Think DSP

Digital Signal Processing in Python

Version 0.6

Think DSP

Digital Signal Processing in Python

Version 0.6

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Preface

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

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

This is a work in progress, so comments are welcome.

Allen B. Downey

Needham MA

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

Contributor List

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

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

- Before I started writing, my thoughts about this book benefited from conversations with Boulos Harb at Google and Aurelio Ramos, formerly at Harmonix Music Systems.

- During the Fall 2013 semester, Nathan Lintz and Ian Daniher worked with me on an independent study project and helped me with the first draft of this book.

- On Reddit's DSP forum, the anonymous user RamjetSoundwave helped me fix a problem with my implementation of Brownian Noise.

Contents

Chapter 1

Sounds and signals

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

A microphone is a device that measures these variations and generates an electronic 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. It focuses on sound signals, but the same methods apply to other signals, like electronic signals and mechanical vibration.

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.

1.1 Periodic signals

We'll also start with **periodic signals**, which are signals that repeat themselves after some period of time. For example, if you strike a tuning fork, it

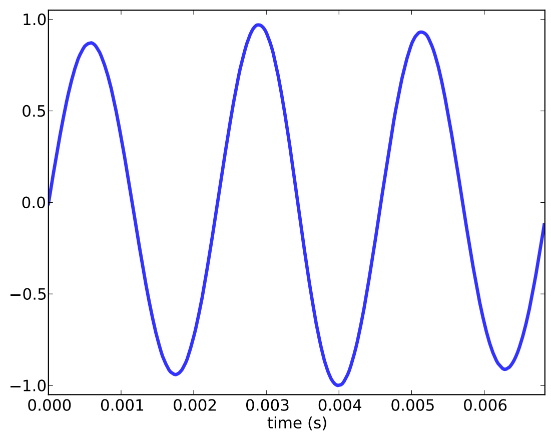

Figure 1.1: Segment from a recording of a tuning fork.

vibrates and generates sound. If you record that sound and plot the transduced signal, it looks like Figure 1.1.[1]

This signal is 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 suffix indicates which octave the note is in. A4 is the A above middle C. A5 is one octave higher. See `http://en.wikipedia.org/wiki/Scientific_pitch_notation`.

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

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

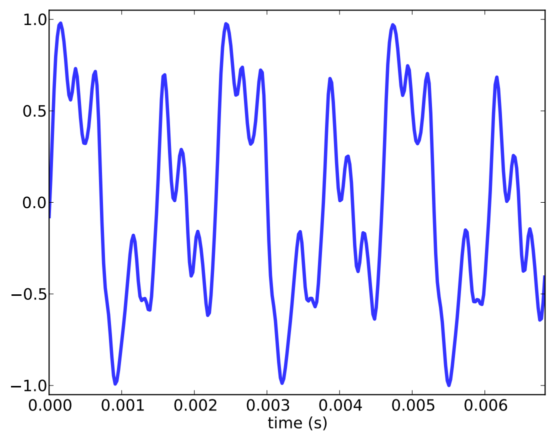

Figure 1.2: Segment from a recording of a violin.

String Quintet No. 5 in E, 3rd movement.[2]

Again we can see that the signal is periodic, but the shape of the signal is much 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 tone quality of the sound. People usually perceive complex waveforms as rich, warm and more interesting than sinusoids.

1.2 Spectral decomposition

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

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

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

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

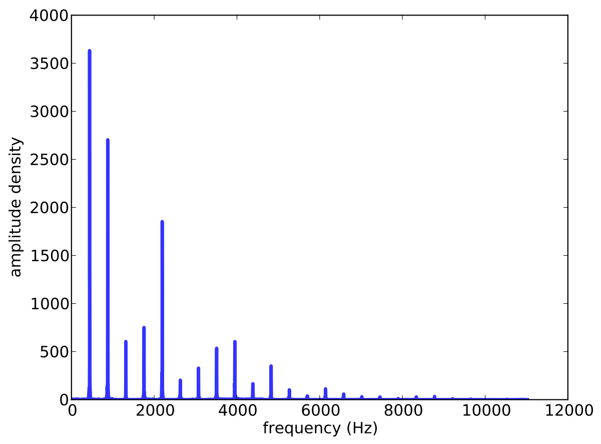

Figure 1.3: Spectrum of a segment from the violin recording.

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.
- 1760 is A6, two octaves above the fundamental.
- 2200 is approximately C♯7, which is a major third above A6.

In other words, 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 (at least approximately) by adding up sinusoids. Or, to reconstruct the signal exactly, you can use the inverse DFT. Next we'll see how.

1.3 Signals

I wrote a Python module called `thinkdsp` that contains classes and function for working with signals and spectrums. [3] You can download it from `http://think-dsp.com/thinkdsp.py`.

To represent signals, `thinkdsp` provides a class named `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 arbitrary units. `offset` is a `phase offset` in radians.

