Lab 0: Audio Processing

The questions below are due on Friday September 04, 2020; 04:00:00 PM.

Table of Contents

Welcome to 6.009! As this is our first lab, part of the time will be spent helping you familiarize yourself with the structure of a 6.009 lab and with the associated infrastructure. As such, compared to a normal lab, this lab is a bit smaller in terms of scope and scale.

1) Infrastructure

1.1) Necessary Software

This lab assumes you have Python 3.6 or later installed on your machine. If you don't already have Python installed, or if you wish to upgrade, please follow our instructions for accessing/installing Python on your operating system of choice before proceeding.

You will also need a program for editing your Python source files. Depending on where you learned Python and how far along you are, you may or may not already have a preferred text editor (or Integrated Development Environment) for working with Python code. If you already have something you are familiar with, feel free to use that! If not, we recommend using the "IDLE" program (which is included with most Python installations) for editing your files.

1.2) The Command Line

Throughout 6.009, our instructions for labs will often refer to using your computer's *command line* (or "*shell*" or "*terminal*") to run programs. We don't expect you to be familiar with using the command line, and we'll try our best to explain exactly what needs to be done when you do need to use the command line. However, you may find it helpful to familiarize yourself with some command line basics before continuing on.

Although learning to use the command line is well worth the effort in the long run, it can be daunting when you're first getting started, so don't worry if things don't come naturally at first. Of course, feel free to ask at office hours (or via the forum) if you need help!

1.3) Running Tests

For all labs, you will also need the `pytest` module installed to run our test cases. Certain labs may have additional requirements, but for today, `pytest` is all we'll need. You can install `pytest` from the command line using a command like the following:

```
$ pip3 install pytest
```

(depending on your setup, you may need to type `pip` instead of `pip3`; or you may need to include `sudo` at the front, i.e. `sudo pip3 install pytest`)

After having done so, if you make a small Python file containing only the line `import pytest`, then evaluating that file with Python should work without error.

For all of our labs, we will include a file called `test.py`; when you run this file (either from the command line or from some other environment), it runs your code through several test cases and shows you the results.

Running `pytest test.py` will run all of the test cases in the file, but you can modify the behavior (including running only a subset of the test cases) as described in this section (about `pytest`) in the notes about the command line.

Note that some of the smaller tests might be useful not only for testing, but also for help with understanding the specification for a given piece of code.

1.4) Submitting Code

As mentioned above, we will distribute a suite of tests with each lab, and you can use this file to test your code for correctness as often as you like.

Once your code passes these test cases on your machine, you should submit your code to the server to be checked. After submitting your code, you will receive feedback on the results of the tests on our server. Your grade for the lab will depend on how many test cases your code passes on the server.

1.5) Checkoff

In order to receive credit for a lab, you must also complete the associated "checkoff," which is a brief conversation with a staff member about your code. We will ask some questions about your code and also provide some feedback on style. The checkoff can also be a good opportunity to learn new Python tricks and alternative ways of aproaching the lab!

Note that your checkoff will be based on the most recent code you have submitted, so if you make stylistic changes, etc., it is worth resubmitting your updated code so that we can discuss it during the checkoff.

Note that, per the lateness policy, submitting style changes after the deadline does not incur any lateness penalty, so long as all previous test cases continue to pass.

2) Preparation

As mentioned above, this lab assumes that you have Python 3.6 (or newer) installed on your machine, as well as `pytest`.

The following file contains code and other resources as a starting point for this lab: `lab0.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, **you should not add any imports to the file**.

You can also see and participate in online discussion about this lab in the "Lab 0" Category in the forum.

This lab is worth a total of 4 points. Your score for the lab is based on:

- answering the questions on this page (1 point),
- passing the tests in `test.py` (2 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (1 point).

All of the questions on this page, including your code submission, are due at 4 p.m. Eastern Time on Friday, 4 September. Checkoffs are due at 10 p.m. Eastern Time on Wednesday, 9 September.

3) Lab Introduction

In this lab, we will be manipulating audio files to produce some neat effects. This week's distribution contains not only a template file that you should use for developing your code (`lab.py`) but also several audio files in the `sounds` directory, with names ending with `.wav` (you can try opening these files in an audio program to hear their contents).

Over the course of this lab, we will refresh ourselves on some important features and structures within Python, and we will also familiarize ourselves with interactions with files on disk (and create some really neat sound effects as well).

