ThinkDSP

This notebook contains solutions to exercises in Chapter 2: Harmonics

In [1]:
```
from __future__ import print_function, division

%matplotlib inline

import thinkdsp
import thinkplot
import numpy as np
import math

import warnings
warnings.filterwarnings('ignore')

from IPython.html.widgets import interact, interact_manual, fixed
from IPython.html import widgets
from IPython.display import display

PI2 = 2 * math.pi
```

Exercise

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 (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?

Solution

My solution is basically a simplified version of TriangleSignal.

In [2]:
```
class SawtoothSignal(thinkdsp.Sinusoid):
    """Represents a sawtooth signal."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        cycles = self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = thinkdsp.normalize(thinkdsp.unbias(frac), self.amp)
        return ys
```

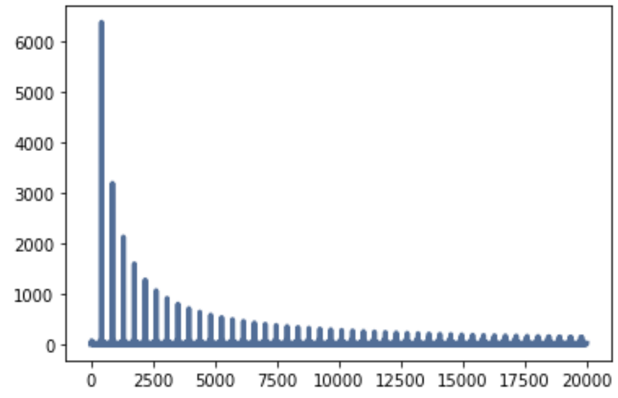
Here's what it sounds like:

```
In [3]: sawtooth = SawtoothSignal().make_wave(duration=0.5, framerate=40000)
        sawtooth.make_audio()
```

```
Out[3]:
        0:00:00 / 14:54:47
```

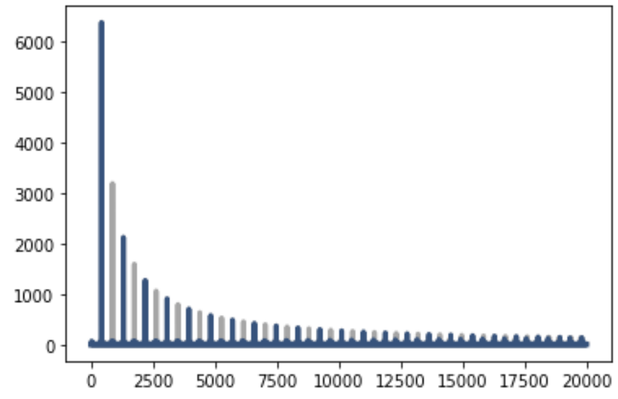
And here's what the spectrum looks like:

```
In [4]: sawtooth.make_spectrum().plot()
```



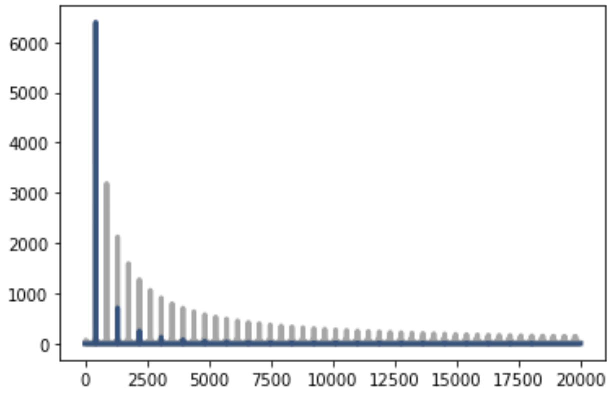
Compared to a square wave, the sawtooth drops off similarly, but it includes both even and odd harmonics. Notice that I had to cut the amplitude of the square wave to make them comparable.

```
In [5]: sawtooth.make_spectrum().plot(color='gray')
        square = thinkdsp.SquareSignal(amp=0.5).make_wave(duration=0.5, framerate=40
        000)
        square.make_spectrum().plot()
```



Compared to a triangle wave, the sawtooth doesn't drop off as fast.

In [6]:
```
sawtooth.make_spectrum().plot(color='gray')
triangle = thinkdsp.TriangleSignal(amp=0.79).make_wave(duration=0.5, framera
te=40000)
triangle.make_spectrum().plot()
```



Specifically, the harmonics of the triangle wave drop off in proportion to $1/f^2$, while the sawtooth drops off like $1/f$.

Exercise

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?

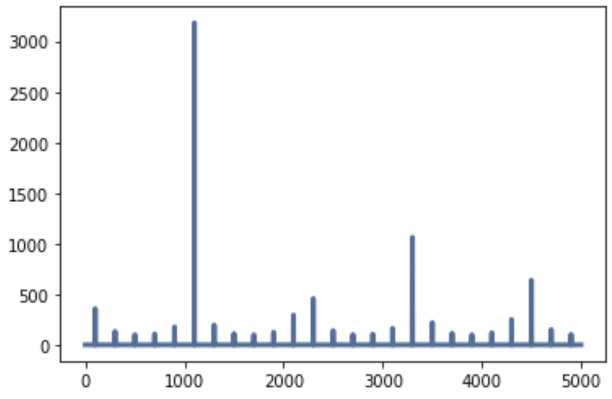
Solution

Here's the square wave:

In [7]:
```
square = thinkdsp.SquareSignal(1100).make_wave(duration=0.5, framerate=1000
0)
```

Here's what the spectrum looks like:

In [8]:
```
square.make_spectrum().plot()
```



The fundamental and the first harmonic are in the right place, but the second harmonic, which should be 5500 Hz, is aliased to 4500 Hz. The third, which should be 7700 Hz, is aliased to 2300 Hz, and so on.

When you listen to the wave, you can hear these aliased harmonics as a low tone.

In [9]:
```
square.make_audio()
```

Out[9]:

0:00 / 0:01

The tone I hear most prominently is at 300 Hz. If you listen to this 300 Hz sine wave, you might hear what I mean.

In [10]:
```
thinkdsp.SinSignal(300).make_wave(duration=0.5, framerate=10000).make_audio
()
```

Out[10]:

0:00 / 0:01

Exercise

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?

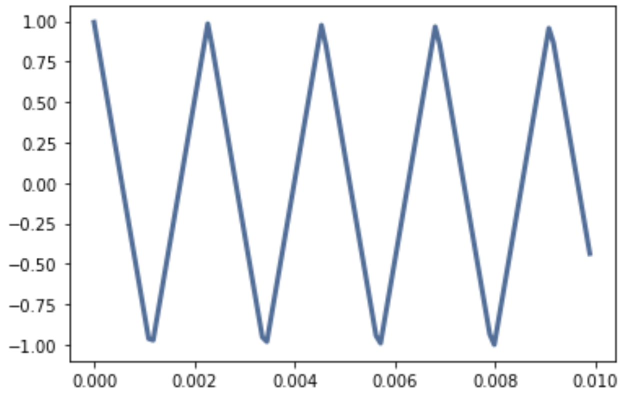
Try this experiment:

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?

Solution

Here's the triangle wave:

In [11]:
```
triangle = thinkdsp.TriangleSignal().make_wave(duration=0.01)
triangle.plot()
```

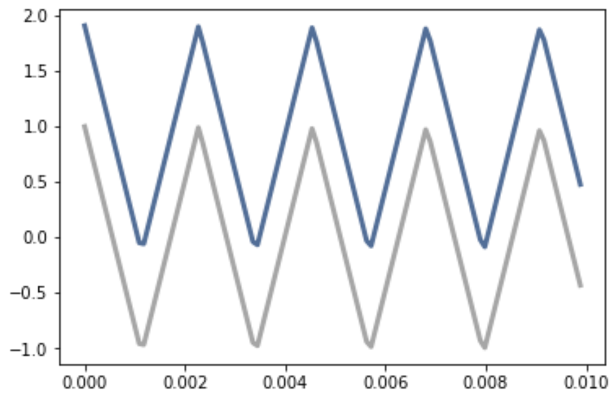


The first element of the spectrum is a complex number close to zero.

```
In [12]: spectrum = triangle.make_spectrum()
         spectrum.hs[0]

Out[12]: (1.0658141036401503e-14+0j)
```

If we add to the zero-frequency component, it has the effect of adding a vertical offset to the wave.

```
In [13]: spectrum.hs[0] = 100
         triangle.plot(color='gray')
         spectrum.make_wave().plot()
```



The zero-frequency component is the total of all the values in the signal, as we'll see when we get into the details of the DFT. If the signal is unbiased, the zero-frequency component is 0. In the context of electrical signals, the zero-frequency term is called the DC offset; that is, a direct current offset added to an AC signal.

