Implementation of the IEEE 802.11a Physical Layer (OFDM)

13.06.2021

Elsayed Mohammed Mostafa 201700316

Mohammed Younis Elsawi 201700915

Salma Hasan Elbess 201601152

Shaimaa Said Hassanen 201700249

University of Science and Technology - Zewail City

Overview

In modern communication systems, our main aim is to attain higher transmission rates with lowest possible resource consumption. Bandwidth, in communications, is a high cost resource that we always try to optimize its usage. OFDM is a method by which the orthogonal frequencies are used to each carry one symbol, so we can attain multiple speed rates with lower bandwidth. The frequencies chosen as carriers are selected in such a way that each carrier signal can be easily separated from the whole transmitted signal.

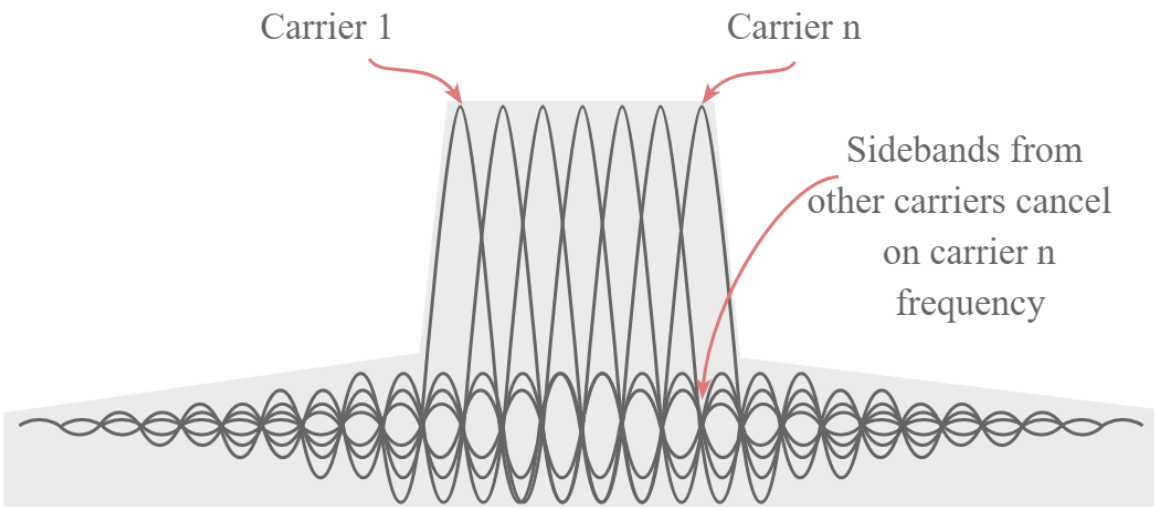

Figure 1. OFDM subcarriers [1]

In this project, we are implementing the WIFI transmitter and receiver using the IEEE 802.11a. We have 64 subcarriers where 48 are used for data transmission and the other 12 are transmitted as 'Zeros' to reduce adjacent channel interference. For forward error correction, the data to be transmitted are channel coded using a convolutional encoder with a rate of ½, ⅔ or ¾ . Veterbi decoder is used at the receiver to correct possible errors accumulated through noise. Data transmitted can be modulated by any of the four modulation techniques "BPSK", "QPSK", "16QAM" and "64QAM". AS for channel estimation and equalization we are using both Zero Forcing (ZF), and Weiner Filter.

Implementation Details

Main Pipeline

Our main code modules are shown in the following figure.

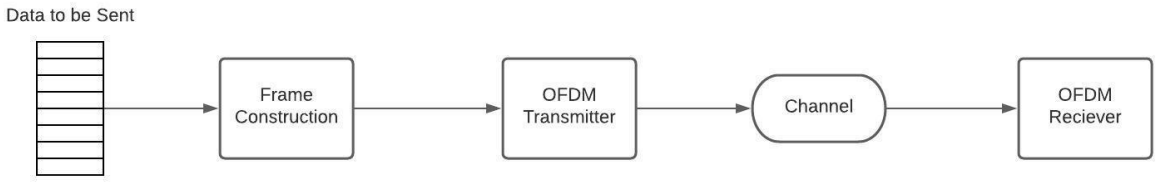

Figure 2. System Structure

We divide the data into frames then send the frames with the OFDM transmitter through the channel and at the receiver we recover the data using the OFDM receiver, each module has many sub-modules that will be discussed and explained in this section.

Frame Construction

The first step in the pipeline is to divide the data to be sent into frames with fixed size, the main idea of the frame is to encapsulate the data with some control information that will be used at the receiver side to decapsulate the data with high reliability. In our project we used the simplified frame structure introduced in the project document that is a simplified version of the frame structure of IEEE802.11a standard. The frame structure is shown in the following figure (3).

