Cairo University
Faculty of Engineering

Department of Computer Engineering

ELC 325B – Spring 2024

Digital Communications

Assignment #1

Quantization

Submitted to

Dr. Hala

Dr. Mai

Eng. Mohamed Khaled

Submitted by

Name	Sec	BN
Mennatallah Ahmed Moustafa	2	25
Michael Ehab Mikhail	2	5

Contents

Figures

Part 1: Uniform Scalar Quantizer Implementation

```python
# Req 1
def UniformQuantizer(in_val, n_bits, xmax, m):
    num_of_levels=2**n_bits
    delta=2*xmax/num_of_levels
    offset=(m) * (delta / 2)
    quantization_index = np.floor((in_val - (offset- xmax)) / delta)
    quantization_index[quantization_index < 0] = 0
    return quantization_index
```

Figure 1 ***Uniform Scalar Quantizer Implementation***

Comment:

- Calculates quantization index (0 to L-1) for each input value with the following steps:
  1. Calculate number of levels through the formula
  $$num\ of\ levels = 2^{n\_bits}$$
  2. Calculate delta through the formula
  $$delta = \frac{2 * xmax}{num\ of\ levels}$$
  3. Calculate the offset which is in terms of m (its value changes according to the chosen quantizer midrise or midtread)
  $$offset = \frac{m * delta}{2}$$
  4. Calculate the quantization index through the formula
  $$floor\left(\frac{input\_value - (offset - xmax)}{delta}\right)$$
  5. Replace any negative quantization index by zero

Part 2: Uniform Scalar De-quantizer Implementation

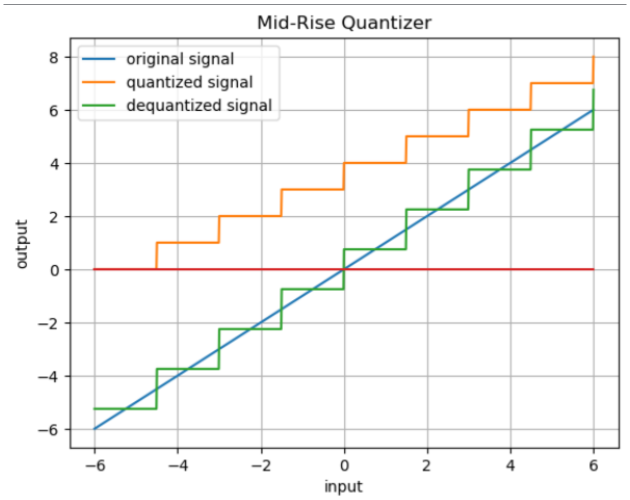
```
# Req 2
def UniformDequantizer(q_ind,n_bits,xmax,m):
    num_of_levels=2**n_bits
    delta=2*xmax/num_of_levels
    offset=(m) * (delta / 2)
    dequantization_value=((q_ind) * delta) + (offset+ (delta / 2) - xmax)
    return dequantization_value
```

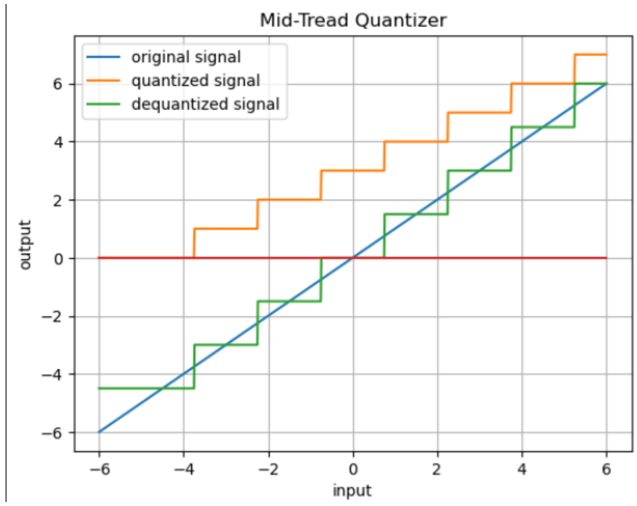
Figure 2 ***Uniform Scalar De-quantizer Implementation***

