Chapter 2
Sampling and Quantization of Signals

Learning Objectives

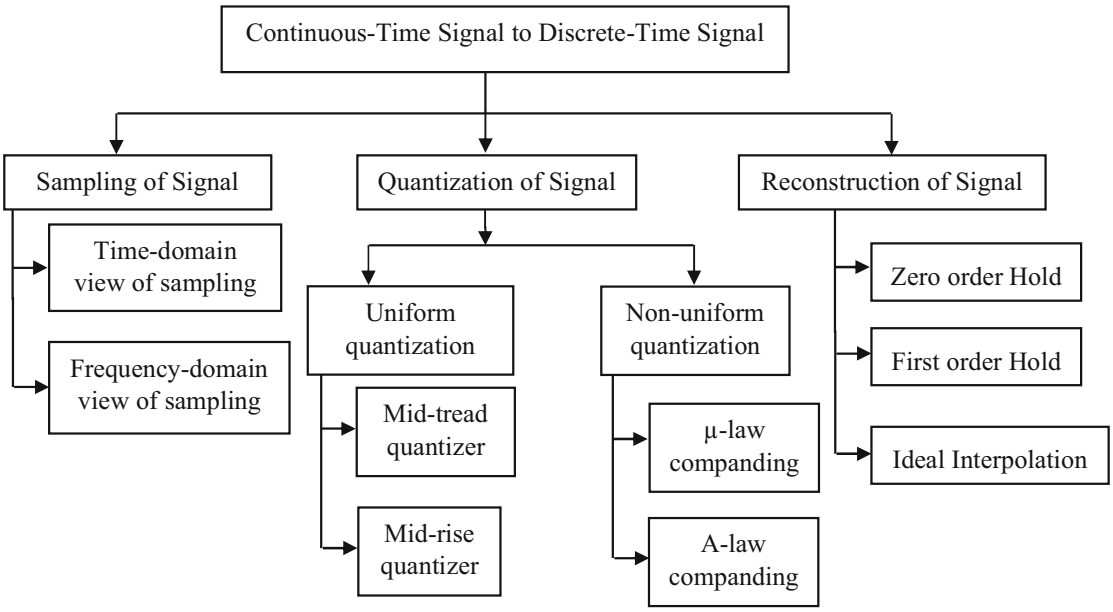
After reading this chapter, the reader is expected to

- Simulate and visualize standard discrete-time signals.
- Simulate and visualize arbitrary discrete-time signals.
- Perform different mathematical operations on discrete-time signals.
- Implement convolution and correlation operations and interpret the obtained results.

Roadmap of the Chapter

The contents discussed in this chapter are given as a flow diagram. The objective is to convert the continuous-time signal into a discrete-time signal. Two important processes in converting the continuous-time signal into a discrete-time signal are (1) sampling and (2) quantization. Also, reconstructing the original signal from the sampled signal is another important task in signal processing. This chapter explores these three processes in detail.

S. Esakkirajan et al., *Digital Signal Processing*,
https://doi.org/10.1007/978-981-99-6752-0_2



PreLab Questions

1. Mention the steps involved in converting the analogue signal into a digital signal.
2. A real-valued signal is known to be bandlimited. The maximum frequency content in the signal is f_{max}. What is the guideline given by the sampling theorem with respect to the choice of sampling frequency such that from the samples, the signal can be reconstructed without aliasing?
3. What is the impact of sampling a bandlimited signal with too low a sampling frequency?
4. Is it possible to reconstruct a periodic square wave of fundamental frequency 5 Hz from its samples? Explain your answer.
5. Mention the reason for aliasing to occur while sampling the signals?
6. What is the meaning of sampling the signal $x(t)$? What is the meaning of the terms (a) sampling rate and (b) sampling interval?
7. A signal has a bandwidth of 5 kHz. What is the Nyquist rate of the signal?
8. Why quantization is considered as a non-linear phenomenon?
9. Why quantization is considered as irreversible phenomenon?
10. What is signal reconstruction? Mention different types of signal reconstruction strategies.

2.1 Sampling of Signal

Sampling is basically taking a specific instant of the signal. In time domain, it is visualized as passing the signal through a switch. Sampling can be considered as multiplying the continuous-time signal $x(t)$ with train of impulse $c(t)$. The train of impulse will take a value of either one or zero; hence, the multiplication of the signal

```
#Generation of comb function
import numpy as np
import matplotlib.pyplot as plt
Fs=100
t = np.arange(0, 2, 1/Fs)
c=np.zeros(len(t))
T = 0.1
c[::int(Fs*T)]=1
plt.stem(t,c),plt.xlabel('Time'),plt.ylabel('Ampliude'),plt.title('c(t)')
```

Fig. 2.1 Python code to generate comb function

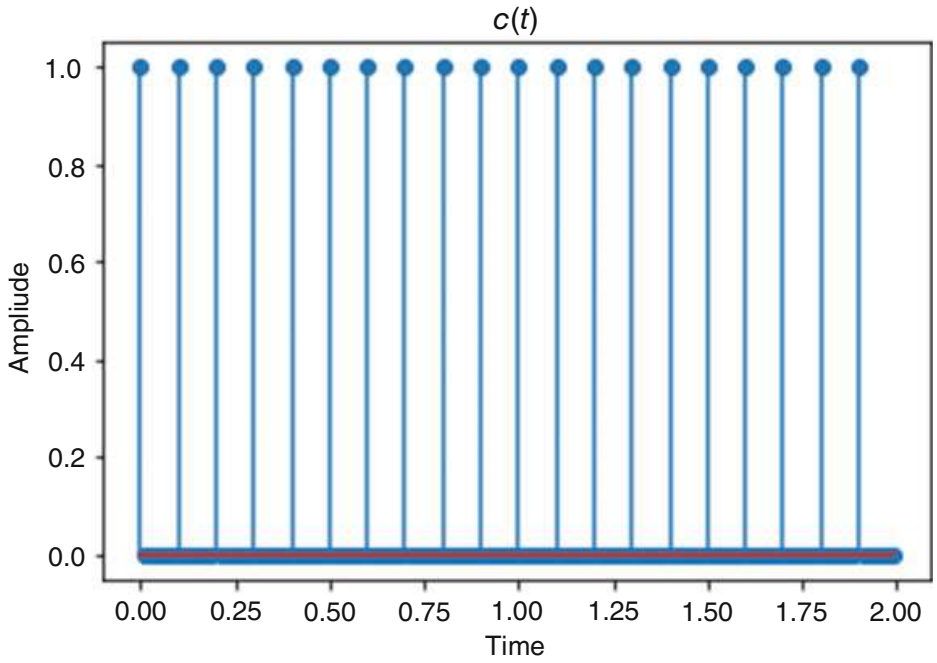


Fig. 2.2 Result of python code shown in Fig. 2.1

$x(t)$ with a train of impulse can be regarded as passing the signal $x(t)$ through a switch. The expression for a train of impulse is given by

$$c(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT) \tag{2.1}$$

The function $c(t)$ takes a value of one whenever $t = nT$; else, it takes a value of zero.

Experiment 2.1 Generation of a Train of Impulse Function
The python code, which generates the train of impulse function or comb function, is given in Fig. 2.1, and the corresponding output is shown in Fig. 2.2.

Inference

1. From Fig. 2.1, it is possible to observe that the variable 'T' (Sampling interval) decides the distance between consecutive samples.
2. From Fig. 2.2, it is possible to confirm that the comb function $c(t)$ takes a value of either '1' or '0'. Whenever $c(t) = 1$, the signal $x(t)$ samples will be collected.

Experiment 2.2 Frequency Domain View of Comb Function

The time-domain expression for the comb function is given by

$$c(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT) \tag{2.2}$$

Upon taking the Fourier transform of the comb function, we get

$$C(\Omega) = \frac{2\pi}{T} \sum_{k=-\infty}^{\infty} \delta(\Omega - k\Omega_s) \tag{2.3}$$

This experiment aims to prove that Fourier transform of a train of impulse will result in a train of impulse function. Here, two comb functions (train of impulse function), namely, $c_1(t)$ and $c_2(t)$ are generated. In the comb function $c_1(t)$, the spacing between consecutive impulses is 0.1 s, whereas in the comb function $c_2(t)$, the spacing between successive impulses is 0.05 s. Upon taking Fourier transform of these two comb functions, the corresponding magnitude spectra $|C_1(f)|$ and $|C_2(f)|$ are obtained. In the magnitude spectrum ($|C_1(f)|$), the spacing between successive peaks is $1/0.1 = 10$, whereas in the magnitude spectrum ($|C_2(f)|$), the spacing between successive peaks is $1/0.05 = 20$. The python code that performs this task is given in Fig. 2.3, and the corresponding output is shown in Fig. 2.4.

Inferences From Fig. 2.4, the following inferences can be drawn:

1. The spacing between two successive samples in the comb function $c_1(t)$ is 0.1 s.
2. The spacing between two consecutive peaks in $C_1(f)$ is 10 Hz.
3. The spacing between two successive samples in the comb function $c_2(t)$ is 0.5 s.
4. The spacing between two consecutive peaks in $C_2(f)$ is 20 Hz.
5. This experiment illustrates the fact that time and frequency are inversely related to each other. That is, compression in one domain is equivalent to expansion in other domain and vice versa.
6. The Fourier transform of a train of impulse function results in a train of impulse function.

Task

1. Write a python code to generate a function expressed as $x[m] = \frac{1}{M} \times \sum_{k=0}^{M-1} e^{j\frac{2\pi}{M}km}$, $-10 < m < 10$ for $M = 1$ and $M = 2$, and comment on the observed result.

```
#Fourier transform of train of impulse
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft,fftshift
#Step 1: Generation of comb functions
Fs=100
t = np.arange(0, 2, 1/Fs)
f = np.linspace(-Fs/2, Fs/2, len(t), endpoint=False)
T1 = 0.1
c1=np.zeros(len(t))
c1[::int(Fs*T1)]=1
T2=0.05
c2=np.zeros(len(t))
c2[::int(Fs*T2)]=1
#Step 2: Fourier transform of comb function
C1=fftshift(fft(c1))
C2=fftshift(fft(c2))
#Step 3: Plotting the result
plt.subplot(2,2,1),plt.stem(t,c1),plt.xlabel('Time'),plt.ylabel('Ampliude'),
plt.title('$c_1(t)$'),plt.subplot(2,2,2),plt.plot(f, np.abs(C1)/len(C1))
plt.xlabel('Frequency'),plt.ylabel('Magnitude'),plt.title('$|C_1(f)|$')
plt.subplot(2,2,3),plt.stem(t,c2),plt.xlabel('Time'),plt.ylabel('Ampliude'),
plt.title('$c_2(t)$'),plt.subplot(2,2,4),plt.plot(f, np.abs(C2)/len(C2))
plt.xlabel('Frequency'),plt.ylabel('Magnitude'),plt.title('$|C_2(f)|$')
plt.tight_layout()
```

Fig. 2.3 Python code to obtain the spectrum of comb function

2.1.1 Violation of Sampling Theorem

The sampling theorem gives the guideline regarding the choice of the sampling rate. According to the sampling theorem, a continuous-time signal with frequencies no higher than $f_{\max}$ (Hz) can be reconstructed exactly from its samples if the samples are taken at a rate greater than $2f_{\max}$. That is, $f_s \geq 2f_{\max}$. Violation of the sampling theorem results in an aliasing, which can be visualized in both the time and frequency domains.

