Communication Systems, January 2024

Dr.Pakravan

Roja Atashkar 400100543

```
import matplotlib.pyplot as plt
import numpy as np
import scipy
```

Implementation of blocks individually

2.1

To simulate a real-time communication system, it's important to consider the concept of interleaving. Interleaving is a process that spreads out consecutive bits in a sequence to mitigate the effects of burst errors that may occur in the transmission channel. By interleaving the bits, errors affecting consecutive bits are spread apart, making it easier to recover the original information at the receiver.

```
def Divide(sequence):
    length = len(sequence)
    if length % 2 != 0:
        raise ValueError("Input sequence length must be even.")

    half_length = length // 2
    sequence1 = []
    sequence2 = []
    for i in range(half_length):
        sequence1.append(sequence[2*i])
        sequence2.append(sequence[2*i + 1])

    return sequence1, sequence2
```

```
def Combine(sequence1, sequence2):
    if len(sequence1) != len(sequence2):
        raise ValueError("Input sequences must have the same length.")

    combined_sequence = []
    for i in range(len(sequence1)):
        combined_sequence.append(sequence1[i])
        combined_sequence.append(sequence2[i])

    return combined_sequence
```

2.2

```
def PulseShaping(sequence, zero_pulse, one_pulse):
    if len(zero_pulse) != len(one_pulse):
        raise ValueError("Length of zero_pulse and one_pulse must be equal.")

    pulse_length = len(zero_pulse)
    waveform = np.zeros(len(sequence) * pulse_length)

    for i, bit in enumerate(sequence):
        if bit == 0:
            waveform[i * pulse_length : (i + 1) * pulse_length] += zero_pulse
        elif bit == 1:
            waveform[i * pulse_length : (i + 1) * pulse_length] += one_pulse
        else:
            raise ValueError("Sequence must contain only zeros and ones.")

    return waveform
```

2.3

AM combines a baseband signal with a carrier signal to generate the modulated signal.

```
def AnalogMod(baseband_signal_1, baseband_signal_2, sampling_frequency, carrier_frequency):
    time = np.arange(len(baseband_signal_1)) / sampling_frequency

    carrier_signal_1 = np.cos(2 * np.pi * carrier_frequency * time)
    carrier_signal_2 = np.sin(2 * np.pi * carrier_frequency * time)
    modulated_signal = baseband_signal_1 * carrier_signal_1 + baseband_signal_2*carrier_signal_2

    return modulated_signal
```

2.4

```
def Channel(signal, sampling_frequency, center_frequency, bandwidth):
    nyquist_frequency = sampling_frequency / 2.0
    low_cutoff = center_frequency - bandwidth / 2.0
    high_cutoff = center_frequency + bandwidth / 2.0

    # Calculate filter coefficients
    b, a = scipy.signal.butter(4, [low_cutoff / nyquist_frequency, high_cutoff / nyquist_frequency], btype='band')

    # Apply the filter to the signal
    filtered_signal = scipy.signal.lfilter(b, a, signal)

    return filtered_signal
```

2.5

```
def AnalogDemod(received_signal, sampling_frequency, signal_bandwidth, carrier_frequency):
    time = np.arange(len(received_signal)) / sampling_frequency


    carrier_signal_1 = np.cos(2 * np.pi * carrier_frequency * time)
    carrier_signal_2 = np.sin(2 * np.pi * carrier_frequency * time)

    demodulated_signal1 = received_signal * carrier_signal_1
    demodulated_signal2 = received_signal * carrier_signal_2

    # Design a low-pass filter
    nyquist_frequency = sampling_frequency / 2.0
    cutoff_frequency = signal_bandwidth / 2.0
    filter_order = 4
    b, a = scipy.signal.butter(filter_order, cutoff_frequency / nyquist_frequency, btype='low')

    # Apply the low-pass filter to the demodulated signals
    demodulated_signal1 = scipy.signal.lfilter(b, a, demodulated_signal1)
    demodulated_signal2 = scipy.signal.lfilter(b, a, demodulated_signal2)

    return demodulated_signal1, demodulated_signal2
```

2.6

```
def MatchedFilter(demodulated_signal, pulse_shape_zero, pulse_shape_one):
    # Matched filter for pulse shape corresponding to zero
    matched_filter_output_zero = scipy.signal.correlate(demodulated_signal, pulse_shape_zero, mode='same')

    # Matched filter for pulse shape corresponding to one
    matched_filter_output_one = scipy.signal.correlate(demodulated_signal, pulse_shape_one, mode='same')

    # Estimate bit values
    estimated_sequence = np.where(matched_filter_output_one > matched_filter_output_zero, 1, 0)

    return  matched_filter_output_one, matched_filter_output_zero, estimated_sequence
```

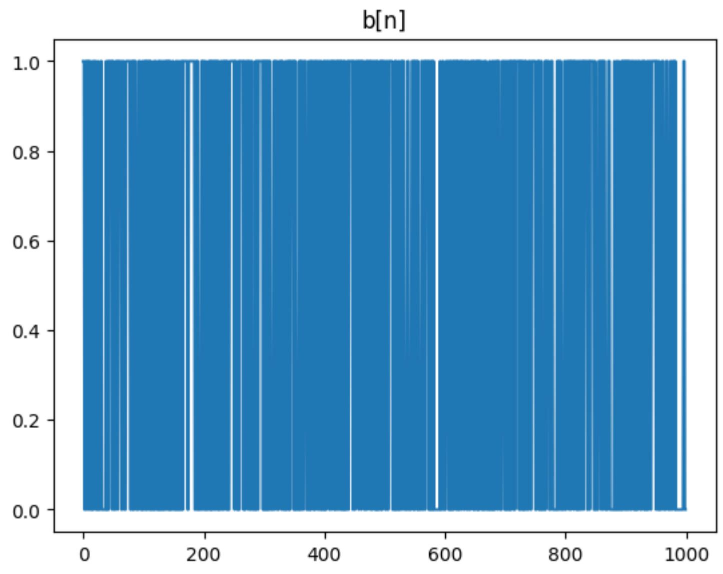
```
def sample_and_quantize(pulse, estimates_sequence):
    l_pulse = len(pulse)
    l = int(len(estimates_sequence)/l_pulse)
    estimated_bits = np.zeros(l)
    step = int(l_pulse/2)
    for i in range(l):
        if estimates_sequence[step + i*l_pulse] > 0:
            estimated_bits[i] = 1
    return estimated_bits
```

Transmission of a random sequence of zeros and ones

```
sampling_frequency = 1e6
carrier_frequency = 10e3
center_frequency = 10e3
pulse_width = 10e-3
channel_bandwidth = 1e3
```

3.1.1

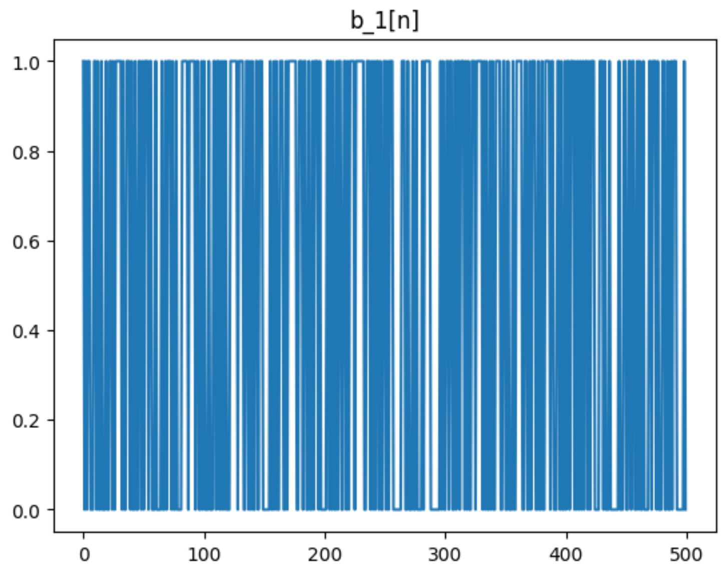
```
import numpy as np
length = 1000
sequence = np.random.randint(2, size=length)
plt.plot(sequence)
plt.title("b[n]")
plt.show()
```

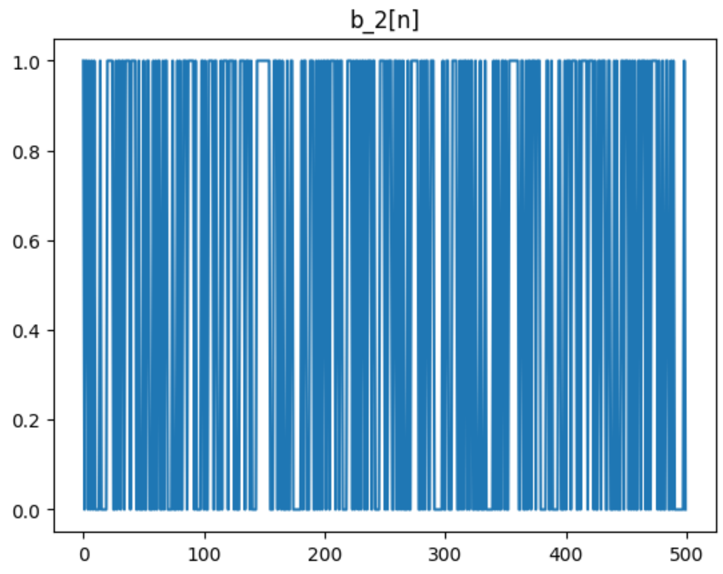


The bandwidth is determined by finding the index where the normalized spectrum drops below a certain threshold. In this case, the threshold is set to 0.05.

```
# finding message signals bandwidth
def calculate_bandwidth(signal, sampling_frequency):
    spectrum = np.fft.fft(signal)
    magnitude_spectrum = np.abs(spectrum)
    half_length = len(signal) // 2
    normalized_spectrum = magnitude_spectrum[:half_length] / np.max(magnitude_spectrum)
    frequency_axis = np.fft.fftfreq(len(signal), 1 / sampling_frequency)[:half_length]

    # Find the index where the spectrum drops below a certain threshold
    threshold = 0.05  # Adjust this threshold as per your requirements
    bandwidth_index = np.where(normalized_spectrum < threshold)[0][0]

    bandwidth = 2 * frequency_axis[bandwidth_index]
    return bandwidth
```

```
# Transmitter
# Divide Block
b_1 , b_2 = Divide(sequence)
plt.plot(b_1)
plt.title("b_1[n]")
plt.show()
plt.plot(b_2)
plt.title("b_2[n]")
plt.show()
```



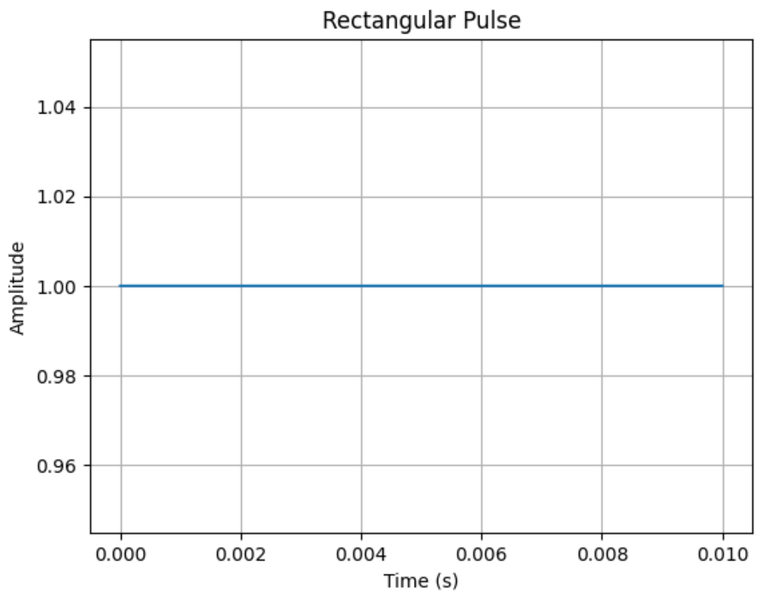


```
bandwidth_1 = calculate_bandwidth(b_1, sampling_frequency)
print("Bandwidth_1:", bandwidth_1)
bandwidth_2 = calculate_bandwidth(b_2, sampling_frequency)
print("Bandwidth_2:", bandwidth_2)
```

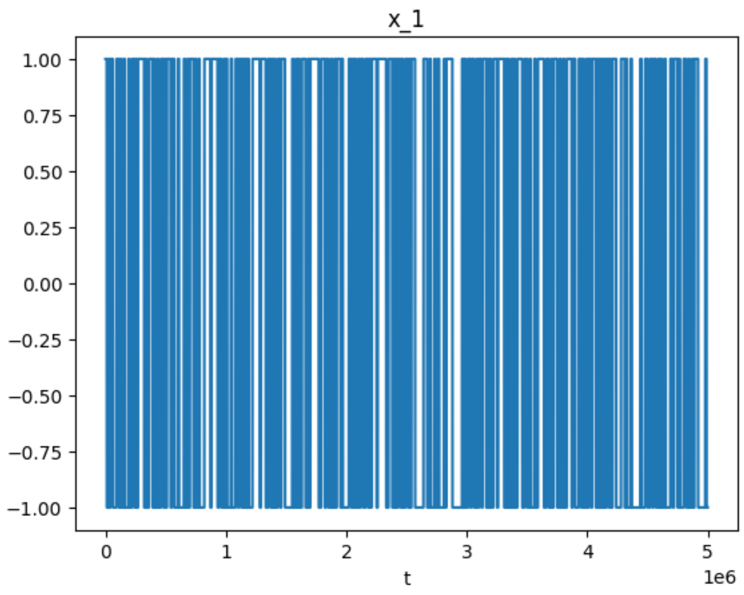
```
Bandwidth_1: 4000.0
Bandwidth_2: 4000.0
```

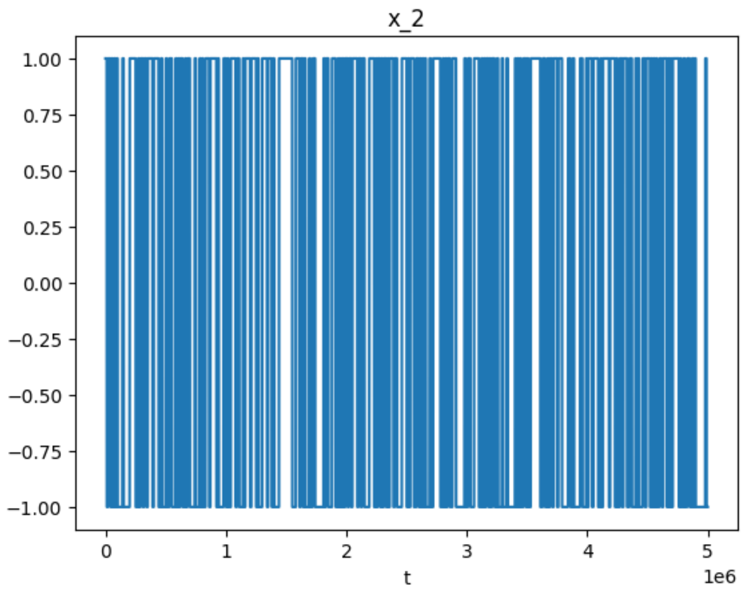
```
signal_bandwidth = max(bandwidth_1, bandwidth_2)
```

```
# Pulse shaping block
def generate_rectangular_pulse(duration, amplitude, sampling_frequency):
    num_samples = int(duration * sampling_frequency)
    pulse = np.ones(num_samples) * amplitude
    return pulse
amp = 1
pulse = generate_rectangular_pulse(pulse_width, 1, sampling_frequency)

# Plotting the pulse
time = np.arange(0, pulse_width, 1 / sampling_frequency)
plt.plot(time, pulse)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Rectangular Pulse')
plt.grid(True)
plt.show()
```

