ENG4052: Digital Communication 4 (2022-23)

Lab5: OFDM Communications Link

Haoshi Huang (2635088H)

Submission Date

29/03/2023

1 Introduction

Orthogonal Frequency Division Multiplexing (OFDM) is a type of **Multi-carrier Modulation (MCM)**. OFDM achieves high spectral efficiency by dividing data into multiple subcarriers which are orthogonal to each other, allowing multiple users to share the same frequency band. The modulation and demodulation of OFDM are based on IFFT and FFT respectively, which is one of the multi-carrier transmission schemes with the lowest implementation complexity and the widest application, as shown in Fig. 1 in AWGN communication channel.

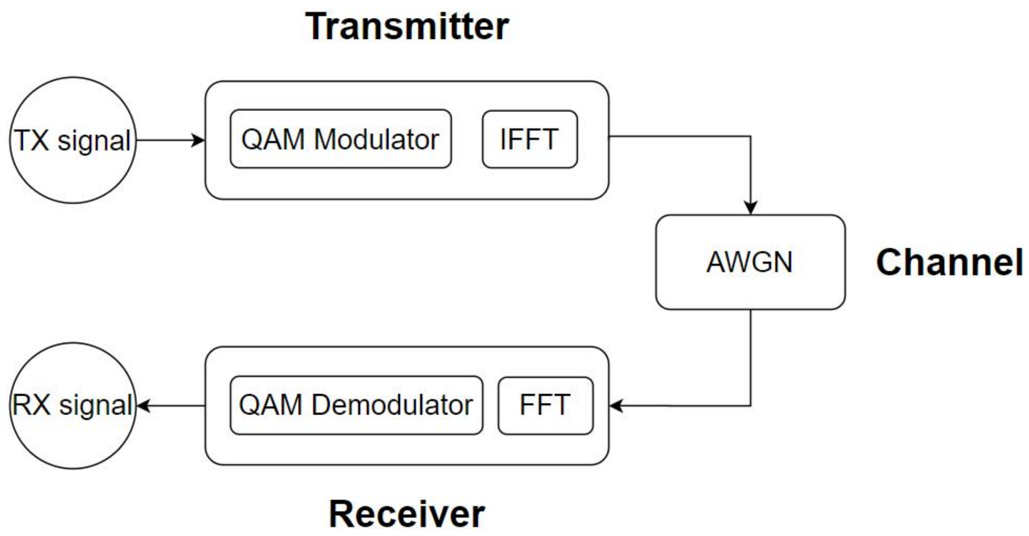

Figure 1: Communications Link

This lab will focus on **OFDM**, simulating a communications link with **Multipath interference** and incorporating **Reed-Solomon Channel Coding**.

2 OFDM Transmitter

2.1 Encoding signal with OFDM

With **Library Pyofdm**, we can configure OFDM object and use *encode method* to encode a set of random uint8 data. We count the length of encoded complex symbol and plot its real and imaginary components as shown in Fig. 2.1.

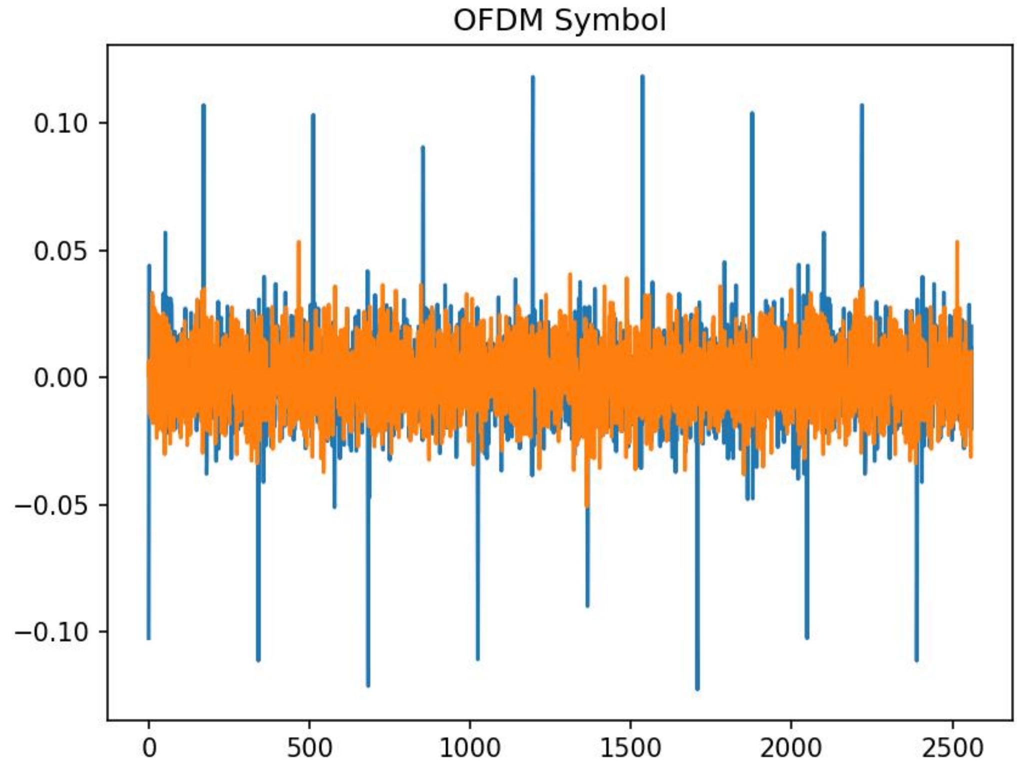

Figure 2.1: Real and Imaginary of OFDM symbol

We also plot the absolute value of Discrete Fourier Transform of the symbol without the prefix as shown in Fig. 2.2.

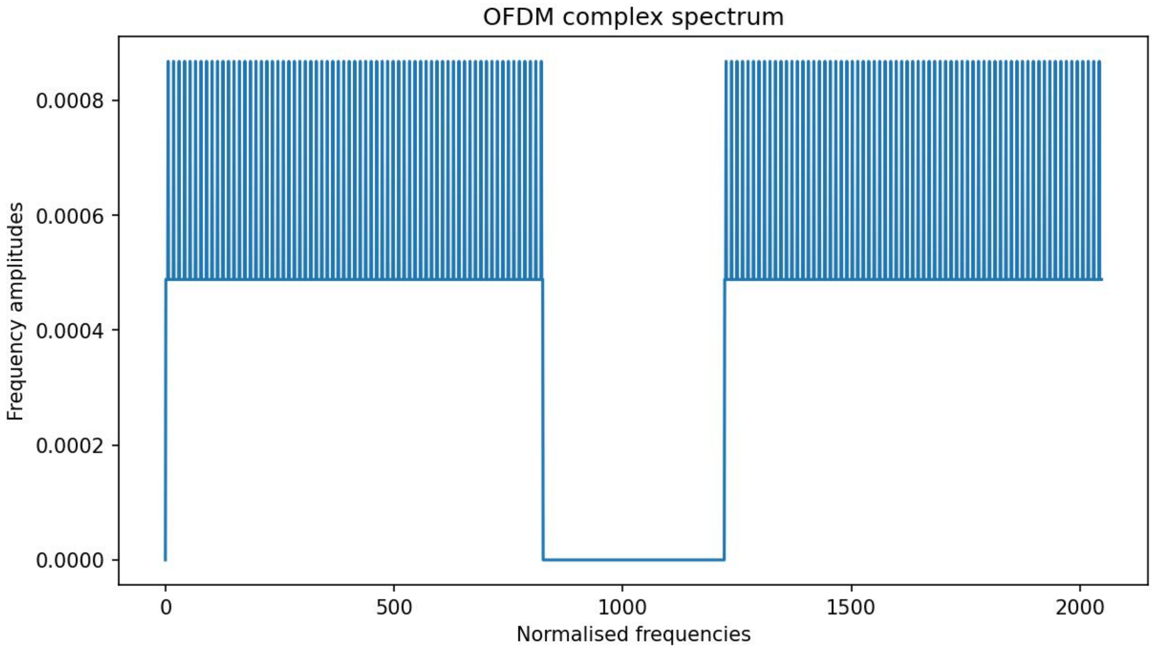

Figure 2.2: DFT of encoded symbol

Conclusion

From the program print result, we can get the length of the complex signal is **2560**, which is the sum of the number of subcarriers **2048** and the length of cyclic prefix, **512 = 2048 * 1/4**, namely **the quarter of the subcarriers count**. The cyclic prefix mainly acts as a protective band between consecutive symbols to overcome the **Inter-symbol Interference (ISI)**, which is the key technique to ensure OFDM correct operation, as shown in Fig. 2.3.

