EITF75: Lab 2 AB Report

Group 39
Ivar Nilsson and Harald Förare

October 12, 2022

Preparation exercises

1. (a)

$$data(n) = 110110100 \longrightarrow x(n) = 1, 1, -1, 1, 1, -1, 1, -1, -1$$

(b) The highest frequency possible in our signal would be alternating -1 and 1. This would be represented by a sinusoid at period $2T_s$ and have a frequency of $\frac{1}{2}F_s$. Therefore the sampling frequency would be given by:

$$F_s = \frac{DataRate}{DataBits/Sample} = \frac{1 * 10^6}{1} = 1MHz$$

(c)

$$D = \frac{SignalSpeed}{Distance * SampleFrequency} = \frac{3 * 10^8}{300 * 1 * 10^6} = 1sample$$

(d)

$$r(n) = s(n) + \alpha s(n - D)$$

(e) Transfer function by Z-transform utilizing mainly the time shift property 2.3.1.4 from the formula collection:

$$Y(z) = X(z) + \alpha z^{-1} X(z)$$

$$H(z) = \frac{Y(z)}{X(z)} = 1 + \alpha z^{-1}$$

$$H(z) = \frac{z + \alpha}{z}$$

The impulse response transfer function can be acquired by multiplying $H(z)$ with the impulse function transfer function $\delta(n) \longleftrightarrow 1$.

$$H_{impulse} = 1H(z)$$

$$H_{impulse} = H(z)$$

Since the transfer function is identical, the difference equation will also be.

$$h(n) = s(n) + \alpha s(n - 1)$$

(f) Since the echo delay of the signal is 1 sample, the maximum window of bits we will have to check is two; the current one and the echoed one before. This gives us $2^2 = 4$ combinations. Using the specified encoding, this yields the possible signals

$$00 \longrightarrow -1 - \alpha$$
$$01 \longrightarrow -1 + \alpha$$
$$10 \longrightarrow +1 - \alpha$$
$$11 \longrightarrow +1 + \alpha$$

which are all different, provided α isn't too close to 1. If α is close to 0, we can ignore the echo.

2. (a)
$$H_R(z) = \frac{1}{H(z)} = \frac{z}{z+\alpha}$$

(b)
$$H_T(z) = \frac{1}{H(z)} = \frac{z}{z+\alpha}$$

$H_R(z) = H_T(z)$ because multiplication of our transfer functions is commutative.

(c) The signal processing without the "genie" will be best placed at the receiving end of the communication since the receiver can listen to and characterize the echo. Furthermore, the transmission protocol could include a predefined test signal which is known to both parties. When the receiver gets the test signal, it could do a signal analysis to characterize the transmission channel and determine an appropriate filter to remove echo and other distortions.

3. (a)
$$\tilde{s}(n) = [\underline{1}, 2, 3, 4]$$

$$s(n) = [\underline{3}, 4, 1, 2, 3, 4]$$

(b)
$$h(n) = [\underline{1}, 0, 1]$$
$$r(n) = s(n) * h(n) = [\underline{3}, 4, 4, 6, 4, 6, 3, 4]$$

(c)
$$r_c(n) = s(n) \circledast_4 h(n) = [\underline{4}, 6, 4, 6]$$

(d) The elements of our real circular convolution $r_c(n)$ match the middle ones of our simulated one using a circular prefix $r(n)$.

4. (a) i.
$$x(n) = \tilde{S}(k) \xrightarrow{\text{inverse DFT}} \tilde{s}(n)$$

ii.
$$\tilde{s}(n) = \frac{1}{N} \sum_{k=0}^{N-1} \tilde{S}(k) e^{j2\pi nk/N}$$

iii.

$$s(n) = \begin{cases} \tilde{s}(n+N-L), & \text{for } 0 \leq n < (L-1) \\ \tilde{s}(n-L), & \text{for } L \leq n < (N+L) \end{cases}$$

(b)

$$\tilde{R}(k) = \tilde{S}(k)H(k)$$

(c)

$$?_n = DFT(h)(n)$$

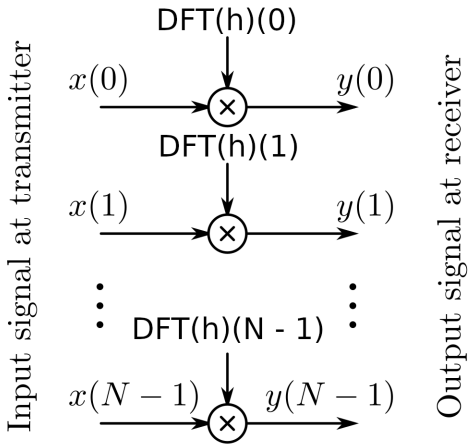

Figure 1: Diagram of our OFDM system

(d)

$$x(n) = \frac{y(n)}{DFT(h)(n)}$$

Lab tasks

Lab task 1

An evenly distributed array of ones and zeros was created using a numpy function. The array was then scaled to between -1 and 1.

