TABLE OF CONTENTS

EXPERIMENT 3 –AMPLITUDE MODULATION AND DEMODULATION

AIM:

To perform amplitude modulation and demodulation for the given message and carrier signals.

SOFTWARE REQUIRED:

1. Oracle VM VirtualBox 6.1.38, Oracle Corporation
2. Ubuntu 22.04 (64-bit) Operating System
3. GNU Radio Companion Application, v3.10.1 (sudo apt-get install gnuradio)

THEORY:

Amplitude modulation or just AM is one of the earliest modulation methods that is used in transmitting information over the radio. This technique was devised in the 20th century at a time when Landell de Moura and Reginald Fessenden were conducting experiments using a radiotelephone in the 1900s.

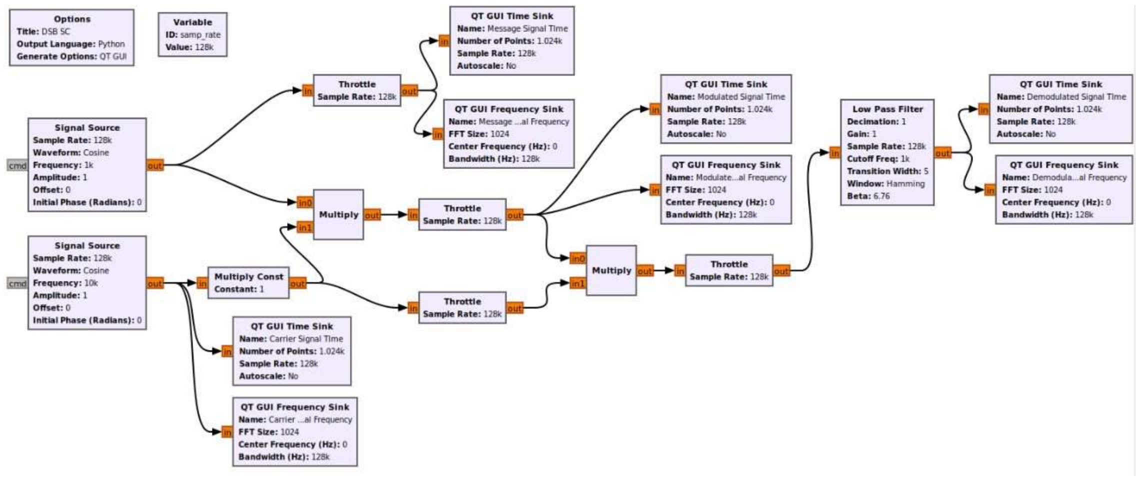
After successful attempts, the modulation technique was established and used in electronic communication. In general, amplitude modulation definition is given as a type of modulation where the amplitude of the carrier wave is varied in some proportion to the modulating data or the signal.

As for the mechanism, when amplitude modulation is used there is a variation in the amplitude of the carrier. Here, the voltage or the power level of the information signal changes the amplitude of the carrier. In AM, the carrier does not vary in amplitude. However, the modulating data is in the form of signal components consisting of frequencies either higher or lower than that of the carrier.

The signal components are known as sidebands and the sideband power is responsible for the variations in the overall amplitude of the signal. The AM technique is different from frequency modulation and phase modulation where the frequency of the carrier signal is varied in the first case and the second one the phase is varied respectively.

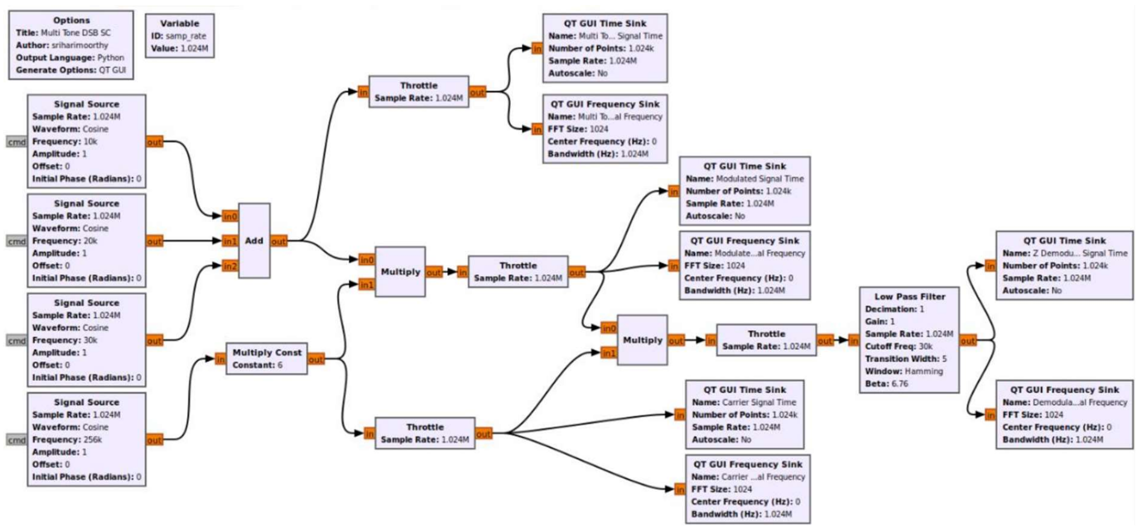
Currently, this technique is used in many areas of communication such as in portable two-way radios; citizens' band radios, VHF aircraft radios and modems for computers. Amplitude modulation is also used to mention mediumwave AM radio broadcasting.

BLOCK DIAGRAM:



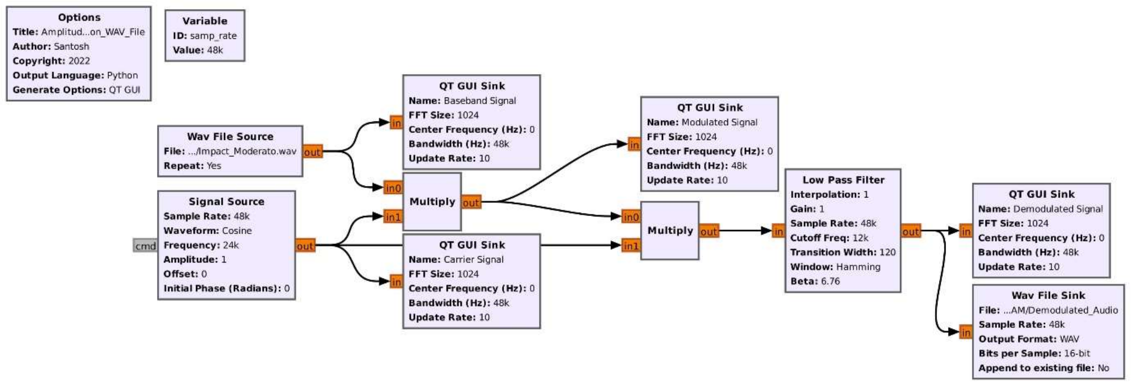
```
Options
Title: DSB SC
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 128k

Signal Source
Sample Rate: 128k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Signal Source
Sample Rate: 128k
Waveform: Cosine
Frequency: 10k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Multiply Const
Constant: 1

Throttle
Sample Rate: 128k

QT GUI Time Sink
Name: Message Signal Time
Number of Points: 1.024k
Sample Rate: 128k
Autoscale: No

QT GUI Frequency Sink
Name: Message ...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 128k

QT GUI Time Sink
Name: Carrier Signal Time
Number of Points: 1.024k
Sample Rate: 128k
Autoscale: No

QT GUI Frequency Sink
Name: Carrier ...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 128k

QT GUI Time Sink
Name: Modulated Signal Time
Number of Points: 1.024k
Sample Rate: 128k
Autoscale: No

QT GUI Frequency Sink
Name: Modulate...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 128k

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 128k
Cutoff Freq: 1k
Transition Width: 5
Window: Hamming
Beta: 6.76

QT GUI Time Sink
Name: Demodulated Signal Time
Number of Points: 1.024k
Sample Rate: 128k
Autoscale: No

QT GUI Frequency Sink
Name: Demodula...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 128k
```

Figure 1 – Singletone Double-Sideband Suppressed-Carrier (DSB-SC) Amplitude Modulation



```
Options
Title: Multi Tone DSB SC
Author: sriharimoorthy
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 1.024M

Signal Source
Sample Rate: 1.024M
Waveform: Cosine
Frequency: 10k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Signal Source
Sample Rate: 1.024M
Waveform: Cosine
Frequency: 20k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Signal Source
Sample Rate: 1.024M
Waveform: Cosine
Frequency: 30k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Signal Source
Sample Rate: 1.024M
Waveform: Cosine
Frequency: 256k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Multiply Const
Constant: 6

Throttle
Sample Rate: 1.024M

QT GUI Time Sink
Name: Multi To... Signal Time
Number of Points: 1.024k
Sample Rate: 1.024M
Autoscale: No

QT GUI Frequency Sink
Name: Multi To...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 1.024M

QT GUI Time Sink
Name: Modulated Signal Time
Number of Points: 1.024k
Sample Rate: 1.024M
Autoscale: No

QT GUI Frequency Sink
Name: Modulate...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 1.024M

QT GUI Time Sink
Name: Carrier Signal Time
Number of Points: 1.024k
Sample Rate: 1.024M
Autoscale: No

QT GUI Frequency Sink
Name: Carrier ...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 1.024M

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 1.024M
Cutoff Freq: 30k
Transition Width: 5
Window: Hamming
Beta: 6.76

QT GUI Time Sink
Name: Z Demodu... Signal Time
Number of Points: 1.024k
Sample Rate: 1.024M
Autoscale: No

QT GUI Frequency Sink
Name: Demodula...al Frequency
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 1.024M
```

Figure 2 – Multitone Double-Sideband Suppressed-Carrier (DSB-SC) Amplitude Modulation



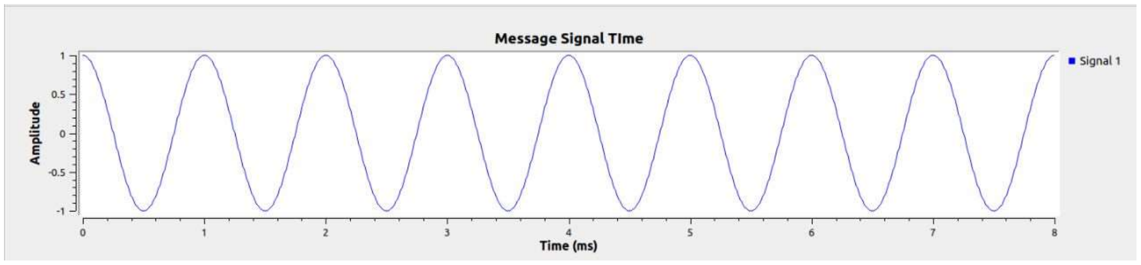
```
Options
Title: Amplitud...on_WAV_File
Author: Santosh
Copyright: 2022
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 48k

Wav File Source
File: .../Impact_Moderato.wav
Repeat: Yes

Signal Source
Sample Rate: 48k
Waveform: Cosine
Frequency: 24k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

QT GUI Sink
Name: Baseband Signal
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 48k
Update Rate: 10

QT GUI Sink
Name: Carrier Signal
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 48k
Update Rate: 10

QT GUI Sink
Name: Modulated Signal
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 48k
Update Rate: 10

Low Pass Filter
Interpolation: 1
Gain: 1
Sample Rate: 48k
Cutoff Freq: 12k
Transition Width: 120
Window: Hamming
Beta: 6.76

QT GUI Sink
Name: Demodulated Signal
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 48k
Update Rate: 10

Wav File Sink
File: ...AM/Demodulated_Audio
Sample Rate: 48k
Output Format: WAV
Bits per Sample: 16-bit
Append to existing file: No
```

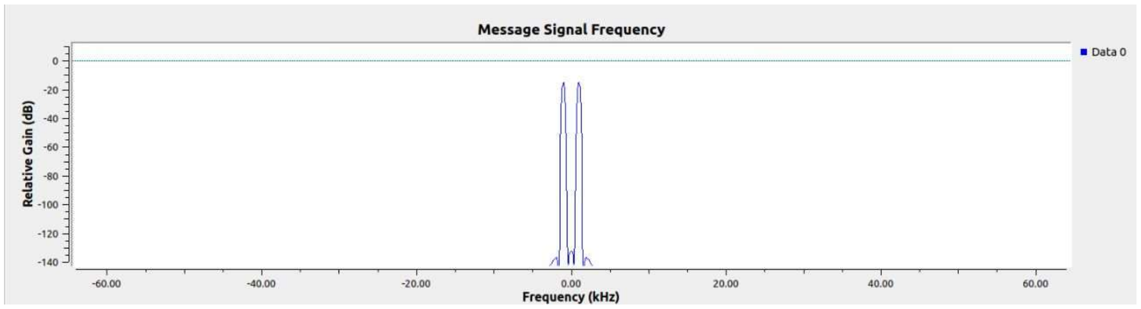
Figure 3 – Amplitude Modulation using WAV File Source

