Question 1

myHuffman function

```
function [CodeWord, dict] = myHuffman(string)
    chars = unique(string);
    p = sum(bsxfun(@eq, string.', chars)) / strlength(string);
    [p, indices] = sort(p);
    chars = chars(indices);

    n = length(p);
    code = cell(1,n);   %Where the codewords are going to be stored
    X = zeros(n,n);   %This matrix helps us track which elemnts we
        have worked on
    temp = p;   %We will work on temp not to temper with original p

%Building the relationship matrix X.This matrix has all elements zero
%except for few entries which are substituted with 10 or 11.Number 10
    denotes this
%entry is the minimum in the column and 11 indicates this is the
    second
%minimum.the minimum is replaced by 20 and second minimum is replaced
    by
%sum of the minimum and the second minimum.And processing for the
    next
%column progresses
    for i = 1:n-1
        [~ ,index] = sort(temp);
        X(index(1),i) = 10;
        X(index(2),i) = 11;
        temp(index(2)) = temp(index(2)) + temp(index(1));
        temp(index(1)) = 20;
    end

%Filling in codewords.The key is the relationship between the 11
    marked
%entry in each columnThis ties the column with the next one.
    i = n-1;
    rows = find(X(:,i) == 10);
    code(rows) = strcat(code(rows),'1');
    rows = find(X(:,i) == 11);
    code(rows) = strcat(code(rows),'0');
    for i = n-2:-1:1
        row11 = X(:,i) == 11;
        row10 = X(:,i) == 10;
        code(row10) = strcat(code(row11),'1');
        code(row11) = strcat(code(row11),'0');
    end
```

Question 1 continued on next page. . .

```
% creating the dict of Huffman coding in order of alphabet
    [~, charIndex] = sort(chars);
    dict = code(charIndex); % final huffman dictionary

    CodeWord = ''; % final Huffman code
    for i = 1:strlength(string)
        CodeWord = [CodeWord, cell2mat(code(string(i) == chars))];
    end
end
```

This $Huffman$ function gets a string in input and outputs the $Huffman$-encoded codeword of it and the corresponding dictionary of encoding process that is the array which contains the code of each symbol used in the string in order of priority in English Alphabet.

This function at first calculates the probability of occurrence of each symbol using its repetition in the text. Then according to this probability vector, builds a matrix X that is the relationship matrix.This matrix has all elements zero except for few entries which are substituted with 10 or 11.Number 10 denotes this entry is the minimum in the column and 11 indicates this is the second minimum.the minimum is replaced by 20 and second minimum is replaced by sum of the minimum and the second minimum.And processing for the next column progresses.

Actually it makes the tree in $Huffman$ algorithm and also its history to being made. By which we can turn back by the tree to make the corresponding code of each symbol(determining the `code` vector). Then, we reorder this vector to determine the `dict` vector that is ordered by the priority of symbols in alphabet. Then, moving forward in characters of the given string, we code them into `CodeWord` vector.

myLempleziv function

```
function [CodeWord, BinaryCode] = myLempelziv(input_string)
    str='';
    SubStrHolder = string(input_string(1)); % the array that holds
       the seen substrings in the main string
    CodeWord = int2str(0) + string(input_string(1));
    BinaryCode = int2str(0) + string(dec2bin(input_string(1)-65, 5));
    SeenIndex = 0; % holds the index of the seen substring in the
       main string
    for k = 2:strlength(input_string)
        str = [str, input_string(k)];
        tempIndex = ismember(SubStrHolder,str);
        if(sum(tempIndex))
            SeenIndex = find(tempIndex);
        else
            SubStrHolder = [SubStrHolder, string(str)];
            CodeWord = CodeWord + int2str(SeenIndex) + string(
```

```
                input_string(k));
            BinaryCode = BinaryCode + string(dec2bin(SeenIndex)) +
                string(dec2bin(input_string(k)-65, 5));
            SeenIndex = 0;
            str = '';
        end
    end
    if(SeenIndex)
        CodeWord = CodeWord + int2str(SeenIndex);
        BinaryCode = BinaryCode + string(dec2bin(SeenIndex));
    end
end
```

This $Lempelziv$ function gets a string in input and outputs the $Lempelziv$ Encoded string as `CodeWord` and its corresponding binary code as `BinaryCode`.