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

def get_random_data(length: int):
    return np.random.randint(0, 2, length)

def get_random_signal(length: int):
    return (get_random_data(length) - 0.5) * 2

if __name__ == "__main__":
    random_signal = get_random_signal(64)
```

```
    print(f"Sequence: [{' '.join(map(str, map(int, random_signal)))
        }]")
    print(f"Average: {np.sum(random_signal) / len(random_signal)}")

    plt.stem(random_signal)
    plt.show()
```

The binary sequence
[0 1 1 0 0 1 1 1 1 0 1 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 0 0 1 1 1 1 0 1 0 1 1]
was generated which coincidentally has an equal number of both values. It has its encoded signal plotted below in Fig. 2

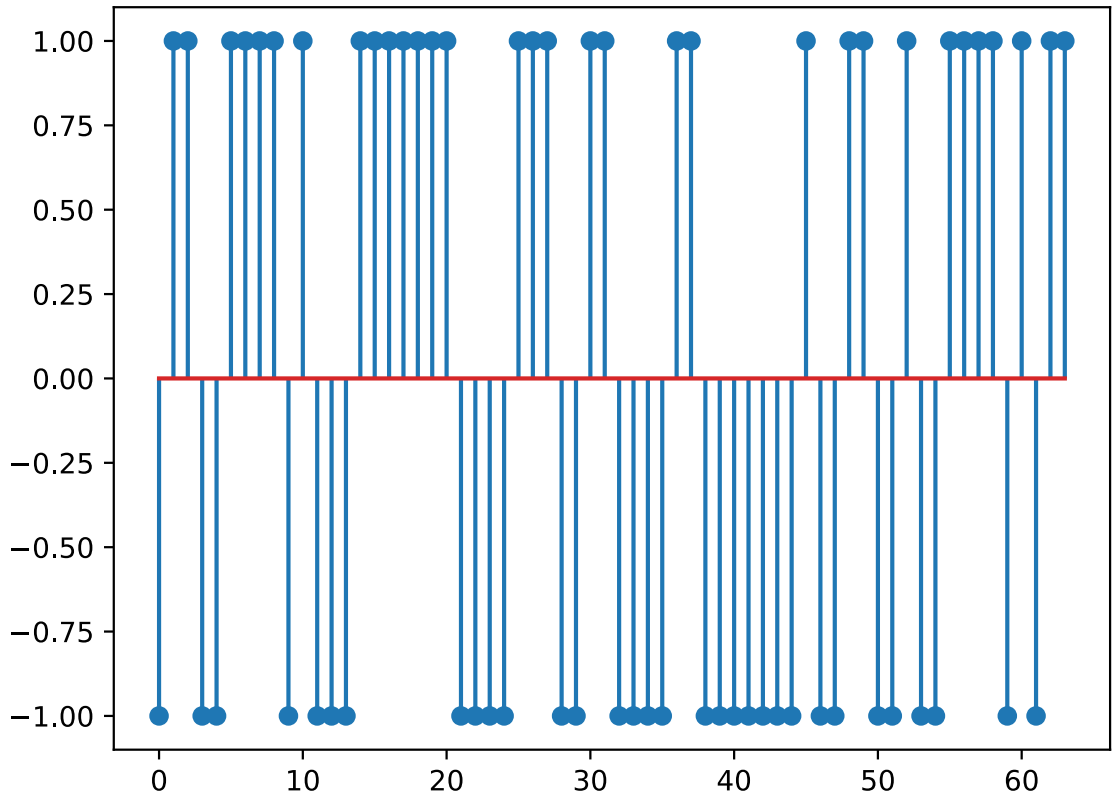

Figure 2: Randomly generated input signal

which seems to be a valid signal encoding of the transmit bits.

Lab task 2

In Fig. 3 the received signal is shown with varying values of the reflection factor. From the figures it can be seen that the different transmitted values of the signal without the echo can simply be recreated

by setting all sample values above 0 to 1 and all sample values below or equal to 0 to -1. This makes it possible to perfectly recreate any signal with a reflection factor within the limits of $|\alpha| < 1$ as seen in Fig. 4. However if noise is added to the received signal the accuracy of the recreation will decrease this can be seen in Fig. 5. The noise makes the signal get more errors for reflection factors that were previously error free.

The received signal was calculated using:

```
import numpy as np

def get_received_signal(transmitted_signal,
   reflection_scaling_coefficient):
    return np.convolve(transmitted_signal, np.array([1,
       reflection_scaling_coefficient]))
```

It was placed in this loop to compare different values of the reflection scaling coefficient:

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

from count_bit_errors import count_bit_errors
from task_1A import get_random_signal
from task_2A_receive import get_received_signal


transmitted_signal = get_random_signal(10_000)

errorSeries = list()
alphaSeries = list()

for alpha in np.linspace(-3, 3, 1000):
    received_signal = get_received_signal(transmitted_signal, alpha)[:
       len(transmitted_signal)]

    # Add noise, (can be commented out)
    received_signal += np.random.normal(0, 0.5, len(received_signal))

    # Signal thresholding detection (not exactly the data format but
       1:1 bijective encoding of it and comparable to transmitted
       signal)
    recovered_signal = ((received_signal > 0).astype("float64") - 0.5)
        * 2

    num_errors = count_bit_errors(transmitted_signal, recovered_signal
       )
    percentage_errors = 100 * num_errors / len(recovered_signal)

    alphaSeries.append(alpha)
    errorSeries.append(percentage_errors)
```

```
if __name__ == "__main__":
    plt.plot(alphaSeries, errorSeries)
    plt.xlabel("reflection scaling coefficient")
    plt.ylabel("percentage errors")
```

Bit errors were counted using:

```
import numpy as np

def count_bit_errors(u, v, epsilon = 0.000001):
    return np.count_nonzero(np.abs(u - v) > epsilon)
```

