18-290 Signals And Systems **Fall 2012**
Lab 2 **Total Points: 100**
Issued: Monday, September 17, 2012 Due: In Lab
Files Needed: `loudness.mat`, `amplitudes.mat`, `ode.mat`, `speech.wav`, `clap1.wav`, `clap2.wav`, `clap3.wav`

Note: It is recommended that you look through the In-Lab section before your Lab session. If you have questions, please come to the lab office hours.

1 Introduction

The goal of this lab is to study the properties of signals and linear time-invariant systems that you learnt in class (and perhaps some that you didn't). The most commonly found simple signal, which is continuous and periodic, is the sine wave. A sine wave makes for a good "test case" for most systems, and in fact, signal processing people like the sine wave so much, they try and represent other signals in the form of sines to analyze it. You will learn this process later in class. As for now, we will stick to the sine because it has only two parameters — the amplitude, or "how big the signal is", and the frequency, or "how fast the signal is" — and is easy to handle using math.

Sine waves are found everywhere in nature. A laser emits light, which is basically sine waves of electromagnetic fluctuations. A sine wave is also the most fundamental signal that you experience when you hear sounds. A sine wave of a fixed amplitude and fixed frequency is called a "pure" tone. Let's try generating it and listening to it. A sine wave x as a function of time t, of amplitude A and frequency f Hz is given by:

$$x(t) = A\sin(2\pi f t).$$

If this is a sound wave, the units of A (and therefore x) are in terms of air pressure. If you are generating sound through electronic means, however, x can represent the voltage you need to apply to the terminals of the speaker, and the units will be volts. Since we are using Matlab, all we need to do is generate a wave with no units — Matlab will properly scale it and generate the sound.

2 Pre-Lab (20 points)

If any of you are familiar with music, you will know the note A4 or A440 or "Concert A". As its name suggests, it has a frequency of 440 Hz and serves as a general tuning standard (ISO 16) for musical pitch. Let us generate the note A4, for 5 seconds. (For all the audio procedures we do for this lab, we shall keep the sampling frequency at 16 kHz.)

Let's begin by specifying the sampling frequency `Fs`:

```
>> Fs = 16000;
```

From this, we need to generate the time step between samples `Ts`:

```
>> Ts = 1/Fs;
```

Now let's generate the time stamps `t`:

```
>> t = (0:Ts:5)';
```

Let's set the frequency `f` at which we want to generate the sine wave:

```
>> f = 440;
```

And the amplitude `A`, although this will not matter since we will use `soundsc`:

```
>> A = 1;
```

We already found out in Lab 1 that certain Matlab functions can take vector inputs and return the appropriate output. So all we need to do to generate `x` is use the formula directly:

```
>> x = A*sin(2*pi*f*t);
```

Awesome, it looks like we have our signal. Let's try and plot the first 37 samples and see if the signal is indeed what we expect:

```
>> plot(t(1:37),x(1:37));xlabel('Time (s)');ylabel('Amplitude');axis tight;
```

Looks like we fortuitously selected almost exactly one cycle of the signal. Anyway, the sine wave seems alright. Let's try and listen to it:

```
>> soundsc(x,Fs);
```

So this is what a pure tone sounds like.

Or is it? What is the pitch and how is it different from the frequency? To put it simply, frequency is what is present, the physical parameter. Pitch is the *sensation* of the frequency. Frequency is an objective, scientific concept, whereas pitch is subjective. Sound waves themselves do not have pitch, and their oscillations can be measured to obtain a frequency. It takes a human brain to map the internal quality of pitch. In the labs, however, we will use pitch and frequency interchangeably.

Now, how about the loudness of a signal? Take a sine wave of the same amplitude, one at 100 Hz and another at 1000 Hz. Both cause the same amplitude of fluctuations in the air, but we will sense the 1000 Hz tone as noticeably louder, because our ears are more sensitive to that frequency. Thus both A and f contribute to how loud we perceive the sound to be.

Let's delve into this further. How much more or less amplitude should a sine wave have at 100 Hz, 125 Hz, 1.25 kHz, etc., in order to sound as loud as a sine wave of amplitude 1 at 1 kHz? If you plot this "required amplitude" as a function of frequency, what you get is an "Equal-Loudness Contour". If you plot this amplitude for different loudness, the graph looks like as shown in Figure 1.