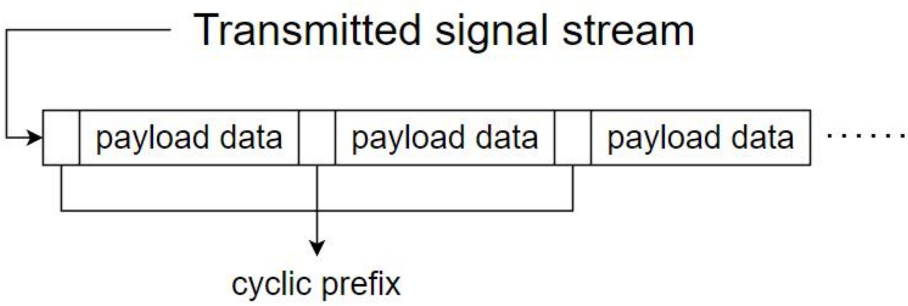

Figure 2.3: Constitution of OFDM symbol

According to the Fig. 2.1, we can draw the conclusion that all the positive and negative peaks of the complex symbol real components are distributed equally. It is because each subcarrier is modulated independently, which minimises interference.

According to the Fig. 2.2, we can draw the conclusion that the distribution of the occupied bandwidth is considerable flat in normalised frequency domain, which is consistent with the above-mentioned conclusion.

2.3 Source signal

In this lab, an image in PGM format will be used as source data as shown in Fig. 2.4. With Library Pillow, we can show the original image and read the image converting to binary data into a NumPy array. First, we pad some "random" length dummy zero data to the start of the signal **assuming the array length is 100**. Then, we use OFDM to encode and modulate the binary image data just like the random number array and save the result as a wav file.

Figure 2.4: Original image

3 OFDM Receiver

3.1 Demodulate OFDM signal

With **Library SciPy** to read wav file created in **Section 2.3**, we need to configure OFDM object by the same parameters. After padding some zeros to the signal of wave file, we demodulate the signal. The key task before decoding is to examine **the cross-correlation at subcarriers (cc)** and **the sum of the squares of the imaginary component of the expected pilot tones (sumofimag)**. In the programme, with Library Pyofdm, *findSymbolStartIndex method* includes the two steps and return **cc**, **sumofimag**. Fig. 3.1 & 3.2 plots them.

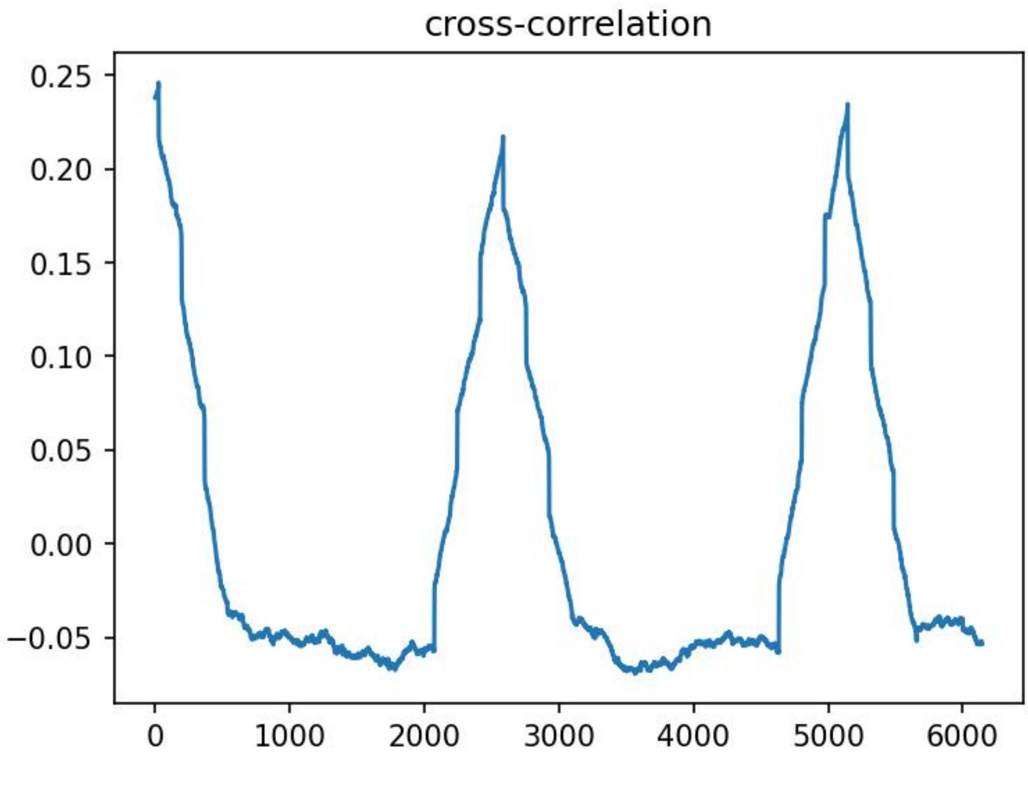

Figure 3.1: Cross-correlation

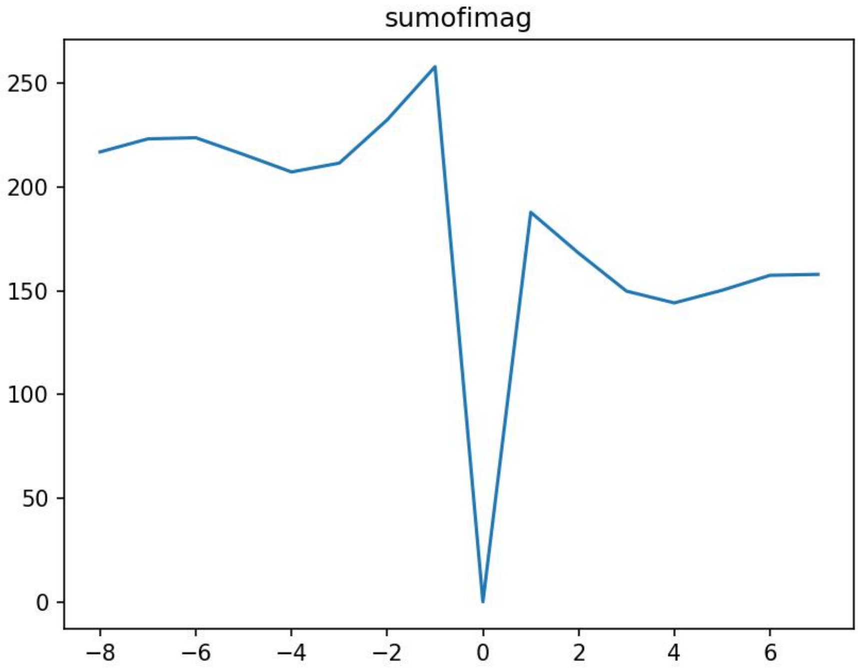

Figure 3.2: Sum of the squares of the imaginary component

Conclusion

The value of *offset* from *findSymbolStartIndex method* in the programme is equal to **2610**. From **Section 2.1**, the **length of the OFDM symbol** is **2560**, so the true value of offset should be **50 = 2610 – 2560 = 100 / 2**. In other word, the real offset is half of the assuming “random” length, namely **100**, of dummy zero data in **Section 2.3**. In the Fig. 3.4, the peaks locating at **50**, **2610** and **5170** indices are plotted by dashed red line. Also, all the **intervals** of these indices, which are **2560**, as the same as the length of the OFDM symbol.

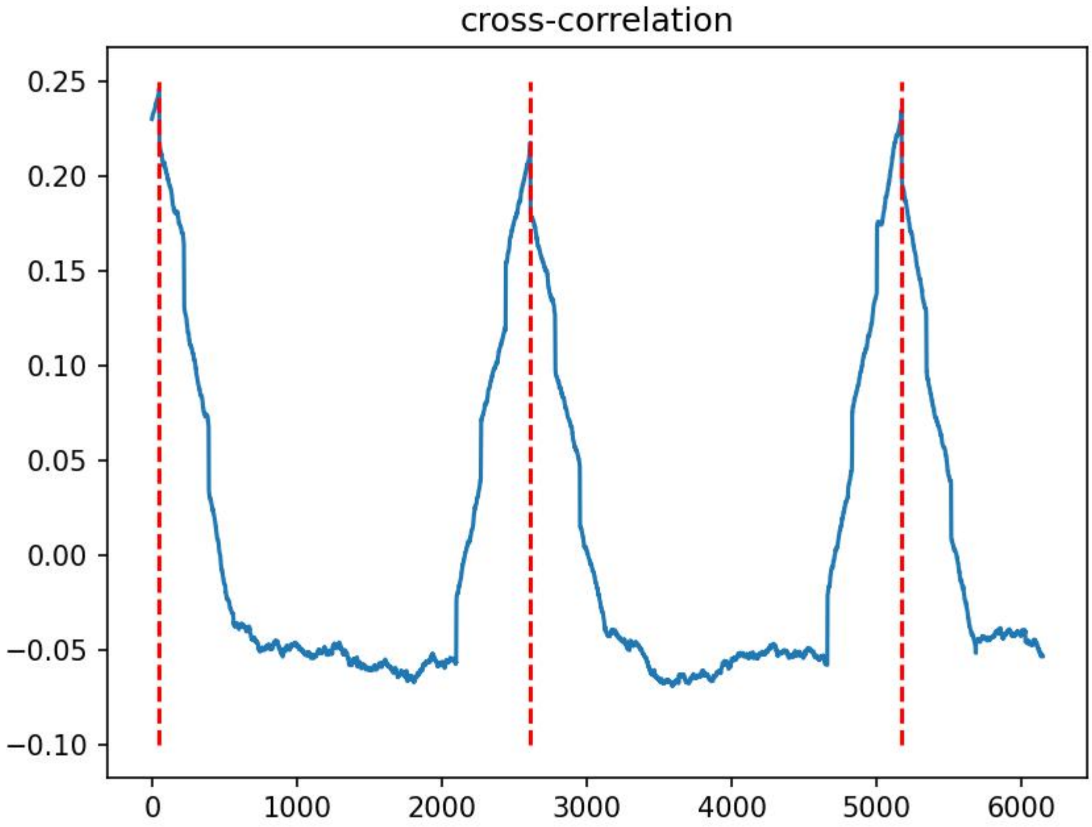

Figure 3.2: Peaks

Additionally, according to the Fig. 3.1, the sumofimag at zero index is much less than others.

