Think DSP

Digital Signal Processing in Python

Version 0.9.8

Think DSP

Digital Signal Processing in Python

Version 0.9.8

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

Preface

The premise of this book (and the other books in the *Think X* series) is that if you know how to program, you can use that skill to learn other things. I am writing this book because I think the conventional approach to digital signal processing is backward: most books (and the classes that use them) present the material bottom-up, starting with mathematical abstractions like phasors.

With a programming-based approach, I can go top-down, which means I can present the most important ideas right away. By the end of the first chapter, you can decompose a sound into its harmonics, modify the harmonics, and generate new sounds.

0.1 Using the code

The code and sound samples used in this book are available from `https://github.com/AllenDowney/ThinkDSP`. Git is a version control system that allows you to keep track of the files that make up a project. A collection of files under Git's control is called a **repository**. GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

The GitHub homepage for my repository provides several ways to work with the code:

- You can create a copy of my repository on GitHub by pressing the Fork button. If you don't already have a GitHub account, you'll need to create one. After forking, you'll have your own repository on GitHub that you can use to keep track of code you write while working on this book. Then you can clone the repo, which means that you make a copy of the files on your computer.
- Or you could clone my repository. You don't need a GitHub account to do this, but you won't be able to write your changes back to GitHub.

- If you don't want to use Git at all, you can download the files in a Zip file using the button in the lower-right corner of the GitHub page.

All of the code is written to work in both Python 2 and Python 3 with no translation.

I developed this book using Anaconda from Continuum Analytics, which is a free Python distribution that includes all the packages you'll need to run the code (and lots more). I found Anaconda easy to install. By default it does a user-level installation, not system-level, so you don't need administrative privileges. And it supports both Python 2 and Python 3. You can download Anaconda from `http://continuum.io/downloads`.

If you don't want to use Anaconda, you will need the following packages:

- NumPy for basic numerical computation, `http://www.numpy.org/`;
- SciPy for scientific computation, `http://www.scipy.org/`;
- matplotlib for visualization, `http://matplotlib.org/`.

Although these are commonly used packages, they are not included with all Python installations, and they can be hard to install in some environments. If you have trouble installing them, I strongly recommend using Anaconda or one of the other Python distributions that include these packages.

Most exercises use Python scripts, but some also use the IPython notebook. If you have not used IPython notebook before, I suggest you start with the documentation at `http://ipython.org/ipython-doc/stable/notebook/notebook.html`.

I wrote this book assuming that the reader is familiar with core Python, including object-oriented features, but not NumPy, and SciPy.

I assume that the reader knows basic mathematics, including complex numbers. I use some linear algebra, but I will explain it as we go along.

Allen B. Downey

Needham MA

Allen B. Downey is a Professor of Computer Science at the Franklin W. Olin College of Engineering.

Contributor List

If you have a suggestion or correction, please send email to `downey@allendowney.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not as easy to work with. Thanks!

- Before I started writing, my thoughts about this book benefited from conversations with Boulos Harb at Google and Aurelio Ramos, formerly at Harmonix Music Systems.
- During the Fall 2013 semester, Nathan Lintz and Ian Daniher worked with me on an independent study project and helped me with the first draft of this book.
- On Reddit's DSP forum, the anonymous user RamjetSoundwave helped me fix a problem with my implementation of Brownian Noise. And andodli found a typo.
- In Spring 2015 I had the pleasure of teaching this material along with Prof. Oscar Mur-Miranda and Prof. Siddhartan Govindasamy. Both made many suggestions and corrections.

Contents

Chapter 1

Sounds and signals

A **signal** is a representation of a quantity that varies in time, or space, or both. That definition is pretty abstract, so let's start with a concrete example: sound. Sound is variation in air pressure. A sound signal represents variations in air pressure over time.

A microphone is a device that measures these variations and generates an electrical signal that represents sound. A speaker is a device that takes an electrical signal and produces sound. Microphones and speakers are called **transducers** because they transduce, or convert, signals from one form to another.

This book is about signal processing, which includes processes for synthesizing, transforming, and analyzing signals. I will focus on sound signals, but the same methods apply to electronic signals, mechanical vibration, and signals in many other domains.

They also apply to signals that vary in space rather than time, like elevation along a hiking trail. And they apply to signals in more than one dimension, like an image, which you can think of as a signal that varies in two-dimensional space. Or a movie, which is a signal that varies in two-dimensional space *and* time.

But we start with simple one-dimensional sound.

The code for this chapter is in `sounds.py`, which is in the repository for this book (see Section 0.1).

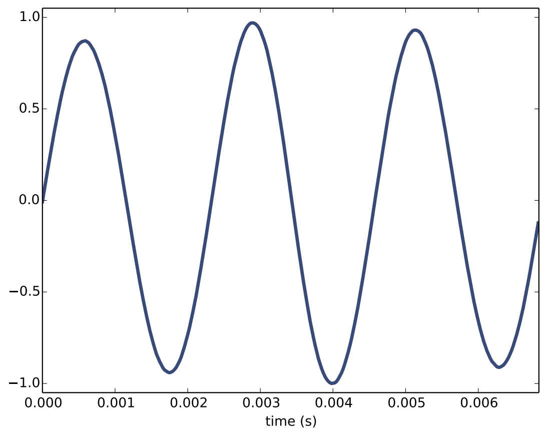

Figure 1.1: Segment from a recording of a tuning fork.

1.1 Periodic signals

We'll start with **periodic signals**, which are signals that repeat themselves after some period of time. For example, if you strike a tuning fork, it vibrates and generates sound. If you record that sound and plot the transduced signal, it looks like Figure 1.1.[1]

This signal is similar to a sinusoid, which means it has the same shape as the trigonometric sine function.

You can see that this signal is periodic. I chose the duration to show three full periods, also known as **cycles**. The duration of each cycle is about 2.3 ms.

The **frequency** of a signal is the number of cycles per second, which is the inverse of the period. The units of frequency are cycles per second, or **Hertz**, abbreviated "Hz".

The frequency of this signal is about 439 Hz, slightly lower than 440 Hz, which is the standard tuning pitch for orchestral music. The musical name of this note is A, or more specifically, A4. If you are not familiar with "scientific pitch notation", the numerical suffix indicates which octave the note is in. A4 is the A above middle C. A5 is one octave higher. See `http://en.wikipedia.org/wiki/Scientific_pitch_notation`.

A tuning fork generates a sinusoid because the vibration of the tines is a

[1] I got this recording from `http://www.freesound.org/people/zippi1/sounds/18871/`.

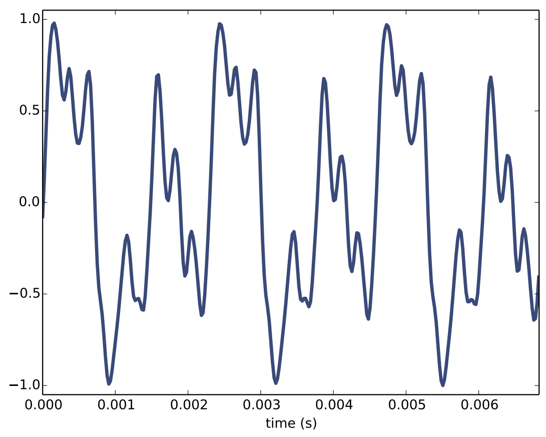

Figure 1.2: Segment from a recording of a violin.

form of simple harmonic motion. Most musical instruments produce periodic signals, but the shape of these signals is not sinusoidal. For example, Figure 1.2 shows a segment from a recording of a violin playing Boccherini's String Quintet No. 5 in E, 3rd movement.[2]

Again we can see that the signal is periodic, but the shape of the signal is more complex. The shape of a periodic signal is called the **waveform**. Most musical instruments produce waveforms more complex than a sinusoid. The shape of the waveform determines the musical **timbre**, which is our perception of the quality of the sound. People usually perceive complex waveforms as rich, warm and more interesting than sinusoids.

1.2 Spectral decomposition

The most important topic in this book is **spectral decomposition**, which is the idea that any signal can be expressed as the sum of simpler signals with different frequencies.

And the most important algorithm in this book is the **discrete Fourier transform**, or **DFT**, which takes a signal (a quantity varying in time) and produces its **spectrum**, which is the set of sinusoids that add up to produce the signal.

[2]The recording is from `http://www.freesound.org/people/jcveliz/sounds/92002/`. I identified the piece using `http://www.musipedia.org`.

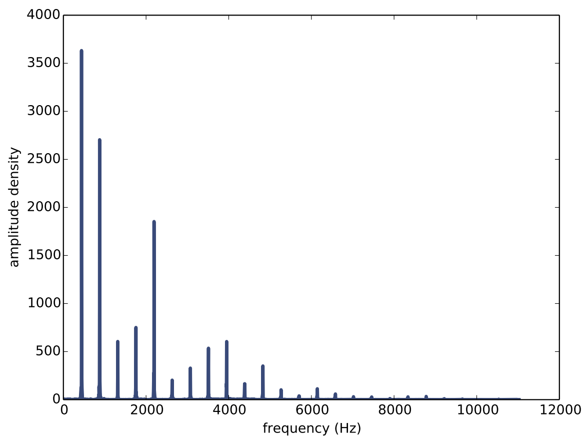

Figure 1.3: Spectrum of a segment from the violin recording.

For example, Figure 1.3 shows the spectrum of the violin recording in Figure 1.2. The x-axis is the range of frequencies that make up the signal. The y-axis shows the strength of each frequency component.

The lowest frequency component is called the **fundamental frequency**. The fundamental frequency of this signal is near 440 Hz (actually a little lower, or "flat").

In this signal the fundamental frequency has the largest amplitude, so it is also the **dominant frequency**. Normally the perceived pitch of a sound is determined by the fundamental frequency, even if it is not dominant.

The other spikes in the spectrum are at frequencies 880, 1320, 1760, and 2200, which are integer multiples of the fundamental. These components are called **harmonics** because they are musically harmonious with the fundamental:

- 880 is the frequency of A5, one octave higher than the fundamental.
- 1320 is approximately E6, which is a major fifth[3] above A5.
- 1760 is A6, two octaves above the fundamental.
- 2200 is approximately C♯7, which is a major third above A6.

These harmonics make up the notes of an A major chord, although not all in the same octave. Some of them are only approximate because the notes

[3]If you are not familiar with musical intervals like "major fifth", see `https://en.wikipedia.org/wiki/Interval_(music)`.

that make up Western music have been adjusted for **equal temperament** (see `http://en.wikipedia.org/wiki/Equal_temperament`).

Given the harmonics and their amplitudes, you can reconstruct the signal by adding up sinusoids. Next we'll see how.

1.3 Signals

I wrote a Python module called `thinkdsp` that contains classes and functions for working with signals and spectrums. [4] You can download it from `http://think-dsp.com/thinkdsp.py`.

To represent signals, `thinkdsp` provides a class called `Signal`, which is the parent class for several signal types, including `Sinusoid`, which represents both sine and cosine signals.

`thinkdsp` provides functions to create sine and cosine signals:

```
    cos_sig = thinkdsp.CosSignal(freq=440, amp=1.0, offset=0)
    sin_sig = thinkdsp.SinSignal(freq=880, amp=0.5, offset=0)
```

`freq` is frequency in Hz. `amp` is amplitude in unspecified units where 1.0 is generally the largest amplitude we can play.

`offset` is a **phase offset** in radians. Phase offset determines where in the period the signal starts (that is, when `t=0`). For example, a cosine signal with `offset=0` starts at $\cos 0$, which is 1. With `offset=pi/2` it starts at $\cos \pi/2$, which is 0. A sine signal with `offset=0` also starts at 0. In fact, a cosine signal with `offset=pi/2` is identical to a sine signal with `offset=0`.

Signals have an `__add__` method, so you can use the `+` operator to add them:

```
    mix = sin_sig + cos_sig
```

The result is a `SumSignal`, which represents the sum of two or more signals.

A Signal is basically a Python representation of a mathematical function. Most signals are defined for all values of `t`, from negative infinity to infinity.

You can't do much with a Signal until you evaluate it. In this context, "evaluate" means taking a sequence of `ts` and computing the corresponding values of the signal, which I call `ys`. I encapsulate `ts` and `ys` in an object called a Wave.

[4]In Latin the plural of "spectrum" is "spectra", but since I am not writing in Latin, I generally use standard English plurals.

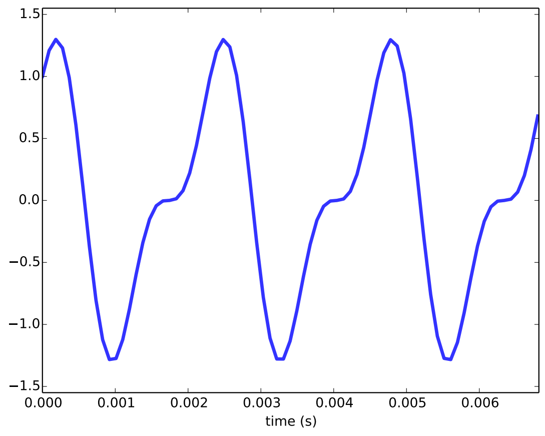

Figure 1.4: Segment from a mixture of two sinusoid signals.

A Wave represents a signal evaluated at a sequence of points in time. Each point in time is called a **frame** (a term borrowed from movies and video). The measurement itself is called a **sample**, although "frame" and "sample" are sometimes used interchangeably.

`Signal` provides `make_wave`, which returns a new Wave object:

```
wave = mix.make_wave(duration=0.5, start=0, framerate=11025)
```

`duration` is the length of the Wave in seconds. `start` is the start time, also in seconds. `framerate` is the (integer) number of frames per second, which is also the number of samples per second.

11,025 frames per second is one of several framerates commonly used in audio file formats, including Waveform Audio File (WAV) and mp3.

This example evaluates the signal from `t=0` to `t=0.5` at 5,513 equally-spaced frames (because 5,513 is half of 11,025). The time between frames, or **timestep**, is `1/11025` seconds, or 91 μs.

`Wave` provides a `plot` method that uses `pyplot`. You can plot the wave like this:

```
wave.plot()
pyplot.show()
```

`pyplot` is part of `matplotlib`; it is included in many Python distributions, or you might have to install it.

At `freq=440` there are 220 periods in 0.5 seconds, so this plot would look like a solid block of color. To zoom in on a small number of periods, we can use `segment`, which copies a segment of a Wave and returns a new wave:

```
    period = mix.period
    segment = wave.segment(start=0, duration=period*3)
```

period is a property of a Signal; it returns the period in seconds.

start and duration are in seconds. This example copies the first three periods from mix. The result is a Wave object.

If we plot segment, it looks like Figure 1.4. This signal contains two frequency components, so it is more complicated than the signal from the tuning fork, but less complicated than the violin.

1.4 Reading and writing Waves

thinkdsp provides read_wave, which reads a WAV file and returns a Wave:

```
    violin_wave = thinkdsp.read_wave('violin1.wav')
```

And Wave provides write, which writes a WAV file:

```
    wave.write(filename='example1.wav')
```

You can listen to the Wave with any media player that plays WAV files. On UNIX systems, I use aplay, which is simple, robust, and included in many Linux distributions.

thinkdsp also provides play_wave, which runs the media player as a subprocess:

```
    thinkdsp.play_wave(filename='example1.wav', player='aplay')
```

It uses aplay by default, but you can provide another player.

1.5 Spectrums

Wave provides make_spectrum, which returns a Spectrum:

```
    spectrum = wave.make_spectrum()
```

And Spectrum provides plot:

```
    spectrum.plot()
    thinkplot.show()
```

thinkplot is a module I wrote to provide wrappers around some of the functions in pyplot. You can download it from http://think-dsp.com/thinkplot.py. It is also included in the Git repository for this book (see Section 0.1).

Spectrum provides three methods that modify the spectrum:

- `low_pass` applies a low-pass filter, which means that components above a given cutoff frequency are attenuated (that is, reduced in magnitude) by a factor.
- `high_pass` applies a high-pass filter, which means that it attenuates components below the cutoff.
- `band_stop` attenuates components in the band of frequencies between two cutoffs.

This example attenuates all frequencies above 600 by 99%:

```
spectrum.low_pass(cutoff=600, factor=0.01)
```

Finally, you can convert a Spectrum back to a Wave:

```
wave = spectrum.make_wave()
```

At this point you know how to use many of the classes and functions in `thinkdsp`, and you are ready to do the exercises at the end of the chapter. In Chapter 2.4 I explain more about how these classes are implemented.

1.6 Exercises

Before you begin this exercises, you should download the code for this book, following the instructions in Section 0.1.

Exercise 1.1 If you have IPython, load `chap01.ipynb`, read through it, and run the examples. You can also view this notebook at `http://tinyurl.com/thinkdsp01`.

Go to `http://freesound.org` and download a sound sample that include music, speech, or other sounds that have a well-defined pitch. Select a segment with duration 0.5 to 2 seconds where the pitch is constant. Compute and plot the spectrum of the segment you selected. What connection can you make between the timbre of the sound and the harmonic structure you see in the spectrum?

Use `high_pass`, `low_pass`, and `band_stop` to filter out some of the harmonics. Then convert the spectrum back to a wave and listen to it. How does the sound relate to the changes you made in the spectrum?

Exercise 1.2 Synthesize a wave by creating a spectrum with arbitrary harmonics, inverting it, and listening. What happens as you add frequency components that are not multiples of the fundamental?

Exercise 1.3 This exercise asks you to write a function that simulates the effect of sound transmission underwater. This is a more open-ended exercise for ambitious readers. It uses decibels, which you can read about at `http://en.wikipedia.org/wiki/Decibel`.

First some background information: when sound travels through water, high frequency components are absorbed more than low frequency components. In pure water, the absorption rate, expressed in decibels per kilometer (dB/km), is proportional to frequency squared.

For example, if the absorption rate for frequency f is 1 dB/km, we expect the absorption rate for $2f$ to be 4 dB/km. In other words, doubling the frequency quadruples the absorption rate.

Over a distance of 10 kilometers, the f component would be attenuated by 10 dB, which corresponds to a factor of 10 in power, or a factor of 3.162 in amplitude.

Over the same distance, the $2f$ component would be attenuated by 40 dB, or a factor or 100 in amplitude.

Write a function that takes a Wave and returns a new Wave that contains the same frequency components as the original, but where each component is attenuated according to the absorption rate of water. Apply this function to the violin recording to see what a violin would sound like under water.

For more about the physics of sound transmission in water, see "Underlying physics and mechanisms for the absorption of sound in seawater" at `http://resource.npl.co.uk/acoustics/techguides/seaabsorption/physics.html`

Chapter 2

Harmonics

The code for this chapter is in `aliasing.py`, which is in the repository for this book (see Section 0.1).

2.1 Implementing Signals and Spectrums

If you have done the exercises, you know how to use the classes and methods in `thinkdsp`. Now let's see how they work.

We'll start with `CosSignal` and SinSignal:

```
def CosSignal(freq=440, amp=1.0, offset=0):
    return Sinusoid(freq, amp, offset, func=numpy.cos)

def SinSignal(freq=440, amp=1.0, offset=0):
    return Sinusoid(freq, amp, offset, func=numpy.sin)
```

These functions are just wrappers for `Sinusoid`, which is a kind of Signal:

```
class Sinusoid(Signal):

    def __init__(self, freq=440, amp=1.0, offset=0, func=numpy.sin):
        Signal.__init__(self)
        self.freq = freq
        self.amp = amp
        self.offset = offset
        self.func = func
```

The parameters of `__init__` are:

- freq: frequency in cycles per second, or Hz.
- amp: amplitude. The units of amplitude are arbitrary, usually chosen so 1.0 corresponds to the maximum input from a microphone or maximum output to a speaker.
- offset: where in its period the signal starts, at $t = 0$. offset is in units of radians, for reasons I explain below.
- func: a Python function used to evaluate the signal at a particular point in time. It is usually either numpy.sin or numpy.cos, yielding a sine or cosine signal.

Like many __init__ methods, this one just tucks the parameters away for future use.

The parent class of Sinusoid, Signal, provides make_wave:

```
    def make_wave(self, duration=1, start=0, framerate=11025):
        dt = 1.0 / framerate
        ts = numpy.arange(start, duration, dt)
        ys = self.evaluate(ts)
        return Wave(ys, framerate)
```

start and duration are the start time and duration in seconds. framerate is the number of frames (samples) per second.

dt is the time between samples, and ts is the sequence of sample times.

make_wave invokes evaluate, which has to be provided by a child class of Signal, in this case Sinusoid.

evaluate takes the sequence of sample times and returns an array of corresponding quantities:

```
    def evaluate(self, ts):
        phases = PI2 * self.freq * ts + self.offset
        ys = self.amp * self.func(phases)
        return ys
```

PI2 is a constant set to 2π.

ts and ys are NumPy arrays. I use NumPy and SciPy throughout the book. If you are familiar with these libraries, that's great, but I will also explain as we go along.

Let's unwind this function one step at time:

1. `self.freq` is frequency in cycles per second, and each element of `ts` a time in seconds, so their product is the number of cycles since the start time.

2. `PI2` is a constant that stores 2π. Multiplying by `PI2` converts from cycles to **phase**. You can think of phase as "cycles since the start time" expressed in radians; each cycle is 2π radians.

3. `self.offset` is the phase, in radians, at the start time. It has the effect of shifting the signal left or right in time.

4. If `self.func` is `sin` or `cos`, the result is a value between -1 and $+1$.

5. Multiplying by `self.amp` yields a signal that ranges from `-self.amp` to `+self.amp`.

In math notation, `evaluate` is written like this:

$$A\cos(2\pi f t + \phi_0)$$

where A is amplitude, f is frequency, t is time, and ϕ_0 is the phase offset. It may seem like I wrote a lot of code to evaluate one simple function, but as we'll see, this code provides a framework for dealing with all kinds of signals, not just sinusoids.

