UNIVERSIDAD DE VALPARAÍSO
Escuela de Ingeniería Civil Biomédica
CBM414 Procesamiento digital de señales biomédicas

Laboratory 3

Maximiliano Antonio Gaete Pizarro

1 Problem 1: Quantization of an Audio Signal

1.1 Explain in detail what the function `cuantizar` does.

The function `cuantizar(s, bit)` quantizes an input signal s using a specified number of bits bit. Below is a detailed explanation of each step:

```
def cuantizar(s,bit):
        # s: senal de entrada
        # bit: bits de cuantificacion
        Plus1=np.power(2, (bit-1))
        X=s*Plus1
        X=np.round(X)
        X=np.minimum(Plus1-1.0,X)
        X=np.maximum(-1.0*Plus1,X)
        X=X/Plus1
        return X
```

1. **Calculate the Positive Quantization Levels:**

   ```
   Plus1 = np.power(2, (bit - 1))
   ```

   Explanation: Computes $2^{(\text{bit}-1)}$, which determines the number of positive quantization levels available. For instance, if bit $= 4$, then Plus1 $= 8$.

2. **Scale the Input Signal:**

   ```
   X = s * Plus1
   ```

   Explanation: Scales the input signal s by multiplying it with Plus1 to adjust it to the range of the quantization levels.

3. **Round the Scaled Signal:**

   ```
   X = np.round(X)
   ```

   Explanation: Rounds the values of X to the nearest integer, assigning each sample to the nearest quantization level.

4. **Limit the Quantized Values to Prevent Saturation:**

   ```
   X = np.minimum(Plus1 - 1.0, X)
   X = np.maximum(-1.0 * Plus1, X)
   ```

   Explanation: Ensures that X does not exceed the maximum and minimum levels allowed by the number of bits. It constrains X within the range $[-2^{(\text{bit}-1)}, 2^{(\text{bit}-1)} - 1]$.

5. **Normalize the Quantized Signal:**

   ```
   X = X / Plus1
   return X
   ```

   Explanation: Normalizes the quantized signal by dividing by Plus1, returning it to the original amplitude range.

Summary: The function `cuantizar` scales the input signal, rounds the values to assign them to discrete quantization levels, limits the values to prevent saturation, and normalizes the quantized signal. This results in a quantized version of the original signal using the specified number of bits.

1.2 Re-quantize the audio signal to 2, 4, and 8 bits. Listen to the signals and describe what you perceive.

We re-quantize the normalized audio signal `x_norm` using the `cuantizar` function:

```
# Re-quantization of the signal to 2, 4, and 8 bits
xq_2bit = cuantizar(x_norm, 2)
xq_4bit = cuantizar(x_norm, 4)
xq_8bit = cuantizar(x_norm, 8)
```

Perception after listening:

- **2 bits:** The audio sounds heavily distorted with significant quantization noise. Due to only 4 quantization levels, the signal loses most of its detail, and artifacts are prominent.
- **4 bits:** The audio quality improves but still contains noticeable distortion. With 16 quantization levels, the signal retains more detail, but quantization noise is still audible.
- **8 bits:** The audio closely resembles the original signal. With 256 quantization levels, the quantization noise is minimal, and the signal preserves much of the original quality.

1.3 Plot the original and quantized signals for 2, 4, and 8 bits (three plots). What do you observe in the graphs?

We plot a segment of the original signal and its quantized versions for visualization.

```
# Define a segment of the signal for visualization
start_sample = 10000
end_sample = 10500

# Plot for 2 bits
plt.figure(figsize=(17, 4))
plt.plot(t[start_sample:end_sample], x_norm[start_sample:end_sample], label='Original')
plt.step(t[start_sample:end_sample], xq_2bit[start_sample:end_sample], label='Quantized␣(2␣
    bits)', where='mid', alpha=0.7)
plt.title('Original␣vs.␣Quantized␣Signal␣(2␣bits)')
plt.xlabel('Time␣[s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()

# Plot for 4 bits
plt.figure(figsize=(17, 4))
plt.plot(t[start_sample:end_sample], x_norm[start_sample:end_sample], label='Original')
plt.step(t[start_sample:end_sample], xq_4bit[start_sample:end_sample], label='Quantized␣(4␣
    bits)', where='mid', alpha=0.7)
plt.title('Original␣vs.␣Quantized␣Signal␣(4␣bits)')
plt.xlabel('Time␣[s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()

# Plot for 8 bits
plt.figure(figsize=(17, 4))
plt.plot(t[start_sample:end_sample], x_norm[start_sample:end_sample], label='Original')
plt.step(t[start_sample:end_sample], xq_8bit[start_sample:end_sample], label='Quantized␣(8␣
    bits)', where='mid', alpha=0.7)
plt.title('Original␣vs.␣Quantized␣Signal␣(8␣bits)')
plt.xlabel('Time␣[s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()
```

