Lab 1 - Music Synthesis with Sinusoidal Signals

Introduction

Sinusoids are used in a wide variety of applications. Perhaps the most obvious one is in music. A musical note is a simple sinusoid of the form $Acos(2\pi ft + \varphi)$, with amplitude A, frequency f, and phase φ. A melody, or musical voice, is composed of an ordered list of notes and rests. A musical composition is simply a sum of melodies. Every pulse in a musical composition is therefore a sum of sinusoids of the form:

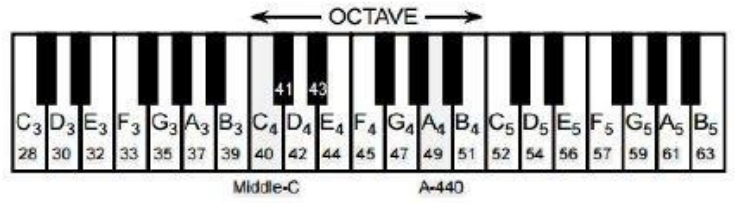
$$x(t) = \sum_k A_k \cos(\omega_k t + \varphi_k)$$

In this lab, you will synthesize some musical pieces, including *Mary Had a Little Lamb*, *Fugue #2* from the *Well-Tempered Clavier* by Bach, *Hava Nashira* by Barukh Rhode, and *Treat You Better* by Shawn Mendes.

1. Lab Part One

1.1 A Function to Play a Note

Piano keyboards are laid out as illustrated by the following image:



Keys on a piano

- C4 refers to the C-key in the fourth octave. C4 is also called middle-C.
- A4 is also called A-440, because its frequency is 440 Hz.
- Every key is given a key number. The key number of middle-C is 40.
- The frequency of any key can be found by substituting its key number into the formula:

$$f = 440\left(2^{\frac{keynumber-49}{12}}\right)$$

Remember that when we construct a sinusoid in MATLAB, we are in fact constructing a sampled sinusoid:

$$x[n] = x(nT_s)$$

The Sampling Theorem tells us that to recover a continuous signal from its samples without aliasing, f_s must be at least two times greater than the maximum frequency component.

Built-in function soundsc(xx,fs) reconstruct the continuous-time version of xx, i.e. . Use this to judge how your synthesis is doing as you work on the lab. In music synthesis, common choices for sampling frequency are 8000 Hz, 11025 Hz, or 44100 Hz.

Write a function to produce a desired note for a given duration at a given complex amplitude. Use the following skeleton code to write your function:

```
function xx = key_to_note(X, keynum, dur)
%{
KEY_TO_NOTE: Produce a sinusoidal waveform corresponding to a given piano key number
Input Args:
    -X: amplitude (default = 1)
    -keynum: number of the note on piano keyboard
    -dur: duration of the note (in seconds)
Output:
    -xx: sinusoidal waveform of the note
%}
fs = 8000;
tt = 0:(1/fs):dur-1/fs;
freq = %<============== fill in this line
xx = real(_______); %<================ fill in this line
end
```

For freq, use the formula given above to determine the frequency for a sinusoid in terms of its key number. Fill in xx to generate the actual sinusoid as the real part of a complex exponential at the proper frequency.

1.2 Synthesize a Song – Mary Had a Little Lamb that NEVER Grew Up!

Multiple notes can be played in order by concatenating their row vectors as follows:

```
xx = [x1 x2];
```

Use key_to_note to write a script, play_mary.m, that plays a series of notes. Use the following skeleton code to write your script:

```
% --------------play_mary.m-------------- %
mary.keys = [44 42 40 42 44 44 44 42 42 42 44 4747];

%Notes: C D E F G
%Key #40 is middle-C

mary.durations = 0.25 * ones(1,length(mary.keys));
fs = 8000; % 11025 Hz also works
xx= zeros(1, fs*sum(mary.durations));
n1 = 1;
for kk = 1:length(mary.keys) keynum = mary.keys(kk);
tone = % <------- Fill in this line
n2 = n1 + length(tone) - 1;
xx(n1:n2) = xx(n1:n2) + tone; %<------- Insert the note
n1 = n2 + 1;
end
soundsc(xx,fs);
```

1. *Generate the sound and play it for a TA.*
   Have your TA check you off for this section.
   (If you can't finish this in time for the TA to check it off, submit it as a .wav file.)

2. *Plot the frequency-time spectrogram of Mary* using the function using specgram(xx,512,fs) .

   Notice that you can zoom in to show the order of notes. Try varying the window length (the NFFT keyword argument in help specgram) to get different looking spectrograms if needed. **Note**: if specgram doesn't work, use plotspec (see help plotspec instead).

1.3 Structures

You may have noticed that mary in the previous section was a structure. MATLAB allows for structures, which group information together. A structure is used to represent information about something more complicated than what can be held by a single number, character, or boolean or a list of any one of them. For example, a Student structure can be defined by his or her name (type str) , GPA (type float) , age (type int) , UFID (type int) , and more. Each of these pieces of information can be labeled with an easily understood descriptive title, and then combined to form a whole (the structure).

