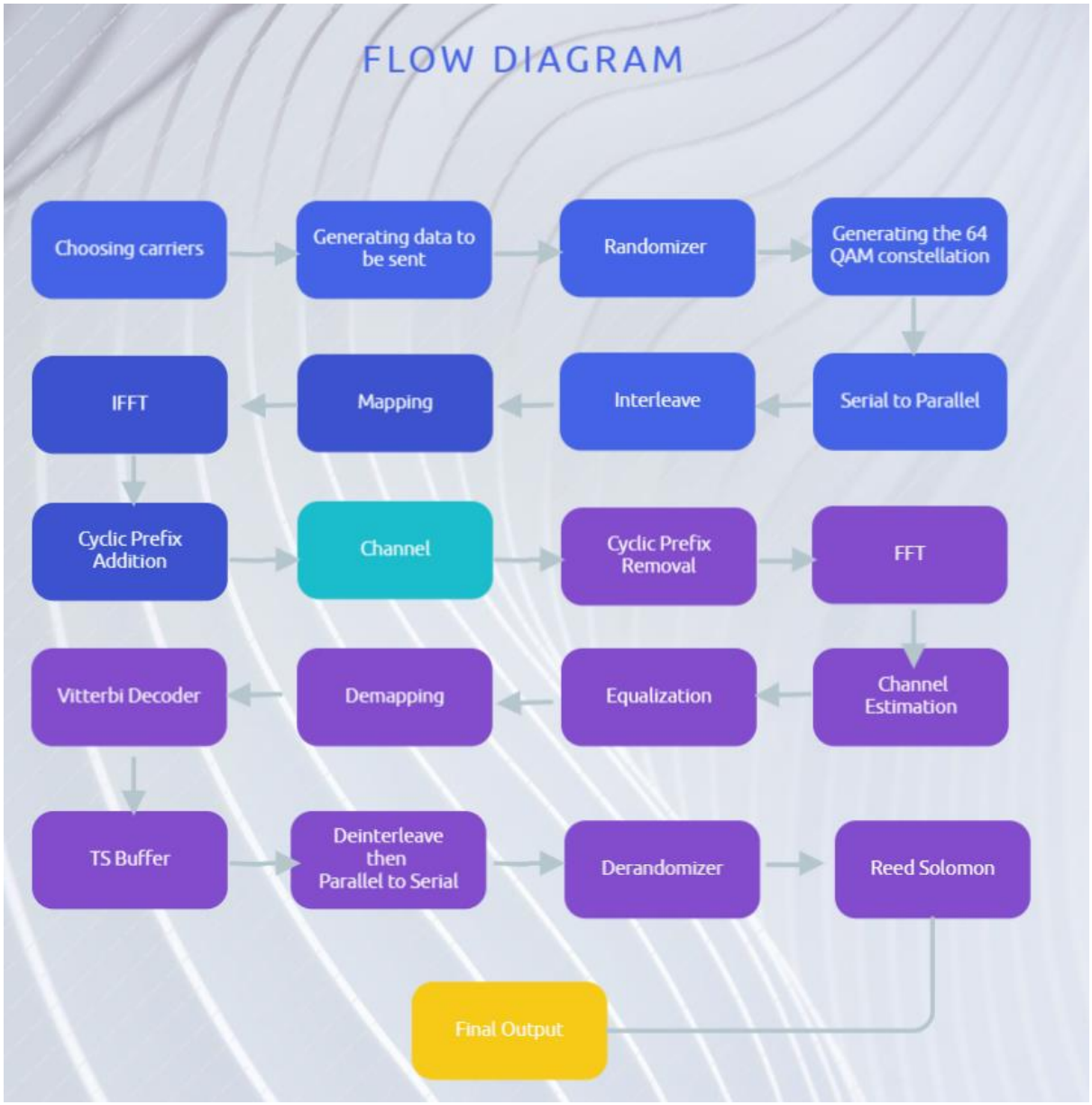
Comprehensive Modern Communication System in Python