Experiment 2.3 Illustration of Aliasing in Time Domain
In this experiment, the aliasing is visualized in time domain. The analogue signal to be sampled is represented as $x(t) = \sin(2\pi ft + \phi)$. The frequency of the signal $x(t)$ is 10 Hz, and the phase angle is zero. This signal is sampled at four different sampling frequencies 8, 15, 50 and 100 Hz. Obviously, the first two sampling frequencies ($f_s = 8$ and 15 Hz) are less than the criteria specified by the sampling theorem. This will result in aliasing. The impact of aliasing is visualized in this experiment. The

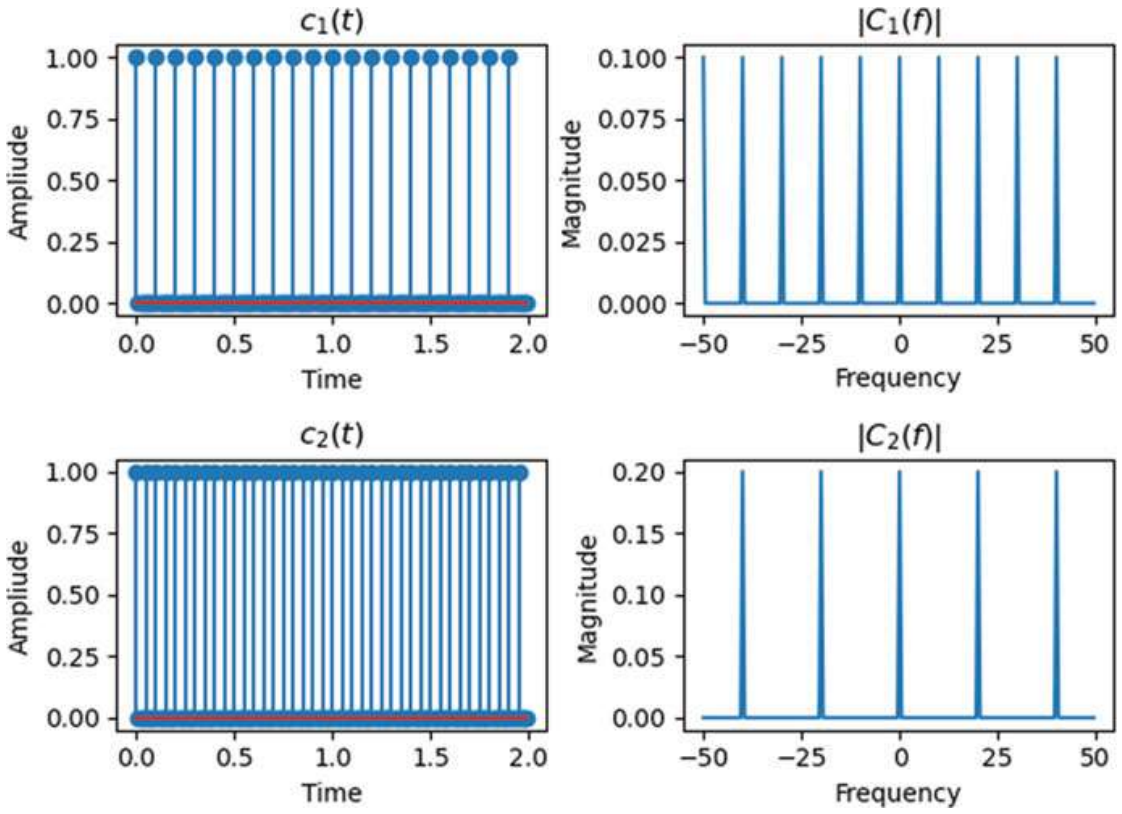


Fig. 2.4 Fourier transform of comb functions

```
#Aliasig in time domain
import numpy as np
import matplotlib.pyplot as plt
f=10   #Signal frequency
fs=[8,15,50,100] #Sampling frequencies
for i in range(len(fs)):
   t=np.arange(0,1,1/fs[i])
   x=np.sin(2*np.pi*f*t)
   plt.subplot(2,2,i+1)
   plt.plot(t,x),plt.xlabel('Time'),plt.ylabel('Amplitude')
   plt.title('$F_s={} $ Hz'.format(fs[i]))
   plt.tight_layout()
```

Fig. 2.5 Python code which illustrates aliasing in time domain

python code that performs this task is shown in Fig. 2.5, and the corresponding output is shown in Fig. 2.6.

Inferences

From Fig. 2.6, the following inferences can be made:

1. The sampling frequency of 8 Hz is insufficient to capture all the information in the signal. The frequency of the sampled signal is given by $f' = f - f_s$. This implies f

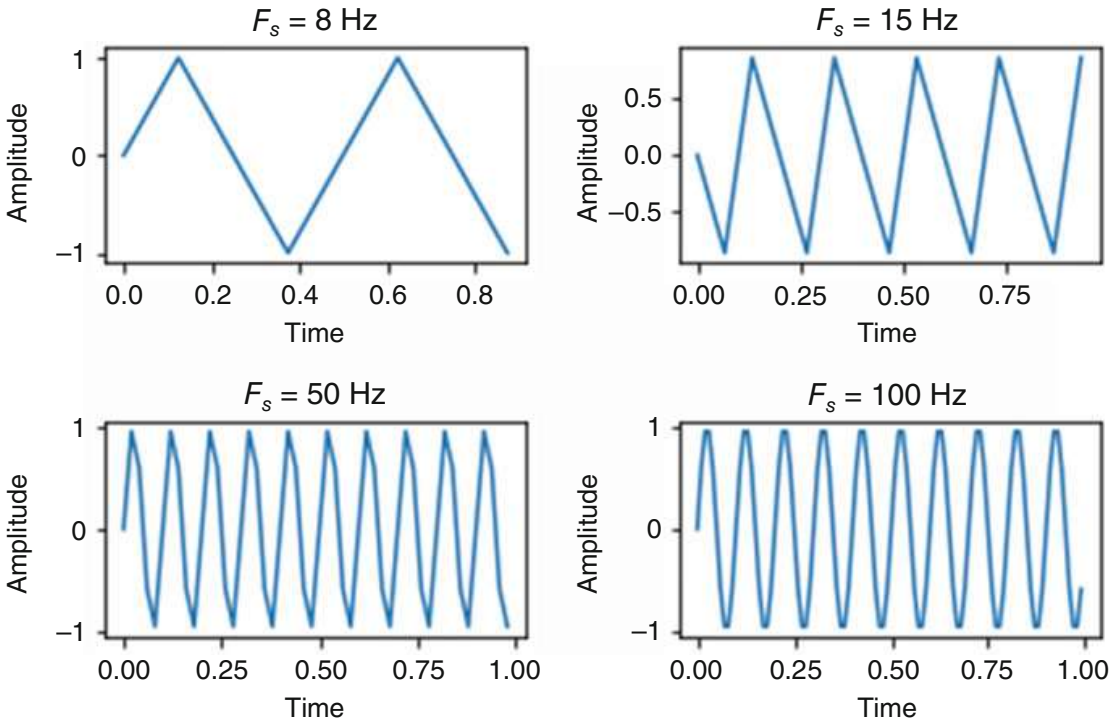


Fig. 2.6 Result of python code shown in Fig. 2.5

$' = 10 - 8 = 2$ Hz. This is the reason that the signal obtained using $f_s = 8$ Hz resembles that of 2 Hz sinusoidal signal.

2. If the sampling frequency is chosen as 15 Hz, then the Nyquist interval is between -7.5 and 7.5 Hz. The signal frequency is not within the Nyquist interval; hence, the frequency of the sampled signal is $f' = f - f_s$. Upon substituting the value, we get $f' = 10 - 15 = -5$ Hz. This is the reason that the signal obtained using $f_s = 15$ Hz resembles that of a 5 Hz sinusoidal signal.
3. For the choice of sampling frequency as 50 and 100 Hz, signal frequency lies well within the Nyquist interval. Hence, no aliasing exists in these cases. As a result, the 10 Hz signal appeared as 10 Hz for $f_s = 50$ and 100 Hz.

Experiment 2.4 Aliasing in the Time Domain

Generate two sinusoidal signals with a frequency of 1 and 6 Hz. Use a sufficiently high sampling frequency to plot the generated signal. Now use the sampling frequency as 5 Hz to plot the 6 Hz frequency component sinusoidal signal, and comment on the observed output. Illustration of this experiment is shown in Fig. 2.7.

The steps involved in the python code implementation of this experiment are as follows:

Step 1: Generation of sine wave of 1 and 6 Hz sinusoidal signals. Let it be represented by the variables '$x1$' and '$x2$'. '$x1$' represents a 1 Hz sine wave, and '$x2$' represents a 6 Hz sine wave. The sampling frequency chosen is 100 Hz ($f_s = 100$ Hz), which is sufficient to represent these two signals without ambiguity.

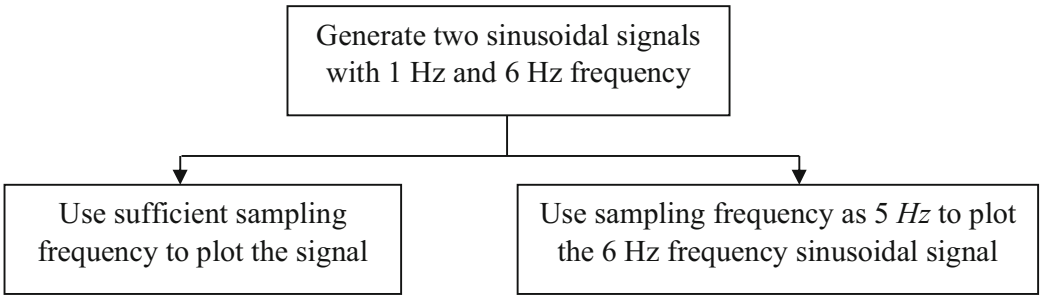


Fig. 2.7 Illustration of Experiment 2.4

```
import numpy as np
import matplotlib.pyplot as plt
#Step 1: To generate x1 and x2
f1=1   #Signal frequency
f2=6
fs=100
t=np.arange(0,1,1/fs)
x1=np.sin(2*np.pi*f1*t)
x2=np.sin(2*np.pi*f2*t)
#Step 2: New sampling frequency is 5 Hz
fs1=5
t1=np.arange(0,1.1,1/fs1)
x3=np.sin(2*np.pi*f1*t1)
#Step 3: Plotting the result
plt.plot(t,x1,'k--',t,x2,'k'),#plt.plot(t,x2,'k')
plt.stem(t1,x3,'r'),plt.xlabel('Time'),plt.ylabel('Amplitude')
plt.legend(['1 Hz Sine wave','6 Hz Sine wave','Sampling with 5 Hz']),
plt.title('Aliasing in Time Domain')
plt.tight_layout()
```

Fig. 2.8 Python code to illustrate aliasing in time domain

Step 2: Now, the new sampling frequency chosen is 5 Hz. That is, $f' = 5$ Hz. This sampling frequency is used to represent a 6 Hz sine wave, which is stored in the variable 'x3'. It is well-known that 5 Hz is insufficient to represent a sine wave of 6 Hz frequency. Because of aliasing, the new frequency will appear at 1 Hz.

Step 3: From the samples taken using $f' = 5$ Hz, it is not possible to distinguish between 1 and 6 Hz sine waves. This phenomenon is termed as 'aliasing'. This occurs due to spectral folding.

The python code used to illustrate this concept is shown in Fig. 2.8, and the corresponding output is shown in Fig. 2.9.

Inferences

From Fig. 2.9, the following inferences can be made:

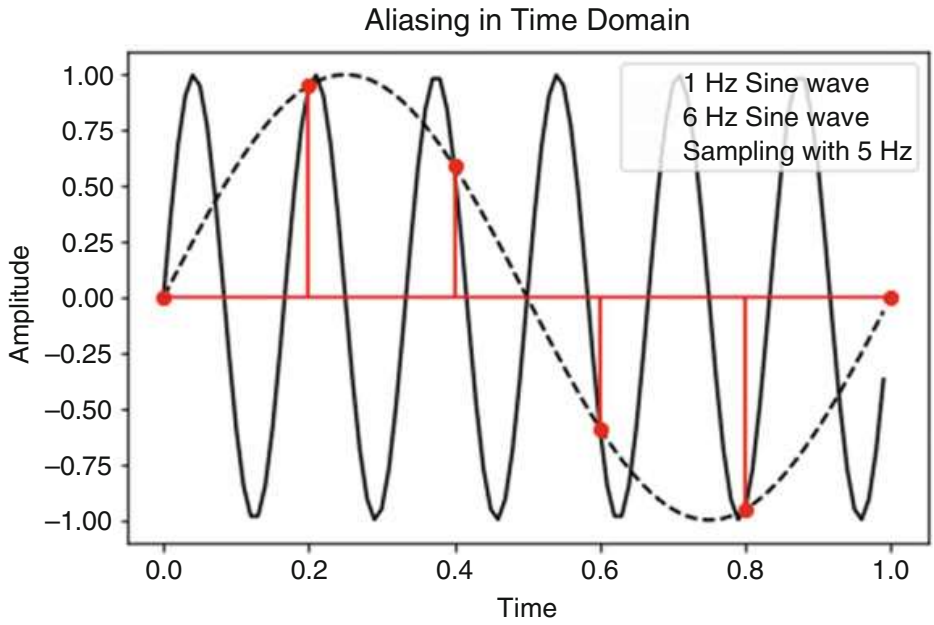


Fig. 2.9 Result of python code shown in Fig. 2.8

1. The solid line shows a sine wave of 6 Hz frequency. The dotted line represents a sine wave of 1 Hz frequency. Since the sampling frequency is 100 Hz, both waveforms appear as desired without ambiguity.
2. The new sampling frequency is chosen as 5 Hz. This sampling frequency is used to represent a 6 Hz sine wave. This sampling frequency is insufficient to represent the 6 Hz. Represent a 6 Hz sine wave; the sampling frequency should be greater than 12 Hz. From the discrete samples, it is not possible to interpret whether the samples are taken from a 6 Hz sine wave or a 1 Hz sine wave. This ambiguity is termed as aliasing, which arises due to spectral folding.