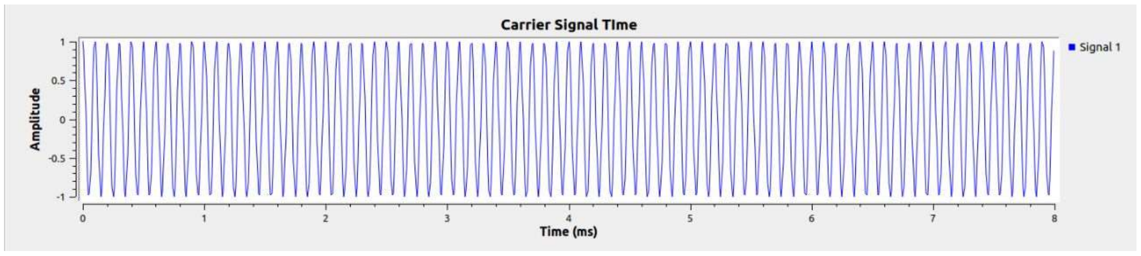
PROCEDURE:

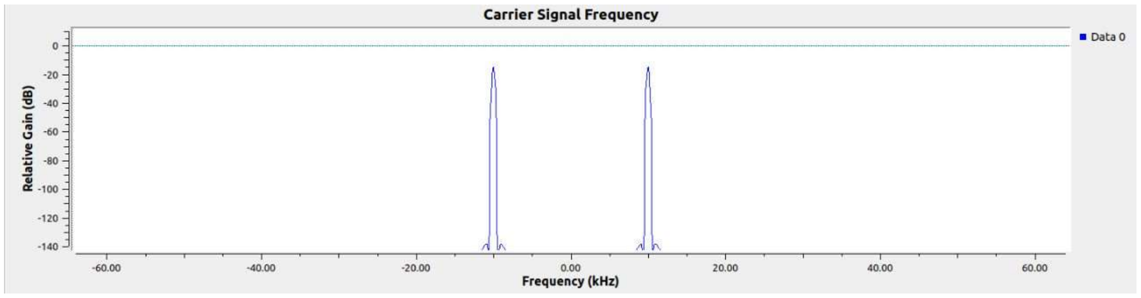
1. Connect the blocks as per the above diagrams.
2. Fix the values for the parameters.
3. Generate the python file for the GRC and execute it.
4. Vary the range of the variables declared during run time.
5. Check the results in the QT GUI Sink.

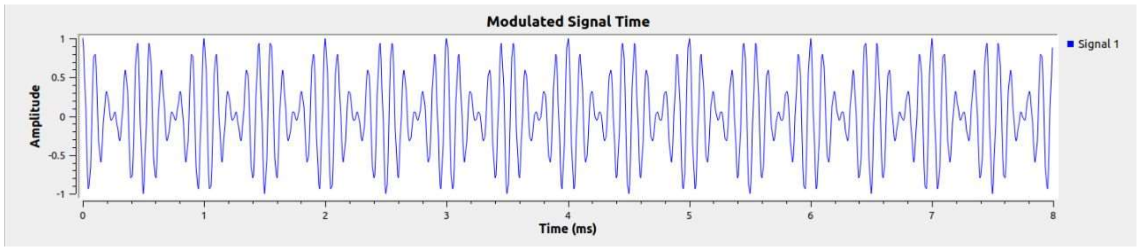
RESULTS:

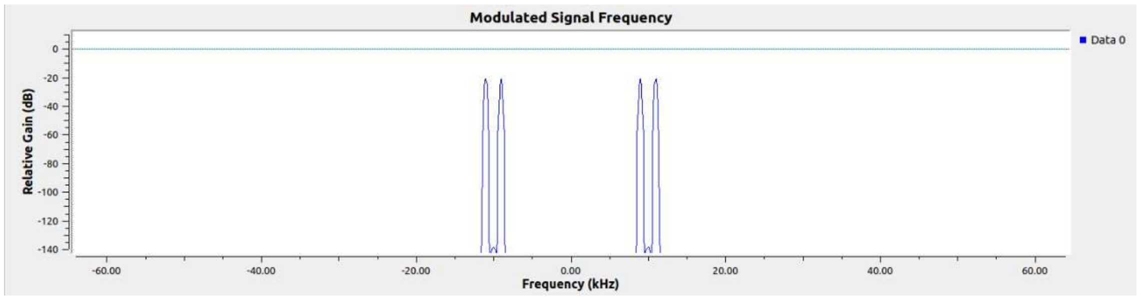


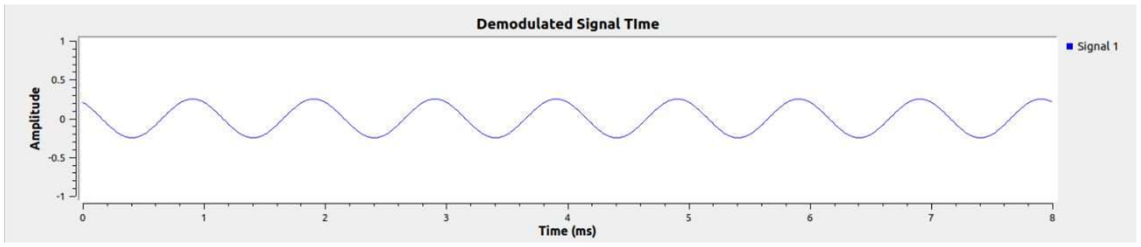












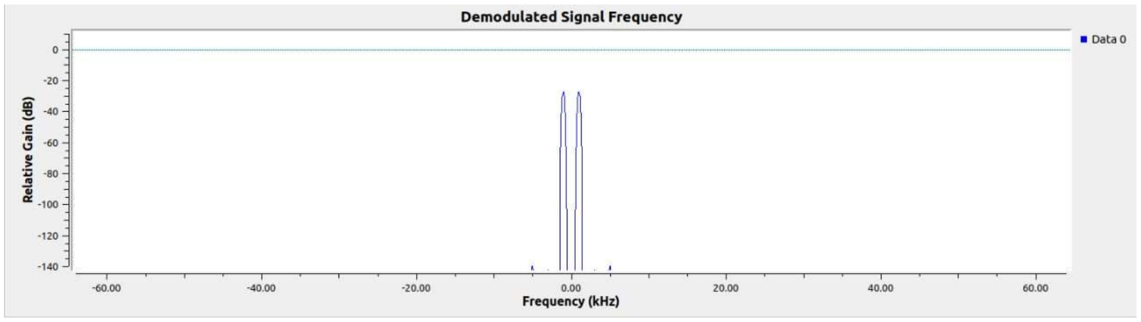
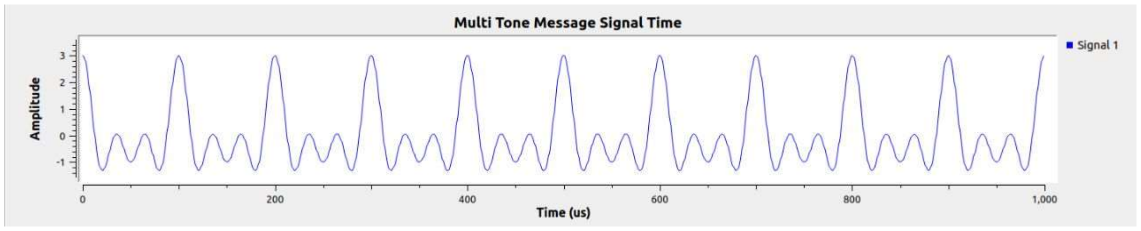
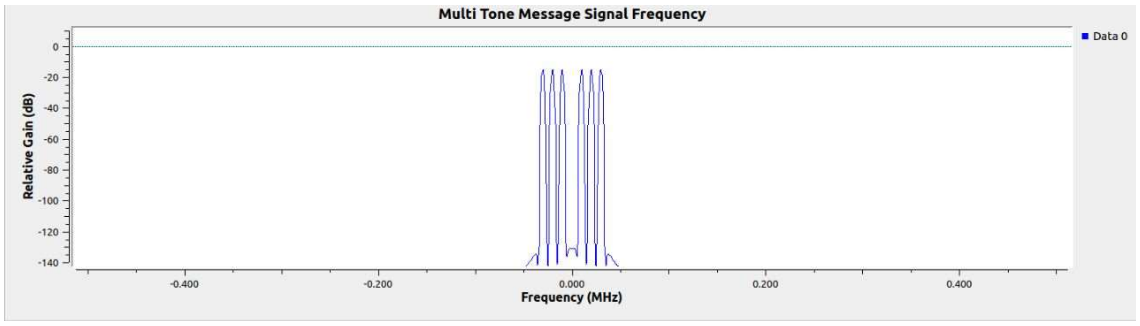
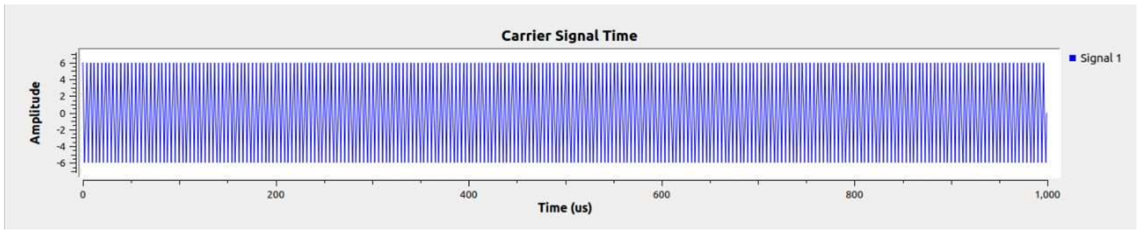
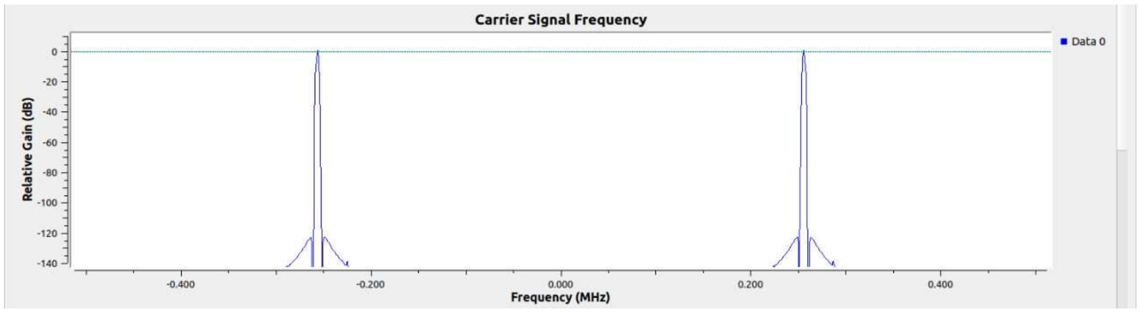


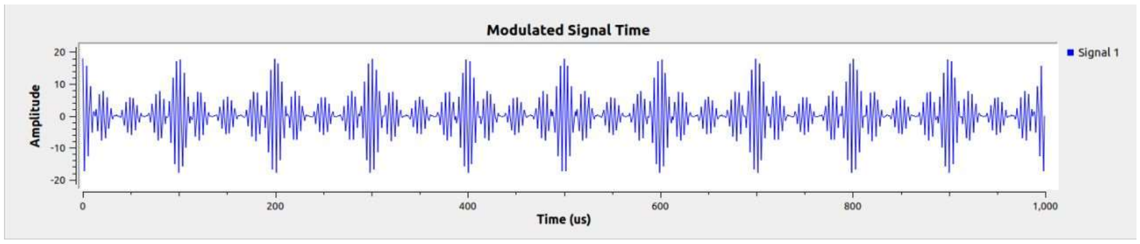
Figure 4 – Singletone Double-Sideband Suppressed-Carrier (DSB-SC) Amplitude Modulation

