University of Glasgow | James Watt School of Engineering

1 Digital Communication 4: OFDM Communications Link

1.1 Introduction

This coding project will introduce you to Orthogonal Frequency Division Multiplexing, simulating a communications link with multipath interference and incorporating Reed-Solomon Channel Coding.

If you are using your own computer, make sure the Python libraries `scipy`, `numpy`, and `matplotlib` are installed. It is recommended that you use a suitable IDE for your project, such as Spyder, PyCharm or Visual Studio. Tip: if you are using Spyder, you can display, manipulate and save graphics in separate windows by setting

`Tools>Preferences>IPython console>Graphics>Backend`

from `Inline` to `Automatic`.

You will need to download and install, if required, the following programmes or libraries:

- `komm` Roberto Nobrega's library, which we used in the previous coding projects
  `pip install komm`
- `pyofdm` by Bernd Porr and David Hutchings `https://pypi.org/project/pyofdm/`
  `pip install pyofdm`
- `reedsolo` a python library to implement Reed-Solomon channel code and decode
  `https://pypi.org/project/reedsolo/` which can be installed using the command
  `pip install reedsolo`
- `Audacity` open source audio editor available for Windows, Mac or Linux at
  `https://audacityteam.org/download`

Each project will be scheduled over a two week period, within which there will be 2 scheduled consultation sessions where you will be able to ask teaching staff for guidance. The project should be written up as a short report describing what you've done and the results you have taken along with any conclusions that you draw. Include your python code(s) in the appendices. Make sure your name and student ID number is on the report. The report should be uploaded to the Moodle assignment by the stated deadline, either using Moodle's inbuilt html editor, or as a single PDF file.

The bespoke OFDM and standard modules to be imported are

```
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
```

1.2 OFDM Transmitter

The arguments and default values (corresponding to 802.11g WiFi) for the OFDM encoder and decoder module are,

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

Energy dispersal is done with a pre-seeded random number generator. Both pilot tones and the cyclic prefix are added so that the start of the symbol can be detected at the receiver. The complex time series after the inverse Fourier Transform is modulated into a real valued stream with a Nyquist quadrature modulator. On the receiver side the start of the symbol is detected by first doing a coarse search with the cyclic prefix and then a precision alignment with the pilots.

`nFreqSamples` sets the number of frequency coefficients of the IFFT (should be a power of 2). Pilot tones are injected at the values from the list `pilotIndices`. The real valued pilot amplitude is `pilotAmplitude` For transmission `nData` bytes are expected in an array. The relative length of the Cyclic prefix is `fracCyclic`; the default cyclic prefix is a $\frac{1}{4}$ of the length of the symbol. `mQAM` is the QAM order and the default `mQAM=2` is identical to QPSK.

Using the following code to initialise the parameters of the OFDM module, commensurate with the DVB-T 2k standard. The supplied routine `pyofdm.codec.setpilotindex()` provides the list of equally spaced pilot tones.

```
# 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)
```

First, encode a single (complex) symbol using random data bytes,

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

Inspect the complex valued time series `complex_signal` in the variable explorer, and by plotting the real and imag components as shown below. Can you account for its length? Can you identify any feature?

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

Now inspect the abs value of its discrete Fourier transform of the symbol without the prefix by plotting the following.

```
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()
```

Note that in the pyofdm code, we have yet to incorporate an appropriate shift to the sub-carrier assignments so that the OBW (Occupied BandWidth) is optimally located in the centre of the complex spectrum. However, the code as-is provides the functionality sufficient for the investigations in the current assignment.

A number of 8 bit depth grayscale (pgm) images of various sizes have been provided for you to use in this coding project. You are also free to try with your own images but make sure you know how they can be represented as arrays of 8 bit integers (bytes). Replace the random byte stream with one of the provided (or your own) images which should be read into a `numpy` array of type `uint8` using the `PIL.Image` module. In this assignment (unlike previous), ensure you leave the data in **bytes**, which is the appropriate datatype for `pyofdm` (and `reedsolo` later).

It is advised to recast your data array into a one-dimensional array of a length (taken to be `tx_byte` in the code snippet below) which is a whole multiple of `nbytes` prior to OFDM encoding, appending an appropriate number of dummy bytes. The OFDM encoding can be performed and appended to `complex_signal`

```
complex_signal = np.array([ofdm.encode(tx_byte[i:i+nbytes])
    for i in range(0,tx_byte.size,nbytes)]).ravel()
```

The stream can be modulated at the Nyquist frequency to give a sampled output using the supplied routine. Add some random length zero data to the start of the double-sampled, real-valued baseband signal in order to demonstrate the symbol start search later. Save the result as a *wav* file, noting that 44.1 kHz is the standard consumer digital audio sampling rate.

```
base_signal = pyofdm.nyquistmodem.mod(complex_signal)
# add some random length dummy zero data to the start of the signal here

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

1.3 OFDM Receiver

Create a separate file for the OFDM receiver code. The transmitted data in the form of a *wav* file is read in using. You append additional zeros to the end of the data so the start search algorithm does not reach the end-of-data.

