Music Synthesis

Adina Goldberg - agoldberg@comm.utoronto.ca; David Ding - davidy.ding@mail.utoronto.ca

Abstract
This experiment introduces the relationship between the time domain and the frequency domain through music synthesis.

Contents

Background

Cell Arrays

Please go to http://www.mathworks.com/help/matlab/cell-arrays.html and its links on this page to get an extensive overview of what cell arrays are in MATLAB. In a nutshell, cell arrays are like matrices, except in each index you can have any data type, not just a number.

Note: Most common mistake from past experience is that students access cell arrays using () like matrices. This is not correct. You must use {} to access contents of cell arrays.

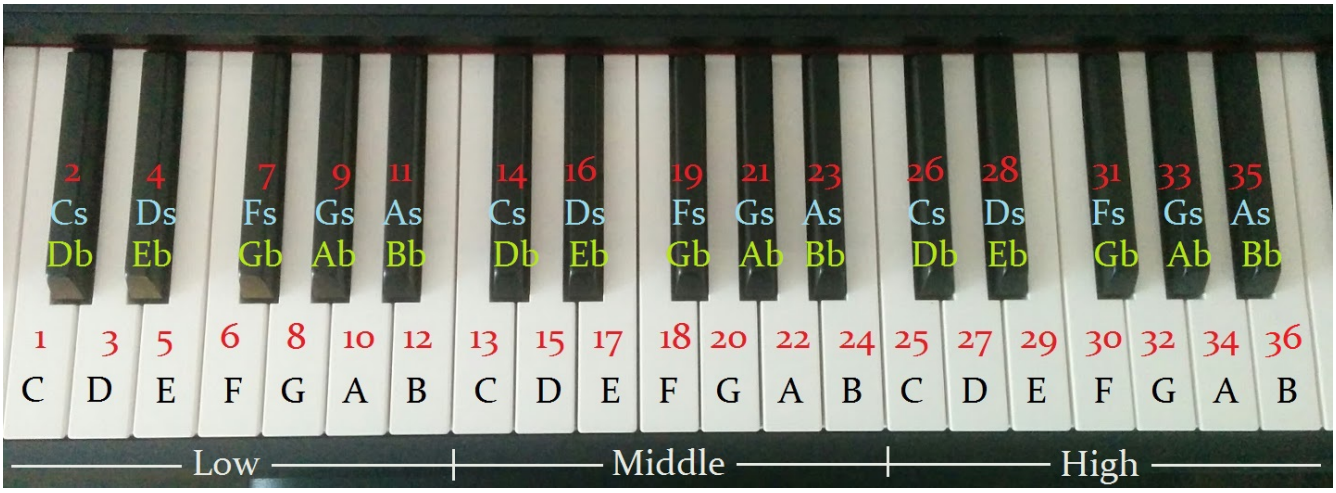
Example:

```
>> my_song_array = {'C', 1; 'Am', 0.5; 'G', 1; 'C', 0.5};
%my_song_array is a 4 x 2 cell array.
%To access the value of the field 'G', at index (3, 1), do:
>> my_song_array{3, 1}
ans =
G
```

Representing music

Note: Section 2.6.2 of your ECE216 Course Notes provides extensive explanation of how frequency of sounds work with respect to a piano keyboard. All students are expected to go over this section carefully prior to the lab. The following blurb only provides an overview.

Each musical note produced by a musical instrument corresponds to a sinusoid wave with a unique frequency. The higher the note, the higher the frequency value, and vice versa. When looking at a piano keyboard, moving up one octave (7 white



keys, or equivalently, 12 keys total) corresponds to doubling the frequency of the current note. Any two neighbouring notes have a constant frequency ratio, which must be $2^{\frac{1}{12}}$.

Notes are labelled with letters of the alphabet, and the labels repeat every octave. On a piano keyboard, the white keys are labeled C, D, E, F, G, A, B and the black keys in between can be referred to in terms of their neighbouring white keys. The black key above (to the right of) a D is called a D♯ (say 'dee sharp') and the black key below (to the left of) a D is called a D♭ (say 'dee flat'). This can be seen in the image above.

For the purposes of this lab, we will work with a small set of notes comprising three octaves, which we have labeled 'Low', 'Middle', and 'High'. We have numbered each note, as indicated in the keyboard image, in order to make notes easier to work with computationally.

