TEAM MAVERICKS

ADITYA RAJ & ARYANSH KUMAR

COMMNET PROBLEM STATEMENT 2

Problem Statement : In this PS, we have design and model a quasi 5G NR system, incorporating some important concepts from 5G that mimic real world communication system

Transmitter

- **Source, sampling and quantization**
  - Implement these blocks as a random digital message.
  - Create floating point values as sampled data
  - 1 frame should consist of 20*10^3 bits
  - Quantize it by taking the floor valu
- **Source Encoder**
  - Convert integers to 2' s complement binary numbers
  - Ensure that you take sufficient bits to accommodate all values
  - Also ensure that you take minimum possible bits needed for communication
- **Forward Error Correction (FEC)**
  - Introduce redundancy in the code by writing each bit 5 times over.
  - This redundancy will help us in detecting and correcting bit errors that occur in the channel
- **Modulation/ Mapping**
  - Introduce QAM - 64 modulation scheme
  - Plot the signal constellation
- **Pulse shaping**
  - Implement raised cosine filter
  - Filter the data stream

Channel Rayleigh

- **Fading channel**

    - Implement 5 path channel with different delays and path gains
- **AWGN**
    - Keep SNR as 6dB
- **Timing Offset**
    - Introduce a random (unknown) timing offset
- **Frequency Offset (Bonus)**
    - Introduce a random (unknown) frequency offset

Receiver

- **Carrier synchronization (Bonus)**
    - Corrects the frequency offset
    - Implement if frequency offset is introduced
    - Implement using standard MATLAB functions only
- **Timing synchronization**
    - Corrects the timing offset
    - Implement using standard MATLAB functions only
- **Inverse pulse shaping**
    - Also known as matched filter
    - Take care of the group delay
    - Take care of the filter gain
- **Channel equalization (Option 1)**
    - The channel parameters is used to create the Rayleigh Fading channel in the Channel part
    - Do channel equalization using zero-forcing method
    - That is divide the signal by the transfer function

    - **Channel decoding**
        - Do inverse repetitive coding
        - That is create a sub-frame of 5 bits
        - Replace the majorly occurring bits among those 5 as the decoded bit

- Logic is even if bit errors are introduced, then the system will be robust and only a minority of the ensemble of 5 bits will be corrupted
- **Demodulation/ Demapping**
  - Demodulate using qamdemod function

OUR APPROACH

Detailed Breakdown of the Code

This code aims to replicate a communication system that transmits digital data and retrieves it at the receiver despite potential disruptions in the channel. Below is a comprehensive explanation of each step involved:

1. Data Initialization (lines 1-5):

- size = 2*10^4;: Establishes the size of the data to be transmitted, set at 20,000 in this instance.
- data = rand([1, size]) * 100;: Generates a random array of size 1 x size with values ranging between 0 and 1 (exclusive), representing the source data for transmission, scaled by 100.
- data_quant = floor(data);: Quantizes the data by rounding down each element in the data array to the nearest integer, reducing the required bits for representation but introducing quantization noise.
- Nbit = 7;: Specifies the number of bits allocated to represent each quantized data value (symbol) in binary form, set at 7 bits here.
- bin = de2bi(data_quant, Nbit, 'left-msb');: Converts the quantized data (data_quant) into a binary representation using the de2bi function. This conversion transforms decimal numbers (integers in data_quant) into binary format. The Nbit parameter dictates the number of bits per symbol, while 'left-msb' dictates the placement of

the most significant bit (MSB) on the left side of the binary representation.

2. **Encoding (lines 6-12):**

    - EncodeData = zeros([sizeNbit5, 1]);: Initializes an empty array EncodeData sized at sizeNbit5 (number of symbols * bits per symbol * redundancy factor) to store the encoded data.
    - for i = 1:size*Nbit loop: Iterates through each bit in the binary representation of the quantized data (bin).
    - *EncodeData(1+5(i-1)) = bin(i);**: Copies the current bit (bin(i)) into the first element of each sub-block within EncodeData. Each sub-block comprises 5 elements due to the introduced redundancy for error correction.
    - The loop continues, replicating the same bit (bin(i)) into the second, third, fourth, and fifth elements of the ongoing sub-block in EncodeData. This repetition enhances signal robustness against noise during transmission.
    - pad = [EncodeData; 0; 0];: Appends two zeros ([0; 0]) to the end of EncodeData, potentially facilitating synchronization at the receiver.