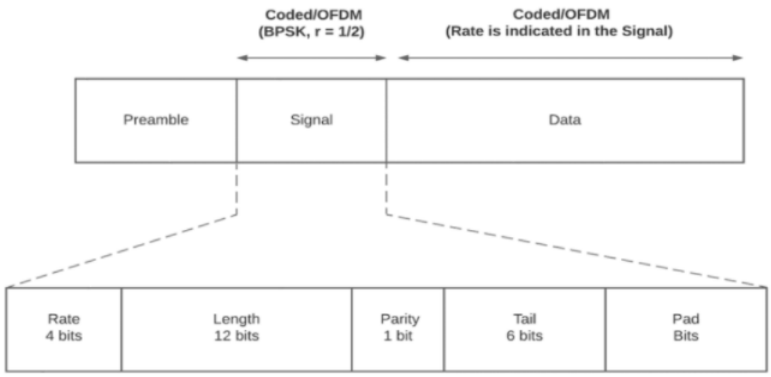

Figure 3. Signal part structure

In our code we have a *function [preamble, signal, data]* = **construct_frame***(data, mod_type, CR)* this function takes the data to be put into the frame (1032 bytes), the modulation type needed and the code rate for channel coding. In the function we construct each field of the frame; Preamble, Signal and Data. We also modulate the signal part with the most reliable configuration (BPSK modulation with code rate of 1/2) as this part is very important at the receiver as all the processing is done depending of the information contained in this field, then we modulate the data with the needed modulation type and finally convolutionally encode the data with the needed code rate. A *function [encoded_data]* = **conv_encoder***(data, rate)* is used to encode each frame of data with the code rate needed where the code rate ½ is achieved by using the matlab function **concev()** and other rates are implemented through puncturing. This function is called in **construct_frame()**. A corresponding *function* is called at the receiver side to decode the data *[decoded_data]* = **viterbi_decoder***(data, CR).* We also do the needed padding to the signal and data fields to be suitable for channel encoding and for OFDM transmission. The function output is the three fields of the frame; Preamble, Signal and Data.

Symbol Mapping and Gray Encoding

To implement an efficient symbol mapping function, we constructed two major vectors representing the x-axis and y-axis on any constellation. The axes values are decided based on the modulation scheme itself. For example, in 16QAM, the axes vectors are [-3, -1, 1, 3] as x-axis and the same vector as y-axis. These vectors are then converted into a meshgrid to have each point in the grid representing a modulated symbol. The symbols are then flattened in one vector that simplifies the modulation reference vector. Given a sample value to be modulated, the value of the sample is modulated into the modulation value that exists at the index equals the input sample minus 1. For example, if the input sample is 13, the modulation value -maybe (1, 3)- that exists at the index 12 is assigned to this sample. In case of gray encoding or any specific encoding scheme, the arrangement of the symbols inside the modulation reference vector is changed according to the mapping scheme. All these functionalities are implemented in our *function [mapped_stream, ref_vector]* = **modulate***(input_stream, input_type, mod_type, coding_scheme_vector).* Moreover, to normalize the modulated symbols as specified in the simplified version of IEEE802.11a standard, a specific value is multiplied by the modulation reference vector as mentioned in the standard such as $(1/\sqrt{42})$ in the case of 64QAM. This function is called at the transmitter to convert the input binary stream into modulation symbols.

At the receiver, once the symbols are received ,which could be noisy in general, the reverse steps are conducted. An implemented *function [output_steam, demod_samples]* = **demodulate***(input_stream, mod_type, output_type, coding_scheme_vector)* is used to get the binary data or the equivalent integer values of these binary output depending on the value

of *output_type.* As the input symbols are often noisy, the nearest possible value of the modulation reference vector is assigned to any given input symbol.

OFDM Implementation

The implementation of OFDM is done through two main modules that are called in a **main_scenario** that calls the transmitter, adds on the channel effect, then calls the receiver function to get the data out of the noisy received encoded frame.

1. Transmitter:

   function *with_cyclic_guards* = **WiFi_transmitter** *(bin_data, mod_type, rate, Nc, guard_len)*

   This function takes in the **bin_data** read from the file, along with the modulation type and coding rate desired, number of subcarriers **Nc**, and the length of cyclic prefix part **guard_len**. It firstly calls the function **construct_frame()** to get the frame of preamble, signal, and data as described earlier. Then it does the needed zero embeddings according to the pre-defined indices by the standard, to have a 64 length OFDM symbol for each part of the frame obtained by the function **construct_frame()**; preamble, signal and data. We then put the **tx_pilots** into the data into the *indices = [44, 57, 8, 22]* defined by the standard to then be used in synchronization. When it is all ready, we reconstruct the frame after preprocessing done to it *[preamble, signal, final_data]*, then apply serial to parallel conversion. The parallel data is then inputted to the IFFT module, adding cyclic guard bits with the length **guard_len**. It is reconverted back to serial to be transmitted through the simulated channel.