The file `loudness.mat` gives you two variables, `spl` and `freq`. `freq` gives you a list of 29 frequencies, and `spl` (short for sound pressure level) gives you the amplitude required for the sine wave (in dB) to hear it at a loudness of 45 dB. For 1 kHz, for example, this is approximately 45.0106 dB and for 100 Hz, this is 68.0227 dB. How do you convert this to amplitudes? An amplitude ratio r are converted to the decibel difference d using the formula $d = 20 \log_{10} r$, thus, the inverse formula is $r = 10^{\frac{d}{20}}$. Consider a sine wave of amplitude 1, at 1 kHz. To sound just as loud at 100 Hz, the difference in dB is $d = 68.0227 - 45.0106 = 23.0120$ dB. The corresponding ratio calculated is $r = 14.1450$. Thus, the amplitude has to be 14.145 times as much.

Using the two vectors given, write down one of the given frequencies, in kHz, at which the human ear is the most sensitive. **(2 points)**

Let's concentrate ourselves to one octave, E3 through E4. The notes and the corresponding frequencies are given in Table 1.

The file `amplitudes.mat` gives you two variables, `octave` and `amplitudes`. `octave` contains the frequencies of Table 1 in order. `amplitudes` gives the amplitudes necessary to make them of equal loudness.

What is an octave? It is supposed to be the interval between one musical frequency and another with half or double its frequency. Look at E3 and E4: the frequency of E4 is double that of E3. An octave is divided into 12 semitones, given in Table 1, usually such that the ratio of the frequencies of a note and the note immediately preceding it is equal for all notes. Test this out. There are 11 such ratios possible, use *one* Matlab command to find them all and save them in one column vector. Write down this command and the corresponding output in your notebook. **(2 points)**

Now, hang on. If the ratio for each semitone is constant at say ρ, and jumping 12 such semitones doubles your frequency (by definition of an octave), then ρ^{12} must equal 2. Verify this for all ratios using a single Matlab command on the column vector generated above, and write down the command used, along with the resulting output. **(1 point)**

To work with these frequencies, we need to make a function that, when given the note, the time duration, and the sampling frequency, will output the tone with the proper frequency, length and amplitude. Using the variables already provided, create such a function named `generateTone`. *You can assume that the notes*

