1 Convolution and Filtering

Lab Objective: *The Fourier transform reveals information in the frequency domain about signals and images that might not be apparent in the usual time (sound) or spatial (image) domain. In this lab, we use the discrete Fourier transform to efficiently convolve sound signals and filter out some types of unwanted noise from both sounds and images. This lab is a continuation of the Discrete Fourier Transform lab and should be completed in the same Jupyter Notebook.*

Convolution

Mixing two sounds signals—a common procedure in signal processing and analysis—is usually done through a *discrete convolution.* Given two periodic sound sample vectors $\mathbf{f}$ and $\mathbf{g}$ of length n, the discrete convolution of $\mathbf{f}$ and $\mathbf{g}$ is a vector of length n where the kth component is given by

$$(\mathbf{f} * \mathbf{g})_k = \sum_{j=0}^{n-1} f_{k-j} g_j, \qquad k = 0, 1, 2, \ldots, n-1. \tag{1.1}$$

Since audio needs to be sampled frequently to create smooth playback, a recording of a song can contain tens of millions of samples; even a one-minute signal has $2,646,000$ samples if it is recorded at the standard rate of $44,100$ samples per second ($44,100$ Hz). The naïve method of using the sum in (1.1) n times is $O(n^2)$, which is often too computationally expensive for convolutions of this size.

Fortunately, the discrete Fourier transform (DFT) can be used compute convolutions efficiently. The *finite convolution theorem* states that the Fourier transform of a convolution is the element-wise product of Fourier transforms:

$$F_n(\mathbf{f} * \mathbf{g}) = n(F_n\mathbf{f}) \odot (F_n\mathbf{g}). \tag{1.2}$$

In other words, convolution in the time domain is equivalent to component-wise multiplication in the frequency domain. Here F_n is the DFT on $\mathbb{R}^n$, $*$ is discrete convolution, and $\odot$ is component-wise multiplication. Thus, the convolution of $\mathbf{f}$ and $\mathbf{g}$ can be computed by

$$\mathbf{f} * \mathbf{g} = nF_n^{-1}((F_n\mathbf{f}) \odot (F_n\mathbf{g})), \tag{1.3}$$

where F_n^{-1} is the *inverse discrete Fourier transform* (IDFT). The fast Fourier transform (FFT) puts the cost of (1.3) at $O(n \log n)$, a huge improvement over the naïve method.

Note

Although individual samples are real numbers, results of the IDFT may have small complex components due to rounding errors. These complex components can be safely discarded by taking only the real part of the output of the IDFT.

```
>>> import numpy
>>> from scipy.fftpack import fft, ifft # Fast DFT and IDFT functions.

>>> f = np.random.random(2048)
>>> f_dft_idft = ifft(fft(f)).real      # Keep only the real part.
>>> np.allclose(f, f_dft_idft)          # Check that IDFT(DFT(f)) = f.
True
```

Achtung!

SciPy uses a different convention to define the DFT and IDFT than this and the previous lab, resulting in a slightly different form of the convolution theorem. Writing SciPy's DFT as $\hat{F}_n$ and its IDFT as $\hat{F}_n^{-1}$, we have $\hat{F}_n = nF_n$, so (1.3) becomes

$$\mathbf{f} * \mathbf{g} = \hat{F}_n^{-1}((\hat{F}_n\mathbf{f}) \odot (\hat{F}_n\mathbf{g})), \tag{1.4}$$

without a factor of n. Use (1.4), not (1.3), when using `fft()` and `ifft()` from `scipy.fftpack`.

Circular Convolution

The definition (1.1) and the identity (1.3) require $\mathbf{f}$ and $\mathbf{g}$ to be periodic vectors. However, the convolution $\mathbf{f}*\mathbf{g}$ can always be computed by simply treating each vector as periodic. The convolution of two raw sample vectors is therefore called the *periodic* or *circular convolution*. This strategy mixes sounds from the end of each signal with sounds at the beginning of each signal.

