

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATICS

Digital Telecommunications

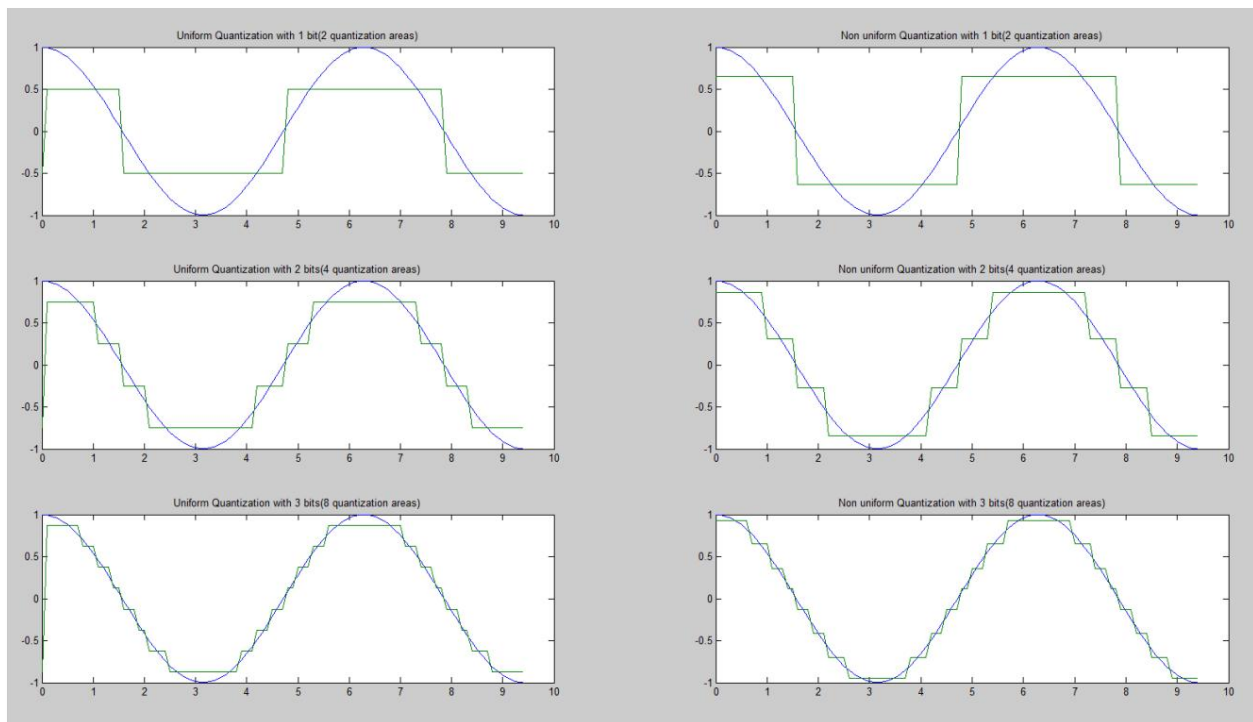
EXERCISE 1st



ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

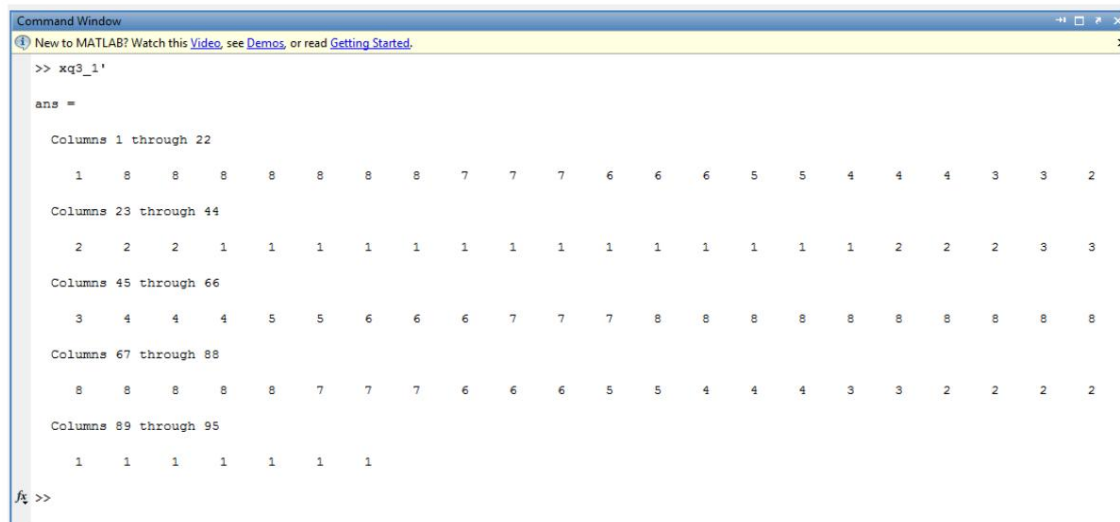
1.1 PCM encoding

Initially, we create the codes that implement the requirements quantizers and are located in the files **uniform_quantizer.m** and **non_uniform_quantizer.m**. Next we present the operation of the two quantizers for the function $f(x) = \cos(x)$ in the interval $[-0, 3\pi]$ for 2, 4, and 8 quantization levels or otherwise for $N = 1, 2, 3$ bits.