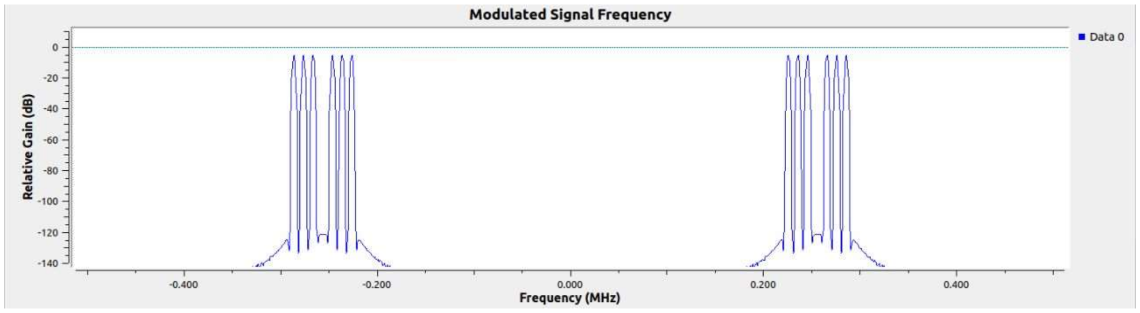


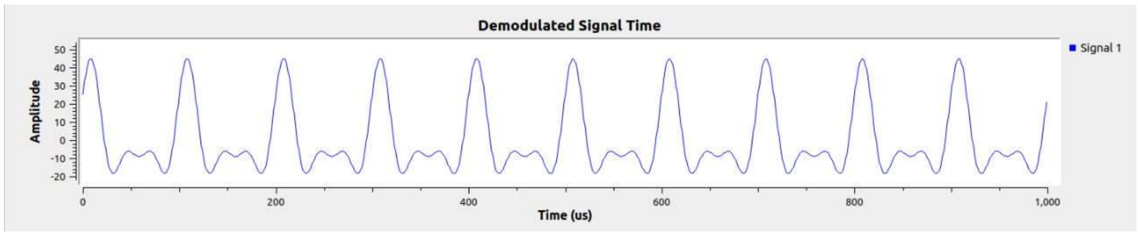

Multi Tone Message Signal Frequency
Relative Gain (dB)
Frequency (MHz)
Data 0


Carrier Signal Time
Amplitude
Time (us)
Signal 1


Carrier Signal Frequency
Relative Gain (dB)
Frequency (MHz)
Data 0


Modulated Signal Time
Amplitude
Time (us)
Signal 1


Modulated Signal Frequency
Relative Gain (dB)
Frequency (MHz)
Data 0


Demodulated Signal Time
Amplitude
Time (us)
Signal 1

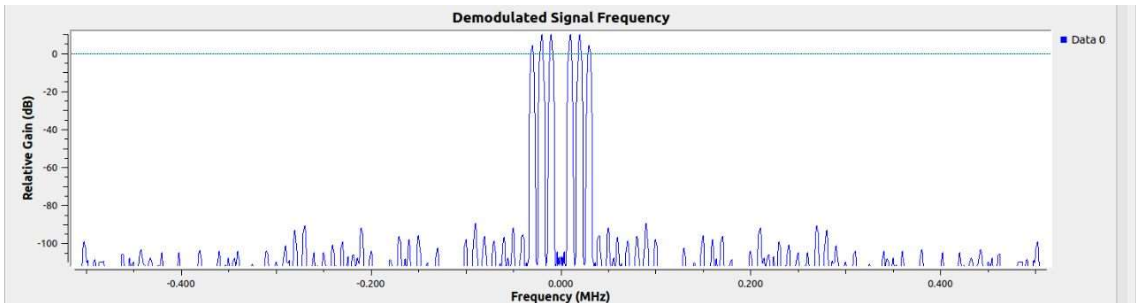


Figure 5 – Multitone Double-Sideband Suppressed-Carrier (DSB-SC) Amplitude Modulation

INFERENCE:

Thus, performed amplitude modulation and demodulation for the given message and carrier signals. All the simulation results were verified successfully.

EXPERIMENT 4 –FREQUENCY MODULATION AND DEMODULATION

AIM:

To perform frequency modulation and demodulation for the given message and carrier signals.

SOFTWARE REQUIRED:

1. Oracle VM VirtualBox 6.1.38, Oracle Corporation
2. Ubuntu 22.04 (64-bit) Operating System
3. GNU Radio Companion Application, v3.10.1 (sudo apt-get install gnuradio)

THEORY:

Frequency modulation is a technique or a process of encoding information on a particular signal (analogue or digital) by varying the carrier wave frequency per the frequency of the modulating signal. As we know, a modulating signal is nothing but information or message that has to be transmitted after being converted into an electronic signal.

Much like amplitude modulation, frequency modulation also has a similar approach where a carrier signal is modulated by the input signal. However, in the case of FM, the amplitude of the modulated signal is kept or remains constant.

The frequency modulation index is mostly over 1 and it usually requires a high bandwidth at a range of 200 kHz. FM operates in a very high-frequency range normally between 88 to 108 Megahertz. There are complex circuits with an infinite number of sidebands that help in receiving high-quality signals with high sound quality.

Meanwhile, broadcast stations in the VHF portion of the frequency spectrum between 88.5 and 108 MHz often use large values of deviation (±75 kHz). This is known as wide-band FM (WBFM). Even though these signals support high-quality transmissions they do occupy a large amount of bandwidth. Normally, 200 kHz is allowed for each wide-band FM transmission.

On the other hand, communications use very little bandwidth. Alternatively, narrowband FM (NBFM) often uses deviation figures of around ±3 kHz. Besides, narrow-band FM is mostly used for two-way radio communication applications.

BLOCK DIAGRAM:

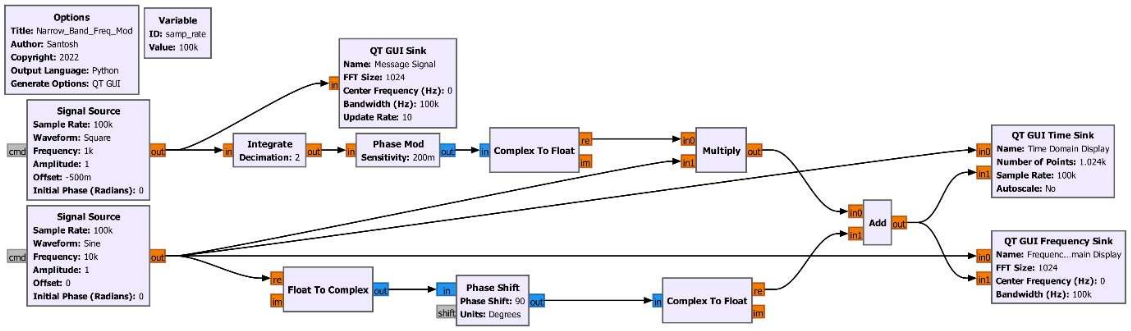


Figure 1 – Narrow-Band Frequency Modulation

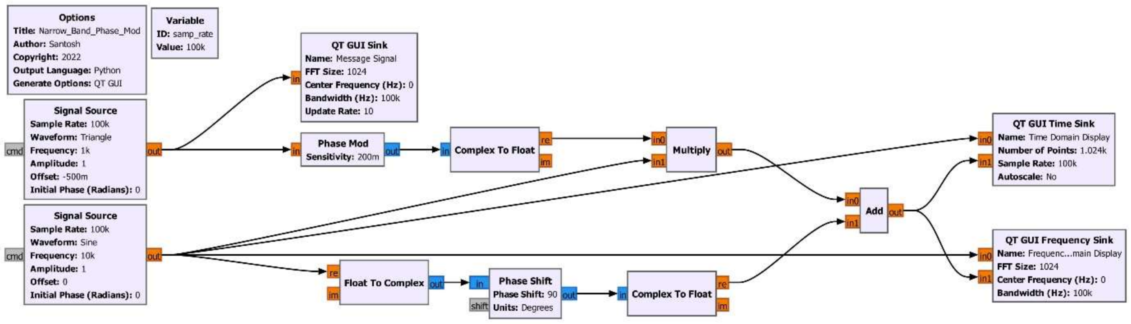


Figure 2 – Narrow-Band Phase Modulation

PROCEDURE:

1. Connect the blocks as per the above diagrams.
2. Fix the values for the parameters.
3. Generate the python file for the GRC and execute it.
4. Vary the range of the variables declared during run time.
5. Check the results in the QT GUI Sink.

RESULTS:

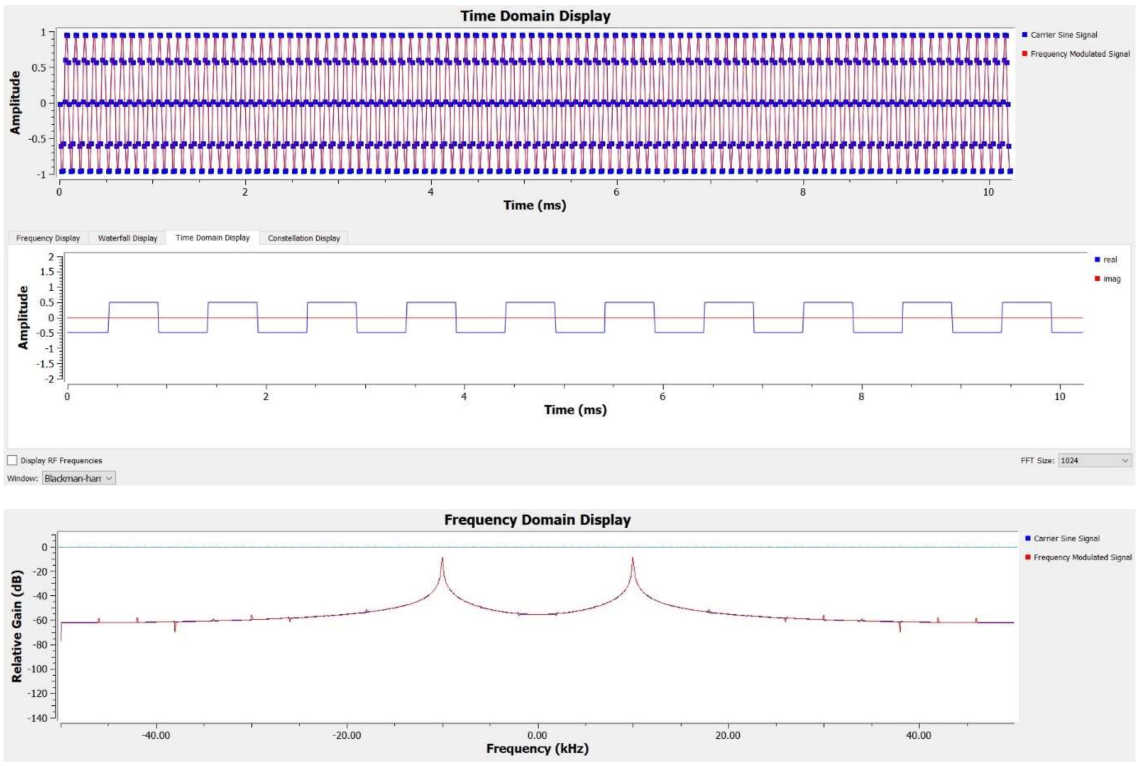


Figure 3 – Narrow-Band Frequency Modulation

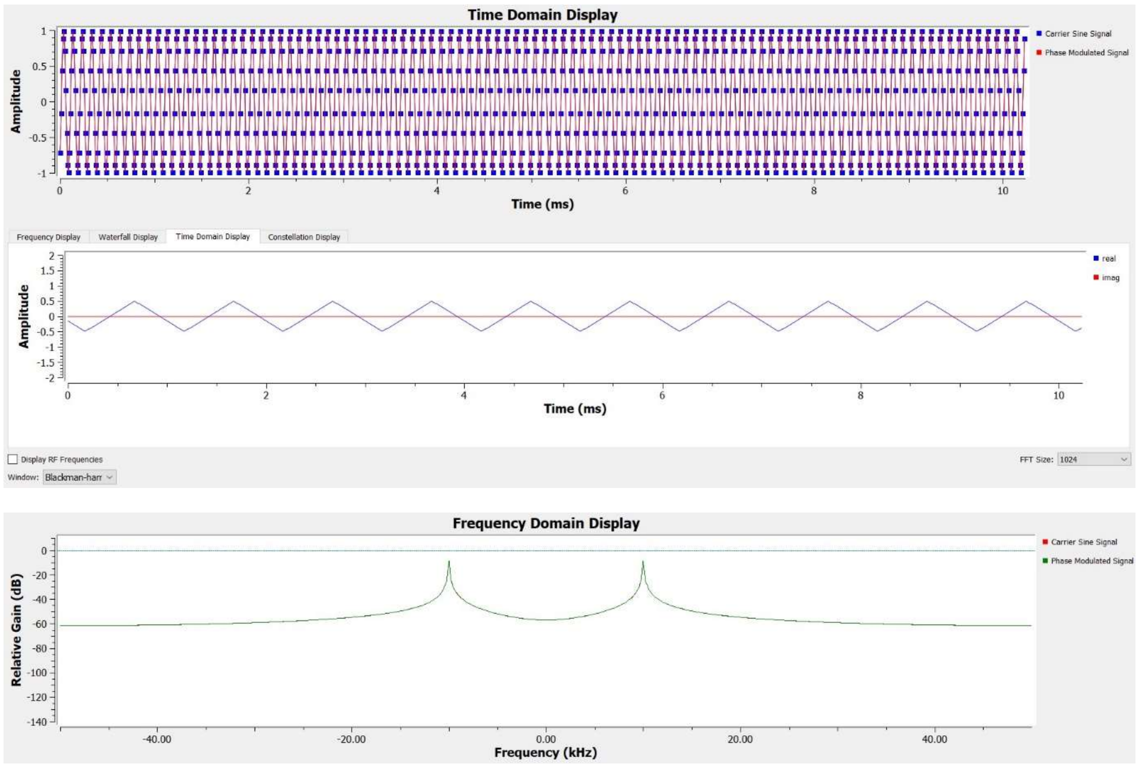


Figure 4 – Narrow-Band Phase Modulation

INFERENCE:

Thus, performed frequency modulation and demodulation for the given message and carrier signals. All the simulation results were verified successfully.