Exercise

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. Test your function using one of the WAV files in the repository or any Wave object.

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?

Solution

Here's my version of the function:

```
In [14]: def filter_spectrum(spectrum):
             spectrum.hs /= spectrum.fs
             spectrum.hs[0] = 0
```

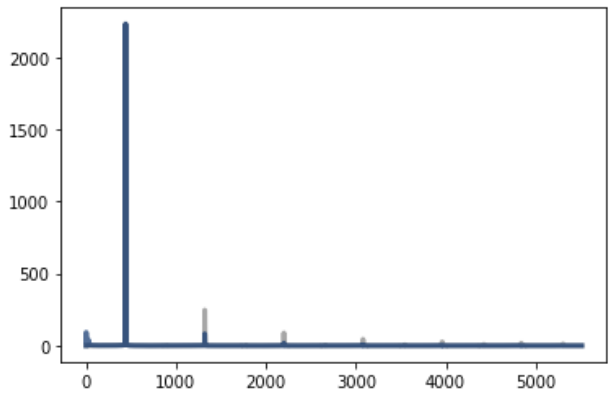
Here's a triangle wave:

```
In [15]: wave = thinkdsp.TriangleSignal(freq=440).make_wave(duration=0.5)
         wave.make_audio()

Out[15]:
```

0:00 / 0:01

Here's what the before and after look like. I scaled the after picture to make it visible on the same scale.

```
In [16]: high = 10000
         spectrum = wave.make_spectrum()
         spectrum.plot(high=high, color='gray')
         filter_spectrum(spectrum)
         spectrum.scale(440)
         spectrum.plot(high=high)
```



The filter clobbers the harmonics, so it acts like a low pass filter.

Here's what it sounds like:

```
In [17]: filtered = spectrum.make_wave()
         filtered.make_audio()
```

```
Out[17]:
```

0:00 / 0:01

The triangle wave now sounds almost like a sine wave.

Exercise

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

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.

Solution

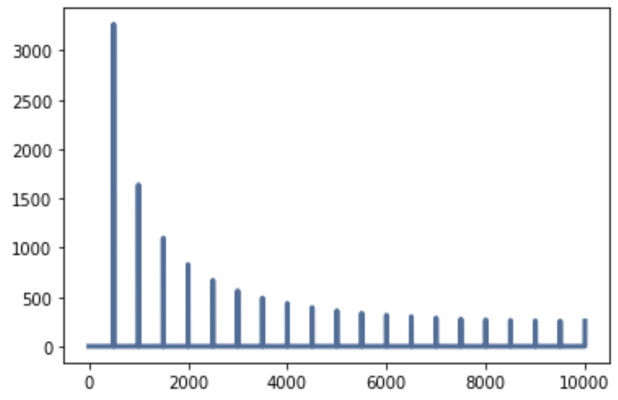
One option is to start with a sawtooth wave, which has all of the harmonics we need:

```
In [18]: freq = 500
         signal = thinkdsp.SawtoothSignal(freq=freq)
         wave = signal.make_wave(duration=0.5, framerate=20000)
         wave.make_audio()
```

```
Out[18]:
         0:00 / 0:01
```

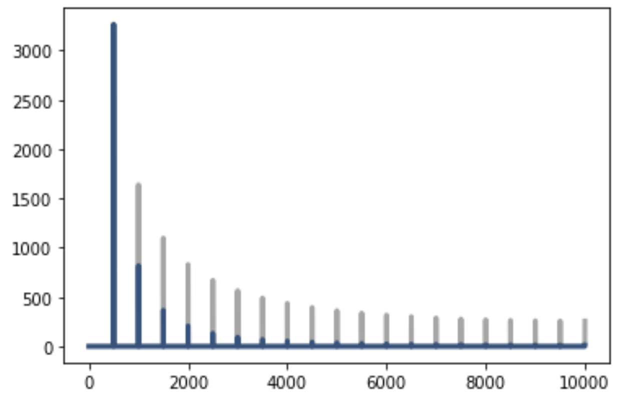
Here's what the spectrum looks like. The harmonics drop off like $1/f$.

```
In [19]: spectrum = wave.make_spectrum()
         spectrum.plot()
```



If we apply the filter we wrote in the previous exercise, we can make the harmonics drop off like $1/f^2$.

```
In [20]: spectrum.plot(color='gray')
         filter_spectrum(spectrum)
         spectrum.scale(freq)
         spectrum.plot()
```



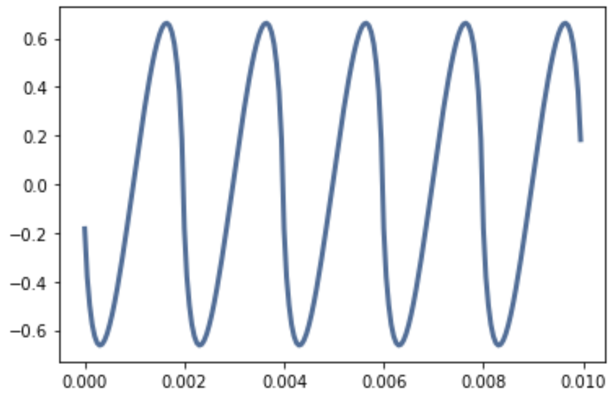
Here's what it sounds like:

```
In [21]: wave = spectrum.make_wave()
         wave.make_audio()
```

```
Out[21]:
         0:00 / 0:01
```

And here's what the waveform looks like.

```
In [22]: wave.segment(duration=0.01).plot()
```



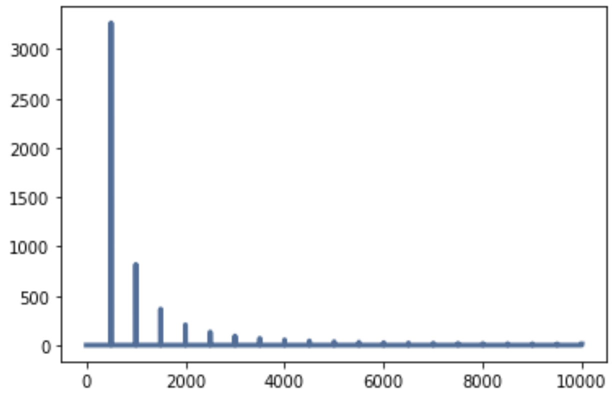
It's an interesting shape, but not easy to see what its functional form is.

Another approach is to add up a series of cosine signals with the right frequencies and amplitudes.

```
In [25]: freqs = np.arange(500, 9500, 500)
         amps = 1 / freqs**2
         signal = sum(thinkdsp.CosSignal(freq, amp) for freq, amp in zip(freqs, amp
         s))
         signal

Out[25]: <thinkdsp.SumSignal at 0x7fb08abfd2e8>
```

Here's what the spectrum looks like:

```
In [26]: spectrum = wave.make_spectrum()
         spectrum.plot()
```



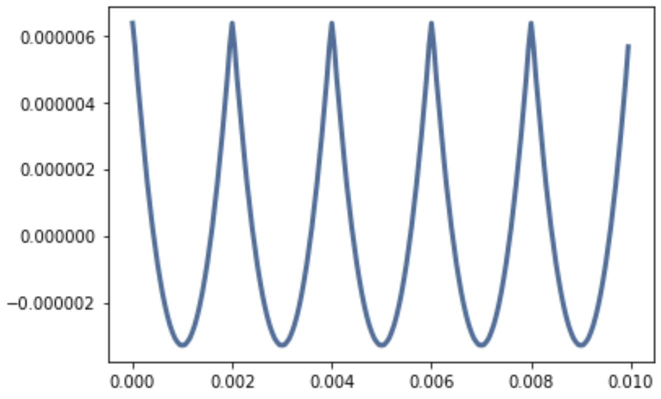
Here's what it sounds like:

```
In [27]: wave = signal.make_wave(duration=0.5, framerate=20000)
         wave.make_audio()

Out[27]:
```

0:00 / 0:01

And here's what the waveform looks like.