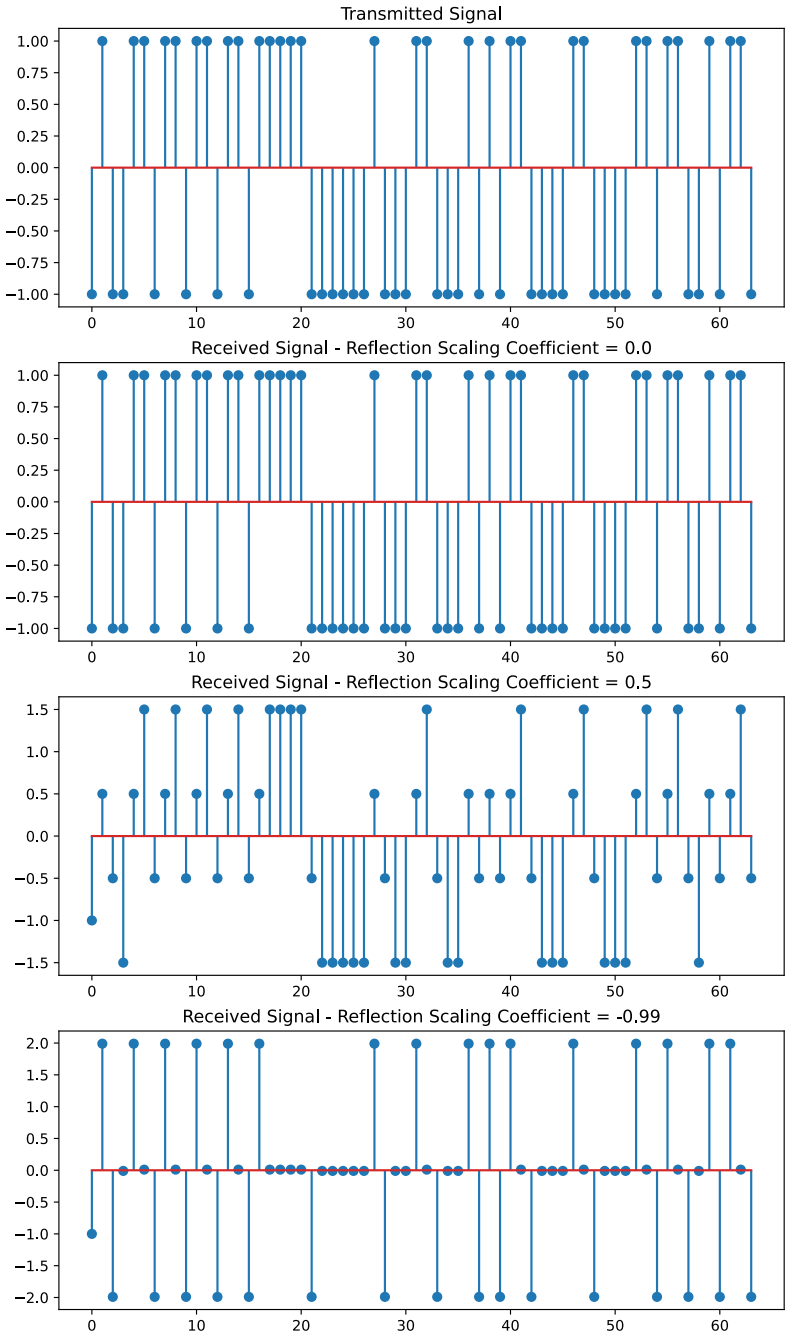

Figure 3: Received signal with different reflection factor values

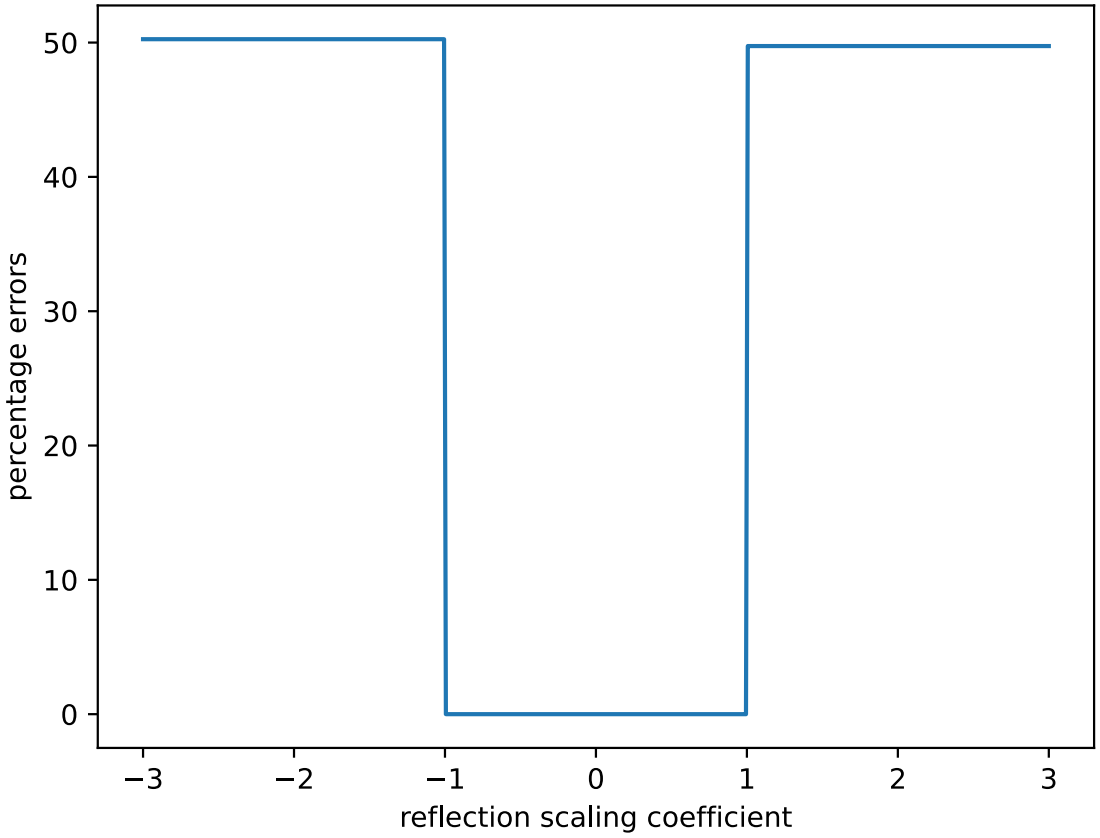

Figure 4: Percentage of wrong detected values for different reflection scaling coefficients α on a signal without noise