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

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

```
mix = sin_sig + cos_sig
```

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

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

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

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)
```

[3] In Latin the plural of "spectrum" is "spectra", but I prefer to use standard English plurals.

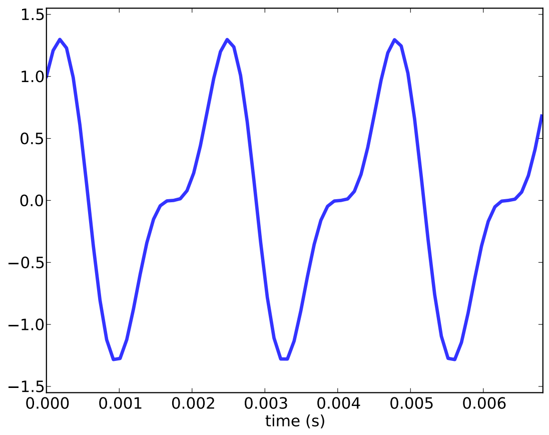

Figure 1.4: Segment from a mixture of two sinusoid signals.

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

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

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

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

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

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

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

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

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

`start` and `duration` are in seconds. This example copies the first three periods from `mix`. If we plot `segment`, it looks like Figure 1.4. This signal contains two frequency components, so it is more complex than the signal from the tuning fork, but less complex than the violin.

1.4 Reading and writing Waves

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

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

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

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

You can listen to the Wave with any media player that plays WAV files. On UNIX systems, I use `aplay`, which is simple, robust, and included in many Linux distributions. For Windows you might like MicroWav, available from `http://bellsouthpwp2.net/b/o/bobad/microwav.htm`.

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

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

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

1.5 Spectrums

`Wave` provides `make_spectrum`, which returns a `Spectrum`:

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

And `Spectrum` provides `plot`:

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

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

`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 a the band of frequencies between two cutoffs.

This example attenuates all frequencies above 600 by 99%:

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

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

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

At this point you know how to use many of the classes and functions in `thinkdsp`, and you are ready to do the exercises at the end of the chapter. In Chapter 2 I explain more about how these classes are implemented.

1.6 Exercises

Exercise 1.1 Read the thinkdsp documentation.

Exercise 1.2 Fork and clone the repo.

Exercise 1.3 The example code in this chapter is in `example1.py`. If you run it, it should ...

Gradually increase the cutoff and listen...

Exercise 1.4 Run violin.py and listen.

Exercise 1.5 Create exercise1.py and copy example1.py

Record a WAV file, or download from ...

Use an audio editor to examine the wave

Find a section where the wave is periodic and plot a few periods.

Compute and plot the spectrum.

Exercise 1.6 Extract a 0.5-2 second segment with constant frequency.

Compute the spectrum.

Remove some of the harmonics, invert the transform and play the wave.

Exercise 1.7 Synthesize a wave by creating a spectrum with arbitrary harmonics, inverting it, and listening. What happens as you add frequency components that are not multiples of the fundamental?

Exercise 1.8 This exercise asks you to write a function that simulates the effect of sound transmission underwater. This is a more open-ended exercise for ambitious readers. It uses decibels, which you can read about at `http://en.wikipedia.org/wiki/Decibel`.

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

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

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

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

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

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

Chapter 2

Harmonics

2.1 Implementing Signals and Spectrums

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

We'll start with `CosSignal` and SinSignal:

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

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

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

```
class Sinusoid(Signal):

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

The parameters of `__init__` are:

- `freq`: frequency in cycles per second, or Hz.
- `amp`: amplitude. The units of amplitude are arbitrary, usually chosen so 1.0 corresponds to the maximum input from a microphone or maximum output to a speaker.

- offset: where in its period the signal starts, at $t = 0$. In units of radians, for reasons I explain below.
- func: a Python function used to evaluate the signal at a particular point in time. It is usually either numpy.sin or numpy.cos, yielding a sine or cosine signal.

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

The parent class of Sinusoid, Signal, provides make_wave:

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

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

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

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

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

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

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

Let's unwind this function one step at time:

1. self.freq is frequency in cycles per second, and each element of ts a time in seconds, so their product is the number of cycles since the start time.
2. PI2 is a constant that stores 2π. Multiplying by PI2 converts from cycles to **phase**. You can think of phase as "cycles since the start time" except that the number of cycles is in units of radians, so each cycle is 2π radians.

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

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

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

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

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

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

2.2 Computing the spectrum

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

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

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

The result of `fft` is a sequence of complex numbers, `hs`, which is stored in a Spectrum. There are two ways to think about complex numbers:

- A complex number is the sum of a real part and an imaginary part, often written $x + iy$, where i is the imaginary unit, $\sqrt{-1}$. You can think of x and y as Cartesian coordinates.

- A complex number is also the product of a magnitude and a complex exponential, $re^{i\phi}$, where r is the **magnitude** and ϕ is the **angle** in radians, also called the "argument." You can think of r and ϕ as polar coordinates.

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

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

Here is the definition of `Spectrum`:

```
class Spectrum(object):

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

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

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

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

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

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

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

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

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

2.3 Other waveforms

A sinusoid contains only one frequency component, so its DFT has only one peak. More complex waveforms, like the violin recording, yield DFTs

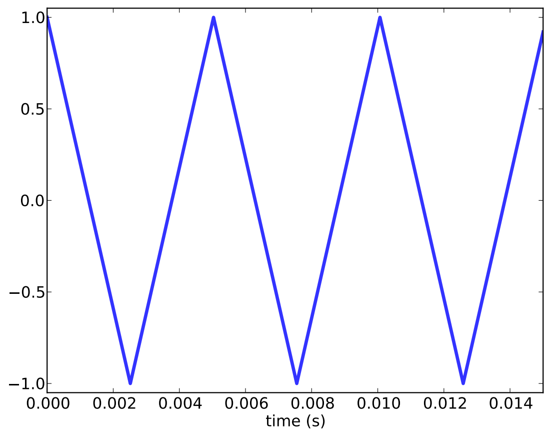

Figure 2.1: Segment of a triangle signal at 200 Hz.

with many peaks. In this section we investigate the relationship between waveforms and their DFTs.

The triangle waveform is like a straight-line version of a sinusoid. Figure 2.1 shows a triangle waveform with frequency 200 Hz.

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