Structures give us a way to "combine" multiple types of information under a single variable. The nice thing about structure is that they allow us to use human-readable descriptions for our data. For example, every note in a melody can be characterized as its note number, a start pulse, and a duration. We can define the note numbers, durations, and start pulses as separate arrays:

```
melody_noteNumbers = [40, 42, 44, 45, 47, 49, 51, 52]
melody_durations = [1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5, 1.5]
melody_startPulses = [1, 3, 5, 7, 9, 11, 13, 15]
```

But fetching all three properties of each note would require accessing three different variables:

```
noteNumber = melody_noteNumbers(2);
duration = melody_durations(2);
startPulse = melody_startPulses(2);
```

Fetching the start pulse of the third note would look like this:

```
melody_startPulses(2)
```

Structure is almost ALWAYS a better way to go then to use nested arrays. To learn more about structures, read MATLAB's official documentation here.

1.4 The Evenly-Timed First Voice

Now that you've received an introduction to musical notes, you're going to play something a little more complicated.

You have been given a data file called barukh_fugue.mat, which contains an array of structures called theVoices, a structure array of the same form as melody above. theVoices is composed of three voices, each voice of which is characterized by three vectors: durations, noteNumbers , and startPulses. The first voice's notes are accessed as the vector:

```
theVoices(1).noteNumbers
```

Modify your play_mary code to create a new script, play_first_voice_even, to *play each note in the first voice of the Barukh Fugue (from Signals.mat)* for 0.5 seconds each.

Have your TA check this off. If you can't finish this in time, submit as a .wav file on Canvas.

1.5 The Correctly-Timed First Voice

Something sounded a little bit off with that, didn't it? That's because in the melody, the notes aren't meant to all be played for the exact same amount of time. Each note is meant to be played for a specified duration, or number of pulses. Provided that there are no intended breaks between notes, a melody can be constructed from a note numbers and durations. The durations can be accessed the same way as the note numbers:

```
theVoices(1).durations(i)
```

Modify your play_first_voice_even code to create a new function, play_first_voice, to *play each note in the first voice for its correct duration of pulses*, with each pulse being 0.15 seconds long.

Have your TA *check this off*. If you can't finish this in time, submit as a .wav file on Canvas.

1.6 Silence and startPulses: Construction of the Better Fugue

Balance is an important part of music. Sometimes, multiple musical voices are simultaneously playing. But sometimes, a given voice is silent in favor of another voice. Sometimes, both voices are silent. We can denote silent periods as time after one note has ended, but before the next note has begun. Suppose a note of note number 49 starts at pulse 13 and has duration 2; and the next note of note number 52 starts at pulse 16 and has duration 1. Then the musical voice is to play Middle A for two pulses, it is to be silent for one pulse, and then it is to play High C for one pulse. Thus, by incorporating information about start pulses, in addition to note numbers and durations, we can incorporate silent periods into our musical reconstruction.

Using the startpulses, durations, and noteNumbers properties of each note in barukh_fugue, construct the three voices in the Barukh Fugue and add them together.

The data file better_fugue.mat is of a similar structure as barukh_fugue, also containing an array of voices, but this time with five voices. Using your barukh_fugue code: Add the five voices, from better_fugue.mat, together to produce the Better Fugue. Note that there may have been errors in the way that you produced the barukh_fugue that aren't noticeable in the better_fugue; the better_fugue is a bit harder.

Produce the Better Fugue at 0.15 seconds per pulse, *play it to a TA for checkoff.* If you can't finish in time, save it as a .wav file, and submit it on Canvas alongside your lab document.

Hint: Next section provides some helpful directions.

This ends Part 1 of this lab.

Helpful Hint

Remember to reference each voice in the song. For example, if you wanted to find the total number of notes in a particular voice, you would have to say:

```
length(theVoices(i).noteNumbers)
```

The skeleton code for your function is provided below:

```
function song = playSong(theVoices)
    %{
    PLAYSONG: Produce a sinusoidal waveform containing the combination of the different notes in theVoices
    Input Args:
        -theVoices: structure contains noteNumbers, durations, and startpulses vectors for multiple voices.
        -Output:
        -song: vector that represents discrete-time version of a musical waveform
        -Usage: song = playSong()
    %}

    load barukh_fugue.mat;
    fs = 8000;
    spp = %seconds per pulse, theVoices is measured in pulses with 4 pulses per beat
    song = %Create a vector of zeros with length equal to the total number of samples in the entire song

    %Then add in the notes
    for i =1:length(theVoices)
        for j = 1:length(theVoices(i).noteNumbers)
            note = %Create sinusoid of correct length to represent a single note
            locstart = %Index of where note starts locend = % index of where note ends
            song(locstart:locend) = song(locstart:locend) + note;
        end
    end
    %soundsc() or audiowrite()
end
```