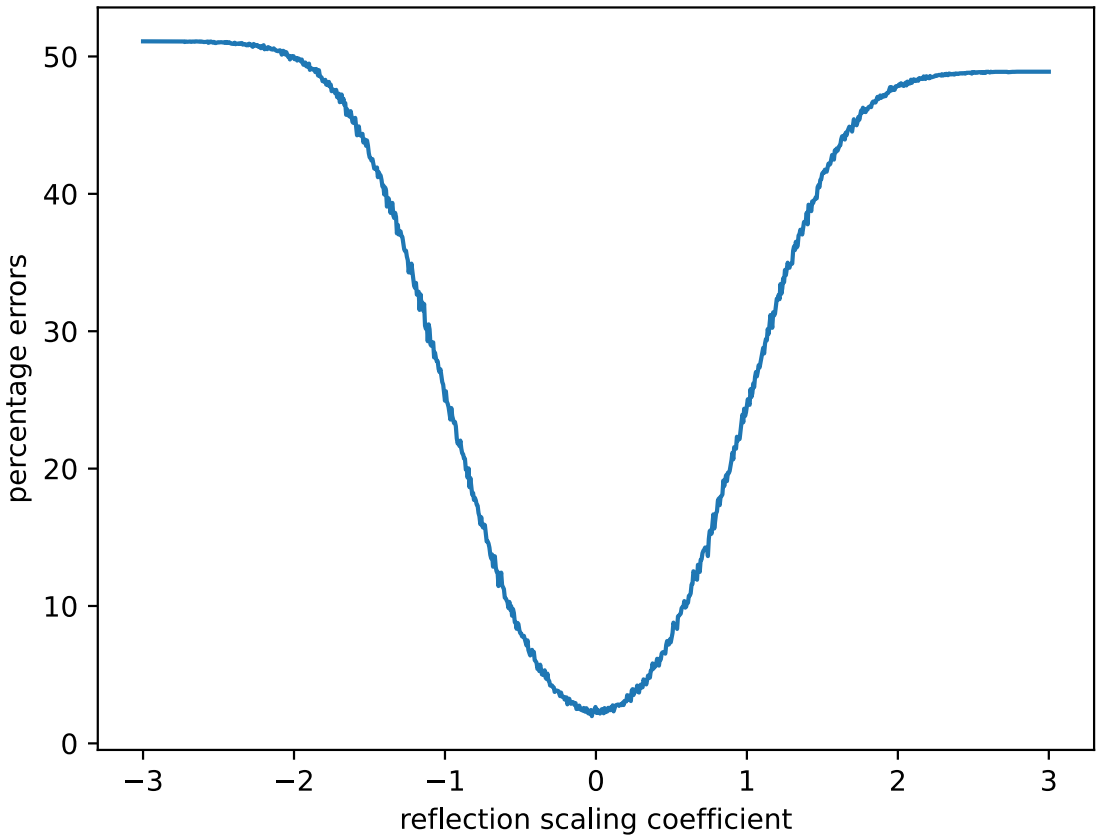

Figure 5: Percentage of wrong detected values for different reflection scaling coefficients α on a signal with gaussian noise of variance 0.25

Lab task 3

This section compares transmitter processing to receiver processing.

Transmitter or receiver processing code:

```
from scipy.signal import lfilter as filter
import matplotlib.pyplot as plt
import numpy as np
import math

from task_1A import get_random_signal
from task_2A_receive import get_received_signal
from count_bit_errors import count_bit_errors


do_include_noise = True

do_processing_in_transmitter = True  # Modify this to change behaviour
```

```
do_processing_in_receiver = not do_processing_in_transmitter


inputted_signal = get_random_signal(10_000)


errorSeries = list()
alphaSeries = list()
for alpha in np.linspace(-3, 3, 1000):
    denominator = np.array([1, alpha])
    numerator = np.array([1])

    if do_processing_in_transmitter:
        transmitted_signal = filter(numerator, denominator,
           inputted_signal)

    else:
        transmitted_signal = inputted_signal


    received_signal = get_received_signal(transmitted_signal, alpha)[:
       len(inputted_signal)]

    if do_include_noise:
        received_signal += np.random.normal(0, math.sqrt(0.25), len(
           received_signal))


    if do_processing_in_receiver:
        recovered_signal = filter(numerator, denominator,
           received_signal)

    else:
        recovered_signal = received_signal


    # Perform bit detection
    recovered_signal = ((recovered_signal > 0).astype("float64") -
       0.5) * 2

    num_bit_errors = count_bit_errors(recovered_signal,
       inputted_signal)

    errorSeries.append(100 * num_bit_errors / len(transmitted_signal))
    alphaSeries.append(alpha)


if __name__ == "__main__":
```

```
    plt.plot(alphaSeries, errorSeries)
    plt.xlabel("reflection scaling coefficient")
    plt.ylabel("percentage bit errors")
    plt.show()
```

