Quadrature Amplitude Modulation: A Comprehensive Analysis and Implementation of Modulation, Demodulation, and Signal Recovery

Alein Miguel P. Capañarihan
University of Santo Tomas
aleinmiguel.capanairhan.eng@ust.edu.ph

Khatelyn Jhuneryll E. Deang
University of Santo Tomas
khatelynjhuneryll.deang.eng@ust.edu.ph

Mary Margot Sofia A. Enriquez
University of Santo Tomas
marymargot.enriquez.eng@ust.edu.ph

***Abstract* - Communications systems greatly impact day-to-day life. They ensure that the message signals reach the correct destination. However, problems may arise during the transmission, such as attenuation. This is why the systems must also ensure that the signal received at the destination is accurate to the original signal. This study aims to recreate a communication system through coding in MATLAB. The specific communication system recreated is the Quadrature Amplitude Modulation (QAM) system. Additionally, the group was tasked with recording and replicating a voice signal. Through this, the group has discovered that the QAM system was able to accurately replicate the message signal, and they are better able to understand the processing that the signals go through.**

Index Terms - Amplitude Modulation, Communication Systems, MATLAB, QAM, Quadrature Amplitude Modulation

Introduction

Communication systems are important in everyday life and have many uses—texting, calling, and even sending files over the Internet are possible because of these systems. However, this brings the question of how do these communication systems transmit these messages. Despite problems like attenuation, distortion, and long distances, how can these systems accurately pass on the input signal? This paper seeks to answer this question by coding and simulating a basic communication system using MATLAB. Specifically, the paper will synthesize a Quadrature Amplitude Modulation scheme.

Quadrature Amplitude Modulation is a method in communications systems used for transmitting signals by combining some aspects of Phase Shift Keying (PSK) and Amplitude Shift Keying (ASK). Based on the name of the process, "quadrature," it can be inferred that the two signals will be phase-shifted so that they have a phase difference of 90 degrees [1]. This method both conserves bandwidth and helps the system stay efficient at utilizing the said bandwidth in the process of transmission [2].

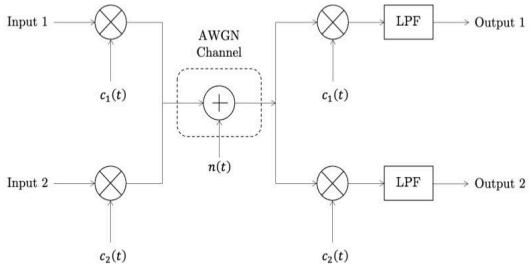

Figure 1. Quadrature Amplitude Modulation Diagram

Figure 1 above shows the diagram of the communication system called Quadrature Amplitude Modulation. In order to simulate the Quadrature Amplitude Modulation, the process will be divided into five distinct parts. These are Loading the Input Signal, Resampling the Input signals, Modulating the Signals, Demodulating the Received Signals, and Resampling the Received Signals. The first two stages are for preparing the input signal for modulation like padding zeros, and up-sampling the signal with the Nyquist Theorem. This theorem states that the sampling frequency should be at least twice the maximum frequency present in the original signal [3]. This theorem is applied during the sampling process to avoid aliasing. If the sampling rate is not sufficient, the signal will not be represented correctly and

a different signal entirely might appear. Moreover, it can be observed in the third stage (Modulation Stage) that there will be modulation of the two input voice signals' amplitude, which is the aspect of Amplitude Shift Keying mentioned earlier. In addition, this is what allows the signal to travel long distances with a high-frequency carrier. Once the two signals that were produced after ASK and PSK are summed up, it will result in the QAM signal with noise. Due to this, the succeeding stages after that are for filtering, demodulating, and resampling the signal to acquire the original message signal again. The specific filter chosen for this system is a low-pass filter that removes any unwanted noise and distortion, and it filters out the high-frequency carrier that damages the quality of the output.

