End-to-End OFDM Simulation in Python for a SISO Communication System

Shahrokh Hamidi
Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario, Canada
Email: shahrokh.hamidi@uwaterloo.ca

I. INTRODUCTION

In this article, an end-to-end Orthogonal Frequency Division Multiplexing (OFDM) simulation in Python for a Single Input Single Output (SISO) communication system is presented. The project has been coded in Python 3 using Object Oriented Programming (OOP) and has been run on Windows and Linux successfully.

The code can be downloaded from "https://github.com/Shahrokh-Hamidi/OFDM-End-to-End-Simulation".

Fig. 1 illustrates the block diagram of the system.

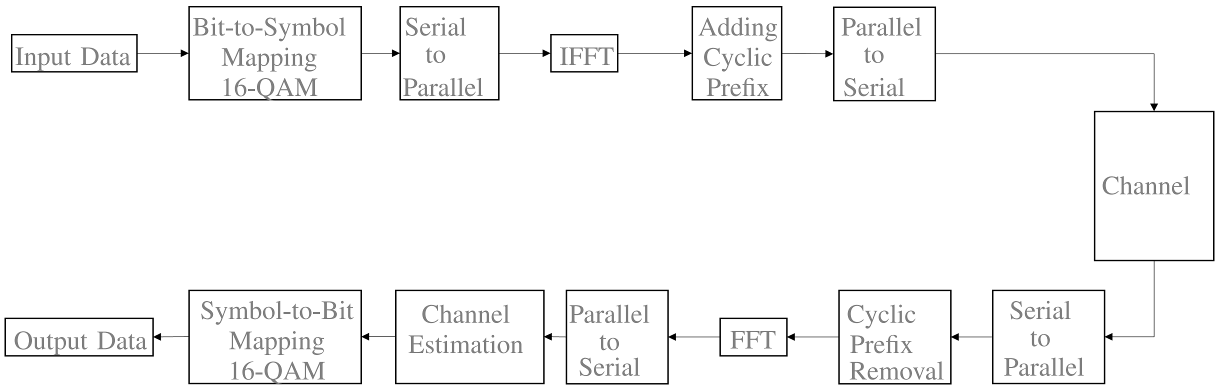

Fig. 1. The block diagram for the OFDM system.

II. CLASSES

In this section, the codes for different classes, which have been used to simulate the project, are presented and their contents are explained.

A. *dataGen*

Class dataGen has been written to generate the input bits to be fed to the system. The function DATA_generation from dataGen class generates random bits based on Bernoulli probability density function (pdf). In the code Binomial pdf has been used. However, it should be noted that the Binomial pdf for $n = 1$ is Bernoulli pdf. In addition, the probability for 0 and 1 has been set to $\%50$.

```
class dataGen:

    def __init__(self):

        pass
    @staticmethod
    def DATA_generation(len):
        return np.random.binomial(n = 1, p = 0.5, size = (len, ))
```

B. Mapping

After data generation, bit-to-symbol mapping is performed. For this simulation we have chosen $16-\mathrm{QAM}$ as the modulation technique. The class $\mathrm{Mapping}$ performs the bit-to-symbol as well as symbol-to-bit mapping based on $16-\mathrm{QAM}$.

```
class Mapping:

    def __init__(self):
        pass


    @staticmethod
    def map_bit2symb():

        dic_ =  {
            (0,0,0,0) : -3-3j,
            (0,0,0,1) : -3-1j,
            (0,0,1,0) : -3+3j,
            (0,0,1,1) : -3+1j,
            (0,1,0,0) : -1-3j,
            (0,1,0,1) : -1-1j,
            (0,1,1,0) : -1+3j,
            (0,1,1,1) : -1+1j,
            (1,0,0,0) :  3-3j,
            (1,0,0,1) :  3-1j,
            (1,0,1,0) :  3+3j,
            (1,0,1,1) :  3+1j,
            (1,1,0,0) :  1-3j,
            (1,1,0,1) :  1-1j,
            (1,1,1,0) :  1+3j,
            (1,1,1,1) :  1+1j
        }

        return dic_

    @staticmethod
    def map_symb2bit():

        dic_ = {symb: bit for bit, symb in Mapping.map_bit2symb().items()}
        return dic_
```

C. Channel

Next we present the $\mathrm{Channel}$ class. The function $\mathrm{channel\_}$ uses convolution method to apply the channel impulse response h to the input data and generate the effect of the wireless channel.