```
class TriangleSignal(Sinusoid):

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

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

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

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

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

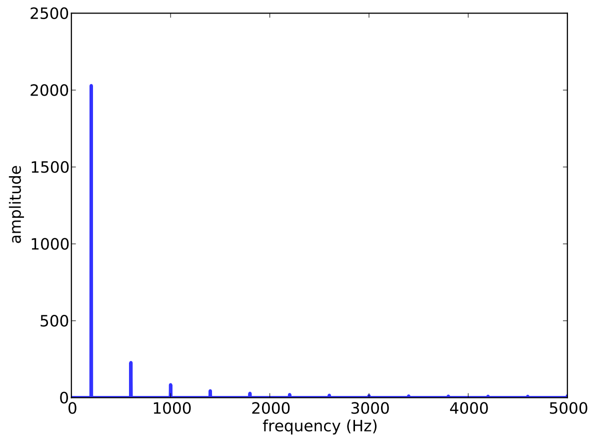

Figure 2.2: Spectrum of a triangle signal at 200 Hz.

integer part, which is ignored. [1]

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

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

Here's the code that generates Figure 2.1:

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

2.4 Harmonics

Next we can compute the spectrum of this waveform:

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

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

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

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

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

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

```
class SquareSignal(Sinusoid):

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

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

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

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

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

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

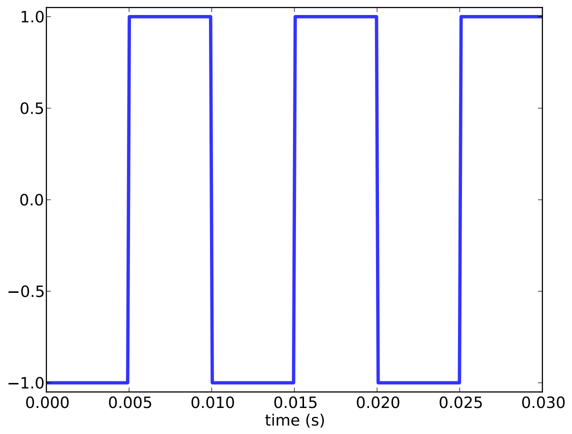

Figure 2.3: Segment of a square signal at 100 Hz.

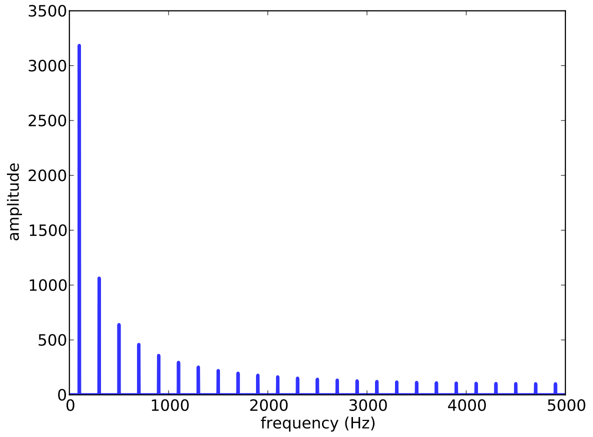

Figure 2.4: Spectrum of a square signal at 100 Hz.

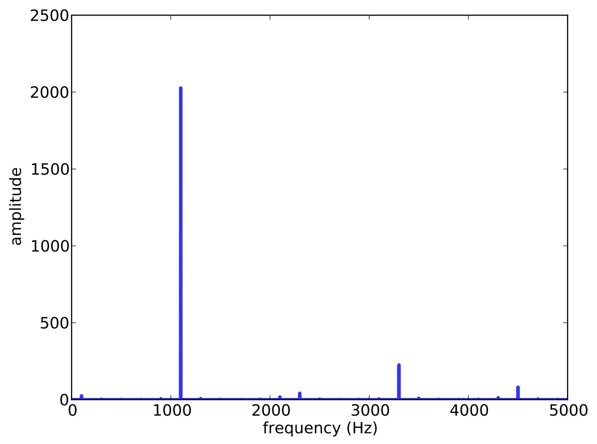

Figure 2.5: Spectrum of a triangle signal at 1100 Hz sampled at 10,000 frames per second.

2.5 Aliasing

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

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

As expected, there are peaks at 1100 and 3300 Hz, but the third peak is at 4500, not 5500 Hz as expected. There is a small fourth peak at 2300, not 7700 Hz. And if you look very closely, the peak that should be at 9900 is actually at 100 Hz. What's going on?

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

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

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

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

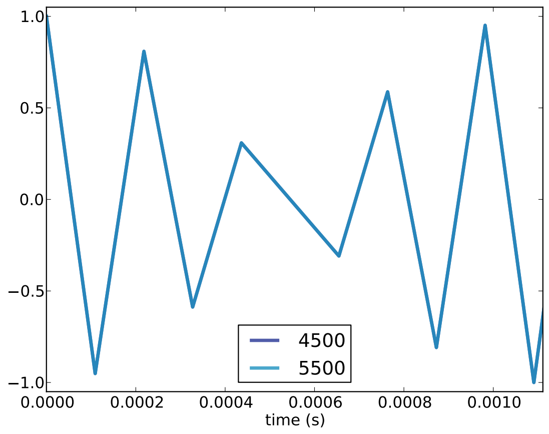

Figure 2.6: Cosine signals at 4500 and 5500 Hz, sampled at 10,000 frames per second.

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

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

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

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

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

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

2.6 Exercises

Exercise 2.1 Listen to the various non-sinusoidal waveforms.

Exercise 2.2 Start with a sawtooth. Compute the DFT. Remove all harmonics above the *i*th. Compute the inverse DFT and see what it looks like. Repeat for a range of values of *i*.

Exercise 2.3 Write a new Signal type, compute it's DFT and listen to it. What happens if the signal is discontinuous? What if the signal is continuous but the slope is discontinuous?

Exercise 2.4 Make a frequency-limited sawtooth wave and see what it looks like and sounds like. See `http://en.wikipedia.org/wiki/Sawtooth_wave`.

Exercise 2.5 What does a square wave sound like underwater?

Exercise 2.6 Sample an 1100 Hz triangle at 10000 frames per second and listen to it. Can you hear the aliased harmonic? Might help to play a sequence of notes with increasing pitch.

Exercise 2.7 Compute the spectrum of an 1100 Hz square wave.

Chapter 3

Non-periodic signals

3.1 Chirp

The signals we have worked with so far are periodic, which means that they repeat forever. It also means that their spectrums do no vary in time. In this chapter, we consider non-periodic signals, which includes any signal whose frequency components change over time. In other words, pretty much all sound signals.

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

