O'REILLY®

Think DSP

DIGITAL SIGNAL PROCESSING IN PYTHON

Allen B. Downey

Think DSP
by Allen B. Downey

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Nan Barber and Susan Conant
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-03-01: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491938485* for release details.

978-1-491-93848-5

[FILL IN]

Table of Contents

Preface ..... **xi**

1. Sounds and signals **15**
Periodic signals 16
Spectral decomposition 18
Signals 19
Reading and writing Waves 21
Spectrums 22
Wave objects 23
Signal objects 23
Exercises 25

2. Harmonics **27**
Triangle waves 27
Square waves 30
Aliasing 32
Computing the spectrum 35
Exercises 36

3. Non-periodic signals **39**
Linear chirp 40
Exponential chirp 42
Spectrum of a chirp 43
Spectrogram 44
The Gabor limit 45
Leakage 46
Windowing 48
Implementing spectrograms 50

Exercises 51

4. Noise. 55
Uncorrelated noise 56
Integrated spectrum 58
Brownian noise 60
Pink Noise 64
Gaussian noise 67
Exercises 68

5. Autocorrelation. 71
Correlation 71
Serial correlation 74
Autocorrelation 75
Autocorrelation of periodic signals 77
Correlation as dot product 81
Using NumPy 82
Exercises 83

6. Discrete cosine transform. 85

7. Discrete Fourier Transform. 87

8. Filtering and Convolution. 89

9. Signals and systems. 91

10. Modulation and sampling. 93

Index. 95

TH EDITION

Think DSP

Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Think DSP
by Allen B. Downey

Table of Contents

Preface ..... **xi**

1. Sounds and signals **15**
Periodic signals 16
Spectral decomposition 18
Signals 19
Reading and writing Waves 21
Spectrums 22
Wave objects 23
Signal objects 23
Exercises 25

2. Harmonics **27**
Triangle waves 27
Square waves 30
Aliasing 32
Computing the spectrum 35
Exercises 36

3. Non-periodic signals **39**
Linear chirp 40
Exponential chirp 42
Spectrum of a chirp 43
Spectrogram 44
The Gabor limit 45
Leakage 46
Windowing 48
Implementing spectrograms 50

Exercises 51

4. Noise. 55
Uncorrelated noise 56
Integrated spectrum 58
Brownian noise 60
Pink Noise 64
Gaussian noise 67
Exercises 68

5. Autocorrelation. 71
Correlation 71
Serial correlation 74
Autocorrelation 75
Autocorrelation of periodic signals 77
Correlation as dot product 81
Using NumPy 82
Exercises 83

6. Discrete cosine transform. 85

7. Discrete Fourier Transform. 87

8. Filtering and Convolution. 89

9. Signals and systems. 91

10. Modulation and sampling. 93

Index. 95

About the Author

Preface

Signal processing is one of my favorite topics. It is useful in many areas of science and engineering, and if you understand the fundamental ideas, it provides insight into many things we see in the world, and especially the things we hear.

But unless you studied electrical or mechanical engineering, you probably haven't had a chance to learn about signal processing. The problem is that most books (and the classes that use them) present the material bottom-up, starting with mathematical abstractions like phasors. And they tend to be theoretical, with few applications and little apparent relevance.

The premise of this book is that if you know how to program, you can use that skill to learn other things, and have fun doing it.

With a programming-based approach, I can present the most important ideas right away. By the end of the first chapter, you can analyze sound recordings and other signals, and generate new sounds. Each chapter introduces a new technique and an application you can apply to real signals. At each step you learn how to use a technique first, and then how it works.

This approach is more practical and, I hope you'll agree, more fun.

Who is this book for?

The examples and supporting code for this book are in Python. You should know core Python and you should be familiar with object-oriented features, at least using objects if not defining your own.

If you are not already familiar with Python, you might want to start with my other book, *Think Python*, which is an introduction to Python for people who have never programmed, or Mark Lutz's *Learning Python*, which might be better for people with programming experience.

I use NumPy and SciPy extensively. If you are familiar with them already, that's great, but I will also explain the functions and data structures I use.

I assume that the reader knows basic mathematics, including complex numbers. You don't need much calculus; if you understand the concepts of integration and differentiation, that will do. I use some linear algebra, but I will explain it as we go along.

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 copy the files to 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 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*.

Good luck, and have fun!

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.
- Silas Gyger corrected an arithmetic error.

Special thanks to Freesound, which is the source of many of the sound samples I use in this book, and to the Freesound users who uploaded those sounds. I include some of their wave files in the GitHub repository for this book, using the original file names, so it should be easy to find their sources.

Unfortunately, most Freesound users don't make their real names available, so I can only thank them using their user names. Samples used in this book were contributed by Freesound users: iluppai, wcfl10, thirsk, docquesting, kleeb, landup, zippi1, themusicalnomad, bcjordan, rockwehrmann, marcgascon7, jcveliz. Thank you all!

CHAPTER 1

Sounds and signals

A **signal** represents 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 `chap01.ipynb`, which is in the repository for this book (see "Using the code" on page xii). You can also view it at *http://tinyurl.com/thinkdsp01*.

Periodic signals

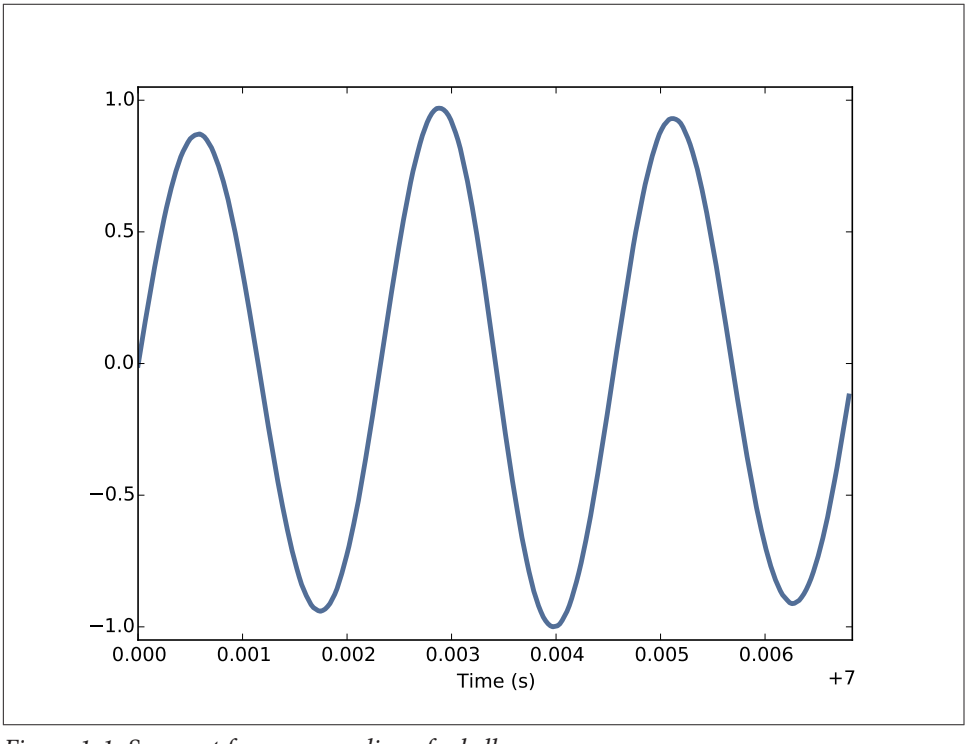

Figure 1-1. Segment from a recording of a bell.

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

This signal resembles 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*.

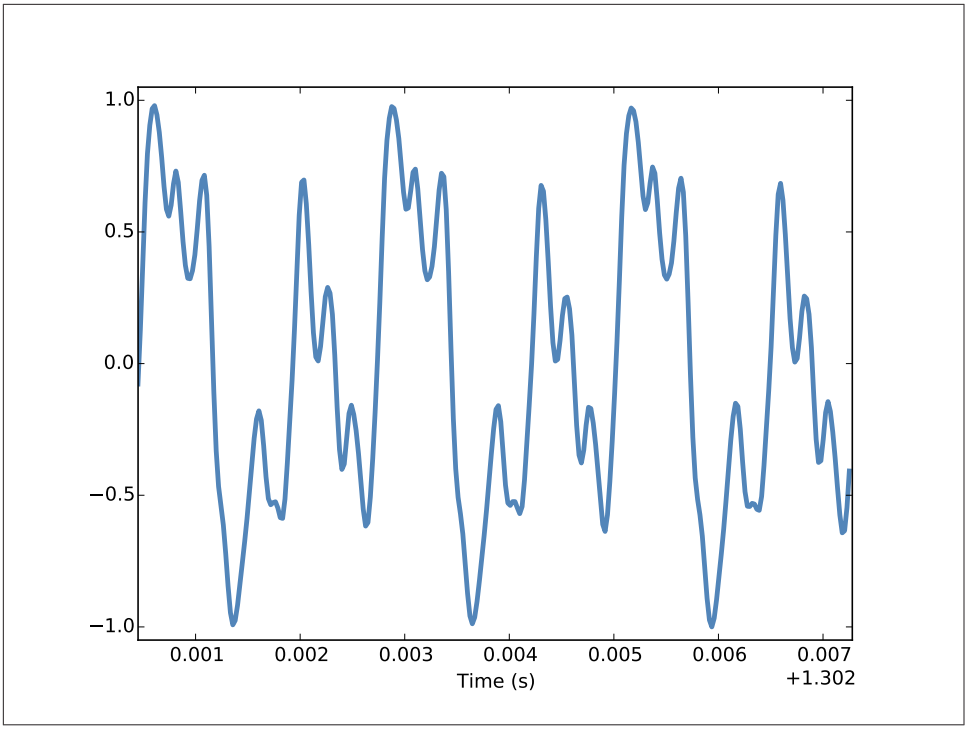

Figure 1-2. Segment from a recording of a violin.

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

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.

Spectral decomposition

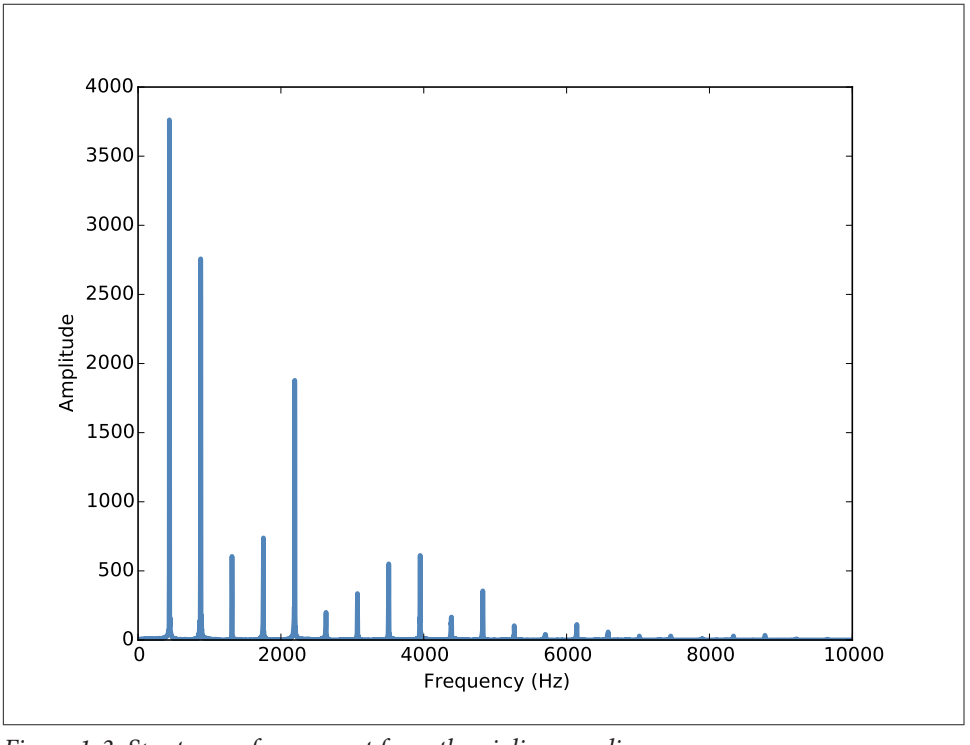

Figure 1-3. Spectrum of a segment from the violin recording.

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

The most important mathematical idea in this book is the **discrete Fourier transform**, or **DFT**, which takes a signal and produces its **spectrum**. The spectrum is the set of sinusoids that add up to produce the signal.

And the most important algorithm in this book is the **Fast Fourier transform**, or **FFT**, which is an efficient way to compute the DFT.

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 or **amplitude** 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 above A5. If you are not familiar with musical intervals like "major fifth", see *https://en.wikipedia.org/wiki/Interval_(music)*.
- 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 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.

Signals

I wrote a Python module called `thinkdsp.py` that contains classes and functions for working with signals and spectrums[1]. You will find it in the repository for this book (see "Using the code" on page xii).

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 defined as the largest amplitude we can record or play back.

1 The plural of "spectrum" is often written "spectra", but I prefer to use standard English plurals. If you are familiar with "spectra", I hope my choice doesn't sound too strange.

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

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 points in time, `ts`, and computing the corresponding values of the signal, `ys`. I represent `ts` and `ys` using NumPy arrays and encapsulate them in an object called a Wave.

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.

