Lab3 - Multimedia Lab

The Multimedia Lab will be a two part lab designed to discuss each aspect of the life-cycle of a multimedia file. This story begins with generation of the file, which is the digitization of some analog phenomenon of interest (e.g. sound -> music, light -> image, both -> film). Once this process is complete, the generated digital media file can be operated on (stored, read, copied, etc.) by a finite precision computer. Each of these operations, however, would be quite a bit faster if the file were small, so we'll investigate compressing the file into a more compact representation. As the honorable Snoop Dogg knew well, it "ain't no fun if the homies can't have none," so you'll naturally want to share this media file with your friends. We'll look at the complications that may arise in transmission, and investigate techniques to ensure that your friend recieves the file error-free. Sadly, all good things must come to an end, and if your file is stored on some physical medium, it will eventually be corrupted.

Naturally, the process of capturing, storing, transmitting, and eventually deleting a media file is composed of a harmony of instruments across **all** EECS disciplines, but this lab will focus in particular on the role of probabilistic analysis, reasoning, and algorithm design in this symphony. Probability in EECS does not exist in a vacuum but, as you will see in this lab, often forms the theoretical underpinning to analyze the efficacy of an idea.

This notebook covers Part 1 of the lab, in which we'll cover the first half of this process: digitization and compression of media. Next week, Part 2 of the lab will be assigned, which covers transmission and corruption. Hope you enjoy the lab!

Part 1: Quantization and Huffman Compression

When capturing media, we must go from a real world analog media to a digital representation. In general, this process is a matter of sampling, quantization, and compression. As technology improves, sampling methods become faster and more precise, quantized representations of media allow for more bits of precision, and both lossy and lossless compression algorithms improve performance. The tools to measure the effectiveness of these improving technologies, however, remain largely the same, and many of them are built on probabilistic analysis.

In this lab, we will talk about quantization for audio signals, and compression of tweets. This should give you an introduction to two of the key aspects of digitally storing analog media.

Audio Quantization

We live in an analog world, where input values may be continous and uncountable. This isn't very useful if we want to do any sort of digital signal processing. What we do instead to deal with this is to create a mapping from our huge set of continuous input values, to a smaller set that we can manage. We essentially bucket-off all of our input values before working with them (rounding is a form of quantization as well). However, as you might imagine, by mapping a large set to a smaller one, we introduce some level of quantization error (think integer division in python). The amount of distortion you experience will be determined by the method and parameters used during the quantization process.

In this portion of the lab, we will step you through the process of generating and playing a wav file in iPython. You will then implement a quantizer method, and play the resulting quantized song. We will wrap up this section of the lab by exploring the effects of quantization error.

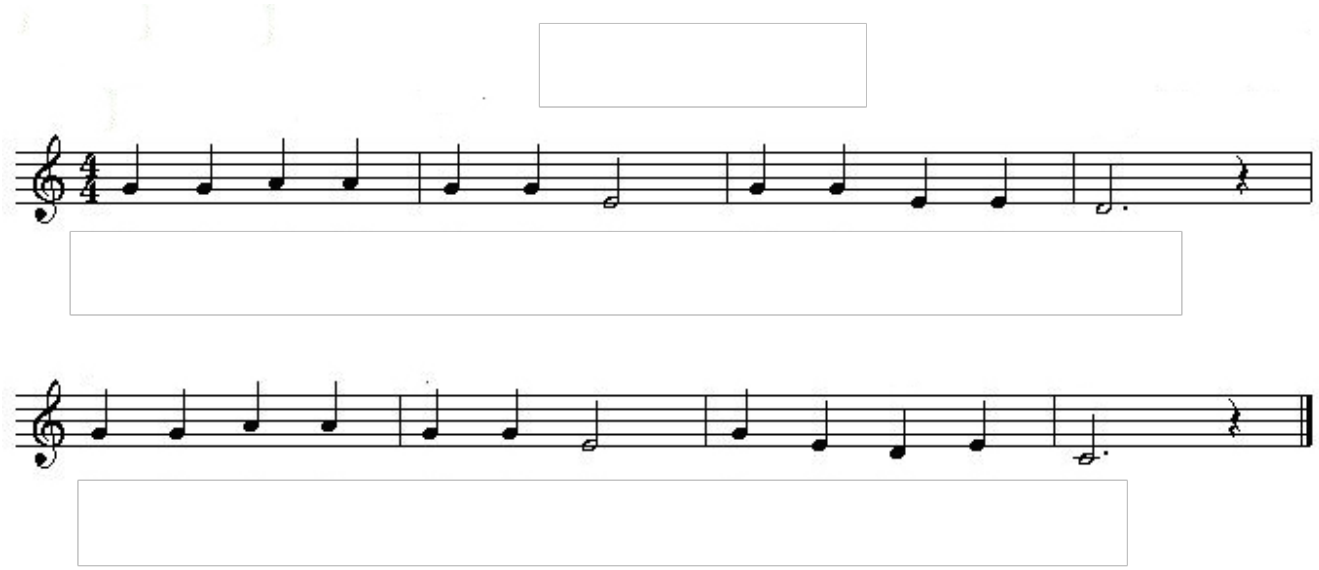
The following modules and functions will allow you to generate, load, and play a wav file in iPython. You don't need to understand how they work, but you should understand how to use them.

