ECE4634 - Digital Communications
November 30, 2020
Nima Mohammadi
nimamo@vt.edu

Course Project
Technical Memorandum

I. Introduction

In this technical report we will design a digital communication channel for transmission of static color images. For this project we use the CIFAR10 image dataset that is widely known within the Machine Learning community. For source coding we design and implement an Autoencoder that is trained on 60000 images. We investigate different configurations of this neural network and choose a setting that achieves adequate results. The images compressed by the Autoencoder are then quantized via a uniform quantization scheme.

In this project, we investigate both AWGN and frequency-flat Rayleigh fading channels. For transmission of the signal we use 16QAM OFDM and show that our SNR vs. BER results match the theoretical results. To combat the impact of fading and allow for more resiliency we also extend our system with channel encoding. We use a Convolutional encoder at the Tx and a Viterbi decoder at the Rx side. Two different coding rates, namely 1/2 and 1/4 are tested. We show that bit interleaving is crucial for this channel encoder to act on the fading channels and discuss about framing when dealing with channel encoding.

II. Source coding

An important part of this project which took a significant part of my time was source coding. We use an Autoencoder to perform image compression. You can think of Autoencoder as

a nonlinear PCA that performs dimensionality reduction. I have designed and implemented an Autoencoder with the architecture depicted in Table 1.

As it can be seen from Table 1, we have parameterized the three convolutional (and decorvolutional) kernels. We have fixed the number of kernels in the third bottleneck layer (last layer of the encoder) named ker3 to 24, to make sure all of our networks compress the images to the same number of values, namely from $3 \times 32 \times 32$ to $24 \times 4 \times 4$, that is reducing from 3072 to 384, an impressive 8-fold compression. However, for ker1 and ker2 we investigate three values for each, namely 3, 6, and 12 kernels.

All Autoencoders were trained on 60000 training images from the CIFAR10 dataset for the same number of iterations (i.e. 40 epochs). The performance of all of these nine configurations are are reported in Table 2. The number of parameters (weights) in the encoder and decoder modules of the Autoencoder are reported in Table 2 along the MSE and the SNR they achieve on 10000 unseen images from the same dataset.

```
Autoencoder(
  (encoder): Sequential(
    (0): Conv2d(3, ker1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(ker1, ker2, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): Conv2d(ker2, ker3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): Sigmoid()
  )
  (decoder): Sequential(
    (0): ConvTranspose2d(ker3, ker2, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): ConvTranspose2d(ker2, ker1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): ReLU()
    (4): ConvTranspose2d(ker1, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (5): Sigmoid()
  )
)
```

Table 1: Architecture of the Autoencoder

(ker1, ker2, ker3)	# Encoder Params	# Decoder Params	MSE (unseen data)	SNR (unseen data)
(3, 3, 24)	1470	1449	0.0109	14.2491
(3, 6, 24)	2769	2748	0.0097	14.7454
(3, 12, 24)	5367	5346	0.0041	18.4609
(6, 3, 24)	1761	1740	0.0081	15.5679
(6, 6, 24)	3204	3183	0.0061	16.7866
(6, 12, 24)	6090	6069	0.0032	19.6340
(12, 3, 24)	2343	2322	0.0090	15.0832
(12, 6, 24)	4074	4053	0.0051	17.5696
(12, 12, 24)	7536	7515	0.0031	19.7785

Table 2. Performance of different realizations of the Autoencoder

From Table 2 we can see that both (6, 12, 24) and (12, 12, 24) Autoencoders achieved the best performance, with (12, 12, 24) being negligibly the superior one. However, we choose the (6, 12, 24) configuration as it requires less parameters to do the same job. Notice that the encoder (decoder) weights are shared with the transmitter (receiver) prior to the transmission. In a more advanced setup, we may augment our system to transfer the decoding weights to the receiving side in a preliminary stage which could potentially enable adaptive fidelity in our system.

The reconstructed images for each of configurations are depicted in Fig. 1. The images have not been seen by the Autoencoder during training, meaning that the neural network has not memorized the images and is indeed capable of generalizing its compression capability. This figure confirms our results based on MSE and SNR and assures us that (**6, 12**, 24) is a relatively accurate image compressor. Notice that this is a proof of concept design and in practice we could come up with more sophisticated designs. From the reconstructed images, we can conclude that with more weights/parameters in our Autoencoder we would get higher accuracy. This is however not a surprising observation. Moreover, we observe from our result that having more kernels at the first layer is preferred to the subsequent layers. This is compatible with our intuition, as the neural networks work based on representation learning and a gradual decrease in

Fig. 1. Sample reconstructed images from the compressed presentations at each configurations

dimensionality at each layer is deemed very important in their performance. Interestingly, for three kernels at the first layers it seems that the Autoencoder had difficulty conveying color information.

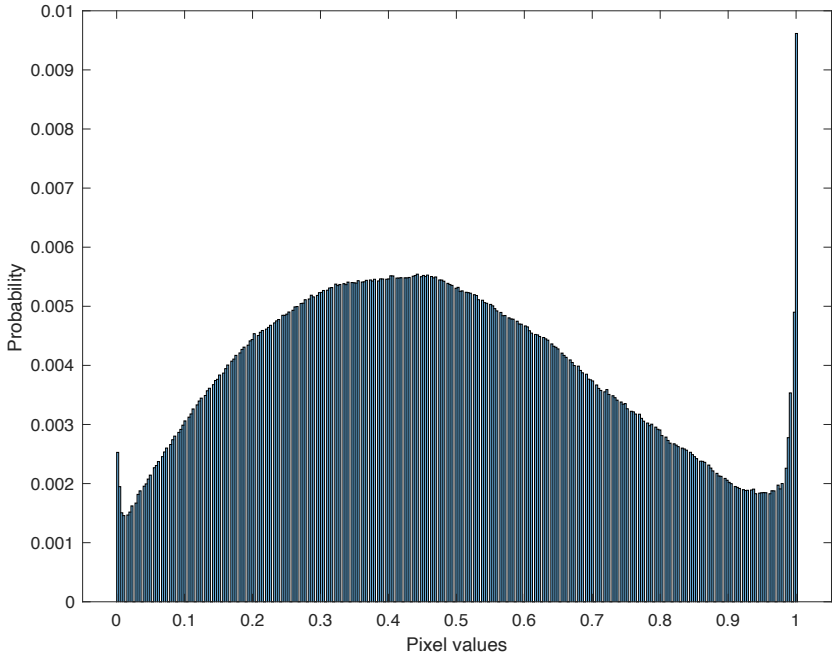

Fig. 2. Histogram of pixel values in CIFAR10

As a technical note, I have implemented the Autoencoder using Python and PyTorch. I have run the code using GPUs on the cloud (via Google Colaboratory service). Then stored trained weights and downloaded them to my local laptop. Next I wrote a MATLAB script that allowed me to load the latent representation (the compressed images) into MATLAB, perform my simulations that resulted in modifications of those latent variable (due to ADC and noise), and then invoke the Python code from MATLAB to reconstruct the images from the "received" signal. This procedure took a significant amount of time for it not to be mentioned!

III. Quantization

Although the Autoencoder has done an impressive dimensionality reduction our images are still "analog". In Fig. 2, the empirical PDF (histogram) of pixel values in the CIFAR10 dataset is plotted. With this information, one may attempt to implement a quantization scheme that exploited the underlying data generating process for more accuracy. However, this is not the aim of this project and I opt to simply use the uniform/linear quantization. The nonlinear $\mu-$law scheme has not been considered since it was originally designed for voice and I expect it to be inferior for our use case of natural images.

Performing uniform quantization with different bits/levels of quantization is reported in Table 3. We observe that for quantization bits more than six bits we do not gain any more meaningful adequacy. The resulting images are also shown in Fig. 3 which visually confirms our findings. Interestingly, for 2-bit quantization it retained an impressive amount of information.

Fig. 3. Sample reconstructed images of (6, 12, 24) autoencoder with different quantization levels

Also for 1-bit quantization we can see a repetitive pattern that is an implication of the convolution layers in our neural network.

(ker1, ker2, ker3)	Quant bits	Quant levels	MSE (unseen data)	SNR (unseen data)
(6, 12, 24)	1	2	0.1424	3.0914
(6, 12, 24)	2	4	0.0890	5.1323
(6, 12, 24)	3	8	0.0288	10.0366
(6, 12, 24)	4	16	0.0089	15.1578
(6, 12, 24)	5	32	0.0044	18.1603
(6, 12, 24)	6	64	0.0035	19.2302
(6, 12, 24)	7	128	0.0032	19.5251
(6, 12, 24)	8	256	0.0032	19.6066

Table 3. Performance for different quantization levels

So far, we made the decisions about our source encoder by choosing among the Autoencoder configurations and the quantization levels, namely (6, 12, 24) and 64, respectively. In the rest of the paper, we aim to transmit 500 images. The reference MSE and SNR for these 500 images, prior to transmission, are 0.0035 and 19.35dB respectively.

IV. Modulation

For modulation we employ a combination of 16QAM and OFDM (orthogonal frequency-division multiplexing) which is a popular multi-carrier system used in many technologies including LTE and 5G. OFDM decomposes the transmission frequency band into a group of narrower contiguous subbands (carriers), and each carrier is individually modulated. By using narrow orthogonal subcarriers, the OFDM signal gains robustness over a frequency-selective fading channel and eliminates adjacent subcarrier crosstalk. Fig. 4 shows how OFDM in time and frequency domains is compared to a single carrier modulation scheme. Since the multiple data streams can be transmitted simultaneously with multiple carriers, OFDM is not influenced by noise to the same degree as single-carrier modulation.