2.2 Computing the spectrum

Given a Signal, we can compute a Wave. Given a Wave, we can compute a Spectrum. `Wave` provides `make_spectrum`, which returns a new `Spectrum` object.

```
    def make_spectrum(self):
        hs = numpy.fft.rfft(self.ys)
        return Spectrum(hs, self.framerate)
```

`make_spectrum` uses `rfft`, which computes the discrete Fourier transform using an algorithm called **Fast Fourier Transform** or FFT.

The result of `rfft` is a sequence of complex numbers, `hs`, which is stored in a Spectrum.

There are two ways to think about complex numbers:

- A complex number is the sum of a real part and an imaginary part, often written $x + iy$, where i is the imaginary unit, $\sqrt{-1}$. You can think of x and y as Cartesian coordinates.
- A complex number is also the product of a magnitude and a complex exponential, $Ae^{i\phi}$, where A is the **magnitude** and ϕ is the **angle** in radians, also called the "argument". You can think of A and ϕ as polar coordinates.

Each element of `hs` corresponds to a frequency component. The magnitude of each element is proportional to the amplitude of the corresponding component. The angle of each element is the phase offset.

NumPy provides `absolute`, which computes the magnitude of a complex number, also called the "absolute value", and `angle`, which computes the angle.

Here is the definition of `Spectrum`:

```
class Spectrum(object):

    def __init__(self, hs, framerate):
        self.hs = hs
        self.framerate = framerate

        n = len(hs)
        f_max = framerate / 2.0
        self.fs = numpy.linspace(0, f_max, n)

        self.amps = numpy.absolute(self.hs)
```

Again, `hs` is the result of the FFT and `framerate` is the number of frames per second.

The elements of `hs` correspond to a sequence of frequencies, `fs`, equally spaced from 0 to the maximum frequency, `f_max`. The maximum frequency is `framerate/2`, for reasons we'll see soon.

Finally, `amps` contains the magnitude of `hs`, which is proportional to the amplitude of the components.

`Spectrum` also provides `plot`, which plots the magnitude for each frequency:

```
    def plot(self, low=0, high=None):
        thinkplot.plot(self.fs[low:high], self.amps[low:high])
```

`low` and `high` specify the slice of the Spectrum that should be plotted.

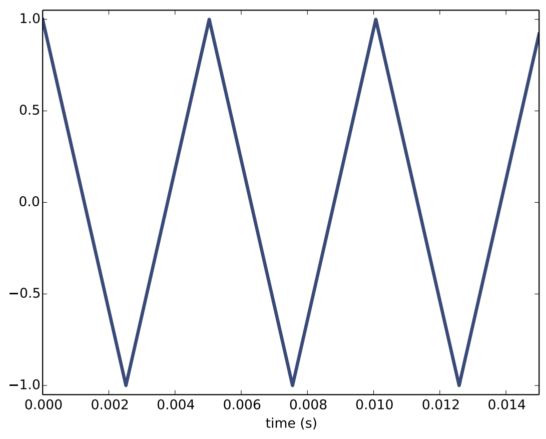

Figure 2.1: Segment of a triangle signal at 200 Hz.

2.3 Other waveforms

A sinusoid contains only one frequency component, so its DFT has only one peak. More complicated waveforms, like the violin recording, yield DFTs with many peaks. In this section we investigate the relationship between waveforms and their DFTs.

I'll start with a triangle waveform, which is like a straight-line version of a sinusoid. Figure 2.1 shows a triangle waveform with frequency 200 Hz.

To generate a triangle wave, you can use `thinkdsp.TriangleSignal`:

```
class TriangleSignal(Sinusoid):

    def evaluate(self, ts):
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = numpy.modf(cycles)
        ys = numpy.abs(frac - 0.5)
        ys = normalize(unbias(ys), self.amp)
        return ys
```

`TriangleSignal` inherits `__init__` from `Sinusoid`, so it takes the same arguments: `freq`, `amp`, and `offset`.

The only difference is `evaluate`. As we saw before, `ts` is the sequence of sample times where we want to evaluate the signal.

There are lots of ways to generate a triangle wave. The details are not important, but here's how `evaluate` works:

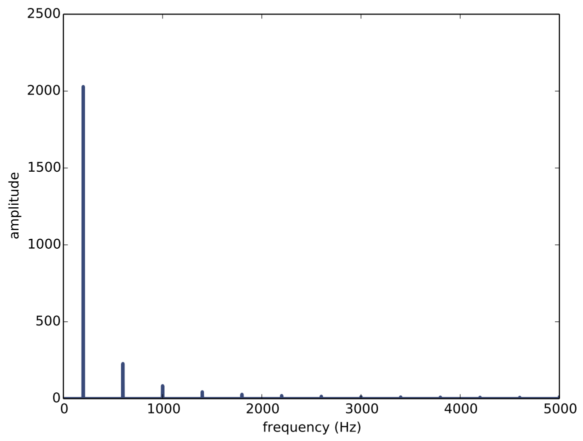

Figure 2.2: Spectrum of a triangle signal at 200 Hz.

1. `cycles` is the number of cycles since the start time. `numpy.modf` splits the number of cycles into the fraction part, stored in `frac`, and the integer part, which is ignored. [1]

2. `frac` is a sequence that ramps from 0 to 1 with the given frequency. Subtracting 0.5 yields values between -0.5 and 0.5. Taking the absolute value yields a waveform that zig-zags between 0.5 and 0.

3. `unbias` shifts the waveform down so it is centered at 0, then `normalize` scales it to the given amplitude, `amp`.

Here's the code that generates Figure 2.1:

```
signal = thinkdsp.TriangleSignal(200)
duration = signal.period * 3
segment = signal.make_wave(duration, framerate=10000)
segment.plot()
```

2.4 Harmonics

Next we can compute the spectrum of this waveform:

```
wave = signal.make_wave(duration=0.5, framerate=10000)
spectrum = wave.make_spectrum()
spectrum.plot()
```

[1]Using an underscore as a variable name is a convention that means, "I don't intend to use this value".

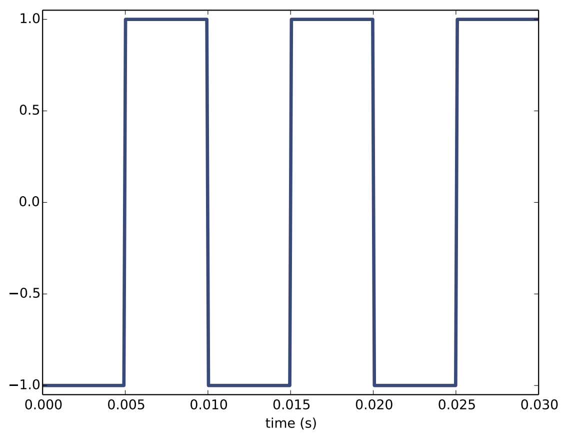

Figure 2.3: Segment of a square signal at 100 Hz.

Figure 2.2 shows the result. As expected, the highest peak is at the fundamental frequency, 200 Hz, and there are additional peaks at harmonic frequencies, which are integer multiples of 200.

But one surprise is that there are no peaks at the even multiples: 400, 800, etc. The harmonics of a triangle wave are all odd multiples of the fundamental frequency, in this example 600, 1000, 1400, etc.

Another feature of this spectrum is the relationship between the amplitude and frequency of the harmonics. The amplitude of the harmonics drops off in proportion to frequency squared. For example the frequency ratio of the first two harmonics (200 and 600 Hz) is 3, and the amplitude ratio is approximately 9. The frequency ratio of the next two harmonics (600 and 1000 Hz) is 1.7, and the amplitude ratio is approximately $1.7^2 = 2.9$.

`thinkdsp` also provides `SquareSignal`, which represents a square signal. Here's the class definition:

```
class SquareSignal(Sinusoid):

    def evaluate(self, ts):
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = numpy.modf(cycles)
        ys = self.amp * numpy.sign(unbias(frac))
        return ys
```

Like `TriangleSignal`, `SquareSignal` inherits `__init__` from `Sinusoid`, so it takes the same parameters.

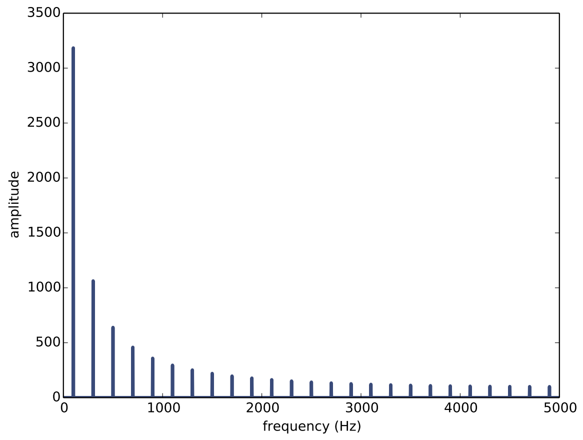

Figure 2.4: Spectrum of a square signal at 100 Hz.

And the `evaluate` method is similar. Again, `cycles` is the number of cycles since the start time, and `frac` is the fractional part, which ramps from 0 to 1 each period.

`unbias` shifts `frac` so it ramps from -0.5 to 0.5, then `numpy.sign` maps the negative values to -1 and the positive values to 1. Multiplying by `amp` yields a square wave that jumps between `-amp` and `amp`.

Figure 2.3 shows three periods of a square wave with frequency 100 Hz, and Figure 2.4 shows its spectrum.

Like a triangle wave, the square wave contains only odd harmonics, which is why there are peaks at 300, 500, and 700 Hz, etc. But the amplitude of the harmonics drops off more slowly. Specifically, amplitude drops in proportion to frequency (not frequency squared).

2.5 Aliasing

I have a confession. I chose the examples in the previous section carefully, to avoid showing you something confusing. But now it's time to get confused.

Figure 2.5 shows the spectrum of a triangle wave at 1100 Hz, sampled at 10,000 frames per second.

As expected, there are peaks at 1100 and 3300 Hz, but the third peak is at 4500, not 5500 Hz as expected. There is a small fourth peak at 2300, not 7700

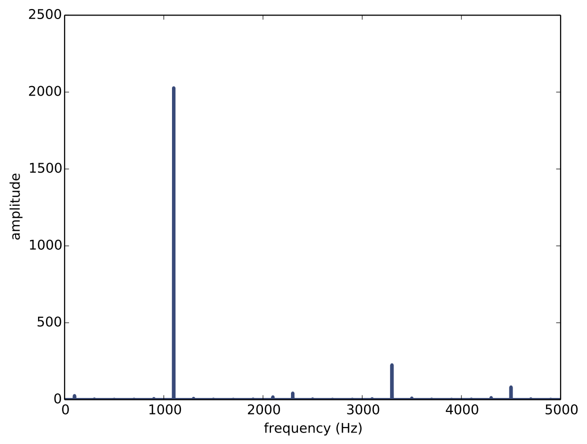

Figure 2.5: Spectrum of a triangle signal at 1100 Hz sampled at 10,000 frames per second.

Hz. And if you look very closely, the peak that should be at 9900 is actually at 100 Hz. What's going on?

The problem is that when you evaluate the signal at discrete points in time, you lose information about what happens between samples. For low frequency components, that's not a problem, because you have lots of samples per period.

But if you sample a signal at 5000 Hz with 10,000 frames per second, you only have two samples per period. That's enough to measure the frequency (it turns out), but it doesn't tell you much about the shape of the signal.

If the frequency is higher, like the 5500 Hz component of the triangle wave, things are worse: you don't even get the frequency right.

To see why, let's generate cosine signals at 4500 and 5500 Hz, and sample them at 10,000 frames per second:

```
framerate = 10000

signal = thinkdsp.CosSignal(4500)
duration = signal.period*5
segment = signal.make_wave(duration, framerate=framerate)
segment.plot()

signal = thinkdsp.CosSignal(5500)
segment = signal.make_wave(duration, framerate=framerate)
segment.plot()
```

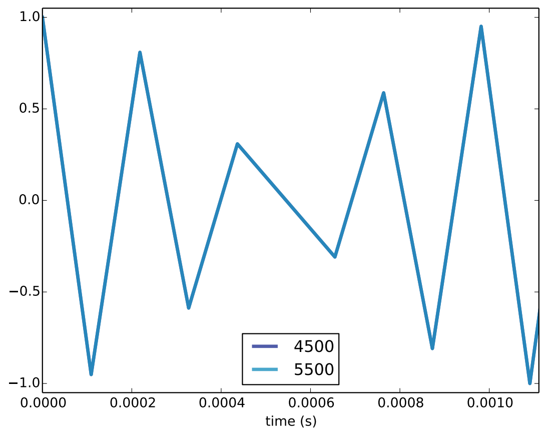

Figure 2.6: Cosine signals at 4500 and 5500 Hz, sampled at 10,000 frames per second. They are identical.

Figure 2.6 shows the result. The sampled waveform doesn't look very much like a sinusoid, but the bigger problem is that the two waveforms are exactly the same!

When we sample a 5500 Hz signal at 10,000 frames per second, the result is indistinguishable from a 4500 Hz signal.

For the same reason, a 7700 Hz signal is indistinguishable from 2300 Hz, and a 9900 Hz is indistinguishable from 100 Hz.

This effect is called **aliasing** because when the high frequency signal is sampled, it disguises itself as a low frequency signal.

In this example, the highest frequency we can measure is 5000 Hz, which is half the sampling rate. Frequencies above 5000 Hz are folded back below 5000 Hz, which is why this threshold is sometimes called the "folding frequency", but more often it is called the **Nyquist frequency**. See `http://en.wikipedia.org/wiki/Nyquist_frequency`.

The folding pattern continues if the aliased frequency goes below zero. For example, the 5th harmonic of the 1100 Hz triangle wave is at 12,100 Hz. Folded at 5000 Hz, it would appear at -2100 Hz, but it gets folded again at 0 Hz, back to 2100 Hz. In fact, you can see a small peak at 2100 Hz in Figure 2.4, and the next one at 4300 Hz.

2.6 Exercises

Exercise 2.1 If you use IPython, load `chap02.ipynb` and try out the examples. You can also view the notebook at `http://tinyurl.com/thinkdsp02`.

Exercise 2.2 A sawtooth signal has a waveform that ramps up linearly from -1 to 1, then drops to -1 and repeats. See `http://en.wikipedia.org/wiki/Sawtooth_wave`

Write a class called `SawtoothSignal` that extends `Signal` and provides `evaluate` to evaluate a sawtooth signal.

Compute the spectrum of a sawtooth wave. How does the harmonic structure compare to triangle and square waves?

Exercise 2.3 Sample an 1100 Hz triangle at 10000 frames per second and listen to it. Can you hear the aliased harmonic? It might help if you play a sequence of notes with increasing pitch.

Exercise 2.4 Compute the spectrum of an 1100 Hz square wave sampled at 10 kHz, and compare it to the spectrum of a triangle wave in Figure 2.5

Exercise 2.5 The triangle and square waves have odd harmonics only; the sawtooth wave has both even and odd harmonics. The harmonics of the square and sawtooth waves drop off in proportion to $1/f$; the harmonics of the triangle wave drop off like $1/f^2$. Can you find a waveform that has even and odd harmonics that drop off like $1/f^2$?

Chapter 3

Non-periodic signals

The code for this chapter is in `chirp.py`, which is in the repository for this book (see Section 0.1).

3.1 Chirp

The signals we have worked with so far are periodic, which means that they repeat forever. It also means that the frequency components they contain do not change over time. In this chapter, we consider non-periodic signals, whose frequency components *do* change over time. In other words, pretty much all sound signals.

We'll start with a **chirp**, which is a signal with variable frequency. Here is a class that represents a chirp:

```
class Chirp(Signal):

    def __init__(self, start=440, end=880, amp=1.0):
        self.start = start
        self.end = end
        self.amp = amp
```

`start` and `end` are the frequencies, in Hz, at the start and end of the chirp. `amp` is amplitude.

In a linear chirp, the frequency increases linearly from `start` to `end`. Here is the function that evaluates the signal:

```
    def evaluate(self, ts):
        freqs = numpy.linspace(self.start, self.end, len(ts)-1)
        return self._evaluate(ts, freqs)
```

`ts` is the sequence of points in time where the signal should be evaluated. If the length of `ts` is n, you can think of it as a sequence of $n - 1$ intervals of time. To compute the frequency during each interval, we use `numpy.linspace`.

`_evaluate` is a private method[1] that does the rest of the math:

```
def _evaluate(self, ts, freqs):
    dts = numpy.diff(ts)
    dphis = PI2 * freqs * dts
    phases = numpy.cumsum(dphis)
    ys = self.amp * numpy.cos(phases)
    return ys
```

`numpy.diff` computes the difference between adjacent elements of `ts`, returning the length of each interval in seconds. In the usual case where the elements of `ts` are equally spaced, the `dts` are all the same.

The next step is to figure out how much the phase changes during each interval. Since we know the frequency and duration of each interval, the *change* in phase during each interval is `dphis = PI2 * freqs * dts`. In math notation:

$$\Delta\phi = 2\pi f(t)\Delta t$$

`numpy.cumsum` computes the cumulative sum, which you can think of as a simple kind of integration. The result is a NumPy array where the `i`th element contains the sum of the first `i` terms from `dphis`; that is, the total phase at the end of the `i`th interval. Finally, `numpy.cos` maps from phase to amplitude.

Here's the code that creates and plays a chirp from 220 to 880 Hz, which is two octaves from A3 to A5:

```
signal = thinkdsp.Chirp(start=220, end=880)
wave1 = signal.make_wave(duration=2)

filename = 'chirp.wav'
wave1.write(filename)
thinkdsp.play_wave(filename)
```

You can run this code in `chap03.ipynb` or view the notebook at `http://tinyurl.com/thinkdsp03`.

[1]Beginning a method name with an underscore makes it "private", indicating that it is not part of the API that should be used outside the class definition.

3.2 Exponential chirp

When you listen to this chirp, you might notice that the pitch rises quickly at first and then slows down. The chirp spans two octaves, but it only takes 2/3 s to span the first octave, and twice as long to span the second.

The reason is that our perception of pitch depends on the logarithm of frequency. As a result, the **interval** we hear between two notes depends on the *ratio* of their frequencies, not the difference. "Interval" is the musical term for the perceived difference between two pitches.

For example, an octave is an interval where the ratio of two pitches is 2. So the interval from 220 to 440 is one octave and the interval from 440 to 880 is also one octave. The difference in frequency is bigger, but the ratio is the same.

As a result, if frequency increases linearly, as in a linear chirp, the perceived pitch increases logarithmically.

If you want the perceived pitch to increase linearly, the frequency has to increase exponentially. A signal with that shape is called an **exponential chirp**.

Here's the code that makes one:

```
class ExpoChirp(Chirp):

    def evaluate(self, ts):
        start, end = math.log10(self.start), math.log10(self.end)
        freqs = numpy.logspace(start, end, len(ts)-1)
        return self._evaluate(ts, freqs)
```

Instead of `numpy.linspace`, this version of evaluate uses `numpy.logspace`, which creates a series of frequencies whose logarithms are equally spaced, which means that they increase exponentially.

That's it; everything else is the same as Chirp. Here's the code that makes one:

```
    signal = thinkdsp.ExpoChirp(start=220, end=880)
    wave1 = signal.make_wave(duration=2)

    filename = 'expo_chirp.wav'
    wave1.write(filename)
    thinkdsp.play_wave(filename)
```

Run this code and listen. If you have a musical ear, this might sound more like music than the linear chirp.

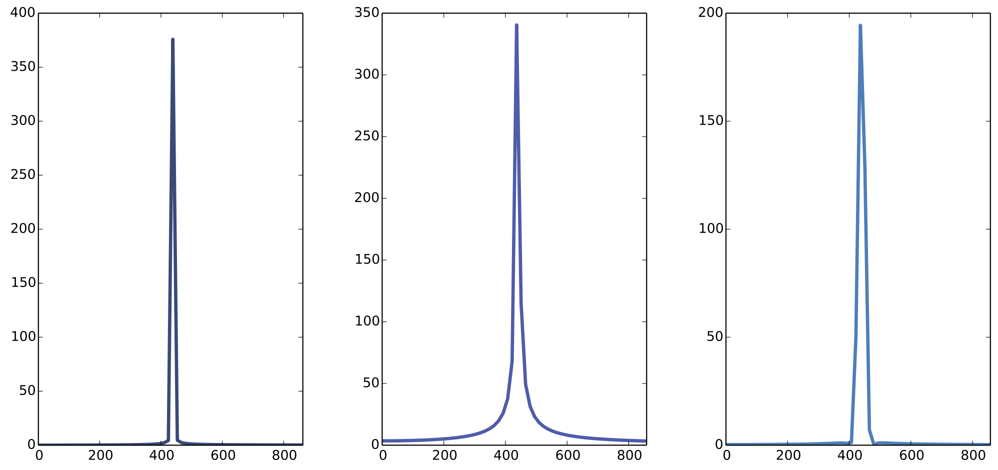

Figure 3.1: Spectrum of (a) a periodic segment of a sinusoid, (b) a non-periodic segment, (c) a tapered non-periodic segment.

3.3 Leakage

In previous chapters, we used the Fast Fourier Transform (FFT) to compute the spectrum of a wave. When we discuss how FFT works in Chapter 7, we will learn that it is based on the assumption that the signal is periodic. In theory, we should not use FFT on non-periodic signals. In practice it happens all the time, but there are a few things you have to be careful about.

One common problem is dealing with discontinuities at the beginning and end of a segment. Because FFT assumes that the signal is periodic, it implicitly connects the end of the segment back to the beginning to make a loop. If the end does not connect smoothly to the beginning, the discontinuity creates additional frequency components in the segment that are not in the signal.