The function is designed so that the input consists of uppercase English letters(to be used in the Question1). It maps each alphabetical character to a 5bit binary code which is determined according to the order of English letters:

$$\begin{cases} 'A' & \rightarrow & 00000 \\ 'B' & \rightarrow & 00001 \\ \vdots & \rightarrow & \vdots \\ 'Z' & \rightarrow & 11001. \end{cases}$$

On the other hand, the function converts any decimal number in `CodeWord` to its binary form for `BinaryCode`.

The function works such that moves forward the input string in a loop and holds any new substring in a vector named `SubStrHolder`. By the vector, in each iteration of the loop we check the last made substring have been seen or not. If seen we return the corresponding index of the substring to be put and the last letter; and if not, we add that to seen substring and move forward.

The code of making the English text string:

```
alphabets = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N',
    'O','P','Q','R','S','T','U','V','W','X','Y','Z'];
p = [8.4966, 2.0720, 4.5388, 3.3844, 11.1607, 1.8121, 2.4705, 3.0034,
     7.5448, 0.1965, 1.1016, 5.4893, 3.0129, 6.6544, 7.1635, 3.1671,
    0.1962, 7.5809, 5.7351, 6.9509, 3.6308, 1.0074, 1.2899, 0.2902,
    1.7779, 0.2722];
Num_chars = 5000;
p = p(:)./sum(p); % normalising the probabilities
index = randsrc(Num_chars, 1, [1:numel(alphabets); p]);
string = alphabets(index);
```

Question 1 continued on next page...

(a)

In this section mapping each character to its corresponding probability, using `randsrc` we made the string text.
Now, we are going to code the alphabets without any specific method of coding.This means that we are going to code the characters with constant length. According to the contents in the class, it implies that the number of bits needed to code each symbol is equal to

$$n = \lceil H_{max} \rceil = \lceil \log_2 \mu \rceil \quad bits$$

In which μ = Number of unique Symbols = 26. Therefore:

$$n = \lceil \log_2 26 \rceil = 5 \quad bits$$

(b)

```
[Huff_CodeWord, dict] = myHuffman(string);
[Lemp_CodeWord, BinaryCode] = myLempelziv(string);
Huff_length = strlength(Huff_CodeWord);
Lemp_length = strlength(BinaryCode);
ASCII_Bits = 5; % number of bits needed to make each symbol to its
    ASCII binary form

output1 = sprintf('The Compression Ratio for Huffman encoding is %f',
    (Num_chars * ASCII_Bits) / Huff_length);
disp(output1);
output2 = sprintf('The Compression Ratio for Lempelziv encoding is %f
    ', (Num_chars * ASCII_Bits) / Lemp_length);
disp(output2);
```

OUTPUT:

```
The Compression Ratio for Huffman encoding is 1.177912
The Compression Ratio for Lempelziv encoding is 1.175696
```

As can be seen, the Compression Ratio for both coding methods is greater that 1; Which means they perform as we want. Also in most of the samples for generated text string, the Compression Ratio for $Huffman$ coding algorithm is a little bit more than the other which shows the better performance.

(c)

```
Num_UniqueSymbols = strlength(unique(string));  % number of unique
    symbols in the English text
% Huffman
Huff_dict_length = sum(cellfun(@length, dict));

% Lempleziv
```

```
Numbers = regexp(Lemp_CodeWord, '\d*', 'match'); % all the numbers in the lempleziv CodeWord
Numbers = unique(str2double(Numbers)); % unique numbers in double
c = 0; % the minimum length of bits needed to save the binary form of each unique number
for i = 1:length(Numbers)
    c = c + strlength(dec2bin(Numbers(i)));
end
LempDictLength1 = c + Num_UniqueSymbols*5; % adding the memory needed to save dictionary of numbers to dictionary of alphabets in method1
LempDictLength2 = Num_UniqueSymbols*5; % memory of saving the dict of alphabets

% without coding
WithoutComp_dict_length = Num_UniqueSymbols*5;

output1 = sprintf('The space needed to save the dictionary of Huffman Code is %d bits', Huff_dict_length);
disp(output1);
output2 = sprintf('The space needed to save the dictionary of Lempleziv Code in the fisrt method of decoding is %d bits and in the second is %d', LempDictLength1, LempDictLength2);
disp(output2);
output3 = sprintf('The space needed to save the dictionary of Coding without compression is %d bits', WithoutComp_dict_length);
disp(output3);
```

Huffman dictionary

The `myHuffman` function itself gives the dictionary needed to decoding that. So, it's just needed to determine the sum of length of the code of each alphabet.

