# THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY ISDN 2602

**Laboratory 4: Source and Channel Coding (5%)**

**Daisy Cheung 21100358**

**Answer Sheet**

Please write down your answer here and submit your answer on GitHub by Wednesday (Oct 29th) 23:59

***Part I: Source Coding***

# Task 1 – Length of the bit streams

In this task, we will compare the lengths of the bit streams for four source coding algorithms applied to a black-and-white image: "raw" image encoding, run-length encoding with lengths encoded as 8-bit binary numbers, and run-length encoding with lengths encoded by Huffman coding with one or two dictionaries.

# Check Point:

1. Write down the lengths of the bit streams using “raw” image encoding and the run-length encoding. Is the run-length code better than the raw encoding? **Explain why**.

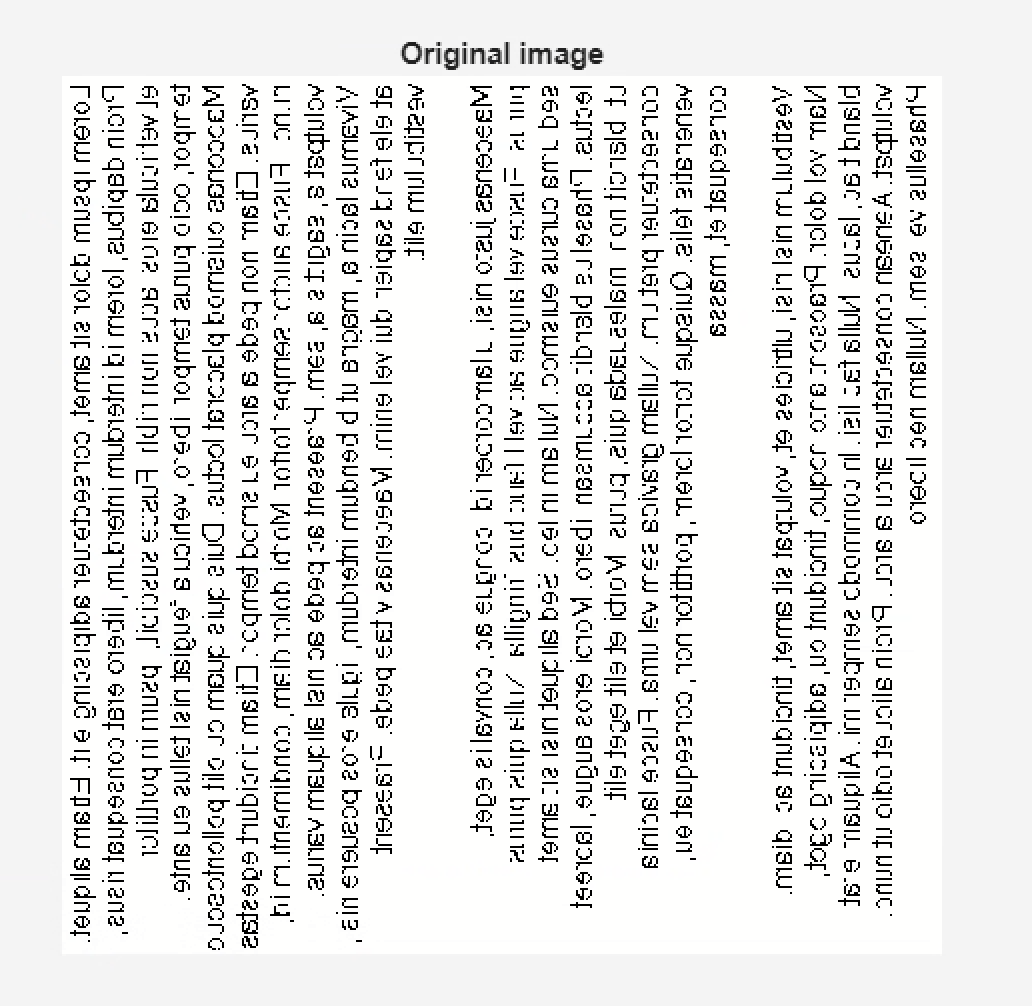
size\_raw\_data =  
 250000  
size\_run\_length =  
 301688  
size\_huffman =  
 117374 100981

