CT216

Introduction to Communication System

Prof. Yash Vasavada

Group 20

Student Name	Student ID
Harita rathod	202301211
Ishti patel	202301212
Prayag kalriya	202301213
Gautam modi	202301214
Jay patoliya	202301215
Archan maru	202301217
Meet gandhi	202301219
Ajaykumar rathod	202301221
Manan chhabhaya	202301222

CT216 – Introduction to Communication Systems

Convolutional Code: Analysis

Group 20

1 Transfer Function of a Convolutional Code

The distance and error performance of a convolutional code can be analyzed using its state diagram. For linear codes, it's sufficient to study sequences generated from the all-zero input since their distance properties reflect all other cases.

To illustrate this, Fig. 1 shows a state diagram where each branch is labeled with D^i, indicating a Hamming distance of i from the all-zero output. The self-loop at node a is ignored as it doesn't affect distance calculations.

Node a is split into two parts: one for input and one for output, creating a five-node diagram.

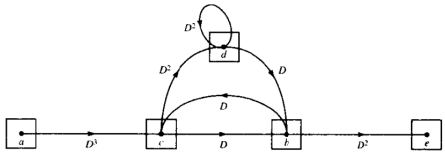


Figure 1: State diagram for rate $\frac{1}{3}$, $K = 3$ convolutional code

The state equations are:

$$X_c = D^3 X_a + D X_b$$
$$X_b = D X_c + D X_d$$
$$X_d = D^2 X_c + D^2 X_d$$
$$X_e = D^2 X_b$$

The transfer function for the code is defined as:

$$T(D) = \frac{X_e}{X_a}$$

By solving the above equations, we obtain:

$$T(D) = \frac{D^6}{1 - 2D^2}$$

Expanding the transfer function as a series:

$$T(D) = D^6 + 2D^8 + 4D^{10} + 8D^{12} + \ldots = \sum_{d=6}^{\infty} a_d D^d$$

where

$$a_d = \begin{cases} 2^{(d-6)/2}, & \text{if } d \text{ is even} \\ 0, & \text{if } d \text{ is odd} \end{cases}$$

The transfer function indicates there is a unique path with Hamming distance $d = 6$ from the all-zero path that remerges with it. From Fig. 1, the path for $d = 6$ is *acbe*. No other such path exists from a to e with this distance.

The second term of the transfer function expansion shows two paths with $d = 8$: *acdbe* and *acbce*. The third term shows four paths with $d = 10$, and so on. The minimum distance, or *free distance*, is denoted d_{free}, and for this code, $d_{\text{free}} = 6$.

To obtain more information beyond distance, variables J and N are introduced. Each branch with input bit 1 increases the exponent of N by 1. Each transition, regardless of input, increases the exponent of J by 1 to count total branches from node a to node e. The modified state diagram in Fig. 2 reflects this change.

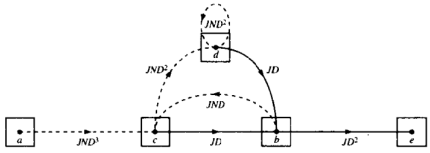


Figure 2: State diagram for rate $\frac{1}{3}$, $K = 3$ convolutional code

The state equations incorporating J and N become:

$$\begin{aligned} X_c &= JND^3X_a + JNDX_b \\ X_b &= JDX_c + JDX_d \\ X_d &= JND^2X_c + JND^2X_d \\ X_e &= JD^2X_b \end{aligned}$$

Solving for the ratio $\frac{X_e}{X_a}$, we get the transfer function:

$$T(D, N, J) = \frac{J^3 N D^6}{1 - JND^2(1 + J)}$$

Expanding this:

$$T(D, N, J) = J^3 N D^6 + J^4 N^2 D^8 + J^5 N^2 D^{10} + J^5 N D^{11} + 2J^6 N^3 D^{10} + J^7 N^3 D^{10} + \ldots$$

In the expansion of $T(D, N, J)$, each term provides information about the structure of the paths:

- The exponent of D represents the Hamming distance from the all-zero path.
- The exponent of N gives the number of input bits equal to 1.
- The exponent of J corresponds to the number of branches or path length.

For example, the first term in $T(D, N, J)$ corresponds to a path with $d = 6$, length 3, and one input bit equal to 1. Two $d = 8$ terms correspond to paths of lengths 4 and 5, with 2 ones in each case.

The factor J is useful when dealing with sequences of finite length m. The transfer function is truncated at J^m for finite-length sequences. For very long (infinite) sequences, we can eliminate the dependence on J by setting $J = 1$, giving:

$$T(D, N, 1) = T(D, N) = \frac{ND^6}{1 - 2ND^2}$$

Expanding this:

$$T(D, N) = ND^6 + 2N^2 D^8 + 4N^3 D^{10} + \ldots$$

$$= \sum_{d=6}^{\infty} a_d N^{(d-4)/2} D^d \qquad \text{(8-2-6)}$$

where the coefficients a_d are defined as in equation for a_d above.

The Viterbi Algorithm

Consider two trellis paths starting and ending at state a after three state transitions (three branches), corresponding to input sequences 000 and 100, and transmitted sequences 000 000 000 and 111 001 011, respectively.

Let the transmitted bits be $\{c_{jm}\}$, where $j = 1, 2, 3$ indicates the branch and $m = 1, 2, 3$ the bit within the branch. The demodulator outputs are $\{r_{jm}\}$.

If soft-decision decoding is used with binary coherent PSK transmission, the decoder input is:

$$r_{jm} = \sqrt{\mathcal{E}_c}(2c_{jm} - 1) + n_{jm} \tag{1}$$

where n_{jm} is additive noise and $\mathcal{E}_c$ is the signal energy per code bit.

A metric for the j-th branch of the i-th path is defined as the logarithm of the joint probability of the sequence $\{r_{jm},\ m = 1, 2, 3\}$.

The metric for the ith path consisting of B branches through the trellis is defined as

$$PM^{(i)} = \sum_{j=1}^{B} \mu_j^{(i)} \tag{2}$$

where

$$\mu_j^{(i)} = \log P(Y_j \mid C_j^{(i)}), \quad j = 1, 2, 3, \ldots \tag{3}$$

The criterion for selecting between two paths through the trellis is to choose the one with the larger metric, which maximizes the probability of correct decision or minimizes the probability of error.

Assuming hard-decision decoding and a received sequence {101 000 100}:

Let $i = 0$ denote the all-zero path, and $i = 1$ denote a path that merges with the all-zero path after three transitions. The metrics are:

$$\begin{aligned} PM^{(0)} &= 6\log(1-p) + 3\log p \\ PM^{(1)} &= 4\log(1-p) + 5\log p \end{aligned} \tag{4}$$

where p is the bit error probability. If $p < \frac{1}{2}$, then $PM^{(0)} > PM^{(1)}$, which aligns with the Hamming distance interpretation (all-zero path has $d = 3$, $i = 1$ path has $d = 5$).

For soft-decision decoding with AWGN, the demodulator output is modeled by:

$$p(r_{jm} \mid c_{jm}^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(r_{jm} - \sqrt{\mathcal{E}_c}(2c_{jm}^{(i)} - 1))^2}{2\sigma^2}\right] \tag{5}$$

where $\sigma^2 = \frac{1}{2}N_0$.

Neglecting common terms, the branch metric becomes:

$$\mu_j^{(i)} = \sum_{m=1}^{n} r_{jm}(2c_{jm}^{(i)} - 1) \tag{6}$$

For $n = 3$, the cumulative metrics are:

$$CM^{(0)} = \sum_{j=1}^{3}\sum_{m=1}^{3} r_{jm}(2c_{jm}^{(0)} - 1)$$
$$CM^{(1)} = \sum_{j=1}^{3}\sum_{m=1}^{3} r_{jm}(2c_{jm}^{(1)} - 1) \tag{7}$$

- For a binary convolutional code with $k = 1$ and constraint length K, the Viterbi algorithm uses 2^{K-1} states. This implies:
  - 2^{K-1} surviving paths at each trellis stage.
  - 2^{K-1} corresponding path metrics.

  More generally, for a code with k bits and K shift-register stages:
  - The trellis has $2^{k(K-1)}$ states.
  - Requires tracking $2^{k(K-1)}$ paths and computing $2^{k(K-1)}$ metrics.

  At each trellis stage:
  - 2^k paths merge at each node.
  - 2^k metrics are computed per node.
  - Only the most likely (minimum-distance) path is retained.

The decoding complexity of the Viterbi algorithm increases exponentially with the number of input bits k and the constraint length K. This exponential growth restricts the practical application of the Viterbi algorithm to codes with small values of k and K.

For long encoded sequences, the Viterbi algorithm introduces significant decoding delay and requires large memory to store all surviving paths, which makes it impractical for many real-world applications.

A common solution is to use a truncated version of the Viterbi algorithm that retains only the most recent δ decoded bits (symbols) in each surviving sequence. When a new bit is received, the decoder compares path metrics over the last δ branches and makes a final decision based on the highest metric.

If δ is chosen sufficiently large, all surviving paths will agree on the decoded bits δ branches back, allowing accurate decisions. Experimental results show that choosing $\delta \geq 5K$ leads to negligible performance degradation compared to the optimal Viterbi algorithm.

Now, we will explore the probability of error in soft-decision decoding using the Viterbi algorithm. The error probability is evaluated based on the branch and path metrics, assuming an all-zero codeword.

Viterbi Algorithm and Soft-Decision Decoding

The Viterbi algorithm with soft-decision decoding calculates the error probability using branch metrics derived from the received signals. These branch metrics are accumulated along trellis paths, and the path with the lowest accumulated metric is selected, assuming that the all-zero codeword was sent.

$$\mu_j^{(i)} = \sum_{m=1}^{n} r_{jm} \left(2c_{jm}^{(i)} - 1\right) \tag{1}$$

The path metric is computed as:

$$CM^{(i)} = \sum_{j=1}^{B} \mu_j^{(i)} = \sum_{j=1}^{B} \sum_{m=1}^{n} \left(2c_{jm}^{(i)} - 1\right) r_{jm} \tag{2}$$

For the all-zero path ($i = 0$), we have $c_{jm} = 0$ for all j and m. Thus, the path metric for the all-zero path becomes:

$$CM^{(0)} = \sum_{j=1}^{B} \sum_{m=1}^{n} (-\sqrt{E_c} + \eta_{jm})(-1) = \sqrt{E_c} Bn + \sum_{j=1}^{B} \sum_{m=1}^{n} \eta_{jm} \tag{3}$$

First Event Error Probability

The first event error probability is defined as the probability that a competing path, which merges with the all-zero path at node B, has a metric greater than the all-zero path for the first time. This is denoted as $P_2(d)$.