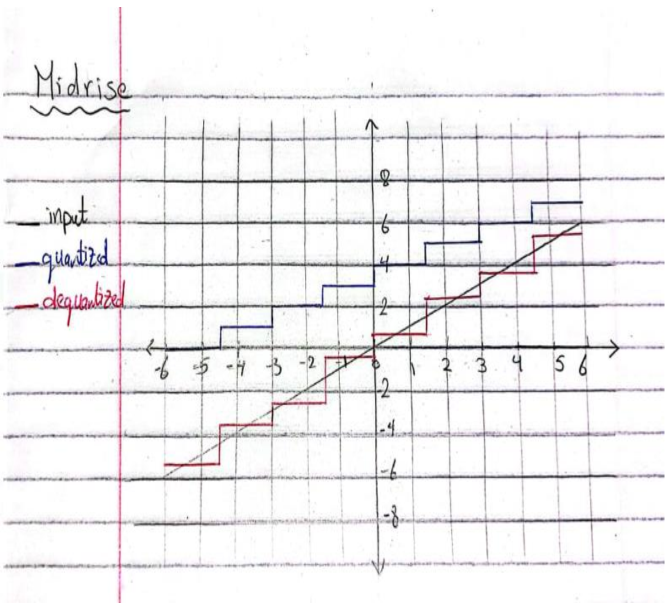
Comment:

- Calculates output value given quantization level:
  1. Calculate number of levels through the formula
  $$num\ of\ levels = 2^{n\_bits}$$
  2. Calculate delta through the formula
  $$delta = \frac{2 * xmax}{num\ of\ levels}$$
  3. Calculate the offset which is in terms of m (its value changes according to the chosen quantizer midrise or midtread)
  $$offset = \frac{m * delta}{2}$$
  4. Calculate the output value through the formula
  $$(level * delta) + offset + \frac{delta}{2} - xmax$$

Part 3: Testing the quantizer/de-quantizer functions on a deterministic input







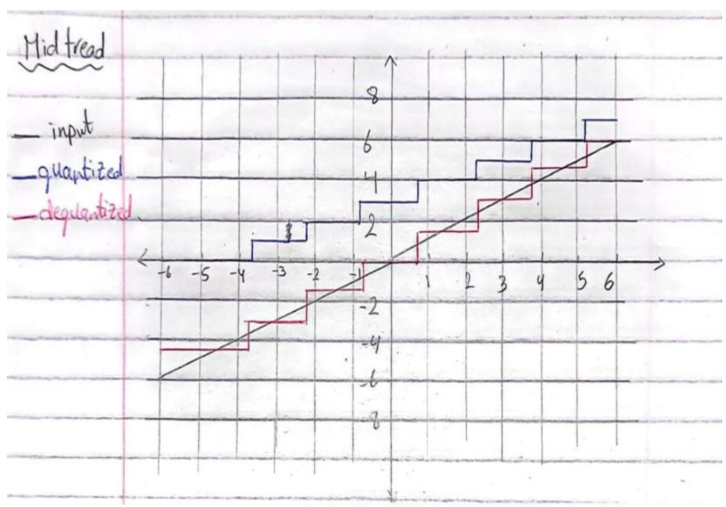


Figure 3&4 ***Quantizer/de-quantizer plot and hand analysis for input ramp signal***

Comment:

- We can notice that the plot follows the configuration of
  - For mid-rise
    - Levels are $\pm\frac{\mathrm{delta}}{2}$, $\pm\frac{3*\mathrm{delta}}{2}$, $\pm\frac{5*\mathrm{delta}}{2}$, $\pm\frac{7*\mathrm{delta}}{2}$
    - For our question,
      - n_bits=3, num of levels =8 , xmax=6 , delta=12/8= 1.5
      - by substituting levels are ± 0.75 , ± 2.25, ± 3.75, ± 5.25
  - For mid-tread
    - Levels are $\pm delta$, $\pm 2 * delta$, $\pm 3 * delta$, $\pm 4 * \mathrm{delta}$
    - For our question,
      - n_bits = 3, num of levels =8 , xmax = 6 , delta=12/8= 1.5
      - by substituting levels are ± 1.5, ± 3, ± 4.5, ± 6

