**Department of Computer Engineering**



**Cairo University**

**Faculty of Engineering**

**ELC 325B – Spring 2023**

**Digital Communications**

**Assignment #1**

**Quantization**

**Submitted to**

Dr. Mai

Dr. Hala

Eng. Mohamed Khaled

**Submitted by**

|  |  |  |
| --- | --- | --- |
| **Name** | **Sec** | **BN** |
| **Abdelrahman Hamdy Ahmed** | **1** | **36** |
| **Zeyad Tarek Khairy** | **1** | **28** |

Contents

[**Part 1:…………** 4](#_Toc129905380)

[**Comment:** 4](#_Toc129905381)

[**Part 2:…………** 5](#_Toc129905382)

[**Comment:** 5](#_Toc129905383)

[**Part 3:…………** 6](#_Toc129905384)

[**Comment:** 6](#_Toc129905385)

[**Part 4:…………** 7](#_Toc129905386)

[**Comment:** 7](#_Toc129905387)

[**Part 5:…………** 9](#_Toc129905388)

[**Comment:** 9](#_Toc129905389)

[**Part 6:…………** 11](#_Toc129905390)

[**Comment:** 11](#_Toc129905391)

[**Index:** 13](#_Toc129905392)

Figures

[Figure 1 Fig 4](#_Toc129905445)

[Figure 2 fig 5](#_Toc129905446)

[Figure 3 Fig 6](#_Toc129905447)

[Figure 4 Fig 7](#_Toc129905448)

[Figure 5 Fig 9](#_Toc129905449)

[Figure 6 Fig 11](#_Toc129905450)

# **Part 1: Uniform Quantizer**

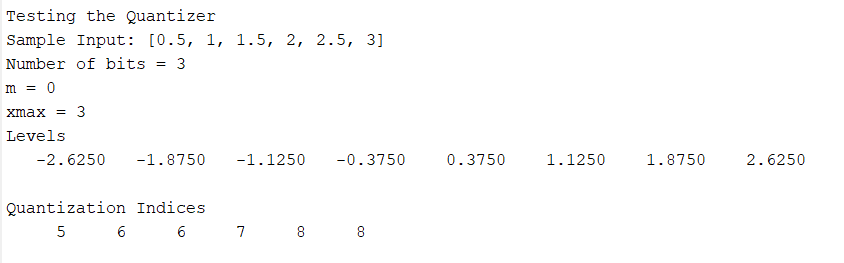


Figure 1 Fig

## **Comment:**

In this first part, our goal is to implement a functional Uniform Quantizer.

Let’s start by first understanding what a quantizer is. It’s basically the process where we map input values from a given set to output values in a smaller set with predetermined fixed voltage levels. We’re going to explain how this is done:

The function takes 4 parameters which are the input values in the form of an array (in\_val), the number of bits we need to sample at (n\_bits), the maximum value of x (xmax) and finally an indicator which specifies whether it’s a midrise or midtread quantization (m = 0 for midrise & 1 for midtread). We first need to calculate the number of quantization levels which is simple 2 power the number of bits (In the above example it’ll be 8). Then we calculate a variable called delta which represents the width of each interval from one level to another and it’s represented by double the xmax value divided by the number of levels. In order to alter the midrise & midtread part, we calculate another parameter called (d) which is (1 – m) \* delta / 2.

We are now ready to compute the range of levels that the input values will map to. The minimum value will be –xmax – d + delta and the maximum value will be xmax – d with the step set to delta.

Now the final part is to compute the quantization indices. We now have the quantization levels so with a simple for loop over the input values, we calculate the absolute difference between each input value and the levels we have in order to identify which level is most closest to this particular point and finally return its index.

# **Part 2: Uniform Dequantizer**

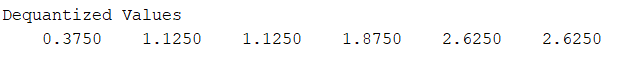


Figure 2 fig

## **Comment:**

Following the above example, these are the dequantized values from the indices we obtained in the quantizer. A dequantizer basically maps quantization indices to their corresponding levels.

