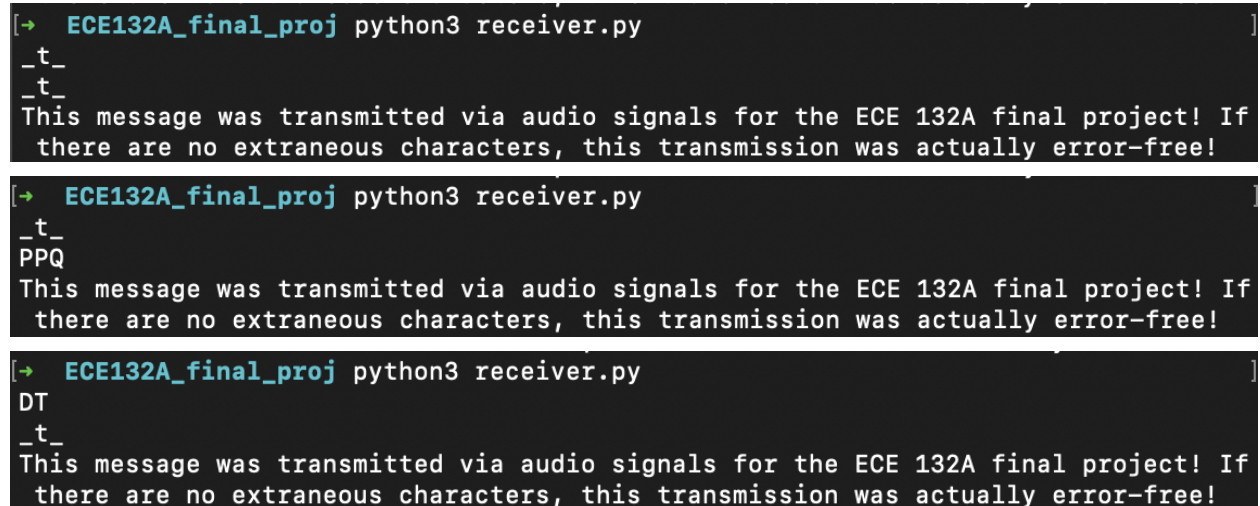Quinn Maloney
UID: 705 016 498
8 March 2021

ECE 132A: Final Project 2
Text Transmission Over the Air (Python implementation)

The provided github code delivers bits via audible tones using quaternary FSK (four carrier frequencies) duplicated over two channels. The receiver and emitter codes use different Python libraries to manipulate audio for some reason, but after installing everything the setup was extremely simple. Their code transmits all 163 ASCII characters (8 bits each) and the 13-bit barker in 69 seconds for a bit rate of $R_b = 19$Hz. For the terminal outputs for a clear transmission, a transmission with AWGN in the 1-2 kHz range (corrupting one channel), and a transmission with AWGN in the 2-3 kHz range (corrupting the other channel), see **Figure 1**. Testing the code against noise was more difficult than I expected because, at the point where either channel can be corrupted by the noise, the transmission is barely audible to human ears.


```
[→  ECE132A_final_proj python3 receiver.py                                   ]
_t_
_t_
This message was transmitted via audio signals for the ECE 132A final project! If
  there are no extraneous characters, this transmission was actually error-free!
[→  ECE132A_final_proj python3 receiver.py                                   ]
_t_
PPQ
This message was transmitted via audio signals for the ECE 132A final project! If
  there are no extraneous characters, this transmission was actually error-free!
[→  ECE132A_final_proj python3 receiver.py                                   ]
DT
_t_
This message was transmitted via audio signals for the ECE 132A final project! If
  there are no extraneous characters, this transmission was actually error-free!
```
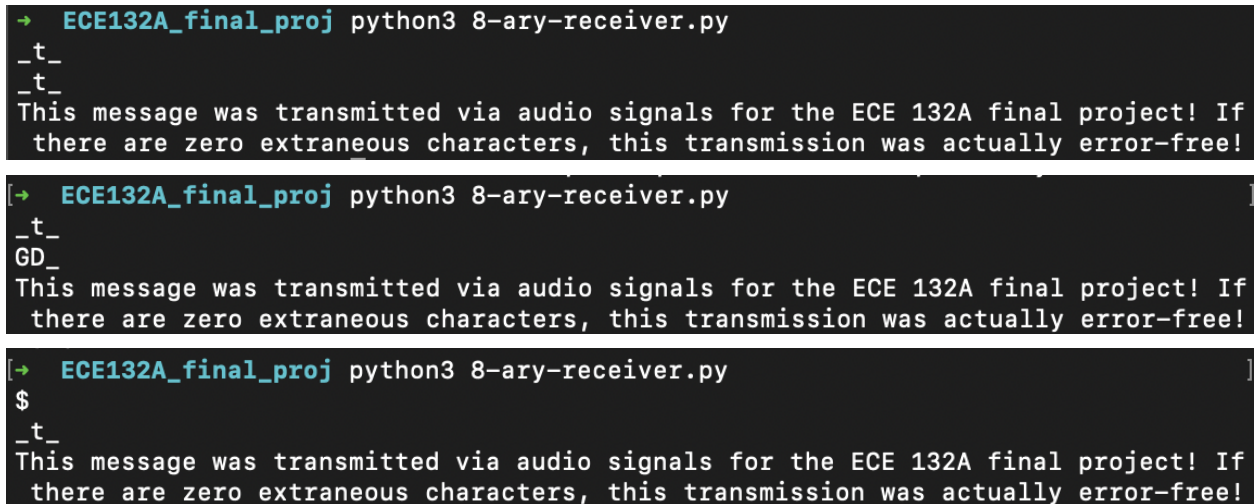
Figure 1- Quaternary FSK Signaling: The responses to clear transmission, 2-3 kHz interference, and 1-2 kHz interference are shown from top to bottom respectively. The received test characters (emitted as '_t_') are decoded from each channel and printed first to indicate corruption.