3.2 Decode OFDM signal

By initialising the OFDM decoder, we can decode the demodulated OFDM signal, and compute the Bit Error Ratio (BER) in programme, which is equal to **zero** in theory. That is because there is no noise in communications channel so far and so is the print result. The received image is shown in the Fig. 3.3, which is the same as the original image.

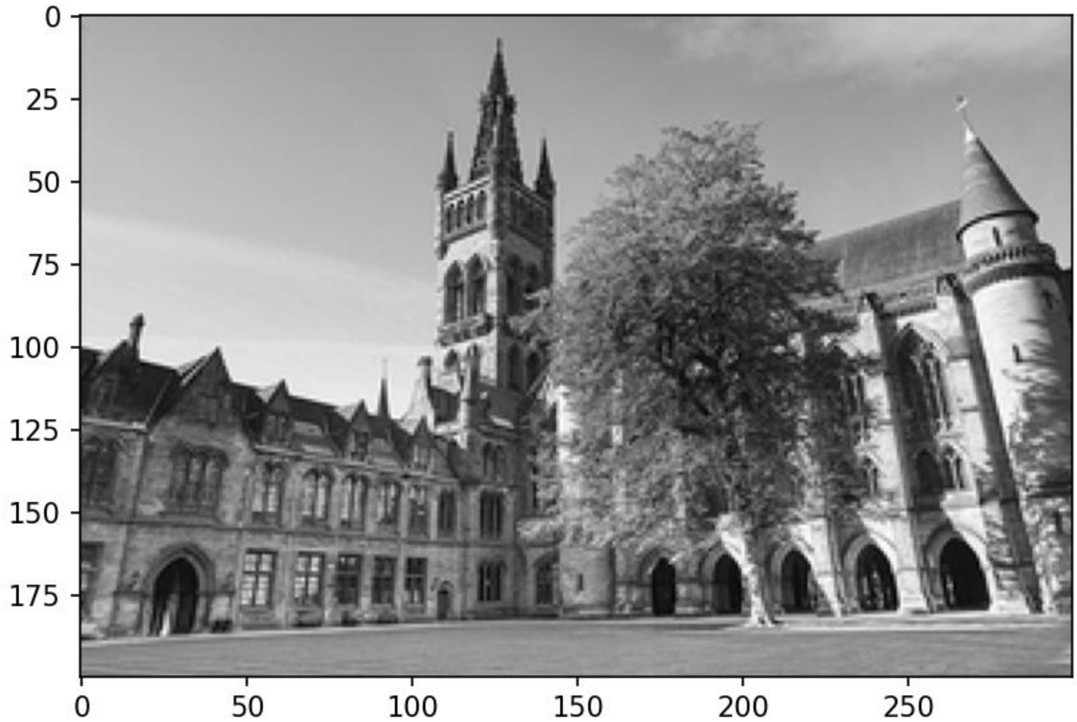

Figure 3.3: Received image

4 Distortion and Noise in the Communications Channel

In this task, I use Audacity, an audio processing tool, to create the following audio files from the original wave file, “ofdm44100.wave” by changing parameters.

1. Reverb 100%: BER is **37.355%**, received image is shown in the Fig. 4.1.
2. Reverb 70%: BER is **7.917%**, received image is shown in the Fig. 4.2.
3. Reverb 50%: BER is **3.928 %**, received image is shown in the Fig. 4.3.
4. Reverb 30%: BER is **0.6167%**, received image is shown in the Fig. 4.4.
5. Reverb 0%: BER is **0.1567%**, received image is shown in the Fig. 4.5.
6. Reverb 100%, damping 100%: BER is **1.032%**, image shown in the Fig. 4.6.
7. Reverb 100%, damping 70%: BER is **1.870%**, image shown in the Fig. 4.7.
8. Reverb 100%, damping 50%: BER is **4.160%**, image shown in the Fig. 4.8.
9. Reverb 100%, damping 30%: BER is **4.153%**, image shown in the Fig. 4.9.