Function $\mathrm{channel\_estimation}$ has been written to perform channel estimation.

The next function in this class is $\mathrm{channel\_estimation\_interpolation}$ which is responsible for interpolation to present better visualization for the channel impulse response that is estimated based on $\mathrm{channel\_estimation}$. Finally, function $\mathrm{channel\_impulse\_response\_visualization}$ plots the estimated impulse response for the channel.

```
class Channel:

    def __init__(self):
        pass


    def channel_(self, data, h):

        return np.convolve(data, h, mode = 'same')


    def channel_estimation(self, data):

        pilot_value_est = data[pilot_index]
        return pilot_value_est/pilot_value


    def channel_estimation_interpolation(self, data):

        h_amp = scipy.interpolate.interp1d(pilot_index, abs(data), kind='cubic')(
                                                subcarrier_index)
        h_phase = scipy.interpolate.interp1d(pilot_index, np.angle(data), kind='cubic')(
                                                subcarrier_index)

        return h_amp*np.exp(1j*h_phase)



    def channel_impulse_response_visualization(self, data):

        H = np.fft.fft(channel_impulse_response, num_subcarriers)
        plt.plot(abs(data), 'k', label = 'estimated impulse response', linewidth = 2)
        plt.plot(abs(H), 'r', label = 'true impulse response', linewidth = 2)
        plt.xlabel('sub-carrier index', fontsize = 16)
        plt.ylabel('|H(f)|', fontsize = 16)
        plt.title('channel impulse response', fontsize = 14)
        plt.legend()
        matplotlib.rc('xtick', labelsize=16)
        matplotlib.rc('ytick', labelsize=16)

        plt.grid()
        plt.show()
```

D. Receiver

The code for the Receiver class is presented. The function add_receiver_noise adds Gaussian noise to the data received at the receiver. This will create the effect of the additive white Gaussian noise channel.

```
class Receiver:

    def __init__(self):

        pass



    @staticmethod
    def add_receiver_noise(data, snr):


        signal_power = np.mean(abs(data**2))
        snr = 10**(snr/10)

        std = np.sqrt(signal_power/snr)
        noise_of_receiver = np.random.normal(0, std, data.shape[0]) + 1j*np.random.normal(0,
                                                             std, data.shape[0])

        return data + (1/np.sqrt(2))*noise_of_receiver
```

E. Detector

In this subsection, the code for the Detector class is presented. The function subcarrier_data_estimation has been written for data estimation.

In this function Zero Forcing, Matched Filtering, and MMSE have been implemented and user can choose one of these methods for the data estimation process.

The function constellation_vis plots the constellation for the estimated data.

Finally, function maximum_likelihood_estimator is the implementation of the maximum likelihood estimator.

```
class Detector:

    def __init__(self):

        pass


    def subcarrier_data_estimation(self, data, h, method):


        if method == 'zero forcing':
            return (data/h)[subcarrier_carrying_data_index]

        if method == 'matched filtering':
            return (np.conj(h)*data)[subcarrier_carrying_data_index]

        if method == 'MMSE':

            H = np.conj(h)/(abs(h)**2 + 0.001)
            return (data*H)[subcarrier_carrying_data_index]



    def constellation_vis(self, data, symbols_info):
        plt.figure(facecolor='black')
        plt.plot(np.real(data), np.imag(data), 'r.', label = 'received signal')
        plt.plot(np.real(symbols_info), np.imag(symbols_info), 'y*', label = 'original signal
                                                        ')
        ax = plt.gca()
        ax.set_facecolor('black')
        ax.xaxis.label.set_color('white')
        ax.yaxis.label.set_color('white')
        plt.xlabel('Real', fontsize = 16)
        plt.ylabel('Imag', fontsize = 16)
        ax.tick_params(axis='x', colors='w')
        ax.tick_params(axis='y', colors='w')
        plt.title(f'   16-QAM     {num_subcarriers} sub-carriers', color = 'w')
        plt.xlim(-6,6)
        plt.ylim(-6,6)
        plt.grid(alpha = 0.3)
        plt.legend()
        plt.show()



    def maximum_likelihood_estimator(self, symbols_estimation, symbols_info):
        estimated_value = []
        for idx in range(symbols_estimation.shape[0]):
            I = np.argmin(abs(symbols_estimation[idx] - symbols_info))
            estimated_value.append(symbols_info[I])


        return np.array(estimated_value)
```