The function takes in the quantization indices, and the other 3 parameters are the same as the quantizer which as the number of bits (n\_bits), maximum value of x (xmax), and m for specifying whether the quantization was done as midrise or midtread. Midrise means that if the levels plotted are observed at the origin, there would be a vertical edge which is basically transitioning from one level to another. In other words, the origin lies in the middle of a rising part of the staircase-like symmetric graph. On the other hand, midtread treats the zero as a level and there wouldn’t be an edge at the origin. We can say in this case that the origin lies in the middle of a tread in the staircase-like graph. Therefore, the graph won’t be symmetric anymore but with the right calculations we will have the same number of levels as midrise.

Just like we did with the quantizer, we will calculate the number of levels from the number of bits given (2 ^ n\_bits). Then get the step between each level defined as delta = 2 \* xmax divided by the number of levels.

Re-define the quantization levels by taking into account the value of m as midrise or midtread and finally map the indices given to its level in the levels array obtained.

These are the dequantization values we are interested in so they are returned from the function for further processing. You’ll notice the values printed above are found in the levels array but according to the quantization indices. Graphical tests will be shown starting from the next part.

# **Part 3: Testing the quantizer & dequantizer on a deterministic input**

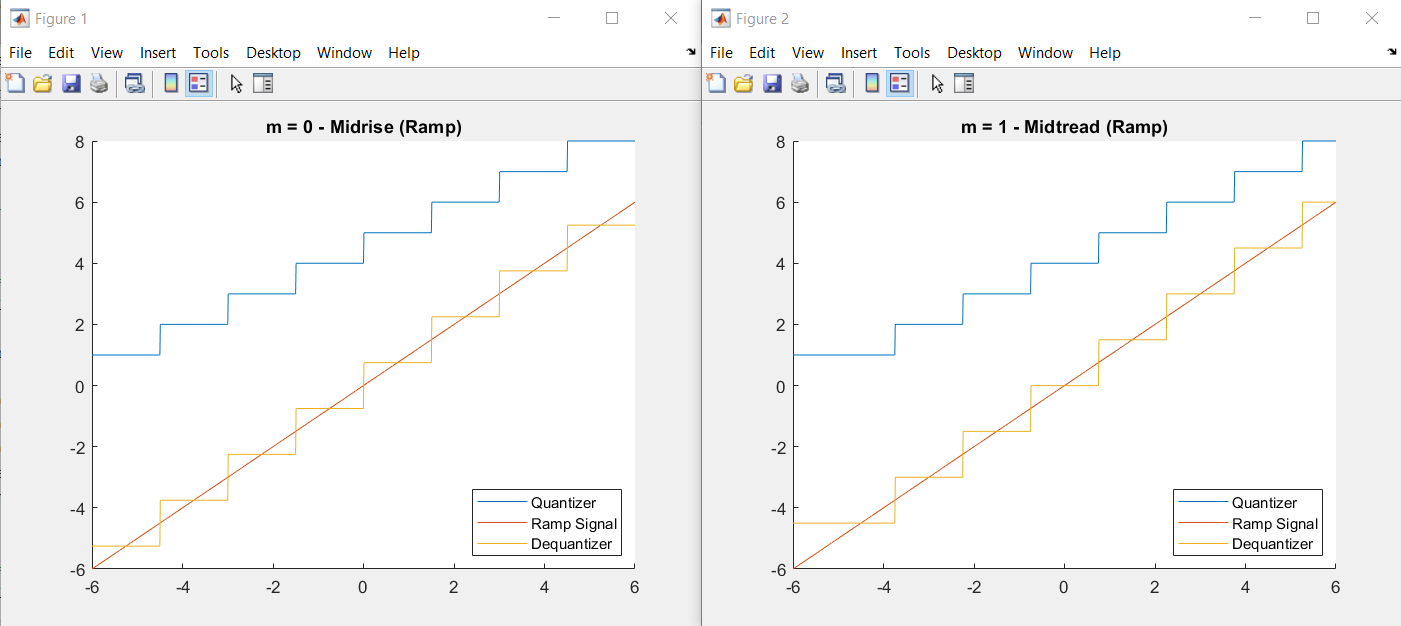


Figure 3 Fig

## **Comment:**

Now we will test the quantizer & dequantizer functions on a simple ramp signal ranging from -6 to 6 and the step is 0.01. The number of bits is set to 3 & the xmax value is 6. Both midrise and midtread samples with m set to 0 for midrise & 1 for midtread. The output above shows the original ramp signal which is the linear straight line, the quantized signal which is the upper staircase & the dequantized signal where the quantized signal is basically changed in amplitude