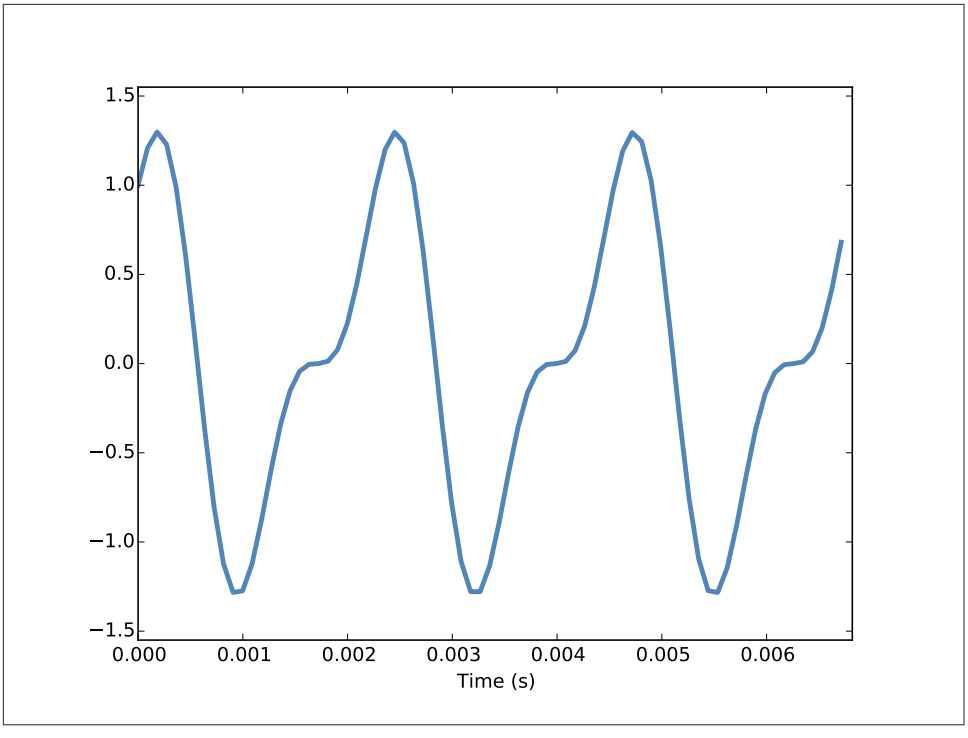

Figure 1-4. Segment from a mixture of two sinusoid signals.

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.

Reading and writing Waves

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

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

And `Wave` provides `write`, which writes a WAV file:

```
wave.write(filename='output.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='output.wav', player='aplay')
```

It uses `aplay` by default, but you can provide the name of another player.

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`. It is included in the Git repository for this book (see "Using the code" on page xii).

`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)
```

A low pass filter removes bright, high-frequency sounds, so the result sounds muffled and darker. To hear what it sounds like, you can convert the Spectrum back to a Wave, and then play it.

```
wave = spectrum.make_wave()
wave.play('temp.wav')
```

The `play` method writes the wave to a file and then plays it. If you use IPython notebooks, you can use `make_audio`, which makes an Audio widget that plays the sound.

Wave objects

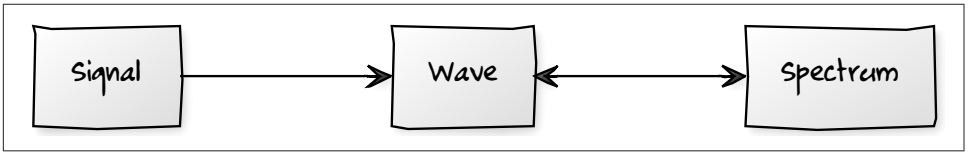

Figure 1-5. Relationships among the classes in `thinkdsp`.

There is nothing very complicated in `thinkdsp.py`. Most of the functions it provides are thin wrappers around functions from NumPy and SciPy.

The primary classes in `thinkdsp` are Signal, Wave, and Spectrum. Given a Signal, you can make a Wave. Given a Wave, you can make a Spectrum, and vice versa. These relationships are shown in Figure 1-5.

A Wave object contains three attributes: `ys` is a NumPy array that contains the values in the signal; `ts` is an array of the times where the signal was evaluated or sampled; and `framerate` is the number of samples per unit of time. The unit of time is usually seconds, but it doesn't have to be. In one of my examples, it's days.

Wave also provides three read-only properties: `start`, `end`, and `duration`. If you modify `ts`, these properties change accordingly.

To modify a wave, you can access the `ts` and `ys` directly. For example:

```
wave.ys *= 2
wave.ts += 1
```

The first line scales the wave by a factor of 2, making it louder. The second line shifts the wave in time, making it start 1 second later.

But Wave provides methods that perform many common operations. For example, the same two transformations could be written:

```
wave.scale(2)
wave.shift(1)
```

You can read the documentation of these methods and others at *http://think-dsp.com/thinkdsp.html*.

Signal objects

Signal is a parent class that provides functions common to all kinds of signals, like `make_wave`. Child classes inherit these methods and provide `evaluate`, which evaluates the signal at a given sequence of times.

For example, Sinusoid is a child class of Signal, with this definition:

```
class Sinusoid(Signal):

    def __init__(self, freq=440, amp=1.0, offset=0, func=np.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`: indicates where in its period the signal starts; `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 `np.sin` or `np.cos`, yielding a sine or cosine signal.

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

Signal provides `make_wave`, which looks like this:

```
    def make_wave(self, duration=1, start=0, framerate=11025):
        n = round(duration * framerate)
        ts = start + np.arange(n) / framerate
        ys = self.evaluate(ts)
        return Wave(ys, ts, framerate=framerate)
```

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

`n` is the number of samples, and `ts` is a NumPy array of sample times.

To compute the `ys`, `make_wave` invokes `evaluate`, is provided by `Sinusoid`:

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

Let's unwind this function one step at time:

1. `self.freq` is frequency in cycles per second, and each element of `ts` is 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 when $t = 0$. It has the effect of shifting the signal left or right in time.
4. If `self.func` is `np.sin` or `np.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:

$$y = A \cos\left(2\pi f t + \phi_0\right)$$

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 expression, but as we'll see, this code provides a framework for dealing with all kinds of signals, not just sinusoids.

Exercises

Before you begin these exercises, you should download the code for this book, following the instructions in "Using the code" on page xii.

Solutions to these exercises are in `chap01soln.ipynb`.

Example 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*.

Example 1-2.

Go to *http://freesound.org* and download a sound sample that includes music, speech, or other sounds that have a well-defined pitch. Select a roughly half-second segment 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?

Example 1-3.

Synthesize a compound signal by creating SinSignal and CosSignal objects and adding them up. Evaluate the signal to get a Wave, and listen to it. Compute its Spectrum and plot it. What happens if you add frequency components that are not multiples of the fundamental?

Example 1-4.

Write a function called `stretch` that takes a Wave and a stretch factor and speeds up or slows down the wave by modifying `ts` and `framerate`. Hint: it should only take two lines of code.

CHAPTER 2

Harmonics

In this chapter I present several new waveforms; we will look and their spectrums to understand their **harmonic structure**, which is the set of sinusoids they are made up of.

I'll also introduce one of the most important phenomena in digital signal processing: aliasing. And I'll explain a little more about how the Spectrum class works.

The code for this chapter is in `chap02.ipynb`, which is in the repository for this book (see "Using the code" on page xii). You can also view it at *http://tinyurl.com/thinkdsp02*.

Triangle waves

A sinusoid contains only one frequency component, so its spectrum 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 spectrums.

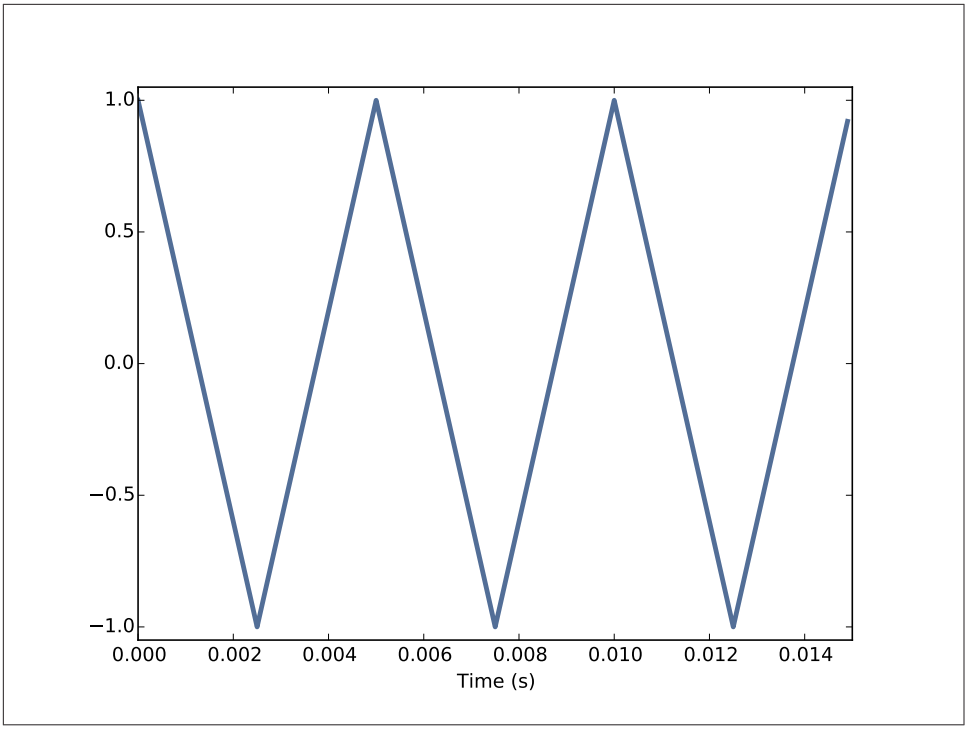

Figure 2-1. Segment of a triangle signal at 200 Hz.

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, _ = np.modf(cycles)
        ys = np.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 many ways to generate a triangle wave. The details are not important, but here's how `evaluate` works:

1. `cycles` is the number of cycles since the start time. `np.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)
signal.plot()
```

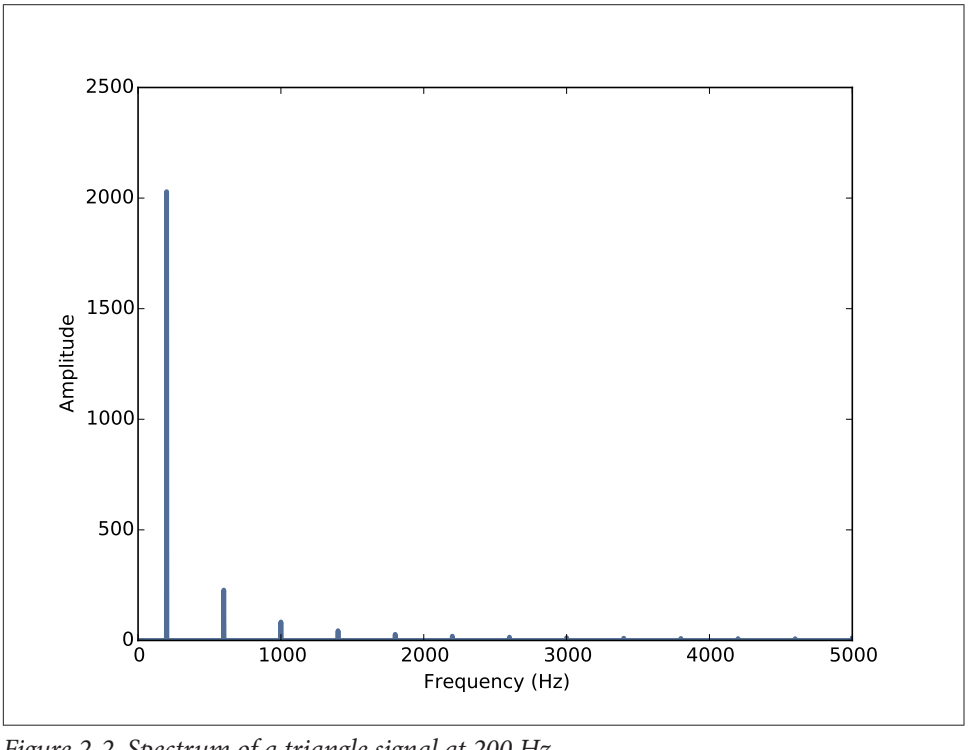

Figure 2-2. Spectrum of a triangle signal at 200 Hz.

Next we can use the Signal to make a Wave, and use the Wave to make a Spectrum:

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

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

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. Their amplitude 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 $1.7^2 = 2.9$. This relationship is called the **harmonic structure**.

Square waves

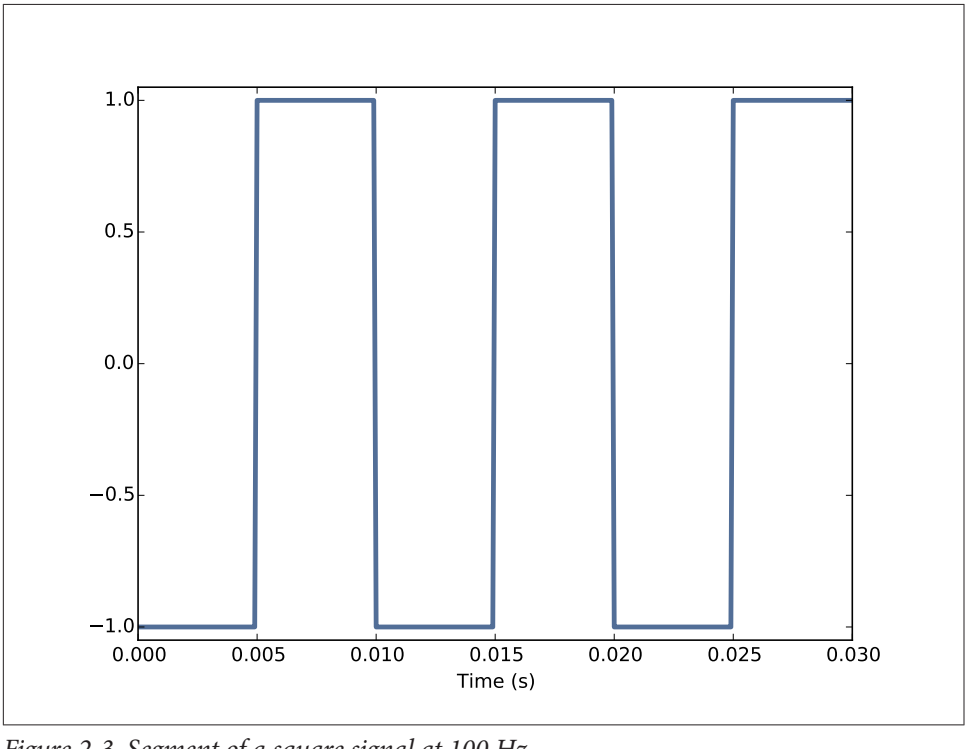

Figure 2-3. Segment of a square signal at 100 Hz.

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, _ = np.modf(cycles)
        ys = self.amp * np.sign(unbias(frac))
        return ys
```

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

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 np.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.