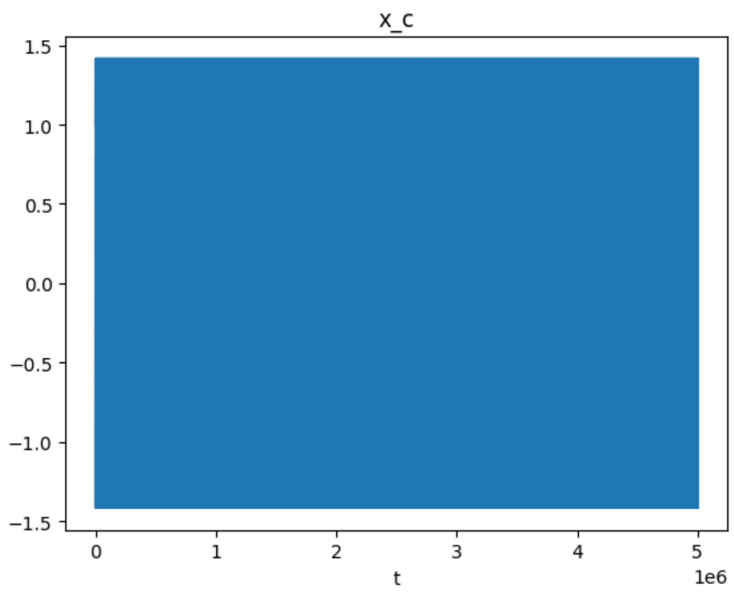


```
x_1 = PulseShaping(b_1, -1*pulse, pulse)
x_2 = PulseShaping(b_2, -1*pulse, pulse)
plt.plot(x_1)
plt.title("x_1")
plt.xlabel("t")
plt.show()
plt.plot(x_2)
plt.title("x_2")
plt.xlabel("t")
plt.show()
```

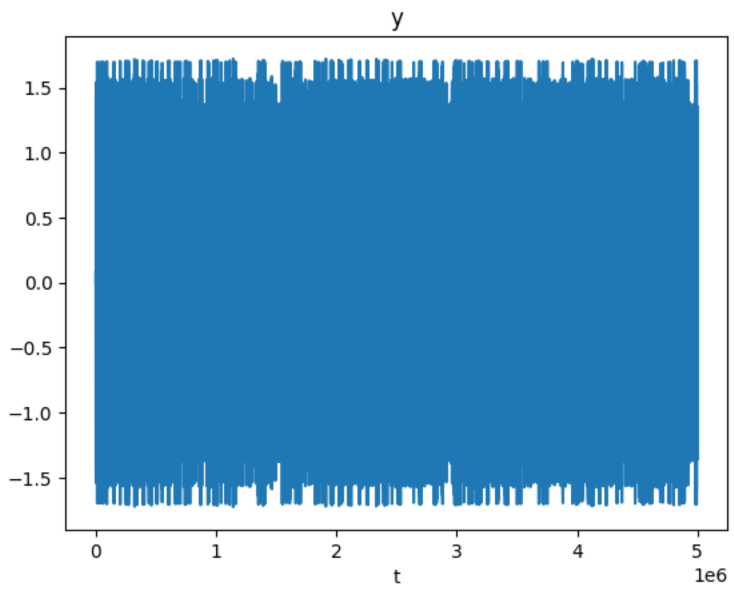




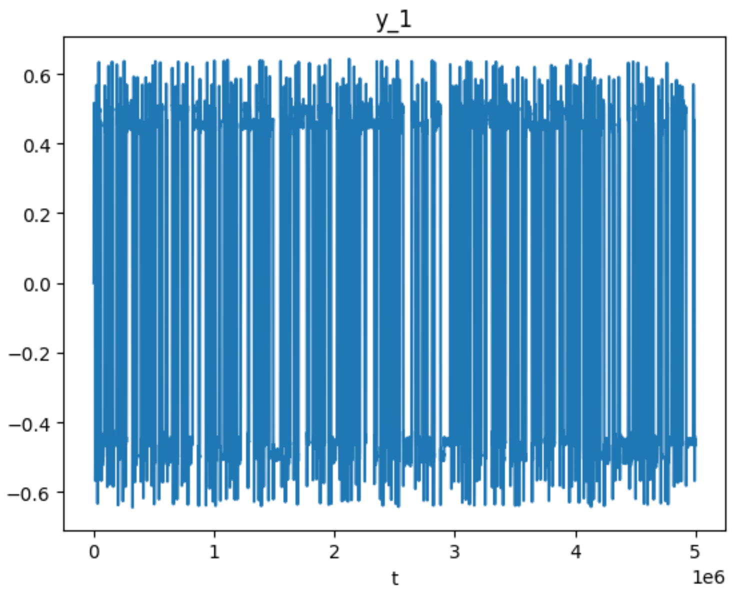
```
# modulation Block
x_c = AnalogMod(x_1, x_2, sampling_frequency, carrier_frequency)
plt.plot(x_c)
plt.title("x_c")
plt.xlabel("t")
plt.show()
```

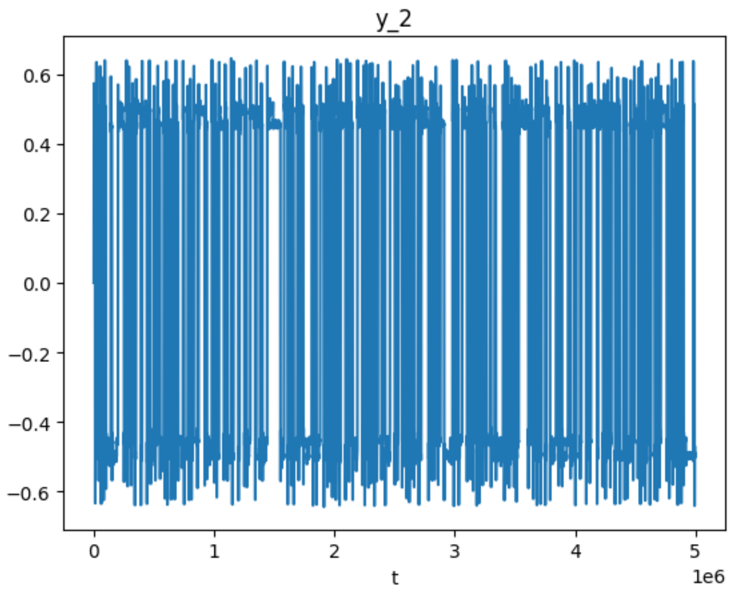


```
# Receiver
# Channel Block
y = Channel(x_c, sampling_frequency,center_frequency,  channel_bandwidth)
plt.plot(y)
plt.title("y")
plt.xlabel("t")
plt.show()
```

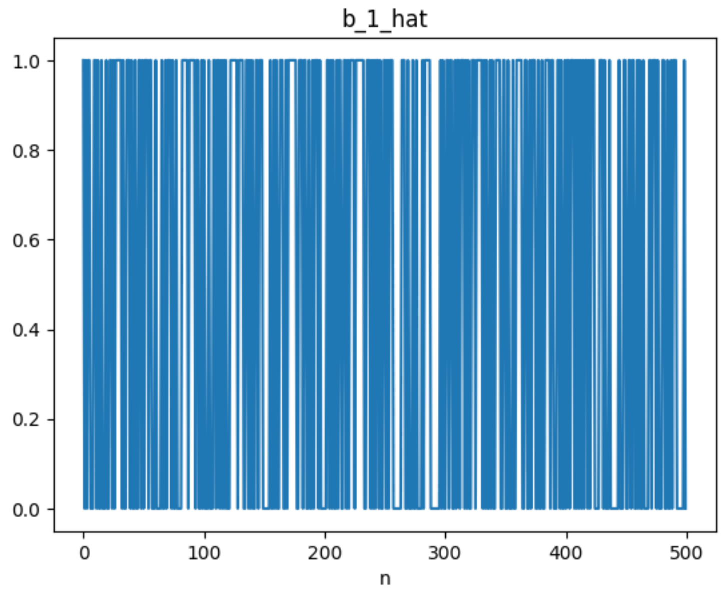


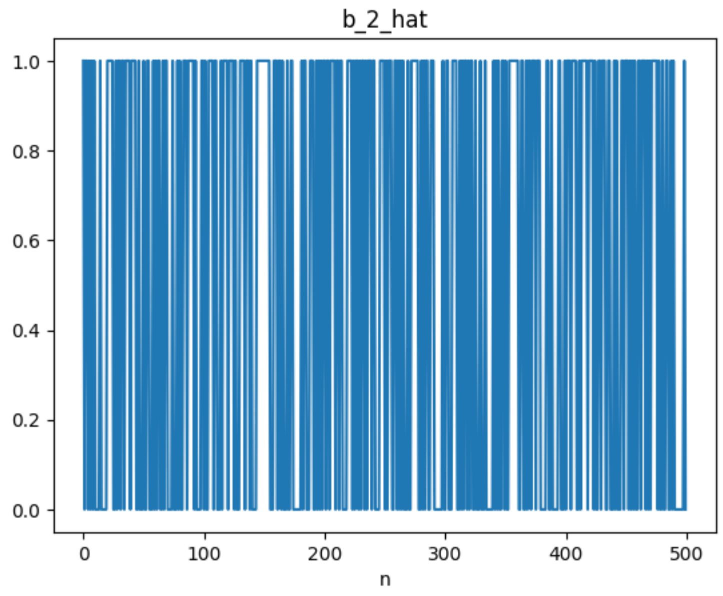
```
# Demodulation block
y_1, y_2 = AnalogDemod(y, sampling_frequency, signal_bandwidth, carrier_frequency)
plt.plot(y_1)
plt.title("y_1")
plt.xlabel("t")
plt.show()
plt.plot(y_2)
plt.title("y_2")
plt.xlabel("t")
plt.show()
```





```
estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
plt.plot(estimated_bit_1)
plt.title("b_1_hat")
plt.xlabel("n")
plt.show()

plt.plot(estimated_bit_2)
plt.title("b_2_hat")
plt.xlabel("n")
plt.show()
```





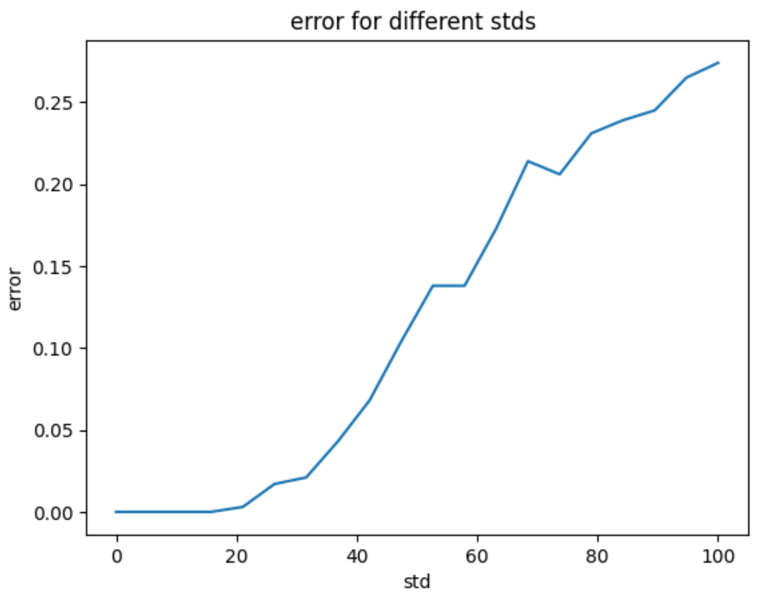
```
b = Combine(estimated_bit_1, estimated_bit_2)
print("bit error rate:" )
print(np.sum(b != sequence)/len(sequence))
```

```
bit error rate:
0.0
```

3.1.2

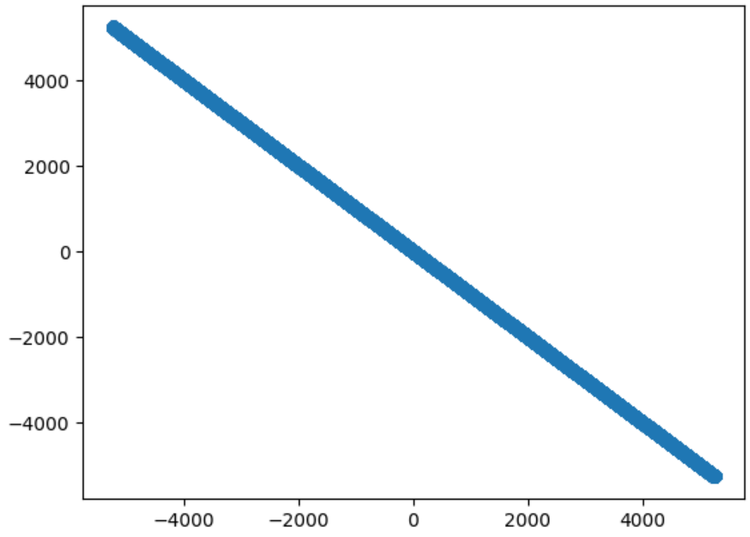
```
mean = 0
num_samples = len(y)
stds = np.linspace(0, 100, 20)
errors = []
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    b = Combine(estimated_bit_1, estimated_bit_2)
    errors.append(np.sum(b != sequence)/len(sequence))
plt.plot(stds, errors)
plt.title("error for different stds ")
plt.xlabel("std")
plt.ylabel("error")
```

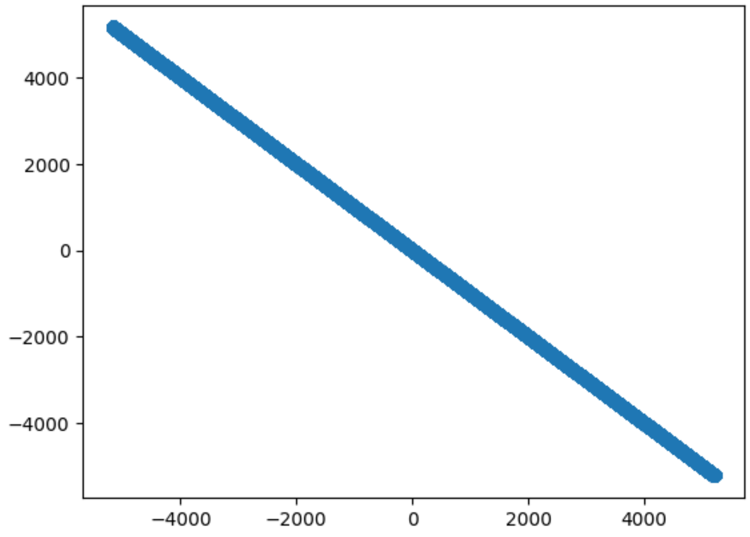
```
Text(0, 0.5, 'error')
```

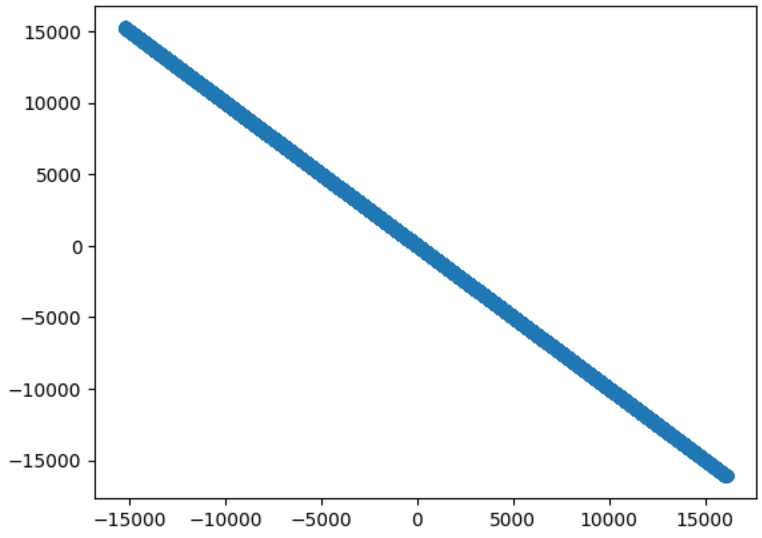


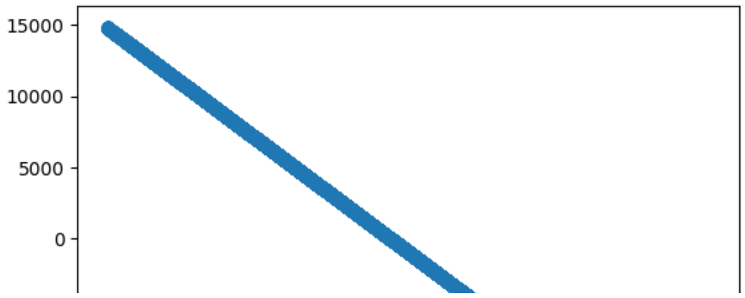
3.1.3

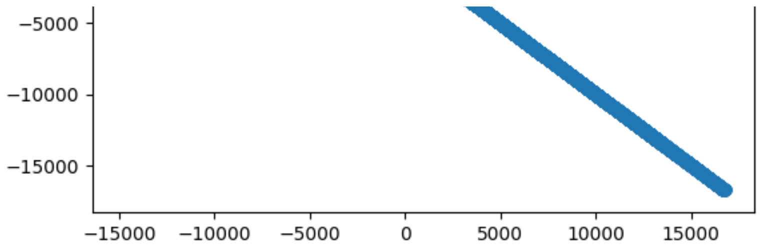
```
mean = 0
stds = [1, 50, 100]
num_samples = len(y)
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    plt.scatter(matched_filter_output_zero_1, matched_filter_output_one_1)
    plt.show()
    plt.scatter(matched_filter_output_zero_2, matched_filter_output_one_2)
    plt.show()
```