Part 4: Testing the input on a random input signal

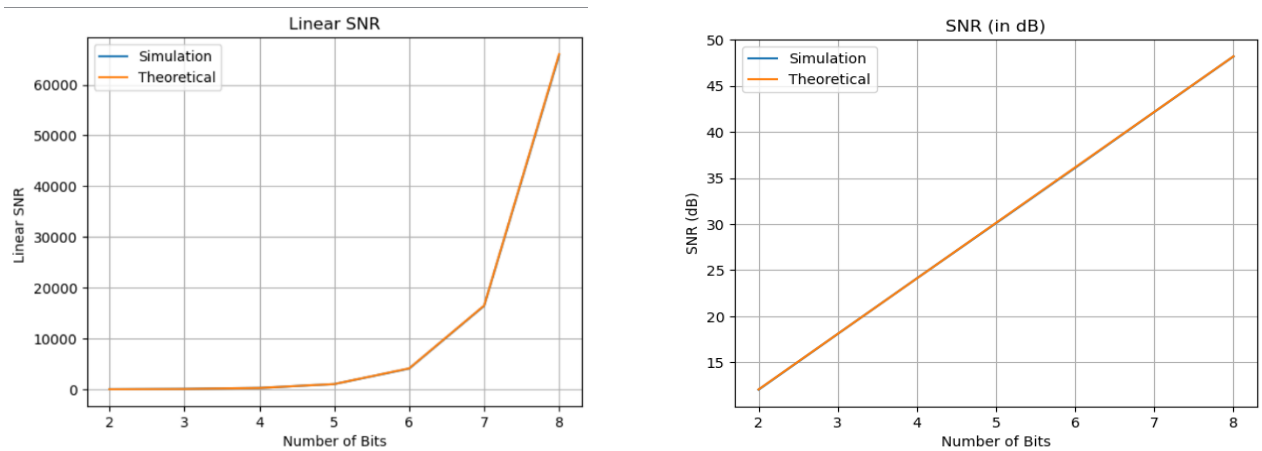


*Figure 5: **plotting simulation and the theoretical SNR***

Comment:

- We are utilizing uniform data with a uniform quantizer and de-quantizer, then the difference between the simulated signal-to-noise ratio (SNR) and the theoretical SNR is extremely minimal.
- This near-zero difference is consistent across all bit resolutions.
- The Simulation SNR is calculated through the formula