EXPERIMENT 5 –BINARY PHASE SHIFT KEYING (BPSK)

AIM:

To perform Binary Phase Shift Keying (BPSK) modulation and demodulation for the given message and carrier signals.

SOFTWARE REQUIRED:

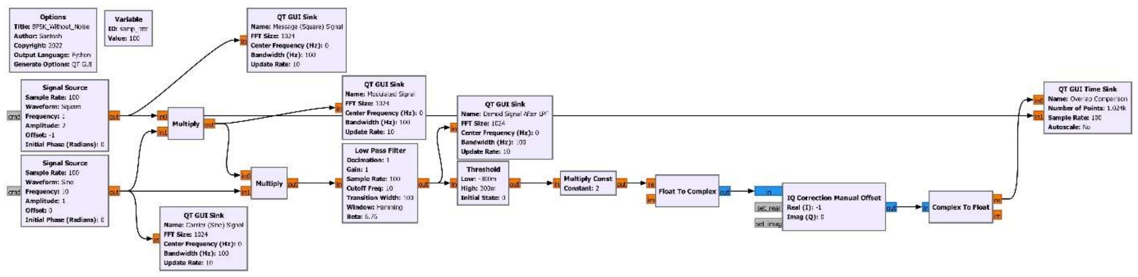
1. Oracle VM VirtualBox 6.1.38, Oracle Corporation
2. Ubuntu 22.04 (64-bit) Operating System
3. GNU Radio Companion Application, v3.10.1 (sudo apt-get install gnuradio)

THEORY:

Phase Shift Keying (PSK) is the digital modulation technique in which the phase of the carrier signal is changed by varying the sine and cosine inputs at a particular time. PSK technique is widely used for wireless LANs, bio-metric, and contactless operations, along with RFID and Bluetooth communications.

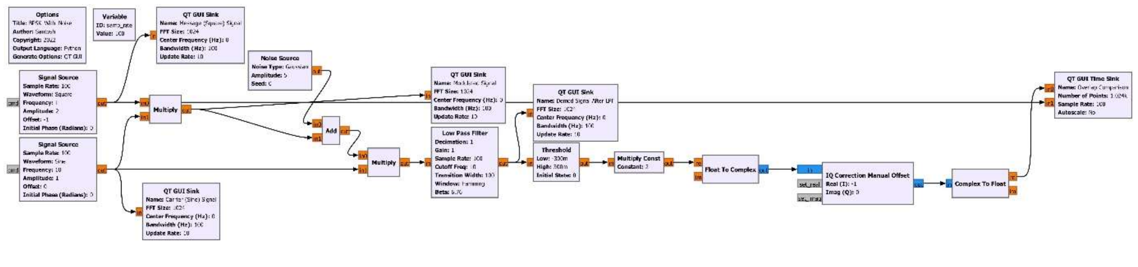
Binary Phase Shift Keying (BPSK) is also called 2-phase PSK or Phase Reversal Keying. In this technique, the sine wave carrier takes two phase reversals such as 0° and 180°. BPSK is a Double Side Band Suppressed Carrier DSB-SC – DSB-SC modulation scheme, for the message being the digital information.

BLOCK DIAGRAM:



Options Title: BPSK_Without_Noise; Author: Santosh; Copyright: 2022; Output Language: Python; Generate Options: QT GUI
Variable ID: samp_rate; Value: 100
Signal Source Sample Rate: 100; Waveform: Square; Frequency: 1; Amplitude: 2; Offset: -1; Initial Phase (Radians): 0
Signal Source Sample Rate: 100; Waveform: Sine; Frequency: 10; Amplitude: 1; Offset: 0; Initial Phase (Radians): 0
QT GUI Sink Name: Message (Square) Signal; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
QT GUI Sink Name: Modulated Signal; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
QT GUI Sink Name: Carrier (Sine) Signal; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
QT GUI Sink Name: Demod Signal After LPF; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
Low Pass Filter Decimation: 1; Gain: 1; Sample Rate: 100; Cutoff Freq: 10; Transition Width: 100; Window: Hamming; Beta: 6.76
Threshold Low: -300m; High: 300m; Initial State: 0
Multiply Const Constant: 2
IQ Correction Manual Offset Real (I): -1; Imag (Q): 0
QT GUI Time Sink Name: Overlap Comparison; Number of Points: 1.024k; Sample Rate: 100; Autoscale: No

Figure 1 – Binary Phase Shift Keying (BPSK) Without Noise



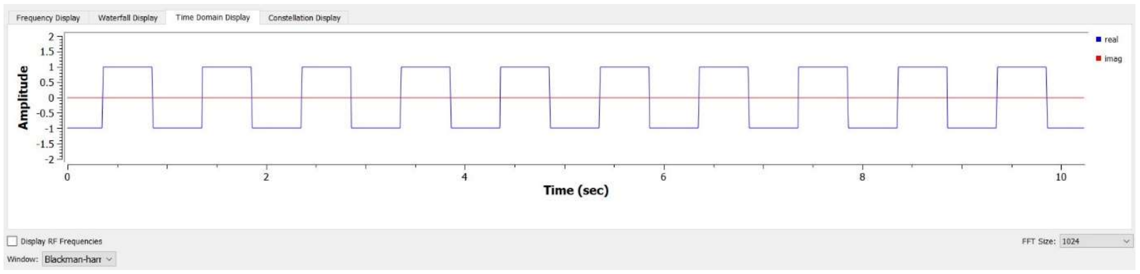
Options Title: BPSK_With_Noise; Author: Santosh; Copyright: 2022; Output Language: Python; Generate Options: QT GUI
Variable ID: samp_rate; Value: 100
Signal Source Sample Rate: 100; Waveform: Square; Frequency: 1; Amplitude: 2; Offset: -1; Initial Phase (Radians): 0
Signal Source Sample Rate: 100; Waveform: Sine; Frequency: 10; Amplitude: 1; Offset: 0; Initial Phase (Radians): 0
Noise Source Noise Type: Gaussian; Amplitude: 5; Seed: 0
QT GUI Sink Name: Message (Square) Signal; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
QT GUI Sink Name: Modulated Signal; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
QT GUI Sink Name: Carrier (Sine) Signal; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
QT GUI Sink Name: Demod Signal After LPF; FFT Size: 1024; Center Frequency (Hz): 0; Bandwidth (Hz): 100; Update Rate: 10
Low Pass Filter Decimation: 1; Gain: 1; Sample Rate: 100; Cutoff Freq: 10; Transition Width: 100; Window: Hamming; Beta: 6.76
Threshold Low: -300m; High: 300m; Initial State: 0
Multiply Const Constant: 2
IQ Correction Manual Offset Real (I): -1; Imag (Q): 0
QT GUI Time Sink Name: Overlap Comparison; Number of Points: 1.024k; Sample Rate: 100; Autoscale: No

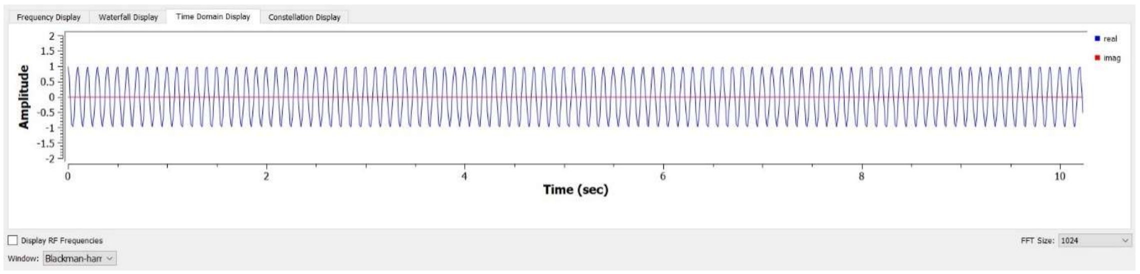
Figure 2 – Binary Phase Shift Keying (BPSK) With Noise

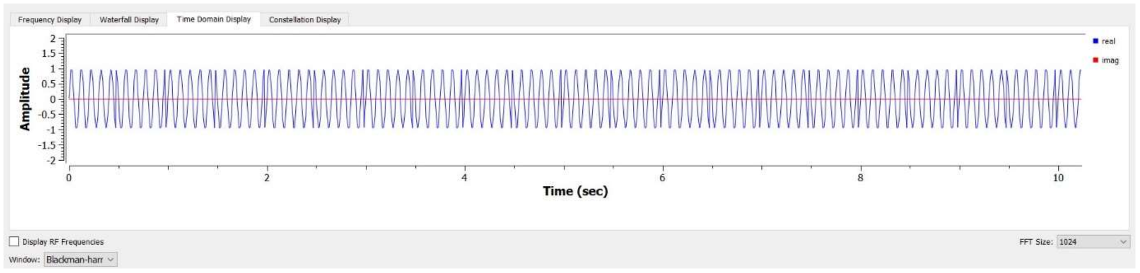
PROCEDURE:

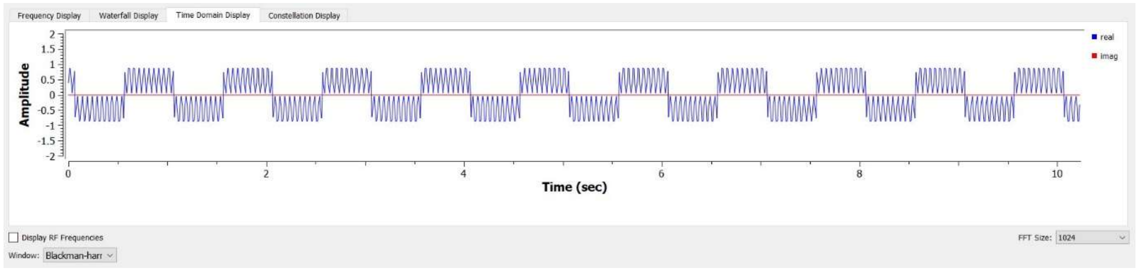
1. Connect the blocks as per the above diagrams.
2. Fix the values for the parameters.
3. Generate the python file for the GRC and execute it.
4. Vary the range of the variables declared during run time.
5. Check the results in the QT GUI Sink.

