Wifi Physical Layer Report

a)

Level 4

At this level, the only thing I did is to filter out the initial zero padding by identifying where the preamble sequence starts in our received message. I do this with the function "find_preamble". This helper function loops from index 0 to the length of the output array minus the length of the preamble sequence in complex numbers. It iterates 1 by 1 and sees if the subarray located at our current index has the highest correlation to the preamble complex array. After looping we should have stored the index that has the highest correlation. To calculate the correlation we just use NumPy's function "np. correlate". We take the absolute value of the correlation to see which might be better since the NumPy function returns a complex number. After we know where the preamble is we can filter the part of the output bits that are behind the preamble since that would just be the initial zero-padding which we do not need.

Level 3

This part was essentially just reversing the OFDM Demodulation. I would say this is the simples part of my code since all I had to do was to reverse the part of the code of level3 in the wifi transmitter and use the function np.fft.fft instead of np.fft.ifft to reverse the inverse Fourier transform done in the wifi transmitter code.

Level 2

To decode the message, I chose to use the Algorithm which is the Hard Viterbi Algorithm. I utilized commpy to demodulate our output bits with the function "demodulate" utilizing the 'hard' argument which returns a sequence of bits of 0s and 1s. Once we have those bits, I removed the preamble and thus this would also remove the initial zero-padding since the preamble was concatenated after the initial zero padding. Then, I proceed to get the length of our message.

To get the length of our message since we know that the length of the array that was concatenated to the sent message is 128, we know the length is in the first 128 bits. We also know the length is padded with a bunch of zeros. Since we know the maximum length of the message sent is 10,000 bytes we know that 2^14 (not inclusive) is the maximum length our message could have. So we can start at position 113 and start iterating until the end of the length array to figure out our length. If we see a 1 we will add the corresponding power of 2 to our message. Otherwise, don't add anything. This will give us the correct length. After obtaining the length we can remove the length bits together from the output bits array.

Since we now know the length of the message I made sure to after obtaining the length, remove the final padding even though it is not necessarily required to do. However, I thought it would be a good idea to do this since then I could just send my output array to the Viterbi solver without having to worry about not sending the padding. To remove the final padding we know that every character in the message is 8 bits, therefore there will be "length"*8 bits. And we also padded zeros to the message sent so we can compute the padded zeros just like they did in the transmitter code "2*nfft-message_bits%(2*nfft)". After that, we know the padding_zero_index should be located at 2*(message_bits*8 + padded_zeros). The reason why we multiply by 2 is that we know our generator polynomial outputs 2 bits for every input. So now we can easily filter out the padding zeros.

After that, we can get the pairs of output bits from the generator polynomial with a simple for loop that keeps track of two pointers (l, r) and appends a list of two bits (output[l], output[r]) to our generator bits array. This is what the "get_output_generator_bits" function is doing.

Once we have obtained the output bits and filtered out all the other information (padding and preamble), we can send these bits to the Viterbi algorithm. Before performing the Viterbi algorithm solution, I chose to build an array called "error_array", with a helper function called "build_error_array". Like the name suggests it calculates the error values of the output bits we received assuming that we matched it to the output of a certain state with an input of 0 or 1. We will store in this array the error values (calculated using the hamming distance between two "two bits pair" values, and this will allow us to have access to all the errors in my Viterbi algorithm without having to compute them (I would just have to look inside this array).

Now, I can call the "viterbi_solver" helper function which performs the Viterbi algorithm. For the Viterbi solver, we have two for loops, the outer for loop is for the number of generator bits and the inner loop is for the number of states. We now need to calculate the minimum error that would be obtained at each state by looking at what would be the error if we came from the different previous states to the current state as well. We store the previous errors in another array called input_tracker which has as keys the previous state and the error value of the path as the value. We can calculate the error that the current state would produce with the different inputs by getting the sum of the error at the possible previous state and the sum of the error that the current input would produce at the current state. Now we just need to check if this will be the input and previous state that will produce the least error at the current state and if so update our min_error_input and min_error_prev_state values. After trying all combinations we store in input_tracker the previous state that caused the least error at the column representing the minimum error previous state and match it to the value of the error. Notice how if we do this for all generator pairs of bits, we will essentially have built the least error path and we can just backtrack to get the best path.

After the Viterbi algorithm is computed we can backtrack. We could backtrack from 4 different states so we look at the state at the last time that has the least value of error since that will be the best path. We calculate this with the "get_state_with_min_path" helper function . This function gets the min error value from our states at the end and then loops again to check which state had that error value and returns that state. Once we have the state, we can use the "build_path" function to backtrack and build the min_error_path. This function utilizes a stack to keep track of the current path, state, and time we are at. We loop until the stack is empty and we would return a dictionary containing our path (with the key "input_bits" matched to our path) when we reach r = 0 (so time = 0) since that is how I designed the algorithm to be. Before returning the path we would have to flip it since we are appending bits to the output. To calculate the previous path we can use my input tracker array which has stored the best current previous state at the current time and state. We can find the input bit of the current state that led to the previous state by using our find_input_bit_based_on_state function (should be pretty easy since for all states it is either just 0 or 1). After that, we append the bit to the current path and store in the stack the current path bit sequence, with time (or r here) updated to be r-1, the current state would be the previous state and the input_tracker which is where I have stored all my data. This will be repeated until r = 0 (or time = 0) which will return a dictionary containing the best path. At the end of the Viterbi algorithm, we just return the path itself.

Level 1

For this level, we first check if the entered level by the user was 1. In this case, we would still have to get the length of the message and remove the length of the array since I do that in my "Level2", but level2 code would not be executed. So we have an if statement that checks if the level entered is 1 we should get the length of the message and remove the length_binary array in the same way that I described above.

After this, we can proceed to first get the message that we received from the Viterbi algorithm. To do this we need to undo the interleave algorithm performed on it. Therefore, I call my helper function "de_interleave" to make sure that this occurs. This function loops through the rows and columns of the array output, (rows being the nsym and columns being the length of the interleave array). We know that to match the output to the correct value at the current index we need to know which row index to access and which column index to access. The row index can be calculated since it would just be whatever the value of Interleave-1 is at the location of the current column we are at. Then we can just say that our output array at the current row times the row_length plus the row index is equal to the input at our current row times the row_length plus the current column. This correctly undoes the interleaving (it's essentially just reversing the algorithm).

Now that we have the correct bits, these bits represent the ASCII values of the message that got sent to us. Since we know every value will be of 8 bits because each letter sent represents a character, then we can decode this by having a for loop that groups every 8 bits into a list and then having another for loop that converts every list of 8 bits to an ASCII character. This is essentially done by my functions "restore_ascii_value" and "convert_ascii_values_to_message". After doing that we should have all the data that we need, including the begin_zero_padding value, message, and length.

b) Since we know that the length of the array that was concatenated to the sent message is 128, we know the length is in the first 128 bits. We also know the length is padded with a bunch of zeros. Since we know the maximum length of the message sent is 10,000 bytes we know that 2^14 (not inclusive) is the maximum length our message could have. So we can start at position 113 and start iterating until the end of the length array to figure out our length. If we see a 1 we will add the corresponding power of 2 to our message. Otherwise, don't add anything.