$$E(input^2)/E(quantization\ error^2)$$

- The theoretical error is calculated through the formula

$$\frac{3 * (2^{nbits})^2 * \ E(input^2)}{xmax^2}$$

Part 5: Testing the uniform quantizer on a non-uniform random input

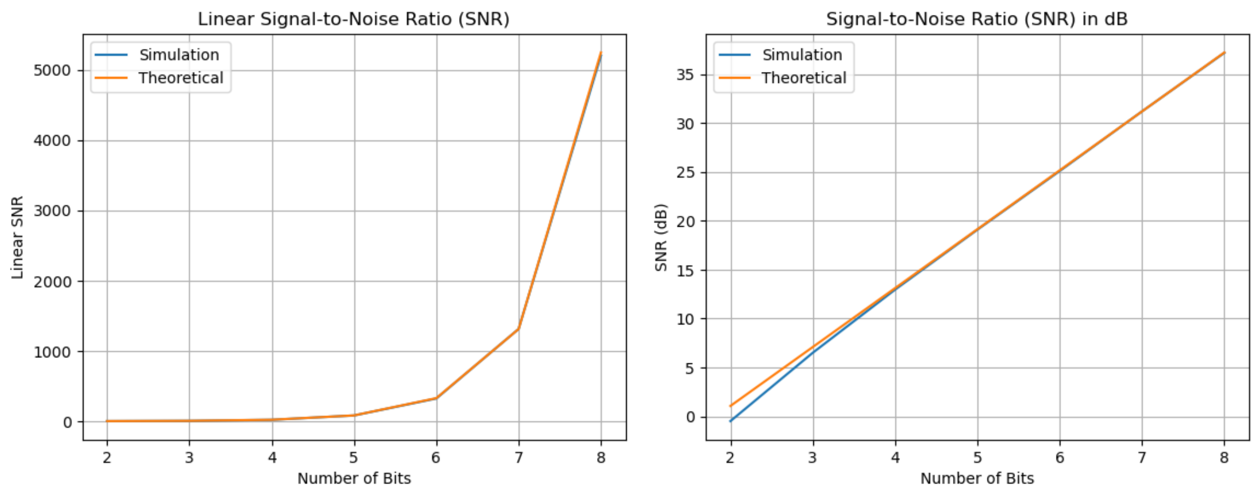


Figure 6: Linear and dB SNR with non-uniform random input

Comment:

- A non-uniform signal (i.i.d. samples with random polarity and exponentially distributed magnitude given by the probability density function (PDF), $f(x) = e^{-x}$) was fed into a uniform quantizer.
- Uniform quantizers work best with uniform signals, but this experiment used a non-uniform one.
- With few bits (low resolution), the Signal-to-Noise Ratio (SNR) suffered due to the mismatch between the signal and quantizer.
- Increasing the number of bits (higher resolution) improved the SNR, making the uniform quantizer more effective.
- While a uniform quantizer can work with non-uniform signals given enough resolution, it might not be the most efficient approach.
- Simulation , theoretical SNR are calculated as the previous part (requirement 4)

Part 6: Testing the non-uniform input signal using a non-uniform µ law quantizer

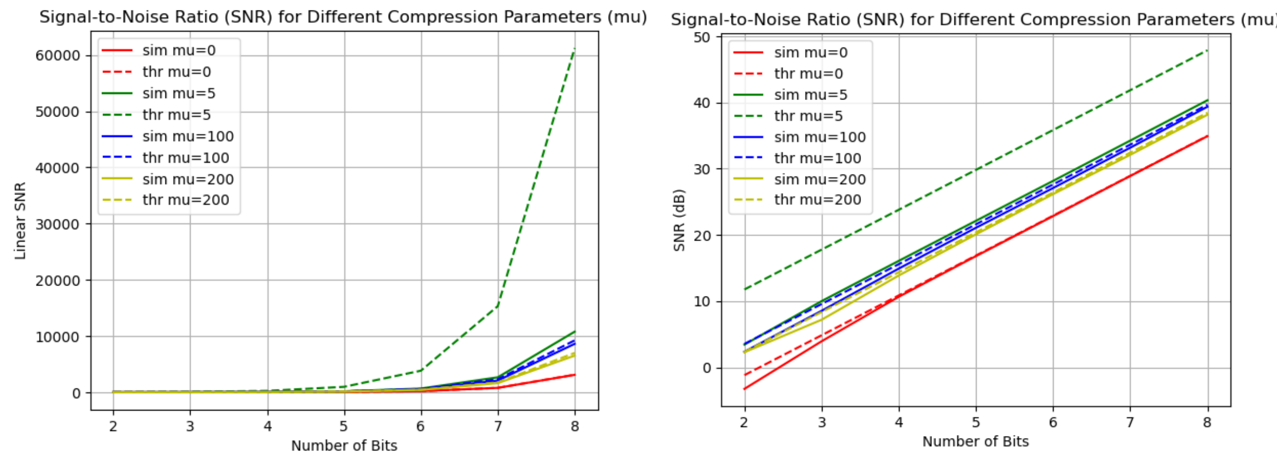


Figure 7: SNR for different Mu values

Comment:

- The experiment tested a µ-law quantizer on a non-uniform signal (same as before).
- Increasing the µ value in the µ-law quantizer improved the match between theoretical and actual SNR. This suggests the quantizer performs better with higher µ for non-uniform signals.
- µ-law compression helps represent the large dynamic range of the signal (exponential distribution) more effectively.
- When µ is 0, the µ-law quantizer acts like a uniform quantizer (requirement 5). This aligns with the previous experiment's results using a uniform quantizer on the same signal (acting as a control point).
- The µ-law quantizer, with an appropriate µ, adapts better to non-uniform signals compared to a uniform quantizer. This leads to a more accurate representation, especially for low-amplitude parts of the signal, resulting in a higher SNR.