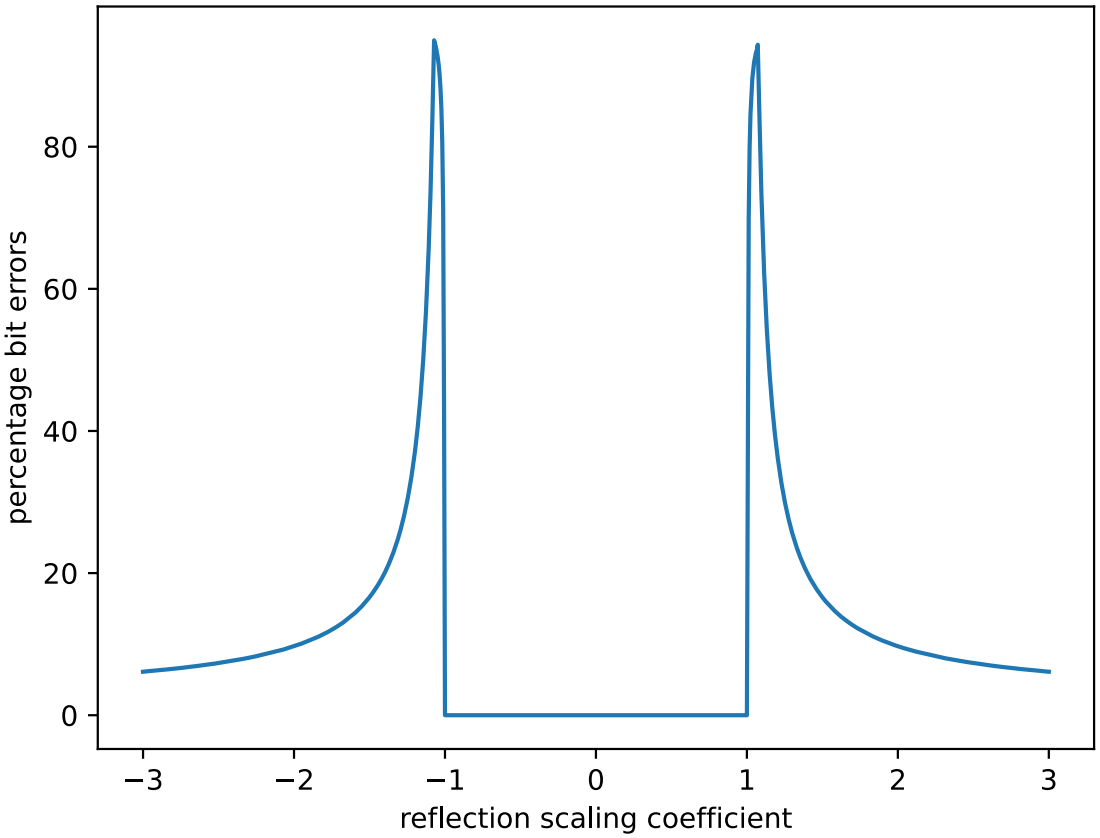

Figure 6: Percentage bit errors for different reflection scaling coefficients using transmitter processing

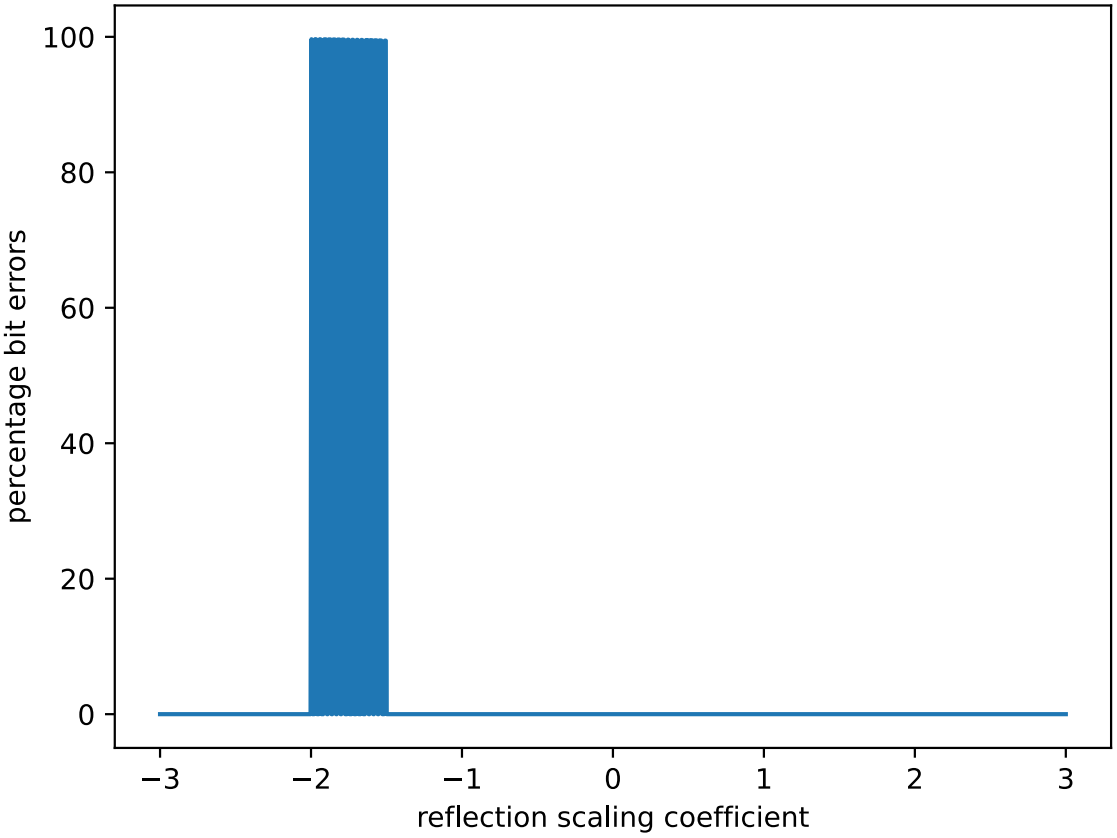

Figure 7: Percentage bit errors for different reflection scaling coefficients using receiver processing

