Chapter 8

Filtering and Convolution

In this chapter I present one of the most important and useful ideas related to signal processing: the Convolution Theorem. But before we can understand the Convolution Theorem, we have to understand convolution. I'll start with a simple example, smoothing, and we'll go from there.

The code for this chapter is in `chap08.ipynb`, which is in the repository for this book (see Section 0.2). You can also view it at `http://tinyurl.com/thinkdsp08`.

8.1 Smoothing

Smoothing is an operation that tries to remove short-term variations from a signal in order to reveal long-term trends. For example, if you plot daily changes in the price of a stock, it would look noisy; a smoothing operator might make it easier to see whether the price was generally going up or down over time.

A common smoothing algorithm is a moving average, which computes the mean of the previous n values, for some value of n.

For example, Figure 8.1 shows the daily closing price of Facebook from May 17, 2012 to December 8, 2015. The gray line is the raw data, the darker line shows the 30-day moving average. Smoothing removes the most extreme changes and makes it easier to see long-term trends.

Smoothing operations also apply to sound signals. As an example, I'll start with a square wave at 440 Hz. As we saw in Section 2.2, the harmonics of

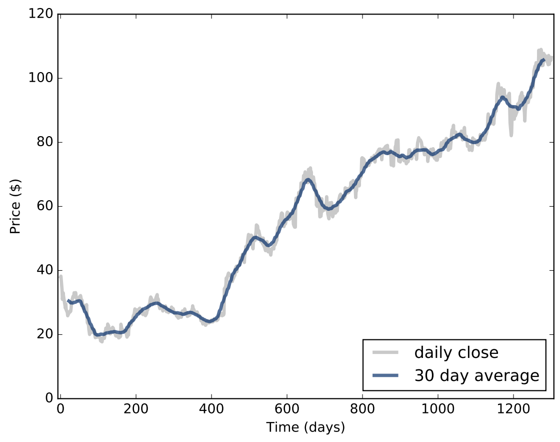

Figure 8.1: Daily closing price of Facebook stock and a 30-day moving average.

a square wave drop off slowly, so it contains many high-frequency components.

First I'll construct the signal and two waves:

```
signal = thinkdsp.SquareSignal(freq=440)
wave = signal.make_wave(duration=1, framerate=44100)
segment = wave.segment(duration=0.01)
```

`wave` is a 1-second slice of the signal; `segment` is a shorter slice I'll use for plotting.

To compute the moving average of this signal, I'll use a window similar to the ones in Section 3.7. Previously we used a Hamming window to avoid spectral leakage caused by discontinuity at the beginning and end of a signal. More generally, we can use windows to compute the weighted sum of samples in a wave.

For example, to compute a moving average, I'll create a window with 11 elements and normalize it so the elements add up to 1.

```
window = np.ones(11)
window /= sum(window)
```

Now I can compute the average of the first 11 elements by multiplying the window by the wave array:

```
ys = segment.ys
N = len(ys)
padded = thinkdsp.zero_pad(window, N)
```

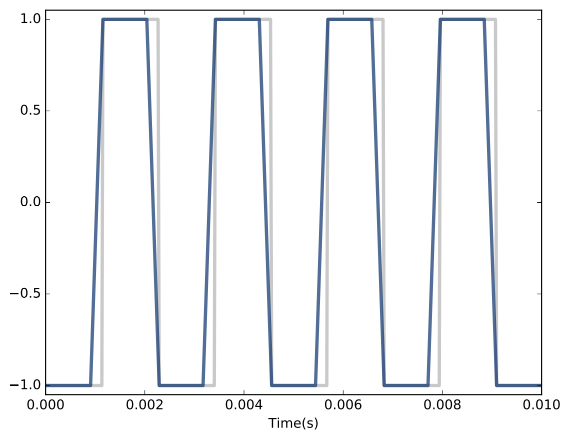

Figure 8.2: A square signal at 400 Hz (gray) and an 11-element moving average.

```
    prod = padded * ys
    sum(prod)
```

`padded` is a version of the window with zeros added to the end so it is the same length as `segment.ys`. Adding zeros like this is called **padding**.