Experiment 2.5 Illustration of Aliasing in Frequency Domain

The python code, which demonstrates the phenomenon of aliasing in the frequency domain, is shown in Fig. 2.10. This experiment generates the signal $x(t) = \sin(10\pi t) + \sin(30\pi t)$ using two different sampling rates: $f_s = 50$ Hz and $f_s = 25$ Hz.

Inferences

The following inferences can be made from Fig. 2.11.

1. The frequency components present in the signal $x(t)$ are $f_1 = 5$ Hz and $f_2 = 15$ Hz.
2. When the sampling rate is 50 Hz, the peak in the magnitude spectrum appears correctly at $f_1 = 5$ Hz and $f_2 = 15$ Hz.
3. On the other hand, if the sampling rate is chosen as $f_s = 25$ Hz, there is no change with respect to $f_1 = 5$ Hz frequency component, whereas the frequency component $f_2 = 15$ Hz appears as $f_2 = 10$ Hz. Observing a 15 Hz frequency component signal as a 10 Hz frequency component is termed as aliasing.

```
#Sampling theorem
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft,fftfreq
#Step 1: Generate the two signals
f1=5
f2=15
fs=[25,50]
N=256
for i in range(len(fs)):
   T=1/fs[i]
   t=np.linspace(0,N*T,N)
   x=np.sin(2*np.pi*f1*t)+np.sin(2*np.pi*f2*t)
   X=fft(x)
   f_axis=fftfreq(N,T)[0:N//2]
   plt.subplot(2,1,i+1)
   plt.plot(f_axis,2/N*np.abs(X[0:N//2]))
   plt.xlabel('$\omega$-->'),plt.ylabel('|X($\omega$)|'),
   plt.title(r'Spectrum corresponding  to $f_s = {} Hz$'.format(fs[i]))
   plt.tight_layout()
```

Fig. 2.10 Python code to illustrate the concept of aliasing in frequency domain

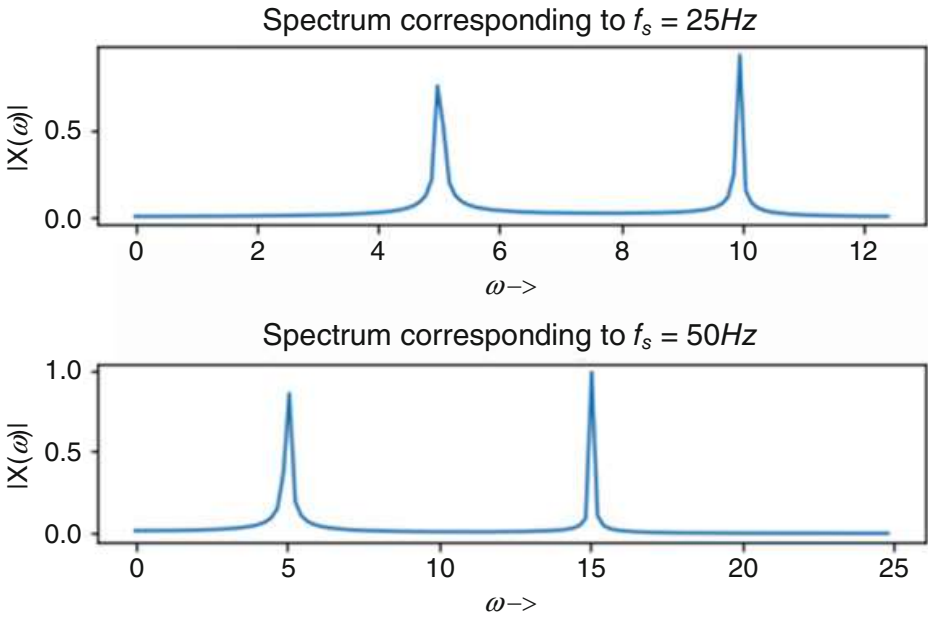


Fig. 2.11 Illustration of aliasing in the frequency domain

Task

1. Change the value of the sampling frequency (f_s) in the python code given in Fig. 2.10, and observe the changes in the output spectrum.

Fig. 2.12 Hearing aliasing effect

```
#Hearing aliasing
import sounddevice as sd
import numpy as np
import matplotlib.pyplot as plt
fs=1500
dur=1
T=1/fs
t=np.linspace(0,1,dur*fs)
x1=np.sin(2*np.pi*500*t)
x2=np.sin(2*np.pi*1000*t)
x=np.concatenate([x1,x2])
sd.play(x,fs)
```

Experiment 2.6 Hearing Aliasing

In this experiment, two sinusoidal tones of frequency $f_1 = 500$ Hz and $f_2 = 1000$ Hz are generated with sampling frequency $f_s = 1500$ Hz. Let $x_1(t)$ and $x_2(t)$ represent the two tones. The maximum signal frequency is 1000 Hz. The minimum sampling rate required is $f_s = 2000$ Hz. Unfortunately, f_s is chosen as 1500 Hz. As a result, 1500 Hz will be heard as 500 Hz. The python code, which illustrates this concept, is given in Fig. 2.12.

Inference

As per the code shown in Fig. 2.12, two sinusoidal tones of frequencies 500 and 1000 Hz are generated. These two tones are appended. Instead of hearing two notes, only one note corresponding to the frequency 500 Hz is heard. This is due to the violation of the sampling theorem. Due to improper sampling, tone of 1000 Hz is heard as a tone of 500 Hz. To overcome the impact of aliasing, the sampling frequency has to be chosen properly.

Task

1. Modify the sampling frequency as 8000 Hz and observe its impact.

2.1.2 Quantization of Signal

Quantization is mapping a large set of values to a smaller set of values. It can be broadly classified into (1) uniform and (2) non-uniform quantization. A uniform quantizer splits the mapped input signal into quantization steps of equal size. The uniform scalar quantization can be broadly classified into (1) mid-tread and (2) mid-rise quantizer.

If 'N' bits are used to represent the value of the signal $x[n]$, then there are 2^N distinct values that $x[n]$ can assume. If the $x_{\min}$ and $x_{\max}$ are the minimum and

maximum values taken by the signal $x[n]$, then the dynamic range of the signal is calculated by

$$\text{Dynamic range} = x_{\max} - x_{\min} \tag{2.4}$$

2.1.2.1 Mid-Tread Quantizer

The relationship between the input and output of a mid-tread uniform quantizer is given by

$$y[n] = Q \times \left\lfloor \frac{x[n]}{Q} + \frac{1}{2} \right\rfloor \tag{2.5}$$

In the above equation, $x[n]$ represents the input signal to be quantized and $y[n]$ represents the quantized signal, 'Q' denotes the quantization step size and the symbol $\lfloor\rfloor$ denotes flooring operation. The expression for quantization step size can be computed by

$$Q = \frac{\text{Dynamic range}}{L} \tag{2.6}$$

where 'dynamic range' represents the difference between the maximum and minimum value of the signal and 'L' denotes the number of reconstruction levels.

The expression for the number of reconstruction levels is given by

$$L = 2^b \tag{2.7}$$

In the above expression, 'b' is the number of bits used to represent the signal.

Experiment 2.7 Transfer Characteristics of Mid-Tread Quantizer
The aim of this experiment is to plot the transfer characteristics of mid-tread quantizer for different bit-rate. The bit-rate (b) chosen is $b = 1, 2, 4$ and 8. The python code, which performs this task, is shown in Fig. 2.13, and the corresponding output is shown in Fig. 2.14.

Inferences
The following inferences can be drawn from Figs. 2.13 and 2.14, which are summarized below:

1. From Fig. 2.13, it is possible to observe that the input signal is represented as the variable 'x' and the quantized signal $Q(x)$ is represented as 'y'. The input signal 'x' varies from −20 to +20; hence, the dynamic range of 'x' is 40.
2. Figure 2.13 shows that the number of bits used to represent the input signal is varied as 1, 2, 4 and 8. It is represented as the variable 'b' in the code.

```
#Transfer characteristics of mid-tread quantizer
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(-20,20)
DR=np.max(x)-np.min(x)  #Dynamic range
b=[1,2,4,8]  #Bits
for i in range(len(b)):
   L=2**b[i]  #Reconstruction level
   q=DR/L
   #Mid-tread quantizer
   y=np.sign(x)*q*np.floor((abs(x)/q)+(1/2))
   plt.subplot(2,2,i+1),plt.plot(x,y),plt.xlabel('x'),plt.ylabel('Q(x)')
   plt.title('Quantizer with b={}' .format(b[i]))
   plt.tight_layout()
```

Fig. 2.13 Python code for transfer characteristics of mid-tread quantizer

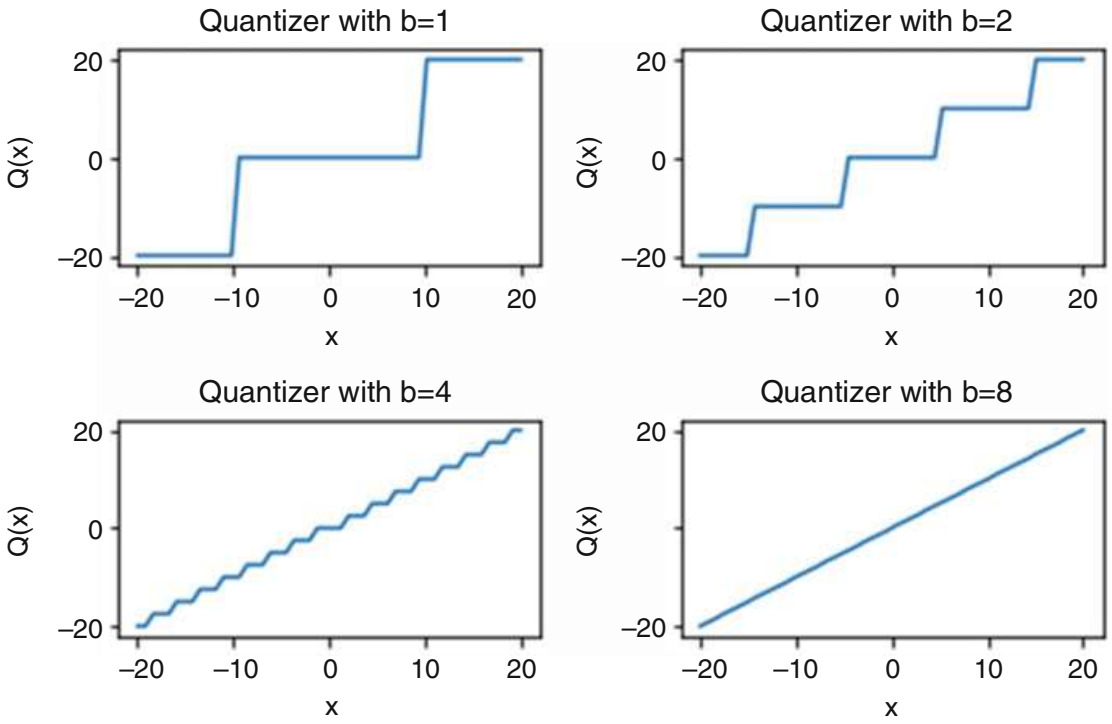


Fig. 2.14 Transfer characteristics of mid-tread quantizer

3. From Fig. 2.14, it is possible to observe that the transfer characteristics of a uniform quantizer is similar to that of a stair-step waveform at low bit rate.
4. At high bit rate ($b = 8$), the relationship between the input signal (x) and the quantized signal ($Q(x)$) is a straight line. This implies that the output follows the input; hence, the error due to quantization will be zero.
5. From Fig. 2.14, it is possible to observe that the number of reconstruction levels depends on the number of bits used to represent the signal.

```
#Uniform Quantization
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
#Step 1: Generate the input signal
t=np.linspace(0,1,100)
x=signal.sawtooth(2*np.pi*5*t)
#Step 2: Parameters of the quantizer
DR=np.max(x)-np.min(x) #Dynamic range
b=[1,2,4,8] #Number of bits
for i in range(len(b)):
   L=2**b[i] #Quantization level
   q=DR/(L)   #Quantization step size
#Step 3: To obtain the quantized signal
   y=np.sign(x)*q*np.floor((abs(x)/q)+(1/2))
   plt.figure(i+1)
   plt.plot(t,x,'b',t,y,'r'),plt.xlabel('Time'),plt.ylabel('Amplitude')
   plt.legend(['Input signal','Quantized Signal'],loc='upper right')
   plt.title('Quantization with b={}'.format(b[i]))
   plt.tight_layout()
```

Fig. 2.15 Python code to perform uniform mid-tread quantization of the signal

6. The stair tread in a ladder is the horizontal walking surface of an individual step. From Fig. 2.14, it is possible to observe that mid-tread quantizer has a zero-valued reconstruction level.