4) Representing Sound

In physics, when we talk about a sound, we are talking about waves of air pressure. When a sound is generated, a sound wave consisting of alternating areas of relatively high pressure ("compressions") and relatively low air pressure ("rarefactions") moves through the air.

When we use a microphone to capture a sound digitally, we do so by making periodic measurements of an electrical signal proportional to this air pressure. Each individual measurement (often called a "sample") corresponds to the air pressure at a single moment in time; by taking repeated measurements at a constant rate (the "sampling rate," usually measured in terms of the number of samples captured per second), these measurements together form a representation of the sound by approximating how the air pressure was changing over time.

When a speaker plays back that sound, it does so by converting these measurements back into waves of alternating air pressure (by moving a diaphragm proportionally to those captured measurements). In order to faithfully represent a sound, we need to know two things: both the sampling rate and the samples that were actually captured.

For sounds recorded in *mono*, each sample is a positive or negative number corresponding to the air pressure at a point in time. For sounds recorded in *stereo* (which we will use for this assignment), each sample can be thought of as consisting of two values: one for the left speaker and one for the right.

4.1) Python Representation

We will be working with files stored in the WAV format. However, you won't need to understand that format in detail, as we have provided some "helper functions" in `lab.py` to load the information from those files into a Pythonic format, as well as to take sounds in that Pythonic representation and save them as WAV files.

Our Pythonic representation of a sound will consist of a dictionary with three key/value pairs:

- `'rate'`: the sampling rate (as an `int`), in units of samples per second
- `'left'`: a list containing samples for the left speaker, where each sample is a `float`
- `'right'`: a similar list containing samples for the right speaker

For example, the following is a valid sound:

```
s = {
    'rate': 8000,
    'left' [0.00, 0.59, 0.95, 0.95, 0.59, 0.00, -0.59, -0.95, -0.95, -0.59],
    'right': [1.00, 0.91, 0.67, 0.31, -0.10, -0.50, -0.81, -0.98, -0.98, -0.81],
}
```

5) Manipulations

In this lab, we will examine the effects of various kinds of manipulations of audio represented in this form.

5.1) Backwards Audio

We'll implement our first manipulation via a function called `backwards`. This function should take a sound (in our representation, as a dictionary) as its input, and it should return a *new* sound that is the reversed version of the original (but without modifying the object representing the original sound!). For example, the reverse of the following sound:

0:00 / 0:04

is this:

0:00 / 0:04

Reversing real-world sounds can create some neat effects. For example, consider the following sound (a crash cymbal):

0:00 / 0:02

When reversed, it sounds like this:

0:00 / 0:02

When we talk about reversing a sound in this way, we are really just talking about reversing the order of its samples (in both the left and right channels) but keeping the sampling rate the same.

Go ahead and implement the `backwards` function in your `lab.py` file. After doing so, your code should pass the first four test cases in `test.py`.

It can also be fun to play around with these things a little bit. For example, `mystery.wav` is a recording of Adam H speaking nonsense. Let's try using our new function to produce a modified version of that file.

Note that we have provided some example code in the `if __name__ == '__main__'` section of the file, which demonstrates how to use the `load_wav` and `write_wav` functions. This is a good place to put code for generating files, or other quick tests.

Try using some similar code to create a reversed version of `mystery.wav` by: loading `mystery.wav`, calling `backwards` on it to produce a new sound, and saving that sound with a different filename (ending with `.wav`). If you listen to that new file, you might be able to interpret the secret message! 🤔🙂

Once you have that file, upload it in the box below to be checked for correctness:

Reversed `mystery.wav`:

Select File No file selected

Download Most Recent Submission

Submit

You have submitted this assignment 1 time.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

5.2) Mixing Audio

Next, we'll look at *mixing* two sounds together to create a new sound. We'll implement this behavior as a function called `mix`. `mix` should take three inputs: two sounds (in our dictionary representation) and a "mixing parameter" p (a `float` such that $0 \leq p \leq 1$).

The resulting sound should take p times the samples in the first sound and $1 - p$ times the samples in the second sound, and add them them together to produce a new sound.

The two input sounds should have the same sampling rate. If you are provided with sounds of two different sampling rates, you should return `None` instead of returning a sound.

However, despite having the same sampling rate, the input sounds might have different durations. The length of the resulting sound should be the *minimum* of the lengths of the two input sounds, so that we are guaranteed a result where we can always hear both sounds (it would be jarring if one of the sounds cut off in the middle).