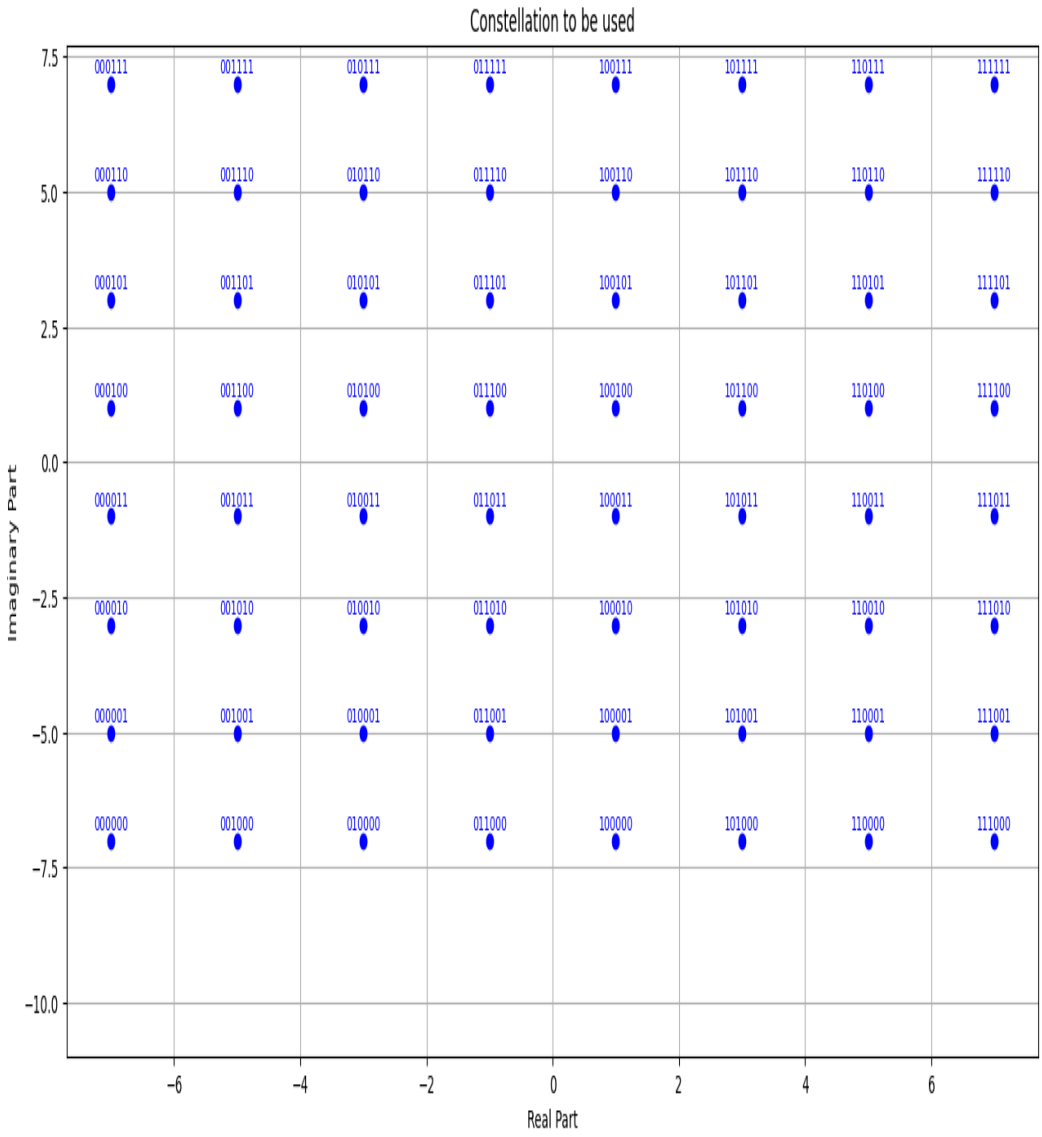
REPORT

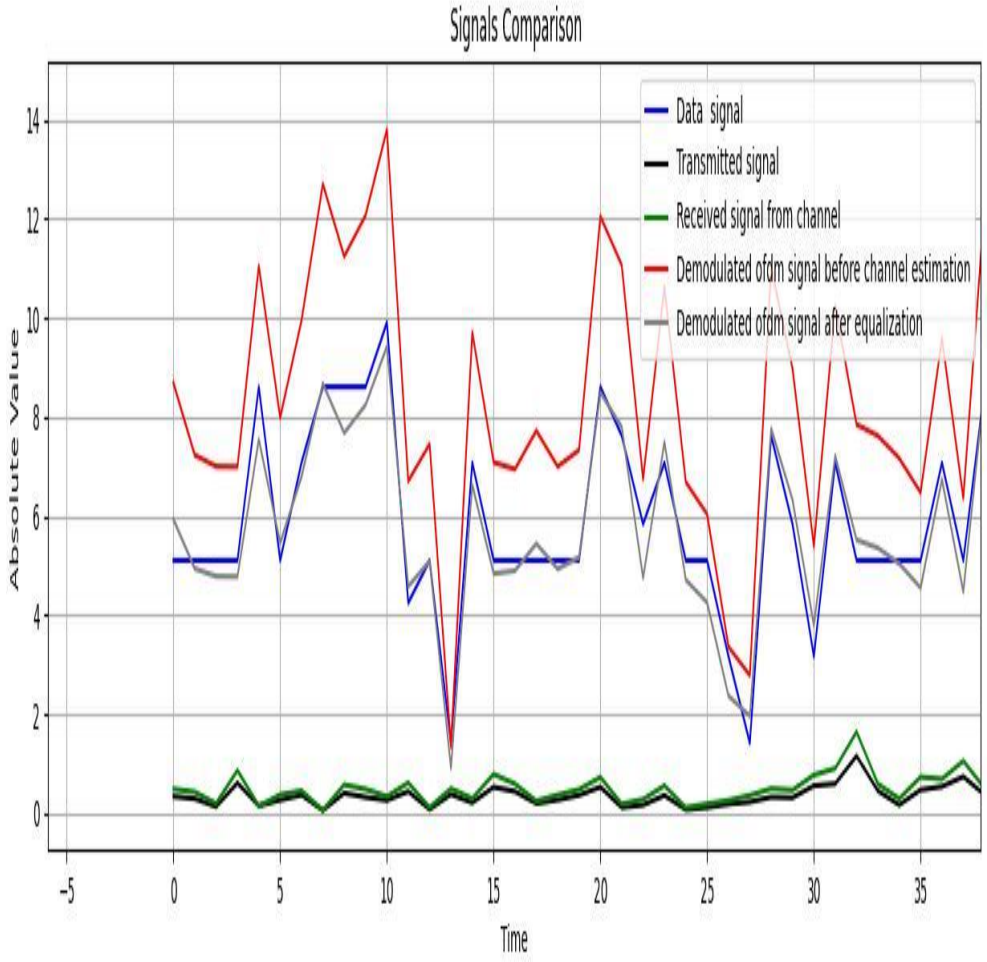
Anshuman Kumar **Email:Anshuman.Kumar@iiitb.ac.in**