```
class Chirp(Signal):

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

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

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

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

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

`_evaluate` is a helper function that does the rest of the math:[1]

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

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

The next step is to figure out how much the phase changes during each segment. Since we know the frequency and duration of each segment, the *change* in phase during each integral is `PI2 * freqs * dts`.

Given the changes in phase, `numpy.cumsum` computes the total phase at the end of each segment. Finally `numpy.cos` maps from phase to amplitude.

It might not be obvious how this function works, so here's another way to think about it. When f is constant, phase increases linearly over time. Mathematically:

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

If f varies in time, we can find the relationship between phase and the instantaneous value of f by taking the time derivative of both sides:

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

In other words, frequency is the derivative of phase. Or equivalently, phase is the integral of frequency. We can write the same equation in differential form:

$$d\phi = 2\pi f dt$$

which looks a whole lot like this line from `_evaluate`:

```
        dps = PI2 * freqs * dts
```

[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.

To find the total phase, you can imagine adding up the elements of dps, or if you like calculus, you can think of it as a numerical solution of an integral.

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

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

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

Before you go on, run this code and listen.

3.2 Exponential chirp

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

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

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

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

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

Here's the code that makes one:

```
class ExpoChirp(Chirp):

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

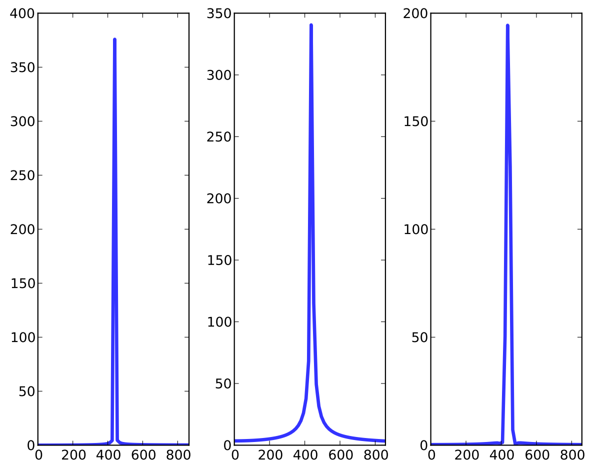

Figure 3.1: Spectrum of (a) a periodic segment of a sinusoid, (b) a non-periodic segment, (c) a tapered non-periodic segment.

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

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

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

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

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

3.3 Leakage

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

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

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

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

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

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

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

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

3.4 Windowing

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

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

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

Figure 3.2c shows the result of multiplying the window by the original signal. Where the window is close to 1, the signal is unchanged. Where the

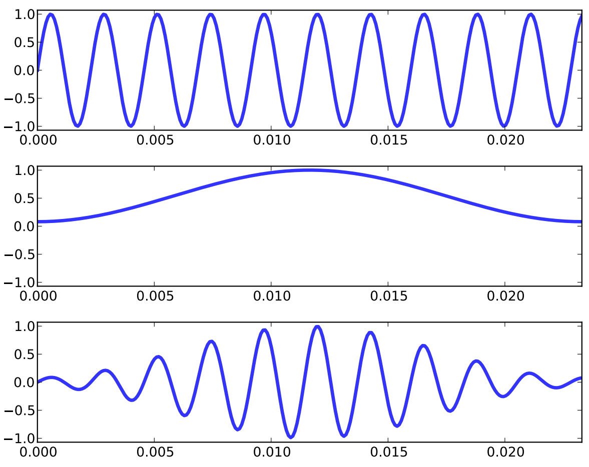

Figure 3.2: (a) Segment of a sinusoid, (b) Hamming window, (c) product of the segment and the window.

window is close to 0, the signal is attenuated. Because the window tapers at both ends, the end of the segment connects smoothly to the beginning.

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

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

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

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

3.5 Spectrum of a chirp

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

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

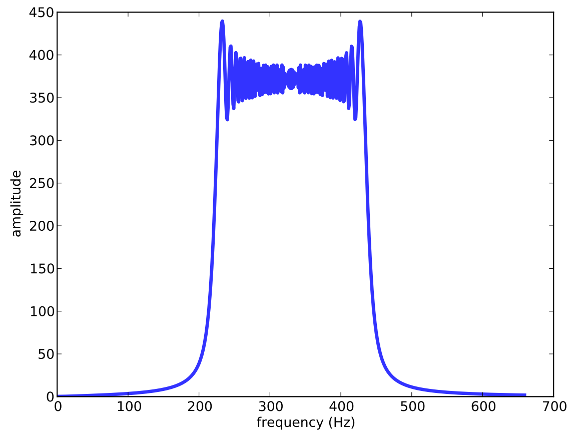

Figure 3.3: Spectrum of a one-second one-octave chirp.

Figure 3.4 shows the result. The spectrum shows components at every frequency from 220 to 440 Hz with variations, caused by leakage, that look a little like the Eye of Sauron.

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 loses the relationship between frequency and time. For example, we cannot tell by looking at this spectrum whether the frequency went up or down, or both.

3.6 Spectrogram

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

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

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

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

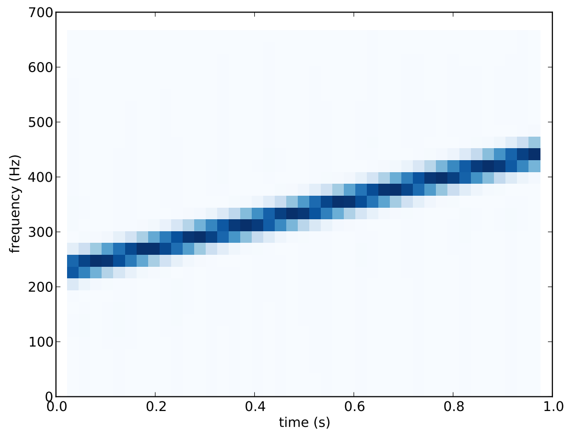

Figure 3.4: Spectrogram of a one-second one-octave chirp.