Implementation

I. Loading the Input Signal

```
function [x1_pad, x2_pad, Fs_orig, L] =
load_input(input1_file, input2_file)
```
(1)

Two audio signals were recorded using a microphone and were then loaded into MATLAB. The first function, named "load input," is shown in (1), The inputs to this function are the filenames of the audio signal. The padded array representation (*x1_pad* and *x2_pad*), sampling frequency (*Fs_orig*), and the length of the padded array (*L*). The outputs in (1) will be explained later in this section.

```
[x1_orig, Fs1] = audioread(input1_file);
[x2_orig, Fs2] = audioread(input2_file);
```
(2)

First, to use the files within MATLAB, they need to be loaded. This can be achieved by using a built-in function called "audioread," shown in (2). This function is needed to read the data from the audio file that can then be utilized in MATLAB. The input requires a filename for which the data is to be read from, it then loads the data into the application by storing it in MATLAB workspace. The file can now be called at any point during coding. The expected output for audioread is the resulting sampled data of the voice files (*x1_orig* and *x2_orig*) and their corresponding sampling frequencies (*Fs1* and *Fs2*).

```
    if Fs1 ~= Fs2
        error('Sampling frequencies do not
match.');
    end
    Fs_orig = Fs1;

    L1 = length(x1_orig);
    L2 = length(x2_orig);
    if L1 > L2
        x1_pad = x1_orig;
        x2_pad = [x2_orig; zeros(L1 - L2,
1)];
    else
        x1_pad = [x1_orig; zeros(L2 - L1,
1)];
        x2_pad = x2_orig;
    end

    L = max(L1, L2);
```
(3)

The next step would be to make sure that the two input signals are the same length. This is tested by comparing the sampling frequency of the two signals in (3). If they are not the same then, it will produce an error message. Next, the code "length()" is used to check how many samples there are in the output of the audioread function. In the loading signals stage at (3), if-else loops are utilized to pad zeros depending on whether they are equal or not. The "zeros()" built-in function creates an array of zeros based on the size inputted. In this case, the size will depend on the difference between the length of the two. The resulting zero array will then be appended to the signal that needs it, thus ensuring that the signals will always be the same length after implementing the first stage. The outputs after this padding will be stored in the variables *x1_pad* and *x2_pad*. Their final length will be stored in the variable *L*.

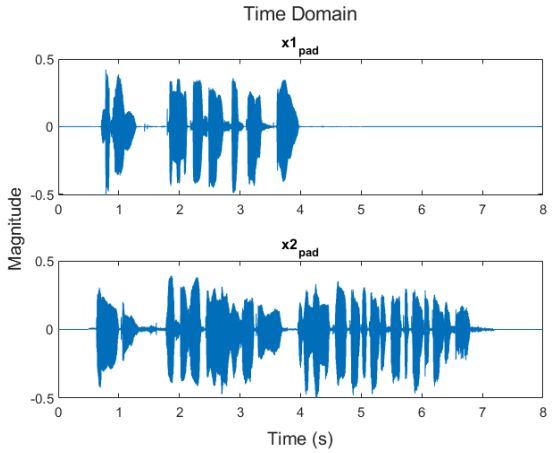

Figure 2. Padded Input Signals In The Time Domain

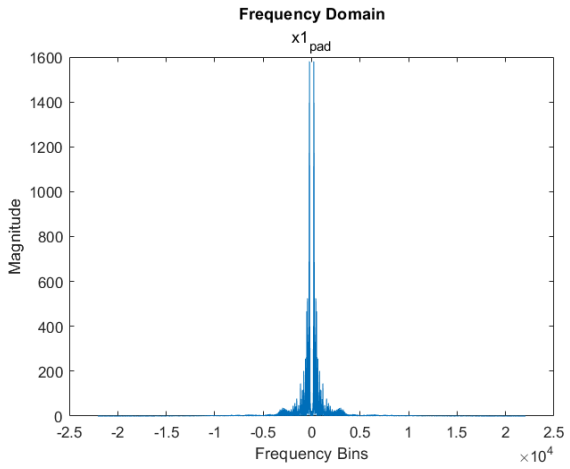

FIGURE 3. PADDED INPUT SIGNALS IN THE FREQUENCY DOMAIN

II. Resampling the Input Signals

```
function [x1_up, x2_up,t] =
resample_signal(x1_pad, x2_pad, Fs,
Fs_orig)
```

(4)

