

Lab 0: Audio Processing

You are not logged in.

If you are a current student, please [Log In](#) for full access to the web site.

Note that this link will take you to an external site (<https://shimmer.mit.edu>) to authenticate, and then you will be redirected back to this page.

Table of Contents

- 1) Infrastructure
 - 1.1) The Command Line
 - 1.2) Necessary Software
 - 1.3) Running Tests
 - 1.4) Submitting Code
 - 1.5) Checkoff
- 2) Preparation
- 3) Lab Introduction
- 4) Representing Sound
 - 4.1) Python Representation
- 5) Manipulations
 - 5.1) Backwards Audio
 - 5.2) Mixing Audio
 - 5.3) Convolutional Filters
 - 5.4) Echo
- 6) Stereo Effects
 - 6.1) Pan
 - 6.2) Removing Vocals from Music
- 7) Code Submission
- 8) Checkoff
- 9) What Comes Next?
- 10) (Optional) Additional Extensions

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. For that reason, and because of a slightly-compressed timetable compared to a normal lab, this lab is a bit smaller in terms of scope and scale than a typical lab.

1) Infrastructure

1.1) 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!

1.2) Necessary Software

Completing and submitting 6.009 assignments will require you to have a few pieces of software installed on your machine. Please follow our instructions for [getting set up for 6.009](#) on your operating system of choice before proceeding.

1.3) Running Tests

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 uses `pytest` to run 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, using the `submit-009-lab` script. More detailed instructions for submitting your code are available further down this page.

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. At the checkoff, 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 approaching the lab!

Note that checkoffs will become available only after the associated lab has come due. The checkoffs themselves generally come due at 10pm on the Wednesday after the code submission came due.

Once you are ready (and the checkoff is available), come to any open lab time and use the [help queue](#) to request a checkoff.

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. 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

This lab assumes that you have Python 3.6 or newer (3.10 recommended) installed on your machine, as well as `pytest`.

The following file contains code and other resources as a starting point for this lab: [lab00.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 ["Labs" 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 5pm on Friday, 4 February. Checkoffs are due at 10 p.m. on Wednesday, 9 February.

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*, 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 *mono* sound (which we will use for the bulk of this lab) will consist of a dictionary with two key/value pairs:

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

For example, the following is a valid sound:

```
s = {
    'rate': 8000,
    'samples': [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 mono sound (using the representation described above, as a dictionary) as its input, and it should return a *new* mono sound that is the reversed version of the original (but without modifying the object representing the original sound!).

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 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

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 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...

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

5.3) Convolutional Filters

It turns out that a wide variety of interesting effects can be applied to audio using an operation known as *convolution*. By applying this operation to the samples in a sound and a "kernel" (another list of samples), we can achieve a wide variety of effects.

In its most concise mathematical form, convolution is usually expressed in the following way:

$$y[n] = (x * h)[n] = \sum_{m=-\infty}^{\infty} x[m]h[n-m]$$

That is to say, the value $y[n]$ at sample number n in the output (y) can be computed using the sum on the right (where x represents the samples in the input sound and h represents the samples in the kernel).

This is a fine way to implement convolution (and you are welcome to do things this way if you please), but it is perhaps easier to think of convolution in a slightly different way. Our kernel is represented by a list of values. For each nonzero value $h[n]$ in our kernel, we create a copy of the samples in the input sound, shifted by n and scaled by $h[n]$; and adding these new lists of signals together give us our output. The total length of the output should be the length of the samples in the input sound, plus the length of the samples in the kernel, minus one.

For example, consider the following sound:

```
s = {
  'rate': 20,
  'samples': [2, 5, 0, 3]
}
```

and consider convolving this signal with the following kernel:

```
kernel = [3, -2]
```

Here, we'll walk through the process of computing the result of `convolve(s, kernel)`, using the process described above. We know that our output is eventually going to have 5 samples in it (the length of the samples, plus the length of the kernel, minus 1).

The 3 at index 0 contributes values to our output samples like a version of the input samples scaled by 3 and shifted by 0: [6, 15, 0, 9, 0]

The `-2` at index `1` contributes values to our output samples like a version of the input samples scaled by `-2` and shifted by `1`: `[0, -4, -10, 0, -6]`.

Adding those together, we get our overall result: `[6, 11, -10, 9, -6]`. These represent the samples of our output sound (which should have the same rate as the input).

Consider the following sound:

```
s = {
    'rate': 30,
    'samples': [-5, 0, 2, 3, 4],
}
```

And the following kernel:

```
k = [0, 1, 0, 0, 3]
```

If we were to make a new sound via `s2 = convolve(s, k)`, what should the value of `s2['samples']` be?

Now implement the `convolve` function in `lab.py`. After it is correctly implemented, your code should pass the first 14 test cases from `test.py`.

Convolution might seem like a weird operation to want to perform, but it turns out to be central to a lot of common and interesting audio manipulations. One example of a kind of effect that can be achieved via convolution is a *bass-boost* effect, where low frequency content in a sound is amplified.

We have provided a function called `bass_boost_kernel(N, scale)`. Calling that function will produce a kernel that can be fed into your `convolve` function (along with a sound) to produce a new sound as output. The `scale` parameter affects the loudness of the low frequencies (`scale=0` means no boosting at all, and larger values boost the low-frequency content more); the `N` parameter provides a way to control the frequencies that are boosted (`N=0` will boost all frequencies roughly equally, and larger values allow more focus on the lowest frequencies in the input sound).

To test this functionality, use your `convolve` function to make a bass-boosted version of the `ice_and_chilli.wav` sound. When creating the kernel, use `N=1000` and `scale=1.5`. Note that this operation may take quite a long while compared to the other files we've generated so far; feel free to grab a cup of tea while it's running 🍵🍰. It might take a couple of minutes, depending on the computer you're running on.

Listen to the result, and compare it against the original `ice_and_chilli.wav`. Can you hear the difference? Note that the difference may be difficult to hear on laptop speakers; but it should be apparent if you use headphones or external speakers.

Upload the resulting file below to test for correctness:

Bass-boosted `ice_and_chilli.wav`:

Select File

No file selected

5.4) Echo

Next, we'll implement a classic effect: an *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_echoes, delay, scale)`. This function should take the following arguments:

- `sound`: a dictionary representing the original sound
- `num_echoes`: 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_echoes * sample_delay` samples longer than the input in order to avoid cutting off any of the echoes.

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,  
    'samples': [1, 2, 3, 4, 5],  
}
```

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

Consider the following sound:

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

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

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.

Note also that there are multiple ways to implement this function. One strategy would involve writing a brand-new `echo` function from scratch; but another would be to construct an appropriate filter and then using `convolve` to implement the echo.

After implementing `echo`, your code should pass the first 18 test cases from `test.py`.

When you have done so, try applying your echo filter to the sound in `chord.wav`, with 5 echoes, 0.3 seconds of delay between echoes, 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

6) Stereo Effects

For the last two audio effects in this lab, we'll focus instead on *stereo* sounds (files that have separate lists of samples for the left and right speakers).

Our Pythonic representation of a *stereo* 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
- `'right'`: a list containing samples for the right speaker

For example, the following is a valid stereo 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],
}
```

Our `load_wav` function can read stereo files to create sounds of this form as well, as long as you provide the `stereo=True` argument to the function. For example, `load_wav('hello.wav')` produces a mono sound, but `load_wav('hello.wav', stereo=True)` produces a stereo sound instead.

6.1) Pan

For our first effect using stereo sound, we'll 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 22 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

6.2) 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. This effect will take a stereo sound as input, but it will produce a mono sound as output.

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 (stereo) input sound, we compute `(left-right)`, i.e., the difference between the left and right channels at that point in time, and use the result as the corresponding sample in the (mono) output sound.

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.

Indeed, it seems weird that subtracting the left and right channels should remove vocals! So how does this work? 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 that 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 in `lookout_mountain.wav`. If you listen to the result, how did this method perform on this example?

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

lookout_mountain.wav after removing vocals:

Select File

No file selected

7) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `submit-009-lab` script. If you haven't already installed it, see the instructions on [this page](#).

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ submit-009-lab -a lab00 /path/to/lab.py
```

Running that script should submit your file to be checked, and it should also provide some information about how and where to get feedback about your submission. Reloading this page after submitting will also show some additional information:

You have not yet made any submissions to this assignment.

8) Checkoff

Once you are finished with the code, you will need to come (in person) to any open lab 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 `convolve` and `echo`.
- Your code for `pan` and `remove_vocals`.
- Your additional code for loading and saving the WAV files from above.

You have not yet received this checkoff.

9) What Comes Next?

If you enjoyed the material in this lab, you might be interested to take 6.003 (Signal Processing) at some point. It covers some of the ideas from this lab in far more detail, so if you're curious about how filters like the bass-boost filter can be designed and understood, that may be a good class to take!

You may also be interested, farther down the line, to take a class like 21M.387 (Fundamentals of Music Processing), which applies these and other techniques to processing of music in particular.

10) (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 versions of your mono effects that apply those same effects to stereo sounds (applying the same effect to the left and right channels separately). Can you implement this without re-writing the core algorithms for each of the effects?
- Make a variant of the echo filter that takes advantage of stereo sound by playing subsequent echoes 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).