to map to the original plot. As we see in both graphs, we have 8 levels (2^3 = 8) and for midrise the graph shows a rising edge at the origin, so the zero level isn’t considered in this case. For midtread, the quantizer has a zero output level and the graph isn’t symmetric. Also since the zero occupies a level, we assumed that for x values before the zero there will be 3 levels and also 4 levels after the zero.

# **Part 4: Testing on a random input signal**

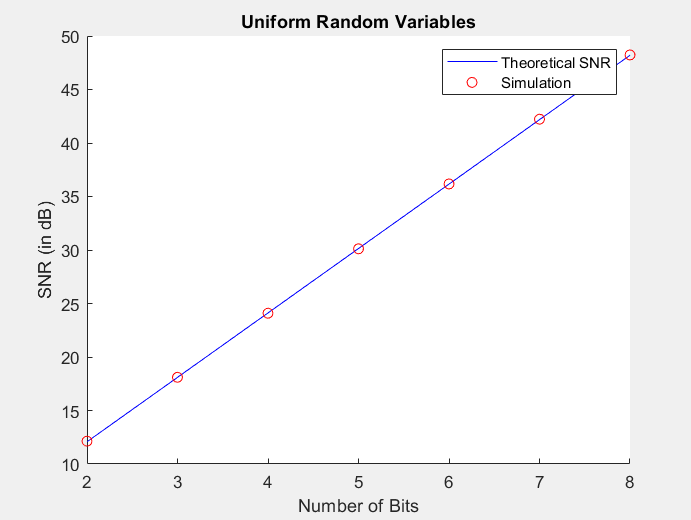


Figure 4 Fig

## **Comment:**

It’s now time to test out quantizer & dequantizer on a continuous set of uniform random variables. We used the matlab built-in function unifrnd(..) which takes in a lower and upper bound, and the size of the input wanted. In this requirement, the upper and lower bound are set to -5 & 5 respectively and the input size is set to 10000.

As for the other parameters regarding quantization and dequantization, xmax is 5 as given and a midrise quantization means that m = 0. The number of bits here vary because we are going to test on 7 different n\_bits values from 2 to 8 (Simply represented as an array). The new aspect here is that we are also interested to calculate the simulated SNR (Signal to Noise Ratio) and the theoretical SNR. We will use the formula discussed in the lecture to calculate the theoretical SNR which is: SNR = (3L^2 / xmax^2) \* P. L is each number of levels from the number of bits (2 ^ n\_bits), we already know xmax and P is the mean of the input squared.

The quantized error is the difference between the input signal and the dequantized values. From this, we will be able to calculate the simulated SNR which is represented by the mean of the input squared divided by mean of the quantization error squared. The above calculations are made of each number of bits (7 times on different n\_bits & therefore different number of levels).

The final step is now to plot both the theoretical SNR and the simulated SNR as shown above in Figure 4. We notice that both SNRs have almost the same values and this is certainly because the distribution of the random variables is uniform.