Tasks

1. Write a python code to plot the error signal. The error signal is the difference between the input and quantized signals. Comment on the observed output.
2. Write a python code to illustrate the fact that quantization error follows a uniform distribution.

Experiment 2.8 Quantization of Input Sawtooth Signal Using Mid-Tread Quantizer

The objective of this python experiment is to perform uniform mid-tread quantization of input sawtooth signal of 5 Hz frequency for different bit rate. The number of bits used to represent the input signal varies as 1, 2, 4 and 8. With an increase in the number of bits used to represent the signal, the quantized signal resembles the input signal. The python code to verify this experiment is shown in Fig. 2.15, and its simulation result is displayed in Fig. 2.16.

Inferences

The following are the inferences can be drawn from Fig. 2.16:

1. The input signal to be quantized is a sawtooth signal whose fundamental frequency is 5 Hz.

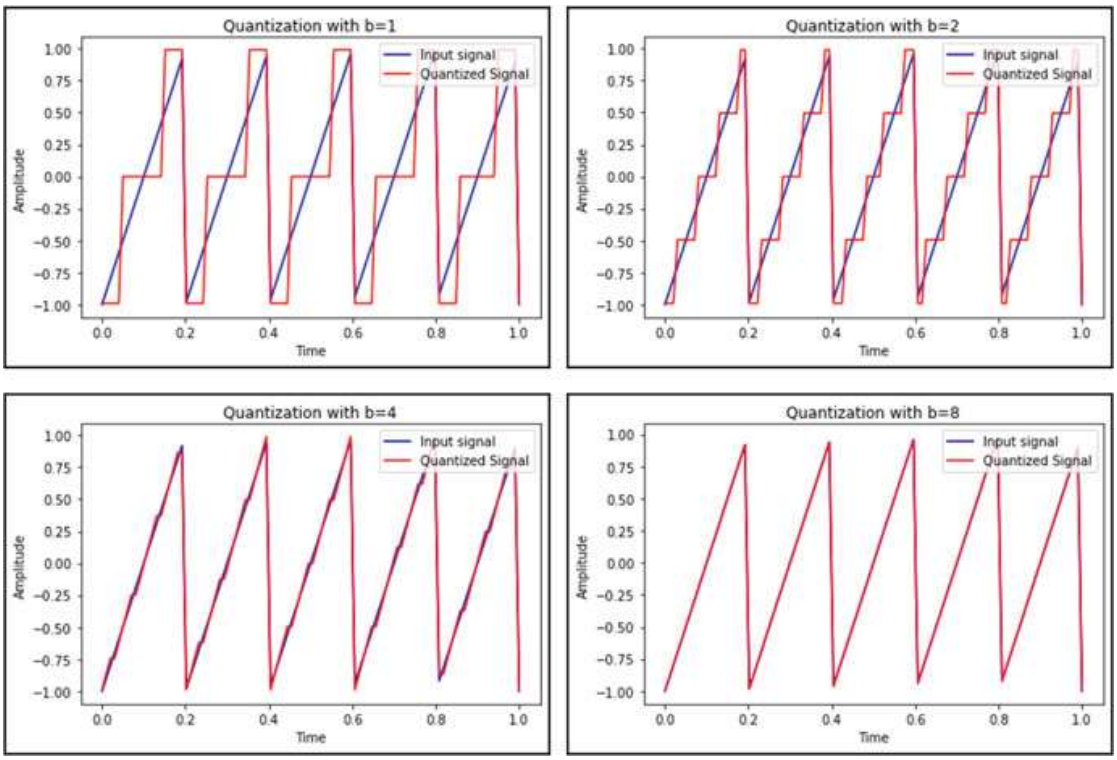


Fig. 2.16 Result of uniform mid-tread quantization

2. The input signal will be uniformly quantized by mid-tread quantizer for different bit rates.
3. It is possible to observe that the quantized signal resembles the input signal with an increase in bit-rate.

2.1.3 Mid-Rise Quantizer

The relationship between the input and output of mid-rise uniform quantizer is given by

$$y[n] = Q \times \left(\left\lfloor \frac{x[n]}{Q} \right\rfloor + \frac{1}{2} \right) \tag{2.8}$$

In the above equation, $x[n]$ represents the input signal to be quantized and $y[n]$ represents the quantized signal, 'Q' denotes the quantization step size and the symbol $\lfloor \rfloor$ denotes flooring operation.

Experiment 2.9 Transfer Characteristics of Mid-Rise Quantizer
The aim of this experiment is to plot the transfer characteristics of mid-rise quantizer for different bit-rate. The bit-rate (b) chosen is $b = 1, 2, 4$ and 8. The python code,

Fig. 2.17 Python code for transfer characteristics of mid-rise quantizer

```
#Transfer characteristics of mid-rise quantizer
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(-20,20)
DR=np.max(x)-np.min(x)  #Dynamic range
b=[1,2,4,8]  #Bits
for i in range(len(b)):
   L=2**b[i]  #Reconstruction level
   q=DR/L
   #Mid-rise quantizer
   y=np.sign(x)*q*(np.floor((abs(x)/q))+(1/2))
   plt.subplot(2,2,i+1)
   plt.plot(x,y),plt.xlabel('x'),plt.ylabel('Q(x)')
   plt.title('Quantizer with b={}' .format(b[i]))
   plt.tight_layout()
```

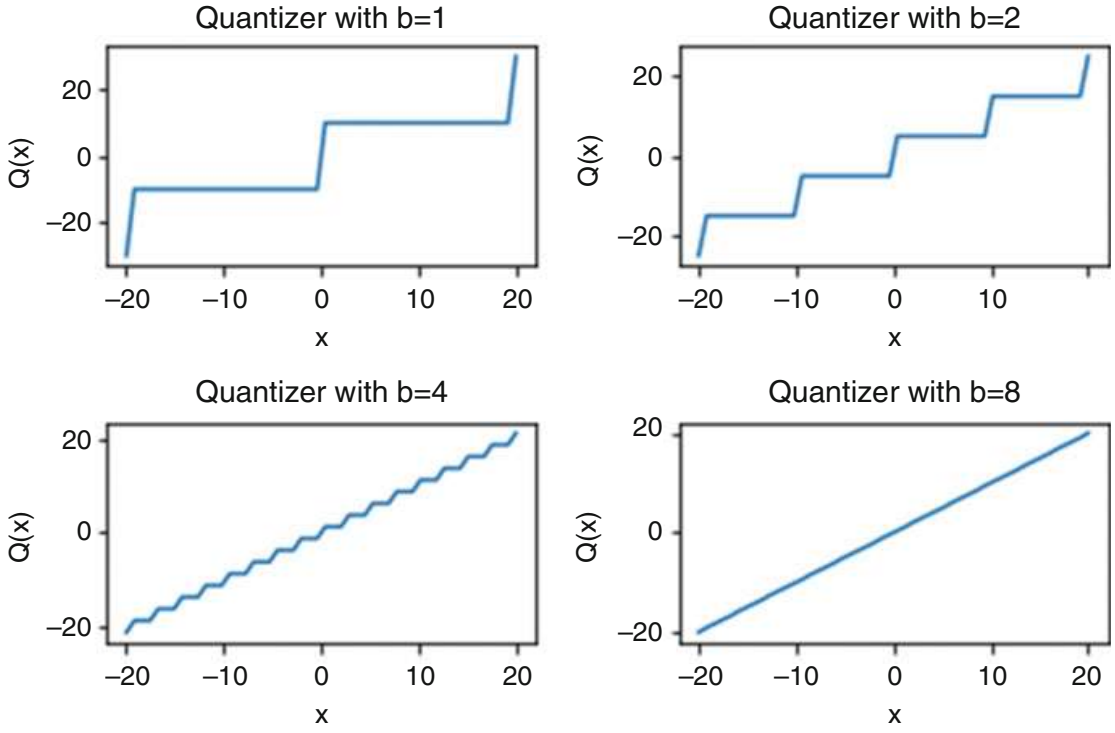


Fig. 2.18 Transfer characteristics of mid-rise quantizer

which performs this task, is shown in Fig. 2.17, and the corresponding output is shown in Fig. 2.18.

Inferences

From Fig. 2.18, it is possible to observe that the reconstruction level rises to the next level at the origin; hence, it is termed as '*mid-rise quantizer*'. It is also possible to observe that with the bit rate increase, the output follows the input. In other words, the quantizer error is minimal with a bit rate increase.

```
#Uniform mid-rise quantizer
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
#Step 1: Generate the input signal
t=np.linspace(0,1,100)
x=signal.sawtooth(2*np.pi*5*t)
#Step 2: Parameters of the quantizer
DR=np.max(x)-np.min(x) #Dynamic range
b=[1,2,4,8] #Number of bits
for i in range(len(b)):
   L=2**b[i] #Quantization level
   q=DR/(L)   #Quantization step size
#Step 3: To obtain the quantized signal
   y=np.sign(x)*q*(np.floor((abs(x)/q))+(1/2))
   plt.figure(i+1)
   plt.plot(t,x,'b',t,y,'r'),plt.xlabel('Time'),plt.ylabel('Amplitude')
   plt.legend(['Input signal','Quantized Signal'],loc='upper right')
   plt.title('Quantization with b={}'.format(b[i]))
   plt.tight_layout()
```

Fig. 2.19 Python code to perform uniform mid-rise quantization

Task

In the python code given in Fig. 2.17, replace '*np.floor*()' by '*np.ceil*()' function, and comment on the change in the transfer characteristics.

Experiment 2.10 Quantization of Input Sawtooth Signal Using Mid-Rise Quantizer

The objective of this experiment is to perform uniform mid-rise quantization of the input sawtooth signal for different bit rate. The python code, which performs this task, is shown in Fig. 2.19, and the corresponding output is shown in Fig. 2.20.

Inference

From Fig. 2.20, it is possible to interpret that with the increase in the number of bits used to represent the signal, the quantized signal resembles the input signal. In other words, the error due to quantization will be minimum with the increase in the number of bits used to represent the signal.

Experiment 2.11 Quantization of Speech Signal

The objective of this experiment is to analyse the performance of uniform mid-tread quantizer for the speech signal. The experiment consists of two steps. Reading the speech signal from a given location is the first step, and performing uniform midtread-quantization of the input speech signal for different bit rates is the second step. The python code, which does this task, is shown in Fig. 2.21, and the corresponding output is shown in Figs. 2.22 and 2.23.