As an example, let's start with a sine wave that contains only one frequency component at 440 Hz.

```
signal = thinkdsp.SinSignal(freq=440)
```

If we select a segment that happens to be an integer multiple of the period, the end of the segment connects smoothly with the beginning, and FFT behaves well.

```
duration = signal.period * 30
wave = signal.make_wave(duration)
spectrum = wave.make_spectrum()
```

Figure 3.1a shows the result. As expected, there is a single peak at 440 Hz.

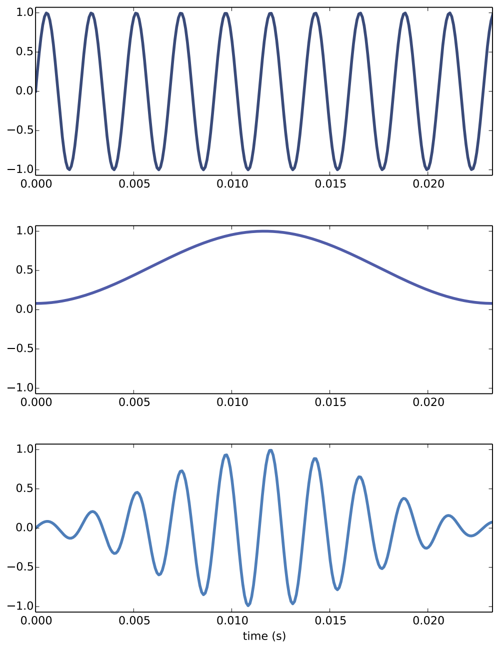

Figure 3.2: (a) Segment of a sinusoid, (b) Hamming window, (c) product of the segment and the window.

But if the duration is not a multiple of the period, bad things happen. With `duration = signal.period * 30.25`, the signal starts at 0 and ends at 1. Figure 3.1b shows the spectrum of this segment. Again, the peak is at 440 Hz, but now there are additional components spread out from 240 to 640 Hz. This spread is called "spectral leakage", because some of the energy that is actually at the fundamental frequency leaks into other frequencies.

In this example, leakage happens because we are using FFT on a segment that is not periodic.

3.4 Windowing

We can reduce leakage by smoothing out the discontinuity between the beginning and end of the segment, and one way to do that is **windowing**.

A "window" is a function designed to transform a non-periodic segment into something that can pass for periodic. Figure 3.2a shows a segment where the end does not connect smoothly to the beginning.

Figure 3.2b shows a "Hamming window", one of the more common window functions. No window function is perfect, but some can be shown to

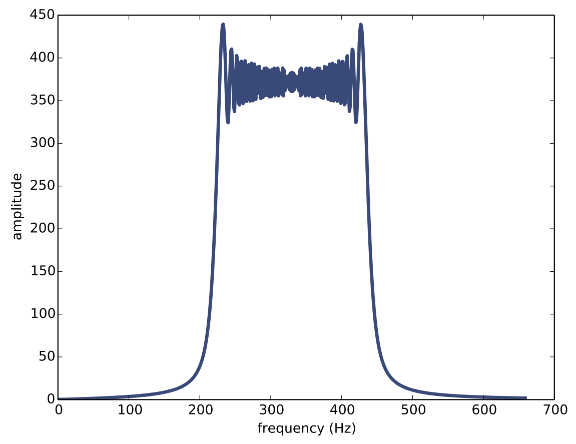

Figure 3.3: Spectrum of a one-second one-octave chirp.

be optimal for different applications, and Hamming is a commonly-used, all-purpose window.

Figure 3.2c shows the result of multiplying the window by the original signal. Where the window is close to 1, the signal is unchanged. Where the window is close to 0, the signal is attenuated. Because the window tapers at both ends, the end of the segment connects smoothly to the beginning.

Figure 3.1c shows the spectrum of the tapered signal. Windowing has reduced leakage substantially, but not completely.

Here's what the code looks like. `Wave` provides `hamming`, which applies a Hamming window:

```
def hamming(self):
    self.ys *= numpy.hamming(len(self.ys))
```

`numpy.hamming` computes the Hamming window with the given number of elements. NumPy provides functions to compute other window functions, including `bartlett`, `blackman`, `hanning`, and `kaiser`. One of the exercises at the end of this chapter asks you to experiment with these other windows.

3.5 Spectrum of a chirp

What do you think happens if you compute the spectrum of a chirp? Here's an example that constructs a one-second one-octave chirp and its spectrum:

```
signal = thinkdsp.Chirp(start=220, end=440)
```

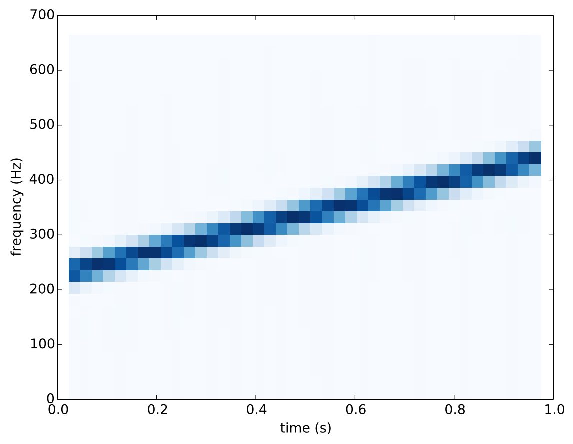

Figure 3.4: Spectrogram of a one-second one-octave chirp.

```
wave = signal.make_wave(duration=1)
spectrum = wave.make_spectrum()
```

Figure 3.3 shows the result. The spectrum shows components at every frequency from 220 to 440 Hz with variations, caused by leakage, that look a little like the Eye of Sauron (see `http://en.wikipedia.org/wiki/Sauron`).

The spectrum is approximately flat between 220 and 440 Hz, which indicates that the signal spends equal time at each frequency in this range. Based on that observation, you should be able to guess what the spectrum of an exponential chirp looks like.

The spectrum gives hints about the structure of the signal, but it obscures the relationship between frequency and time. For example, we cannot tell by looking at this spectrum whether the frequency went up or down, or both.

3.6 Spectrogram

To recover the relationship between frequency and time, we can break the chirp into segments and plot the spectrum of each segment. The result is called a **short-time Fourier transform** (STFT).

There are several ways to visualize a STFT, but the most common is a **spectrogram**, which shows time on the x-axis and frequency on the y-axis. Each column in the spectrogram shows the spectrum of a short segment, using color or grayscale to represent amplitude.

Wave provides make_spectrogram, which returns a Spectrogram object:

```
    signal = thinkdsp.Chirp(start=220, end=440)
    wave = signal.make_wave(duration=1)
    spectrogram = wave.make_spectrogram(seg_length=512)
    spectrogram.plot(high=32)
```

seg_length is the number of samples in each segment. I chose 512 because FFT is most efficient when the number of samples is a power of 2.

Figure 3.4 shows the result. The x-axis shows time from 0 to 1 seconds. The y-axis shows frequency from 0 to 700 Hz. I cut off the top part of the spectrogram; the full range goes to 5012.5 Hz, which is half of the framerate.

The spectrogram shows clearly that frequency increases linearly over time. Similarly, in the spectrogram of an exponential chirp, we can see the shape of the exponential curve.

However, notice that the peak in each column is blurred across 2–3 cells. This blurring reflects the limited resolution of the spectrogram.

3.7 The Gabor limit

The **time resolution** of the spectrogram is the duration of the segments, which corresponds to the width of the cells in the spectrogram. Since each segment is 512 frames, and there are 11,025 frames per second, there are 0.046 seconds per segment.

The **frequency resolution** is the frequency range between elements in the spectrum, which corresponds to the height of the cells. With 512 frames, we get 256 frequency components over a range from 0 to 5012.5 Hz, so the range between components is 21.5 Hz.

More generally, if n is the segment length, the spectrum contains $n/2$ components. If the framerate is r, the maximum frequency in the spectrum is $r/2$. So the time resolution is n/r and the frequency resolution is

$$\frac{r/2}{n/2}$$

which is r/n.

Ideally we would like time resolution to be small, so we can see rapid changes in frequency. And we would like frequency resolution to be small

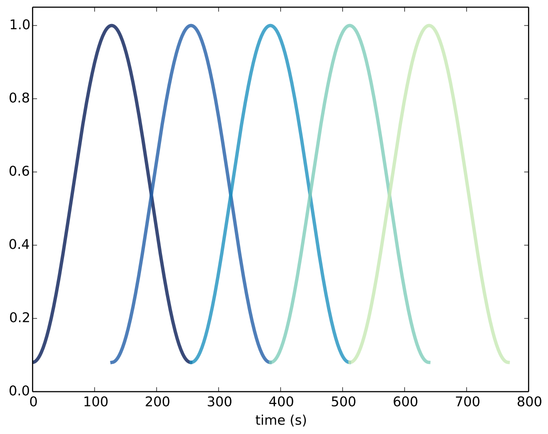

Figure 3.5: Overlapping Hamming windows.

so we can see small changes in frequency. But you can't have both. Notice that time resolution, n/r, is the inverse of frequency resolution, r/n. So if one gets smaller, the other gets bigger.

For example, if you double the segment length, you cut frequency resolution in half (which is good), but you double time resolution (which is bad). Even increasing the framerate doesn't help. You get more samples, but the range of frequencies increases at the same time.

This tradeoff is called the **Gabor limit** and it is a fundamental limitation of this kind of time-frequency analysis.

3.8 Implementing spectrograms

Here is the `Wave` method that computes spectrograms:

```
def make_spectrogram(self, seg_length):
    n = len(self.ys)
    window = numpy.hamming(seg_length)

    start, end, step = 0, seg_length, seg_length / 2
    spec_map = {}

    while end < n:
        ys = self.ys[start:end] * window
        hs = numpy.fft.rfft(ys)
```

```
            t = (start + end) / 2.0 / self.framerate
            spec_map[t] = Spectrum(hs, self.framerate)

            start += step
            end += step

        return Spectrogram(spec_map, seg_length)
```

`seg_length` is the number of samples in each segment. `n` is the number of samples in the wave. `window` is a Hamming window with the same length as the segments.

`start` and `end` are the slice indices that select the segments from the wave. `step` is the offset between segments. Since `step` is half of `seg_length`, the segments overlap by half. Figure 3.5 shows what these overlapping windows look like.

Inside the while loop, we select a slice from the wave, multiply by the window, and compute the FFT. Then we construct a Spectrum object and add it to `spec_map` which is a map from the midpoint of the segment in time to the Spectrum object.

Finally, the method constructs and returns a Spectrogram. Here is the definition of Spectrogram:

```
class Spectrogram(object):

    def __init__(self, spec_map, seg_length):
        self.spec_map = spec_map
        self.seg_length = seg_length
```

Like many `__init__` methods, this one just stores the parameters as attributes.

`Spectrogram` provides `plot`, which generates a pseudocolor plot:

```
    def plot(self, low=0, high=None):
        ts = self.times()
        fs = self.frequencies()[low:high]

        # copy amplitude from each spectrum into a column of the array
        size = len(fs), len(ts)
        array = numpy.zeros(size, dtype=numpy.float)

        # copy each spectrum into a column of the array
        for i, t in enumerate(ts):
            spectrum = self.spec_map[t]
            array[:,i] = spectrum.amps[low:high]

        thinkplot.pcolor(ts, fs, array)
```

`plot` uses `times`, which the returns the midpoint of each time segment in a sorted sequence, and `frequencies`, which returns the frequencies of the components in the spectrums.

`array` is a numpy array that holds the amplitudes from the spectrums, with one column for each point in time and one row for each frequency. The `for` loop iterates through the times and copies the amplitudes from each spectrum into a column of the array.

Finally `thinkplot.pcolor` is a wrapper around `pyplot.pcolor`, which generates the pseudocolor plot.

And that's how Spectrograms are implemented.

3.9 Exercises

Exercise 3.1 Run and listen to the examples in `chap03.ipynb`, which is in the repository for this book, and also available at `http://tinyurl.com/thinkdsp03`.

In the leakage example, try replacing the Hamming window with one of the other windows provided by NumPy, and see what effect they have on leakage. See `http://docs.scipy.org/doc/numpy/reference/routines.window.html`

Exercise 3.2 Write a class called `SawtoothChirp` that extends `Chirp` and overrides `evaluate` to generate a sawtooth waveform with frequency that increases (or decreases) linearly.

Hint: combine the evaluate functions from `Chirp` and `SawtoothSignal`.

Draw a sketch of what you think the spectrogram of this signal looks like, and then plot it. The effect of aliasing should be visually apparent, and if you listen carefully, you can hear it.

Exercise 3.3 Another way to generate a sawtooth chirp is to add up a harmonic series of sinusoidal chirps. Write another version of `SawtoothChirp` that uses this method and plot the spectrogram.

Exercise 3.4 In musical terminology, a "glissando" is a note that slides from one pitch to another, so it is similar to a chirp. A trombone player can play a glissando by extending the trombone slide while blowing continuously. As the slide extends, the total length of the tube gets longer, and the resulting pitch is inversely proportional to length.

Assuming that the player moves the slide at a constant speed, how does frequency vary with time? Is a trombone glissando more like a linear or exponential chirp?

Write a function that simulates a trombone glissando from C3 up to F3 and back down to C3. C3 is 262 Hz; F3 is 349 Hz.

Exercise 3.5 George Gershwin's *Rhapsody in Blue* starts with a famous clarinet glissando. Find a recording of this piece and plot a spectrogram of the first few seconds.

Exercise 3.6 Make or find a recording of a series of vowel sounds and look at the spectrogram. Can you identify different vowels?

Chapter 4

Noise

In English, "noise" means an unwanted or unpleasant sound. In the context of digital signal processing, it has two different senses:

1. As in English, it can mean an unwanted signal of any kind. If two signals interfere with each other, each signal would consider the other to be noise.
2. "Noise" also refers to a signal that contains components at many frequencies, so it lacks the harmonic structure of the periodic signals we saw in previous chapters.

This chapter is about the second kind.

The code for this chapter is in `noise.py`, which is in the repository for this book (see Section 0.1). You can listen to the examples in `chap04.ipynb`, which you can view at `http://tinyurl.com/thinkdsp04`.

4.1 Uncorrelated noise

The simplest way to understand noise is to generate it, and the simplest kind to generate is uncorrelated uniform noise (UU noise). "Uniform" means the signal contains random values from a uniform distribution; that is, every value in the range is equally likely. "Uncorrelated" means that the values are independent; that is, knowing one value provides no information about the others.

This code generates UU noise:

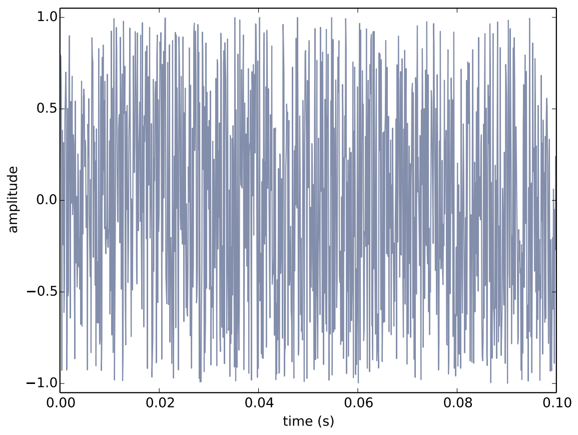

Figure 4.1: Waveform of uncorrelated uniform noise.

```
duration = 0.5
framerate = 11025
n = framerate * duration
ys = numpy.random.uniform(-1, 1, n)
wave = thinkdsp.Wave(ys, framerate)
wave.plot()
```

The result is a wave with duration 0.5 seconds at 11,025 samples per second. Each sample is drawn from a uniform distribution between -1 and 1.

If you play this wave, it sounds like the static you hear if you tune a radio between channels. Figure 4.1 shows what the waveform looks like. As expected, it looks pretty random.

Now let's take a look at the spectrum:

```
spectrum = wave.make_spectrum()
spectrum.plot_power()
```

`Spectrum.plot_power` is similar to `Spectrum.plot`, except that it plots power density instead of amplitude. Power density is the square of amplitude, expressed in units of power per Hz. (I am switching from amplitude to power in this section because it is more conventional in the context of noise.)

Figure 4.2 shows the result. Like the signal, the spectrum looks pretty random. And it is, but we have to be more precise about the word "random". There are at least three things we might like to know about a noise signal or its spectrum:

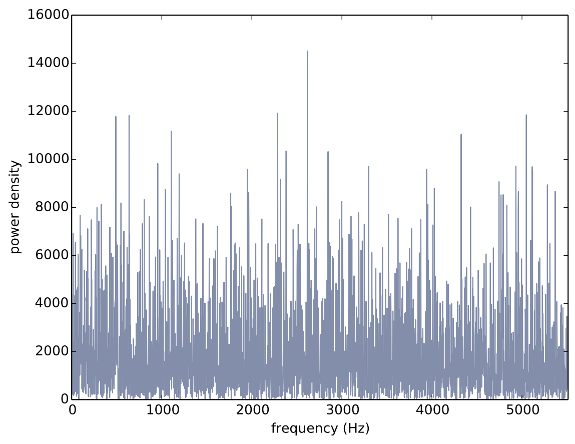

Figure 4.2: Spectrum of uncorrelated uniform noise.

- Distribution: The distribution of a random signal is the set of possible values and their probabilities. For example, in the uniform noise signal, the set of values is the range from -1 to 1, and all values have the same probability. An alternative is **Gaussian noise**, where the set of values is the range from negative to positive infinity, but values near 0 are the most likely, with probability that drops off according to the Gaussian or "bell" curve.

- Correlation: Is each value in the signal independent of the others, or are there dependencies between them? In UU noise, the values are independent. An alternative is **Brownian noise**, where each value is the sum of the previous value and a random "step". So if the value of the signal is high at a particular point in time, we expect it to stay high, and if it is low, we expect it to stay low.

- Relationship between power and frequency: In the spectrum of UU noise, the power at all frequencies is drawn from the same distribution; that is, there is no relationship between power and frequency (or, if you like, the relationship is a constant). An alternative is **pink noise**, where power is inversely related to frequency; that is, the power at frequency f is drawn from a distribution whose mean is proportional to $1/f$.

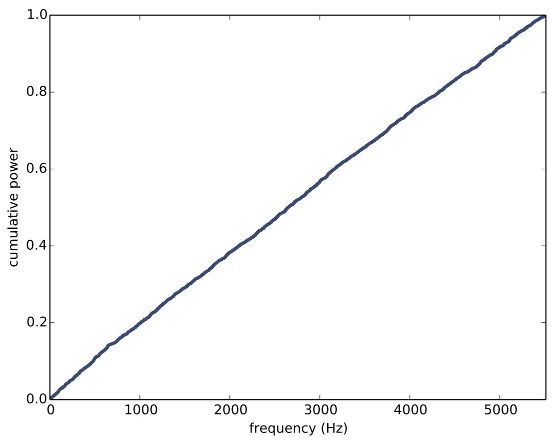

Figure 4.3: Integrated spectrum of uncorrelated uniform noise.

4.2 Integrated spectrum

For UU noise we can see the relationship between power and frequency more clearly by looking at the **integrated spectrum**, which is a function of frequency, f, that shows the cumulative total power in the spectrum up to f.

`Spectrum` provides a method that computes the IntegratedSpectrum:

```
    def make_integrated_spectrum(self):
        cs = numpy.cumsum(self.power)
        cs /= cs[-1]
        return IntegratedSpectrum(cs, self.fs)
```

`self.power` is a NumPy array containing power for each frequency. `numpy.cumsum` computes the cumulative sum of the powers. Dividing through by the last element normalizes the integrated spectrum so it runs from 0 to 1.

The result is an IntegratedSpectrum. Here is the class definition:

```
class IntegratedSpectrum(object):
    def __init__(self, cs, fs):
        self.cs = cs
        self.fs = fs
```

Like Spectrum, IntegratedSpectrum provides `plot_power`, so we can compute and plot the integrated spectrum like this:

```
    integ = spectrum.make_integrated_spectrum()
    integ.plot_power()
```

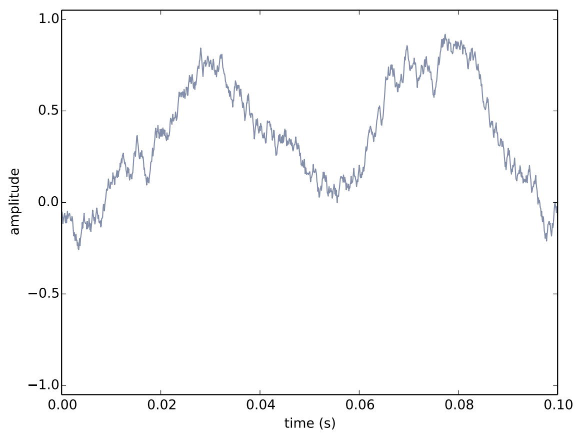

Figure 4.4: Waveform of Brownian noise.

```
    thinkplot.show(xlabel='frequency (Hz)',
                   ylabel='cumulative power')
```

The result, shown in Figure 4.3, is a straight line, which indicates that power at all frequencies is constant, on average. Noise with equal power at all frequencies is called **white noise** by analogy with light, because an equal mixture of light at all visible frequencies is white.

4.3 Brownian noise

UU noise is uncorrelated, which means that each value does not depend on the others. An alternative is Brownian noise, in which each value is the sum of the previous value and a random "step".

It is called "Brownian" by analogy with Brownian motion, in which a particle suspended in a fluid moves apparently at random, due to unseen interactions with the fluid. Brownian motion is often described using a **random walk**, which is a mathematical model of a path where the distance between steps is characterized by a random distribution.

In a one-dimensional random walk, the particle moves up or down by a random amount at each time step. The location of the particle at any point in time is the sum of all previous steps.