- **For 2 bits:**

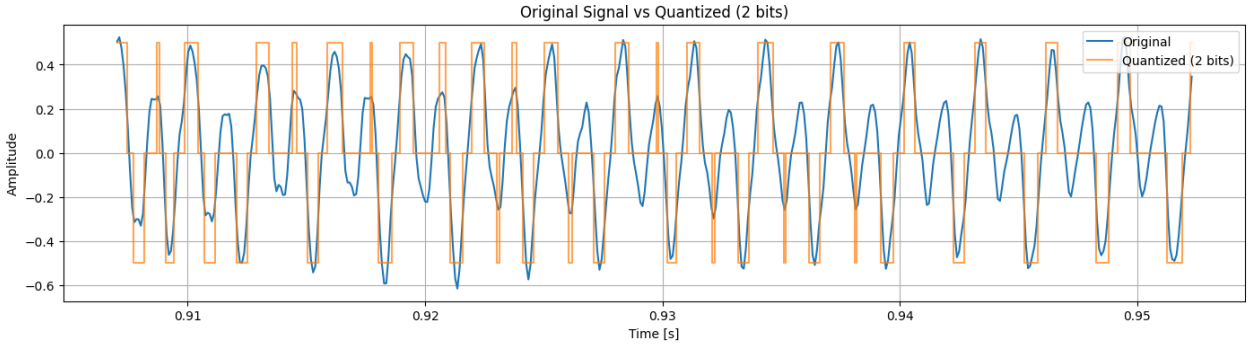

Figure 1: Original vs. Quantized Signal (2 bits)

- **For 4 bits:**

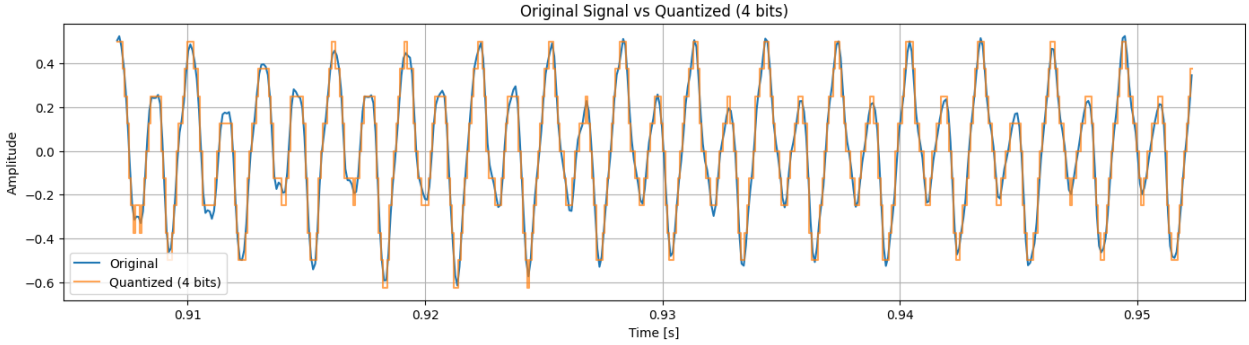

Figure 2: Original vs. Quantized Signal (4 bits)

- **For 8 bits:**

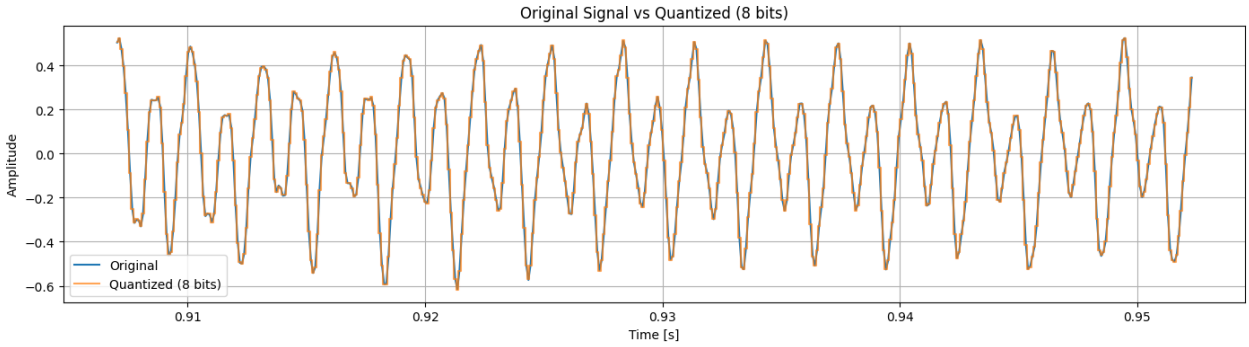

Figure 3: Original vs. Quantized Signal (8 bits)

Observations:

- **2 bits:** The quantized signal approximates the original signal with very coarse steps, resulting in a blocky waveform. The lack of levels causes a poor representation of the signal.
- **4 bits:** The quantization steps are finer compared to the 2-bit case. The quantized signal follows the original more closely but still shows noticeable stepping.

- **8 bits:** The quantized signal closely matches the original waveform. The steps are so fine that they are hardly noticeable, indicating a high-quality quantization.