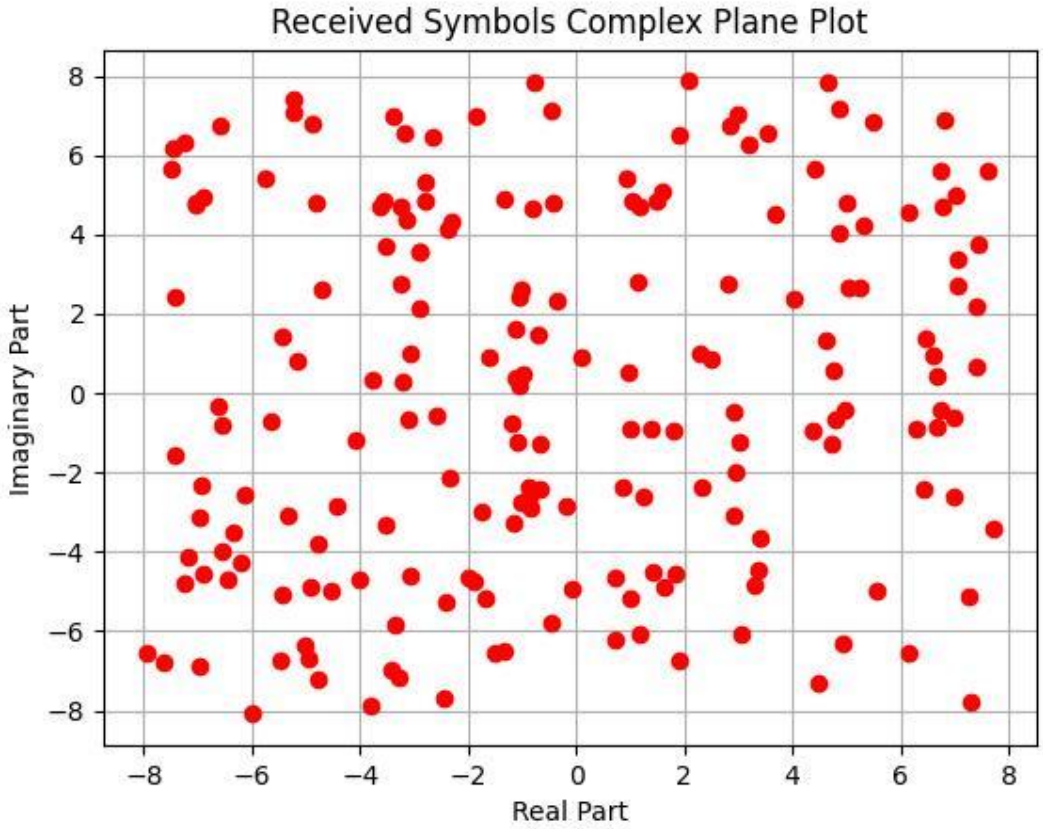
FLOWCHART

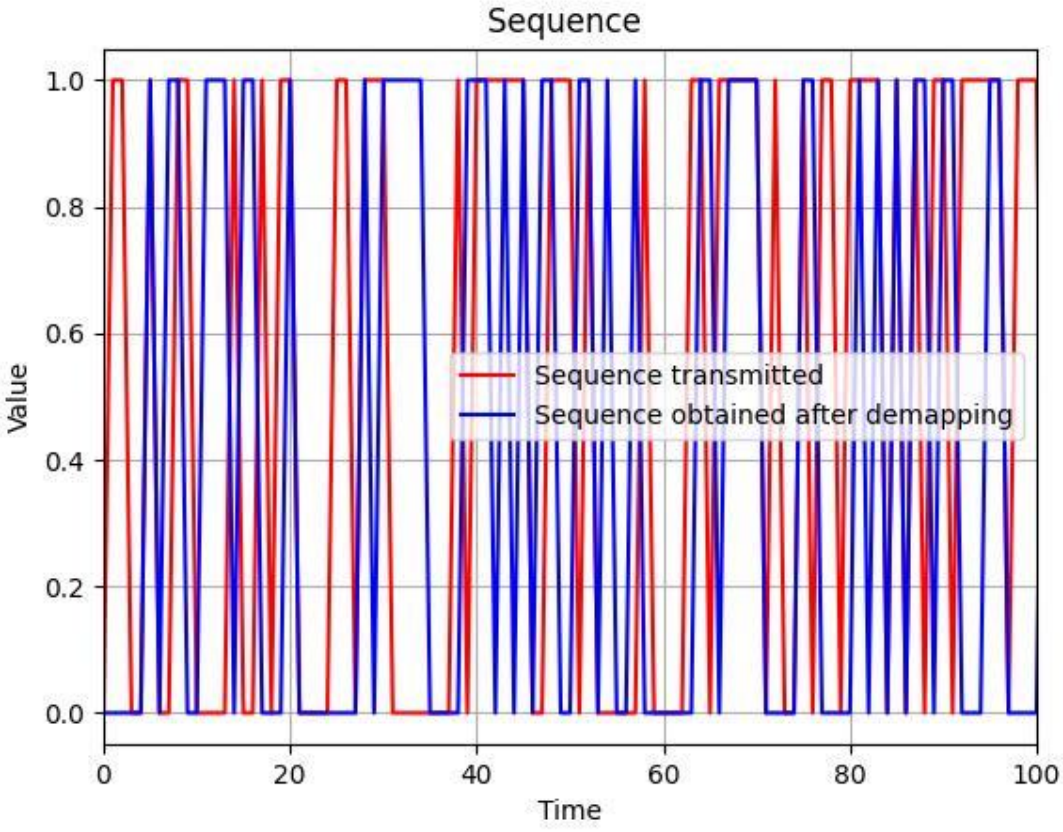


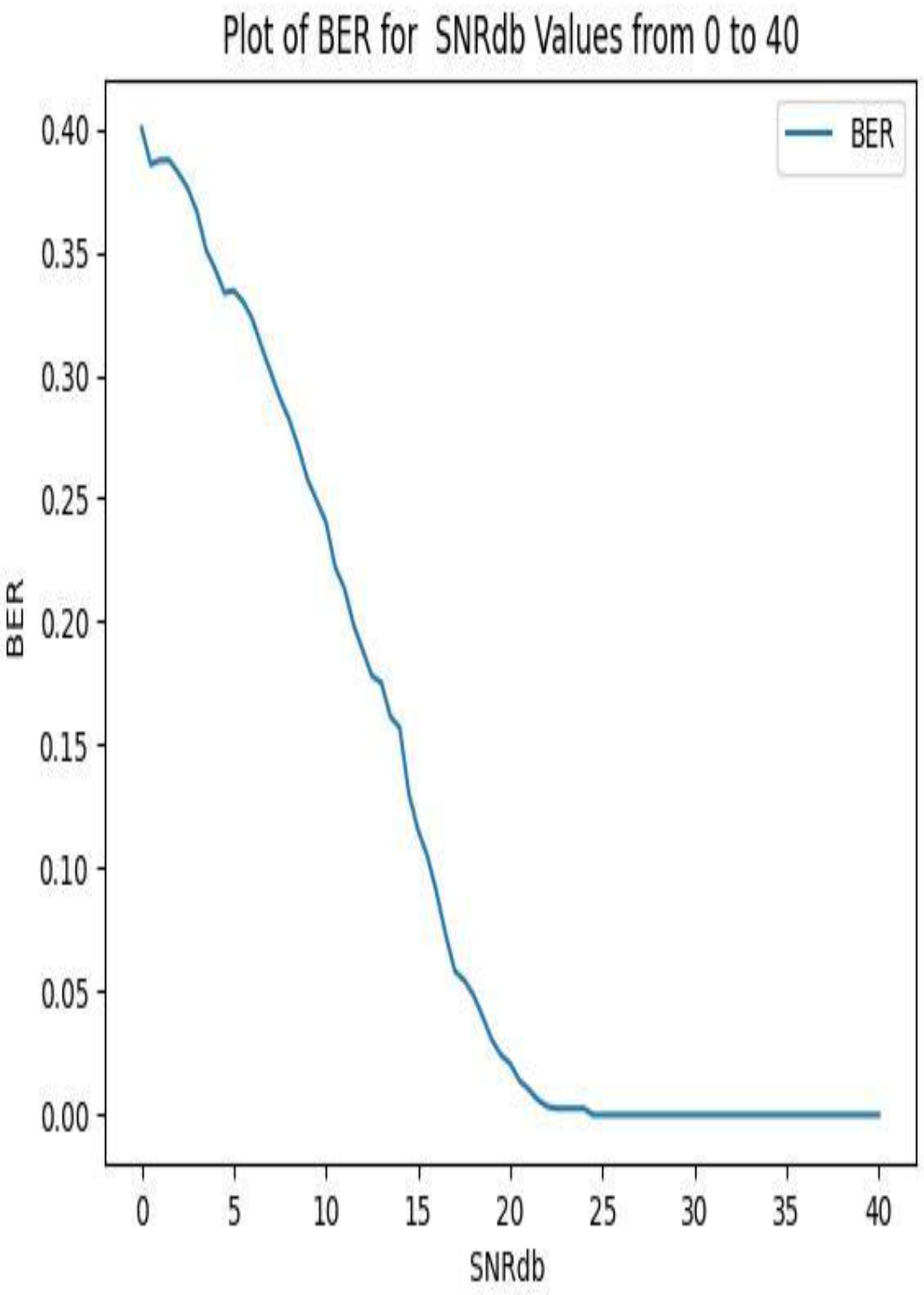
PLOTS:

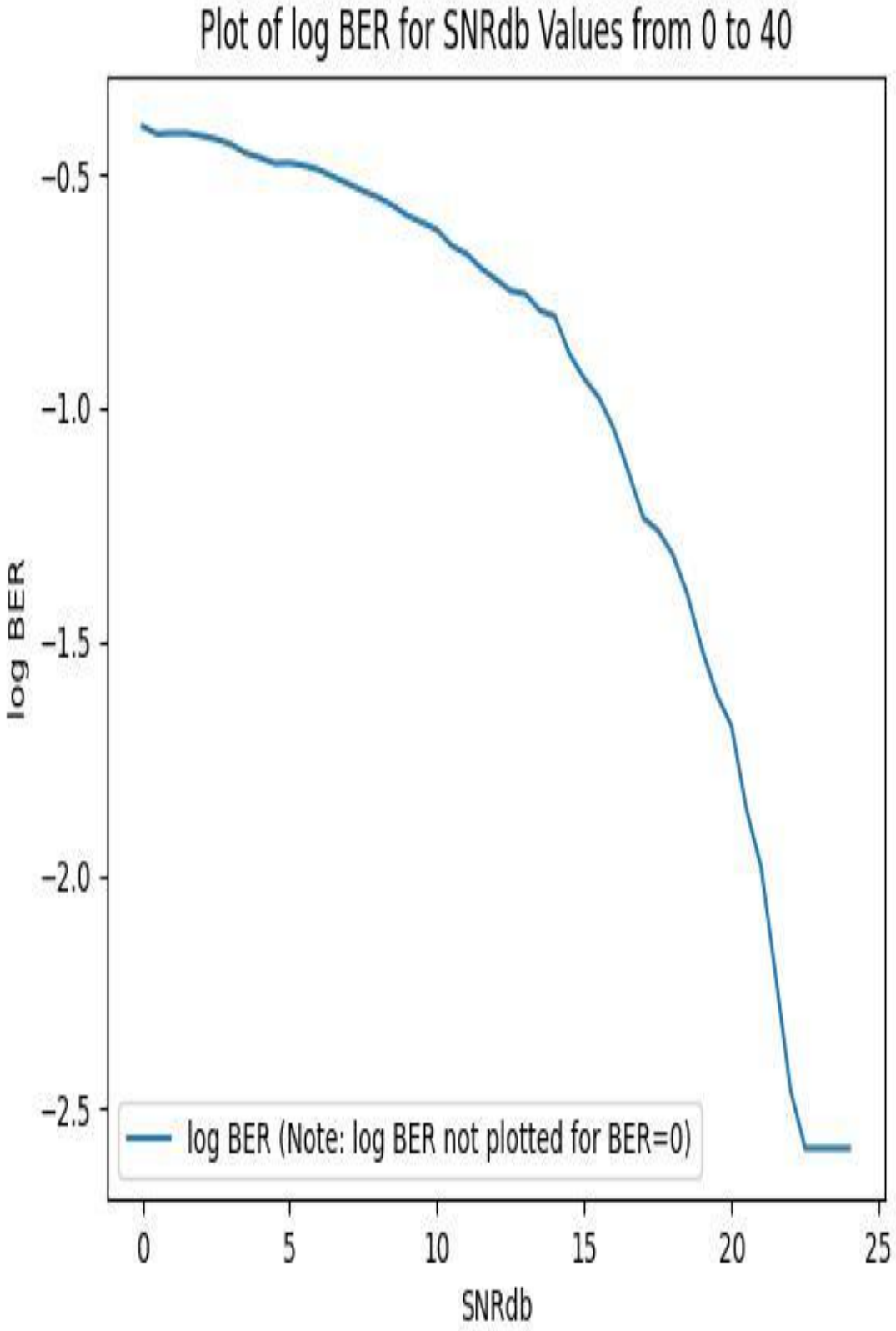

Constellation to be used
000111 001111 010111 011111 100111 101111 110111 111111
000110 001110 010110 011110 100110 101110 110110 111110
000101 001101 010101 011101 100101 101101 110101 111101
000100 001100 010100 011100 100100 101100 110100 111100
000011 001011 010011 011011 100011 101011 110011 111011
000010 001010 010010 011010 100010 101010 110010 111010
000001 001001 010001 011001 100001 101001 110001 111001
000000 001000 010000 011000 100000 101000 110000 111000
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
Imaginary Part
−6
−4
−2
0
2
4
6
Real Part


Signals Comparison
Data signal
Transmitted signal
Received signal from channel
Demodulated ofdm signal before channel estimation
Demodulated ofdm signal after equalization
Absolute Value
Time
14
12
10
8
6
4
2
0
-5
0
5
10
15
20
25
30
35


Received Symbols Complex Plane Plot
Imaginary Part
Real Part
8
6
4
2
0
−2
−4
−6
−8
−8
−6
−4
−2
0
2
4
6
8


Sequence
Value
Time
Sequence transmitted
Sequence obtained after demapping
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100


Plot of BER for SNRdb Values from 0 to 40
BER
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
BER
0
5
10
15
20
25
30
35
40
SNRdb


Plot of log BER for SNRdb Values from 0 to 40
log BER
−0.5
−1.0
−1.5
−2.0
−2.5
0
5
10
15
20
25
SNRdb
log BER (Note: log BER not plotted for BER=0)

Choosing carriers and pilots

Code

```
bits=6  #bits per symbol (64 QAM)

noofcarriers = 256

carriers = [i for i in range(noofcarriers)]  # Carriers
print(carriers)

blocksize=16

nblocks=noofcarriers//blocksize

cyclicprefix = noofcarriers // 4

# Choosing pilot carriers

pilots_per_block=4    #can be changed


pilotval=1+5j   #choosen pilot value

pilotCarriers = np.empty(0,dtype=int)

for i in range (0,noofcarriers,blocksize):      #Selecting the pilot carriers
,starting carriers in each block
    for j in range (0,pilots_per_block):
        pilotCarriers=np.append(pilotCarriers,i+j)




print('pilot carriers',pilotCarriers)

#Carriers which carry the data
datacar = np.delete(carriers, pilotCarriers)

print('data carriers',datacar)
```

The number of carriers to be used is chosen . Then blocks of carriers are made. Each block of carriers has a particular no. of carriers . The next task is allocating pilots . A know value is selected as a pilot and the starting few carriers of each block are assigned the pilot value .The rest of the carriers carry the data. Also the amount of cyclic prefix to be used is also decided in this

stage. The allocation can be seen here(Here 4 carriers in each block are pilot carriers.)

```
52, 253, 254, 255]
pilot carriers [  0   1   2   3  16  17  18  19  32  33  34  35  48  49  50  51  64  65
  66  67  80  81  82  83  96  97  98  99 112 113 114 115 128 129 130 131
 144 145 146 147 160 161 162 163 176 177 178 179 192 193 194 195 208 209
 210 211 224 225 226 227 240 241 242 243]
data carriers [  4   5   6   7   8   9  10  11  12  13  14  15  20  21  22  23  24  25
  26  27  28  29  30  31  36  37  38  39  40  41  42  43  44  45  46  47
