7 Filtering and Convolution

Lab Objective: *The Fourier transform reveals information in the frequency domain about signals and images that might not be apparent in the time (sound) or spatial domain (image). It provides an extremely powerful tool to analyze difficult problems simply. In this lab, we learn how to use the Fourier transform to effectively convolve sound signals and filter out unwanted noise.*

Convolution

Mixing two sounds signals is a common method used in signal processing and analysis. This is done through a discrete *convolution*. Given two periodic sound samples f and g of length n, the convolution of these two samples is an n-dimensional vector where the kth component is given by

$$(f * g)_k = \sum_{j=1}^{n-1} f_{k-j} g_j, \qquad k = 0, 1, 2, \ldots, n-1. \tag{7.1}$$

Since audio needs to be sampled frequently to create smooth playback, a recording of a song can contain tens of millions of samples. For example, a signal that is one minute long would have 2646000 samples if the signal was sampled at 44100 per second (which is the standard rate). Therefore, convolving samples using the naïve method of convolution (7.1) is very computationally expensive and often infeasible.

Fortunately, the Fourier transform can calculate convolutions quickly. The Convolution Theorem states that

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g), \tag{7.2}$$

where $\mathcal{F}$ is the discrete Fourier transform, $*$ is convolution, and $\cdot$ is component-wise multiplication. Thus, the convolution of two arrays is

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)), \tag{7.3}$$

where $\mathcal{F}^{-1}$ is the inverse discrete Fourier transform.

Recall that although the samples used are real numbers, taking the inverse Fourier transform of the samples may have small complex components due to rounding errors. To avoid this error, remember to take the real part of the inverse Fourier transform.

Circular Convolution

The Convolution Theorem requires that the samples be periodic in the time domain. Due to the mathematical properties of the Fourier transform, the result of this convolution is a *circular convolution*. This means that the convolution is assumed to be periodic and the sounds at the end of the signal will tend to mix with sounds at the beginning of the signal. A circular convolution creates an interesting effect on a signal when convoluted with a segment of white noise; the sound will loop seamlessly from the end back to the beginning.

Problem 1. Use SciPy's `sp.fft()` and `sp.ifft()` to create a `SoundWave` object that is the circular convolution of `tada.wav` with two seconds of white noise. Note that the length of the two samples must be the same, so pad an array of zeros to the end of the `tada.wav` sample to match the length of the noise. You may use `np.append()` to add an array of zeros to the end of a sample.

Make a sample of white noise by using NumPy's `random` module:

```
# Create 2 seconds of mono white noise.
samplerate = 22050
noise = np.int16(np.random.randint(-32767, 32767, samplerate*2))
```

To test your result, use the `append()` method in the `SoundWave` class to add multiple copies of the signal consecutively. Your new signal should have a continuous, seamless transition.

Linear Convolution

Although circular convolution can have unique results, common mixtures of sounds do not have sound at the beginning of a signal to mix with the sound at the end of the signal. This form of convolution is called a *linear convolution*. The linear convolution of two samples with lengths N and M has length $N+M-1$. The simplest way to achieve this length when using the Convolution Theorem is to pad zeros at the end of both of the samples to have at least lengths $N+M-1$ and return only the first $N+M-1$ samples of the convolution.

Problem 2. Write a function that accepts two arrays of sound samples and returns the linear convolution of the samples using the Fourier transform. Make sure to pad the right amount of zeros to the end of both samples to avoid circular convolution and return the correct length of sample.