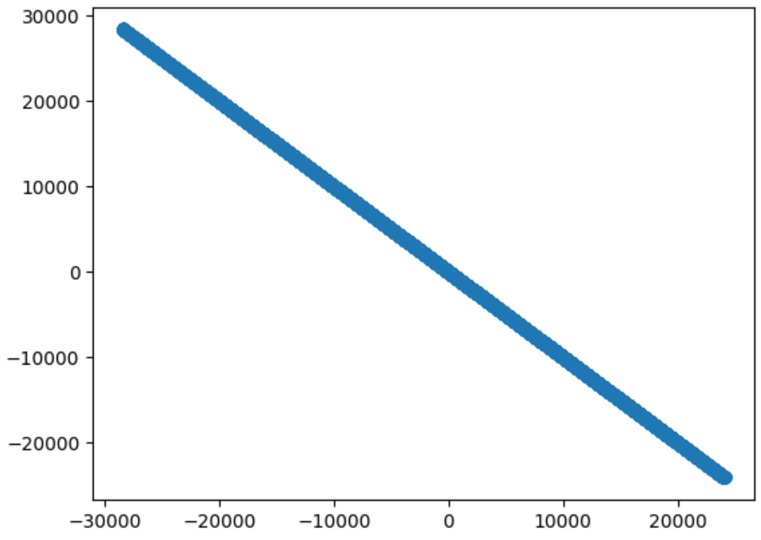

4000
2000
0
−2000
−4000
−4000
−2000
0
2000
4000

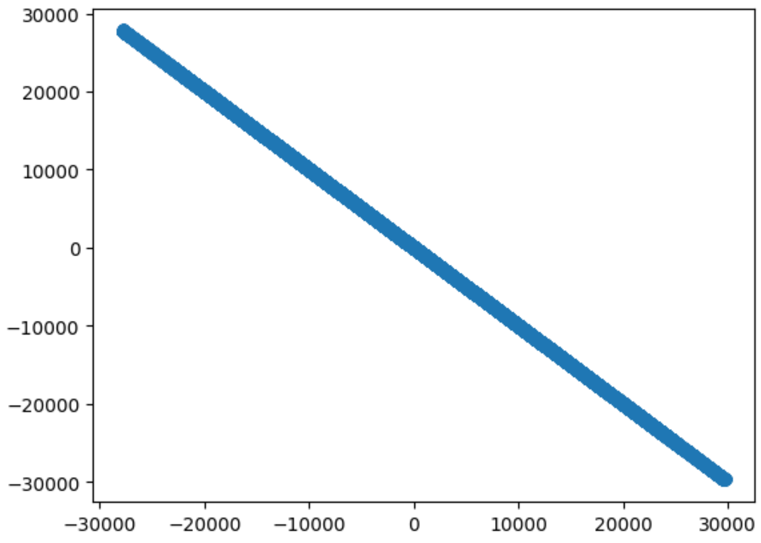

4000
2000
0
−2000
−4000
−4000
−2000
0
2000
4000


15000
10000
5000
0
−5000
−10000
−15000
−15000
−10000
−5000
0
5000
10000
15000


15000
10000
5000
0

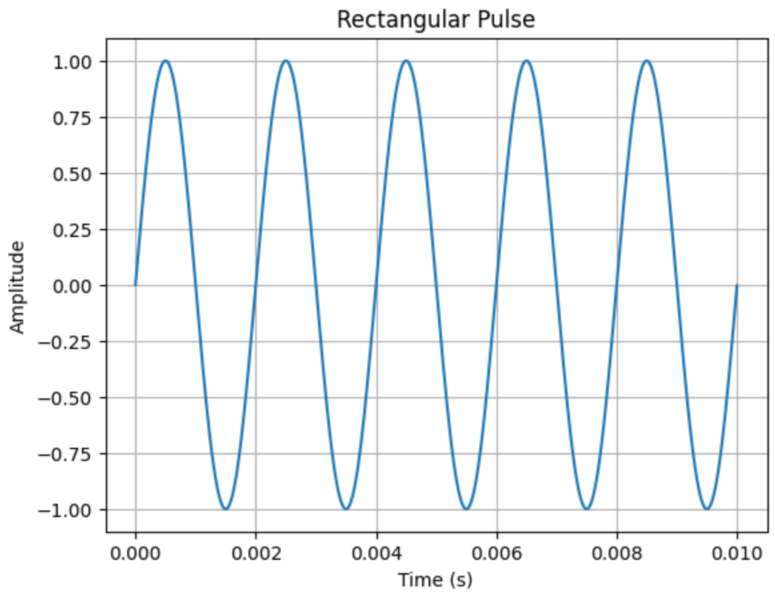

−5000
−10000
−15000
−15000
−10000
−5000
0
5000
10000
15000


30000
20000
10000
0
−10000
−20000
−30000
−20000
−10000
0
10000
20000

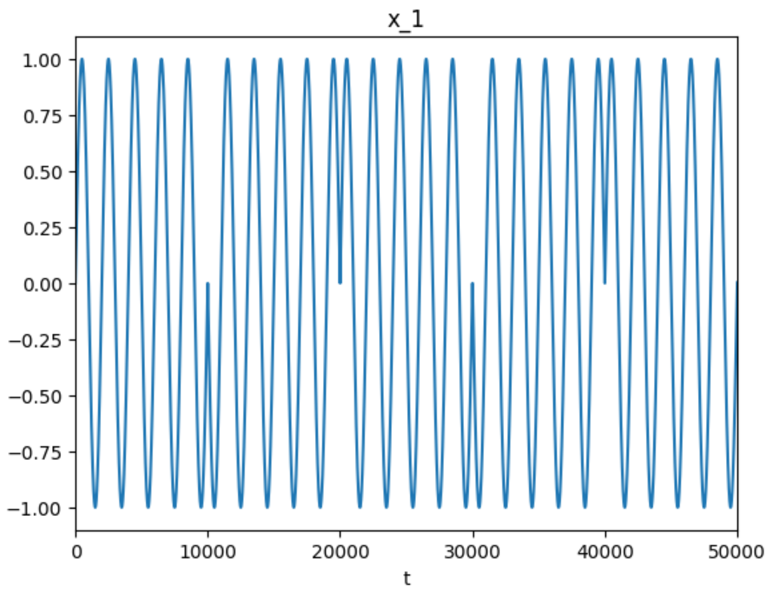

30000
20000
10000
0
−10000
−20000
−30000
−30000
−20000
−10000
0
10000
20000
30000

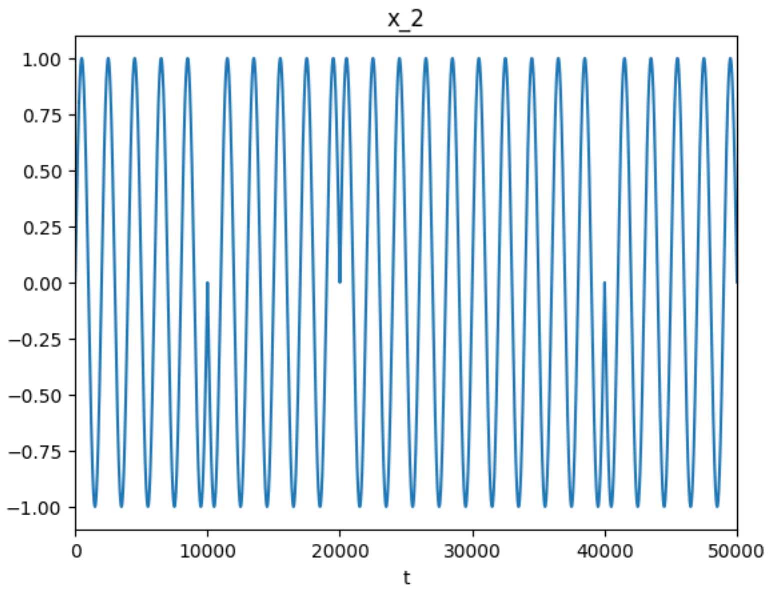
3.2.1

```
# Pulse shaping block
def generate_sinusoidal_pulse(frequency, duration, sampling_rate):
    # Generate the time axis
    t = np.linspace(0, duration, int(duration * sampling_rate), endpoint=False)

    # Generate the sinusoidal pulse
    pulse = np.sin(2 * np.pi * frequency * t)

    return pulse
sin_freq = 500
pulse = generate_sinusoidal_pulse(sin_freq, pulse_width, sampling_frequency)

# Plotting the pulse
time = np.arange(0, pulse_width, 1 / sampling_frequency)
plt.plot(time, pulse)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Rectangular Pulse')
plt.grid(True)
plt.show()
```

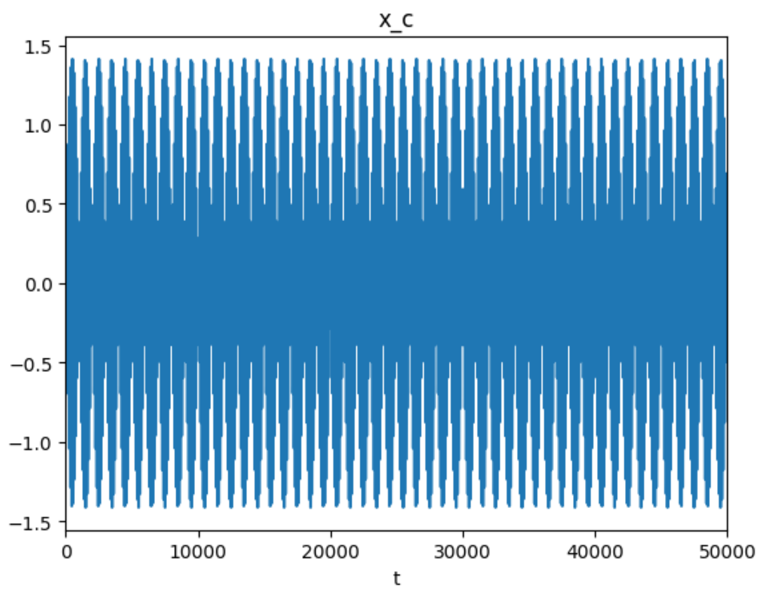


```
x_1 = PulseShaping(b_1, -1*pulse, pulse)
x_2 = PulseShaping(b_2, -1*pulse, pulse)
plt.plot(x_1)
plt.title("x_1")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
plt.plot(x_2)
plt.title("x_2")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
```

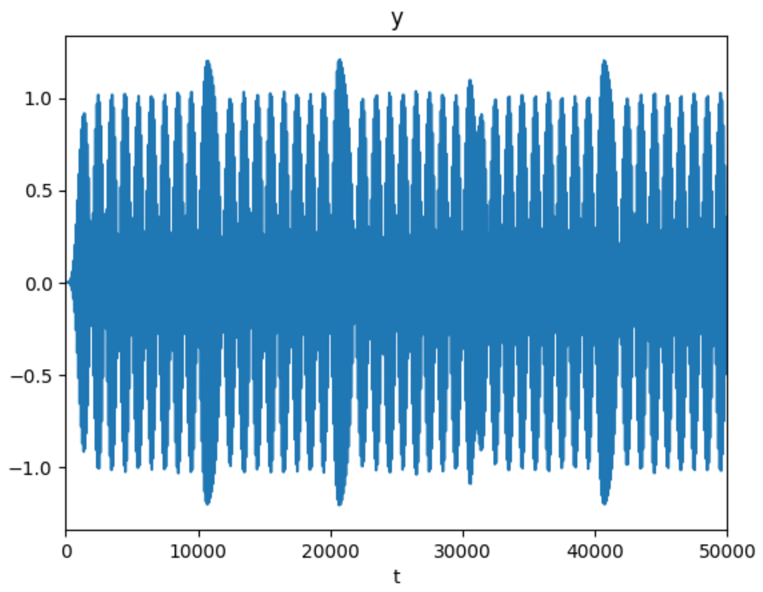




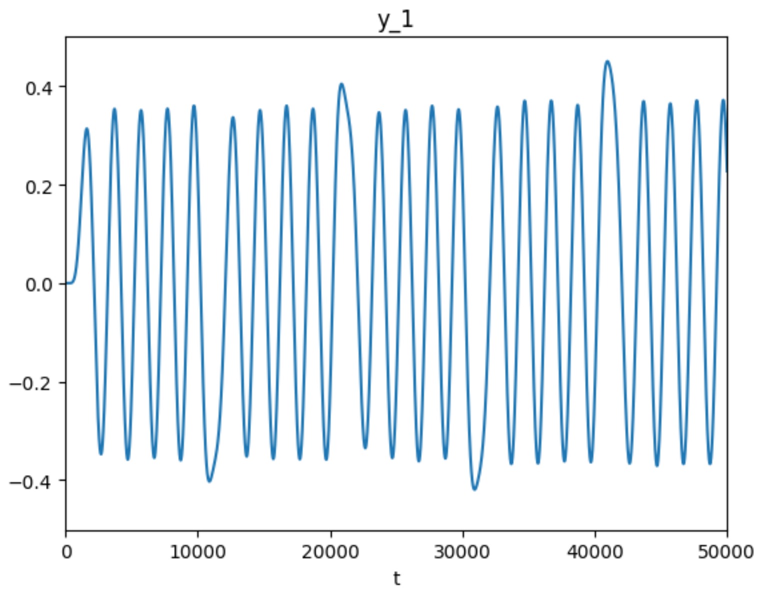
```
# modulation Block
x_c = AnalogMod(x_1, x_2, sampling_frequency, carrier_frequency)
plt.plot(x_c)
plt.title("x_c")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
```

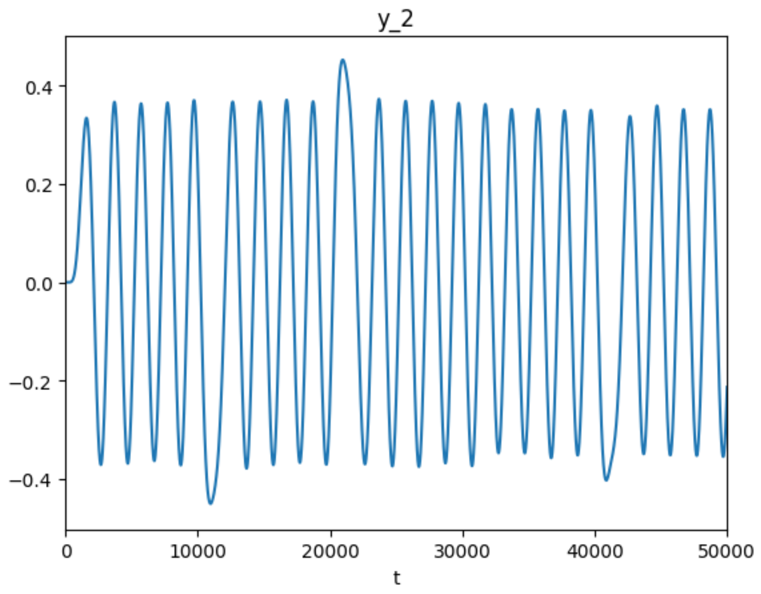


```
plt.show()
# Receiver
# Channel Block
y = Channel(x_c, sampling_frequency, center_frequency, channel_bandwidth)
plt.plot(y)
plt.title("y")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
```