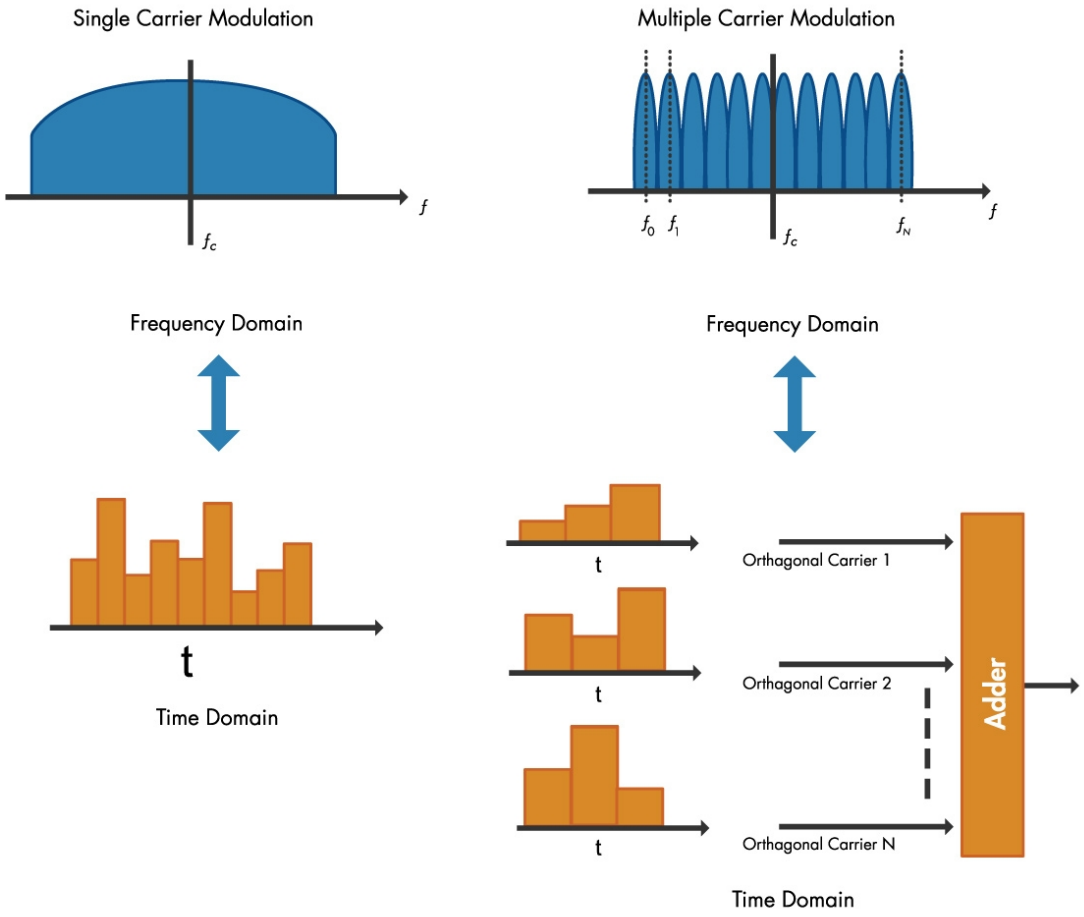

Fig. 4. Single carrier modulation and OFDM in time and frequency domains.

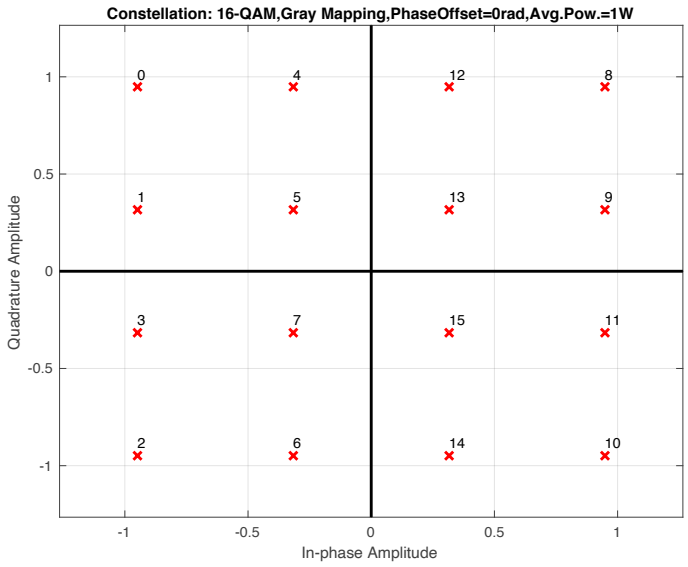

Fig. 5. Constellation diagram for 16QAM

In OFDM, when the amplitude of each subcarrier reaches the maximum, the carriers are arranged at intervals of 1 / symbol time so that the amplitude of other subcarriers is 0, thereby preventing interference between symbols. Moreover, multicarrier OFDM transmission is effective in multipath environments because the influence of multipath is concentrated on specific subcarriers compared with a single-carrier transmission. In the case of a single-carrier transmission, the multipath affects the whole.

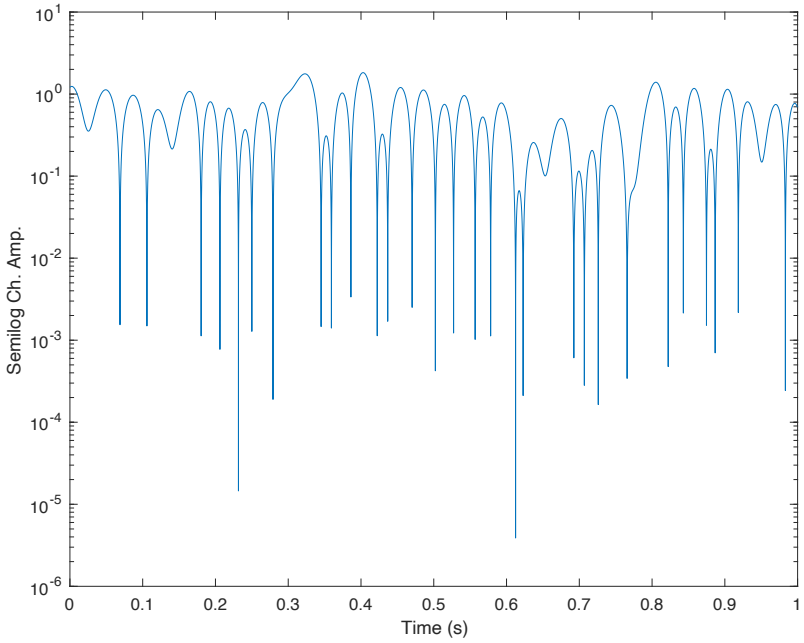

Fig. 6. Fading channel amplitude during the first second of transmission

The constellation diagram of 16QAM used in this project is depicted in Fig. 5. As it can be seen from this diagram, I have normalized the signal constellation to the average power. In this project I use an OFDM modulator with a 128 Fast Fourier Transform (FFT) points, that is 128 subcarriers, with length of the cyclic prefix being equal to 32. At each frame of OFDM, 117 16QAM symbols (i.e. 468 bits) are transmitted.

V. Channel and Channel Coding

After modulation, 16QAM and OFDM, the signal is passed through an additive white Gaussian noise channel (AWGN) and also a multi-path Rayleigh fading channel. For the fading channel, however, I have not used the frequency-selective setting and opted for the frequency-flat case. The maximum doppler shift across all path is set to 0.001Hz (default MATLAB comm toolbox setting).

For 500 compressed images of size $24 \times 4 \times 4$and 6-bit quantization the total length of the message is $500 \times (24 \times 4 \times 4) \times 6 = 1152000$ bits. With a sampling rate of 64kHz the transmission of the image takes about $17.57s$ to transmit the images. We are using 16QAM, meaning that we have $1152000/4 = 288000$ symbols to transmit. At each instance, 117 of these symbols are multiplexed together via OFDM and transmitted, that is $\lceil 288000/117 \rceil = 2462$ frames. To get a sense of how the Rayleigh channel is behaving, the channel amplitude during the transmission of the first second (140 OFDM frames / 16520 16QAM symbols) are depicted in Fig. 6. The bandwidth containing 99% of the 16QAM single-carrier and OFDM multi carrier transmitted signal was found to be 32362Hz and 29900Hz. The ESD of these modulations are depicted in Fig 7.