- Suppose an incorrect path, denoted as $i = 1$, merges with the all-zero path and differs in d bits.
- For the correct path, $CM^{(i)} < CM^{(0)}$, but an error occurs when $CM^{(i)} \geq CM^{(0)}$.

Thus, the error probability is:

$$P_2(d) = P(CM^{(i)} \geq CM^{(0)}) = P(CM^{(i)} - CM^{(0)} \geq 0) \tag{4}$$

Expanding the metrics:

$$CM^{(0)} = \sum_{j=1}^{B} \sum_{m=1}^{n} r_{jm}(2c_{jm}^{(0)} - 1) \tag{5}$$

$$CM^{(i)} = \sum_{j=1}^{B} \sum_{m=1}^{n} r_{jm}(2c_{jm}^{(i)} - 1) \tag{6}$$

Substituting into the expression for $P_2(d)$:

$$P\left(\sum_{j=1}^{B} \sum_{m=1}^{n} r_{jm}(2c_{jm}^{(i)} - 1) - \sum_{j=1}^{B} \sum_{m=1}^{n} r_{jm}(2c_{jm}^{(0)} - 1) \geq 0\right) \tag{7}$$

Simplifying:

$$P\left(2\sum_{j=1}^{B} \sum_{m=1}^{n} r_{jm}(c_{jm}^{(i)} - c_{jm}^{(0)}) \geq 0\right) \tag{8}$$

Since only d bits differ between the paths, the expression becomes:

$$P\left(\sum_{l=1}^{d} r'_l \geq 0\right) \tag{9}$$

Statistics of r'_l

- l runs over the d bits where the two paths differ.
- For the all-zero path, $c_{jm} = 0$ for all j and m.
- The expectation of r'_l is:

$$E\{r'_l\} = E\left\{\sqrt{\varepsilon_c}(0 - 1) + n_{jm}\right\} = -\sqrt{\varepsilon_c}$$

- The variance of r'_l is:

$$\mathrm{Var}\{r'_l\} = \frac{N_0}{2}$$

- Therefore, $\{r_l'\} \sim \mathcal{N}(-\sqrt{\varepsilon_c}, N_0/2)$, i.e., the r_l''s are identically and independently Gaussian distributed.

Using the Central Limit Theorem (CLT), the sum of the d bits follows a normal distribution:

$$\sum_{l=1}^{d} r_l' \sim \mathcal{N}\left(-d\sqrt{\varepsilon_c}, \frac{dN_0}{2}\right)$$

Converting to a standard normal variate $Z \sim \mathcal{N}(0,1)$:

$$P\left(\sum_{l=1}^{d} r_l' \geq 0\right) = P\left(Z \geq \sqrt{\frac{2\varepsilon_c d}{N_0}}\right)$$

This is equivalent to:

$$P\left(z \geq \sqrt{\frac{2\varepsilon_c d}{N_0}}\right)$$

The standard normal cumulative distribution function $Q(x)$ is defined as:

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{u^2}{2}} du$$

Thus, the first event error probability is:

$$P_2(d) = Q\left(\sqrt{\frac{2\varepsilon_c d}{N_0}}\right)$$

The signal-to-noise ratio (SNR) is given by:

$$SNR(\gamma_b) = \frac{\varepsilon_b}{N_0}$$

where the energy per bit ε_b is related to the symbol energy ε_c as:

$$\varepsilon_b = \frac{\varepsilon_c}{R_c}$$

Therefore, the SNR can be expressed as:

$$\gamma_b = \frac{\varepsilon_b}{N_0} = \frac{\varepsilon_c}{R_c N_0}$$

ε_b - Energy per bit

ε_c - Transmitted symbol energy

R_c - Code Rate

$$P_2(d) = Q\left(\sqrt{2\gamma_b R_c d}\right)$$

Although we have derived the first-event error probability for a path of distance d from the all-zero path, there are many possible paths with different distances that merge with the all-zero path at a given node B. The transfer function $T(D)$ describes all such paths and their distances. Summing the error probability over all possible distances gives an upper bound on the first-event error probability:

$$P_e \leq \sum_{d=d_{\text{free}}}^{\infty} a_d P_2(d) \leq \sum_{d=d_{\text{free}}}^{\infty} a_d Q\left(\sqrt{2\gamma_b R_c d}\right) \tag{10}$$

where a_d is the number of paths of distance d from the all-zero path that merge with it for the first time.

This is an upper bound because:

- The error events may overlap, meaning the probabilities are not disjoint.
- Summing for all $d \geq d_{\text{free}}$ assumes infinite code length. For practical truncated codes, the sum can be limited to $d_{\text{free}} \leq d \leq B$.

The Q-function can be upper-bounded by an exponential function to simplify the expression:

$$Q\left(\sqrt{2\gamma_b R_c d}\right) \leq e^{-\gamma_b R_c d} = D^d \quad \text{where } D = e^{-\gamma_b R_c} \tag{11}$$

Bit Error Probability

The first-event error probability gives a rough idea of code performance. But the bit error probability is more accurate and useful.

We can upper-bound bit error probability using first-event error:

$$P_e < T(D)\big|_{D=e^{-\gamma_b R_c}} \tag{12}$$

Here, errors happen when an incorrect path is chosen. Each wrong path causes some information bits to be decoded wrongly.

We count how many bits are wrong using the transfer function $T(D, N)$:

$$T(D, N) = \sum_{d=d_{\text{free}}}^{\infty} a_d D^d N^{f(d)} \tag{13}$$

To find the number of bit errors for each path, differentiate $T(D, N)$ w.r.t N, and set $N = 1$:

$$\left.\frac{dT(D,N)}{dN}\right|_{N=1} = \sum_{d=d_{\text{free}}}^{\infty} a_d f(d) D^d = \sum_{d=d_{\text{free}}}^{\infty} \beta_d D^d \tag{14}$$

Now we can bound the bit error probability using:

$$P_b \leq \sum_{d=d_{\text{free}}}^{\infty} \beta_d P_2(d) < \sum_{d=d_{\text{free}}}^{\infty} \beta_d Q\left(\sqrt{2\gamma_b R_c d}\right) \tag{15}$$

This gives a clear upper bound for P_b, based on path distances and number of bit errors. If the Q function is upper-bounded by an exponential, then bit error probability becomes:

$$P_b < \sum_{d=d_{\text{free}}}^{\infty} \beta_d D^d \Big|_{D=e^{-\gamma_b R_c}} = \left.\frac{dT(D,N)}{dN}\right|_{N=1, D=e^{-\gamma_b R_c}}$$

For $k > 1$, divide this result by k to get the average bit error probability.

This expression assumes binary coherent PSK is used. It also works for 4-phase PSK since it behaves like two binary PSKs.

For other modulations like FSK (coherent or not), just recalculate $P_2(d)$ depending on the modulation used. The rest of the expression remains unchanged.

This approach extends to nonbinary convolutional codes, where each symbol maps to a different waveform. The coefficients β_d still represent the number of symbol errors at distance d.

Let $P_2(d)$ be the pairwise symbol error probability between two paths separated by d symbols. Then the symbol error probability is given by:

$$P_M \leq \sum_{d=d_{\text{free}}}^{\infty} \beta_d P_2(d)$$

For 2^k symbols, the bit error probability is:

$$P_b = P_M \cdot \frac{2^k - 1}{2^k}$$

+

Hard Decision Decoding of Convolutional Codes

(Based on Proakis - Digital Communications)

Overview

The metrics in the Viterbi algorithm for convolutional code decoding are the Hamming distances between the received sequence and $2^{K(k-1)}$ surviving sequences in the trellis. This explanation focuses on hard decision decoding.

We assume the use of hard decision decoding, where each received bit is assumed to be either a 0 or a 1 (no soft information like likelihood or confidence). Let all-zero sequence be transmitted.

Error Event Probability $P_2(d)$

Case 1: When d is odd

$$P_2(d) = \sum_{k=\lceil \frac{d+1}{2} \rceil}^{d} \binom{d}{k} p^k (1-p)^{d-k} \tag{16}$$

Case 2: When d is even

$$P_2(d) = \sum_{k=\frac{d}{2}+1}^{d} \binom{d}{k} p^k (1-p)^{d-k} + \frac{1}{2}\binom{d}{d/2} p^{d/2}(1-p)^{d/2} \tag{17}$$

Here, p is the bit flip probability in the binary symmetric channel (BSC). If the number of errors is greater than $\lfloor \frac{d-1}{2} \rfloor$, the bits cannot be corrected and produce an error.

Upper Bound:

$$P_2(d) \leq \sum_{k=\lceil d/2 \rceil}^{d} \binom{d}{k} p^k (1-p)^{d-k} \leq 2^d p^{d/2}(1-p)^{d/2} \Rightarrow P_2(d) < [4p(1-p)]^{d/2} \tag{18}$$

First Event Error Probability P_e

Summing over all $d \geq d_{\text{free}}$, the overall first event probability is:

$$P_e < \sum_{d=d_{\text{free}}}^{\infty} a_d [4p(1-p)]^{d/2} \tag{19}$$

Transfer Function $T(D, N)$

Define the transfer function:

$$T(D, N) = \sum_{d=d_{\text{free}}}^{\infty} \sum_{k} a_{d,k} D^d N^k \tag{20}$$

Let $f(d)$ denote the exponent of N as a function of d. Then:

$$P_e < T(D,N)\Big|_{N=1, D=\sqrt{4p(1-p)}} \tag{21}$$

Taking derivative with respect to N:

$$\frac{dT(D,N)}{dN}\Big|_{N=1} = \sum_{d=d_{\text{free}}}^{\infty} a_d D^d f(d) \tag{22}$$

Bit Error Probability P_b

1.1 Hard Decision Decoding

In hard decision decoding, the probability of a pairwise error when the transmitted codeword has Hamming weight d can be expressed as:

$$P_2(d) = \sum_{k=\lceil \frac{d+1}{2} \rceil}^{d} \binom{d}{k} p^k (1-p)^{d-k} \tag{23}$$

This summation accounts for the fact that at least half the bits must be in error for the decoder to choose an incorrect path.