`prod` is the product of the window and the wave array. The sum of the elementwise products is the average of the first 11 elements of the array. Since these elements are all -1, their average is -1.

To compute the next element of the moving average, we **roll** the window, which shifts the ones to the right and wraps one of the zeros from the end around to the beginning.

When we multiply the rolled window and the wave array, we get the average of the next 11 elements of the wave array, starting with the second.

```
    rolled = np.roll(rolled, 1)
    prod = rolled * ys
    sum(prod)
```

The result is -1 again.

We can compute the rest of the elements the same way. The following function wraps the code we have seen so far in a loop and stores the results in an array.

```
def smooth(ys, window):
    N = len(ys)
    smoothed = np.zeros(N)
    padded = thinkdsp.zero_pad(window, N)
    rolled = padded

    for i in range(N):
        smoothed[i] = sum(rolled * ys)
        rolled = np.roll(rolled, 1)
    return smoothed
```

`smoothed` is the array that will contain the results; `padded` is an array that contains the window and enough zeros to have length `N`; and `rolled` is a copy of `padded` that gets shifted to the right by one element each time through the loop.

Inside the loop, we multiply `ys` by `rolled` to select 11 elements and add them up.

Figure 8.2 shows the result for a square wave. The gray line is the original signal; the darker line is the smoothed signal. The smoothed signal starts to ramp up when the leading edge of the window reaches the first transition, and levels off when the window crosses the transition. As a result, the transitions are less abrupt, and the corners less sharp. If you listen to the smoothed signal, it sounds less buzzy and slightly muffled.

8.2 Convolution

The operation we just performed – applying a window function to each overlapping segment of a wave – is called **convolution**.

Convolution it is such a common operation that NumPy provides an implementation that is simpler and faster than my version:

```
    convolved = np.convolve(ys, window, mode='valid')
    smooth2 = thinkdsp.Wave(convolved, framerate=wave.framerate)
```

`np.convolve` computes the convolution of the wave array and the window. The mode flag `valid` indicates that it should only compute values when the window and the wave array overlap completely, so it stops when the right edge of the window reaches the end of the wave array. Other than that, the result is the same as in Figure 8.2.

Actually, there is one other difference. The loop in the previous section actually computes **cross-correlation**:

$$(f \star g)[n] = \sum_{m=0}^{N-1} f[m]g[n+m]$$

where f is a wave array with length N, g is the window, and $\star$ is the symbol for cross-correlation. To compute the nth element of the result, we shift g to the right, which is why the index is $n+m$.

The definition of convolution is slightly different:

$$(f * g)[n] = \sum_{m=0}^{N-1} f[m]g[n-m]$$

The symbol $*$ represents convolution. The difference is in the index of g: m has been negated, so the summation iterates the elements of g backward (assuming that negative indices wrap around to the end of the array).

Because the window we used in this example is symmetric, cross-correlation and convolution yield the same result. When we use other windows, we will have to be more careful.

You might wonder why convolution is defined like this, with the window applied in a way that seems backwards. There are two reasons:

- This definition comes up naturally for several applications, especially analysis of signal-processing systems, which is the topic of Chapter 10.
- Also, this definition is the basis of the Convolution Theorem, coming up very soon.

Finally, a note for people who know too much: in the presentation so far I have not distinguished between convolution and circular convolution. We'll get to it.