The above waveforms are created with the help of the script file **correct_samples.m**. We observe that the cosine signal we create compared to the corresponding quantized one, it has a better approximation than the original one as we increase the number of bits or otherwise the quantization levels. The quantizers created in the above files represent values in intervals $1, 2, 3, \dots, N=1, 2, 3$ bits used in quantization. That is, if we look at the vector x_q which contains the indices for the vector *centers*, this vector that we use to construct the quantized signal, we will confirm the above position. For the above example,

we look at the cosine signal x_q for 3 bits and quantization range $[-1, 1]$ that it is



```

Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
>> xq3_1'
ans =
Columns 1 through 22
     1     8     8     8     8     8     8     8     7     7     7     6     6     6     5     5     4     4     4     3     3     2
Columns 23 through 44
     2     2     2     1     1     1     1     1     1     1     1     1     1     1     1     1     2     2     2     3     3
Columns 45 through 66
     3     4     4     4     5     5     6     6     6     7     7     7     8     8     8     8     8     8     8     8     8
Columns 67 through 88
     8     8     8     8     8     7     7     7     6     6     6     5     5     4     4     4     3     3     2     2     2
Columns 89 through 95
     1     1     1     1     1     1     1
fx >>

```

1.1.1 Uniform quantizer

The uniform quantizer is implemented by the function:

function [x_q , centers, up_bounds, down_bounds] = uniform_quantizer(x, N, min_value, max_value)

The logic of implementing the code is as follows: initially we define "D" quantization step", after we first make some necessary checks. Based on of this step, we uniformly create the quantization field and the we fragment it into quantization regions. The average of the regions, which translated to the centers of these areas, the corresponding levels will be quantization that returns the vector *centers*. From the samples of the signal we assign them appropriately so that they belong to the quantization regions, we give them index values and return them together with the vector x_q . The samples that are outside of the quantization levels we adjust it according to the ends of the fields. We can reproduce the quantized signal with the command **centers(x_q)**.

1.1.2 Non-uniform quantizer

The uniform quantizer is implemented by the function:

```
function [ xq, centers, D, bounds ] = non_uniform_quantizer( x, N, min_value, max_value )
```

The logic of the code implementation is similar to that of uniform quantizer and is as follows: initially we give the same arguments as the uniform quantizer. In this implementation we correspond each sample to in each region and then calculate the centroids of each region, finding the average of these samples. The corresponding centers of of the uniform quantizer are the centroids in the non-uniform quantizer that will returns the vector *centers*. Then each of the samples that will be belongs to a region, it will be assigned to the centroid value, it will we store in the vector *xq*. In this way we will create the quantized signal and in addition we will calculate its distortion quantized signal relative to the original by storing it in the vector *D*.

Furthermore, we apply the steps of the Lloyd-Max algorithm which allows the design of an optimal quantizer for any number of levels. Since we have the centroids of the quantization regions, we use them appropriately to create the new quantization regions as follows: We calculate the average of two successive centroids and we consider these as the new edges of the regions quantization, having included in the data the values **min_value** and **max_value**, since the signal is always limited to the range [min_value, max_value]. As in the first step, we match the signal samples to the regions quantization that belong, we recalculate the centroids and store them in the vector *centers*. Each sample belonging to a region takes the value of the corresponding centroid, thus creating the new quantized signal *xq*. In new quantized signal we calculate the distortion relative to the original and we store it in the vector *D*. Finally we compare the last two elements, if they exceed the **eps** of our machine and it is indeed true, we repeat the above step, we assume that our algorithm converges and we keep the last quantized signal in order.

Creating sources

The sources described in the exercise are created in the script file **createSourceScript.m**. It should be noted the following change made to transformation of the third source. Because the image is not in the desired state but in the range $[-0.5, 1]$ instead of the desired $[-1, 1]$. This is corrected as follows:

```
% -----THIRD SOURCE-----
% loading image
load mercs2.mat
% imshow(uint8(mercs))
% pause(3)
% close

% transform the image in order to bring it's dynamic range to
% [-1,1];
xr = mercs;
x = mercs(:); % third x[] source
x = (x-min(x));
f=(min(x)+max(x))/2;
x_C = (x-f) / f;
```

Question 1

The following answers regarding quantization, SQNR probability of occurrence of quantization levels and overload probabilities, have been based both on the course descriptions as well as Wikipedia searches.

- a) For **source A** after running the codes **main.m**, **createSourceScript.m**, **1st_source.m** we get the following results:

SQNR_exper_uni_sA					
SQNR_exper_uni_sA <1x3 double>					
	1	2	3	4	5
1	11.2022	15.6589	16.3961		
2					
3					
4					
SQNR_theor_uni_sA					
SQNR_theor_uni_sA <1x3 double>					
	1	2	3	4	5
1	11.4291	16.3251	17.1996		
2					
3					
4					
SQNR_exper_non_sA					
SQNR_exper_non_sA <1x3 double>					
	1	2	3	4	5
1	11.8921	16.1169	16.5024		
2					
3					
4					
SQNR_theor_non_sA					
SQNR_theor_non_sA <1x3 double>					
	1	2	3	4	5
1	12.1476	16.8630	17.3314		
2					
3					
4					

For the SQNRs of this source we observe that the experimental ones we get is quite close to the theoretical ones, which makes sense since our source is random. Although the distribution followed by source A is exponential, for 10000 elements this distribution does not seem to be followed. Probably for 1,000,000 elements the results would be better and the approach still. However, what is seen is that as N increases, the SQNRs these are increasing. This is due to the better approach that we achieve each time the quantized signal with respect to the original. Thus by better approximating the quantized shape to the original, we reduce the distortion, since SQNR is inversely proportional to deformation.

For **source B** after running the codes **main.m**, **createSourceScript.m**, **2nd_source.m** we get the following results:

SQNR_theor_non_sB				
SQNR_theor_non_sB <1x3 double>				
	1	2	3	4
1	18.0723	29.0012	39.3128	
!!!				
SQNR_exper_non_sB				
SQNR_exper_non_sB <1x3 double>				
	1	2	3	4
1	18.2945	28.1298	38.9345	
!!!				
SQNR_theor_uni_sB				
SQNR_theor_uni_sB <1x3 double>				
	1	2	3	4
1	18.0921	29.0288	39.4712	
!!!				
SQNR_exper_uni_sB				
SQNR_exper_uni_sB <1x3 double>				
	1	2	3	4
1	18.0343	28.3874	39.1642	
!!!				

And for source B the same things apply as for source A with the difference that here a uniform distribution is followed in the interval $[0, 4]$. And here again it is not appears to follow a uniform distribution. Therefore, the experimental quantized signal has less distortion than the theoretical one, which would be true respectively in their SQNRs.