```
wave = signal.make_wave(duration=1)
spectrogram = wave.make_spectrogram(seg_length=512)
spectrogram.plot(high=32)
```

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

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

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

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

3.7 The Gabor limit

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

The **frequency resolution** is the frequency range between components in the spectrum, which corresponds to the height of the cells. With 512 frames,

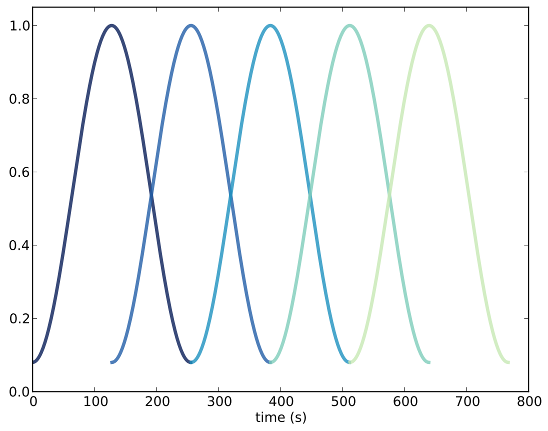

Figure 3.5: Overlapping Hamming windows.

we get 256 frequency components over a range from 0 to 5012.5 Hz, so the range between components is 21.5 Hz.

More generally, if n is the segment length, the spectrum contains $n/2$ components. If the framerate is r, the maximum frequency in the spectrum is $r/2$. So the time resolution is n/r and the frequency resolution is $r/2$ divided by $n/2$, 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 to play with, but the range of frequencies increases at the same time.

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

3.8 Implementing spectrograms

Here is the `Wave` method that computes spectrograms:

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

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

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

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

            start += step
            end += step

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

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

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

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

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

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

And Spectrogram provides plot, which generates a pseudocolor plot:

```
    def plot(self, low=0, high=None):
        ts = self.times()
        fs = self.frequencies()[low:high]
        # copy amplitude from each spectrum into a column of the array
        size = len(fs), len(ts)
        array = numpy.zeros(size, dtype=numpy.float)

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

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

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

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

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

So that's how Spectrograms are implemented.

3.9 Exercises

Exercise 3.1 Write a class called `SawtoothChirp` that extends `Chirp` and overrides `evaluate` to generate a sawtooth waveform with frequency that increases (or decreases) linearly.

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

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

Exercise 3.2 Another way to generate a sawtooth chirp is to add up a harmonic series of sinusoidal chirps. Write another version of `SawtoothChirp` that uses this method and plot the histogram.

If you truncate the harmonic series at the Nyquist frequency, you should be able to avoid aliasing.

Exercise 3.3 In musical terminology, a "glissando" is a note that slides from one pitch to another, so it is similar to a chirp. A trombone player can play a glissando by extending the trombone slide while blowing continuously. As the slide extends, the total length of the tube gets longer, and the resulting pitch is inversely proportional to length.

Write a function that simulates a trombone glissando from C4 up to F4 and back down to C4. C3 is 262 Hz; F3 is 349 Hz.

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

Exercise 3.4 Try other window functions.

Exercise 3.5 Make a spectrogram of the clarinet glissando at the beginning of Rhapsody in Blue.

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

Chapter 4

Noise

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

1. As in English, it can mean an unwanted signal of any kind. If two signals interfere with each other, then for each signal the other would be considered 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 noise in the second sense. The simplest way to understand noise is to generate it, and the simplest way to generate it is uncorrelated uniform noise (UU noise). “Uniform” means that the values that make up the signal are drawn at random from a uniform distribution, and “uncorrelated” means that each value is independent of the others.