8.3 The frequency domain

Smoothing makes the transitions in a square signal less abrupt, and makes the sound slightly muffled. Let's see what effect this operation has on the spectrum. First I'll plot the spectrum of the original wave:

```
spectrum = wave.make_spectrum()
spectrum.plot(color=GRAY)
```

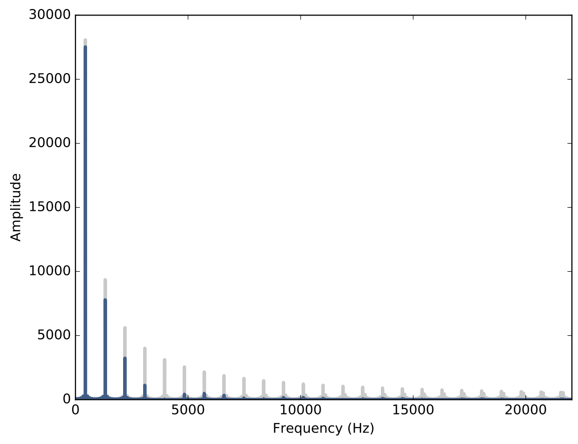

Figure 8.3: Spectrum of the square wave before and after smoothing.

Then the smoothed wave:

```
    convolved = np.convolve(wave.ys, window, mode='same')
    smooth = thinkdsp.Wave(convolved, framerate=wave.framerate)
    spectrum2 = smooth.make_spectrum()
    spectrum2.plot()
```

The mode flag `same` indicates that the result should have the same length as the input. In this example, it will include a few values that "wrap around", but that's ok for now.

Figure 8.3 shows the result. The fundamental frequency is almost unchanged; the first few harmonics are attenuated, and the higher harmonics are almost eliminated. So smoothing has the effect of a low-pass filter, which we saw in Section 1.5 and Section 4.4.

To see how much each component has been attenuated, we can compute the ratio of the two spectrums:

```
    amps = spectrum.amps
    amps2 = spectrum2.amps
    ratio = amps2 / amps
    ratio[amps<560] = 0
    thinkplot.plot(ratio)
```

`ratio` is the ratio of the amplitude before and after smoothing. When `amps` is small, this ratio can be big and noisy, so for simplicity I set the ratio to 0 except where the harmonics are.

Figure 8.4 shows the result. As expected, the ratio is high for low frequencies and drops off at a cutoff frequency near 4000 Hz. But there is another

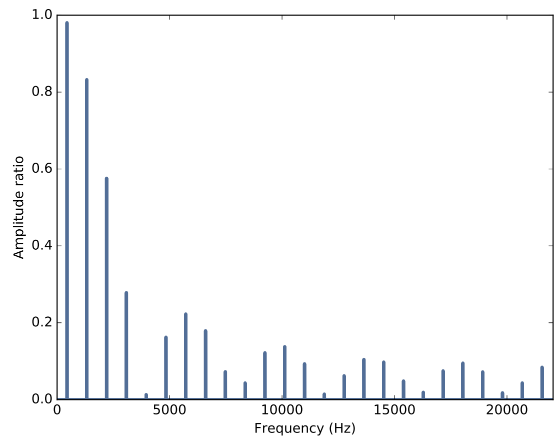

Figure 8.4: Ratio of spectrums for the square wave, before and after smoothing.

feature we did not expect: above the cutoff, the ratio bounces around between 0 and 0.2. What's up with that?

8.4 The Convolution Theorem

The answer is the Convolution Theorem. Stated mathematically:

$$\mathrm{DFT}(f * g) = \mathrm{DFT}(f) \cdot \mathrm{DFT}(g)$$

where f is a wave array and g is a window. In words, the Convolution Theorem says that if we convolve f and g, and then compute the DFT, we get the same answer as computing the DFT of f and g, and then multiplying the results element-wise.

When we apply an operation like convolution to a wave a wave, we say we are working in the **time domain**, because the wave is a function of time. When we apply an operation like multiplication to the DFT, we are working in the **frequency domain**, because the DFT is a function of frequency.

Using these terms, we can state the Convolution Theorem more concisely:

> Convolution in the time domain corresponds to multiplication in the frequency domain.