  52  53  54  55  56  57  58  59  60  61  62  63  68  69  70  71  72  73
  74  75  76  77  78  79  84  85  86  87  88  89  90  91  92  93  94  95
 100 101 102 103 104 105 106 107 108 109 110 111 116 117 118 119 120 121
 122 123 124 125 126 127 132 133 134 135 136 137 138 139 140 141 142 143
 148 149 150 151 152 153 154 155 156 157 158 159 164 165 166 167 168 169
 170 171 172 173 174 175 180 181 182 183 184 185 186 187 188 189 190 191
```

Generating data to be sent ,randomizer followed by serial to parallelization of data:

We then randomly generate data to be sent. After this we pass through the randomizer block. In a communication system, a randomizer block is often used to introduce randomness into the transmitted data stream. This process is particularly important in scenarios where the original data might have patterns or sequences that could lead to suboptimal performance during transmission or reception

```
def randomize(data, seed):      #randomization
    random.seed(seed)
    key = [random.randint(0, 1) for _ in range(len(data))]
    randomized_data = [data[i] ^ key[i] for i in range(len(data))]
    randomized_data=np.array(randomized_data)
    return randomized_data, key

randomsp=randombits.reshape((len(datacar),bits))
```

This function takes two parameters, data (the input data sequence) and seed (the seed for the random number generator). It begins by setting the seed to ensure reproducibility of random numbers. Then, it generates a random key of 0s and 1s with the same length as the input data. The function performs a bitwise XOR operation between each corresponding pair of bits in the data and the key, resulting in a randomized data sequence. Finally, the randomized data

is converted to a NumPy array, and the function returns a tuple containing the randomized data and the key used for the randomization. This type of randomization is commonly applied in communication systems to introduce controlled randomness, enhancing the robustness of transmitted data. After randomizing the data now we make it parallel by organizing it into parallel streams

Generating a constellation

Now a 64 QAM constellation is generated to be used in our system.

```
values=[-7,-5,-3,-1,1,3,5,7]

mapping_table={}
mapkeys=0

for i in range(0,len(values)):
    for j in range(0,len(values)):
        mapping_table[int_to_binary_tuple(mapkeys)]=values[i]+1j*values[j]
        mapkeys+=1
```

The provided Python code creates a mapping table for complex values using a set of specified real values. It iterates through all combinations of real values and associates each combination with a unique binary tuple key in the mapping table. The resulting complex values are formed by combining the real and imaginary parts from the given list of real values. This type of mapping is often used in digital communication systems for modulation schemes where different combinations of real values represent distinct symbols. Here we have 64 mappings each one mapping a 6 bit sequence to a symbol in the constellation.