Here’s the code to generate some:

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

The result is a wave with duration 0.5 seconds at 11,025 samples per second. Each sample is drawn from a uniform distribution between -1 and 1, which means that every value in that range is equally likely.

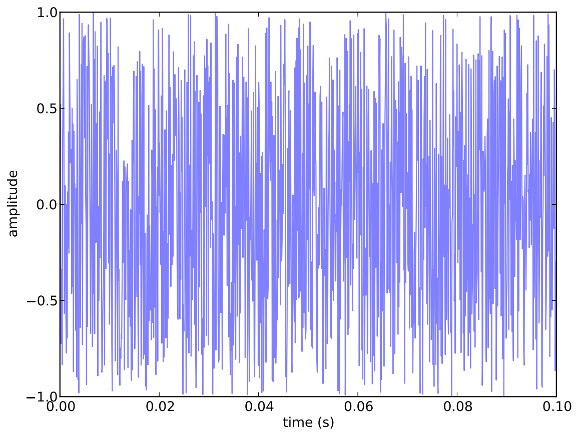

Figure 4.1: Waveform of uncorrelated uniform noise.

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

Now let's take a look at the spectrum. Here's the code:

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

`Spectrum.plot_power` is similar to `Spectrum.plot`, except that it plots power density against frequency. Power density is the square of amplitude density, expressed in units of power per Hz.

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

- Distribution: The distribution of a random signal is the set of possibly 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 distribution or "bell curve."

- Correlation: Is each value in the signal independent of the others, or are there dependencies between them? In UU noise, each value is independent of the others, so knowing the value of the signal at one

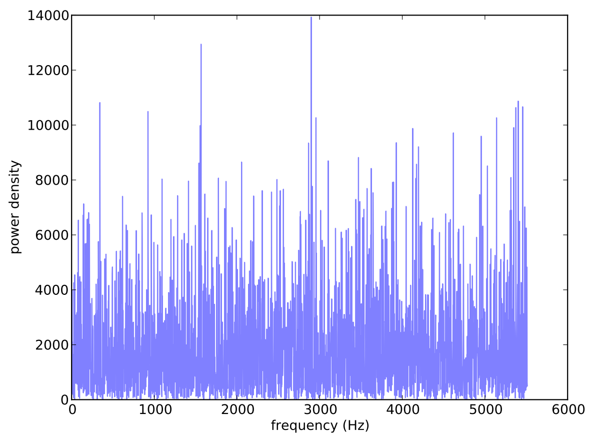

Figure 4.2: Spectrum of uncorrelated uniform noise.

point in time provides no new information about the value at any other time. An alternative is **Brownian noise**, where each value is the sum of the previous value and a random value. So if the value of the signal is high at a particular point in time, we expect it to stay high for a least the next few values. 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 is no relationship between power and frequency (or, if you like, the relationship is a constant). An alternative is **pink noise**, where power is inversely related to frequency; that is, the power at frequency f is drawn from a distribution whose mean is proportional to $1/f$.

4.1 Integrated spectrum

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

`Spectrum` provides a method that computes the IntegratedSpectrum:

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

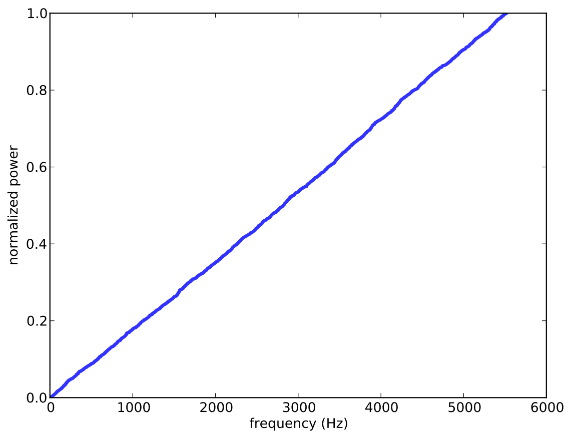

Figure 4.3: Integrated spectrum of uncorrelated uniform noise.

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

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

```
class IntegratedSpectrum(object):

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

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

```
    integ = spectrum.make_integrated_spectrum()
    integ.plot_power()
    thinkplot.show(xlabel='frequency (Hz)',
                   ylabel='cumulative power')
```

The result, shown in Figure 4.3, is a remarkably 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.

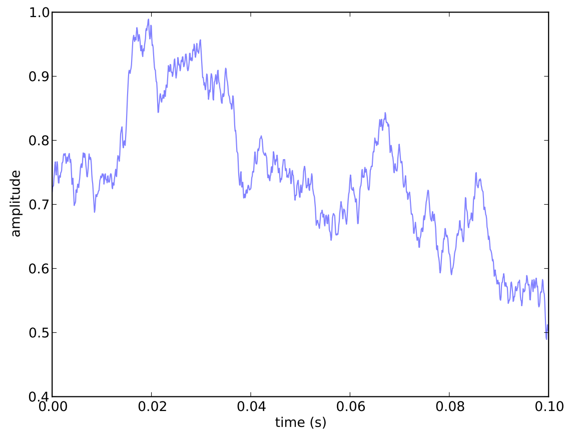

Figure 4.4: Waveform of Brownian noise.

4.2 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.

The simplest way to generate Brownian noise is to generate an uncorrelated signal and then compute it cumulative sum.

Here is a class definition that implements this algorithm:

```
class BrownianNoise(_Noise):

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

We use `numpy.random.uniform` to generate an uncorrelated signal and `scipy.integrate.cumtrapz` to approximate the integral using the trapezoid rule.