In [ ]:
```
import scipy.constants as const
import scipy
from scipy.io import wavfile
from IPython.core.display import HTML
from IPython.display import display
import numpy as np
import matplotlib.pyplot as plt
from __future__ import division
%matplotlib inline

# this is a wrapper that take a filename and publish an html <audio> tag to li
sten to it
def wavPlayer(filepath):
    src = """
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Simple Test</title>
    </head>

    <body>
    <audio controls="controls" style="width:600px" >
      <source src="files/%s" type="audio/wav" />
      Your browser does not support the audio element.
    </audio>
    </body>
    """%(filepath)
    display(HTML(src))

def playMusic(title, rate, music):
```

```
    # write the file on disk, and show in in a Html 5 audio player
    music_wav = music * 2**13
    wavfile.write(title + '.wav', rate, music_wav.astype(np.int16))
    wavPlayer(title + '.wav')
```

Generating a Song

Each key of a piano can be thought as a sinusoidal wave with a certain frequency. Let us consider the following score of a song, 'School Bell Ringing'.



Let us assume that each note is a half second long sinusoidal wave. Because the song consists of 32 notes, the song is 16 seconds long. If we specify a sampling rate of 44.1 kHz, then our song will have \(44100\times16\) samples in the entire song. The frequency associated with each note will be specified by the formula given for piano key frequencies (http://en.wikipedia.org/wiki/Piano_key_frequencies), namely that the frequency for the \(n^\text{th}\) key is given by

\[{\huge f(n) = 2^{\frac{n-49}{12}}\times 440 \text{Hz} }\]

The list \(\operatorname{notes}\) specified below holds the key values \(n\) for each note in 'School Bell Ringing' in sequential order.

In [ ]:
```
# C:40, D:42, E:44, F:45, G:47, A:49, B:51, Higher C:52
# 0 represents 'pause'
notes = [47, 47, 49, 49, 47, 47, 44, 44, 47, 47, 44, 44, 42, 42, 42, 0,
         47, 47, 49, 49, 47, 47, 44, 44, 47, 44, 42, 44, 40, 40, 40, 0]
```

From these \(n\) values, we can generate the sinusoids which represent this song as follows:

In [ ]:
```
note_duration = 0.5 # per key, in sec
rate = 44100 #44.1 khz
music_duration = note_duration * len(notes)
time = np.linspace(0, music_duration, num=rate*music_duration) # indices for 0
-16 secs spaced apart by 1/44100
song = np.zeros(len(time))
```

```
sinwave = lambda f,t : np.sin(2*np.pi*f*t) # generate a sinusoid of frequency
f Hz over time t
key2freq = lambda n : 2 ** ((n-49)/12) * 440 # convert a piano key n to a freq
uency in Hz

idx_note = 0
for note in notes:
    if note: # if note == 0, skip.
        freq = key2freq(note) # frequency of each note
        song[idx_note*rate*note_duration : (idx_note+1)*rate*note_duration-1]
= \
                sinwave( freq,
                        time[idx_note*rate*note_duration : (idx_note+1)*rate*n
ote_duration-1 ]
                        ) #generates the sinusoids for the song, in .5 sec int
ervals
    idx_note += 1
```

Now that we've generated the song, let's plot the first 0.01 seconds to see that we actually turned notes into sinusoidal waves.

In []:
```
plt.plot(time[0:441], song[0:441]) # Plot the first 0.01 second
plt.title("First 0.01 seconds of 'School Bell Ringing'")
plt.xlabel("Time")
plt.ylabel("Amplitude")
```

And now, let's listen to the song that's been generated.