This observation suggests a way to generate Brownian noise: generate uncorrelated random steps and then add them up. Here is a class definition that implements this algorithm:

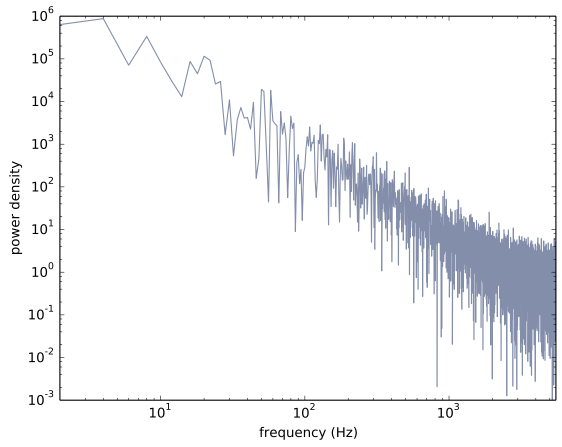

Figure 4.5: Spectrum of Brownian noise on a log-log scale.

```
class BrownianNoise(_Noise):

    def evaluate(self, ts):
        dys = numpy.random.uniform(-1, 1, len(ts))
        ys = numpy.cumsum(dys)
        ys = normalize(unbias(ys), self.amp)
        return ys
```

We use `numpy.random.uniform` to generate an uncorrelated signal and `numpy.cumsum` to compute their cumulative sum.

Since the sum is likely to escape the range from -1 to 1, we have to use `unbias` to shift the mean to 0, and `normalize` to get the desired maximum amplitude.

Here's the code that generates a BrownianNoise object and plots the waveform.

```
    signal = thinkdsp.BrownianNoise()
    wave = signal.make_wave(duration=0.5, framerate=11025)
    wave.plot()
```

Figure 4.4 shows the result. The waveform wanders up and down, but there is a clear correlation between successive values. When the amplitude is high, it tends to stay high, and vice versa.

If you plot the spectrum of Brownian noise, it doesn't look like much. Nearly all of the power is at the lowest frequencies; on a linear scale, the higher frequency components are not visible.

To see the shape of the spectrum, we have to plot power density and frequency on a log-log scale. Here's the code:

```
    spectrum = wave.make_spectrum()
    spectrum.plot_power(low=1, linewidth=1, alpha=0.5)
    thinkplot.show(xlabel='frequency (Hz)',
                   ylabel='power density',
                   xscale='log',
                   yscale='log')
```

The slope of this line is approximately -2 (we'll see why in Chapter 9), so we can write this relationship:

$$\log P = k - 2 \log f$$

where P is power, f is frequency, and k is the intercept of the line (which is not relevant for our purposes). Exponentiating both sides yields:

$$P = K/f^2$$

where K is e^k, but still not relevant. More important is that power is proportional to $1/f^2$, which is characteristic of Brownian noise.

Brownian noise is also called **red noise**, for the same reason that white noise is called "white". If you combine visible light with the same relationship between frequency and power, most of the power would be at the low-frequency end of the spectrum, which is red. Brownian noise is also sometimes called "brown noise", but I think that's confusing, so I won't use it.

4.4 Pink Noise

For red noise, the relationship between frequency and power is

$$P = K/f^2$$

There is nothing special about the exponent 2. More generally, we can synthesize noise with any exponent, β.

$$P = K/f^\beta$$

When $\beta = 0$, power is constant at all frequencies, so the result is white noise. When $\beta = 2$ the result is red noise.

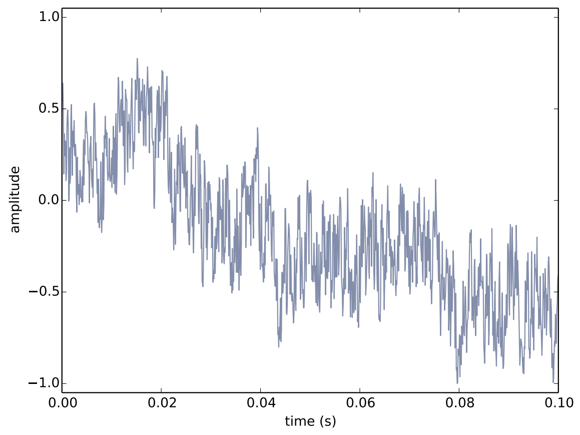

Figure 4.6: Waveform of pink noise with $\beta = 1$.

When β is between 0 and 2, the result is between white and red noise, so it is called "pink noise".

There are several ways to generate pink noise. The simplest is to generate white noise and then apply a low-pass filter with the desired exponent. `thinkdsp` provides a class that represents a pink noise signal:

```
class PinkNoise(_Noise):

    def __init__(self, amp=1.0, beta=1.0):
        self.amp = amp
        self.beta = beta
```

`amp` is the desired amplitude of the signal. `beta` is the desired exponent. `PinkNoise` provides `make_wave`, which generates a Wave.

```
    def make_wave(self, duration=1, start=0, framerate=11025):
        signal = UncorrelatedUniformNoise()
        wave = signal.make_wave(duration, start, framerate)
        spectrum = wave.make_spectrum()

        spectrum.pink_filter(beta=self.beta)

        wave2 = spectrum.make_wave()
        wave2.unbias()
        wave2.normalize()
        return wave2
```

`duration` is the length of the wave in seconds. `start` is the start time of the wave, which is included so that `make_wave` has the same interface for

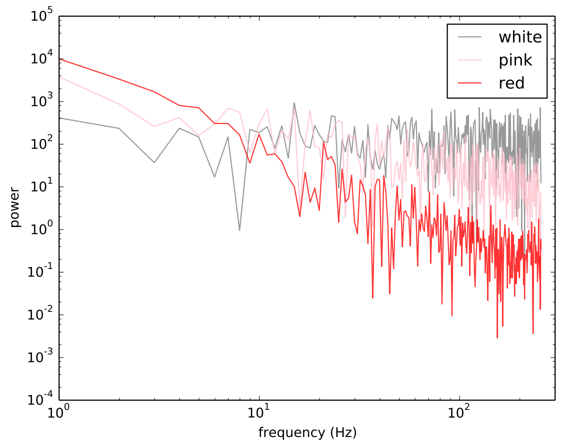

Figure 4.7: Spectrum of white, pink, and red noise on a log-log scale.

all types of noise, but for random noises, start time is irrelevant. And `framerate` is the number of samples per second.

`make_wave` creates a white noise wave, computes its spectrum, applies a filter with the desired exponent, and then converts the filtered spectrum back to a wave. Then it unbiases and normalizes the wave.

`Spectrum` provides `pink_filter`:

```
    def pink_filter(self, beta=1.0):
        denom = self.fs ** (beta/2.0)
        denom[0] = 1
        self.hs /= denom
```

`pink_filter` divides each element of the spectrum by $f^{\beta/2}$. Since power is the square of amplitude, this operation divides the power at each component by f^{β}. It treats the component at $f = 0$ as a special case, partly to avoid dividing by 0, but also because this element represents the bias of the signal, which we are going to set to 0 anyway.

Figure 4.6 shows the resulting waveform. Like Brownian noise, it wanders up and down in a way that suggests correlation between successive values, but at least visually, it looks more random. In the next chapter we will come back to this observation and I will be more precise about what I mean by "correlation" and "more random".

Finally, Figure 4.7 shows a spectrum for white, pink, and red noise on the same log-log scale. The relationship between the exponent, β, and the slope of the spectrum is apparent in this figure.

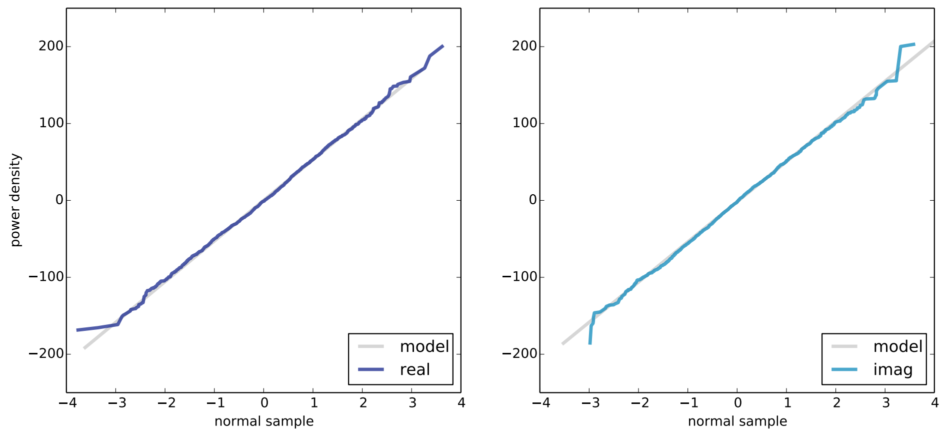

Figure 4.8: Normal probability plot for the real and imaginary parts of the spectrum of Gaussian noise.

4.5 Gaussian noise

We started with uncorrelated uniform (UU) noise and showed that, because its spectrum has equal power at all frequencies, on average, UU noise is white.

But when people talk about "white noise", they don't always mean UU noise. In fact, more often they mean uncorrelated Gaussian (UG) noise.

`thinkdsp` provides an implementation of UG noise:

```
class UncorrelatedGaussianNoise(_Noise):

    def evaluate(self, ts):
        ys = numpy.random.normal(0, self.amp, len(ts))
        return ys
```

`numpy.random.normal` returns a NumPy array of values from a Gaussian distribution, in this case with mean 0 and standard deviation `self.amp`. In theory the range of values is from negative to positive infinity, but we expect about 99% of the values to be between -3 and 3.

UG noise is similar in many ways to UU noise. The spectrum has equal power at all frequencies, on average, so UG is also white. And it has one other interesting property: the spectrum of UG noise is also UG noise. More precisely, the real and imaginary parts of the spectrum are uncorrelated Gaussian values.

To test that claim, we can generate the spectrum of UG noise and then generate a "normal probability plot", which is a graphical way to test whether a distribution is Gaussian.

```
    signal = thinkdsp.UncorrelatedGaussianNoise()
    wave = signal.make_wave(duration=0.5, framerate=11025)
    spectrum = wave.make_spectrum()

    thinkstats2.NormalProbabilityPlot(spectrum.real)
    thinkstats2.NormalProbabilityPlot(spectrum.imag)
```

`NormalProbabilityPlot` is provided by `thinkstats2`, which is included in the repository for this book. If you are not familiar with normal probability plots, you can read about them in *Think Stats* at `http://thinkstats2.com`.

Figure 4.8 shows the results. The gray lines show a linear model fit to the data; the dark lines show the data.

A straight line on a normal probability plot indicates that the data come from a Gaussian distribution. Except for some random variation at the extremes, these lines are straight, which indicates that the spectrum of UG noise is UG noise.

The spectrum of UU noise is also UG noise, at least approximately. In fact, by the Central Limit Theorem, the spectrum of almost any uncorrelated noise is approximately Gaussian, as long as the distribution has finite mean and standard deviation and the number of samples is large.

4.6 Exercises

Exercise 4.1 The algorithm in this chapter for generating pink noise is conceptually simple but computationally expensive. There are more efficient alternatives, like the Voss-McCartney algorithm. Research this method, implement it, compute the spectrum of the result, and confirm that it has the desired relationship between power and frequency.

Exercise 4.2 "A Soft Murmur" is a web site that plays a mixture of natural noise sources, including rain, waves, wind, etc. At `http://asoftmurmur.com/about/` you can find their list of recordings, most of which are at `http://freesound.org`.

Download a few of these files and compute the spectrum of each signal. Does the power spectrum look like white noise, pink noise, or Brownian noise? How does the spectrum vary over time?

Exercise 4.3 In a white noise signal, the mixture of frequencies changes over time. In the long run, we expect the power at all frequencies to be equal, but in any sample, the power at each frequency is random.

To estimate the long-term average power at each frequency, we can break a long signal into segments, compute the power spectral density for each segment, and then compute the average across the segments. You can read more about this algorithm at `http://en.wikipedia.org/wiki/Bartlett's_method`.

At `http://www.coindesk.com` you can download the daily price of a BitCoin as a CSV file. Read this file and compute the power spectrum of BitCoin prices as a function of time. Does it resemble white, pink, or Brownian noise?

Chapter 5

Autocorrelation

In the previous chapter I characterized white noise as "uncorrelated", which means that each value is independent of the others, and Brownian noise as "correlated", because each value depends on the preceding value. In this chapter I define these terms more precisely and present the **autocorrelation function**, which is a useful tool for signal analysis.

But before we get to autocorrelation, let's start with correlation.

The code for this chapter is in `autocorr.py`, which is in the repository for this book (see Section 0.1).

5.1 Correlation

In general, correlation between variables means that if you know the value of one, you have some information about the other. There are several ways to quantify correlation, but the most common is the Pearson product-moment correlation coefficient, usually denoted ρ. For two variables, x and y, that each contain N values:

$$\rho = \frac{\sum_i (x_i - \mu_x)(y_i - \mu_y)}{N \sigma_x \sigma_y}$$

Where μ_x and μ_y are the means of x and y, and σ_x and σ_y are their standard deviations.

Pearson's correlation is always between -1 and +1 (including both). If ρ is positive, we say that the correlation is positive, which means that when one

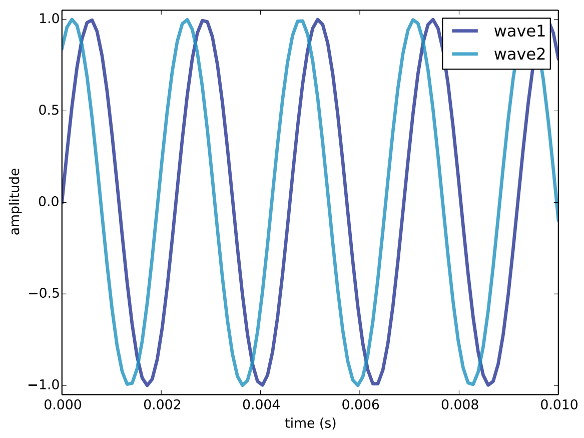

Figure 5.1: Two sine waves that differ by a phase offset of 1 radian; their coefficient of correlation is 0.54.

variable is high, the other tends to be high. If ρ is negative, the correlation is negative, so when one variable is high, the other is low.

The magnitude of ρ indicates the strength of the correlation. If ρ is 1 or -1, the variables are perfectly correlated, which means that if you know one, you can make a perfect prediction about the other.

If ρ is near zero, the correlation might be weak, or there might be a nonlinear relationship that is not captured by this coefficient of correlation. Nonlinear relationships are often important in statistics, but less often relevant for signal processing, so I won't say more about them here.

Python provides several ways to compute correlations. `numpy.corrcoef` takes any number of variables and computes a **correlation matrix** that includes correlations between each pair of variables.

I'll present an example with only two variables. First, I define a function that constructs sine waves with different phase offsets:

```
def make_wave(offset):
    signal = thinkdsp.SinSignal(freq=440, offset=offset)
    wave = signal.make_wave(duration=0.5, framerate=10000)
    return wave
```

Next I instantiate two waves with different offsets:

```
    wave1 = make_wave(offset=0)
    wave2 = make_wave(offset=1)
```

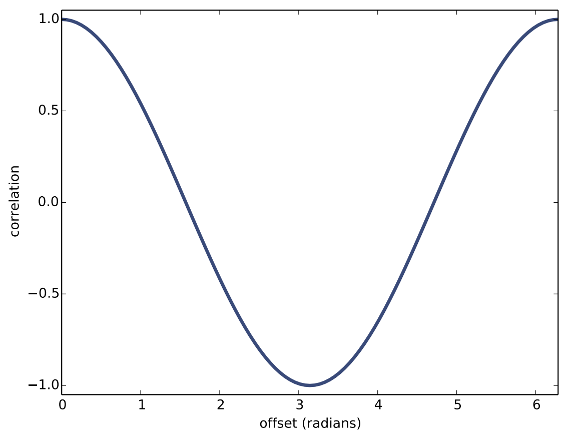

Figure 5.2: The correlation of two sine waves as a function of the phase offset between them. The result is a cosine.

Figure 5.1 shows what the first few periods of these waves look like. When one wave is high, the other is usually high, so we expect them to be correlated.

```
>>> corr_matrix = numpy.corrcoef(wave1.ys, wave2.ys, ddof=0)
[[ 1.    0.54]
 [ 0.54  1.  ]]
```

The option `ddof=0` indicates that `corrcoef` should divide by N rather than the default, $N-1$.

The result is a correlation matrix: the first element is the correlation of `wave1` with itself, which is always 1. Similarly, the last element is the correlation of `wave2` with itself.

The off-diagonal elements contain the value we're interested in, the correlation of `wave1` and `wave2`. The value 0.54 indicates that the strength of the correlation is moderate.

As the phase offset increases, this correlation decreases until the waves are 180 degrees out of phase, which yields correlation -1. Then it increases until the offset differs by 360 degrees. At that point we have come full circle and the correlation is 1.

Figure 5.2 shows the relationship between correlation and phase offset for a sine wave. The shape of that curve should look familiar; it is a cosine.

`thinkdsp` also provides a simple interface for computing the correlation between waves:

```
>>> wave1.corr(wave2)
0.54
```

5.2 Serial correlation

Signals often represent measurements of quantities that vary in time. For example, the sound signals we've worked with represent measurements of voltage (or current), which represent the changes in air pressure we perceive as sound.

Measurements like this almost always have serial correlation, which is the correlation between each element and the next (or the previous). To compute serial correlation, we can shift a signal and then compute the correlation of the shifted version with the original.

```
def corrcoef(xs, ys):
    return numpy.corrcoef(xs, ys, ddof=0)[0, 1]

def serial_corr(wave, lag=1):
    n = len(wave)
    y1 = wave.ys[lag:]
    y2 = wave.ys[:n-lag]
    corr = corrcoef(y1, y2)
    return corr
```

`corrcoef` is a convenience function that simplifies the interface to `numpy.corrcoef`.

`serial_corr` takes a Wave object and `lag`, which is the integer number of places to shift the waves.

We can test this function with the noise signals from the previous chapter. We expect UU noise to be uncorrelated, based on the way it's generated (not to mention the name):

```
signal = thinkdsp.UncorrelatedGaussianNoise()
wave = signal.make_wave(duration=0.5, framerate=11025)
serial_corr(wave)
```

When I ran this example, I got 0.006, which indicates a very small serial correlation. You might get a different value when you run it, but it should be comparably small.

In a Brownian noise signal, each value is the sum of the previous value and a random "step", so we expect a strong serial correlation:

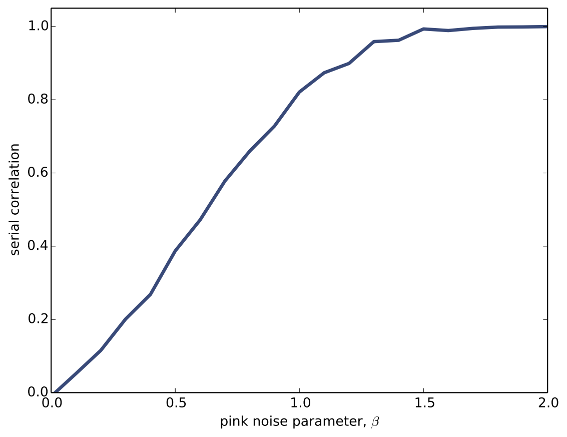

Figure 5.3: Serial correlation for pink noise with a range of parameters.

```
signal = thinkdsp.BrownianNoise()
wave = signal.make_wave(duration=0.5, framerate=11025)
serial_corr(wave)
```

Sure enough, the result I got is greater than 0.999.

Since pink noise is in some sense between Brownian noise and UU noise, we might expect an intermediate correlation:

```
signal = thinkdsp.PinkNoise(beta=1)
wave = signal.make_wave(duration=0.5, framerate=11025)
serial_corr(wave)
```

With parameter $\beta = 1$, I got a serial correlation of 0.851. As we vary the parameter from $\beta = 0$, which is uncorrelated noise, to $\beta = 2$, which is Brownian, serial correlation ranges from 0 to almost 1, as shown in Figure 5.3.

5.3 Autocorrelation

In the previous section we computed the correlation between each value and the next, so we shifted the elements of the array by 1. But if you noticed the `lag` parameter in `serial_corr`, you realize we can compute serial correlations with different lags.

You can think of `serial_corr` as a function that maps from each value of lag to the corresponding correlation, and we can evaluate that function by looping through values of `lag`:

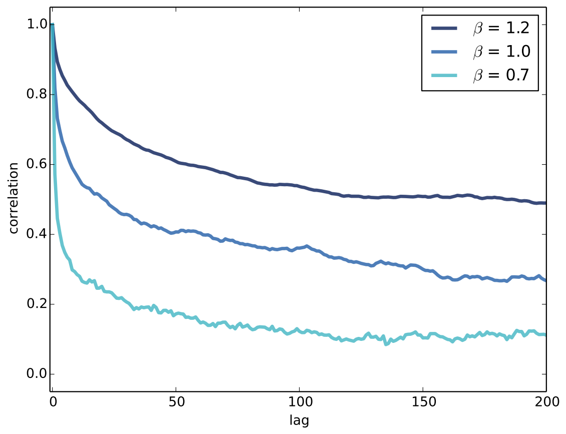

Figure 5.4: Autocorrelation functions for pink noise with a range of parameters.

```
def autocorr(wave):
    lags = range(len(wave.ys)//2)
    corrs = [serial_corr(wave, lag) for lag in lags]
    return lags, corrs
```

`autocorr` takes a Wave object and returns the autocorrelation function as a pair of sequences: `lags` is a sequence of integers from 0 to half the length of the wave; `corrs` is the sequence of serial correlations for each lag.