The simulation results for different SNRs are plotted in Fig. 8. The SNR vs. BER pairs are calculated for transmission of the 500 images and as shown in the plot, closely matches the

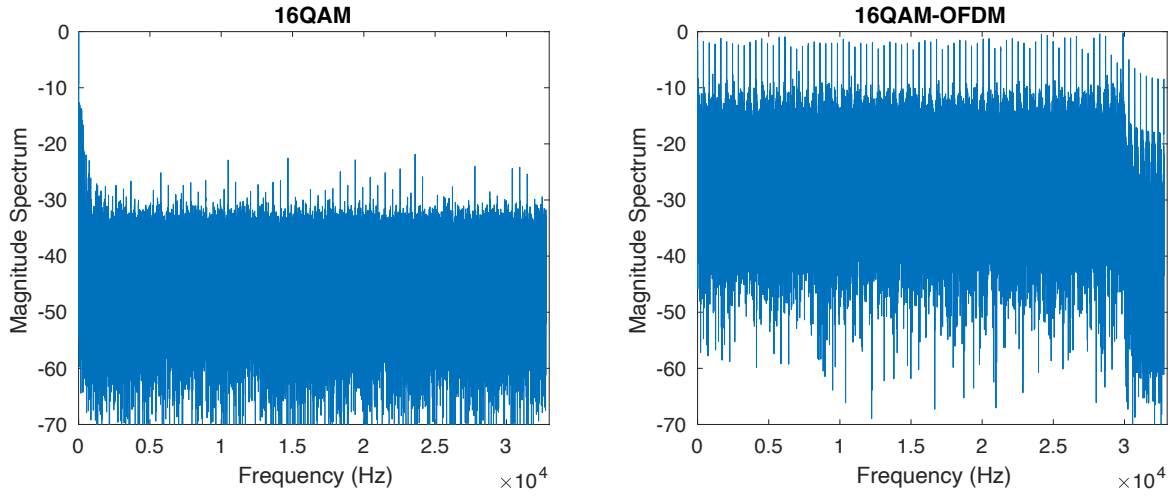

Fig. 7. Energy Spectral Density of 16QAM and 16QAM-OFDM modulated signal

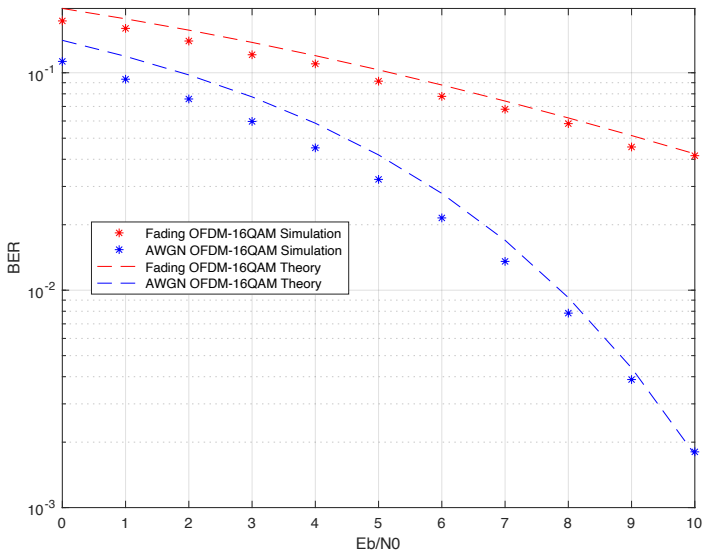

Fig. 8. SNR (dB) vs. BER plot of 16QAM-OFDM for transmission of the 500 images closely matched the theoretical values

theoretical values. We observe that the BER of the AWGN channel quickly decline whereas thereof the fading channel declines with with a less steep slope.

For channel encoding I use Convolutional encoder at the transmitting side and Viterbi decoder on the receiving side. I have experimented with two different code rates, namely 1/2 and 1/4. For the fading channel, channel encoder at first seemed to even deteriorate the result. This proved to be a challenging aspect and took me a lot of time and tweaking to figure out that it might be due to the burst distortions made by the fading channel that is problematic. To fix this issue, I used bit-interleaving where the bitstream is randomly permuted and then at the receiving side is de-interleaved to put the bitstream back to the original ordering.

I also investigated two cases regarding when the channel encoding (and bit interleaving) is carried out. Since OFDM is transferring the bitstream in form of frames there are two possibilities. One could perform channel encoding (and bit-interleaving) inside each frame, or alternatively perform this outside the framings on the whole bit-stream prior to the transmission of the message. I have implemented both and report the result below.

In Fig. 9 the results for the case where channel encoding is performed on the whole message first is depicted. First of all, we can see that, as we expected, bit-interleaving has no impact on the AWGN channel where errors are independent. There is however, a pronounced impact for the

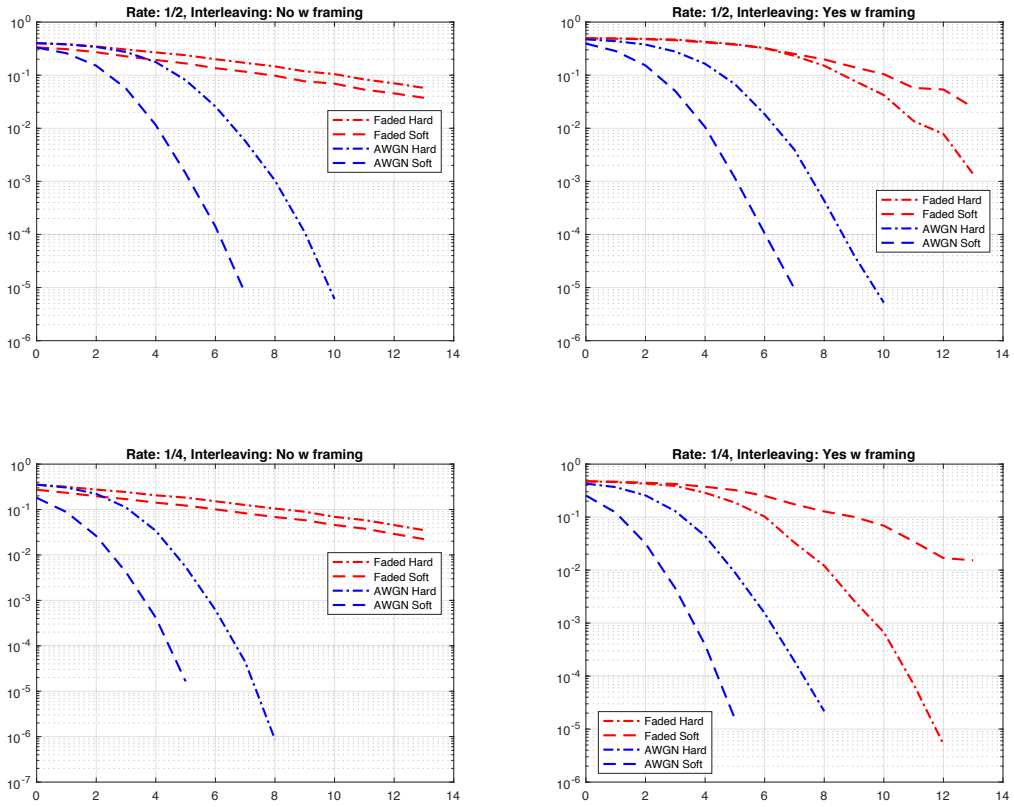

Fig. 9. SNR vs. BER for AWGN and Rayleigh frequency-flat fading channel with soft and hard decoders when channel encoding is performed **outside** the frames

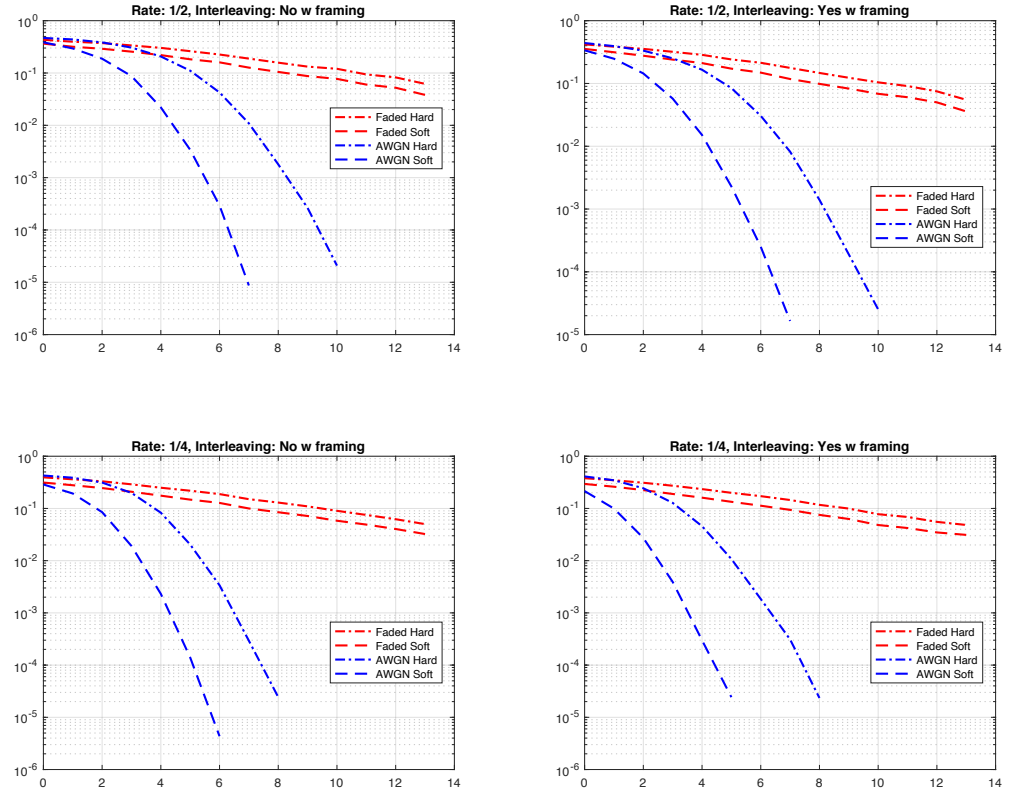

Fig. 10. SNR vs. BER for AWGN and Rayleigh frequency-flat fading channel with soft and hard decoders when channel encoding is performed **inside** the frames

fading channel where there exists correlation in the channel. It shows that bit-interleaving is in fact very crucial for these types of channels. Interestingly the hard decoder has a better performance which I did not expect. I suspect that the soft decoding is not correctly working for the case that channel encoding is performed on the whole message instead of each frame.