Problem 1.

Implement the `__mul__()` magic method for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A * B` creates a new `SoundWave` object whose samples are the circular convolution of the samples from `A` and `B`. If the samples from `A` and `B` are not the same length, append zeros to the shorter array to make them the same length before convolving. Use `scipy.fftpack` and (1.4) to compute the convolution, and raise a `ValueError` if the sample rates from `A` and `B` are not equal.

A circular convolution creates an interesting effect on a signal when convolved with a segment of white noise: the sound loops seamlessly from the end back to the beginning. To see this, generate two seconds of white noise (at the same sample rate as `tada.wav`) with the following code.

```
>>> rate = 22050        # Create 2 seconds of white noise at a given rate.
>>> white_noise = np.random.randint(-32767, 32767, rate*4, dtype=np.int16)
```

Next, convolve `tada.wav` with the white noise. Finally, use the `>>` operator to append the convolution result to itself. This final signal sounds the same from beginning to end, even though it is the concatenation of two signals.

Linear Convolution

Although circular convolutions can give interesting results, most common sound mixtures do not combine sounds at the beginning of one signal with sounds at the end of another. Whereas circular convolution assumes that the samples represent a full period of a periodic function, *linear convolution* aims to combine non-periodic discrete signals in a way that prevents the beginnings and endings from interacting. Given two samples with lengths n and m, the simplest way to achieve this is to pad both samples with zeros so that they each have length $n+m-1$, compute the convolution of these larger arrays, and take the first $n+m-1$ entries of that convolution.

Problem 2.

Implement the `__pow__()` magic method for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A ** B` creates a new `SoundWave` object whose samples are the linear convolution of the samples from `A` and `B`. Raise a `ValueError` if the sample rates from `A` and `B` are not equal.

Because `scipy.fftpack` performs best when the length of the inputs is a power of 2, start by computing the smallest 2^a such that $2^a \geq n+m-1$, where $a \in \mathbb{N}$ and n and m are the number of samples from `A` and `B`, respectively. Append zeros to each sample so that they each have 2^a entries, then compute the convolution of these padded samples using (1.4). Use only the first $n+m-1$ entries of this convolution as the samples of the returned `SoundWave` object.

To test your method, read `CGC.wav` and `GCG.wav`. Time (separately) the convolution of these signals with `SoundWave.__pow__()` and with `scipy.signal.fftconvolve()`. Compare the results by listening to the original and convolved signals.

Problem 3. Clapping in a large room with an echo produces a sound that resonates 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 sound of a single instrument in a carpeted room is convolved with the impulse response from a concert hall, the new signal sounds as if the instrument is being played in the concert hall.

The file `chopin.wav` contains a short clip of a piano being played in a room with little or no echo, and `balloon.wav` is a recording of a balloon being popped in a room with a substantial echo (the impulse). Use your method from Problem 2 or `scipy.signal.fftconvolve()` to compute the linear convolution of `chopin.wav` and `balloon.wav`.

Filtering Frequencies with the DFT

The DFT also provides a way to clean a signal by altering some of its frequencies. Consider `noisy1.wav`, a noisy recording of a short voice clip. The time-domain plot of the signal only shows that the signal has a lot of static. On the other hand, the signal's DFT suggests that the static may be the result of some concentrated noise between about 1250–2600 Hz. Removing these frequencies could result in a much cleaner signal.