$$F_s > 2F_{max} \tag{5}$$

The goal of this stage is to up-sample the original message signals in preparation for Modulation. The function "resample_signal" is shown in (4). The function requires four inputs: The padded message signals from the previous stage (*x1_pad* and *x2_pad*), the original sampling frequency (*Fs_orig)*, and the desired sampling rate (*Fs*). Proper selection of *Fs* is important, as a low sampling frequency value results in aliasing, resulting in distortion or artifacts in the signal. The minimum sampling frequency is computed using the Nyquist Theorem in Equation (5). In this design, a sampling frequency of 500kHz was chosen.

```
x1_up = resample(x1_pad, Fs, Fs_orig);
x2_up = resample(x2_pad, Fs, Fs_orig);
```

(6)

This stage's code is simple and straightforward due to the use of the built-in MATLAB function "resample," shown in (6). This function takes the input signal to be resampled and uses the ratio of the second and third input (*Fs* and *Fs_orig*) multiplied by the original sampling frequency of the padded signal to resample it. For further security, it also applies an Anti Aliasing Low Pass Filter. The harder part in this stage was finding the correct ratio to get an accurate replica of the two voice signals at the output. The two outputs for this stage will now be named *x1_new* and *x2_new*. These signals are now ready to be modulated in the next stage.

III. Modulation Stage

```
function [x_AM] = qam_modulation(x1_up,
x2_up, set_fc, t)
```

(7)

```
t = 0:1/Fs:(length(x1_up)-1)/Fs;
```

(8)

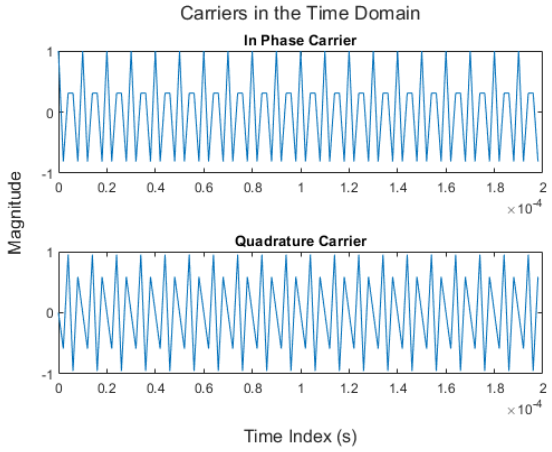

FIGURE 4. IN PHASE AND QUADRATURE CARRIERS

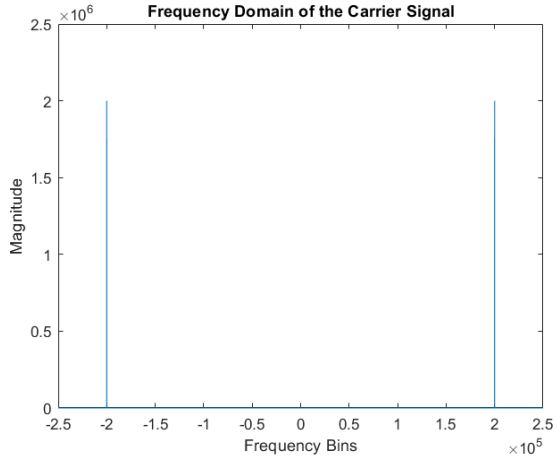

FIGURE 5. IN PHASE CARRIER IN THE FREQUENCY DOMAIN

In this stage, the two signals will separate and move into two different branches. These branches are the in-phase branch and the quadrature branch. Their main difference is that at the quadrature branch, the signals are not in phase and have a phase difference of 90 degrees, unlike in the in-phase branch. The name of the main function file for this stage will be "qam_modulation." The first step is to pick a suitable carrier. The time domain representation for the chosen carriers is shown in Fig. 4. The carriers must be high enough to carry the message signal over long distances. In this particular case, the frequency of the carrier chosen is 200kHz, as seen in Fig.