In Fig. 10 we see the corresponding result for the case where channel-encoding is performed within each frame. Although the accuracy still improves negligible, specially for the 1/4 coding rate, but we no longer see the same gain here and the curve of the fading channel resists to decline. I believe that the reason is that applying this procedure on only 117 symbols are not sufficient for dispersing the distortion in a frequency-flat fading channel. My conjecture is that if we have frequency-selective fading, the channel encoding scheme would have a better performance. I have not tested this as running the simulations took hours and I did not have time to repeat the experiment. Here, the soft decoder has a better performance, as we expected. A good solution could be applying channel encoding over multiple frames (instead of the whole message or a single frame), where the distortion is dispersed enough for channel encoding to combat bursts of errors in the Rayleigh channel.

I end this section by reporting the result when noise is set to 6.5dB, on AWGN and Rayleigh frequency-flat fading channel with a coding rate of 1/4. In this specific case both decoders for the AWGN were able to reconstruct the original message perfectly. Fig. 11 shows eight of the reconstructed image.

Channel	Decoder	BER	MSE	SNR
AWGN	Hard	0.00000	0.0035	19.3540
AWGN	Soft	0.00000	0.0035	19.3540
Rayleigh	Hard	0.00483	0.0250	10.7792
Rayleigh	Soft	0.10409	0.1577	2.7814

Table 4. BER, MSE and SNR for 6.5dB noise AWGN and Rayleigh frequency-flat fading channels with channel encoding of rate 1/4 and bit interleaving across the whole message

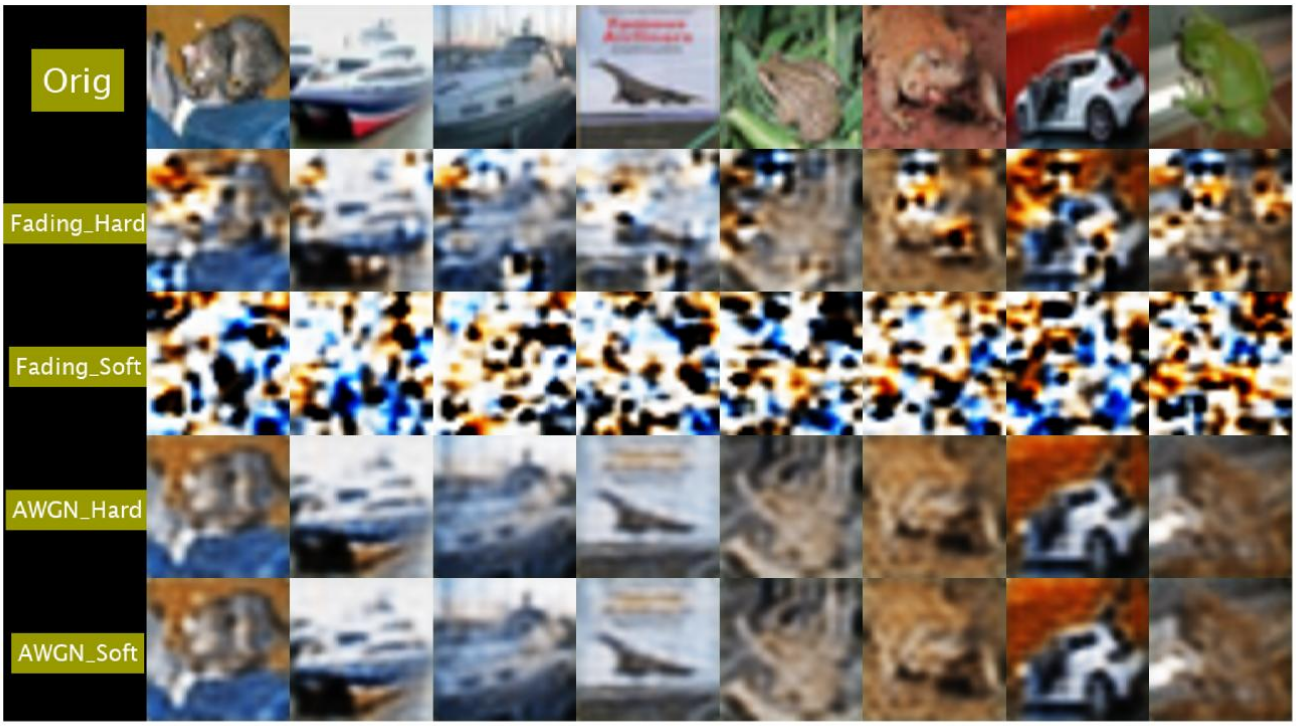

Fig. 11. Received images

VI. Conclusion

In this project we designed a digital communication system for transmission of natural images. We designed an Autoencoder for source coding that performs an 8-fold image compression. We also chose among possible configurations of the Autoencoder based on the performance criteria. The uniform quantization was found to be a proper quantization scheme and the adequate level of quantization was then determined. We proceeded by designing a 16QAM OFDM modulation scheme and showed that our simulated results closely match the theoretical results. To combat noise, we added channel coding to the mix via a Convolutional encoder and a corresponding Viterbi decoder. We also showed the importance of bit-interleaving in forward-error correcting in face of a fading channel.

VII. Appendix 1: Autoencoder (Python Code)

```
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.utils.data.sampler import SubsetRandomSampler

from torch.utils.data import DataLoader
from torchvision import datasets, transforms

from tqdm import tqdm, trange
import pickle
import os.path

## Loading dataset

# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# load the training and test datasets
train_data = datasets.CIFAR10(root='data', train=True,
                                   download=True, transform=transform)
test_data = datasets.CIFAR10(root='data', train=False,
                                  download=True, transform=transform)

# Create training and test dataloaders

num_workers = 0
# how many samples per batch to load
batch_size = 32

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
num_workers=num_workers)

## Convolutional Autoencoder
```

```
class Autoencoder(nn.Module):
    def __init__(self, kernel_size1=12, kernel_size2=24, kernel_size3=32):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, kernel_size1, 4, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(kernel_size1, kernel_size2, 4, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(kernel_size2, kernel_size3, 4, stride=2, padding=1),
            nn.Sigmoid(),
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(kernel_size3, kernel_size2, 4, stride=2, padding=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(kernel_size2, kernel_size1, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(kernel_size1, 3, 4, stride=2, padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

## Training Autoencoders
for kernel_size1 in [3, 6, 12]:
  for kernel_size2 in [3, 6, 12,]:
    if os.path.exists('drive/MyDrive/AutoEncModels/AutoEnc_{}_{}.model'.format(
      kernel_size1, kernel_size2
    )):
      continue
    print('****** kernel_size1:', kernel_size1, '\tkernel_size2:', kernel_size2)
    myautoenc = Autoencoder(kernel_size1=kernel_size1,
                            kernel_size2=kernel_size2,
                            kernel_size3=24)

    learning_rate = .005

    criterion = nn.BCELoss()
    optimizer = optim.Adam(myautoenc.parameters(), lr=learning_rate)

    train_losses = []
```

```
    test_losses = []

    n_epochs = 40
    for epoch in range(n_epochs):
        train_loss = 0.0
        test_loss = 0.0
        for i, (inputs, _) in enumerate(train_loader, 0):
          # inputs = get_torch_vars(inputs)

          # ============ Forward ============
          outputs = myautoenc(inputs)
          loss = criterion(outputs, inputs)
          # ============ Backward ============
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()

          # ============ Logging ============
          train_loss += loss.data

        with torch.no_grad():
          for i, (inputs, _) in enumerate(test_loader, 0):
            outputs = myautoenc(inputs)
            loss = criterion(outputs, inputs)
            test_loss += loss.data

        train_losses.append(train_loss)
        test_losses.append(test_loss)
        if not bool((epoch+1) % 5):
          print("iter: %d\tTrain_Loss: %.4f\tVal_Loss: %.4f" % (epoch, train_loss /
len(train_loader),
                                                                 test_loss /
len(test_loader)))
    torch.save(myautoenc, 'drive/MyDrive/AutoEncModels/AutoEnc_{}_{}.model'.format(
      kernel_size1, kernel_size2
    ))
    with open('drive/MyDrive/AutoEncModels/train_loss_{}_{}.traj'.format(
      kernel_size1, kernel_size2
    ), 'wb') as handle:
      pickle.dump({'train': train_losses,
                   'test': test_losses},
                  handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
## Counting params
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

for kernel_size1 in [3, 6, 12]:
  for kernel_size2 in [3, 6, 12]:
    myautoenc2 = torch.load('drive/MyDrive/AutoEncModels/AutoEnc_{}_{}.model'.format(
          kernel_size1, kernel_size2
        ))
    myautoenc2.eval()

    print([kernel_size1, kernel_size2,
           count_parameters(myautoenc2.encoder),
count_parameters(myautoenc2.decoder)])
```

```
## Python helper function for MATLAB
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.utils.data.sampler import SubsetRandomSampler

from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import sys

from scipy import io

# convert data to torch.FloatTensor
transform = transforms.ToTensor()


# Create training and test dataloaders

num_workers = 0
# how many samples per batch to load
```

```
batch_size = 32


class Autoencoder(nn.Module):
    def __init__(self, kernel_size1=12, kernel_size2=24, kernel_size3=32):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, kernel_size1, 4, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(kernel_size1, kernel_size2, 4, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(kernel_size2, kernel_size3, 4, stride=2, padding=1),
            nn.Sigmoid(),
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(kernel_size3, kernel_size2, 4, stride=2, padding=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(kernel_size2, kernel_size1, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(kernel_size1, 3, 4, stride=2, padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

#print(sys.argv)

ker1 = sys.argv[1]
ker2 = sys.argv[2]
command = sys.argv[3]
dataset = sys.argv[4]

#ker1 = '6'
#ker2 = '6'
#command = 'recons'
#dataset = ''

myautoenc = torch.load('/Users/nima/VTCoursesECE/DigComm/Course_Project/python_src/
AutoEncModels/AutoEnc_{}_{}.model'.format(
      ker1, ker2
```

```
    ))
myautoenc.eval()

if dataset in ['train', 'train_lim']:
    train_data = datasets.CIFAR10(root='/Users/nima/VTCoursesECE/DigComm/
Course_Project/python_src/data', train=True,
                                        download=False, transform=transform)
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
num_workers=num_workers)
    loader = train_loader
else:
    test_data = datasets.CIFAR10(root='/Users/nima/VTCoursesECE/DigComm/
Course_Project/python_src/data', train=False,
                                       download=False, transform=transform)
    test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
num_workers=num_workers)
    loader = test_loader

origs = []
comprs = []
recons = []

if command == 'orig':
    for i, (orig_imgs, _) in enumerate(loader, 0):
        origs.append(orig_imgs)
    origs = np.concatenate(origs)
    if dataset.endswith('_lim'):
        origs = origs[:500,:,:,:]
    io.savemat('/tmp/tmp_py_mat.mat', {'data': origs})
elif command == 'comp':
    for i, (orig_imgs, _) in enumerate(loader, 0):
        comprs.append(myautoenc.encoder(orig_imgs).detach())
    comprs = np.concatenate(comprs)
    if dataset.endswith('_lim'):
        comprs = comprs[:500,:,:,:]
    io.savemat('/tmp/tmp_py_mat.mat', {'data': comprs})
elif command == 'recons':
    data = io.loadmat('/tmp/tmp_mat_py.mat')['dataset']
    recons = myautoenc.decoder(torch.Tensor(data))
    io.savemat('/tmp/tmp_py_mat.mat', {'data': np.array(recons.detach())})
```