RESULTS:









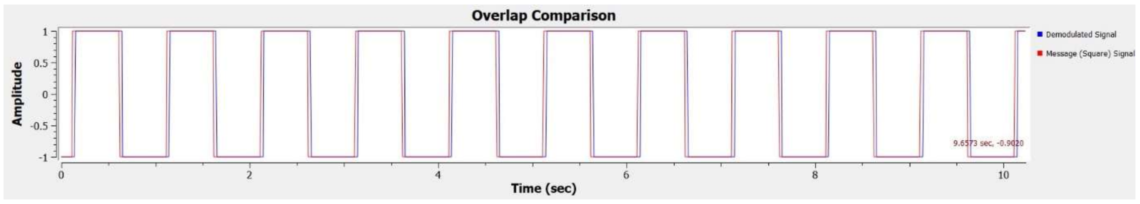
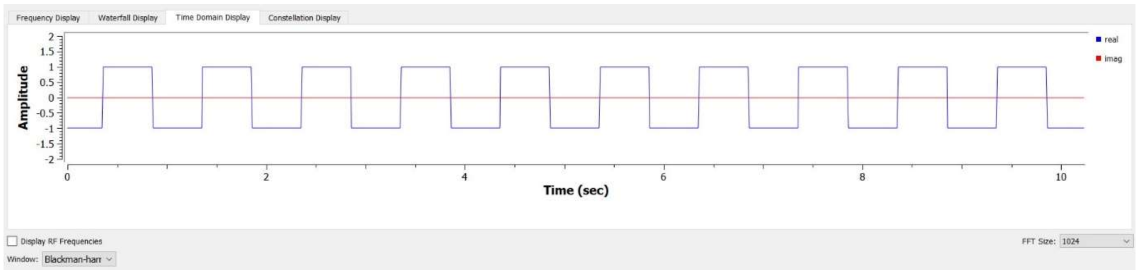
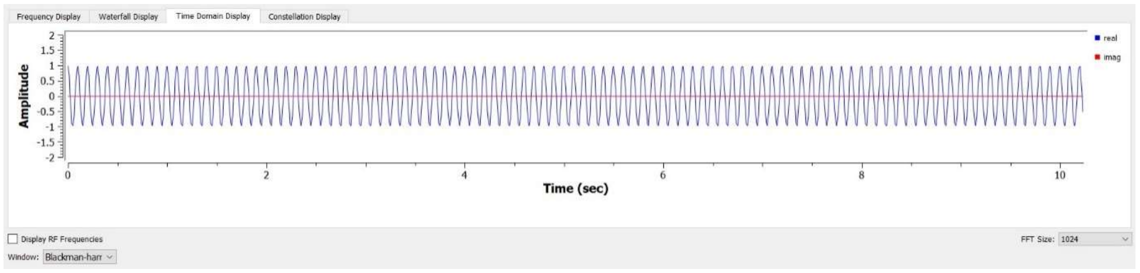
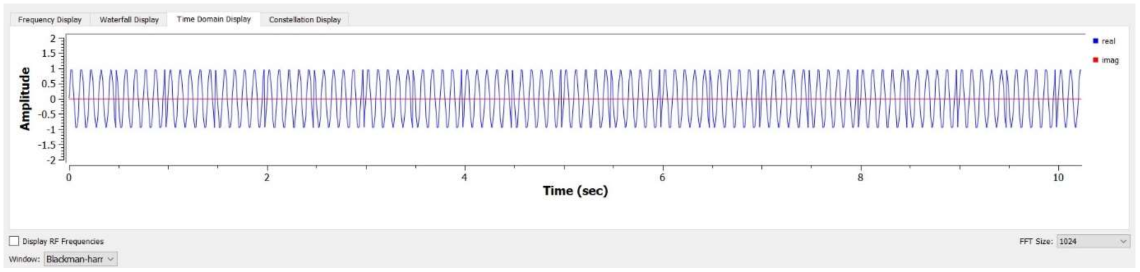
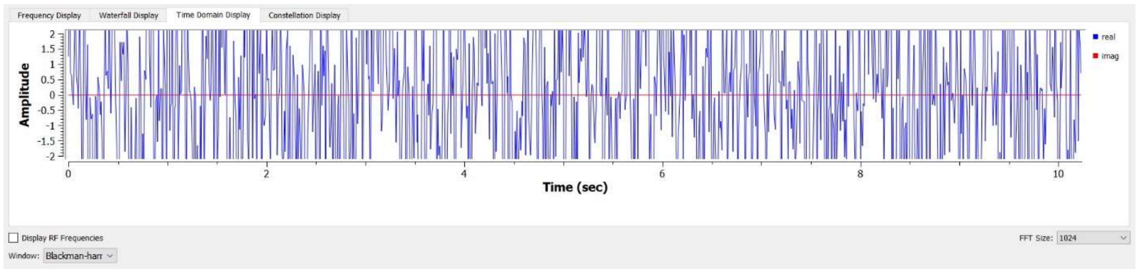


Figure 1 – Binary Phase Shift Keying (BPSK) Without Noise









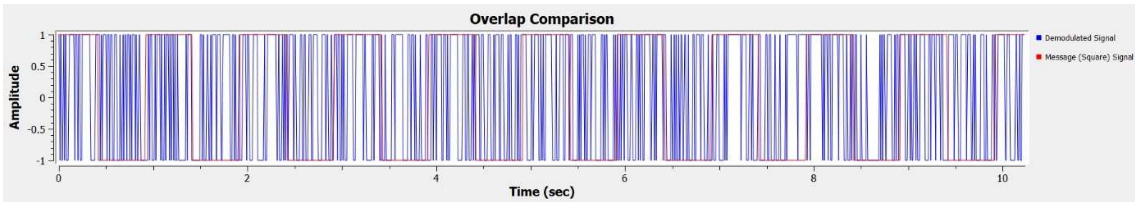


Figure 2 – Binary Phase Shift Keying (BPSK) With Noise

INFERENCE:

Thus, performed Binary Phase Shift Keying (BPSK) modulation and demodulation for the given message and carrier signals. All the simulation results were verified successfully.

EXPERIMENT 6 –QUADRATURE AMPLITUDE MODULATION (QAM)

AIM:

To perform Quadrature Amplitude Modulation (QAM) for the given input signals.

SOFTWARE REQUIRED:

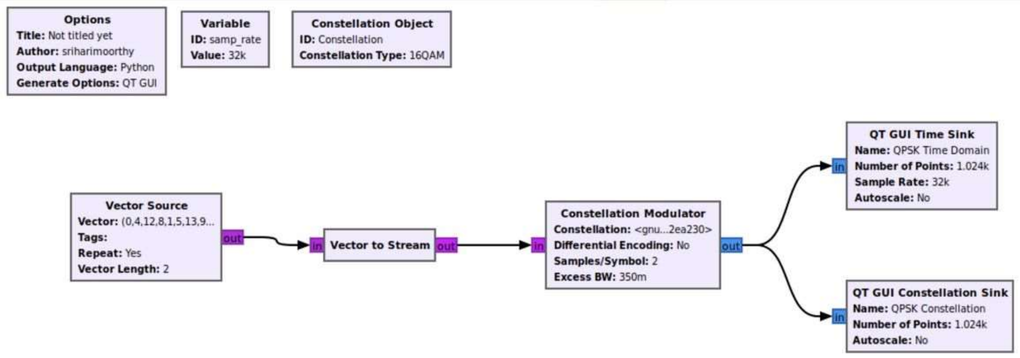
1. Oracle VM VirtualBox 6.1.38, Oracle Corporation
2. Ubuntu 22.04 (64-bit) Operating System
3. GNU Radio Companion Application, v3.10.1 (sudo apt-get install gnuradio)

THEORY:

Quadrature Amplitude Modulation (QAM) is a form of modulation that is a combination of phase and amplitude modulation. In modulation, there are two carrier signals with a phase shift of 90° between them. These are then amplitude modulated with two data streams known as in-phase and quadrature phases.

These signals are added thus converting them to the required frequency and amplifying them. During modulation, the modulated signal is split and each side is applied to a mixer. One half has an impulse local oscillator applied, then other half has a quadrature oscillator signal applied. Since it includes both amplitude and phase modulation, we obtain a grid called a constellation map.

BLOCK DIAGRAM:



Options
Title: Not titled yet
Author: sriharimoorthy
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Constellation Object
ID: Constellation
Constellation Type: 16QAM

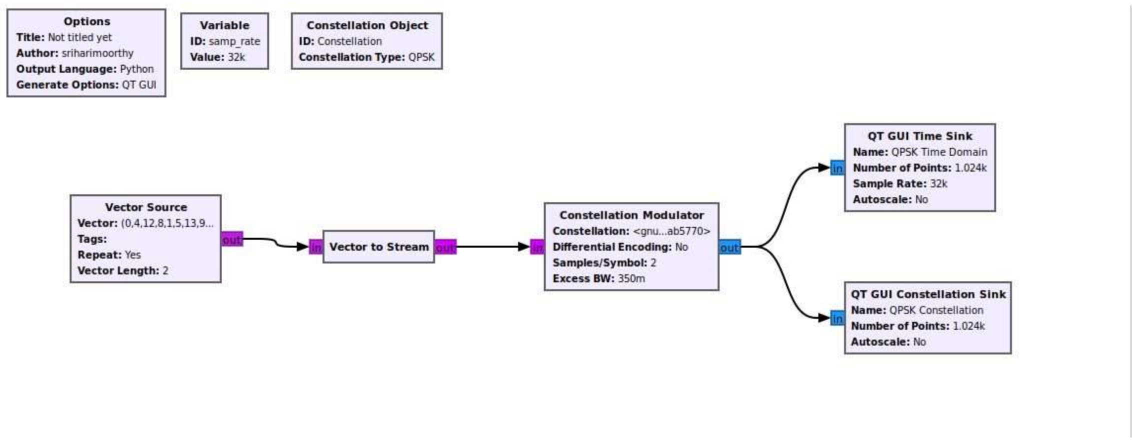
Vector Source
Vector: (0,4,12,8,1,5,13,9...
Tags:
Repeat: Yes
Vector Length: 2

Constellation Modulator
Constellation: <gnu...2ea230>
Differential Encoding: No
Samples/Symbol: 2
Excess BW: 350m

QT GUI Time Sink
Name: QPSK Time Domain
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

QT GUI Constellation Sink
Name: QPSK Constellation
Number of Points: 1.024k
Autoscale: No

Figure 1 – 16 Quadrature Amplitude Modulation (QAM)



Options
Title: Not titled yet
Author: sriharimoorthy
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 32k

Constellation Object
ID: Constellation
Constellation Type: QPSK

Vector Source
Vector: (0,4,12,8,1,5,13,9...
Tags:
Repeat: Yes
Vector Length: 2

Vector to Stream

Constellation Modulator
Constellation: <gnu...ab5770>
Differential Encoding: No
Samples/Symbol: 2
Excess BW: 350m

QT GUI Time Sink
Name: QPSK Time Domain
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

QT GUI Constellation Sink
Name: QPSK Constellation
Number of Points: 1.024k
Autoscale: No

Figure 2 – Quadrature Phase Shift Keying (QPSK)

PROCEDURE:

1. Connect the blocks as per the above diagram.
2. Fix the values for the parameters.
3. Generate the python file for the GRC and execute it.
4. Vary the range of the variables declared during run time.
5. Check the results in the QT GUI Sink.

RESULTS:

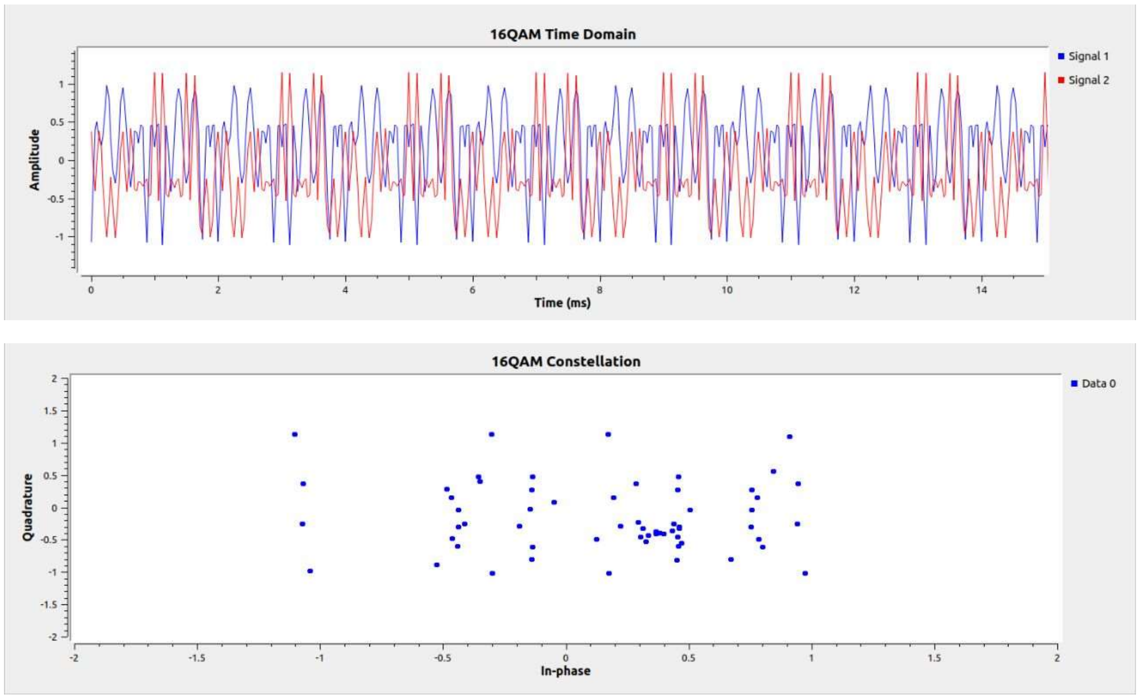
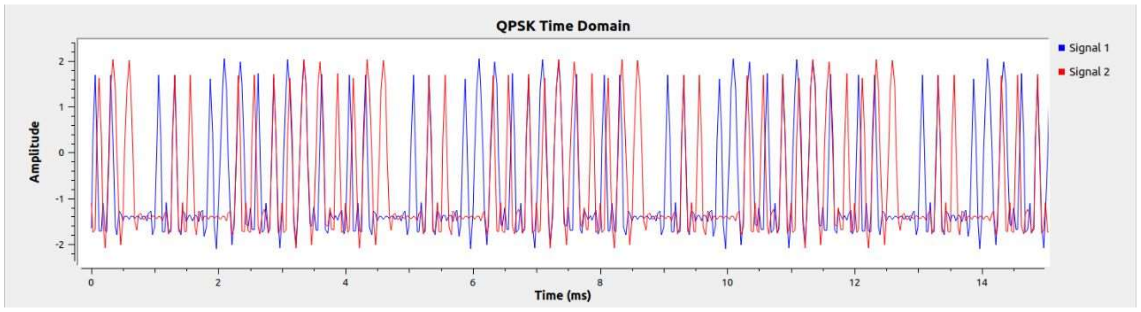