In this lab, we will be working with chords, which are groups of three or more notes played at the same time. One reference for chord names and corresponding component notes is
`http://www.piano-lessons-info.com/chart-of-piano-chords.html`

You can also see how to build chords visually on a piano here:
`http://www.onlinepianist.com/chords/`

We will be using a sequence of chords to build a song.

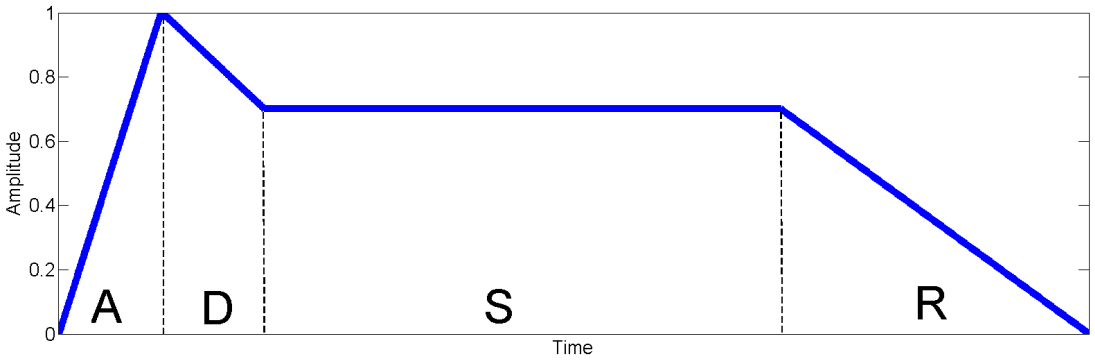
Synthesizing music

When synthesizing music, the fundamental *frequency* of each note is used to parameterize a corresponding sinusoidal wave. In this lab, to play a note with fundamental frequency f, we will produce the wave: $\cos(2\pi ft - \frac{\pi}{2})$. (The $-\frac{\pi}{2}$ shift is used so that the signal at time zero has value zero.) This creates a pure, clean sound, but one which doesn't really resemble mechanically produced music. When playing a musical instrument, hitting a key or plucking a string induces vibrations at certain other frequencies. The additional sounds are referred to as *harmonics*. We don't synthesize the harmonic frequencies in this lab, which is one of the many examples of how we are using a simplified model of real sound in order to perform music synthesis.

However, we can still improve our model by manipulating synthesized signals in order to make them more realistic. Different manipulations can produce different instrument sounds, rhythms, volumes, etc.

At the end of this lab, we will multiply a sequence of synthesized chords by an envelope function in order to change the sound by introducing strumming. We will use an ADSR envelope. ADSR stands for "Attack, Decay, Sustain, Release". The ADSR envelope can be thought of as a continuous function comprised of four sections, each of a specified duration. The attack section of the function starts from zero and is increasing. The decay section is slowly decreasing. The sustain section is constant. The release section decreases the function to zero. These sections correspond to roughly what happens when, for instance, one presses a piano key. As the key is pushed down, the volume rises to a peak (depending on how hard the key is hit), corresponding to the attack section, then decays to a given level until the key is released. After that, the volume decreases to zero, corresponding to the release section.

Enveloping is a time domain operation, done by pointwise multiplication (MATLAB command is .*) of two time signals. However, other sound modifications are more easily done in the frequency domain. Any manipulation that's done in one domain can be performed in the other, but often with much greater difficulty. This is part of the motivation for working in both domains.



Working with audio in MATLAB

Audio in MATLAB is stored as a column vector (or two column vectors, in the case of stereo) of amplitude samples of the real sound wave. In order to store good quality sound, it is important for these samples to be sufficiently close together. Theory regarding this will be covered later in the course, but intuitively it should make sense. In order to play back sound faithfully in MATLAB, it is necessary to know the sampling frequency of the data. The frequency mostly used in this lab is 11025 Hz. MP3 files usually have a sampling frequency of 44100 Hz.

Throughout this lab, you will use the function `soundsc(X,fs)` to listen to a sound signal `X` which was stored with a sampling frequency `fs`.