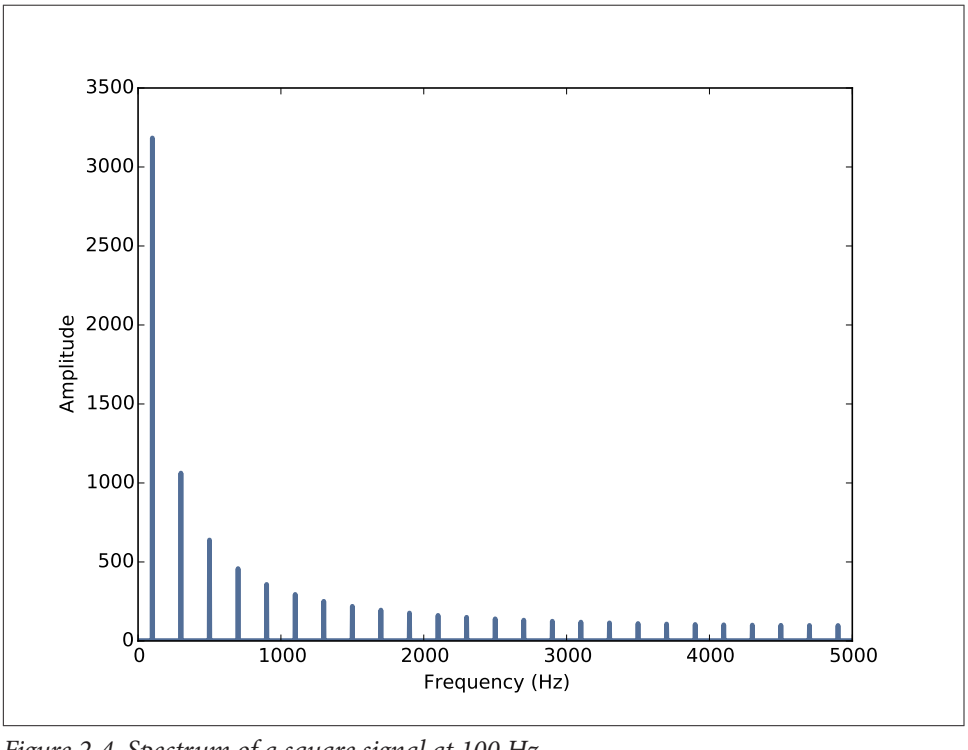

Figure 2-4. Spectrum of a square signal at 100 Hz.

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).

The exercises at the end of this chapter give you a chance to explore other waveforms and other harmonic structures.

Aliasing

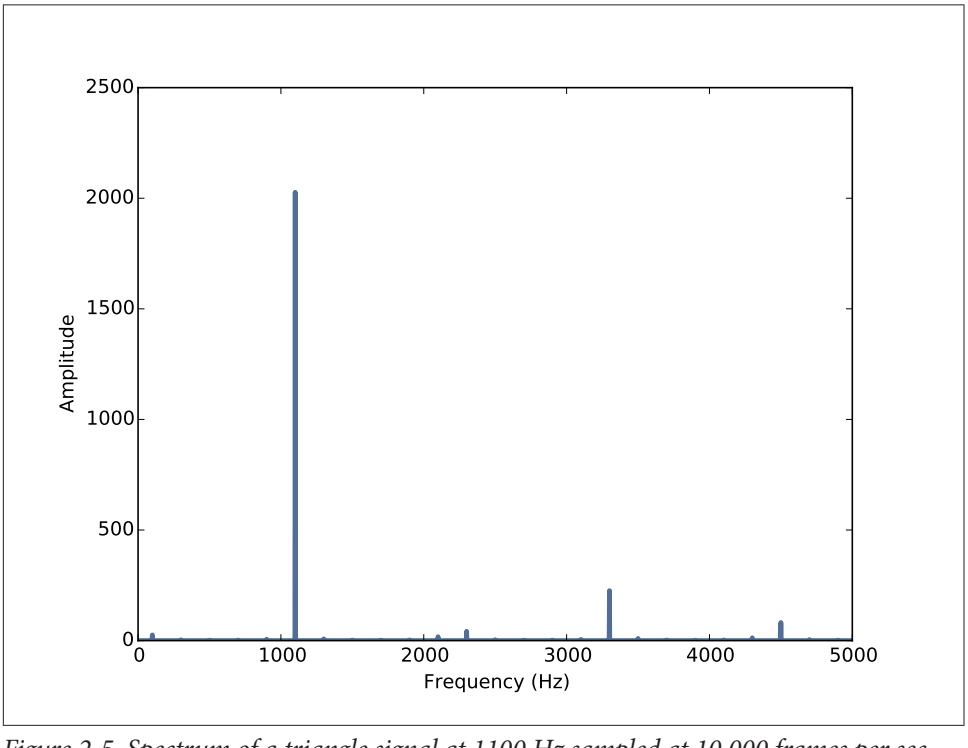

Figure 2-5. Spectrum of a triangle signal at 1100 Hz sampled at 10,000 frames per second.

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. The harmonics of this wave should be at 3300, 5500, 7700, and 9900 Hz.

In the figure, there are peaks at 1100 and 3300 Hz, as expected, but the third peak is at 4500, not 5500 Hz. The fourth peak is at 2300, not 7700 Hz. And if you look 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 happened 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 turns out to be enough, just barely, but if the frequency is higher, it's not.

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()
```

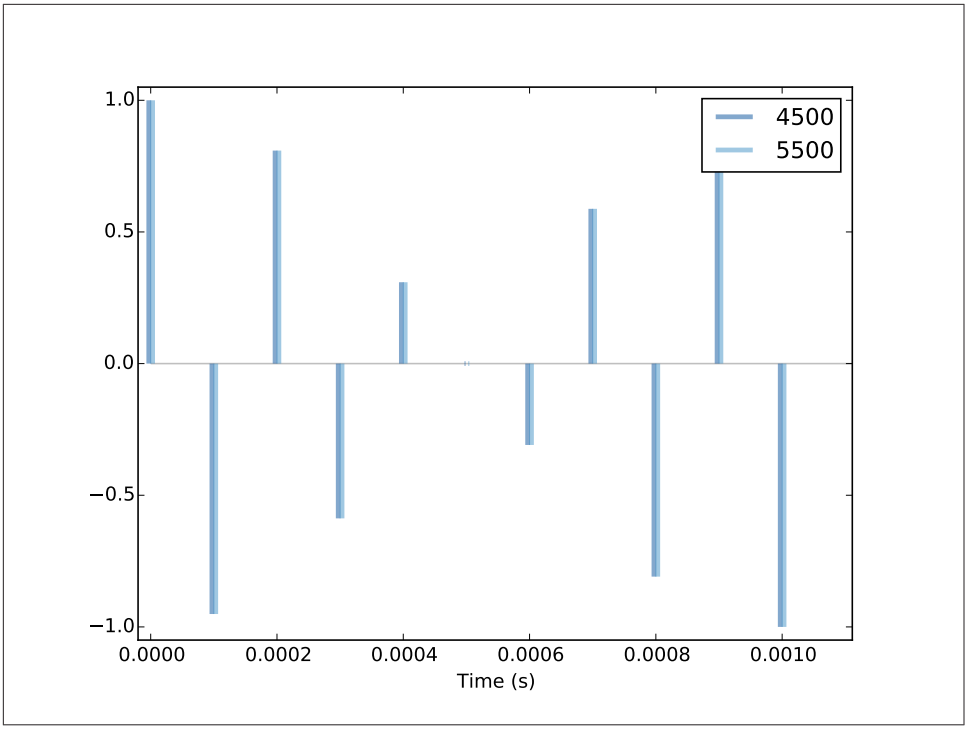

Figure 2-6. Cosine signals at 4500 and 5500 Hz, sampled at 10,000 frames per second. They are identical.

Figure 2-6 shows the result. I plotted the samples using vertical lines, to make it easier to compare the two Waves, and I offset them slightly in time. The problem should be clear: the the two Waves 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 appears to be 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". Is is sometimes also 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.

Computing the spectrum

We have seen the Wave method `make_spectrum` several times. Here is the implementation (leaving out some details we'll get to later):

```
from np.fft import rfft, rfftfreq

# class Wave:
    def make_spectrum(self):
        n = len(self.ys)
        d = 1 / self.framerate

        hs = rfft(self.ys)
        fs = rfftfreq(n, d)

        return Spectrum(hs, fs, self.framerate)
```

The parameter `self` is a Wave object. `n` is the number of samples in the wave, and `d` is the inverse of the frame rate, which is the time between samples.

`np.fft` is the NumPy module that provides functions related to the **Fast Fourier Transform** (FFT), which is an efficient algorithm that computes the Discrete Fourier Transform (DFT).

`make_spectrum` uses `rfft`, which stands for "real FFT", because the Wave contains real values, not complex. Later we'll see the full FFT, which can handle complex signals. The result of `rfft`, which I call `hs`, is a NumPy array of complex numbers that represents the amplitude and phase offset of each frequency component in the wave.

The result of `rfftfreq`, which I call `fs`, is an array that contains frequencies corresponding to the `hs`.

To understand the values in `hs`, consider these 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 value in `hs` corresponds to a frequency component: its magnitude is proportional to the amplitude of the corresponding component; its angle is the phase offset.

The Spectrum class provides two read-only properties, `amps` and `angles`, which return NumPy arrays representing the magnitudes and angles of the `hs`. When we

plot a Spectrum object, we usually plot `amps` versus `fs`. Sometimes it is also useful to plot `angles` versus `fs`.

Although it might be tempting to look at the real and imaginary parts of `hs`, you will almost never need to. I encourage you to think of the DFT as a vector of amplitudes and phase offsets that happen to be encoded in the form of complex numbers.

To modify a Spectrum, you can access the `hs` directly. For example:

```
spectrum.hs *= 2
spectrum.hs[spectrum.fs > cutoff] = 0
```

The first line multiples the elements of `hs` by 2, which doubles the amplitudes of all components. The second line sets to 0 only the elements of `hs` where the corresponding frequency exceeds some cutoff frequency.

But Spectrum also provides methods to perform these operations:

```
spectrum.scale(2)
spectrum.low_pass(cutoff)
```

You can read the documentation of these methods and others at *http://think-dsp.com/thinkdsp.html*.

At this point you should have a better idea of how the Signal, Wave, and Spectrum classes work, but I have not explained how the Fast Fourier Transform works. That will take a few more chapters.

Exercises

Solutions to these exercises are in `chap02soln.ipynb`.

Example 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*.

Example 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?

Example 2-3.

Make a square signal at 1100 Hz and make a wave that samples it at 10000 frames per second. If you plot the spectrum, you can see that most of the harmonics are aliased. When you listen to the wave, can you hear the aliased harmonics?

Example 2-4.

If you have a spectrum object, `spectrum`, and print the first few values of `spectrum.fs`, you'll see that they start at zero. So `spectrum.hs[0]` is the magnitude of the component with frequency 0. But what does that mean?

Try this experiment:

1. Make a triangle signal with frequency 440 and make a Wave with duration 0.01 seconds. Plot the waveform.
2. Make a Spectrum object and print `spectrum.hs[0]`. What is the amplitude and phase of this component?
3. Set `spectrum.hs[0] = 100`. Make a Wave from the modified Spectrum and plot it. What effect does this operation have on the waveform?

Example 2-5.

Write a function that takes a Spectrum as a parameter and modifies it by dividing each element of `hs` by the corresponding frequency from `fs`. Hint: since division by zero is undefined, you might want to set `spectrum.hs[0] = 0`.

Test your function using a square, triangle, or sawtooth wave.

1. Compute the Spectrum and plot it.
2. Modify the Spectrum using your function and plot it again.
3. Make a Wave from the modified Spectrum and listen to it. What effect does this operation have on the signal?

Example 2-6.

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 $1/f^2$?

Hint: There are two ways you could approach this: you could construct the signal you want by adding up sinusoids, or you could start with a signal that is similar to what you want and modify it.

CHAPTER 3
Non-periodic signals

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.

This chapter also presents spectrograms, a common way to visualize non-periodic signals.

The code for this chapter is in `chap03.ipynb`, which is in the repository for this book (see "Using the code" on page xii). You can also view it at *http://tinyurl.com/thinkdsp03*.

Linear chirp

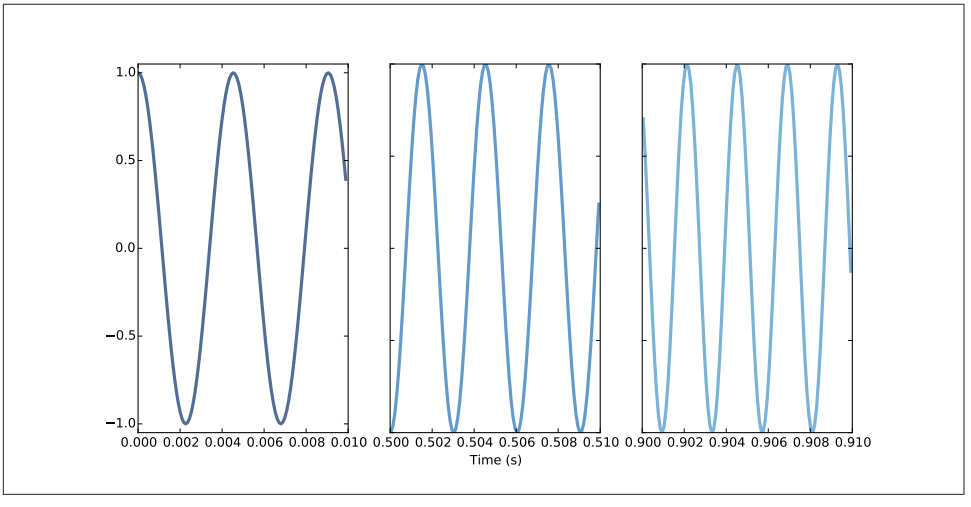

Figure 3-1. Chirp waveform near the beginning, middle, and end.

We'll start with a **chirp**, which is a signal with variable frequency. `thinkdsp` provides a Signal called Chirp that makes a sinusoid that sweeps linearly through a range of frequencies.

Here's an example what sweeps from 220 to 880 Hz, which is two octaves from A3 to A5:

```
signal = thinkdsp.Chirp(start=220, end=880)
wave = signal.make_wave()
```

Figure 3-1 shows segments of this wave near the beginning, middle, and end. It's clear that the frequency is increasing.

Before we go on, let's see how Chirp is implemented. Here is the class definition:

```
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.