Interleaving

```
def block_interleave(data, block_size):
    interleaved_data = []
    num_blocks = len(data) // block_size

    for i in range(block_size):
        for j in range(num_blocks):
            interleaved_data.append(data[j * block_size + i])

    return interleaved_data
```

```
interleaved_data = block_interleave(datatobesent, 6)

interleaved_data=np.array(interleaved_data)
```

It rearranges the elements of the input data by taking values from different blocks, resulting in an interleaved output. In this specific case, it uses a block size of 6. The interleaved data is then returned as a NumPy array.

The function uses two nested loops. The outer loop (for i in range(block_size)) iterates over each position within a block, and the inner loop (for j in range(num_blocks)) iterates over each block in the data.

Inside the loops, the function appends elements to the interleaved_data list by selecting them from different blocks. The selection is based on the current position i within a block and the block index j.

The formula data[j * block_size + i] calculates the index of the element in the original data sequence.

Loading data on carriers

```
def ofdm_data(qam):
    ofdmsym=np.zeros(noofcarriers,dtype=complex)
    ofdmsym[pilotCarriers]=pilotval         #pilot carriers
    #print('pilot values',ofdmsym[pilotCarriers])
    ofdmsym[datacar]=qam
    return ofdmsym

ofdmdata=ofdm_data(qam)
print(ofdmdata)
print('ofdmdata  1',ofdmdata[datacar])
```

The pilot carriers are provided with pilot values and the data carriers are provided with the data

IFFT and cyclic prefix addition:

In digital communication systems employing Orthogonal Frequency Division Multiplexing (OFDM), the Inverse Fast Fourier Transform (IFFT) and the addition of a cyclic prefix are integral components of the modulation process. The IFFT is employed to convert a set of parallel data symbols into a time-domain signal,

which is crucial for transmitting data over the channel. This transformation allows the simultaneous transmission of multiple narrowband signals, contributing to the efficiency of OFDM. However, the transmission over communication channels can introduce intersymbol interference, especially due to multipath propagation. To mitigate this, a cyclic prefix is added to the IFFT output. The cyclic prefix is a copy of the end portion of the time-domain signal that is prepended to the original signal. This redundancy assists in overcoming channel-induced delays and echoes, ensuring that the received signal can be accurately demodulated. Together, the IFFT and cyclic prefix addition play a key role in enhancing the robustness and reliability of OFDM-based communication systems in the presence of channel impairments.

```
def ifft(ofdmdata):
    return np.fft.ifft(ofdmdata)

ofdmt=ifft(ofdmdata)

print('size after ifft',ofdmt.shape)
print(ofdmt)


#cyclic prefix addition



def cyclicprefixadd(ofdmt):
    prefix = ofdmt[-cyclicprefix:]
    return np.array(list(prefix) + list(ofdmt))


ofdmcp=cyclicprefixadd(ofdmt)

print('size after adding cp',ofdmcp.shape)
```

Simulating the channel:

Now the signal is passed through the channel. A noisy wireless channel is simulated.

```
def channel(signal, SNRdb):
    h=1+ 1j
    signal=np.convolve(h,signal)
```

```
    signalpower = np.mean(abs(signal**2))  #signal power
    No = signalpower / (10**(SNRdb / 10))  # noise power
    #generating noise
    n = np.sqrt(No / 2) * (np.random.normal(size=signal.shape) + 1j *
np.random.normal(size=signal.shape))
    signal=signal + n # adding noise
    return signal

ofdmsend=ofdmcp
ofdmrec=channel(ofdmsend,SNRdb) #received signal
```

The code defines a channel function that simulates the effects of a communication channel on a given signal. The channel is represented by a complex-valued coefficient h, which represents the channel gain and phase. The function convolves the input signal with this channel response, effectively introducing channel-induced distortions to the signal. To emulate the impact of noise in the channel, the function calculates the noise power based on the desired Signal-to-Noise Ratio (SNR) in decibels (SNRdb). Gaussian noise is then generated and added to the convolved signal. The resulting signal, now including both the channel effects and noise, is returned.

In the second paragraph, the code is applied to a specific scenario. The variable ofdmsend represents an OFDM signal, likely with a cyclic prefix (ofdmcp). The ofdmsend signal is passed through the simulated channel by calling the channel function with the OFDM signal and a specified SNR. The received signal, ofdmrec, is the output of this channel simulation. This process simulates the transmission of an OFDM signal through a communication channel, considering both the channel characteristics (modelled by h) and the impact of noise at the specified SNR. Such simulations are valuable for assessing the performance of communication systems and evaluating the robustness of signal transmission under realistic channel conditions.