1.4 Calculate the quantization error (RMS) for 2, 4, and 8 bits. What can you conclude from the results?

We calculate the Root Mean Square (RMS) error using the `rms` function:

```
# Calculation of the RMS error
error_2bit = rms(x_norm, xq_2bit)
error_4bit = rms(x_norm, xq_4bit)
error_8bit = rms(x_norm, xq_8bit)

print(f"RMS␣Error␣for␣2␣bits:␣{error_2bit}")
print(f"RMS␣Error␣for␣4␣bits:␣{error_4bit}")
print(f"RMS␣Error␣for␣8␣bits:␣{error_8bit}")
```

Computed RMS Errors:

- **2 bits:** $\text{Error}_{\text{RMS}}^{(2\,\text{bits})} = 0.1411$
- **4 bits:** $\text{Error}_{\text{RMS}}^{(4\,\text{bits})} = 0.0358$
- **8 bits:** $\text{Error}_{\text{RMS}}^{(8\,\text{bits})} = 0.0022$

Conclusions:

- The RMS error decreases significantly as the number of bits increases.
- Increasing the bit depth reduces quantization error, leading to a more accurate representation of the original signal.
- There is a substantial improvement in error reduction from 2 bits to 4 bits, and from 4 bits to 8 bits, indicating the importance of bit depth in quantization.

1.5 Plot the difference (point by point) between the original and re-quantized signals for 2, 4, and 8 bits (single plot). What can you conclude from the above graph?

We compute the differences and plot them:

```
# Calculation of the differences
diff_2bit = x_norm - xq_2bit
diff_4bit = x_norm - xq_4bit
diff_8bit = x_norm - xq_8bit

# Plotting the quantization errors
plt.figure(figsize=(20, 6))
plt.plot(t[start_sample:end_sample], diff_2bit[start_sample:end_sample], label='Error␣(2␣
    bits)')
plt.plot(t[start_sample:end_sample], diff_4bit[start_sample:end_sample], label='Error␣(4␣
    bits)')
plt.plot(t[start_sample:end_sample], diff_8bit[start_sample:end_sample], label='Error␣(8␣
    bits)')
plt.title('Quantization␣Error␣for␣2,␣4,␣and␣8␣bits')
plt.xlabel('Time␣[s]')
plt.ylabel('Amplitude␣Difference')
plt.legend()
plt.grid()
plt.show()
```

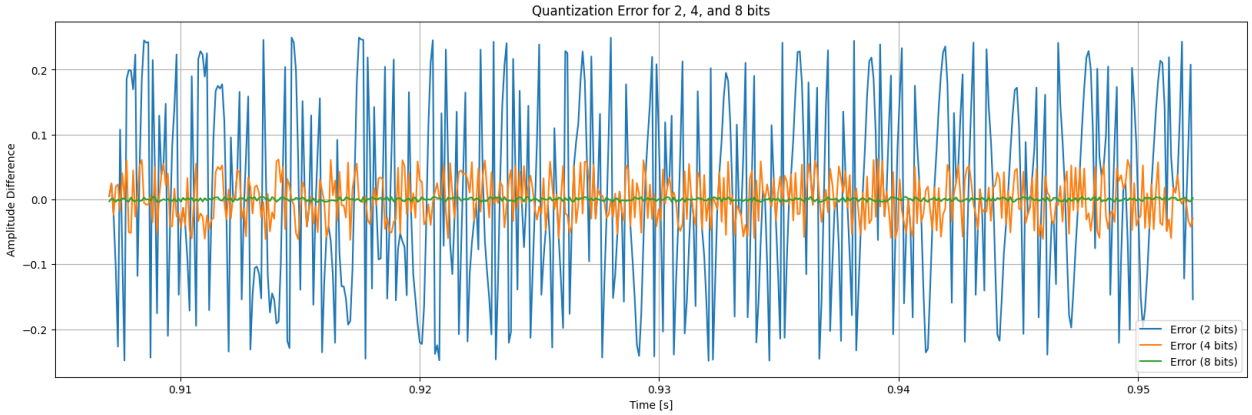

Figure 4: Quantization Error for 2, 4, and 8 bits

Conclusions:

- **Error Magnitude:** The magnitude of the quantization error decreases with increasing bit depth.
- **2 bits:** Shows large error values, indicating significant distortion and loss of information.
- **4 bits:** Error values are smaller but still noticeable, reflecting moderate quantization noise.
- **8 bits:** Error values are minimal, indicating that the quantized signal closely approximates the original signal.
- **Pattern of Errors:** The error signals resemble random noise superimposed on the signal, characteristic of quantization noise.

Overall Conclusions:

- **Impact of Bit Depth:** Increasing the number of quantization bits greatly improves the quality of the quantized signal by reducing quantization error.
- **Quantization Noise:** Quantization introduces noise that can be perceived in the audio signal, especially at lower bit depths.
- **Optimal Bit Depth:** There is a trade-off between the number of bits (which affects data size and processing requirements) and the quality of the quantized signal. Selecting an appropriate bit depth depends on the application's requirements for quality and resources.