In MATLAB, audio can be imported from files and exported to files (with extensions such as `.wav` and `.mp4`) using the functions `audioread(filename, fs)` and `audiowrite(filename,y,fs)`. These functions will prove useful if you'll want to save the song you produce.

How to read and plot a spectrogram

A spectrogram is a way of visualizing the change in frequency components of a signal over time. The vertical axis corresponds to time and the horizontal axis corresponds to frequency. The colour of the graph at a particular time, t, and frequency, f, indicates the magnitude of frequency component, f, in the neighbourhood of t.

For spectrograms created in MATLAB, hotter colours (i.e. red) indicate greater magnitudes, and cooler colours (i.e. blue) indicate lesser magnitudes.

In order to plot the spectrogram of a signal `y` sampled at frequency `fs`, execute `spectrogram(y,[],[],[],fs)`. The middle three arguments can be left empty and will revert to their default values. Type `help spectrogram` into MATLAB in order to learn more about what you can do by modifying the arguments.

It might be helpful to execute the command `colorbar` after plotting a spectrogram, to get a colour legend.

Modelling a song

Here is some terminology that will help you in Section 4 of this lab. There will be two main components to the songs we produce, which will work in parallel to create the sound. One is the sequence of chords, and the other is the strum pattern. We will define both of these terms shortly. The lengths of chords and strums will be measured in units called *beats*. The number of seconds per beat will be called the *beat duration*. A shorter beat duration corresponds to speeding up the song, and a longer beat duration corresponds to slowing it down.

The sequence of chords will be characterized by a list of chord names, and a corresponding number of beats for which each chord is to be played. You will store these in a *cell array*, for example:

```
>> my_song_array = {'C', 1; 'Am', 0.5; 'G', 1; 'C', 0.5; ... };
```

A strum pattern is essentially a rhythm. It lasts for a specified number of beats, and then repeats. It may sometimes span multiple chords, and may sometimes repeat multiple times over the duration of a single chord. If you try listening to a song, and tapping along to the beat, this tapping pattern should eventually repeat, and that should give you an idea of your strum pattern. In order to specify a strum pattern to put into MATLAB, we will use an increasing sequence of fractions in $[0,1)$. If you wanted 4 equal strums, for example, your strum pattern would be specified as $(0,\frac{1}{4},\frac{1}{2},\frac{3}{4})$. If you wanted two equal short strums followed by one long strum, your strum pattern might look like $(0,\frac{1}{8},\frac{1}{4})$. Strum patterns don't specify absolute lengths of time or numbers of beats, but just relatively how long each strum is held for.

In order to create each strum in the strum pattern, the function `adsr_env` uses one ADSR envelope, with the duration of each segment determined by the duration of the strum.

1. Preliminaries

1. Open `MATLAB`.

2. Set the current directory of MATLAB to your local directory (similar to Lab 2, we suggest that you create a new folder called ”Lab3”).

3. Download the ”Lab3-Code.zip” zip file from Blackboard and unzip its contents into your current working directory set up in the previous step.

4. At different points in this lab, you will be asked to complete various `MATLAB` functions by filling in lines of code according to comments in the files. An example of an incomplete `MATLAB` function that you'll need to fill in is provided in Appendix B.

5. You will also need to use some built-in `MATLAB` functions throughout the lab, such as `soundsc`. To figure out how to use any built-in function, just execute the command: `help <name of function>`.

6. You will be producing data in this lab that you will need to use in subsequent sections. Make sure to save your variables.

2. Build some chords and listen

1. Complete the function `get_freq` to output the frequency of a note in Hz, given the key number as an input. Get the frequency of key number 9. (See *Representing music* in the *Background* section for help.)

2. Complete the function `get_wave` to produce a vector containing the time signal, $\cos(2\pi ft - \frac{\pi}{2})$, for a note with a given name and duration. (Note that you should use the provided function `get_num`.)

3. Use `get_wave` to generate a time signal for middle C at 11025 Hz for 2 seconds. Listen to the signal using `soundsc`.

4. A C major chord is comprised of the notes: C, E, and G. Use the middle octave. Create the time signal for a C chord by (a) creating the time signal corresponding to each component note and then (b) summing the three signals. The chord should last for 2 seconds, and have a sampling frequency of 11025 Hz. Listen to the synthesized C chord using `soundsc`.