2. Receiver:

function out_decoded = **WiFi_receiver***(input_stream, Nc, guard_len)*

At the receiver side, we merely do the opposite processing done at the transmitter, to extract the data back and this may include some extra processing for noise removal and reliable data retrieval. The frame received is firstly converted to parallel in odere to remove the cyclic prefix from each symbol, and input it to the FFT module. It is converted back to serial frame, so be able to divide it back to preamble, signal and data by the length specified for each part by the standard. The **preamble**, particularly the long symbols, are used for channel estimation using the implemented function **estimate_channel**. They are then equalized with the desired equalizer using the implemented function ***equalize_channel***. The **signal** part is first decoded and demodulated back with “BPSK” and CR = ½ and then used to extract the needed information - data length, code rate, and modulation type - to get the binary data back. The data is then decoded using the implemented function **demodulate()** by the mod_type extracted from the **signal**. We then

remove the padding and decode the data using **viterbi_decoder()** with code rate extracted from the **signal** part.

AWGN Addition

To add Additive white gaussian noise to the received signal after simulating the wireless channel effect, we compute the AWGN power based on the signal power and the simulated SNR value. As our SNR value represents Eb/No which is the ratio between the energy per bit and the noise energy per bit, the power of the noise added to the signal part is different from the power of the noise added to the preamble and the data part of the transmitted frame. Hence, we separate the frame into its 3 parts and add the noise separately taking into consideration the number of bits per symbol depending on the modulation scheme.

Channel Estimation

Channel estimation is one of the very crucial processing steps that is done by the receiver as it is very important for channel equalization as will be discussed below. The preamble part -in particular, the long symbols of the preamble- is used for channel estimation. We have a *function channel_gains* = **estimate_channel***(rec_preamble),* This function takes the received preamble symbols as input and returns the channel gains as output. In the function we remove the nulls from the long symbols then simply divide the first received symbol over the value of the long symbol at the transmitter (value specified by IEEE802.11a standard), then repeating the same process for the second long symbols and finally averaging the gains resulting from the two symbols.

Channel Equalization

Channel equalization is used to equalize the effect of the channel on the received data. In our implementation we have Zero Forcing (ZF) equalization and Weiner (LMMSE) equalization both are implemented in *function* equalized_data = **equalize_channel***(data,channel_gains,equalization_method, Pz, Px)* This function takes as inputs the data to be equalized, channel gains obtained from channel estimation, equalization method, noise power level and data power level. In the function there are two main modules

1. ZF equalization

   In ZF we extract the channel gains at the data indices then we divide each OFDM data symbol by the channel gains to obtain the equalized data.

2. Weiner equalization

In the LMMSE method we just apply the frequency domain equation stated below to obtain Weiner weights, then we multiply the weights by each OFDM symbol to obtain the equalized data.

$$W_k^{LMMSE} = \frac{H_k^*}{|H_k|^2 + (\frac{P_z}{Px})}$$

Fixed point Implementation

After several experiments, the parameters of the fixed point implementation were optimized to balance between usage of resources and quantization errors. The parameters in the transmitter are window size = 8 and fraction size = 8. The parameters in the receiver, after FFT, are window size = 16 and fraction size = 12. The metric used for optimizing the parameters was the BER difference between fixed and floating point implementations. The rationale was to reduce the resources without exceeding a certain margin of difference.

Results and Discussion

In this section, we introduce and discuss the results we got using our implementation for different experiments.