Lempelziv dictionary

In this coding, we have mapped each alphabet to its 5bit binary code and each number to its binary form. The code dictionary for alphabets must be saved; But not for numbers! So, based on the method of decoding, we need different dictionary:

method1 :

In this method, we save the binary form of each number that comes in the `CodeWord`. Therefore, the memory needed to save that would be too much! Also, the number of bits needed to save the dictionary is specified by `LempDictLength1` in the code.

method2 :

We know in the $Lempelziv$ codeword, after any alphabet there is a number. So, in some ways we can specify first the number 0 and then the 5bits of first alphabet then the second number(that is 0) and second alphabet and so on... . In this way,we can specify the position of numbers and according to the value of detected numbers, predict the bits of each number so that it can be converted from binary to decimal and therefore the numbers easily can be decoded. So, there is no need to save any dictionary for numbers but just alphabets! Also, the number of bits needed to save the dictionary is specified by

`LempDictLength2` in the code.

Coding without compression dictionary
Obviously, in this way we just need to allocate $n = 5 \quad bits$ for each alphabet.(It is similar to method2 of $Lempelziv$ decoding.) Therefore, we need

$$NumberOfAlphabets \times n = 26 \times 5 = 130 \quad bits$$

for the dictionary.

OUTPUT:

```
The space needed to save the dictionary of Huffman Code is 137 bits
The space needed to save the dictionary of Lempleziv Code in the fisrt method of decoding is 4508 bits and in the second is 130
The space needed to save the dictionary of Coding without compression is 130 bits
```

As you can see, the dictionary of the $Huffman$ encoding needs more space than two other codings(if considering method2 for decoding $Lempelziv$ codeword.)

(d)

```
Source_Entropy = sum(-p .* log2(p));
Huff_mean = Huff_length / Num_chars; % the mean length of binary code
    of characters in huffman coding.
Lemp_mean = Lemp_length / Num_chars; % the mean length of binary code
    of characters in Lempelziv coding.

output1 = sprintf('The Source Entropy is %f bits/symbol',
    Source_Entropy);
disp(output1);
output2 = sprintf('The mean length of code for each symbols in
    Huffman coding is %f bits/symbol', Huff_mean);
disp(output2);
output3 = sprintf('The mean length of code for each symbols in
    Lempelziv coding is %f bits/symbol', Lemp_mean);
disp(output3);
```

OUTPUT:

```
The Source Entropy is 4.246828 bits/symbol
The mean length of code for each symbols in Huffman coding is 4.255800 bits/symbol
The mean length of code for each symbols in Lempelziv coding is 4.255600 bits/symbol
```

Clearly, the mean length of codewords(codes for each symbol) for both compression methods is so close to the source entropy. But it is a little bit more. This result was completely expected. Because according to the $Source\ Coding\ Theorem$, the memoryless source output with entropy H, can be recovered with a sufficiently small arbitrary error if

$$\bar{n} \geq H \qquad \bar{n} = Mean\ length\ of\ code\ words$$

. So, here too, the average length of the code words in both methods will be more than the entropy, but due to the high performance of the coding methods of $Huffman$ and $Lempelziv$, this value is very close to the entropy.

Question 1 continued on next page. . .

(e)

```
function code = Arithmetic_Coder(string , p)
    low = 0;
    high = 1;
    range = 1;
    % Encode each symbol in source
    for i = 1:length(string)
        % Find range for current symbol
        symbol = string(i);
        symbolRange = [0, 0];
        for j = 1:symbol-1
            symbolRange(1) = symbolRange(2);
            symbolRange(2) = symbolRange(2) + p(j);
        end
        symbolRange(1) = symbolRange(2);
        symbolRange(2) = symbolRange(2) + p(symbol);

        % Update encoder range
        width = high - low;
        high = low + width*symbolRange(2)/range;
        low = low + width*symbolRange(1)/range;
        range = high - low;
    end
    code = (low + high)/2;
end
```