2 Problem 2: Noise in an Audio Signal

2.1 Briefly explain what the function `add_gaussian_noise` does. Use the definition of SNR_{dB} to explain, step by step mathematically, how the variable `noise_variance` was derived within the function.

Code for `add_gaussian_noise` and `calculate_snr`

```
def add_gaussian_noise(signal, snr_db):

    # Calculate the signal power
    signal_power = np.mean(signal**2)

    # Calculate the desired noise power
    snr_linear = 10**(snr_db / 10)
    noise_power = signal_power / snr_linear
```

```
    # Calculate the noise variance
    noise_variance = noise_power

    # Generate Gaussian noise
    noise = np.sqrt(noise_variance) * np.random.randn(*signal.shape)

    # Add the noise to the signal
    noisy_signal = signal + noise

    return noisy_signal, noise

def calculate_snr(signal, noise):
    """ Calculates the SNR between an input signal "signal" and noise "noise" """
    # Ensure numpy arrays
    signal, noise = np.array(signal), np.array(noise)

    # Calculate the power of the signal and the noise
    signal_power = np.mean(signal**2)
    noise_power = np.mean(noise**2)

    # Calculate the SNR in decibels (dB)
    snr = 10 * np.log10(signal_power / noise_power)
    return snr
```

Explanation of the `add_gaussian_noise` Function:

The function `add_gaussian_noise` adds Gaussian noise to a signal to achieve a specified Signal-to-Noise Ratio (SNR) in decibels (dB). Below is a step-by-step mathematical explanation of how the variable `noise_variance` is derived.

Step 1: Calculate the Signal Power $E[S^2]$

$$\text{signal\_power} = E[S^2] = \frac{1}{N}\sum_{n=1}^{N} s[n]^2$$

This computes the average power of the signal.

Step 2: Convert SNR from dB to Linear Scale

$$\text{snr\_linear} = 10^{(\text{snr\_db}/10)}$$

This step converts the desired SNR from decibels to a linear scale.

Step 3: Calculate the Desired Noise Power $E[N^2]$

Using the definition of SNR in linear terms:

$$\text{SNR}_{\text{linear}} = \frac{E[S^2]}{E[N^2]}$$

Solving for $E[N^2]$:

$$E[N^2] = \frac{E[S^2]}{\text{SNR}_{\text{linear}}}$$

Therefore:

$$\text{noise\_power} = \frac{\text{signal\_power}}{\text{snr\_linear}}$$

Step 4: Calculate the Noise Variance

For zero-mean Gaussian noise, the variance equals the power:

$$\text{noise\_variance} = \text{noise\_power}$$

Step 5: Generate Gaussian Noise

$$\text{noise} = \sqrt{\text{noise\_variance}} \times \text{randn}(N)$$

Where randn(N) generates N samples of standard normal random variables.

Step 6: Add the Noise to the Signal

$$\text{noisy\_signal} = \text{signal} + \text{noise}$$

Thus, the variable `noise_variance` is derived to ensure that the added Gaussian noise results in the desired SNR.

2.2 Contaminate the audio signal with Gaussian noise using the function `noise = np.random.normal(0, var, len(signal))`, where Var $= 0.1, 0.01, 0.001$. Listen to the signals and briefly describe what you perceive.

Code for Adding Gaussian Noise with Specified Variance

```
# Variances to use
variances = [0.1, 0.01, 0.001]

# Contaminate the signal and listen
for var in variances:
    # Generate Gaussian noise
    noise = np.random.normal(0, np.sqrt(var), len(x_norm))

    # Noisy signal
    noisy_signal = x_norm + noise

    # Listen to the signal
    print(f"Signal␣with␣Gaussian␣noise␣of␣variance␣{var}:")
    display(Audio(noisy_signal, rate=fs))
```

Perceptions after Listening:

- **Variance** $= 0.1$**:** The noise is very prominent, significantly distorting the original audio signal. The speech or music becomes difficult to understand due to the high level of noise.
- **Variance** $= 0.01$**:** The noise is noticeable but less intrusive. The original audio can still be heard, but the quality is degraded.
- **Variance** $= 0.001$**:** The noise is minimal, and the audio quality is close to the original. Any noise present is barely perceptible.

2.3 Calculate the SNR_{dB} using the function `calculate_snr` and report the values for the three given variances. Does SNR_{dB} increase, decrease, or stay the same? What can you conclude from the results, emphasizing the meaning of decibels?