I. Comparison of BER performance using both zero forcing equalizer and LMMSE equalizer

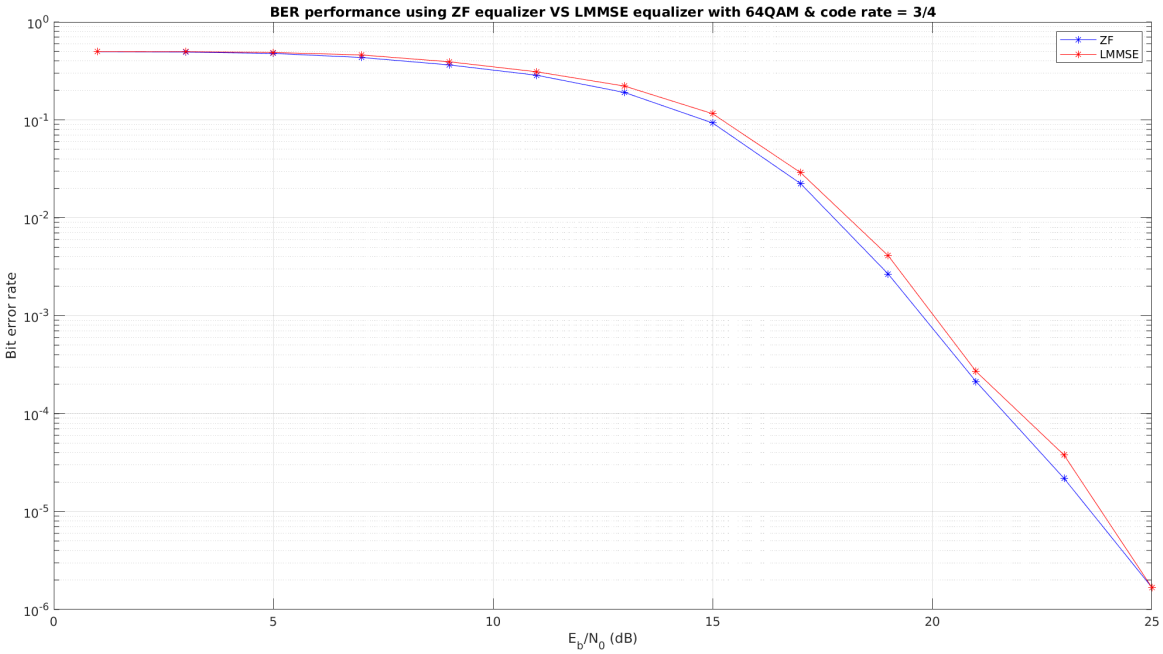

Figure 3. BER Comparison

Actually, the BER curve we got is not what we expected. The LMMSE BER performance is expected to be better than the ZF one especially at the low SNR. The worst performance of ZF is due to the noise enhancement occurring for the noise at low SNR.

II. Constellation diagram of the received symbols using both ZF equalizer and LMMSE equalizer at different SNR values.

As a point of comparison and to investigate the noise enhancement effect at using ZF equalizer, we plotted the constellation diagrams of the received symbols at both cases with 4 different SNR values as shown in the following figures. The used modulation scheme is 64QAM with ¾ as the code rate.

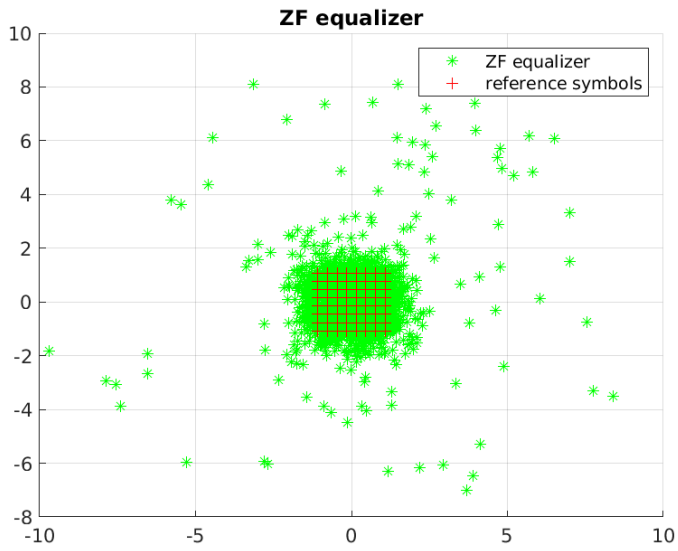

Figure 5. ZF equalizer constellation (SNR = 6 dB)

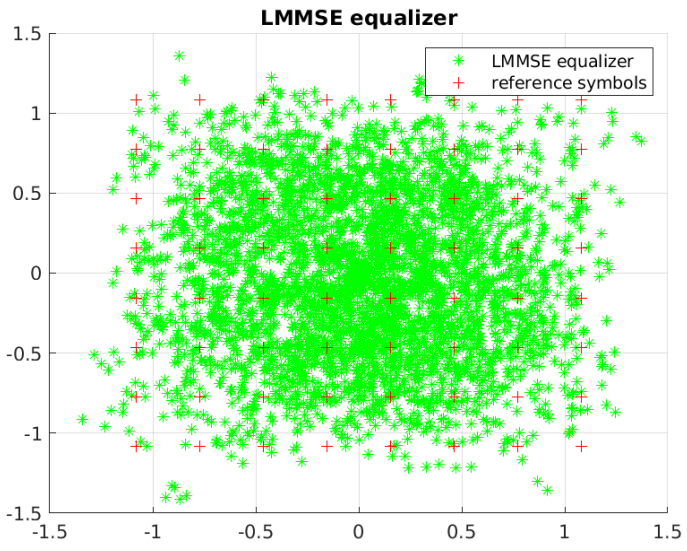

Figure 6. LMMSE equalizer constellation (SNR = 6 dB)