Run length encoding is usually better if the image has long stretches of the same color pixels because it compresses runs into fewer bits

1. Type “help transpose” in the command window to learn how to perform matrix transpose operation on a matrix in MATLAB. Revise the MATLAB codes so that the image will be rotated along the diagonal. Then, write down and compare the lengths of the bitstreams for these four source coding algorithms before and after the rotation. **Explain why**.

size\_raw\_data =  
  
 250000  
  
size\_run\_length =  
  
 196680  
  
size\_huffman =  
  
 134892 120565

After transposing, if pixel values change more frequently along the new direction, I got shorter run lengths and a greater number of runs. This increased the encoding size because there are more run values to encode.



***Fill in the answers to the blanks and Show your result to the TA.***

# Task 2 – Huffman code

In this task, you will generate the Huffman code for a set of run-lengths, and use it to encode the run- lengths of black or white pixels. You will find that Huffman coding enables us to encode the sequence of run lengths using fewer bits than the standard 8-bit encoding.

# Check point:

1. Find an optimal dictionary to represent these 11 symbols using the symbol probabilities and the Huffman coding algorithm. Once you have found it, replace the value of **dict** defined between the line:

*% % % % Revise the following code to generate a valid and efficient dictionary % % % %*

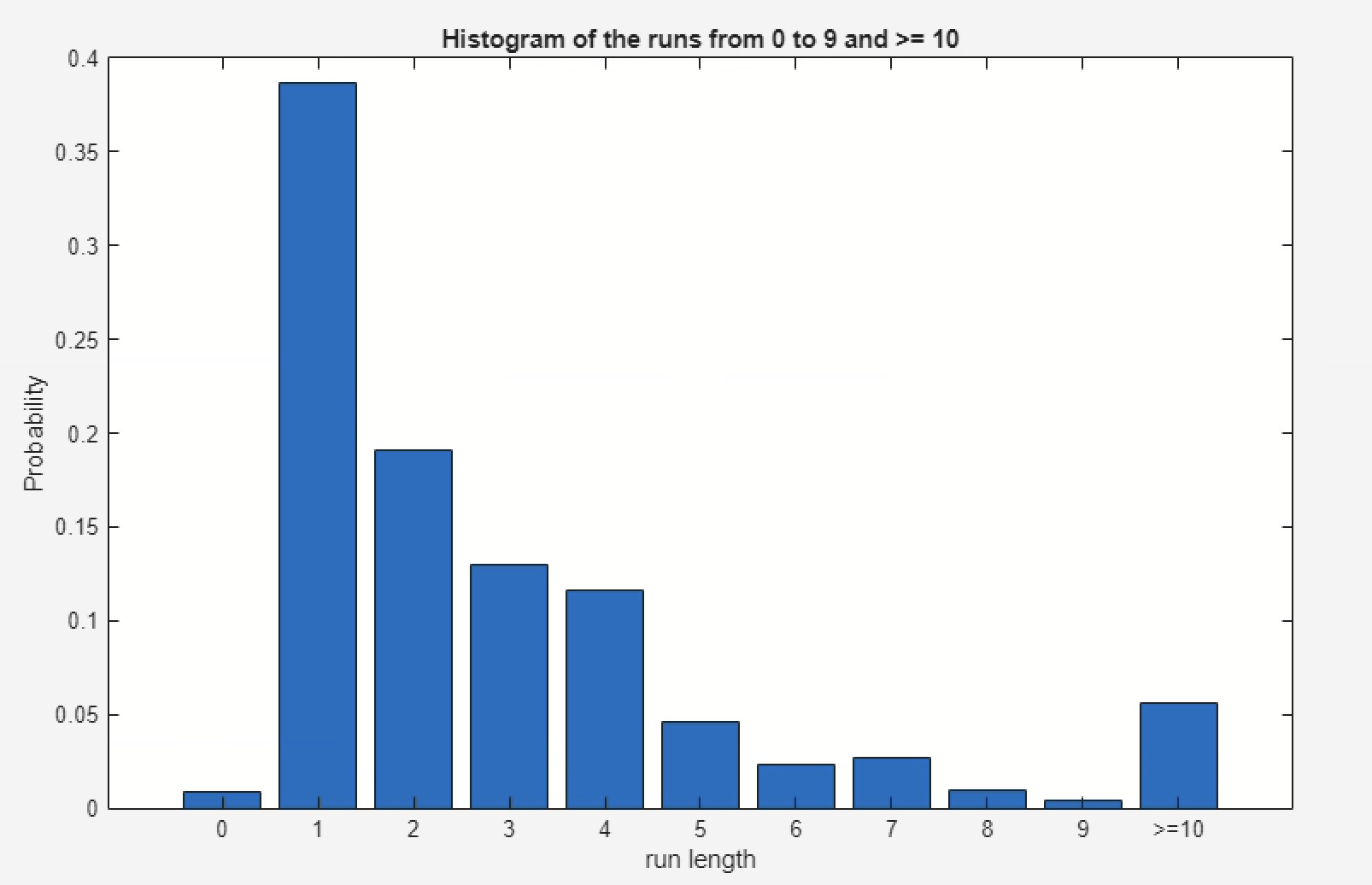
and

*% % % % Do not change the code below % % % %*

The remaining part of the code uses this dictionary to encode the run lengths, and to measure the length of the resulting bit stream. It also checks whether the dictionary is valid by reconstructing the image from the run lengths encoded by the dictionary using the function **huffman\_encode\_dict**. If your dictionary is correct, the original and reconstructed images should be the same and the **size\_huffman** should be equal to 117374.

prob =  
  
 0.0092 0.3870 0.1911 0.1294 0.1162 0.0461 0.0232 0.0274 0.0099 0.0040 0.0565

size\_raw\_data =  
  
 250000  
  
  
size\_huffman =  
  
 167884  
  
  
size\_reconstructed =  
  
 250000



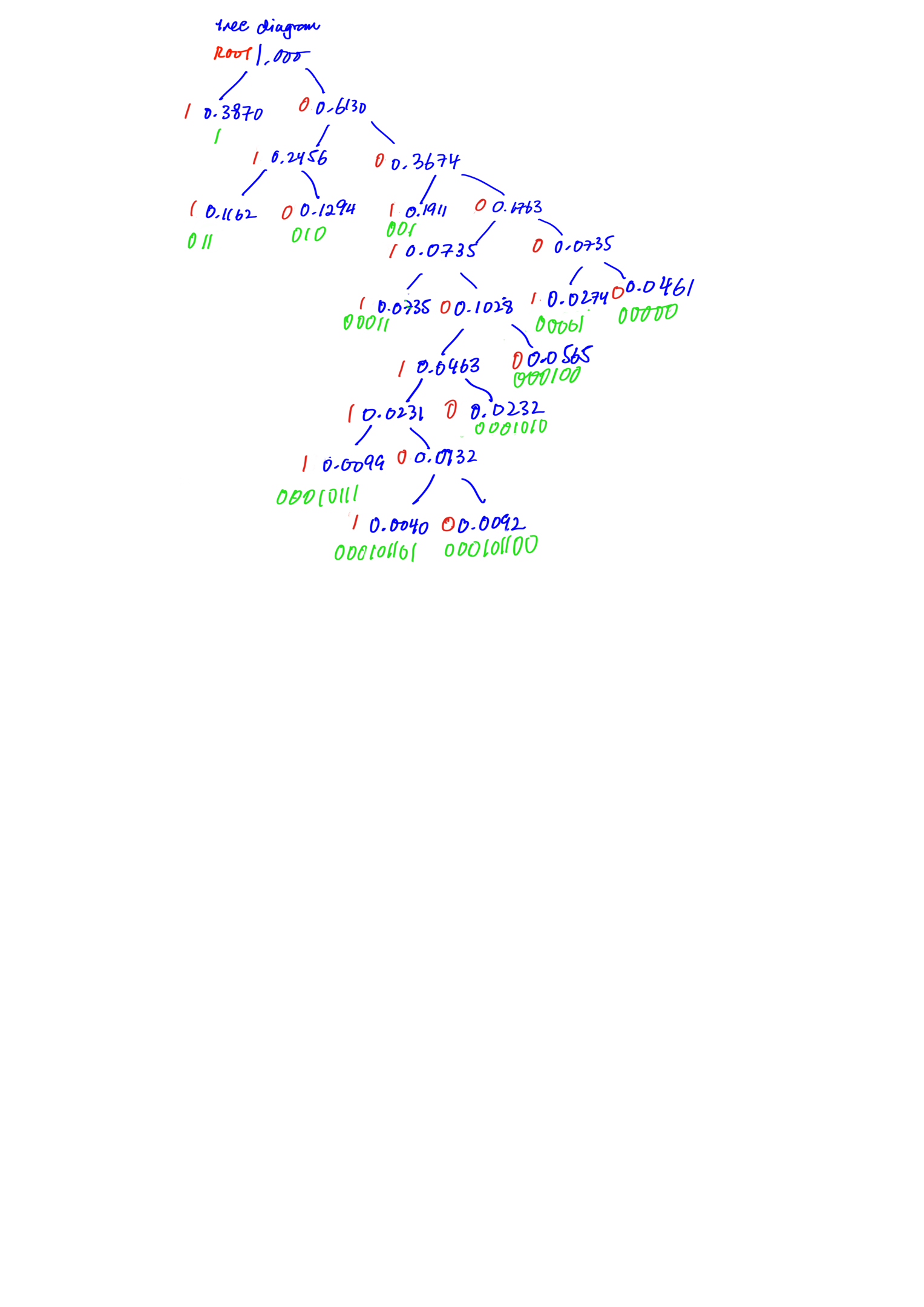
# (Commit the revised codes to GitHub. Show your results to TAs.)

