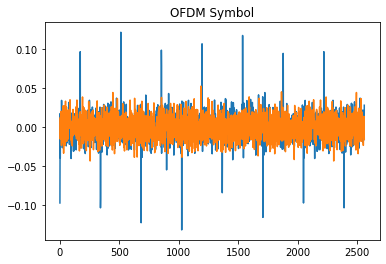
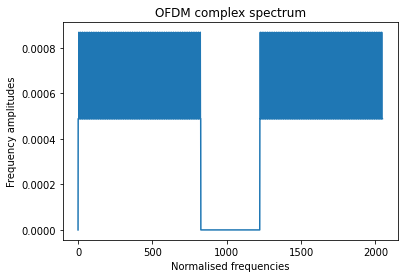
1.2 OFDM Transmitter

First, we encoded a single symbol using random data bytes, and plotted the real and imaginary components of encoded signal as follows.



The length of the encoded signal is 2560, which is the sum of 2048 and 512. 2048 is the number of subcarriers, and 512 is the length of the cyclic prefix, which is 1/4 of 2048. The real part of the encoded signal has equally spaced positive and negative peaks because the pilot carriers are evenly distributed.

We also plotted the absolute value of its discrete Fourier transform without the prefix. As we can see, the lines are relatively flat at the occupied bandwidth.

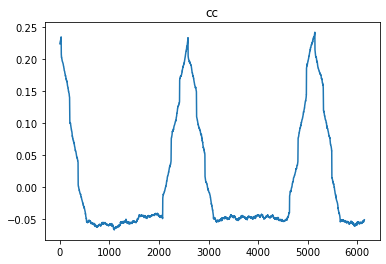


Then we encoded a 8-bit-depth grayscale image named DC4\_300x200.pgm, and added 50 bytes zero data to the double-sampled, real-valued result. Finally, we saved the result as a wav file named ofdm44100.wav.

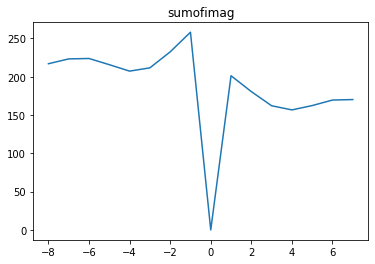
1.3 OFDM Receiver

First, we read the wav file mentioned above, and found the start of the OFDM symbol. Running the function findSymbolStartIndex, we found the value of offset is 2585. However, the length of an OFDM symbol is 2560, so the true value of offset should be 25, which is half of 50, length of the previously inserted dummy data in transmitter code.

The plot of cc is shown as below. The index of peak is also the value of offset, i.e. 25, 2585 and so on.



We also plotted the quantity sumofimag, and confirmed that the value at zero is substantially lower than the others.



Then, we initialized the OFDM decoder and determined the expected total number of OFDM symbols Nsig\_sym. The total number of bytes per OFDM symbol is 378 and the length of input signal is 200x300=60000, so the total number of OFDM symbols is ceil(60000/378)=159.

Finally, using the offset and Nsig\_sym, we decoded the OFDM symbols and display the result as an image. The received image is shown below and it matches the original image. The bit error ratio is zero.



1.3 Distortion and Noise in the Communications Channel

First, we opened the wav file using Audacity, and it sounded disorganized, like noise. We then added a reverb effect to the original audio, using the default settings of 50% reverberance and 50% damping. The resulting image is as follows, and the bit error ratio is 0.878%.



Then we adjusted the reverberance to 20% and 80% respectively, and the corresponding bit error ratios were 0.291% and 2.045% respectively. It can be concluded that the higher the reverberance, the greater the impact on transmission and the higher the bit error ratio.

On the premise of maintaining the reverberance of 50%, we adjusted the damping to 20% and 80% respectively, and got the bit error ratios of 1.303% and 0.625%. This means that the lower the damping, the higher the bit error ratio.

We also explored the effects of additive white noise. First of all, we added white noise with a amplitude of 0.8, corresponding to a signal-to-noise ratio of -29.63db, which is very low. Therefore, the final bit error ratio is 99.5%, as shown in the image below.



Then we try to add a smaller amplitude of noise. Noise of 0.05 amplitude corresponds to a signal-to-noise ratio of -5.555db, with the bit error ratio of 93.8%. Noise of 0.015 amplitude corresponds to a signal-to-noise ratio of 4.9127db, with the bit error ratio of 25.11%. The signal-to-noise ratio of 0.01 amplitude noise is 8.432db, and the bit error ratio decreases to 2.61%. 0.003 amplitude noise corresponds to a signal-to-noise ratio of 18.892db and zero bit error ratio.

1.5 Reed-Solomon Channel Coding

We repeated the code in 1.2 and 1.3, using Reed-Solomon channel coding. The recovered image is as follows, which is identical to the original image. Therefore, the bit error ratio is zero.



In the cases of white noise, compared to the RSC(255,223) QPSK plot in the lecture notes, when snr approached 5db, ber was less than 10% with RSC, but close to 25% without RSC. When the snr was close to 8db, ber was 0.1% with the RSC, but close to 2.5% without the RSC. It can be concluded that ber could be effectively reduced with the RSC.

Appendix

1.2

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)

row = np.random.randint(256, size=nbytes, dtype="uint8")

complex\_signal = ofdm.encode(row)

plt.figure()

plt.title("OFDM Symbol")

plt.plot(complex\_signal.real)

plt.plot(complex\_signal.imag)

plt.show()

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()

# read image

img = Image.open("DC4\_300x200.pgm")

tx\_byte = np.array(img).ravel()

# append dummy bytes in order to make the data array is a whole multiple of nbytes

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\_signal = np.array([ofdm.encode(tx\_byte[i:i+nbytes])

for i in range(0, tx\_byte.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("ofdm44100.wav", 44100, base\_signal)

1.3

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("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 of the OFDM symbol

searchRangeForPilotPeak = 8

cc, sumofimag, offset = ofdm.findSymbolStartIndex(complex\_signal, searchrangefine=searchRangeForPilotPeak)

print("Symbol start sample index =", offset)

plt.plot(cc)

plt.title("cc")

plt.show()

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(receive\_img, plt.cm.gray)

# calculate bit error ratio

origin\_img = Image.open("DC4\_300x200.pgm")

origin\_img = np.array(origin\_img)

ber = np.sum(origin\_img != receive\_img) / origin\_img.size

print("Bit error ratio = ", ber)

1.4

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)

def receive(wave\_file):

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(receive\_img, plt.cm.gray)

# calculate bit error ratio

origin\_img = Image.open("DC4\_300x200.pgm")

origin\_img = np.array(origin\_img)

ber = np.sum(origin\_img != receive\_img) / origin\_img.size

print("Bit error ratio = ", ber)

receive("ofdm44100\_reverb.wav")

receive("ofdm44100\_reverb\_B20.wav")

receive("ofdm44100\_reverb\_B80.wav")

receive("ofdm44100\_reverb\_M20.wav")

receive("ofdm44100\_reverb\_M80.wav")

receive("ofdm44100\_white\_noise0.8.wav")

receive("ofdm44100\_white\_noise0.05.wav")

receive("ofdm44100\_white\_noise0.015.wav")

receive("ofdm44100\_white\_noise0.01.wav")

receive("ofdm44100\_white\_noise0.003.wav")

1.5

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("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("ofdm44100\_channel.wav", 44100, base\_signal)

samp\_rate, base\_signal = wav.read("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("DC4\_300x200.pgm")

origin\_img = np.array(origin\_img)

ber = np.sum(origin\_img != receive\_img) / origin\_img.size

print("Bit error ratio = ", ber)