Arithmetic coding is a method of lossless data compression that encodes a sequence of symbols into a single decimal number in the interval [0,1). It works by dividing the interval [0,1) into sub-intervals corresponding to the probabilities of the symbols in the sequence, and then encoding each symbol by shrinking the sub-interval to fit within its range.
Here, he function initializes the encoder with low set to 0, high set to 1, and range set to 1. It then encodes each symbol in string by finding the range of values for the symbol in the probability distribution and updating the encoder's range accordingly. Finally, it outputs the average of the new low and high values as the code value.
Note that the output of the arithmetic encoding algorithm is a real number between 0 and 1, which is not easily representable in binary format. Therefore, this implementation assumes that the output will be converted to a different format.
Also, unfortunately because this algorithm works for about 10 characters and the number corresponding to each interval in more iterations tend to zero quickly, I were unable to encode the 5000 character string in the question with this algorithm because it is not possible with MATLAB.(in small intervals, we are adding a really small number that is the half of length of interval to the beginning of that which is too relatively large. So, the first number is ignored and we can't code anymore!)

Question 2 continued on next page...

Question 2

(a)

$$\begin{cases} P_{12} = 0.5 \Rightarrow P_{11} = 0.5 \\ P_{21} = 0.8 \Rightarrow P_{22} = 0.2 \end{cases} \Rightarrow \begin{cases} P_1 = 0.5P_1 + 0.8P_2 \\ P_2 = 0.5P_1 + 0.2P_2 \\ P_1 + P_2 = 1 \end{cases} \Rightarrow \begin{cases} P_1 = \frac{8}{13} \\ P_2 = \frac{5}{13} \end{cases}$$

$$\Rightarrow H = \sum_j P_j H_j = P_1 H_j + P_2 H_2 = \frac{8}{13} h(0.5) + \frac{5}{13} h(0.2) \approx 0.89305 \quad bits/symbols$$

The result concludes that in a discrete two-symbol source with memory, The entropy rate is less than $1\ bits/symbols$. Because, the source has memory and every output symbol in average, doesn't give us all the information of the memoryless form; Cause it is dependent of the previous output of the source. Which means, we can kind of predict the next output approximately. Therefore, it has less average information or entropy than the memoryless form(that is a binary memoryless source code which has obviously the entropy $H = 1\ bits/symbols$.

(b)

```
clear; clc; close all
% creating symbols
Num_Symbols = 10^4;
symbols = randsrc(1, Num_Symbols, [[-1, 1]; [5/13, 8/13]]);
G_k = zeros([1, 10]);
Huff_mean = zeros([1, 10]);
Coding_efficiency = zeros([1, 10]);
H_X = 0.8930492673;
for k = 1:10
    %determining probabilities vector for Xk
    n = numel(symbols);
    symbols_k = nan(k, ceil(n/k));
    symbols_k(1:n) = symbols;
    symbols_k = symbols_k.';
    Unq_syms_k = unique(symbols_k, 'rows');
    row_rpt = zeros([1, length(Unq_syms_k)]);
    for i = 1:size(Unq_syms_k, 1)
        row = Unq_syms_k(i, :);
        row_rpt(i) = sum(ismember(symbols_k, row, "rows"));
    end
    p = row_rpt(:)/sum(row_rpt);
    p = p.';

    % calculating G_k
    G_k(k) = sum(-p .* mylog2(p)) / k;

    %calculating Huffman mean codeword length
    Huffman_code = myHuffman_k(p, symbols_k, Unq_syms_k);
```