Prefix removing and fft

```
def prefixremover(signal):
    return signal[cyclicprefix:(cyclicprefix+noofcarriers)]

ofdmdem=prefixremover(ofdmrec)

print('size after removing cp',ofdmdem.shape)
```

```
def fft(signal):                #fft
    return np.fft.fft(signal)
ofdmdem=fft(ofdmdem)


print('size after fft',ofdmdem.shape)
```

The prefixremover function removes the cyclic prefix from the received OFDM signal, and the fft function applies the Fast Fourier Transform (FFT) to the signal. When the signal is received, the cyclic prefix is removed to obtain the original OFDM symbol. This operation is necessary before further processing to prevent interference between adjacent symbols and ensure accurate demodulation In OFDM, each subcarrier corresponds to a particular frequency, and the FFT allows the receiver to separate the contributions of each subcarrier. This frequency-domain representation facilitates channel equalization, simplifies the detection of transmitted symbols, and enables efficient data demodulation

Channel estimation and equalization:

```
def channelestimate(signal1):
    #estimating block wise
    hest=np.empty(0,dtype=complex)
    pilots=signal1[pilotCarriers]
    print('observed pilots',pilots)
    print('pilots size',pilots.size)
    knormsq=pilots_per_block*pilotval**2
    for block in range(0,pilots.size,pilots_per_block):
        normsq=0
        obsv=pilots[block:block+pilots_per_block]
        for i in range (0,pilots_per_block):
            normsq+=np.abs(pilots[block+i])**2
        arr = np.full((pilots_per_block,), pilotval)
        hblock=np.dot((arr/knormsq),obsv)               #Applying linear
regression
        print('hblock ',hblock)
        hblock1=np.full((blocksize,),hblock)
        hest=np.append(hest,hblock1)        #different estimated h for
different slots
        print('hest size',hest.size)
        print('norm in  block',block/4,'is',normsq)
    return hest
hest=channelestimate(ofdmdem)
```

```
print('hest size',hest.shape)
print('ofdmdem size',ofdmdem.shape)

def equalizer(hest,ofdm):
    return ofdm/hest

ofdmdem=equalizer(hest,ofdmdem)

ofdmdata1=ofdmdem[datacar]
print('data ofdm symbols',ofdmdata1)
```

The channelestimate function aims to estimate the channel response based on the observed pilots in the received signal. The function iterates through blocks of observed pilot symbols, calculates the squared norm of the observed pilots, and applies linear regression to estimate the channel response for each block. The estimated channel responses for different blocks are then concatenated to form the complete channel estimate (hest).

The equalizer function employs the channel estimate obtained from channelestimate to perform equalization on the received OFDM signal (ofdm). Equalization is a crucial step in combating the effects of channel distortion. By dividing the received signal by the estimated channel response, the equalizer compensates for the channel-induced distortion, enhancing the recovery of the transmitted symbols.

In the provided code, these functions are applied to an OFDM-demodulated signal (ofdmdem). The channel estimate (hest) is obtained through the channelestimate function, and subsequently, the equalizer function is used to equalize the OFDM signal. The resulting equalized signal, denoted as ofdmdem, is then used to extract the data OFDM symbols (ofdmdata1). This process is crucial in communication systems to mitigate the impact of the channel on transmitted data, enhancing the overall performance of the OFDM-based communication system

```
hblock  (0.9892571339235585+1.0798114923901438j)
hest size 16
norm in  block 0.0 is 225.36061515108614
hblock  (1.0522993606199131+0.9522687048832087j)
hest size 32
norm in  block 1.0 is 210.3052000589322
hblock  (0.9971775882675614+1.0147941549786743j)
hest size 48
norm in  block 2.0 is 212.58737580989026
hblock  (1.0082066332118496+0.9983628509341055j)
hest size 64
norm in  block 3.0 is 212.06821803133943
hblock  (0.9507956034991815+0.9522764812447657j)
hest size 80
norm in  block 4.0 is 189.59884333699452
hblock  (1.0191598883576176+1.0031814708273123j)
hest size 96
norm in  block 5.0 is 216.88623730559664
hblock  (1.0017975376828858+1.0225095603675352j)
hest size 112
norm in  block 6.0 is 216.25844443951328
```

Fig:Code calculating Hest for different blocks

Demapping:

Now we have the OFDM symbols and we want to get back to the original bit sequence.

```
def demapping(ofdmdata1):
    points=np.empty(0,dtype=tuple)
    ofdmdata1=ofdmdata1[datacar]
    estsymbol=np.empty(0,dtype=complex)
    for i in range (0,ofdmdata1.size):
        btup=int_to_binary_tuple(0)
        dist=abs(ofdmdata1[i]-mapping_table[btup])
        symbol=mapping_table[btup]
        for j in range(0,64):
            btj=int_to_binary_tuple(j)
            if (abs(ofdmdata1[i]-mapping_table[btj])<dist):
                dist=np.abs(ofdmdata1[i]-mapping_table[btj])
                #print(' op1',ofdmdata1[i],' op2 ',mapping_table[btj],' =
',dist)
                symbol=mapping_table[btj]
                btup=btj
        #print('btup',btup)
        points=np.append(points,btup)
```

```
        estsymbol=np.append(estsymbol,symbol)
    return points,estsymbol

sequence,estsymbol=demapping(ofdmdem)
```

The function takes as input the data symbols (ofdmdata1) obtained after equalization and extracts the symbols corresponding to the data carriers. It then demaps these symbols back to their original binary representations based on the closest match with the entries in the mapping_table. The resulting binary tuples and the estimated symbols are returned as outputs.

Within the function:

points: Initializes an empty array to store the binary tuples representing the demapped symbols.

estsymbol: Initializes an empty array to store the estimated symbols corresponding to the demapped data.

The function iterates over each received data symbol in ofdmdata1. For each symbol, it performs a comparison with each entry in the mapping_table to find the closest match. The comparison is based on the Euclidean distance between the received symbol and each possible symbol in the table. The binary tuple (btup) corresponding to the closest match is stored in the points array, and the estimated symbol is stored in the estsymbol array.

Finally, the function returns two arrays: points, containing the binary tuples representing the demapped symbols, and estsymbol, containing the estimated symbols corresponding to the demapped data. This demapping process is crucial for recovering the transmitted information and completing the demodulation process in an OFDM communication system.