Code for Calculating SNR

```
# List to store the calculated SNRs
snr_values = []

# Original signal power
signal_power = np.mean(x_norm**2)

for var in variances:
    # Generate Gaussian noise
    noise = np.random.normal(0, np.sqrt(var), len(x_norm))

    # Noisy signal
    noisy_signal = x_norm + noise

    # Calculate the SNR
    snr = calculate_snr(x_norm, noise)
    snr_values.append(snr)
```

```
print(f"SNR_dB for variance {var}: {snr:.2f} dB")
```

Calculated SNR_{dB} Values:

- **Variance** $= 0.1$**:** $SNR_{dB} \approx -3.09$ dB
- **Variance** $= 0.01$**:** $SNR_{dB} \approx 6.87$ dB
- **Variance** $= 0.001$**:** $SNR_{dB} \approx 16.91$ dB

Conclusions:

- As the variance decreases from 0.1 to 0.001, the SNR_{dB} increases.
- A lower noise variance means less noise power, resulting in a higher SNR.
- The increase in SNR_{dB} is logarithmic due to the decibel scale. Each 10-fold decrease in noise power results in a 10 dB increase in SNR.
- Decibels provide a logarithmic measure of the ratio between signal and noise power, making it easier to represent large variations in power ratios.

2.4 Use the function `add_gaussian_noise` and add Gaussian noise with $SNR_{dB} = 1, 10, 30$. Use the second output of the function, i.e., `noise`, and calculate the variance. What can you conclude from the results?

Code for Adding Noise and Calculating Variance

```
# SNR_dB values to use
snr_db_values = [1, 10, 30]

# List to store the calculated variances
noise_variances = []

for snr_db in snr_db_values:
    # Add Gaussian noise with the specified SNR_dB
    noisy_signal, noise = add_gaussian_noise(x_norm, snr_db)

    # Calculate the variance of the noise
    noise_variance = np.var(noise)
    noise_variances.append(noise_variance)

    print(f"For SNR_dB = {snr_db}:")
    print(f"Noise variance = {noise_variance:.6f}\n")
```

Calculated Noise Variances:

- $SNR_{dB} = 1$**:** Noise variance ≈ 0.039517
- $SNR_{dB} = 10$**:** Noise variance ≈ 0.004969
- $SNR_{dB} = 30$**:** Noise variance ≈ 0.000049

Conclusions:

- As SNR_{dB} increases, the calculated noise variance decreases.
- A higher SNR_{dB} implies a higher signal-to-noise ratio, which requires a lower noise power (variance).
- The relationship between SNR_{dB} and noise variance is logarithmic. Each 10 dB increase in SNR corresponds to a 10-fold decrease in noise variance.
- These results confirm that to achieve a higher SNR, the noise added to the signal must have a lower variance.

2.5 Overall Conclusions

- The functions `add_gaussian_noise` and `calculate_snr` are consistent with the mathematical definitions of signal power, noise power, and SNR.
- Adding Gaussian noise with specified variances affects the perceptual quality of the audio signal, with higher variances leading to more audible noise.
- The SNR_{dB} is inversely related to the noise variance; decreasing noise variance increases SNR_{dB}.
- Understanding the relationship between SNR, noise variance, and their impact on signal quality is crucial in signal processing applications.

3 Problem 3: Noise in an ECG Signal

3.1 Plot the signal without noise and with noise. Also, calculate the SNR in dB of the signal with respect to the noise that contaminates it.

Code for Generating and Plotting the ECG Signals

```
# Parameters
sampling_rate = 200  # Sampling frequency in Hz
capture_length = 10  # Signal duration in seconds

def create_ECG_signal():
    bpm = 60  # Beats per minute
    bps = bpm / 60  # Beats per second

    # Create a wave similar to a heartbeat
    pqrst = signal.wavelets.daub(10)
    samples_rest = 10
    zero_array = np.zeros(samples_rest, dtype=float)
    pqrst_full = np.concatenate([pqrst, zero_array])

    # Concatenate the beats to cover the entire duration
    ecg_template = np.tile(pqrst_full, int(capture_length * bps))

    # Resample the signal to match the desired sampling frequency
    ecg_sampled = signal.resample(ecg_template, int(sampling_rate * capture_length))
    ecg_sampled = ecg_sampled / np.max(np.abs(ecg_sampled))  # Normalize
    ecg_sampled = 1.5 * ecg_sampled + 1  # Adjust amplitude and DC level

    return ecg_sampled

# Generate the ECG signal without noise
ecg_sampled = create_ECG_signal()

# Time vector
time = np.arange(0, capture_length, 1 / sampling_rate)

# Generate random noise and 50 Hz noise
np.random.seed(0)  # For reproducibility
noise = np.random.normal(0, 0.1, len(ecg_sampled))
noise50hz = 0.5 * np.sin(2 * np.pi * 50 * time)

# ECG signal with noise
ecg_noisy_sampled = ecg_sampled + noise + noise50hz

# Plot the signals
plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)
plt.plot(time, ecg_sampled)
plt.title('ECG␣Signal␣without␣Noise')
plt.xlabel('Time␣[s]')
```

```
plt.ylabel('Amplitude')
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(time, ecg_noisy_sampled, color='r')
plt.title('ECG Signal with Noise')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.grid()

plt.tight_layout()
plt.show()
```