Print out and compare the time it takes to compute the convolution of the signals of `AEA.wav` and `EAE.wav` using the method you have written with SciPy's `sp.signal.fftconvolve()` function and a naive convolution function using Equation (7.1) given below.

```
def naive_convolve(sample1,sample2):
    sig1 = np.append(sample1, np.zeros(len(sample2)-1))
    sig2 = np.append(sample2, np.zeros(len(sample1)-1))

    final = np.zeros_like(sig1)
    rsig1 = sig1[::-1]
    for k in range(len(sig1)-1):
        final[i+1] = np.sum((np.append(rsig1[i:],rsig1[::-i][:i]))*sig2)
    return final
```

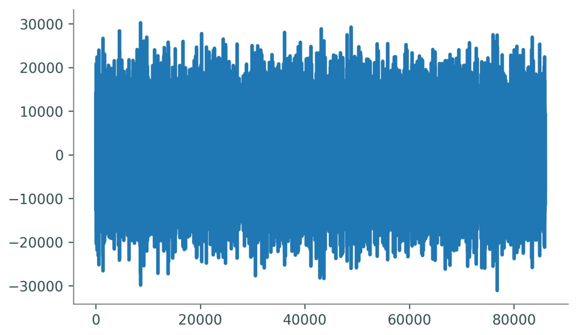
To test the convolution method, listen to the signal created with the fast Fourier transformation convolution. Compare your audio with the convolution created with SciPy's `signal.fftconvolve()`.

Suppose there is a recording of a musical piece played in a small, carpeted room with essentially no acoustics (little or no echo). The discrete linear convolution can mix this signal with an echoing sound to make the piece sound like it were played in a large concert hall with echo.

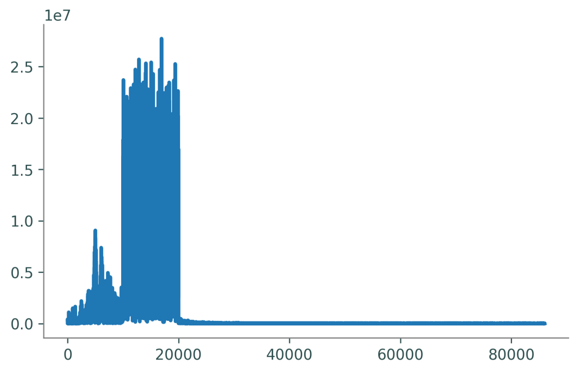
When a balloon is popped in a large room with echo, the sound resonate in the room for up to several seconds. This echoing sound is referred to as the *impulse response* of the room, and is a way of approximating the acoustics of a room. When the individual sounds of an instrument in a carpeted room is convoluted with the impulse response from a concert hall, the new signal will sound as if the instrument is being played in the concert hall.

Problem 3. The `chopin.wav` file is a signal with piano being played in a dead room with little or no acoustics, and the `balloon.wav` file is a recording of a balloon pop in an echoic room. Use SciPy's `signal.fftconvolve()` to take the convolution of `chopin.wav` with `balloon.wav`.

Listen to the new signal, there should be echo in the piano recording.



(a) Signal in the time domain.



(b) Signal in the frequency domain.

Figure 7.1: The `Noisysignal1.wav` signal

Cleaning up Noise

The Fourier transform also reveals important information about images and signals in the frequency domain. For example, Figure 7.1a plots a noisy recording of a voice over time. This plot has a lot of static and does not reveal a lot of information about the signal.

The Fourier transform of the signal in Figure 7.1b, however, reveals that the static in the time domain is the result of some concentrated noise between 1250 Hz to 2500 Hz. Remove this noise at those frequencies to remove the noise of the signal.

Recall that the plot of a signal in the frequency domain is the plot of the discrete Fourier transform (DFT) with the x-axis scaled to reveal the amplitude at each frequency. This was done by multiplying the domain of the DFT by the sampling rate and dividing by the number of samples in the signal.