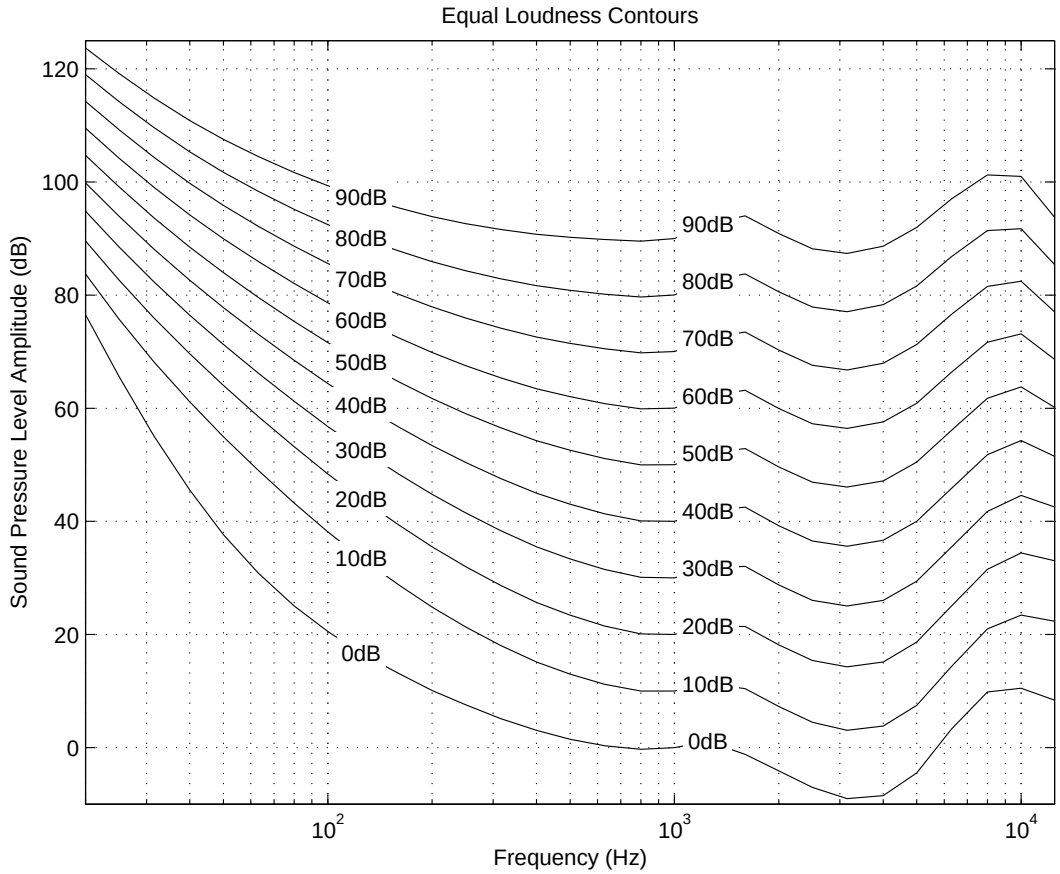

Figure 1: Equal Loudness Contours. Data calculated as per ISO 226.

Note	Frequency (Hz)
E3	164.814
F3	174.614
F♯3/G♭3	184.997
G3	195.998
G♯3/A♭3	207.652
A3	220.000
A♯3/B♭3	233.082
B3	246.942
C4	261.626
C♯4/D♭4	277.183
D4	293.665
D♯4/E♭4	311.127
E4	329.628

Table 1: Some notes and their frequencies.

given will not have sharps or flats or the note E5, so you can work with only 7 out of the 12 frequencies, i.e, E3, F3, G3, A3, B3, C4 and D4. That means the input will be a single character, containing an uppercase letter from A through G only. You might need to use the `switch` and `case` commands. (To load `amplitudes.mat` inside the function, use the command `load amplitudes.mat`.) Take a print out of this function, and submit it with the pre-lab. Add comments using the character `%` and explain the steps of your code. *Under-commented code will lose you points.* But don't over-comment your code. Yes, there is such a thing as over-commenting, although the threshold is pretty high. If you manage to reach and cross this threshold, we need to talk, and you will lose points too. **(15 points)**

Please keep the code for this function (and the files included in this lab) with you when you come to the lab, either via email, dropbox, USB drive, or on your own computer. You will need to use it in the lab, and if you cannot access your code, you will have to type it out again using the print out of your code.

To help you, here is how the first line of your `generateTone.m` file should look like:

```
function [output] = generateTone(f, T, Fs)
```

3 In-Lab (60 Points)

1. You have learnt convolution in class. For the non-Matlab questions below, write down the answers in your lab notebook and show how you worked out the solution. Just writing the answer will not get you any points. Let $x = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix}$.

   (a) Without using Matlab, calculate the convolution of x with itself. Let the output be y. **(2 points)**

   (b) Without using Matlab, calculate the convolution of y with x. Let the output be z. **(2 points)**

   (c) Without using Matlab, calculate the convolution of z with x. Let the output be a. **(2 points)**

   (d) Now use Matlab to calculate these for you, using its convolution function `conv`. Plot the four signals x, y, z and a in a single window. You don't have to specify anything for the x-axis. **(2 points)**

   (e) Without using Matlab, calculate the convolution of y with itself. Verify that it is the same as convolving z with x. Prove why they are the same using convolution properties. **(2 points)**

2. Remember the function we wrote in the last lab, `myEcho1(x, d, a)`? Let's use it on some real speech. Load the file `speech.wav`. You might want to test playing it in Matlab.