- b) Based on the SQNRs it is observed that the non-uniform quantizer succeeds larger SQNRs compared to the homogeneous one, which means that the quantized signal undergoes less distortion. This practically means that the non-uniform quantizer is *optimal* with respect to the uniform.
- c) For *source A* the following overload probabilities are observed:

p_overload_experimental			
p_overload_experimental <1x1 double>			
	1	2	3
1	0.0182		
2			
3			

p_overload_theoretical			
p_overload_theoretical <1x1 double>			
	1	2	3
1	0.0183		
2			
3			

For *source B* the overload probability will be equal to 0 since we quantize in the same dynamic range as the dynamic range of the signal.

- d) To theoretically calculate the probability of each level of occurrence quantizer, we create the following in the file **layer_probability.m** function

```
function probability = layer_probability(x,N,min_value,max_value,type)
```

This will be called from the script files of each source source A, B respectively in order to automatically calculate the requested values. This function, since calls the appropriate quantizer, uses the quantization ranges that are returned with the ***bounds register***. These are used as edges in the integral of each distribution on which the samples of each are based source. In addition, the probability of overloading the quantizer is calculated and stored in a vector. This process is performed either for uniform or for non-uniform quantization depending on the *type argument*.

Similarly for the experimental data we again use the **bounds register**.

In this case we find the number of samples belonging to the each quantization region and divide it by the total number of samples in order to find the probability of each quantization level. This process is done with the help of the following code (*below is code for source A for the uniform quantizer but the same applies to the other sources*).

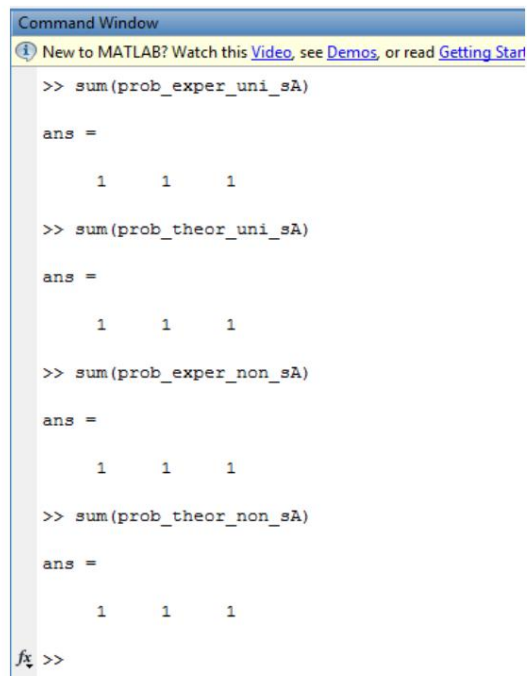
```
prob_exper_uni_sA(1, i) = numel(x_A(x_A>-Inf & x_A<bounds(1,1)))/
numel(x_A);

prob_exper_uni_sA(2^(2*i)+2,i) = numel(x_A(x_A>bounds(end,2) & x_A<Inf))/numel(x_A);

% we compute the rest of the probabilities for j=1:size(bounds,1)

    prob_exper_uni_sA(j+1,i)=numel(x_A(x_A>=bounds(j,1) & x_A<bounds(j,2)))/
    numel(x_A);
end
```

Additionally, we sum the probabilities of each column and check if they have a sum of 1.



```
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started

>> sum(prob_exper_uni_sA)

ans =

    1    1    1

>> sum(prob_theor_uni_sA)

ans =

    1    1    1

>> sum(prob_exper_non_sA)

ans =

    1    1    1

>> sum(prob_theor_non_sA)

ans =

    1    1    1

fx >>
```

In the above code the index i is the line of each of the vectors and shows at $N=2*i$.

Additionally, to calculate the efficiency of PCM encoding we will exploit the entropy from the experimental probabilities of occurrence. This is implemented with the help of the following code for uniform and non-uniform quantizer, after first filtering and keeping only the non-zero ones chances.

```
% FOR UNIFORM QUANTIZER
```

```
prob_exp_uni_sA_1 = prob_exp_uni_sA(:,1)'; prob_exp_uni_sA_2 =  
prob_exp_uni_sA(:,2)'; prob_exp_uni_sA_3 = prob_exp_uni_sA(:,3)';  
prob_exp_uni_sA_1 = prob_exp_uni_sA_1(2:2^(2*1)+2); prob_exp_uni_sA_2  
= prob_exp_uni_sA_2(2:2^(2*2)+2); prob_exp_uni_sA_3 = prob_exp_uni_sA_3(2:2^(2*3)+2);
```

```
% FOR NON-UNIFORM QUANTIZER
```

```
prob_exp_non_sA_1 = prob_exp_non_sA(:,1)'; prob_exp_non_sA_2 =  
prob_exp_non_sA(:,2)'; prob_exp_non_sA_3 = prob_exp_non_sA(:,3)';  
prob_exp_non_sA_1 = prob_exp_non_sA_1(2:2^(2*1)+2);  
prob_exp_non_sA_2 = prob_exp_non_sA_2(2:2^(2*2)+2); prob_exp_non_sA_3 =  
prob_exp_non_sA_3(2:2^(2*3)+2);
```

Then the efficiency is found as follows

```
%-----PCM CODING EFFICIENCY PART-----
```