Figure 1 – 16 Quadrature Amplitude Modulation (QAM)



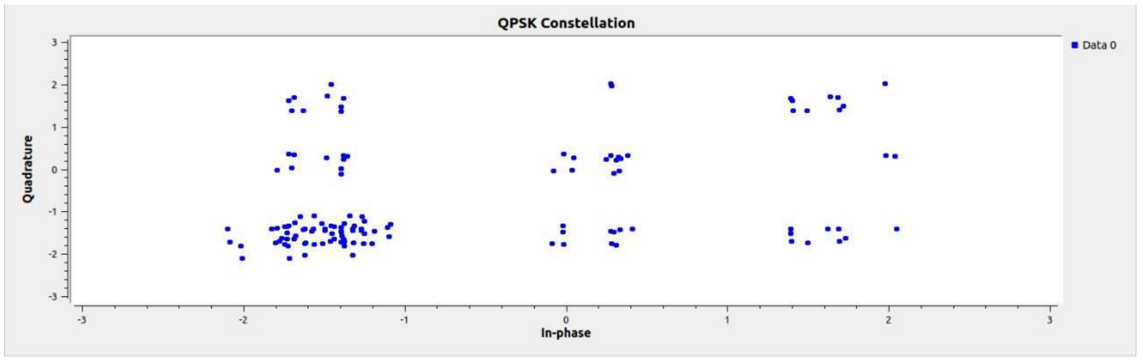


Figure 2 – Quadrature Phase Shift Keying (QPSK)

INFERENCE:

Thus, Quadrature Amplitude Modulation (QAM) has been performed for digital signal and signal added with noise. All the simulation results were verified successfully.

EXPERIMENT 7 – PROBABILITY OF ERROR USING PULSE AMPLITUDE MODULATION (PAM)

AIM:

To estimate and plot the probability of error when binary antipodal signals are transmitted over an additive white Gaussian noise channel.

SOFTWARE REQUIRED:

1. Anaconda3 2021.11 (Python 3.9.7 64-bit)
2. Spyder 5.1.5 Integrated Development Environment (IDE)

THEORY:

The purpose of this problem is to estimate and plot the probability of error when binary antipodal signals are transmitted over an additive white Gaussian noise channel. The model of the binary communication system employing antipodal signals is shown in Figure. As shown, we simulate the generation of the random variable y, which is the input to the detector.

A uniform random number generator is used to generate the binary information sequence of zeros and ones from the data source. The sequence of zeros and ones is mapped into a sequence of ±1. A Gaussian noise generator is used to generate a sequence of zero-mean Gaussian numbers with variance σ^2.

For equally probable zeros and ones, the detector compares the random variable y with the threshold zero. (Cross correlation can also be applied) If $y > 0$, the decision is made that the transmitted bit is a zero. If $y < 0$, the decision is made that the transmitted bit is a 1.

The output of the detector is compared with the transmitted sequence of information bits, and the bit errors are counted. Perform the simulation for the transmission of 10,000 bits at several different values of SNR, which covers the range of SNR 0 to 10. Plot the error probability as a function of the SNR. Compare the estimated error probability with the theoretical error probability given by the formula:

$$P_2 = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

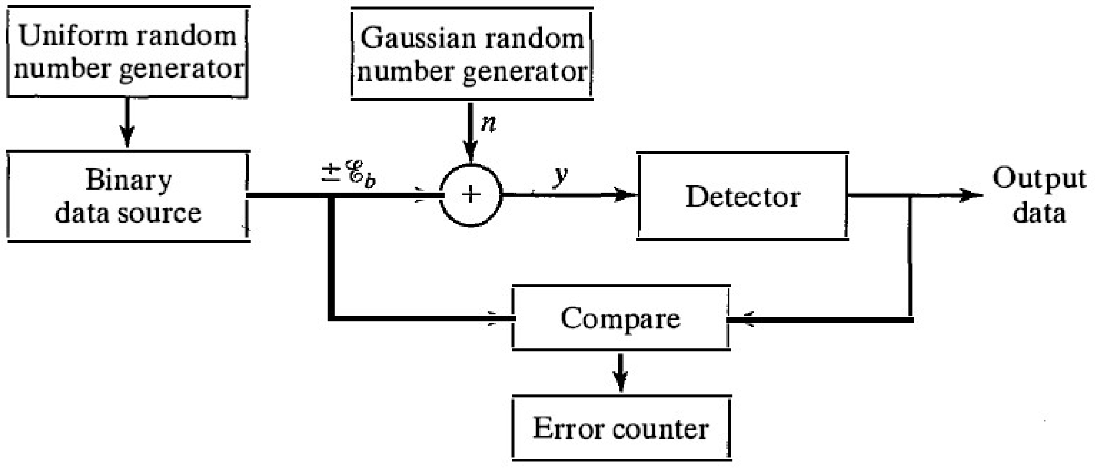


Figure 1 – Model of the Binary Communication System Employing Antipodal Signals

PROCEDURE:

1. In this part, you will simulate the performance of a BPAM communication system transmitting 10,000 binary antipodal signals over an additive white Gaussian noise (AWGN) channel.
2. Generate the random binary information sequence b of 0's and 1's from the data source. Assume that the bits are equally probable. You can use a uniform random number generator function to generate numbers in the range [0, 1].
3. If a number generated is in the range [0, 0.5], then you can consider that the binary source output is a '0'. Otherwise, it is a 1'. Map the previously generated sequence of bits into a sequence of symbols s.
4. The bits '0' and '1' will be mapped into symbols 1 and -1 respectively, to represent the signal energy per bit. For convenience, you may normalize the signal energy unity.
5. Use a Gaussian noise generator is used to generate a sequence n of zero-mean Gaussian numbers with variance $=N_0/2$. The values may be defined by the values of the SNRs in dB.
6. Assume that the detector will use the optimal threshold a*, and compare the received signal r = s + n with a* to decide whether the originally transmitted bit is '0' or '1'. Count the number of bit errors.
7. Vary the values of σ^2, and perform the simulation for the transmission of the 10,000 bits at several different values of SNR, which covers the range of SNRs $0 < 10 \log_{10}(E_b/N) = 10$ dB.

DELIVERABLES:

1. In this part, you will assume that the transmitted symbol is always +1. Plot the distributions of the received signals r when the SNR is 2 dB, and 10 dB. Use the histogram,

```
import numpy as np
import math as ma
import matplotlib.pyplot as plt

s = np.random.normal(0,1,10000)

count, bins, ignored = plt.hist(s, 100, density=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
plt.show()
```

2. In this part, you will assume that the transmitted symbol could be +1 or -1. In a new figure, plot the error probability P as a function of the SNR.
3. In the same figure of part b), plot the theoretical error probability given by the formula,

$$P_2 = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

4. Use the math.erfc(x),

$$Q(z) = \frac{1}{2} erfc\left(\frac{z}{\sqrt{2}}\right)$$

PYTHON CODE:

```
import matplotlib.pyplot as plt # Provides an implicit way of plotting
import numpy as np # Support for large, multi-dimensional arrays and matrices
import math # Provides access to the mathematical functions defined by the C
standard

t = []
xt = np.random.normal(0, 1, 10000)
for i in range(len(xt)):
    if i>0.5:
        t.append(i)
    elif i<=0.5:
        t.append(0)

y = []
for i in t:
    if i==0:
        y.append(1)
    else:
        y.append(-1)

x = np.arange(0, len(y), 1); plt.step(x, y)
plt.xlim(0, 100); plt.show()

l = p = []
for num in range(1,11):
    final = error = rt = []
    sigma = (10*10**(-num/10)/2)
    print("\n" + str(num) + " - Sigma Value: " + str(sigma))

    noise = np.random.normal(0,sigma,10000)
    count, bins, ignored = plt.hist(noise, 1000, density=True)
    plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
    plt.show()

    rt = y + noise
    plt.plot(rt); plt.title("Signal with Noise")
    plt.xlim(0,100); plt.show()

    for i in range(len(rt)):
        if rt[i]>0:
            final.append(1)
        else:
            final.append(-1)

    error_count = 0
    for i in range(len(rt)):
        if final[i] != y[i]:
            error_count += 1
    print("    Error Count: " + str(error_count)); error.append(error_count)

    plt.stem(final)
    plt.xlabel("Sequence"); plt.ylabel("Ampltiude"); plt.title("Demodulated
Signal")
    plt.xlim(0,100); plt.show()

    for i in range(len(error)):
        l.append(error[i]/10000)
    p.append(math.erfc(np.sqrt(num)*np.sqrt(2))) # Returns the complementary
error function of a number

plt.semilogx(p); plt.semilogy(l)
plt.xlabel("SNR in dB"); plt.ylabel("Probability of Error (PAM)")
plt.show()
```

RESULTS:

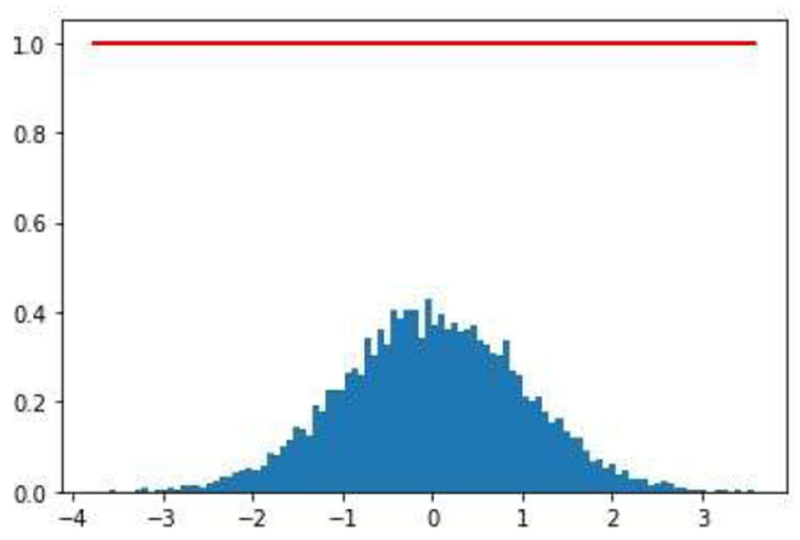


Figure 2 – Histogram of the Noise Added

Mean = 0 & Variance = 1

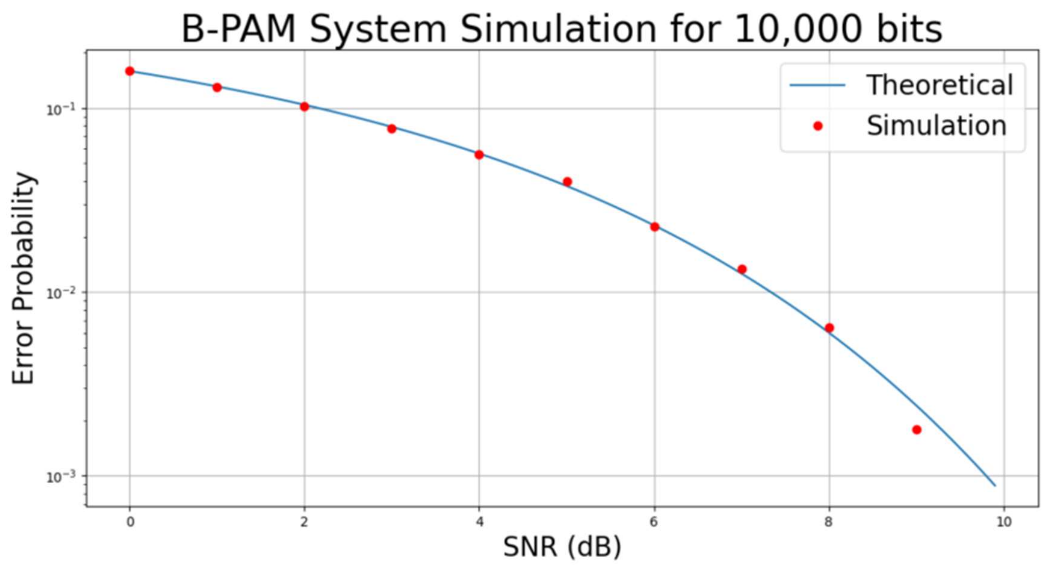


Figure 3 – B-PAM System Simulation for 10,000 Bits

INFERENCE:

Plotted the probability of error for binary antipodal signals over an additive white gaussian noise channel and found that SNR increases as the probability of error decreases.