Figure 5.4 shows autocorrelation functions for pink noise with three values of β. For low values of β, the signal is less correlated, and the autocorrelation function drops toward zero relatively quickly. For larger values, serial correlation is stronger and drops off more slowly. With $\beta = 1.2$ serial correlation is strong even for long lags; this phenomenon is called **long-range dependence**, because it indicates that each value in the signal depends on many preceding values.

If you plot these autocorrelation functions on a log-x scale, they are approximately straight lines, which indicates that they are decreasing exponentially. An exponential autocorrelation function is one of the identifying characteristics of pink noise.

5.4 Autocorrelation of periodic signals

The autocorrelation of pink noise has interesting mathematical properties, but limited applications. The autocorrelation of periodic signals is more

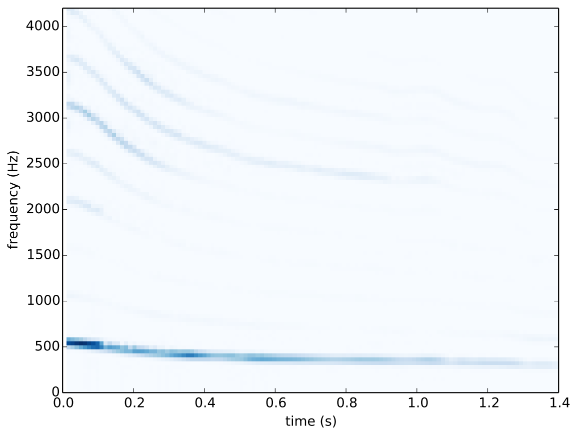

Figure 5.5: Spectrogram of a vocal chirp.

useful.

As an example, I downloaded from `freesound.org` a recording of someone singing a chirp; the repository for this book includes the file: `28042__bcjordan__voicedownbew.wav`. And you can use the IPython notebook for this chapter, `chap05.ipynb`, to play it.

Figure 5.5 shows the spectrogram of this wave. The fundamental frequency and some of the harmonics show up clearly. The chirp starts near 500 Hz and drops down to about 300 Hz, roughly from C5 to E4.

To estimate pitch at a particular point in time, we can take a short segment from the wave and plot its spectrum:

```
    duration = 0.01
    segment = wave.segment(start=0.2, duration=duration)
    spectrum = segment.make_spectrum()
    spectrum.plot(high=15)
```

This segment starts at 0.2 seconds and lasts 0.01 seconds. Figure 5.6 shows its spectrum. There is a clear peak near 400 Hz, but it is hard to identify the pitch precisely. The length of the segment is 441 samples at a framerate of 44100 Hz, so the frequency resolution is 100 Hz (see Section 3.7).

If we take a longer segment, we get better frequency resolution, but since the pitch is changing over time, we would see "motion blur"; that is, the peak would be spread between the start and end pitch of the segment.

We can estimate pitch more precisely using autocorrelation. If a signal is periodic, we expect the autocorrelation to spike when the lag equals the

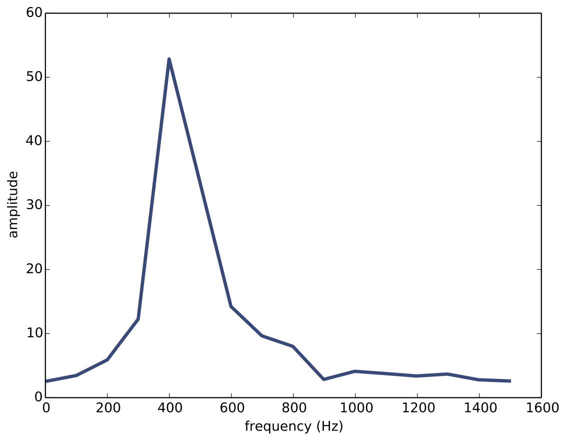

Figure 5.6: Spectrum of a segment from a vocal chirp.

period.

To show why that works, I'll plot two segments from the same recording.

```
def plot_shifted(wave, shift=0.0023, start=0.2):
    thinkplot.preplot(2)
    segment1 = wave.segment(start=start, duration=0.01)
    segment1.plot(linewidth=2, alpha=0.8)

    segment2 = wave.segment(start=start-shift, duration=0.01)
    segment2.plot(linewidth=2, alpha=0.4)

    thinkplot.config(xlabel='time (s)', ylim=[-1, 1])
```

One segment starts at 0.2 seconds; the other starts 0.0023 seconds later. Figure 5.7 shows the result. The segments are similar, and their correlation is 0.94. This result suggests that the period is near 0.0023 seconds, which corresponds to a frequency of 435 Hz.

In this example, I estimated the period by trial and error. To automate this process, we can use the autocorrelation function.

```
    lags, corrs = autocorr(segment)
    thinkplot.plot(lags, corrs)
```

Figure 5.8 shows the autocorrelation function for the segment starting at $t = 0.2$ seconds. The first peak occurs at `lag=101`. Then we can compute the frequency that corresponds to that period:

```
    period = lag / segment.framerate
    frequency = 1 / period
```

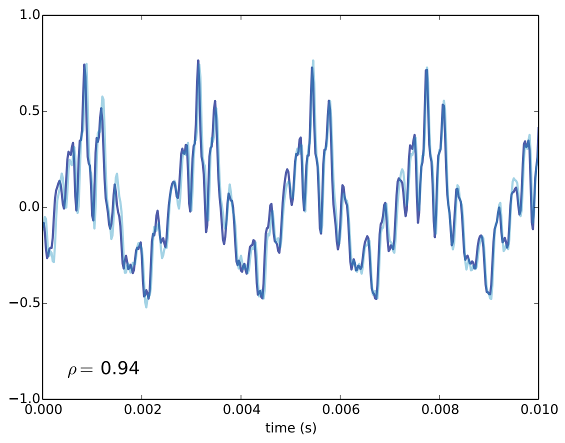

Figure 5.7: Two segments from a chirp, one starting 0.0023 seconds after the other.

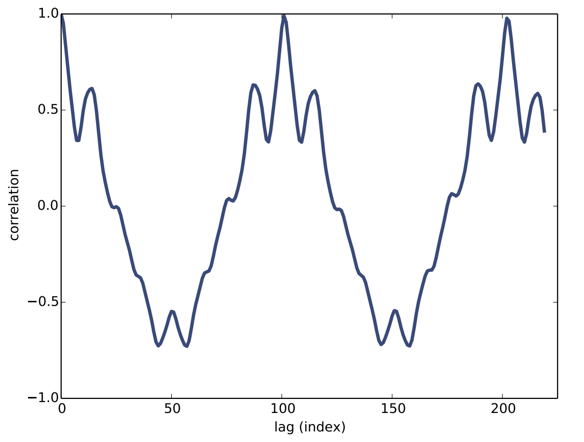

Figure 5.8: Autocorrelation function for a segment from a chirp.

The estimated fundamental frequency is 437 Hz. To evaluate the precision of the estimate, we can run the same computation with lags 100 and 102, which correspond to frequencies 432 and 441 Hz. So the frequency precision using autocorrelation is less than 10 Hz, compared with 100 Hz using the spectrum.

5.5 Correlation as dot product

I started this chapter with this definition of Pearson's correlation coefficient:

$$\rho = \frac{\sum_i (x_i - \mu_x)(y_i - \mu_y)}{N\sigma_x\sigma_y}$$

Then I used ρ to define serial correlation and autocorrelation. That's consistent with how these terms are used in statistics, but in the context of signal processing, the definitions are a little different.

In signal processing, we are often working with unbiased signals, where the mean is 0, and normalized signals, where the standard deviation is 1. In that case, the definition of ρ simplifies to:

$$\rho = \frac{1}{N} \sum_i x_i y_i$$

And it is common to simplify even further:

$$r = \sum_i x_i y_i$$

This definition of correlation is not "standardized", so it doesn't generally fall between -1 and 1. But it has other useful properties.

If you think of x and y as vectors, you might recognize this formula as the **dot product**, $x \cdot y$. See `http://en.wikipedia.org/wiki/Dot_product`.

The dot product indicates the degree to which the signals are similar. If they are normalized so their standard deviations are 1,

$$x \cdot y = \cos\theta$$

where θ is the angle between the vectors. And that explains why Figure 5.2 is a cosine curve.

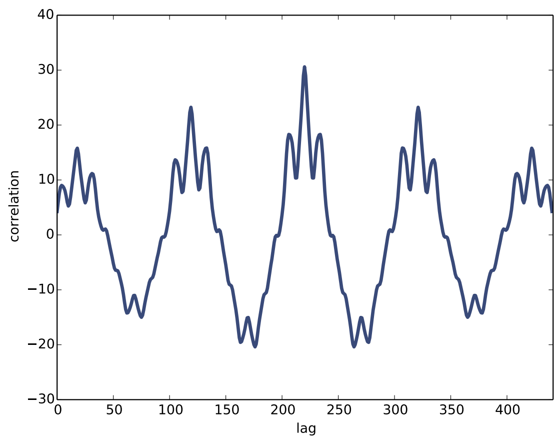

Figure 5.9: Autocorrelation function computed with `numpy.correlate`.

5.6 Using NumPy

NumPy provides a function, `correlate`, that computes the correlation of two functions or the autocorrelation of one function. We can use it to compute the autocorrelation of the segment from the previous section:

```
corrs2 = numpy.correlate(segment.ys, segment.ys, mode='same')
```

The option `mode` tells `correlate` what range of `lag` it should use. With the value `'same'`, the range is from $-N/2$ to $N/2$, where N is the length of the wave array.

Figure 5.9 shows the result. It is symmetric because the two signals are identical, so a negative lag on one has the same effect as a positive lag on the other. To compare with the results from `autocorr`, we can select the second half:

```
    N = len(corrs2)
    half = corrs2[N//2:]
```

If you compare Figure 5.9 to Figure 5.8, you'll notice that the correlations computed by `numpy.correlate` get smaller as the lags increase. That's because `numpy.correlate` uses the unstandardized definition of correlation; as the lag gets bigger, the overlap between the two signals gets smaller, so the magnitude of the correlations decreases.

We can correct that by dividing through by the lengths:

```
    lengths = range(N, N//2, -1)
    half /= lengths
```

Finally, we can standardize the results so the correlation with `lag=0` is 1.

```
    half /= half[0]
```

With these adjustments, the results computed by `autocorr` and `numpy.correlate` are nearly the same. They still differ by 1-2%. The reason not important, but if you are curious: `autocorr` standardizes the correlations independently for each lag; for `numpy.correlate`, we standardized them all at the end.

More importantly, now you know what autocorrelation is, how to use it to estimate the fundamental period of a signal, and two ways to compute it.

5.7 Exercises

Exercise 5.1 If you did the exercises in the previous chapter, you downloaded the historical price of BitCoins and estimated the power spectrum of the price changes. Using the same data, compute the autocorrelation of BitCoin prices. Does the autocorrelation function decay exponentially? Is there evidence of periodic behavior?

Exercise 5.2 Estimate the pitch of a segment from the saxophone recording.

Chapter 6

Discrete cosine transform

The code for this chapter is in `dct.py`, which is in the repository for this book (see Section 0.1).

The topic of this chapter is the **Discrete Cosine Transform** (DCT), which is used in MP3 and related formats for compressing music, JPEG and similar formats for images, and the MPEG family of formats for video.

DCT is similar in many ways to the discrete Fourier transform (DFT), which we have been using for spectral analysis. Once we learn how DCT works, it will be easier to explain DFT.

Here are the steps we'll follow to get there:

1. We'll start with the synthesis problem: given a set of frequency components and their amplitudes, how can we construct a waveform?

2. Next we'll rewrite the synthesis problem using NumPy arrays. This move is good for performance, and also provides insight for the next step.

3. We'll look at the analysis problem: given a signal and a set of frequencies, how can we find the amplitude of each frequency component? We'll start with a solution that is conceptually simple but slow.

4. Finally, we'll use some principles from linear algebra to find a more efficient algorithm. If you already know linear algebra, that's great, but I will explain what you need as we go.

Let's get started.

6.1 Synthesis

Suppose I give you a list of amplitudes and a list of frequencies, and ask you to construct a signal that is the sum of these frequency components. Using objects in the `thinkdsp` module, there is a simple way to perform this operation, which is called **synthesis**:

```
def synthesize1(amps, freqs, ts):
    components = [thinkdsp.CosSignal(freq, amp)
                  for amp, freq in zip(amps, freqs)]
    signal = thinkdsp.SumSignal(*components)

    ys = signal.evaluate(ts)
    return ys
```

`amps` is a list of amplitudes, `freqs` is the list of frequencies, and `ts` is the sequence of times where the signal should be evaluated.

`components` is a list of `CosSignal` objects, one for each amplitude-frequency pair. `SumSignal` represents the sum of these frequency components.

Finally, `evaluate` computes the value of the signal at each time in `ts`.

We can test this function like this:

```
    amps = numpy.array([0.6, 0.25, 0.1, 0.05])
    freqs = [100, 200, 300, 400]
    framerate = 11025

    ts = numpy.linspace(0, 1, framerate)
    ys = synthesize1(amps, freqs, ts)
    wave = thinkdsp.Wave(ys, framerate)
    wave.play()
```

This example makes a signal that contains a fundamental frequency at 100 Hz and three harmonics (100 Hz is a sharp G2). It renders the signal for one second at 11,025 frames per second and plays the resulting wave.

Conceptually, synthesis is pretty simple. But in this form it doesn't help much with **analysis**, which is the inverse problem: given the wave, how do we identify the frequency components and their amplitudes?

6.2 Synthesis with arrays

Here's another way to write `synthesize`:

$$\begin{array}{ccc}
 & \mathtt{M} & \begin{bmatrix}0.6\\0.25\\0.1\\0.05\end{bmatrix} = \mathtt{amps} \\
\begin{array}{c}\cdot\\\cdot\\t_k\\\cdot\\\cdot\end{array} & \begin{bmatrix}\cdot&\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot\\ a&b&c&d\\ \cdot&\cdot&\cdot&\cdot\\ \cdot&\cdot&\cdot&\cdot\end{bmatrix} & \begin{bmatrix}\cdot\\\cdot\\e\\\cdot\\\cdot\end{bmatrix} = \mathtt{ys} \\
 & \begin{array}{cccc}\cdot & f_j & \cdot & \cdot\end{array} &
\end{array}$$

Figure 6.1: Synthesis with arrays.

```
def synthesize2(amps, freqs, ts):
    args = numpy.outer(ts, freqs)
    M = numpy.cos(PI2 * args)
    ys = numpy.dot(M, amps)
    return ys
```

This function looks very different, but it does the same thing. Let's see how it works:

1. `numpy.outer` computes the outer product of `ts` and `freqs`. The result is an array with one row for each element of `ts` and one column for each element of `freqs`. Each element in the array is the product of a frequency and a time, ft.

2. We multiply `args` by 2π and apply `cos`, so each element of the result is $\cos(2\pi ft)$. Since the `ts` run down the columns, each column contains a cosine signal at a particular frequency, evaluated at a sequence of times.

3. `numpy.dot` multiplies each row of `M` by `amps`, element-wise, and then adds up the products. In terms of linear algebra, we are multiplying a matrix, `M`, by a vector, `amps`.

Figure 6.1 shows the structure of this computation. Each row of the matrix, `M`, corresponds to a time from 0.0 to 1.0 seconds; t_k is the time of the kth row. Each column corresponds to a frequency from 100 to 400 Hz; f_j is the frequency of the jth column.

I labeled the kth row with the letters a through d; as an example, the value of a is $\cos[2\pi(100)t_k]$.

The result of the dot product, ys, is a vector with one element for each row of M. The kth element, labeled e, is the sum of products:

$$e = 0.6a + 0.25b + 0.1c + 0.05d$$

And likewise with the other elements of ys. So each element of y is the sum of four frequency components, evaluated at a point in time, and multiplied by the corresponding amplitudes. And that's exactly what we wanted. We can use the code from the previous section to check that the two versions of synthesize produce the same results.

```
ys1 = synthesize1(amps, freqs, ts)
ys2 = synthesize2(amps, freqs, ts)
print max(abs(ys1 - ys2))
```

The biggest difference between ys1 and y2 is about 1e-13, which is what we expect due to floating-point errors.

Writing this computation in terms of linear algebra makes the code smaller and faster. Another benefit of linear algebra is that it provides concise notation for operations on matrices and vectors. For example, we could write synthesize like this:

$$\begin{aligned} M &= \cos(2\pi t \otimes f) \\ y &= Ma \end{aligned}$$

where a is a vector of amplitudes, t is a vector of times, f is a vector of frequencies, and $\otimes$ is the symbol for the outer product of two vectors.

6.3 Analysis

Now we are ready to solve the analysis problem. Suppose I give you a wave and tell you that it is the sum of cosines with a given set of frequencies. How would you find the amplitude for each frequency component? In other words, given ys, ts and freqs, can you recover amps?

In terms of linear algebra, the first step is the same as for synthesis: we compute $M = \cos(2\pi t \otimes f)$. Then we want to find a so that $y = Ma$; in other words, we want to solve a linear system. NumPy provides linalg.solve, which does exactly that.

Here's what the code looks like:

```
def analyze1(ys, freqs, ts):
    args = numpy.outer(ts, freqs)
    M = numpy.cos(PI2 * args)
    amps = numpy.linalg.solve(M, ys)
    return amps
```

The first two lines use `ts` and `freqs` to build the matrix, `M`. Then `numpy.linalg.solve` computes `amps`.

But there's a hitch. In general we can only solve a system of linear equations if the matrix is square; that is, the number of equations (rows) is the same as the number of unknowns (columns).

In this example, we have only 4 frequencies, but we evaluated the signal at 11,025 times. So we have many more equations than unknowns.

In general if `ys` contains more than 4 elements, it is unlikely that we can analyze it using only 4 frequencies.

But in this case, we know that the `ys` were actually generated by adding only 4 frequency components, so we can use any 4 values from the wave array to recover `amps`.

For simplicity, I'll use the first 4 samples from the signal. Using the values of `ys`, `freqs` and `ts` from the previous section, we can run `analyze1` like this:

```
    n = len(freqs)
    amps2 = analyze1(ys[:n], freqs, ts[:n])
    print(amps2)
```

And sure enough, we get

```
[ 0.6   0.25  0.1   0.05 ]
```

This algorithm works, but it is slow. Solving a linear system of equations takes time proportional to n^3, where n is the number of columns in M. We can do better.

6.4 Orthogonal matrices

One way to solve linear systems is by inverting matrices. The inverse of a matrix M is written M^{-1}, and it has the property that $M^{-1}M = I$. I is the identity matrix, which has the value 1 on all diagonal elements and 0 everywhere else.

So, to solve the equation $y = Ma$, we can multiply both sides by M^{-1}, which yields:

$$M^{-1}y = M^{-1}Ma$$

On the right side, we can replace $M^{-1}M$ with I:

$$M^{-1}y = Ia$$

If we multiply I by any vector a, the result is a, so

$$M^{-1}y = a$$

This implies that if we can compute M^{-1} efficiently, we can find a with a simple matrix multiplication (using `numpy.dot`). That takes time proportional to n^2, which is better than n^3.

In general inverting a matrix is slow, but some special cases are faster. In particular, if M is **orthogonal**, the inverse of M is just the transpose of M, written M^T. In NumPy transposing an array is a constant-time operation. It doesn't actually move the elements of the array; instead, it creates a "view" that changes the way the elements are accessed.

Again, a matrix is orthogonal if its transpose is also its inverse; that is, $M^T = M^{-1}$. That implies that $M^T M = I$, which means we can check whether a matrix is orthogonal by computing $M^T M$.

So let's see what the matrix looks like in `synthesize2`. In the previous example, M has 11,025 rows, so it might be a good idea to work with a smaller example:

```
def test1():
    amps = numpy.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    time_unit = 0.001
    ts = numpy.arange(N) / N * time_unit
    max_freq = N / time_unit / 2
    freqs = numpy.arange(N) / N * max_freq
    ys = synthesize2(amps, freqs, ts)
```

`amps` is the same vector of amplitudes we saw before. Since we have 4 frequency components, we'll sample the signal at 4 points in time. That way, M is square.

`ts` is a vector of equally spaced sample times in the range from 0 to 1 time unit. I chose the time unit to be 1 millisecond, but it is an arbitrary choice, and we will see in a minute that it drops out of the computation anyway.

Since the frame rate is N samples per time unit, the Nyquist frequency is `N / time_unit / 2`, which is 2000 Hz in this example. So `freqs` is a vector of equally spaced frequencies between 0 and 2000 Hz.

With these values of `ts` and `freqs`, the matrix, M, is:

```
[[ 1.      1.      1.      1.    ]
 [ 1.      0.707  0.     -0.707]
 [ 1.      0.     -1.     -0.    ]
 [ 1.     -0.707 -0.      0.707]]
```

You might recognize 0.707 as an approximation of $\sqrt{2}/2$, which is $\cos \pi/4$. You also might notice that this matrix is **symmetric**, which means that the element at (j,k) always equals the element at (k,j). This implies that M is its own transpose; that is, $M^T = M$.

But sadly, M is not orthogonal. If we compute $M^T M$, we get:

```
[[ 4.  1. -0.  1.]
 [ 1.  2.  1. -0.]
 [-0.  1.  2.  1.]
 [ 1. -0.  1.  2.]]
```

And that's not the identity matrix.

6.5 DCT-IV

But if we choose `ts` and `freqs` carefully, we can make M orthogonal. There are several ways to do it, which is why there are several versions of the discrete cosine transform (DCT).

One simple option is to shift `ts` and `freqs` by a half unit. This version is called DCT-IV, where "IV" is a roman numeral indicating that this is the fourth of eight versions of the DCT.

Here's an updated version of `test1`:

```
def test2():
    amps = numpy.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    ts = (0.5 + numpy.arange(N)) / N
    freqs = (0.5 + numpy.arange(N)) / 2
    ys = synthesize2(amps, freqs, ts)
```

If you compare this to the previous version, you'll notice two changes. First, I added 0.5 to `ts` and `freqs`. Second, I cancelled out `time_units`, which simplifies the expression for `freqs`.