VIII. Appendix 2: MATLAB Code

```
%% Calculate MSE and SNR of different Autoencs
orig_imgs = autoenc(12, 12, 'orig', 'test');
orgimg = cell(1, 10);
for i=1:10
    img = permute(reshape(orig_imgs(i,:,:,:), [3 32 32]), [2 3 1]);
    orgimg{i} = img;
end

kers_1 = [3 3 3 6 6 6 12 12 12];
kers_2 = [3 6 12 3 6 12 3 6 12];

recsimg = cell(9, 10);
MSEs = cell(1, 9);
SNRs = cell(1, 9);
for j=1:9
    comp_imgs = autoenc(kers_1(j), kers_2(j), 'comp', 'test');
    recons_imgs = autoenc(kers_1(j), kers_2(j), 'recons', comp_imgs);

    MSEs{j} = immse(orig_imgs, recons_imgs);
    SNRs{j} = snr(orig_imgs, orig_imgs-recons_imgs);
    for i=1:10
        img = permute(reshape(recons_imgs(i,:,:,:), [3 32 32]), [2 3 1]);
        recsimg{j, i} = img;
    end
end

%% Plot compressed and reconstructed images

img = zeros(128, 128, 3);
num_img = 8;
mont = cell(num_img*10);

mont{1} = insertText(img, [68 68], 'Orig', 'FontSize', 30, 'TextColor','white', 'AnchorPoint',
'Center');
for i=1:num_img
    mont{i+1} = orgimg{i};
end
for j=1:9
    mont{j*(num_img+1)+1} = insertText(img, [68 68],...
        ['(' int2str(kers_1(j)) ', ' int2str(kers_2(j)) ')'],...
        'FontSize',30, 'TextColor','white', 'AnchorPoint', 'Center');
```

```
    for i=1:num_img
        mont{j*(num_img+1)+i+1} = recsimg{j, i};
    end
end
montage(mont, 'Size', [10, num_img+1]);

%% Plot histogram of image
histogram(orig_imgs,'Normalization','probability');
xlabel('Pixel values')
ylabel('Probability')


%% Load compressed images from the selected Autoencoder (6, 12)
comp_imgs = autoenc(6, 12, 'comp', 'test');

%%
clc;
[sn, sx, sy, sz] = size(comp_imgs);
comp_flattened = reshape(comp_imgs, [sn*sx*sy*sz 1]);


recsimg = cell(8, 10);
MSEs = cell(1, 8);
SNRs = cell(1, 8);

for bits=1:8
    display(['******** ', int2str(bits)])
    comp_dig = Analog2Digital(double(comp_flattened), 65536, bits, 0);
    comp_anal = Digital2Analog(comp_dig, bits, 0);
    comp_anal_s = reshape(comp_anal, [sn sx sy sz]);
    recons_imgs = autoenc(6, 12, 'recons', comp_anal_s);

    MSEs{bits} = immse(orig_imgs, recons_imgs);
    SNRs{bits} = snr(orig_imgs, orig_imgs-recons_imgs);
    for i=1:10
        img = permute(reshape(recons_imgs(i,:,:,:), [3 32 32]), [2 3 1]);
        recsimg{bits, i} = img;
    end
end
%% Plot images for different quantization levels

img = zeros(128, 128, 3);
num_img = 8;
```

```
mont = cell(num_img*10);

mont{1} = insertText(img, [68 68], 'Orig', 'FontSize', 30, 'TextColor','white', 'AnchorPoint',
'Center');
for i=1:num_img
    mont{i+1} = orgimg{i};
end
for j=1:8
    mont{j*(num_img+1)+1} = insertText(img, [68 68],...
        [ int2str(j) '-bit'],...
        'FontSize',30, 'TextColor','white', 'AnchorPoint', 'Center');
    for i=1:num_img
        mont{j*(num_img+1)+i+1} = recsimg{j, i};
    end
end
montage(mont, 'Size', [9, num_img+1]);
%% Select autoencoder and quantization levels for the rest of the project
clear;
orig_imgs = autoenc(6, 12, 'orig', 'test_lim');
bits = 6;
comp_imgs = autoenc(6, 12, 'comp', 'test_lim');
[sn, sx, sy, sz] = size(comp_imgs);
comp_flattened = reshape(comp_imgs, [sn*sx*sy*sz 1]);
inpsig = Analog2Digital(double(comp_flattened), 65536, bits, 0);

outsig = Digital2Analog(inpsig, bits, 0);
outsig_orig_shape = reshape(outsig, [sn sx sy sz]);
recons_imgs = autoenc(6, 12, 'recons', outsig_orig_shape);
ref_MSEs = immse(orig_imgs, recons_imgs)
ref_SNRs = snr(orig_imgs, orig_imgs-recons_imgs)
save inpsig


%% OFDM-QAM over AWGN and Rayleigh channels
clc;
clear;
load inpsig;

M = 16;                  % Modulation alphabet
k = log2(M);             % Bits/symbol
numSC = 128;             % Number of OFDM subcarriers
cpLen = 32;              % OFDM cyclic prefix length
```

```
ofdmMod = comm.OFDMModulator('FFTLength', numSC, 'CyclicPrefixLength', cpLen);
ofdmDemod = comm.OFDMDemodulator('FFTLength', numSC, 'CyclicPrefixLength', cpLen);

ofdmDims = info(ofdmMod);
numDC = ofdmDims.DataInputSize(1);



rayleighChannel = comm.RayleighChannel;
rayleighChannel.PathGainsOutputPort = true;



awgnChannel = comm.AWGNChannel('NoiseMethod','Variance', 'VarianceSource','Input port');

EbNoVec = (0:10)';
snrVec = EbNoVec + 10*log10(k) + 10*log10(numDC/numSC);

BERqamOFDM = zeros(length(EbNoVec), 1);
BERqamOFDMfaded = zeros(length(EbNoVec), 1);

orig_msg_length = length(inpsig);
framesize = numDC*k;
nFrames = ceil(orig_msg_length / framesize);
paded_inpsig = inpsig;
paded_inpsig(end:nFrames * framesize) = 0;

rxBits = zeros([length(paded_inpsig), 1]);
rxBitsFaded = zeros([length(paded_inpsig), 1]);

fadings = zeros([(numSC + cpLen) * nFrames 1]);
for m = 1:length(EbNoVec)
    for j=1:nFrames
        frame = paded_inpsig((j-1)*framesize+1:j*framesize);
        qamTx = qammod(frame, M, 'InputType', 'bit', 'UnitAveragePower', true);
        qamOfdmTx = ofdmMod(qamTx);

        [qamFadedSig pathGains] = rayleighChannel(qamOfdmTx);

        fadings((numSC + cpLen)*(j-1)+1:(numSC + cpLen)*j) = pathGains;

        powerDB = 10*log10(var(qamOfdmTx));
        noiseVar = 10.^(0.1*(powerDB-snrVec(m)));

        qamRxFadedNoisy = awgnChannel(qamFadedSig, noiseVar);
```

```
        qamRxNoisy = awgnChannel(qamOfdmTx, noiseVar);

        qamRxNoisyEq = qamRxFadedNoisy./sum(pathGains, 2);

        qamOfdmRxFaded = ofdmDemod(qamRxNoisyEq);
        qamOfdmRx = ofdmDemod(qamRxNoisy);

        qamRxFaded = qamdemod(qamOfdmRxFaded, M, 'OutputType', 'bit', 'UnitAveragePower', true);
        qamRx = qamdemod(qamOfdmRx, M, 'OutputType', 'bit', 'UnitAveragePower', true);

        rxBitsFaded((j-1)*framesize+1:j*framesize)  = qamRxFaded;
        rxBits((j-1)*framesize+1:j*framesize)  = qamRx;
    end
    [~, ratio] = biterr(inpsig, rxBitsFaded(1:length(inpsig)))
    BERqamOFDMfaded(m) = ratio;
    [~, ratio] = biterr(inpsig, rxBits(1:length(inpsig)))
    BERqamOFDM(m) = ratio;
end

%% Plot Channel Amplitude
ll = numSC + cpLen;
ff = 140;
semilogy((1:ll*ff) * 117 / ll / 16384, sqrt(fadings(1:ll*ff).^2));
ylabel('Semilog Ch. Amp.');
xlabel('Time (s)');


%% Semilog SNR-BER plots
close all;
semilogy(EbNoVec, BERqamOFDMfaded, 'r*');
hold on;
semilogy(EbNoVec, BERqamOFDM, 'b*');
grid on;
hold on;
berTheoryQamFading = berfading(EbNoVec,'qam', M, 1);
semilogy(EbNoVec, berTheoryQamFading, 'r--');
hold on;
berTheoryQamAwgn = berawgn(EbNoVec,'qam', M);
semilogy(EbNoVec, berTheoryQamAwgn, 'b--');
legend('Fading OFDM-16QAM Simulation', 'AWGN OFDM-16QAM Simulation',...
    'Fading OFDM-16QAM Theory', 'AWGN OFDM-16QAM Theory', 'location', 'best')
xlabel('Eb/N0')
ylabel('BER')
```

```
%% Convolutional Encoder and Viterbi decoder
clc;
% clear;
load inpsig.mat;

M = 16;                  % Modulation alphabet
k = log2(M);             % Bits/symbol
numSC = 128;             % Number of OFDM subcarriers