In [28]:
```
wave.segment(duration=0.01).plot()
```



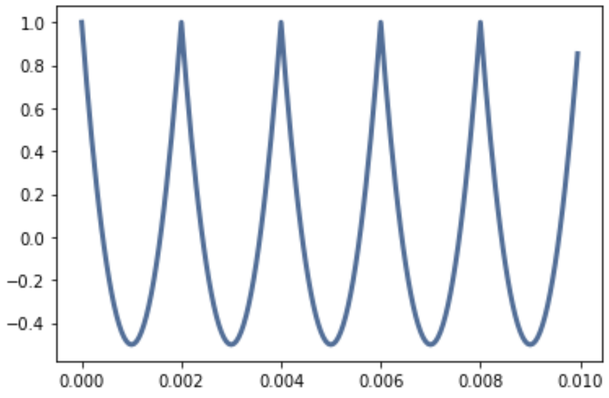
If those look to you like parabolas, you might be right. `thinkdsp` provides `ParabolicSignal` , which computes parabolic waveforms.

In [29]:
```
wave = thinkdsp.ParabolicSignal(freq=500).make_wave(duration=0.5, framerate=
20000)
wave.make_audio()
```

Out[29]: 0:00 / 0:01

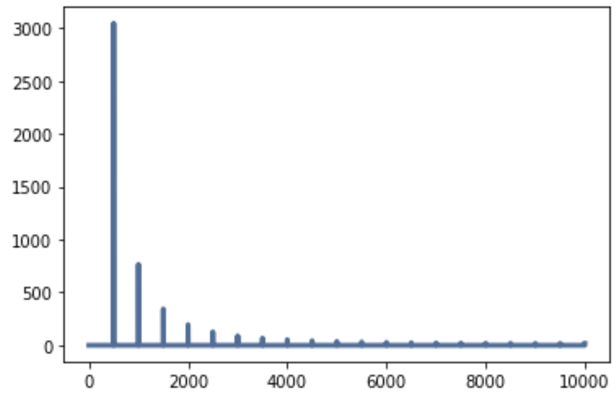
Here's what the waveform looks like:

In [30]:
```
wave.segment(duration=0.01).plot()
```



A parabolic signal has even and odd harmonics which drop off like $1/f^2$:

In [31]:
```
spectrum = wave.make_spectrum()
spectrum.plot()
```



ThinkDSP

This notebook contains solutions to exercises in Chapter 3: Non-periodic signals

In [1]:
```
from __future__ import print_function, division

%matplotlib inline

import thinkdsp
import thinkplot
import numpy as np

from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets
```

Exercise

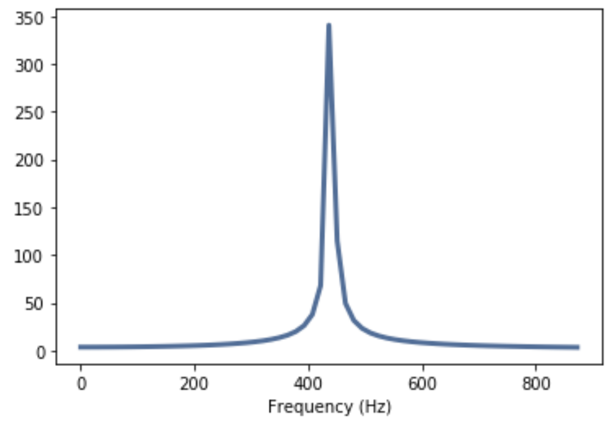
Run and listen to the examples in chap03.ipynb. In the leakage example, try replacing the Hamming window with one of the other windows provided by NumPy, and see what effect they have on leakage.

See http://docs.scipy.org/doc/numpy/reference/routines.window.html (http://docs.scipy.org/doc/numpy/reference/routines.window.html)

Solution

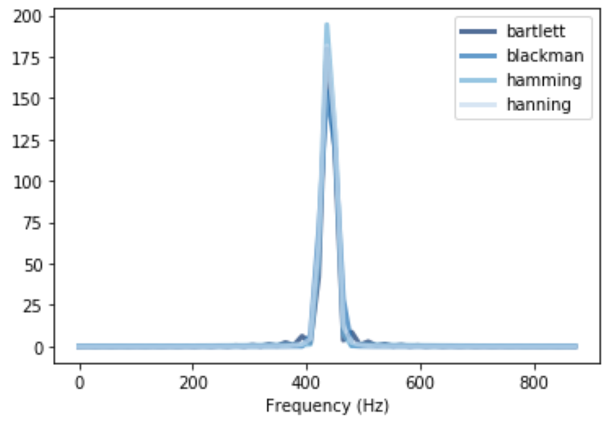
Here's the leakage example:

In [2]:
```
signal = thinkdsp.SinSignal(freq=440)
duration = signal.period * 30.25
wave = signal.make_wave(duration)
spectrum = wave.make_spectrum()

spectrum.plot(high=880)
thinkplot.config(xlabel='Frequency (Hz)')
```



The following figure shows the effect of 4 different windows.

In [3]:
```
thinkplot.preplot(4)

for window_func in [np.bartlett, np.blackman, np.hamming, np.hanning]:
    wave = signal.make_wave(duration)
    wave.ys *= window_func(len(wave.ys))

    spectrum = wave.make_spectrum()
    spectrum.plot(high=880, label=window_func.__name__)

thinkplot.config(xlabel='Frequency (Hz)', legend=True)
```



All four do a good job of reducing leakage. The Bartlett filter leaves some residual "ringing". The Hamming filter dissipates the least amount of energy.

Exercise

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

In [4]:
```
import math
PI2 = 2 * math.pi

class SawtoothChirp(thinkdsp.Chirp):
    """Represents a sawtooth signal with varying frequency."""

    def _evaluate(self, ts, freqs):
        """Helper function that evaluates the signal.

        ts: float array of times
        freqs: float array of frequencies during each interval
        """
        dts = np.diff(ts)
        dps = PI2 * freqs * dts
        phases = np.cumsum(dps)
        phases = np.insert(phases, 0, 0)
        cycles = phases / PI2
        frac, _ = np.modf(cycles)
        ys = thinkdsp.normalize(thinkdsp.unbias(frac), self.amp)
        return ys
```

Here's what it sounds like.

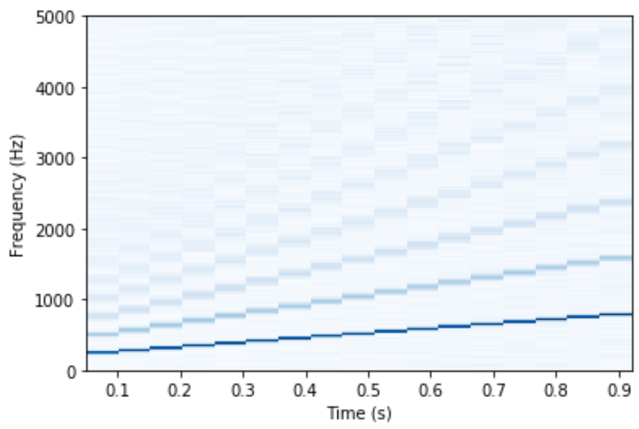
In [5]:
```
signal = SawtoothChirp(start=220, end=880)
wave = signal.make_wave(duration=1, framerate=10000)
wave.apodize()
wave.make_audio()
```

Out[5]:

0:00 / 0:01

And here's the spectrogram.

In [6]:
```
sp = wave.make_spectrogram(1024)
sp.plot()
thinkplot.config(xlabel='Time (s)', ylabel='Frequency (Hz)')
```



At a relatively low frame rate, you can see the aliased harmonics bouncing off the folding frequency. And you can hear them as a background hiss. If you crank up the frame rate, they go away.

By the way, if you are a fan of the original Star Trek series, you might recognize the sawtooth chirp as the red alert signal:

In [7]:
```
thinkdsp.read_wave('tos-redalert.wav').make_audio()
```

Out[7]:

0:00 / 0:01

Exercise

Make a sawtooth chirp that sweeps from 2500 to 3000 Hz, then make a wave with duration 1 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.

Solution

Since the fundamental sweeps from 2500 to 3000 Hz, I expect to see something like the Eye of Sauron in that range. The first harmonic sweeps from 5000 to 6000 Hz, so I expect a shorter tower in that range, like the Outhouse of Sauron. The second harmonic sweeps from 7500 to 9000 Hz, so I expect something even shorter in that range, like the Patio of Sauron.