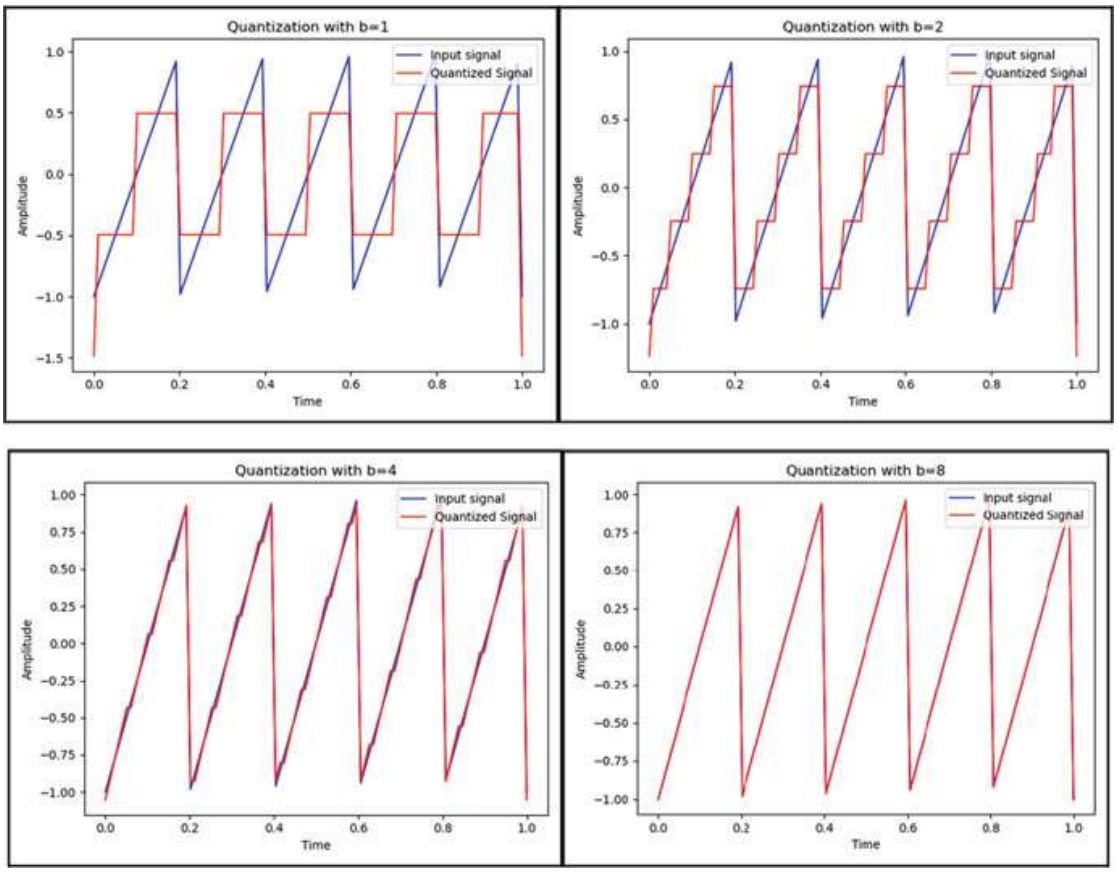


Fig. 2.20 Results of uniform mid-rise quantizer

Inference

The following inference can be made from this experiment:

1. The input speech signal belongs to the uttered word 'Hello'.
2. The quantized signal resembles the original speech signal with the increase in the number of bits of the quantizer.

Experiment 2.12 Uniform Mid-Tread Quantization of Image

In this experiment, a greyscale image, whose intensity varies gradually from black to white, is generated first. This image is subjected to uniform quantization with bit rates 1, 2, 4 and 8. The python code, which performs this task, is shown in Fig. 2.24, and the corresponding output is shown in Fig. 2.25.

Inferences

The following inferences can be drawn from this experiment:

1. The grey level of the input image varies gradually from black to white.
2. The input image is quantized uniformly with a bit rate of $b = 1, 2, 4$ and 8. When $b = 1$, the number of grey levels used to represent the image is minimum. The quantized image is different from the input image.
3. With the increase in the number of bits used to represent the pixel value, the quantized image resembles the input image.

```
#Uniform quantization of speech signal
from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt
#Step 1: Reading of speech waveform
samplerate, x = wavfile.read('C:\\Users\\Admin\\Desktop\\speech1.wav')
duration = x.shape[0] / samplerate
t = np.linspace(0, duration, x.shape[0])
plt.figure(1)
plt.plot(t,x,'k',linewidth=2)
plt.xlabel('Time'),plt.ylabel('Amplitude')
plt.title('Input speech signal')
#Step 2: Performing uniform quantization of the signal
DR=np.max(x)-np.min(x) #Dynamic range
b=[1,2,4,8] #Number of bits
for i in range(len(b)):
   L=2**b[i] #Quantization level
   q=DR/L    #Quantization step size
#Step 3: To obtain the quantized signal
   y=np.floor(x/q)*q-(q/2)
   plt.figure(2)
   plt.subplot(2,2,i+1)
   plt.plot(t,y,'k',linewidth=2),plt.xlabel('Time'),plt.ylabel('Amplitude')
   plt.title('Quantized signal with b={}'.format(b[i]))
   plt.tight_layout()
```

Fig. 2.21 Performing uniform mid-tread quantization of the speech signal

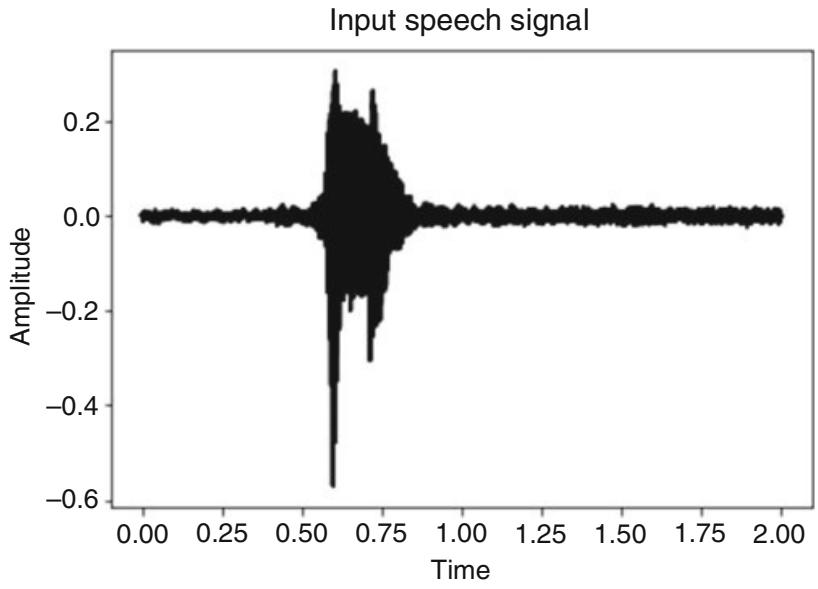


Fig. 2.22 Input speech signal

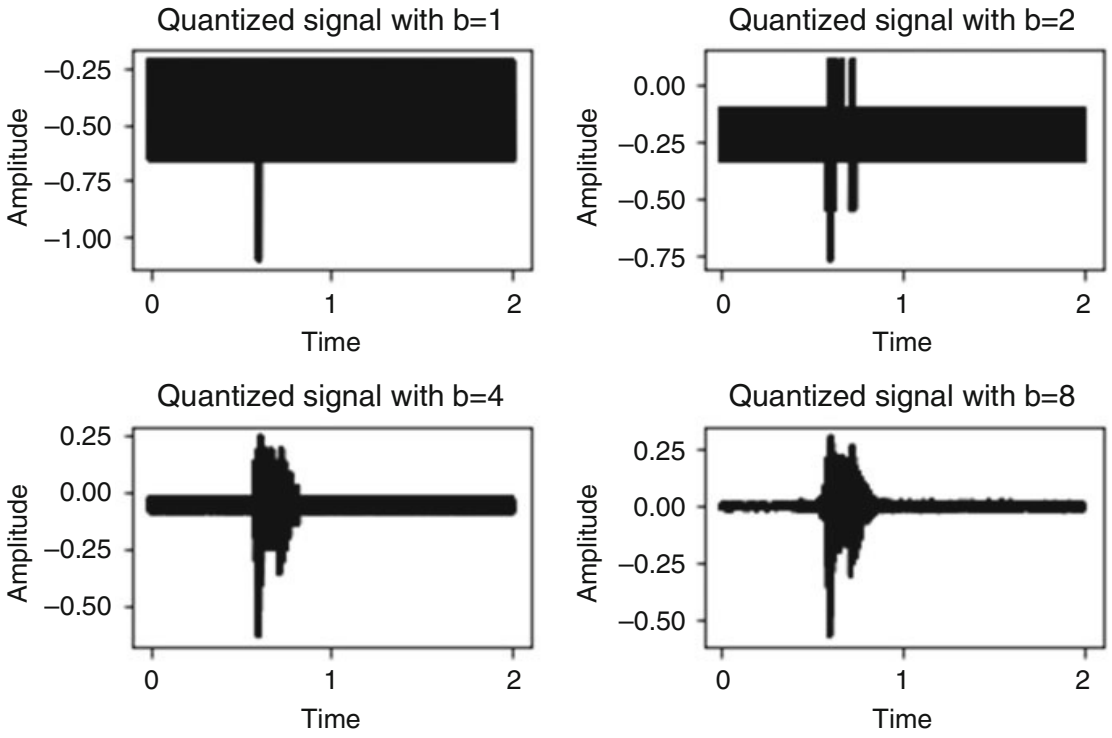


Fig. 2.23 Uniformly quantized speech signal for different bit-rate

```
#Uniform mid-tread quantization of image
import numpy as np
import matplotlib.pyplot as plt
#Step 1: Generation of test image
img=np.zeros([256,256])
img[:,0:256]=np.arange(0,256,1)
plt.figure(1)
plt.imshow(img,cmap='gray')
plt.title('Input image')
#Step 2: Parameters of uniform quantizer
DR=np.max(img)-np.min(img) #Dynamic range
b=[1,2,4,8] #Number of bits
for i in range(len(b)):
   L=2**b[i] #Quantization level
   q=DR/(L)    #Quantization step size
#Step 3: To obtain the quantized signal
   y=np.sign(img)*q*np.floor((abs(img)/q)+(1/2))
   plt.figure(2)
   plt.subplot(2,2,i+1)
   plt.imshow(y,cmap='gray')
   plt.title('b={} '.format(b[i]))
plt.tight_layout()
```

Fig. 2.24 Uniform mid-tread quantization of the image

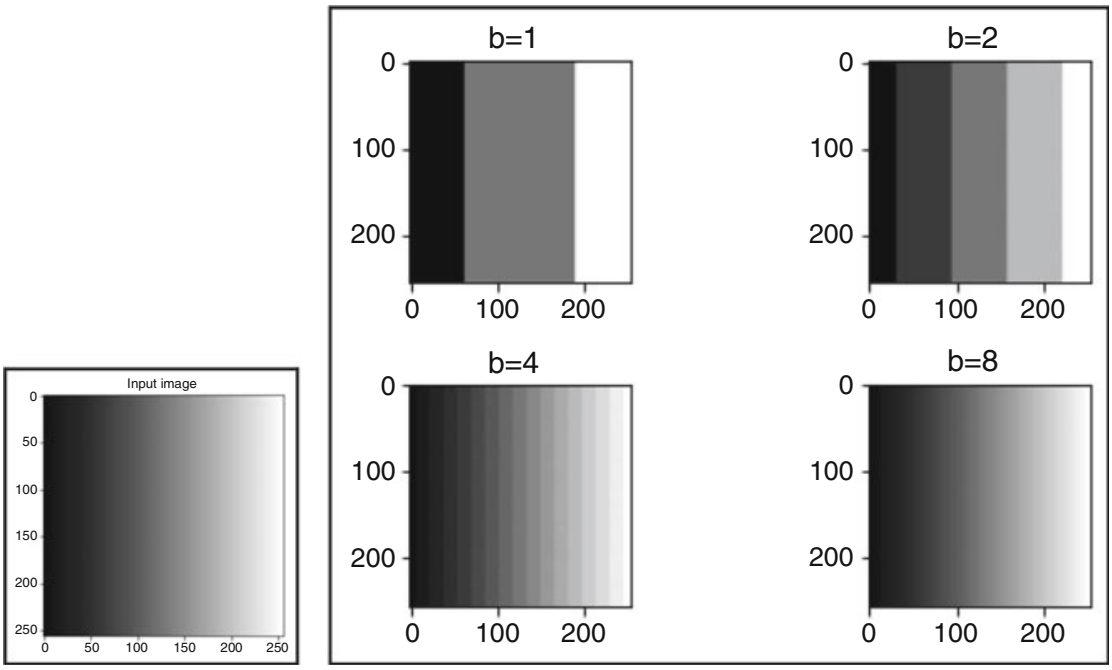


Fig. 2.25 Input and Output of uniform mid-tread quantizer

Task

1. Generate a 256 × 256 image in which half of the pixels are white (grey level 255) and half of the pixels are black (grey level 0). The columns 0 to 127 is white, whereas column 128 to 256 is black. Try to quantize this image for different bit rate and comment on the observed result.

2.2 Non-uniform Quantization

One way to construct non-uniform quantizer is to perform companding.

Companding = **Comp**ression + Exp**anding**

The three steps involved in companding are (1) compression, (2) uniform quantization and (3) expanding. In the first step, the input signal is applied to a logarithmic function, and the output of this function is given to a uniform quantizer. Finally, the inverse of the logarithmic function is applied to the output of the quantizer. There are two standards for non-uniform quantizer companding. They are (1) μ-law companding for North America and (2) A-law companding for Europe.