And that explains Figure 8.4, because when we convolve a wave and a window, we multiply the spectrum of the wave with the spectrum of the window. To see how that works, we can compute the DFT of the window:

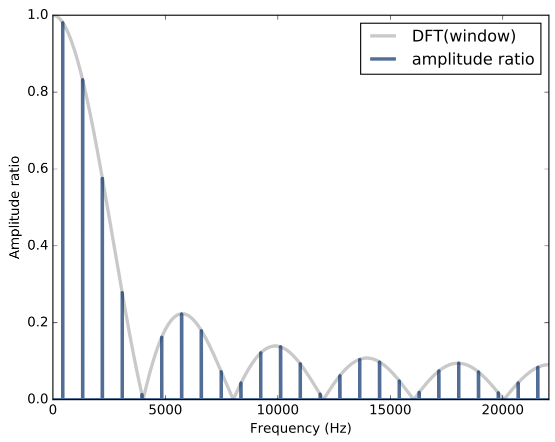

Figure 8.5: Ratio of spectrums for the square wave, before and after smoothing, along with the DFT of the smoothing window.

```
    padded = zero_pad(window, N)
    dft_window = np.fft.rfft(padded)
    thinkplot.plot(abs(dft_window))
```

`padded` contains the smoothing window, padded with zeros to be the same length as `wave`; `dft_window` contains the DFT of `padded`.

Figure 8.5 shows the result, along with the ratios we computed in the previous section. The ratios are exactly the amplitudes in `dft_window`. Mathematically:

$$\text{abs}(\text{DFT}(f * g))/\text{abs}(\text{DFT}(f)) = \text{abs}(\text{DFT}(g))$$

In this context, the DFT of a window is called a **filter**. For any convolution window in the time domain, there is a corresponding filter in the frequency domain. And for any filter that can be expressed by element-wise multiplication in the frequency domain, there is a corresponding window.

8.5 Gaussian filter

The moving average window we used in the previous section is a low-pass filter, but it is not a very good one. The DFT drops off steeply at first, but then it bounces around. Those bounces are called **sidelobes**, and they are there because the moving average window is like a square wave, so its spectrum contains high-frequency harmonics that drop off proportionally to $1/f$, which is relatively slow.

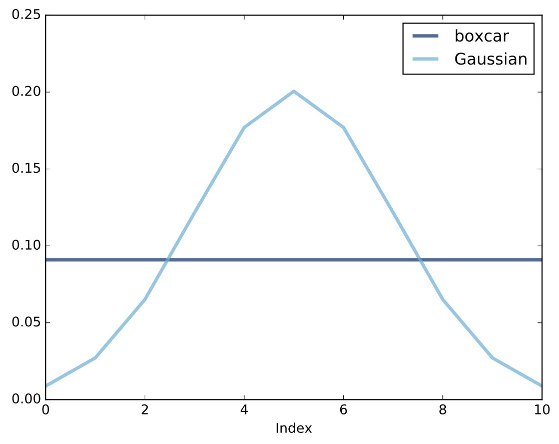

Figure 8.6: Boxcar and Gaussian windows.

We can do better with a Gaussian window. SciPy provides functions that compute many common convolution windows, including `gaussian`:

```
gaussian = scipy.signal.gaussian(M=11, std=2)
gaussian /= sum(gaussian)
```

`M` is the number of elements in the window; `std` is the standard deviation of the Gaussian distribution used to compute it. Figure 8.6 shows the shape of the window. It is a discrete approximation of the Gaussian "bell curve". The figure also shows the moving average window from the previous example, which is sometimes called a **boxcar window** because it looks like a rectangular railway car.

I ran the computations from the previous sections again with this window, and generated Figure 8.7, which shows the ratio of the spectrums before and after smoothing, along with the DFT of the Gaussian window.