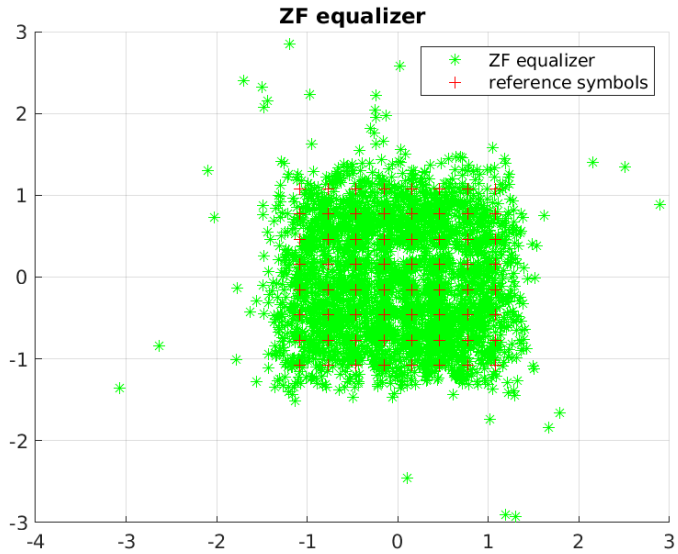

Figure 7. ZF equalizer constellation (SNR = 10 dB)

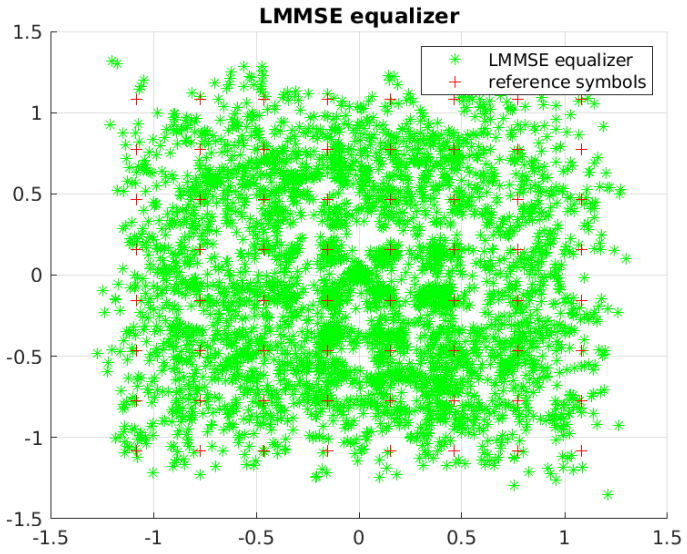

Figure 8. LMMSE equalizer constellation (SNR = 10 dB)

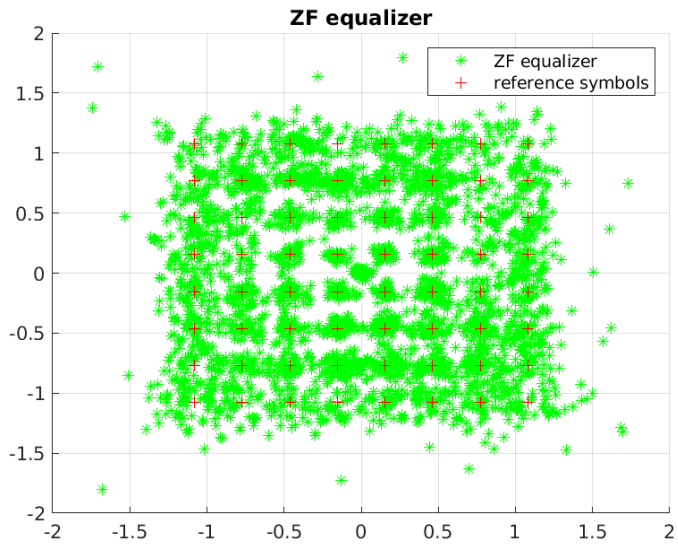

Figure 9. ZF equalizer constellation (SNR =15 dB)