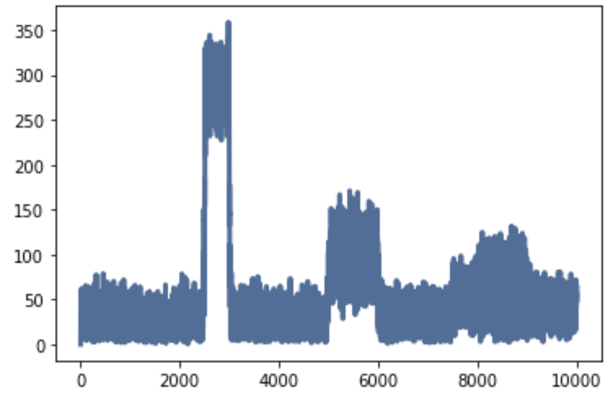
The other harmonics get aliased all over the place, so I expect to see some energy at all other frequencies. This distributed energy creates some interesting sounds.

In [8]:
```
signal = SawtoothChirp(start=2500, end=3000)
wave = signal.make_wave(duration=1, framerate=20000)
wave.make_audio()
```

Out[8]:

```
0:00:00 / 29:49:34
```

In [9]:
```
wave.make_spectrum().plot()
```



Exercise

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 its spectrogram.

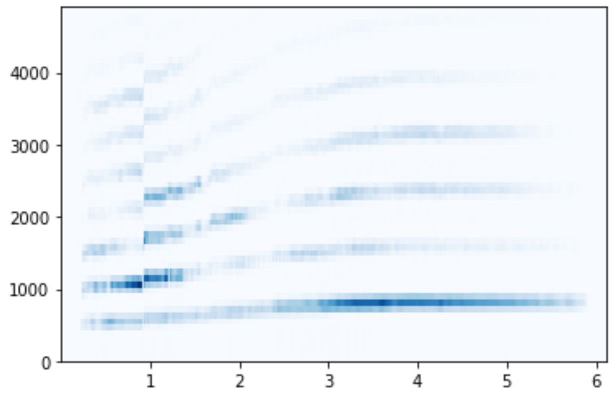
One suggestion: George Gershwin's *Rhapsody in Blue* starts with a famous clarinet glissando; you can download a recording from http://archive.org/details/rhapblue11924 (http://archive.org/details/rhapblue11924).

In [10]:
```
wave = thinkdsp.read_wave('72475__rockwehrmann__glissup02.wav')
wave.make_audio()
```

Out[10]:

```
0:00:00 / 13:31:36
```

In [11]:
```
wave.make_spectrogram(512).plot(high=5000)
```



Exercise

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 F3 down to C3 and back up to F3. 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?

In [12]:
```
class TromboneGliss(thinkdsp.Chirp):
    """Represents a trombone-like signal with varying frequency."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        l1, l2 = 1.0 / self.start, 1.0 / self.end
        lengths = np.linspace(l1, l2, len(ts)-1)
        freqs = 1 / lengths
        return self._evaluate(ts, freqs)
```

Here's the first part of the wave:

In [13]:
```
low = 262
high = 349
signal = TromboneGliss(high, low)
wave1 = signal.make_wave(duration=1)
wave1.apodize()
wave1.make_audio()
```

Out[13]:

0:00 / 0:01

And the second part:

In [14]:
```
signal = TromboneGliss(low, high)
wave2 = signal.make_wave(duration=1)
wave2.apodize()
wave2.make_audio()
```

Out[14]:

0:00 / 0:01

Putting them together:

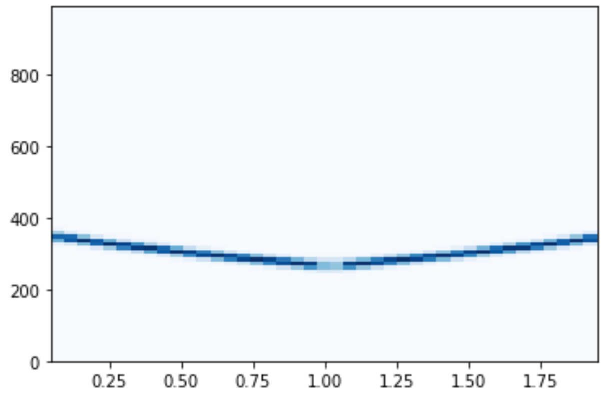
In [15]:
```
wave = wave1 | wave2
wave.make_audio()
```

Out[15]:

0:00:00 / 54:06:23

Here's the spectrogram:

In [16]:
```
sp = wave.make_spectrogram(1024)
sp.plot(high=1000)
```



Exercise

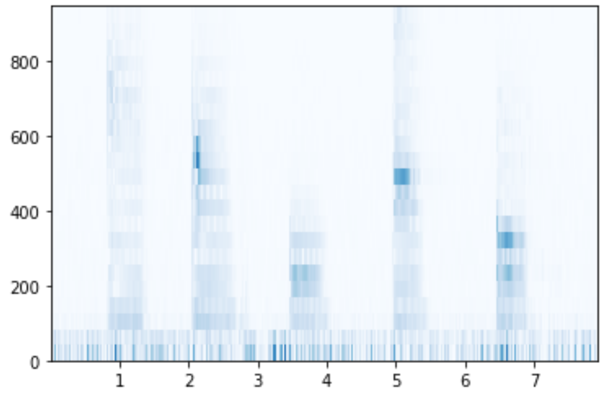
Make or find a recording of a series of vowel sounds and look at the spectrogram. Can you identify different vowels?

In [17]:
```
wave = thinkdsp.read_wave('87778__marcgascon7__vocals.wav')
wave.make_audio()
```

Out[17]:

0:00:00 / 13:31:36

In [18]:
```
wave.make_spectrogram(1024).plot(high=1000)
```

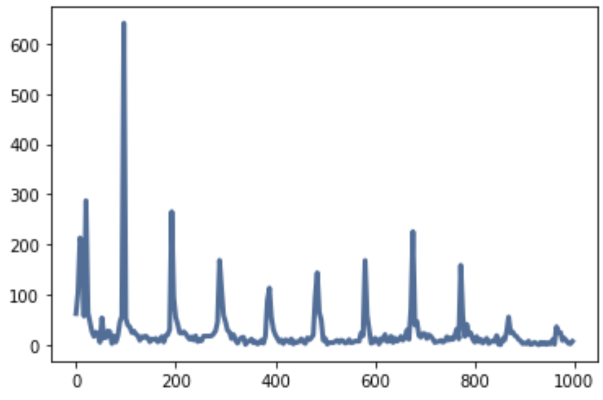


The stripe across the bottom is probably background noise. The peaks in the spectrogram are called "formants".

In general, vowel sounds are distinguished by the amplitude ratios of the first two formants relative to the fundamental. For more, see [https://en.wikipedia.org/wiki/Formant (https://en.wikipedia.org/wiki/Formant)](https://en.wikipedia.org/wiki/Formant)

We can see the formats more clearly by selecting a segment during 'ah'.

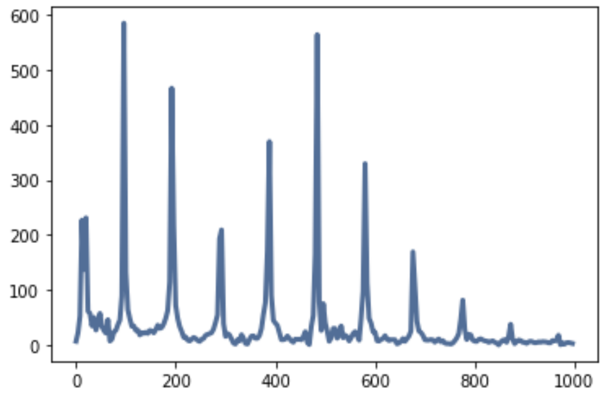
In [19]:
```
high = 1000
thinkplot.preplot(5)

segment = wave.segment(start=1, duration=0.25)
segment.make_spectrum().plot(high=high)
```



The fundamental is near 100 Hz. The next highest peaks are at 200 Hz and 700 Hz. People who know more about this than I do can identify vowels by looking at spectrums, but I can't.