In []:
```
playMusic('school_bell_original', rate, song)
print "Wow! What a great song. I should try playing other ones too."
```

Q1. Quantize the song

Now for the fun part. We want to quantize this audio signal and analyze the resulting error. We will use an ℓ-bit uniform-quantizer. We can use only $L = 2^\ell$ quantized values: $\{-1, -\frac{L-3}{L-1}, -\frac{L-5}{L-1}, ..., -\frac{1}{L-1}, \frac{1}{L-1}, ..., \frac{L-5}{L-1}, \frac{L-3}{L-1}, 1\}$ to represent each value of the original signal $-1 \leq x[n] \leq 1$. Our signal only takes values between -1 and 1, so our ℓ-bit uniform-quantizer allows our signal to take on 2^ℓ evenly spaced values in the range of interest. Each point in the original signal $x[n]$ is binned according to a nearest neighbor strategy.

a. Implement the quantizer method below, under the above conditions.

In []:
```
def quantizer(num_of_bits, original_signal):
    # Complete this quantizer method.
    # It should return a numpy array of the same length that contains quantize
```

```
d values of the original signal.

    # your beautiful code here... #

    return quantized_signal
```

b. What do you expect the quantized signal to look like if `number_of_bits` is set to 1?

1b. Your Answer Here

c. Quantize the song using your quantizer.Plot and compare the original signal and the quantized signal. Play around with `num_of_bits` to see how the song changes

In [ ]:
```
num_of_bits = 4
quantized_song = quantizer(num_of_bits,song)
# Plot the first 0.01 second of songs.
plt.figure(figsize=(10,6))
plt.plot(time[0:441], song[0:441])
plt.plot(time[0:441], quantized_song[0:441])
plt.legend(['Original Signal', 'Quantized Signal'])
plt.title('Effects of ' + str(num_of_bits) + '-bit quantizer')

playMusic('quantized_song_' + str(num_of_bits) + '_bits', rate, quantized_song
)
```

As noted before, the difference you are witnessing between the original signal and the quantized signal is the **quantization error**. As confirmed from the previous exercise, the quantization noise is almost negligible if one uses a sufficient number of bits. Let's take a look at the quantization error more carefully.

In [ ]:
```
# Run code in this cell to see the difference between the original and quantiz
ed signals (quantization error)
num_of_bits = 1
quantized_song = quantizer(num_of_bits,song)
plt.plot(time[0:441], song[0:441] - quantized_song[0:441])
plt.title('Quantization Error')
```

Q2. Plot the mean squared error between the original and quantized signal as a function of number of bits used for a quantizer. Consider $1 \leq \ell \leq 10$. Observe how fast the MSE drops as you increase ℓ. Is it linearly decreasing? Or is it exponentially decreasing? First plot the MSE, then change the y-axis to a log-scale.

Note: The MSE is the l^2-norm (http://mathworld.wolfram.com/L2-Norm.html) of the quantization noise divided by the number of noise samples.

In [ ]:
```
# Q2. Your Beautiful Code Here
```

You should see that the MSE does indeed drop exponentially! This result was easily confirmed via simulation, so let's do some calculations to obtain the same results via analysis.

Sampling & Quantization: Analysis

In order to analyze the effect of quantization noise, we need to come up with a good model for the quantization noise. Although everything we did was deterministic, what we saw above was really just one specific realization of quantization. For each sample, the original value was mapped to the nearest quantization point. The distance between the chosen quantization point and the original value became the quantization noise for the sample. But how bad is the error on the whole?

Let's just start by modelling the noise as a **uniform random variable**. If the original signal is assumed to take each value chosen at uniformly random, the model will be precise. However, is this model also valid for structured signals such as a combination of sinusoidal waves? Let's take a look at a 'noise histogram' to find out.

In [ ]:
```
num_of_bits = 8
# time, song = sampler(sampling_rate)
quantized_song = quantizer(num_of_bits, song)
matplotlib.pyplot.hist( (song - quantized_song ) , 30 );
plt.title('Noise Histogram')
plt.xlabel('Quantization Error')
plt.ylabel('Occurances')
```

We see that the quantization error is not exactly uniform, but extremely close. There is one value which occurs significantly more frequently than the rest, but that is an artifact of the structure of a sinusoid (they like to hang around -1 and 1 for a while). If we consider general signals, the model of quantization noise as uniform is even more accurate, although it is not too bad in this case either.

Q3. Given that the error between a quantized signal and the original is modeled as a uniform random variable, justify that the MSE will drop exponentially. Find an equation for $\mathbb{E}[\text{MSE}]$ given an arbitrary number of bits ℓ. On a log scale, plot the expected MSE given the uniform model along with the achieved MSE for the signal above for $1 \leq \ell \leq 10$. Do they decay similarly?