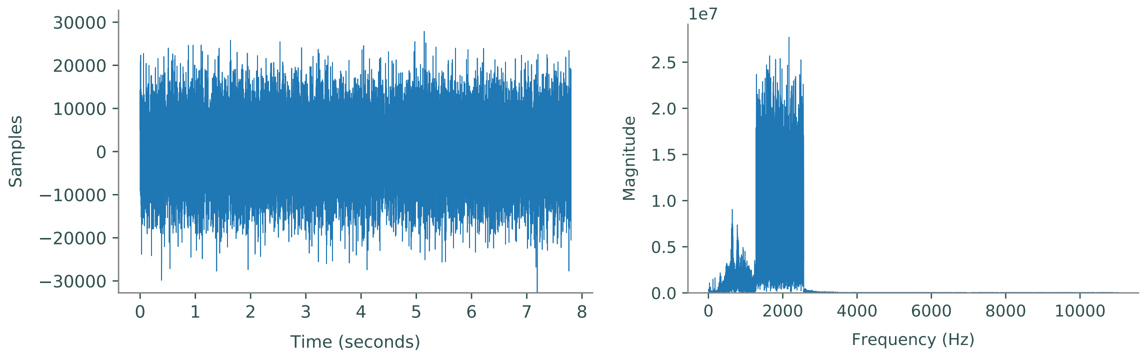


Figure 1.1: The time-domain plot (left) and DFT (right) of `noisy1.wav`.

To implement this idea, recall that the kth entry of the DFT array $\mathbf{c} = F_n\mathbf{f}$ corresponds to the frequency $v = kr/n$ in Hertz, where r is the sample rate and n is the number of samples. Hence, the DFT entry c_k corresponding to a given frequency v in Hertz has index $k = vn/r$, rounded to an integer if needed. In addition, since the DFT is symmetric, c_{n-k} also corresponds to this frequency. This suggests a strategy for filtering out an unwanted interval of frequencies $[v_{\text{low}}, v_{\text{high}}]$ from a signal:

1. Compute the integer indices k_{low} and k_{high} corresponding to v_{low} and v_{high}, respectively.
2. Set the entries of the signal's DFT from k_{low} to k_{high} and from $n - k_{\text{high}}$ to $n - k_{low}$ to zero, effectively removing those frequencies from the signal.
3. Take the IDFT of the modified DFT to obtain the cleaned signal.

Using this strategy to filter `noisy1.wav` results in a much cleaner signal. However, any "good" frequencies in the affected range are also removed, which may decrease the overall sound quality. The goal, then, is to remove only as many frequencies as necessary.

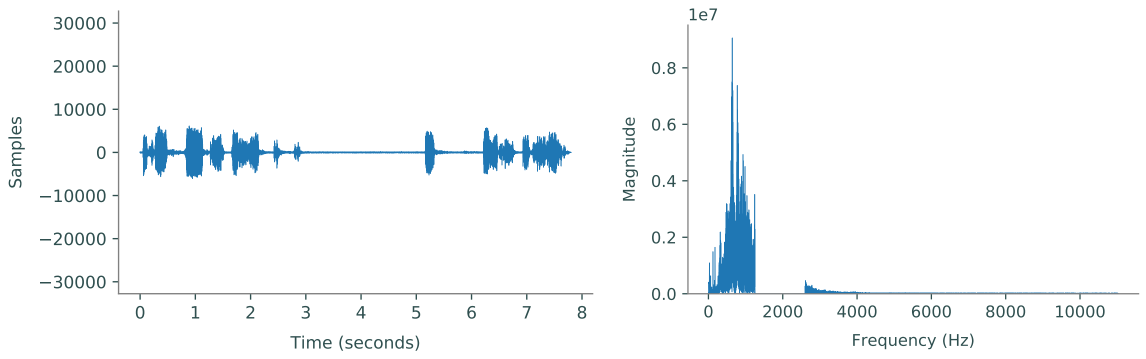


Figure 1.2: The time-domain plot (left) and DFT (right) of `noisy1.wav` after being cleaned.

Problem 4. Add a method to the `SoundWave` class that accepts two frequencies v_{low} and v_{high} in Hertz. Compute the DFT of the stored samples and zero out the frequencies in the range $[v_{\text{low}}, v_{\text{high}}]$ (remember to account for the symmetry DFT). Take the IDFT of the altered array and store it as the sample array.

