FOR BETTER UNDERSTANDING OF THE MATLAB CODES WE HAVE WRITTEN THIS PSUEDO CODE FOR HARD AND SOFT DECODING, ENCODING, MODULATOR, AND OUR MAIN FUNCTION. WE HAVE WRITTEN THE COMMENTS IN THE PSUEDO CODE TO GET THE DEPTH OF THE CODE. WE USED THE SAME CODE IN THE MATLAB FILE (.mlx) SO IT DOES NOT CONTAIN ANY COMMENTS.

PSUEDO CODE :

- FOR ENCODING:

```
FUNCTION encoding(kc, generation_polynomials, inp)

   // Initialize variables

   n = LENGTH(generation_polynomials)  // Number of generation polynomials

   ans = []                        // Initialize empty list for storing encoded sequence

   regval = 0                      // Initialize register value to zero


   // Loop through each input bit

   FOR i FROM 1 TO LENGTH(inp)

      curbit = inp[i]                  // Get current input bit

      regval = BITOR(BITSHIFT(regval, -1), BITSHIFT(curbit, (kc - 1)))  // Update register value --> here if
second arguement is

     negative then perfom shift right otherwise perfom shift left.


      // Loop through each generation polynomial

      FOR j FROM 1 TO n

         cur_poly = generation_polynomials[j]  // Get current generation polynomial

         output = 0                       // Initialize output to zero
```

```
        p = DEC2BIN(cur_poly, kc)           // Convert current polynomial to binary
        p = FLIPLR(p) - '0'                 // Reverse the binary representation and convert to integer array
        rv = DEC2BIN(regval, kc) - '0'      // Convert register value to binary
        output = 0                          // Initialize output to zero

        // Compute output using bitwise AND and XOR operations
        FOR k FROM 1 TO kc
          output = XOR(output, (p[k] AND rv[k]))  // Update output using XOR and AND operations
        END FOR

        ans = CONCATENATE(ans, output)  // Append output to result list
      END FOR
    END FOR

    RETURN ans  // Return the encoded output
END FUNCTION
```

- Soft_Decoding :

```
FUNCTION decoding_soft(kc, generation_polynomials, inp)
    // Initialize variables
    ns = BITSHIFT(1, (kc - 1))                 // Number of states : here we compute 2^(kc-1)
```

```
n = LENGTH(generation_polynomials)          // Number of generation polynomials

mtr = ONES(FLOOR(LENGTH(inp) / n) + 1, ns) * 1e9 // Initialize metric matrix with large values

previous_states = CELL(FLOOR(LENGTH(inp) / n) + 1, 1) // Initialize previous states cell array


// Initialize previous_states with 1e9 - large values

FOR i FROM 1 TO LENGTH(previous_states)

   previous_states[i] = CELL(ns, 1)

   FOR j FROM 1 TO ns

      previous_states[i][j] = {1e9, 1e9}

   END FOR

END FOR

idx = 0

FOR t FROM 1 TO (LENGTH(inp) / n)

   // Loop through states

   FOR st FROM 0 TO ns - 1

      // Initialize metric for the first time instance and state 0

      IF (t == 1) AND (st == 0)

         mtr(t, st + 1) = 0

      END IF


      // Loop through input bits (0 and 1)

      FOR input_bits FROM 0 TO 1

         curstate = st

         next_state = BITOR(BITSHIFT(curstate, -1), (input_bits * BITSHIFT(1, (kc - 2))))

         euclidean_distance = 0


         // Compute output and update euclidean distance
```

```
        FOR i FROM 1 TO n
          output = 0
          poly = generation_polynomials[i]
          regvalue = BITOR(curstate, BITSHIFT(input_bits, (kc - 1)))

          // Compute output using bitwise operations
          FOR k FROM 0 TO (kc - 1)
            output = BITXOR(output, BITAND(BITSHIFT(poly, -k), 1) AND BITAND(BITSHIFT(regvalue,
-(kc - k - 1)), 1))
          END FOR

          output = 1 - 2 * output
          euclidean_distance = euclidean_distance + (ABS(output * inp(idx + i)) * ABS(output -
inp(idx + i)))
        END FOR

        euclidean = SQRT(euclidean_distance)
        a = mtr(t + 1, next_state + 1)
        b = mtr(t, curstate + 1)

        // Update metric and previous states
        IF a > b + euclidean_distance
          mtr(t + 1, next_state + 1) = b + euclidean_distance
          previous_states[t + 1][next_state + 1] = {curstate, input_bits}
        END IF
      END FOR
    END FOR
    idx = idx + n
```

```
    END FOR

    // Initialize ans list
    ans = []

    // Trace back to find the optimal path
    temp = previous_states[(LENGTH(inp) / n) + 1][1]
    ans = [temp[2] ans]
    cur = LENGTH(inp) / n

    WHILE cur > 1
        temp = previous_states[cur][temp[1] + 1]
        ans = [temp[2] ans]
        cur = cur - 1
    END WHILE

    RETURN ans  // Return the decoded output
END FUNCTION
```