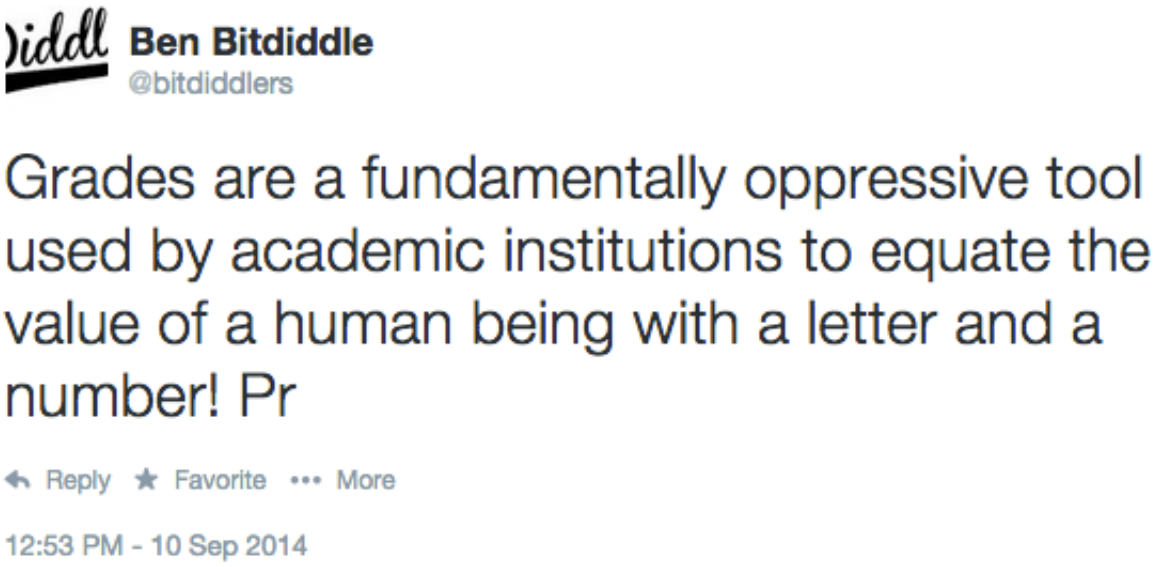
Q3. Your Solution Here

In [ ]:
```
# Q3. Plotting Code Here
```

Compression: Twitter Revolution

Before attempting this section, brush up on (or learn for the first time) Huffman coding (http://www.arbitrarylabs.com/huffman.pdf).

Your friend Ben Bitdiddle just returned from Tunisia and wants to start a revolution at UC Berkeley. He knows Twitter can be an important tool for revolution, but it has a major problem: 140 characters just isn't enough. For example, Ben wants to send the tweet:



The tweet exceeded 140 characters and he had to send it in **2 separate tweets**, which dramatically reduces the impact of his revolutionary message. If people read the first tweet, they won't know where the protest is located; if they read the second, they may get to the protest, but they won't know why they are protesting!

Ben has devised a new scheme to *compress* his tweets so he can fit his longer messages into 140 characters. There are only 32 characters that he uses in his tweets (26 letters + 6 punctuation), but not all of them occur equally often, so he will use a Huffman code as follows: 1. He will encode his tweet as a binary string using a Huffman code based on the letter frequencies determined by the hash table `freq_dict` below. 2. He will convert this bit-string back into letters according to the `bits2letters` mapping defined below. If the bitstring created by the Huffman code does not have a length which is a multiple of 5, he will append the Huffman code for the character `SPACE` to the bitstring until it *does* have length which is a multiple of 5.

In []:
```
english_freq_dict = {'!': 0.0008, "'": 0.0004, ' ': 0.1033, ',': 0.0011,\
             '.': 0.0011, '?': 0.0006, 'a': 0.069, 'c': 0.027000000000000003,\
             'b': 0.013999999999999999, 'e': 0.1, 'd': 0.039, 'g': 0.021, 'f':
 0.022000000000000002,\
             'i': 0.06, 'h': 0.048, 'k': 0.008, 'j': 0.0014000000000000002, 'm
': 0.025,\
             'l': 0.043, 'o': 0.065, 'n': 0.06, 'q': 0.0017000000000000001, 'p
': 0.02,\
             's': 0.059999999999999994, 'r': 0.052000000000000005, 'u': 0.0270
00000000000003,\
             't': 0.083, 'w': 0.017, 'v': 0.0108, 'y': 0.018000000000000002, '
x': 0.0036, 'z': 0.0012}
```