An upper bound for this probability is given by:

$$P_2(d) < [4p(1-p)]^{d/2} \tag{24}$$

Using the union bound, we can derive an upper bound for the bit error rate (BER) as:

$$P_b < \sum_{d=d_{\text{free}}}^{\infty} \beta_d P_2(d) \tag{25}$$

To simplify analysis, we can also relate this to the derivative of the transfer function:

$$P_b < \frac{dT(D,N)}{dN}\Big|_{N=1,\, D=\sqrt{4p(1-p)}} \tag{26}$$

Now, to better reflect practical systems, we use the Q-function to estimate the bit error probability p for a BSC-equivalent channel over AWGN:

$$p = Q\left(\sqrt{2\gamma_b R_c}\right) \tag{27}$$

Substituting this into $P_2(d)$ gives:

$$P_2(d) = \sum_{k=\lceil \frac{d+1}{2} \rceil}^{d} \binom{d}{k} \left(Q\left(\sqrt{2\gamma_b R_c}\right)\right)^k \left(1 - Q\left(\sqrt{2\gamma_b R_c}\right)\right)^{d-k} \tag{28}$$

Since summing up to infinity isn't practical, we truncate the sum to a few dominant terms (e.g., 6 terms beyond d_{free}), which still gives a good approximation:

$$P_b < \sum_{d=d_{\text{free}}}^{d_{\text{free}}+6} \beta_d P_2(d) \tag{29}$$

The resulting BER curve for this approximation is shown below:

Example

Consider a rate $\frac{1}{2}$ convolutional code with generator polynomials $g_1 = (1,1,1)$ and $g_2 = (1,0,1)$ (octal: 7, 5). This code has a free distance $d_{\text{free}} = 5$ and corresponding $a_d = [0,0,0,0,2,5,12,\ldots]$.
Let the binary symmetric channel have $p = 0.01$. Then, for $d = 5$:

$$\begin{aligned} P_2(5) &= \sum_{k=3}^{5} \binom{5}{k} p^k (1-p)^{5-k} \\ &= \binom{5}{3}(0.01)^3(0.99)^2 + \binom{5}{4}(0.01)^4(0.99) + \binom{5}{5}(0.01)^5 \\ &\approx 0.000098 \end{aligned}$$

So the contribution to P_b from $d = 5$ is:

$$P_b^{(5)} < B_5 P_2(5) = 2f(5) \cdot 0.000098$$

Soft decision

For a rate $R_c = 1/2$ convolutional code, the bit error probability using soft decision Viterbi decoding is upper-bounded as:

$$P_b < \sum_{d=d_{\text{free}}}^{\infty} \beta_d Q\left(\sqrt{2\gamma_b R_c d}\right)$$

Here, β_d is the average number of bit errors for paths at Hamming distance d from the correct path. Using the union bound, this can be approximated

as:

$$P_b < \sum_{d=d_{\text{free}}}^{\infty} \beta_d e^{-\gamma_b R_c d}$$

To compute this, we use the **transfer function** of the code. For example, the transfer function for a $(5, 7)$ code is:

$$T(D, N) = D^5 N + 2D^6 N^2 + 4D^7 N^3 + \cdots$$

Differentiating with respect to N and evaluating at $N = 1$ gives the total bit errors per distance:

$$\left.\frac{dT(D, N)}{dN}\right|_{N=1} = D^5 + 4D^6 + 12D^7 + 32D^8 + \cdots$$

This gives the coefficients β_d directly:

$$\beta_5 = 1, \quad \beta_6 = 4, \quad \beta_7 = 12, \quad \beta_8 = 32, \quad \ldots$$

So the approximate BER becomes:

$$P_b < \sum_{d=d_{\text{free}}}^{d_{\text{free}}+6} \beta_d e^{-\gamma_b R_c d}$$

This truncated bound captures the dominant error events and provides a tight and practical estimate of performance. The resulting BER curve is shown in the figure below.

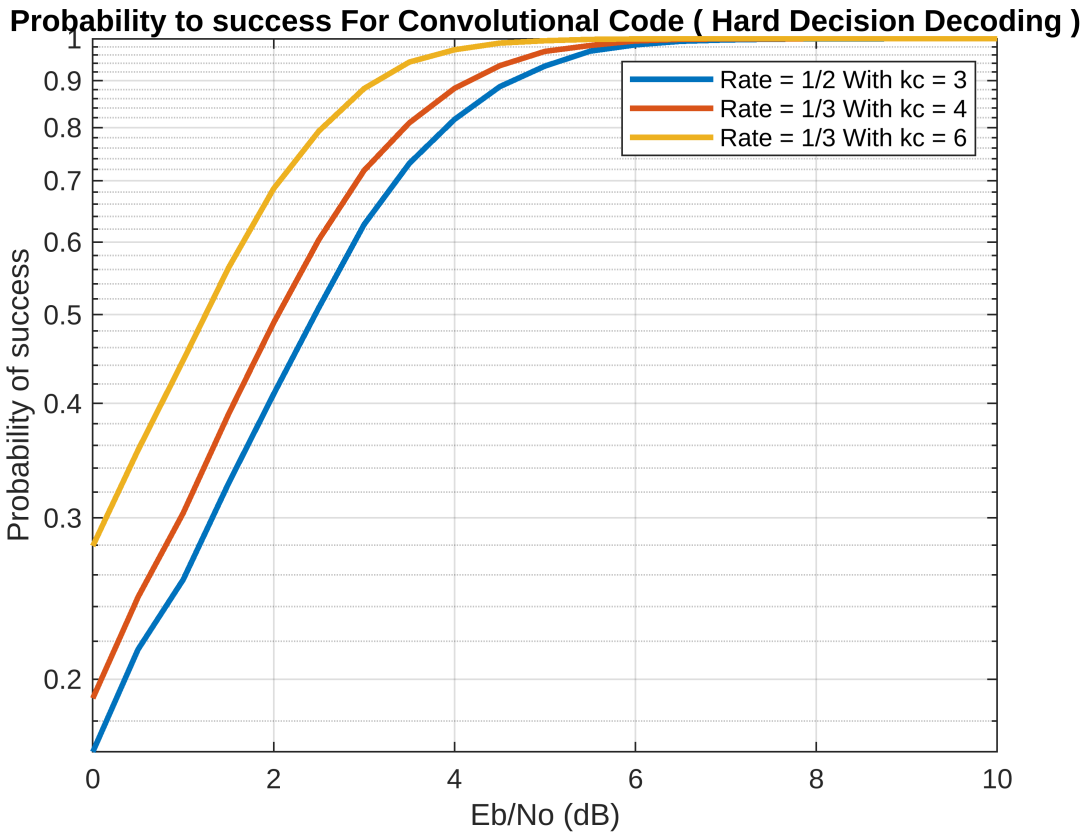
Conclusion

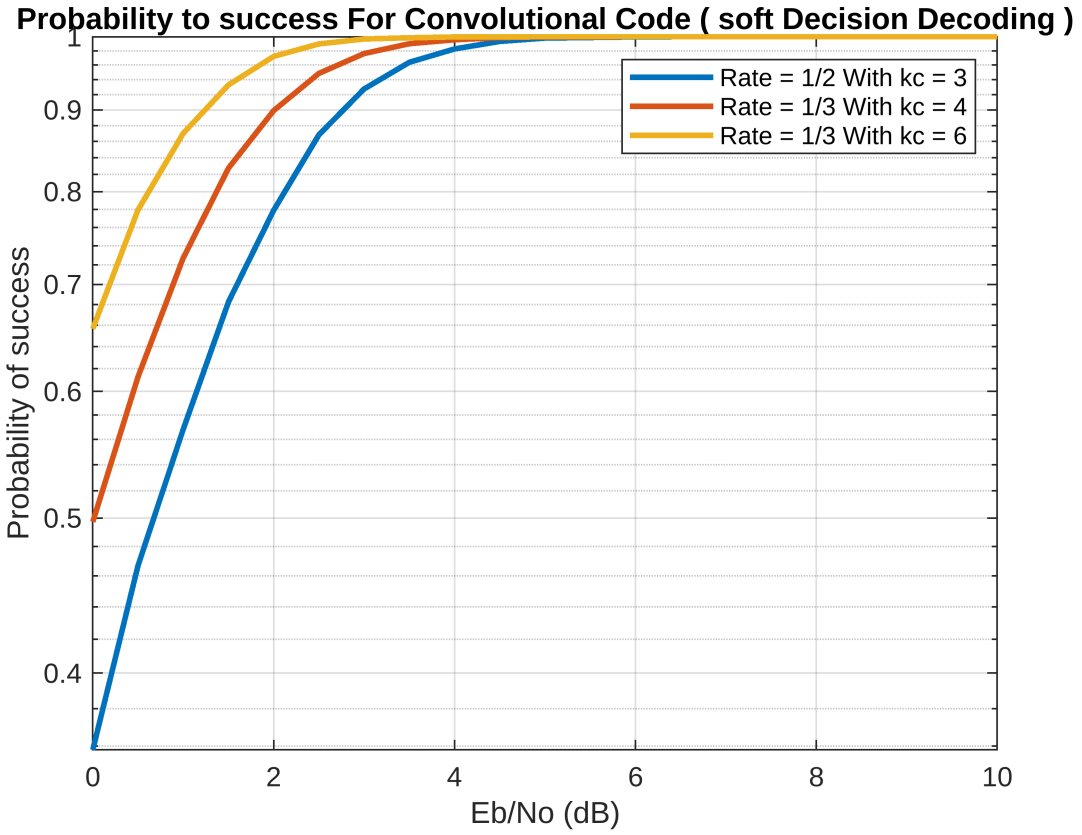
Hard decision decoding simplifies decoding by using binary decisions but suffers from slightly worse performance compared to soft decision decoding. The performance can still be bounded effectively using union bounds and transfer function techniques as described in Proakis' *Digital Communications*.

