EE-3221 LABORATORY
Quantization Error

Overview
In this lab exercise you will investigate sampled audio signals and then explore the effects of quantization error through MATLAB simulation.

Sampled Audio in MATLAB
A vector of samples in MATLAB can be played as an audio signal using:

```
sound(x, fs, BITS)
```

where x is the vector (or sequence) of samples, fs is the sampling frequency of the sequence, and BITS is the number of bits used to represent the audio signal in the digital-to-analog converter (DAC) of your sound card. If fs and BITS are not specified, they default to 8192 samples/second and 16 bits/sample.

Sinusoids
Run MATLAB and create an index vector that will correspond to 2 seconds of time.

```
fs = 8192;          % fs = 8192 samples/second, default sample rate
Ts = <?>            % sample period Ts = 1/fs seconds/sample & is inverse of fs
t_max = 4;          % max time is 4 seconds
N = t_max/<?>       % determine the number of samples that corresponds to 4
seconds of a sampled sequence (care must be taken that N is an integer; here
the values are arranged so it will be, but try help round and doc ceil for
common ways to convert floating point numbers to integer values)
n = 0:1:(N-1); % maximum value is N-1 since we start at 0
% alternatively we can define N = fs*t_max
```

Now we create the sine wave signal. Let's create a 440 Hz (concert pitch "A") tone. We take the desired frequency and multiply it by 2π/fs or 2π·Ts, which are equivalent. Note that Ts*n is the sampled time in seconds.

```
f_A = 440               % concert pitch A is 440 Hz
x1 = sin(2*pi*f_A*Ts*n);
```

We can listen to the signal through the computer's sound card using **`sound(x, fs)`**. **Always turn down the volume on your computer before playing a sound. The full scale volume is very loud!**

```
sound(x1, fs)
```

```
sound(0.1*x1, fs)
```

The second call attempts to scale the output voltage by 0.1. We would expect this to result in $20\times\log_{10}(0.1) = -20$ dB power gain, which would clearly give an audible difference. This rests on a few assumptions, the key one being that the sound card driver isn't one that automatically increases the gain for quiet sounds in an attempt to be helpful.

Next create a sine wave signal that represents an 880 Hz tone.

```
x2 = <?>
```

1. *What is the expression used to find the sequence vector for x2?*

Listen to x2:

```
sound(x2, fs)
```

2. *Describe the differences between the two sounds. What causes the difference?*

Musical Chords

In western music, a chord is a combination of tones that are related by ratios of the fundamental tone. For example, an A-major chord consists of an A note at 440 Hz, a C# note at about 1.26*440 Hz, and an E note at about 1.5*440 Hz. Those who are musically inclined may know that the exact frequency multipliers for the 3 components are 0, 4, and 7 semitones above the root, which can be computed in MATLAB with 2.^([0 4 7]/12)

Let's build the tones for the notes that make up an A-major chord.

```
x1 = sin(         440*2*pi/fs*n);
x2 = sin(2^(4/12)*440*2*pi/fs*n); % Approx. 1.26 of fundamental
x3 = sin(2^(7/12)*440*2*pi/fs*n); % Approx. 1.5  of fundamental
xs = [x1 x2 x3]; % explain what this does
sound(xs, fs)
xc = mean([x1; x2; x3]);  % explain what this does
sound(xc, fs)
```

3. *Explain what **[x1 x2 x3]** and **mean([x1; x2; x3])** do. Hint: Examine the sizes of the inputs and outputs.*
4. *Suppose you were to examine the frequency spectrum of signals **xs** and **xc** (e.g., using a spectrum analyzer). Describe the ways in which the spectrum of **xs** and **xc** are similar. Describe how they differ.*

.wav Audio Files

Download the file plumclip.wav from https://faculty-web.msoe.edu/prust/EE3221 .
To find MATLAB's default folder on your system, start MATLAB and enter pwd ("print working directory"). This default folder is a convenient place to save the WAV file. You can change MATLAB's current folder at the top of the command window or with the cd ("change directory") command. To import the file into a vector, type:

```
[original_clip, fs] = audioread('plumclip.wav');
```

This .wav file has a sampling rate of 44100 Hz with 16 bits per sample. These are standard industry specifications for CD quality audio. Listen to the clip:

```
sound(original_clip, fs)
```

Low Resolution Samples: The Sound of Quantization Noise and Signal to Noise Ratio (SNR)

The original .wav file used 16 bits to represent the amplitude of each sample. This represents a resolution of the amplitude equal to 2^{16} = 65,536 different levels. To hear the effect of the integer approximations used in quantization, let's reduce the resolution to 6 bits per sample.