In []:
```
letters2bits = {'!': '11101', "'": '11111', ' ': '11010', ',': '11100', \
           '.': '11011', '?': '11110', 'a': '00000', 'c': '00010', 'b': '0000
1', 'e': '00100', \
           'd': '00011', 'g': '00110', 'f': '00101', 'i': '01000', 'h': '0011
1', 'k': '01010',\
           'j': '01001', 'm': '01100', 'l': '01011', 'o': '01110', 'n': '0110
1', 'q': '10000', \
           'p': '01111', 's': '10010', 'r': '10001', 'u': '10100', 't': '1001
```

```
1', 'w': '10110', 'v': '10101',\
            'y': '11000', 'x': '10111', 'z': '11001'}

def reverse_dict(original_dict):
    # create a new dict with the keys of original_dict as values and the value
s of original_dict as keys
    new_dict = {}
    for key in original_dict:
        new_dict[original_dict[key]] = key
    return new_dict

bits2letters = reverse_dict(letters2bits)
```

Characters and their frequencies in the English language:

Character	Frequency
SPACE	0.1033
e	0.1
t	0.083
a	0.069
o	0.065
i	0.06
n	0.06
s	0.056
r	0.052
h	0.048
l	0.043
d	0.039
c	0.027
u	0.027
m	0.025
f	0.022
g	0.021
p	0.02
y	0.018
w	0.017

b	0.014
v	0.0108
k	0.008
x	0.0036
q	0.0017
j	0.0014
z	0.0012
,	0.0011
.	0.0011
!	0.0008
?	0.0006
'	0.0004

Q4. Implement Huffman. In this question we will develop Ben's strategy for Huffman Coded Tweets.

a. Implement a method that, given a list of frequencies, will output the corresponding mapping of input symbol to Huffman codewords

In [ ]:
```
from heapq import heappush, heappop, heapify
from collections import defaultdict

def HuffEncode(freq_dict):
    """Return a dictionary (letters2huff) which maps keys from the input dicti
onary freq_dict
       to bitstrings using a Huffman code based on the frequencies of each key
"""

    # Your Beautiful Code Here #

    return letters2huff
```

b. Using this method, generate the mapping from our alphabet to their corresponding Huffman codewords. Print this mapping in a readable format.

In [ ]:
```
letters2huff = HuffEncode(english_freq_dict)
huff2letters = reverse_dict(letters2huff)
# print dictionary in readable format
```

c. Write a method to encode a string using this Huffman mapping into a binary string and then use the `bits2letters` mapping above to turn this binary string back into english characters. These characters will be different than the original message, and there should be fewer of them. Encode the string containing the desired tweet (provided below as `original_string`).

In []:
```
def encode_string(string, letters2huff):
    """Return a bitstring encoded according to the Huffman code defined in the
 dictionary letters2huff.
       If your resulting bitstring does not have a length which is a multiple
of five, add the binary for
       SPACE characters until it is."""

    # Your Beautiful Code Here

    return encoded_string
```

In []:
```
original_string = 'grades are a fundamentally oppressive tool used by academic
 institutions to equate the value of a human being with a letter and a number!
 protest coryhall tonight'
encoded_string = encode_string(original_string)

print "Original String: " , original_string
print "Length of Original String: " , len(original_string)
print "Encoded String: " , encoded_string
print "Length of Encoded String: " , len(encoded_string)
```

Can you successfully post your message now?

d. Write a function to decode a Huffman coded tweet back into the original english tweet that generated it. The result should be that

```
>>> decode_string(encode_string('hello world', letters2huff), huff2letters)

"hello world"
```

In [4]:
```
def decode_string(coded_string, letters2huff):
    """Translate from a Huffman coded string to a regular string"""

    # Your Code Here #

    return decoded_string
```

In []:
```
print decode_string(encode_string('hello world', letters2huff), huff2letters)
print decode_string(encode_string(original_string, letters2huff), huff2letters
)
```

After implementing the coding system described above, Ben was able to tweet the following (138 characters), while still conveying his full, original message:

Wow! That's the kind of tweet that will really get me excited to revolt!

Q4. Are Huffman codes unique? Should you be worried that the tweet you generate may not precisely match the tweet Ben generated above?

A4. Your Solution Here

Congratulations, you can now quantize children's songs and Huffman code tweets. You're really moving up in the world!

In [ ]: