EE-3220 LABORATORY

Week 2

MATLAB Discrete Audio Waveforms

Goal – Investigate sampled audio signals containing single and multiple frequencies in both the time and frequency domains.

Background – A sequence generated in MATLAB can be played as an audio signal using:

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

where x is the sequence, 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.

Procedure – 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 = 2;        % max time is 2 seconds
N = t_max/<?>     % determine the number of samples that corresponds to 2
seconds of a sampled sequence (care must be taken that n_max 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
```

1. *Show that N = fs*t_max results in the maximum sequence integer for a given sampling frequency and t_max. That is, show this calculation agrees with the one for N above.*

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.

2. *Experiment with different scaling factors. What is the largest factor less than 1 that allows you to hear a difference? What is the smallest positive factor that gives a sound that you can hear at all? State what type of laptop/computer you are using and what type of speakers (built-in, headphones, external speakers, with model number if available) you are using.*

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

```
x2 = <?>
```

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

Listen to x2:

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

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

Musical Chords

In western music, a chord is a combination of tones that sound as if (and perhaps are) sounded simultaneously 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)
```

5. *Explain what **[x1 x2 x3]** and **mean([x1; x2; x3])** do. Hint: Examine the sizes of the inputs and outputs.*

Viewing the Spectrum of a Sine Wave

The FFT, which we will discuss later, can be used to plot the spectrum, or frequency domain representation, of a signal. The FFT is a measure of "how much energy" is in each frequency of the signal sequence. To get an idea of what this means, we create a series of 4 s tones and view it on the oscilloscope, first in normal mode, then in FFT mode.

```
t_max = <?>       % max time length to 4 seconds
N = t_max/<?>     % number of samples in a 4 s sampled sequence
n = 0:<?>   % % step size defaults to 1 when omitted
x1 = sin(440*2*pi/fs*n);
x2 = sin(660*2*pi/fs*n);
x3 = sin(880*2*pi/fs*n);
x = [x1 x2 x3];
sound(x, fs)
```

Attach the eighth-inch-to-banana cable available from TSC to the headphone output of your laptop and to the oscilloscope. Play the three-tone sequence you just created and view it on the oscilloscope. Make sure the scope setting shows the wave in its entirety without clipping or it could affect the FFT results to come.

Now follow the instructions on the oscilloscope FFT handout to view the frequency content (aka spectrum) of the signal. Keep in mind that you will be viewing a signal with frequencies 440, 660, and 880 Hz, so set the viewing span and center appropriately.

6. *Play the sound again and describe what happens to the FFT picture on the scope.*

Next play:

```
x_sum = [.1*(x1+x2+x3)  .1*(x1+x2+x3)  .1*(x1+x2+x3)];
% Look at the documentation for repmat for a better way to do this.
```

7. *Describe the differences of the FFTs for x and x_sum observed. What were the peak values for each? Why? Check your results with max() and min(). **Demonstrate** this to the lab instructor.*

The FFT shows you "how much energy" is in each frequency of the signal sequence. If you play a song with a lot of bass tones into the oscilloscope you will see activity in the lower frequencies, and high notes create activity in the higher frequencies.

Viewing the Spectrum of Music

Download the file plumclip.wav from https://faculty-web.msoe.edu/prust/EE3220 .
MATLAB's default folder is D:\MyDocs\Documents\MATLAB so that might be a good place to save it. You can change MATLAB's current folder at the top of the command window. 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. Detach the audio cable from your laptop and listen to the clip:

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

Now attach the audio cable and show the spectrum of the clip on the oscilloscope.

8. *What sounds do the spikes in the FFT represent?*

Sampling Below the Nyquist Rate: The Sound of Aliasing

Now let's investigate the effects of aliasing due to low sampling frequency by downsampling an audio clip.

The original audio clip was obtained by $x(n) = x_a(nTs) = x_a(n/fs)$. If we decrease the sampling rate, we will increase the sampling period by the same ratio. For example, if Ts = 1/44100, then x(1), x(2), x(3), x(4), … represent the values of $x_s(t)$ at t = 1/44100, 2/44100, 3/44100, 4/44100, etc.

We can "downsample" the sequence by retaining fewer elements from the original sequence. If we downsample by a factor of 2, we would keep the sequence values of $x_s(t)$ at t = 1/22050, 2/22050, …

9. *What values from the original clip sequence would need to be kept if we downsampled by a factor of 2? That is, what are the values for n?*

Let's create some sampled sequence tones 4 s long with a sample rate of fs = 2000 samples/sec and examine them on the scope with the FFT.

```
fs = <?>
N = 4*fs    % check units (4 s)*(fs samples/s) has units of?
n = 0:(N-1);
y = (sin(400*2*pi/fs*n) + sin(700*2*pi/fs*n)) / 2;
sound(y, fs)
```

10. *What does y represent and how is this verified by the FFT? Explain.*

We downsample the sequence, y, to a new sample rate of 1000 samples/sec , which is a factor of 2, by selecting every other sequence value in y(n). That is y(0), y(2), y(4), …etc.

The downsampled sequence will be a subset of the original sequence. Individual elements and parts of vectors can be accessed easily in MATLAB. One of the ways to get a subset of the vector in MATLAB is to provide only the indices that you want to keep. Note that the first MATLAB index always starts with 1. Also, each vector has a property called **end** that is equal to the last index. This will make it easy to downsample the sequence. Type:

```
y_ds = y(1:2:end);      % start at index 1, then take every 2 indices,
                        % until end
sound(y_ds, fs/2)
```

11. *Describe what you hear and what you see on the FFT for y_ds and compare to the FFT for y. What frequency does the 700 Hz tone map to in the downsampled sequence?*

Equivalently, we could have simply resampled the signal y directly, using fs = 1000.

```
fs = <?>
N = 4*fs    % check units (4 s)*(fs samples/s) has units of?
n = 0:(N-1);
y_new= .5*(sin(400*2*pi/fs*n) + sin(700*2*pi/fs*n);
sound(y_new, fs)
```

12. *Compare the first ten samples of y_ds and y_new in MATLAB. Are the sequences the same?*

Let's listen to what happens when we downsample 'plumclip.wav' by a factor of 20. Recall, this had an original sample rate of 44100 samples/s.

```
downsampled_clip = original_clip(1:20:end);
```

The downsampled clip has a new sampling rate of 44100/20 = ? samples/s. To play the new clip, type:

```
sound(downsampled_clip, <?>)  % compute the sampling rate for fs/20
```

Notice the effect. Every note above ½ times the sampling frequency, 1102.5 Hz (High C) in this case, will be "aliased" to a wrong note. The lower notes still sound correct. This aliasing effect occurs when the sampling rate is too low to properly represent the higher frequency sine waves present in the song.

We will discuss aliasing in detail later in the course.

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 8 bits per sample.

13. *How many possible amplitude levels are possible when 8 bits are used to quantize the signal?*

To do this, we will make use of MATLAB's round function, which rounds to the nearest integer.

To obtain the integer values present in the original .wav file, we must rescale the sound vector original_clip, which currently takes values from -1 to 1. We want to represent the sound vector the way it was stored in the .wav file, with each sample represented by a 16-bit unsigned integer. The following will offset the signal so it ranges from 0 to 2 instead of -1 to 1, and then scale the samples so they range from 0 to 2^16.

```
integer_original_clip = (original_clip+1) * pow2(15);
```

Now, reduce the resolution to 8 bits by dividing by 2^(16-8) 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 eight times will discard the lowest 8 bits. This leaves only the 8 most significant bits out of the original 16 bits.

```
integer_lowres_clip = round(integer_original_clip/pow2(8));
```

Now, each sample ranges from 0 to 255 since we are down to 8 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 8-bit integer vector back to the range of -1 to 1 for playing in MATLAB and listen to the result:

```
lowres_clip = integer_lowres_clip/pow2(7) – 1;
sound(lowres_clip, fs); % make sure you have the correct fs here.
```

14. *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 bit resolutions between ranging from 2 to 16 bits. Here's the function:

```
function new_audio_clip = ChangeBitRes( original_audio_clip, resolution_bits )
% Convert the amplitude of the original 16-bit audio clip to an unsigned
% integer range from 0 to 2^16.
integer_original_clip=(original_audio_clip+1)*(pow2(16-1));
% Discard 16-resolution_bits to keep a vector that is in the integer range
% of 0 to 2^(resolution_bits)
integer_lowres_clip = round(integer_original_clip/(pow2(16-resolution_bits)));
% Scale the amplitudes that range from 0 to 2^resolution_bits back to the
% range of -1 to +1.
new_audio_clip = integer_lowres_clip/(pow2(resolution_bits-1))-1;
end
```

Save the function and experiment with the song using different bits of resolution. For example, try:

```
sound(ChangeBitRes(original_clip,12), fs)     % 12 bits used to quantize amplitude
sound(ChangeBitRes(original_clip, 6), fs)     %  6 bits used to quantize amplitude
sound(ChangeBitRes(original_clip, 4), fs)     %  4 bits used to quantize amplitude
```

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 sound which we call white noise. We will discuss this effect later in the course as well.

Play the low resolution clip into the oscilloscope and observe the FFT.

15. *What do you see that is different compared to the original clip in the FFT?*

16. *Describe, qualitatively, what happens when fewer bits are used to represent the samples.*

Submittal

The submittal should follow all instructions on the provided grading checklist and cover sheet.

Provide short answers (in sentence format) to the questions listed in italics.

Additional Problems

Submit answers to the following additional problems with your lab. Be sure to include any MATLAB code.

- P2.4.1.
- P2.4.3. Hint: Note that the right signal being added contains a folded (time-reversed) component, x(1-n) == x(-n + 1), which can be found by flipping the input from left to right (so that the value at the lowest time index moves to the highest time index, etc.) and then delaying it for one sample. The input is non-0 for n = -3…3, so the flipped version will also be non-zero for n = -3…3 and after delaying it will be non-zero for n = -2…4. Note that x(n+1) is non-0 for n = -4..2. Thus, we need to let n = -4:4 to cover all the samples and insert extra 0s so the 2 parts are the right size for sample-by-sample multiplication: [0 0 fliplr(x)] .* [x 0 0].
- Calculate by hand the convolution y(n) = x(n) * h(n) for the following sequences and verify your result using the conv function in MATLAB. Both sequences start at n=0. x = [2, -5, 3, -1, 2, 6]; h = [5 -4 -3 2 1];