Test your method by cleaning `noisy1.wav`, then clean `noisy2.wav`, which also has some artificial noise that obscures the intended sound.
(Hint: plot the DFT of `noisy2.wav` to determine which frequencies to eliminate.)

A digital audio signal made of a single sample vector with is called *monoaural* or *mono*. When several sample vectors with the same sample rate and number of samples are combined into a matrix, the overall signal is called *stereophonic* or *stereo*. This allows multiple speakers to each play one *channel*—one of the original sample vectors—simultaneously. "Stereo" usually means there are two channels, but there may be any number of channels (5.1 surround sound, for instance, has five).

Most stereo sounds are read as $n \times m$ matrices, where n is the number of samples and m is the number of channels (i.e., each column is a channel). However, some functions, including Jupyter's embedding tool `IPython.display.Audio()`, receive stereo signals as $m \times n$ matrices (each row is a channel). Be aware that both conventions are common.

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 incessant noise level. Eventually, audio filtering techniques were used to cancel out the sound of the vuvuzela, which has a frequency of around 200–500 Hz.

The file `vuvuzela.wav`[a] is a stereo sound with two channels. Use your function from Problem 4 to clean the sound clip by filtering out the vuvuzela frequencies in each channel. Recombine the two cleaned samples.

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

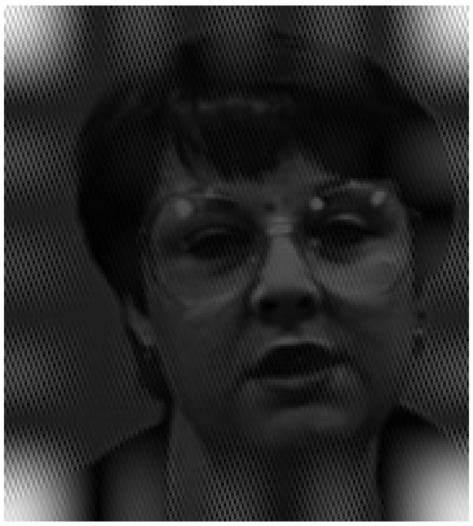
The Two-dimensional Discrete Fourier Transform

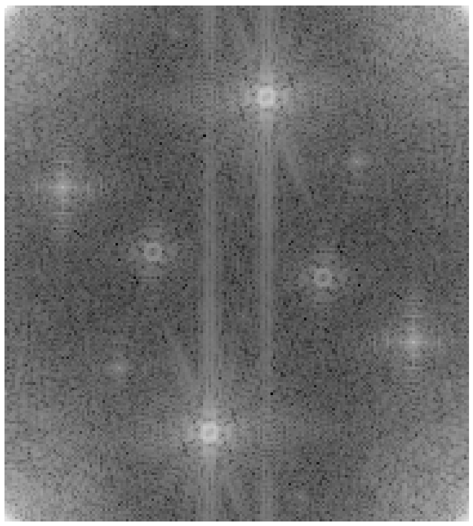
The DFT can be easily extended to any number of dimensions. Computationally, the problem reduces to performing the usual one-dimensional DFT iteratively along each of the dimensions. For example, to compute the two-dimensional DFT of an $m \times n$ matrix, calculate the usual DFT of each of the n columns, then take the DFT of each of the m rows of the resulting matrix. Calculating the two-dimensional IDFT is done in a similar fashion, but in reverse order: first calculate the IDFT of the rows, then the IDFT of the resulting columns.