1. Attach the corresponding Huffman tree of the revised optimal dictionary.

dict = { [1], [0 1 1], [1 1 0], [0 0 1], [0 0 0 1 1], ...

[0 0 0 0 1], [0 0 0 0 0], [0 0 0 1 0 0], ...

[0 0 0 1 0 1 0],[0 0 0 1 0 1 1 1], [0 0 0 1 0 1 1 0 1], [0 0 0 1 0 1 1 0 0] };



***Fill in the answers, commit the revised codes to GitHub***

***and Show your result to the TA.***

***Part II: Channel Coding***

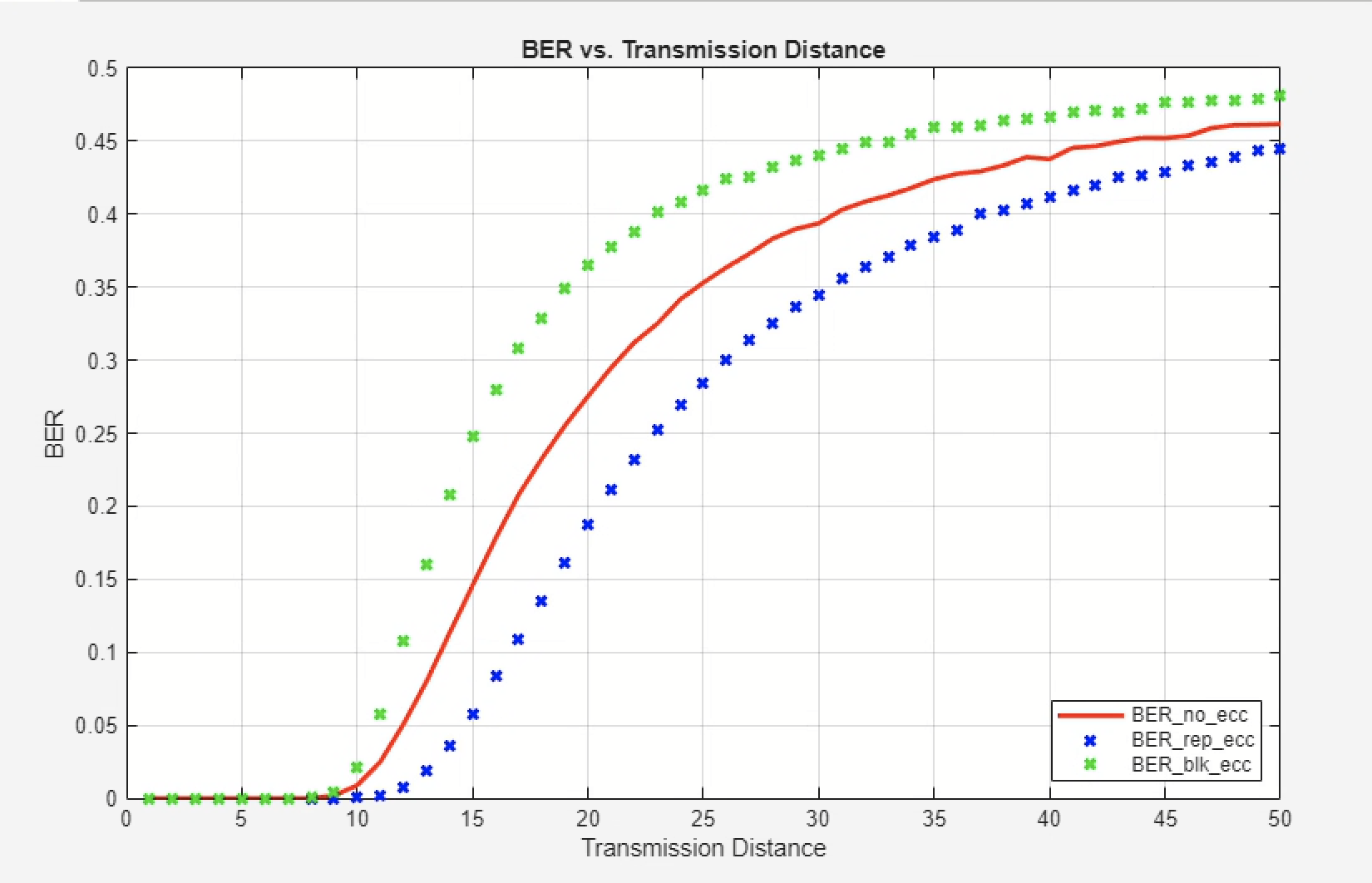


# Task 3 – (n,k) block code decoder and Error Correction Capability

In this task, we will implement the (n,k) block code decoder and compare the error correction capability of the repetition code, hamming block code, and no error correction code.

# Check point:

1. Generate a figure with three curves representing the BER performance.



# （Show your results to the TA）

1. Write down/Insert a screenshot of the modified code in “**blk\_decoder.m**”.

function msgblk = blk\_decoder(codeword)

% Compute syndrome bits

% Rearranging the codeword bits 1 to 8 as follows:

% 1 2 5

% 3 4 6

% 7 8

ind = [1 3 1 2;...

2 4 3 4;...

5 6 7 8 ];

% Check parity by summing down the rows and taking mod 2

S = mod(sum(codeword(ind)), 2);

% Assume no error at first

msgblk = codeword(1:4);

% Check for errors based on the syndrome bits

if S(1) == 1 && S(2) == 1

% Single bit error detected in the message block

msgblk(1) = not(msgblk(1)); % Error in msgblk(1)

elseif S(1) == 1 && S(3) == 1

msgblk(2) = not(msgblk(2)); % Error in msgblk(2)

elseif S(2) == 1 && S(4) == 1

msgblk(3) = not(msgblk(3)); % Error in msgblk(3)

elseif S(3) == 1 && S(4) == 1

msgblk(4) = not(msgblk(4)); % Error in msgblk(4)

elseif sum(S) == 1

% Error in one of the parity bits

error\_col = find(S == 1);

% You can handle parity bit correction here if needed

disp(['Error in parity bit ' num2str(error\_col) '.']);

end

end

**(Commit the revised codes to GitHub. )**

1. Based on your observations, which coding scheme performs the best? **Explain why**.

**(8,4) block code** generally performs better than the (3,1,3) repetition code, in scenarios with higher noise levels or longer transmission distances. It balances redundancy and error correction capability, allowing it to handle multiple errors more efficiently while maintaining a lower BER.

***Fill in the answers, commit the revised codes to GitHub***

***and Show your result to the TA.***

**----------------------------------End-----------------------------------**