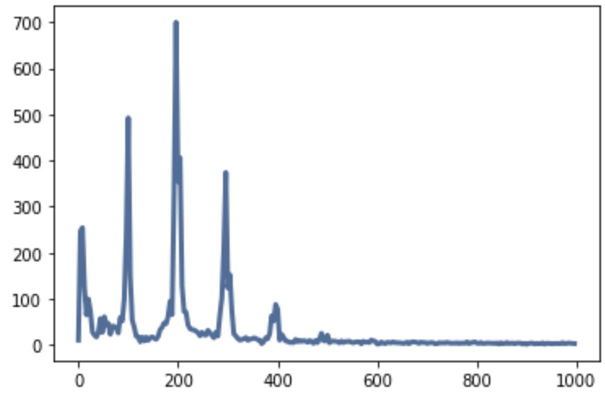
The 'eh' segment has a high-amplitude formant near 500 Hz.

In [20]:
```
segment = wave.segment(start=2.2, duration=0.25)
segment.make_spectrum().plot(high=high)
```



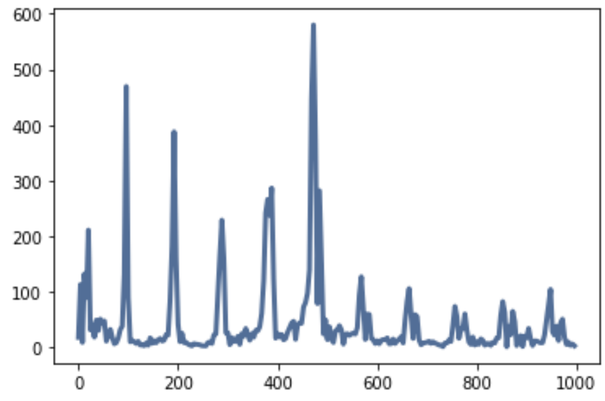
The 'ih' segment has no high frequency components.

In [21]:
```
segment = wave.segment(start=3.5, duration=0.25)
segment.make_spectrum().plot(high=high)
```



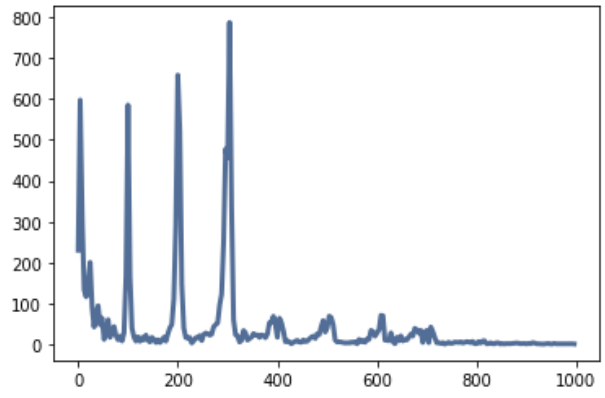
The 'oh' segment has a high-amplitude formant near 500 Hz, even higher than the fundamental.

In [22]:
```
segment = wave.segment(start=5.1, duration=0.25)
segment.make_spectrum().plot(high=high)
```



The 'oo' segment has a high-amplitude formant near 300 Hz and no high-frequency components

In [23]:
```
segment = wave.segment(start=6.5, duration=0.25)
segment.make_spectrum().plot(high=high)
```



ThinkDSP

This notebook contains solutions to exercises in Chapter 4: Noise

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot
import thinkstats2

import numpy as np
import pandas as pd

import warnings
warnings.filterwarnings('ignore')

from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets

%matplotlib inline
```

Exercise: ``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/ (http://asoftmurmur.com/about/) you can find their list of recordings, most of which are at http://freesound.org (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?

In [2]:
```
wave = thinkdsp.read_wave('132736__ciccarelli__ocean-waves.wav')
wave.make_audio()
```

Out[2]:
```
0:00:00 / 12:25:39
```

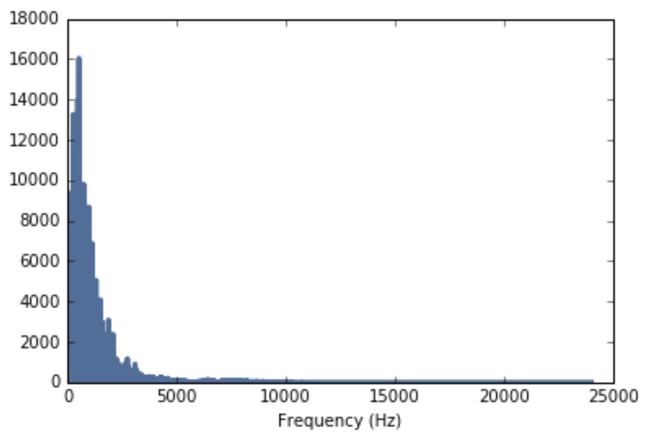
I chose a recording of ocean waves. I selected a short segment:

In [3]:
```
segment = wave.segment(start=1.5, duration=1.0)
segment.make_audio()
```

Out[3]:
```
0:00:00 / 12:25:39
```

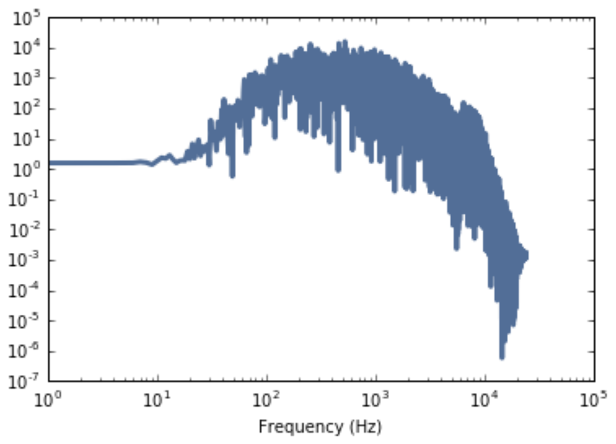
And here's its spectrum:

In [4]:
```
spectrum = segment.make_spectrum()
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (Hz)')
```



Amplitude drops off with frequency, so this might be red or pink noise. We can check by looking at the power spectrum on a log-log scale.

In [5]:
```
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (Hz)',
                 xscale='log',
                 yscale='log')
```



This structure, with increasing and then decreasing amplitude, seems to be common in natural noise sources.

Above $f = 10^3$, it might be dropping off linearly, but we can't really tell.

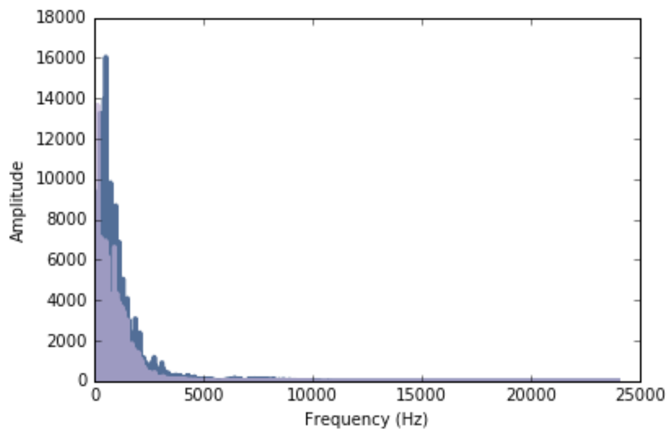
To see how the spectrum changes over time, I'll select another segment:

In [6]:
```
segment2 = wave.segment(start=2.5, duration=1.0)
segment2.make_audio()
```

Out[6]:
```
0:00:00 / 12:25:39
```

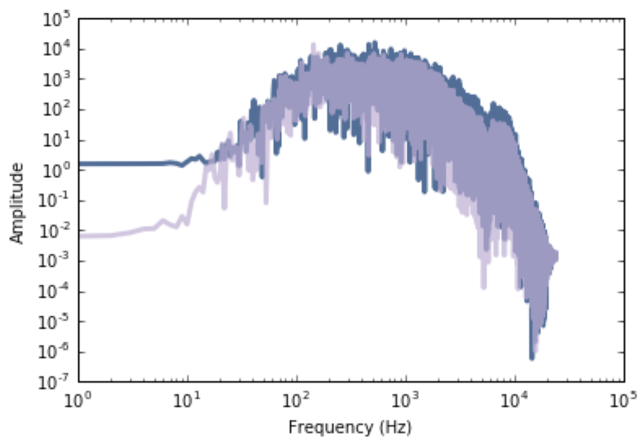
And plot the two spectrums:

In [7]:
```
spectrum2 = segment2.make_spectrum()
spectrum.plot_power()
spectrum2.plot_power(color='#beaed4')
thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude')
```



Here they are again, plotting power on a log-log scale.

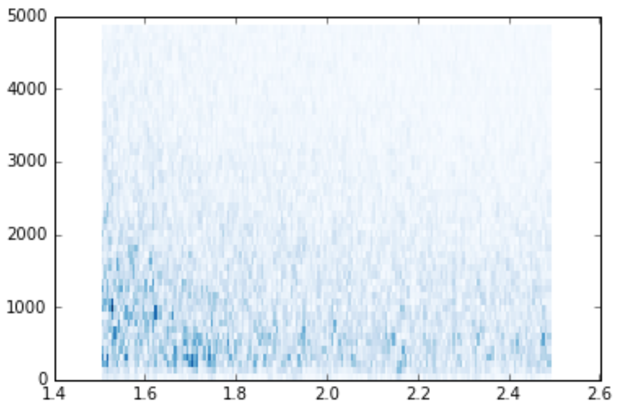
In [8]:
```
spectrum.plot_power()
spectrum2.plot_power(color='#beaed4')
thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude',
                 xscale='log',
                 yscale='log')
```



So the structure seems to be consistent over time.

We can also look at a spectrogram:

In [9]:
```
segment.make_spectrogram(512).plot(high=5000)
```



Within this segment, the overall amplitude drops off, but the mixture of frequencies seems consistent.

Exercise: 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.

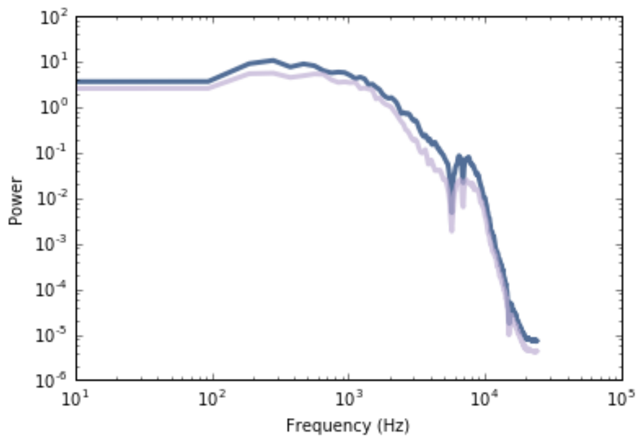
To estimate the long-term average power at each frequency, we can break a long signal into segments, compute the power spectrum for each segment, and then compute the average across the segments. You can read more about this algorithm at http://en.wikipedia.org/wiki/Bartlett's_method (http://en.wikipedia.org/wiki/Bartlett's_method).

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

In [10]:
```
def bartlett_method(wave, seg_length=512, win_flag=True):
    """Estimates the power spectrum of a noise wave.