# **Part 5: Testing on non-uniform random input (mu = 0)**

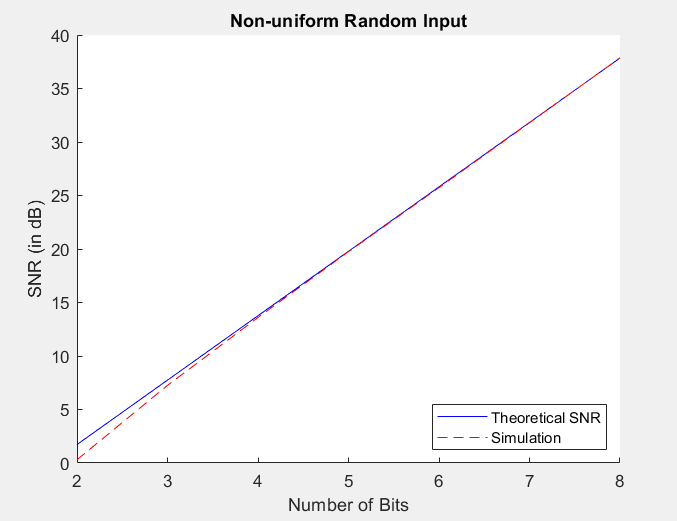


Figure 5 Fig

## **Comment:**

We’ll basically repeat what we did in Part 4 but now on random variables with an exponential distribution. For this, we will use the matlab built-in function exprnd which takes in the mu value for the exponential distribution and the size of the input (again 10000). There will be an adjustment here that we want to set the polarity of each of these values to have a probability of 50% from being positive or negative. Therefore, we’ll use the function Randi to generate either 0 or 1 with 0.5 probability and then multiply by 2 & subtract 1 to obtain either 1 or -1 which represents the sign. We will undergo the same steps as Part 4 regarding the quantization & dequantization but xmax will be set to the absolute maximum of these random variables. Theoretical SNR is calculated using the same equation but the simulated SNR is equal to the signal power divided by the error power, where the signal power is the variance of the input signal and the error power is the mean of the difference between the dequantized values and the input values squared (Quantization Error). By observing the plot, we can see that as the number of bits increases, the simulated SNR approaches the theoretical SNR until it eventually almost equals it, but for smaller number of bits the SNR of the simulation is lower than the Theoretical SNR. This is because the random variable distribution is in the shape of an exponential curve.

# **Part 6: Quantizing the non-uniform signal using a non-uniform μ law quantizer**

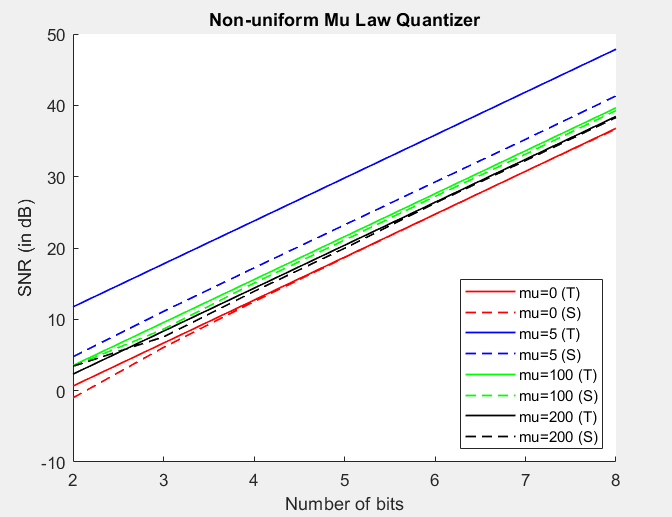


Figure 6 Fig

## **Comment:**

Using the same input from Part 5, we will now apply non-uniform μ quantization and compare their results. We will define values for μ as [0, 5, 100, and 200] for each number of bits we have from 2 to 8. An extra step before processing will be done and it is to normalize the input (dividing all input values by their absolute maximum).

For each value of μ we will calculate the theoretical and simulated SNR. Using the μ-Law concept for each μ and each number of bits, we will compress the signal before quantization and dequantization with the same 0.5 probability for the polarity by multiplying the absolute normalized input by the value of μ + 1 divided by 1 + μ both enclosed in a log function and then multiply the sign in the end. A special condition is made for when μ is equal to zero since it will result in an undefined behavior so we will just continue using the normalized input. After quantization & dequantization, we will take the dequantized signal and expand it to restore its original form. Now we will calculate the quantized error which is the difference between the input & the expanded signal. In the same way as Part 4 we will calculate the simulated SNR. For the theoretical SNR, if μ is greater than zero, we will use this formula SNR = 3L^2 / log(1 + μ)^2. However, if μ is zero then we will use the original formula as in part 4. Lastly, the output is shown above when we plotted the theoretical and simulated SNR for each μ.