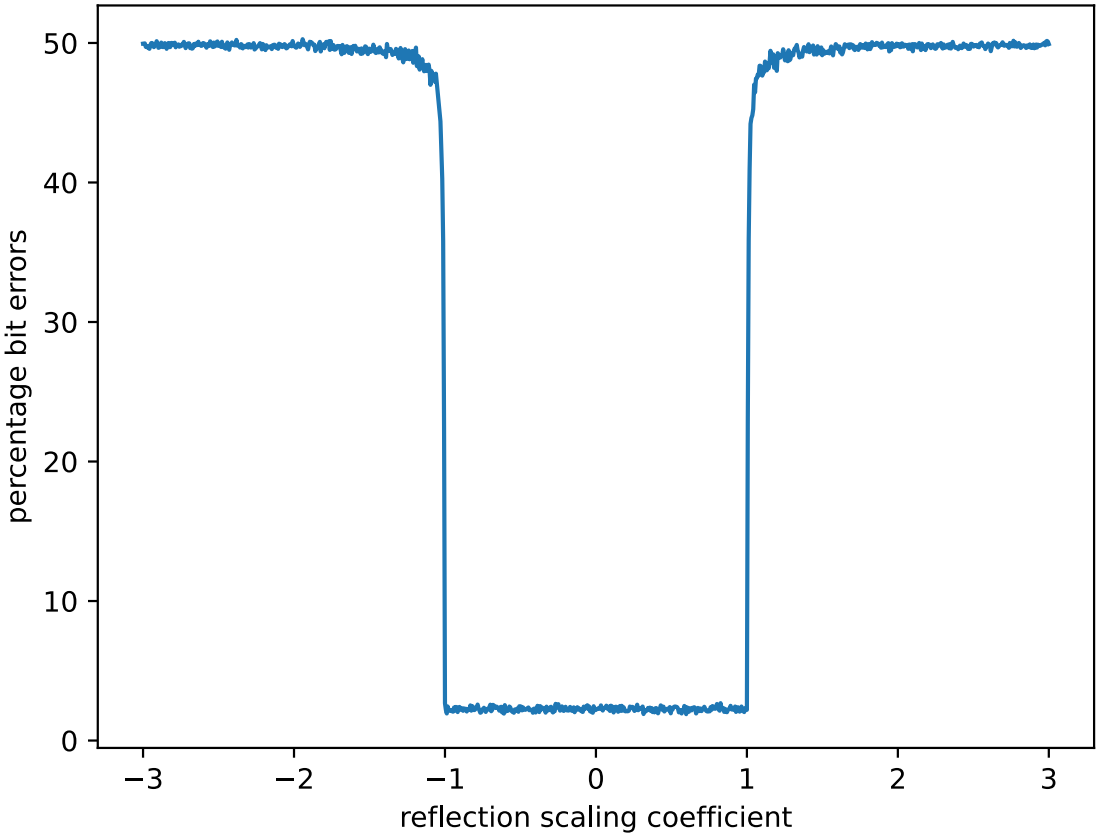

Figure 8: Percentage bit errors for different reflection scaling coefficients using transmitter processing with noise

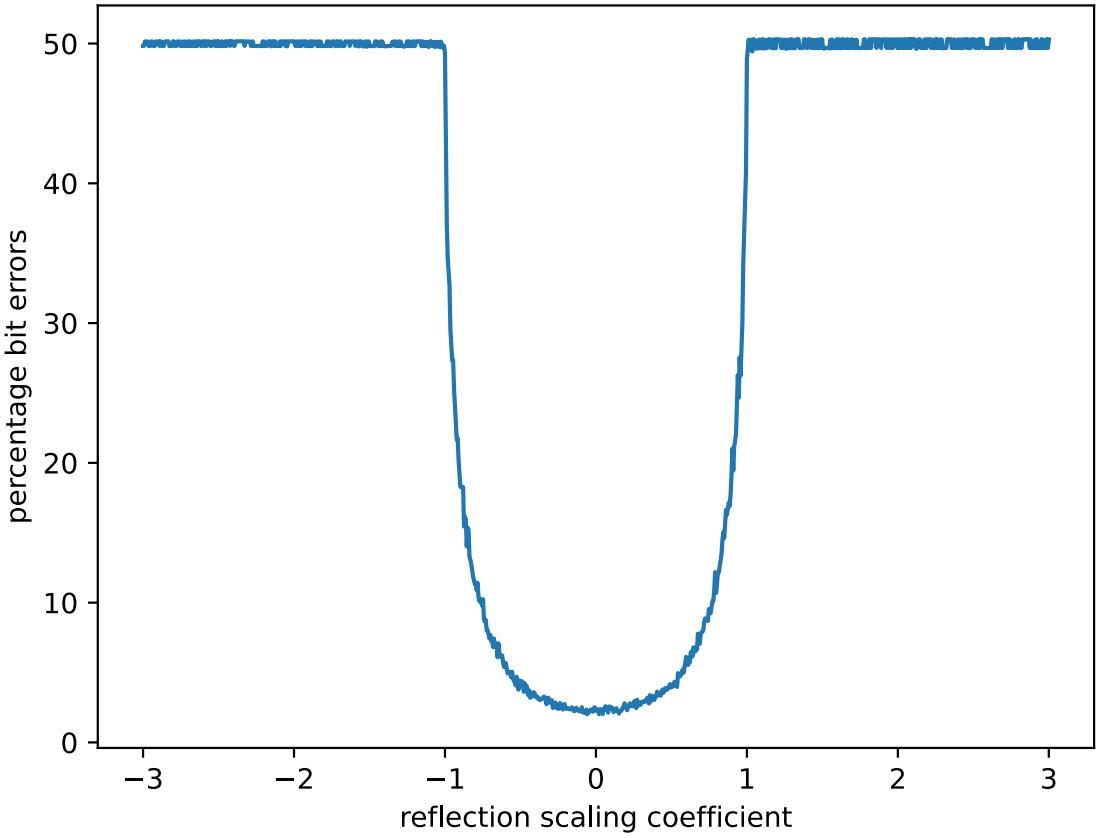

Figure 9: Percentage bit errors for different reflection scaling coefficients using receiver processing with noise

All added noise has a variance of 0.25 as seen in the code above.

