# ECE 72: Introduction to Electrical and Computer Engineering Tools

## **Project 3: Music Synthesis** Due: 12/5/17 by 2:00 PM

You are to write MATLAB script that will play a song of your choosing with correct timing, and then plays the notes. You will also create a script that allows a user to modify the song you created.

You will also prepare a document discussing possibilities for the computer-generated synthesis of music.

In order to complete this project, you will need a computer with both MATLAB and sound. If you use a computer in the ECE department, you may need a set of headphones.

Each student will submit his/her own work. You may discuss this project with other students. But your submissions must be the fruit of your own labor.

**Music Synthesis**

As discussed in class, MATLAB can be used to synthesize music, by creating a sinusoid that can be used as an output to a speaker. In MATLAB, sound waves can be represented as a collection of amplitude samples of that sinusoid, and an indication of the sampling rate. There is a standard, called Musical Instrument Digital Interface (MIDI), and a MIDI note number, which represents a specific pitch. Using the symbol $p$ to represent the pitch number, the frequency of this note is given by:

$$440\times2^{(p-69)/12}\text{ Hz}$$

A couple of examples might be helpful. When $p = 69$, the frequency is 440 Hz. When $p = 57$, the frequency is 220 Hz.

In musical terms, each of the pitch numbers represents a key on the piano keyboard. This scheme accounts for both black and white keys. The pitch number increases as your finger moves from left to right on the keyboard. Middle C has been assigned the pitch number 60. When you move your finger on the keyboard to the right by 12 keys, the frequency doubles. For example, your finger moves 12 keys to the right in shifting from the key associated with pitch number 57 to the key associated with pitch number 69, and the corresponding frequencies are 220 Hz and 440 Hz, respectively. A set of 12 consecutive keys (including both black and white keys) is called an *octave*, corresponding to a doubling of the frequency.

More information, including a table of notes and frequencies can be found on Wikipedia: https://en.wikipedia.org/wiki/Scientific_pitch_notation

Music is made both with a specific pitch, but also with a specific duration. When reading sheet music, different note durations are specified with special symbols, as shown on this Wikipedia page: https://en.wikipedia.org/wiki/Note_value .

## Instructions

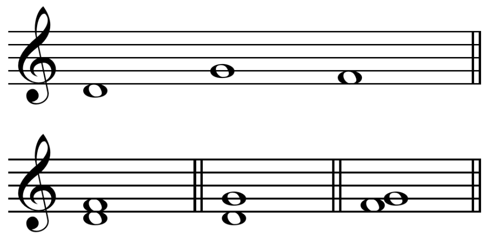
Your project will be completed in two parts. Firstly, you will synthesize some choice of music in MATLAB, and in the second, you will write a script to perform modifications to your synthesized music, based on the input given by a user.

### Part 1

You can choose any song you like, or write your own, and create a vector for MATLAB to play. You can find sheet music online and use the above information to translate it into pitches, find specific songs written with pitch or notes, or use year ear to find correct note values. You will also need to find specific durations of each note. You should use the same standard of 44.1 kHz sampling rate used by CDs.

There are a few requirements that your song must satisfy:

- Your song must have more than one location of multiple notes playing at the same time, as most songs do, with harmonics and melodies. Finding piano sheet music will be helpful with this as usually both hands (and fingers) play different notes at the same time. An example of multiple notes in sheet music is shown below:



- Your song must be longer than a few notes, though you don't have to code in a whole song (unless you want to). At minimum, I want around 5-10 seconds of music.
- It must be a song, not just a random collection of notes.
- You must have a unique song, I don't want any repeated songs in the class. To select your song, go to the Blackboard Discussion forum, and post your song

(don't claim a song that has already been posted). You may also not use Johann Sebastian Bach's *Fugue #2 for the Well-Tempered Clavier* which has been used in an ECE 72 project in the past.

At minimum, you will write a script m-file and a function m-file, you can write more functions if you think it would be useful. The function(s) you write will be called from the script. The script will also define a variable representing the sampling frequency and assign this variable the value 44100 (hertz or, equivalently, samples per second).

Your first function will produce a signal vector representing one note. There will be three input arguments to this function. The first of these is the pitch number *p*. The next argument is the duration in seconds. The final argument is the sampling frequency in hertz. This function will not need the start time because the signal vector produced by this function will later be shifted to start at the correct time. (That shifting will occur in the script.) There will be two outputs from this function: one signal vector and one scalar. The scalar is the number of samples in the signal vector.

A single pure note having frequency **f** (Hz) and duration **T** (seconds) could be synthesized like this:

```
t = 0:Ts:T;

x = cos(2*pi*f*t);
```

In the above sample code, **Ts** is the sample period (the time between adjacent sample values) in seconds. The sample period equals the reciprocal of the sampling frequency. (Don't confuse the sampling frequency with the frequency **f** of the note.) The frequency **f** is related to the pitch number as explained in the section *Music Synthesis*. If you were to synthesize the notes as shown above, however, there would be audible clicks in your synthesized music. These clicks are due to the sudden arrival and departure of the notes.

You should eliminate the clicks by having each note fade in and fade out. You can accomplish this by multiplying each pure note (the **x** defined above) by a fading mask. The mask is a vector having the same length as **x**. The first **k** samples of the mask vector should increase linearly from **0** to **1**. The last **k** samples of the mask should decrease linearly from **1** to **0**. All other samples in the mask should equal **1**. You should use at least **k = 1000**; this is a suitable choice for the sampling frequency of 44.1 kHz that you will be using in this project. In other words, with **k = 1000**, the fade-in of each note will last about 23 ms as will the fade-out of each note. The duration of the note will be the time from the beginning of the fade-in to the end of the fade-out. So, the mask for each note will equal **1** for a period of time that is less than the duration of the note by about 46 ms. Since different notes will, in general, have different durations, you can decide to make the fade different for different notes. (However, the duration of every note should be greater than 50 ms, so the fade-out should never overlap the fade-in.)

The vector returned by your function will represent one masked note. As mentioned above, the function will also return a scalar equal to the number of samples in the output vector.

Your script will call your function once for each note of each voice. Your script will shift the signal for each note so that it starts at the proper time. The script will then compose a signal vector that incorporates all notes of all three voices. We will call this the composite-signal vector.

In the interest of efficiency, you should pre-allocate memory for the signal vector that will be passed to **sound**. As usual, this pre-allocation will be accomplished with the function **zeros**. But first, it will be necessary to determine how long this signal vector must be, which will be based on how much music you plan on playing (note, this is likely not just the summation of the notes you will be playing, as notes can (and should) overlap, and there may be rests where no note is played.) Be aware that your calculations will, by default, employ (double-precision) floating-pointing numbers. The length that you input to **zeros** cannot have a fractional part. You can use the function **uint32** to round a floating-point number to the nearest unsigned integer. (You need **uint32** because **uint8** and **uint16** likely can't represent a large enough number for this application.)

In the script, the masked signal vector for each note must be added to the composite-signal vector (that has been pre-allocated and that, after normalization, will eventually be passed to **sound**). For example, the masked signal vector **w** of a note can be added in to the long signal vector **y** like this:

```
y(a:b) = y(a:b) + w;
```

where **a** is the index of **y** corresponding to the start time of the note and **b** is the index corresponding to the end time of the note. These index values **a** and **b** may be computed from the start time of the note, the sampling frequency (44100 Hz) and the number of samples for the note's signal vector. Make sure you round-off your calculated index values using **uint32**. (If you use a floating-point number with a nonzero fractional part as an index in an array, MATLAB will complain.) You may decide it is easier for you to write your vector using something else learned in class, and I encourage you to code in your own style, you don't need to use the example above if you don't want to. Note that a requirement is that you have multiple notes playing at a single time at multiple parts of your song, in those cases you will be adding the signal values over each other.

It is essential to scale properly the composite-signal vector. The function **sound**, which will be used to play the composite-signal vector, and the function **audiowrite**, which will be used to record the composite-signal vector, both expect that all sample values will be between −1 and +1. Any sample value that does not meet this criterion will be clipped. (A sample value of −1.27 will, for example, be clipped to −1, and a sample value of +2.41 will be clipped to +1.) As you might imagine, clipping will distort your music. You should recognize that even if each component note meets this criterion, the composite-signal vector, obtained by summing notes together, may not. You can achieve the desired normalization by first finding the maximum of the absolute value of the (composite-signal) samples and then dividing all samples by this maximum. It is recommended that you additionally multiply the result by 0.9; this ensures that the new, normalized composite-signal vector does not even touch the limits of −1 and +1.

After the normalized, composite-signal vector has been constructed, the script will use **sound** in order to play the music. You should use three input arguments to **sound**: the (normalized, composite-signal) vector, a scalar equal to the sampling frequency, and a scalar equal to 16. (This last input is the number of bits to be used per sample when playing the vector. If you don't specify 16 bits, **sound** will use 8 bits by default.)

When the script is executed, the command **sound** should be executed only once. In this single invocation of **sound**, the script should send the normalized, composite-signal vector, which includes every note of all voices. If your script invokes **sound** more than once, your music won't sound right. With multiple invocations of **sound**, computational delay between invocations will cause the timing of the notes to be wrong.

Finally, you will record the normalized, composite-signal vector in a WAVE (waveform audio format) file (***.wav**) using the **audiowrite** function. (A WAVE file does not employ compression and is much less memory efficient than MP3; but a WAVE file is acceptable for small sound snippets.) Use **audiowrite** with three inputs: a string for the filename, the (normalized, composite-signal) vector, and the sampling frequency. In the interest of conserving memory, **audiowrite** saves each sample using just 16 bits, rather than with the full 64 bits (that is, 8 bytes) of a **double**. After saving the signal vector to a WAVE file, use the function **audioinfo** to obtain some basic facts about the saved WAVE file. Then print out the sample rate and the total (number of) samples for the WAVE file (with information provided by the function **audioinfo**). A WAVE file can be played even outside of MATLAB.

**Part 2**

The second part of the project is to use the **input** function to modify the sound vector you made in Part 1 based on user input. Your script should perform an infinite loop until some "exit" input is given. You should clearly print to the screen what options the user has to modify the script, including the method to exit. This could be a menu with numbers, keywords, etc. Exactly how and what the user inputs for these different options is up to you. For all of these modification options, you are not allowed to recreate your sound vector, you must modify your vector from Part 1, and you are not allowed to modify the 44.1 kHz sampling rate. After the user gives an input, your script should perform the appropriate modification, and then play the new sound vector with the **sound** function. All subsequent modifications should be made to the same waveform, so for example, if I perform "fade in" and "fade out" it should gradually get louder until halfway through the song, then gradually get softer until the end. When the user exits, you should write the modified sound to a WAVE file, similar to what you did in Part 1.

The modifications/actions you need to input are as follows:

- **Fade in** – you should modify the entire vector to start from a volume of 0 to a volume of 1 by the end of the song.
- **Fade out** - you should modify the entire vector to start from a volume of 1 to a volume of 0 by the end of the song.
- **Pitch up** – you should modify the entire sound vector to be twice as high (the frequency should be close to doubled for each note). Remember you can't simply modify the sampling rate, you must modify your vector. It is ok if your song goes twice as fast after your modification.
    - **Extra credit**: if you are able to get the pitch to be higher, just by modifying the vector (again you can't recreate it), while the length of every note is the same, I will give some extra credit points.
- **Pitch down** – you should modify the entire sound vector to be twice as low (the frequency should be close to halved for each note). Remember you can't simply modify the sampling rate, you must modify your vector. It is ok if your song goes twice as slow after your modification.
    - **Extra credit**: if you are able to get the pitch to be lower, just by modifying the vector (again you can't recreate it), while the length of every note is the same, I will give some extra credit points.

- **Plot** – plot the entire current sound vector to Figure 1. Noting is modified with this action.
- **Normalize** – do a similar operation to what you did in Part 1 to normalize the sound vector so that the maximum amplitude of your vector is 1.
- **Exit** – save a WAVE file with your modified sound, as a different name than your file from Part 1, and then exit your infinite loop.

As long as the above requirements given are met, you are free to be as artistic as you want. You can add crescendos, decrescendos, vibrato, or any other interesting musical modification you like.

**Publishing Your Script**

As explained above, you will create one script m-file and at least one function m-file for this project. In the script, you will include a **type** command for each custom function that you created for this project. Each **type** command will be of the form:

```
type functionname
```

where *functionname* is the name of a function.

You will publish your script to a ***.pdf** document. The **type** command in the script causes the code in the named function to be printed to the published ***.pdf** document.

**What You Will Submit**

You will publish your script to a ***.pdf** document. Make sure that the code for your custom function or functions appear along with the code for your script in your published ***.pdf** document. You will submit to Blackboard your published ***.pdf** document, and all of your ***.m** files used in your project.

You will also submit to Blackboard an essay (preferably submitted as a ***.pdf** document) in which you tell the meaningfulness of the song you used to you, as well as possibilities for computer-generated synthesis of music. Your document should be approximately one page long.

You will *not* submit any WAVE files.

Checklist:

1. Published ***.pdf** document showing the code in your script and function m-files,

2. All your ***.m** script and function files that you created, and

4. Essay (as a ***.pdf** file) containing the story of your collage.

**Grade**

Your grade will reflect primarily the technical merit of your MATLAB solution. However, the artistic merit (including, for example, your discussion of music synthesis) will also be taken into account. The grader will be checking that you followed all instructions.