5. Complete the function `get_chord_wave` which will return the time signal for a chord given the name of the chord, duration, and sampling frequency. This function uses the `switch/case` environment. Some common chords are already predefined for you as cases in this function. If you need to use more chords later, simply add cases. You may need to look up how to access elements of a cell array.

6. The function you just completed automates what you have already done manually for a C chord. Use `get_chord_wave` to do the same thing for two other chords: ‘G’ and ‘Am’, both of which are predefined. The chords should last for 2 seconds each, and have a sampling frequency of 11025 Hz. Listen to each of the synthesized chords using `soundsc`.

3. Visualize frequency of recorded and synthesized sounds

1. Look at the time domain signals of the following chords synthesized in Matlab vs. those that are produced on the piano in figure 1. What main difference do you see between the real and synthesized signals in the time domain? Be sure to explain the cause of this difference.

2. Look at the frequency spectrum of the signals of the following chords synthesized in Matlab vs. those that are produced on the piano in figure 2. What main difference do you see between the real and synthesized signals in the frequency domain? Be sure to explain the cause of this difference. Also explain the similarities observed here.

3. Look at the spectrogram of a synthesized C chord shown in figure 3, and interpret the changes you see as you move along the x-axis and the y-axis of the graph. What does this spectrogram have in common with the frequency spectrum of the synthesized C chord found in figure 2?

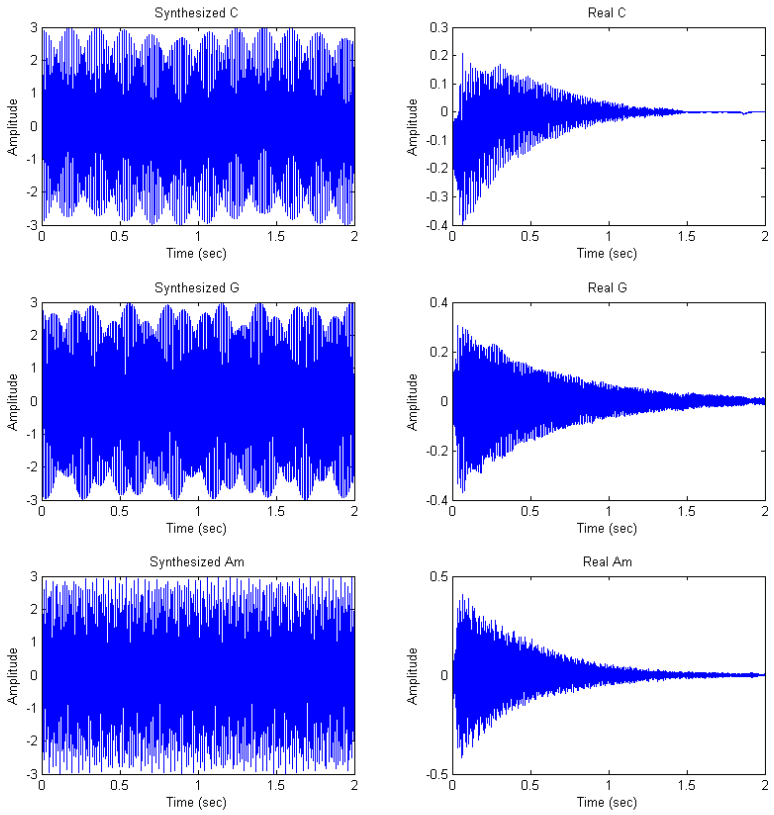


Figure 1. Matlab-synthesized chords vs. real chords played on the piano

4. Synthesize a song

1. Complete a function called `get_song_wave` which will take in a song notated in a cell-array (an example is `lua` in `songs.mat`, which you can use for testing) and return the time signal given a beat duration and a sampling frequency. It may help to look up how to access data in cell arrays.

   Note: A ”song” here is a cell array of size n x 2, where ”n” is the number of chords in the song. Each row of this cell array denotes a chord (first column) and how many beats (second column) this chord lasts (number of beats can be less than 1, such as half a beat). Just a reminder that `beat_length` is how many seconds per beat.

2. Go on `www.ultimateguitar.com` or an equivalent website and look up the chords of a song you know. Hint: if you are running out of time, just Google Image Search ”twinkle twinkle little star chords” and you will be able to see the chords to ”twinkle twinkle little star”, which is very few, so you can implement them on time. You can just implement the first verse of a song if you are pressed for time!

3. Choose a part of the song (a verse, the chorus, or more if you are feeling ambitious) and create a *cell array* where the first column contains the names of **chords** from your section of the song, in order, and the second column contains the corresponding **number of beats** to play each chord. It is up to you to decide what counts as a single beat. It doesn't matter what you pick, as long as a chord lasting twice the number of beats is played for twice as long in the real song, etc. You may need to listen to the song in order to enter the numbers of beats relatively accurately. If any of your chords is not listed in the switch-case environment inside `get_chord_wave`, you will need to add a case for it.

   Tip: If you are musically-trained, you can recognize that at the beginning of a music score, there is a ”time signature”, which is in the form of a fraction (usually). The numerator is how many beats per bar, and the denominator is what counts as a beat. For example, if a song is 4/4, then it means that there are 4 beats per bar and a quarter-note (filled note

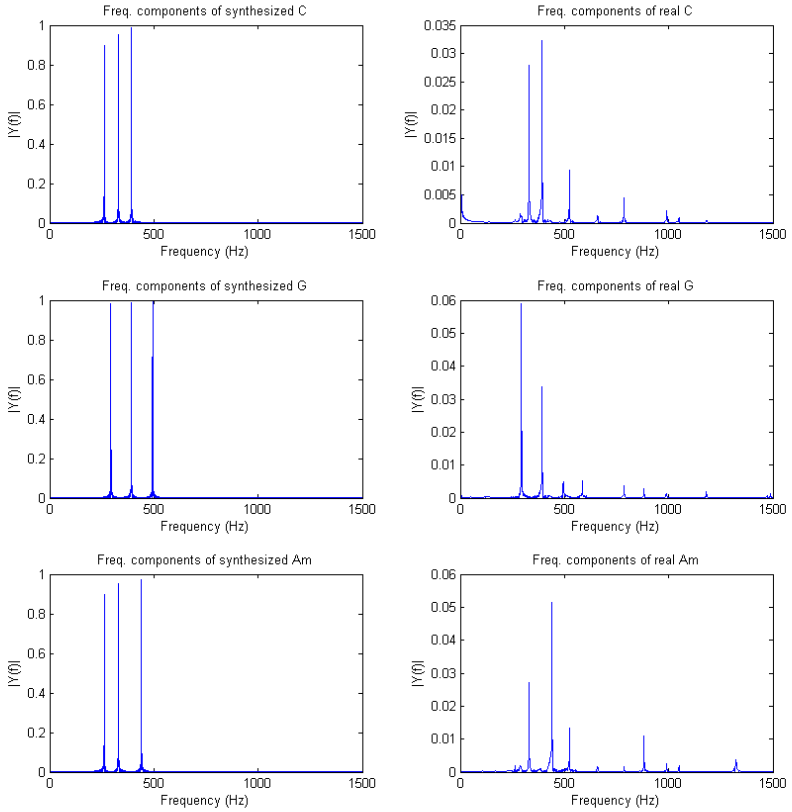


Figure 2. Matlab-synthesized chords vs. real chords played on the piano (frequency spectrum)