As a low-pass filter, Gaussian smoothing is better than a simple moving average. After the ratio drops off, it stays low, with almost none of the sidelobes we saw with the boxcar window. So it does a better job of cutting off the higher frequencies.

The reason it does so well is that the DFT of a Gaussian curve is also a Gaussian curve. So the ratio drops off in proportion to $\exp(-f^2)$, which is much faster than $1/f$.

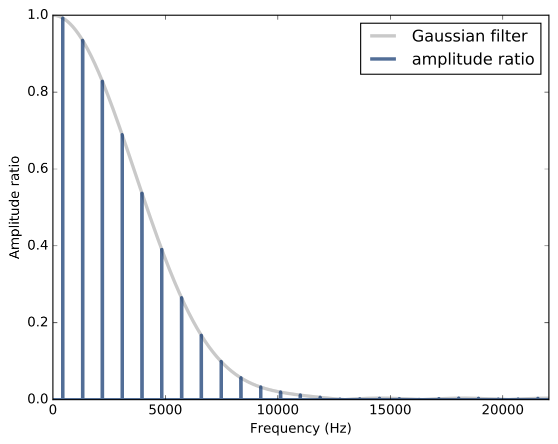

Figure 8.7: Ratio of spectrums before and after Gaussian smoothing, and the DFT of the window.

8.6 Efficient convolution

One of the reasons the FFT is such an important algorithm is that, combined with the Convolution Theorem, it provides an efficient way to compute convolution, cross-correlation, and autocorrelation.

Again, the Convolution Theorem states

$$\mathrm{DFT}(f * g) = \mathrm{DFT}(f) \cdot \mathrm{DFT}(g)$$

So one way to compute a convolution is:

$$f * g = \mathrm{IDFT}(\mathrm{DFT}(f) \cdot \mathrm{DFT}(g))$$

where $IDFT$ is the inverse DFT. A simple implementation of convolution takes time proportional to N^2; this algorithm, using FFT, takes time proportional to $N \log N$.

We can confirm that it works by computing the same convolution both ways. As an example, I'll apply it to the Facebook data shown in Figure 8.1.

```
import pandas as pd

names = ['date', 'open', 'high', 'low', 'close', 'volume']
df = pd.read_csv('fb.csv', header=0, names=names)
ys = df.close.values[::-1]
```

This example uses Pandas to read the data from the CSV file (included in the repository for this book). If you are not familiar with Pandas, don't worry:

I'm not going to do much with it in this book. But if you're interested, you can learn more about it in *Think Stats* at `http://thinkstats2.com`.

The result, `df`, is a DataFrame, one of the data structures provided by Pandas. `close` is a NumPy array that contains daily closing prices.

Next I'll create a Gaussian window and convolve it with `close`:

```
    window = scipy.signal.gaussian(M=30, std=6)
    window /= window.sum()
    smoothed = np.convolve(ys, window, mode='valid')
```

`fft_convolve` computes the same thing using FFT:

```
from np.fft import fft, ifft

def fft_convolve(signal, window):
    fft_signal = fft(signal)
    fft_window = fft(window)
    return ifft(fft_signal * fft_window)
```

We can test it by padding the window to the same length as `ys` and then computing the convolution:

```
    padded = zero_pad(window, N)
    smoothed2 = fft_convolve(ys, padded)
```

The result has $M-1$ bogus values at the beginning, where M is the length of the window. We can slice off the bogus values like this:

```
    M = len(window)
    smoothed2 = smoothed2[M-1:]
```

The result agrees with `fft_convolve` with about 12 digits of precision.

8.7 Efficient autocorrelation

In Section 8.2 I presented definitions of cross-correlation and convolution, and we saw that they are almost the same, except that in convolution the window is reversed.