Timing

The time duration of a single pulse will determine the speed of the song. This is typically specified in beats per minute. You can perform a series of conversions to convert beats per minute into seconds per pulse, given that pulses_per_beat = 4 and beats_per_minute = 120:

```
beats_per_minute = 120;
beats_per_second = beats_per_minute / 60;
seconds_per_beat = 1 / beats_per_second;
seconds_per_pulse = seconds_per_beat / 4;
```

2. Lab Part Two

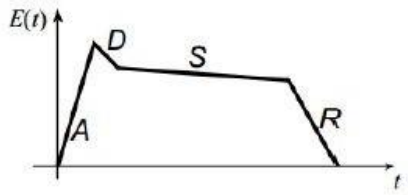
2.1 Construction of the Bach Fugue

The bach_fugue.mat song is of the same structure as barukh_fugue.mat , also containing a list of voices. *Synthesize the bach_fugue . Have your TA check this off.* If you can't finish this in time, submit as a .wav file on Canvas.

2.2 Musical Tweaks - Enveloping

After creating the bach_fugue, you may notice that the product sounds artificial. This is because it is created from pure sinusoids. In this section, you will incorporate an envelope into your music synthesis process in order to improve the quality of the final product.

When played by an instrument, notes are not single pure evenly-amplified tones. Their amplitudes change over the course of a single note, with attack, decay, sustain, and release parts of a note, together making up an 'ADSR' profile. Enveloping creates short pauses between musical notes, as a scaling vector where all values are between 0 and 1. Envelopes can be implemented in the synthesis process by multiplying each note by an enveloping function $E(t)$.



ADSR envelope

Your envelope should be a mathematical representation of the above. You can get such a curve by using the linspace(). Alternatively, you can use the hanning() function (see help hanning) , which will create a bell-shaped curve. The curve does not exactly represent $E(t)$. However, it creates similar effects.

Implement an envelope to improve the sound quality of your Bach Fugue. Play it for a TA for checkoff. If you can't finish it in time, submit a .wav file on Canvas.

Note: grading here is based on your ability to improve the sound quality. This part is intentionally left vague; we want you to be creative here!

2.3 Musical Tweaks – Fourier Series of a Trumpet

Remember that any real-world periodic signal can be written as a sum of sinusoids whose frequencies (in Hertz) are integer multiples of the fundamental frequency. The amplitudes and phases of the sinusoids are all that is needed to specify the Fourier series expansion. Musical signals are a natural application of Fourier series, since musical signals (specifically, an instrument playing a specific note) are periodic. The sinusoid with the same period as the musical pitch is called the fundamental; it is the pure tone that sounds most like the pitch. To obtain the richer sound of an instrument playing a note, we add in harmonics; musicians call them overtones. Harmonics are

sinusoids at frequencies double, triple, etc. the frequency of the fundamental. The amplitudes of the harmonics determine the timbre (sound) of the instrument playing the note. The reason a violin playing note B# sounds different from a trumpet playing note B# is that the amplitudes of the harmonics are different.

The Fourier series representation of a musical signal is specified by only a few numbers (the amplitudes and phases of the harmonics, and the note):

$$x(t) = \sum R\left\{ A_k e^{jw_\delta kt} e^{-j\phi_k} \right\} = \sum A_k \cos(w_\delta kt - \varphi_k)$$

In practice, the infinite series is truncated to a finite number of terms, giving a finite Fourier series. Finite Fourier series still give good approximations to most periodic signals.

Since instruments playing musical notes create periodic signals, musical signals have Fourier series expansions. The Fourier series can be truncated to a finite number of terms and still do a good job of representing the musical signal. The timbre (sound) of the trumpet is produced by the amplitudes A_k . The fundamental frequency alone would be a pure tone; the overtones (harmonics) create the richer sound of the trumpet.

Here is a table of and values for nine harmonics:

***k*- Harmonic**	A_k **- Relative Amplitude**	φ_k **- Relative Phase**
1 (fundamental)	0.1155	-2.1299
2	0.3417	+1.6727
3	0.1789	-2.5454
4	0.1232	+0.6607
5	0.0678	-2.0390
6	0.0473	+2.1597
7	0.0260	-1.0467
8	0.0045	+1.8581
9	0.0020	-2.3925

Using a sampling frequency of 44100 Hz, *construct your Bach Fugue in trumpet. Play it for a TA for check off*. If you can't finish in time, submit a .wav file on Canvas.
Question 1: Suppose the maximum frequency in the Bach Fugue is 1200 Hz. What is the minimum sampling frequency needed to synthesize, without aliasing, a trumpet sound containing nine harmonics?

2.4 Extra Credit – Synthesize a Musical Piece of your own!

For up to fifteen points of extra credit, out of 100 for the entire lab, synthesize a piece of music into a song, calling it yourname_song. Save it as yourname_song.mat. Submit your code, explanation, and the .wav of your musical composition.

Extra Credit Grading:

- A simple composition might get 5 points.
- A more complex one might get 10.
- A really well-done one might earn the full 15 points.

Again, be creative!

We look forward to hearing your creations!