```
samp_rate, base_signal = wav.read('ofdm44100.wav')
# append some extra zeros to the base_signal here

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

As described in lectures, it is necessary to accurately finding the start of the OFDM symbol. Here we use a two-step process. First, make use of the Cyclic Prefix by examining the cross-correlation at `totalFreqSamples`, and then second using the Pilot Tones by examining the sum of the squares of the imaginary component of the expected pilot tones. The `ofdm.findSymbolStartIndex` module performs both of these methods sequentially.

```
searchRangeForPilotPeak = 8
cc, sumofimag, offset = ofdm.findSymbolStartIndex(complex_signal,
    searchrangefine = searchRangeForPilotPeak)
print('Symbol start sample index =',offset)
```

Inspect the value of `offset`. Does it correspond to the previously inserted dummy data in your transmitter code? Plot the quantity `cc` to show the cross-correlation and identify the peak corresponding to this offset. Plot the quantity `sumofimag`, given for the range `[-search_range:search_range]` and confirm that it is substantially lower than the others for one unique delay value.

Initialise the OFDM decoder and decode one symbol at a time as follows. You will need to determine the expected total number of OFDM symbols `Nsig_sym`

```
ofdm.initDecode(complex_signal,offset)

rx_byte = np.uint8([ofdm.decode()[0] for i in range(Nsig_sym)]).ravel()
```

Display the received data as an image and confirm that it matches your original image as used in the transmitter code. Determine the bit error ratio (you will need to read in the original image data), which should be zero for this back-to-back simulation if the offset is correctly determined, as we haven't yet included distortion or noise.

1.4 Distortion and Noise in the Communications Channel

The `Audacity` open source audio editor will be used for adding noise and distortion. Open the *wav* file created by your OFDM transmitter code. Play a few seconds of it and comment on how it sounds.

Multipath Interference can be simulated using the *reverb* effect. Select `Effect>Reverb` and apply the default. Export the result as a 32-bit floating point *wav* file and apply your OFDM receiver code to it (I suggest using a different file name so that you avoid overwriting your original signal). Include an example of the resulting image in your report and note the bit error ratio value. You can vary the *reverb* options to explore the degradation of the signal and note the `ber` (remember to load the original signal into Audacity each time). Try at least 2 more of the *reverb* factory settings available under the *Manage* selection.

You can also explore the effects of additive white noise. If necessary, under the `Analyze` menu enable the `Measure RMS` plugin. Create a white noise track using `Generate>Noise`. Use `Analyze>Measure RMS` to obtain values for both the signal and noise track and hence note the value (in dB) for the signal-to-noise ratio. Mix the tracks using `Tracks>Mix` and export the result. Again, apply your OFDM receiver code and note the `ber`. Repeat for an additional couple of different signal-to-noise ratios by adjusting the noise amplitude.

1.5 Reed-Solomon Channel Coding

In this final section forward error encoding will be incorporated using the Reed-Solomon method. We will use the popular $(N = 255, K = 223)$ code which can correct up to 16 symbol (byte) errors in the code word. The RSC import and initialisation is done as follows,

```
from reedsolo import RSCodec
from reedsolo import ReedSolomonError #only required in receiver

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

Prior to RSC encoding, we need to ensure that the number of bytes is a multiple of K, and therefore additional zero bytes should be added if required.

```
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])))
```

Now treat `tx_enc` as the data input stream to the OFDM encoder.

For the OFDM receiver code, first the OFDM start is identified and the OFDM decoding performed. Make sure you decode the appropriate number of OFDM symbols. The Reed-Solomon decoding is then done using the following. If the decoding is unsuccessful due to the number of byte errors, an exception will be raised. In that case, since the code is implemented in a systematic fashion, we can ignore the parity bytes and use the received data bytes which will undoubtedly contain errors.

```
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])
```

Display your recovered data as an image and compare to your original image as used in the transmitter code. Determine the bit error ratio, identifying the change in bit error ratios due to addition of the Reed-Solomon channel coding. Additionally, in the cases of white noise, compare your ber dependence on snr to the RSC(255,223) QPSK plot in the lecture notes.

1.6 Documentation

python 3 `https://docs.python.org/3/`

numpy and scipy `https://docs.scipy.org/doc/`

matplotlib `https://matplotlib.org/contents.html`

spyder `https://docs.spyder-ide.org/`

Getting the python libraries

If you are using your own computer, make sure the Python libraries `scipy`, `numpy` and `matplotlib` are installed. These libraries are installed by default with the Anaconda python distribution. It is recommended that you use a suitable IDE for your project, such as Spyder.