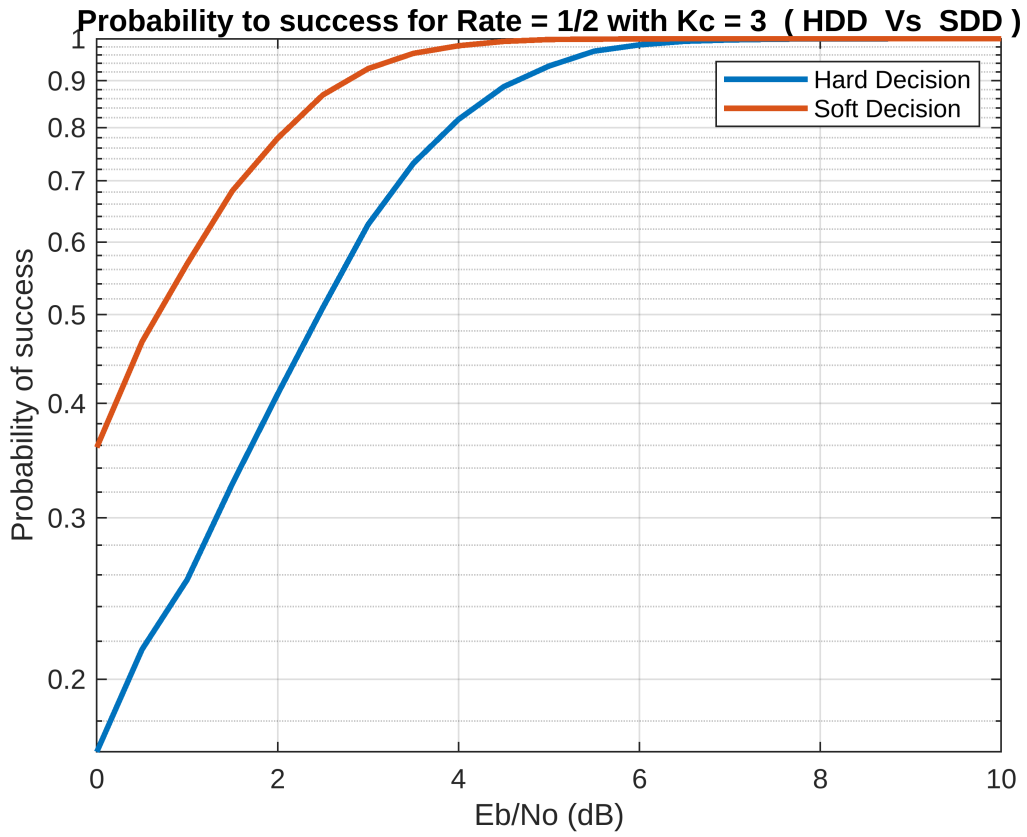
Group - 20 Convolution Code (MATLAB)

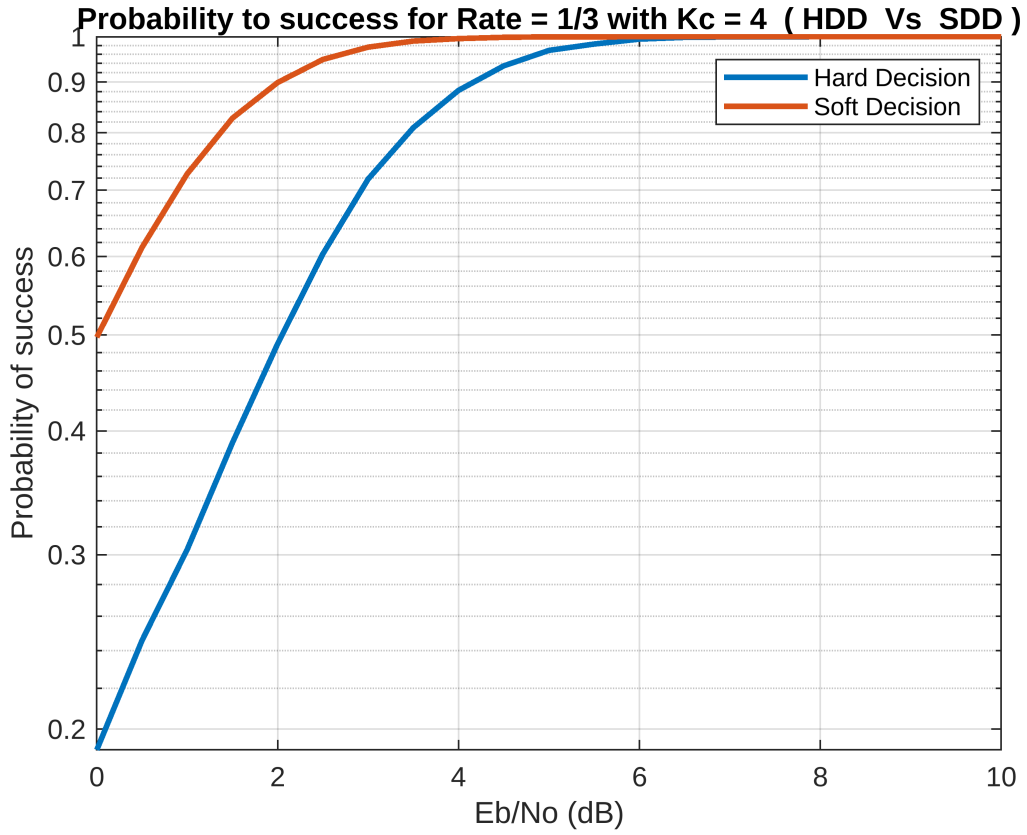
```
%defining parameter
KC = [3,4,6]; % constraint lenght
N = [2,3,3]; % number of ouput bits per input + prev bit
G = {[5,7],[13,15,17],[47,53,75]}; % generator polynomials

errorPerformance(0:0.5:10, 10000,100, KC, N, G);
```

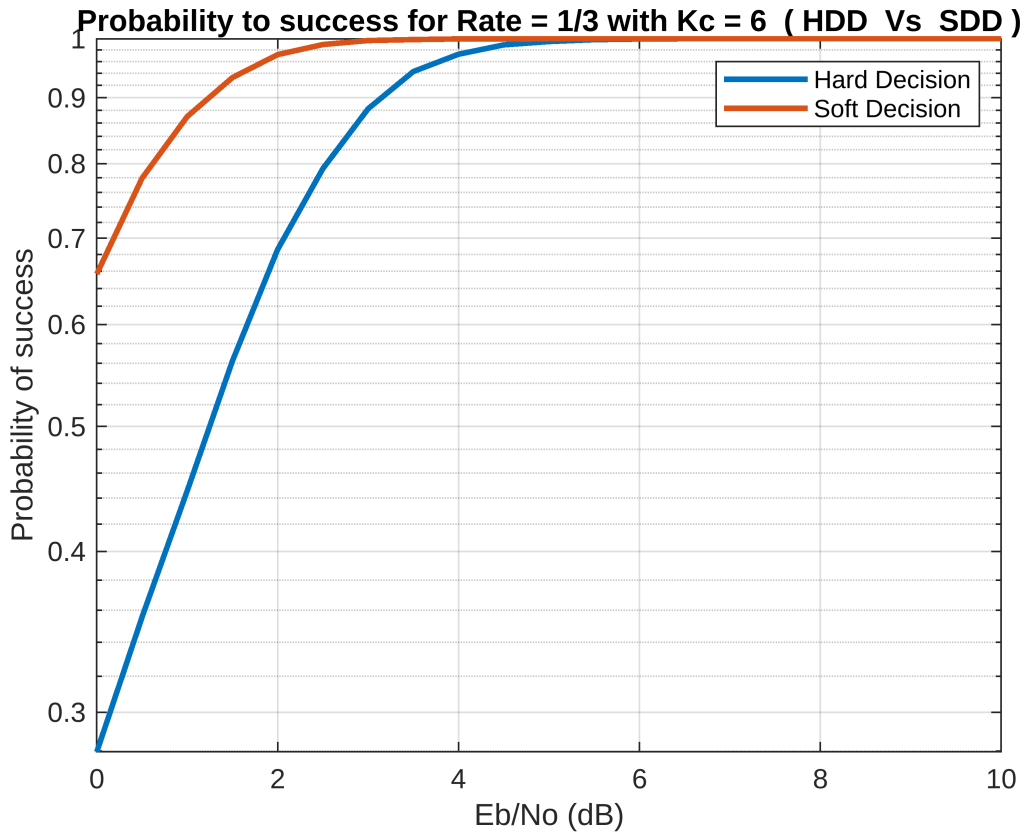
```
EbNo_loop =
0
EbNo_loop =
0.5000
EbNo_loop =
1
EbNo_loop =
1.5000
EbNo_loop =
2
EbNo_loop =
2.5000
EbNo_loop =
3
EbNo_loop =
3.5000
EbNo_loop =
4
EbNo_loop =
4.5000
EbNo_loop =
5
EbNo_loop =
5.5000
EbNo_loop =
6
EbNo_loop =
6.5000
EbNo_loop =
7
EbNo_loop =
7.5000
EbNo_loop =
8
EbNo_loop =
8.5000
EbNo_loop =
9
EbNo_loop =
9.5000
EbNo_loop =
10
```

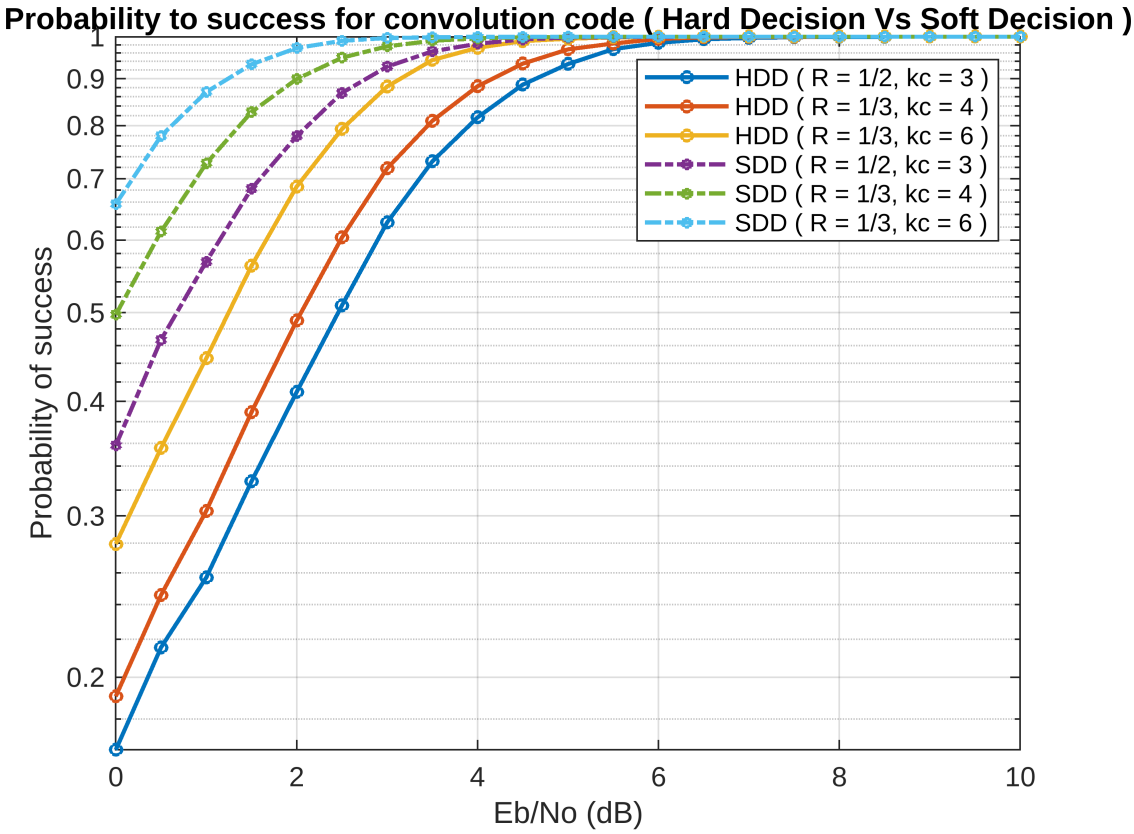

Probability to success For Convolutional Code (Hard Decision Decoding)
Rate = 1/2 With kc = 3
Rate = 1/3 With kc = 4
Rate = 1/3 With kc = 6
Probability of success
Eb/No (dB)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10