After familiarizing myself with the code, I noticed that the designers had spectral resolution to spare. Hoping that it may spare future students (and the roommates of future students) a few unnecessary seconds of computer beeping during Week 10, I decided to expand the code into an octonary FSK system.

To facilitate increased complexity, I replaced the nested if/else structure with a dictionary implementation. This directly associated frequency bins from both channels with their respective bit representations. Then I adjusted the signal length and bit indexing to accommodate 3-bits per carrier frequency, including extending the text file to 162 characters (to allow dividing bits into groups of three). I also had to spend some time debugging the receiver-side audio sample

indexing. After ironing out the bugs, the updated code delivers 165 characters and a 15-bit barker in 47 seconds, yielding the expected bit rate of $R_b' = 1.5R_b = 28$Hz. Maybe their professor will give me extra points for the quick and accurate transmission. The responses of my version to clear and noisy transmissions are shown in **Figure 2**.


```
→  ECE132A_final_proj python3 8-ary-receiver.py
_t_
_t_
This message was transmitted via audio signals for the ECE 132A final project! If
 there are zero extraneous characters, this transmission was actually error-free!
[→  ECE132A_final_proj python3 8-ary-receiver.py                                  ]
_t_
GD_
This message was transmitted via audio signals for the ECE 132A final project! If
 there are zero extraneous characters, this transmission was actually error-free!
[→  ECE132A_final_proj python3 8-ary-receiver.py                                  ]
$
_t_
This message was transmitted via audio signals for the ECE 132A final project! If
 there are zero extraneous characters, this transmission was actually error-free!
```
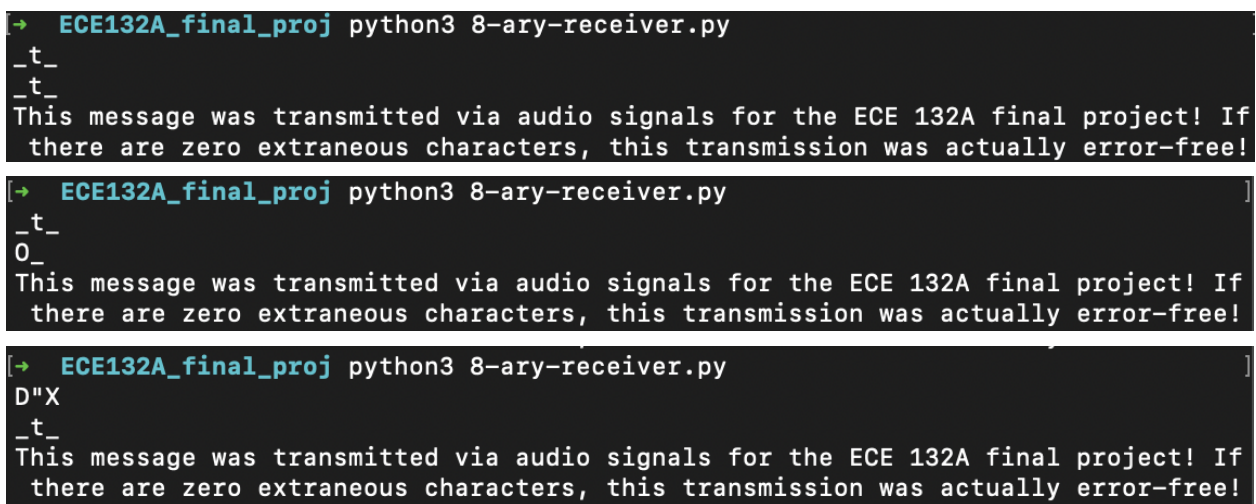
Figure 2- Octonary FSK Signaling: The same corruption from the filters is visible here, but the other channel maintained accuracy.