cpLen = 32;              % OFDM cyclic prefix length


for framing_encoder=[1]
for ch_coding=[1 2]
    for interleaving=[0 1]

        ofdmMod = comm.OFDMModulator('FFTLength', numSC, 'CyclicPrefixLength', cpLen);
        ofdmDemod = comm.OFDMDemodulator('FFTLength', numSC, 'CyclicPrefixLength', cpLen);

        ofdmDims = info(ofdmMod);
        numDC = ofdmDims.DataInputSize(1);


        rayleighChannel = comm.RayleighChannel;
        rayleighChannel.PathGainsOutputPort =true;
        awgnChannel = comm.AWGNChannel('NoiseMethod','Variance', 'VarianceSource','Input port');


        trellis = {poly2trellis(7,[171 133]), poly2trellis(8, [367 323 275 271])};
        tbl = {32, 32};
        rate = {1/2, 1/4};


        if framing_encoder
            inpsig_msg = inpsig;
        else
            inpsig_pad = inpsig;
            inpsig_pad(length(inpsig):length(inpsig) + tbl{ch_coding}) = 0;
            inpsig_msg = convenc(inpsig_pad, trellis{ch_coding});

            if interleaving
                inpsig_msg = randintrlv(inpsig_msg, 123);
            end
```

```
end


orig_msg_length = length(inpsig_msg);
if framing_encoder
    framesize = numDC*k*rate{ch_coding}-tbl{ch_coding};
else
    framesize = numDC*k;
end
nFrames = ceil(orig_msg_length / framesize);
paded_inpsig = inpsig_msg;
paded_inpsig(end: nFrames * framesize) = 0;

rxBitsFadedSoftEnc = zeros([length(paded_inpsig), 1]);
rxBitsFadedHardEnc = zeros([length(paded_inpsig), 1]);
rxBitsSoftEnc = zeros([length(paded_inpsig), 1]);
rxBitsHardEnc = zeros([length(paded_inpsig), 1]);

EbNoVec = (0:13)';
snrdB = EbNoVec + 10*log10(k*rate{ch_coding}) + 10*log10(numDC/numSC);

BERqamOfdmFadedEncodedSoft = zeros(length(EbNoVec), 1);
BERqamOfdmFadedEncodedHard = zeros(length(EbNoVec), 1);
BERqamOfdmEncodedSoft = zeros(length(EbNoVec), 1);
BERqamOfdmEncodedHard = zeros(length(EbNoVec), 1);


for m = 1:length(EbNoVec)
    for j=1:nFrames
        if framing_encoder
            frame = paded_inpsig((j-1)*framesize+1:j*framesize);
            frame_padded = frame;
            frame_padded(framesize+1:framesize+tbl{ch_coding}) = 0;

            dataEnc = convenc(frame_padded, trellis{ch_coding});
            if interleaving
                dataEnc = randintrlv(dataEnc, j);
            end
            qamTx = qammod(dataEnc, M, 'InputType', 'bit', 'UnitAveragePower',true);
        else
            frame = paded_inpsig((j-1)*framesize+1:j*framesize);
            qamTx = qammod(frame, M, 'InputType', 'bit', 'UnitAveragePower',true);
        end
```

```
                qamOfdmTx = ofdmMod(qamTx);

                powerDB = 10*log10(var(qamOfdmTx));                % Calculate Tx signal power
                noiseVar = 10.^(0.1*(powerDB-snrdB(m)));           % Calculate the noise variance

                % Faded Channel
                [fadedSig pathGains] = rayleighChannel(qamOfdmTx);
                qamOfdmRxFaded = awgnChannel(fadedSig, noiseVar);
                qamOfdmRxFaded_eq = qamOfdmRxFaded./sum(pathGains, 2);
                qamOfdmRxFaded = ofdmDemod(qamOfdmRxFaded_eq);
                % AWGN Channel
                qamOfdmRx =  awgnChannel(qamOfdmTx, noiseVar);
                qamOfdmRx = ofdmDemod(qamOfdmRx);

                % QAM Demod - Faded
                rxDataHardFaded = qamdemod(qamOfdmRxFaded, M,
'OutputType','bit','UnitAveragePower', true);
                rxDataSoftFaded = qamdemod(qamOfdmRxFaded, M, 'OutputType','approxllr', ...
                            'UnitAveragePower', true, 'NoiseVariance',noiseVar);

                % QAM Demod - AWGN
                rxDataHard = qamdemod(qamOfdmRx, M, 'OutputType','bit','UnitAveragePower', true);
                rxDataSoft = qamdemod(qamOfdmRx, M, 'OutputType','approxllr', ...
                            'UnitAveragePower', true, 'NoiseVariance',noiseVar);
                if framing_encoder
                    if interleaving
                        rxDataHardFaded = randdeintrlv(rxDataHardFaded, j);
                        rxDataSoftFaded = randdeintrlv(rxDataSoftFaded, j);
                        rxDataHard = randdeintrlv(rxDataHard, j);
                        rxDataSoft = randdeintrlv(rxDataSoft, j);
                    end
                    % Decode - Faded
                    dataHardFaded = vitdec(rxDataHardFaded, trellis{ch_coding}, tbl{ch_coding},
'cont', 'hard');
                    dataSoftFaded = vitdec(rxDataSoftFaded, trellis{ch_coding}, tbl{ch_coding},
'cont', 'unquant');

                    % Decode - AWGN
                    dataHard = vitdec(rxDataHard, trellis{ch_coding}, tbl{ch_coding}, 'cont',
'hard');
```

```
                    dataSoft = vitdec(rxDataSoft, trellis{ch_coding}, tbl{ch_coding}, 'cont',
'unquant');

                    % Faded
                    rxBitsFadedHardEnc((j-1)*framesize+1:j*framesize)  =
dataHardFaded(tbl{ch_coding}+1:end);
                    rxBitsFadedSoftEnc((j-1)*framesize+1:j*framesize)  =
dataSoftFaded(tbl{ch_coding}+1:end);
                    % AWGN
                    rxBitsHardEnc((j-1)*framesize+1:j*framesize)  = dataHard(tbl{ch_coding}
+1:end);
                    rxBitsSoftEnc((j-1)*framesize+1:j*framesize)  = dataSoft(tbl{ch_coding}
+1:end);
                else
                    % Faded
                    rxBitsFadedHardEnc((j-1)*framesize+1:j*framesize)  = rxDataHardFaded;
                    rxBitsFadedSoftEnc((j-1)*framesize+1:j*framesize)  = rxDataSoftFaded;
                    % AWGN
                    rxBitsHardEnc((j-1)*framesize+1:j*framesize)  = rxDataHard;
                    rxBitsSoftEnc((j-1)*framesize+1:j*framesize)  = rxDataSoft;
                end

            end
            display('***************');
            fprintf('framing: %d\t coding_rate: %.2f\t interleaving: %d\t EbN0: %d \n',
framing_encoder, rate{ch_coding}, interleaving, EbNoVec(m))
            if framing_encoder
                [~, ratio] = biterr(inpsig, rxBitsFadedHardEnc(1:length(inpsig)));
                BERqamOfdmFadedEncodedHard(m) = ratio;
                [~, ratio] = biterr(inpsig, rxBitsFadedSoftEnc(1:length(inpsig)));
                BERqamOfdmFadedEncodedSoft(m) = ratio;
                [~, ratio] = biterr(inpsig, rxBitsHardEnc(1:length(inpsig)));
                BERqamOfdmEncodedHard(m) = ratio;
                [~, ratio] = biterr(inpsig, rxBitsSoftEnc(1:length(inpsig)));
                BERqamOfdmEncodedSoft(m) = ratio;
            else
                if interleaving
                    rxBitsFadedHardEnc = randdeintrlv(rxBitsFadedHardEnc(1:orig_msg_length),
123);
                    rxBitsFadedSoftEnc = randdeintrlv(rxBitsFadedSoftEnc(1:orig_msg_length),
123);