- Hard_Decoding :

```
FUNCTION decoding(kc, generation_polynomials, inp)
    // Initialize variables
    ns = BITSHIFT(1, (kc - 1))                    // Number of states
    n = LENGTH(generation_polynomials)            // Number of generation polynomials
    mtr = ONES(FLOOR(LENGTH(inp) / n) + 1, ns) * 1e9 // Initialize metric matrix with large values
```

```
previous_states = CELL(FLOOR(LENGTH(inp) / n) + 1, 1) // Initialize previous states cell array

// Initialize previous_states with large values
FOR i FROM 1 TO LENGTH(previous_states)
   previous_states[i] = CELL(ns, 1)
   FOR j FROM 1 TO ns
      previous_states[i][j] = {1e9, 1e9}
   END FOR
END FOR

idx = 0

// Loop through time instances
FOR t FROM 1 TO (LENGTH(inp) / n)
   // Loop through states
   FOR st FROM 0 TO ns - 1
      // Initialize metric for the first time instance and state 0
      IF (t == 1) AND (st == 0)
         mtr(t, st + 1) = 0
      END IF

      // Loop through input bits (0 and 1)
      FOR input_bits FROM 0 TO 1
         curstate = st
         next_state = BITOR(BITSHIFT(curstate, -1), (input_bits * BITSHIFT(1, (kc - 2))))
         hamming_distance = 0
```

```
            // Compute output and update hamming distance
            FOR i FROM 1 TO n
               output = 0
               poly = generation_polynomials[i]
               regvalue = BITOR(curstate, BITSHIFT(input_bits, (kc - 1)))

               // Compute output using bitwise operations
               FOR k FROM 0 TO (kc - 1)
                  output = BITXOR(output, BITAND(BITSHIFT(poly, -k), 1) AND BITAND(BITSHIFT(regvalue, -(kc - k - 1)), 1))
               END FOR

               // Update hamming distance
               IF output != inp(idx + i)
                  hamming_distance = hamming_distance + 1
               END IF
            END FOR

            a = mtr(t + 1, next_state + 1)
            b = mtr(t, curstate + 1)

            // Update metric and previous states
            IF a > b + hamming_distance
               mtr(t + 1, next_state + 1) = b + hamming_distance
               previous_states[t + 1][next_state + 1] = {curstate, input_bits}
            END IF
         END FOR
      END FOR
```

```
    idx = idx + n
  END FOR

  // Initialize ans list
  ans = []

  // Trace back to find the optimal path
  temp = previous_states[(LENGTH(inp) / n) + 1][1]
  ans = [temp[2] ans]
  cur = LENGTH(inp) / n

  WHILE cur > 1
    temp = previous_states[cur][temp[1] + 1]
    ans = [temp[2] ans]
    cur = cur - 1
  END WHILE

  RETURN ans  // Return the decoded output
END FUNCTION
```

- Modulator :

```
FUNCTION modulator(encoded_message, sigma)
```

```
    // BPSK modulation: Mapping binary values to -1 or 1.
    s = 1 - 2 * encoded_message

    // Add Gaussian noise to the modulated signal
    modulated_op = s + sigma * RANDN(1, LENGTH(encoded_message))

    RETURN modulated_op  // Return the modulated signal with noise
END FUNCTION
```

- Main code :

```
// Initialize variables
EbNodB = 0:0.5:10
R = 1/2
k = 1
n = 2
kc = 3
practical_error = zeros(1, length(EbNodB))
theoretical_error = zeros(1, length(EbNodB))
soft_error = zeros(1, length(EbNodB))
idx = 1
idx2 = 1
N = 10000
```

```
// Loop through each EbNodB value
for j in EbNodB
   // Calculate EbNo and sigma
   EbNo = 10^(j/10)
   sigma = SQRT(1 / (2 * R * EbNo))

   // Calculate theoretical bit error rate (BER)
   BER_th = 0.5 * erfc(SQRT(EbNo))

   Nerrs = 0
   Nerr_soft = 0

   // Loop for N iterations
   for i in 1:N
      // Generate random message
      msg = RANDI([0 1], 1, 100)
      msg = [msg ZEROS(1, kc - 1)]

      // Encode the message
      encoded_array = encoding(kc, [5 7], msg)

      // Modulate the encoded message
      modulated_message = modulator(encoded_array, sigma)

      // Demodulate the modulated message
      demodulated_message = modulated_message < 0
```

```
        // Decode the demodulated message using hard decision
        decoded_message = decoding(kc, [5 7], demodulated_message)

        // Decode the demodulated message using soft decision
        soft_decoded_message = decoding_soft(kc, [5 7], modulated_message)

        // Calculate errors
        Nerr_soft = Nerr_soft + SUM(soft_decoded_message != msg)
        Nerrs = Nerrs + SUM(msg != decoded_message)
    end

    // Calculate error rates
    soft_error[idx] = Nerr_soft / (N * LENGTH(msg))
    practical_error[idx] = Nerrs / (N * LENGTH(msg))
    theoretical_error[idx2] = theoretical_error[idx2] + BER_th

    idx = idx + 1
    idx2 = idx2 + 1
end

// Plotting
semilogy(EbNodB, practical_error, 'LineWidth', 2.0)
HOLD ON
semilogy(EbNodB, theoretical_error, 'LineWidth', 2.0)
semilogy(EbNodB, soft_error, 'LineWidth', 2.0)
LEGEND('Hard error', 'Theoretical error', 'Soft error')
```

```
TITLE('kc=3 rate=1/2')

XLABEL('EbNo(dB)')

YLABEL('BER')

HOLD OFF
```