Conclusion: Increasing the value of μ decreases the difference between the theoretical and simulated SNR. Similarly with part 5, increasing the number of bits will decrease the difference between them as well using an exponential distribution for the random variables generated.

# **Index:**

close all;

clear all;

% ---------------- Req 3 ----------------

ramp = -6:0.01:6;

% m = 0

q = UniformQuantizer(ramp, 3, 6, 0);

dq = UniformDequantizer(q, 3, 6, 0);

figure();

hold on

plot(ramp, q);

plot(ramp, ramp);

plot(ramp, dq);

title('m = 0 - Midrise (Ramp)');

legend('Quantizer', 'Ramp Signal', 'Dequantizer');

hold off

% m = 1

q = UniformQuantizer(ramp, 3, 6, 1);

dq = UniformDequantizer(q, 3, 6, 1);

figure();

hold on

plot(ramp, q);

plot(ramp, ramp);

plot(ramp, dq);

title('m = 1 - Midtread (Ramp)');

legend('Quantizer', 'Ramp Signal', 'Dequantizer');

hold off

% ---------------- Req 4 ----------------

% Continuous Uniform Random Variables between -5 & 5

N = 10000;

lowerBound = -5;

upperBound = 5;

input = unifrnd(lowerBound, upperBound, 1, N);

% Set xmax = 5 & m to midrise as given in the requirement

xmax = 5;

m = 0;

% Define the number of bits array & get its length

n\_bits = [2, 3, 4, 5, 6, 7, 8];

len = length(n\_bits);

% Initialize Theoretical and Simulated SNR

theoreticalSNR = zeros(1, len);

simulatedSNR = zeros(1, len);

% Define the levels array (2 ^ n) & calculate P

Levels = 2 .^ n\_bits;

P = mean(input .^ 2);

for i = 1:len

% Get the dequantized signal with corresponding inputs

q = UniformQuantizer(input, n\_bits(i), xmax, m);

dq = UniformDequantizer(q, n\_bits(i), xmax, m);

% Calculate the Theoretical SNR for each number of bits

theoreticalSNR(i) = 10 \* log10(P / (((xmax) .^ 2) / (3 \* ((Levels(i) .^ 2)))));

% Get the quantization error by subtracting input values & the

% dequantized signal

quantizedError = input - dq;

% Calculate the simulated SNR using the quantized error

simulatedSNR(i) = 10 \* log10(P / mean(quantizedError .^ 2));

end

% Sketching the simulation & Theoretical SNR

figure();

hold on

plot(n\_bits, theoreticalSNR, 'B-');

plot(n\_bits, simulatedSNR, 'Ro');

xlabel('Number of Bits');

ylabel('SNR (in dB)');

title('Uniform Random Variables');

legend('Theoretical SNR', 'Simulation');

hold off

% ---------------- Req 5 ----------------

% Define polarity & give it a value +/- with probability 0.5

polarity = 2 \* randi([0 1], 1, N) - 1;

% Sample magnitude following an exponential distribution

magnitude = exprnd(1, 1, N);

% Apply the random polarity to magnitude

input = magnitude .\* polarity;

% Re-initialize Theoretical & Simulated SNR

theoreticalSNR = zeros(1, len);

simulatedSNR = zeros(1, len);

% Calculate signal power as the variance of the input

signalPower = var(input);

% Re-define P & xmax accordingly (m is still 0)

P = mean(input .^ 2);

xmax = max(abs(input));

for i = 1:len

% Get the dequantized signal with corresponding inputs

q = UniformQuantizer(input, n\_bits(i), xmax, m);

dq = UniformDequantizer(q, n\_bits(i), xmax, m);