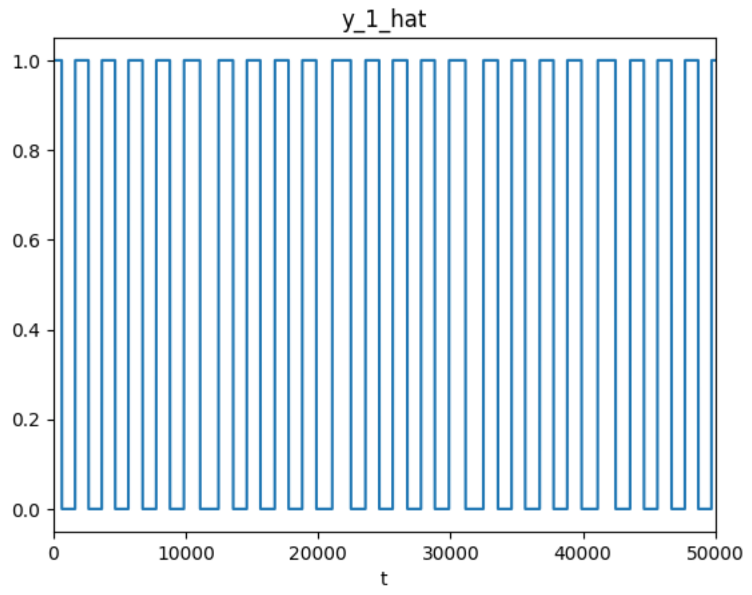


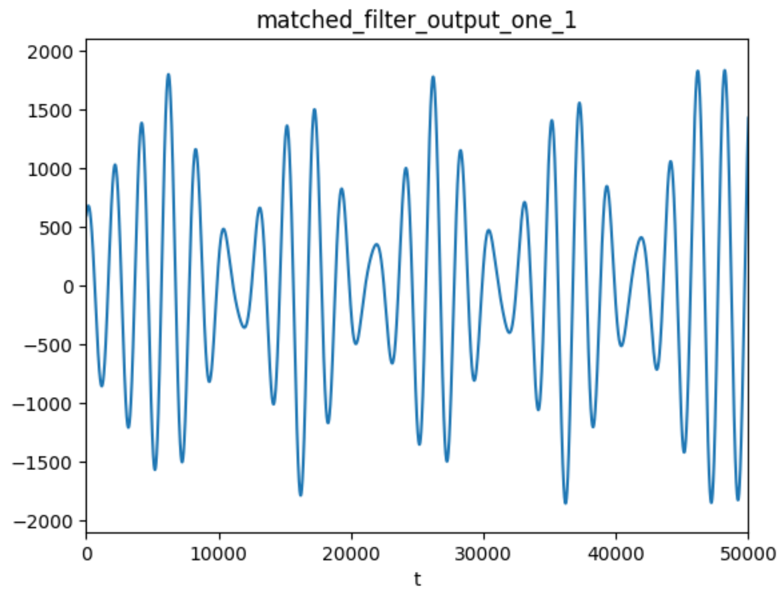
```
# Demodulation block
y_1, y_2 = AnalogDemod(y, sampling_frequency, signal_bandwidth, carrier_frequency)
plt.plot(y_1)
plt.title("y_1")
plt.xlabel("t")
plt.xlim(0, 5e4)
plt.show()
plt.plot(y_2)
plt.title("y_2")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
```

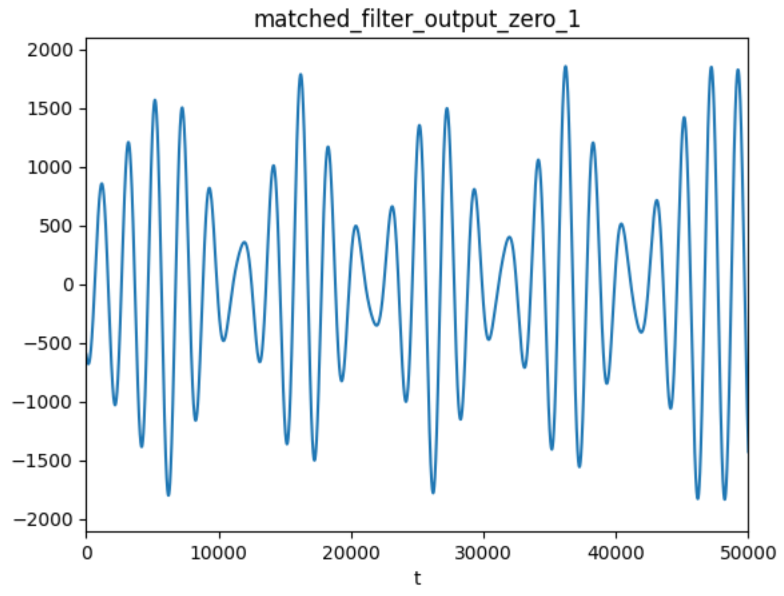

y_1
0.4
0.2
0.0
−0.2
−0.4
0
10000
20000
30000
40000
50000
t

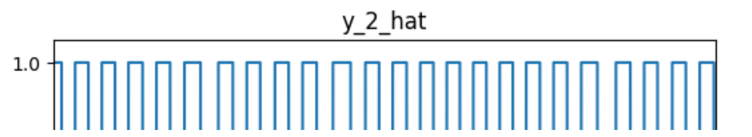

y_2
0.4
0.2
0.0
−0.2
−0.4
0
10000
20000
30000
40000
50000
t

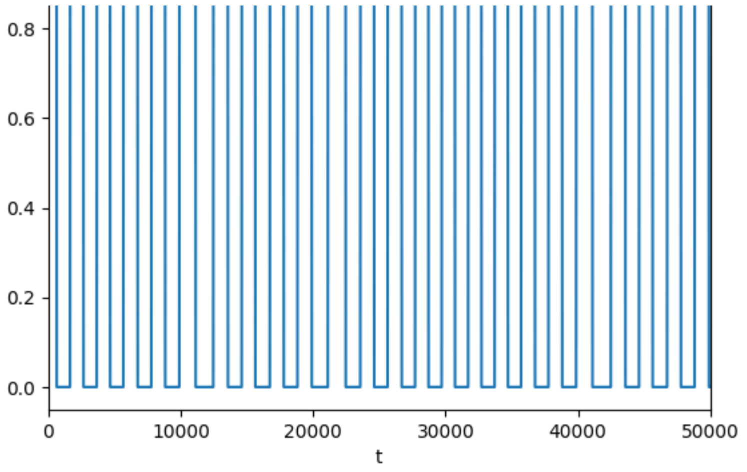
```
# Matched filter and threshold
matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
plt.plot(y_1_hat)
plt.title("y_1_hat")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
plt.plot(matched_filter_output_one_1)
plt.title("matched_filter_output_one_1")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
plt.plot(matched_filter_output_zero_1)
plt.xlim(0, 5e4)
plt.title("matched_filter_output_zero_1")
plt.xlabel("t")
plt.show()

plt.plot(y_2_hat)
plt.xlim(0, 5e4)
plt.title("y_2_hat")
plt.xlabel("t")
plt.show()
plt.plot(matched_filter_output_one_2)
plt.xlim(0, 5e4)
plt.title("matched_filter_output_one_2")
plt.xlabel("t")
plt.show()
plt.plot(matched_filter_output_zero_2)
plt.xlim(0, 5e4)
plt.title("matched_filter_output_zero_2")
plt.xlabel("t")
plt.show()
```

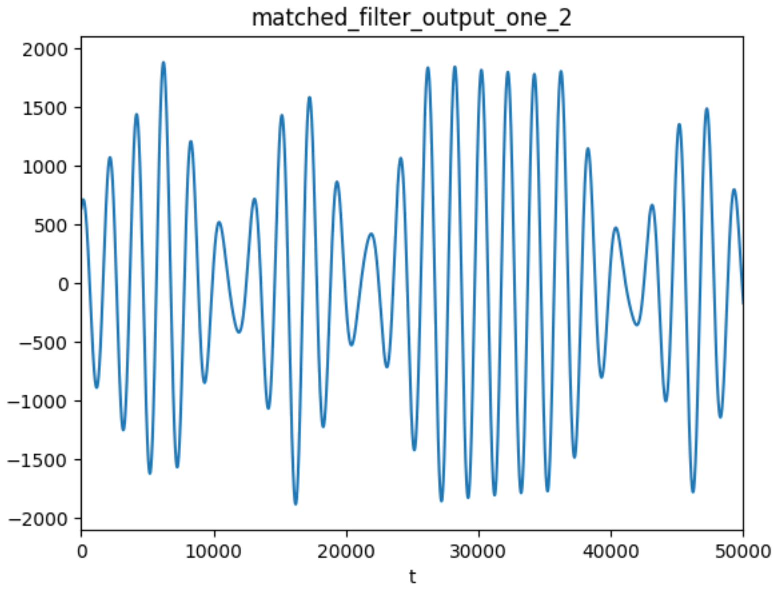

y_1_hat
1.0
0.8
0.6
0.4
0.2
0.0
0
10000
20000
30000
40000
50000
t

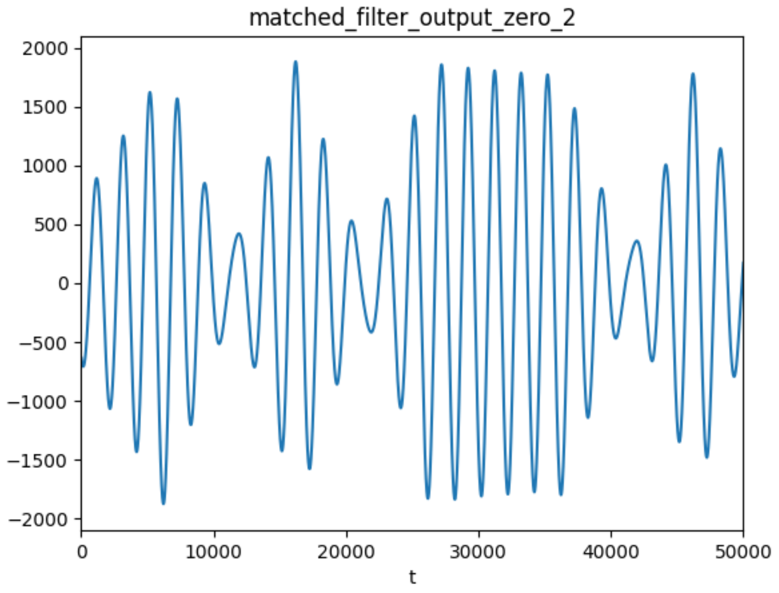

matched_filter_output_one_1
2000
1500
1000
500
0
−500
−1000
−1500
−2000
0
10000
20000
30000
40000
50000
t


matched_filter_output_zero_1
2000
1500
1000
500
0
−500
−1000
−1500
−2000
0
10000
20000
30000
40000
50000
t

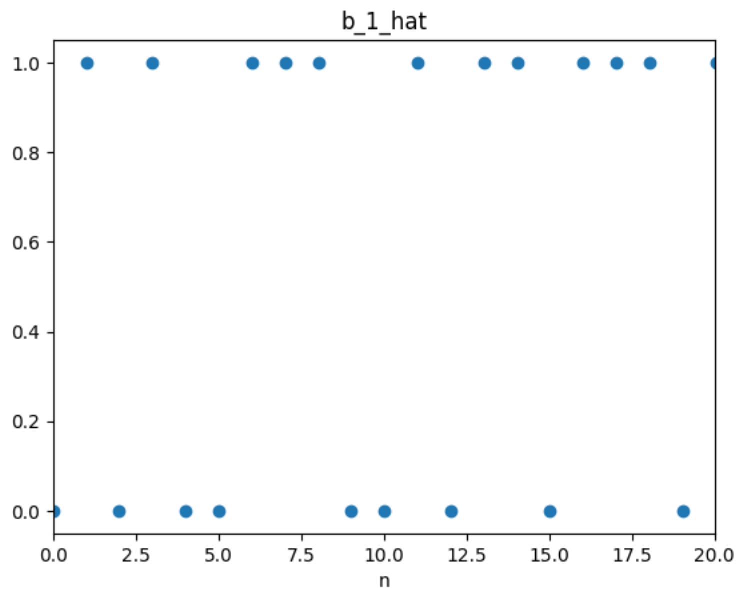

y_2_hat
1.0

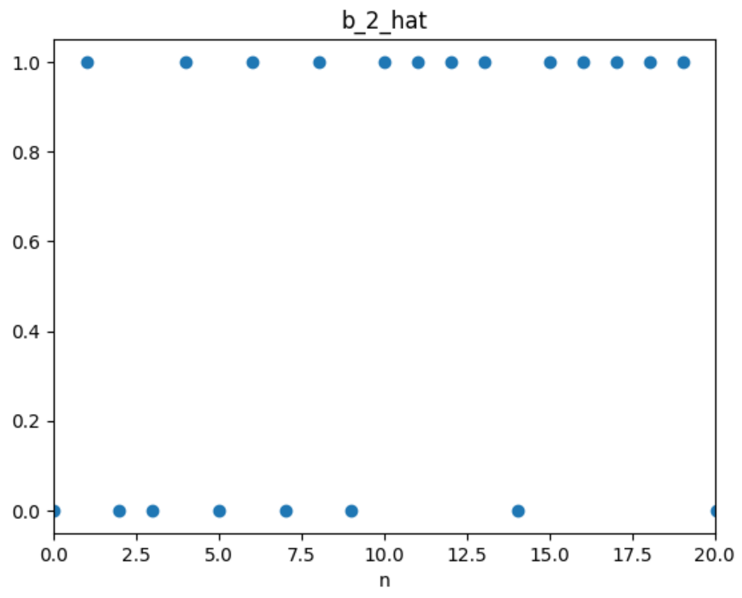

0.8
0.6
0.4
0.2
0.0
0
10000
20000
30000
40000
50000
t


matched_filter_output_one_2
2000
1500
1000
500
0
−500
−1000
−1500
−2000
0
10000
20000
30000
40000
50000
t


matched_filter_output_zero_2
2000
1500
1000
500
0
−500
−1000
−1500
−2000
0
10000
20000
30000
40000
50000
t

```
estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
n = np.linspace(0, 500, len(estimated_bit_1))
plt.scatter(n, estimated_bit_1)
plt.title("b_1_hat")
plt.xlim(0, 20)
plt.xlabel("n")
plt.show()

plt.scatter(n, estimated_bit_2)
plt.xlim(0, 20)
plt.title("b_2_hat")
plt.xlabel("n")
plt.show()
```





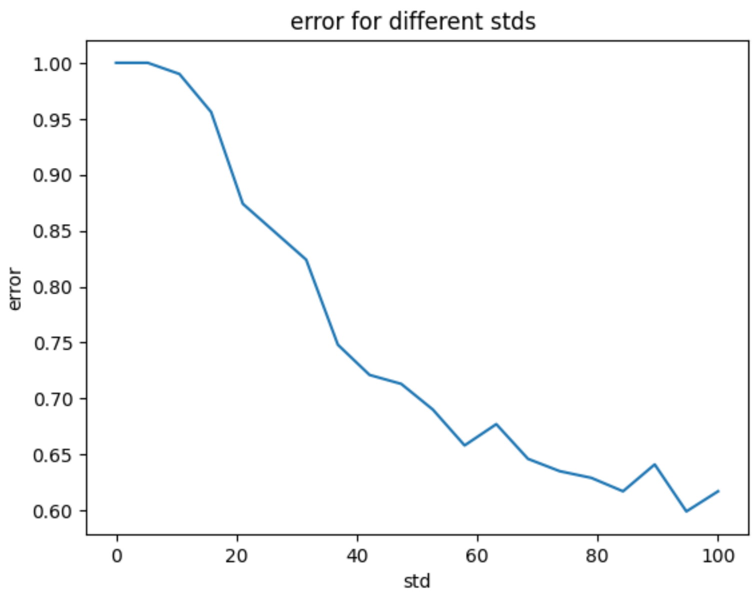
```
b = Combine(estimated_bit_1, estimated_bit_2)
print("bit error rate:" )
print(np.sum(b != sequence)/len(sequence))
```

```
    bit error rate:
    1.0
```

