ENGR 151

Fall 2023

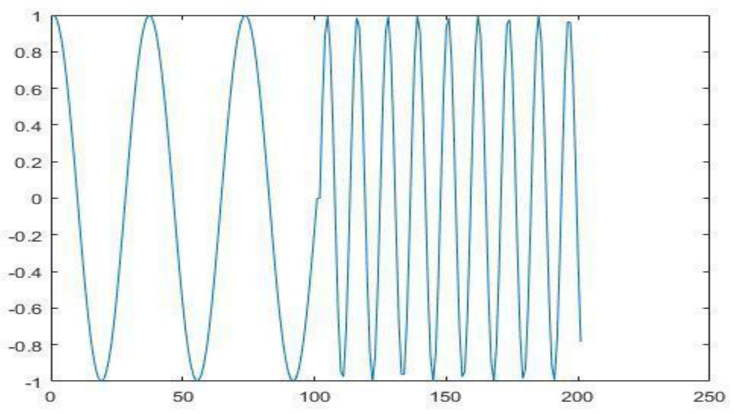
Lab 3: Flow Control in MATLAB

Music Animation (Selection, Iteration, and Functions)

In these exercises, you get to play around with sound! Talk to your GSI if you do not have a speaker to use. Successive exercises pick up on where the previous one left off, so approach these problems in order.

Sound Basics

First, let's go over some basics. Sound is vibrating air. These vibrations occur at different frequencies. When the frequency is very regular, we hear this as music. Music notes can be modeled by sine waves with certain amplitudes and frequencies. The amplitude is how loud the sound is and the frequency changes the pitch of the note. A low note has a low frequency and a high note has a high frequency. For example, the graph below shows a low note being played followed by a higher note.



In music, certain frequencies are given names. For example: a frequency of 440 Hz is the note 'A' (in octave 4). More on this later.

Exercise 1: First Musical steps

MATLAB has a function called `sound()`, which will play a note if given a sine wave as an argument. Let's start by playing one note.

To create a sine wave, we will generate many points using the **`sin()`** function. **For this lab, you must use a resolution of 1/8192**. (8192 is referred to as the "sampling frequency".)

For example, you can use the code:

```
duration = 1;                 %play note for 1 second
frequency = 440;              %In Hz; play the note "A"
samp_freq = 1/8192;           %Sampling frequency
time_arr = 0:samp_freq:duration; %create 1-sec worth of points
wave = sin(2*pi*frequency*time_arr);
```

Now if you call `sound(wave)` it will play the note A for 1 second!

sound(wave) must be executed in Matlab

(Will not work in Canvas Matlab Environment)

Make a function called **play_note** that generates a musical note for 1 second, plays it on your speaker, and plots the wave! Make sure to use the `sound()` function to play the note! **Comment out or remove the sound() function when submitting to the canvas autograder.**

Inputs: the **frequency** of the note. (`freq` - it must be a number!)
Output: the output **wave** (`wave`)
Function must plot the wave and return the wave as output.

Exercise 2: Write a note generator!

Let's start to write your first song! First, let's create a function that generates the sounds for a given input array of notes (eg: `["A", "G"]`). Now, we also want to be able to influence how long the pauses are between these notes.

You can use the **pause()** function to space your notes, otherwise they will play at the same time. For example, a script with :

```
sound(wave1);
pause(duration);
sound(wave2);
```

will play a note for one second, pause for `duration` seconds, and play the second note for 1 second.

Use the table below to store the relationships between the note and its corresponding frequency. If you're musically inclined or want to explore some more, see Appendix A for information on how to calculate all note frequencies :)

Note	**Frequency**
C	261.63
D	293.66
E	329.63
F	349.23
G	392.00
A	440.00

Write a function **'play_notes'** that takes in an array of the notes as its argument, and plays those notes in succession. Use a loop to go through each note and its corresponding pause!

Inputs: A **string** (not char) array of notes (`notes`), and a **double** array of pause durations (`pauses`)
Sample: notes = ["G", "C", "D", "E"], pauses = [0.5, 0.1, 1, 0.7]
Note: Enclose your note within " ", NOT ' '!
Outputs: A wave with all the generated notes joined one after another. (`wave`) (Make sure to add the blank pause durations in the wave as well!)

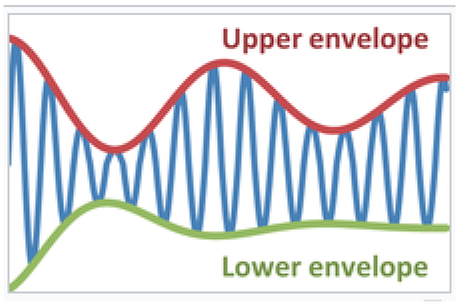
To test your function, write an array of notes like `notes =["A", "G", "C", "D"]` and an array of pause durations like `pauses = [0.5, 1, 0, 0.2]` and run it with a command like `out_wave = play_notes(notes, pauses)`.
Comment out or remove the sound() function when submitting to the canvas autograder.