Here is the function that evaluates the signal:

```
    def evaluate(self, ts):
        freqs = np.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; to keep this function simple, I assume they are equally-spaced.

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, I use `np.linspace`, which returns a NumPy array of $n - 1$ values between `start` and `end`.

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

```
def _evaluate(self, ts, freqs):
    dts = np.diff(ts)
    dphis = PI2 * freqs * dts
    phases = np.cumsum(dphis)
    phases = np.insert(phases, 0, 0)
    ys = self.amp * np.cos(phases)
    return ys
```

`np.diff` computes the difference between adjacent elements of `ts`, returning the length of each interval in seconds. If 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. In "Signal objects" on page 23 we saw that when frequency is constant, the phase, φ, increases linearly over time:

$$\phi = 2\pi f t$$

When frequency is a function of time, the *change* in phase during a short time interval, Δt is:

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

In Python, since `freqs` contains $f(t)$ and `dts` contains the time intervals, we can write

```
dphis = PI2 * freqs * dts
```

Now, since `dphis` contains the changes in phase, we can get the total phase at each timestep by adding up the changes:

```
phases = np.cumsum(dphis)
phases = np.insert(phases, 0, 0)
```

`np.cumsum` computes the cumulative sum, which is almost what we want, but it doesn't start at 0. So I use `np.insert` to add a 0 at the beginning.

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.

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, `np.cos` computes the amplitude of the wave as a function of phase (remember that phase is expressed in radians).

If you know calculus, you might notice that the limit as Δt gets small is

$$d\phi = 2\pi f(t)dt$$

Dividing through by dt yields

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

In other words, frequency is the derivative of phase. Conversely, phase is the integral of frequency. When we used `cumsum` to go from frequency to phase, we were approximating integration.

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 = np.log10(self.start), np.log10(self.end)
        freqs = np.logspace(start, end, len(ts)-1)
        return self._evaluate(ts, freqs)
```

Instead of `np.linspace`, this version of evaluate uses `np.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)
wave = signal.make_wave(duration=1)
```

You can listen to these examples in `chap03.ipynb` and compare the linear and exponential chirps.

Spectrum of a chirp

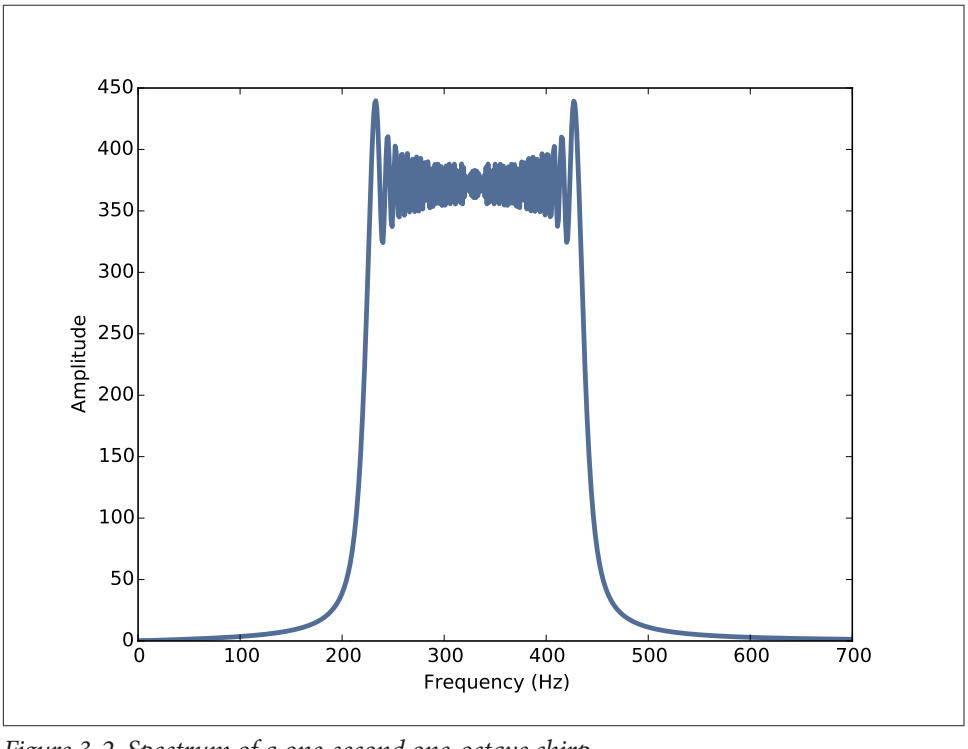

Figure 3-2. Spectrum of a one-second one-octave chirp.

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

Figure 3-2 shows the result. The spectrum has components at every frequency from 220 to 440 Hz, with variations 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.

Spectrogram

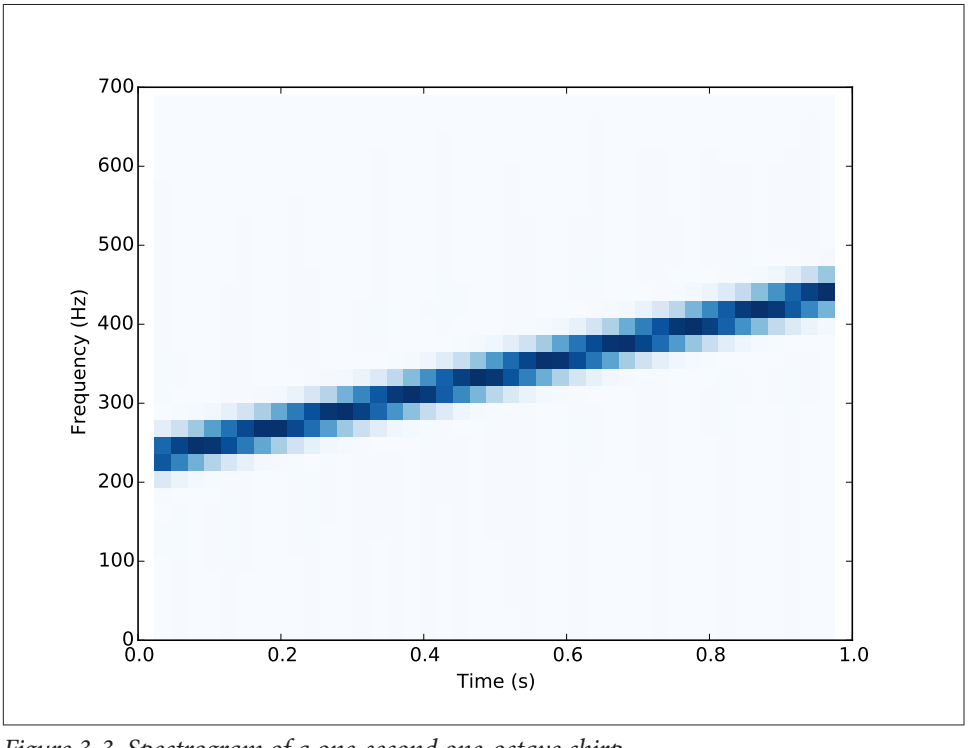

Figure 3-3. Spectrogram of a one-second one-octave chirp.

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.

As an example, I'll compute the spectrogram of this chirp:

```
signal = thinkdsp.Chirp(start=220, end=440)
wave = signal.make_wave(duration=1, framerate=11025)
```

`Wave` provides `make_spectrogram`, which returns a `Spectrogram` object:

```
spectrogram = wave.make_spectrogram(seg_length=512)
spectrogram.plot(high=700)
```

`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 3-3 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 5512.5 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.

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 5512.5 Hz, so the range between components is 21.6 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 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.

Leakage

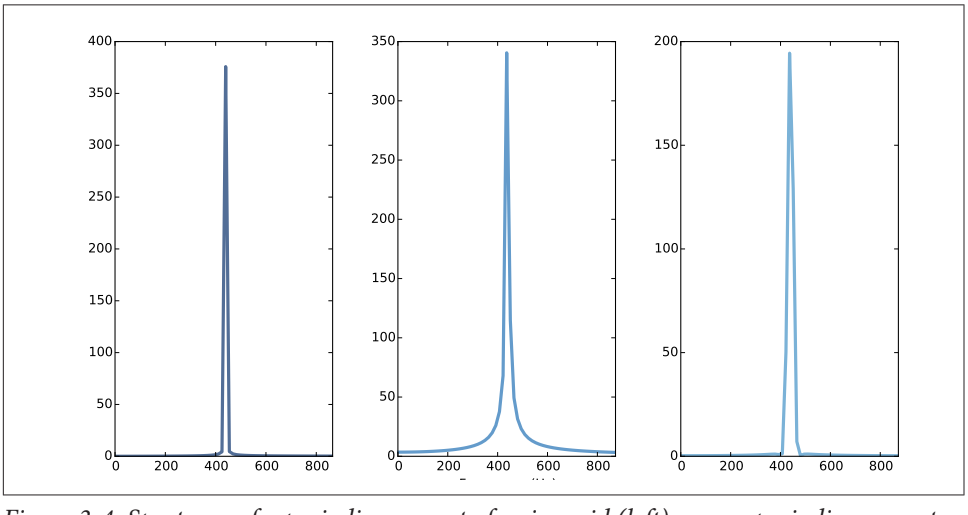

Figure 3-4. Spectrum of a periodic segment of a sinusoid (left), a non-periodic segment (middle), a windowed non-periodic segment (right).

In order to explain how `make_spectrogram` works, I have to explain windowing; and in order to explain windowing, I have to show you the problem it is meant to address, which is leakage.

The Discrete Fourier Transform (DFT), which we use to compute Spectrums, treats waves as if they are periodic; that is, it assumes that the finite segment it operates on is a complete period from an infinite signal that repeats over all time. In practice, this assumption is often false, which creates problems.

One common problem is discontinuity at the beginning and end of the segment. Because DFT 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 signal 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 DFT behaves well.

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

Figure 3-4 (left) shows the result. As expected, there is a single peak at 440 Hz.

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 3-4 (middle) 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 DFT on a segment that becomes discontinuous when we treat it as periodic.

Windowing

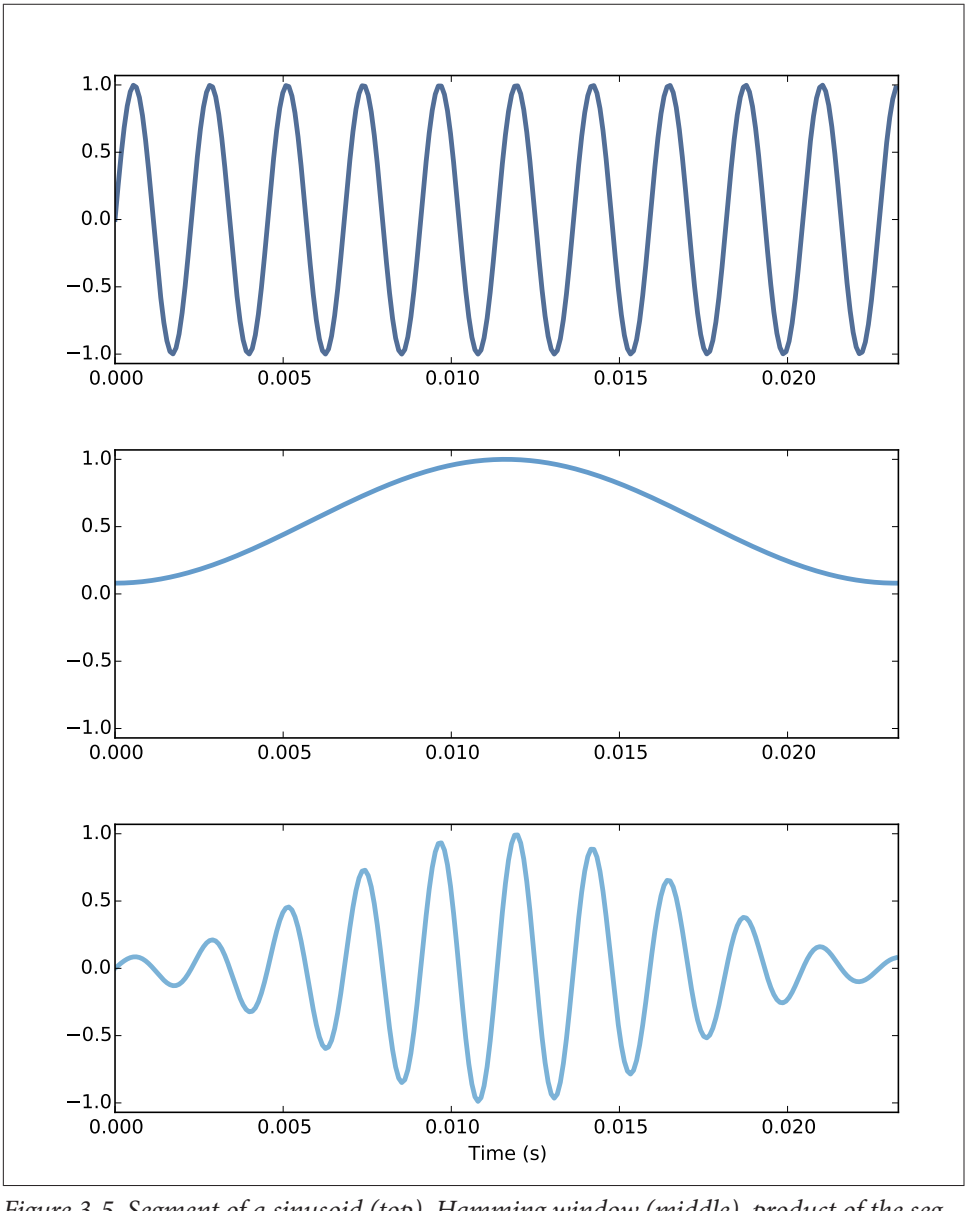

Figure 3-5. Segment of a sinusoid (top), Hamming window (middle), product of the segment and the window (bottom).

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 3-5 (top) shows a segment where the end does not connect smoothly to the beginning.

Figure 3-5 (middle) shows a "Hamming window", one of the more common window functions. No window function is perfect, but some can be shown to be optimal for different applications, and Hamming is a good, all-purpose window.

Figure 3-5 (bottom) 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 3-4 (right) shows the spectrum of the windowed signal. Windowing has reduced leakage substantially, but not completely.

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