3.2.2

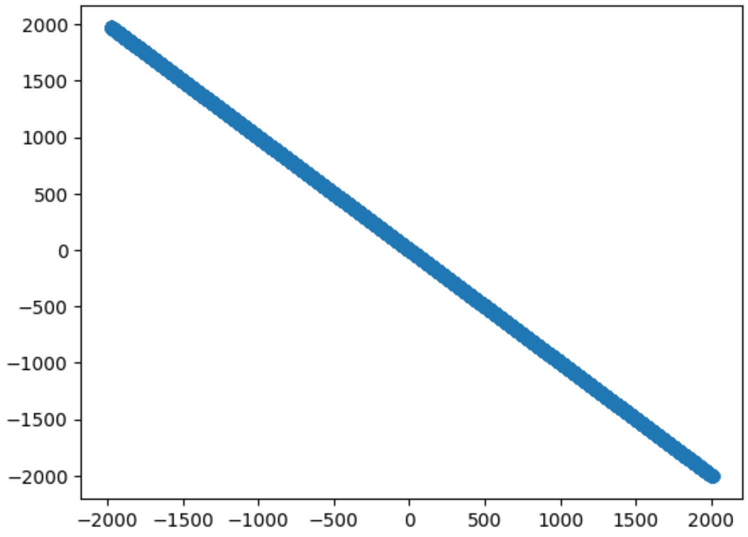
```
mean = 0
num_samples = len(y)
stds = np.linspace(0, 100, 20)
errors = []
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    b = Combine(estimated_bit_1, estimated_bit_2)
    errors.append(np.sum(b != sequence)/len(sequence))
plt.plot(stds, errors)
plt.title("error for different stds ")
plt.xlabel("std")
plt.ylabel("error")
```

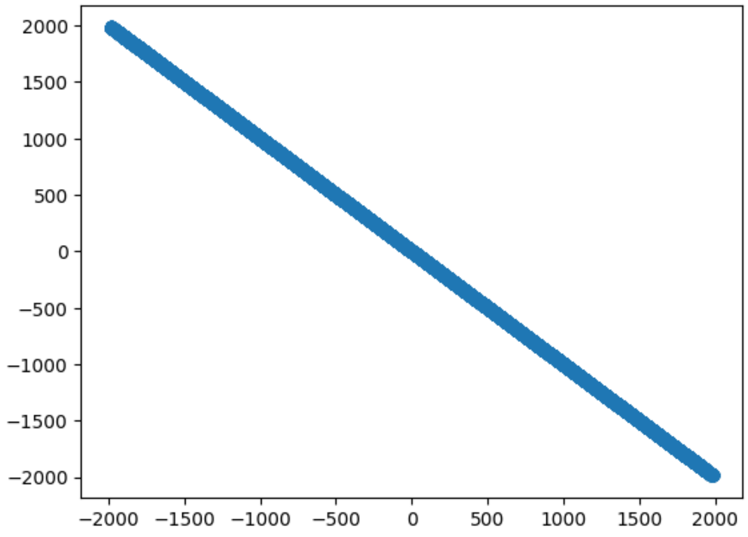
```
Text(0, 0.5, 'error')
```

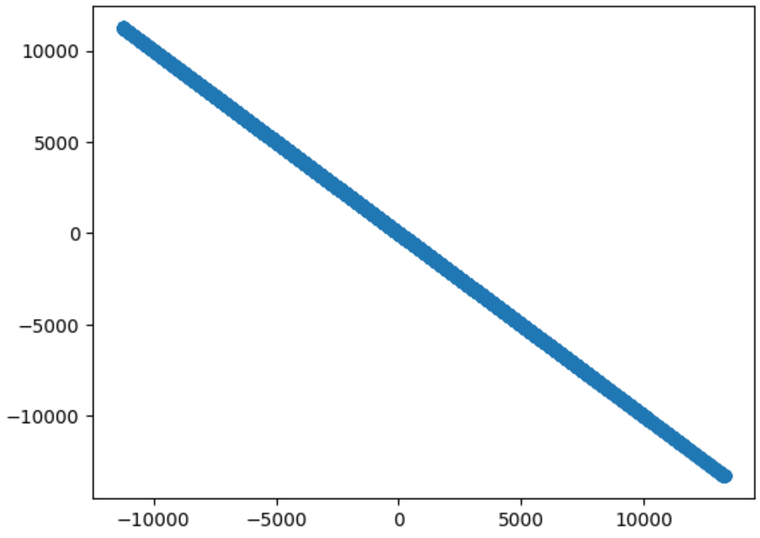


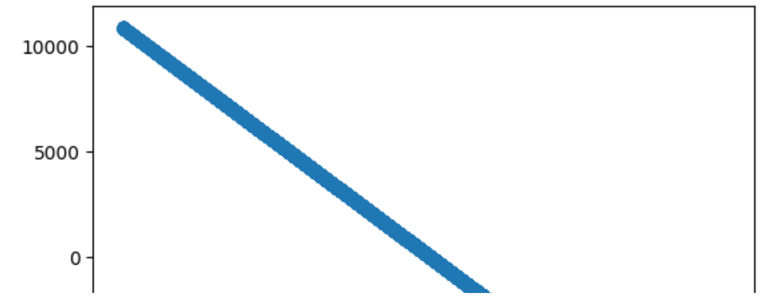
3.2.3

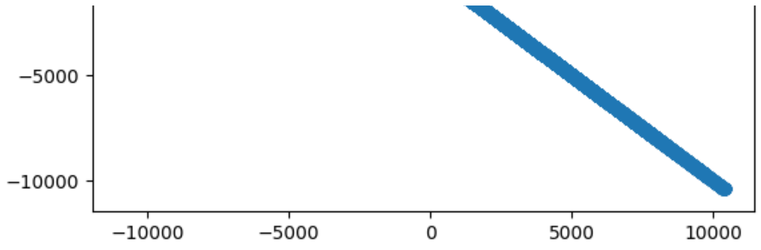
```
mean = 0
stds = [1, 50, 100]
num_samples = len(y)
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    plt.scatter(matched_filter_output_zero_1, matched_filter_output_one_1)
    plt.show()
    plt.scatter(matched_filter_output_zero_2, matched_filter_output_one_2)
    plt.show()
```

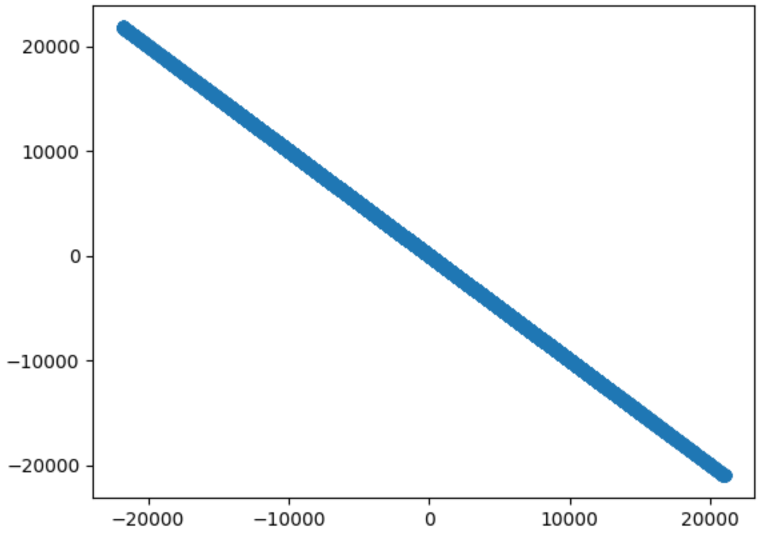

2000
1500
1000
500
0
−500
−1000
−1500
−2000
−2000 −1500 −1000 −500 0 500 1000 1500 2000

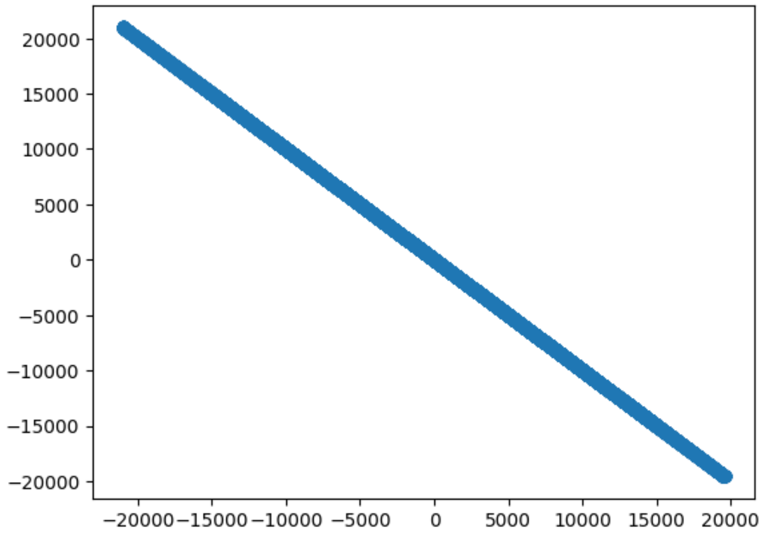

2000
1500
1000
500
0
−500
−1000
−1500
−2000
−2000 −1500 −1000 −500 0 500 1000 1500 2000


10000
5000
0
−5000
−10000
−10000 −5000 0 5000 10000


10000
5000
0







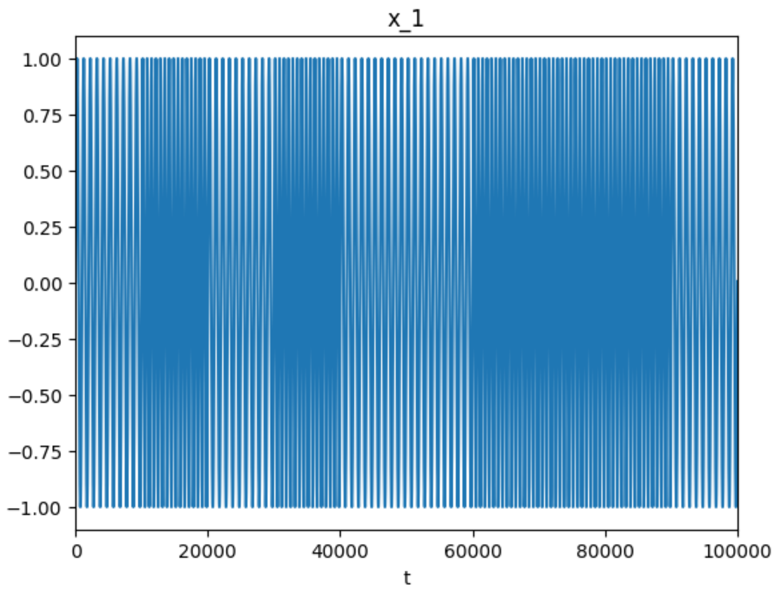
5.3.1

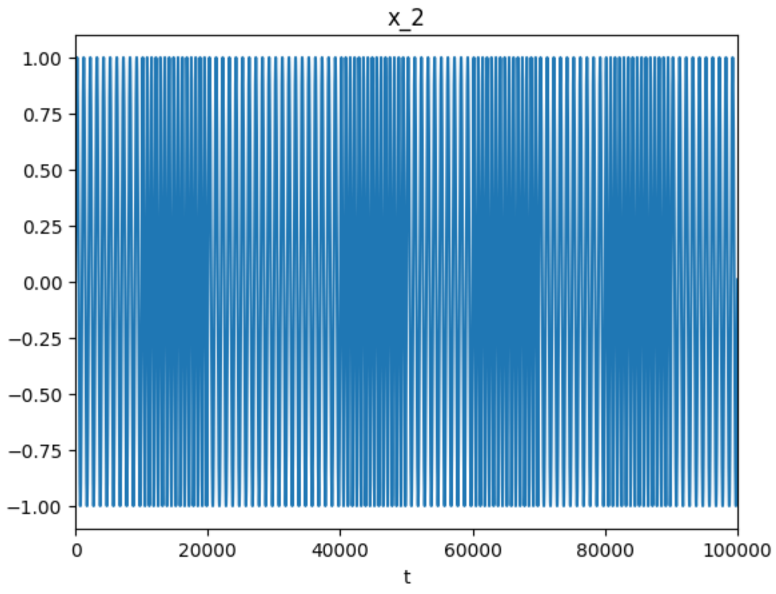
For the frequencies given in the FSK modulation scenario mentioned (1 kHz for bit 1 and 1.5 kHz for bit 0), these frequencies would not inherently form orthogonal signaling waveforms.

5.3.2

```
frequencies = {0: 1500, 1: 1000}  # Frequency mapping for bits

# Generate random bit sequence
num_bits = 1000
bit_sequence = np.random.randint(0, 2, num_bits)
b_1 , b_2 = Divide(sequence)
```

```
x_1 = PulseShaping(b_1, generate_sinusoidal_pulse(frequencies[0], pulse_width, sampling_frequency), generate_sinusoidal_pulse(frequencies[1
x_2 = PulseShaping(b_2, generate_sinusoidal_pulse(frequencies[0], pulse_width, sampling_frequency), generate_sinusoidal_pulse(frequencies[1
plt.plot(x_1)
plt.title("x_1")
plt.xlim(0, 1e5)
plt.xlabel("t")
plt.show()
plt.plot(x_2)
plt.title("x_2")
plt.xlim(0, 1e5)
plt.xlabel("t")
plt.show()
```



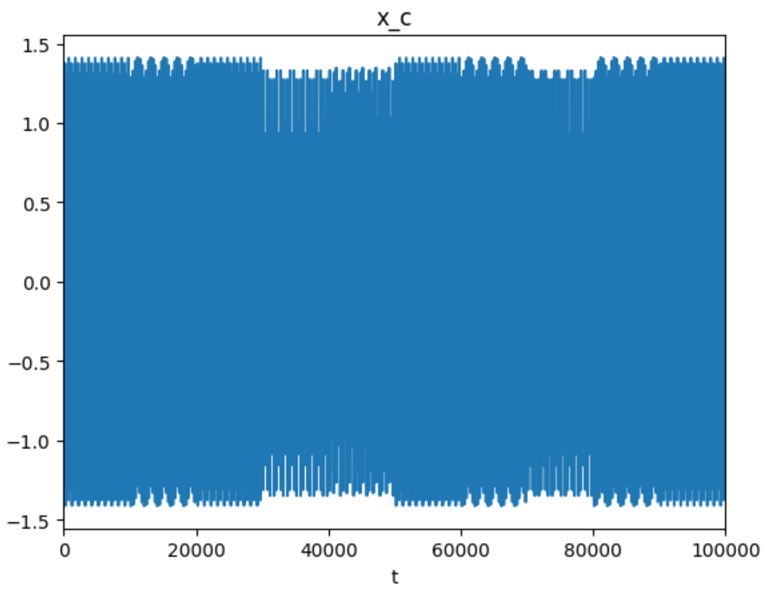


```
bandwidth_1 = calculate_bandwidth(b_1, sampling_frequency)
print("Bandwidth_1:", bandwidth_1)
bandwidth_2 = calculate_bandwidth(b_2, sampling_frequency)
print("Bandwidth_2:", bandwidth_2)
```

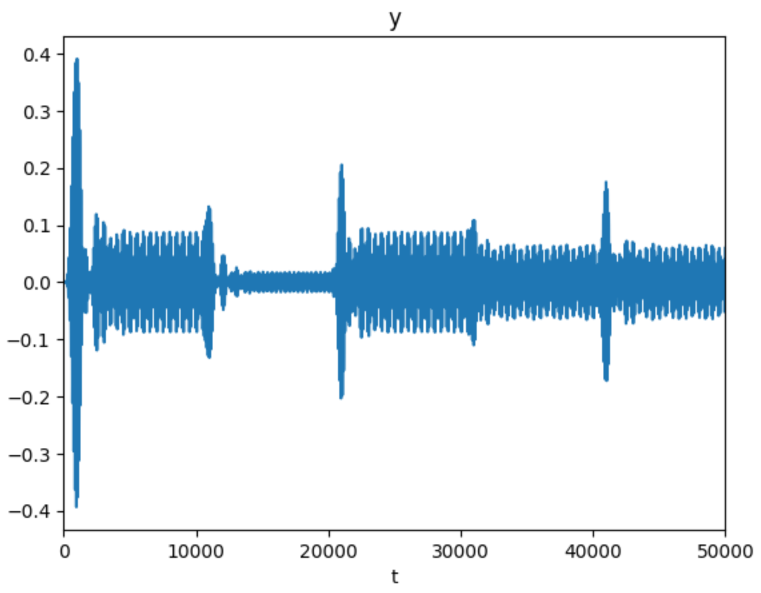
```
Bandwidth_1: 4000.0
Bandwidth_2: 4000.0
```

```
signal_bandwidth = max(bandwidth_1, bandwidth_2)
```