Hence, to remove the high amplitudes at certain frequencies between between 1250 to 2500 Hz in the signal, find the indices of the DFT array that correspond to those frequencies and set them to zero. In order to find the correct indices of the DFT array corresponding to these frequencies, multiply the frequency by the number of samples and dividing by the sampling rate. Make sure that the index is an integer.

```
# Find the indices of the DFT that corresponds with the frequencies 1250 and ↩
    2500.
>>> low = 1250*len(samples)//rate
>>> high = 2500*len(samples)//rate
```

The plot in Figure 7.1b has also been cut in half since the DFT is symmetric. Therefore, if the coefficient at index j is set to 0, then the coefficient at index $N - j$ must be set to 0 as well, where N is the number of discrete Fourier samples.

```
# Set the chosen coefficients between low and high to 0.
>>> fft_sig[low:high]=0
>>> fft_sig[-high:-low]=0
```

Then calculate the inverse Fourier transform to get a new, clean signal.

The plot of this new signal in the time domain now reveals individual syllables as they are spoken. See Figure 7.2.

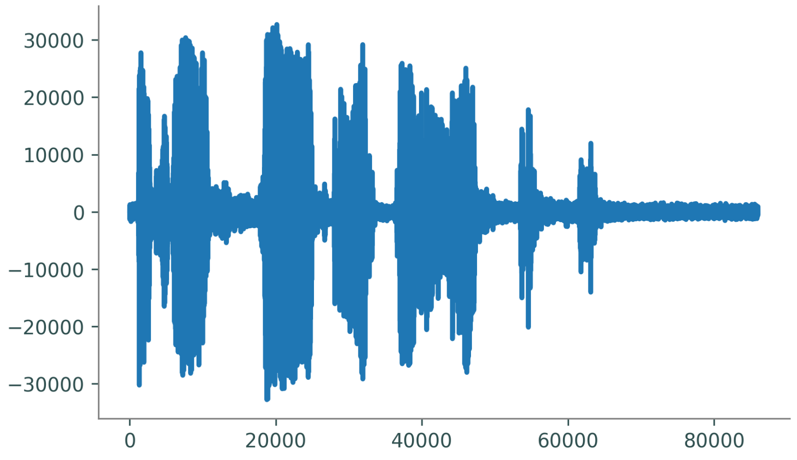


Figure 7.2: The plot of `Noisysignal1.wav` in the time domain after being cleaned.

Problem 4. Write a function that accepts a sound sample, a sample rate and low and high frequencies that define a range of frequencies to remove using the technique described above. It should return an array of samples that has the indicated frequencies removed.

Listen to `Noisysignal2.wav`, which just sounds like random noise. Use the `SoundWave` class to plot the discrete Fourier transform of this signal to see at what frequencies is the noise present. Remove the noise using the function you have written and create a `SoundWave` object from the new signal.

It may be helpful to plot the DFT of the new signal to fine-tune the low and high frequencies chosen to filter out.

Listen to the filtered signal and see if you can recognize the famous person speaking.

When a digital audio signal is played on a computer, the signal is sent to a speaker, which vibrates, producing sound waves. When multiple speakers are used, they can all produce the same signal or they can produce different signal. When there is only one signal, the sound is *monoaural*, or *mono*. When speakers produce more than one signal, 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).

All of the sounds used in this lab so far were mono; hence, the samples were only one dimensional arrays. More commonly, signals are multi-dimensional, and can be treated similarly to mono signals.

Problem 5. During the 2010 World Cup in South Africa, large plastic horns called vuvuzelas were blown excessively throughout the games. Broadcasting organizations faced difficulties with their programs due to the noise level of these vuvuzelas. To solve this problem, audio filtering techniques were used to cancel out the sound of the vuvuzela which has a frequency of around 200-500 Hz.

Listen to `vuvuzela.wav`[a] and notice the low humming sound of the vuvuzelas in the background. Use the function written previously to create a new `SoundWave` object that removed the vuvuzela noise. Note that the sound file is a stereo sound with two sound signals. The first and second column of the array sample corresponds to the signals for the left and right speaker respectively. Filter out the frequencies of the vuvuzela in each signal. Then combine the two samples back to its original form.

Listen to the resulted signal to see whether the vuvuzela horns has successfully been filtered out.

[a]A clip of `https://www.youtube.com/watch?v=g_0NoBKWCT8`.