```
#class Wave:
    def window(self, window):
        self.ys *= window
```

And NumPy provides `hamming`, which computes a Hamming window with a given length:

```
window = np.hamming(len(wave))
wave.window(window)
```

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.

Implementing spectrograms

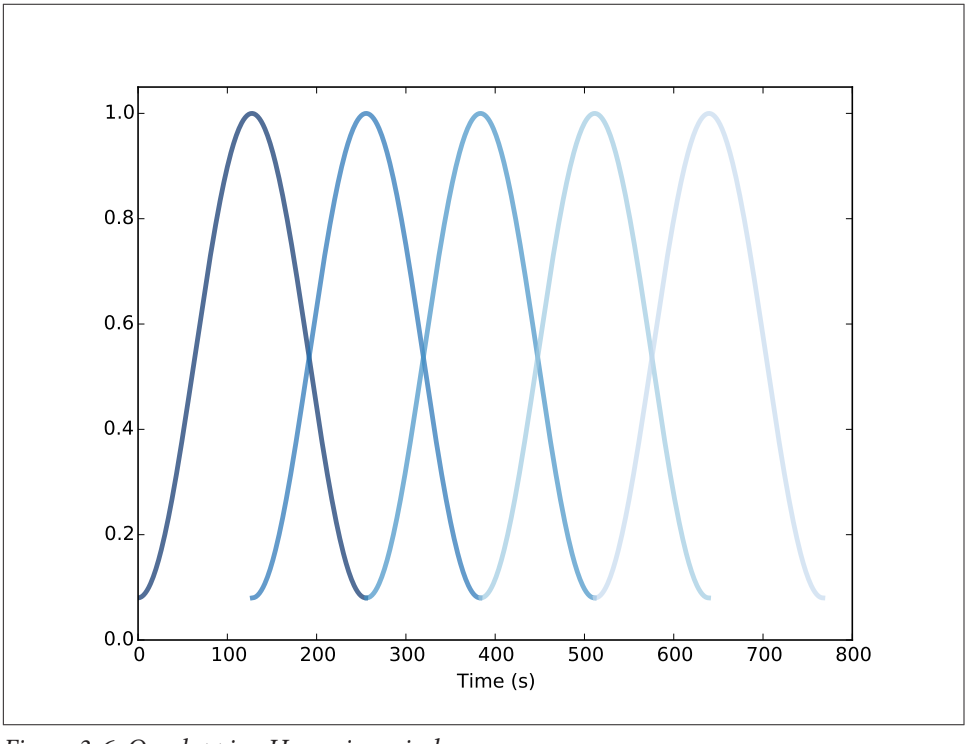

Figure 3-6. Overlapping Hamming windows.

Now that we understand windowing, we can understand the implementation of spectrogram. Here is the `Wave` method that computes spectrograms:

```
#class Wave:
    def make_spectrogram(self, seg_length):
        window = np.hamming(seg_length)
        i, j = 0, seg_length
        step = seg_length / 2

        spec_map = {}

        while j < len(self.ys):
            segment = self.slice(i, j)
            segment.window(window)

            t = (segment.start + segment.end) / 2
            spec_map[t] = segment.make_spectrum()

            i += step
            j += step
```

```
        return Spectrogram(spec_map, seg_length)
```

This is the longest function in the book, so if you can handle this, you can handle anything.

The parameter, `self`, is a Wave object. `seg_length` is the number of samples in each segment.

`window` is a Hamming window with the same length as the segments.

`i` and `j` are the slice indices that select segments from the wave. `step` is the offset between segments. Since `step` is half of `seg_length`, the segments overlap by half. Figure 3-6 shows what these overlapping windows look like.

`spec_map` is a dictionary that maps from a timestamp to a Spectrum.

Inside the while loop, we select a slice from the wave and apply the window; then we construct a Spectrum object and add it to `spec_map`. The nominal time of each segment, `t`, is the midpoint.

Then we advance `i` and `j`, and continue as long as `j` doesn't go past the end of the Wave.

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 with time along the x-axis and frequency along the y-axis.

And that's how Spectrograms are implemented.

Exercises

Solutions to these exercises are in `chap03soln.ipynb`.

Example 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*

Example 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.

Example 3-3.

Make a sawtooth chirp that sweeps from 2500 to 3000 Hz, then use it to make a wave with duration 1 s and framerate 20 kHz. Draw a sketch of what you think the spectrum will look like. Then plot the spectrum and see if you got it right.

Example 3-4.

In musical terminology, a "glissando" is a note that slides from one pitch to another, so it is similar to a chirp.

Find or make a recording of a glissando and plot a spectrogram of the first few seconds. One suggestion: George Gershwin's *Rhapsody in Blue* starts with a famous clarinet glissando, which you can download from *http://archive.org/details/rhapblue11924*.

Example 3-5.

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?

Write a class called `TromboneGliss` that extends `Chirp` and provides `evaluate`. Make a wave that simulates a trombone glissando from C3 up to F3 and back down to C3. C3 is 262 Hz; F3 is 349 Hz.

Plot a spectrogram of the resulting wave. Is a trombone glissando more like a linear or exponential chirp?

Example 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 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 `chap04.ipynb`, which is in the repository for this book (see "Using the code" on page xii). You can also view it at *http://tinyurl.com/thinkdsp04*.

Uncorrelated noise

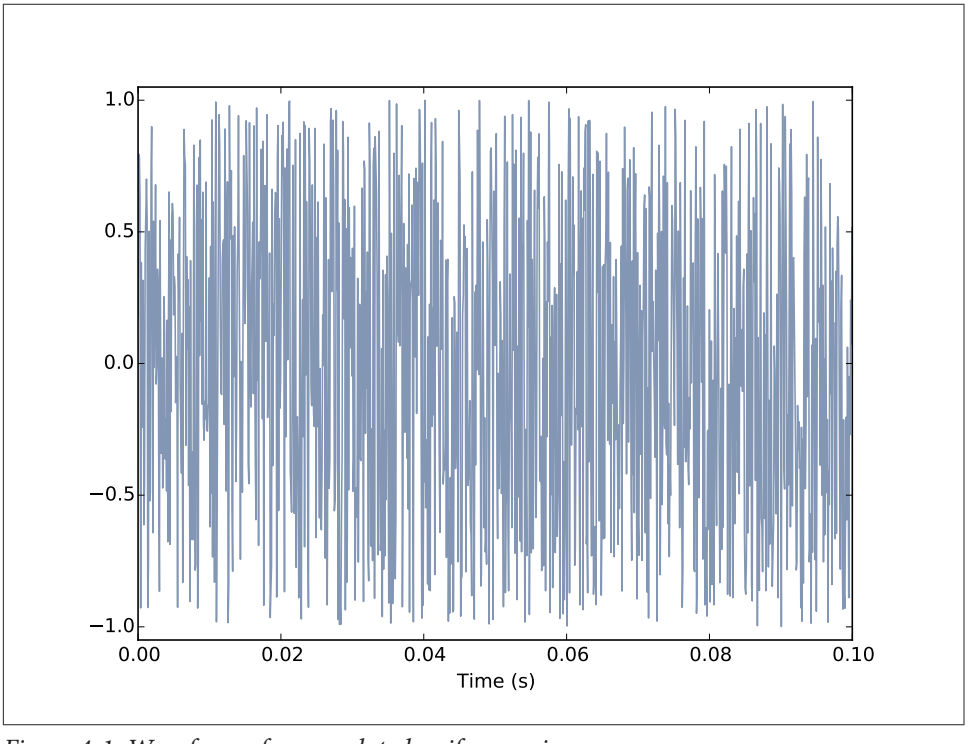

Figure 4-1. Waveform of uncorrelated uniform noise.

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.

Here's a class that represents UU noise:

```
class UncorrelatedUniformNoise(_Noise):

    def evaluate(self, ts):
        ys = np.random.uniform(-self.amp, self.amp, len(ts))
        return ys
```

`UncorrelatedUniformNoise` inherits from `_Noise`, which inherits from `Signal`.

As usual, the evaluate function takes `ts`, the times when the signal should be evaluated. It uses `np.random.uniform`, which generates values from a uniform distribution. In this example, the values are in the range between `-amp` to `amp`.

The following example generates UU noise with duration 0.5 seconds at 11,025 samples per second.

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

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.

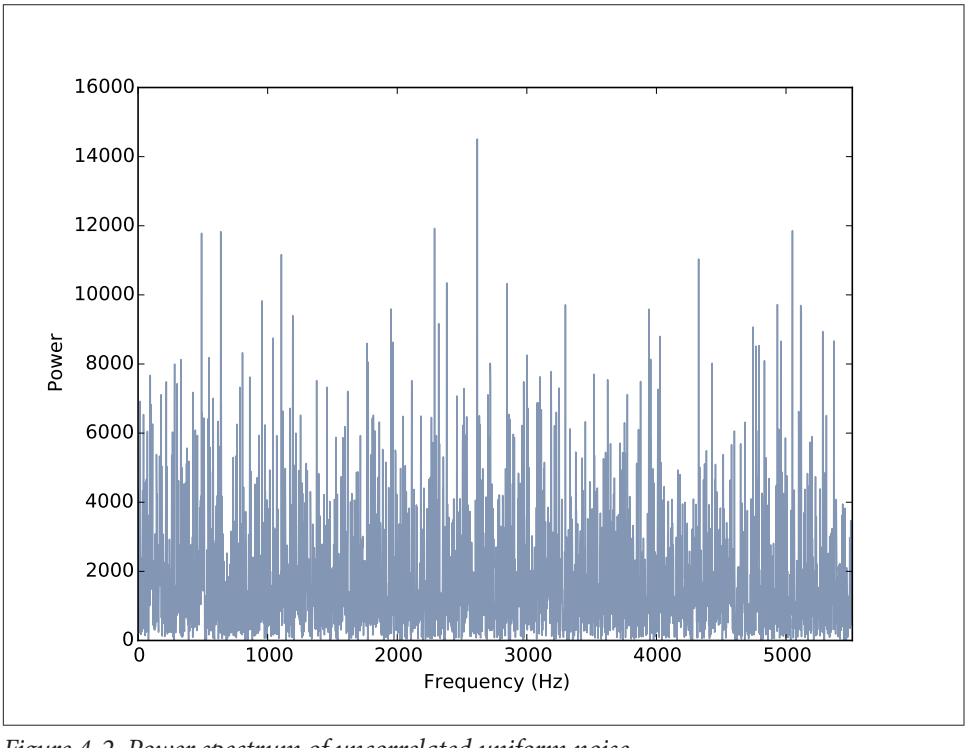

Figure 4-2. Power spectrum of uncorrelated uniform noise.

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 instead of amplitude. Power is the square of amplitude. I am switching from amplitude to power in this chapter because it is more conventional in the context of noise.

Figure 4-2 shows the result. Like the signal, the spectrum looks pretty random. In fact, it *is* random, 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:

- 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, the average power is the same for all frequencies. 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 $1/f$.

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 power in the spectrum up to f.

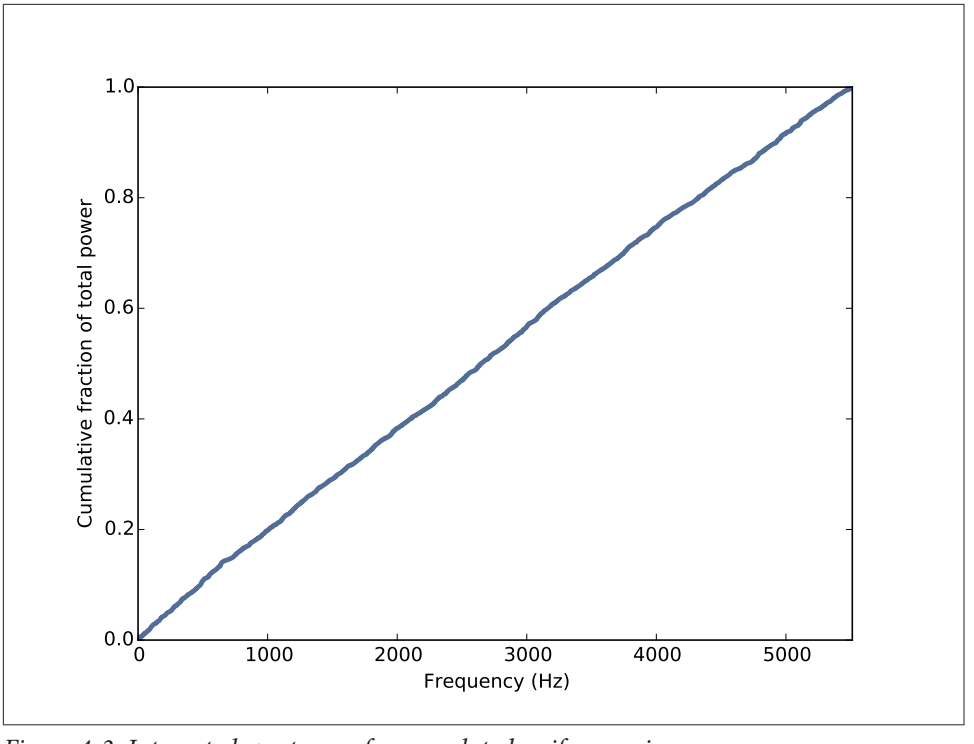

Figure 4-3. Integrated spectrum of uncorrelated uniform noise.

`Spectrum` provides a method that computes the IntegratedSpectrum:

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

`self.power` is a NumPy array containing power for each frequency. `np.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()
```

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.