The μ-law compression expression in terms of the input signal $x(t)$ is expressed as

$$x_1(t) = \text{sgn}(x) \ln \frac{(1 + \mu|x|)}{\ln(1 + \mu)} \tag{2.9}$$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
#Step 1: Generate the input signal
t1=np.linspace(0,1,100)
x=signal.sawtooth(2*np.pi*5*t1)
#Step 2: Mu law Encoding (Non-uniform encoding)
mu=255 # 8 bit Quantization
y1=np.sign(x)*((np.log(1+(mu*abs(x))))/np.log(1+mu))
plt.figure(1)
plt.plot(t1,x,'b',t1,y1,'g'),plt.xlabel('Time'),plt.ylabel('Amplitude')
plt.legend(['Input signal','Encoded'],loc='upper right')
plt.title('Degree of Compression with mu={}'.format(mu))
#Step 3: Parameters of the quantizer
DR=np.max(y1)-np.min(y1) #Dynamic range
b=[1,2,4,8] #Number of bits
for i in range(len(b)):
   L=2**b[i] #Quantization level
   q=DR/(L)    #Quantization step size
#Step 3: To obtain the quantized signal
   y2=np.sign(y1)*q*np.floor((abs(y1)/q)+(1/2))
   y=np.sign(y2)*(((1+mu)**(abs(y2))-1)/mu)
   plt.figure(i+2)
   plt.plot(t1,y2,'r',t1,y),plt.xlabel('Time'),plt.ylabel('Amplitude')
   plt.legend(['Quantized Before decoding','Non-Uniform Quantized'],loc='upper right')
   plt.title('Quantization with b={} and mu={}'.format(b[i],mu))
   plt.tight_layout()
```

Fig. 2.26 Python code for μ-law companding

In the above expression, 'μ' is the compression parameter, which is 255 for the USA and Japan. During compression, the least significant bits of large amplitude values are discarded.

Experiment 2.13 μ-Law Companding

The python code which performs μ-law companding is shown in Fig. 2.26, and the corresponding output is shown in Figs. 2.27 and 2.28.

Inference

The input signal to be companded is a sawtooth signal. The fundamental frequency of a sawtooth signal is 5 Hz. Figure 2.27 illustrates the signal to be encoded using μ-law companding with $\mu = 255$. Here the signal is basically compressed before passing it to the uniform quantizer. Figure 2.28 shows the uniform quantizer results for different bit-rate values. With increase in bit-rate, the quantized signal resembles the input signal.

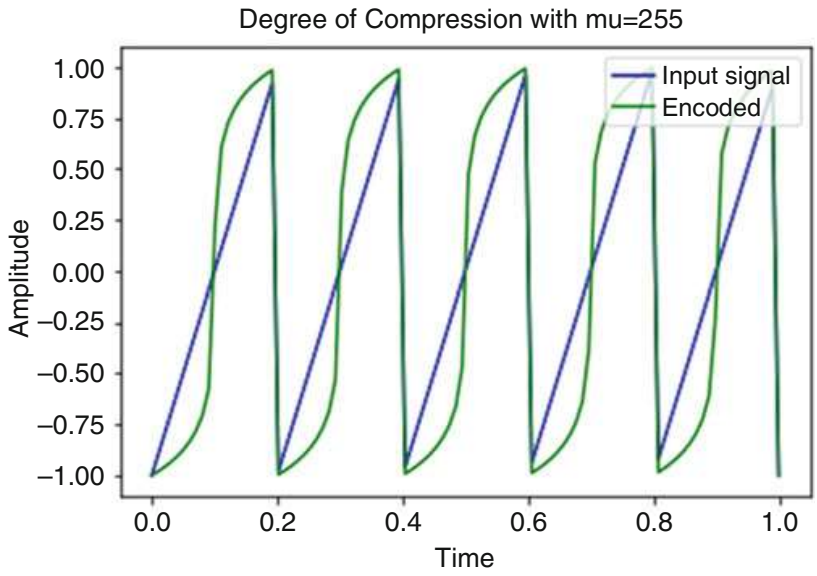


Fig. 2.27 Encoded signal using μ-law companding

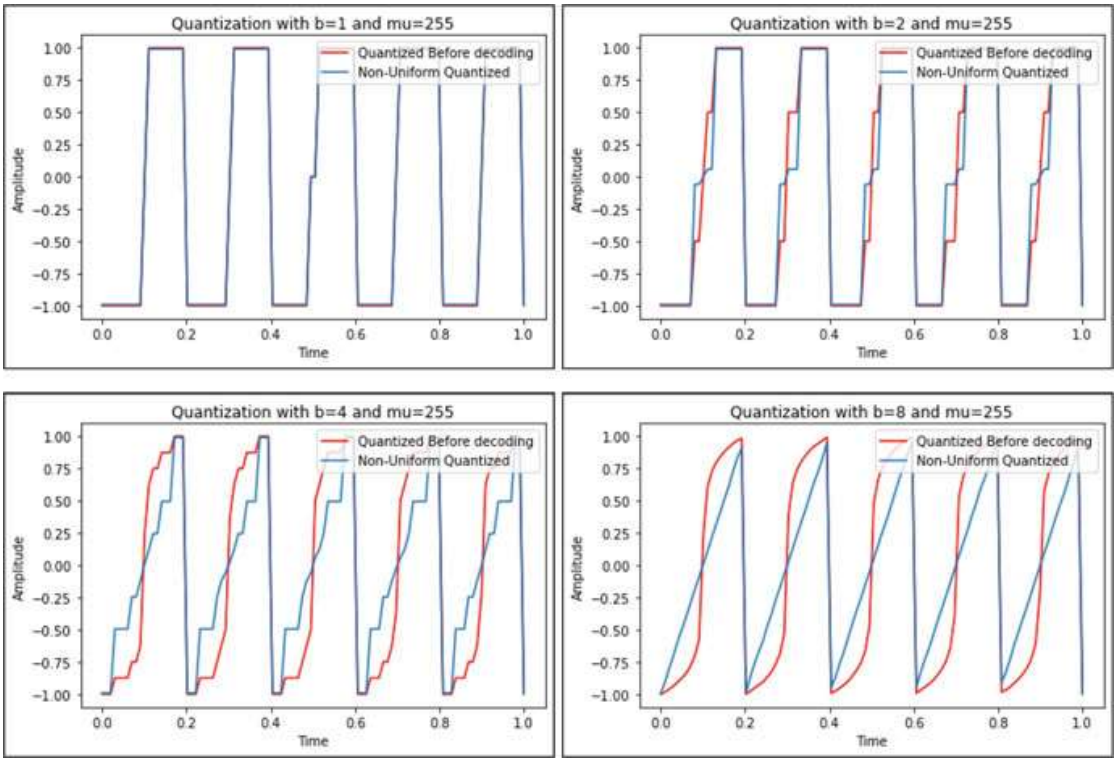


Fig. 2.28 Quantized signal

Experiment 2.14 Error Due to Quantization

Quantization is basically mapping a large set of values to a smaller set of values. It is a non-linear and irreversible process. Quantization leads to loss of information. The loss of information due to quantization can be considered as an error. The error signal is considered as the difference between the quantized signal ($y[n]$) and the input signal ($x[n]$). The objective of this experiment is to quantize the input

```
#Error due to quantization
import numpy as np
import matplotlib.pyplot as plt
#Step 1: Generate the input signal
t=np.linspace(0,1,100)
x=np.sin(2*np.pi*5*t)
#Step 2: Parameters of the quantizer
DR=np.max(x)-np.min(x) #Dynamic range
b=[1,2,4,8] #Number of bits
for i in range(len(b)):
   L=2**b[i] #Quantization level
   q=DR/(L)   #Quantization step size
#Step 3: To obtain the quantized signal
   y=np.sign(x)*q*(np.floor((abs(x)/q))+(1/2))
#Step 4: Obtain the error signal
   e=y-x
#Plot the error signal
   plt.subplot(2,2,i+1), plt.plot(e),plt.xlabel('Time'), plt.ylabel('Amplitude'),
   plt.title('Error signal for b={}'.format(b[i]))
   plt.tight_layout()
```

Fig. 2.29 Error due to quantization

sinusoidal signal of 5 Hz frequency for different bit rate. Then, plot the error signal for different bit-rate. The python code, which performs this task, is shown in Fig. 2.29, and the corresponding output is shown in Fig. 2.30.

Inferences

From Fig. 2.30, the following inferences can be made:

1. The error signal is oscillatory in nature. The magnitude of the error signal varies between positive and negative values.
2. The magnitude of the error signal decreases with increase in bit-rate of the quantizer.
3. Error due to quantization is inevitable; hence, quantization is considered as irreversible phenomenon.

Experiment 2.15 Probability Density Function of Quantization Error

From the previous experiment, it is possible to confirm that error is inevitable in quantization process. The objective of this experiment is to prove that quantization error follows a uniform distribution. The steps followed in this experiment are displayed in Fig. 2.31.

The python code which performs the task mentioned above is shown in Fig. 2.32, and the corresponding output is shown in Fig. 2.33.

Inferences

The following inferences can be drawn from Fig. 2.33:

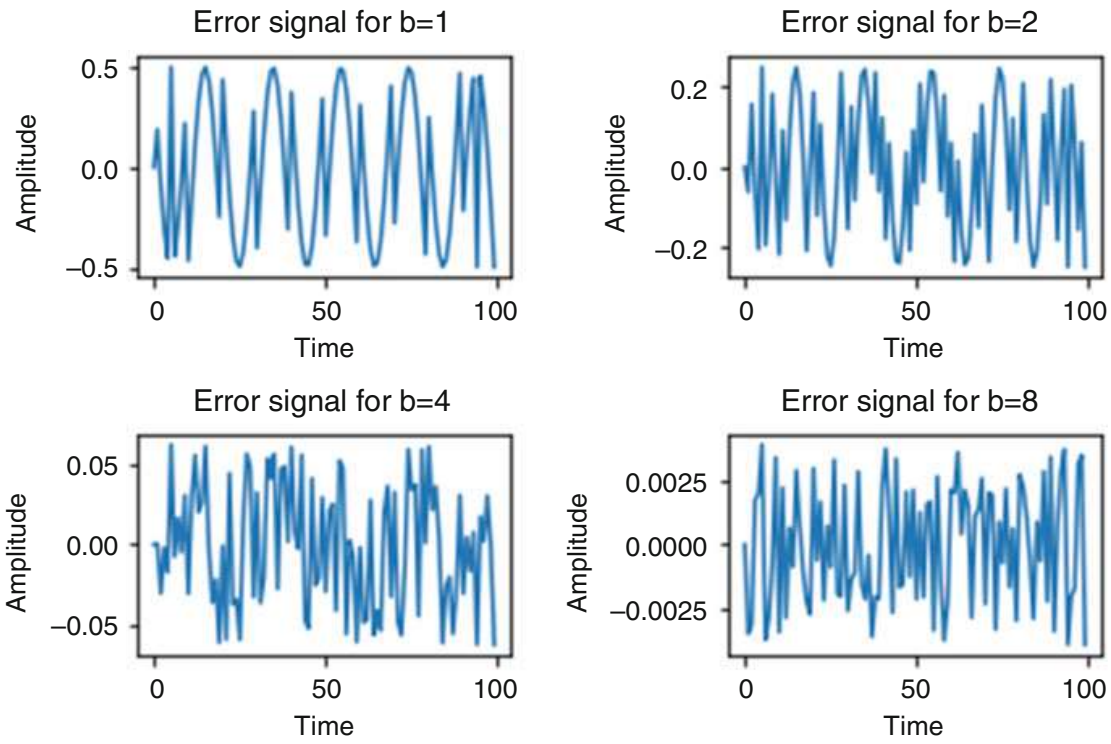
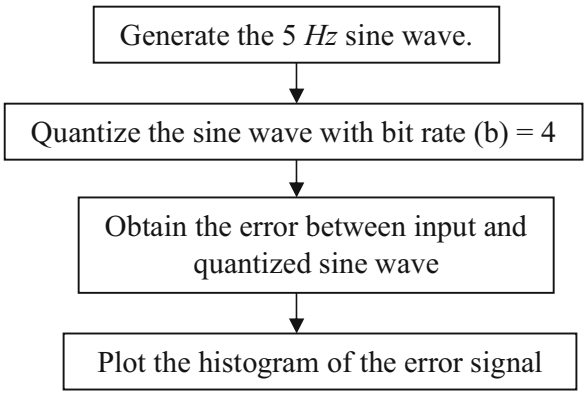


Fig. 2.30 Error signal for different bit-rate of the quantizer

Fig. 2.31 Flow diagram of Experiment 2.13



1. The quantization error follows uniform distribution in the range $(-\Delta/2, \Delta/2)$, where 'Δ' is the quantization step size.
2. In this example, the value of 'Δ' is 0.125; hence, $\Delta/2$ value is 0.0625.

2.3 Signal Reconstruction

Signal reconstruction is an attempt to obtain the continuous-time signal from the samples. This is also termed as interpolation. Different types of interpolation schemes include (1) zero-order hold interpolation, (2) first-order hold or linear interpolation and (3) ideal interpolation.