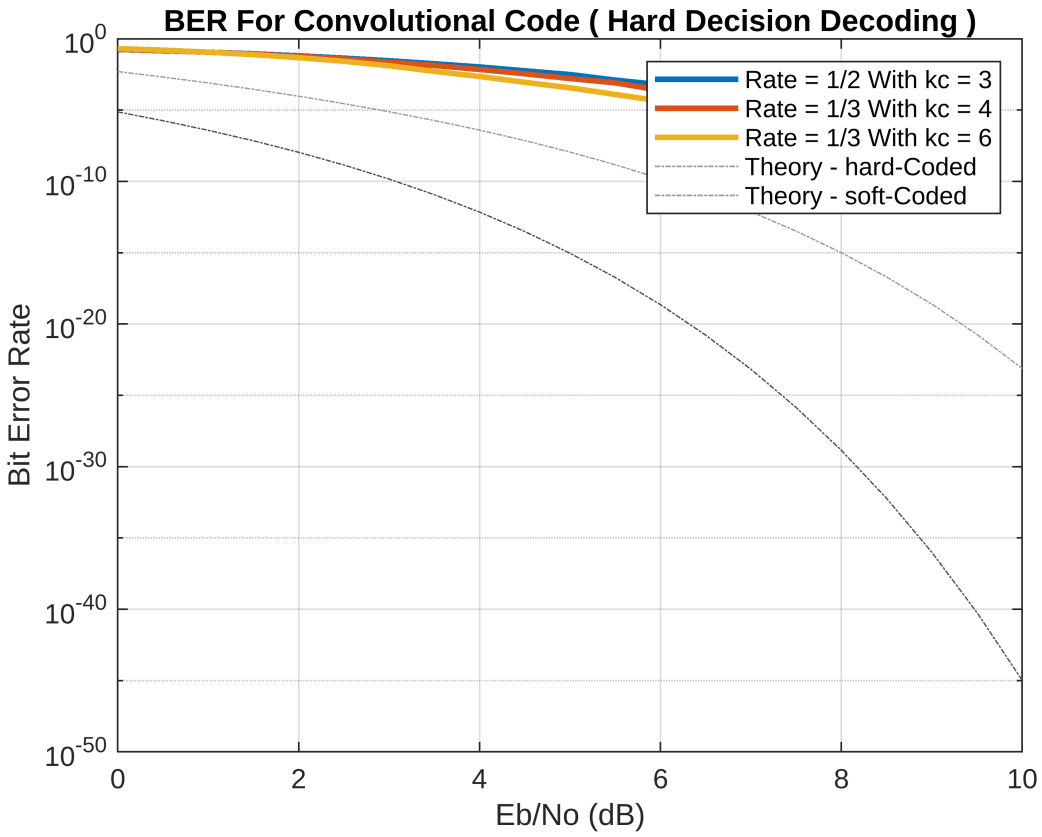

Probability to success For Convolutional Code (soft Decision Decoding)
Rate = 1/2 With kc = 3
Rate = 1/3 With kc = 4
Rate = 1/3 With kc = 6
Probability of success
Eb/No (dB)
1
0.9
0.8
0.7
0.6
0.5
0.4
0
2
4
6
8
10

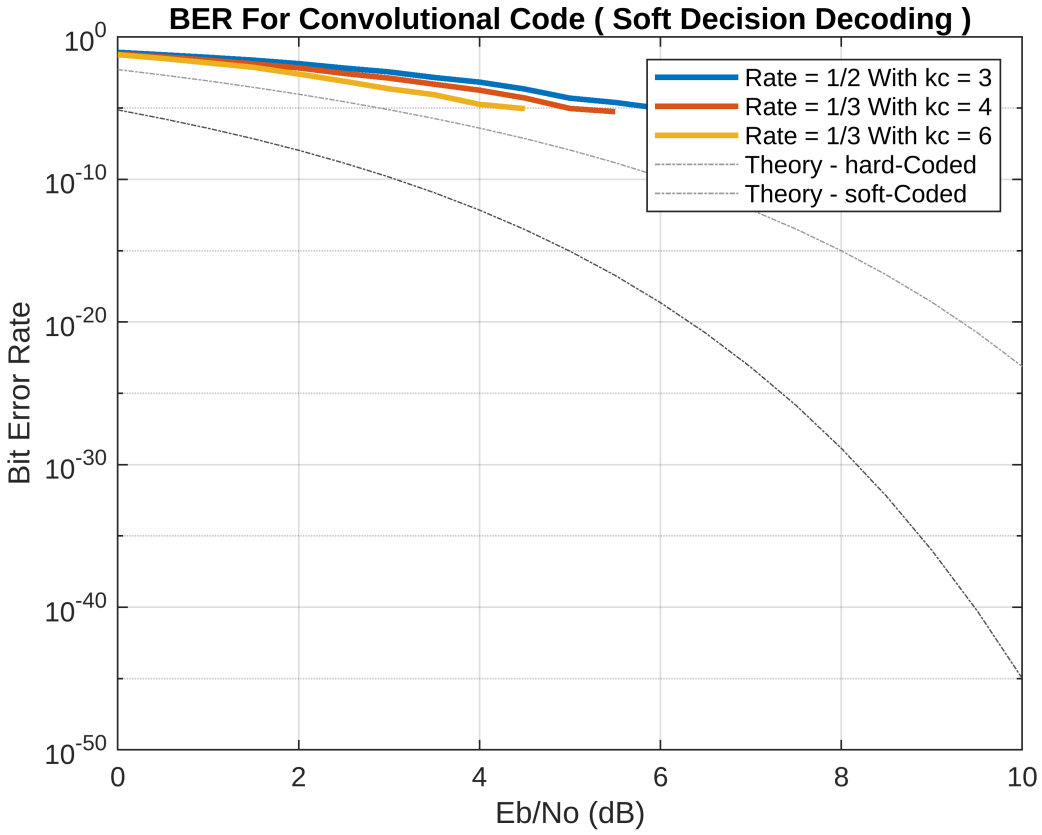

Probability to success for Rate = 1/2 with Kc = 3 (HDD Vs SDD)
Hard Decision
Soft Decision
Probability of success
Eb/No (dB)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10

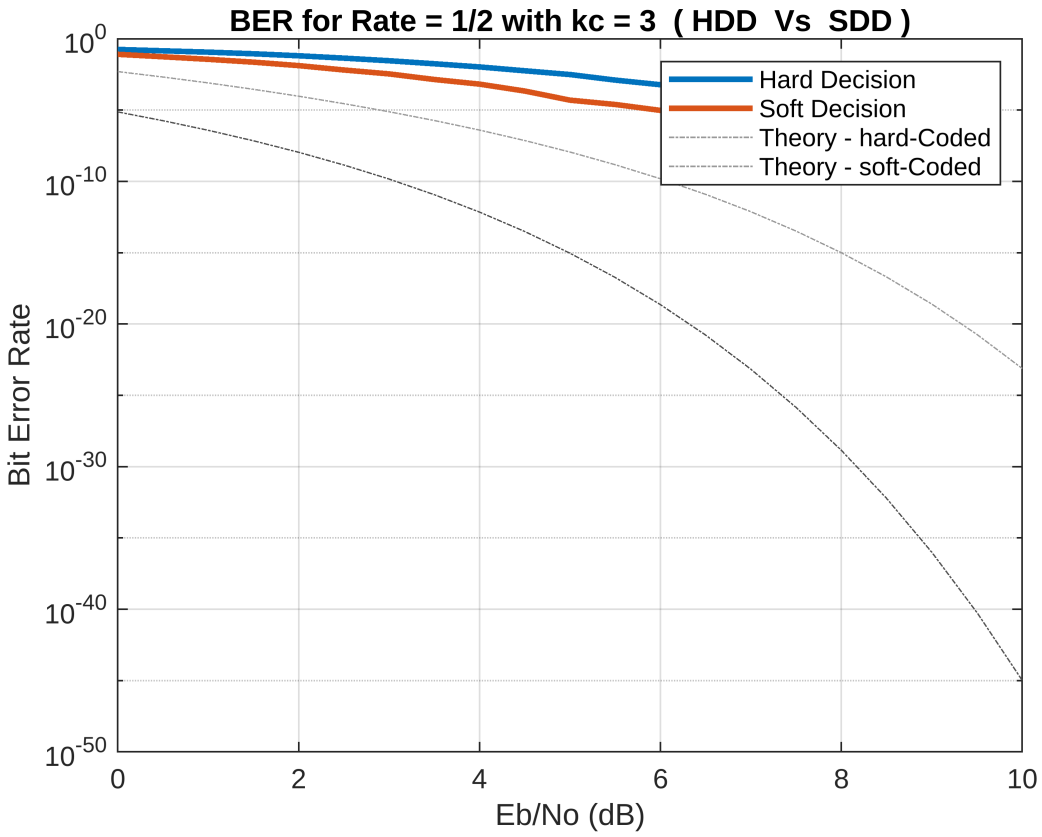

Probability to success for Rate = 1/3 with Kc = 4 (HDD Vs SDD)
Hard Decision
Soft Decision
Probability of success
Eb/No (dB)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10