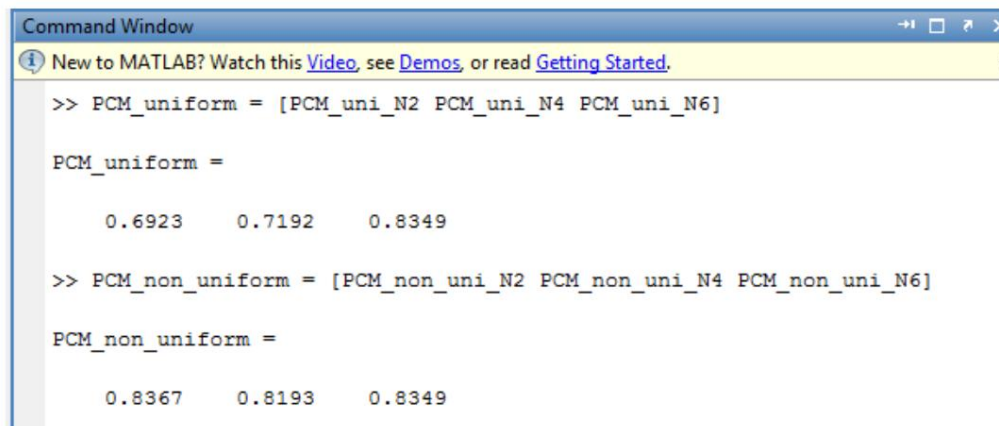
```
% FOR UNIFORM QUANTIZER
```

```
PCM_uni_N2 = -  
sum(prob_exp_uni_sA_1.*log2(prob_exp_uni_sA_1))/(2*1); PCM_uni_N4 = -  
  
sum(prob_exp_uni_sA_2.*log2(prob_exp_uni_sA_2))/(2*2); PCM_uni_N6 = -  
  
sum(prob_exp_uni_sA_3.*log2(prob_exp_uni_sA_3))/(2*3);
```

```
% FOR NON-UNIFORM QUANTIZER
```

```
PCM_non_uni_N2 = -  
sum(prob_exp_non_sA_1.*log2(prob_exp_non_sA_1))/(2*1); PCM_non_uni_N4 = -  
  
sum(prob_exp_non_sA_2.*log2(prob_exp_non_sA_2))/(2*2); PCM_non_uni_N6 = -  
  
sum(prob_exp_non_sA_3.*log2(prob_exp_non_sA_3))/(2*3);
```

Summarizing the results of the efficiency of PCM encoding with use of uniform and non-uniform are the following



```

Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
>> PCM_uniform = [PCM_uni_N2 PCM_uni_N4 PCM_uni_N6]

PCM_uniform =

    0.6923    0.7192    0.8349

>> PCM_non_uniform = [PCM_non_uni_N2 PCM_non_uni_N4 PCM_non_uni_N6]

PCM_non_uniform =

    0.8367    0.8193    0.8349

```

e) In the last question with the help of the following code and given code **huffman.m** we calculate the efficiency of Huffman algorithm and compare the results with the encoding PCM.

% FOR UNIFORM QUANTIZER

% we call after Huffman function to find the bits huffman's coding user for

% PCM uniform quantization

```

[~,bits_1] = huffman(prob_exper_uni_sA_1);
[~,bits_2] = huffman(prob_exper_uni_sA_2);
[~,bits_3] = huffman(prob_exper_uni_sA_3);

```

% then we compute Huffman Coding's efficiency using source A's entropy

```

Huff_uni_N2 = -
sum(prob_exper_uni_sA_1.*log2(prob_exper_uni_sA_1))/sum(prob_exper_uni_sA_1.*bits_1);
Huff_uni_N4 = -
sum(prob_exper_uni_sA_2.*log2(prob_exper_uni_sA_2))/sum(prob_exper_uni_sA_2.*bits_2);
Huff_uni_N6 = -
sum(prob_exper_uni_sA_3.*log2(prob_exper_uni_sA_3))/sum(prob_exper_uni_sA_3.*bits_3);

```

% FOR NON-UNIFORM QUANTIZER

% we call after Huffman function to find the bits huffman's coding user for

% PCM non-uniform quantization

```

[~,bits_1] = huffman(prob_exper_non_sA_1);
[~,bits_2] = huffman(prob_exper_non_sA_2);
[~,bits_3] = huffman(prob_exper_non_sA_3);

```

```
% then we compute Huffman Coding's efficiency using source A's entropy
```

```
Huff_non_uni_N2 = -
sum(prob_exper_non_sA_1.*log2(prob_exper_non_sA_1))/sum(prob_exper_no
n_sA_1.*bits_1);
Huff_non_uni_N4 = -
sum(prob_exper_non_sA_2.*log2(prob_exper_non_sA_2))/sum(prob_exper_no
n_sA_2.*bits_2);
Huff_non_uni_N6 = -
sum(prob_exper_non_sA_3.*log2(prob_exper_non_sA_3))/sum(prob_exper_no
n_sA_3.*bits_3);
```

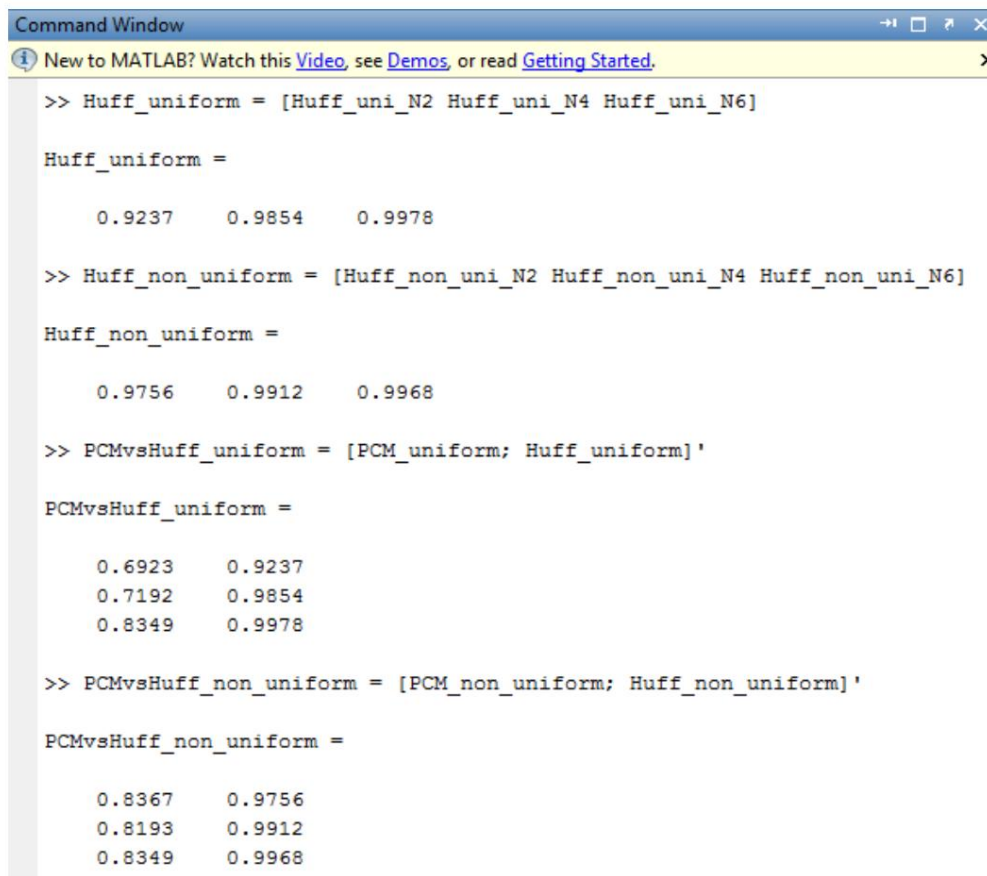
Summarizing the results of the efficiency of Huffman coding in contrast to those of PCM coding using uniform and non-uniform uniform are the following

```
%-----PCM -VS- HUFFMAN EFFICIENCY PART-----
```

```
% present the results into two columns
```

```
PCM_vs_Huff_uni = [PCM_uni_N2 PCM_uni_N4 PCM_uni_N6; Huff_uni_N2
Huff_uni_N4 Huff_uni_N6]';
```

```
PCM_vs_Huff_non_uni = [PCM_non_uni_N2 PCM_non_uni_N4 PCM_non_uni_N6;
Huff_non_uni_N2 Huff_non_uni_N4 Huff_non_uni_N6]';
```