3. **Modulation (line 13):**

    - mod = qammod(pad, 64, 'InputType', 'bit');: Modulates the encoded data (pad) using QAM-64 modulation. QAM (Quadrature Amplitude Modulation) integrates amplitude and phase variations to represent multiple bits within a single symbol. Here, 64 distinct symbols are utilized, with each representing 6 bits of data. The 'InputType' parameter specifies that the input data is in bit format.

4. **Pulse Shaping (lines 14-15):**

    - rcTx = comm.RaisedCosineTransmitFilter(FilterSpanInSymbols=4, OutputSamplesPerSymbol=4);: Defines a raised cosine transmit filter object (rcTx) utilizing the comm.RaisedCosineTransmitFilter function. This filter molds the individual pulses within the modulated signal (mod) to curtail spectral bandwidth and mitigate inter-symbol interference (ISI) during transmission. The FilterSpanInSymbols parameter determines the filter's roll-off factor, while OutputSamplesPerSymbol specifies the number of samples produced per symbol at the filter output.

- txData = rcTx(mod);: Applies the raised cosine transmit filter (rcTx) to the modulated signal (mod), shaping the pulses and resulting in the transmitted signal txData.

5. Channel Simulation (lines 16-27):

- Eb_N0_dB = 6;: Sets the Eb/N0 value (ratio of energy per bit to noise power spectral density) in decibels (dB), used to compute the signal-to-noise ratio (SNR) at the receiver.
- SNR = 10^(Eb_N0_dB/20);: Calculates the SNR based on the Eb/N0 value. A higher SNR indicates a stronger signal relative to noise, leading to improved transmission quality.
- h = zeros(length(txData), 1);: Initializes an empty array h of the same length as the transmitted signal txData to represent the channel response, describing how the channel distorts the transmitted signal.
- path_delays = [0, 1, 3, 5, 7];: Specifies an array path_delays indicating the delays (in symbol periods) encountered by various signal paths through the channel. In a multipath channel scenario, signals may reach the receiver through multiple paths with differing delays.
- path_gains = [1, 0.7, 0.5, 0.3, 0.1];: Defines an array path_gains indicating the gains (attenuations) associated with each path delay. Longer paths typically result in weaker signals due to attenuation.
- for i = 1:length(path_delays) loop: Iterates through each path delay.
- h = h + path_gains(i)1/sqrt(2)(randn(length(txData),1) + 1jrandn(length(txData),1)).exp(-1j2pipath_delays(i)/length(txData)(1:

length(txData))');: Models the contribution of each path to the overall channel response h. This calculation incorporates:

- path_gains(i): The attenuation factor for the current path.
- randn(length(txData),1) + 1j*randn(length(txData),1): Generation of random noise with zero mean and Gaussian distribution for the in-phase (real) and quadrature (imaginary) components. The sqrt(2) factor scales the noise to accommodate the average power of QAM-64 symbols.
- exp(-1j2pipath_delays(i)/length(txData)(1:length(txData))'): Creation of a complex exponential term introducing a phase shift based on the path delay path_delays(i). This models the time delay experienced by the signal on that path.
- The loop sums the contributions from all paths to generate a realistic channel response accounting for multipath propagation and attenuation.
- n = 1/sqrt(2)(randn(length(txData),1) + 1jrandn(length(txData),1));: Generates random noise (n) with zero mean and Gaussian distribution for both the in-phase and quadrature components. The noise power is scaled by 1/sqrt(2) to achieve the desired SNR based on the previously calculated SNR value.
- rxData = h.*txData + sqrt(1/SNR)*n;: Simulates the received signal (rxData), involving:
- h.*txData: Convolution of the transmitted signal (txData) with the channel response (h), simulating channel-induced signal distortion.
- sqrt(1/SNR)*n: Addition of generated noise (n) scaled by the square root of the inverse SNR to the convolved signal, introducing noise into the received signal as per simulated channel conditions.