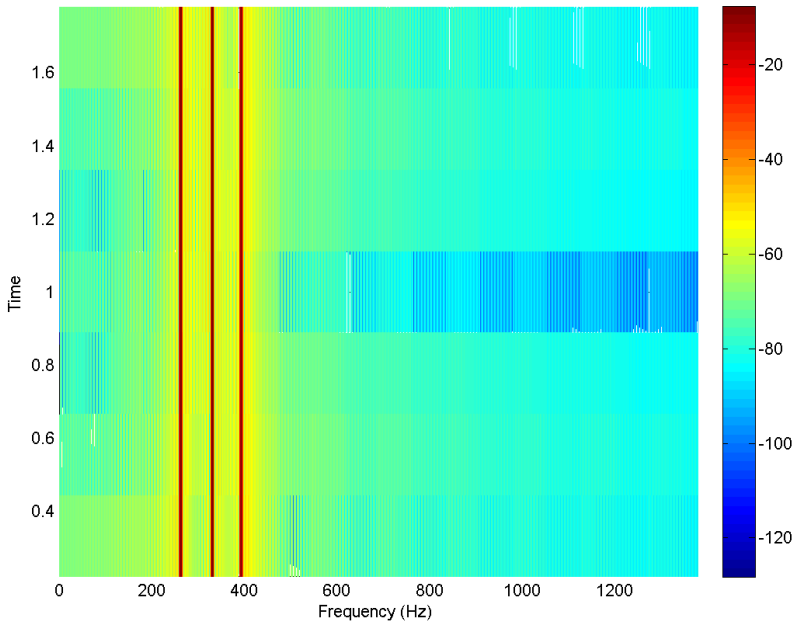


Figure 3. Matlab-synthesized chords vs. real chords played on the piano (frequency spectrum)

with straight stem) counts as a beat (i.e. beat = 1). In this case, a half note (hollow note with straight stem) would have beat = 2, and an eighth-note would have beat = 0.5, etc.

4. Figure out an approximate *beat duration* for your song, by listening to the song and relating that to the values in your cell array. Then get the time signal for your song using `get_song_wave`, and listen to it. Play it for your TA.

5. Listen to your song, and see how many beats it takes for the strum pattern (rhythm) to repeat. This will be your `num_beats`. Then, using the function `ADSR_env`, produce a strummed version of your song. (Use the same beat duration as you used for `get_song_wave`.) Listen to the result.

6. Now go inside the `ADSR_env` function and change the strum pattern, ADSR lengths, and peak and sustain volumes to sound more like the original song. If you like, you may also edit the `get_chord_wave` function to introduce slight delays so the chord is formed sequentially instead of each note starting at the same time. (This step is crucial in making your song actually sound something like the original, but if you're pressed for time, don't spend too much time on this.)

7. Take a spectrogram of the final signal. Use appropriate window lengths (the second argument of `spectrogram`) to see consecutive chords separately. The window duration is the number of samples that gets transformed at once. Set `noverlap = 0` (the third argument of `spectrogram`). Explain what you're seeing in the spectrogram to your TA.

8. If you want to save your song in a `.wav` file, use the `audiowrite` command which is built in to `MATLAB`.

Appendix A: Provided code and data

Functions

- `f = get_freq(num)` – outputs the frequency of a note in Hz, given the key number
- `X = get_wave(note, oct, duration, fs)` – given a note (`'A'`, `'Gb'`, `'Fs'`, etc.) and an octave designation high (`'h'`), middle (`'m'`), or low (`'l'`), a duration of a time in seconds, and a sampling frequency, returns a vector of values, `X`, to represent the time signal corresponding to that note
- `num = get_num(note, oct)` – given a note (`'A'`, `'Gb'`, `'Fs'`, etc.) and an octave designation high (`'h'`), middle (`'m'`), or low (`'l'`), returns the corresponding key number by looking in `key_data.mat`
- `X = get_chord_wave(chord, duration, fs)` – given a chord name (`'Am'`, `'C'`, `'G'`, etc.), a duration of time, and a sampling frequency, this function generates the corresponding time signal
- `Z = get_song_wave(song, beat_duration, fs)` – given a song (notated in a cell array in terms of chords and beats), a beat duration in seconds, and a sampling frequency, `Z` is the time signal of the song to be used in playback
- `Y = ADSR_env(X, beat_duration, num_beats, fs)` – given a signal `X`, the beat duration in seconds, the number of beats to be covered by a single repetition of the strum pattern, and the sampling frequency of the original signal, this function outputs `Y`, a strummed version of `X`

Files

- `key_data.mat` – needed for the `get_num` function, contains array of notes and their key numbers

Appendix B: Example of an incomplete function

```
function [ f ] = get_freq( num )
%get_freq produces the frequency of a note in Hz given the key number

middle_A_freq = 440; %Hz
middle_A_num = 22; %this is the key number of middle A

octave_mult = 2; %going up an octave doubles the frequency
num_octave_keys = 12; %there are 12 keys in an octave

%fill in the following lines in terms of num and the constants defined
%above
semitone_mult =  %frequency ratio of a semitone (two neighbouring keys)
semitone_jump =   %how many semitones away from middle A our key is
f =  %our key's frequency
```

```
end
```

Acknowledgments

Thanks to the students who have provided input on the previous versions of this experiment.