It is less "easy" to figure out what bit sequence was transmitted when $|\alpha|$ becomes greater. Values more than 1 away from zero causes "wrap-around" and also makes it harder. The transfer function convolution does a better job than us "looking" at the plot of the received signal.

The performance within $-1 < \alpha < 1$ is "perfect" for both transmitter- and receiver processing in a noise free environment. What happens outside is less important since extreme α values carry little physical meaning.

When adding noise it becomes apparent that transmitter processing has better performance, with it achieving the same peak performance, but keeping it throughout $-1 < \alpha < 1$. This is because receiver processing effectively cancels out the transmission channel effects, causing the receiver to receive the same signal regardless of α. Receiver processing doesn't have this luxury and has to apply processing to an already noisy signal. The receiver doesn't know the transfer function of the noise, and therefore can't cancel it out. This causes more strongly echoed noise to pass through increasing errors for bigger $|\alpha|$.

Including either transmitter- or receiver processing increases detection performance, as should be expected. Processing largely removes effects of the transmission channel, with some restrictions for receiver processing as discussed in the paragraph above, leaving only noise to be contended with.

In transmitter processing (Fig. 8), the effects of the transmission channel were completely negated leaving only noise left, compared to what can be seen in Fig. 5.

Lab task 4

OFDM transmissions were modelled using the function:

```
import numpy as np
import math

def OFDM_transmission(x_n, noise_variance, alpha):
    N = len(x_n)

    S_tilde_k = x_n

    s_tilde_n = np.fft.ifft(S_tilde_k, N)

    H_n = np.array([1, alpha])
    L = len(H_n)

    s_n = np.append(s_tilde_n[-(L - 1):], s_tilde_n)

    r_n = np.convolve(s_n, H_n)

    # Add noise to both real and imaginary components
    noise = np.random.normal(0, math.sqrt(noise_variance / N / 2), len
       (r_n))
    noise = noise + 1j*noise

    r_n += noise

    r_tilde_n = r_n[L - 1:-(L - 1)]

    R_tilde_k = np.fft.fft(r_tilde_n, N)

    y_n = R_tilde_k

    H_k = np.fft.fft(H_n, N)

    x_n_recovered = y_n / H_k
    x_n_recovered = np.real(x_n_recovered)

    return x_n_recovered
```

The code

```
import numpy as np

from OFDM_transmission import OFDM_transmission
```

```
from task_1A import get_random_signal

alpha_series = list()
error_series = list()

for alpha in np.linspace(-3, 3, 1_000):
    X = get_random_signal(64)
    Y = OFDM_transmission(X, 0.0, alpha)

    Y = ((Y > 0).astype("float64") - 0.5) * 2

    number_of_errors = np.count_nonzero(np.abs(X - Y) > 10**(-8))

    alpha_series.append(alpha)
    error_series.append(number_of_errors)

if __name__ == "__main__":
    import matplotlib.pyplot as plt
    plt.plot(alpha_series, error_series)
    plt.show()
```

generated the graph

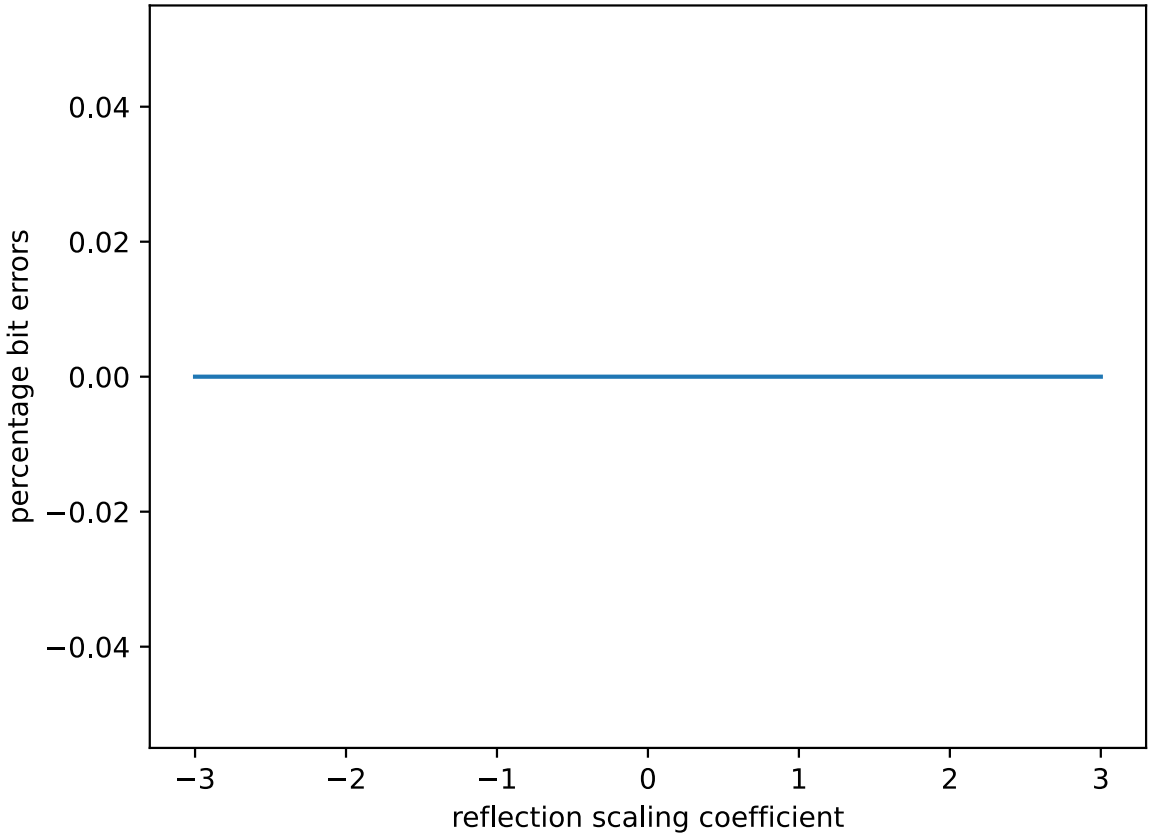

Figure 10: Percentage bit errors in OFDM transmission without noise

which shows, by example, that our OFDM-implementation is likely to be correct.