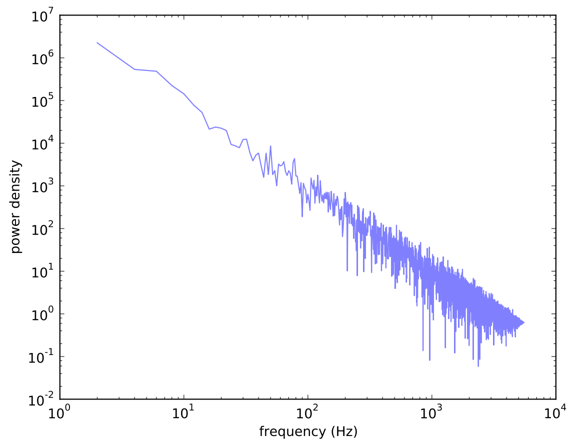

Figure 4.5: Spectrum of Brownian noise on a log-log scale.

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

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

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

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

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

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

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

In theory, we expect the result to be a straight line with slope -2. To see why, it helps to know that if $H(f)$ is the Fourier transform of a wave, $h(t)$, and

$g(t)$ is the integral of $h(t)$, then the Fourier transform of $g(t)$ is $H(f)/i2\pi f$. In the denominator, the factor of f is the important part; it indicates that integrating a waveform has the effect of a low-pass filter. Each component is attenuated in proportion to its frequency.

Since power is related to the square of amplitude, dividing the Fourier components by f has the effect of dividing power by f^2. Since we started with white noise, which has equal power at all frequencies, we expect Brownian noise to have power at frequency f proportional to $1/f^2$, on average. In other words:

$$P = kf^{-2}$$

with some unknown constant, k. Taking the log of both sides yields:

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

Which means that the result, in Figure 4.5, should be a straight line with slope -2. It is quite noisy, so we can't be sure it is a straight line, but at least we can estimate its slope.

Spectrum provides `estimate_slope`, which uses `scipy` to do linear regression (also known as a least squares fit):

```
    def estimate_slope(self):
        x = numpy.log(self.fs[1:])
        y = numpy.log(self.power[1:])
        t = scipy.stats.linregress(x,y)
        return t
```

I discard the first component of the spectrum for two reasons: (1) this component corresponds to $f = 0$, and $\log 0$ is undefined, and (2) the amplitude at $f = 0$ indicates the mean value of the signal (sometimes called the "DC offset" or "bias"). Since we unbiased the signal, this component should be 0 anyway.

`estimate_slope` returns the result from `scipy.stats.linregress` which is a tuple that contains the estimated slope and intercept, coefficient of determination (R^2), p-value, and standard error. For now, we only need the slope, which for this example is -1.8.

Brownian noise is also called "red noise," for the same reason that white noise is called "white." If you combine visible light with the same relationship between frequency and power, most of the power would be at the low-frequency end of the spectrum, which is red.

Brownian noise is also sometimes called "brown noise", but I think that's confusing, so I avoid it.

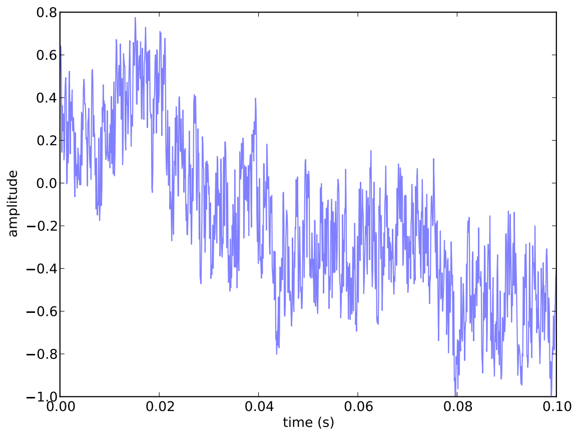

Figure 4.6: Waveform of pink noise with $\beta = 1$.

4.3 Pink Noise

To review, for red noise, the relationship between frequency and power is

$$P = kf^{-2}$$

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

$$P = kf^{-\beta}$$

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.

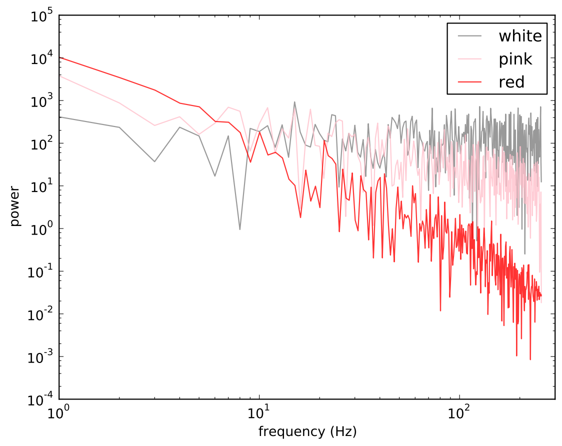

Figure 4.7: Spectrum of white, pink, and red noise on a log-log scale.

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

    spectrum.pink_filter(beta=self.beta)

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

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

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

`Spectrum` provides `pink_filter`:

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

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

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

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

4.4 Exercises

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

Exercise 4.2 Spectrum of natural noise sources: symbal crash, applause, etc.