Brownian noise

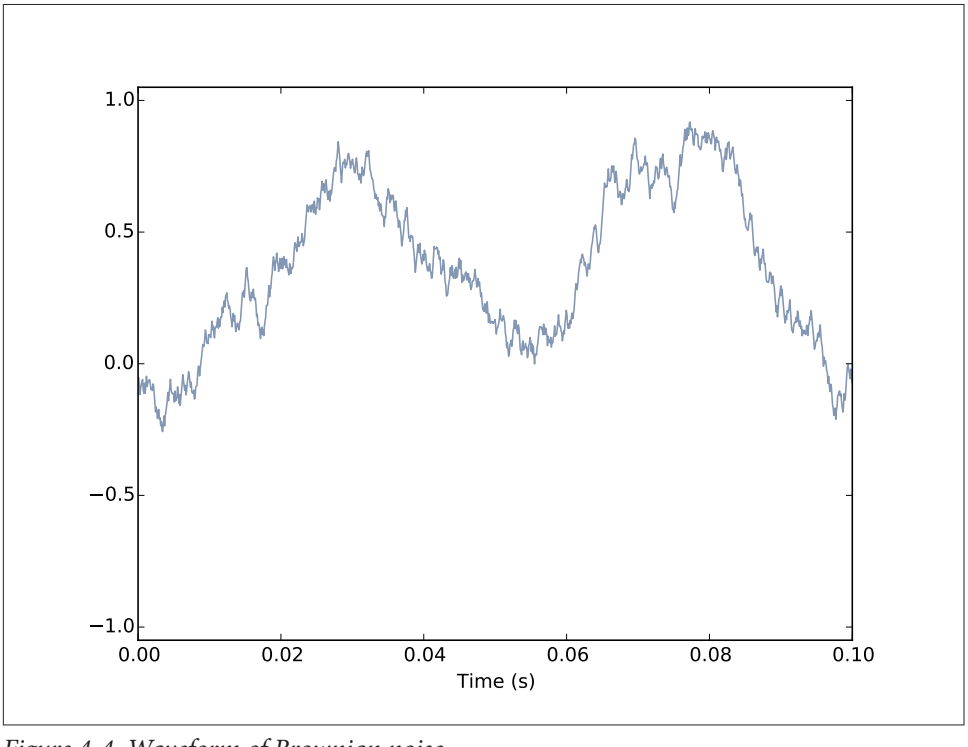

Figure 4-4. Waveform of Brownian noise.

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:

```
class BrownianNoise(_Noise):

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

`evaluate` uses `np.random.uniform` to generate an uncorrelated signal and `np.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.

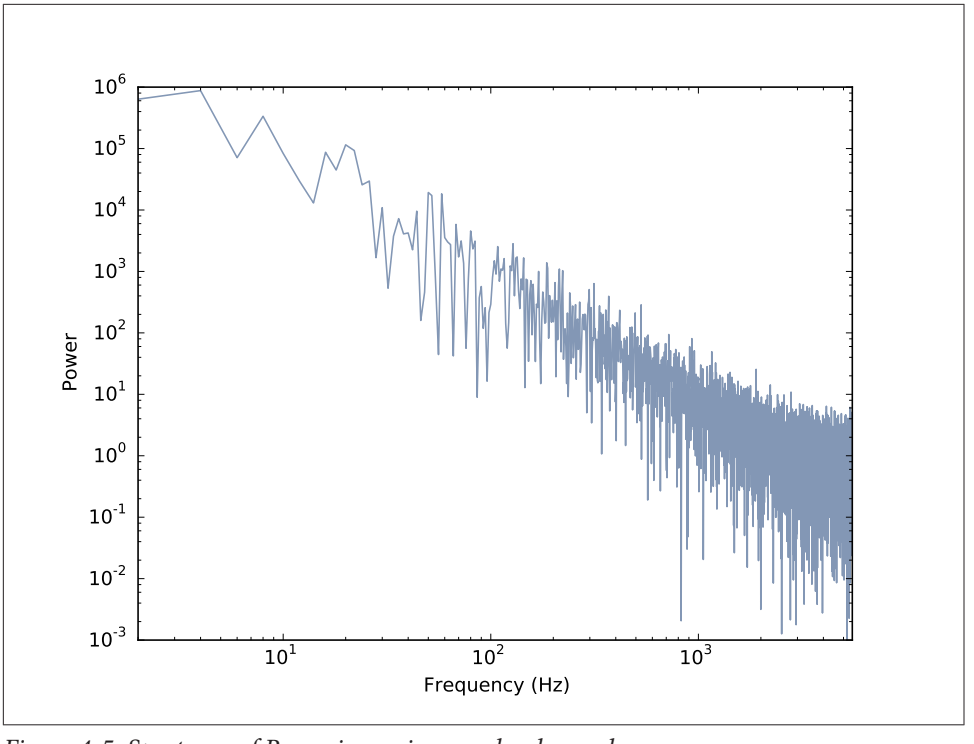

Figure 4-5. Spectrum of Brownian noise on a log-log scale.

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 more clearly, we can plot power and frequency on a log-log scale. Here's the code:

```
spectrum = wave.make_spectrum()
spectrum.plot_power(linewidth=1, alpha=0.5)
thinkplot.config(xscale='log', yscale='log')
```

The result is in Figure 4-5. The relationship between power and frequency is noisy, but roughly linear.

`Spectrum` provides `estimate_slope`, which uses SciPy to compute a least squares fit to the power spectrum:

```
#class Spectrum

    def estimate_slope(self):
        x = np.log(self.fs[1:])
        y = np.log(self.power[1:])
```

```
    t = scipy.stats.linregress(x,y)
    return t
```

It discards the first component of the spectrum because this component corresponds to $f = 0$, and log 0 is undefined.

`estimate_slope` returns the result from `scipy.stats.linregress` which is an object that contains the estimated slope and intercept, coefficient of determination (R^2), p-value, and standard error. For our purposes, we only need the slope.

For Brownian noise, the slope of the power spectrum is -2 (we'll see why in ???), 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 important for our purposes. Exponentiating both sides yields:

$$P = K/f^2$$

where K is e^k, but still not important. More relevant 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 power proportional to $1/f^2$, 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.

Pink Noise

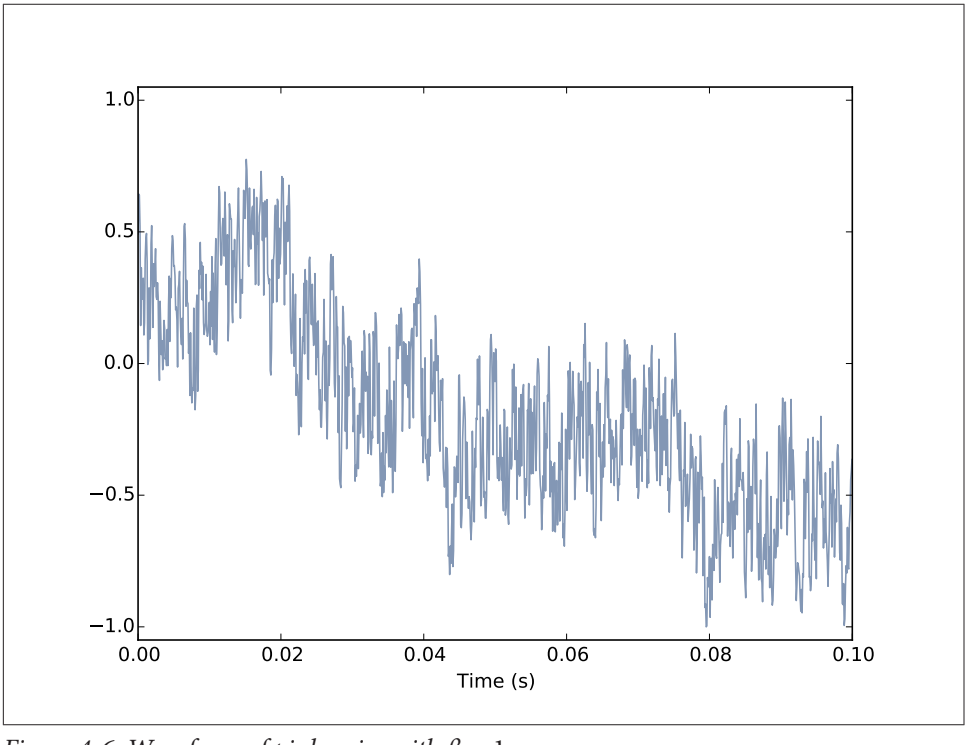

Figure 4-6. Waveform of pink noise with $\beta = 1$.

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.

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(self.amp)
        return wave2
```

`duration` is the length of the wave in seconds. `start` is the start time of the wave; it is included so that `make_wave` has the same interface for all types of signal, but for random noise, start time is irrelevant. And `framerate` is the number of samples per second.

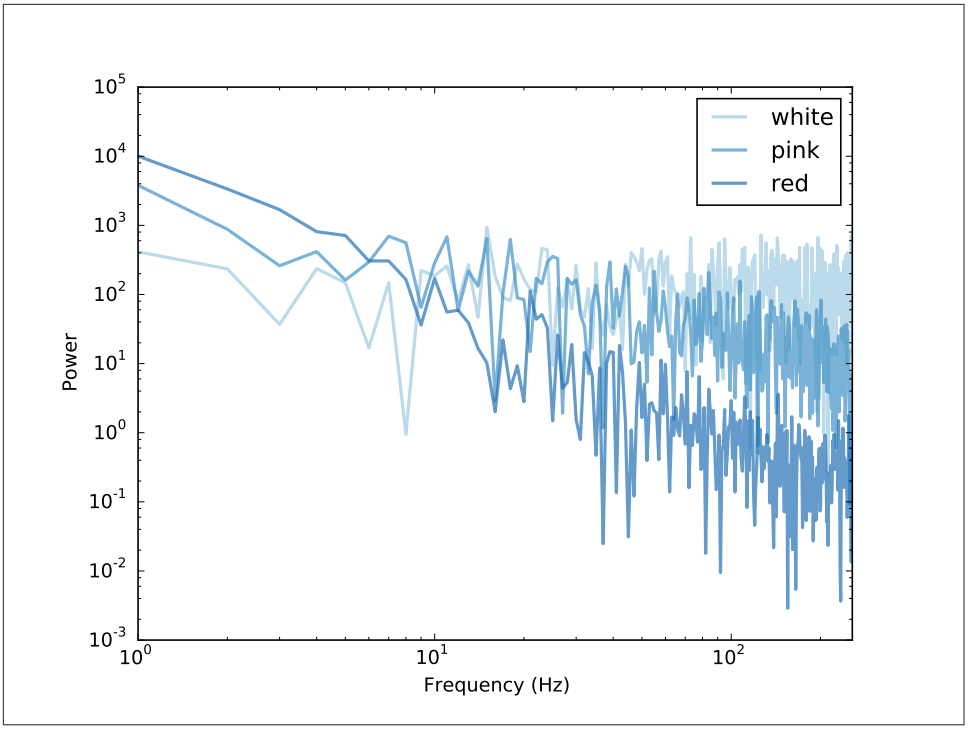

Figure 4-7. Spectrum of white, pink, and red noise on a log-log scale.

`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, and partly 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.

Gaussian noise

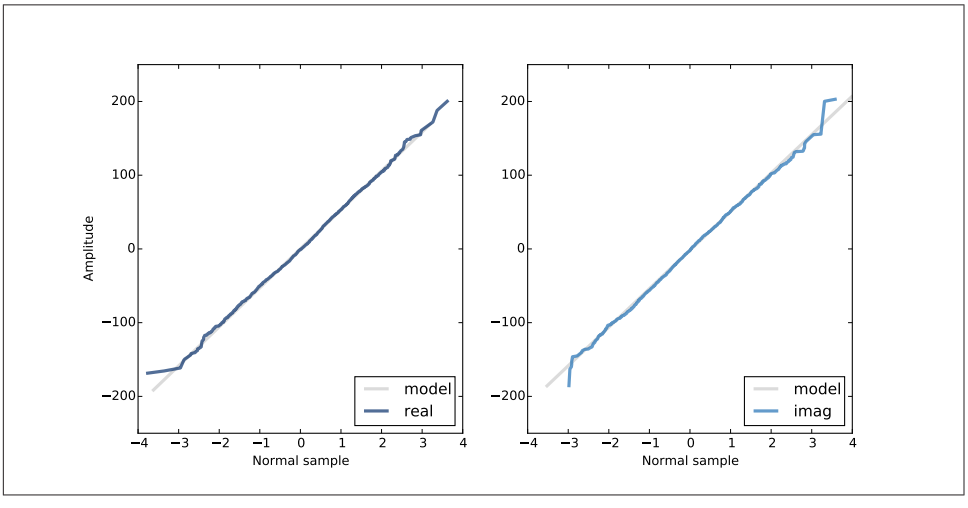

Figure 4-8. Normal probability plot for the real and imaginary parts of the spectrum of Gaussian noise.