With these values, M is

```
[[ 0.981  0.831  0.556  0.195]
 [ 0.831 -0.195 -0.981 -0.556]
 [ 0.556 -0.981  0.195  0.831]
 [ 0.195 -0.556  0.831 -0.981]]
```

And $M^T M$ is

```
[[ 2.  0.  0.  0.]
 [ 0.  2. -0.  0.]
 [ 0. -0.  2. -0.]
 [ 0.  0. -0.  2.]]
```

Some of the off-diagonal elements are displayed as -0, which means that the floating-point representation is a small negative number. So this matrix is very close to $2I$, which means M is almost orthogonal; it's just off by a factor of 2. And for our purposes, that's good enough.

Because M is symmetric and (almost) orthogonal, the inverse of M is just $M/2$. Now we can write a more efficient version of `analyze`:

```
def analyze2(ys, freqs, ts):
    args = numpy.outer(ts, freqs)
    M = numpy.cos(PI2 * args)
    amps = numpy.dot(M, ys) / 2
    return amps
```

Instead of using `numpy.linalg.solve`, we just multiply by $M/2$.

Combining `test2` and `analyze2`, we can write an implementation of DCT-IV:

```
def dct_iv(ys):
    N = len(ys)
    ts = (0.5 + numpy.arange(N)) / N
    freqs = (0.5 + numpy.arange(N)) / 2
    args = numpy.outer(ts, freqs)
    M = numpy.cos(PI2 * args)
    amps = numpy.dot(M, ys) / 2
    return amps
```

Again, `ys` is the wave array. We don't have to pass `ts` and `freqs` as parameters; `dct_iv` can figure them out based on `N`, the length of `ys`.

We can test `dct_iv` like this

```
    amps = numpy.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    ts = (0.5 + numpy.arange(N)) / N
    freqs = (0.5 + numpy.arange(N)) / 2
    ys = synthesize2(amps, freqs, ts)

    amps2 = dct_iv(ys)
    print max(abs(amps - amps2))
```

Starting with amps, we synthesize a wave array, then use dct_iv to see if we can recover the amplitudes. The biggest difference between amps and amps2 is about 1e-16, which is what we expect due to floating-point errors.

6.6 Inverse DCT

Finally, notice that analyze2 and synthesize2 are almost identical. The only difference is that analyze2 divides the result by 2. We can use this insight to compute the inverse DCT:

```
def inverse_dct_iv(amps):
    return dct_iv(amps) * 2
```

inverse_dct_iv takes the vector of amplitudes and returns the wave array, ys. We can confirm that it works by starting with amps, applying inverse_dct_iv and dct_iv, and testing that we get back what we started with.

```
    amps = [0.6, 0.25, 0.1, 0.05]
    ys = inverse_dct_iv(amps)
    amps2 = dct_iv(ys)
    print max(abs(amps - amps2))
```

Again, the biggest difference is about 1e-16.

6.7 Exercises

Exercise 6.1 Test the algorithmic complexity of analyze1, analyze2 and scipy.fftpack.dct.

Exercise 6.2 One of the major applications of the DCT is compression for both sound and images. In its simplest form, DCT-based compression works like this:

1. Break a long signal into segments.
2. Compute the DCT of each segment.
3. Identify frequency components with amplitudes so low they are inaudible, and remove them. Store only the frequencies and amplitudes that remain.
4. To play back the signal, load the frequencies and amplitudes for each segment and apply the inverse DCT.

Implement a version of this algorithm and apply it to a recording of music or speech. How many components can you eliminate before the difference is perceptible?

Chapter 7

Discrete Fourier Transform

We've been using the discrete Fourier transform (DFT) since Chapter 1, but I haven't explained how it works. Now is the time.

If you understand the discrete cosine transform (DCT), you will understand the DFT. The only difference is that instead of using the cosine function, we'll use the complex exponential function. I'll start by explaining complex exponentials, then I'll follow the same progression as in Chapter 6:

1. We'll start with the synthesis problem: given a set of frequency components and their amplitudes, how can we construct a signal? The synthesis problem is equivalent to the inverse DFT.
2. Then I'll rewrite the synthesis problem in the form of matrix multiplication using NumPy arrays.
3. Next we'll solve the analysis problem, which is equivalent to the DFT: given a signal, how to we find the amplitude and phase offset of its frequency components?
4. Finally, we'll use linear algebra to find a more efficient way to compute the DFT.

The code for this chapter is in `dft.py`, which is in the repository for this book (see Section 0.1).

7.1 Complex exponentials

One of the more interesting moves in mathematics is the generalization of an operation from one type to another. For example, factorial is a function that operates on integers; the natural definition for factorial of n is the product of all integers from 1 to n.

If you are of a certain inclination, you might wonder how to compute the factorial of a non-integer like 3.5. Since the natural definition doesn't apply, you might look for other ways to compute the factorial function, ways that would work with non-integers.

In 1730, Leonhard Euler found one, a generalization of the factorial function that we know as the gamma function (see `http://en.wikipedia.org/wiki/Gamma_function`).

Euler also found one of the most useful generalizations in applied mathematics, the complex exponential function.

The natural definition of exponentiation is repeated multiplication. For example, $\phi^3 = \phi \cdot \phi \cdot \phi$. But this definition doesn't apply to non-integers, including complex numbers.

However, exponentiation can also be expressed as a power series:

$$e^{\phi} = 1 + \phi + \phi^2/2! + \phi^3/3! + ...$$

This definition works with imaginary numbers and, by a simple extension, with complex numbers. It turns out that:

$$e^{i\phi} = \cos\phi + i\sin\phi$$

You can see the derivation at `http://en.wikipedia.org/wiki/Euler's_formula`. This formula implies that $e^{i\phi}$ is a complex number with magnitude 1; if you think of it as a point in the complex plane, it is always on the unit circle. And if you think of it as a vector, the angle in radians between the vector and the positive x-axis is the argument, ϕ.

In the case where the argument is a complex number, we have:

$$e^{a+i\phi} = e^a e^{i\phi} = Ae^{i\phi}$$

where A is a real number that indicates amplitude and $e^{i\phi}$ is a unit complex number that indicates angle.

NumPy provides a version of `exp` that works with complex numbers:

```
>>> i = complex(0, 1)
>>> phi = 1.5
>>> z = numpy.exp(i * phi)
>>> z
(0.0707+0.997j)
```

`complex` is a built-in function that takes the real and imaginary parts of a complex number and returns a Python complex object.

When the argument to `numpy.exp` is imaginary or complex, the result is a complex number; specifically, a `numpy.complex128`, which is represented by two 64-bit floating-point numbers. The result is `0.0707+0.997j`. Note that Python uses `j` to represent the imaginary unit, rather than `i`.

Complex numbers have attributes `real` and `imag`:

```
>>> z.real
0.0707
>>> z.imag
0.997
```

To get the magnitude, you can use the built-in function `abs` or `numpy.absolute`:

```
>>> abs(z)
1.0
>>> numpy.absolute(z)
1.0
```

To get the angle, you can use `numpy.angle`:

```
>>> numpy.angle(z)
1.5
```

This example confirms that $e^{i\phi}$ is a complex number with magnitude 1 and angle ϕ radians.

7.2 Complex signals

If $\phi(t)$ is a function of time, $e^{i\phi(t)}$ is also a function of time. Specifically,

$$e^{i\phi(t)} = \cos\phi(t) + i\sin\phi(t)$$

This function describes a quantity that varies in time, so it is a signal. Specifically, it is a **complex exponential signal**.

In the special case where the frequency of the signal is constant, so $\phi(t) = 2\pi f t$, the result is a **complex sinusoid**:

$$e^{i2\pi f t} = \cos 2\pi f t + i\sin 2\pi f t$$

Or more generally, the signal might start at a phase offset ϕ_0, yielding

$$e^{i(2\pi f t + \phi_0)}$$

thinkdsp provides an implementation of this signal, ComplexSinusoid:

```
class ComplexSinusoid(Sinusoid):

   def evaluate(self, ts):
        i = complex(0, 1)
        phases = PI2 * self.freq * ts + self.offset
        ys = self.amp * numpy.exp(i * phases)
        return ys
```

ComplexSinusoid inherits __init__ from Sinusoid. It provides a version of evaluate that is almost identical to Sinusoid.evaluate; the difference is that it uses numpy.exp instead of numpy.sin.

The result is a NumPy array of complex numbers:

```
>>> signal = thinkdsp.ComplexSinusoid(freq=1, amp=0.6, offset=1)
>>> wave = signal.make_wave(duration=1, framerate=4)
>>> print(wave.ys)
[ 0.324+0.505j -0.505+0.324j -0.324-0.505j  0.505-0.324j]
```

The frequency of this signal is 1 cycle per second; the amplitude is 0.6 (in unspecified units); and the phase offset is 1 radian.

This example evaluates the signal at 4 places equally spaced between 0 and 1 second. The resulting samples are complex numbers. You might wonder how to interpret these values. We'll get to it soon.

7.3 The synthesis problem

We can create compound signals by adding up complex sinusoids with different frequencies. And that brings us to the synthesis problem: given the frequency and amplitude of each component, how do we evaluate the signal?

The simplest solution is to instantiate ComplexSignal objects and add them up.

```
def synthesize1(amps, freqs, ts):
    components = [thinkdsp.ComplexSinusoid(freq, amp)
                  for amp, freq in zip(amps, freqs)]
```

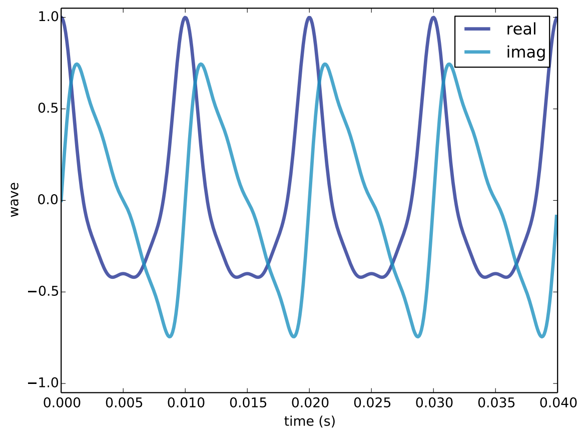

Figure 7.1: Real and imaginary parts of a mixture of complex sinusoids.

```
    signal = thinkdsp.SumSignal(*components)
    ys = signal.evaluate(ts)
    return ys
```

This solution is almost identical to `synthesize1` in Section 6.1; the only difference is that I replaced `CosSignal` with `ComplexSinusoid`.

Here's an example:

```
>>> amps = numpy.array([0.6, 0.25, 0.1, 0.05])
>>> freqs = [100, 200, 300, 400]
>>> framerate = 11025
>>> ts = numpy.linspace(0, 1, framerate)
>>> ys = synthesize1(amps, freqs, ts)
>>> print(ys)
[ 1.000 +0.000e+00j  0.995 +9.093e-02j  0.979 +1.803e-01j ...,
  0.979 -1.803e-01j  0.995 -9.093e-02j  1.000 -5.081e-15j]
```

At the lowest level, a complex signal is a sequence of complex numbers. But how should we interpret it? We have some intuition for real signals: they represent quantities that vary in time; for example, a sound signal represents changes in air pressure. But nothing we measure in the world yields complex numbers.

So what is a complex signal? I don't have a good answer to this question. The best I can offer is two unsatisfying answers:

1. A complex signal is a mathematical abstraction that is useful for computation and analysis, but it does not correspond directly with anything in the real world.

2. You can think of a complex signal as a sequence of complex numbers that contains two signals as its real and imaginary parts.

Taking the second point of view, we can split the previous signal into its real and imaginary parts:

```
    n = framerate / 25
    thinkplot.plot(ts[:n], ys[:n].real, label='real')
    thinkplot.plot(ts[:n], ys[:n].imag, label='imag')
```

Figure 7.1 shows a segment of the result. The real part is a sum of cosines; the imaginary part is the sum of sines. Although the waveforms look different, they contain the same frequency components in the same proportions. To our ears, they sound the same; in general, we don't hear phase offsets.

7.4 Synthesis with matrices

As we saw in Section 6.2, we can also express the synthesis problem in terms of matrix multiplication:

```
PI2 = 2 * math.pi
i = complex(0, 1)

def synthesize2(amps, freqs, ts):
    args = numpy.outer(ts, freqs)
    M = numpy.exp(i * PI2 * args)
    ys = numpy.dot(M, amps)
    return ys
```

Again, `amps` is a NumPy array that contains a sequence of amplitudes.

`freqs` is a sequence containing the frequencies of the components. `ts` contains the times where we will evaluate the signal.

`args` contains the outer product of `ts` and `freqs`, with the `ts` running down the rows and the `freqs` running across the columns (you might want to refer back to Figure 6.1).

Each column of matrix `M` contains a complex sinusoid with a particular frequency, evaluated at a sequence of `ts`.

When we multiply `M` by the amplitudes, the result is a vectors whose elements correspond to the `ts`; each element is the sum of several complex sinusoids, evaluated at a particular time.

Here's the example from the previous section again:

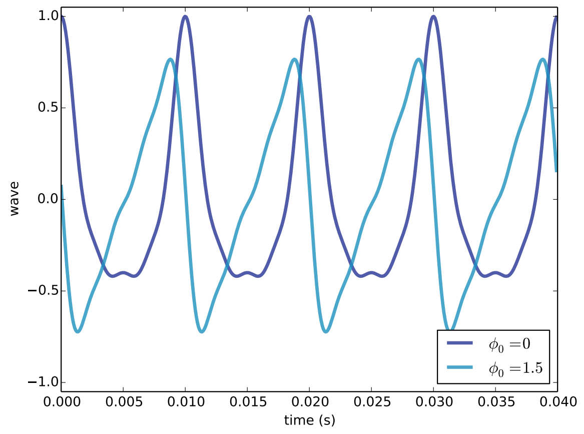

Figure 7.2: Real part of two complex signals that differ by a phase offset.

```
>>> ys = synthesize2(amps, freqs, ts)
>>> print(ys)
[ 1.000 +0.000e+00j  0.995 +9.093e-02j  0.979 +1.803e-01j ...,
  0.979 -1.803e-01j  0.995 -9.093e-02j  1.000 -5.081e-15j]
```

The result is the same.

In this example the amplitudes are real, but they could also be complex. What effect does a complex amplitude have on the result? Remember that we can think of a complex number in two ways: either the sum of a real and imaginary part, $x + iy$, or the product of a real amplitude and a complex exponential, $Ae^{i\phi_0}$. Using the second interpretation, we can see what happens when we multiply a complex amplitude by a complex sinusoid. For each frequency, f, we have:

$$Ae^{i\phi_0} \cdot e^{i2\pi ft} = Ae^{i2\pi ft + \phi_0}$$

Multiplying by $Ae^{i\phi_0}$ multiplies the amplitude by A and adds the phase offset ϕ_0.

We can test that claim by running the previous example with $\phi_0 = 1.5$ for all frequency components:

```
    amps2 = amps * numpy.exp(1.5j)
    ys2 = synthesize2(amps2, freqs, ts)

    thinkplot.plot(ts[:n], ys.real[:n])
    thinkplot.plot(ts[:n], ys2.real[:n])
```

Python recognizes `1.5j` as a complex number. Since `amps` is an array of reals, multiplying by `numpy.exp(1.5j)` yields an array of complex numbers with phase offset 1.5 radians, and the same magnitudes as `amps`.

Figure 7.2 shows the result. The phase offset $\phi_0 = 1.5$ shifts the wave to the left by about one quarter of a cycle; it also changes the waveform, because the same phase offset applied to different frequencies changes how the frequency components line up with each other.

Now that we have the more general solution to the synthesis problem – one that handles complex amplitudes – we are ready for the analysis problem.

7.5 The analysis problem

The analysis problem is the inverse of the synthesis problem: given a sequence of samples, y, and knowing the frequencies that make up the signal, can we compute the complex amplitudes of the components, a?

As we saw in Section 6.3, we can solve this problem by forming the synthesis matrix, M, and solving the system of linear equations, $Ma = y$ for a.

```
def analyze1(ys, freqs, ts):
    args = numpy.outer(ts, freqs)
    M = numpy.exp(i * PI2 * args)
    amps = numpy.linalg.solve(M, ys)
    return amps
```

`analyze1` takes a (possibly complex) wave array, `ys`, a sequence of real frequencies, `freqs`, and a sequence of real times, `ts`. It returns a sequence of complex amplitudes, `amps`.

Continuing the previous example, we can confirm that `analyze1` recovers the amplitudes we started with. For the linear system solver to work, `M` has to be square, so we need `ys`, `freqs` and `ts` to have the same length. I'll insure that by slicing `ys` and `ts` down to the length of `freqs`:

```
>>> n = len(freqs)
>>> amps2 = analyze1(ys[:n], freqs, ts[:n])
>>> print(amps2)
[ 0.60 +4.6e-13j  0.25 -1.4e-12j  0.10 +1.4e-12j  0.05 -4.6e-13j]
```

Each component of `amps` has a small imaginary part due to floating-point errors, but the results are approximately correct.

7.6 Efficient analysis

Unfortunately, solving a linear system of equations is slow. For the DCT, we were able to speed things up by choosing `freqs` and `ts` so that `M` is orthogonal. That way, the inverse of `M` is the transpose of `M`, and we can compute both DCT and inverse DCT by matrix multiplication.

We'll do the same thing for the DFT, with one small change. Since `M` is complex, we need it to be **unitary**, rather than orthogonal, which means that the inverse of `M` is the conjugate transpose of `M`, which we can compute by transposing the matrix and negating the imaginary part of each element. See `http://en.wikipedia.org/wiki/Unitary_matrix`.

The NumPy methods `conj` and `transpose` do what we want. Here's the code that computes `M` for $N = 4$ components:

```
>>> N = 4
>>> ts = numpy.arange(N) / N
>>> freqs = numpy.arange(N)
>>> args = numpy.outer(ts, freqs)
>>> M = numpy.exp(i * PI2 * args)
```

If M is unitary, $M^*M = I$, where M^* is the conjugate transpose of M, and I is the identity matrix. We can test whether M is unitary like this:

```
>>> MstarM = M.conj().transpose().dot(M)
```

The result, within the tolerance of floating-point error, is $4I$, so M is unitary except for an extra factor of N, similar to the extra factor of 2 we found with the DCT.

We can use this result to write a faster version of `analyze1`:

```
def analyze2(ys, freqs, ts):
    args = numpy.outer(ts, freqs)
    M = numpy.exp(i * PI2 * args)
    amps = M.conj().transpose().dot(ys) / N
    return amps
```

And test it with appropriate values of `freqs` and `ts`:

```
>>> N = 4
>>> amps = numpy.array([0.6, 0.25, 0.1, 0.05])
>>> freqs = numpy.arange(N)
>>> ts = numpy.arange(N) / N
>>> ys = synthesize2(amps, freqs, ts)

>>> amps2 = analyze2(ys, freqs, ts)
>>> print(amps2)
[ 0.60 +2.1e-17j  0.25 +1.3e-17j  0.10 -3.9e-17j  0.05 -8.3e-17j]
```

Again, the result is correct within the tolerance of floating-point arithmetic.

7.7 DFT

As a function, `analyze2` would be hard to use because it only works if `freqs` and `ts` are chosen correctly. So the usual definition of `dft` takes `ys` and computes `freqs` and `ts` accordingly:

```
def dft(ys):
    N = len(ys)
    ts = numpy.arange(N) / N
    freqs = numpy.arange(N)
    args = numpy.outer(ts, freqs)
    M = numpy.exp(i * PI2 * args)
    amps = M.conj().transpose().dot(ys)
    return amps
```

Notice that I made one small change; this version of `DFT` does not divide through by `N`. I did that to make it consistent with the conventional definition of DFT. Here's my version:

```
>>> print(dft(ys))
[ 2.4 +8.3e-17j  1.0 +5.6e-17j  0.4 -1.6e-16j  0.2 -3.3e-16j]
```

And here's the version in `numpy.fft`:

```
>>> print(numpy.fft.fft(ys))
[ 2.4 +8.0e-17j  1.0 -2.5e-17j  0.4 -3.1e-17j  0.2 -2.5e-17j]
```

They are approximately the same.

The inverse DFT is almost the same, except *now* we have to divide through by N:

```
def idft(amps):
    ys = dft(amps) / N
    return ys
```

Finally, we can confirm that `dft(idft(amps))` yields `amps`.

```
>>> ys = idft(amps)
>>> print(dft(ys))
[ 0.60 -2.1e-17j  0.05 +0.0e+00j  0.10 -2.5e-17j  0.25 -1.4e-17j]
```

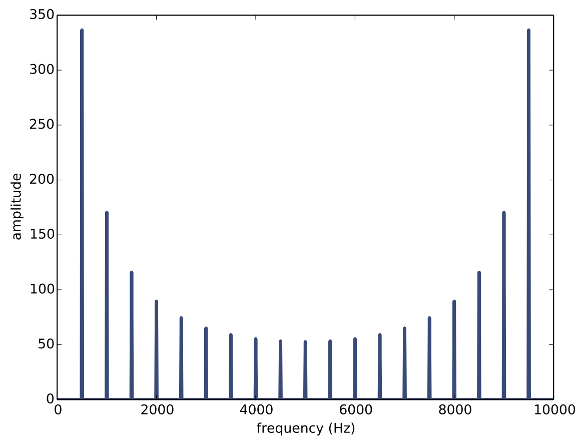

Figure 7.3: DFT of a 500 Hz sawtooth signal sampled at 10 kHz.

If I could go back in time, I might change the definition of DFT so it divides by N and the inverse DFT doesn't. That would be more consistent with my presentation of the synthesis and analysis problems.