5. The number of samples in this array can be taken from the sampling frequency, 500kHz, chosen in the second stage.

$$m_1(t)\, A_c cos(\omega_c t) = m_1(t)\, cos(2\pi F_c t) \tag{9}$$

$$- m_2(t)\, sin(2\pi F_c t). \tag{10}$$

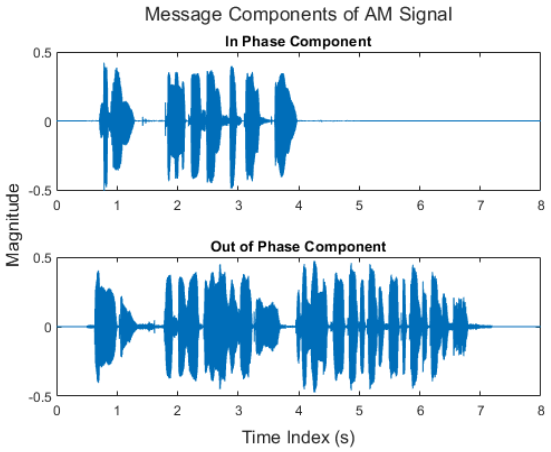

FIGURE 6. MODULATED MESSAGE SIGNALS

Since there are two branches, each will have a different equation assigned to them. For the in-phase branch, the equation is (9). Additionally, ω_c is equivalent to $2\pi F_c$ and the amplitude of the carrier is 1. Substituting the variables that were already obtained will result in the final equation being (9). Equation (10) shows the formula for the quadrature branch will be mostly the same, except that a negative sine function will be used instead of cosine because it is out of phase by 90 degrees. The final equation for this branch will be (10). After applying the code, it results in the plot Fig. 6.

```
I_component = x1_up .* cos(2 * pi * set_fc
* t).';
Q_component = x2_up .* -sin(2 * pi * set_fc
* t).';
x_AM = I_component + Q_component;
```
(11)

In (11), the code will be relatively simple and straightforward since it only substitutes the variables that the group already has in MATLAB into the two equations obtained for the in-phase and quadrature branches. After the two results from the equations are transposed, they are added together to get the modulated QAM signal using the (+) operator. This QAM signal is the final output for this stage and will be represented and stored in the variable *x_AM*. The plot for *x_AM* is shown in Fig. 7.

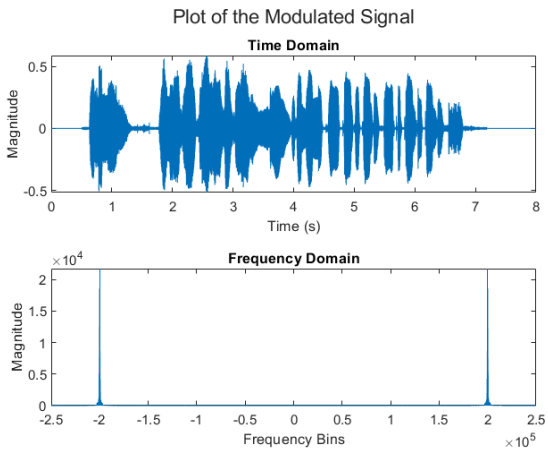

FIGURE 7. OUTPUT AM SIGNAL

IV. Demodulation Stage

```
function [y_1, y_2, y_filt_1, y_filt_2] =
qam_demodulation(rx, set_fc, fcut, Fs, t)
```
(12)

```
t = 0:1/Fs:(length(x1_up)-1)/Fs;
```
(8)

Here, (12) will essentially simulate what happens when the signal has already reached the receiver side at the destination. This starts the process of getting this received signal (*rx*) back to its original state and basically reverses the modulation process. The 'qam_demodulation function will need five inputs (*rx, set_fc, fcut, Fs, t*). All these were already obtained in the preceding stages except *fcut*, which is the cutoff frequency for the low pass filter used. For example, *rx* is simply just equal to *x_AM*, and in order, the other values are 200kHz, 20kHz, 500kHz, and *t* is the time array utilized for graphing (8). The value of the cutoff frequency was chosen in accordance with the knowledge that most audio frequency signals lie between the range of 20Hz to 20kHz. Therefore a cut-off frequency of 20kHz was chosen to preserve the message signal.

```
y_1 = rx .* cos(2 * pi * set_fc * t).';
y_2 = rx .* sin(2 * pi * set_fc * t).';
```
(13)

```
y_filt_1 = lowpass(y_1, fcut, Fs);
y_filt_2 = lowpass(y_2, fcut, Fs);
```
(14)

