ECE 414: OFDM Receiver

Ahsen Qureshi, 20964433
Rahavan Sivaguganantha, 20945290

July 27, 2025

1 Introduction

The goal of our project is to design an OFDM receiver to demodulate a signal in a communication pipeline. As outlined in the project description, this involves a series of key components including a Serial to Parallel Converter, a Remove CP block, a DFT module, and additional downstream elements such as Channel Effects Equalization and Maximum Likelihood Detection. Figure 1 below represents our system level approach for recovering the transmitted signal accurately.

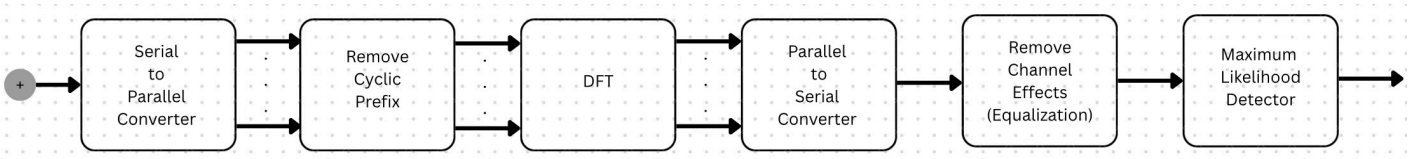

Figure 1: Block diagram of digital data for receiving end of system.

2 Receiver Design Flow

Pilot Cyclic Prefix	Pilot	OFDM Cyclic Prefix	OFDM Symbol
32 Samples	128 Samples	32 Samples	128 Samples

Table 1: Received Signal Structure

2.1 Bitstream Parallelizing for OFDM Subcarriers

The Serial to Parallel Converter takes in a continuous stream of bits and parallelizes them to match the OFDM symbol structure. This step is essential because OFDM uses frequency division to transmit multiple parallel data streams simultaneously, with each sub-stream mapped onto a separate orthonormal basis. The code below extracts the received signal in the .csv file into a signal structure that aligns with the multicarrier nature of the system.

```
signal = received_signal[:, 0] + (1j * received_signal[:, 1])
```

Since the .csv file is divided into separate columns for the real and imaginary parts of the received signal, we are able to combine them both to obtain the complex time-domain representation of the signal, which is required for further processing.

2.2 Pilot Handling and Removing the Cyclic Prefix

At the transmitter, a pilot signal is transmitted along with the OFDM symbol to assist the receiver in estimating the channel behaviour. The pilot signal is a mutually agreed upon signal between the transmitter and receiver entities, that

is used to calculate the effects that the channel has on the transmitted signal, as seen by the receiver. As we saw in Table 1, the pilot signal and OFDM symbol both also have cyclic prefixes that come before them.

At the transmitter, a cyclic prefix is included to both the pilot and OFDM symbol to help combat inter-symbol interference (ISI) caused by multipath propagation, to help preserve the integrity of the data. We know that ISI occurs when a signal reflects off surfaces and arrives at the receiver at slightly different times, resulting in overlap with the next symbol, and ultimately leading to distortion. The cyclic prefix helps prevent this by copying the end portion of the symbol and placing it at the beginning, creating a buffer zone.

As per Table 1, our signal structure is arranged such that the pilot signal and its cyclic prefix, come before the OFDM symbol and its cyclic prefix. Accordingly, our receiver design accounts for this by processing the signal in sequential order. The code below shows the sequential order of how this is handled, beginning with the pilot's cyclic prefix being removed, followed by the capturing of the pilot signal, and then finally the removal of the OFDM symbol's cyclic prefix.

```
# Pilot Cyclic Prefix Removal

    pilot_cyclic_prefix = np.arange(0,32)
    removed_pilot_cyclic_prefix_signal = np.delete(signal, pilot_cyclic_prefix)

# Capture of Pilot Signal

    pilot = removed_pilot_cyclic_prefix_signal[0:128].copy()
    pilot_samples = np.arange(0,128)
    removed_pilot_signal = np.delete(removed_pilot_cyclic_prefix_signal, pilot_samples)

# OFDM Symbol Cyclic Prefix Removal

    OFDM_signal_cyclic_prefix = np.arange(0,32)
    OFDM_signal = np.delete(removed_pilot_signal, OFDM_signal_cyclic_prefix)
```

This ensures that the pilot signal is isolated and captured, so that it can be used for channel equalization, and that the OFDM symbol is extracted for demodulation and decoding.

2.3 Frequency Domain Transformation

The Discrete Fourier Transform (DFT) converts the received digital baseband time domain signal into its frequency domain representation, allowing each subcarrier in the OFDM symbol to be individually processed. Here, the pilot signal also needs to be converted into the frequency-domain before we begin to analyze the channel effects. The code below performs this conversion by applying the DFT (using the Fast Fourier Transform method) to both the pilot signal and OFDM symbol individually.

```
dft_pilot = np.fft.fft(pilot)

dft_OFDM_signal = np.fft.fft(OFDM_signal)
```

2.4 Removing Channel Effects

To accurately recover the transmitted data, it is necessary to account for distortions and AWGN introduced by the communication channel. To account for this, we perform Zero-Forcing (ZF) channel equalization, where we divide the received signal in the frequency domain by an estimate of the channel response. This channel estimate, denoted as $\hat{H}[q]$, is obtained using a known reference pilot signal, $x_p[q]$. Since both the received pilot and the reference pilot are in the frequency domain, dividing them gives us $\hat{H}[q] = \frac{\text{received DFT } x_p[q]}{\text{reference } x_p[q]}$, which is used as the equalization factor.