                    rxBitsHardEnc = randdeintrlv(rxBitsHardEnc(1:orig_msg_length), 123);
                    rxBitsSoftEnc = randdeintrlv(rxBitsSoftEnc(1:orig_msg_length), 123);
```

```
                end

                inpsig_dec_FadedHardEnc = vitdec(rxBitsFadedHardEnc, trellis{ch_coding},
tbl{ch_coding}, 'cont', 'hard');
                inpsig_dec_FadedSoftEnc = vitdec(rxBitsFadedSoftEnc, trellis{ch_coding},
tbl{ch_coding}, 'cont', 'unquant');
                inpsig_dec_HardEnc = vitdec(rxBitsHardEnc, trellis{ch_coding}, tbl{ch_coding},
'cont', 'hard');
                inpsig_dec_SoftEnc = vitdec(rxBitsSoftEnc, trellis{ch_coding}, tbl{ch_coding},
'cont', 'unquant');

                [~, ratio] = biterr(inpsig, inpsig_dec_FadedHardEnc(tbl{ch_coding}
+1:tbl{ch_coding}+length(inpsig)));
                BERqamOfdmFadedEncodedHard(m) = ratio;
                [~, ratio] = biterr(inpsig, inpsig_dec_FadedSoftEnc(tbl{ch_coding}
+1:tbl{ch_coding}+length(inpsig)));
                BERqamOfdmFadedEncodedSoft(m) = ratio;
                [~, ratio] = biterr(inpsig, inpsig_dec_HardEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)));
                BERqamOfdmEncodedHard(m) = ratio;
                [~, ratio] = biterr(inpsig, inpsig_dec_SoftEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)));
                BERqamOfdmEncodedSoft(m) = ratio;
            end
            fprintf('\tFadedHard: %.5f\tFadedSoft: %.5f\tHard: %.5f\tSoft: %.5f \n',
BERqamOfdmFadedEncodedHard(m),...
                BERqamOfdmFadedEncodedSoft(m), BERqamOfdmEncodedHard(m),
BERqamOfdmEncodedSoft(m))
        end
        BERqamOfdmFadedEncodedHard_r{ch_coding, interleaving+1, framing_encoder+1} =
BERqamOfdmFadedEncodedHard;
        BERqamOfdmFadedEncodedSoft_r{ch_coding, interleaving+1, framing_encoder+1} =
BERqamOfdmFadedEncodedSoft;
        BERqamOfdmEncodedHard_r{ch_coding, interleaving+1, framing_encoder+1} =
BERqamOfdmEncodedHard;
        BERqamOfdmEncodedSoft_r{ch_coding, interleaving+1, framing_encoder+1} =
BERqamOfdmEncodedSoft;
    end
end
end

save BERqamOfdmFadedEncodedHard_r
save BERqamOfdmFadedEncodedSoft_r
```

```
save BERqamOfdmEncodedHard_r
save BERqamOfdmEncodedSoft_r


%% Semilogy BERvsSNR plots for various channel coding configs

close all;

set(gcf,'PaperSize',[16 16]);
set(gcf,'position',[0 0 1000 1000]);

framing = 1;
rates = {'1/2', '1/4'};
inter = {'No', 'Yes'};
for pp = 1:4
    subplot(2, 2, pp);
    for coding_rate = [1 2]
        for interleaving = [0 1]
            if pp == (coding_rate-1) * 2 + interleaving + 1
                % main plot
                lw = 1.7;
            else
                continue
                lw = 1.0;
                % other plots
            end
            semilogy(EbNoVec, BERqamOfdmFadedEncodedHard_r{coding_rate, interleaving+1,
framing+1}, 'r:', 'linewidth', lw);
            hold on;
            semilogy(EbNoVec, BERqamOfdmFadedEncodedSoft_r{coding_rate, interleaving+1,
framing+1}, 'r-', 'linewidth', lw);
            hold on;
            semilogy(EbNoVec, BERqamOfdmEncodedHard_r{coding_rate, interleaving+1, framing+1},
'b--', 'linewidth', lw);
            hold on;
            semilogy(EbNoVec, BERqamOfdmEncodedSoft_r{coding_rate, interleaving+1, framing+1},
'b-.', 'linewidth', lw);
            legend('Faded Hard', 'Faded Soft', 'AWGN Hard', 'AWGN Soft', 'location', 'best')
            title(sprintf('Rate: %s, Interleaving: %s w framing', rates{coding_rate},
inter{interleaving+1}));
            grid on;
        end
    end
```

```
end

%% Convolutional Encoder and Viterbi decoder
clc;
clear;
load inpsig.mat;

M = 16;                  % Modulation alphabet
k = log2(M);            % Bits/symbol
numSC = 128;            % Number of OFDM subcarriers
cpLen = 32;             % OFDM cyclic prefix length


framing_encoder = 0;
ch_coding = 2;
interleaving = 1;

ofdmMod = comm.OFDMModulator('FFTLength', numSC, 'CyclicPrefixLength', cpLen);
ofdmDemod = comm.OFDMDemodulator('FFTLength', numSC, 'CyclicPrefixLength', cpLen);

ofdmDims = info(ofdmMod);
numDC = ofdmDims.DataInputSize(1);


rayleighChannel = comm.RayleighChannel;
rayleighChannel.PathGainsOutputPort =true;
awgnChannel = comm.AWGNChannel('NoiseMethod','Variance', 'VarianceSource','Input port');


trellis = {poly2trellis(7,[171 133]), poly2trellis(8, [367 323 275 271])};
tbl = {32, 32};
rate = {1/2, 1/4};


if framing_encoder
    inpsig_msg = inpsig;
else
    inpsig_pad = inpsig;
    inpsig_pad(length(inpsig):length(inpsig) + tbl{ch_coding}) = 0;
    inpsig_msg = convenc(inpsig_pad, trellis{ch_coding});

    if interleaving
        inpsig_msg = randintrlv(inpsig_msg, 123);
```

```
    end
end



orig_msg_length = length(inpsig_msg);
if framing_encoder
    framesize = numDC*k*rate{ch_coding}-tbl{ch_coding};
else
    framesize = numDC*k;
end
nFrames = ceil(orig_msg_length / framesize);
paded_inpsig = inpsig_msg;
paded_inpsig(end: nFrames * framesize) = 0;

rxBitsFadedSoftEnc = zeros([length(paded_inpsig), 1]);
rxBitsFadedHardEnc = zeros([length(paded_inpsig), 1]);
rxBitsSoftEnc = zeros([length(paded_inpsig), 1]);
rxBitsHardEnc = zeros([length(paded_inpsig), 1]);

EbNoVec = [8.5];
snrdB = EbNoVec + 10*log10(k*rate{ch_coding}) + 10*log10(numDC/numSC);

BERqamOfdmFadedEncodedSoft = zeros(length(EbNoVec), 1);
BERqamOfdmFadedEncodedHard = zeros(length(EbNoVec), 1);
BERqamOfdmEncodedSoft = zeros(length(EbNoVec), 1);
BERqamOfdmEncodedHard = zeros(length(EbNoVec), 1);



for m = 1:length(EbNoVec)
    for j=1:nFrames
        if framing_encoder
            frame = paded_inpsig((j-1)*framesize+1:j*framesize);
            frame_padded = frame;
            frame_padded(framesize+1:framesize+tbl{ch_coding}) = 0;

            dataEnc = convenc(frame_padded, trellis{ch_coding});
            if interleaving
                dataEnc = randintrlv(dataEnc, j);
            end
            qamTx = qammod(dataEnc, M, 'InputType', 'bit', 'UnitAveragePower',true);
        else
            frame = paded_inpsig((j-1)*framesize+1:j*framesize);
            qamTx = qammod(frame, M, 'InputType', 'bit', 'UnitAveragePower',true);
```

```
        end


        qamOfdmTx = ofdmMod(qamTx);

        powerDB = 10*log10(var(qamOfdmTx));              % Calculate Tx signal power
        noiseVar = 10.^(0.1*(powerDB-snrdB(m)));          % Calculate the noise variance

        % Faded Channel
        [fadedSig pathGains] = rayleighChannel(qamOfdmTx);
        qamOfdmRxFaded = awgnChannel(fadedSig, noiseVar);
        qamOfdmRxFaded_eq = qamOfdmRxFaded./sum(pathGains, 2);
        qamOfdmRxFaded = ofdmDemod(qamOfdmRxFaded_eq);
        % AWGN Channel
        qamOfdmRx =  awgnChannel(qamOfdmTx, noiseVar);
        qamOfdmRx = ofdmDemod(qamOfdmRx);

        % QAM Demod - Faded