    wave: Wave
    seg_length: segment length
    """
    # make a spectrogram and extract the spectrums
    spectro = wave.make_spectrogram(seg_length, win_flag)
    spectrums = spectro.spec_map.values()

    # extract the power array from each spectrum
    psds = [spectrum.power for spectrum in spectrums]

    # compute the root mean power (which is like an amplitude)
    hs = np.sqrt(sum(psds) / len(psds))
    fs = next(iter(spectrums)).fs

    # make a Spectrum with the mean amplitudes
    spectrum = thinkdsp.Spectrum(hs, fs, wave.framerate)
    return spectrum
```

`bartlett_method` makes a spectrogram and extracts `spec_map`, which maps from times to Spectrum objects. It computes the PSD for each spectrum, adds them up, and puts the results into a Spectrum object.

In [11]:
```
psd = bartlett_method(segment)
psd2 = bartlett_method(segment2)

psd.plot_power()
psd2.plot_power(color='#beaed4')

thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Power',
                 xscale='log',
                 yscale='log')
```

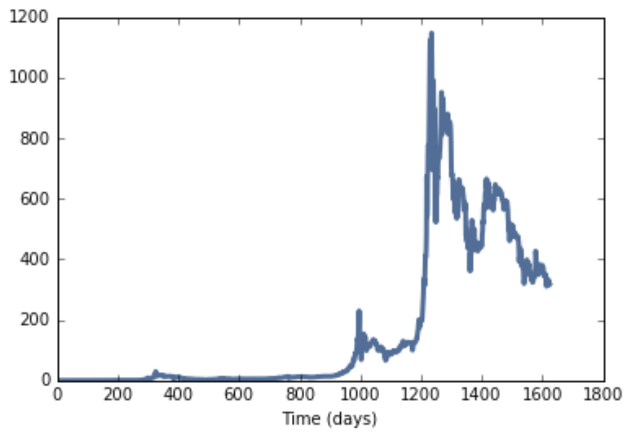


Now we can see the relationship between power and frequency more clearly. It is not a simple linear relationship, but it is consistent across different segments, even in details like the notches near 5000 Hz, 6000 Hz, and above 10,000 Hz.

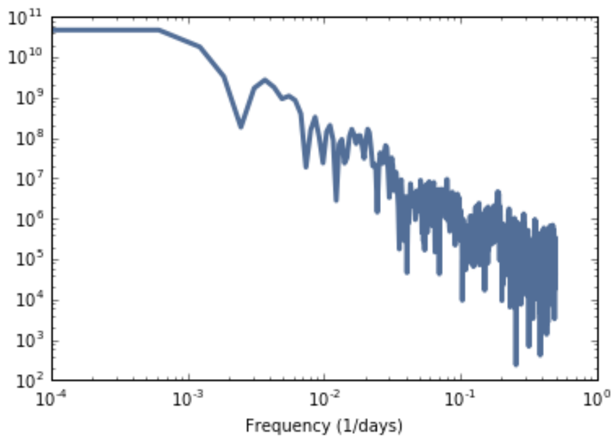
Exercise: At http://www.coindesk.com (http://www.coindesk.com) you can download the daily price of a BitCoin as a CSV file. Read this file and compute the spectrum of BitCoin prices as a function of time. Does it resemble white, pink, or Brownian noise?

In [12]:
```
df = pd.read_csv('coindesk-bpi-USD-close.csv', nrows=1625, parse_dates=[0])
ys = df.Close.values
ts = np.arange(len(ys))
```

In [13]:
```
wave = thinkdsp.Wave(ys, ts, framerate=1)
wave.plot()
thinkplot.config(xlabel='Time (days)')
```



In [14]:
```
spectrum = wave.make_spectrum()
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (1/days)',
                 xscale='log', yscale='log')
```



The slope is -1.8, which is similar to red noise (which should have a slope of -2).

In [15]:
```
spectrum.estimate_slope()[0]
```

Out[15]:
```
-1.8048752734169031
```

Exercise: 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.

In [16]:
```
class UncorrelatedPoissonNoise(thinkdsp._Noise):
    """Represents uncorrelated Poisson noise."""

    def evaluate(self, ts):
        """Evaluates the signal at the given times.

        ts: float array of times

        returns: float wave array
        """
        ys = np.random.poisson(self.amp, len(ts))
        return ys
```

Here's what it sounds like at low levels of "radiation".

In [17]:
```
amp = 0.001
framerate = 10000
duration = 1

signal = UncorrelatedPoissonNoise(amp=amp)
wave = signal.make_wave(duration=duration, framerate=framerate)
wave.make_audio()
```

Out[17]:

0:00 / 0:01

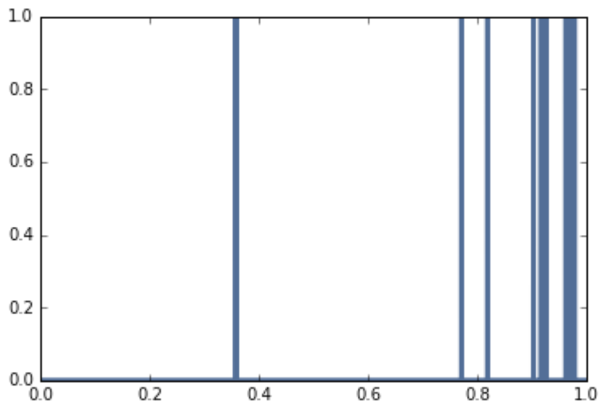
To check that things worked, we compare the expected number of particles and the actual number:

In [18]:
```
expected = amp * framerate * duration
actual = sum(wave.ys)
print(expected, actual)
```