6. Timing and Frequency Offsets (lines 28-33):

- timing_offset = rand() - 0.5;: Generates a random timing offset ranging from -0.5 to 0.5 symbol periods.

- rxData_shifted = circshift(rxData, round(timing_offset * length(rxData)));: Applies the timing offset by circularly shifting the received signal (rxData) by round(timing_offset * length(rxData)) samples, simulating a scenario where the received signal is misaligned with expected symbol boundaries due to synchronization issues.
- freq_offset = (rand() - 0.5) * 0.02;: Generates a random frequency offset ranging from -0.01 to 0.01 of the symbol rate.
- phase_shift = exp(1j * 2 * pi * freq_offset * (1:length(rxData))');: Creates a complex exponential term varying with time, modeling a linearly increasing phase shift due to frequency offset in the received signal. This offset may cause symbol rotation in the constellation diagram, complicating correct decoding.
- *rxData_freq_offset = rxData . phase_shift;**: Applies the frequency offset by multiplying the received signal (rxData) with the complex exponential term (phase_shift), incorporating the phase shift induced by the frequency offset into the received signal.
- rxData = rxData_corrected;: Corrects the frequency offset by dividing the received signal with the complex exponential term (phase_shift), ideally restoring the original signal. However, this step assumes knowledge of the exact frequency offset, which may not be realistic in practice.

7. Matched Filtering and Equalization (lines 34-44):

- (Create a constellation diagram object): This step involves creating a visualization tool to analyze the signal in the constellation diagram. The constellation diagram plots received symbols on a complex plane, where each symbol position represents a combination of in-phase and quadrature values.
- constDiagram = comm.ConstellationDiagram('SamplesPerSymbol', 4,'SymbolsToDisplaySource', 'Property', 'Title', 'Frequency Synchronized Signal');: Defines a constellation diagram object (constDiagram) using the comm.ConstellationDiagram function,

specifying the number of samples per symbol (4 for QAM-64), symbol display method, and diagram title.

- constDiagram(rxData);: Plots the frequency-synchronized received signal (rxData) on the constellation diagram, aiding in assessing the frequency offset's impact on the signal.
- release(constDiagram);: Releases the constellation diagram object after use.
- symbolSync = comm.SymbolSynchronizer('SamplesPerSymbol', 4);: Defines a symbol synchronizer object (symbolSync) using the comm.SymbolSynchronizer function, facilitating timing offset recovery by aligning received symbols with expected symbol boundaries.
- rxData_corrected = step(symbolSync, rxData);: Applies the symbol synchronizer (symbolSync) to the received signal (rxData), correcting the timing offset and aligning symbols for improved decoding. The corrected signal is stored in rxData_corrected.
- constDiagram = comm.ConstellationDiagram('SamplesPerSymbol', 4,'SymbolsToDisplaySource', 'Property', 'Title', 'Timing Synchronized Signal');: Defines another constellation diagram object to visualize the timing-synchronized signal.
- constDiagram(rxData);: Plots the timing-synchronized received signal (rxData_corrected) on the constellation diagram, aiding in assessing the effectiveness of symbol synchronization.
- release(constDiagram);: Releases the constellation diagram object after use.
- (Define the matched filter): This step involves designing a filter matched to the transmit filter (rcTx) used earlier. The matched filter helps recover original pulses from the received signal.
- rcRx = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols', 4, 'InputSamplesPerSymbol', 4, 'DecimationFactor', 1);: Defines a raised cosine receive filter object (rcRx) using the comm.RaisedCosineReceiveFilter function, matched to the transmit

filter (rcTx) to recover transmitted pulses from the received signal. Filter parameters include filter span in symbols, input and output samples per symbol, and no downsampling (decimation factor of 1).