10. Reverb 100%, damping 0%: BER is **7.915%**, image is shown in the Fig. 4.10.

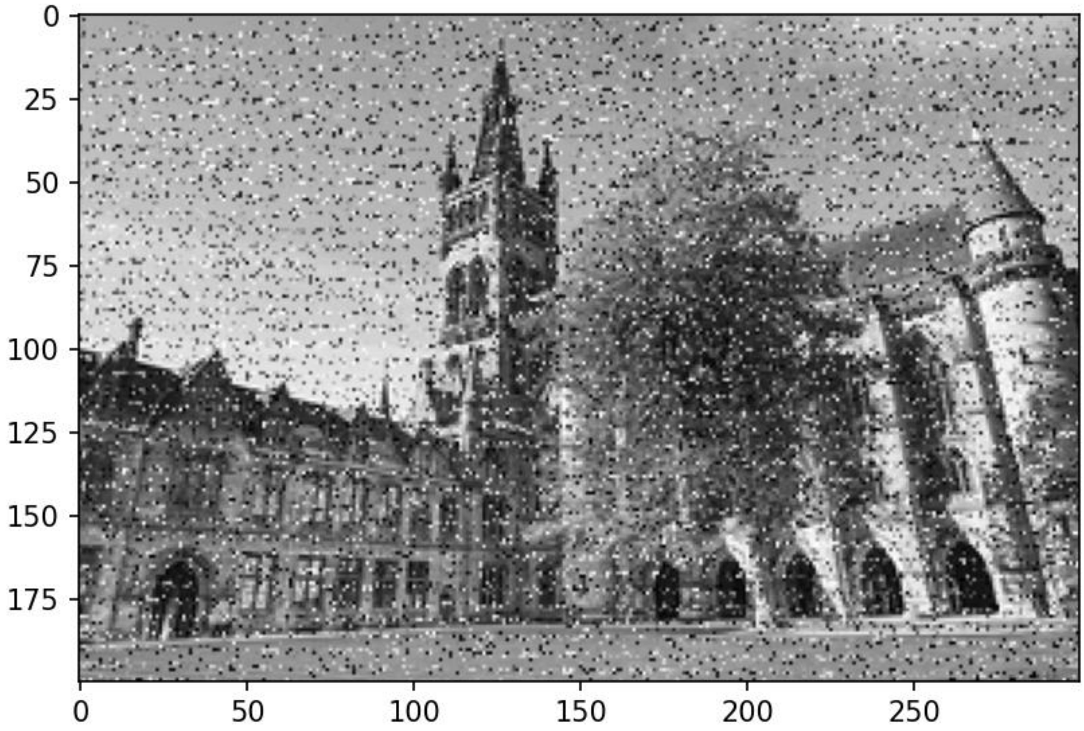

Figure 4.1: Reverb 100%   BER is 37.355%

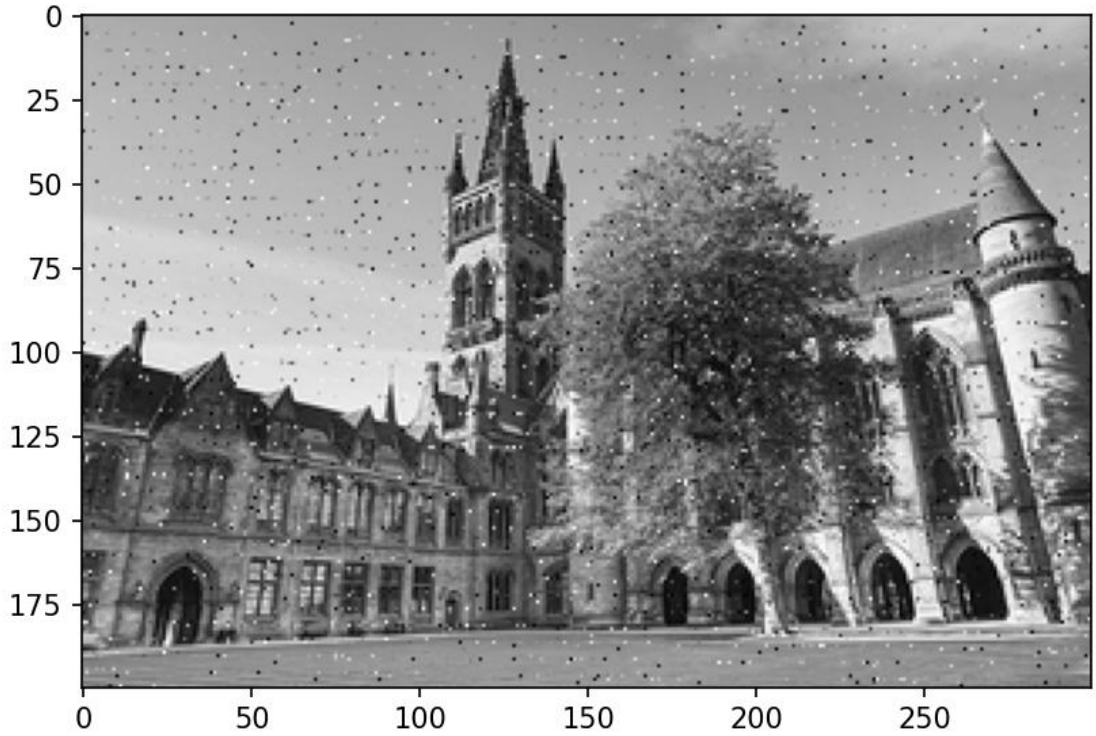

Figure 4.2: Reverb 70%   BER is 7.917%

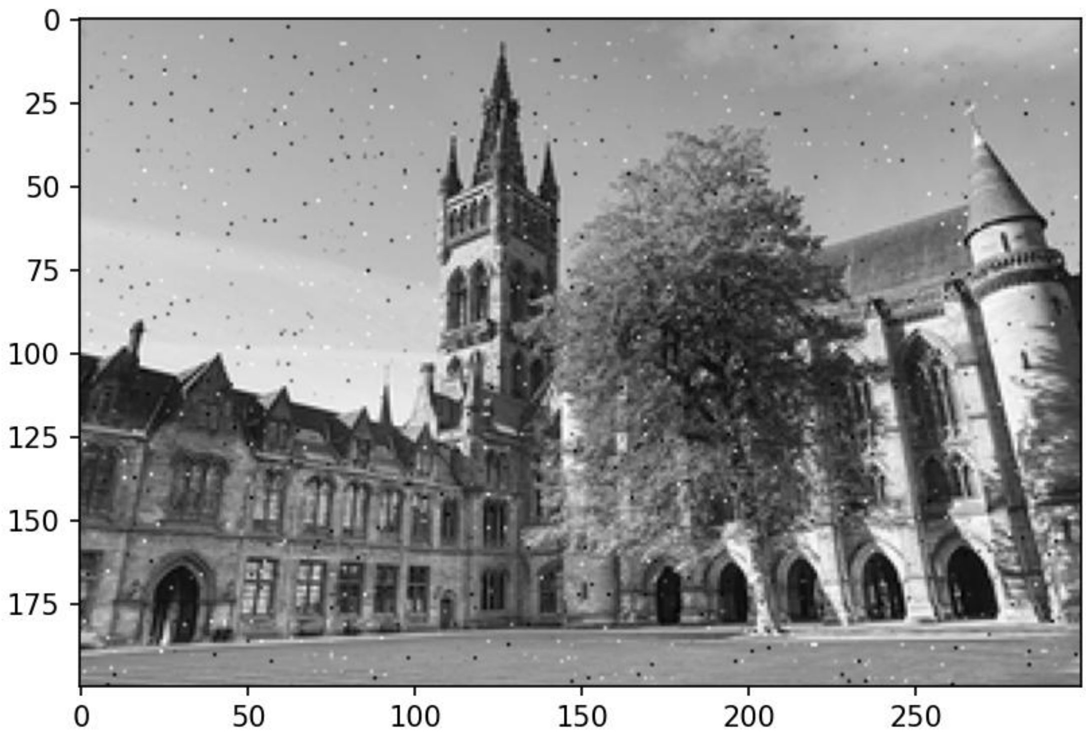

Figure 4.3: Reverb 50%   BER is 3.928 %

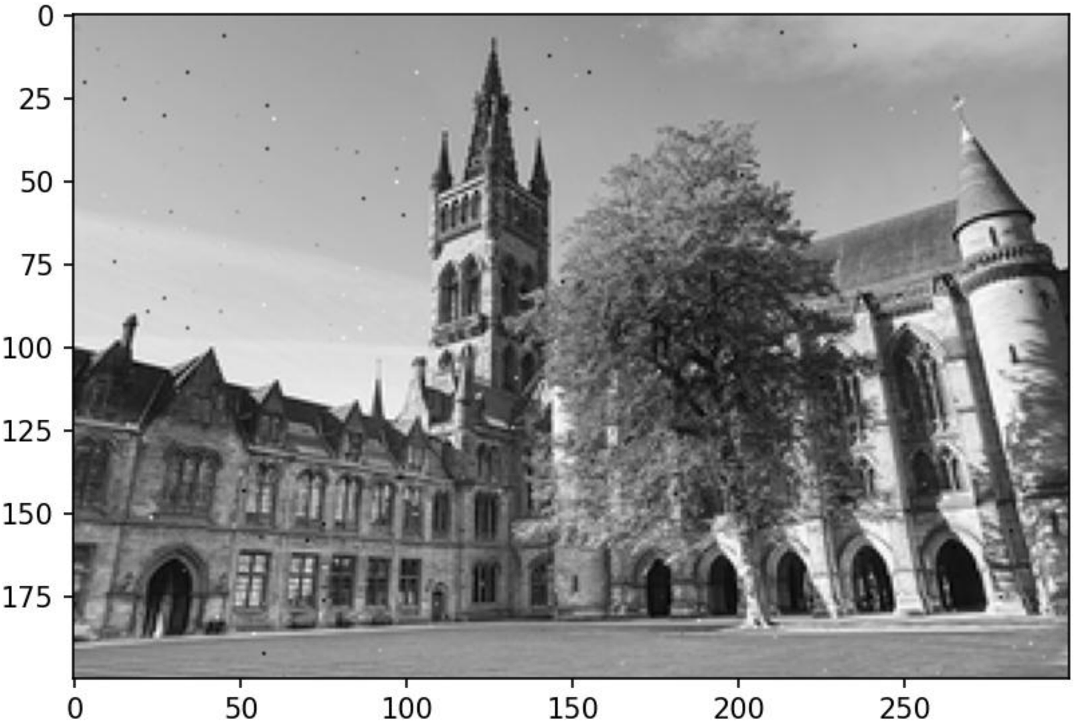

Figure 4.4: Reverb 30%   BER is 0.6167%

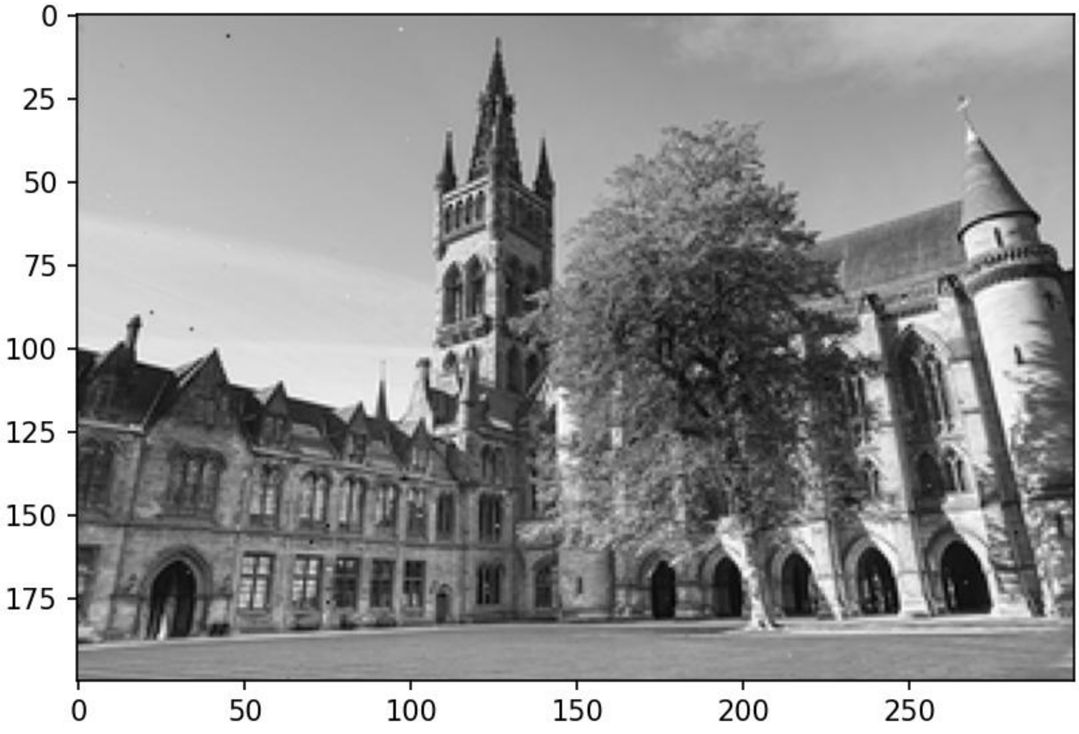