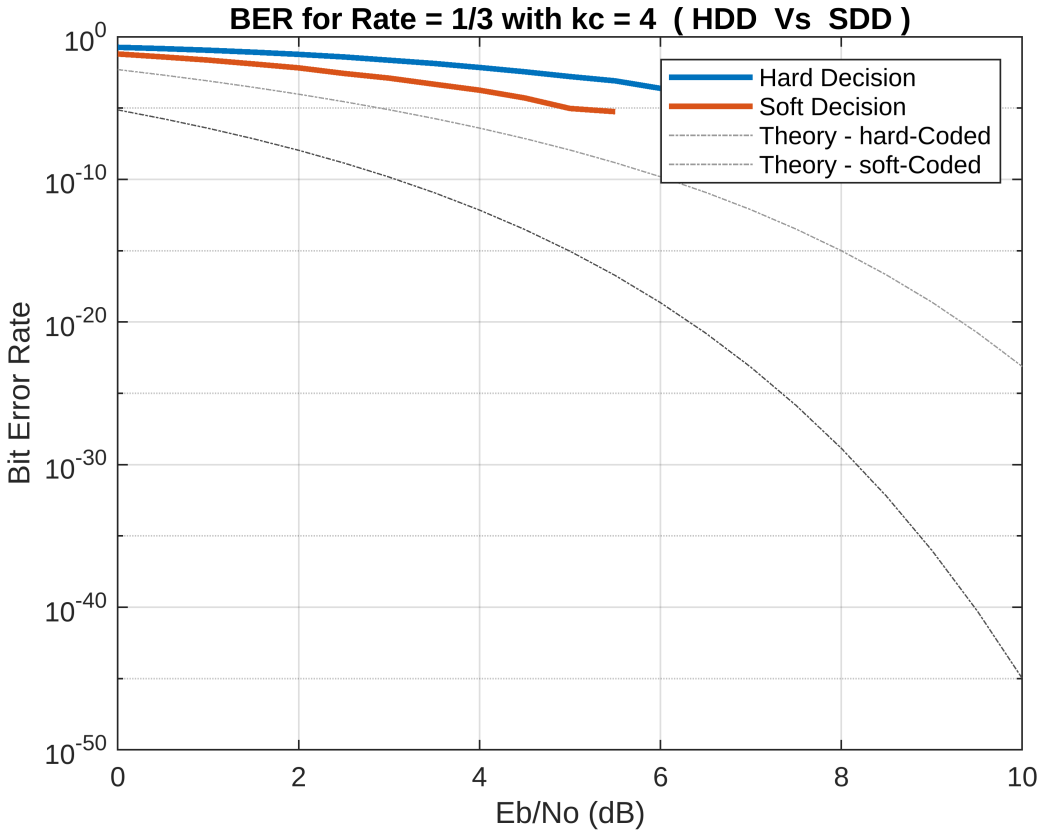

Probability to success for Rate = 1/3 with Kc = 6 (HDD Vs SDD)
Hard Decision
Soft Decision
Probability of success
Eb/No (dB)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
2
4
6
8
10

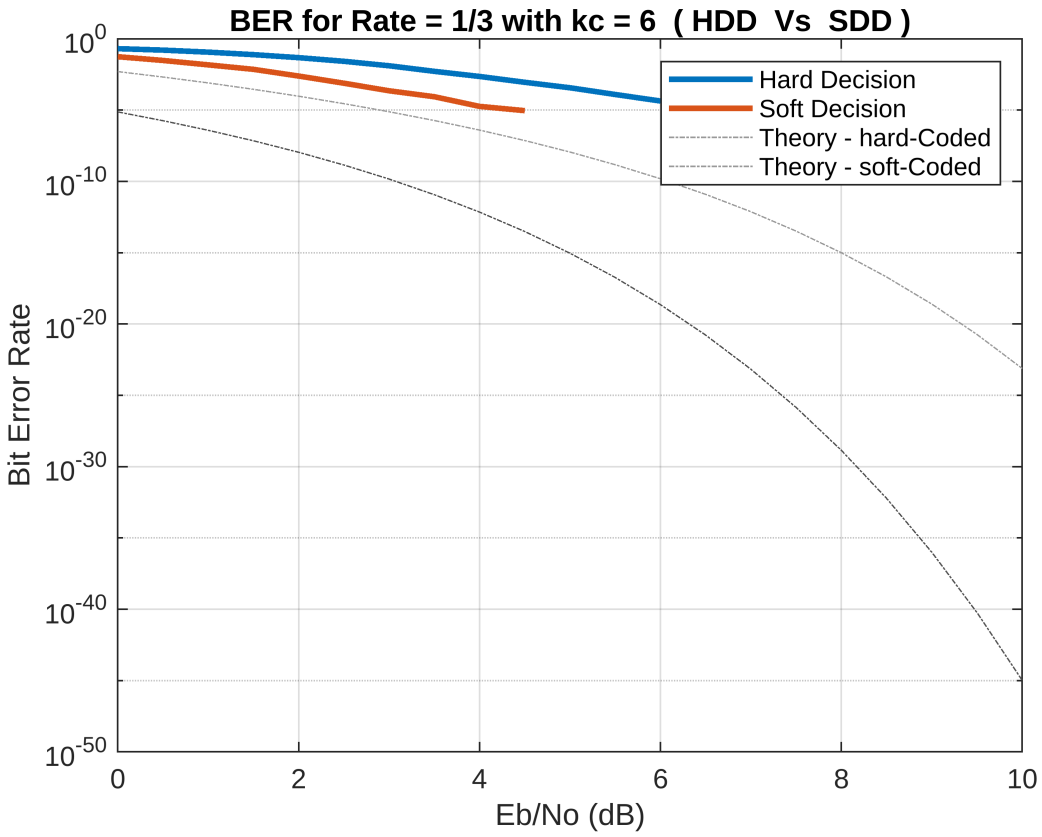

Probability to success for convolution code (Hard Decision Vs Soft Decision)
HDD (R = 1/2, kc = 3)
HDD (R = 1/3, kc = 4)
HDD (R = 1/3, kc = 6)
SDD (R = 1/2, kc = 3)
SDD (R = 1/3, kc = 4)
SDD (R = 1/3, kc = 6)
Probability of success
Eb/No (dB)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10

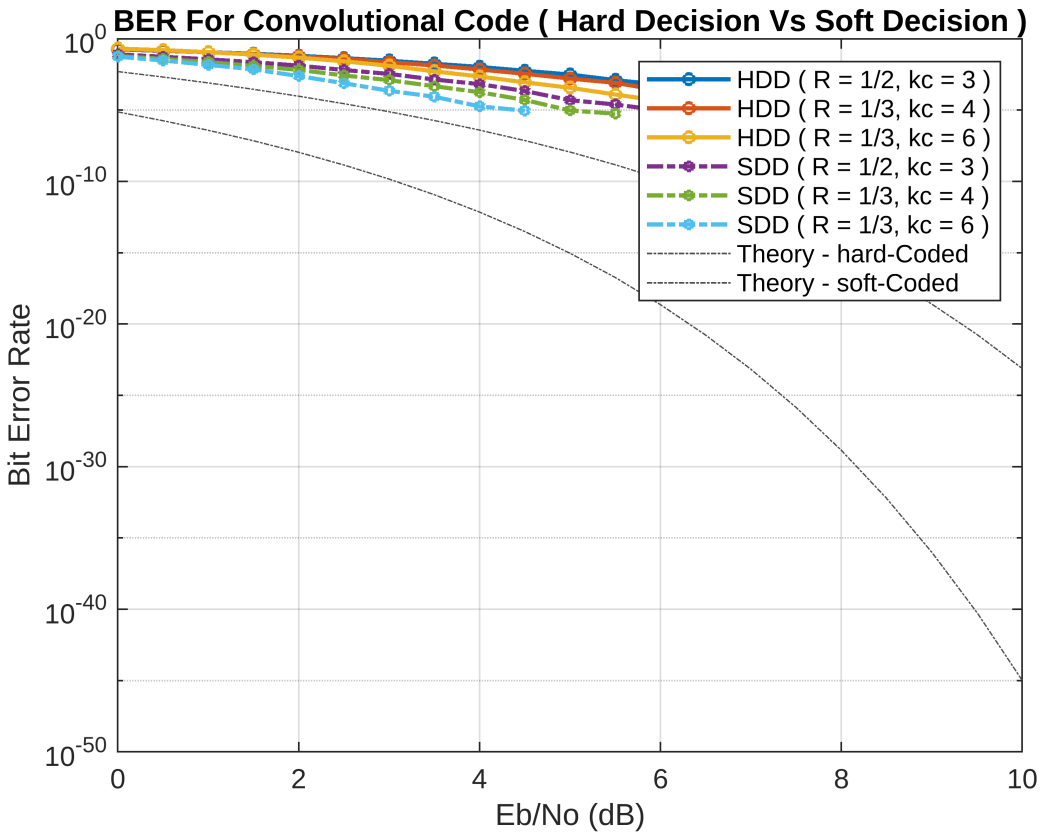

BER For Convolutional Code (Hard Decision Decoding)
Bit Error Rate
Eb/No (dB)
Rate = 1/2 With kc = 3
Rate = 1/3 With kc = 4
Rate = 1/3 With kc = 6
Theory - hard-Coded
Theory - soft-Coded

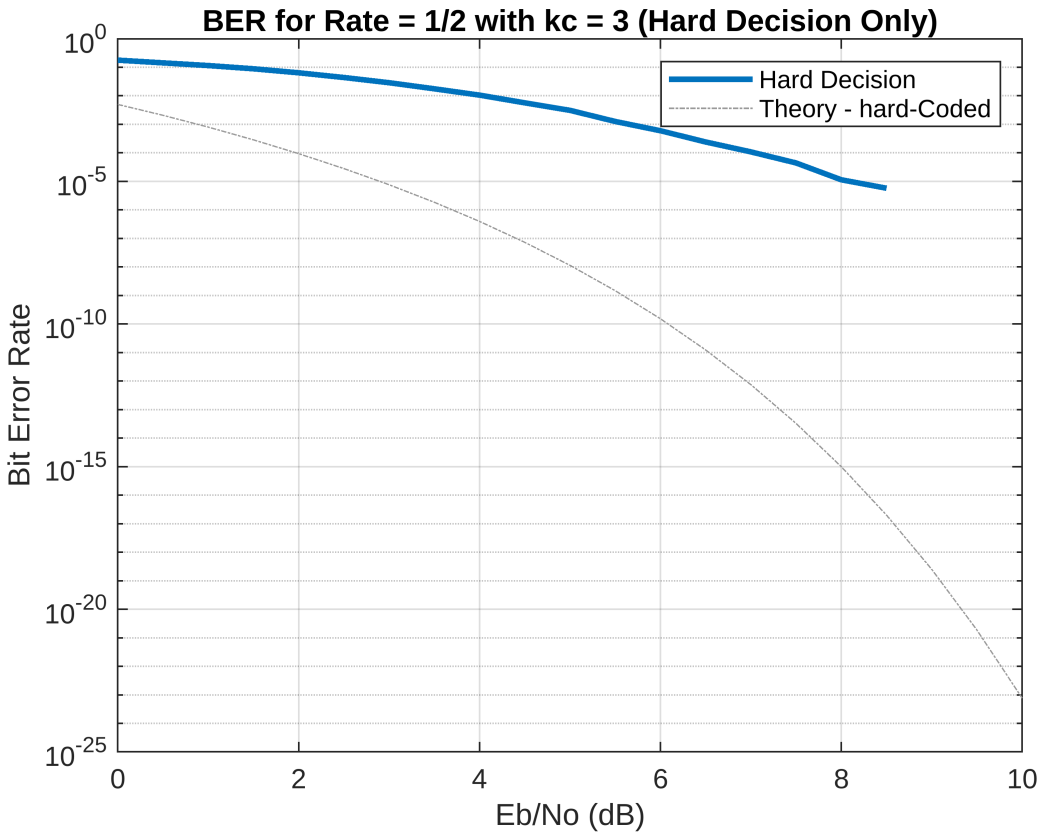

BER For Convolutional Code (Soft Decision Decoding)
Bit Error Rate
Eb/No (dB)
Rate = 1/2 With kc = 3
Rate = 1/3 With kc = 4
Rate = 1/3 With kc = 6
Theory - hard-Coded
Theory - soft-Coded