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 = np.random.normal(0, self.amp, len(ts))
        return ys
```

`np.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 prop-

erty: 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 Chapter 5 of *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.

Exercises

Solutions to these exercises are in `chap04soln.ipynb`.

Example 4-1.

"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?

Example 4-2.

In a 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 spectrum 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*.

Implement Bartlett's method and use it to estimate the power spectrum for a noise wave. Hint: look at the implementation of `make_spectrogram`.

Example 4-3.

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

Example 4-4.

A Geiger counter is a device that detects radiation. When an ionizing particle strikes the detector, it outputs a surge of current. The total output at a point in time can be modeled as uncorrelated Poisson (UP) noise, where each sample is a random quantity from a Poisson distribution, which corresponds to the number of particles detected during an interval.

Write a class called `UncorrelatedPoissonNoise` that inherits from `thinkdsp._Noise` and provides `evaluate`. It should use `np.random.poisson` to generate random values from a Poisson distribution. The parameter of this function, `lam`, is the average number of particles during each interval. You can use the attribute `amp` to specify `lam`. For example, if the framerate is 10 kHz and `amp` is 0.001, we expect about 10 "clicks" per second.

Generate about a second of UP noise and listen to it. For low values of `amp`, like 0.001, it should sound like a Geiger counter. For higher values it should sound like white noise. Compute and plot the power spectrum to see whether it looks like white noise.

Example 4-5.

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.

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.

The code for this chapter is in `chap05.ipynb`, which is in the repository for this book (see "Using the code" on page xii). You can also view it at *http://tinyurl.com/thinkdsp05*.

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:

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

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 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 tends to be 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 is probably weak, so if you know one, it doesn't tell you much about the others,

I say "probably weak" because it is also possible that there is a nonlinear relationship that is not captured by the 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. `np.corrcoef` takes any number of variables and computes a **correlation matrix** that includes correlations between each pair of variables.

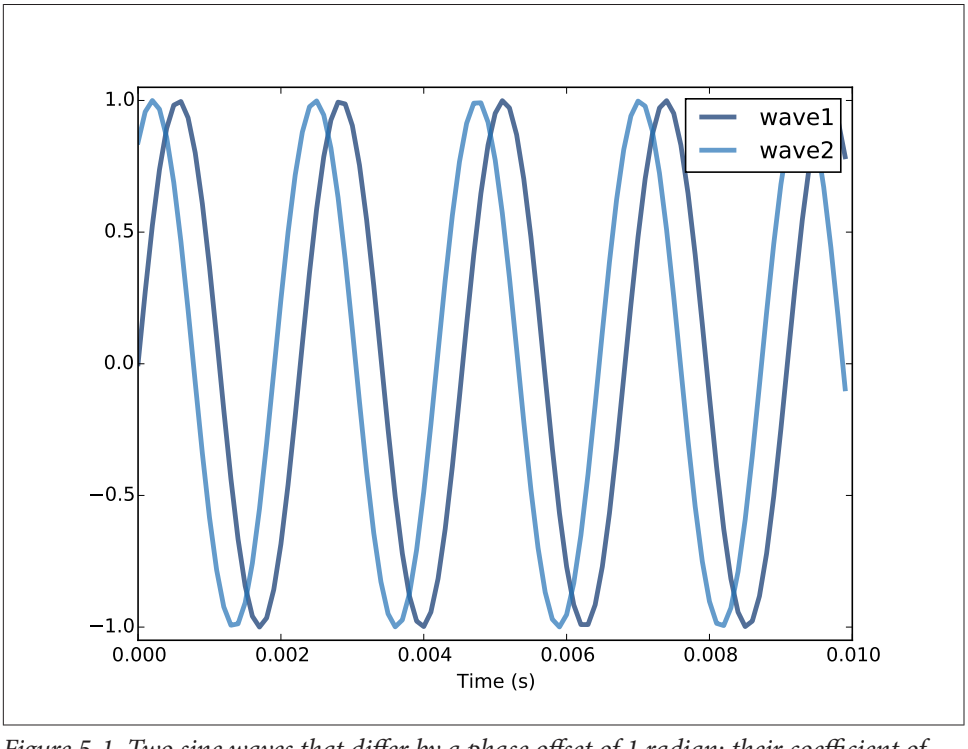

Figure 5-1. Two sine waves that differ by a phase offset of 1 radian; their coefficient of correlation is 0.54.

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

```
def make_sine(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_sine(offset=0)
wave2 = make_sine(offset=1)
```

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

The option `ddof=0` indicates that `corrcoef` should divide by N, as in the equation above, rather than use 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.

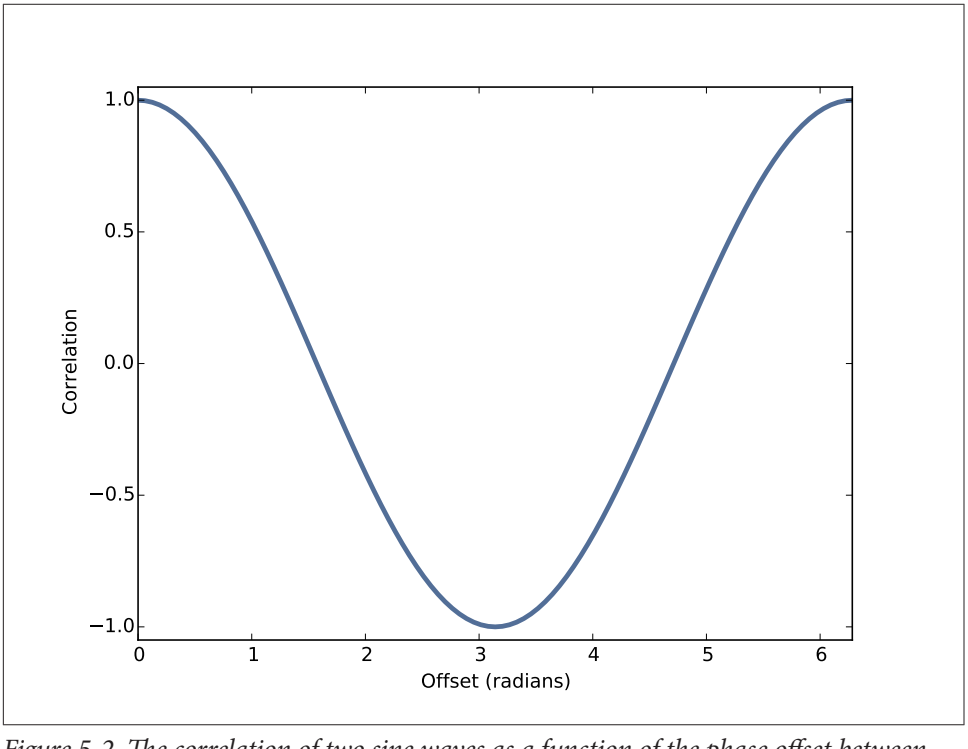

Figure 5-2. The correlation of two sine waves as a function of the phase offset between them. The result is a cosine.

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

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

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 correspond to 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 serial_corr(wave, lag=1):
    n = len(wave)
    y1 = wave.ys[lag:]
    y2 = wave.ys[:n-lag]
    corr = np.corrcoef(y1, y2, ddof=0)[0, 1]
    return corr
```

`serial_corr` takes a Wave object and `lag`, which is the integer number of places to shift the wave. It computes the correlation of the wave with a shifted version of itself.

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:

```
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.

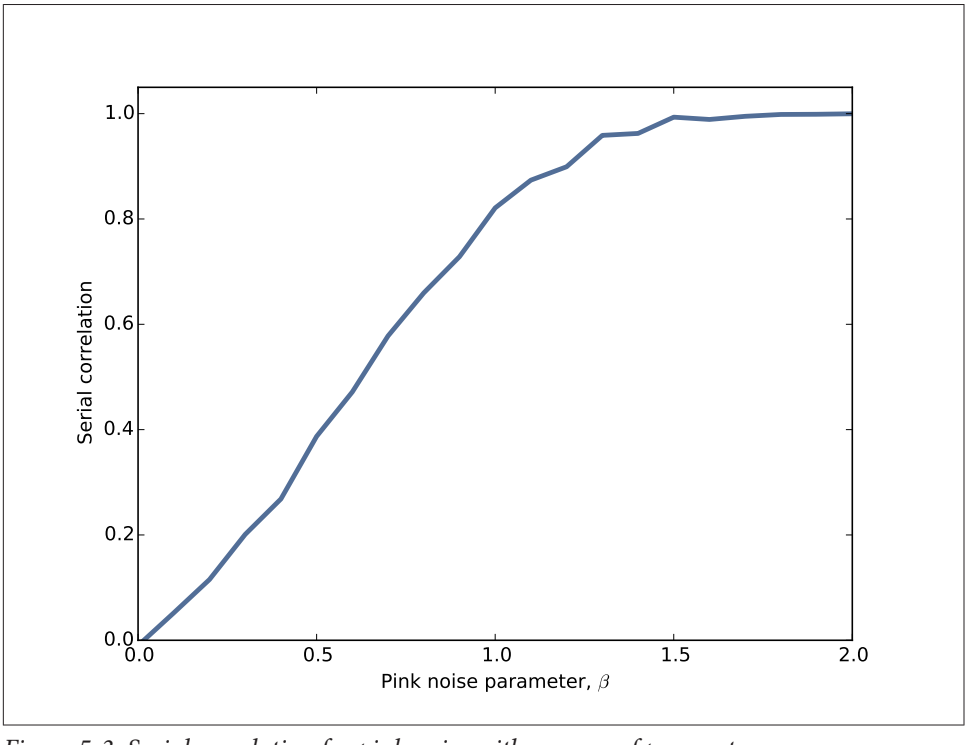

Figure 5-3. Serial correlation for pink noise with a range of parameters.

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.

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 we can easily compute serial correlations with different lags.

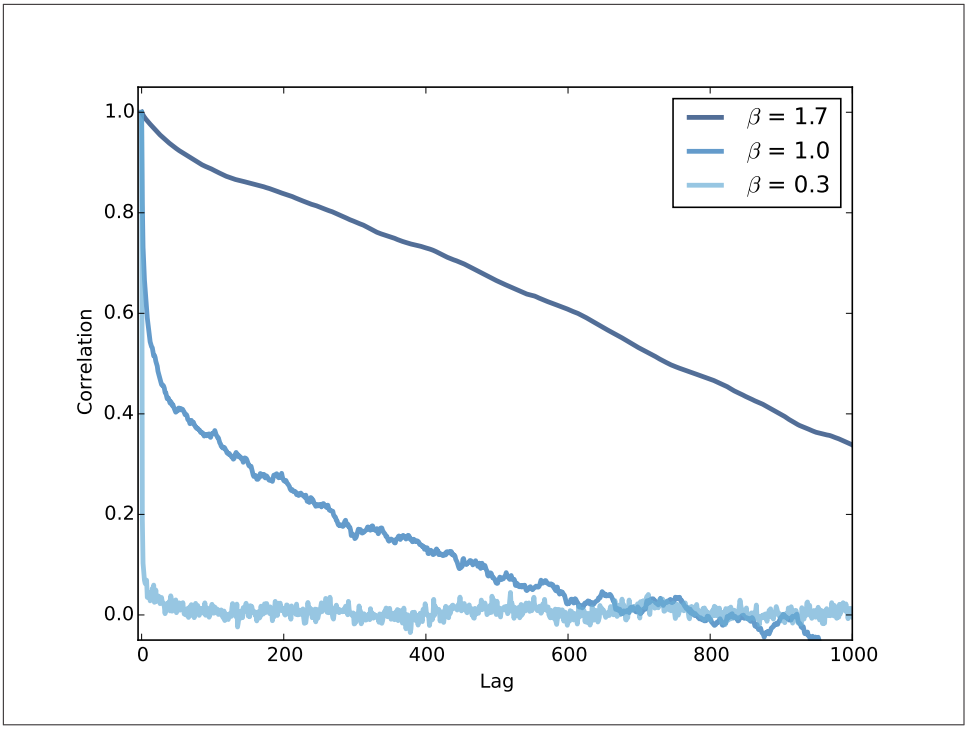

Figure 5-4. Autocorrelation functions for pink noise with a range of parameters.

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`:

```
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 off to zero quickly. For larger values, serial correlation is stronger and drops off more slowly. With $\beta = 1.7$ 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.

Autocorrelation of periodic signals

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

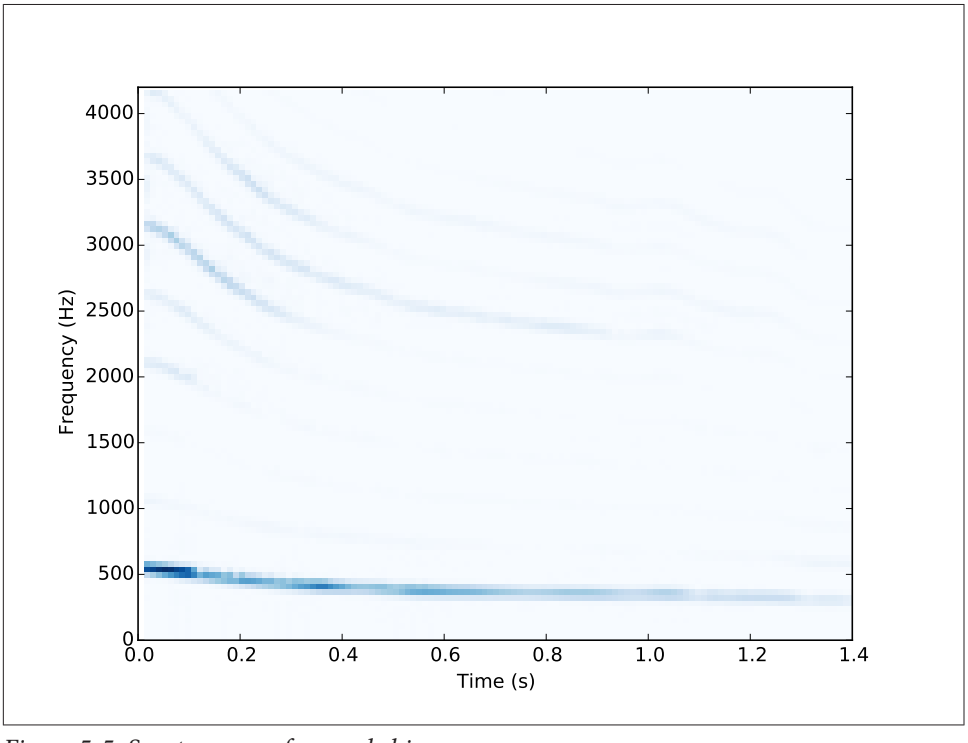

Figure 5-5. Spectrogram of a vocal chirp.

