title
xlabel('Sample Index') % add x-axis label
ylabel('Amplitude') % add y-axis label
subplot(2,1,2) % subplot for output in
frequency domain
plot(w,abs(Sy(1:256))); % create figure
title('Output Sequence (Frequency Domain)')
% add title
xlabel('Frequency (rad/sample)') % add
x-axis label
ylabel('Amplitude') % add y-axis label
X = fft(x); % FFT of input signal
Y = fft(y); % FFT of output signal
f = (0:length(X)-1)*(fs/length(X)); %
Frequency vector
H = abs(Y) ./ abs(X); % Magnitude response
figure(3); % plot figure
plot(f, abs(H)); % create plot
title('Estimated Magnitude Response of the
Transmission Line'); % add title
xlabel('Frequency (rad/sample)'); % add
x-axis label
ylabel('Magnitude Response'); % add y-axis
label
grid on; % add grid to plot
%% Note:
%{
Plot the signals in time and frequency
domain as necessary.
In designing the input sequence, vary
parameters to observe trends.
%}
```

B. Code for Application 2

```
function [equi] = rad2im(radian)
% rad2im - converts angle in radians to
polar form.
% z = cosx + jsinx
%
% Input:
% radian - radian
%
% Output:
% equi - radian in Euler's Form
%
% For Application 2: ECE21113L - Grp 8
equi = cos(radian) + 1j*sin(radian);


function [b,a] = pole_zero(p,z)
% pole_zero - plots pole zero plot and both
magnitude and frequency
% response. has a gain to increase the gain
of the system.
%
% Inputs:
% p - poles
% z - zeroes
% gain - gain
%
% Outputs:
% b - numerator
% a - denominator
% Pole-Zero Plot
% Frequency Response
%
% For Application 2: ECE21113L - Grp 8
b = poly(z); % numerator
a = poly(p); % denominator
figure;
zplane(b,a);
figure;
freqz(b,a);


% ECE21113L - Application 2
% b and a are generated from:
% >> % [b,a] = pole_zero(0.5*rad2im([0,
pi/4, pi/2, 3*pi/4, pi, -pi/4, -pi/2,
-3*pi/4]),
% [0.8*rad2im([pi/4, -pi/4, pi*3/4,
-pi*3/4]) 1.2*rad2im([pi/4, -pi/4, pi*3/4,
-pi*3/4])],1)
n = 0:511; % 512 points
x_n =
(cos((pi*3/20)*n)+cos((pi/2)*n)+cos((pi*3/4
)*n)).*ustep(length(n)); % the input signal
% the plotted signal in the time domain
figure;
subplot(2,1,1);
plot(n,x_n)
title('The Input Signal in the Time
Domain')
xlabel('Time (s)')
ylabel('Amplitude')
grid on
% Frequency domain
S_input = fft(x_n, 512);
w = linspace(0, pi, 512); % the frequency
vector
subplot(2,1,2);
plot(w/pi,20*log10(S_input))
title('The Input Signal in the Frequency
Domain')
xlabel('Normalized Frequency (\times \pi
/rad)')
ylabel('Magnitude (dB)')
grid on
filtered_signal = filter(b,a,x_n); % the
filtered signal using the coefficients of
the designed filter
figure;
subplot(2,1,1);
plot(n,filtered_signal);
title('The Filtered Input Signal in the
Time Domain')
xlabel('Time (s)')
```

```
ylabel('Amplitude')
grid on
S_filter = fft(filtered_signal, 512);
subplot(2,1,2);
plot(w/pi,20*log10(S_filter))
title('The Filtered Input Signal in the
Frequency Domain')
xlabel('Normalized Frequency (\times \pi
/rad)')
ylabel('Magnitude (dB)')
grid on
```

C. Code for Application 3

```
% ECE21113L: Grp 8 - Application 3
f = [3*pi/20, pi/2, 3*pi/4];  % frequencies
for the three sine waves

% z transform of system 1,
cos(pi*3/20)*u(n)
num1 = [1 -cos(f(1))];
denom1 = [1 -2*cos(f(1)) 1];

% z transform of system 2, cos(pi/2)*u(n)
num2 = [1 -cos(f(2))];
denom2 = [1 -2*cos(f(2)) 1];

% z transform of system 3, cos(pi*3/4)*u(n)
num3 = [1 -cos(f(3))];
denom3 = [1 -2*cos(f(3)) 1];

% Compute the impulse response for each
oscillator, with the first 100 samples
% (parallel components)
[h1, n1] = impz(num1, denom1, 100);
[h2, n2] = impz(num2, denom2, 100);
[h3, n3] = impz(num3, denom3, 100);

% Combine the impulse responses for the
overall response
[h_12,n_12] = sigadd(h1,n1,h2,n2);
[h_total,n_total] =
sigadd(h_12,n_12,h3,n3);

% comparison of plots of the impulse
response vs the original signal x(n)
figure;
subplot(2,1,1);
plot(n_total,h_total)
title('The Impulse Response of h(n)')
xlabel('Index')
ylabel('Impulse Response')
grid on
% the input signal
n = n_total;
x_n =
(cos((f(1))*n')+cos((pi/2)*n')+cos((pi*3/4)
*n')).*ustep(length(n));
subplot(2,1,2);
plot(n,x_n)
title('The Original Signal x(n)')
xlabel('Time (s)')
ylabel('Amplitude')
grid on
```

D. Code for Application 4

A.