Question 2 continued on next page...

```
    Huff_mean(k) = strlength(Huffman_code) / Num_Symbols; % the mean length of binary code of characters in huffman coding.

    %calculating Coding gain
    Coding_efficiency(k) = H_X / Huff_mean(k);
end

k = 1:10;
figure
subplot(3,1,1)
plot(k, G_k, LineWidth=2, Color='b')
title('The entropy rate $G_k = \frac{H(X_1,X_2,\cdots,X_k)}{k}$ for different $k$s', 'Interpreter','latex', FontSize=25)
ylabel('$G_k$', 'Interpreter','latex', FontSize=20)
xlabel('$k$', 'Interpreter','latex', FontSize=20)
ylim([0.8, 1])
grid minor

subplot(3,1,2)
plot(k, Huff_mean, LineWidth=2, Color='g')
title('The mean length of $Huffman$ code words for different $k$s', 'Interpreter','latex', FontSize=25)
ylabel('$\bar{R}$', 'Interpreter','latex', FontSize=20)
xlabel('$k$', 'Interpreter','latex', FontSize=20)
ylim([0.8, 1])
grid minor

subplot(3,1,3)
plot(k, Coding_efficiency, LineWidth=2, Color='r')
title('The Coding Efficiency $\eta_N = \frac{H(X)}{\tilde{H}_N}$ for different $k$s', 'Interpreter','latex', FontSize=25)
ylabel('$\eta_N$', 'Interpreter','latex', FontSize=20)
xlabel('$k$', 'Interpreter','latex', FontSize=20)
ylim([0.8, inf])
grid minor


%% functions
function [CodeWord] = myHuffman_k(p, symbols_k, Unq_syms_k)
    n = length(p);
    code = cell(1,n);   %Where the codewords are going to be stored
    X = zeros(n,n);   %This matrix helps us track which elemnts we have worked on
    temp = p;   %We will work on temp not to temper with original p

    for i = 1:n-1
        [~ ,index] = sort(temp);
        X(index(1),i) = 10;
        X(index(2),i) = 11;
```

```
        temp(index(2)) = temp(index(2)) + temp(index(1));
        temp(index(1)) = 20;
    end

%Filling in codewords.The key is the relationship between the 11 marked
%entry in each columnThis ties the column with the next one.
    i = n-1;
    rows = find(X(:,i) == 10);
    code(rows) = strcat(code(rows),'1');
    rows = find(X(:,i) == 11);
    code(rows) = strcat(code(rows),'0');
    for i = n-2:-1:1
        row11 = X(:,i) == 11;
        row10 = X(:,i) == 10;
        code(row10) = strcat(code(row11),'1');
        code(row11) = strcat(code(row11),'0');
    end

    CodeWord = []; % final Huffman code
    for i = 1:size(symbols_k, 1)
        CodeWord = [CodeWord, cell2mat(code(ismember(Unq_syms_k, symbols_k(i, :), "rows")))];
    end
end

% returns zero if the input is zero(to solve the MATLAB problem of 0 * log2(0) =
% -Inf)
function out = mylog2(in)
    out = log2(in);
    out(~in) = 0;
end
```

In this question, we are going to further examine a discrete source with memory.
At first we create the symbols:
Hint : we put the number of produced symbols by the source to 10^4 because for greater values, the probability vector described further, takes so small values that MATLAB rounds them and makes big errors in our calculations.
To do so, we just need to make a random vector named `symbols` which takes the value of 1 and -1 with the probabilities of $\frac{8}{13}$ and $\frac{5}{13}$ in order. It is determined from the previous part of the question and the fact that

$$P(Being\ in\ state\ S_i\ in\ the\ current\ state) = P(Being\ in\ state\ S_i\ in\ the\ next\ state)$$
$$= P(generation\ of\ the\ corresponding\ symbol\ of\ state\ S_i)$$

Now, the vector of symbols produced by the source is created. In the next step, we are going to describe the code for a specific k:

Notice that X_k denotes the output of the source at time k. Therefore, the G_k term described in the question is determined such that:

$$G_k = \frac{H(X_1, X_2, \cdots, X_k)}{k} = \frac{H(\mathbf{X}^k)}{k}$$

Where $\mathbf{X}^k$ denotes a $k-ary$ random vector. So, we just need to calculate the entropy of this random vector.
To do this, in the first step we make a 2D matrix named `symbols_k` which contains the entries of the `symbols` vector. ith Row of this matrix is the ith $k-length$ vector of the main `symbols` vector. In other words, we have split the `symbols` vector with length of n, into $\lfloor \frac{n}{k} \rfloor$ subvectors with length of k that make the rows of the `symbols_k` matrix. So, this rows are outcomes for the random variable $\mathbf{X}^k$ which can be used to determine the Entropy of $H(\mathbf{X}^k)$.
In this step we specify the unique rows of the matrix and according to their number of occurrence, set the probability vector for them.
Finally according to this probability vector that is named `p`, we can easily calculate the Entropy $H(\mathbf{X}^k)$ and consequently G_k from that.

For $Huffman$ encoding the main symbols vector, we should just consider each row of the `symbols_k` matrix as a symbol. Therefore, we would need a M-ary $Huffman$ encoder function. It can be easily determined by some little changes in $myHuffman$ function used in the previous question.(like moving forward into the symbols vector by k-steps, ignore making dict to encode, getting the probability vector directly at the input and ...).
In this way, we get the $Huffman$-encoded code and similar to part d of the Question1, by dividing the length of the code by length of symbols we can find out the value of mean code word length:

$$\bar{R} = mean\ length\ of\ code\ words = \frac{length\ of\ Huffman\ code}{length\ of\ symbols\ vector} = \sum_{x \in \mathbb{X}} p(x) l(x)$$

where $l(x)$ is the length of the code word corresponding to the source output x, satisfies the following inequality:

$$H(\mathbf{X}^k) \leq k\bar{R} < H(\mathbf{X}^k) + 1 \Rightarrow \frac{H(\mathbf{X}^k)}{k} \leq \bar{R} < \frac{H(\mathbf{X}^k)}{k} + \frac{1}{k}$$

$$\Rightarrow G_k \leq \bar{R} < G_k + \frac{1}{k}$$

Means by increasing the value of k, the mean $Huffman$ code word length becomes so close to the G_k.
On the other hand we have by the definition:

$$H(X) = \lim_{k \to \infty} \frac{H(X_1, X_2, \cdots, X_k)}{k} = \lim_{k \to \infty} G_k$$

Therefore, we can conclude with increasing k, the value of $\bar{R}$ gets close to G_k, and also G_k itself approaches $H(X)$, the entropy rate of the source.

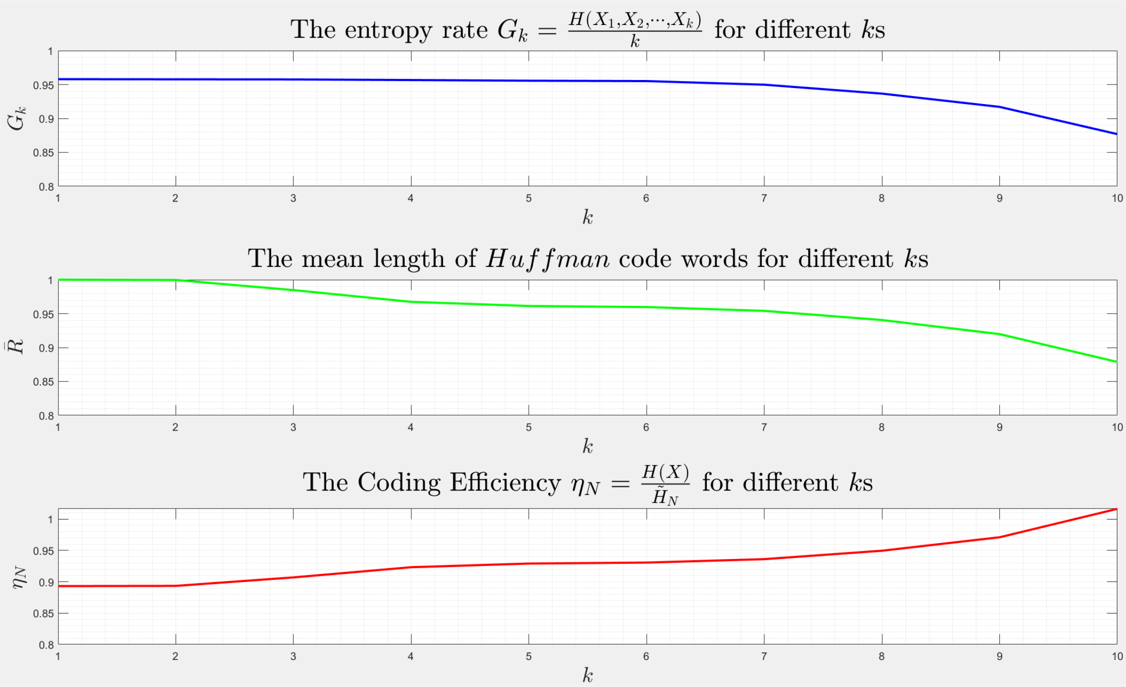
Question 2 continued on next page...

Now we are going to find the $Coding\ Efficiency$. By the definition, it is clear that

$$\tilde{H}_N = \bar{R} \Rightarrow \eta_N = \frac{H(X)}{\bar{R}}$$

So, putting the value of $H(X)$ calculated theoretically in part a and the value of $\bar{R}$ determined before, we can find the $Coding\ Efficiency$.
At last, by repeating the process described above, for different values of k from 1 to 10, we get the following outputs.

OUTPUT:



As expected and described before, G_k for small values of k is close to the entropy of a memoryless 2-symbol source code with $H = h(\frac{5}{13})$ but increasing k makes G_k become so close to the entropy rate of the source in the question.

Also, with increasing k, the mean length of $Huffman$ code words become that were $1bit/symbol$ at first(because we have 2 symbols and no compression needed. It is in the most compressed form!) becomes more closer to G_k and consequently approaches $H(X)$ at last.

Therefore obviously we would have the $Coding\ Efficiency$ η_N in the increasing form approaching to 1. that shows the performance of coding, getting better with increasing k.(but it is a little bit more than 1 for $k = 10$ because of the errors in MATLAB calculations against theory calculations.)

BONUS QUESTIONS

Question 3

Sum_Product_BPSK function

```
function reconstructed_codeword = Sum_Product_BPSK(BPSK_signal, repeatance, E, N0, n_check_nodes, l)
    L = 4 * sqrt(E)/N0 * BPSK_signal; % creating the initial likelihood
    M_matrix = repmat(L', 1, n_check_nodes);  %Lji matrix
    N_matrix = M_matrix.';  % Lij matrix

    for i = 1:repeatance
        for k = 1:l
            for j = 1:n_check_nodes
                current_message = M_matrix(:,j);
                current_message(k) = [];  % To remove the message in the current time from the product accordign to formula
                N_matrix(j, k) = 2 * atanh(prod(tanh(current_message / 2))); % Updating from check nodes to variable nodes with the formula for Lji
            end
        end
        for k = 1:l
            for j = 1:n_check_nodes
                current_message = N_matrix(:,k);
                current_message(j) = []; % To remove the message in the current time from the product accordign to formula
                M_matrix(k, j) =  L(k) + sum(current_message); % Updating from variable nodes to check nodes with the formula for Lij
            end
        end
    end
    L = L + sum(N_matrix, 1); % updating likelihood
    reconstructed_codeword = L < 0;     %detected codeword
end
```

In BPSK modulation, the transmitted bits are represented as phase shifts of a carrier signal, (here 0 and +1). The received signal is then corrupted by noise, and the goal of decoding is to maximize the likelihood of the transmitted bits given the received signal.

The sum-product algorithm works by constructing a graph, which is a graphical model representation of the problem. The graph consists of variable nodes which are

Question 3 continued on next page...

$M(j)$ here, corresponding to the received symbols, and check nodes which are $N(j)$ here, corresponding to the channel and the code. Each variable node is connected to its corresponding check nodes, and each node is connected to its corresponding variable nodes.

The algorithm iteratively refines estimates of the variable nodes' probabilities of being a 0 or +1. At each iteration, the algorithm performs two message-passing steps: a sum step and a product step. In the sum step, each check node calculates the weighted sum of the probabilities of its connected variable nodes, using the log-likelihood ratio (LLR) of the received symbol. In the product step, each variable node calculates the product of the messages received from its connected check nodes and normalizes the result.

Also in this algorithm, the `M_matrix` and `N_matrix` are the matrices that are initialized with the likelihood vector L and are used to get the information from the check nodes to the variable ones and visa versa.(Used to update the prior and get the posterior)

Example:

```
clear;clc;

CodeWord = randi([0,1], 1, 1000);
channel_noise = sqrt(2) * randn(size(CodeWord));
E = 3.5^2;
N0 = 1.5;
BPSK_signal = (-sqrt(E)) * CodeWord + sqrt(E) * (~CodeWord) +
    channel_noise; % creating BPSK-modulated signal
l = length(BPSK_signal);
repeatance = 20;
reconstructed_codeword = Sum_Product_BPSK(BPSK_signal, repeatance, E,
     N0, 4, l);
estimation_accuracy = sum(reconstructed_codeword == CodeWord) / l;

output = sprintf('The estimation accuracy percentage in decoding is %
    f', estimation_accuracy * 100);
disp(output);
```

There was an example in the above, in which the $Sum\_Product\_BPSK$ function got tested on a code word. The BPSK-modulation of the codeword is done by $E = 3.5^2$ and a Gaussian noise with the variance of $\frac{N_0}{2}$ is superimposed on the modulated signal. Also, the number of check nodes is 4 and the number of iterations is 25. Giving the function the polluted signal with the given data, after codeword detection it outputs

OUTPUT:

```
The estimation accuracy percentage in decoding is 99.300000
```

That is good for the given approximately low SNR in the input(The noise power is high!).

Question 3 continued on next page...

For example if we increase E to $E = 6^2$, we will have in the output
OUTPUT:

```
The estimation accuracy percentage in decoding is 100.000000
```

which shows that the detection was completely correct and accurate.