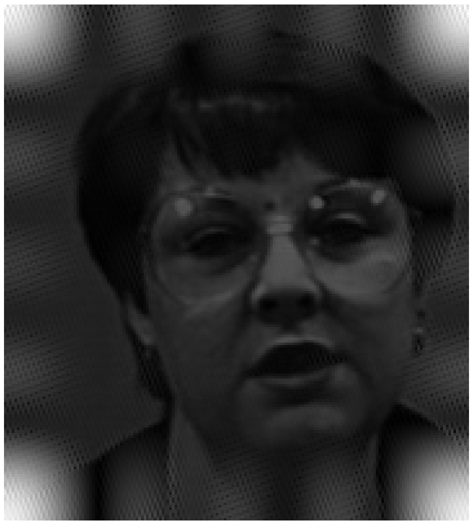
```
>>> from scipy.fftpack import fft2, ifft2

>>> A = np.random.random((10,10))
>>> A_dft = fft2(A)                 # Calculate the 2d DFT of A.
>>> A_dft_ifft = ifft2(A_dft).real  # Calculate the 2d IDFT.
>>> np.allclose(A, A_dft_ifft)
True
```

Just as the one-dimensional DFT can be used to remove noise in sounds, its two-dimensional counterpart can be used to remove "noise" in images. The procedure is similar to the filtering technique in Problems 4 and 5: take the two-dimensional DFT of the image matrix, modify certain entries of the DFT matrix to remove unwanted frequencies, then take the IDFT to get a cleaner version of the original image. This strategy makes the fairly strong assumption that the noise in the image is periodic and corresponds to certain frequencies. While this may seem like an unlikely scenario, it does actually occur in many digital images—for an example, try taking a picture of a computer screen with a digital camera.


(a) The original blurry image.


(b) The DFT of the original image.


(c) The improved image.

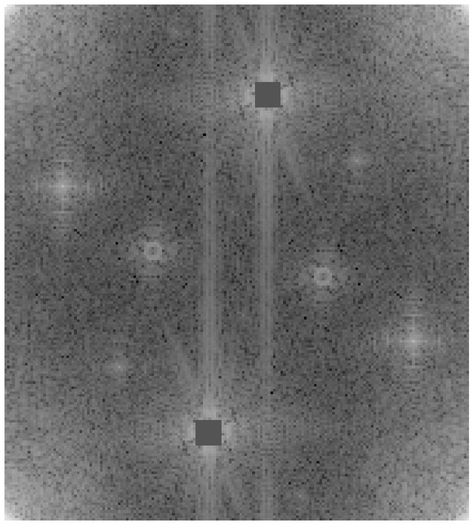

(d) The DFT of the improved image.

Figure 1.3: To remove noise from an image, take the DFT of the image and replace the abnormalities with values more consistent with the rest of the DFT. Notice that the new image is less noisy, but only slightly. This is because only some of the abnormalities in the DFT were changed; in order to further decrease the noise, we would need to further alter the DFT.

To begin cleaning an image with the DFT, take the two-dimensional DFT of the image matrix. Identify *spikes*—abnormally high frequency values that may be causing the noise—in the image DFT by plotting the log of the magnitudes of the Fourier coefficients. With `cmap="gray"`, spikes show up as bright spots. See Figures 1.3a–1.3b.

```
# Read the image.
>>> import imageio
>>> image = imageio.imread("noisy_face.png")

# Plot the log magnitude of the image's DFT.
>>> im_dft = fft2(image)
>>> plt.imshow(np.log(np.abs(im_dft)), cmap="gray")
>>> plt.show()
```

Instead of setting spike frequencies to zero (as was the case for sounds), replace them with values that are similar to those around them. There are many ways to do this, but one convention is to simply "patch" each spike by setting portions of the DFT matrix to some set value, such as the mean of the DFT array. See Figure 1.3d.

Once the spikes have been covered, take the IDFT of the modified DFT to get a (hopefully cleaner) image. Notice that Figure 1.3c still has noise present, but it is a slight improvement over the original. However, it often suffices to remove some of the noise, even if it is not possible to remove it all with this method.

Problem 6. The file `license_plate.png` contains a noisy image of a license plate. The bottom right corner of the plate has is a sticker with information about the month and year that the vehicle registration was renewed. However, in its current state, the year is not clearly legible.

Use the two-dimensional DFT to clean up the image enough so that the year in the bottom right corner is legible. This may require a little trial and error.