```
10.0 10
```

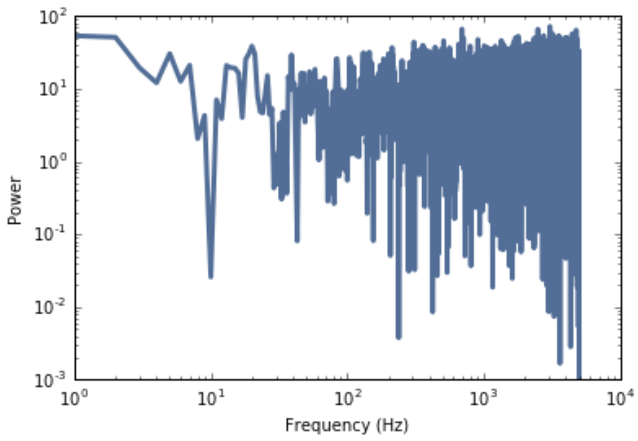
Here's what the wave looks like:

In [19]:
```
wave.plot()
```



And here's its power spectrum on a log-log scale.

In [20]:
```
spectrum = wave.make_spectrum()
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Power',
                 xscale='log',
                 yscale='log')
```



Looks like white noise, and the slope is close to 0.

In [21]:
```
spectrum.estimate_slope().slope
```

Out[21]:
```
nan
```

With a higher arrival rate, it sounds more like white noise:

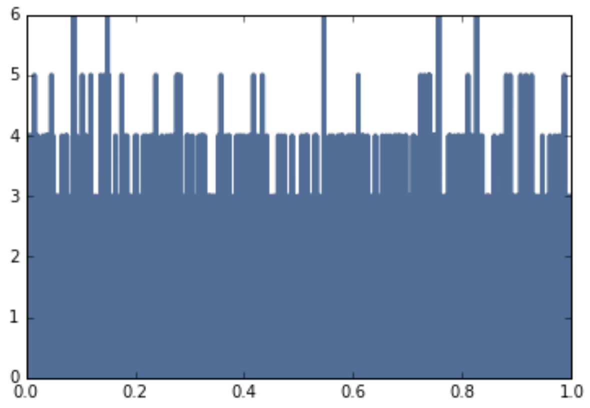
In [22]:
```
amp = 1
framerate = 10000
duration = 1

signal = UncorrelatedPoissonNoise(amp=amp)
wave = signal.make_wave(duration=duration, framerate=framerate)
wave.make_audio()
```

Out[22]:

0:00 / 0:01

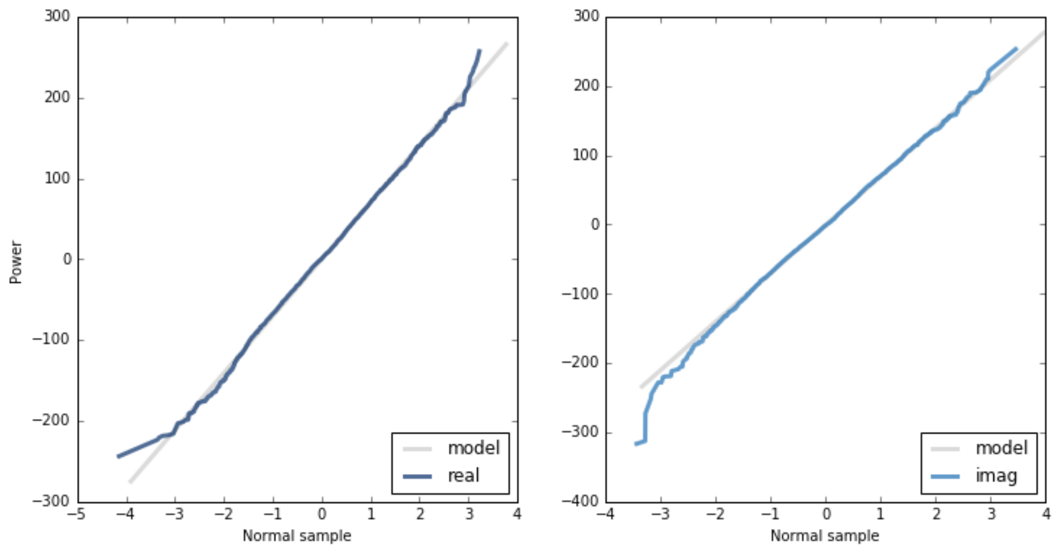
It looks more like a signal:

In [23]:
```
wave.plot()
```



And the spectrum converges on Gaussian noise.

In [24]:
```
spectrum = wave.make_spectrum()
spectrum.hs[0] = 0

thinkplot.preplot(2, cols=2)
thinkstats2.NormalProbabilityPlot(spectrum.real, label='real')
thinkplot.config(xlabel='Normal sample',
                 ylabel='Power',
                 legend=True,
                 loc='lower right')

thinkplot.subplot(2)
thinkstats2.NormalProbabilityPlot(spectrum.imag, label='imag')
thinkplot.config(xlabel='Normal sample',
                     loc='lower right')
```



Exercise: 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.

Solution: The fundamental idea of this algorithm is to add up several sequences of random numbers that get updates at different sampling rates. The first source should get updated at every time step; the second source every other time step, the third source ever fourth step, and so on.

In the original algorithm, the updates are evenly spaced. In an alternative proposed at http://www.firstpr.com.au/dsp/pink-noise/ (http://www.firstpr.com.au/dsp/pink-noise/), they are randomly spaced.

My implementation starts with an array with one row per timestep and one column for each of the white noise sources. Initially, the first row and the first column are random and the rest of the array is Nan.

In [25]:
```
nrows = 100
ncols = 5