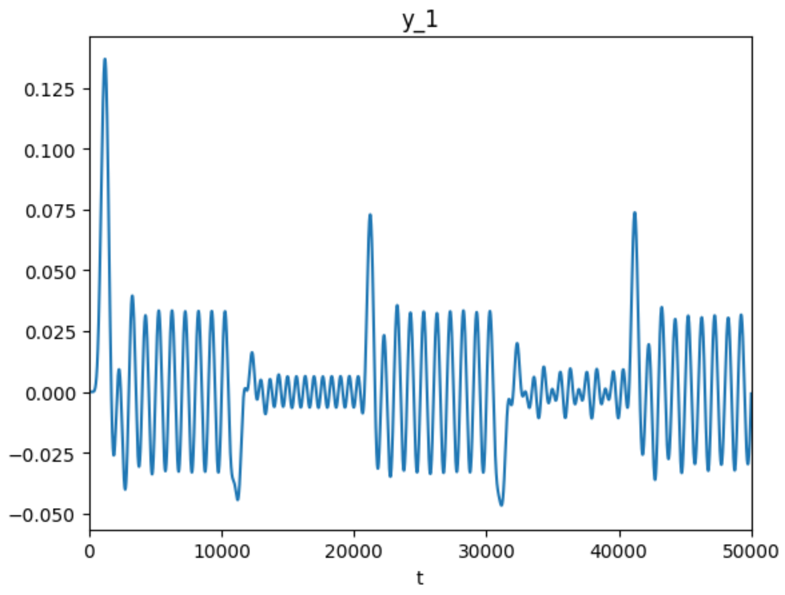
```
# modulation Block
x_c = AnalogMod(x_1, x_2, sampling_frequency, carrier_frequency)
plt.plot(x_c)
plt.title("x_c")
plt.xlabel("t")
plt.xlim(0, 1e5)
plt.show()
```

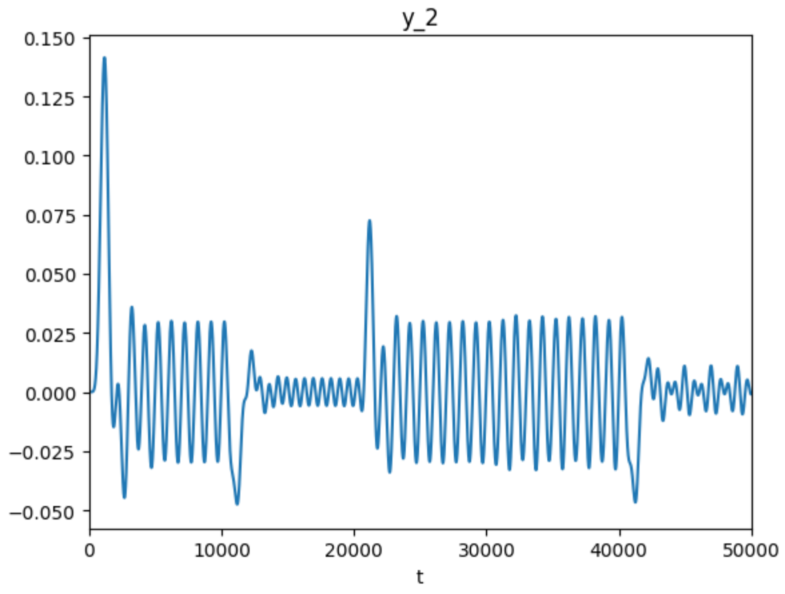


```
# Receiver
# Channel Block
y = Channel(x_c, sampling_frequency, center_frequency, channel_bandwidth)
plt.plot(y)
plt.title("y")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
```



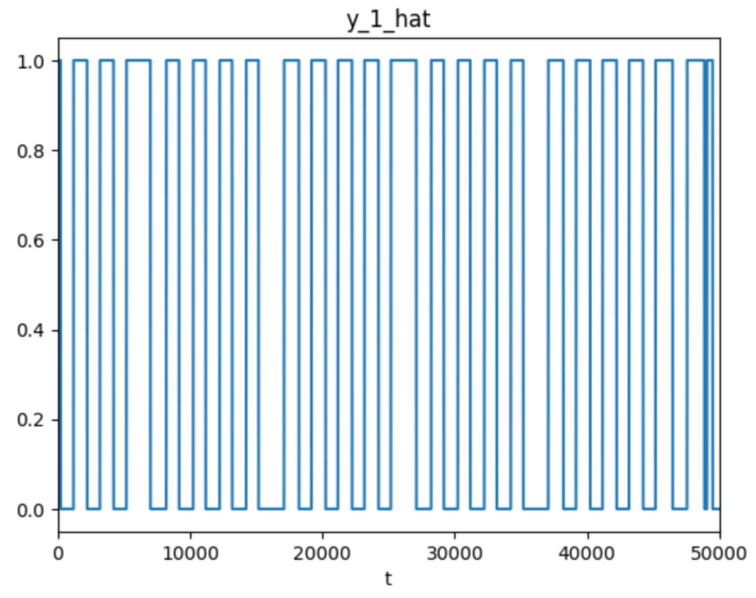
```
# Demodulation block
y_1, y_2 = AnalogDemod(y, sampling_frequency, signal_bandwidth, carrier_frequency)
plt.plot(y_1)
plt.title("y_1")
plt.xlabel("t")
plt.xlim(0, 5e4)
plt.show()
plt.plot(y_2)
plt.title("y_2")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
```

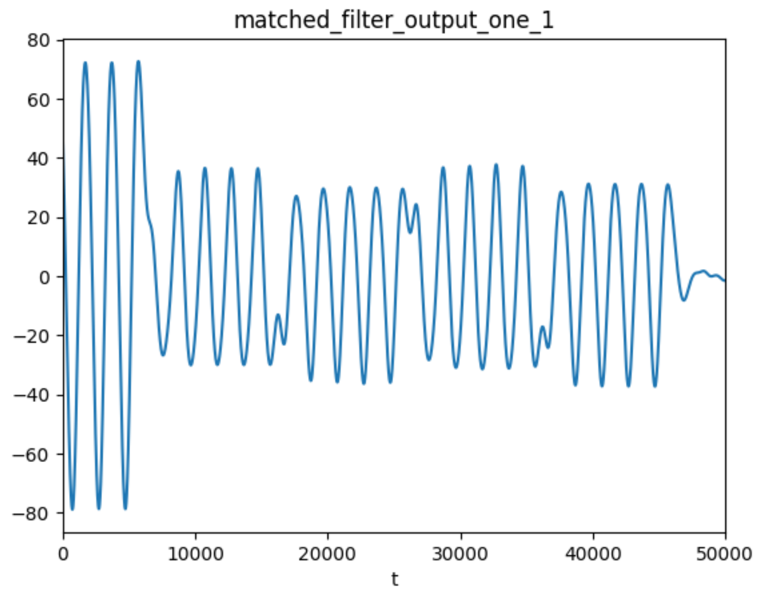


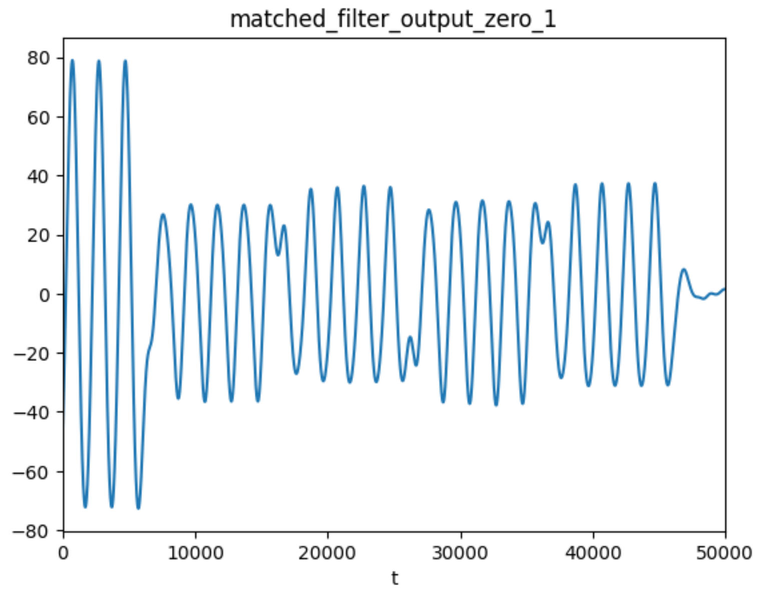


Double-click (or enter) to edit

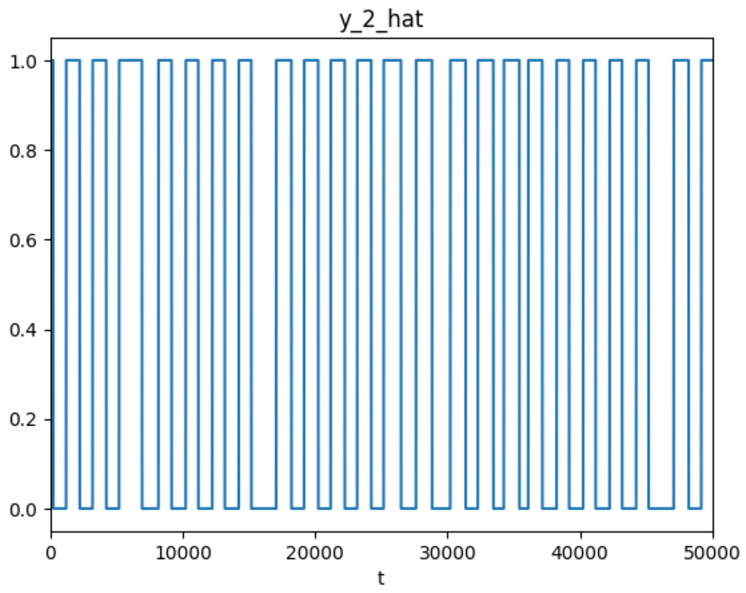
```
# Matched filter and threshold
matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
plt.plot(y_1_hat)
plt.title("y_1_hat")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
plt.plot(matched_filter_output_one_1)
plt.title("matched_filter_output_one_1")
plt.xlim(0, 5e4)
plt.xlabel("t")
plt.show()
plt.plot(matched_filter_output_zero_1)
plt.xlim(0, 5e4)
plt.title("matched_filter_output_zero_1")
plt.xlabel("t")
plt.show()
```

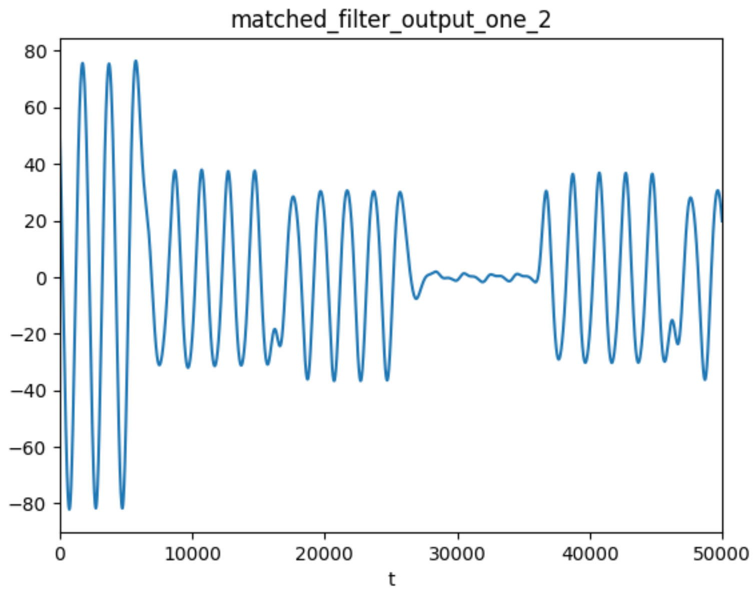

y_1_hat
1.0
0.8
0.6
0.4
0.2
0.0
0
10000
20000
30000
40000
50000
t

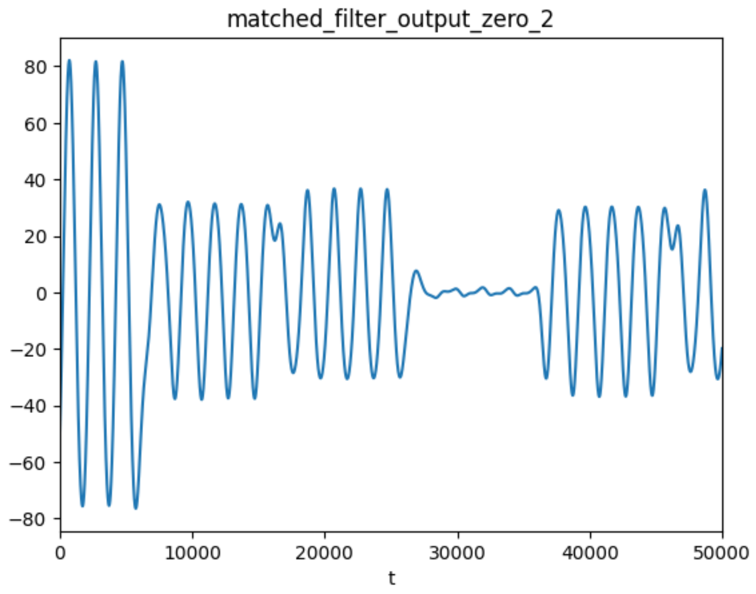

matched_filter_output_one_1
80
60
40
20
0
−20
−40
−60
−80
0
10000
20000
30000
40000
50000
t


matched_filter_output_zero_1
80
60
40
20
0
−20
−40
−60
−80
0
10000
20000
30000
40000
50000
t

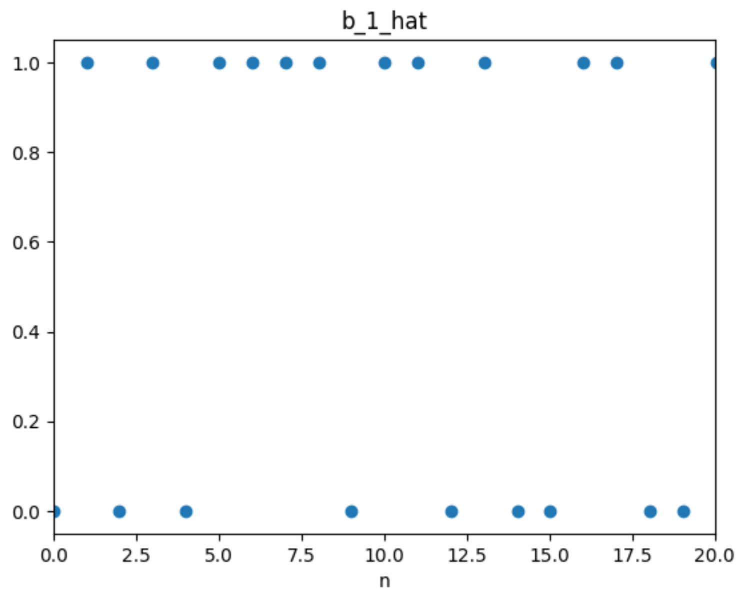
```
    plt.plot(y_2_hat)
    plt.xlim(0, 5e4)
    plt.title("y_2_hat")
    plt.xlabel("t")
    plt.show()
    plt.plot(matched_filter_output_one_2)
    plt.xlim(0, 5e4)
    plt.title("matched_filter_output_one_2")
    plt.xlabel("t")
    plt.show()
    plt.plot(matched_filter_output_zero_2)
    plt.xlim(0, 5e4)
    plt.title("matched_filter_output_zero_2")
    plt.xlabel("t")
    plt.show()
```

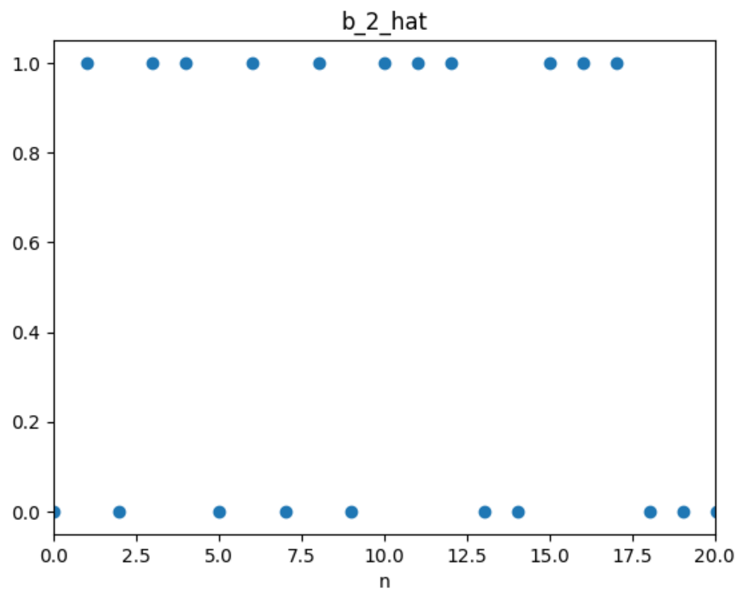

y_2_hat
1.0
0.8
0.6
0.4
0.2
0.0
0
10000
20000
30000
40000
50000
t


matched_filter_output_one_2
80
60
40
20
0
−20
−40
−60
−80
0
10000
20000
30000
40000
50000
t


matched_filter_output_zero_2
80
60
40
20
0
−20
−40
−60
−80
0
10000
20000
30000
40000
50000
t

```
estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
n = np.linspace(0, 500, len(estimated_bit_1))
plt.scatter(n, estimated_bit_1)
plt.title("b_1_hat")
plt.xlim(0, 20)
plt.xlabel("n")
plt.show()

plt.scatter(n, estimated_bit_2)
plt.xlim(0, 20)
plt.title("b_2_hat")
plt.xlabel("n")
plt.show()
```