Exercise 3: Smoothen note transitions (Optional)

Envelope Function!

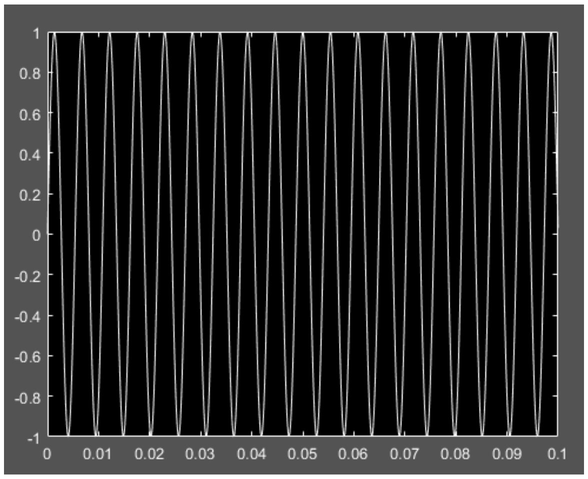
You may notice an unmusical clicking at the beginning and end of each note. This is caused by the abrupt start of the sine wave (a discontinuity). If you plot the wave in exercise 2 you can see the jump where the frequencies change. You will write a function to alleviate this effect.

According to Wikipedia, "in physics and engineering, the envelope of an oscillating signal is a smooth curve outlining its extremes" as shown in the figure below:

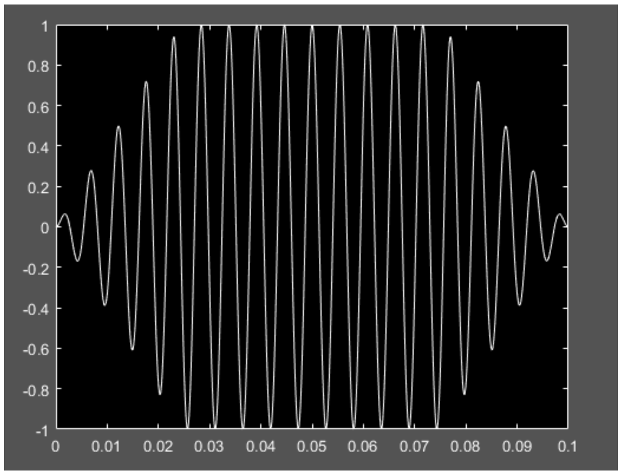


Currently, you are probably making the wave a sine wave with a constant amplitude such as: `a*sin(freq*2*pi*time)` where the value of `a` may very well be 1. This results in a wave like the one in the left plot below.

If you decrease the amplitudes zero at the beginning and end of the note, the discontinuity will be faded out. The amplitude can start at zero, increase to a reasonable volume, and decrease to zero again at the end. This can be achieved by multiplying the signal by an envelope array that scales the amplitude. Note the `sound()` function expects input with a max amplitude of -1 to 1.



Before envelope



After envelope

You can write a function named `envelope` to perform this operation.

<u>Suggested Function structure:</u>

Input : Input wave (**`in_wave`**)

Output: Output wave (**`out_wave`**) which is an array of the input signal multiplied by an envelope. The figures above are an example input/output pair of how your function should function.

To check your function works, call your function (i.e. `tapered_wave = envelope(orig_wave)`). The resulting figure should look ***similar*** to the above figure on the right. **Don't worry about it being the exact same !!**

Now call your envelope function on each note before it plays. There is not a specific amount of attenuating (decreasing the amplitude) that needs to happen, but you should hear an improved quality of music.

Files to turn in

This lab assignment is due before the beginning of your next lab. Please turn in following files on Canvas under Lab 3 assignment:

- L3 : Exercise 1: play_note.m
- L3 : Exercise 2: play_notes.m

Appendix A

The notes of Western music are A, A#, B, C, C#, D, D#, E, F, F#, G and G#. Each note frequency is related to an adjacent note frequency by a factor of $2^{1/12}$ which is ~1.06.

You read in the lab that A4 (concert A, used for tuning) has a frequency of 440 Hz. Thus the next note, A#4 would be $440*2^{1/12}$, about 466.16 Hz.

Since there are 12 notes in an octave, the A in the next octave, A5, is exactly double the frequency of A4 - 880 Hz! Super cool, right? Now you can think about why certain intervals between notes go well together depending on their ratios.