Calculation of SNR

```
# Calculate the power of the signal and the noise
signal_power = np.mean(ecg_sampled ** 2)
noise_power = np.mean((noise + noise50hz) ** 2)

# Calculate the SNR in decibels
SNR_dB = 10 * np.log10(signal_power / noise_power)
print(f"The SNR of the signal with respect to the noise is: {SNR_dB:.2f} dB")
```

Results and Discussion

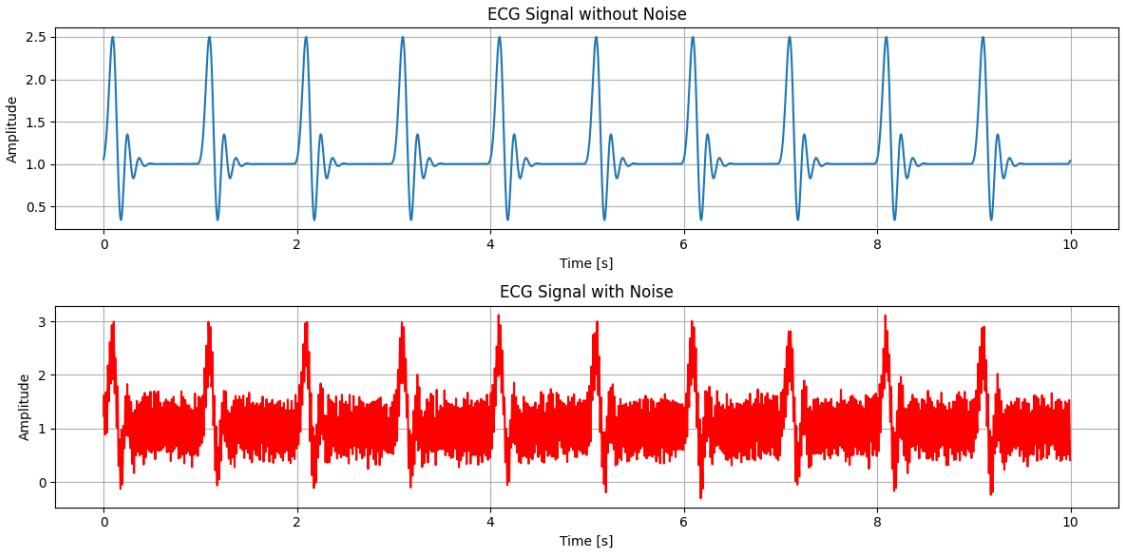

Figure 5: ECG Signal without Noise (Top) and with Noise (Bottom)

Calculated SNR:

$$\text{SNR}_{dB} \approx 10.11 \text{ dB}$$

- The SNR is approximately 10.11 dB, indicating that the signal power is higher than the noise power.
- A positive SNR in decibels means that the signal is stronger than the noise.
- The ECG signal with noise shows visible interference, but the main features of the ECG are still distinguishable.

3.2 Use the FFT and obtain the spectrum of the signal without noise and with noise. Analyze both spectra and describe what you can conclude from the graphs.

Code for Computing and Plotting the Spectra

```
# Compute FFT of the clean signal
yf_clean = fft(ecg_sampled)
xf = fftfreq(len(ecg_sampled), 1 / sampling_rate)

# Compute FFT of the noisy signal
yf_noisy = fft(ecg_noisy_sampled)

# Plot the amplitude spectrum of the clean signal
plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)
plt.plot(xf[:len(xf)//2], 2.0 / len(ecg_sampled) * np.abs(yf_clean[:len(yf_clean)//2]))
plt.title('Spectrum␣of␣the␣Clean␣ECG␣Signal')
plt.xlabel('Frequency␣[Hz]')
plt.ylabel('Amplitude')
plt.grid()

# Plot the amplitude spectrum of the noisy signal
plt.subplot(2, 1, 2)
plt.plot(xf[:len(xf)//2], 2.0 / len(ecg_noisy_sampled) * np.abs(yf_noisy[:len(yf_noisy)//2])
    , color='r')
plt.title('Spectrum␣of␣the␣Noisy␣ECG␣Signal')
plt.xlabel('Frequency␣[Hz]')
plt.ylabel('Amplitude')
plt.grid()

plt.tight_layout()
plt.show()
```