A working execution is shown below:

```
Symbol  sent (-1-5j)
Symbol received (-0.07882196648568088-4.920122457133111j)
Estimated constellation (-1-5j)
Symbol  sent (-7-7j)
Symbol received (-7.639314716655899-6.789193368598335j)
Estimated constellation (-7-7j)
Symbol  sent (-1+1j)
Symbol received (-0.9727860970858117+0.498854181288707j)
Estimated constellation (-1+1j)
Symbol  sent (-5+7j)
Symbol received (-5.226286356970196+7.415528024329684j)
Estimated constellation (-5+7j)
```

Vitterbi decoder and TS Buffer:

```
#convolution encoding using vitterbi algorithm
dot11a_codec = Viterbi(7, [0o133, 0o171])

sequence = np.array(dot11a_codec.encode(sequence),dtype=int)


sequence=np.array(dot11a_codec.decode(sequence),dtype=int)
print('sequence first few samples',sequence[0:64])

print('randombits first few samples',randombits[0:64])
class Buffer:
    def __init__(self):
        self.my_list = []

    def add_element(self, element):
        self.my_list.append(element)
        self.make_tuple_if_size()

    def make_tuple_if_size(self):
        if len(self.my_list) == 6:
            my_tuple = self.convert_to_tuple()
            self.clear_list()
            afterbuffer.append(my_tuple)
            return None


    def convert_to_tuple(self):
        return tuple(self.my_list)

    def clear_list(self):
        self.my_list.clear()
```

```
bufferobj= Buffer()

for i in range (0,sequence.size):
   bufferobj.add_element(sequence[i])
print(' After buffer ',afterbuffer[:24])

afterbuffer=np.array(afterbuffer)
afterbuffer=afterbuffer.flatten()
```

The Viterbi algorithm is a dynamic programming algorithm used for decoding convolutionally encoded sequences. It works by finding the most likely sequence of states in a trellis that could have produced the observed sequence, taking into account the probabilities associated with each transition. Viterbi decoding is particularly effective in correcting errors introduced in noisy communication channels.

The code defines a Buffer class with methods to add elements to a list and convert the list to a tuple when a specific size (6 in this case) is reached. An instance of the Buffer class, named bufferobj, is created. The code then iterates over the elements in a given sequence, and for each element, it is added to the bufferobj. When the size of the internal list reaches the specified limit, the list is converted to a tuple, cleared, and the tuple is appended to the afterbuffer list. Finally, the content of afterbuffer is converted into a NumPy array. This code essentially performs buffering by collecting elements into a list until a threshold size is reached, at which point the elements are converted to tuples and stored for further processing or analysis. Basically we are waiting for bits to come so that they can be clubbed to use in further operations.

De-interleaving and derandomizing:

```
def block_deinterleave(interleaved_data, block_size):
    deinterleaved_data = []
    num_blocks = len(interleaved_data) // block_size

    for i in range(num_blocks):
        for j in range(block_size):
            deinterleaved_data.append(interleaved_data[j * num_blocks + i])

    return deinterleaved_data
```

```
deinterleaved=block_deinterleave(afterbuffer,block_size=6)

deinterleaved=np.array(deinterleaved)

def derandomize(data, key):
    derandomized_data = [data[i] ^ key[i] for i in range(len(data))]
    return np.array(derandomized_data)

received_data = derandomize(deinterleaved, keyr)

deinterleaved=received_data

deinterleaved = deinterleaved.reshape((len(datacar),bits))

print(" deinter ",deinterleaved[0:20])
```

The provided Python code comprises two functions: block_deinterleave and derandomize, applied to a sequence of received data. The block_deinterleave function is designed to reverse the block interleaving process, where the data has been organized into blocks during transmission for error resilience. It rearranges the elements by extracting them in a systematic manner to recover the original order. The derandomize function then performs the inverse of the randomization process applied during transmission. It takes the deinterleaved data and a randomization key to bitwise XOR each element, effectively reversing the randomization step.

In the code execution, the block_deinterleave function is applied to the afterbuffer data, which likely contains interleaved and randomized symbols. The resulting deinterleaved sequence is then derandomized using the derandomize function with a provided key. The deinterleaved array is reshaped to match the original block and bit organization, resulting in the deinterleaved data. This process is crucial in recovering the original transmitted symbols, providing a crucial step in the demodulation and error correction procedures of a communication system, particularly in scenarios where interleaving and randomization were employed for improved performance in the presence of channel impairments.

Reed Solomon and final Bit Error Rate:

```
rsc = RSCodec(10)

afterrsc = []


for i in range(0, len(deinterleaved)):
    # Data to be encoded
    encoded = rsc.encode(tuple(deinterleaved[i]))

    # Simulating distortion by setting the last element to 1
    distorted = encoded
    distorted[-1] = 1

    # Decoding the distorted data
    decoded_msg, decoded_msgecc, errata_pos = rsc.decode(distorted)

    # Appending the result as a tuple to afterrsc
    afterrsc.append(tuple(decoded_msg[0:6]))
    #print(' encoded ',deinterleaved[i],' decoded ',decoded_msg)

print('afterrsc',afterrsc[0:20])

afterrsc=np.array(afterrsc)

afterrsc=afterrsc.flatten()


print(' original size ',randombits.size,' received size ',afterrsc.size)

wrongbits=0
for i in range(0,randombits.size):
    if(sequencetobesent[i]!=afterrsc[i]):
        wrongbits+=1

print("BER final :",wrongbits/randombits.size)
```

Now the Reed Solomon algorithm is used .The RSCodec(10) object is created, representing a Reed-Solomon code with a coding strength of 10. The code then iterates over the deinterleaved data, encoding each block with Reed-Solomon encoding. Simulating channel distortion, the last element of the encoded block is set to 1, and the distorted block is then decoded. The result is appended to the afterrsc list as a tuple.

After processing all blocks, the afterrsc list is converted to a NumPy array and flattened. The code then compares the original size of the transmitted bits (randombits.size) with the size of the received and decoded bits (afterrsc.size). Subsequently, a bit error rate (BER) is calculated by comparing the received and original sequences bit by bit. The number of differing bits is counted, and the BER is computed as the ratio of the wrong bits to the total number of transmitted bits.