Lab 0: Getting to some basic sound processing

Below you will find a series of tasks that will help you get started with audio processing using Python. We will start with learning how to generate test sounds, how to read/write audio files, how to perform simple waveform editing and finally how to use real time audio I/O. Look at the provided hints and try to figure out how to do these tasks on your own. If you get stuck, talk to me or our helpers for more hints.

You can get the necessary files from this link: http://courses.engr.illinois.edu/cs498ps3/lab0.zip

Exercise 1. Generating and playing basic sounds

It is important to be able to generate multiple types of test sounds to test various parts of an audio processing chain. Some of the most important ones are sinusoids, chirps, and certain types of noise. For this exercise you need to generate the following test signals, at a sampling rate of 8kHz and for a duration of a second. Plot them if you like and see if they look right. You will probably have to zoom into the plots to check these waveforms. Also, play these sounds from your computer's speakers (IMPORTANT: Turn your computer's volume down first, some of these sounds might be very loud!)

1. White noise (use a random number generator)
2. A sinusoidal tone of frequency 440 Hz
3. A linear chirp from 0 to 4kHz (do not use existing chirp functions)
4. A sinusoidal tone with an exponentially decreasing amplitude from 100 to 0.0001

You will now make a stereo file. This is represented as a 2d array, one part containing the left channel and the other containing the right channel. For the left channel generate a quarter-second sinusoidal tone of frequency 523.24Hz with an exponentially decaying amplitude from 100 to 0.0001. For the right channel do the same thing but use a frequency of 784Hz. Start the right channel tone after a quarter second. Play this and verify that it sounds ok (it should sound like a video game "ping-pong" sound).

Load the provided file `handel.dat`. It contains a sound waveform encoded as a series of 16-bit values. Find out what it's sample rate is (there's no trick here, this one is trial and error).

To play a sound inline when using Jupyter you can use:

```
IPython.display.display( IPython.display.Audio( sound_array, rate=sample_rate))
```

Useful `numpy` *functions:* `random.randn`, `sin`, `linspace`, `logspace`, `fromfile`

Exercise 2. Saving sounds

What good are sounds if we can't store them? For most of this class we will be using what is known as a PCM format (more on Tuesday). The most popular of these formats is the WAVE file, which we will use most often. When saving a sound to a file we need to be careful and make sure we don't lose any information.

Take the "ping-pong" sound from above and save it to a WAVE file. Play the file back, or open it with an audio editor and find out if there's anything wrong. If so, find a way to fix it.

Useful python package: `scipy.io.wavfile`

Exercise 3. Basic sound editing

In this exercise we will learn to do some simple manipulations of sounds. Ordinarily you would do this with an audio editor with a graphical interface, but hey life sucks and you have to do this with code.

1. Load the file `1234.wav` and listen to it. Clearly something is wrong. Try to fix the problem using code.
2. Use the above file to create a countdown instead.
3. Load the file `uneven.wav`. There's something wrong here too. Fix it!
4. Load the two files `m1.wav` and `m2.wav`. They are roughly at the same tempo and you want to make a music mix out of them. Play the first sound for two seconds, then fade it out over four seconds. While the first sound fades out the second one should fade in at the same speed. Congrats, you just learned how to (poorly) DJ in python!

Useful python commands: `scipy.io.wavfile.wavread`, `numpy.hstack`

Exercise 4. Real-time processing

In real-life you can't just load an existing soundfile, process at your leisure and save it. You have to be able to process sound in real-time. This means that you will record tiny snippets of sound, quickly process each one and then move to the rest without looking back again. In this exercise we will try a couple of real-time things to get the hang of it. Interpreted languages is generally horrible for real-time systems, but we'll stick with them since it's much simpler than writing low-level code. If you use python you can use the package `pysoundcard` to get some low-level audio control. Open an audio stream with a sample rate go 16kHz and a single channel. Use a buffer size of 1024 samples.

Now we will create a loop in which we get a snippet of sound from the microphone at each pass. Inside the loop you will `read` from the stream (which should be taking samples off the microphone). Using this, measure the standard deviation of each incoming sound snippet of sound and after eight seconds of recording plot these as a sequence.

Now let's try to add some output as well. We will make a robot voice effect that makes use of a *ring modulator*. This is the same effect that's been used to generate robot voices for many older films and TV shows (e.g. the Daleks in Dr. Who).

We will reuse the loop that we made above, but this time we will additionally have an audio output. Do the same as above, but this time you can also `write` to the stream to send a buffer of samples to the speaker. For a test you can simply pass the input buffer from read to write, and this would simply play from the speakers the sounds you make to the microphone (*tip #1*: wear headphones to avoid a feedback loop! *tip #2*: Every time you put on headphones set the volume to a very low value to avoid any painful surprises). Once you verify that a passthrough works, multiply each input snippet with a a 440Hz sine and send that to the output to create a voice transformation. If successful, it should sound robotic. Congrats, you just made your first audio effect!

Useful python commands: `pysoundcard.Stream.read`, `pysoundcard.Stream.write`