- The theoretical error is calculated through the formula
    - at μ =0

$$\frac{3*(2^{nbits})^2 * \ E(input^2)}{xmax^2}$$

    - otherwise

$$\frac{3*(2^{nbits})^2}{\log(1+\mu)^2}$$

- The Compression formula is

$$\frac{\pm\log\,(1+\mu|x|)}{\log\,(1+\mu)}$$

- The expansion formula is

$$\frac{\pm(1+\mu)^{|x|}-1}{\mu}$$

Index:

```python
from math import floor
import numpy as np
import matplotlib.pyplot as plt

# Req 1
def UniformQuantizer(in_val, n_bits, xmax, m):
    num_of_levels=2**n_bits
    delta=2*xmax/num_of_levels
    offset=(m) * (delta / 2)
    quantization_index = np.floor((in_val - (offset- xmax)) / delta)
    quantization_index[quantization_index < 0] = 0
    return quantization_index
# Req 2
def UniformDequantizer(q_ind,n_bits,xmax,m):
    num_of_levels=2**n_bits
    delta=2*xmax/num_of_levels
    offset=(m) * (delta / 2)
    dequantization_value=((q_ind) * delta) + (offset+ (delta / 2) - xmax)
    return dequantization_value
def plot_four_outputs(x1, y1, x2, y2, x3, y3, x4, y4, label, labelx, labely,legends):
  plt.figure()
  plt.plot(x1, y1, label=legends[0])
  plt.plot(x2, y2, label=legends[1])
  plt.plot(x3, y3, label=legends[2])
  plt.plot(x4, y4)
  plt.title(label)
  plt.xlabel(labelx)
  plt.ylabel(labely)
  plt.grid(True)
  plt.legend()
  plt.show()
def plot_two_outputs(x1, y1, x2, y2, label, labelx, labely, legends):
    plt.figure()
    plt.plot(x1, y1, label=legends[0])
    plt.plot(x2, y2, label=legends[1])
    plt.title(label)
    plt.xlabel(labelx)
    plt.ylabel(labely)
    plt.grid(True)
    plt.legend()
     plt.show()
def convert_snr_to_db(snr):
    """Converts signal-to-noise ratio (SNR) to decibel (dB) scale."""
    return 10 * np.log10(snr)
def compress_signal(x, u, sign):
    return sign * (np.log(1 + u * np.abs(x)) / np.log(1 + u))
```

```python
def expand_signal(x, u, sign):
    return sign * (((1 + u) ** np.abs(x) - 1) / u)
# Req 3
# Generate a linearly spaced array from -6 to 6 with 1001 points
input_signal = np.linspace(-6, 6, 1001)
# Set the number of bits for quantization and the maximum value of the signal
number_of_bits = 3
maximum_value = 6
# Mid-rise quantizer (midpoint=0)
midpoint = 0
quantized_signal_midrise = UniformQuantizer(input_signal, number_of_bits, maximum_value, midpoint)
dequantized_signal_midrise = UniformDequantizer(quantized_signal_midrise, number_of_bits, maximum_value, midpoint)
plot_four_outputs(input_signal,input_signal,input_signal,quantized_signal_midrise,input_signal,dequantized_signal_
midrise,input_signal,input_signal*0,"Mid-Rise Quantizer","input","output",["original signal","quantized
signal","dequantized signal"])
# Mid-tread quantizer (midpoint=1)
midpoint = 1
quantized_signal_midtread = UniformQuantizer(input_signal, number_of_bits, maximum_value, midpoint)
dequantized_signal_midtread = UniformDequantizer(quantized_signal_midtread, number_of_bits, maximum_value,
midpoint)
plot_four_outputs(input_signal,input_signal,input_signal,quantized_signal_midtread,input_signal,dequantized_signal
_midtread,input_signal,input_signal*0,"Mid-Tread Quantizer","input","output",["original signal","quantized
signal","dequantized signal"])
#Req 4
# Generate a list of random numbers within a specified range
signal_samples = np.random.uniform(-5, 5, size=10000)
# Calculate the maximum absolute value from the signal samples
max_signal_value = 5