The 2-Dimensional FFT

The Fourier transform can be readily extended to any number of dimensions. Computationally, the problem reduces to performing the one-dimensional Fourier transform iteratively along each of the dimensions. This lab focuses only on the Fourier transform of two-dimensional matrices. The Fourier transform of two-dimensional matrices is useful for image denoising, image compression, edge-detection, image enhancement, and more.

Given a matrix A, first calculate the one-dimensional Fourier transform of each column, storing the result column-wise in an array the same shape as A. Then calculate the Fourier transform of each row of this resulting array. This yields the two-dimensional Fourier transform of A. Calculating the two-dimensional inverse Fourier transform is done in a similar fashion, but in the opposite order: first calculate the inverse Fourier transform of the rows, then the columns.

The NumPy module has functions that perform these operations.

```
>>> A = np.array([[5, 3, 1], [4, 2, 7], [8, 9, 3]])
# Calculate the Fourier transform of A.
>>> fft = np.fft.fft2(A)
# Calculate the inverse Fourier transform.
>>> ifft = np.fft.ifft2(fft)
>>> np.allclose(A, ifft)
True
```

Images

Just as the one-dimensional Fourier transform can be used to remove noise in sounds, its two-dimensional counterpart can be used to remove noise in images. By taking the two-dimensional Fourier transform of a noisy, or blurry, image as described above, we can determine the frequencies that are causing the blur and alter them. Taking the inverse Fourier transform of this produces a less-blurry version of the original image. Note that in order for this process to work perfectly, the noise must be periodic and all problem areas in the Fourier transform must be changed correctly. As a result, it is very difficult to completely remove all noise from an image using only the Fourier transform, but depending on the situation, it may be capable of removing enough noise to be useful.

Below is an example of how this process works. The following code refers to figure **??**. First plot the original blurry image.

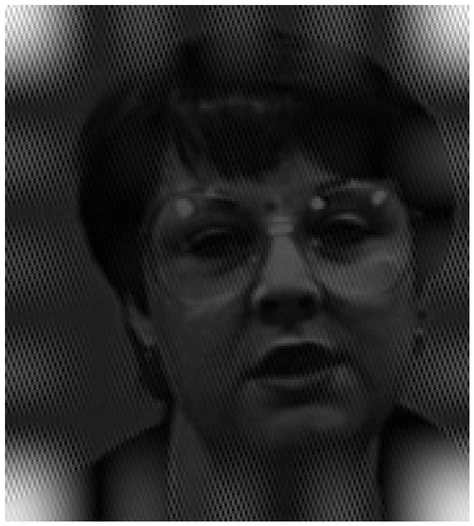
```
# Plot the blurry image (figure 1.3(a)).
>>> from scipy.misc import imread
>>> image = imread("face.png", True)
>>> plt.imshow(image, cmap='gray')
>>> plt.show()
```

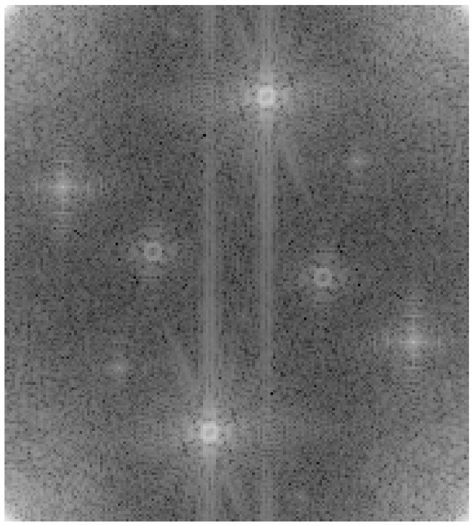
Now plot the Fourier transform of this image in order to see where the spikes in frequency are. To effectively visualize this plot, plot the log of the absolute value of the result.

