TECHIN 513 – Lab 1
Review on Python and Signals

Lab Instructions

1. Please approach the course instructor or grader for assistance if you need help.
2. We highly recommend working in groups of **2 members**. When collaborating with students in other groups, please do not share code but only discuss the relevant concepts and implementation methods.
3. Please document your code well by using appropriate comments, variable names, spacing, indentation, docstring, etc. Please refer to TECHIN 509 for more information.
4. The starter code is not binding on you. Feel free to modify it as you wish. Everything is fine so long as you are getting the correct results.
5. Please upload the .ipynb file to canvas. Use Markdown cells appropriately to answer the questions. Each team only needs to **submit one report**.
6. Please enter the names of all the team members in your Jupyter notebook so that you do not lose credit for your work.

Lab Submission Requirements

In your notebook, briefly describe your results and discuss the problems you encountered, the solutions that you came up with and any choices you made regarding to the length of rests, details of the envelope, etc. Also, answer the questions asked in each section. Make sure to include the code of all the tasks in code cells.

Lab Objectives

The purpose of this lab is to get yourself familiarized with constructing and operating on discrete-time signals in Python. You will also gain some understanding of the physical meaning of the signals by using audio playback. Specifically, you will

1. You will learn how to **synthesize** music notes and **play** them in Python environment.
2. You will **concatenate** a series of music notes into a small music piece.
3. You will perform signal transformation, e.g., add **volume variation** to the music piece.
4. You will **overlap** the adjacent notes to further make the music smoother and more realistic.

Task 1: Generating Musical Notes

Background

For this task, we will review how to use simple sinusoidal waves to realize musical notes and how to bring them together to compose a segment of music. At the end of this task, you will construct the first few measures of the song Scarborough Fair.

There are seven natural notes: A, B, C, D, E, F and G. After G, we begin again with A. Music is written on a "staff" consisting of five lines with four spaces between the lines. The notes on the staff are written in alphabetical order. The first line is E as shown in Figure 1. Notes can extend above and below the staff. When they do, ledger lines are added.

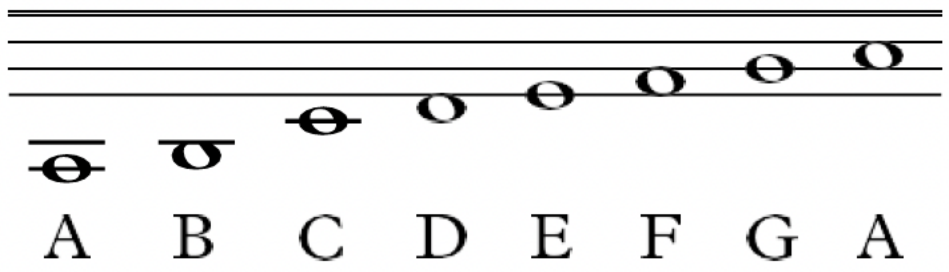

Figure 1 Natural notes

Musical notes are arranged in groups of twelve notes called octaves. The notes that we'll be using for Scarborough Fair are in the octave containing frequencies from 220 Hz to 440 Hz. The twelve notes in each octave are logarithmically spaced in frequency, with each note being of a frequency $2^{1/12}$ times the frequency of the note of lower frequency. Thus, a 1-octave pitch shift corresponds to a doubling of the frequencies of the notes in the original octave.

Table 1 shows the ordering of notes in the octave to be used to synthesize the opening of Scarborough Fair, as well as the fundamental frequencies for these notes.

Note	Frequency
A	220
A♯, B♭	$220 \times 2^{1/12}$
B	$220 \times 2^{2/12}$
C	$220 \times 2^{3/12}$
C♯, D♭	$220 \times 2^{4/12}$
D	$220 \times 2^{5/12}$
D♯, E♭	$220 \times 2^{6/12}$
E	$220 \times 2^{7/12}$
F	$220 \times 2^{8/12}$
F♯, G♭	$220 \times 2^{9/12}$
G	$220 \times 2^{10/12}$
G♯, A♭	$220 \times 2^{11/12}$
A	440

Table 1: Notes in the 220-440 Hz Octave

A musical score is essentially a plot of frequencies (notes) on the vertical scale versus time (measures) on the horizontal scale. The musical sequence of notes for the piece you will synthesize is given in Figure 2. The following discussion identifies how musical scores can be mapped to tones of specific pitch and duration.