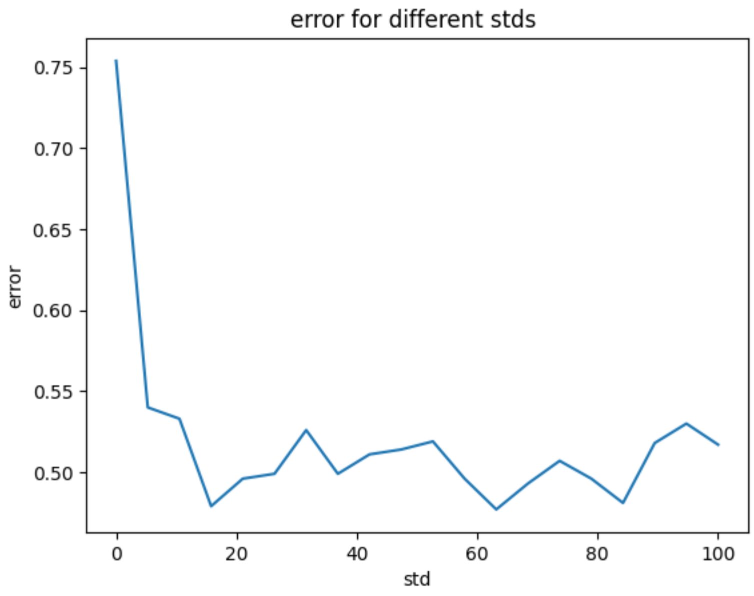
```
b = Combine(estimated_bit_1, estimated_bit_2)
print("bit error rate:" )
print(np.sum(b != sequence)/len(sequence))
```

```
bit error rate:
0.754
```

5.3

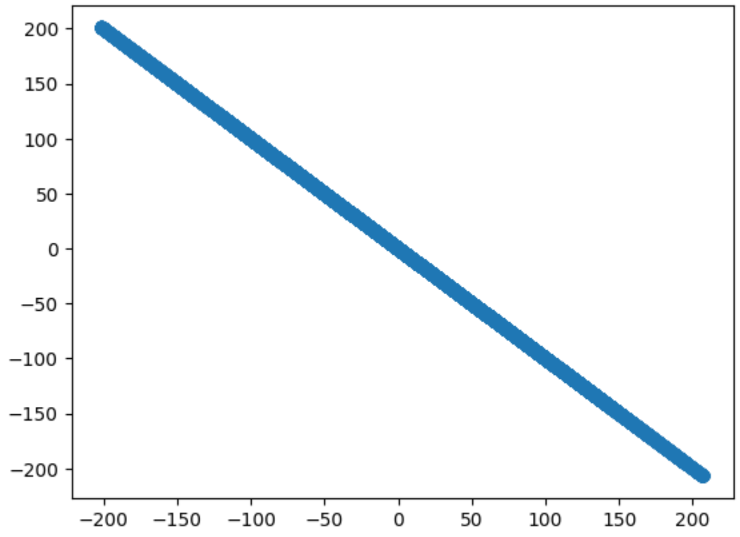
```
mean = 0
num_samples = len(y)
stds = np.linspace(0, 100, 20)
errors = []
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    b = Combine(estimated_bit_1, estimated_bit_2)
    errors.append(np.sum(b != sequence)/len(sequence))
plt.plot(stds, errors)
plt.title("error for different stds ")
plt.xlabel("std")
plt.ylabel("error")
```

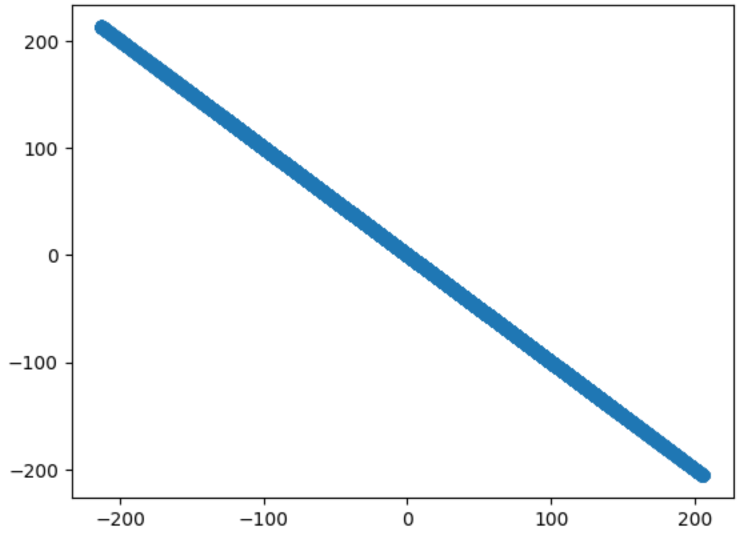
```
 Text(0, 0.5, 'error')
```

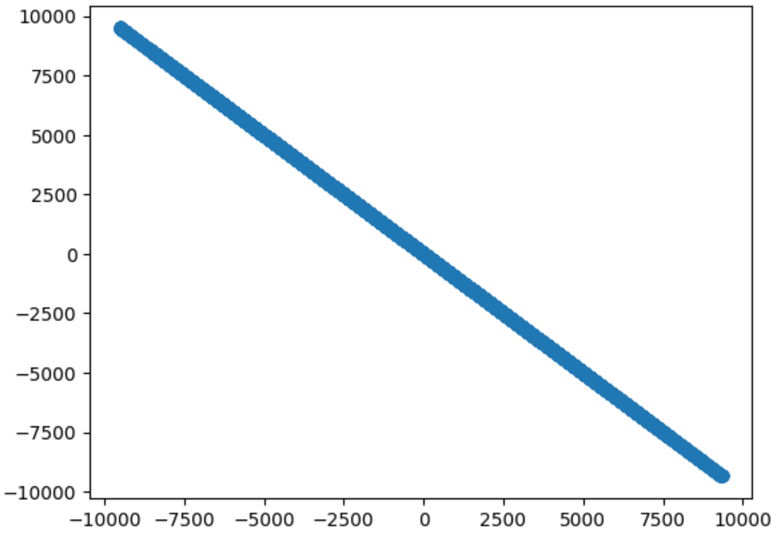


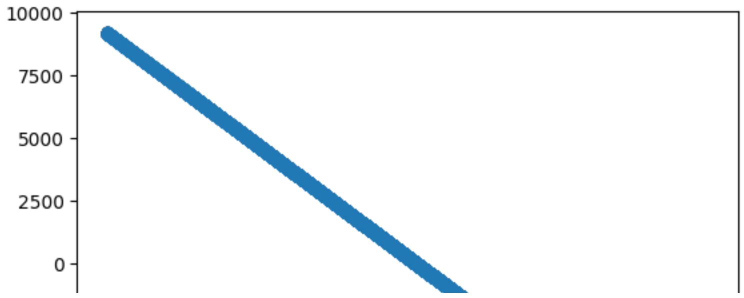
5.4

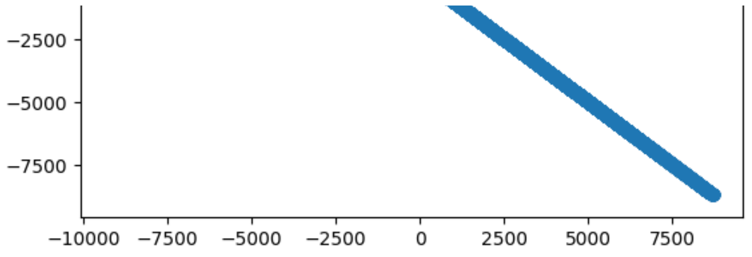
```
mean = 0
stds = [1, 50, 100]
num_samples = len(y)
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    plt.scatter(matched_filter_output_zero_1, matched_filter_output_one_1)
    plt.show()
    plt.scatter(matched_filter_output_zero_2, matched_filter_output_one_2)
    plt.show()
```

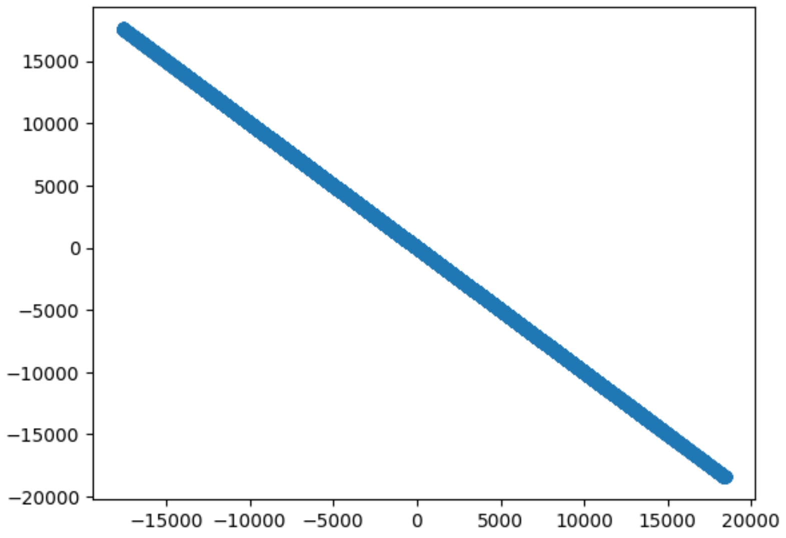


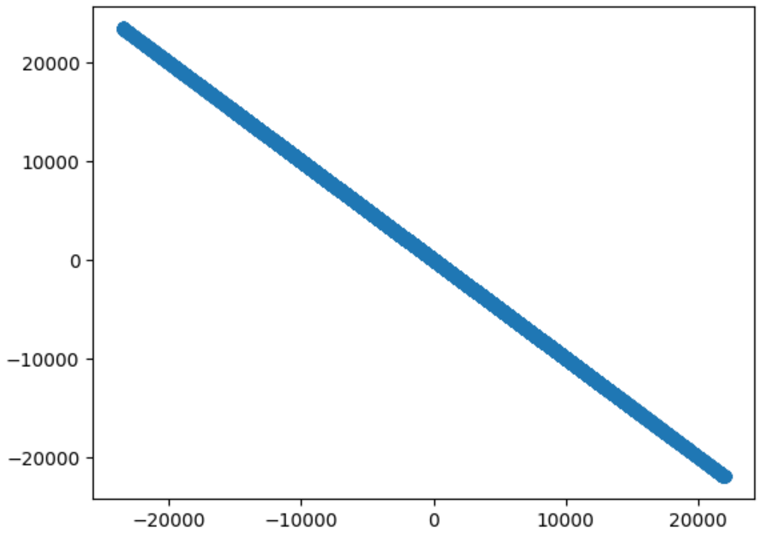












3.4

FSK (Frequency Shift Keying):

Definition: FSK is a digital modulation technique in which the frequency of the carrier signal is varied in proportion to the digital signal. Key Point: In FSK, different frequencies represent different binary states (0 or 1). Advantages: It is relatively simple to implement and can be less affected by noise. Applications: Used in telecommunications, RFID systems, and wireless networks. PSK (Phase Shift Keying):

Definition: PSK is a digital modulation technique where the phase of the carrier signal is varied in relation to the digital input signal. Key Point: In PSK, different phase shifts represent different binary states. Advantages: PSK is efficient in terms of bandwidth utilization and is commonly used in high-speed data transmission. Applications: Widely used in wireless communication systems, satellite communication, and radar systems. PAM (Pulse Amplitude Modulation):

Definition: PAM is a digital modulation technique in which the amplitude of the pulse carrier is varied in proportion to the analog signal being transmitted. Key Point: In PAM, different levels of amplitude represent different signal levels. Advantages: PAM is relatively simple and can be implemented using basic electronics. Applications: Used in pulse code modulation (PCM) for digital audio transmission, ADSL (Asymmetric Digital Subscriber Line) communication, and some fiber optic communication systems.

4 Transferring a sequence of 8-bit numbers

4.1

```
def SourceGenerator(sequence):
    binary_sequence = []
    for num in sequence:
        binary_num = bin(num)[2:].zfill(8)
        binary_sequence.append(binary_num)
    result = []
    for binary_num in binary_sequence:
        for digit in binary_num:
            result.append(int(digit))
    return result
    return binary_array

def OutputDecoder(binary_sequence):
    #print(binary_sequence)

    binary_string = ''.join(str(bit) for bit in binary_sequence)
    #print(binary_string)
    sequence = int(binary_string, 2)

    return sequence
```

4.2

```
length = 1000
sequence = [np.random.randint(0, 255) for _ in range(length)]
binary_sequence = SourceGenerator(sequence)
print(binary_sequence)
```

```
[0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
```

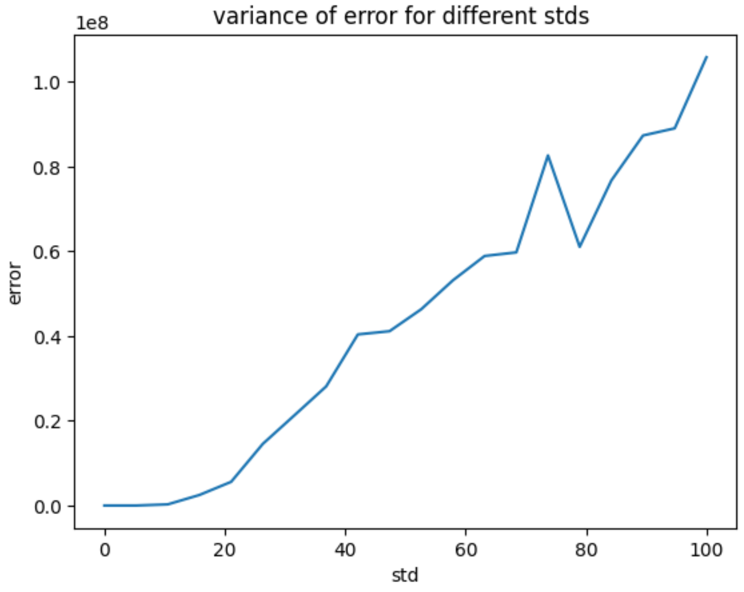
```
b_1 , b_2 = Divide(binary_sequence)
bandwidth_1 = calculate_bandwidth(b_1, sampling_frequency)
print("Bandwidth_1:", bandwidth_1)
bandwidth_2 = calculate_bandwidth(b_2, sampling_frequency)
print("Bandwidth_2:", bandwidth_2)
signal_bandwidth = max(bandwidth_1, bandwidth_2)
```

```
Bandwidth_1: 500.0
Bandwidth_2: 500.0
```

```
amp = 1
pulse = generate_rectangular_pulse(pulse_width, 1, sampling_frequency)

# Plotting the pulse
time = np.arange(0, pulse_width, 1 / sampling_frequency)


x_1 = PulseShaping(b_1, -1*pulse, pulse)
x_2 = PulseShaping(b_2, -1*pulse, pulse)
x_c = AnalogMod(x_1, x_2, sampling_frequency, carrier_frequency)
y = Channel(x_c, sampling_frequency,center_frequency,  channel_bandwidth)
```

```
mean = 0
num_samples = len(y)
stds = np.linspace(0, 100, 20)
variance_of_errors = []
for std in stds:
    white_noise= np.random.normal(mean, std, size=num_samples)
    y_final = y + white_noise
    #plt.plot(y_final)
    #plt.show()
    y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
    matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
    matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
    estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
    estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
    b = Combine(estimated_bit_1, estimated_bit_2)
    b = np.split(np.asarray(b), 1000)
    estimated_num = []
    for i in b:
        estimated_num.append(OutputDecoder(i.astype(int)))
    # Calculate the error for each data point
    errors = [predicted - true for predicted, true in zip(estimated_num, sequence)]
    # Square each error
    squared_errors = [error ** 2 for error in errors]
    # Calculate the mean squared error
    mean_squared_error = np.mean(squared_errors)
    # Calculate the variance of the error
    variance_of_error = np.var(squared_errors)

    variance_of_errors.append(variance_of_error)



plt.plot(stds, variance_of_errors)
plt.title("variance of error for different stds ")
plt.xlabel("std")
plt.ylabel("error")
```