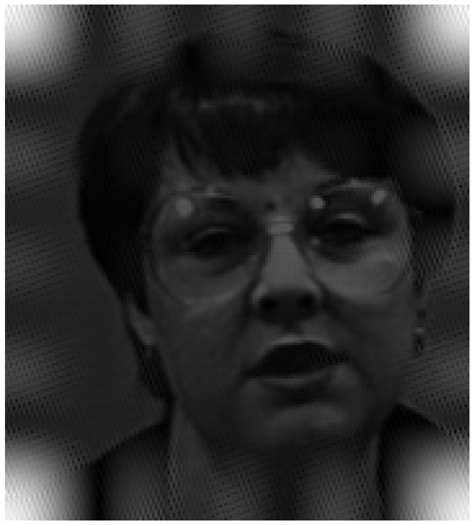
```
# Plot the Fourier transform of the blurry image (figure 1.3(b)).
>>> fft = np.fft.fft2(image)
>>> plt.imshow(np.log(np.abs(fft)), cmap='gray')
>>> plt.show()
```

Notice the spikes in the plot. In order to reduce the noise in the plot, replace these abnormally high values with values that are more similar to those around them. There are many ways to do this, but one possibility is to simply "patch" it by covering the area with a small matrix of values similar to other values in the plot.

```
# Cover the spikes in the Fourier transform (figure 1.3(d)).
>>> fft[30:40, 97:107] = np.ones((10,10)) * fft[33][50]
>>> fft[-39:-29, -106:-96] = np.ones((10,10)) * fft[33][50]
>>> plt.imshow(np.log(np.abs(fft)),cmap='gray')
>>> plt.show()
```


(a) The original blurry image.


(b) The FFT of the original image.


(c) The improved image.

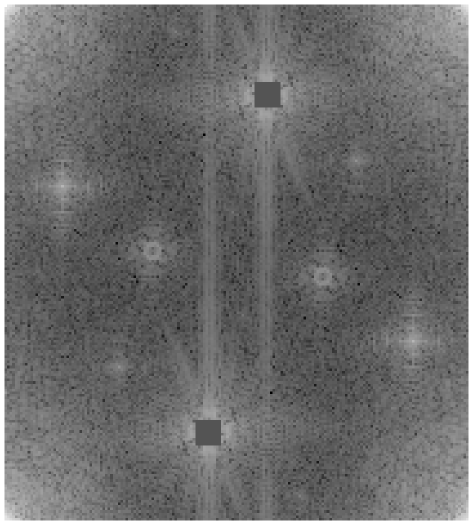

(d) The FFT of the improved image.

Figure 7.3: To remove noise from an image, take the Fourier transform of the image and replace the abnormalities with values more consistent with the rest of the FFT. Notice that the new image is less noisy, but only slightly. This is because only some of the abnormalities in the FFT were changed. In order to further decrease the noise, we would need to further alter the FFT.

Finally, take the inverse Fourier transform of this to get an image similar to the original, but with less noise. Notice that this new image still has noise present, but is less noisy than the original. Once again, use the absolute value of this result to plot it.

```
# Plot the new image (figure 1.3(c)).
>>> new_image = np.abs(np.fft.ifft2(fft))
>>> plt.imshow(new_image, cmap='gray')
>>> plt.show()
```

In practice, it often will suffice to remove some of the noise, even if it is not possible to remove all of it. For example, if the purpose of cleaning up an image was to retrieve some information from it that was not easily distinguishable before, then it would be sufficient to only remove enough noise to accurately extract that information. To further improve the image, a similar process to the one described above must be done to cover the remaining spikes.

Problem 6. One potential application of removing noise from an image is cleaning up an image of a license plate to retrieve its information. The file `license_plate.png` contains a blurred image of a license plate. In the bottom right corner of this image, there is a sticker that has information about the month and year that the license plate was renewed. However, in its current state the year is not clearly legible. Use the two-dimensional Fourier transform to clean up the image enough that the year in the bottom right corner is legible.