Project Progress till 3rd April

We delved deeper into understanding the NOMA system and implemented the code available on github. We will explain our understanding of NOMA and code in this report.

NOMA is generally introduced through Signal-to-Interference-and-Noise ratio (SINR) and sum rate analyses. Also to compare the performance with OMA techniques, we use high Signal-to-Noise (SNR) analysis. In the downlink scenario, the transmitter combines signals of different users with different allocated power coefficients which are inversely related to their channel conditions. Now the users need to perform SIC where each user detects components that are stronger than its own desired power and subtracts it from the received signal. This process is continued until the user's own signal is determined and it treats the signals with lower power as noise compared to its own signal.

$$s = \sum_{i=1}^{L} \sqrt{a_i P_s} x_i,$$

Encoded Signal

$$y_l = h_l s + n_l = h_l \sum_{i=1}^{L} \sqrt{a_i P_s} x_i + n_l,$$

Signal at receiver

$$\mathrm{SINR}_l = \frac{a_l \gamma |h_l|^2}{\gamma |h_l|^2 \sum_{i=l+1}^{L} a_i + 1}.$$

SINR expression

Code for OFDMA-

```
import commpy
from commpy.modulation import PSKModem, QAMModem
import numpy as np

# This function modulates a given baseband signal using the specified modulation type
def modulate(BasebandSignal, type):
    # Determine the appropriate modulation scheme based on the specified type
    if(type=='4PSK'):
        modulator = PSKModem(4)
    elif (type=='8PSK'):
        modulator = PSKModem(8)
    elif (type=='16PSK'):
        modulator = PSKModem(16)
    elif (type=='4QAM'):
        modulator = QAMModem(4)
    elif (type=='8QAM'):
        modulator = QAMModem(8)
    else:
        modulator = QAMModem(16)
    # Modulate the baseband signal using the determined modulation scheme
    return modulator, modulator.modulate(BasebandSignal)

# This function takes in a signal, samples it at a given frequency, and returns the result
def sample(Signal, k, carr_num, sign):
    # Initialize variables to hold the real and imaginary components of the sampled signal
    x_real = 0
    x_img = 0
    # Loop through each carrier in the signal
    for i in range(carr_num):
        # Compute the real and imaginary components of the sampled signal using the given
sampling frequency and sign
        x_real += Signal[i].real * np.cos(2 * np.pi * i * k / carr_num) - Signal[i].imag *
np.sin(2 * sign * np.pi * i * k / carr_num)
        x_img += Signal[i].real * np.sin(2 * sign * np.pi * i * k / carr_num) +
Signal[i].imag * np.cos(2 * np.pi * i * k / carr_num)
    # Normalize the resulting signal and return it
    return (complex(x_real, x_img)) / np.sqrt(carr_num)
import numpy as np

def OFDM_encode(Signal, H, Symbol_rate, Bandwidth):
    # Assume the cyclic prefix is equal to maximum delay spread
    max_delay = len(H) - 1
    # Calculate the number of sub-carriers
    Carr_num = np.ceil(max_delay / (Bandwidth / Symbol_rate - 1))
    Carr_num = int(Carr_num)
    # Do sampling in every window
    block_index = 0
```

```
    block_num = int(len(Signal) / Carr_num)
    encoded_sig = np.zeros((block_num, Carr_num + len(H) - 1), dtype=complex)
    trans_sig = np.zeros((block_num, Carr_num), dtype=complex)
    # Do OFDM encoder
    while block_index < block_num:
        # Current block and next block
        s_cu = Signal[block_index * Carr_num:(block_index + 1) * Carr_num]
        s_next = Signal[(block_index + 1) * Carr_num:(block_index + 2) * Carr_num]
        for t in range(Carr_num):
            # Do sampling
            encoded_sig[block_index][t + len(H) - 1] = sample(Signal=s_cu, k=t,
carr_num=Carr_num, sign=1)
            # Overlap
            # Discard surplus data
            if t > Carr_num - len(H) and len(s_next) == Carr_num:
                encoded_sig[block_index + 1][t - Carr_num + len(H) - 1] =
sample(Signal=s_next, k=(t - Carr_num), carr_num=Carr_num, sign=1)
        block_index += 1
    # Encoded signal cross a multipath channel
    for i in range(block_num):
        # Ignore cyclic prefix
        for j in range(Carr_num):
            for k in range(len(H)):
                trans_sig[i][j] += H[k] * encoded_sig[i][j - k + len(H) - 1]
            j += 1

    return Carr_num, trans_sig

def OFDM_Decode(trans_sig, H, carr_num):
    # Create a list of indices for the channel frequency response
    channel_index = []
    for i in range(carr_num):
        if i < len(H):
            channel_index.append(H[i])
        else:
            channel_index.append(0)
    # Compute the frequency response of the channel using the FFT
    channel_index = np.fft.fft(channel_index)

    decode_sig = []
    for t in range(np.shape(trans_sig)[0]):
        for k in range(carr_num):
            # Recover the transmitted symbol using the inverse FFT
            temp = sample(Signal=trans_sig[t], k=k, carr_num=carr_num, sign=-1)
            # Equalize the symbol by dividing by the channel frequency response
            temp = temp / channel_index[k]
            decode_sig.append(temp)
    # Print the length of the decoded signal
    print(len(decode_sig))
```

```
    return decode_sig

if __name__=='__main__':
    BasebandSignal=[]
    Seq_len=76
    for i in range(Seq_len):
        BasebandSignal.append(np.random.randint(0,2))
    modulator,BandpassSignal=modulate(BasebandSignal,'8PSK')
    # Parameter of channel
    H=[0.5,0.6]
    # Desired Symbol rate
    Symbol_rate=5e+05
    #available bandwith
    Bandwidth=6e+05
    carrnum,tran_sig=OFDM_encode(BandpassSignal,H,Symbol_rate,Bandwidth)
    decode_sig=OFDM_Decode(tran_sig,H,carrnum)
    demodulate_sig=modulator.demodulate(decode_sig,'hard')
```