The screenshot shows a MATLAB Command Window with the following code and output:

```
>> Huff_uniform = [Huff_uni_N2 Huff_uni_N4 Huff_uni_N6]

Huff_uniform =

    0.9237    0.9854    0.9978

>> Huff_non_uniform = [Huff_non_uni_N2 Huff_non_uni_N4 Huff_non_uni_N6]

Huff_non_uniform =

    0.9756    0.9912    0.9968

>> PCUvsHuff_uniform = [PCM_uniform; Huff_uniform]

PCUvsHuff_uniform =

    0.6923    0.9237
    0.7192    0.9854
    0.8349    0.9978

>> PCUvsHuff_non_uniform = [PCM_non_uniform; Huff_non_uniform]

PCUvsHuff_non_uniform =

    0.8367    0.9756
    0.8193    0.9912
    0.8349    0.9968
```

Conclusion: We observe that as the number of bits used in encodings increases, the performance also increases.

Question 2

- a) To calculate the SQNRs of the uniform and non-uniform quantizer and to list the images resulting from each quantizer for $N = 2, 4, 6$ bits, we use the code files **main.m**, **createSourceScript.m**, **3rd_source.m** and we get the following results:

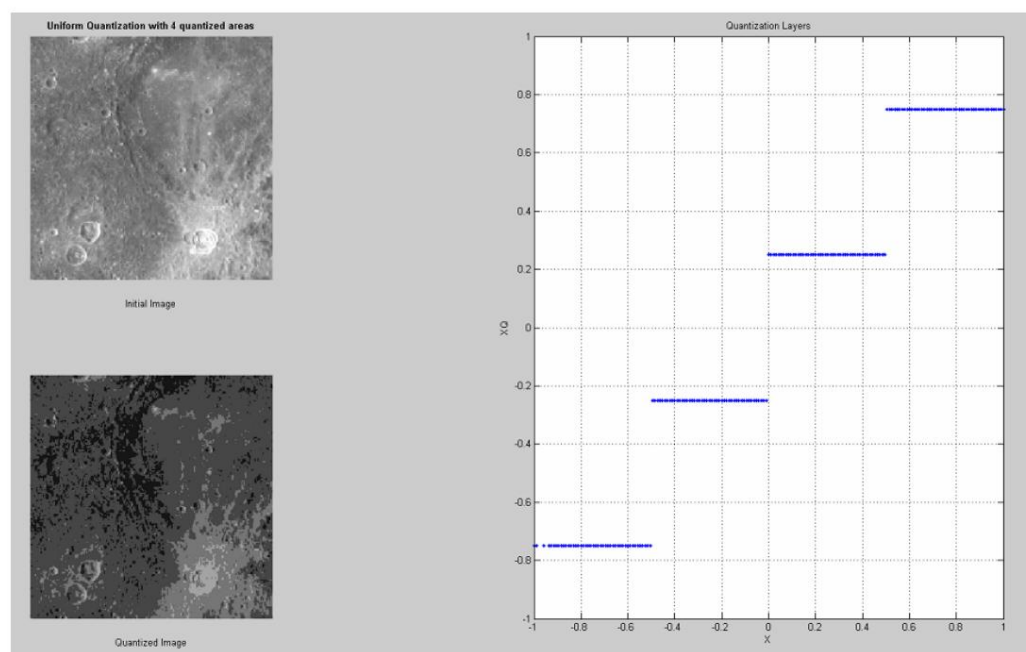
Initially for the SQNRs for the uniform and non-uniform for each of the N bits.