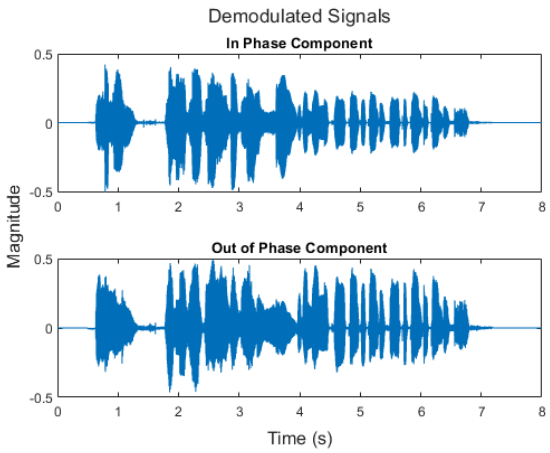

FIGURE 8. QAM SIGNAL AFTER DEMODULATION

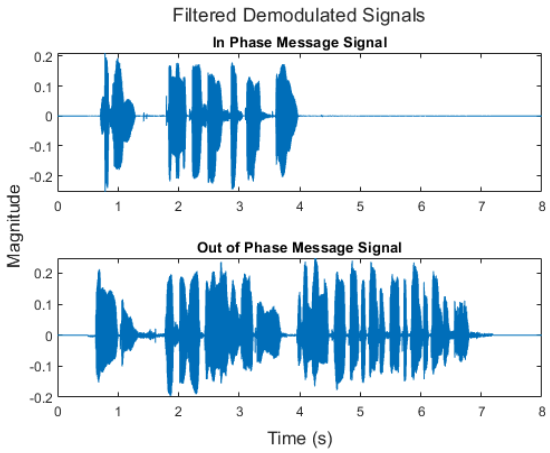

FIGURE 9. THE DEMODULATED SIGNAL AFTER FILTERING

For the code to divide the *x_AM* signal into two separate signals, the equations for the In-phase and Quadrature branches, shown in (13) and (14), must be utilized again, but instead of *x1_new* or *x2_new* in the equation, it will be replaced by *x_AM*. The output of these equations will be stored in *y_1* and *y_2*. These signals will then be passed through a low-pass filter with a cut-off frequency of *fcut*, this is done to isolate the low-frequency message signal from the high-frequency carrier signal This filter will be constructed using the built-in MATLAB function "lowpass." This specific filter function needs three inputs (*y_n, fcut, Fs*). This function will create a lowpass filter based on the specifications inputted, like the cut-off frequency and the sampling frequency defined earlier when calling the main function. Finally, the two outputs from the equations from this stage are inserted because they are the signals to be filtered. After filtering, the plotted signal should be similar to the plot from the Message Components of the AM signal. The filtered signals will then be stored in the variables *y_filt_1* and *y_filt_2,* and they will move on to the next stage. The plot for these should be similar to the plot in Fig. 6. Once, the output for this has been plotted in Fig. 9., it was confirmed that they are similar.

V. Resampling the Received Signals

```
function [y_down1, y_down2] =
downsample_signal(y_filt_1, y_filt_2, Fs,
Fs_orig)
   y_down1 = resample(y_filt_1, Fs_orig,
Fs);
   y_down2 = resample(y_filt_2, Fs_orig,
Fs);
end
```

(15)

This will be the final stage of the QAM scheme. Since the message signals were up-sampled, there is now a need to down-sample the filtered signals from the preceding stage to allow the playback device to properly reproduce the original signals. This process effectively reverses the up-sampling done in stage 2. The main function file name will be "downsample_signal." This will need four inputs (*y_filt_1, y_filt_2, Fs, Fs_orig*). The first two are the outputs from the Demodulation stage. The next two are the current sampling frequency after the up-sampling and the original message signal's sampling frequency. The code for this will be the same as the code from Stage 2. The only difference will be that the group has now switched the position of *Fs* and *Fs_orig* in the ratio, shown in (15). The output after the down-sampling shall be placed in the variables *y_down1* and *y_down2*. These signals should now be very close to the message signal. The results can be verified using the built-in function "sound()." This plays the audio of the signal that is inputted into its parentheses. The Sampling Frequency can also be set to be as accurate as possible. In this case, the value used will be the original sampling frequency stored in *Fs_orig*.