Or I might change the definition so that both operations divide through by $\sqrt{N}$. Then the DFT and inverse DFT would be identical.

I can't go back in time (yet!), so we're stuck with a slightly weird convention. But for practical purposes it doesn't really matter.

7.8 Just one more thing

The Spectrum class in `thinkdsp` is based on `numpy.ftt.rfft`, which computes the "real DFT"; that is, it works with real signals. The DFT is more general than the real DFT; it works with complex signals.

So what happens when we apply DFT to a real signal? Let's look at an example:

```
>>> framerate = 10000
>>> signal = thinkdsp.SawtoothSignal(freq=500)
>>> wave = signal.make_wave(duration=0.1, framerate=framerate)
>>> hs = dft(wave.ys)
>>> amps = numpy.absolute(hs)
```

This code makes a sawtooth wave with frequency 500 Hz, sampled at framerate 10 kHz. `hs` contains the complex DFT of the wave; `amps` contains the amplitude at each frequency. But what frequency do these amplitudes correspond to? If we look at the body of `dft`, we see:

```
>>> fs = numpy.arange(N)
```

So it's tempting to think that these values are the right frequencies. The problem is that `dft` doesn't know the sampling rate. The DFT assumes that the duration of the wave is 1 time unit, so the sampling rate is N per time unit. In order to interpret the frequencies, we have to convert from these arbitrary time units back to seconds, like this:

```
>>> fs = numpy.arange(N) * framerate / N
```

With this change, the range of frequencies is from 0 to the actual framerate, 10 kHz. Now we can plot the spectrum:

```
>>> thinkplot.plot(fs, amps)
>>> thinkplot.config(xlabel='frequency (Hz)',
                     ylabel='amplitude')
```

Figure 7.3 shows the amplitude of the signal for each frequency component from 0 to 10 kHz. The left half of the figure is what we should expect: the dominant frequency is at 500 Hz, with harmonics dropping off like $1/f$.

But the right half of the figure is a surprise. Past 5000 Hz, the amplitude of the harmonics start growing again, peaking at 9500 Hz. What's going on?

The answer: aliasing. Remember that with framerate 10000 Hz, the folding frequency is 5000 Hz. As we saw in Section 2.5, a component at 5500 Hz is indistinguishable from a component at 4500 Hz. So when we evaluate the DFT at 5500 Hz, we get the same value as at 4500 Hz. Similarly, the value at 6000 Hz is the same as the one at 4000 Hz, and so on.

So the DFT of a real signal is symmetric around the folding frequency. Since there is no additional information past this point, we can save time be evaluating only the first half of the DFT, and that's exactly what `rfft` does.

7.9 Exercises

Exercise 7.1 In this chapter, I showed how we can express the DFT and inverse DFT as matrix multiplications. These operations are relatively fast, taking time proportional to N^2, where N is the length of the wave array. That would be fast enough for many applications, but it turns out that there is a faster algorithm, the Fast Fourier Transform (FFT), which takes time proportional to $N \log N$.

Read about the FFT at `http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`, and write an implementation. Hint: I suggest you write a simple version as a recursive function; don't worry about "data reordering, bit reversal, and in-place algorithms".

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 one of the simplest examples: smoothing.

The code for this chapter is in `convolution.py`, which is in the repository for this book (see Section 0.1).

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 BitCoin from July 18, 2010 to December 28, 2014 (downloaded from `http://www.coindesk.com`). The gray line is the raw data, the darker line shows the 30-day moving average. In the short term, the price of BitCoin is quite volatile, often changing by more than 10% in a day. 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 2.4, the harmonics of

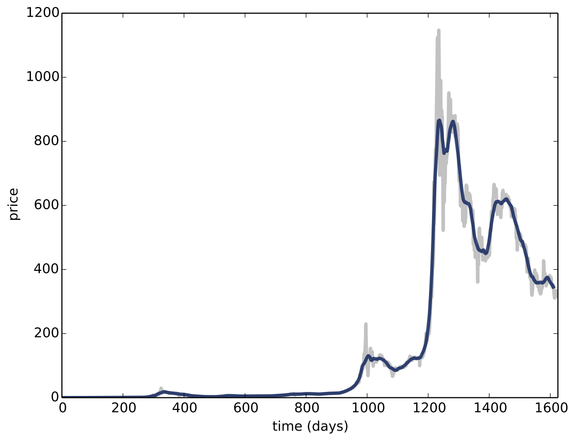

Figure 8.1: Daily closing price of BitCoin and a 30-day moving average.

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

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

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

You can try out this example, and listen to it, in `chap08.ipynb`, which is in the repository for this book and also available at `http://tinyurl.com/thinkdsp08`.

To compute the moving average of this signal, I'll create a window with 11 elements and normalize it so the elements add up to 1.

```
>>> window = numpy.ones(11)
>>> window /= sum(window)
```

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

```
>>> padded = zero_pad(window, len(segment))
>>> prod = padded * segment.ys
```

`padded` is a version of the window with zeros added to the end so it has the same length as `segment.ys` (`zero_pad` is defined in `convolution.py`). `prod` is the product of the window and the wave array.

The first 11 elements of this wave array are -1, so their average is -1, and (sure enough) that's the value of `prod`.

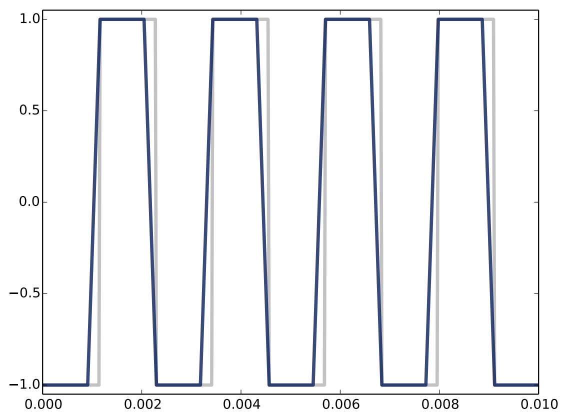

Figure 8.2: A square signal at 400 Hz (gray) and an 11-element moving average (darker).

To compute the next element of the smoothed signal, we shift the window to the right and compute the average of the next 11 elements of the wave array, starting with the second.

The following code computes the rolling average for the whole wave:

```
>>> smoothed = numpy.zeros_like(segment.ys)
>>> rolled = padded
>>> for i in range(len(segment.ys)):
        smoothed[i] = sum(rolled * segment.ys)
        rolled = numpy.roll(rolled, 1)
```

`smoothed` is the wave array where I store the results. `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 the segment by `rolled` to select 11 elements, and then add them up.

Figure 8.2 shows the result. 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 computed is called **convolution**, and it is such a common operation that NumPy provides an implementation that is simpler and faster than my version:

```
>>> ys = numpy.convolve(segment.ys, window, mode='valid')
>>> smooth2 = thinkdsp.Wave(ys, framerate=wave.framerate)
```

`numpy.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**:

$$(f \star g)[j] = \sum_{k=0}^{N-1} f[k]g[k-j]$$

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

The mathematical definition of convolution is slightly different:

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

The symbol $*$ represents convolution. The difference is that the index of g has been negated, which has the same effect as reversing the elements of g (assuming that negative indices wrap around to the end of the array).

Because the moving average window 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 the way it is. There are two reasons:

- Convolution comes up naturally for several applications, especially analysis of signal-processing systems, which is the topic of Chapter 9.
- Also, convolution is the basis of the Convolution Theorem, coming up very soon.

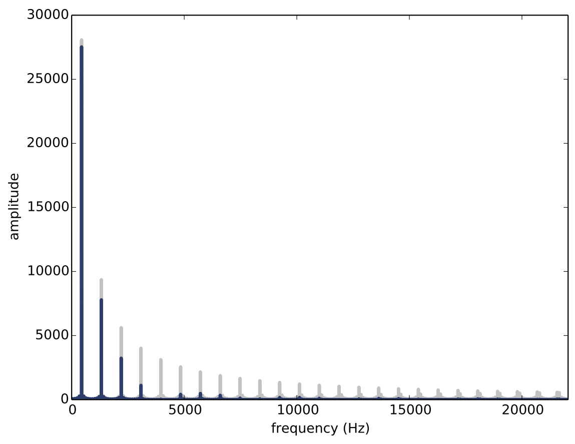

Figure 8.3: Spectrum of the square wave before and after smoothing.

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='0.7')
```

Then the smoothed wave:

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

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)
```

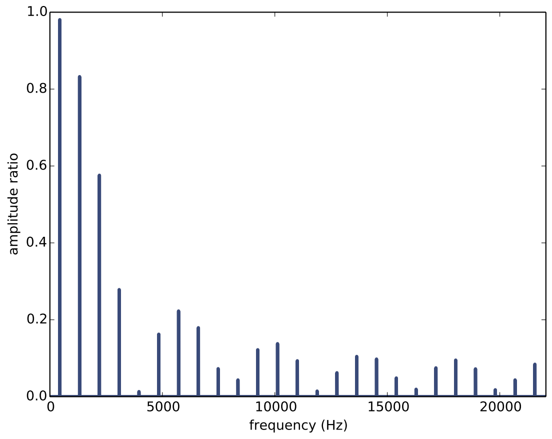

Figure 8.4: Ratio of spectrums for the square wave, before and after smoothing.

`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 for 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 feature we did not expect: above the cutoff, the ratio seems to oscillate between 0 and 0.2. What's up with that?

8.4 The convolution theorem

The answer is the convolution theorem. Stated mathematically:

$$DFT(f * g) = DFT(f) \cdot DFT(g)$$

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. 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:

```
    padded = zero_pad(window, len(wave))
```

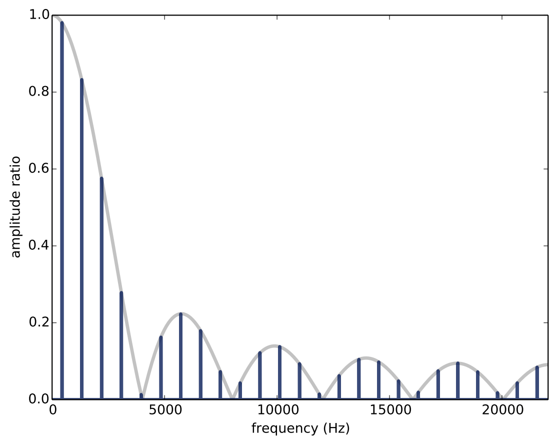

Figure 8.5: Ratio of spectrums for the square wave, before and after smoothing, along with the DFT of the smoothing window.

```
dft_window = numpy.fft.rfft(padded)
thinkplot.plot(abs(dft_window))
```

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

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:

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

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 than 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. It cuts off at around 4000 Hz, but then it bounces around for a while. For reasons we will see soon, we can do better with a Gaussian window.

SciPy provides functions that compute many common convolution windows, including `gaussian`:

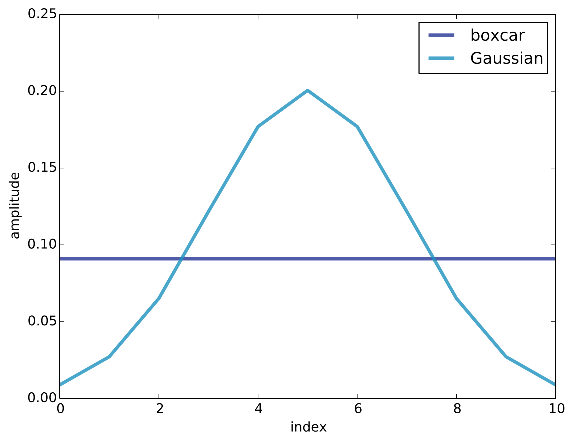

Figure 8.6: Boxcar and Gaussian windows.

```
    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 expression 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 curve, 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 much better than a simple moving average. The cutoff frequency is a little higher; near 7000 Hz, the ratio is 0.1. But above that, the ratio drops quickly and stays low, with almost none of the oscillation we saw with the boxcar window.

In Figure 8.7, the shape of the filter might look familiar. As it turns out, the DFT of a Gaussian curve is also a Gaussian curve. In this example, the window is a discrete approximation, so the resulting filter is only approximately Gaussian.

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 con-

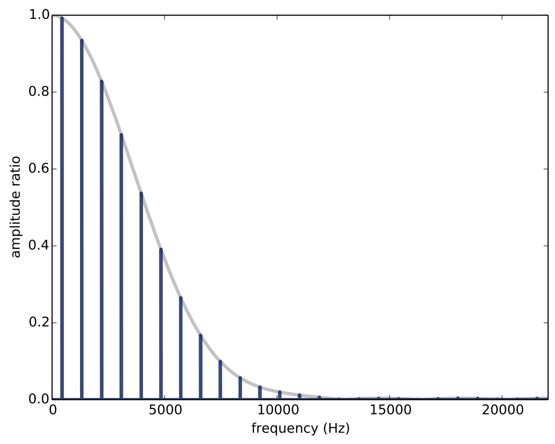

Figure 8.7: Ratio of spectrums before and after Gaussian smoothing, and the DFT of the window.

volution, cross-correlation, and autocorrelation.

Again, the Convolution Theorem states

$$DFT(f * g) = DFT(f) \cdot DFT(g)$$

So one way to compute a convolution is:

$$f * g = IDFT(DFT(f) \cdot DFT(g))$$

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 BitCoin data shown in Figure 8.1.

```
df = pandas.read_csv('coindesk-bpi-USD-close.csv',
                     nrows=1625,
                     parse_dates=[0])
ys = df.Close.values
```

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. `ys` is a NumPy array that contains daily closing prices.

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

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

With `mode='valid'`, `numpy.convolve` returns values only where the window and the signal overlap completely, avoiding bogus values at the beginning and end.

`fft_convolve` computes the same thing using FFT:

```
def fft_convolve(signal, window):
    fft_signal = numpy.fft.fft(signal)
    fft_window = numpy.fft.fft(window)
    return numpy.fft.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, len(ys))
    smoothed2 = fft_convolve(ys, padded)
```

The result has $M-1$ bogus values at the beginning, where M is the length of the window. If we slice off the bogus values, we get the same result as before, with about 12 digits of precision.

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

You can run this example in `chap08.ipynb`, which is in the repository for this book and also available at `http://tinyurl.com/thinkdsp08`.

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 of BitCoin prices:

```
N = len(ys)
corrs = numpy.correlate(ys, ys, mode='same')
corrs = corrs[N//2:]
```

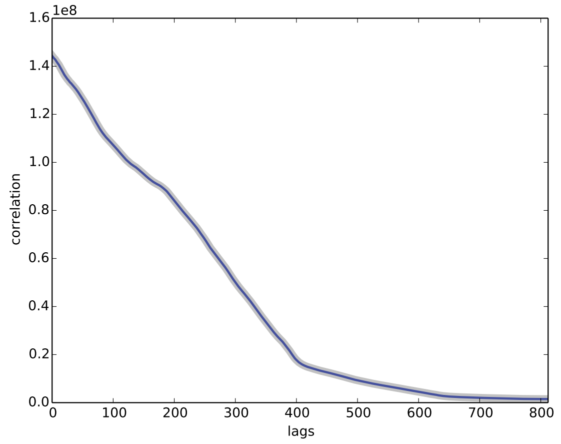

Figure 8.8: Autocorrelation functions computed by NumPy and `fft_correlate`.

With `mode='same'`, the result has the same length as *ys*, corresponding to lags from $-N/2$ to $N/2-1$. By selecting the second half, we get only the non-negative lags. 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.

To compute autocorrelation using convolution, we make a copy of the signal and reverse it. Then we pad the copy and the original to double their 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.

```
def fft_autocorr(signal):
    N = len(signal)
    window = signal[::-1]
    signal = zero_pad(signal, 2*N)
    window = zero_pad(window, 2*N)

    corrs = fft_convolve(signal, window)
    corrs = corrs[N//2: 3*N//2]
    return corrs
```

When `fft_autocorr` calls `fft_convolve`, the result has length $2N$, corresponding to lags from $-N+1$ to N. From these I slice out the middle half, corresponding to lags from $-N/2$ to $N/2-1$.

Finally, we call `fft_autocorr` and select the second half, which contains correlations with non-negative lags.

```
    corrs2 = fft_autocorr(ys)
    corrs2 = corrs2[N//2:]
```

The thin, dark line in Figure 8.8 shows the result, which is identical to the result from `numpy.correlate`, with about 7 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 packing and unpacking is a nuisance, but that's why libraries like NumPy provide functions to do it for you.

8.8 Exercises

Exercise 8.1 Experiment with the parameters of the Gaussian window to see what effect they have on the cutoff frequency.

Chapter 9

Signals and Systems

The code for this chapter is in `systems.py`, which is in the repository for this book (see Section 0.1).

9.1 Finite differences

In Section 8.1, I applied a smoothing window to the historical price of BitCoin, and we found that a smoothing window in the time domain corresponds to a low-pass filter in the frequency domain.

In this section, we'll look at daily price changes, and we'll see that computing the difference between successive elements, in the time domain, corresponds to a high-pass filter.

Here's the code to read the data, store it as a wave, and compute its spectrum.

```
    df = pandas.read_csv('coindesk-bpi-USD-close.csv',
                         nrows=1625,
                         parse_dates=[0])
    ys = df.Close.values
    wave = thinkdsp.Wave(ys, framerate=1)
    spectrum = wave.make_spectrum()
```

The framerate is 1 sample per day. Figure 9.1 shows the time series and its spectrum plotted on a log-log scale. Visually, the time series resembles Brownian noise (see Section 4.3). The estimated slope of the spectrum is -1.8, which is consistent with Brownian noise.

Now let's compute the daily price change using `numpy.diff`:

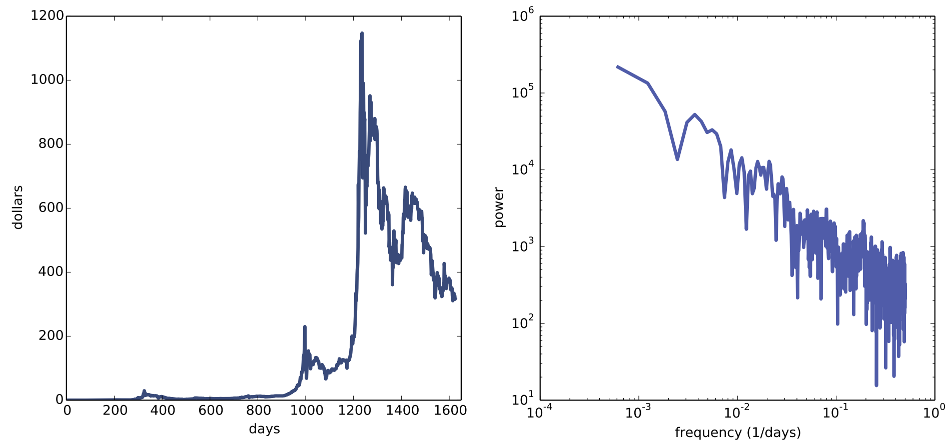

Figure 9.1: Daily closing price of BitCoin and the spectrum of this time series.

```
diff = numpy.diff(ys)
wave2 = thinkdsp.Wave(diff, framerate=1)
```

Figure 9.2 shows the resulting wave and its spectrum. The daily changes and their spectrum resemble white noise, and the estimated parameter, -0.023, is near zero, which is what we expect for white noise.

9.2 The frequency domain

To see the effect of this operation in the frequency domain, we can compute the ratio of amplitudes before and after:

```
amps = spectrum.amps
amps2 = spectrum2.amps
n = min(len(amps), len(amps2))
ratio = amps2[:n] / amps[:n]
thinkplot.plot(ratio)
```

We can also characterize the effect in the frequency domain by computing the DFT of the window. Computing the difference between successive elements is the same as convolution with the window `[1, -1]`.