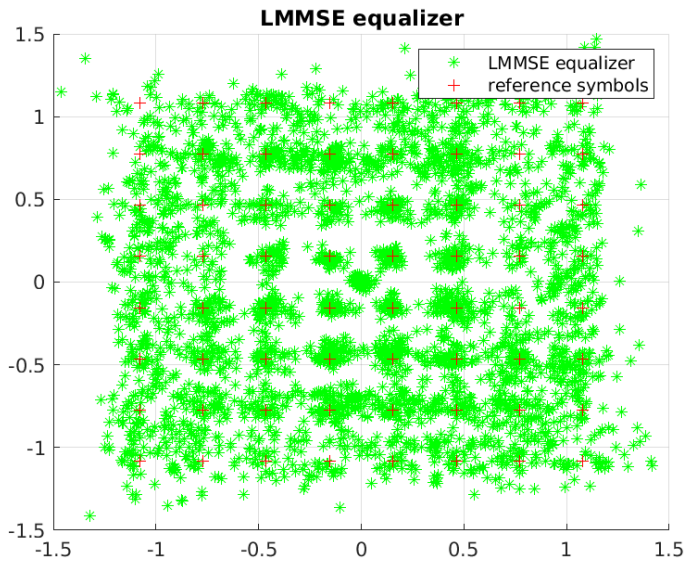

Figure 10. LMMSE equalizer constellation (SNR=15 dB)

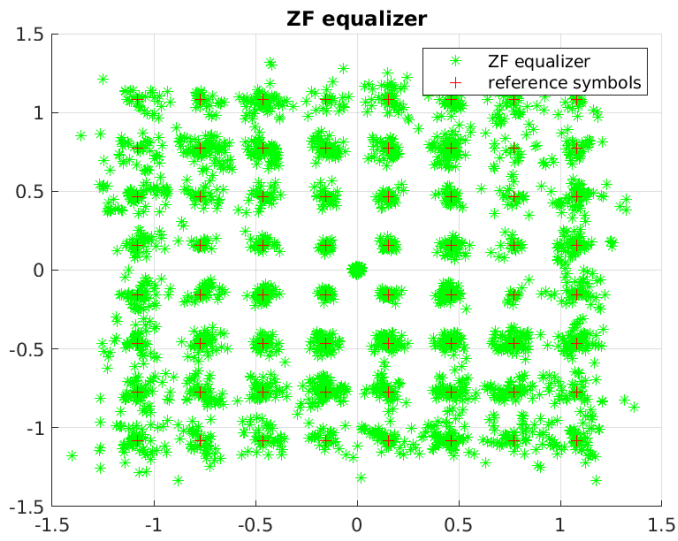

Figure 11. ZF equalizer constellation (SNR = 21 dB)

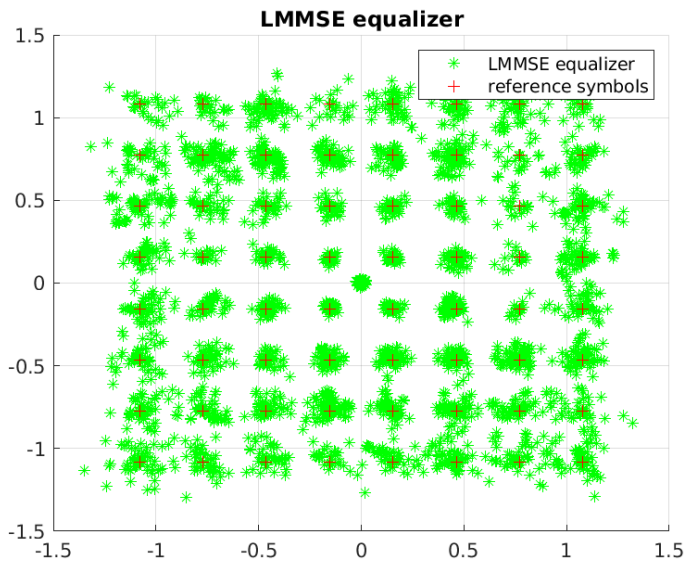

Figure 12. LMMSE equalizer constellation (SNR=21 dB)

From the previous figures, the performance of LMMSE equalizer is better than the performance of ZF equalizer especially at the low SNR as expected. This better performance is visually shown in the noise enhancement occurring at SNR of value 6 dB and 10 dB. The noise enhancement effect is shown in the abnormal high constellation values. The higher the SNR value, the similar the performance of LMMSE and ZF regarding the SER (symbol error rate) as expected.

For the zero symbols occurring at the middle of the constellations, these are the zero symbols used for padding the last data symbols in order to be multiple of 48 before adding the known zero symbols and pilots. They are not removed until decoding the bits as after

decoding we know exactly the number of bits (the length sent in the signal part), so we simply ignore the additional bits resulting from these zeros symbols.