Figure 4.5: Reverb 0%   BER is 0.1567%

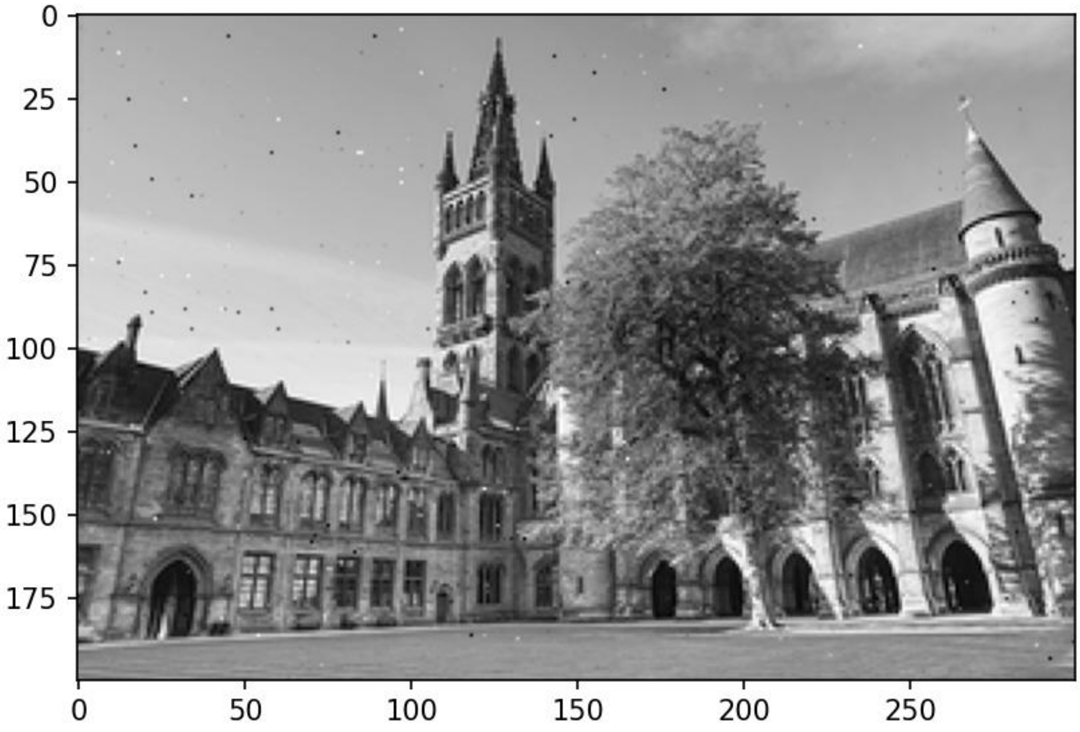

Figure 4.6: Reverb 100%, damping 100%   BER is 1.032%

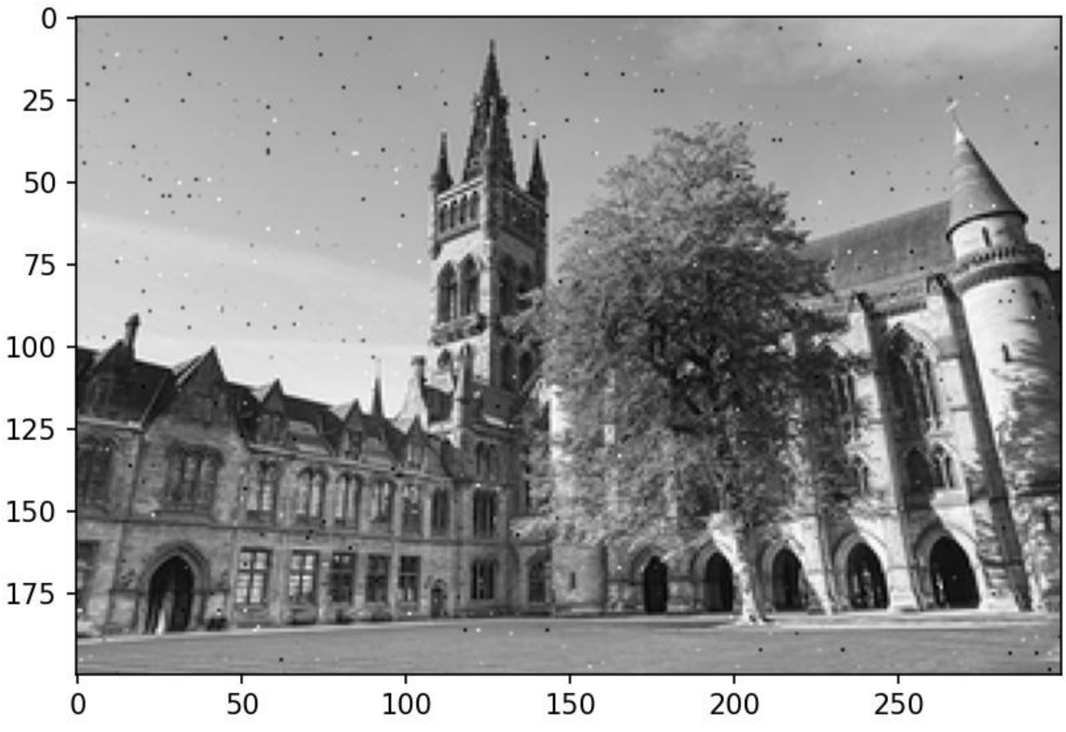

Figure 4.7: Reverb 100%,damping 70%   BER is 1.870%

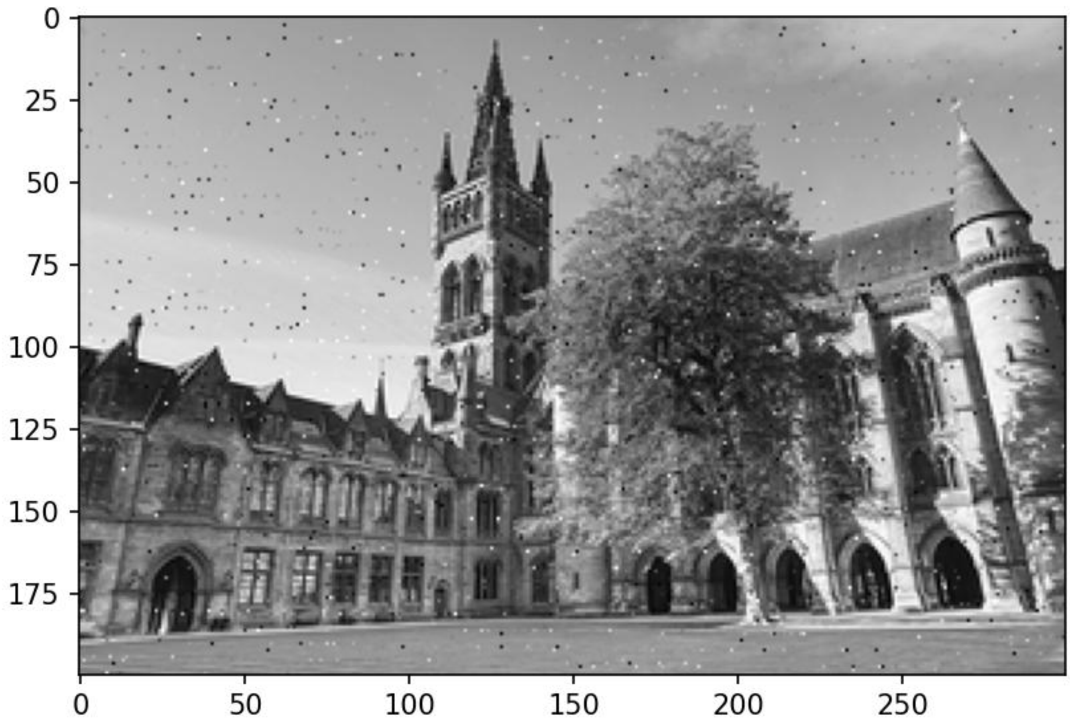

Figure 4.8: Reverb 100%,damping 50%   BER is 4.160%

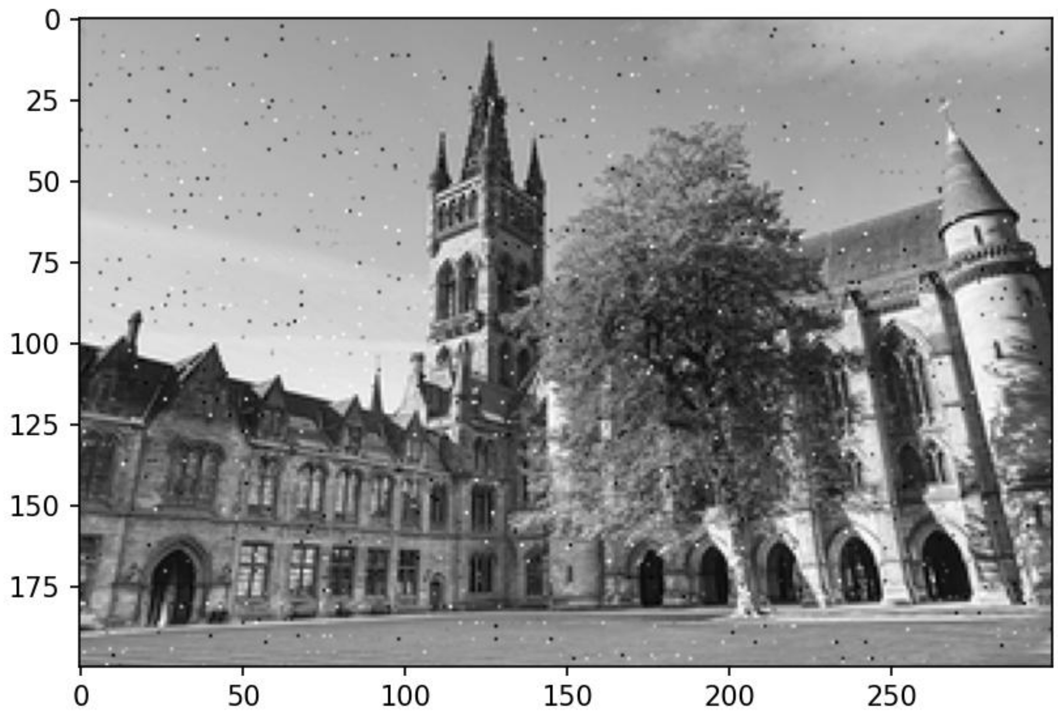

Figure 4.9: Reverb 100%,damping 30%   BER is 4.153%

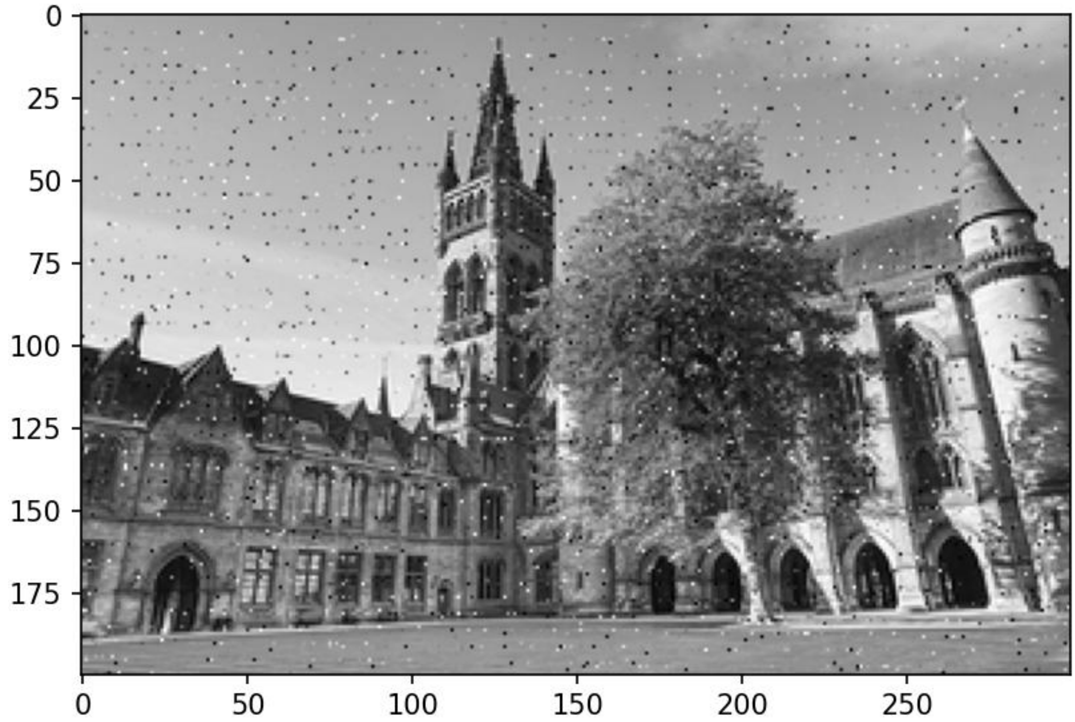

Figure 4.10: Reverb 100%,damping 0%   BER is 7.915%

Add noise, the original wave is -36.342dB

Setting the noise = 0.1A, as -24.7702dB, so **SNR = -11.572** = -36.342 – (-24.770), **BER = 98.158%**, as shown in the Fig. 4.11.

Setting the noise = 0.05A, as -30.7872dB, so **SNR = -5.555** = -36.342 – (-30.787) , **BER = 93.773%**, as shown in the Fig. 4.12

Setting the noise = 0.01A, as -44.769dB, so **SNR = 8.427** = -36.342 – (-44.769) , BER = **2.648%**, as shown in the Fig. 4.13.