EXPERIMENT 8 – SIMULATION OF EYE DIAGRAM FOR BINARY PHASE SHIFT KEYING (BPSK)

AIM:

To simulate and measure the eye diagram metrics for Binary Phase Shift Keying (BPSK) modulation and demodulation.

SOFTWARE REQUIRED:

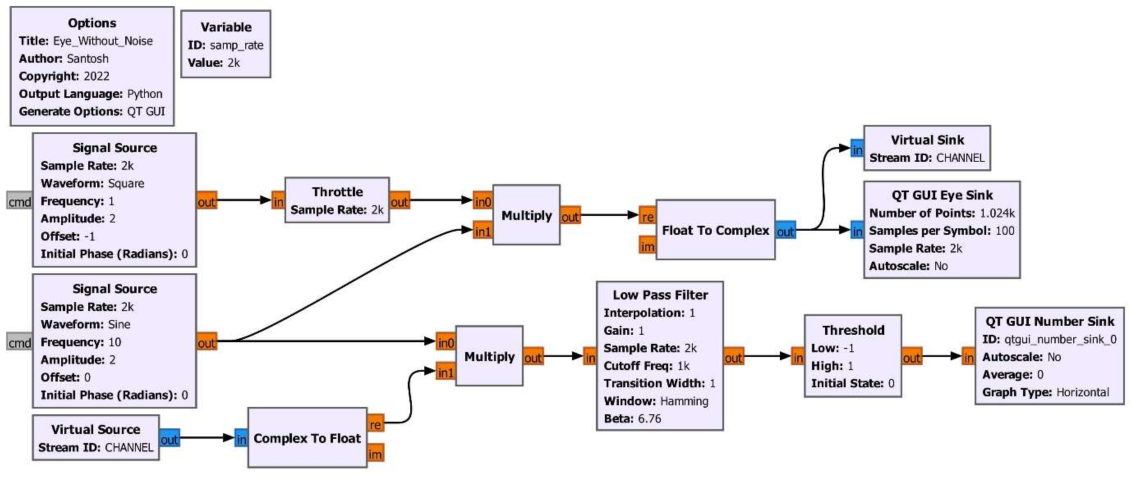
1. Oracle VM VirtualBox 6.1.38, Oracle Corporation
2. Ubuntu 22.04 (64-bit) Operating System
3. GNU Radio Companion Application, v3.10.1 (sudo apt-get install gnuradio)

THEORY:

The width of the eye-opening defines the time interval over which the received wave can be sampled, without an error due to ISI. the best time for sampling is when the eye is open widest. The sensitivity of the system to the timing error is determined by the rate of closure of the eye as the sampling rate is varied.

The height of eye-opening at a specified sampling time defines the margin over the noise. When the effect of ISI is severe, the eye is completely closed and it is impossible to avoid error due to the combined presence of ISI and noise in the system.

BLOCK DIAGRAM:



Options
Title: Eye_Without_Noise
Author: Santosh
Copyright: 2022
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 2k

Signal Source
Sample Rate: 2k
Waveform: Square
Frequency: 1
Amplitude: 2
Offset: -1
Initial Phase (Radians): 0

Throttle
Sample Rate: 2k

Multiply

Float To Complex

Virtual Sink
Stream ID: CHANNEL

QT GUI Eye Sink
Number of Points: 1.024k
Samples per Symbol: 100
Sample Rate: 2k
Autoscale: No

Signal Source
Sample Rate: 2k
Waveform: Sine
Frequency: 10
Amplitude: 2
Offset: 0
Initial Phase (Radians): 0

Multiply

Low Pass Filter
Interpolation: 1
Gain: 1
Sample Rate: 2k
Cutoff Freq: 1k
Transition Width: 1
Window: Hamming
Beta: 6.76

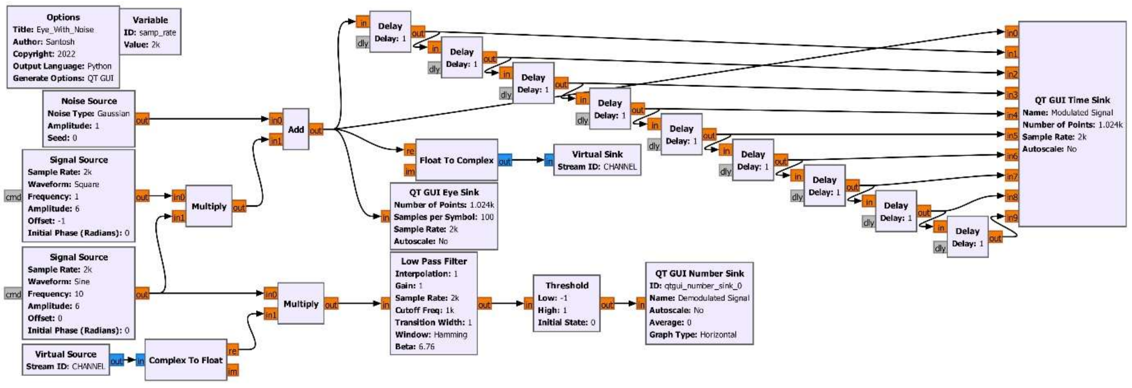
Threshold
Low: -1
High: 1
Initial State: 0

QT GUI Number Sink
ID: qtgui_number_sink_0
Autoscale: No
Average: 0
Graph Type: Horizontal

Virtual Source
Stream ID: CHANNEL

Complex To Float

Figure 1 – Eye Without Noise


Options
Title: Eye_With_Noise
Author: Santosh
Copyright: 2022
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 2k

Noise Source
Noise Type: Gaussian
Amplitude: 1
Seed: 0

Signal Source
Sample Rate: 2k
Waveform: Square
Frequency: 1
Amplitude: 6
Offset: -1
Initial Phase (Radians): 0

Signal Source
Sample Rate: 2k
Waveform: Sine
Frequency: 10
Amplitude: 6
Offset: 0
Initial Phase (Radians): 0

Virtual Source
Stream ID: CHANNEL

Complex To Float

Multiply

Add

Multiply

Float To Complex

Virtual Sink
Stream ID: CHANNEL

QT GUI Eye Sink
Number of Points: 1.024k
Samples per Symbol: 100
Sample Rate: 2k
Autoscale: No

Low Pass Filter
Interpolation: 1
Gain: 1
Sample Rate: 2k
Cutoff Freq: 1k
Transition Width: 1
Window: Hamming
Beta: 6.76

Threshold
Low: -1
High: 1
Initial State: 0

QT GUI Number Sink
ID: qtgui_number_sink_0
Name: Demodulated Signal
Autoscale: No
Average: 0
Graph Type: Horizontal

Delay
Delay: 1

QT GUI Time Sink
Name: Modulated Signal
Number of Points: 1.024k
Sample Rate: 2k
Autoscale: No

Figure 2 – Eye With Noise

PROCEDURE:

1. Connect the blocks as per the above diagram.
2. Fix the values for the parameters.
3. Generate the python file for the GRC and execute it.
4. Vary the range of the variables declared during run time.
5. Check the results in the QT GUI Sink.

RESULTS:

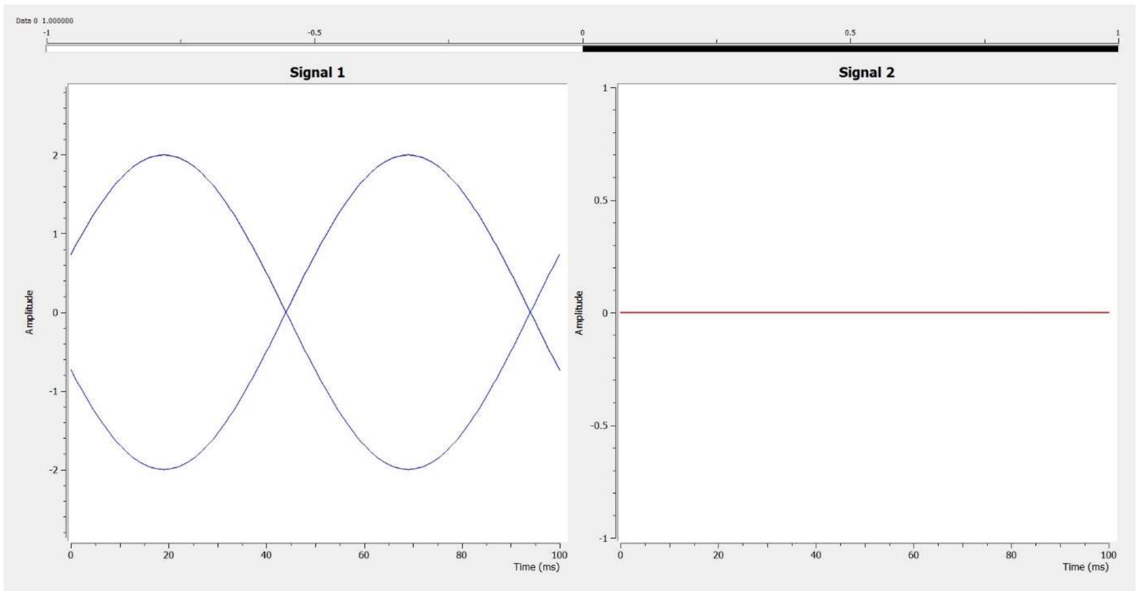


Figure 3 – Eye Without Noise

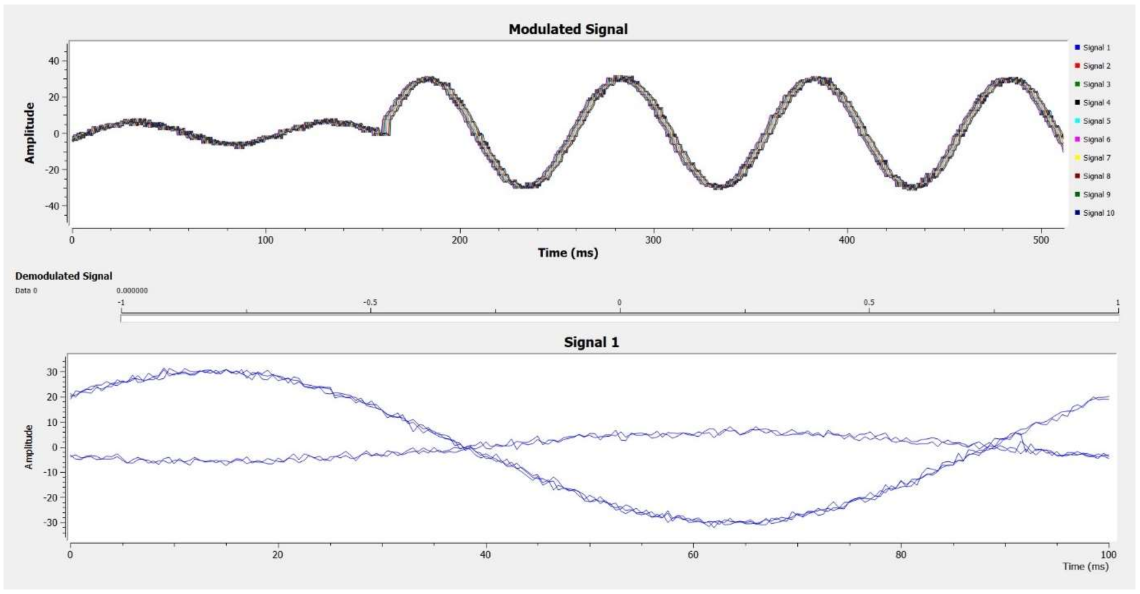


Figure 4 – Eye With Noise

INFERENCE:

Thus, plotted the eye diagram for Binary Phase Shift Keying (BPSK) with and without noise. Also, explored other parameters associated with it such as added delay of symbols, increased amplitude modulation and changes such as slope, distortion, etc. All the simulation results were verified successfully.