ANALYSIS AND DISCUSSION OF RESULTS

The results of the MATLAB implementation clearly demonstrate the accuracy and effectiveness of the QAM modulation and demodulation process. Initially, the input signal was upsampled to 500 kHz, increasing its sampling rate to allow smooth modulation at the chosen carrier frequency of 200 kHz. The upsampling ensured that the signal met the Nyquist criterion, which prevents aliasing during modulation. The successful generation of the modulated signal indicates that the system accurately shifted the original audio spectrum to the passband defined by the carrier. This step is essential for practical communication systems where signals need to be transmitted over higher-frequency carriers to travel long distances.

After modulation, the signal was demodulated using the same carrier frequency and subsequently passed through a low-pass filter with a cutoff frequency of 20 kHz. This cutoff frequency was chosen to preserve the essential frequency components of the original audio signal, which typically fall within the human audible range (20Hz to 20 kHz) while eliminating high-frequency components introduced during modulation. The low-pass filtering stage was crucial for suppressing residual carrier signals and harmonics that could cause interference or aliasing during downsampling. The appropriate choice ensured that the baseband signal was retained with minimal distortion, closely resembling the original signal before modulation. This highlights the importance of filter design and parameter selection in maintaining the integrity of the recovered signal.

Finally, the downsampling step brought the signal back to its original sampling frequency, aligning it with the specifications of the input audio. Thus, downsampling ensures compatibility with standard playback devices like speakers. Without downsampling, the higher sampling rate of the signal would exceed the playback device's capabilities, causing distortion or improper audio reproduction. The fact that the output audio is perceptually identical to the input audio demonstrates that the entire process—from upsampling and modulation to demodulation, filtering, and downsampling—was implemented without introducing any noticeable distortion or degradation. Moreover, the inclusion of a well-designed low-pass filter was instrumental in ensuring the signal's fidelity, as it preserved the audio frequency content while suppressing unwanted artifacts.

As for the analysis of the frequency response of the QAM signals, it reveals key insights into the modulation process. In the time domain, the padded input signals, *x1,* and *x2*, show significant amplitude variations, which are characteristic of QAM, as they encode data in both in-phase and quadrature components. Furthermore, in the frequency domain, the signal exhibits a concentrated energy distribution around the carrier frequency, with minimal spectral spread, indicating efficient bandwidth usage—a hallmark of well-implemented modulation. With that in mind, the message components in the in-phase and quadrature domains retain their distinct patterns, demonstrating successful separation of the orthogonal data streams during demodulation. In line with this, the time-domain representation of the modulated signal shows clear amplitude variations, indicating successful embedding of the message signal onto the carrier. In the frequency domain, the modulated signal exhibits distinct peaks corresponding to the carrier frequency and its sidebands, confirming proper spectral allocation. Additionally, the carrier signal's frequency-domain plot reveals symmetrical peaks, validating its role in supporting modulation.

The result of the project emphasizes the robustness of QAM as a modulation technique and the effectiveness of the processing chain in maintaining audio quality. It also showcases the system's suitability for applications where high-fidelity signal transmission and recovery are essential, such as audio broadcasting or digital communication systems.

Conclusion and Recommendations

In conclusion, due to the need for messages to traverse long distances, there is a need to modulate these signals to better suit the chosen medium, in this case, wirelessly. One such scheme is Quadrature Amplitude Modulation (QAM), which was utilized in this paper. To make it easier to comprehend, the group split this QAM scheme into five distinct stages. These are Loading the Input Signal, Upsampling, Modulation, Demodulation, and Downsampling.

After recording two voice signals and sending these signals through the QAM scheme, the group has found that the message signals were able to be retrieved accurately. Some key decisions that were important for ensuring accurate results were (i) choosing the sampling frequency, (ii) finding the correct equation for the phase shifting, (iii) picking an appropriate filter, etc.