Results and Discussion

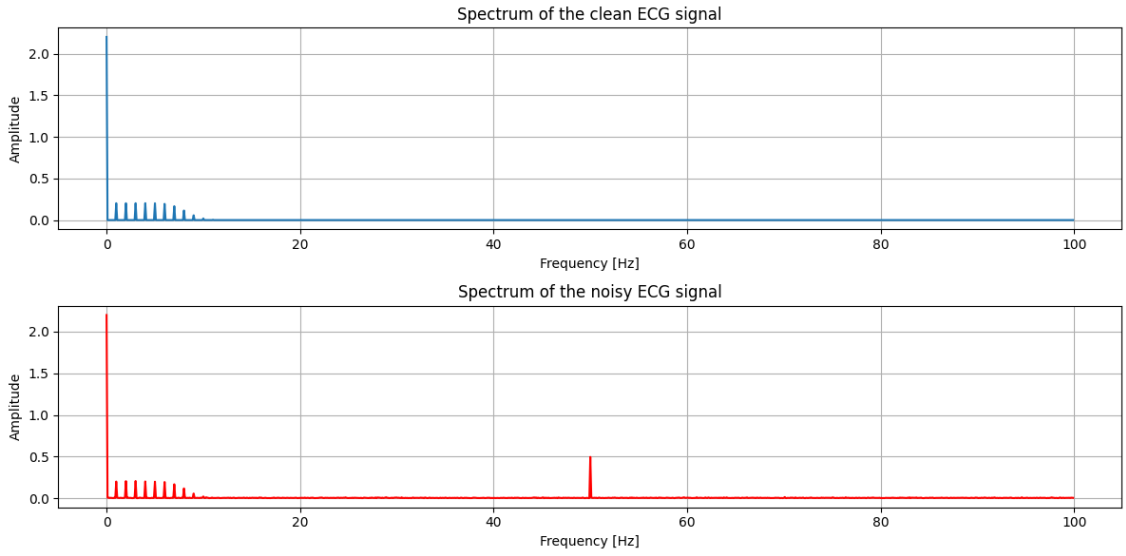

Figure 6: Spectrum of the Clean ECG Signal (Top) and Noisy ECG Signal (Bottom)

Observations:

- In the clean ECG spectrum, the main frequency components correspond to the heart rate and its harmonics.
- In the noisy ECG spectrum, there is a prominent peak at 50 Hz due to the power line interference.
- The noise introduces additional frequency components, increasing the overall noise floor in the spectrum.

Conclusions:

- The 50 Hz interference is clearly visible in the noisy signal's spectrum, which can be targeted for filtering.
- The random noise raises the baseline of the spectrum, indicating increased noise across all frequencies.

3.3 Normalize the signal from -1 to 1. Plot the original and quantized signal for 2, 4, and 8 bits (there are 3 graphs). What do you observe in the graphs?

Code for Normalization and Quantization

```
# Normalize the signal from -1 to 1
ecg_normalized = 2 * (ecg_sampled - np.min(ecg_sampled)) / (np.max(ecg_sampled) - np.min(
    ecg_sampled)) - 1

def cuantizar(s, bit):
    # s: input signal normalized between -1 and 1
    # bit: number of quantization bits
    levels = 2 ** bit
    s_quantized = np.round(((s + 1) / 2) * (levels - 1))  # Scale and quantize
    s_quantized = s_quantized / (levels - 1) * 2 - 1  # Scale back to -1 and 1
    return s_quantized

# Quantization
ecg_quantized_2bit = cuantizar(ecg_normalized, 2)
ecg_quantized_4bit = cuantizar(ecg_normalized, 4)
ecg_quantized_8bit = cuantizar(ecg_normalized, 8)

# Select a segment to plot
start_sample = 950
end_sample = 1150
time_segment = time[start_sample:end_sample]

# Plot for 2 bits
plt.figure(figsize=(10, 4))
plt.plot(time_segment, ecg_normalized[start_sample:end_sample], label='Original', alpha=0.7)
plt.step(time_segment, ecg_quantized_2bit[start_sample:end_sample], label='Quantized␣(2␣bits
    )', where='mid')
plt.title('Original␣Signal␣vs␣Quantized␣(2␣bits)')
plt.xlabel('Time␣[s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()

# Plot for 4 bits
plt.figure(figsize=(10, 4))
plt.plot(time_segment, ecg_normalized[start_sample:end_sample], label='Original', alpha=0.7)
plt.step(time_segment, ecg_quantized_4bit[start_sample:end_sample], label='Quantized␣(4␣bits
    )', where='mid')
plt.title('Original␣Signal␣vs␣Quantized␣(4␣bits)')
plt.xlabel('Time␣[s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()

# Plot for 8 bits
```

```
plt.figure(figsize=(10, 4))
plt.plot(time_segment, ecg_normalized[start_sample:end_sample], label='Original', alpha=0.7)
plt.step(time_segment, ecg_quantized_8bit[start_sample:end_sample], label='Quantized (8 bits
    )', where='mid')
plt.title('Original Signal vs Quantized (8 bits)')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.show()
```

Results and Discussion

Quantization with 2 Bits:

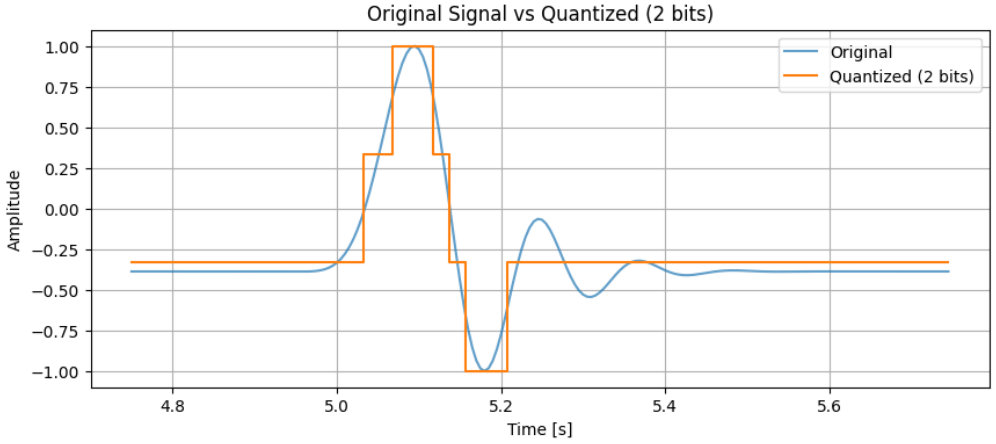

Figure 7: Original vs Quantized ECG Signal (2 bits)

Quantization with 4 Bits:

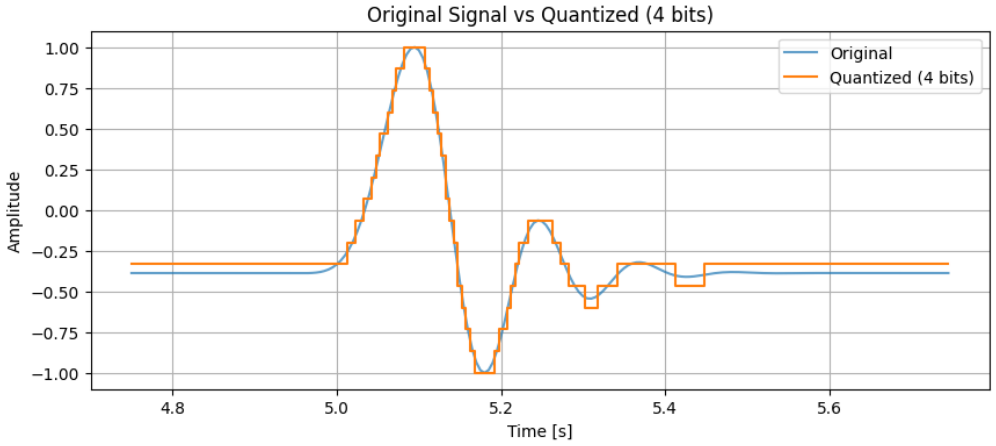

Figure 8: Original vs Quantized ECG Signal (4 bits)

Quantization with 8 Bits:

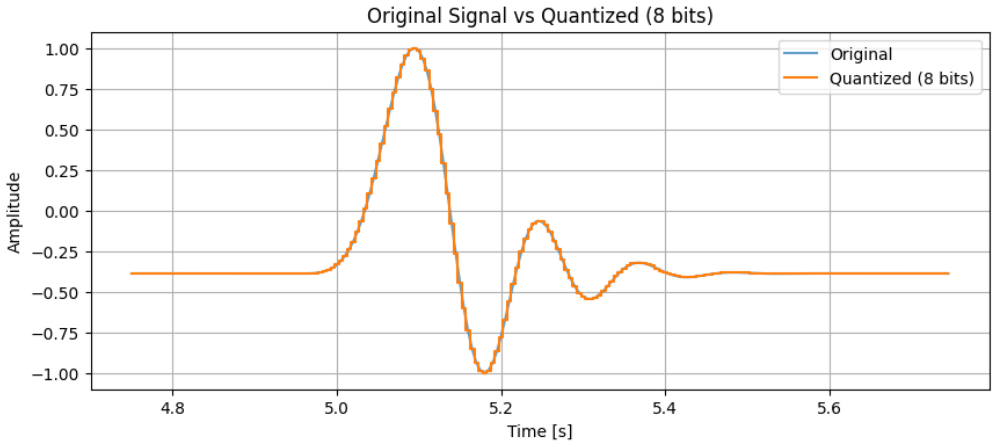

Figure 9: Original vs Quantized ECG Signal (8 bits)

Observations:

- **2 Bits:** The quantized signal appears as a coarse approximation of the original signal with only 4 quantization levels. The waveform is heavily distorted, and fine details are lost.
- **4 Bits:** With 16 quantization levels, the quantized signal follows the original signal more closely. However, there are still noticeable steps in the waveform.
- **8 Bits:** The quantized signal closely matches the original signal. The increased number of levels (256) allows for a much finer representation, and the steps are barely perceptible.

Conclusions:

- Increasing the number of quantization bits improves the fidelity of the quantized signal.
- At lower bit depths, quantization error is significant, leading to distortion of the signal.
- For applications requiring high accuracy, a higher bit depth is necessary to preserve the integrity of the ECG signal.

3.4 Overall Conclusions

- The presence of noise, especially the 50 Hz interference, significantly affects the quality of the ECG signal.
- Frequency analysis using FFT allows us to identify and potentially filter out unwanted frequency components.
- Quantization introduces errors that depend on the number of bits used. Adequate bit depth is essential for accurate signal representation in digital systems.