BER for Rate = 1/2 with kc = 3 (HDD Vs SDD)
Hard Decision
Soft Decision
Theory - hard-Coded
Theory - soft-Coded
Bit Error Rate
Eb/No (dB)
10^0
10^-10
10^-20
10^-30
10^-40
10^-50
0
2
4
6
8
10


BER for Rate = 1/3 with kc = 4 (HDD Vs SDD)
Hard Decision
Soft Decision
Theory - hard-Coded
Theory - soft-Coded
Bit Error Rate
Eb/No (dB)
10^0
10^-10
10^-20
10^-30
10^-40
10^-50
0
2
4
6
8
10


BER for Rate = 1/3 with kc = 6 (HDD Vs SDD)
Hard Decision
Soft Decision
Theory - hard-Coded
Theory - soft-Coded
Bit Error Rate
Eb/No (dB)


BER For Convolutional Code (Hard Decision Vs Soft Decision)
HDD (R = 1/2, kc = 3)
HDD (R = 1/3, kc = 4)
HDD (R = 1/3, kc = 6)
SDD (R = 1/2, kc = 3)
SDD (R = 1/3, kc = 4)
SDD (R = 1/3, kc = 6)
Theory - hard-Coded
Theory - soft-Coded
Bit Error Rate
Eb/No (dB)




```
function encoded_msg = Encoder(KC,N,G,Input_msg,prov_kc,prov_n)

value_of_g = [];
for i = 1  : length(KC)
    if KC(i) == prov_kc && N(i) == prov_n
        value_of_g = octal_to_binary(cell2mat(G(i)),prov_kc,prov_n);
        break;
    end
end
 idx = 1;

m = [Input_msg, zeros(1, prov_kc - 1)];
    shift_reg = zeros(1, prov_kc);
    encoded_msg = zeros(1, length(m) * prov_n);


    for i = 1:length(m)
        current_bit = m(i);
        shift_reg = [current_bit, shift_reg(1:end-1)];

        for j = 1 : prov_n
            encoded_bit = mod(sum(shift_reg .* value_of_g(j,:)), 2);
            encoded_msg(idx) = encoded_bit;
```

```
            idx = idx + 1;
        end
    end
end



function noisy_signal = BPSK_with_AWGN(message_bits, Sigma)

    bpsk_signal = 1 - 2*message_bits;

    noise = Sigma * randn(1,length(message_bits));

    noisy_signal = bpsk_signal + noise;
end


function decoded_msg = hard_decode(Mod_seq, prov_kc, prov_n, KC, N, G)
    % HARD_DECODE  Hard-decision Viterbi decoder with traceback pointers

    % Lookup/convert generator matrix if provided in octal form
    id = find(KC == prov_kc & N == prov_n, 1);
    if ~isempty(id)
        G = octal_to_binary(G{id}, prov_kc, prov_n);
    end

    % Convert received sequence (BPSK: <0 -> 1, >=0 -> 0)
    received_sequence = Mod_seq < 0;
    received_len      = length(received_sequence);

    % Trellis dimensions
    no_of_state  = 2^(prov_kc - 1);
    no_of_stages = received_len/prov_n + 1;

    % Initialize path metrics and traceback pointers
    smallest_path = inf(no_of_state, no_of_stages);
    PrevState     = zeros(no_of_state, no_of_stages);
    PrevInput     = zeros(no_of_state, no_of_stages);
    smallest_path(1,1) = 0;

    % Build state transition/output table
    state_table = zeros(no_of_state, 4); % [NS0, NS1, OUT0, OUT1]
    for state = 0:(no_of_state-1)
        for input = 0:1
            nxt_st = bitshift(state, -1);
            if input == 1
                nxt_st = nxt_st + 2^(prov_kc-2);
            end
            curr_reg = [input, dec2bin(state, log2(no_of_state)) - '0'];
            out_bin  = mod(curr_reg * G', 2);
```

```
            out_dec  = sum(out_bin .* 2.^(prov_n-1:-1:0));
            state_table(state+1, input+1) = nxt_st;
            state_table(state+1, input+3) = out_dec;
        end
    end

    % Forward pass: update smallest_path and store traceback pointers
    for stage = 1:(no_of_stages-1)
        idx      = (stage-1)*prov_n + (1:prov_n);
        rx_block = received_sequence(idx);
        for st = 1:no_of_state
            for input = 0:1
                nst        = state_table(st, input+1) + 1;
                out        = dec2bin(state_table(st, input+3), prov_n) - '0';
                dH         = sum(xor(rx_block, out));
                new_metric = smallest_path(st, stage) + dH;
                if new_metric < smallest_path(nst, stage+1)
                    smallest_path(nst, stage+1) = new_metric;
                    PrevState(nst, stage+1)     = st-1;
                    PrevInput(nst, stage+1)     = input;
                end
            end
        end
    end

    % Traceback: reconstruct decoded bits from pointers
    decoded = zeros(1, no_of_stages-1);
    myState = 0;
    for stage = no_of_stages:-1:2
        decoded(stage-1) = PrevInput(myState+1, stage);
        myState          = PrevState(myState+1, stage);
    end

    decoded_msg = decoded;
end



function decoded_msg = soft_decode(Mod_seq, prov_kc, prov_n, KC, N, G)
    % SOFT_DECODE  soft-decision Viterbi decoder with traceback pointers

    % Lookup/convert generator matrix if provided in octal form
    id = find(KC == prov_kc & N == prov_n, 1);
    if ~isempty(id)
        G = octal_to_binary(G{id}, prov_kc, prov_n);
    end

    received_sequence=Mod_seq;
    received_len      = length(received_sequence);
```

```
    % Trellis dimensions
    no_of_state  = 2^(prov_kc - 1);
    no_of_stages = received_len/prov_n + 1;

    % Initialize path metrics and traceback pointers
    smallest_path = inf(no_of_state, no_of_stages);
    PrevState     = zeros(no_of_state, no_of_stages);
    PrevInput     = zeros(no_of_state, no_of_stages);
    smallest_path(1,1) = 0;

    % Build state transition/output table
    state_table = zeros(no_of_state, 4); % [NS0, NS1, OUT0, OUT1]
    for state = 0:(no_of_state-1)
        for input = 0:1
            nxt_st = bitshift(state, -1);
            if input == 1
                nxt_st = nxt_st + 2^(prov_kc-2);
            end
            curr_reg = [input, dec2bin(state, log2(no_of_state)) - '0'];
            out_bin  = mod(curr_reg * G', 2);
            out_dec  = sum(out_bin .* 2.^(prov_n-1:-1:0));
            state_table(state+1, input+1) = nxt_st;
            state_table(state+1, input+3) = out_dec;
        end
    end

    % Forward pass: update smallest_path and store traceback pointers
    for stage = 1:(no_of_stages-1)
        idx      = (stage-1)*prov_n + (1:prov_n);
        msg = received_sequence(idx);
        for st = 1:no_of_state
            for input = 0:1
                nst        = state_table(st, input+1) + 1;
                out        = dec2bin(state_table(st, input+3), prov_n) - '0';
                ref_signal = 1 - 2 * out;  % Map bits to BPSK (+1, -1)
                dE = sum((msg - ref_signal).^2);  % Euclidean distance

                new_metric = smallest_path(st, stage) + dE;
                if new_metric < smallest_path(nst, stage+1)
                    smallest_path(nst, stage+1) = new_metric;
                    PrevState(nst, stage+1)     = st-1;
                    PrevInput(nst, stage+1)     = input;
                end
            end
        end
    end

    % Traceback: reconstruct decoded bits from pointers
    decoded = zeros(1, no_of_stages-1);
    myState = 0;
```

```
    for stage = no_of_stages:-1:2
        decoded(stage-1) = PrevInput(myState+1, stage);
        myState          = PrevState(myState+1, stage);
    end

    decoded_msg = decoded;
end




function errorPerformance(Eb_by_No_Range, N_sim, max_msg_size, KC, N, G)

practical_BER_hard = zeros(length(KC), length(Eb_by_No_Range));
practical_BER_soft = zeros(length(KC), length(Eb_by_No_Range));

prob_success_hard = zeros(length(KC), length(Eb_by_No_Range));
prob_success_soft = zeros(length(KC), length(Eb_by_No_Range));

theoretical_BER_coded_hard = zeros(length(KC), length(Eb_by_No_Range));
theoretical_BER_coded_soft = zeros(length(KC), length(Eb_by_No_Range));

idx = 1;

for EbNo_loop = Eb_by_No_Range
    EbNo = 10^(EbNo_loop/10); % Convert Eb/No from dB to linear scale

    % Distance spectrum (example; replace with actual for your code)
    d_free = 5;
    a_d = [1 5 14 30 58];
    d_range = d_free:(d_free + length(a_d) - 1);

    % Initialize BER values
    BER_th_hard = 0;
    BER_th_soft = 0;

    for id = 1:length(a_d)
        d = d_range(id);
        Bd = a_d(id);

        % Hard-decision union bound
        P2_d_hard = 0.5 * erfc(sqrt(d * EbNo));
        BER_th_hard = BER_th_hard + Bd * P2_d_hard;

        % Soft-decision union bound
        P2_d_soft = 0.5 * erfc(sqrt(2 * d * EbNo));
        BER_th_soft = BER_th_soft + Bd * P2_d_soft;
    end

    % Initialize accumulators
```

```
    bit_errors_hard = zeros(1, length(KC));  % total bit errors
    bit_errors_soft = zeros(1, length(KC));
    total_bits = zeros(1, length(KC));       % total transmitted bits
    count_success_messages_hard = zeros(1, length(KC));% messages with ≥1
error
    count_success_messages_soft = zeros(1, length(KC));

    for sim = 1:N_sim
        msg_size = randi([1, max_msg_size]); % Random message size
        msg = randi([0, 1], 1, msg_size);     % Generate random bits

        for it = 1:length(KC)
            kc = KC(it);
            n = N(it);

            R = 1/n;
            Sigma = sqrt(1 / (2 * R * EbNo));

            encoded = Encoder(KC, N, G, msg, kc, n);
            modulated = BPSK_with_AWGN(encoded, Sigma);