```
#PDF of quantized error signal
import numpy as np
import matplotlib.pyplot as plt
#Step 1: Generate the input signal
t=np.linspace(0,1,1000)
x=np.sin(2*np.pi*5*t)
#Step 2: Parameters of the quantizer
DR=np.max(x)-np.min(x) #Dynamic range
b=4
L=2**b #Quantization level
q=DR/(L) #Quantization step size
#Step 3: Quantize the input signal
y=np.sign(x)*q*(np.floor((abs(x)/q))+(1/2))
#Step 3: Obtain the error signal
e=y-x
#Step 4: Plot the histogram of the error signal
plt.hist(e,10),plt.xlabel('e'),plt.ylabel('$P_e(e)$')
plt.title('PDF of error signal')
```

Fig. 2.32 Python code for PDF of quantization error

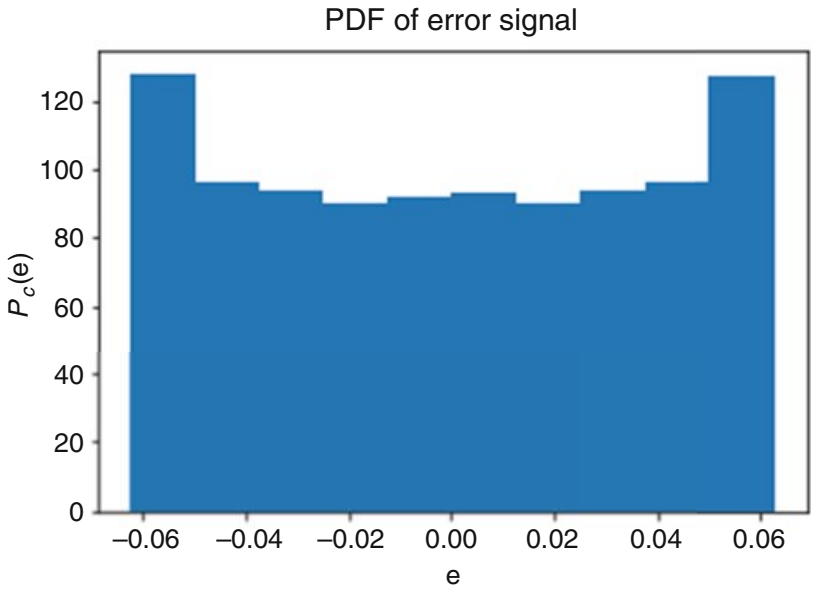


Fig. 2.33 Histogram plot quantization error

2.3.1 Zero-Order Hold Interpolation

A zero-order hold (ZoH) system is a form of simple interpolation, where a line of zero-slope connects discrete samples. The zero-order hold maintains the signal level of the previous pulse until the next pulse arrives. The reconstructed signal will resemble a staircase curve. This is depicted in Fig. 2.34.

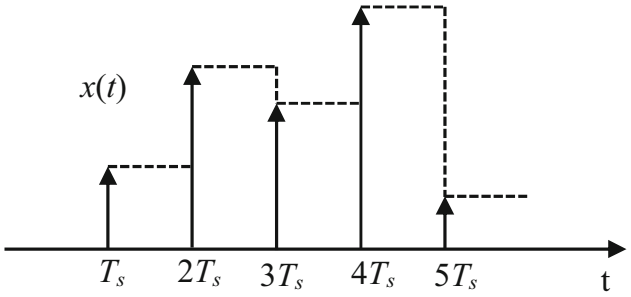


Fig. 2.34 Zero-order hold interpolation

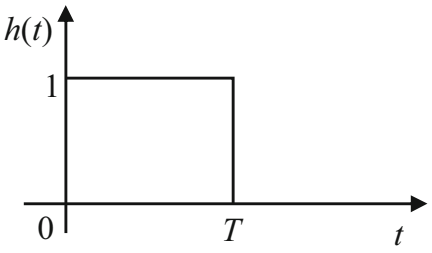


Fig. 2.35 Impulse response of ZoH interpolation function

```
#Zero-order hold interpolation
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
#Step 1: Generation of sine wave
t=np.linspace(0,2*np.pi,10)
x=np.sin(t)
#Step 2: Performing zero-order hold interpolation
f=interp1d (t,x,kind='previous')
#Step 3: Plotting the results
t1=np.linspace(0,2*np.pi,500)
plt.plot(t1,f(t1),'k--'),plt.stem(t,x,'r'),plt.xlabel('Time'),
plt.ylabel('Amplitude'),plt.title('Sine wave')
plt.legend(['ZOH interpolation','Sine wave samples'],loc=1)
```

Fig. 2.36 Python code of zero-order hold interpolation

The impulse response of a zero-order hold is shown in Fig. 2.35.
The transfer function of zero-order hold function is given by

$$H(s) = \frac{1 - e^{-Ts}}{s} \tag{2.10}$$

Experiment 2.16 Zero-Order Hold Interpolation
The python example, which performs zero-order hold interpolation of the sinusoidal signal, is shown in Fig. 2.36, and the corresponding output is shown in Fig. 2.37. In the scipy package, the built-in function 'interp1d' performs the zero-order hold interpolation.

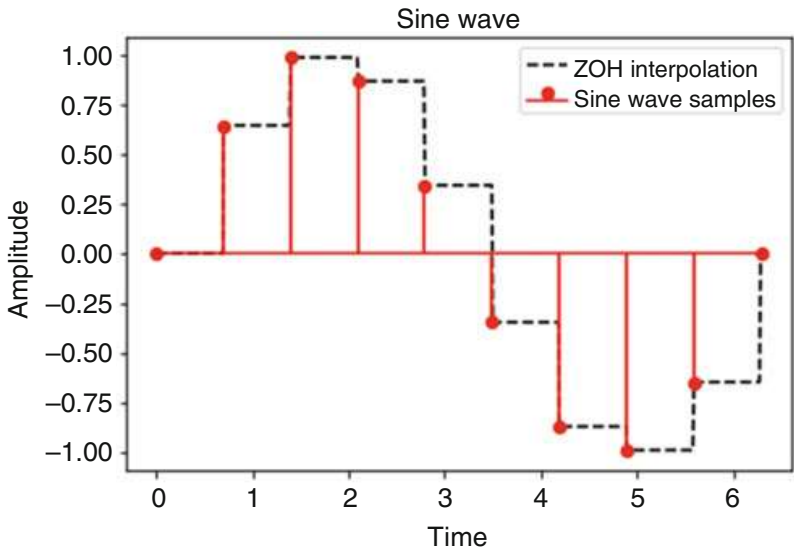
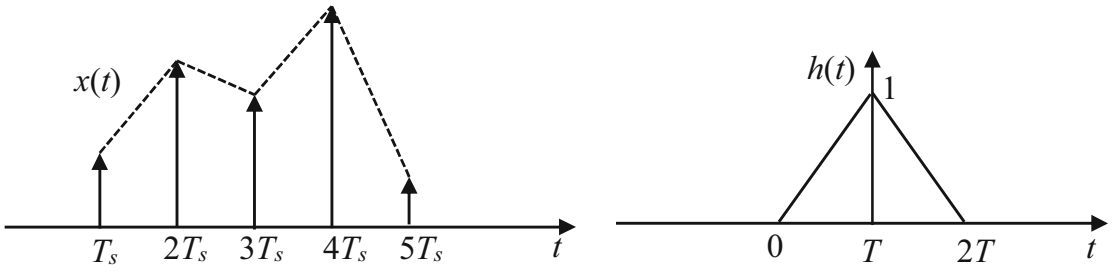


Fig. 2.37 Result of python code shown in Fig. 2.36



(a) First-order hold interpolation **(b) Impulse response**

Fig. 2.38 First-order hold interpolation. (**a**) First-order hold interpolation. (**b**) Impulse response

Inference

From Fig. 2.37, it is possible to interpret that zero-order hold interpolation converts the input signal into a piece-wise constant signal. It is possible to observe discontinuity in the zero-order hold interpolated signal.

2.3.2 First-Order Hold Interpolation

In first-order hold (FoH) interpolation, the signal samples are connected by a straight line. This idea is illustrated in Fig. 2.38a.

The first-order hold performs linear interpolation between samples. The impulse response of first-order hold is shown in Fig. 2.38b.

The transfer function of first-order hold is expressed as

```
#First-order hold interpolation
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
#Step 1: Generation of sine wave
t=np.linspace(0,2*np.pi,10)
x=np.sin(t)
#Step 2: Performing zero-order hold interpolation
f=interp1d(t,x,kind='linear')
#Step 3: Plotting the results
t1=np.linspace(0,2*np.pi,10)
plt.plot(t1,f(t1),'k--'),plt.stem(t,x,'r'),plt.xlabel('Time'),
plt.ylabel('Amplitude'),plt.title('Sine wave')
plt.legend(['FOH interpolation','Sine wave samples'],loc=1)
plt.tight_layout()
```

Fig. 2.39 Python code to perform first-order hold interpolation

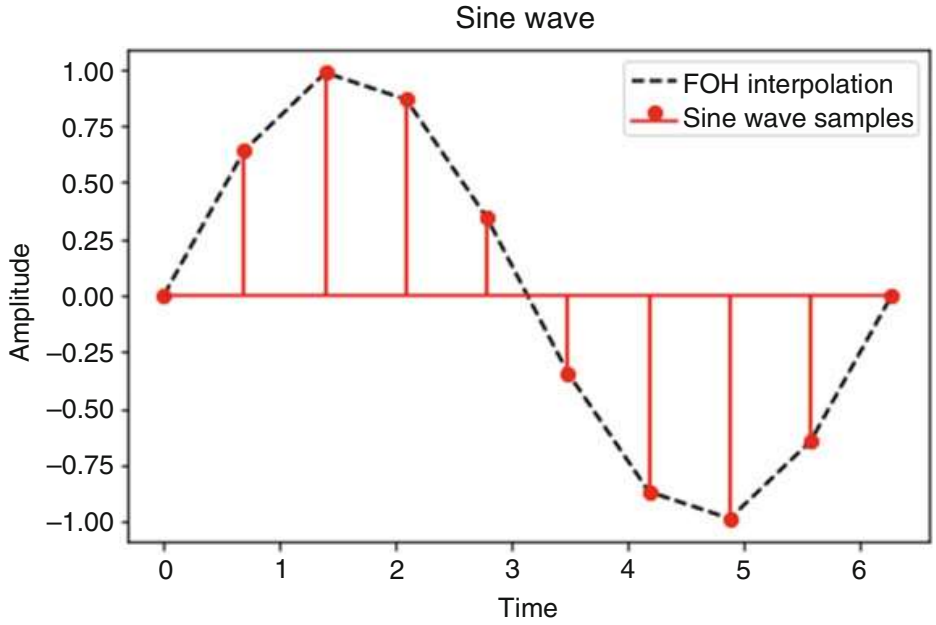


Fig. 2.40 Result of python code shown in Fig. 2.39

$$H(s) = \left(\frac{1 - e^{-sT}}{s}\right)^2 \tag{2.11}$$

Experiment 2.17 First-Order Hold Interpolation

The python code to illustrate first-order hold interpolation is shown in Fig. 2.39, and the corresponding output is shown in Fig. 2.40.

From Fig. 2.40, it is possible to interpret that first-order hold interpolation attempts to connect the sample points through a straight line.

Inferences
The following inference can be drawn from Fig. 2.40:

1. The zero-order hold yields a staircase approximation of the signal.
2. The first-order hold yields a linear approximation of the signal.
3. The first-order hold connects the samples with straight lines.

2.3.3 *Ideal or Sinc Interpolation*

The expression for continuous-time signal obtained using sinc interpolation is expressed as

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \sin c\left(\frac{t - nT_s}{T_s}\right) \tag{2.12}$$

The sinc function is a symmetric function which is square integrable. The decay of the sinc function is slow. The sinc function has infinite support; hence, it is termed as ideal interpolation. The sinc interpolation produces the smoothest possible interpolation of the samples.

Experiment 2.18 Ideal or Sinc Interpolation of a Sinusoidal Signal
The python code, which performs the ideal interpolation of the sine waveform, is shown in Fig. 2.41, and the corresponding output is shown in Fig. 2.42.

Inference
The sinc interpolation produces the smoothest possible interpolation of the samples.

Experiment 2.19 Comparison of Zero-Order Hold and Sinc Interpolation
The python code, which performs the zero-order hold and sinc interpolation of a given sinusoidal signal, is shown in Fig. 2.43, and the corresponding output is in Fig. 2.44.

Inference
By observing Fig. 2.44, it is possible to infer that sinc interpolation smooths the successive samples in the sine wave when compared to zero-order hold interpolation method.

Exercises
1. Write a python code to demonstrate the phenomenon of aliasing in the frequency domain for which the signal $x(t) = \sin(20\pi t) + \sin(50\pi t)$ is generated using two different sampling rates: $f_s = 100$ Hz and $f_s = 25$ Hz. Plot the corresponding spectrum and comment on the observed result.