OFDM is a very clever algorithm. Our implementation seems to work and we didn't expect anything else in particular. Let's move on to task 5 to actually test it in the presence of noise.

Lab task 5

The code

```
import numpy as np

from OFDM_transmission import OFDM_transmission
from task_1A import get_random_signal

alpha_series = list()
error_series = list()

for alpha in np.linspace(-3, 3, 1000):
    X = get_random_signal(1000)
    Y = OFDM_transmission(X, 0.25, alpha)
```

```
    X_bits = (X > 0).astype("int8")
    Y_bits = (Y > 0).astype("int8")

    n_bit_errors = np.count_nonzero(X_bits - Y_bits)

    percentage_bit_errors = 100 * n_bit_errors / len(X)

    alpha_series.append(alpha)
    error_series.append(percentage_bit_errors)


if __name__ == "__main__":
    import matplotlib.pyplot as plt
    plt.plot(alpha_series, error_series)
    plt.xlabel("reflection scaling coefficient")
    plt.ylabel("percentage bit errors")
    plt.show()
```

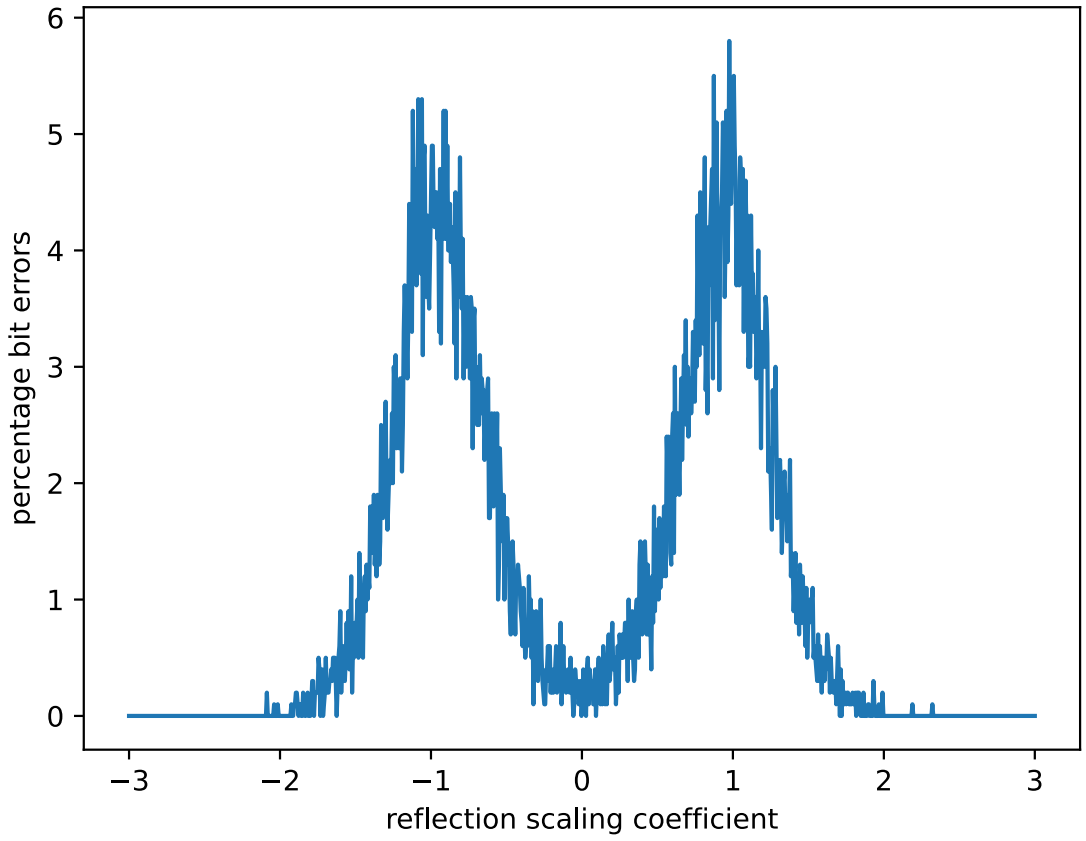

Figure 11: Percentage bit errors in OFDM transmission with noise

was used to test our OFDM implementation in the presence of noise. *We increased transmission length N to make the produced plot clearer by reducing quantization.*

The regular bit recovery code is a good strategy since it should be independent of the underlying transmission channel. Since OFDM is a good way to send signals in an error-robust way we would expect the received signal to look like the transmitted one a lot. OFDM is the closest to an ideal channel in the presence of noise of all the ones we've tried.

OFDM is better than the transmission system from task 2A in almost every way. When comparing to Fig. 5 we can see that for the naive implementation from task 2A, percentage bit errors quickly increases to almost 30% when $|\alpha| \longrightarrow 1$, whereas the same metric tops out at below 6% for the OFDM-system under the same conditions. The OFDM-system has approximately $\frac{1}{5}$ as many errors as the system from task 2A at the same α. On the other hand; the downsides of an OFDM-system are one: The need to model the transmission channel. Two: A more complex to understand and power/-computation demanding implementation due to having a higher number of steps. Three: Potentially also longer latency due to needing a longer buffer to circularly convolve over, with the cyclic prefix. But in terms of bit detection error performance, as asked in the question, OFDM is clearly superior to the system from task 2A.

1. Need to model the transmission channel
2. Need to perform regular and inverse DFT
3. Increased processing/power requirements

Fig. 10 shows that our detection strategy works without errors for noise-free OFDM transmission.