# Initialize the midpoint value for quantization
midpoint = 0
# Lists to store Signal-to-Noise Ratio (SNR) values from simulation and theoretical calculation
simulation_snr_list = []
theoretical_snr_list = []
# Loop through different numbers of bits for quantization
for num_bits in range(2, 9):
    # Quantize the signal based on the number of bits and range
    quantized_signal = UniformQuantizer(signal_samples, num_bits, max_signal_value, midpoint)
    # Dequantize the signal back to the original value range
    dequantized_signal = UniformDequantizer(quantized_signal, num_bits, max_signal_value, midpoint)
    # Calculate the quantization error
    quantization_error = abs(signal_samples - dequantized_signal)
    quantization_error = np.asarray(quantization_error)
    # Calculate and store the simulation SNR value
    simulation_snr = np.mean(signal_samples**2) / np.mean(quantization_error**2)
    simulation_snr_list.append(simulation_snr)
    # Calculate and store the theoretical SNR value
    theoretical_factor = (3 * ((2**num_bits)**2)) / (max_signal_value**2)
    theoretical_snr = theoretical_factor * np.mean(signal_samples**2)
```

```
    theoretical_snr_list.append(theoretical_snr)
# Create an array of bit numbers for plotting
bit_numbers = np.arange(2, 9)
# Convert SNR values from linear scale to decibels (dB)
simulation_snr_db = convert_snr_to_db(simulation_snr_list)
theoretical_snr_db = convert_snr_to_db(theoretical_snr_list)
# Plot the Linear SNR values against the number of bits
plot_two_outputs(x1=bit_numbers, y1=simulation_snr_list, x2=bit_numbers, y2=theoretical_snr_list,
                 label="Linear SNR", labelx="Number of Bits", labely="Linear SNR ",
                 legends=['Simulation', 'Theoretical'])
# Plot the SNR values in dB against the number of bits
plot_two_outputs(x1=bit_numbers, y1=simulation_snr_db, x2=bit_numbers, y2=theoretical_snr_db,
                 label="SNR (in dB)", labelx="Number of Bits", labely="SNR (dB)",
                 legends=['Simulation', 'Theoretical'])
#Req 5
# Define the size of the sample array for random number generation
sample_dimensions = (1, 10000)
# Generate a sample of exponentially distributed random numbers
exponential_random_sample = np.random.exponential(scale=1, size=sample_dimensions)
# Create an array of random signs (-1 or 1)
random_signs = (np.random.randint(0, 2, size=sample_dimensions) * 2) - 1
# Combine the exponential samples with their respective signs to form bipolar samples
bipolar_samples = exponential_random_sample * random_signs
# Determine the maximum absolute value from the bipolar samples
max_bipolar_value = np.max(np.abs(bipolar_samples))
# Initialize lists to hold SNR values for simulation and theoretical analysis
simulated_snr_values = []
theoretical_snr_values = []
# Iterate over a range of bit resolutions for quantization
for bit_resolution in range(2, 9):
    # Quantize the bipolar samples
    quantized_samples = UniformQuantizer(bipolar_samples, bit_resolution, max_bipolar_value, 0)
    # Dequantize the samples to recover the signal
    recovered_samples = UniformDequantizer(quantized_samples, bit_resolution, max_bipolar_value, 0)
    # Compute the quantization error
    quantization_errors = np.abs(bipolar_samples - recovered_samples)
    # Calculate and store the simulated SNR value