```
#Ideal or sinc interpolation
import numpy as np
import matplotlib.pyplot as plt
t=np.linspace(0,2*np.pi,10)
t1=np.linspace(0,2*np.pi,100)
x=np.sin(t)
def sinc_interp(x, s, u):
   if len(x) != len(s):
      raise ValueError('x and s must be the same length')
   T = s[1] - s[0]
   sincM = np.tile(u, (len(s), 1)) - np.tile(s[:, np.newaxis], (1, len(u)))
   y = np.dot(x, np.sinc(sincM/T))
   return y
y=sinc_interp(x,t,t1)
plt.plot(t1,y,'r--'),plt.stem(t,x,'k'),plt.xlabel('Time'),
plt.ylabel('Amplitude'),plt.title('Sinc interpolation')
plt.legend(['Ideal interpolation','Sine wave samples'],loc=1)
plt.tight_layout()
```

Fig. 2.41 Python code to perform sinc interpolation

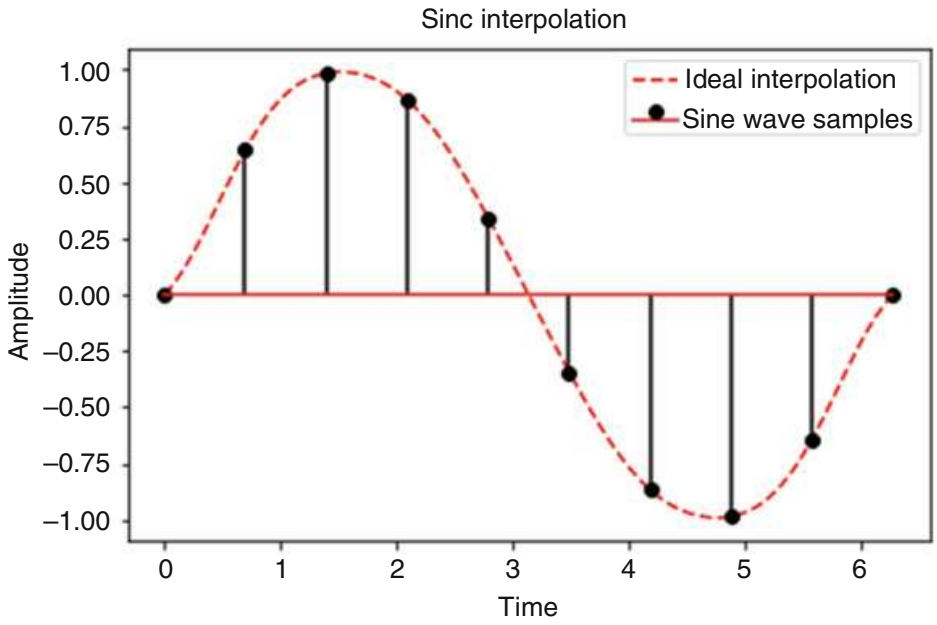


Fig. 2.42 Result of python code shown in Fig. 2.41

2. Generate a sinusoidal signal of 5 Hz frequency. Quantize this signal using uniform mid-rise quantizer with bit-rate, $b = 1$, 2 and 4. Use a subplot to plot the input signal and the quantized signal.
3. Consider an analogue signal $x(t) = \cos(2\pi t) + \cos(14\pi t) + \cos(18\pi t)$, where '$t$' is in seconds. If this signal is sampled at $f_s = 8$ Hz, then it will be aliased with the

```
#Ideal and sinc interpolation
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
t=np.linspace(0,2*np.pi,10)
t1=np.linspace(0,2*np.pi,500)
x=np.sin(t)
#Zero-order hold interpolation
f=interp1d(t,x,kind='previous')
#Sinc interpolation
def sinc_interp(x, s, u):
   if len(x) != len(s):
      raise ValueError('x and s must be the same length')
   T = s[1] - s[0]
   sincM = np.tile(u, (len(s), 1)) - np.tile(s[:, np.newaxis], (1, len(u)))
   y = np.dot(x, np.sinc(sincM/T))
   return y
y=sinc_interp(x,t,t1)
plt.plot(t1,f(t1),'b:'),plt.plot(t1,y,'k--'),plt.stem(t,x,'r'),plt.xlabel('Time'),
plt.ylabel('Amplitude'),plt.title('Comparison of interpolation methods')
plt.legend(['ZOH interpolation','Sinc interpolation','Sine wave samples',],loc=1)
```

Fig. 2.43 Comparison of zero-order hold and sinc interpolation

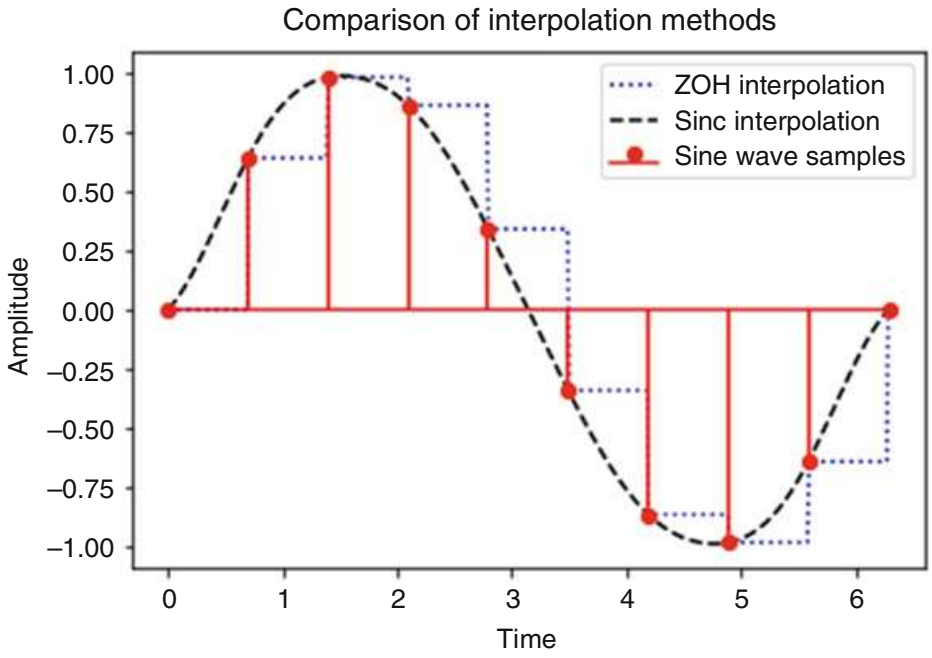


Fig. 2.44 Result of ZOH and sinc interpolation

signal, which is expressed as $x^{'}(t) = 3\cos(2\pi t)$. Plot $x(t)$ and $x'(t)$ on the same graph to verify the signals inter at the sampling instants.

4. Write a python code to generate a sinusoidal signal of fundamental frequency 1300 Hz and sampling frequency $f_s = 8$ kHz. Hear this tone. Now downsample this signal by a factor of 2 and hear the tone. Comment on the heard tones.
5. Write a python code to generate a sinusoidal signal of 10 Hz frequency. Quantize this signal using 4-bit uniform mid-tread quantizer. Use a subplot to plot the input, quantized and error signals. Comment on the observed output.

Objective Questions

1. What will be the output if the following code is executed?

```
import numpy as np
x=4.25
print(np.floor(x))
```

A. 4.5
B. 4.0
C. 4.25
D. 5.0

2. What will be the output if the following code is executed?

```
import numpy as np
x=-4.25
print(np.ceil(x))
```

A. −4.0
B. −5.0
C. −4.25
D. −5.25

3. The following python code segment produces

```
t=np.linspace(0,2*np.pi,20)
x=np.sin(t)
f=interp1d(t,x,kind='nearest')
```

A. Zero-order hold interpolation
B. Linear interpolation
C. Polynomial interpolation
D. Sinc interpolation

4. Fourier transform of train of impulse function results in

A. Train of step function
B. Train of impulse
C. Sinc function
D. Triangular function

5. A sinusoidal signal of the form $x(t) = \sin(2\pi ft)$, where '$f = 5$ Hz' is sampled at the rate $f_s = 100$ Hz to obtain the discrete-time sequence $x[n]$. The expression for the signal $x[n]$ is

$$x[n] = \sin\left(\frac{\pi}{2}n\right)$$

$$x[n] = \sin\left(\frac{\pi}{4}n\right)$$

$$x[n] = \sin\left(\frac{\pi}{5}n\right)$$

$$x[n] = \sin\left(\frac{\pi}{10}n\right)$$

6. Assertion: Quantization is an irreversible process.
   Reason: Quantization is many-to-one mapping:

   A. Assertion and reason are true.
   B. Assertion is wrong; reason is true.
   C. Assertion is true; reason is wrong.
   D. Assertion and reason are wrong.

7. Statement 1: Quantization is a non-linear phenomenon
   Statement 2: Quantization is an irreversible phenomenon

   A. Statement 1 and 2 are false.
   B. Statement 1 and 2 are true.
   C. Statement 1 is true; statement 2 is false.
   D. Statement 1 is false; statement 2 is true.

8. The transfer function of zero-order hold is

$$H(s) = 1$$

$$H(s) = \frac{1}{s}$$

$$H(s) = \frac{1 - e^{-sT}}{s}$$

$$H(s) = 1 - e^{-sT}$$

9. An analogue voltage in the range 0–4 V is divided into 32 equal intervals. The quantization step size of this uniform quantizer is

   A. 0.0625
   B. 0.125
   C. 0.25
   D. 0.5

10. If 'Δ' represents the quantization step size of a uniform quantizer, the expression for mean square quantization error is

    A. $\frac{\Delta^2}{2}$
    B. $\frac{\Delta^2}{4}$
    C. $\frac{\Delta^2}{8}$
    D. $\frac{\Delta^2}{12}$

11. The quantization error follows

    A. Normal distribution
    B. Uniform distribution
    C. Chi-square distribution
    D. Exponential distribution

12. The transfer function of first-order hold is

    A. $H(s) = \frac{1 - e^{-sT}}{s}$
    B. $H(s) = \frac{1}{s}$
    C. $H(s) = \left(\frac{1 - e^{-sT}}{s}\right)^2$
    D. $H(s) = 1 - e^{-sT}$

13. The signal to be quantized takes the value in the range $(-1,1)$. The dynamic range of the signal is

    A. 1
    B. -1
    C. 0
    D. 2

14. If f_s represents the sampling frequency, then the expression for Nyquist frequency is

    A. f_s
    B. $f_s/2$
    C. $f_s/4$
    D. $f_s/8$

15. The quantization step size of a two-bit quantizer which accepts the input signal, which varies from 0 to 2 V, is

    A. 0.125
    B. 0.25
    C. 0.5
    D. 0.75

Bibliography

1. Alan V. Oppenheim, and Ronald W. Schafer, "Discrete-Time Signal Processing", Pearson, 2009.
2. Michael Roberts, and Govind Sharma, "Fundamentals of Signals and Systems", McGraw Hill Education, 2017.
3. John G. Proakis, and Dmitris G. Manolakis, "Digital Signal Processing: Principles, Algorithms and Applications", Pearson Education, 2007.
4. Barrie Jervis, Emmanuel Ifeachor, "Digital Signal Processing: A Practical Approach", Pearson, 2001.
5. Allen B. Downey, "Think DSP: Digital Signal Processing in Python", O' Reilly Media, 2016.