EXPERIMENT 9 – RAISED-COSINE SPECTRUM

AIM:

Implement a raised-cosine filter using Python and study its properties by plotting the impulse response with various roll-off factors.

SOFTWARE REQUIRED:

1. Anaconda3 2021.11 (Python 3.9.7 64-bit)
2. Spyder 5.1.5 Integrated Development Environment (IDE)

THEORY:

The raised-cosine filter is a filter frequently used for pulse-shaping in digital modulation due to its ability to minimise intersymbol interference (ISI). Its name stems from the fact that the non-zero portion of the frequency spectrum of its simplest form (β =1) is a cosine function, 'raised' up to sit above the f (horizontal) axis.

The raised-cosine filter is an implementation of a low-pass Nyquist filter, i.e., one that has the property of vestigial symmetry. This means that its spectrum exhibits odd symmetry of about 1/2T, where T is the symbol period of the communications system.

The impulse response of such a filter is given by:

$$h(t) = \begin{cases} \frac{\pi}{4T}\,\mathrm{sinc}\left(\frac{1}{2\beta}\right), & t = \pm\frac{T}{2\beta} \\ \frac{1}{T}\,\mathrm{sinc}\left(\frac{t}{T}\right)\frac{\cos\left(\frac{\pi\beta t}{T}\right)}{1-\left(\frac{2\beta t}{T}\right)^2}, & \text{otherwise} \end{cases}$$

in terms of the normalised sinc function. Here, this is the "communications sinc" sin(πx)/(πx) rather than the mathematical one. The roll-off factor, β, is a measure of the excess bandwidth of the filter, i.e. the bandwidth occupied beyond the Nyquist bandwidth of 1/2T. Some authors use α = β.

As β approaches 0, the roll-off zone becomes infinitesimally narrow, so the impulse response approaches h(t) = 1/T * sinc(t/T). Hence, it converges to an ideal or brick-wall filter in this case. When β =1, the non-zero portion of the spectrum is a pure raised cosine.

The bandwidth of a raised cosine filter is most commonly defined as the width of the non-zero frequency-positive portion of its spectrum, i.e.:

$$BW = Rs * (\beta+1) / 2, \quad (0<\beta<1)$$

As measured using a spectrum analyzer, the radio bandwidth B in Hz of the modulated signal is twice the baseband bandwidth BW, i.e.:

$$B = 2BW = Rs(\beta+1), \quad (0<\beta<1)$$

PYTHON CODE:

```
# PySDR: A Guide to SDR and DSP using Python -
https://pysdr.org/content/pulse_shaping.html

import matplotlib.pyplot as plt # Provides an implicit way of plotting
import numpy as np # Support for large, multi-dimensional arrays and matrices
import warnings
warnings.filterwarnings('ignore') # Never print matching warnings

# Compute DFT coefficients using the linear transformation method:
def DFT(x, plot_name):

    # Compute W(N) 1D Array:
    r1 = c1 = len(x)
    wn = []
    for i in range(r1):
        for j in range(c1):
            wn.append(np.exp(-2j * np.pi * i * j / len(x)))

    # numpy.reshape() is used to give a new shape to an array without changing
its data.

    wn_multidim = np.reshape(wn, (r1, c1)) # An N*N W(N) matrix
    r2 = len(x); c2 = 1
    x_multidim = np.reshape(x, (r2, c2)) # An N*1 x(N) matrix

    # Compute X(N) = W(N) * x(N), an N*1 matrix
    fourier_transform_multidim = [[0]*c2]*r1 # NULL Multidimensional Array
    fourier_transform_l_t = [] # Convert Multidimensional Array to 1D
    for i in range(r1):
        for j in range(c2):
            fourier_transform_multidim[i][j] = 0
            for k in range(c1):
                fourier_transform_multidim[i][j] += wn_multidim[i][k] *
float(x_multidim[k][j])
            fourier_transform_l_t.append(abs(fourier_transform_multidim[i][j])
)

    plt.subplot(1,2,2)
    plt.xlabel("Frequency (f)"); plt.ylabel("Freqeuncy Response, H(f)")
    plt.title(str(plot_name) + "\n" + "in Frequency Domain")
    plt.stem(np.arange(0, len(fourier_transform_l_t)), fourier_transform_l_t)
    plt.grid(True); plt.tight_layout(); plt.show()

# The overlap is fine, as long as your pulse-shaping filter meets this one
criterion: all of the pulses must add up to zero at every multiple of our
symbol period T, except for one of the pulses.
```

```
num_symbols = 10
sps = 8 # 8 samples per symbol
bits = np.random.randint(0, 2, num_symbols) # Our data to be transmitted, 1's
and 0's
x = np.array([])

for bit in bits:
    pulse = np.zeros(sps)
    pulse[0] = bit*2-1 # Set the first value to either a 1 or -1
    x = np.concatenate((x, pulse)) # Add the 8 samples to the signal

plt.figure(); plt.plot(x, '.-')
plt.title("Pulse Train of Impulses")
plt.grid(True); plt.show()

"""
bits: [0, 1, 1, 1, 1, 0, 0, 0, 1, 1]
BPSK symbols: [-1, 1, 1, 1, 1, -1, -1, -1, 1, 1]
Applying 8 samples per symbol: [-1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, ...]
"""

# Create our raised-cosine filter: -
beta = [0, 0.3, 0.6, 0.9] # Range of β values
for i in range(len(beta)):

    Ts = sps # Assume the sample rate is 1 Hz, so the sample period is 1, and
the *symbol* period is 8
    t = np.arange(-50, 51) # Remember it's not inclusive of final number
    h = 1/Ts*np.sinc(t/Ts) * np.cos(np.pi*beta[i]*t/Ts) / (1 -
(2*beta[i]*t/Ts)**2)

    plot_name = "For β = " + str(beta[i])
    plt.figure(); plt.subplot(1,3,1)
    plt.xlabel("Time (t)"); plt.ylabel("Impulse Response, h(t)")
    plt.plot(t, h, '.'); plt.grid(True)
    plt.title(str(plot_name) + "\n" + "in Time Domain")

    DFT(h, plot_name)

    """

    from scipy.fft import fft # Compute the 1-D discrete Fourier Transform
    y = fft(x); plt.subplot(1,3,2)

    plt.xlabel("Frequency (f)"); plt.ylabel("Freqeuncy Response, H(f)")
    plt.title(str(plot_name) + "\n" + "Fast Fourier Transform")
    plt.stem(np.arange(0, len(y)), y); plt.grid(True)
    from scipy.fft import ifft # Compute the 1-D inverse discrete Fourier
Transform
    yinv = ifft(y); plt.subplot(1,3,3)

    plt.xlabel("Frequency (f)"); plt.ylabel("Freqeuncy Response, H(f)")
    plt.title(str(plot_name) + "\n" + "Inverse Fast Fourier Transform")
    plt.stem(np.arange(0, len(yinv)), yinv)
    plt.grid(True); plt.tight_layout(); plt.show()

    """
```

RESULTS:

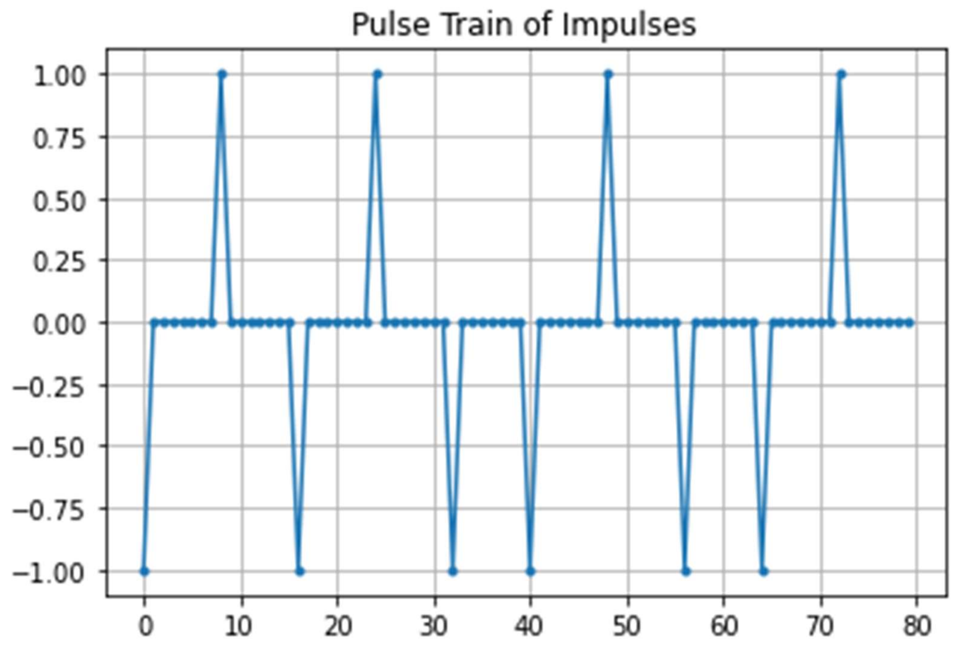


Figure 1 – Pulse Train of Impulses

- Our data to be transmitted, 1s and 0s.
- Set the first value to either a 1 or a -1.
- Add the 8 samples to the signal.

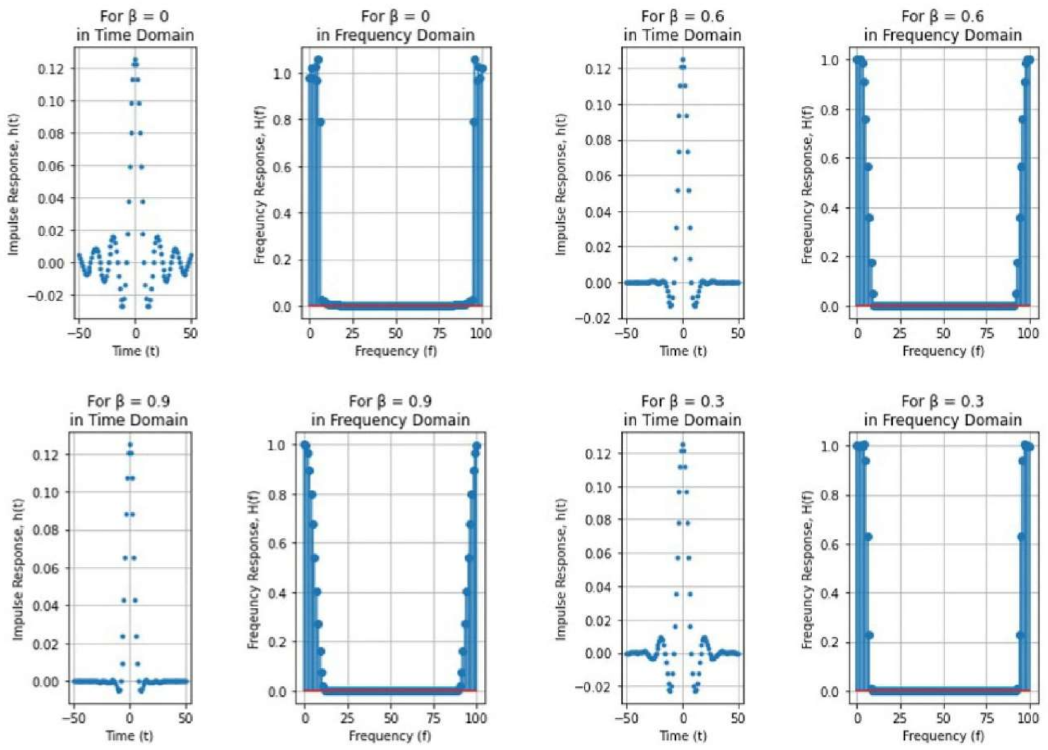


Figure 2 – Raised-Cosine Spectrum Outputs for Various Values of β

INFERENCE:

Thus, implemented a raised-cosine filter using Python and studied its properties. All the simulation results were verified successfully.

REFERENCES:

1. PySDR: A Guide to SDR and DSP using Python – https://pysdr.org/content/pulse_shaping.html
2. Raised-Cosine Filter. From Wikipedia, the Free Encyclopedia.
3. Digital Communications, John G. Proakis, Masoud Salehi, Fifth Edition, McGraw-Hill Higher Education, 2008. ISBN-10: 0072957166. ISBN-13: 978-0072957167.