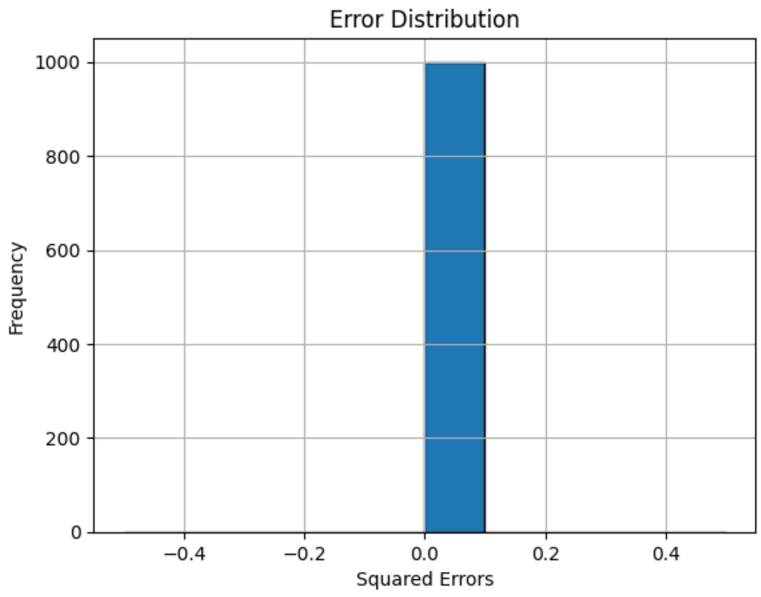
```
Text(0, 0.5, 'error')
```



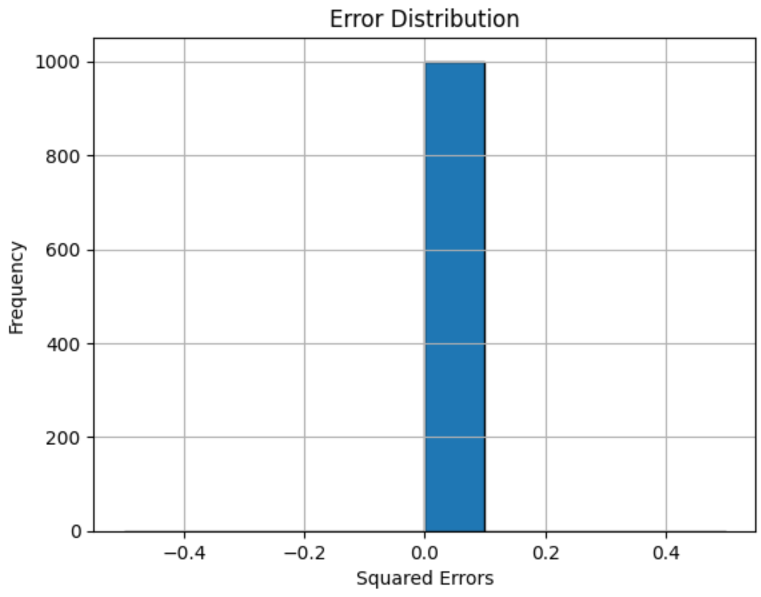
4.3

```
    mean = 0
    num_samples = len(y)
    stds = [1, 10, 30, 50, 70, 100, 1000]
    variance_of_errors = []
    for std in stds:
        white_noise= np.random.normal(mean, std, size=num_samples)
        y_final = y + white_noise
        #plt.plot(y_final)
        #plt.show()
        y_1, y_2 = AnalogDemod(y_final, sampling_frequency, signal_bandwidth, carrier_frequency)
        matched_filter_output_one_1, matched_filter_output_zero_1, y_1_hat = MatchedFilter(y_1, -1*pulse, pulse)
        matched_filter_output_one_2, matched_filter_output_zero_2, y_2_hat = MatchedFilter(y_2, -1*pulse, pulse)
        estimated_bit_1 = sample_and_quantize(pulse, y_1_hat)
        estimated_bit_2 = sample_and_quantize(pulse, y_2_hat)
        b = Combine(estimated_bit_1, estimated_bit_2)
        b = np.split(np.asarray(b), 1000)
        estimated_num = []
        for i in b:
            estimated_num.append(OutputDecoder(i.astype(int)))
        # Calculate the error for each data point
        errors = [predicted - true for predicted, true in zip(estimated_num, sequence)]
        # Square each error
        squared_errors = [error ** 2 for error in errors]
        # Plot the error distribution
        print("std is:")
        print(std)
        plt.hist(squared_errors, bins=10, edgecolor='black')
        plt.xlabel('Squared Errors')
        plt.ylabel('Frequency')
        plt.title('Error Distribution')
        plt.grid(True)
        plt.show()
```

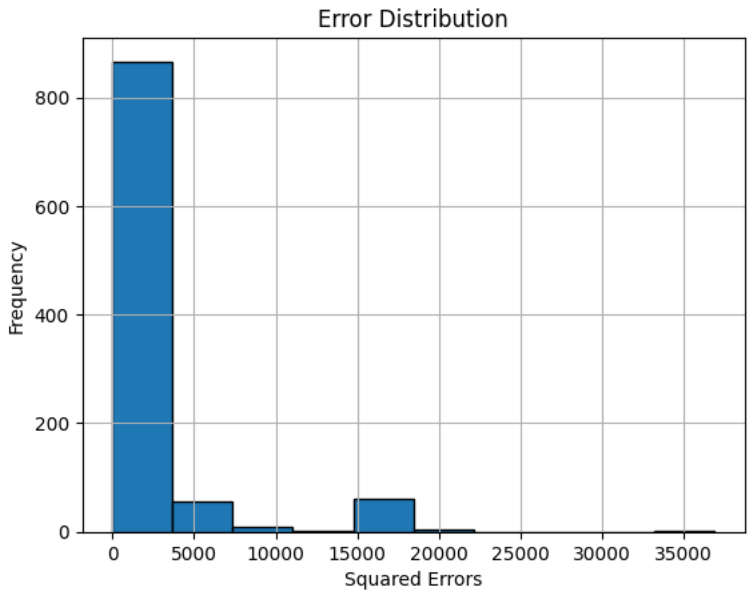
```
std is:
1
```



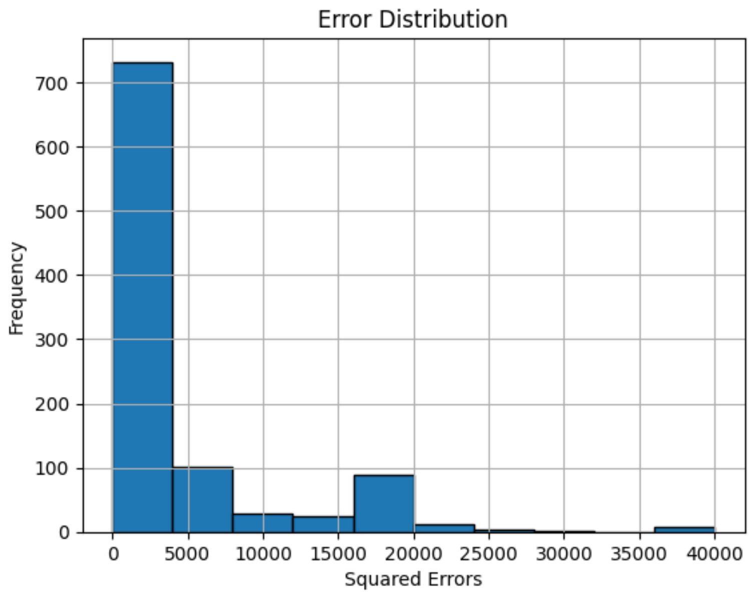
```
std is:
10
```



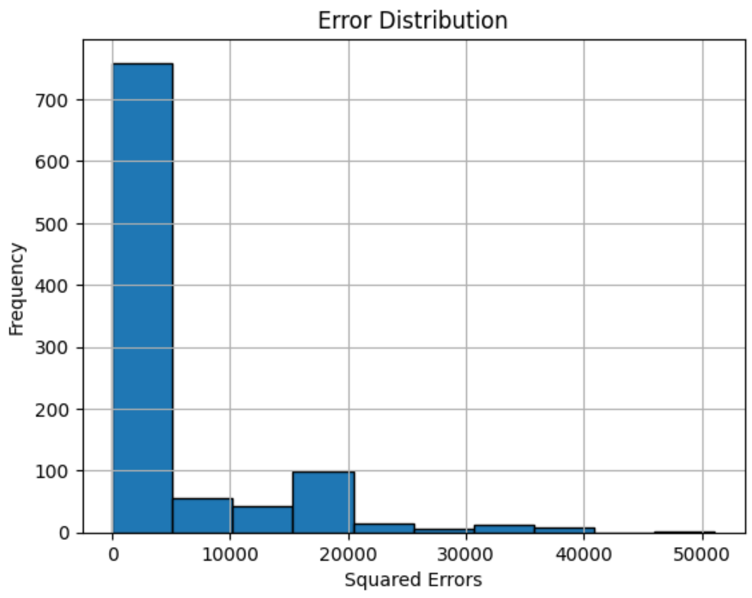
```
std is:
30
```



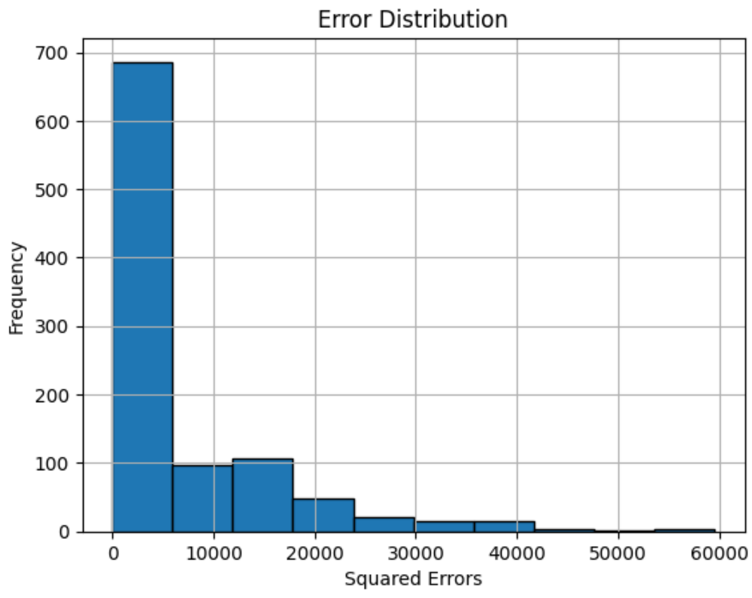
```
std is:
50
```



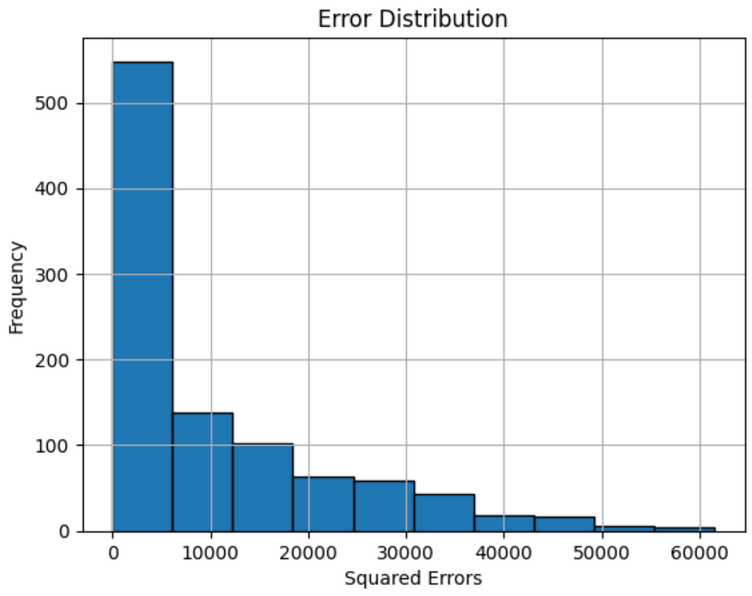
```
std is:
70
```



```
std is:
100
```



```
std is:
1000
```



The limiting behavior of the error distribution, as mentioned earlier, would tend towards a right-skewed distribution. This happens because squared errors are always non-negative values, causing the distribution to accumulate more towards the higher error values.

To find this limiting behavior using the histogram plotted based on the squared errors:

Shape of the Distribution: it is positive skewed.

Peak and Tail: A right-skewed distribution typically has a longer tail on the right side as we can see.

Mean and Median: the mean is usually greater than the median due to the influence of extreme values on the right side.

Mode: In a right-skewed distribution, the mode should be less than both the mean and the median. we can see that the variance tends to infinity.

4.4

When the noise tends to infinity in a communication system, the error variance can be calculated analytically. In such cases, the error variance will also tend to infinity due to the uncontrollable and overwhelming influence of the noise.

To calculate the error variance analytically in this scenario, we can consider the ideal case where the received signal is the sum of the transmitted signal and the noise:

[Y = X + N]

where:

(Y) is the received signal, (X) is the transmitted signal, (N) is the noise. The error is given by:

[E = Y - X = N]

Given that the noise tends to infinity, the variance of the error, (\sigma_E^2), can be calculated as the variance of the noise, (\sigma_N^2), which is also tending to infinity.

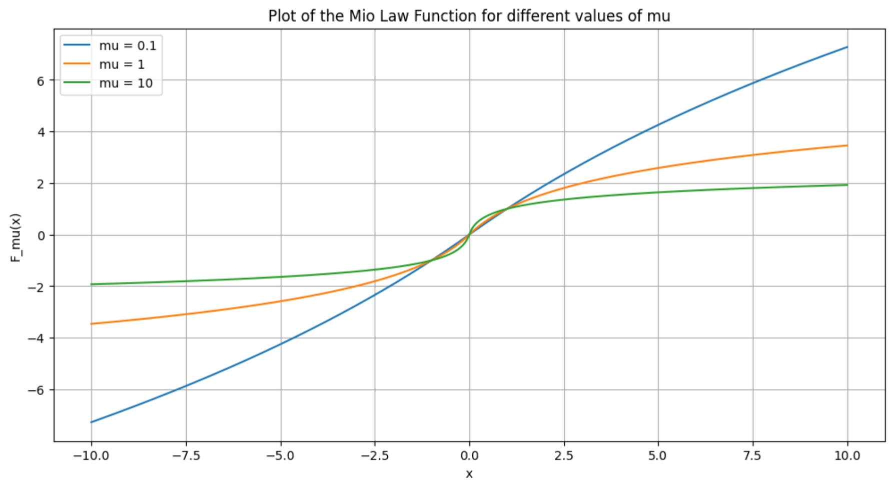
Thus, when the noise tends to infinity, the error variance, (\sigma_E^2), will also tend to infinity.

This result is compatible with the expectation that as the noise level increases without bounds, the error in the system will become unbounded as well, leading to an infinitely large error variance.

5 Compander

5.1

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

def mio_law(x, mu):
    return np.sign(x) * np.log(1 + mu * np.abs(x)) / np.log(1 + mu)

# Define the range of x values
x = np.linspace(-10, 10, 1000)

# Define different values of mu
mu_values = [0.1, 1, 10]

plt.figure(figsize=(12, 6))

for mu in mu_values:
    plt.plot(x, mio_law(x, mu), label=f'mu = {mu}')

plt.xlabel('x')
plt.ylabel('F_mu(x)')
plt.title('Plot of the Mio Law Function for different values of mu')
plt.legend()
plt.grid(True)
plt.show()
```



5.2

```
import pyaudio
import wave
#
# # Constants for recording
# FORMAT = pyaudio.paInt16
# CHANNELS = 1
# RATE = 44100  # Sampling rate
# CHUNK = 1024
# RECORD_SECONDS = 120
OUTPUT_FILENAME = "output.wav"
#
# # Initialize PyAudio
# audio = pyaudio.PyAudio()
#
# # Open a stream for recording
```