            decoded_hard = hard_decode(modulated, kc, n, KC, N, G);
            decoded_soft = soft_decode(modulated, kc, n, KC, N, G);

            % Padding
            msg_padded = [msg, zeros(1, kc - 1)];

            % Count errors
            errors_hard = sum(msg_padded ~= decoded_hard);
            errors_soft = sum(msg_padded ~= decoded_soft);

            % Accumulate
            bit_errors_hard(it) = bit_errors_hard(it) + errors_hard;
            bit_errors_soft(it) = bit_errors_soft(it) + errors_soft;

            total_bits(it) = total_bits(it) + length(msg_padded);

            if errors_hard == 0
                count_success_messages_hard(it) =
count_success_messages_hard(it) + 1;
            end
            if errors_soft == 0
                count_success_messages_soft(it) =
count_success_messages_soft(it) + 1;
            end
        end
    end

    % Compute practical BER
    practical_BER_hard(:, idx) = bit_errors_hard ./ total_bits;
```

```
    practical_BER_soft(:, idx) = bit_errors_soft ./ total_bits;

    % Compute frame error rate
    prob_success_hard(:, idx) = count_success_messages_hard / N_sim;
    prob_success_soft(:, idx) = count_success_messages_soft / N_sim;

    % Store theoretical uncoded BER
    theoretical_BER_coded_hard(:, idx) = BER_th_hard;
    theoretical_BER_coded_soft(:,idx) = BER_th_soft;

    idx = idx + 1;
    display(EbNo_loop);
end

    % Plot error curves for hard decision decoding
    figure;
    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, prob_success_hard(i, :), 'LineWidth', 2);
        if i == 1
            hold on;
        end
    end

    xlabel('Eb/No (dB)');
    ylabel('Probability of success');
    title('Probability to success For Convolutional Code ( Hard Decision
Decoding )');
    legends_practical = arrayfun(@(x) ['Rate = 1/', num2str(N(x)), ' With kc
= ', num2str(KC(x))], 1:length(KC), 'UniformOutput', false);
    legends = [legends_practical];
    legend(legends);
    grid on;
    hold off;

    % Plot error curves for soft decision decoding
    figure;
    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, prob_success_soft(i, :), 'LineWidth', 2);
        if i == 1
            hold on;
        end
    end
    xlabel('Eb/No (dB)');
    ylabel('Probability of success');
    title('Probability to success For Convolutional Code ( soft Decision
Decoding )');
    legends_practical = arrayfun(@(x) ['Rate = 1/', num2str(N(x)), ' With kc
= ', num2str(KC(x))], 1:length(KC), 'UniformOutput', false);
    legends = [legends_practical];
    legend(legends);
```

```
    grid on;
    hold off;

    % Plot both hard and soft decision error curves for each convolutional
encoding
    for i = 1:length(KC)
        figure;
        semilogy(Eb_by_No_Range, prob_success_hard(i, :), 'LineWidth', 2);
        hold on;
        semilogy(Eb_by_No_Range, prob_success_soft(i, :), 'LineWidth', 2);
        xlabel('Eb/No (dB)');
        ylabel('Probability of success');
        legend('Hard Decision', 'Soft Decision');
        title(['Probability to success for Rate = 1/', num2str(N(i)), ' with
Kc = ', num2str(KC(i)), '  ( HDD  Vs  SDD )']);
        grid on;
        hold off;
    end

    figure;
    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, prob_success_hard(i, :), 'LineStyle', '-',
'Marker', 'o', 'MarkerSize', 4, 'LineWidth', 1.5);
        if i == 1
            hold on;
        end
    end

    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, prob_success_soft(i, :), 'LineStyle', '-.',
'Marker', 'hexagram', 'MarkerSize', 4, 'LineWidth', 1.5);
    end

    xlabel('Eb/No (dB)');
    ylabel('Probability of success');
    title('Probability to success for convolution code ( Hard Decision Vs
Soft Decision )');

    legends_practical_hard = arrayfun(@(x) ['HDD ( R = 1/', num2str(N(x)),
', kc = ', num2str(KC(x)), ' )'], 1:length(KC), 'UniformOutput', false);
    legends_practical_soft = arrayfun(@(x) ['SDD ( R = 1/', num2str(N(x)),
', kc = ', num2str(KC(x)), ' )'], 1:length(KC), 'UniformOutput', false);
    legends = [legends_practical_hard, legends_practical_soft];
    legend(legends);
    grid on;
    hold off;

figure;
    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, practical_BER_hard(i, :), 'LineWidth', 2);
```

```
        if i == 1
            hold on;
        end
    end
    semilogy(Eb_by_No_Range, theoretical_BER_coded_hard, 'Color', [0.6, 0.6,
0.6], 'LineStyle', '-.', 'LineWidth', 0.5);
    semilogy(Eb_by_No_Range, theoretical_BER_coded_soft, 'Color', [0.3, 0.3,
0.3], 'LineStyle', '-.', 'LineWidth', 0.5);
    xlabel('Eb/No (dB)');
    ylabel('Bit Error Rate');
    title('BER For Convolutional Code ( Hard Decision Decoding )');
    legends_practical = arrayfun(@(x) ['Rate = 1/', num2str(N(x)), ' With kc
= ', num2str(KC(x))], 1:length(KC), 'UniformOutput', false);
    legends = [legends_practical, 'Theory - hard-Coded','Theory - soft-
Coded'];
    legend(legends);
    grid on;
    hold off;

    % Plot error curves for soft decision decoding
    figure;
    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, practical_BER_soft(i, :), 'LineWidth', 2);
        if i == 1
            hold on;
        end
    end
    semilogy(Eb_by_No_Range, theoretical_BER_coded_hard, 'Color', [0.6, 0.6,
0.6], 'LineStyle', '-.', 'LineWidth', 0.5);
    semilogy(Eb_by_No_Range, theoretical_BER_coded_soft, 'Color', [0.3, 0.3,
0.3], 'LineStyle', '-.', 'LineWidth', 0.5);
    xlabel('Eb/No (dB)');
    ylabel('Bit Error Rate');
    title('BER For Convolutional Code ( Soft Decision Decoding )');
    legends_practical = arrayfun(@(x) ['Rate = 1/', num2str(N(x)), ' With kc
= ', num2str(KC(x))], 1:length(KC), 'UniformOutput', false);
    legends = [legends_practical, 'Theory - hard-Coded','Theory - soft-
Coded'];
    legend(legends);
    grid on;
    hold off;

    % Plot both hard and soft decision error curves for each convolutional
encoding
    for i = 1:length(KC)
        figure;
        semilogy(Eb_by_No_Range, practical_BER_hard(i, :), 'LineWidth', 2);
        hold on;
        semilogy(Eb_by_No_Range, practical_BER_soft(i, :), 'LineWidth', 2);
```

```
        semilogy(Eb_by_No_Range, theoretical_BER_coded_hard, 'Color', [0.6,
0.6, 0.6], 'LineStyle', '-.', 'LineWidth', 0.5);
        semilogy(Eb_by_No_Range, theoretical_BER_coded_soft, 'Color', [0.3,
0.3, 0.3], 'LineStyle', '-.', 'LineWidth', 0.5);
        xlabel('Eb/No (dB)');
        ylabel('Bit Error Rate');
        legend('Hard Decision', 'Soft Decision', 'Theory - hard-
Coded','Theory - soft-Coded');
        title(['BER for Rate = 1/', num2str(N(i)), ' with kc = ',
num2str(KC(i)), '  ( HDD  Vs  SDD )']);
        grid on;
        hold off;
    end

    figure;
    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, practical_BER_hard(i, :), 'LineStyle', '-',
'Marker', 'o', 'MarkerSize', 4, 'LineWidth', 1.5);
        if i == 1
            hold on;
        end
    end

    for i = 1:length(KC)
        semilogy(Eb_by_No_Range, practical_BER_soft(i, :), 'LineStyle',
'-.', 'Marker', 'hexagram', 'MarkerSize', 4, 'LineWidth', 1.5);
    end

    semilogy(Eb_by_No_Range, theoretical_BER_coded_hard, 'Color', [0.3, 0.3,
0.3], 'LineStyle', '-.', 'LineWidth', 0.5);
    semilogy(Eb_by_No_Range, theoretical_BER_coded_soft, 'Color', [0.3, 0.3,
0.3], 'LineStyle', '-.', 'LineWidth', 0.5);
    xlabel('Eb/No (dB)');
    ylabel('Bit Error Rate');
    title('BER For Convolutional Code ( Hard Decision Vs Soft Decision )');

    legends_practical_hard = arrayfun(@(x) ['HDD ( R = 1/', num2str(N(x)),
', kc = ', num2str(KC(x)), ' )'], 1:length(KC), 'UniformOutput', false);
    legends_practical_soft = arrayfun(@(x) ['SDD ( R = 1/', num2str(N(x)),
', kc = ', num2str(KC(x)), ' )'], 1:length(KC), 'UniformOutput', false);
    legends = [legends_practical_hard, legends_practical_soft, 'Theory -
hard-Coded','Theory - soft-Coded'];
    legend(legends);
    grid on;
    hold off;

    % Plot BER for kc = 3 and rate = 1/2 (i.e., N(1) = 2, KC(1) = 3)
    figure;
    semilogy(Eb_by_No_Range, practical_BER_hard(1, :), 'LineWidth', 2);
    hold on;
```

```
    semilogy(Eb_by_No_Range, theoretical_BER_coded_hard, 'Color', [0.6, 0.6,
0.6], 'LineStyle', '-.', 'LineWidth', 0.5);
    xlabel('Eb/No (dB)');
    ylabel('Bit Error Rate');
    title(['BER for Rate = 1/', num2str(N(1)), ' with kc = ',
num2str(KC(1)), ' (Hard Decision Only)']);
    legend('Hard Decision', 'Theory - hard-Coded');
    grid on;
    hold off;

end


function binary_matrix = octal_to_binary(octal_vector, Kc, n)

    binary_matrix = zeros(n, Kc);

    for i = 1:n
        octal_number = octal_vector(i);
        decimal_number = oct2dec(octal_number);
        binary_string = dec2bin(decimal_number, Kc);
        binary_matrix(i, :) = binary_string - '0'; %character to digit
conversion
    end
end
```