        rxDataHardFaded = qamdemod(qamOfdmRxFaded, M, 'OutputType','bit','UnitAveragePower',
true);
        rxDataSoftFaded = qamdemod(qamOfdmRxFaded, M, 'OutputType','approxllr', ...
                    'UnitAveragePower', true, 'NoiseVariance',noiseVar);

        % QAM Demod - AWGN
        rxDataHard = qamdemod(qamOfdmRx, M, 'OutputType','bit','UnitAveragePower', true);
        rxDataSoft = qamdemod(qamOfdmRx, M, 'OutputType','approxllr', ...
                    'UnitAveragePower', true, 'NoiseVariance',noiseVar);
        if framing_encoder
            if interleaving
                rxDataHardFaded = randdeintrlv(rxDataHardFaded, j);
                rxDataSoftFaded = randdeintrlv(rxDataSoftFaded, j);
                rxDataHard = randdeintrlv(rxDataHard, j);
                rxDataSoft = randdeintrlv(rxDataSoft, j);
            end
            % Decode - Faded
            dataHardFaded = vitdec(rxDataHardFaded, trellis{ch_coding}, tbl{ch_coding}, 'cont',
'hard');
            dataSoftFaded = vitdec(rxDataSoftFaded, trellis{ch_coding}, tbl{ch_coding}, 'cont',
'unquant');

            % Decode - AWGN
            dataHard = vitdec(rxDataHard, trellis{ch_coding}, tbl{ch_coding}, 'cont', 'hard');
            dataSoft = vitdec(rxDataSoft, trellis{ch_coding}, tbl{ch_coding}, 'cont', 'unquant');
```

```
            % Faded
            rxBitsFadedHardEnc((j-1)*framesize+1:j*framesize)  = dataHardFaded(tbl{ch_coding}
+1:end);
            rxBitsFadedSoftEnc((j-1)*framesize+1:j*framesize)  = dataSoftFaded(tbl{ch_coding}
+1:end);
            % AWGN
            rxBitsHardEnc((j-1)*framesize+1:j*framesize)  = dataHard(tbl{ch_coding}+1:end);
            rxBitsSoftEnc((j-1)*framesize+1:j*framesize)  = dataSoft(tbl{ch_coding}+1:end);
        else
            % Faded
            rxBitsFadedHardEnc((j-1)*framesize+1:j*framesize)  = rxDataHardFaded;
            rxBitsFadedSoftEnc((j-1)*framesize+1:j*framesize)  = rxDataSoftFaded;
            % AWGN
            rxBitsHardEnc((j-1)*framesize+1:j*framesize)  = rxDataHard;
            rxBitsSoftEnc((j-1)*framesize+1:j*framesize)  = rxDataSoft;
        end

    end
    display('***************');
    fprintf('framing: %d\t coding_rate: %.2f\t interleaving: %d\t EbN0: %d \n', framing_encoder,
rate{ch_coding}, interleaving, EbNoVec(m))
    if framing_encoder
        [~, ratio] = biterr(inpsig, rxBitsFadedHardEnc(1:length(inpsig)));
        BERqamOfdmFadedEncodedHard(m) = ratio;
        [~, ratio] = biterr(inpsig, rxBitsFadedSoftEnc(1:length(inpsig)));
        BERqamOfdmFadedEncodedSoft(m) = ratio;
        [~, ratio] = biterr(inpsig, rxBitsHardEnc(1:length(inpsig)));
        BERqamOfdmEncodedHard(m) = ratio;
        [~, ratio] = biterr(inpsig, rxBitsSoftEnc(1:length(inpsig)));
        BERqamOfdmEncodedSoft(m) = ratio;
    else
        if interleaving
            rxBitsFadedHardEnc = randdeintrlv(rxBitsFadedHardEnc(1:orig_msg_length), 123);
            rxBitsFadedSoftEnc = randdeintrlv(rxBitsFadedSoftEnc(1:orig_msg_length), 123);
            rxBitsHardEnc = randdeintrlv(rxBitsHardEnc(1:orig_msg_length), 123);
            rxBitsSoftEnc = randdeintrlv(rxBitsSoftEnc(1:orig_msg_length), 123);
        end

        inpsig_dec_FadedHardEnc = vitdec(rxBitsFadedHardEnc, trellis{ch_coding}, tbl{ch_coding},
'cont', 'hard');
        inpsig_dec_FadedSoftEnc = vitdec(rxBitsFadedSoftEnc, trellis{ch_coding}, tbl{ch_coding},
'cont', 'unquant');
```

```
        inpsig_dec_HardEnc = vitdec(rxBitsHardEnc, trellis{ch_coding}, tbl{ch_coding}, 'cont',
'hard');
        inpsig_dec_SoftEnc = vitdec(rxBitsSoftEnc, trellis{ch_coding}, tbl{ch_coding}, 'cont',
'unquant');

        [~, ratio] = biterr(inpsig, inpsig_dec_FadedHardEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)));
        BERqamOfdmFadedEncodedHard(m) = ratio;
        [~, ratio] = biterr(inpsig, inpsig_dec_FadedSoftEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)));
        BERqamOfdmFadedEncodedSoft(m) = ratio;
        [~, ratio] = biterr(inpsig, inpsig_dec_HardEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)));
        BERqamOfdmEncodedHard(m) = ratio;
        [~, ratio] = biterr(inpsig, inpsig_dec_SoftEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)));
        BERqamOfdmEncodedSoft(m) = ratio;
    end
    fprintf('\tFadedHard: %.5f\tFadedSoft: %.5f\tHard: %.5f\tSoft: %.5f \n',
BERqamOfdmFadedEncodedHard(m),...
        BERqamOfdmFadedEncodedSoft(m), BERqamOfdmEncodedHard(m), BERqamOfdmEncodedSoft(m))
end
BERqamOfdmFadedEncodedHard_r{ch_coding, interleaving+1, framing_encoder+1} =
BERqamOfdmFadedEncodedHard;
BERqamOfdmFadedEncodedSoft_r{ch_coding, interleaving+1, framing_encoder+1} =
BERqamOfdmFadedEncodedSoft;
BERqamOfdmEncodedHard_r{ch_coding, interleaving+1, framing_encoder+1} = BERqamOfdmEncodedHard;
BERqamOfdmEncodedSoft_r{ch_coding, interleaving+1, framing_encoder+1} = BERqamOfdmEncodedSoft;

%% Use the Autoencoder on received signals
orig_imgs = autoenc(6, 12, 'orig', 'test_lim');
bits = 6;
comp_imgs = autoenc(6, 12, 'comp', 'test_lim');
[sn, sx, sy, sz] = size(comp_imgs);

outsigs = {Digital2Analog(inpsig_dec_FadedHardEnc(tbl{ch_coding}+1:tbl{ch_coding}
+length(inpsig)), bits, 0),...
    Digital2Analog(inpsig_dec_FadedSoftEnc(tbl{ch_coding}+1:tbl{ch_coding}+length(inpsig)), bits,
0),...
    Digital2Analog(inpsig_dec_HardEnc(tbl{ch_coding}+1:tbl{ch_coding}+length(inpsig)), bits,
0),...
    Digital2Analog(inpsig_dec_SoftEnc(tbl{ch_coding}+1:tbl{ch_coding}+length(inpsig)), bits, 0)};
```

```
recons_imgs = cell([4 1]);
for i=1:4
    outsigs{i} = reshape(outsigs{i}, [sn sx sy sz]);
    recons_imgs{i} = autoenc(6, 12, 'recons', outsigs{i});
end

%% Calculate MSE and SNR of received signals
MSEs = cell([4 1]);
SNRs = cell([4 1]);
for i=1:4
    MSEs{i} = immse(orig_imgs, recons_imgs{i});
    SNRs{i} = snr(orig_imgs, orig_imgs-recons_imgs{i});
end
MSEs
SNRs

%% Montage reconstructed images
orgimg = cell(1, 10);
for i=1:10
    img = permute(reshape(orig_imgs(i,:,:,:), [3 32 32]), [2 3 1]);
    orgimg{i} = img;
end

for j=1:4
    for i=1:10
        img = permute(reshape(recons_imgs{j}(i,:,:,:), [3 32 32]), [2 3 1]);
        recsimg{j, i} = img;
    end
end

img = zeros(128, 128, 3);
num_img = 8;
mont = cell(num_img*5, 1);

mont{1} = insertText(img, [68 68], 'Orig', 'FontSize', 30, 'TextColor','white', 'AnchorPoint',
'Center');
for i=1:num_img
    mont{i+1} = orgimg{i};
end
labels = {'Fading_Hard', 'Fading_Soft', 'AWGN_Hard', 'AWGN_Soft'};
for j=1:4
    mont{j*(num_img+1)+1} = insertText(img, [68 68],...
        labels{j},...
```

```
        'FontSize',20, 'TextColor','white', 'AnchorPoint', 'Center');
    for i=1:num_img
        mont{j*(num_img+1)+i+1} = recsimg{j, i};
    end
end
montage(mont, 'Size', [5, num_img+1]);
```