For example, consider the following two sounds:

0:00 / 0:06

0:00 / 0:01

Mixing them together with a mixing parameter $p = 0.7$, we hear the sound of a frustrated cat whose human is paying too much attention to a guitar and not enough to the cat....

0:00 / 0:01

Go ahead and implement `mix` in `lab.py`. As with the function above, `mix` should produce a new Python object representing the new sound, and it should not modify either of the sounds that are passed to it. After having implemented `mix`, you should pass the first 8 test cases in `test.py`.

As one example of a neat result, try mixing together `synth.wav` and `water.wav` with a mixing parameter of $p = 0.2$. Give this one a listen, and you should hear a sound mimicking what you would hear listening to some weird new-age music while standing next to a stream....

Once you have that file, upload it below to be checked for correctness.

Mixed synth and water:

Select File `No file selected`

Download Most Recent Submission

Submit

You have submitted this assignment 2 times.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

5.3) Echo

Next, we'll implement a classic effect: the *echo* filter. We simulate an echo by starting with our original sound, and adding one or more additional copies of the sound, each delayed by some amount and scaled down so as to be quieter.

We will implement this filter as a function called `echo(sound, num_echos, delay, scale)`. This function should take the following arguments:

- `sound`: a dictionary representing the original sound
- `num_echos`: the number of additional copies of the sound to add
- `delay`: the amount (in **seconds**) by which each "echo" should be delayed
- `scale`: the amount by which each echo's samples should be scaled

A good first place to start is by determining how many *samples* each copy should be delayed by. To make sure your results are consistent with our checker, you should use Python's `round` function: `sample_delay = round(delay * sound['rate'])`

We should add in a delayed and scaled-down copy of the sound's samples (scaled by the given `scale` value and offset by `sample_delay` samples). Note that each new copy should be scaled down more than the one preceding it (the first should be multiplied by `scale`, the second by a total of `scale**2`, the third by a total of `scale**3`, and so on).

All told, the output should be `num_echos * sample_delay` samples longer than the input in order to avoid cutting off any of the echos.

As an example, consider the following sound:

0:00 / 0:02

If we invoke `echo` with this sound, 4 copies, a 0.4-second delay, and a scaling factor of 0.4, we end up with the following:

0:00 / 0:04

Consider the following sound:

```
s = {
    'rate': 8,
    'left': [1, 2, 3, 4, 5],
    'right': [5, 4, 3, 2, 1],
}
```

If we were to make a new sound via `s2 = echo(s, 1, 0.4, 0.2)`, what should the value of `s2['left']` be?
Enter a Python list in the box below:

[1, 2, 3, 4.2, 5.4, 0.6000000000000001, 0.8, 1.0]

Submit

You have submitted this assignment 1 time.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

Consider the following sound:

```
s = {
    'rate': 8,
    'left': [1, 2, 3, 4, 5],
    'right': [5, 4, 3, 2, 1],
}
```

If we were to make a new sound via `s2 = echo(s, 2, 0.4, 0.2)`, what should the value of `s2['left']` be?
Enter a Python list in the box below:

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

Implement the echo filter by filling in the definition of the `echo` function in `lab.py`. Note that `echo` should create a new sound and should not modify its inputs.

When you have done so, try applying your echo filter to the sound in `chord.wav`, with 5 echos, 0.3 seconds of delay between echos, and a scaling factor of 0.6. Save the result as a WAV file, give it a listen, and upload it in the box below to check for correctness:

Echo-y `chord.wav`:

Select File No file selected

Download Most Recent Submission

Submit

You have submitted this assignment 1 time.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

5.4) Pan

So far, we have not really taken advantage of the fact that we are working with stereo sounds (for each of the effects above, we applied the same results to the left and right channels). We'll change that in this section, using stereo sound to create a really neat spatial effect. **Note** that this effect is most noticeable if you are wearing headphones; it may not be too apparent through laptop speakers.

Let's hear an example before we describe the process. If you listen to the sound below (a creaky door), it should sound like the door is more or less directly in front of you:

0:00 / 0:03

However, if we manipulate things a bit, we can make it seem like we are moving by the door. Or maybe the door is moving by us? 😕 Either way, the sound seems to start off to our left and end up on our right!

0:00 / 0:03

We achieve this effect by adjusting the volume in the left and right channels separately, so that the left channel starts out at full volume and ends at 0 volume (and *vice versa* for the right channel).