Exercise 4.3 Estimate the spectrum of a long time series (stock market data? astronomical data?) Make a periodogram.

Chapter 5

Pitch tracking

Autocorrelation

5.1 Spectrum of vowels

5.2 Spectrum of a piano

5.3 Music synthesis

Additive synthesis.

Envelope generation.

5.4 Convert music notation to a signal

5.5 Compute the spectrogram of an idealized performance

5.6 Compute the spectrogram of an actual performance

5.7 Pitch tracking algorithm

Autocorrelation (or maybe this gets introduced in the noise chapter first?)

5.8 Pitch shifting

Auto-tune?

Chapter 6

Discrete cosine transform

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

DCT is similar in many ways to the discrete Fourier transform (DFT). Once we learn how DCT works, it will be easier to explain DFT.

Here are the steps we will follow to get there:

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

2. Next we'll rewrite the synthesis problem using numpy arrays. This move is good for performance, and also provides insight into the analysis problem.

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

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

Let's get started.

6.1 Synthesis

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

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

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

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

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

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

We can test this function like this:

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

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

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

Conceptually, synthesis is pretty simple. But in this form it doesn't help much with **analysis**; that is, given the wave, identifying the frequency components and their amplitudes. To do that, we have to take another approach.

6.2 Synthesis with arrays

Here's another way to write `synthesize`:

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

Figure 6.1: Synthesis with arrays.

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

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

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

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

3. `numpy.dot` computes a dot product. In terms of linear algebra, we are multiplying a matrix, `M`, by a vector, `amps`. Figure 6.1 is a diagram of this multiplication. `numpy.dot` multiplies each row of `M` by `amps`, element-wise, and then adds up the products.

In the diagram, each row of the matrix, `M`, corresponds to a time from 0.0 to 1.0 s; t_k is the time of the kth row. Each column corresponds to a frequency from 100 to 400 Hz; f_j is the frequency of the jth column.

I have labeled the kth row with the letters a through d; as an example, the value of a is $\cos[2\pi(220)t_k]$.

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

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

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

```
    amps = numpy.array([0.6, 0.25, 0.1, 0.05])
    freqs = [100, 200, 300, 400]
    framerate = 11025
    ts = numpy.linspace(0, 1, framerate)
    ys1 = synthesize1(amps, freqs, ts)
    ys2 = synthesize2(amps, freqs, ts)
    print max(abs(ys1 == ys2))
```

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

One note on linear algebra vocabulary: I am using "matrix" to refer to a two-dimensional array, and "vector" for a one-dimensional array or sequence. To be a bit pedantic, it would be more correct to think of matrices and vectors as abstract mathematical concepts, and numpy arrays as one way (and not the only way) to represent them.

One advantage of linear algebra is that it provides concise notation for operations on matrices and vectors. For example, we could write synthesize like this:

$$M = \cos[2\pi(t \otimes f)]$$
$$y = Ma$$

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

6.3 Analysis

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

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

Here's what the code looks like:

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

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

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

In this example, we have only 4 frequencies, but we evaluated the signal at 11,025 times. So we have many more equations than unknowns. That suggests that we should be able to recover a using only 4 elements of `ys`. Here's what that looks like:

Using the values of `ys`, `freqs` and `ts` from the previous section, we can run `analyze1` like this:

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

And sure enough, we get

```
amps2 = [ 0.6   0.25  0.1   0.05]
```

In this case, we know that the `ys` were actually generated by adding only 4 frequency components, so we can use any 4 values from the wave array to recover `amps`. But in general if `ys` contains more than 4 elements, it is unlikely that we can analyze it using only 4 frequencies.

We'll come back to this issue later, but first I want to address a different problem – this algorithm is slow. Solving a linear system of equations takes time proportional to n^3, where n is the number of columns in M.

It turns out that we can do better.

6.4 Orthogonal matrixes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

And that's not the identity matrix.

6.5 DCT-IV

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

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

Here's an updated version of `test1`:

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

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

With these values, M is

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

And $M^T M$ is

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

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

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

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

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

Putting it all together, we can write an implementation of DCT-IV:

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

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

We can test `dct_iv` like this

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

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

Starting with `amps`, we synthesize a wave array, then use `dct_iv` to see if we can recover the amplitudes. The biggest difference between `amps` and `amps2` is about `1e-16`, which is what we expect with 64-bit floating point arithmetic.

6.6 Inverse DCT

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

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

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

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

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

6.7 Summary

This chapter explores the relationship between the synthesis problem and the analysis problem. By writing the synthesis problem in the form of matrix operations, we derive an efficient algorithm for computing the DCT.

This algorithm is based on a matrix, M, that is orthogonal and symmetric, so it has the unusual property that the inverse of M is M (within a factor of two, anyway). As a result, the function we wrote to compute DCT-IV also computes the inverse DCT.

For the analysis problem, we started with a solution that takes time proportional to n^3 and improved it to take time proportional to n^2. It turns out that there is another optimization that get the run time down to $n \log n$. That algorithm is implemented in `scipy`, so we will use it in the next chapter. And then in Chapter **??** we will see how it works.

The code from this chapter is available from `http://think-dsp.com/example4.py`.

6.8 Exercises

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

Chapter 7

Using the DCT

In the previous chapter we derived the DCT and wrote a simple implementation of DCT-IV. I chose DCT-IV because it is the easiest to explain, but in practice it is more common to use DCT-II and its inverse, DCT-III. And unlike DCT-IV, the more common versions are available in `scipy`.