% Calculate the error power

errorPower = mean((dq - input) .^ 2);

% Get theoratical SNR (similar to Req 4)

theoreticalSNR(i) = 10 \* log10(P / (((xmax) .^ 2) / (3 \* ((Levels(i) .^ 2)))));

% Define simulated SNR as signal power divided by error power

simulatedSNR(i) = 10 \* log10(signalPower / errorPower);

end

% Plot

figure();

hold on

plot(n\_bits, theoreticalSNR, 'B-');

plot(n\_bits, simulatedSNR, 'R--');

xlabel('Number of Bits');

ylabel('SNR (in dB)');

title('Non-uniform Random Input');

legend('Theoretical SNR', 'Simulation');

hold off

% ---------------- Req 6 ----------------

% Non-uniform mu quantization

mu = [0 ,5, 100, 200];

colors = ['r' , 'b' , 'g' , 'k'];

% Normalize the Input

input\_N = input / xmax;

figure();

hold on

for j = 1:length(mu)

% Initialize Theoretical and Simulated SNR

simulatedSNR = n\_bits;

theoreticalSNR = zeros(1, len);

for i = 1:len

% Compress the Signal if mu is greater than zero

if (mu(j) > 0)

y = polarity .\* (log(1+mu(j) \* abs(input\_N)) / log(1+mu(j)));

else

y = input\_N;

end

ymax = max(abs(y));

q = UniformQuantizer(y, n\_bits(i), ymax, m);

dq = UniformDequantizer(q, n\_bits(i), ymax, m);

if (mu(j) > 0)

z = polarity .\*(((1+mu(j)) .^ abs(dq)-1) / mu(j));

else

z = dq;

end

ex = z \* xmax;

error = abs(input - ex);

simulatedSNR(i) = 10\*log10(mean(input .^ 2) / mean(error .^ 2));

if (mu(j) > 0)

theoreticalSNR(i) = 10\*log10((3\*(Levels(i) .^ 2))/((log(1 + mu(j))) .^ 2));

else

theoreticalSNR(i) = 10\*log10(P / (((xmax) .^ 2) / (3 \* ((Levels(i) .^ 2)))));

end

end

% Plot

plot(n\_bits, theoreticalSNR, sprintf('%s-' , colors(j)) , 'LineWidth', 1);

plot(n\_bits, simulatedSNR, sprintf('%s--' , colors(j)) , 'LineWidth', 1);

end

hold off

title('Non-uniform Mu Law Quantizer');

xlabel('Number of bits');

ylabel('SNR (in dB)');

legend('mu=0 (T)', 'mu=0 (S)', 'mu=5 (T)', 'mu=5 (S)', 'mu=100 (T)', 'mu=100 (S)', ...

'mu=200 (T)', 'mu=200 (S)');

% ---------------- Req 1 ----------------

function q\_ind = UniformQuantizer(in\_val, n\_bits, xmax, m)

% Calculate the number of quantization levels

numberOfLevels = 2^n\_bits;

% Get the width of each interval

delta = 2\*xmax / numberOfLevels;

% Define the range of the quantizer reconstruction levels

d = ((1 - m)\*delta) / 2;

levels = (-xmax - d + delta):delta:(xmax - d);

% Get size of the input

size = length(in\_val);

% Initialize quantized indices

q\_ind = zeros(1, size);

% Find the closest reconstruction level to the input sample

for i = 1:size

M = abs(in\_val(i) - levels);

q\_ind(i) = find(M == min(M), 1);

end

end

% ---------------- Req 2 ----------------

function deq\_val = UniformDequantizer(q\_ind, n\_bits, xmax, m)

% Calculate the number of levels

numberOfLevels = 2 ^ n\_bits;

% Calculate Delta

delta = 2 \* xmax / numberOfLevels;

% Get the Levels

d = ((1 - m)\*delta) / 2;

levels = (-xmax - d + delta):delta:(xmax - d);

% Restore each level to original amplitude

deq\_val = levels(q\_ind);

end