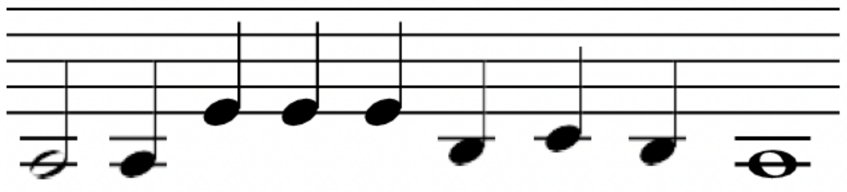

Figure 2: Musical score for Scarborough Fair

In the simplest case, each note may be represented by a series of samples of a sinusoid followed by a shorter period of silence (samples of zeros, which are a pause). The pauses allow us to distinguish between separate notes of the same pitch. The duration of each note burst is determined by whether the note is a whole note, a half note, or a quarter note (see Figure 3). In this Lab, use a duration of **4,000 samples** for 1 count.

Therefore, your whole notes should be four times the duration of your quarter notes. The short pause you use to follow each note should be of the same duration regardless of the length of the note. Longer periods of silence that are part of the musical score are indicated by one or more rest symbols. There are no rest symbols in the score you are given.

Note that A – G only yield seven notes; the additional notes are due to changes in pitch called sharps (denoted by the symbol ♯) or flats (denoted by the symbol ♭) that follows a given note. A sharp increases the pitch by $2^{1/12}$ and a flat decreases it by $2^{1/12}$. There are no sharp or flat symbols in the music you are going to create.

In the musical score in Figure 2, the first half note and quarter note are both A. The next three quarter notes are all E and so on. You can get the fundamental frequencies for these notes from Table 1.

Python Notes

You can play audio samples using python using display.Audio() function of the “IPython” package.

```python
import IPython.display as ipd
ipd.Audio(audio_array, rate=sampling_rate, autoplay=True)
```

Notes

- audio_array is the name of the variable that stores your generated audio samples and the second argument sampling_rate is the sample rate (**8 kHz**).
- The Audio Object must be returned when used in a function.

Assignment 1 (6 points)

Write a Python program that generates the **audio samples** of the provided musical score (Scarborough Fair) using a **sampling rate** of **8 kHz.** As part of the task, fill in the relevant function definitions in the provided template (play() and note()). Use the play function to listen to your generated sample.

Instructions and Helpful Hints

- **Generating a Pure Tone.** In this Lab, a musical note is a **sinusoid** of a certain **frequency**, a certain **duration**, and sampled at a certain **sampling rate**. To create the sinusoid in Python, you need to **choose values for the three parameters**. You can then use those parameters to generate an ndarray that you can use as input to the `np.sin` function. Once you have created the sinusoid, you can play it using the `play` function shown above in the Assignment section.
- **Generating a Pure Tone.** Visualize the sinusoid waves for A, B, C, and E using **subplot**. Based on the given frequency, calculate their periods. Compare your calculations with the plots and verify them.
- **Concatenating Notes and Rests.** Now that we know how to create a note, the next step is to concatenate a series of notes, separated by short rests. Recall that `np.concatenate([a, b, c])` joins the arrays a, b, and c together. The rest between notes is just some silence, that you can generate with the `np.zeros` function (You may also find the function `np.pad` useful). Experiment with the amount of rest to find a duration that sounds good to you.
- **Efficient Implementation.** Consider how you can efficiently implement the code required to play the entire sequence of notes. For example, it may be effective to use a for-loop, as in the following (incomplete) example.:

```
rest = ...            # define the rest between notes here
song = np.array([])   # start with an empty song
for i in range(N):
  thisNote = note(frequency[i], duration[i], samplingRate)
  song = ...          # concatenate this note and a rest to the song
```

Note that, in this case you need to store the frequency and duration of each note in one (or two) array(s). You can also consider using Python's list comprehension syntax:

```
score = [(freq1,dur1),(freq2,dur2),...] #define frequencies and durations
notes = [note(f,d) for (f,d) in score] # create list of notes
song = np.concatenate(notes) # concatenate
```

How would you incorporate the results using this syntax?

Task 2: Time Scaling and Time Shifting Audio Signals

Assignment 2 (4 points)

In this task, you will explore how time scaling and time shifting operations will impact audio signals. Load the train.wav file and store it as train_audio. Complete the following tasks on time scaling:

- Use the time_scale function to create a DT signal w[n] = train_audio[2n] and a DT signal v[n] = train_audio[n/2]. Store the output from time_scale function.
- Create a time-reversed version of train_audio, z[n] = w[-n].
- Save each resulting signal in a wav file.
- Use subplot to visualize train_audio, w, v, and z. Adjust the axis limits to have the same ranges. Clearly label your plots.

Complete the following tasks for time shifting.

- Read the provided function time_shift. This function takes a signal x[n] and time shift b as input, and returns a signal y[n] = x[n+b]. Since x[n] is of finite length, you can assume that the signal value is zero outside the time window.
- Test the function using train_audio with positive, negative, and zero b values. Visualize the resulting signals.