```
window = numpy.array([1.0, -1.0])
padded = zero_pad(window, len(wave))
fft_window = numpy.fft.rfft(padded)
thinkplot.plot(abs(fft_window), color='0.7')
```

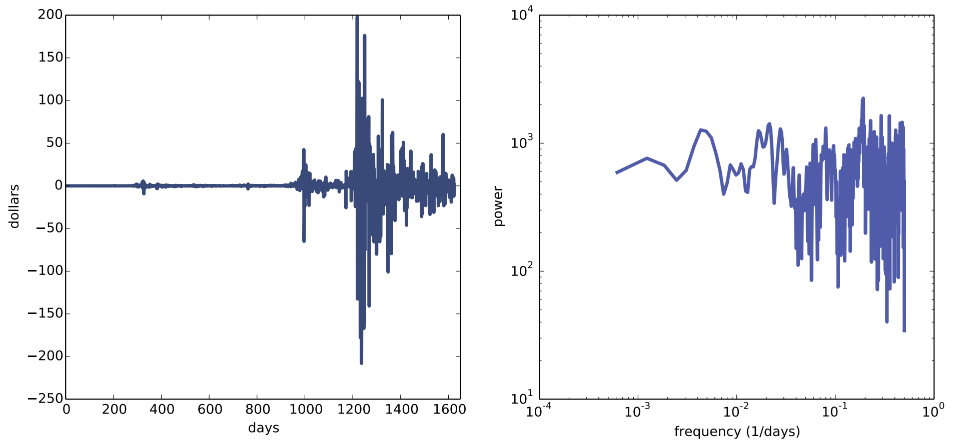

Figure 9.2: Daily price changes of BitCoin and their spectrum.

If the order of the elements in the window seems backward, remember that convolution reverses the window before applying it to the signal.

Figure 9.3 shows the computed ratio and the DFT of the window, which is the filter. The computed ratio is noisy, but generally follows the shape of the filter.

The filter increases with frequency, linearly for low frequencies, and then sublinearly after that. In the next section, we'll see why.

9.3 Differentiation

The window we used in the previous section is a numerical approximation of the first derivative, so the filter approximates the effect of differentiation, which we can figure out mathematically.

Suppose we have a complex sinusoid with frequency f:

$$E_f(t) = e^{2\pi i f t}$$

The first derivative of E_f is

$$\frac{d}{dt} E_f(t) = 2\pi i f e^{2\pi i f t}$$

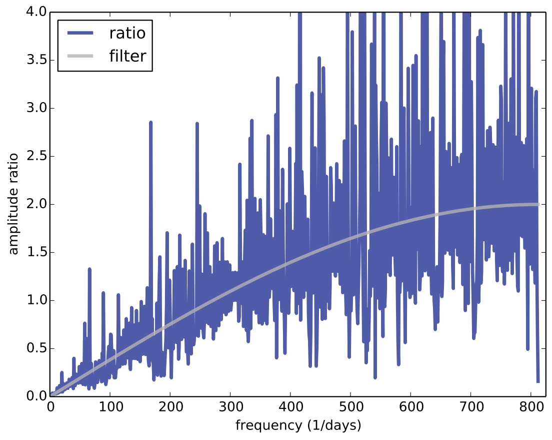

Figure 9.3: Ratio of amplitudes before and after filtering, and the filter corresponding to the `diff` operation.

which we can rewrite as

$$\frac{d}{dt}E_f(t) = 2\pi i f E_f(t)$$

In other words, taking the derivative of E_f is the same as multiplying by $2\pi i f$, which is a complex scalar (more specifically, an imaginary scalar).

As we saw in Section 7.4, multiplying a complex sinusoid by a complex scalar has two effects: it multiplies the amplitude, in this case by $2\pi f$, and shifts the phase offset, in this case by $\pi/2$ (because that's the angle of a positive imaginary number).

If you are familiar with the language of operators and eigenfunctions, each E_f is an eigenfunction of the differentiation operator, with the corresponding eigenvalue $2\pi i f$. See `http://en.wikipedia.org/wiki/Eigenfunction`.

If you are not familiar with that language, I'll tell you what it means:

- An operator is a function that takes a function and returns another function. For example, differentiation is an operator.

- A function, g, is an eigenfunction of an operator, $\mathcal{A}$, if applying $\mathcal{A}$ to g has the effect of multiplying the function by a scalar. That is, $\mathcal{A}g = \lambda g$.

- In that case, the scalar λ is the eigenvalue that corresponds to the eigenfunction g.

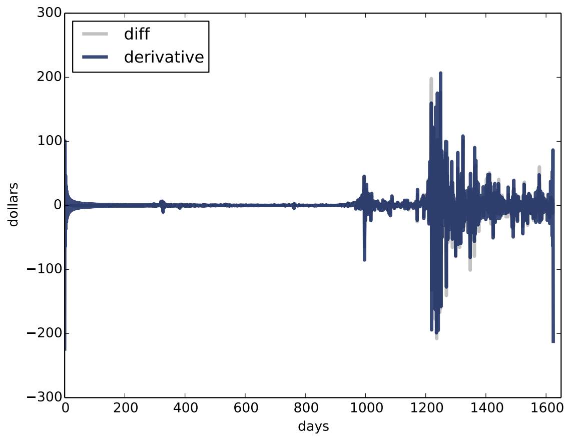

Figure 9.4: Comparison of daily price changes computed by `numpy.diff` and by filtering in the frequency domain.

- A given operator might have many eigenfunctions, each with a corresponding eigenvalue.

For a complex signal with only one frequency component, differentiation is easy. For signals with more than one component, the process is only slightly harder:

1. Express the signal as the sum of complex sinusoids.
2. Compute the derivative of each component by multiplication.
3. Add up the differentiated components.

If that process sounds familiar, that's because it is identical to the algorithm for convolution in Section 8.6: compute the DFT, multiply by a filter, and compute the inverse DFT.

`Spectrum` provides a method that applies the differentiation filter:

```
# class Spectrum:

    def differentiate(self):
        i = complex(0, 1)
        filtr = PI2 * i * self.fs
        self.hs *= filtr
```

We can use it to compute the derivative of the BitCoin time series:

```
spectrum3 = wave.make_spectrum()
spectrum3.differentiate()
wave3 = spectrum3.make_wave()
```

And then plot `wave3`, compared to the previous result using `numpy.diff`, `wave3`:

```
wave2.plot(color='0.7', label='diff')
wave3.plot(label='derivative')
```

Figure 9.4 shows the results. They are similar except where the changes are most abrupt. They also differ at the boundaries, because the DFT-based derivative is based on the assumption that the signal is periodic. It wraps around from the end to the beginning, creating artifacts at the boundaries.

To summarize, we have shown:

- Computing the difference between successive values in a signal can be expressed as convolution with a simple window. The result is an approximation of the first derivative.
- Differentiation in the time domain corresponds to a simple filter in the frequency domain.

The second conclusion is the basis of **spectral methods** for solving differential equations (see `http://en.wikipedia.org/wiki/Spectral_method`).

In particular, it is useful for the analysis of linear, time-invariant systems, which is the next topic.

9.4 LTI systems

In the context of signal processing, a **system** is an abstract representation of anything that takes a signal as input and produces a signal as output.

For example, an electronic amplifier is a circuit that takes an electrical signal as input and produces a (louder) signal as output.

As another example, for one of the exercises in Chapter **??**, we computed the sound of a violin as heard through a kilometer of sea water. In that case, we model the sea water as a system that takes the original recording as input and produces a filtered version as output.

A **linear, time-invariant system**[1] is a system with these additional properties:

[1]My presentation here follows `http://en.wikipedia.org/wiki/LTI_system_theory`.

1. Linearity: If the input x_1 produces output y_1 and input x_2 produces y_2, then $ax_1 + bx_2$ produces $ay_1 + by_2$, where a and b are scalars. In other words, if you put two inputs into the system at the same time, the result is the sum of their outputs.

2. Time invariance: If x_1 and x_2 differ by a shift in time, y_1 and y_2 differ by the same shift. In other words, the effect of the system doesn't vary over time, or depend on the state of the system.

Many physical systems have these properties, at least approximately.

- Circuits that contain only resistors, capacitors and inductors are LTI, to the degree that the components behave like their idealized models.
- Mechanical systems that contain springs, masses and dashpots are also LTI, assuming linear springs (force proportional to displacement) and dashpots (force proportional to velocity).
- Also, and most relevant to applications in this book, the media that transmit sounds (including air, water and solids) are well-modeled by LTI systems.

LTI systems are described by linear differential equations, and the solutions of those equations are complex sinusoids (see `http://en.wikipedia.org/wiki/Linear_differential_equation`).

This result provides an algorithm for computing the effect of an LTI system on an input signal:

1. Express the signal as the sum of complex sinusoids.
2. For each input component, compute the corresponding output component.
3. Add up the output components.

At this point, I hope this algorithm sounds familiar. It's the same algorithm we used for convolution in Section 8.6, and for differentiation in Section 9.3. This process is called **spectral decomposition** because we "decompose" the input signal into its spectral components.

In order to apply this process to an LTI system, we have to **characterize** the system by finding its effect on each component of the input signal. For mechanical systems, it turns out that there is a simple and efficient way to do that: you kick it and measure the output.

Technically, the "kick" is called an **impulse** and the output is called the **impulse response**. You might wonder how a single impulse can completely characterize a system. You can see the answer by computing the DFT of an impulse. Here's a wave array with an impulse at $t = 0$:

```
>>> impulse = numpy.zeros(8)
>>> impulse[0] = 1
>>> print(impulse)
[ 1.  0.  0.  0.  0.  0.  0.  0.]
```

And here's its spectrum:

```
>>> spectrum = numpy.fft.fft(impulse)
>>> print(spectrum)
[ 1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j  1.+0.j]
```

That's right: an impulse is the sum of components with equal magnitudes at all frequencies (and a phase offset of 0).

What that means is that when you test a system by inputting an impulse, you are testing the response of the system at all frequencies. And you can test them all at the same time because the system is linear, so simultaneous tests don't interfere with each other.

9.5 Transfer functions

For characterizing the acoustic response of a room or open space, a simple and effective way to generate an impulse is to fire a gun (or a starter pistol). The gunshot puts an impulse into the system; the sound you hear is the impulse response.

As an example, I'll use a recording of a gunshot to characterize the room where the gun was fired, then use the impulse response to simulate the effect of that room on a violin recording.

This example is in `chap09.ipynb`, which is in the repository for this book; you can also view it, and listen to the examples, at `http://tinyurl.com/thinkdsp09`.

Here's the gunshot:

```
    response = thinkdsp.read_wave('180961__kleeb__gunshots.wav')
    response = response.segment(start=0.26, duration=5.0)
    response.normalize()
    response.plot()
```

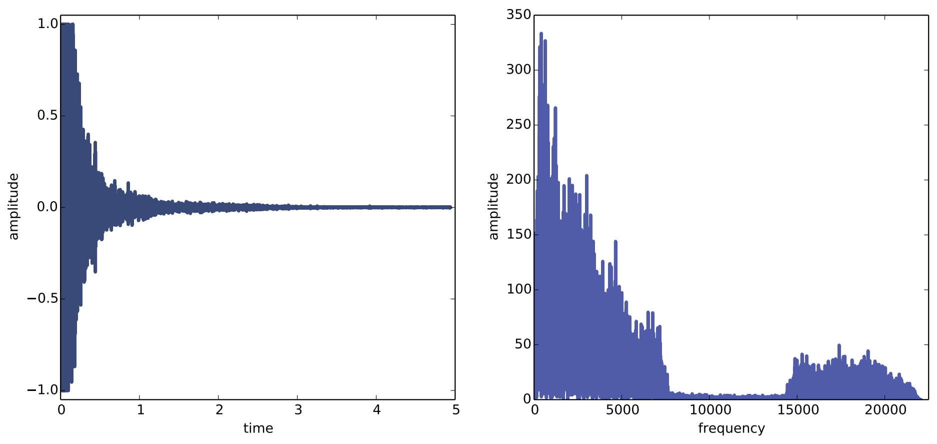

Figure 9.5: Waveform of a gunshot.

I select a segment starting at 0.26 seconds to remove the silence before the gunshot. Figure **??** (left) shows the waveform of the gunshot. Next we compute the DFT of `response`:

```
transfer = response.make_spectrum()
transfer.plot()
```

Figure **??** (right) shows the result. This spectrum encodes the response of the room; for each frequency, the spectrum contains a complex number that represents an amplitude multiplier and a phase shift. This spectrum is called a **transfer function** because it contains information about how the system transfers the input to the output.

Now we can simulate the effect this room would have on the sound of a violin. Here is the violin recording we used in Chapter **??**

```
wave = thinkdsp.read_wave('92002__jcveliz__violin-origional.wav')
wave.ys = wave.ys[:len(response)]
wave.normalize()
```

The violin and gunshot waves were sampled at the same framerate, 44,100 Hz. And coincidentally, the duration of both is about 5 seconds. I trimmed the violin wave to the same length as the gunshot.

Next I'll compute the DFT of the violin wave:

```
spectrum = wave.make_spectrum()
```

Now I know the magnitude and phase of each component in the input, and I know the transfer function of the system. Their product is the DFT of the output:

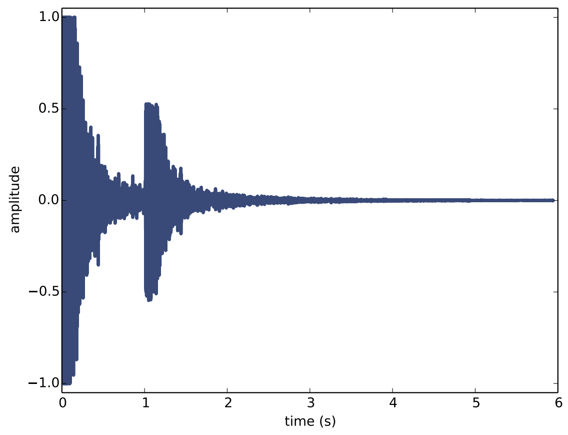

Figure 9.6: Sum of a gunshot waveform with a shifted, scaled version of itself.

```
output = (spectrum * transfer).make_wave()
output.normalize()
output.plot()
```

Figure **??** shows the output of the system superimposed on the input. The overall shape is similar, but there are substantial differences. And those differences are clearly audible. Load `chap09.ipynb` and listen to them. One thing I find striking about this example is that you can get a sense of what the room was like; to me, it sounds like a long, narrow room with hard floors and ceilings. That is, a firing range.

There's one things I glossed over in this example that I'll mention in case it bothers anyone. The violin recording I started with has already been transformed by one system: the room where it was recorded. So what I really computed in my example is the sound of the violin after two transformations. To really simulate the sound of a violin in a different room, I should have characterized the room where the violin was recorded and applied the inverse of that transfer function first.

9.6 Systems and convolution

If you think the previous example is black magic, you are not alone. I've been thinking about it for a while and it still makes my head hurt.

In the previous section, I suggested one way to think about it:

- An impulse is made up of components with amplitude 1 at all frequencies.
- The impulse response contains sum of the responses of the system to all of these components.
- The transfer function, which is the DFT of the impulse response, encodes effect of the system on each frequency component in the form of an amplitude multiplier and a phase shift.
- For any input, we can compute the response of the system by breaking the input into components, computing the response to each component, and adding them up.

But if you don't like that, there's another way to think about it altogether: convolution!

Suppose that instead of firing one gun, you fire two guns: a big one with amplitude 1 at $t = 0$ and a smaller one with amplitude 0.5 at $t = 1$.

We can compute the response of the system by adding up the original impulse response and a scaled, shifted version of itself. Here's a function that makes a shifted, scaled version of a wave:

```
def shifted_scaled(wave, shift, factor):
    res = wave.copy()
    res.shift(shift)
    res.scale(factor)
    return res
```

Here's how we use it to compute the response to a two-gun salute:

```
dt = 1
shift = dt * response.framerate
factor = 0.5
response2 = response + shifted_scaled(response, shift, factor)
```

Figure 9.6 shows the result. You can hear what it sounds like in `chap09.ipynb`, which you can view at `http://tinyurl.com/thinkdsp09`. Not surprisingly, it sounds like two gunshots, the first one louder than the second.

Now suppose instead of two guns, you add up 100 guns, with a shift of 100 between them. This loop computes the result:

```
    total = 0
    for j in range(100):
        total += shifted_scaled(response, j*100, 1.0)
```

$$\begin{array}{ccccccccc}
f[0] & [& g[0] & g[1] & g[2] & \ldots & & &] \\
f[1] & [& & g[0] & g[1] & g[2] & \ldots & &] \\
f[2] & [& & & g[0] & g[1] & g[2] & \ldots &] \\
\hline
 & [& & & h[2] & & & &]
\end{array}$$

Figure 9.7: Diagram of the sum of scaled and shifted copies of g.

At a framerate of 44,100 Hz, there are 441 gunshots per second, so you don't hear the individual shots. Instead, it sounds like a periodic signal at 441 Hz. If you play this example, it sounds like a car horn in a garage.

And that brings us to a key insight: you can think of any wave as a series of samples, where each sample is an impulse with a different magnitude.

As a example, I'll generate a sawtooth signal at 410 Hz:

```
signal = thinkdsp.SawtoothSignal(freq=410)
wave = signal.make_wave(duration=0.1,
                        framerate=response.framerate)
```

Now I'll loop through the series of impulses that make up the sawtooth, and add up the impulse responses:

```
total = 0
for j, y in enumerate(wave.ys):
    total += shifted_scaled(response, j, y)
```

The result is what it would sound like to play a sawtooth wave in a firing range. Again, you can listen to it in `chap09.ipynb`.

Figure 9.7 shows a diagram of this computation, where f is the sawtooth, g is the impulse response, and h is the sum of the shifted, scaled copies of g.

For the example shown:

$$h[2] = f[0]g[2] + f[1]g[1] + f[2]g[0]$$

And more generally,

$$h[i] = \sum_{j=0}^{N-1} f[j]g[i-j]$$

You might recognize this equation from Section 8.2. It is the convolution of f and g.

In summary, there are two ways to think about the effect of a system on a signal:

1. The input is a sequence of impulses, so the output is the sum of scaled, shifted copies of the impulse response, which is the convolution of the input and the impulse response.
2. The DFT of the impulse response is a transfer function that encodes the effect of the system on each frequency component as a magnitude and phase offset. The DFT of the input encodes the magnitude and phase offset of the frequency components it contains. Multiplying the DFT of the input by the transfer function yields the DFT of the output.

The equivalence of these two descriptions should not be a surprise; it is basically a statement of the Convolution Theorem: convolution in the time domain corresponds to multiplication in the frequency domain.

9.7 Proof of the Convolution Theorem

Well, I've put it off long enough. It's time to prove the Convolution Theorem (CT), which states:

$$DFT(f * g) = DFT(f)DFT(g)$$

where f and g are vectors with the same length, N.

I'll proceed in two steps:

1. I'll show that in the special case where f is a complex exponential, convolution with g has the effect of multiplying f by a scalar.
2. In the more general case where f is not a complex exponential, we can use the DFT to express it as a sum of exponential components, compute the convolution of each component (by multiplication) and then add up the results.

The second step demonstrates the Convolution Theorem.

First, let's assemble the pieces we'll need. The DFT of g, which I'll call G, is:

$$G[n] = \sum_k g[k] \exp(-2\pi i n k / N)$$

where N is the length of g, and n is an index that runs from 0 to $N-1$. So G is a vector of complex numbers, with length N.

And here's the definition of convolution:

$$(f * g)[j] = \sum_k f[k]g[j-k]$$

This is the definition we saw in Section 8.2, except that I've changed the names of the index variables.

Convolution is commutative, so I could equivalently write:

$$(f * g)[j] = \sum_k f[j-k]g[k]$$

Now let's consider the special case where f is a complex exponential; specifically,

$$f_n[\kappa] = \exp(2\pi i n \kappa / N)$$

where N is the length of f and g, and n is one of the integers from 0 to $N-1$. I'm using κ here as a placekeeper to avoid confusion because very soon I will need to substitute $j-k$ for κ.

Plugging f_n into the second definition of convolution yields

$$(f_n * g)[j] = \sum_k \exp(2\pi i n(j-k)/N)g[k]$$

We can split the first term into a product:

$$(f_n * g)[j] = \sum_k \exp(2\pi i n j/N)\exp(-2\pi i n k/N)g[k]$$

The first half does not depend on k, so we can pull it out of the summation:

$$(f_n * g)[j] = \exp(2\pi i n j/N)\sum_k \exp(-2\pi i n k/N)g[k]$$

Now we recognize the first term is f_n (with j substituted for κ), and the summation is $G[n]$. So we can write:

$$(f_n * g)[j] = f_n[j]G[n]$$

which shows that for each complex exponential, f_n, convolution with g has the effect of multiplying f_n by $G[n]$. In other words, each f_n is an eigenvector of this operation, and $G[n]$ is the corresponding eigenvalue.

9.8 Exercises

Exercise 9.1 Are kazoos nonlinear?

Also see `http://www.google.com/patents/US20140256218`

And `http://designingsound.org/2014/07/spectral-analysis-interview-with-rob-blake/`

Chapter 10

Fourier analysis of images

Appendix A

Linear Algebra

Dot product

In the special case where y has length 1, we have

$$x \cdot \hat{y} = \mathrm{norm}(x) \cos \theta$$

which is the **scalar projection** of x onto $\hat{y}$; that is, it is the length of the component of x that points in the direction of $\hat{y}$.

I mention this now because it will come in handy when we talk about transforms.

In the context of computation, it is common to think of a vector as a sequence of numbers, and people often use "vector" and "array" interchangeably. But it is more correct to think of a vector as an abstract mathematical concept, with an array as one way to represent one.

From a mathematical point of view, a vector is a sequence of coordinates relative to a **basis**. Most vectors are implicitly defined relative to the "standard basis", which is a sequence of unit vectors.

For example, in a 3-D Euclidean space, the standard basis is the sequence of vectors:

$$\begin{aligned} e_x &= (1, 0, 0) && \text{(A.1)} \\ e_y &= (0, 1, 0) && \text{(A.2)} \\ e_z &= (0, 0, 1) && \text{(A.3)} \\ & && \text{(A.4)} \end{aligned}$$

Relative to this basis, the vector $(1, 2, 3)$ is understood to mean $1e_x + 2e_y + 3e_z$. This is similar to the way we understand that the number 123 means $100 + 20 + 3$, provided that it is in base 10.

But just as we can write numbers in other bases, we can define vectors relative to other bases. A basis is a sequence of vectors that are linearly-independent, which means that no vector in the set can be written as a weighted sum of the others.

One note on linear algebra vocabulary: I am using "matrix" to refer to a two-dimensional array, and "vector" for a one-dimensional array or sequence. To be pedantic, it would be more correct to think of matrices and vectors as abstract mathematical concepts, and NumPy arrays as one way (and not the only way) to represent them.

This chapter explores the relationship between the synthesis problem and the analysis problem. By writing the synthesis problem in the form of matrix operations, we derive an efficient algorithm for computing the DCT.

That's useful for analysis and synthesis, but it also provides an alternative way to think about these operations: in the vocabulary of linear algebra, they are linear transformations, or **transforms**.

To understand transforms, we have to start with bases. A **basis** is a set of linearly-independent vectors, which means that no vector in the set can be written as a weighted sum of the others.

In this context, a vector is a sequence of coordinates defined relative to a basis.

This algorithm is based on a matrix, M, that is orthogonal and symmetric, so it has the unusual property that the inverse of M is M (within a factor of two, anyway). As a result, the function we wrote to compute DCT-IV also computes the inverse DCT.

From

For the analysis problem, we started with a solution that takes time proportional to n^3 and improved it to take time proportional to n^2. It turns out that there is another optimization that gets the run time down to $n \log n$. We'll see how that works in Chapter 7.