It is worth noting that the resulting output files had a smaller file size than the original input message signals. However, the sound itself was still accurate to the original recordings. This smaller file size could be due to the loss of data somewhere in the process of modulating and demodulating the signal. Another notable observation is the reduction in volume of the received message signals, though this has relatively minor effects, the group believes it can still be remedied. Therefore, the group recommends that the code be improved upon and for future researchers to explore different parameters to input into the system to yield better results.

References

[1] D. T, (n.d.). Quadrature amplitude modulation. SRM University. https://webstor.srmist.edu.in/web_assets/srm_mainsite/files/files/Quadrature%20Amplitutde%20Modulation.pdf (accessed Nov. 28, 2024).

[2] B. Zhang, et al., Simulation and Performance Study of Quadrature Amplitude Modulation and Demodulation System. In IOP Conference Series: Materials Science and Engineering (Vol. 782, No. 4, p. 042048). IOP Publishing, 2020

[3] H. Austerlitz, Data acquisition techniques using PCs. Academic press, 2002.
[4] I. Kaminow, and T. Li, (Eds.). (2002). Optical fiber telecommunications IV-B: systems and impairments (Vol. 2). Elsevier, 2002.
[5] S. Haykin and M. Moher, An Introduction to Analog and Digital Communications, Wiley, 2012.
[6] S. Haykin, Communication Systems, 4th ed., Wiley, 2001.
[7] A. B. Carlson, et al., Communication Systems: An Introduction to Signals and Noise in Electrical Communication, 5th ed., McGraw-Hill, 2010.

Distribution of Tasks

Name	Tasks
Capañarihan, Alein Miguel P.	➢ QAM Code ➢ Implementation
Deang, Khatelyn Jhuneryll E.	➢ QAM Code ➢ Analysis and Discussion of Results
Enriquez, Mary Margot Sofia A.	➢ Abstract ➢ Introduction ➢ Implementation ➢ Conclusion and Recommendation

Table 1. Summary of Contributed Tasks

Appendix A: Matlab Code

1. Loading Input Signals:

```
function [x1_pad, x2_pad, Fs_orig, L] =
load_input(input1_file, input2_file)
   % Load the audio files
   [x1_orig, Fs1] = audioread(input1_file);
   [x2_orig, Fs2] = audioread(input2_file);
   % Test if sampling frequencies are
matching
   if Fs1 ~= Fs2
       error('Sampling frequencies do not
match.');
   end
   Fs_orig = Fs1; % Set original sampling
frequency
   % If length does not match, pad zeros
   L1 = length(x1_orig);
   L2 = length(x2_orig);
   if L1 > L2
       x1_pad = x1_orig;
       x2_pad = [x2_orig; zeros(L1 - L2,
1)];
   else
       x1_pad = [x1_orig; zeros(L2 - L1,
1)];
       x2_pad = x2_orig;
   end

   L = max(L1, L2);
end
```

2. Upsampling:

```
function [x1_up, x2_up,t] =
resample_signal(x1_pad, x2_pad, Fs,
Fs_orig)
   x1_up = resample(x1_pad, Fs,
Fs_orig);
   x2_up = resample(x2_pad, Fs,
Fs_orig);
   t = 0:1/Fs:(length(x1_up)-1)/Fs;
end
```

3. Modulation:

```
function [x_AM] = qam_modulation(x1_up,
x2_up, set_fc, t)

   I_component = x1_up .* cos(2 * pi *
set_fc * t).';
   Q_component = x2_up .* -sin(2 * pi *
set_fc * t).';
   x_AM = I_component + Q_component;
end
```

4. Demodulation:

```
function [y_1, y_2, y_filt_1, y_filt_2]
= qam_demodulation(rx, set_fc, fcut,
Fs, t)

   y_1 = rx .* cos(2 * pi * set_fc *
t).';
   y_2 = rx .* sin(2 * pi * set_fc *
t).';

   y_filt_1 = lowpass(y_1, fcut, Fs);
   y_filt_2 = lowpass(y_2, fcut, Fs);
end
```

5. Downsampling:

```
function [y_down1, y_down2] =
downsample_signal(y_filt_1, y_filt_2,
Fs, Fs_orig)
   y_down1 = resample(y_filt_1,
Fs_orig, Fs);
   y_down2 = resample(y_filt_2,
Fs_orig, Fs);
end
```