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*. 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.

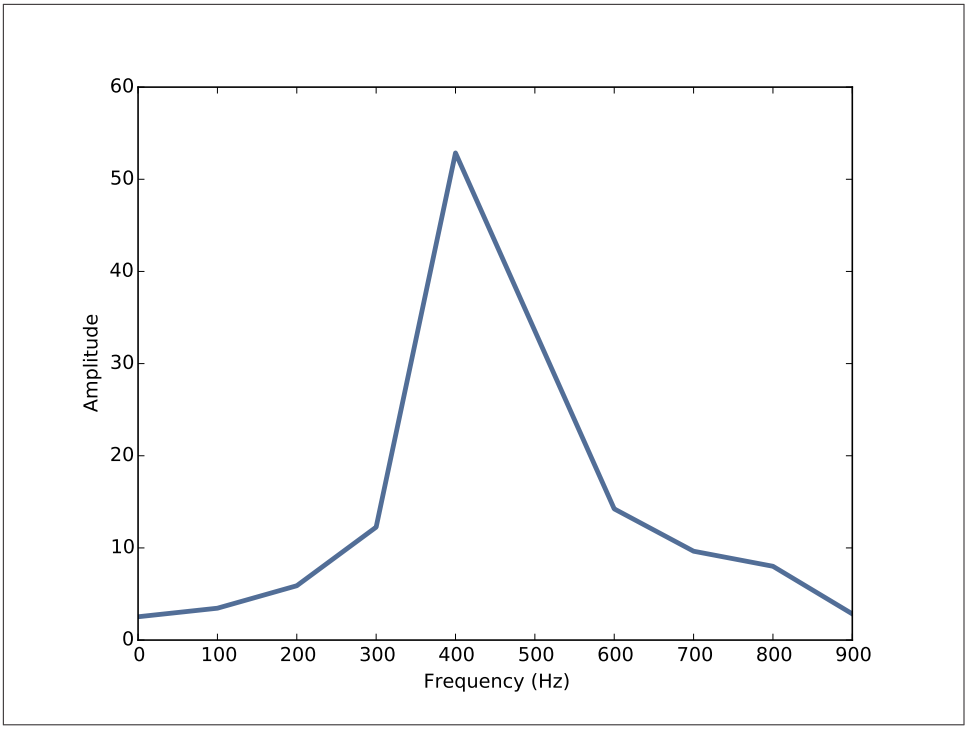

Figure 5-6. Spectrum of a segment from a vocal chirp.

To estimate pitch at a particular point in time, we could use the spectrum, but it doesn't work very well. To see why not, I'll 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=1000)
```

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 "The Gabor limit" on page 45). That means the estimated pitch might be off by 50 Hz; in musical terms, the range from 350 Hz to 450 Hz is about 5 semitones, which is a big difference!

We could get better frequency resolution by taking a longer segment, but since the pitch is changing over time, we would also get "motion blur"; that is, the peak would spread between the start and end pitch of the segment, as we saw in "Spectrum of a chirp" on page 43.

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

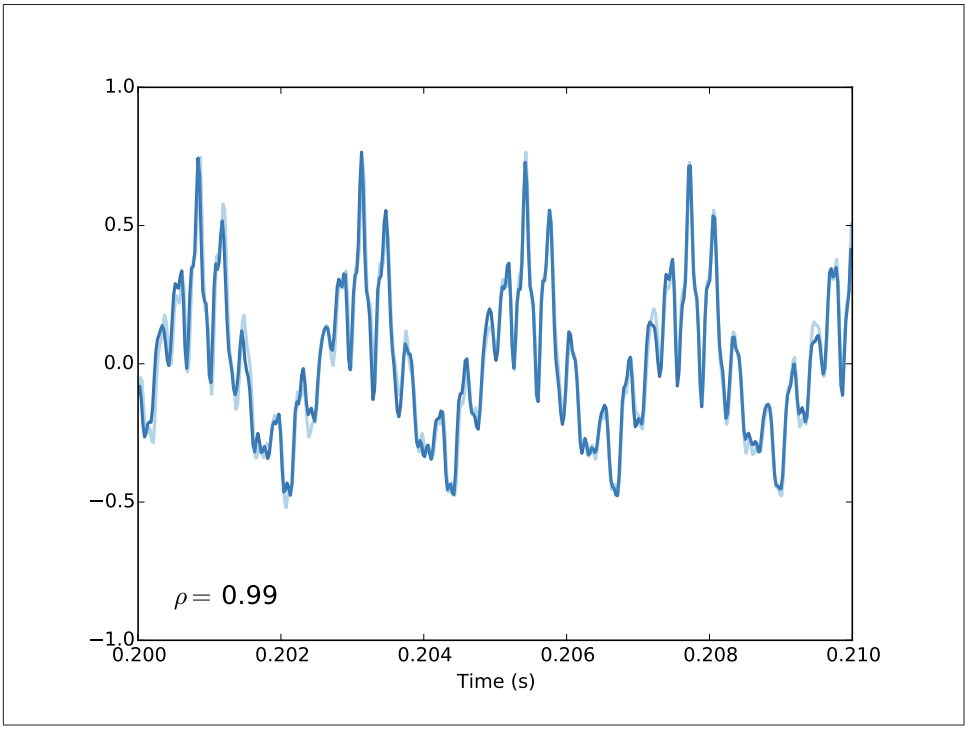

Figure 5-7. Two segments from a chirp, one starting 0.0023 seconds after the other.

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

```
def plot_shifted(wave, offset=0.001, 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-offset, duration=0.01)
    segment2.shift(offset)
    segment2.plot(linewidth=2, alpha=0.4)

    corr = segment1.corr(segment2)
    text = r'$\rho =$ %.2g' % corr
    thinkplot.text(segment1.start+0.0005, -0.8, text)
    thinkplot.config(xlabel='Time (s)')
```

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 0.99. This result suggests that the period is near 0.0023 seconds, which corresponds to a frequency of 435 Hz.

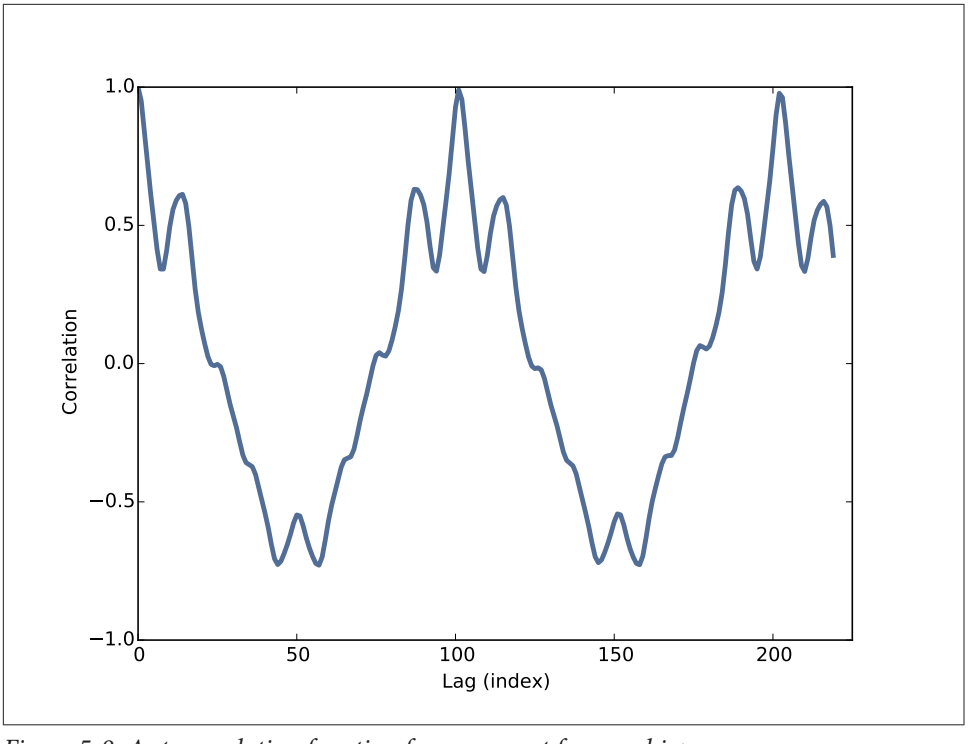

Figure 5-8. Autocorrelation function for a segment from a chirp.

For this example, I estimated the period by trial and error. To automate the 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`. We can compute the frequency that corresponds to that period like this:

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

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. The frequency precision using autocorrelation is less than 10 Hz, compared with 100 Hz using the spectrum. In musical terms, the expected error is about 30 cents (a third of a semitone).

Correlation as dot product

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

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

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.

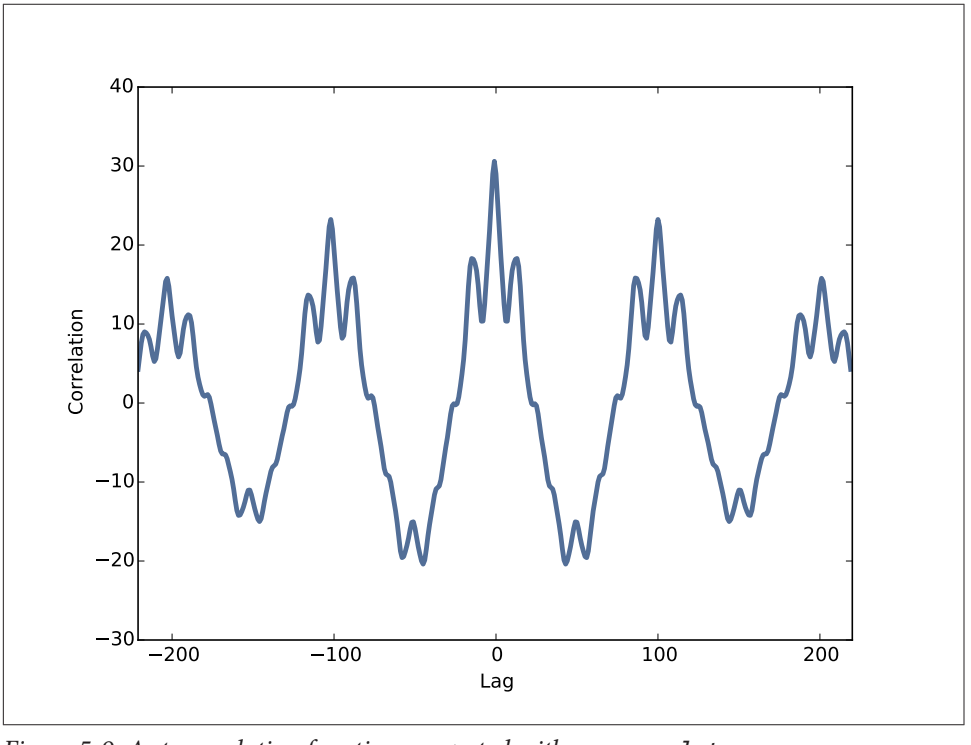

Figure 5-9. Autocorrelation function computed with `np.correlate`.

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

The option `mode` tells `correlate` what range of `lag` to 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 `np.correlate` get smaller as the lags increase. That's because `np.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 `np.correlate` are nearly the same. They still differ by 1-2%. The reason is not important, but if you are curious: `autocorr` standardizes the correlations independently for each lag; for `np.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.

Exercises

Solutions to these exercises are in `chap05soln.ipynb`.

Example 5-1.

The IPython notebook for this chapter, `chap05.ipynb`, includes an interaction that lets you compute autocorrelations for different lags. Use this interaction to estimate the pitch of the vocal chirp for a few different start times.

Example 5-2.

The example code in `chap05.ipynb` shows how to use autocorrelation to estimate the fundamental frequency of a periodic signal. Encapsulate this code in a function called `estimate_fundamental`, and use it to track the pitch of a recorded sound.

To see how well it works, try superimposing your pitch estimates on a spectrogram of the recording.

Example 5-3.

If you did the exercises in the previous Autocorrelation, 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 drop off quickly? Is there evidence of periodic behavior?

Example 5-4.

In the repository for this book you will find an IPython notebook called `saxophone.ipynb` that explores autocorrelation, pitch perception, and a phenomenon called the **missing fundamental**. Read through this notebook and run the examples. Try selecting a different segment of the recording and running the examples again.

Vi Hart has an excellent video called "What is up with Noises? (The Science and Mathematics of Sound, Frequency, and Pitch)"; it demonstrates the missing fundamental phenomenon and explains how pitch perception works (at least, to the degree that we know). Watch it at *https://www.youtube.com/watch?v=i_0DXxNeaQ0*.

CHAPTER 6

Discrete cosine transform

Chapter to Come

CHAPTER 7

Discrete Fourier Transform

Chapter to Come

CHAPTER 8

Filtering and Convolution

Chapter to Come

CHAPTER 9

Signals and systems

Chapter to Come

CHAPTER 10

Modulation and sampling

Chapter to Come

Index

A
Anaconda, xii

C
clone, xii
contributors, xiii

F
fork, xii

G
Git, xii
GitHub, xii

I
installation, xiii
IPython, xiii

M
matplotlib, xii

N
NumPy, xii

R
repository, xii

S
SciPy, xii