- rxData_matched_filter = rcRx(rxData);: Applies the matched filter (rcRx) to the received signal (rxData), reducing inter-symbol interference (ISI) and recovering original pulses encoded in the signal.
- (Adjust for group delay): This step involves compensating for the delay introduced by the matched filter (rcRx). Group delay refers to the time difference between the center of the filter's impulse response and its ideal output.
- group_delay = rcRx.FilterSpanInSymbols / 2;: Calculates the group delay based on the filter span in symbols (rcRx.FilterSpanInSymbols).
- rxData_matched_filter = rxData_matched_filter(group_delay+1:end);: Adjusts for group delay by discarding the first group_delay samples from the filtered signal (rxData_matched_filter), ensuring remaining samples are aligned with original symbols.
- (Adjust for filter gain): This step involves normalizing the filtered signal's amplitude to account for any gain introduced by the matched filter.
- rxData_matched_filter = rxData_matched_filter / max(abs(rxData_matched_filter));: Normalizes the filtered signal (rxData_matched_filter) by dividing it by the maximum absolute value within the signal, ensuring consistent amplitude for further processing.
- constDiagram = comm.ConstellationDiagram('SamplesPerSymbol', 4,'SymbolsToDisplaySource', 'Property', 'Title', 'Signal after inverse pulse shaping');: Defines another constellation diagram object to visualize the signal after matched filtering.
- constDiagram(rxData_matched_filter);: Plots the signal after matched filtering (rxData_matched_filter) on the constellation diagram, aiding in assessing the matched filter's effectiveness in recovering pulses.

- release(constDiagram);: Releases the constellation diagram object after use.
- (Extract the relevant portion of the channel response): This step involves isolating the channel response experienced by the signal post-symbol synchronization. With symbol synchronization aligning symbols, the channel response can now be estimated based on the filtered signal.
- h_equalization = h(group_delay+1:group_delay+length(rxData_matched_filter));: Extracts the relevant portion of the channel response (h_equalization) from the original channel response (h). This portion corresponds to symbols post-symbol synchronization, accounting for channel distortion experienced by the data.
- (Zero-forcing channel equalization): This step involves applying an equalization technique to remove channel distortion from the signal.
- rxData_equalized = rxData_matched_filter ./ h_equalization;: Conducts zero-forcing channel equalization by dividing the filtered signal (rxData_matched_filter) by the estimated channel response (h_equalization), ideally eliminating the channel's filtering effect and restoring original transmitted symbols.
- rxData = rxData_equalized;: Updates the received signal variable (rxData) to hold the equalized signal (rxData_equalized).
- constDiagram = comm.ConstellationDiagram('SamplesPerSymbol', 4,'SymbolsToDisplaySource', 'Property', 'Title', 'Channel equalization using zero forcing');: Defines another constellation diagram object to visualize the signal post-channel equalization.
- constDiagram(rxData);: Plots the equalized signal (rxData) on the constellation diagram, aiding in assessing channel equalization effectiveness in correcting channel distortion.
- release(constDiagram);: Releases the constellation diagram object after use.

8. Decoding:

- rxData(5(i-1) + 1 : 5i) = repmat(major_bit, 5, 1);**: Replaces all bits within the current sub-frame (rxData(5*(i-1) + 1 : 5*i)) with the majority bit value (major_bit) determined earlier. This redundancy introduced during decoding aims to compensate for potential errors in individual bits within the sub-frame.
- end: The loop iterates through all sub-frames, applying the majority vote decoding technique.
- 9. Demodulation and Error Calculation (lines 53-60):
-
- demod = qamdemod(rxData, 64, 'OutputType', 'bit');: Demodulates the decoded received signal (rxData) using QAM-64 demodulation with the qamdemod function, recovering the original binary representation of the data from the received symbols.
- reshape(demod, [], Nbit);: Reshapes the demodulated data (demod) into a column vector, with each element representing one bit. The Nbit (7 in this case) specifies the number of bits used to represent each symbol.
- num_errors = sum(abs(demod - bin(:)) > 0);: Calculates the number of bit errors (num_errors) by comparing the demodulated data (demod) with the original binary data (bin(:)). The abs function calculates the absolute difference between corresponding elements, and the sum function counts the number of elements where the difference is greater than 0 (indicating an error).
- BER = num_errors / numel(bin);: Calculates the Bit Error Rate (BER) by dividing the number of bit errors (num_errors) by the total number of transmitted bits (numel(bin)).
- fprintf('BER = %f, Number of errors = %d\n', BER, num_errors);: Prints the calculated BER and the number of bit errors to the console.