I also tried increasing the bit rate by decreasing the period between carrier switches (I increased Fbit from 10 to 15). Although rapid jumps in frequency do introduce some distortion (in the form of slight clicking noises) from my laptop speakers at some point, I was able to produce accurate decodings from a 31-second transmission. This puts my system at $R_b'' = 43$Hz, and I'm sure we could push it further.


```
→  ECE132A_final_proj python3 8-ary-receiver.py
_t_
_t_
This message was transmitted via audio signals for the ECE 132A final project! If
 there are zero extraneous characters, this transmission was actually error-free!
[→  ECE132A_final_proj python3 8-ary-receiver.py                                  ]
_t_
O_
This message was transmitted via audio signals for the ECE 132A final project! If
 there are zero extraneous characters, this transmission was actually error-free!
[→  ECE132A_final_proj python3 8-ary-receiver.py                                  ]
D"X
_t_
This message was transmitted via audio signals for the ECE 132A final project! If
 there are zero extraneous characters, this transmission was actually error-free!
```

Figure 3- Transmission Accuracy At Doubled Bitrate: This system runs at over twice the bitrate of the original system. However, by decreasing the frame size I have also decreased the

spectral resolution, which could cause problems distinguishing one carrier from the next if taken too far. At this point there's some risk of losing points for accuracy.

All code can be found at https://github.com/sudo-quinnmaloney/transmit_text_via_audio; see the original project at https://github.com/Mortiniera/text_transfert_over_audio_channel.

OCTONARY_TRANSMITTER.py

```python
import numpy as np
import scipy.io.wavfile as sc
import soundfile as sf
import sounddevice as sd
import sys

frequency_dict = [{'000':1200, '001':1300, '010':1400, '011':1500, '100':1600, '101':1700, '110':1800, '111':1900}, {'000':2200,
'001':2300, '010':2400, '011':2500, '100':2600, '101':2700, '110':2800, '111':2900}]
Fbit = 10 #bit frequency
Fs = 44100 #sampling frequency
A = 1 #amplitude of the signal

BARKER = [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1]
BARKER = BARKER + BARKER
TEST = "_t_"

def encode(text):
    int_text = int.from_bytes(text.encode(), 'big')
    bin_text = str(bin(int_text))
    bin_text = [0] + [int(d) for d in bin_text[2:]] #take out the b from the binary string"
    return bin_text

def modulation(choice,bin_text):
    t = np.arange(0,1/float(Fbit),1/float(Fs), dtype=np.float)
    signal = np.ndarray(len(bin_text)//3*len(t),dtype=np.float)

    i=0
    for j in range(0,len(bin_text),3):
        bits = ''.join([str(n) for n in bin_text[j:j+3]])
        signal[i:i+len(t)] = A * np.cos(2*np.pi*(frequency_dict[choice][bits])*t)
        i += len(t)

    return signal

def double_cosinus(bin_text):
    cos0 = modulation(0,bin_text)
    cos1 = modulation(1,bin_text)
    return cos0 + cos1

def get_sync_signal():
    return double_cosinus(BARKER)

def get_test_signal():
    return double_cosinus(encode(TEST))

def emitter_real(text):
    modul = double_cosinus(encode(text))
    sync = get_sync_signal()
    test = get_test_signal()
    return np.append(np.append(np.append(sync,np.append(test,modul)),test), test)

def emitter(filepath):
    with open(filepath, "r") as data:
        x = emitter_real(data.read())
```

```
        sc.write("emitter2b_expanded.wav",Fs,x)
        #samples, samplerate = sf.read('emitter2b_expanded.wav')
        #sd.play(samples, samplerate)
        sd.wait()


def main():
    path = 'test162.txt'
    emitter(path)

if __name__ == "__main__":
    main()
```

OCTONARY_RECIEVER.py

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

frequency_dict = [{'000':1200, '001':1300, '010':1400, '011':1500, '100':1600, '101':1700, '110':1800, '111':1900}, {'000':2200,
'001':2300, '010':2400, '011':2500, '100':2600, '101':2700, '110':2800, '111':2900}]
frequency_dict_inv = [{v: k for k, v in channel.items()} for channel in frequency_dict]


Fbit = 10 #bit frequency
Fs = 44100 #sampling frequency
A = 1 #amplitude of the signal
FREQ_THRESHOLD = 30 #must be smaller than Fdev