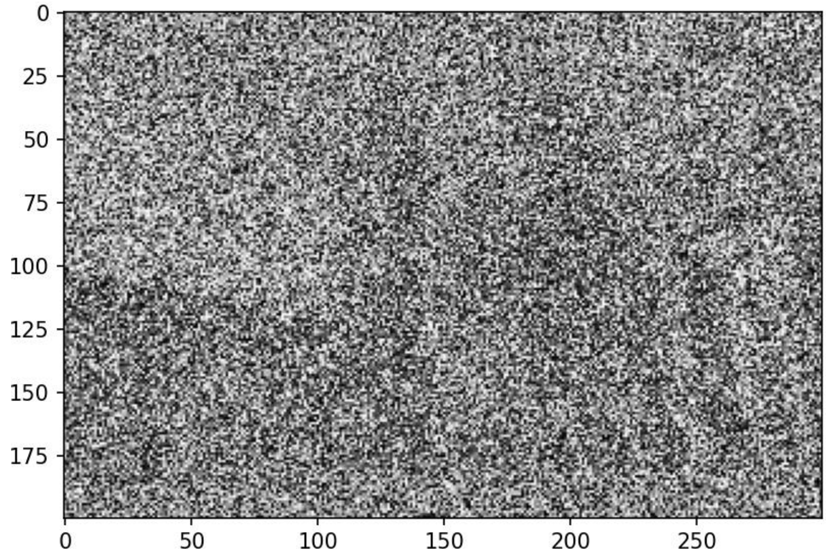

Figure 4.11: SNR is -11.572   BER is 98.158%

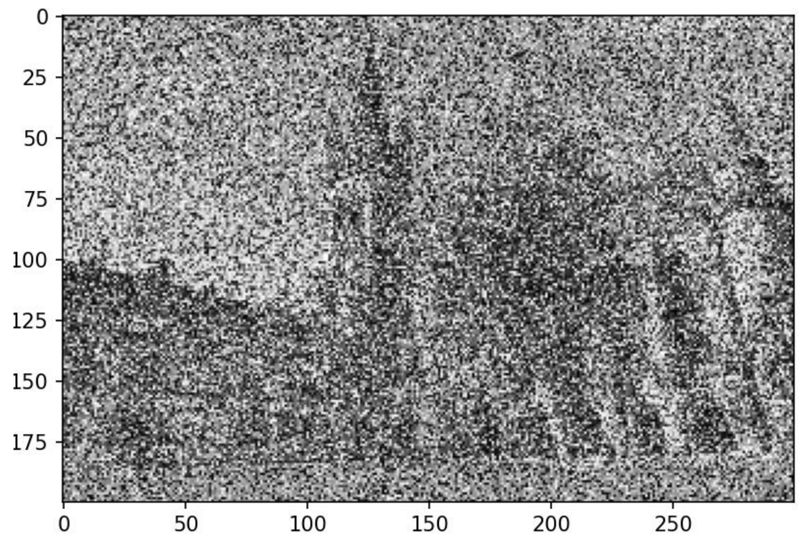

Figure 4.12: SNR is -5.555   BER is 93.773%

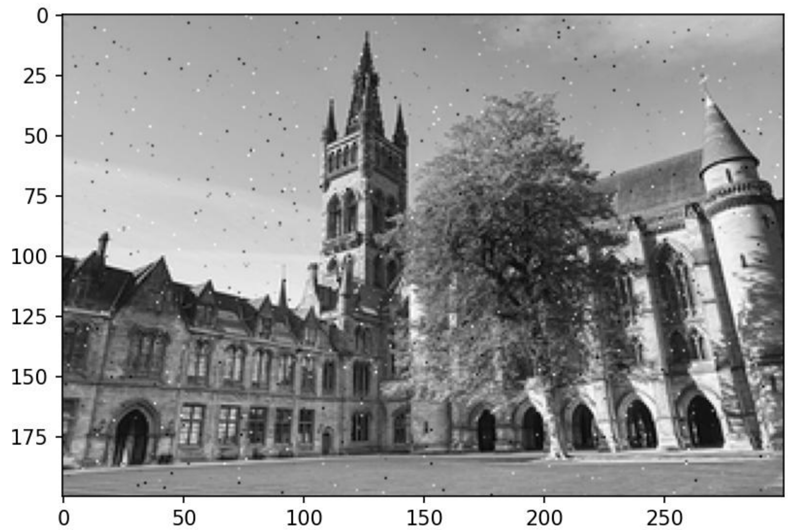

Figure 4.13: SNR is 8.427   BER is 2.648%

Conclusion

When other conditions are the same, the higher reverberation is, the more signal noise generated, namely the greater BER is.

When other conditions are the same, the lower damping is, the more signal noise generated, namely the greater BER is.

When other conditions are the same, the lower noise is, the higher SNR is, namely the greater BER is.

5 Reed-Solomon Channel Coding

The previous programme can be used in this task. We just need to using Reed-Solomon Channel coding. The Fig. 5.1 is the received image, which looks the same as the original image and BER = 0. So the conclusion can be drawn that RSC is a valid method to reduced BER.

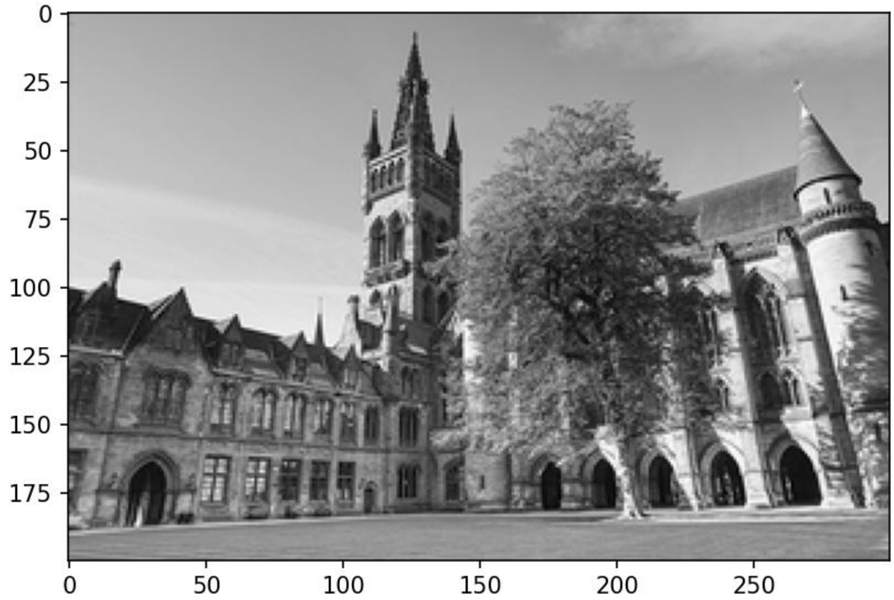

Figure 5.1: Received image using Reed-Solomon channel coding

Appendix:

OFDMTransmitter.py

from PIL import Image

import numpy as np

import scipy.io.wavfile as wav

import pyofdm.codec

```
import pyofdm.nyquistmodem
import matplotlib.pyplot as plt

# pyofdm.codec.OFDM(nFreqSamples=64,
#     pilotIndices=[-21, -7, 7, 21],
#     pilotAmplitude=1,
#     nData=12,
#     fracCyclic=0.25,
#     mQAM=2)

# Number of total frequency samples
totalFreqSamples = 2048
# Number of useful data carriers / frequency samples
sym_slots = 1512
# QAM Order
QAMorder = 2
# Total number of bytes per OFDM symbol
nbytes = sym_slots*QAMorder//8
# Distance of the evenly spaced pilots
distanceOfPilots = 12
pilotlist = pyofdm.codec.setpilotindex(nbytes, QAMorder, distanceOfPilots)

ofdm = pyofdm.codec.OFDM(pilotAmplitude = 16/9,
                                                    nData=nbytes,
                                                    pilotIndices = pilotlist,
                                                    mQAM = QAMorder,
                                                    nFreqSamples                  =
totalFreqSamples)

# Take a uint8 as a simple example
```

```
row = np.random.randint(256,size=nbytes,dtype='uint8')
complex_signal = ofdm.encode(row)
print("complex_signal num:", complex_signal.size)

# plot OFDM symbol
plt.figure()
plt.title('OFDM Symbol')
plt.plot(complex_signal.real)
plt.plot(complex_signal.imag)
plt.show()

# plot OFDM complec spectrum
plt.figure()
plt.title("OFDM complex spectrum")
plt.xlabel("Normalised frequencies")
plt.ylabel("Frequency amplitudes")
plt.plot(np.abs(np.fft.fft(complex_signal[-totalFreqSamples:])/totalFreqSamples))
plt.show()

# open image and binary information
# img = Image.open("./Lab5/DC4_600x400.pgm")
img = Image.open("./Lab5/DC4_300x200.pgm")
# plot imag
plt.figure()
plt.title('Original Image')
plt.imshow(np.array(img),cmap="gray",vmin=0,vmax=255)
plt.show()

# binary data
tx_byte = np.array(img).ravel()
```

```
## add some random length dummy zero data to the start of the signal here
pad_num = nbytes - tx_byte.shape[0] % nbytes
tx_byte = np.pad(tx_byte, (0, pad_num), mode="constant", constant_values=127)
##

# OFDM encoding
complex_img_signal = np.array([ofdm.encode(tx_byte[i:i+nbytes])
    for i in range(0,tx_byte.size,nbytes)]).ravel()

# modulation
base_signal = pyofdm.nyquistmodem.mod(complex_img_signal)

## add some random length dummy zero to the start of the signal
random_pad_length = 50
base_signal = np.pad(base_signal, (random_pad_length, 0), mode="constant")
##

# save it as a wav file
wav.write('./Lab5/ofdm44100.wav',44100,base_signal)
```

OFDMReceiver.py

```
from PIL import Image
import numpy as np
import scipy.io.wavfile as wav
import pyofdm.codec
import pyofdm.nyquistmodem
import matplotlib.pyplot as plt
```

```
# Number of total frequency samples
totalFreqSamples = 2048

# Number of useful data carriers / frequency samples
sym_slots = 1512

# QAM Order
QAMorder = 2

# Total number of bytes per OFDM symbol
nbytes = sym_slots * QAMorder // 8

# Distance of the evenly spaced pilots
distanceOfPilots = 12
pilotlist = pyofdm.codec.setpilotindex(nbytes, QAMorder, distanceOfPilots)

ofdm = pyofdm.codec.OFDM(pilotAmplitude=16/9,
                         nData=nbytes,
                         pilotIndices=pilotlist,
                         mQAM=QAMorder,
                         nFreqSamples=totalFreqSamples)

# read wav file to decode OFDM
samp_rate, base_signal = wav.read("./Lab5/ofdm44100.wav")

## append some extra zeros to the base_signal
extra_pad_length = 60
base_signal = np.pad(base_signal, (0, extra_pad_length), "constant")
##
```

```
complex_signal = pyofdm.nyquistmodem.demod(base_signal)

## find the start
searchRangeForPilotPeak = 8
cc, sumofimag, offset = ofdm.findSymbolStartIndex(complex_signal,
searchrangefine=searchRangeForPilotPeak)
print("Symbol start sample index =", offset)
##

## plot cross-correlation
plt.plot(cc)
plt.title("cross-correlation")
# plt.vlines([50, offset, 5170], -0.1, 0.25, linestyles='dashed', colors='red')
plt.show()

## plot sumofimag
search_range = np.arange(-searchRangeForPilotPeak, searchRangeForPilotPeak)
plt.plot(search_range, sumofimag)
plt.title("sumofimag")
plt.show()
##

Nsig_sym = 159

ofdm.initDecode(complex_signal, 25)
rx_byte = np.uint8([ofdm.decode()[0] for i in range(Nsig_sym)]).ravel()
rx_byte = 255 - rx_byte

rx_byte = rx_byte[:60000].reshape(200, 300)
```

```
receive_img = Image.fromarray(rx_byte)

plt.imshow(np.array(receive_img),cmap="gray",vmin=0,vmax=255)

# plt.imshow(receive_img, plt.cm.gray)

plt.show()


# calculate bit error ratio

# origin_img = Image.open("./Lab5/DC4_600x400.pgm")

origin_img = Image.open("./Lab5/DC4_300x200.pgm")

origin_img = np.array(origin_img)

# compute ber in practice

practiceBer = lambda tx_bin, rx_bin : np.sum([pix[0] != pix[1] for pix in zip(tx_bin, rx_bin)])
/ tx_bin.size

ber = practiceBer(origin_img, rx_byte)

# ber = np.sum(origin_img != receive_img) / origin_img.size

print("Bit error ratio : ", round(ber, 6))
```