III. Comparison of BER performance using both floating point and fixed point implementations.

To investigate the effect of limitations of resources, the following figure shows the comparison of BER performance using floating point implementation versus fixed point implementation . The used modulation scheme is 16QAM with ¾ as the code rate.

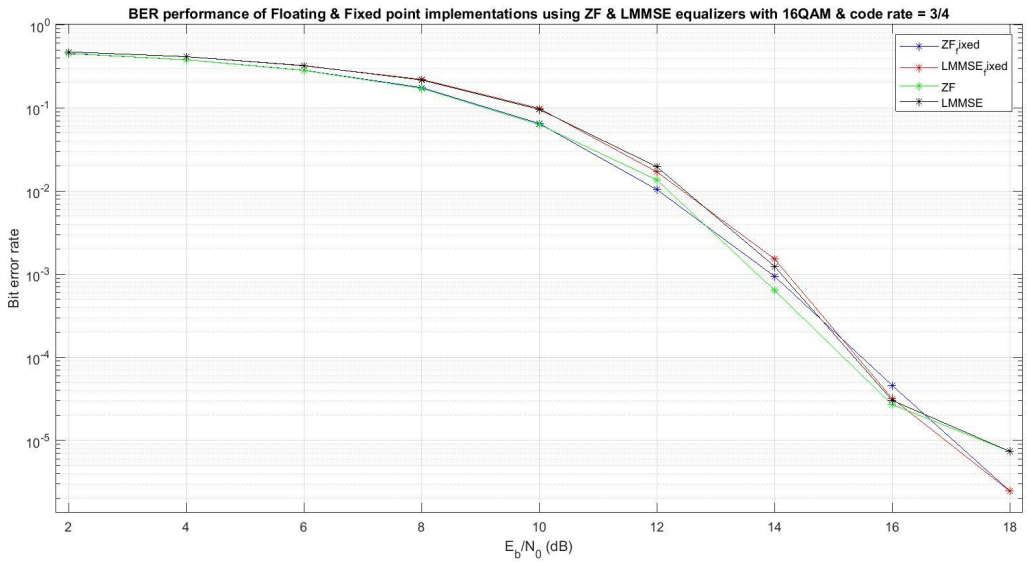

Figure 14. BER for floating and fixed point implementations

The fixed point with the chosen parameters is a representative of the floating point implementation. As, the fixed point curve lies around the floating point curve using both wiener and zero forcing equalizers. The curves do not perfectly match due to the quantization error that occurs when shifting from floating to fixed point representation.

IV. Comparison of BER performance using all the supported rates and modulation schemes in floating point implementation.

To investigate the effect of different modulation schemes and code rates and their robustness against noise, the following figure shows the BER curves for all the supported modulation schemes and code rates combinations for multiple SNR

values. The BER values were taken after running 6 experiments to smooth the curves.