5. *How many possible amplitude levels are possible when 6 bits are used to quantize the signal?*

To obtain the integer values present in the original .wav file, we could rescale the sound vector original_clip, but it is easier to read it directly:

```
integer_original_clip = double(audioread('plumclip.wav', 'native'));
```

double() causes the signed integers, which range from -32,768 to 32,767, to be stored in MATLAB's default double-precision floating point data type, which makes them easy to work with. Note that these integers are small enough that they can be stored as doubles with no loss of precision. We can check that these integers when divided by 2^{15} are exactly equal to the scaled values loaded earlier by calculating the error between them:

```
norm(integer_original_clip/pow2(15)-original_clip)
```

Now, reduce the resolution to 6 bits by dividing by 2^(16-6) and using round() to round the results of the sequence values. Remember, our original clip has 16 bits of resolution. Dividing a binary number by 2 discards the least significant bit and dividing by 2 ten times will discard the lowest 10 bits. This leaves only the 6 most significant bits out of the original 16 bits.

```
integer_lowres_clip = round(integer_original_clip/pow2(10)); %discard 10 bits
integer_lowres_clip = min(integer_lowres_clip,pow2(5)-1);
```

The last line limits the output to a maximum of 31; replacing any out-of-range values of 32 with 31. Now, each sample ranges from -32 to 31 since we are down to 6 bits of resolution. Note that MATLAB uses doubles to store the integer results in this case; MATLAB also supports true integer data types, but they generally aren't used unless utmost efficiency is required. Scale and offset the 6-bit integer vector back to the standard range of -1 to 1 for playing in MATLAB and listen to the result. Note that the maximum value isn't quite 1, but 31/32, approaching it closely.

```
lowres_clip = integer_lowres_clip/pow2(5);
sound(lowres_clip, fs); % make sure you have the correct fs here.
```

6. *Describe what you hear. What might be causing this?*

Let’s create a function that will let us hear what the clip will sound like with different resolutions ranging all the way down to 1 bit. Here’s the function:

```
function xq = ChangeBitRes(x, Nbits)
% x - original clip in the range -1 to nearly 1, [-1,1)
% Nbits - number of bits in output
L = 2^Nbits; % number of levels
xq_int = floor((x+1) * (L/2)) - L/2; % quantization level, details below
% The steps are:
%   +1: shift to [0,2)
%   *(L/2): shift to [0,L), still floating point
%   floor: for non-negative, discard fractional part: [0,L) as integer
%   -L/2: shift to signed range [L/2,L/2-1]
% For example, if Nbits is 6, there are 64 levels in the range [-32,31].
% If the user accidentally inputs a value of exactly +1, it will get
% quantized to, e.g., 32, which doesn't fit in a 6-bit signed integer.

% Now, convert back to values in [-1,1)
xq = (xq_int + 1/2) / (L/2); % quantized value (+1/2 rounds to nearest)
```

Save the function and experiment with the song using different resolutions. For example, try:

```
sound(ChangeBitRes(original_clip,12), fs) % 12 bits used to quantize amplitude
sound(ChangeBitRes(original_clip(1:fs*5), 6), fs) % limit to first 5 seconds
sound(ChangeBitRes(original_clip, 4), fs)
sound(ChangeBitRes(original_clip, 1), fs)
```

Rounding and quantization adds error to each sample. The amount of error in each sample does not follow any particular pattern, so it sounds like a shhhh-ing or static sound that we call white noise.

7. *Listen to the audio clip using 1 bit of resolution.* ***IMPORTANT: Be sure to turn down your laptop volume, as the noise will be quite loud.*** *You should be able to recognize the song. Explain why the song is still recognizable despite the resolution being just 1 bit.*

Quantization Error and SQNR

As observed in the previous exercise, quantization of a sampled signal results in errors between the original (unquantized) amplitudes and the quantized amplitudes. This difference is referred to as *quantization error* and is often modelled as a noise that is added to the original signal. The ratio of the signal power to the quantization noise power is known as the *signal-to-quantization-noise ratio*, or *SQNR*.

$$SQNR = 10\log_{10}\frac{P_x}{P_n}$$

P_x = signal power

P_n = quantization noise power

In this portion of the exercise, we will carefully simulate the quantization process, examine characteristics of the quantization noise, and compare measured SQNR to theoretical predictions.