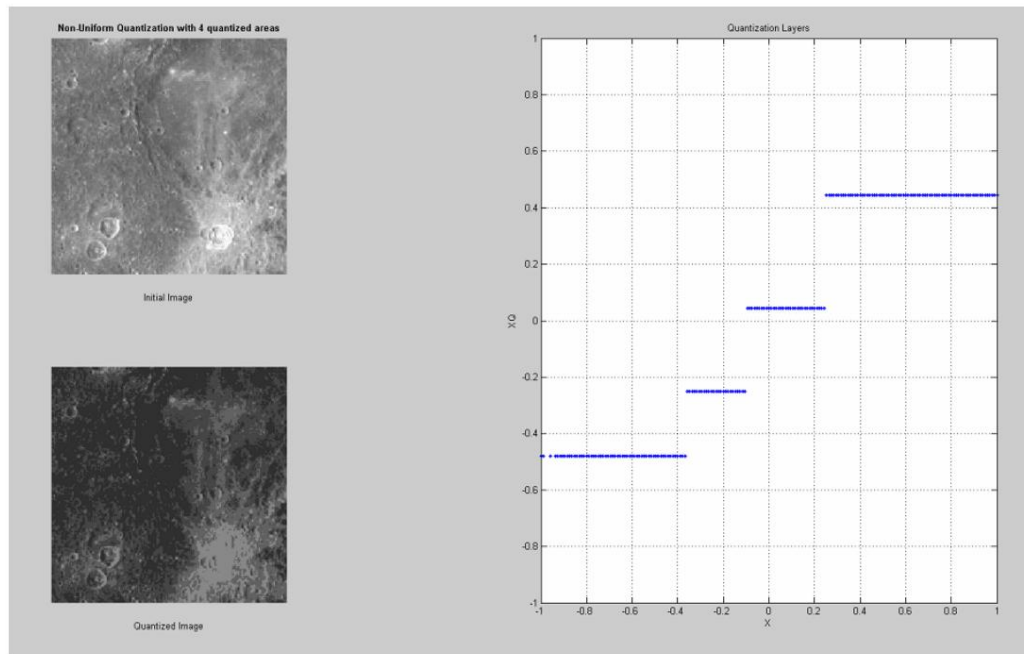
SQNR_exper_non_sC				
SQNR_exper_non_sC <1x3 double>				
	1	2	3	4
1	11.7116	21.8590	32.7211	
2				

SQNR_exper_uni_sC				
SQNR_exper_uni_sC <1x3 double>				
	1	2	3	4
1	7.9183	20.1988	32.3650	
2				

Next follow the images resulting from quantization

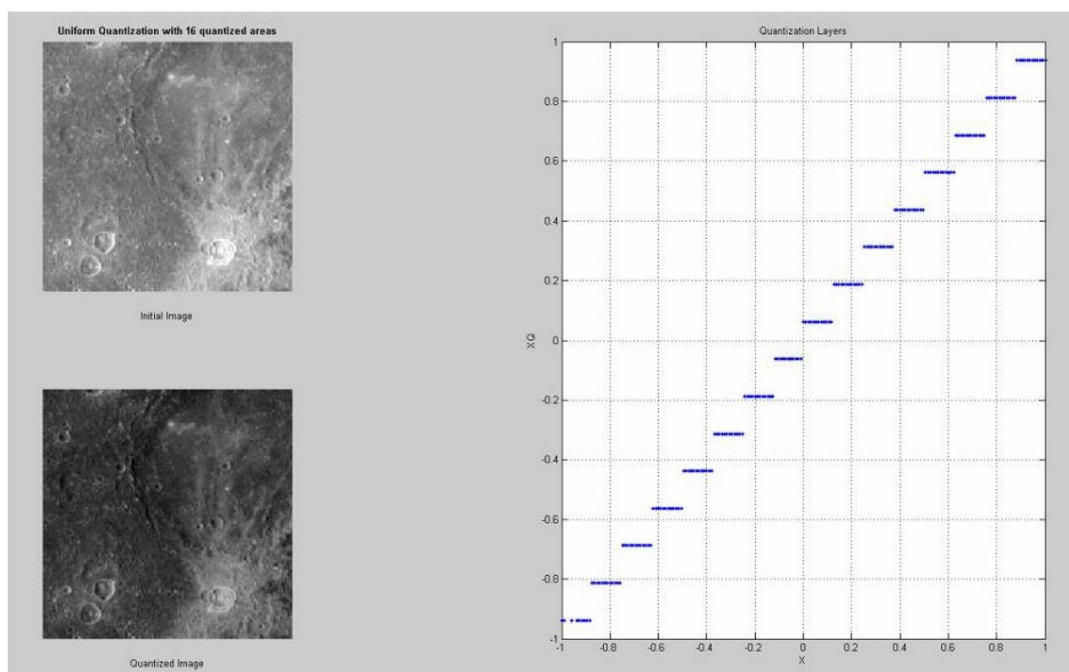
For $N = 2$ bits, i.e. 4 quantization levels

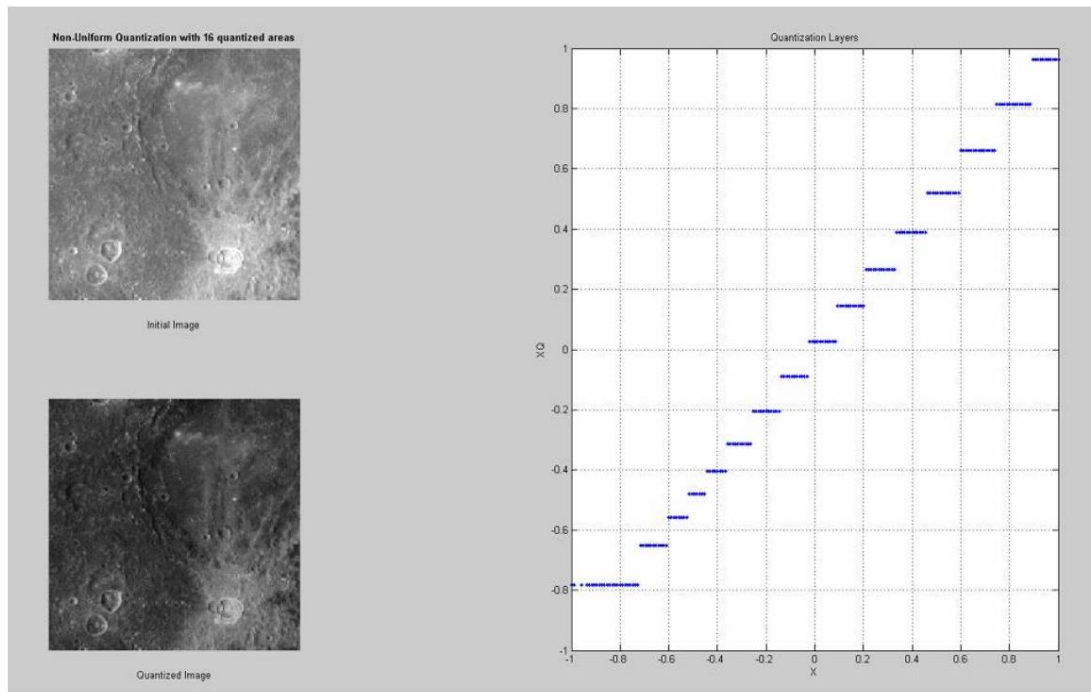




Conclusion: In this subcase, we can observe that the quantized images have shades of gray in as many as the number of the quantization levels. The quantized images additionally contain some from the details of the original image although it is not quite obvious.