   (a) Now apply the function `myEcho1` to it. Play around with the parameters `d` and `a` to make it sound somewhat realistic. Write down your final values of the parameters. I will ask you to play the echo during checkout. **(5 points)**

   (b) Is the echo function an LTI system? Why or why not? Support your answer with a proof. Determine the impulse response of the function for your values of `d` and `a`. **(5 points)**

   (c) Use Matlab convolution function `conv` to write a function `myEcho2(x, d, a)` that is equivalent to `myEcho1(x, d, a)`. Write the code in your lab notebook. Test it on `speech.wav`. Write a single command that allows you to see that the outputs of both the functions are exactly equal (in Matlab and in your notebook). **(5 points)**

3. Last time, almost all of you answered that the echo model used can be made better by adding more delayed and attenuated signals. This answer is absolutely correct for reverberations in a room. The parameters are determined by the shape and the dimensions of the room, the absorbing materials used, and the frequency of the sound. Such reverberations are desired in auditoriums as it helps overcome the drop-off of sound intensity with distance — that is the reason for the shapes of the walls and materials used for their construction. A particular enclosure is said to be "live" if it takes longer for the sound to die away. Absorbent enclosures, where the sound dies away quickly are said to be acoustically "dead".

   Approximately, the reverberation effect can be viewed as an LTI system, and as you learnt in class, an impulse response is all that is required to completely characterize the system. It is impossible to

produce an exact continuous impulse, but a clap is a good approximation of one. The files `clap1.wav`, `clap2.wav` and `clap3.wav` are recordings resulting from a single loud clap in three different enclosures, thus approximating their impulse responses.

(a) Generate a plot containing the plots of all three recordings one below the other. (Don't forget to label the x-axis with the appropriate time units). **(5 points)**

(b) Using `conv`, convolve these recordings with the speech recording of the earlier problem. Listen to the three outputs and rank the enclosures in order of increasing "liveness". **(5 points)**

4. Now we will write a function to generate some music, `playMusic(notes, beats, Fs)`. Suppose I give you a list of notes to play, and specify that 2 notes should be played per second. I want you to write a function that will take these two inputs, along with the sampling frequency, and generate a single wave output that I can play on Matlab.

   We will play a very simplistic rendition of "Ode to Joy". Converting the half notes to 2 quarter notes, the notes to be played for this tune are "BBCDDCBAGGABBAAABBCDDCBAGGABAGGGAABGACB-GACBGGADDBBCDDCBAGGABAGGG"; make sure the notes are within the E3 to E4 octave, as shown in Table 1. The file `ode.mat` contains the variable `ode` with these notes represented as a character array. The function you write should take this variable, the number of notes per second (2 in this case) and the sampling frequency (`Fs`). The function header should be as follows:

   ```
   function [output] = playMusic(notes, beats, Fs);
   ```

   The output should be a column vector, such as can be played using `sound` (*not* `soundsc`) as follows:

   ```
   >> sound(music, Fs);
   ```

   Obviously, you can use the function `generateTone` that you have written during pre-lab, and a `for` loop. Note the absence of sharps and flats and the note E5, so `generateTone` will suffice. First find the time for which each quarter note has to be played. Then take each character in the array one by one, generate the sine wave for each using `generateTone` and in the end, concatenate all of them together. Take a print out of your code for `playMusic(notes, beats, Fs)` or write it down in your lab notebook. Commenting is again essential. Save this function with you for further usage. **(15 points)**

   I will ask you to play the music you generate during checkout. Music that sounds significantly different from what it should be will lose points.

   If you generated the music piece as described above, you should observe "clicks" at several points during the playback, around the time one note ends and another begins. Find out why this happens. It might help if you plot a small part of the signal around such a click. What could be done to avoid these clicks? You don't have to write a new function code to generate the music without clicks, but just write down what you think in the lab notebook. **(5 points)**

   A lowpass filter takes in a signal and cuts out frequencies above a certain frequency cutoff. Let's apply a lowpass filter to the music you have above. If you put the musical signal above through an ideal lowpass filter with cutoff around 270 Hz, what do you expect to hear? (Look back at Table 1) Now you will create a filter we provide and pass the melody above through it. Given that `x` is your signal, and `Fs` is your sampling frequency, execute the following commands:

   ```
   >> [b, a] = butter(12, 270 / (Fs / 2), 'low');
   >> y = filter(b, a, x);
   >> sound(y, Fs);
   ```

   Did you hear what you expected? Why or why not? For now, you don't have to understand what is going on in the above commands nor the specifics of filtering, though you can find more information using the `help` command. **(5 points)**