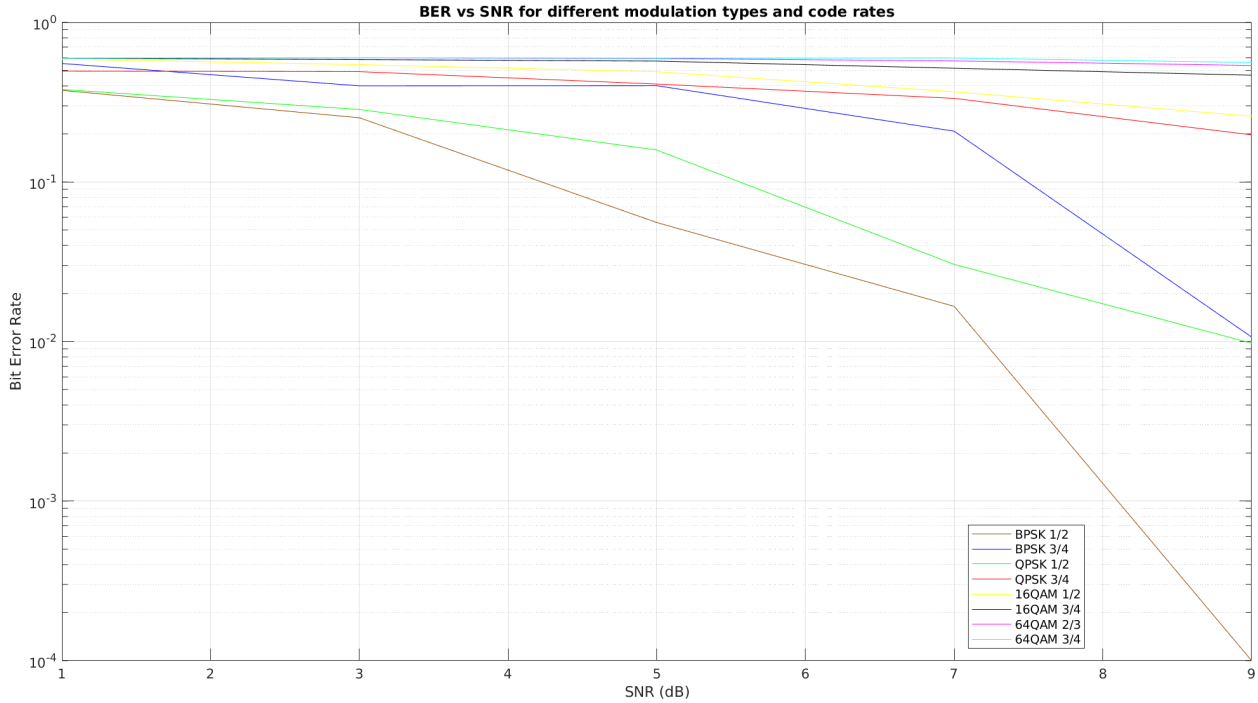

Figure 15. BER for different modulation schemes and code rates.

Just as expected, the BER curves show the variability of modulation schemes robustness against noise. The lower the curves (such as BPSK ½), the better the performance. The code rate effect is also shown by figuring that the lower rate has better performance with the same modulation scheme. This is because the lower rate means that more additional bits are added for a smaller number of information bits which is more secure.

V. Comparison of BER performance using all the supported rates and modulation schemes in fixed point implementation.

The same previous comparison is repeated again but with fixed point implementation this time. It is actually expected to get the same trend of each BER curve and the same differences between the different combinations of code rate and modulation scheme.

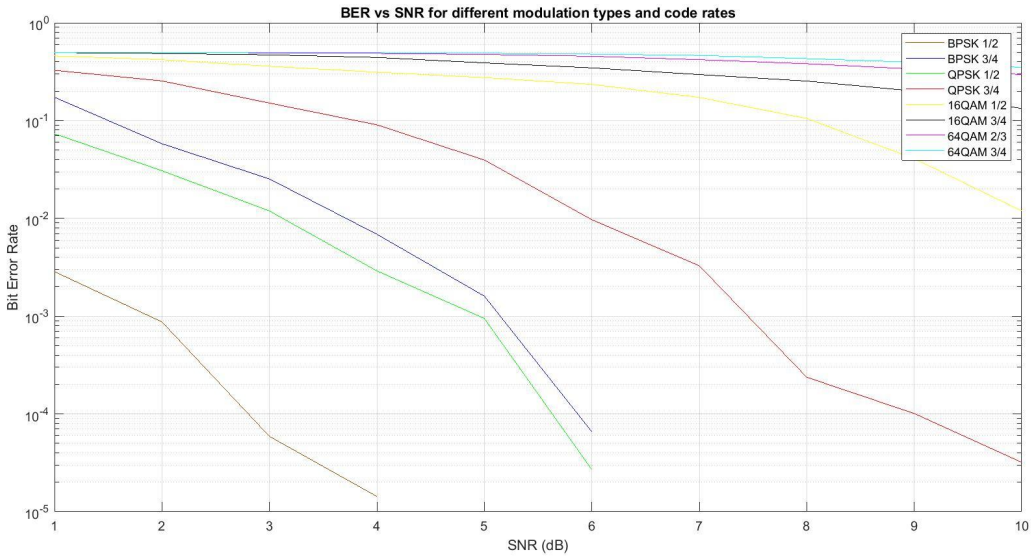

Figure 16. BER for different modulation schemes and code rates)

The results of fixed point implementation for different modulation schemes follow the same trend as the floating point implementation. However, the BER as values changes from one experiment to another due to the randomness of the noise but the trend of the curves is preserved. The above graph shows only the result of one experiment. We have to run multiple experiments in order to accurately compare performance of the fixed point and floating point representations. Figure 14 shows that for one experiment the fixed point BER may be above or below the floating point implementation. This is due to the quantization errors and lack of multiple experiments. In addition, one fixed point experiment was computationally expensive. So, it was hard to run multiple experiments and take the average BERs across all experiments.

References:

[1] E. Notes, "What is OFDM: Orthogonal Frequency Division Multiplexing," *Electronics Notes*. [Online]. Available: https://www.electronics-notes.com/articles/radio/multicarrier-modulation/ofdm-orthogonal-frequency-division-multiplexing-what-is-tutorial-basics.php. [Accessed: 14-Jun-2021].