DistortionNoise.py

```
from PIL import Image

import numpy as np

import scipy.io.wavfile as wav

import pyofdm.codec

import pyofdm.nyquistmodem

import matplotlib.pyplot as plt
```

```
# Number of total frequency samples
totalFreqSamples = 2048

# Number of useful data carriers / frequency samples
sym_slots = 1512

# QAM Order
QAMorder = 2

# Total number of bytes per OFDM symbol
nbytes = sym_slots * QAMorder // 8

# Distance of the evenly spaced pilots
distanceOfPilots = 12
pilotlist = pyofdm.codec.setpilotindex(nbytes, QAMorder, distanceOfPilots)

ofdm = pyofdm.codec.OFDM(pilotAmplitude=16/9,
                         nData=nbytes,
                         pilotIndices=pilotlist,
                         mQAM=QAMorder,
                         nFreqSamples=totalFreqSamples)

samp_rate, base_signal = wav.read("./Lab5/ofdm44100_reverb.wav")

# append some extra zeros to the base_signal
extra_pad_length = 60
base_signal = np.pad(base_signal, (0, extra_pad_length), "constant")

complex_signal = pyofdm.nyquistmodem.demod(base_signal)
```

```
# find the start of the OFDM symbol
searchRangeForPilotPeak = 8
cc, sumofimag, offset = ofdm.findSymbolStartIndex(complex_signal,
searchrangefine=searchRangeForPilotPeak)
print("Symbol start sample index =", offset)

Nsig_sym = 159
ofdm.initDecode(complex_signal, 25)
rx_byte = np.uint8([ofdm.decode()[0] for i in range(Nsig_sym)]).ravel()
rx_byte = 255 - rx_byte

rx_byte = rx_byte[:60000].reshape(200, 300)
receive_img = Image.fromarray(rx_byte)
plt.imshow(receive_img, plt.cm.gray)

# calculate bit error ratio
# origin_img = Image.open("./Lab5/DC4_600x400.pgm")
origin_img = Image.open("./Lab5/DC4_300x200.pgm")
origin_img = np.array(origin_img)
# ber = np.sum(origin_img != receive_img) / origin_img.size
# compute ber in practice
practiceBer = lambda tx_bin, rx_bin : np.sum([pix[0] != pix[1] for pix in zip(tx_bin, rx_bin)])
/ tx_bin.size
ber = practiceBer(origin_img, rx_byte)
print("Bit error ratio : ", ber)




def receive(wave_file):
```

```python
    samp_rate, base_signal = wav.read(wave_file)
    # append some extra zeros to the base_signal
    extra_pad_length = 60
    base_signal = np.pad(base_signal, (0, extra_pad_length), "constant")
    complex_signal = pyofdm.nyquistmodem.demod(base_signal)

    Nsig_sym = 159
    ofdm.initDecode(complex_signal, 25)
    rx_byte = np.uint8([ofdm.decode()[0] for i in range(Nsig_sym)]).ravel()
    rx_byte = 255 - rx_byte

    rx_byte = rx_byte[:60000].reshape(200, 300)
    receive_img = Image.fromarray(rx_byte)
    plt.imshow(np.array(receive_img),cmap="gray",vmin=0,vmax=255)
    plt.show()
    # plt.imshow(receive_img, plt.cm.gray)

    # calculate bit error ratio
    origin_img = Image.open("./Lab5/DC4_300x200.pgm")
    origin_img = np.array(origin_img)
    ber = practiceBer(origin_img, rx_byte)
    print("Bit error ratio = ", ber)


receive("./Lab5/ofdm44100_reverb.wav")

receive("./Lab5/ofdm44100_reverb100.wav")

receive("./Lab5/ofdm44100_reverb70.wav")
```

```
receive("./Lab5/ofdm44100_reverb50.wav")

receive("./Lab5/ofdm44100_reverb30.wav")

receive("./Lab5/ofdm44100_reverb0.wav")

receive("./Lab5/ofdm44100_damping100.wav")

receive("./Lab5/ofdm44100_damping70.wav")

receive("./Lab5/ofdm44100_damping50.wav")

receive("./Lab5/ofdm44100_damping30.wav")

receive("./Lab5/ofdm44100_damping0.wav")
receive("./Lab5/ofdm44100_noise0.1.wav")
receive("./Lab5/ofdm44100_noise0.05.wav")
receive("./Lab5/ofdm44100_noise0.01.wav")
```

Reed-SolomonChannelCoding.py

```
from PIL import Image
import numpy as np
import scipy.io.wavfile as wav
import pyofdm.codec
import pyofdm.nyquistmodem
import matplotlib.pyplot as plt

from reedsolo import RSCodec
```

```
from reedsolo import ReedSolomonError

N, K = 255, 223
rsc = RSCodec(N-K, nsize=N)

tx_im = Image.open("./Lab5/DC4_300x200.pgm")
tx_byte = np.append(np.array(tx_im, dtype="uint8").flatten(),
                    np.zeros(K-tx_im.size[1]*tx_im.size[0]%K, dtype="uint8"))
tx_enc = np.empty(0, "uint8")
for i in range(0, tx_im.size[1]*tx_im.size[0], K):
    tx_enc = np.append(tx_enc, np.uint8(rsc.encode(tx_byte[i:i+K])))

        # Number of total frequency samples
totalFreqSamples = 2048

# Number of useful data carriers / frequency samples
sym_slots = 1512

# QAM Order
QAMorder = 2

# Total number of bytes per OFDM symbol
nbytes = sym_slots * QAMorder // 8

# Distance of the evenly spaced pilots
distanceOfPilots = 12
pilotlist = pyofdm.codec.setpilotindex(nbytes, QAMorder, distanceOfPilots)

ofdm = pyofdm.codec.OFDM(pilotAmplitude=16/9,
                         nData=nbytes,
```

```
                              pilotIndices=pilotlist,
                              mQAM=QAMorder,
                              nFreqSamples=totalFreqSamples)



# append dummy bytes in order to make the data array is a whole multiple of nbytes
pad_num = nbytes - tx_enc.shape[0] % nbytes
tx_enc = np.pad(tx_enc, (0, pad_num), mode="constant", constant_values=127)

# OFDM encoding
complex_signal = np.array([ofdm.encode(tx_enc[i:i+nbytes])
                              for i in range(0, tx_enc.size, nbytes)]).ravel()

# modulate
base_signal = pyofdm.nyquistmodem.mod(complex_signal)

# add some random length dummy zero to the start of the signal
random_pad_length = 50
base_signal = np.pad(base_signal, (random_pad_length, 0), mode="constant")

# save it as a wav file
wav.write("./Lab5/ofdm44100_channel.wav", 44100, base_signal)

samp_rate, base_signal = wav.read("./Lab5/ofdm44100_channel.wav")

# append some extra zeros to the base_signal
extra_pad_length = 60
base_signal = np.pad(base_signal, (0, extra_pad_length), "constant")

complex_signal = pyofdm.nyquistmodem.demod(base_signal)
```

```
# find the start of the OFDM symbol
searchRangeForPilotPeak = 8
cc, sumofimag, offset = ofdm.findSymbolStartIndex(complex_signal, searchrangefine=searchRangeForPilotPeak)
print("Symbol start sample index =", offset)

Nsig_sym = 183
ofdm.initDecode(complex_signal, 25)
rx_enc = np.uint8([ofdm.decode()[0] for i in range(Nsig_sym)]).ravel()
rx_enc = 255 - rx_enc

rx_byte = np.empty(0, dtype="uint8")
for i in range(0, tx_im.size[1]*tx_im.size[0]*N//K, N):
    try:
        rx_byte = np.append(rx_byte, np.uint8(rsc.decode(rx_enc[i:i+N])[0]))
    except ReedSolomonError:
        rx_byte = np.append(rx_byte, rx_enc[i:i+K])

rx_byte = rx_byte[:60000].reshape(200, 300)
receive_img = Image.fromarray(rx_byte)
plt.imshow(receive_img, plt.cm.gray)

# calculate bit error ratio
origin_img = Image.open("./Lab5/DC4_300x200.pgm")
origin_img = np.array(origin_img)
ber = np.sum(origin_img != receive_img) / origin_img.size
print("Bit error ratio = ", ber)
```