# THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY ISDN 2602

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

**Answer Sheet**

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

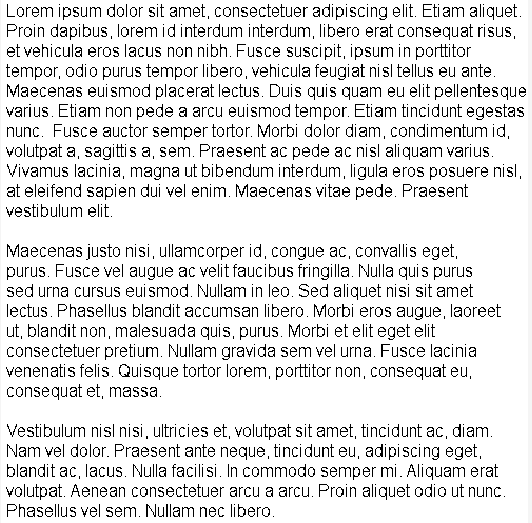
***Part I: Source Coding***

# 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.

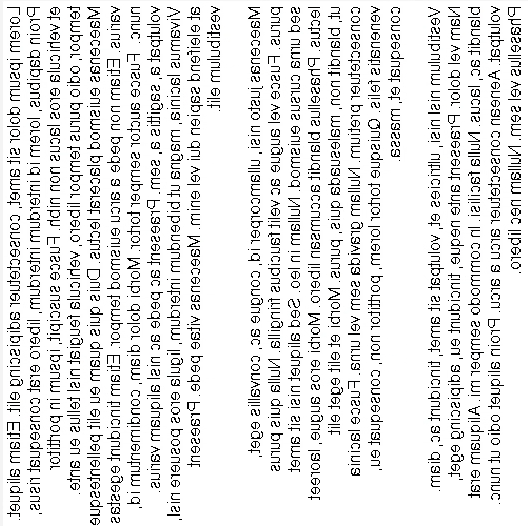
# Check Point:

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**.



The raw length is 25000, and the run code is 301688. Because the run code is represented with numbers, it can be longer as seen in the code. This is because each number from the run code is represented with 8 bits, so big length will take up less bit and smaller length will take up more bit. In this case, the raw code is better, but this is not always the case.

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 length: 250000

Size run length: 196680  
Size Huffman: raw:134892 run:120565

When the image is transposed, the run length gets longer, because there are more horizontal black in the transposed picture. And when the run length in general gets longer, the size of the run length gets smaller.

With the Huffman code, you can see that the size of the raw run length is bigger than the raw run length, because the raw has more of the same run length that the raw code has.

***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 % % % %*

dict = {[0 0 0 0 1 0 1 1], [1], [0 0 1], [0 1 0], [0 1 1], [0 0 0 1 0], [0 0 0 0 1 1], [0 0 0 0 0 ], [0 0 0 0 1 0 0 ], [0 0 0 0 1 0 1 0],[0 0 0 1 1]};

*% % % % 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.

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

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

dict = {[0 0 0 0 1 0 1 1], [1], [0 0 1], [0 1 0], [0 1 1], [0 0 0 1 0], [0 0 0 0 1 1], [0 0 0 0 0 ], [0 0 0 0 1 0 0 ], [0 0 0 0 1 0 1 0],[0 0 0 1 1]};

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

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

***Part II: Channel Coding***

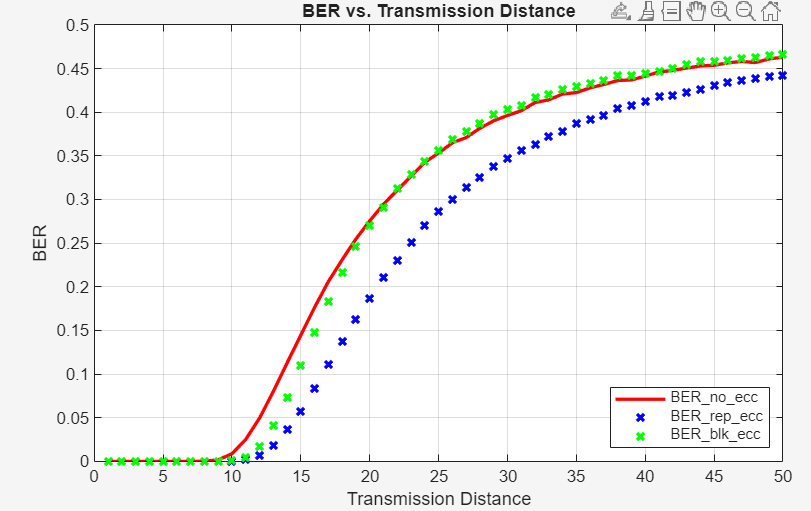


# Task 3 – (n,k) block code decoder and Error Correction Capability

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

% If we rearrange the codeword bits 1 to 8 as

% 1 2 5

% 3 4 6

%` 7 8

% The parity checks correspond to checking the parity of the bits indexed

% by the columns of the matrix below

ind = [1 3 1 2;...

2 4 3 4;...

5 6 7 8 ];

% We can check parity by summing down the rows and then taking the modulus

% after division by two.

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

% assume no error at first

msgblk = codeword(1:4);

% compute syndrome bits

% S = zeros(1,4);

% S(1) = rem(sum(codeword([1 2 5])),2);

% S(2) = rem(sum(codeword([3 4 6])),2);

% S(3) = rem(sum(codeword([1 3 7])),2);

% S(4) = rem(sum(codeword([2 4 8])),2);

%

% % check for one bit errors in the message block only

% % There are four possible one bit errors in the message block

%

% % Modify the code below

if (S(1)==1) && (S(2)==0) && (S(3)==1) && (S(4)==0)

msgblk(1)=not(msgblk(1));%when one bit error is in msgblk(1)

elseif (S(1)==1) && (S(2)==0) && (S(3)==0) && (S(4)==1)

msgblk(2)=not(msgblk(2));%when one bit error is in msgblk(2)

elseif (S(1)==0) && (S(2)==1) && (S(3)==1) && (S(4)==0)

msgblk(3)=not(msgblk(3));%when one bit error is in msgblk(3)

elseif (S(1)==0) && (S(2)==1) && (S(3)==0) && (S(4)==1)

msgblk(4)=not(msgblk(4));%when one bit error is in msgblk(4)

end

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

The BER of blk preformes the best because

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

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

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