Now that we have an efficient algorithm for convolution, we can also use it to compute cross-correlations and autocorrelations. Using the data from the previous section, we can compute the autocorrelation Facebook stock prices:

```
corrs = np.correlate(close, close, mode='same')
```

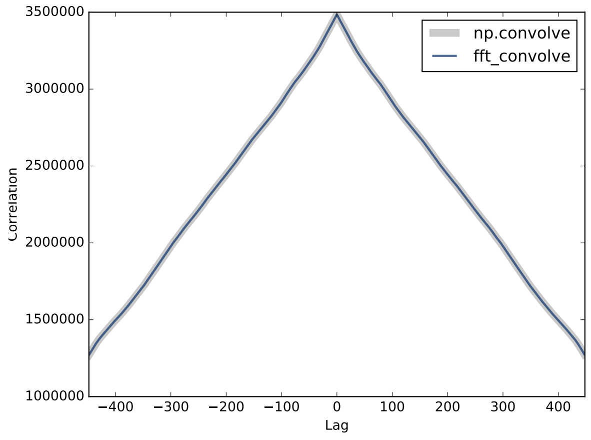

Figure 8.8: Autocorrelation functions computed by NumPy and `fft_correlate`.

With `mode='same'`, the result has the same length as `close`, corresponding to lags from $-N/2$ to $N/2-1$. The gray line in Figure 8.8 shows the result. Except at `lag=0`, there are no peaks, so there is no apparent periodic behavior in this signal. However, the autocorrelation function drops off slowly, suggesting that this signal resembles pink noise, as we saw in Section 5.3.

To compute autocorrelation using convolution, we have to zero-pad the signal to double the length. This trick is necessary because the FFT is based on the assumption that the signal is periodic; that is, that it wraps around from the end to the beginning. With time-series data like this, that assumption is invalid. Adding zeros, and then trimming the results, removes the bogus values.

Also, remember that convolution reverses the direction of the window. In order to cancel that effect, we reverse the direction of the window before calling `fft_convolve`, using `np.flipud`, which flips a NumPy array. The result is a view of the array, not a copy, so this operation is fast.

```
def fft_autocorr(signal):
    N = len(signal)
    signal = thinkdsp.zero_pad(signal, 2*N)
    window = np.flipud(signal)

    corrs = fft_convolve(signal, window)
    corrs = np.roll(corrs, N//2+1)[:N]
    return corrs
```

The result from `fft_convolve` has length $2N$. Of those, the first and last $N/2$

are valid; the rest are the result of zero-padding. To select the valid element, we roll the results and select the first N, corresponding to lags from $-N/2$ to $N/2-1$.

As shown in Figure 8.8 the results from `fft_autocorr` and `np.correlate` are identical (with about 9 digits of precision).

Notice that the correlations in Figure 8.8 are large numbers; we could normalize them (between -1 and 1) as shown in Section 5.6.

The strategy we used here for auto-correlation also works for cross-correlation. Again, you have to prepare the signals by flipping one and padding both, and then you have to trim the invalid parts of the result. This padding and trimming is a nuisance, but that's why libraries like NumPy provide functions to do it for you.

8.8 Exercises

Solutions to these exercises are in `chap08soln.ipynb`.

Exercise 8.1 The notebook for this chapter is `chap08.ipynb`. Read through it and run the code.

It contains an interactive widget that lets you experiment with the parameters of the Gaussian window to see what effect they have on the cutoff frequency.

What goes wrong when you increase the width of the Gaussian, `std`, without increasing the number of elements in the window, `M`?

Exercise 8.2 In this chapter I claimed that the Fourier transform of a Gaussian curve is also a Gaussian curve. For Discrete Fourier Transforms, this relationship is approximately true.

Try it out for a few examples. What happens to the Fourier transform as you vary `std`?

Exercise 8.3 If you did the exercises in Chapter 3, you saw the effect of the Hamming window, and some of the other windows provided by NumPy, on spectral leakage. We can get some insight into the effect of these windows by looking at their DFTs.

In addition to the Gaussian window we used in this chapter, create a Hamming window with the same size. Zero-pad the windows and plot their