In particular, if our sound is N samples long, then:

- We scale the first sample in the right channel by 0, the second by $\frac{1}{N-1}$, the third by $\frac{2}{N-1}$, ... and the last by 1.
- At the same time, we scale the first sample in the left channel by 1, the second by $1 - \frac{1}{N-1}$, the third by $1 - \frac{2}{N-1}$, ... and the last by 0.

Go ahead and implement this as a function `pan` in your `lab.py` file. As with the functions above, this function should not modify its input; rather, it should make a brand-new object representing the new sound. After implementing `pan`, your code should pass the first 16 test cases in `test.py`.

Once you have done so, we'll once again test by applying this function to a piece of audio. Try applying this to `car.wav`, then save the result and listen to it. This should be more impressive (or at least more realistic) than the door example....

When you have that file, upload it below to check for correctness:

Left-to-right `car.wav`:

Select File No file selected

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

5.5) Removing Vocals from Music

As a final example for this lab (unless you are interested to try some of the optional additional pieces discussed below!) is a little trick for (kind of) removing vocals from a piece of music, creating a version of the song that would be appropriate as a backing track for karaoke night.

Our approach is going to seem weird at first, but we will explain why it works (at least on some pieces of music) in a bit. For now, we'll describe our algorithm. For each sample in the input, we compute `(left-right)`, i.e., the difference between the left and right channels at that point in time, and use that value as **both** the left and right values in the output at that time.

That might seem like a weird approach to take, but we can hear that the results are pretty good. For example, here is a short sample from a song that was popular before you were born ("Lido Shuffle" by Boz Scaggs):

0:00 / 0:25

And here is the result after applying the algorithm above:

0:00 / 0:25

Although some of the instruments are a little bit distorted, and some trace of the vocal track remains, this approach did a pretty good job of removing the vocals while preserving most everything else.

So how does it work? Indeed, it seems weird that subtracting the left and right channels should remove vocals... and why does it only work on certain songs? Well, it comes down to a little bit of a trick of the way songs tend to be recorded. Typically, many instruments are recorded so they they favor one side of the stereo track over the other (for example, the guitar track might be slightly off to one side, the bass slightly off to the other, and various drums at various "positions" as well). By contrast, vocals are often recorded *mono* and played equally in both channels. When we subtract the two, we are removing everything that is the same in both channels, which often includes the main vocal track (and often not much else). However, there are certainly exceptions to this rule; and, beyond differences in recording technique, certain vocal effects like reverb tend to introduce differences between the two channels that make this technique less effective.

Anyway, now would be a good time to go ahead and implement this manipulation by filling in the definition of the `remove_vocals` function in `lab.py`. As with all of the previous filters, this function should not modify its input; rather, it should produce a new sound. After implementing `remove_vocals`, your code should pass all of the test cases in `test.py`!

Try applying `remove_vocals` to the sound from `coffee.wav`. This one doesn't do quite as well as the example above, but you can hopefully still notice that a substantial portion of the main vocal track has indeed gone away.

Save the result as a WAV file and upload it below to be checked for correctness:

`coffee.wav` after removing vocals:

Select File `No file selected`

Download Most Recent Submission

Submit

You have submitted this assignment 1 time.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

6) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` by clicking on "Choose File", then clicking the `Submit` button to send the file to the 6.009 server. The server will run the tests and report back the results below.

Select File `No file selected`

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 04, 2020 at 04:00:00 PM Eastern time.

7) Checkoff

Once you are finished with the code, you will need to come to any office-hour time and add yourself to the queue asking for a checkoff in order to receive credit for the lab. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions; see the notes on style for more information). In particular, be prepared to discuss:

- Your code for `backwards` and `mix`.
- Your code for `echo`.
- Your code for `pan` and `remove_vocals`.
- Your additional code for loading and saving the WAV files from above.

7.1) Grade

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

8) (Optional) Additional Extensions

If you have found this lab interesting, you might be interested in trying some additional things (and if you do, please feel free to share during the checkoff!):

- Try manipulating your own sounds! You can use a tool like Audacity to record sounds of your own (or to clip and convert music files)! Make sure that you save your files as WAV files. In Audacity, the right option to choose is "WAV (Microsoft) signed 16-bit PCM".

- Make a variant of the echo filter that takes advantage of stereo sound by playing subsequent echos in alternate speakers (first echo on the left, second on the right, and so on) rather than playing them equally in both channels.
- Make a variant of `mix` that takes arbitrarily many sounds (and associated mixing parameters) and mixing them *all* together.
- Try making a function that speeds up or slows down a given sound (either by manipulating the samples themselves, or the sampling rate).