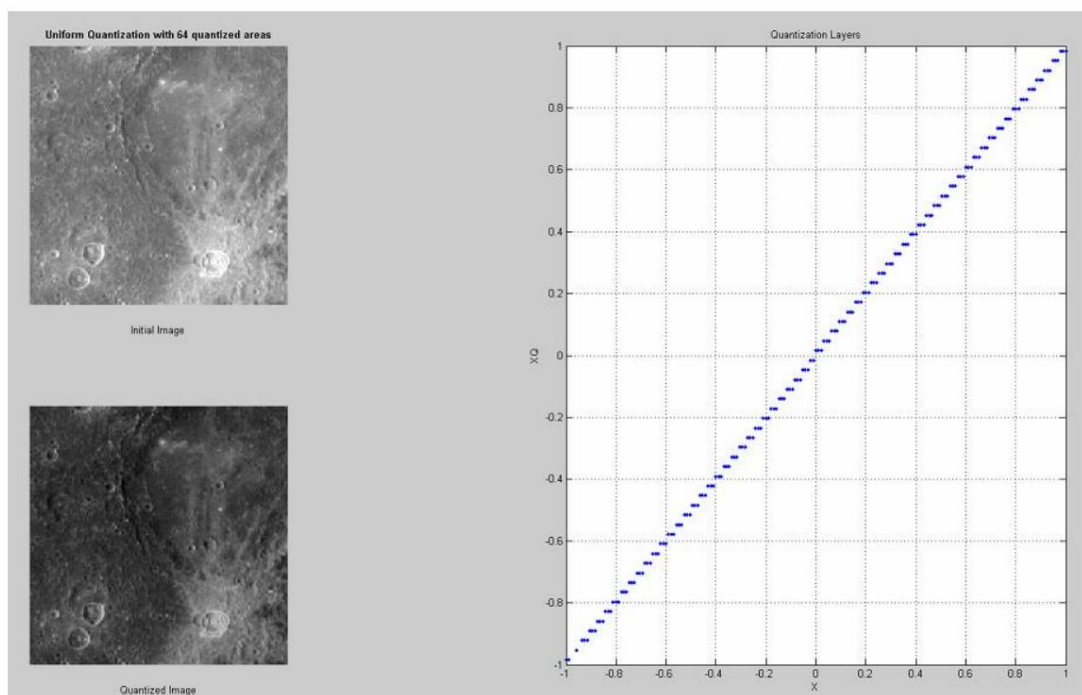
For **$N = 4$ bits**, i.e. 16 quantization levels

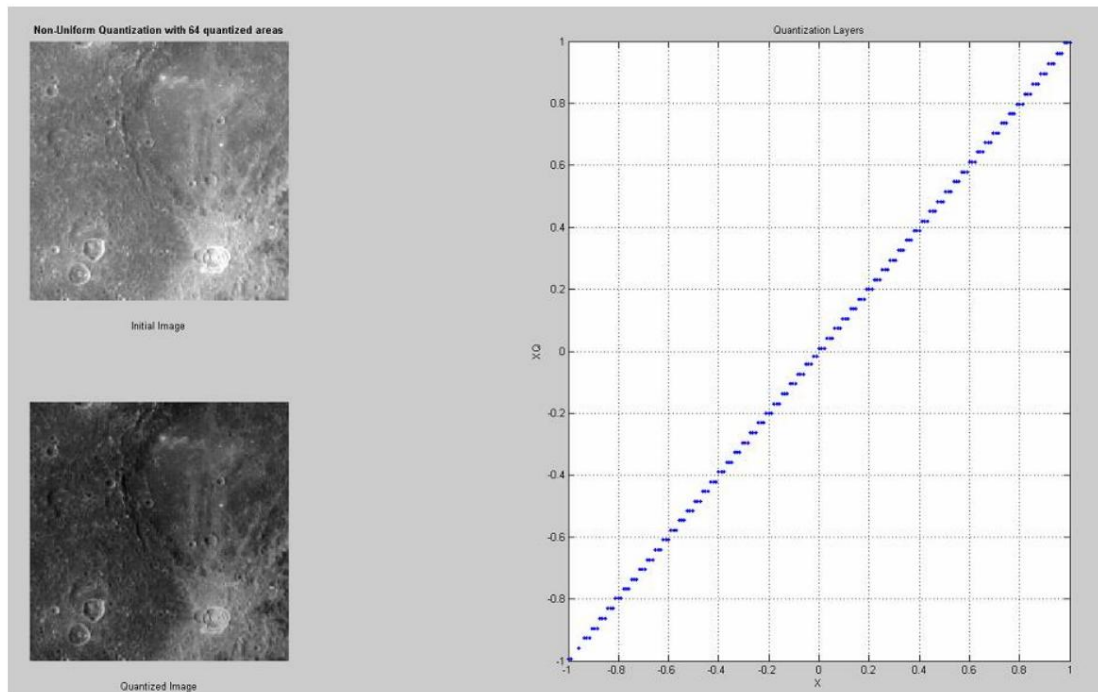




Conclusion: In this subcase, we can observe that the quantized images have a clearly greater “variety” in the shades of gray. Because the quantization we achieve is clearly better than before the differences in the two quantizers cannot be seen strongly.

For **$N = 6$ bits**, i.e. 64 quantization levels





Conclusion: In this subcase, we can observe that the quantized images do not have any particular difference, since the There are many shades. The quantization levels are certainly enough for a satisfactory result of the image approximation.