A well-known result in the DSP community is that SQNR increases by 6.02 dB for each additional bit used in the quantizer. The exact SQNR for a given signal depends on various factors, including the statistics of the signal itself. For example, the SQNR of a sinusoidal signal can be shown (under a set of assumptions) to be

$$SQNR = 6.02b + 1.76 \text{ dB}$$

where b is the number of bits used in the quantizer. For example, the SQNR of a sinusoid sampled using a 14-bit quantizer is, in theory, 6.02(14)+1.76 dB = 86.04 dB. Since 86.04 dB corresponds to 4.02E8, the signal power in the quantized signal is *400 million* times larger than the quantization noise power!

Let's now simulate a quantizer in MATLAB. We begin by generating samples of a sinusoid:

```
fs = 1000;
Ts = 1/fs;
t = 0:Ts:1;
x = sin(2*pi*4*t); % 4 Hz sinusoid
```

By default, MATLAB uses "double" precision (according to IEEE Standard 754) in calculating these sample points, which utilizes a 64-bit floating-point representation. Suffice to say, the representation is extremely accurate.

We now create a 3-bit quantized version of the sinusoid, paying careful attention to placement of the quantization levels.

```
xq = ChangeBitRes(x,3);
```

We can plot the original signal x and the quantized signal xq with the following:

```
figure
plot(t,x,t,xq)
xlabel('t')
legend('x','xq')
```

Carefully inspect this plot, paying special attention to the placement of the quantization levels. Note that values of x are "rounded" to the nearest quantization level. This particular quantizer is known as a "midrise" quantizer.

The quantization error can be calculated as

```
e = x-xq;
```

8. *Create a plot showing x, xq, and e on the same time axis (i.e., one figure showing all three signals). Label the time axis and include a legend. Include this plot in your submittal. What is the range of e, and how does this range correspond to the step size of the quantizer?*

We now calculate the simulated SQNR as

```
S = mean(xq.^2); % signal power is mean square value
Q = mean(e.^2); % noise power is mean square value
SQNR = 10*log10(S/Q); % in dB
```

You should find the SQNR to be 18.7 dB. The theoretical SQNR for a 3-bit quantizer is 19.8 dB.

9. *Repeat this simulation for each of the quantizers listed in the table below. Complete the table and include in your submittal. Interpret the results.*

Number of bits	Simulated SQNR (dB)	Theoretical SQNR (dB)
2		
4		
8		
12		
16		

A modified version of the quantizer can be simulated by replacing the previous quantization function with one that truncates (i.e., rounds down) instead of rounding to the nearest level.

```
function xq = ChangeBitRes_Truncate(x, Nbits)
% x - original clip in the range -1 to nearly 1, [-1,1)
% Nbits - number of bits in output
L = 2^Nbits; % number of levels
xq_int = floor((x+1) * (L/2)) - L/2; % quantization level, details below
% The steps are:
%   +1: shift to [0,2)
%   *(L/2): shift to [0,L), still floating point
%   floor: for non-negative, discard fractional part: [0,L) as integer
%   -L/2: shift to signed range [L/2,L/2-1]
% For example, if Nbits is 6, there are 64 levels in the range [-32,31].
% If the user accidentally inputs a value of exactly +1, it will get
% quantized to, e.g., 32, which doesn't fit in a 6-bit signed integer.

% Now, convert back to values in [-1,1)
xq = xq_int / (L/2); % quantized value (always rounds down)
```

Notice the line that computes xq differs from the previous quantizer.

10. *Create a plot showing x, xq, and e on the same time axis (i.e., one figure showing all three signals) for this* ***3-bit*** *"truncating" quantizer. Label the time axis and include a legend. Include this plot in your submittal. What is the range of e, and how does this range correspond to the step size of the quantizer?*
11. *Simulate this "truncating" quantizer using 8 bits. What is the resulting SQNR? How does this SQNR compare to the SQNR for the 8 bit "rounding" quantizer that was previously simulated? Explain the results. (Hint: Use the plot you generated in the previous question to support your explanation.)*