Task 3: Volume Variations: Amplitude Operations

Background

There are many ways of improving the perceived quality of a synthesized sound. In this task you will learn about one such method: varying the note **volume** over time.

Typically, when a note is played, the volume rises quickly from zero and then decays over time, depending on how hard the key is struck and how long it is depressed. The variation of the volume over time is divided into four segments: **Attack, Decay, Sustain, and Release (ADSR)**. For a given note, volume changes can be achieved by **multiplying** the sinusoid by another function (**range of [0, 1]**) called an **envelope function**. An example of such function is shown in Figure 4.

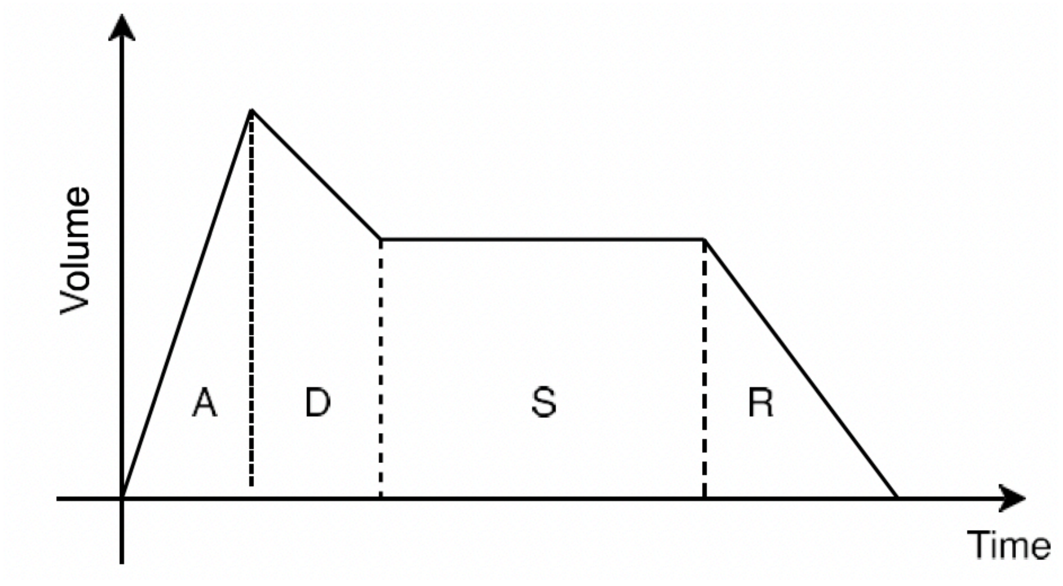

Figure 4: An ADSR Envelope

Assignment 3 (6 points)

Improve the quality of the sound you produced in Task 1 by applying an ADSR envelope to each note.

- Do not directly modify your code for Task 1. Instead, copy it over and modify the copy so that you have both your original code and the improved code in your notebook.
- Experiment with different durations and heights of the different segments. Find a combination that sounds good to you.
- In addition to listening to your sample with the `play()` function, inspect the envelope using the `display_envelope()` function included in the template. Confirm that it looks as expected and include the plot in your submitted report Make sure to properly label the x- and y-axis of your plot.
- Instead of enveloping with the ADSR function, how would you envelope with a decaying exponential? Write a `display_exp_envelope()` function to apply the decaying exponential envelope. Describe what would be the consequence of applying a decaying exponential as the envelope. Plot the original note with the enveloped one. Clearly label your graph. You can verify your answer by playing both notes.

Helpful Hints

- You do not need to copy the functions or constants (e.g., the frequency/duration array) you have defined in Task 1 if you do not plan to modify them.
- In Python, the envelope function should consist of a Numpy array which will be multiplied (element by element) by the sample in Task 1.
- You may find the `arange` or `linspace` functions useful for generating the ramping regions of the envelope.

Task 4: Overlapping Notes

Another improvement in perceived quality can be achieved by **overlapping** some notes as done by advanced piano players. As the volume of one note is decaying, another note is played. Mathematically, this can be accomplished by allowing the time regions occupied by subsequent sinusoids to overlap. This will yield a much smoother, less staccato-sounding piece.

Assignment 4 (4 points)

Improve the quality of the sound you produced in Task 2 by allowing the end of one note and the beginning of the next note to overlap slightly in time. Again, work on a copy instead of directly modifying your original Task 2 code. Keep in mind that the samples of your final song should not exceed the ±1.0 range. If you are not sure, you can double check your waveform by plotting it. Consider whether you should still use concatenation of notes to accomplish the desired blending effect. If so, how? If not, why not? How would you do it instead?

In addition to listening to your sample with the `play()` function, inspect the envelope using the `display_envelope()` function included in the template. Confirm that it looks as expected and include the plot in your submitted report.

Helpful Hints

- You may find the function to `np.pad` be helpful.