F. Utils

Class Utils contains multiple functions that are used for the simulation. The input of the system is a stream of bits that they arrive in series. In order to feed these bits to the IFFT block serial-to-parallel block which has been implemented in function Serial_to_Parallel is used.

The function map_bit_to_symb maps the bits to the corresponding symbols based on the modulation technique used in the simulation. The function map_symb_to_bit does the reverse on the receiver side.

Functions apply_ifft and apply_fft apply IFFT and FFT transformations to the data, respectively.

Moreover, function adding_CP adds cyclic prefix on the transmit side and and function remove_CP removes the cyclic prefix on the receiver side.

```
class Utils(Mapping):

    def __init__(self):

        pass


    def Serial_to_Parallel(self, data):
        return data.reshape(-1, num_bits_per_symb)


    def map_bit_to_symb(self, data):

        return np.array([Mapping.map_bit2symb()[tuple(v)]  for v in data])


    def map_symb_to_bit(self, data):

        return np.array([Mapping.map_symb2bit()[v]  for v in data])



    def apply_ifft(self, data):

        return np.fft.ifft(data)


    def adding_CP(self, data, CP):

        return np.hstack((data[-CP:], data))


    def remove_CP(self, data):

        return data[num_cyclic_prefix: num_cyclic_prefix + num_subcarriers]


    def apply_fft(self, data):

        return np.fft.fft(data)
```

III. Main Function

In this section, the code for the main part of the project is presented and different components are explained one by one.

```
if __name__ == '__main__':

    len_data = 476
    num_subcarriers = 128
    num_pilots = 8
    num_bits_per_symb = 4
    num_cyclic_prefix = int(num_subcarriers/4)
    channel_impulse_response = np.array([1, 1j, 0.1 + 0.5j, 0.2 + 0.2j])
    snr = 30 #db


    subcarrier_index = np.arange(0,num_subcarriers)
    pilot_index = subcarrier_index[::int(num_subcarriers/num_pilots)].tolist()
    pilot_index.append(subcarrier_index[-1])
    num_pilots += 1
    pilot_index = np.array(pilot_index)
    subcarrier_carrying_data_index = np.delete(subcarrier_index, pilot_index)


    pilot_value = 3 + 3j

    utils = Utils()
    channel = Channel()
    detector = Detector()
    equalizer = 'MMSE'    #'matched filtering'    'zero forcing'

    bits_of_info_serial = dataGen.DATA_generation(len_data)


    bits_of_info_parallel = utils.Serial_to_Parallel(bits_of_info_serial)

    symbols_info = utils.map_bit_to_symb(bits_of_info_parallel)
    subcarrier_data_pilot_frequency = np.zeros(num_subcarriers, dtype = complex)
    subcarrier_data_pilot_frequency[subcarrier_carrying_data_index] = symbols_info
    subcarrier_data_pilot_frequency[pilot_index] = pilot_value

    subcarrier_data_pilot_time = utils.apply_ifft(subcarrier_data_pilot_frequency)

    subcarrier_data_pilot_time_CP = utils.adding_CP(subcarrier_data_pilot_time,
                                                   num_cyclic_prefix)

    subcarrier_data_pilot_time_CP_channel = channel.channel_(subcarrier_data_pilot_time_CP,
                                                   channel_impulse_response)

    subcarrier_data_pilot_time_CP_channel_receiver = Receiver.add_receiver_noise(
                                                   subcarrier_data_pilot_time_CP_channel, snr
                                                   )

    subcarrier_data_pilot_time_rx = utils.remove_CP(
                                                   subcarrier_data_pilot_time_CP_channel_receiver
                                                   )

    subcarrier_data_pilot_frequency_rx = utils.apply_fft(subcarrier_data_pilot_time_rx)

    channel_impulse_response_est = channel.channel_estimation(
                                                   subcarrier_data_pilot_frequency_rx)

    channel_impulse_response_est = channel.channel_estimation_interpolation(
                                                   channel_impulse_response_est)
    channel.channel_impulse_response_visualization(channel_impulse_response_est)

    symbols_estimation = detector.subcarrier_data_estimation(
                                        subcarrier_data_pilot_frequency_rx,
                                        channel_impulse_response_est, equalizer) #

    detector.constellation_vis(symbols_estimation, symbols_info)

    symbols_estimation = detector.maximum_likelihood_estimator(symbols_estimation,
                                        symbols_info)

    #detector.constellation_vis(symbols_estimation, symbols_info)


    bits_estimation = utils.map_symb_to_bit(symbols_estimation).reshape(-1)
```