Knowing this, you can define the frequency of notes relative to A4.

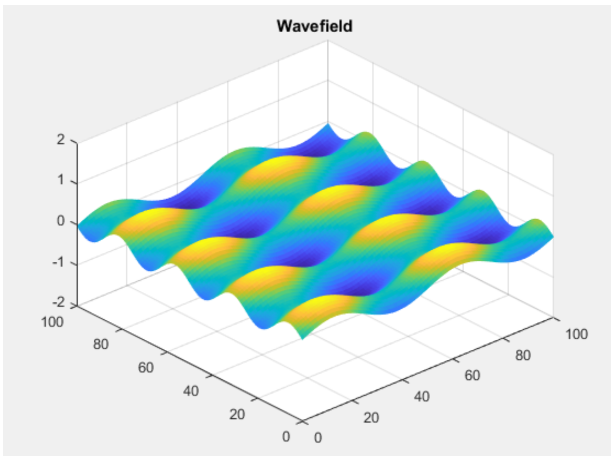
$$\texttt{frequency} = 2*\texttt{pi}*440*(2^{1/12})^{\texttt{step}}$$

where “step” is how many notes above or below A the note is. If playing a C4, step would be 3. If playing a G3, step would be -2.

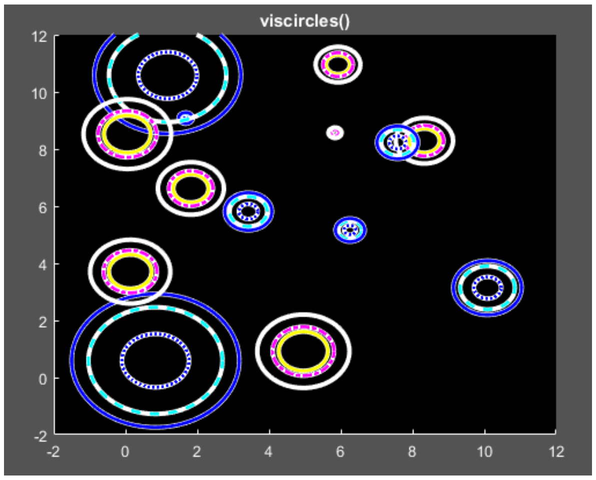
Appendix B - Plot Function Options

Browse the plot gallery: https://www.mathworks.com/products/matlab/plot-gallery.html
Below are pictures of a couple samples

meshgrid() and **surf()**

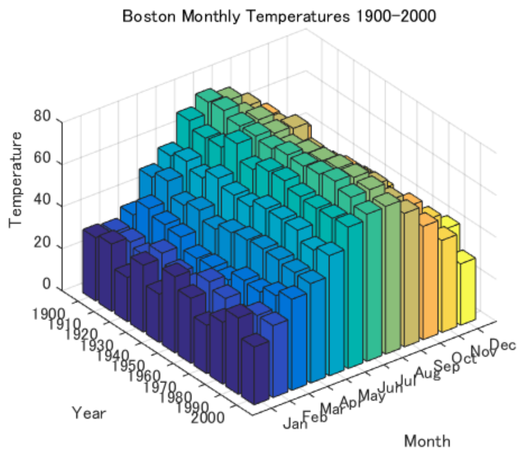


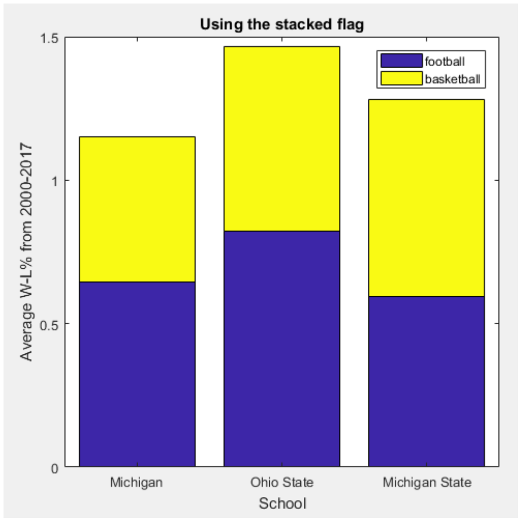
viscircles()



bar3() function

bar() with the **‘stacked’** flag


Boston Monthly Temperatures 1900–2000
Temperature
0
20
40
60
80
Year
1900
1910
1920
1930
1940
1950
1960
1970
1980
1990
2000
Month
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec


Using the stacked flag
Average W-L% from 2000-2017
0
0.5
1
1.5
football
basketball
Michigan
Ohio State
Michigan State
School