Earlier, we recovered the pilot signal, and had converted it into its frequency domain representation using the DFT. We are also aware of what the reference pilot signal should be in the frequency domain. Thus, we can directly calculate for $\hat{H}[q] = \frac{\text{received DFT } x_p[q]}{\text{reference } x_p[q]}$. The code below does this by constructing $x_p[q]$ and then solving $\hat{H}[q] = \frac{\text{received DFT } x_p[q]}{\text{reference } x_p[q]}$ with the recovered pilot signal.

```
pilot_4_symbols = [complex(1,1), complex(-1,1), complex(-1,-1), complex(1,-1)]

PILOT_REF = np.tile(pilot_4_symbols, 32)

H_hat = dft_pilot/PILOT_REF
```

Now that we have $\hat{H}[q]$, we can apply it to the received OFDM symbol by dividing the frequency domain representation of the OFDM symbol by $\hat{H}[q]$. In the code below, we divide the frequency domain representation of the OFDM symbol by $\hat{H}[q]$, effectively removing the channel's influence and recovering the originally transmitted frequency domain data, with only the effects of AWGN remaining.

```
equalized_OFDM_signal = dft_OFDM_signal/H_hat
```

2.5 Symbol Detection using Maximum Likelihood

After channel equalization, the received signal is still affected by residual noise, typically modeled as Additive White Gaussian Noise (AWGN). This noise causes the received symbols to deviate slightly from their ideal positions in the constellation map. To recover the original transmitted data, we use a Maximum Likelihood Estimate (MLE), which selects the constellation point closest to each received symbol. In the case of 16-QAM, this involves comparing the noisy symbol to all possible constellation points and choosing the one with the smallest Euclidean distance. This approach minimizes the probability of symbol error under AWGN and allows us to map the received signal back to the correct 4-bit binary sequence associated with each 16-QAM symbol.

The code below shows the calculation of the closest constellation point to the received symbol, using Euclidian distance.

```
def nearest_constellation_symbol(symbol):
    minimum = math.inf
    point = None
    for coordinate in QAM_16_map.keys():
        distance = math.sqrt((symbol.real - coordinate.real)**2 + (symbol.imag - coordinate.imag)**2)
        if(distance < minimum):
            minimum = distance
            point = coordinate

    return point
```

The code below shows the complete AWGN handling for constellation mapping, using the Maximum Likelihood Estimate.

```
demodulated_OFDM_signal = []

for symbol in equalized_OFDM_signal:
    constellation_symbol = nearest_constellation_symbol(symbol)
    demodulated_OFDM_signal.append(QAM_16_map[constellation_symbol])
```

Once the nearest constellation symbol is identified, the corresponding 4-bit sequence to that constellation symbol is identified and appended to a list to recover the transmitted bitstream.

2.6 Message Decoding and Formation

Now, we have retrieved the 4-bit sequences for all the OFDM symbol samples, and can begin to decode the transmitted message. Since we know that the message is a set of ASCII characters, we know each character is 8 bits long. Currently, the bit sequences are 4 bits long each, so we will have to form 8-bit sequences by joining every two 4-bit sequences. The code below shows the joining of every two 4-bit sequences to make up the 8-bit representations.

```
combined_results = []
for i in range(0, len(demodulated_OFDM_signal)-1, 2):
    ascii_binary = demodulated_OFDM_signal[i] + demodulated_OFDM_signal[i+1]
    combined_results.append(ascii_binary)
```

Now we have the 8-bit representation of each ASCII character. Hence, we can convert the bit representation into their decimal representations, and retrieve the corresponding ASCII character. If we repeat this for every 8-bit sequence, we will recover the entire OFDM symbol that was . The code below does this conversion, and builds the recovered message.

```
message = ""
for ascii_binary in combined_results:
    ascii_decimal = int(ascii_binary, 2)
    character = chr(ascii_decimal)
    message += character
```

3 Message

Using our receiver-side demodulator, we were able to recover the following message from the received signal:

Why can't you ever trust atoms? Because they make up everything.

References

[1] TutorialsPoint. *NumPy - Reading data from files*. Available at: https://www.tutorialspoint.com/numpy/numpy_reading_data_from_files.htm.

[2] TutorialsPoint. *NumPy - Array creation routines*. Available at: https://www.tutorialspoint.com/numpy/numpy_array_creation_routines.htm.

[3] TutorialsPoint. *NumPy - Indexing and Slicing*. Available at: https://www.tutorialspoint.com/numpy/numpy_indexing_and_slicing.htm.

[4] TutorialsPoint. *NumPy - tile() function*. Available at: https://www.tutorialspoint.com/numpy/numpy_tile_function.htm.

[5] TutorialsPoint. *NumPy - Discrete Fourier Transform*. Available at: https://www.tutorialspoint.com/numpy/numpy_discrete_fourier_transform.htm.

[6] W3Schools. *Python math Module*. Available at: https://www.w3schools.com/python/module_math.asp.