The data length has been set to $\text{len\_data} = 476$. The number of sub-carriers is equal to $\text{num\_subcarriers} = 128$. In order to estimate the channel pilot signals are used which their number is set as $\text{num\_pilots} = 8$. The value for the pilot signal has been chosen as $\text{pilot\_value} = 3 + 3\text{j}$.

Since $16 - QAM$ has been used for this simulation, therefore, the number of bits per symbol is $\text{num\_bits\_per\_symb} = 4$. The number of cyclic prefix is set to $\%25$ of the number of sub-carriers $\text{num\_cyclic\_prefix} = \text{num\_subcarriers}/4$.

All 3 well-known equalizers, namely MMSE, matchedfiltering, and zeroforcing have been implemented and the user can select the right approach based on the SNR level chosen for the simulation.

IV. Results

In the following example the impulse response of the channel has been set as an FIR system with 2 taps: $\text{h} = \{1.0, \ 0.1\text{j}\}$.

The SNR is 30 dB and the constellation is based on 16-QAM.

For this simulation the MMSE estimator has been used. It should be noted that, for large SNR the MMSE reduces to the zero-forcing and in the case of low SNR it approaches the matched-filtering equalizer.

The number of sub-carriers is 128 out of which 9 of them are pilot signals that have been used to estimate the channel. In Fig. 2, the result for the simulation has been presented. Fig. 2-(a) shows the true as well as the estimated channel impulse response. In Fig. 2-(b), the constellation for the true and estimated signals are shown.

