Lab 10

Filtering and Convolution

Lab Objective: *The Fourier transform reveals things about an audio signal that are not immediately apparent from the soundwave. In this lab we learn to filter noise out of a signal using the discrete Fourier transform, and explore the effect of convolution on sound files.*

Cleaning up a Noisy Signal

Digital audio signals can be used to produce actual sound waves. When you play a digital audio signal on your computer, the signal is sent to a speaker, which vibrates, producing sound waves. When more than one speaker is used, they can both produce the same signal, or each could produce a different signal. When there is only one signal, we say that the sound is *monoaural*, or simply *mono*. When speakers produce different signals, we say that the overall signal is *stereophonic*, or *stereo*. Usually stereo means two, but there may be any number of signals (5.1 surround sound, for instance, has 5).

Listen to `Noisysignal1.wav`. This is a mono recording of a (probably familiar) voice with some annoying noise over it. The plot of the soundwave isn't very descriptive; in fact, it looks like static. See Figure 10.1a.

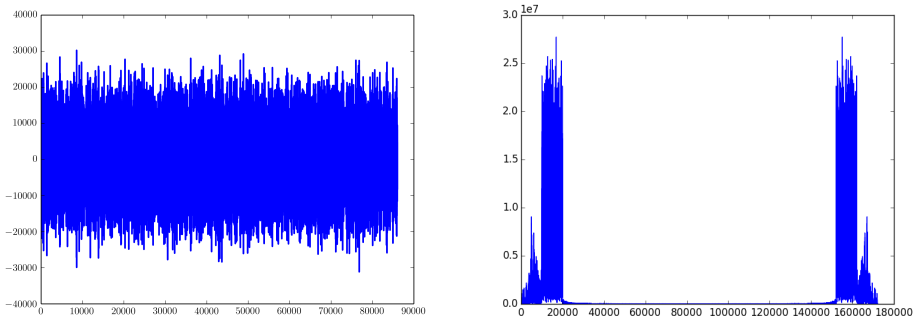
However, if we take the Fourier transform of the signal, we see that the static in Figure 10.1a is the result of some concentrated high-frequency noise. (In this case, artificially added). See Figure 10.1b.

The noise can be removed by setting the coefficients of the high frequencies to zero. Since the discrete Fourier transform is symmetric, if we set coeffecient j to 0, then we must set coefficient $N-j$ to 0 as well, where N is the number of coefficients. Then we calculate the inverse Fourier transform to get a new, clean signal.

```
>>> rate,data = wavfile.read('Noisysignal1.wav')

# Calculate the Fourier transform
>>> fsig = sp.fft(data, axis = 0)

# Coefficients 10000 to 20000 were chosen by inspecting the
# plot of the Fourier transform.
```



(a) The plot of `Noisysignal1.wav`. (b) Spectrum of `Noisysignal1.wav`

Figure 10.1

```
>>> for j in xrange(10000, 20000):
...     # Set the chosen coefficients to 0
...     fsig[j] = 0
...     fsig[-j] = 0

# Calculate the inverse Fourier transform, cast it as real,
# and scale it to be compatible with the wavfile format.
>>> newsig = sp.ifft(fsig)
>>> newsig = sp.real(newsig)
>>> newsig = sp.int16(newsig / sp.absolute(newsig).max() * 32767)
```

Now we can save the resulting cleaned-up signal `newsig` to a `.wav` file. The plot of the wave now reveals individual syllables as they are spoken. See Figure 10.2.

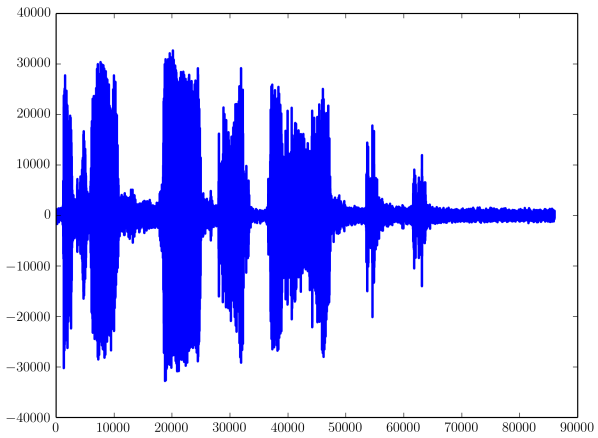


Figure 10.2: The plot of `Noisysignal1.wav` after being cleaned.

Problem 1. Listen to `Noisysignal2.wav`. You will probably just hear noise. Inspect the discrete Fourier transform to see where there is noise. Remove the noise using the technique described above in order to make the cleaned-up signal audible. What does the voice say? Who is the speaker? (If you don't know the answer to this last question, try a quick Google search.)

The DFT is commonly used in sound filtering, though identifying the particular frequencies to zero out can be difficult.

Filtering and Convolution

The DFT is useful for more than filtering noise out of a signal. Suppose we have a recording of a musical piece played in a small, carpeted room with essentially no acoustics (little or no echo), and suppose we would like to apply an effect to make it sound as if the piece were played in a large concert hall or some other room. The DFT makes this possible when used together with the idea of *convolution*.

When a balloon is popped in large, echoic room, although the sound of the actual pop only lasts a few milliseconds, the sound echoes about the room for up to several seconds. This echoing sound is referred to as the *impulse response* of the room, and is a way of approximating the acoustics of a room.