    simulated_snr = np.mean(bipolar_samples**2) / np.mean(quantization_errors**2)
    simulated_snr_values.append(simulated_snr)
    # Calculate and store the theoretical SNR value
    theoretical_snr = (3 * ((2**bit_resolution)**2)) / (max_bipolar_value**2) * np.mean(bipolar_samples**2)
    theoretical_snr_values.append(theoretical_snr)
# Create an array of bit numbers for plotting
bit_numbers = np.arange(2, 9)
# Convert SNR values from linear scale to decibels (dB)
simulation_snr_db = convert_snr_to_db(simulated_snr_values)
theoretical_snr_db = convert_snr_to_db(theoretical_snr_values)
# Plot the Linear SNR values against the number of bits
```

```python
plot_two_outputs(x1=bit_numbers, y1=simulated_snr_values, x2=bit_numbers, y2=theoretical_snr_values,
                 label="Linear Signal-to-Noise Ratio (SNR)", labelx="Number of Bits", labely="Linear SNR",
                 legends=['Simulation', 'Theoretical'])
# Plot the SNR values in dB against the number of bits
plot_two_outputs(x1=bit_numbers, y1=simulation_snr_db, x2=bit_numbers, y2=theoretical_snr_db,
                 label="Signal-to-Noise Ratio (SNR) in dB", labelx="Number of Bits", labely="SNR (dB)",
                 legends=['Simulation', 'Theoretical'])
#Req 6
normalized_samples = bipolar_samples / max_bipolar_value
colors = ['r', 'g', 'b', 'y']
mu_values = [0, 5, 100, 200]
n_bits_range = range(2, 9)
sim_snr, thr_snr = [], []
for mu in mu_values:
    sim_snr_mu, thr_snr_mu = [], []
    compressed_samples = bipolar_samples if mu == 0 else compress_signal(normalized_samples, mu, random_signs)
    ymax = np.max(np.abs(compressed_samples))
    for n_bits in n_bits_range:
        quantized = UniformQuantizer(compressed_samples, n_bits, ymax, 0)
        dequantized = UniformDequantizer(quantized, n_bits, ymax, 0)
        if mu != 0:
            expanded_samples = expand_signal(dequantized, mu, random_signs)
            dequantized = expanded_samples * max_bipolar_value
        error = np.abs(bipolar_samples - dequantized)
        sim_snr_mu.append(np.mean(bipolar_samples**2) / np.mean(error**2))
        if mu != 0:
            scale = 3 * (2**n_bits)**2
            thr_snr_mu.append(scale / (np.log(1 + mu))**2)
        else:
            scale = 3 * (2**n_bits)**2 / max_bipolar_value**2
            thr_snr_mu.append(scale * np.mean(bipolar_samples**2))
    sim_snr.append(sim_snr_mu)
    thr_snr.append(thr_snr_mu)
# Linear Plot
plt.figure()
for i, mu in enumerate(mu_values):
    label_sim = f"sim mu={mu}"
    label_thr = f"thr mu={mu}"
    plt.plot(n_bits_range, sim_snr[i], '-', color=colors[i], label=label_sim)
    plt.plot(n_bits_range, thr_snr[i], '--', color=colors[i], label=label_thr)
plt.title("Signal-to-Noise Ratio (SNR) for Different Compression Parameters (mu)")
plt.xlabel("Number of Bits")
plt.ylabel("Linear SNR")
plt.legend()
plt.grid(True)
plt.show()
# Plot in dB
plt.figure()
```

```
for i, mu in enumerate(mu_values):
    label_sim = f"sim mu={mu}"
    label_thr = f"thr mu={mu}"
    plt.plot(n_bits_range, convert_snr_to_db(sim_snr[i]), '-', color=colors[i], label=label_sim)
    plt.plot(n_bits_range, convert_snr_to_db(thr_snr[i]), '--', color=colors[i], label=label_thr)
plt.title("Signal-to-Noise Ratio (SNR) for Different Compression Parameters (mu)")
plt.xlabel("Number of Bits")
plt.ylabel("SNR (dB)")
plt.legend()
plt.grid(True)
plt.show()
```