- b) To theoretically calculate the probability of each symbol appearing in quantizer, we create the following in the file **symbol_probability.m** function

```
function probability
=symbol_probability(x,N,min_value,max_value,type)
```

Its operation is the same as the **layer_probability function**. Then for to calculate the probabilities of occurrence for each of the $N = 2, 4, 6$ bits we use the following code.

```
% FOR UNIFORM QUANTIZER
prob_exper_uni_sC_1 =
symbol_probability(x_C,2,min_value,max_value,'uni');
prob_exper_uni_sC_2 =
symbol_probability(x_C,4,min_value,max_value,'uni');
prob_exper_uni_sC_3 =
symbol_probability(x_C,6,min_value,max_value,'uni');
```



```
% FOR NON-UNIFORM QUANTIZER
```

```
prob_exper_non_sC_1 =
```

```
symbol_probability(x_C,2,min_value,max_value,'non'); prob_exper_non_sC_2 =
```

```
symbol_probability(x_C,4,min_value,max_value,'non'); prob_exper_non_sC_3 =
```

```
symbol_probability(x_C,6,min_value,max_value,'non');
```

Additionally, to calculate the efficiency of PCM encoding we will exploit the following code for uniform and non-uniform quantizer

```
%-----PCM CODING EFFICIENCY PART-----
```

```
% FOR UNIFORM QUANTIZER
```

```
PCM_uni_N2 = -
```

```
sum(prob_exper_uni_sC_1.*log2(prob_exper_uni_sC_1))/(2*1); PCM_uni_N4 = -
```

```
sum(prob_exper_uni_sC_2.*log2(prob_exper_uni_sC_2))/(2*2); PCM_uni_N6 = -
```

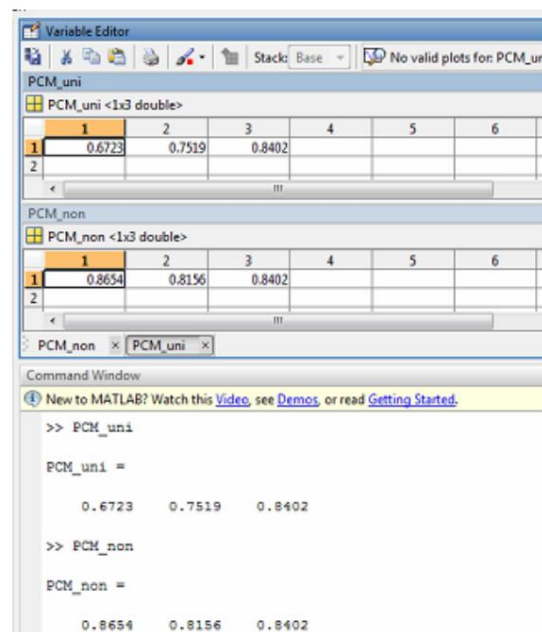
```
sum(prob_exper_uni_sC_3.*log2(prob_exper_uni_sC_3))/(2*3); % FOR NON-UNIFORM QUANTIZER
```

```
PCM_non_uni_N2 = -
```

```
sum(prob_exper_non_sC_1.*log2(prob_exper_non_sC_1))/(2*1); PCM_non_uni_N4 = -
```

```
sum(prob_exper_non_sC_2.*log2(prob_exper_non_sC_2))/(2*2); PCM_non_uni_N6 = -
```

```
sum(prob_exper_non_sC_3.*log2(prob_exper_non_sC_3))/(2*3);
```



- c) In the last question with the help of the following code and of the given **huffman** code.m we calculate in the same way as in Question 1 the efficiency of the Huffman algorithm and compare the results with PCM encoding.

```
% FOR UNIFORM QUANTIZER
```

```
% we call after Huffman function to find the bits huffman's coding user for
```

```
% PCM uniform quantization
```

```
[~,bits_1] = huffman(prob_exper_uni_sC_1);
```

```
[~,bits_2] = huffman(prob_exper_uni_sC_2);
```

```
[~,bits_3] = huffman(prob_exper_uni_sC_3);
```

```
%
```

```
% then we compute Huffman Coding's efficiency using source C's entropy
```

```
Huff_uni_N2 = -
```

```
sum(prob_exper_uni_sC_1.*log2(prob_exper_uni_sC_1))/sum(prob_exper_uni_sC_1.*bits_1);
```

```
Huff_uni_N4 = -
```

```
sum(prob_exper_uni_sC_2.*log2(prob_exper_uni_sC_2))/sum(prob_exper_uni_sC_2.*bits_2);
```

```
Huff_uni_N6 = -
```

```
sum(prob_exper_uni_sC_3.*log2(prob_exper_uni_sC_3))/sum(prob_exper_uni_sC_3.*bits_3);
```

```
% FOR NON-UNIFORM QUANTIZER
```

```
% we call after Huffman function to find the bits huffman's coding user for
```

```
% PCM non-uniform quantization
```

```
[~,bits_1] = huffman(prob_exper_non_sC_1);
```

```
[~,bits_2] = huffman(prob_exper_non_sC_2);
```

```
[~,bits_3] = huffman(prob_exper_non_sC_3);
```

```
%
```

```
% then we compute Huffman Coding's efficiency using source C's entropy
```

```
Huff_non_uni_N2 = -
```

```
sum(prob_exper_non_sC_1.*log2(prob_exper_non_sC_1))/sum(prob_exper_non_sC_1.*bits_1);
```

```
Huff_non_uni_N4 = -
```

```
sum(prob_exper_non_sC_2.*log2(prob_exper_non_sC_2))/sum(prob_exper_non_sC_2.*bits_2);
```

```
Huff_non_uni_N6 = -
```

```
sum(prob_exper_non_sC_3.*log2(prob_exper_non_sC_3))/sum(prob_exper_non_sC_3.*bits_3);
```

%-----PCM -VS- HUFFMAN EFFICIENCY PART-----

%

% present the results into two columns

```
PCM_vs_Huff_uni = [PCM_uni_N2 PCM_uni_N4 PCM_uni_N6; Huff_uni_N2
Huff_uni_N4 Huff_uni_N6]';
```

%

```
PCM_vs_Huff_non = [PCM_non_uni_N2 PCM_non_uni_N4
```

```
PCM_non_uni_N6; Huff_non_uni_N2 Huff_non_uni_N4 Huff_non_uni_N6]';
```