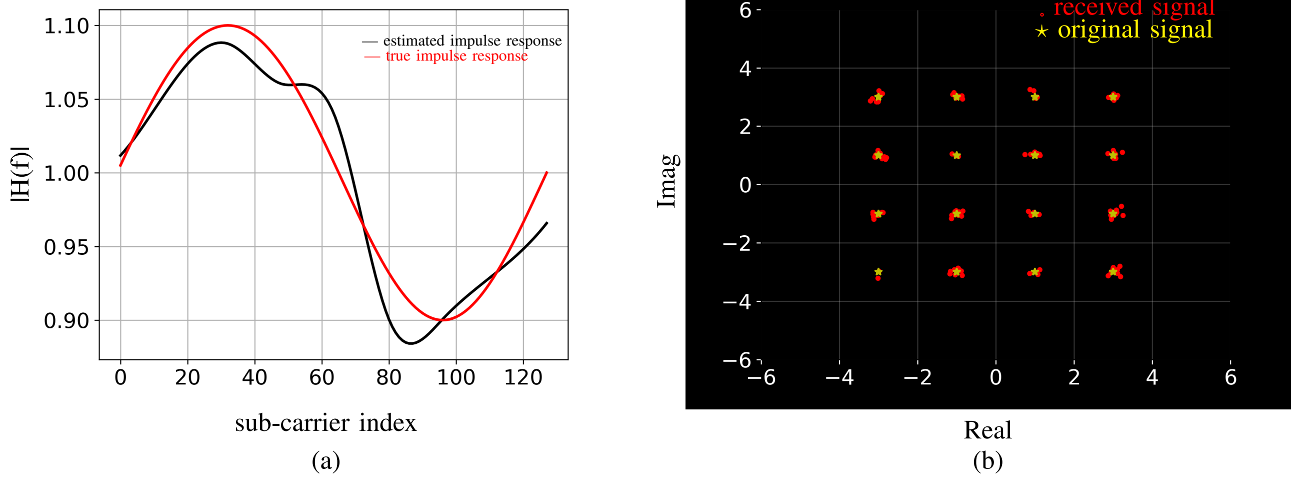

Fig. 2. a) the estimated and true channel impulse response, b) the constellation for the true and reconstructed data.

In the second simulation the channel impulse response has been set to $\mathrm{h} = \{1.0,\ 1.0\mathrm{j},\ 0.1+0.5\mathrm{j},\ 0.2+0.2\mathrm{j}\}$. the other parameters have are similar to the previous simulation. The result for this simulation is presented in Fig. 3. Fig. 3-(a) illustrates the true as well as the estimated impulse response for the wireless channel. Fig. 3-(b) depicts the constellation for the true and estimated signals.

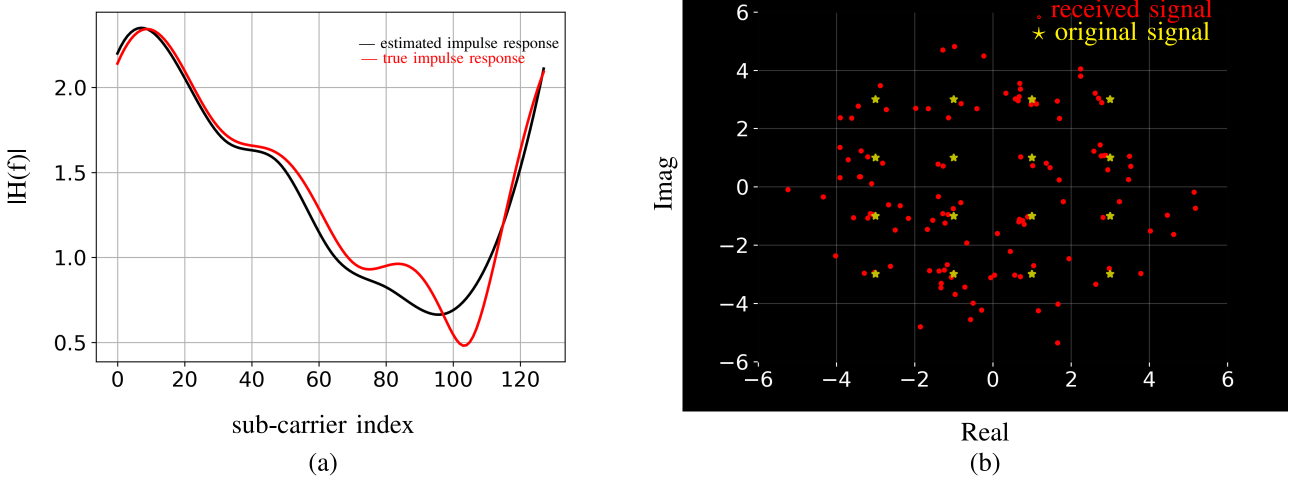

Fig. 3. a) the estimated and true channel impulse response, b) the constellation for the true and reconstructed data.

Shahrokh Hamidi was born in Iran, in 1983. He received his B.Sc., M.Sc., and Ph.D. degrees all in Electrical and Computer Engineering. He is with the faculty of Electrical and Computer Engineering at the University of Waterloo, Waterloo, Ontario, Canada. His current research areas include statistical signal processing, mmWave imaging, Terahertz imaging, image processing, system design, multi-target tracking, wireless and digital communication systems, machine learning, optimization, and array processing.