First, we need a recording of how the room responds to a short pulse of sound. Effective ways of producing a loud sound approximating a pulse–other than creating an actual pulse with a computer–include firing a (preferably blank) gunshot, popping a balloon, or, if neither of those options are available, clapping the hands one time.

Recall that we model sound with discrete samples of a soundwave in rapid succession. When these sounds are played back, the ear percieves them as a continuous soundwave. In other words, sound playback is a series of pulses of varying intensities, similar to the pulse in an impulse response. If we "mix" the individual sounds of an instrument in a carpeted room with the impulse response from a concert hall, then the new soundwave will sound as if the instrument is being played in the concert hall.

This "mixing" is better referred to as *convolution*. With the impulse response, we can see how sound echoes and decays. This "echoicness" can then be combined with each audio sample to reproduce the echo at the appropriate time and amplitude. Since audio needs to be sampled frequently (44100 samples per second is standard) to create smooth playback, a recording of a song can contain tens of millions of samples (one minutes at 44100 samples per second gives 2646000 samples). Each of these samples needs to be combined with the impulse response, which may be several seconds long. This may be starting to seem computationally infeasible or at least very difficult, but surprisingly, it is not. The key is to recognize that this process can be described as a convolution: namely, the final sound is simply the convolution of the our original sound with the impulse response. In other words, it is the original sound with the echoes of the prexious n samples, where n is the number of samples in the impulse response. We can calculate convolutions quickly

using the convolution theorem:

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

where $\mathcal{F}$ is the Fourier Transform, $*$ is convolution, and $\cdot$ is component-wise multiplication. Thus we calculate the convolution of two arrays by simply taking the Fourier transform of each, multiplying them pointwise, and then taking the inverse Fourier transform.

Problem 2. (Optional)[a] Find a large room or area with good acoustics, and record (an approximation to) its impulse response using a balloon pop. To record the sound, you will want to use at least a decent microphone. You may want to record it using the program Audacity[b] and a laptop. If you use a unidirectional microphone, be sure the microphone is pointing at the balloon when you pop it, so that the direct sound from the pop is picked up. (If you don't, the result will still be okay. However, after the convolution it will probably sound somewhat distant, as if we were standing somewhere where we couldn't hear the music directly.) If you've chosen a good room, the response should be audible for at least a full second.

Include a plot of both the waveform and spectrum of the impulse response you recorded.

[a] If the instructor does not require this problem then students may use the provided `balloon.wav` file which contains the sound of a balloon pop in a large room.

[b] Audacity is free sound manipulation software and may be downloaded at http://audacity.sourceforge.net

Problem 3. Download and listen to the file `chopin.wav`. You will hear a piano being played in a dead room with little or no acoustics. Using the Convolution Theorem, take the convolution of this signal with the impulse response recorded in the previous problem. The convolution given in the theorem is *circular*, meaning that sounds at the end of the signal will tend to mix with sounds at the beginning of the signal. To avoid this effect, add several seconds of silence (as long as the impulse response echo) to the end of `chopin.wav` by appending zeroes to the end of the signal. Also, keep in mind that the Convolution Theorem requires both signals to have the same length; therefore you will need to pad the smaller of your two *transformed* signals (namely, the transformed impulse response signal) with zeros in order to make it the same size as the other transformed signal. These zeros should be added to the middles of the transformed signal, as we need to maintain its symmetric structure. Describe the resulting sound.

To summarize:

1. Read in `chopin.wav` and the impulse response with `wavfile`,

2. Add several seconds of silence to the signal from `chopin.wav`,
3. Insert zeros into the middle of the impulse response transform so that it is the same length as `chopin.wav`,
4. Calculate the convolution of the signals,
5. And finally, calculate the inverse Fourier transform.

In some instances, a circular convolution is actually desirable. For instance, an interesting effect is achieved by taking the circular convolution of a long segment of white noise with some other (shorter) sound. We can create white noise using SciPy's `random` module:

```
# Create 10 seconds of mono white noise.
samplerate = 22050
noise = sp.int16(sp.random.randint(-32767, 32767, samplerate * 10))
```

Problem 4. Create white noise and listen to the resulting sound (**CAUTION:** Turn your volume *way* down! It may be very, *very* loud). This kind of noise is called "white" because it contains all frequencies with the same strength, or rather, with the same expected strength (since the amplitude of a specific frequency is a matter of chance). In order to see this, plot the spectrum of the noise.

Now we can take the circular convolution of this noise with some other sound. For instance, let's use `tada.wav`. The result is in `tada-conv.wav`. We notice that the original short sound has been sustained to an indefinite length. The result is not a set of static tones, but rather a rich sound which preserves not only the tones, but the texture, of the original sound; you can hear different tones fluctuating randomly in amplitude over time. If you were to play this `tada-conv.wav` on repeat, you would find that, because we used a circular convolution, the sound loops seamlessly from the end back to the beginning; however, most sound players are not capable of doing this properly, so you will probably hear a break in the sound. To demonstrate the "seamlessness", we can paste together multiple copies of the sound consecutively:

```
rate, sig = wavfile.read('tada-conv.wav')
sig = sp.append(sig, sig)
sig = sp.append(sig, sig)
```

Listen to the resulting sound, and notice that we are not able to identify where the sound loops back to the beginning, because there is no break or click.