CHUNK = 1024
RECORD_SECONDS = 50
CHANNELS = 1
FORMAT = pyaudio.paInt16

BARKER = [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1]
BARKER = BARKER + BARKER
TEST = "_t_"
#TEST_LEN = len(TEST)*(Fs//Fbit)*3
TEST_LEN = int(len(TEST)*Fs/Fbit*8/3)

def fft(sig):
    return np.abs(np.fft.rfft(sig))

def index_freq(f) :
    return round(f/Fbit)

def fft_plot(ns):
    yf = np.abs(np.fft.rfft(ns))
    plt.plot(yf[index_freq(1000):index_freq(2000)])
    plt.grid()
    plt.show()

def content_at_freq(fft_sample, f, Fbit) :
    lower = f-FREQ_THRESHOLD
    upper = f+FREQ_THRESHOLD
    indexFreqs = [index_freq(f) for f in range(lower,upper, 1)]
    freqSample = max([fft_sample[index] for index in indexFreqs])
    return freqSample

def filtered(sig, choice, full):
```

```python
    data = []
    if(full):
        x = len(sig)//(Fs//Fbit)*(Fs//Fbit)
    else :
        x = TEST_LEN
    for i in range(0,x,Fs//Fbit):
        bit = sig[i:i+Fs//Fbit]
        bitfft = fft(bit)

        xData = [content_at_freq(bitfft, level, Fbit) for level in frequency_dict_inv[choice].keys()]

        toAppend = list(frequency_dict_inv[choice][list(frequency_dict_inv[choice].keys())[np.argmax(xData)]])
        for n in toAppend:
            data.append(int(n))

    return data

def decode_letter(letter):
    binary = int("0b"+letter,base=2)
    m = ""
    try :
        m = binary.to_bytes((binary.bit_length() + 7) // 8, 'big').decode() #decoding from built-in functions in python
    except (UnicodeDecodeError) :
        m = False
    return m

def decode(bin_text):
    str_text = "".join([ str(c) for c in bin_text])
    string = ""
    for i in range(0,len(str_text),8):
        letter = decode_letter(str_text[i:i+8])
        if(letter != False):
            string += letter
        if len(string)>len(TEST) and string[len(string)-len(TEST):]==TEST:
            #print('This happened.')
            return string[:len(string)-len(TEST)]
    return string

def fromstring_buffer(buffer) :
    return struct.unpack('%dh' % (len(buffer)/2), buffer)

def process_raw(in_data):
    data = np.array(fromstring_buffer(in_data))
    return data

def listen_to_signal():
    audio = pyaudio.PyAudio()
    stream = audio.open(format=FORMAT, channels=CHANNELS,rate=Fs, input=True,frames_per_buffer=CHUNK)
    frames = []
    stream.start_stream()

    for i in range(0, int(Fs / CHUNK * RECORD_SECONDS)):
        data = stream.read(CHUNK)
        data = process_raw(data)
        frames.append(data)

    stream.stop_stream()
    stream.close()
    audio.terminate()
    return np.hstack(frames)
```

```python
def modulation(choice,bin_text):
    t = np.arange(0,1/float(Fbit),1/float(Fs), dtype=np.float)
    signal = np.ndarray(len(bin_text)//3*len(t),dtype=np.float)

    i=0
    for j in range(0,len(bin_text),3):
        bits = ''.join([str(n) for n in bin_text[j:j+3]])
        signal[i:i+len(t)] = A * np.cos(2*np.pi*(frequency_dict[choice][bits])*t)
        i += len(t)
    return signal

def double_cosinus(bin_text):
    cos0 = modulation(0,bin_text)
    cos1 = modulation(1,bin_text)
    return cos0 + cos1

def get_sync_signal():
    return double_cosinus(BARKER)

def try_sync(sig):
    sync_signal = get_sync_signal()
    sync_padded = np.hstack((sync_signal, np.zeros(sig.size-sync_signal.size)))
    correlation = np.abs(np.fft.ifft(np.fft.fft(sig) * np.conj(np.fft.fft(sync_padded))))
    return np.argmax(correlation)

def sync(sig):
    SYNC = False
    i=0
    x = len(get_sync_signal())
    index = 0
    while not SYNC:
        chunk = sig[i:i+(5*Fs)]
        index = try_sync(chunk)
        if index + x < len(chunk):
            SYNC = True
        i += x
    return sig[index+x:]

import time
def receiver():
    ls = listen_to_signal()
    x = sync(ls)

    t0 = decode(filtered(x,0,False))
    t1 = decode(filtered(x,1,False))
    x = x[TEST_LEN:]
    print(t0)
    print(t1)
    if(t0 == TEST):
        return decode(filtered(x,0,True))
    if(t1 == TEST):
        return decode(filtered(x,1,True))
    else:
        print("Fatal Error !")

def main():
    x = receiver()
    print(x)

if __name__ == "__main__":
    main()
```