```
%% ECE21113L 3ECE-A MP Group8; Item 4a
% Parameters
fs = 8000;            % Sampling frequency
in Hz
N = 256;              % DFT size
f_low = 852;          % Low frequency (row
frequency for '9')
f_high = 1477;        % High frequency
(column frequency for '9')
% Generate the DTMF signal for button '9'
t = (0:N-1) / fs;     % Time vector for 256
samples
signal = sin(2*pi*f_low*t) +
sin(2*pi*f_high*t);   % Combined DTMF signal
% Apply a windowing function (Hamming
window)
window = hamming(N)';
windowed_signal = signal .* window;
% Compute the 256-point DFT
dft_signal = fft(windowed_signal, N);
% Compute the amplitude spectrum
amplitude_spectrum = abs(dft_signal);
% Find the peak indices
[~, k_low] =
max(amplitude_spectrum(1:N/2)); % Search
for low frequency
k_high = round(f_high * N / fs);
% Compute high-frequency index manually
% Output results
fprintf('Peak frequency indices:\n');
fprintf('Low frequency (852 Hz): Index
%d\n', k_low);
fprintf('High frequency (1477 Hz): Index
%d\n', k_high);
% Plot the amplitude spectrum
frequencies = (0:N-1) * (fs / N);
% Frequency vector
plot(frequencies, amplitude_spectrum);
xlabel('Frequency (Hz)');
ylabel('Amplitude');
title('Amplitude Spectrum of the DTMF
Signal');
grid on;
```

B.

```
%% Inputting of DTMF audio
[audio, fs] = audioread('touchtone.wav');
sound(audio,fs)

%% Initializing keypad
%row / low frequencies keypad
row = [697, 770, 852, 941];
%column / high frequencies keypad
column = [1209, 1336, 1477, 1633];
%keypad buttons
buttons = ['1', '2', '3', 'A'; '4', '5',
'6', 'B'; '7', '8', '9', 'C'; '*', '0',
'#', 'D'];

%% Plotting the input DTMF audio in
Frequency Domain
% Plot of the input signal in frequency
domain
L = length(audio);
f = (0:L-1)*(fs/L);
fft_audio = abs(fft(audio));
L=floor(L/2);
figure;
plot(f(1:L), fft_audio(1:L));
xlabel('Frequency (Hz)');
ylabel('Magnitude');
title('Frequency domain of touchtone.wav');
xlim([0 1800]);
ylim([0 100]);
```

```
%% Initializing the Goertzel algorithm
frequencies = [row, column];
% Frame size captured in samples
N = 0.21 * fs;
% Normalizing frequencies & equation of
Goertzel
k = (frequencies / fs * N);
coeffiecient = 2 * cos(2 * pi * k / N);
% Initialize the keypand
decoded = '';
% Analyze audio frame by frame
for i = 1:N:(length(audio) - N)
   % The current frame
   current_frame = audio(i:i + N - 1);
   magnitude = zeros(size(frequencies));
   % Using the Goertzel algorithm
   for freq_idx = 1:length(frequencies)
       prev_value = 0;
       prev_value2 = 0;

       for sample = (current_frame)'
           current_value = sample +
coeffiecient(freq_idx) * prev_value -
prev_value2;
           prev_value2 = prev_value;
           prev_value = current_value;
       end
       magnitude(freq_idx) =
(prev_value2).^2 + (prev_value).^2 -
coeffiecient(freq_idx) * prev_value *
prev_value2;
   end
   % Detect keypad used
   [~, row_key] = max(magnitude(1:4));
   [~, col_key] = max(magnitude(5:8));
   % Append detected key
   decoded = [decoded, buttons(row_key,
col_key)];
end

%% Display decoded keypad sequence
disp('Decoded keypad sequence: ');
disp(decoded);
```

E. Code for Application 5

```
%% load images
image1=imread('stars-blurred.png');
psf= imread('stars-psf.png');

%% displays the given images
figure;
imshow(image1);
title('Blurred Image');
figure;
imshow(psf);
title('psf');

%% double precision for images
image1_doubled = im2double(image1);
psf_doubled = im2double(psf);

%% normalizing PSF
psf_normalized = psf_doubled /
sum(psf_doubled(:));
% deconvolve using lucy-richardson
iteration = 150;
deconvolved_imageNW =
deconvlucy(image1_doubled, psf_doubled,
iteration);

%% applying windowing to the margins
[mrow, mcolumn] = size(image1_doubled);
windowed = tukeywin(mrow) *
tukeywin(mcolumn)';
image1_windowed = image1_doubled .*
windowed;

%% deconvolve using lucy-richardson with
window
deconvolved_image =
deconvlucy(image1_windowed, psf_normalized,
iteration);

%% reduce size of pixels
scale = 5;
Deconvolved_image =
imresize(deconvolved_image, scale,
'bilinear');

%% output
figure;
imshow(deconvolved_imageNW,[]);
title('Deconvolved Image without
Windowing');
figure;
imshow(deconvolved_image,[]);
title('Deconvolved Image with Windowing ');
figure;
imshow(Deconvolved_image,[]);
title('Deconvolved Image');
```