array = np.empty((nrows, ncols))
array.fill(np.nan)
array[0, :] = np.random.random(ncols)
array[:, 0] = np.random.random(nrows)
array[0:6]
```

Out[25]:
```
array([[ 0.67651918,  0.78247984,  0.19970621,  0.74959508,  0.79686574],
       [ 0.01947641,         nan,         nan,         nan,         nan],
       [ 0.86137914,         nan,         nan,         nan,         nan],
       [ 0.99768391,         nan,         nan,         nan,         nan],
       [ 0.7667415 ,         nan,         nan,         nan,         nan],
       [ 0.79678027,         nan,         nan,         nan,         nan]])
```

The next step is to choose the locations where the random sources change. If the number of rows is n, the number of changes in the first column is n, the number in the second column is $n/2$ on average, the number in the third column is $n/4$ on average, etc.

So the total number of changes in the matrix is $2n$ on average; since n of those are in the first column, the other n are in the rest of the matrix.

To place the remaining n changes, we generate random columns from a geometric distribution with $p = 0.5$. If we generate a value out of bounds, we set it to 0 (so the first column gets the extras).

In [26]:
```
p = 0.5
n = nrows
cols = np.random.geometric(p, n)
cols[cols >= ncols] = 0
cols
```

Out[26]:
```
array([0, 1, 2, 1, 2, 1, 1, 0, 2, 2, 1, 4, 2, 1, 2, 1, 2, 1, 1, 1, 2, 2, 0,
       1, 1, 1, 1, 2, 1, 2, 3, 1, 1, 3, 1, 1, 2, 1, 2, 1, 3, 2, 1, 1, 1, 3,
       1, 1, 3, 1, 3, 3, 2, 1, 2, 3, 3, 2, 1, 1, 1, 1, 3, 3, 1, 1, 1, 1, 1,
       1, 4, 3, 1, 2, 3, 4, 1, 1, 1, 1, 2, 1, 3, 1, 4, 1, 1, 1, 2, 2, 1, 3,
       2, 2, 0, 1, 4, 2, 1, 4])
```

Within each column, we choose a random row from a uniform distribution. Ideally we would choose without replacement, but it is faster and easier to choose with replacement, and I doubt it matters.

In [27]:
```
rows = np.random.randint(nrows, size=n)
rows
```

Out[27]:
```
array([23, 98, 31, 67, 40, 67, 72,  9, 90, 29, 29, 34, 53,  2, 73, 90, 57,
       57, 82, 85, 22, 28, 34, 80, 60, 20, 49, 87, 41, 26, 45, 91, 62, 82,
       62, 84, 83, 66, 98, 46,  7, 19, 59, 76, 87, 64, 81, 11, 82, 40, 29,
       66, 86, 13,  2, 38, 85, 20, 42, 14,  8, 99, 13, 69, 53, 52,  9, 54,
       94, 26, 74, 52, 84, 25, 89,  3, 75, 57, 46, 57, 58, 29, 37, 87, 90,
       28, 43, 85, 28, 65, 13, 74, 13,  3, 63, 16, 27, 21, 88, 69])
```

Now we can put random values at rach of the change points.

In [28]:
```
array[rows, cols] = np.random.random(n)
array[0:6]
```

Out[28]:
```
array([[ 0.67651918,  0.78247984,  0.19970621,  0.74959508,  0.79686574],
       [ 0.01947641,         nan,         nan,         nan,         nan],
       [ 0.86137914,  0.76081171,  0.83727687,         nan,         nan],
       [ 0.99768391,         nan,  0.88194749,         nan,  0.4568749 ],
       [ 0.7667415 ,         nan,         nan,         nan,         nan],
       [ 0.79678027,         nan,         nan,         nan,         nan]])
```

Next we want to do a zero-order hold to fill in the NaNs. NumPy doesn't do that, but Pandas does. So I'll create a DataFrame:

In [29]:
```
df = pd.DataFrame(array)
df.head()
```

Out[29]:

	0	1	2	3	4
0	0.676519	0.782480	0.199706	0.749595	0.796866
1	0.019476	NaN	NaN	NaN	NaN
2	0.861379	0.760812	0.837277	NaN	NaN
3	0.997684	NaN	0.881947	NaN	0.456875
4	0.766742	NaN	NaN	NaN	NaN

And then use `fillna` along the columns.

In [30]:
```
filled = df.fillna(method='ffill', axis=0)
filled.head()
```

Out[30]:

	0	1	2	3	4
0	0.676519	0.782480	0.199706	0.749595	0.796866
1	0.019476	0.782480	0.199706	0.749595	0.796866
2	0.861379	0.760812	0.837277	0.749595	0.796866
3	0.997684	0.760812	0.881947	0.749595	0.456875
4	0.766742	0.760812	0.881947	0.749595	0.456875

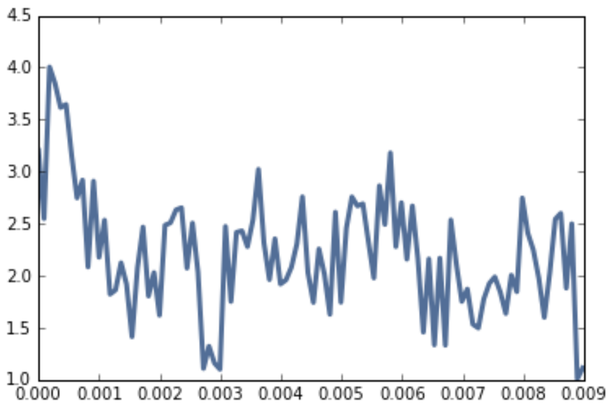
Finally we add up the rows.

In [31]:
```
total = filled.sum(axis=1)
total.head()
```

Out[31]:
```
0    3.205166
1    2.548123
2    4.005929
3    3.846913
4    3.615971
dtype: float64
```

If we put the results into a Wave, here's what it looks like:

In [32]:
```
wave = thinkdsp.Wave(total.values)
wave.plot()
```



Here's the whole process in a function:

In [33]:
```
def voss(nrows, ncols=16):
    """Generates pink noise using the Voss-McCartney algorithm.

    nrows: number of values to generate
    rcols: number of random sources to add

    returns: NumPy array
    """
    array = np.empty((nrows, ncols))
    array.fill(np.nan)
    array[0, :] = np.random.random(ncols)
    array[:, 0] = np.random.random(nrows)

    # the total number of changes is nrows
    n = nrows
    cols = np.random.geometric(0.5, n)
    cols[cols >= ncols] = 0
    rows = np.random.randint(nrows, size=n)
    array[rows, cols] = np.random.random(n)

    df = pd.DataFrame(array)
    df.fillna(method='ffill', axis=0, inplace=True)
    total = df.sum(axis=1)

    return total.values
```

To test it I'll generate 11025 values:

In [34]:
```
ys = voss(11025)
ys
```

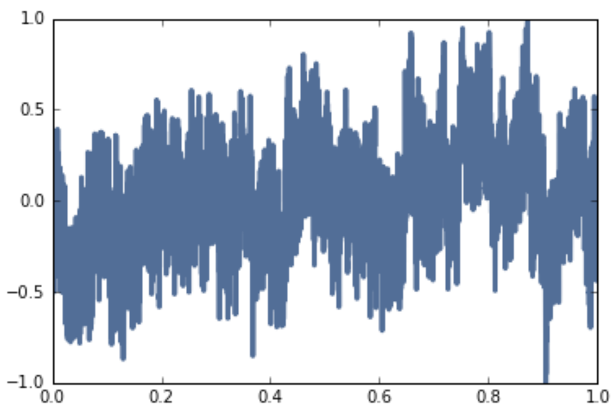
Out[34]:
```
array([ 9.0222124 ,  7.07720943,  7.5834063 , ...,  6.12007538,
        7.62269653,  7.81563447])
```

And make them into a Wave:

In [35]:
```
wave = thinkdsp.Wave(ys)
wave.unbias()
wave.normalize()
```

Here's what it looks like:

```
In [36]: wave.plot()
```



As expected, it is more random-walk-like than white noise, but more random looking than red noise.

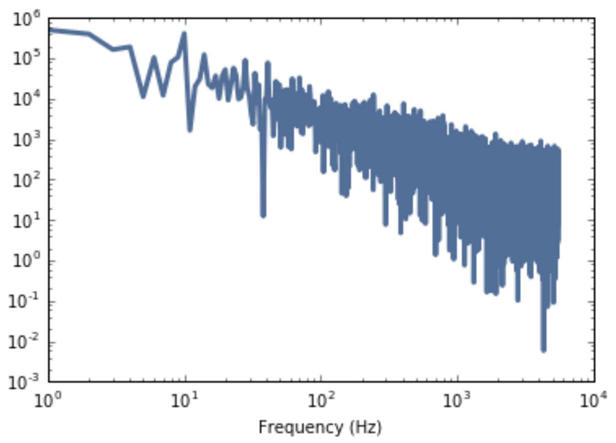
Here's what it sounds like:

```
In [37]: wave.make_audio()
Out[37]:
```

0:00 / 0:01

And here's the power spectrum:

```
In [38]: spectrum = wave.make_spectrum()
         spectrum.hs[0] = 0
         spectrum.plot_power()
         thinkplot.config(xlabel='Frequency (Hz)',
                          xscale='log',
                          yscale='log')
```



The estimated slope is close to -1.

```
In [39]: spectrum.estimate_slope().slope

Out[39]: -1.0169216785741335
```

We can get a better sense of the average power spectrum by generating a longer sample:

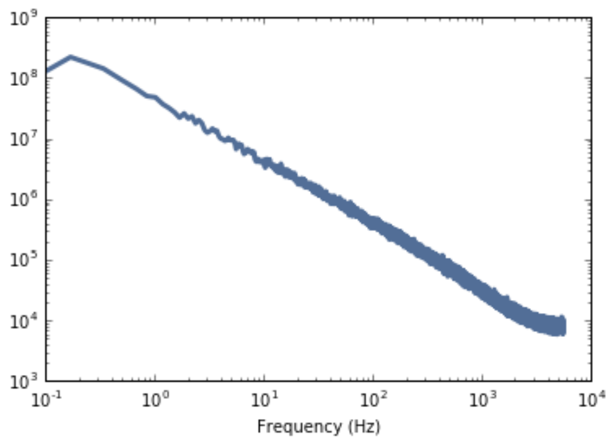
```
In [40]: seg_length = 64 * 1024
         iters = 100
         wave = thinkdsp.Wave(voss(seg_length * iters))
         len(wave)

Out[40]: 6553600
```

And using Barlett's method to compute the average.

```
In [41]: spectrum = bartlett_method(wave, seg_length=seg_length, win_flag=False)
         spectrum.hs[0] = 0
         len(spectrum)

Out[41]: 32769
```

It's pretty close to a straight line, with some curvature at the highest frequencies.

```
In [42]: spectrum.plot_power()
         thinkplot.config(xlabel='Frequency (Hz)',
                          xscale='log',
                          yscale='log')
```



And the slope is close to -1.

```
In [43]: spectrum.estimate_slope().slope

Out[43]: -1.0019307825072716

In [ ]:
```

ThinkDSP

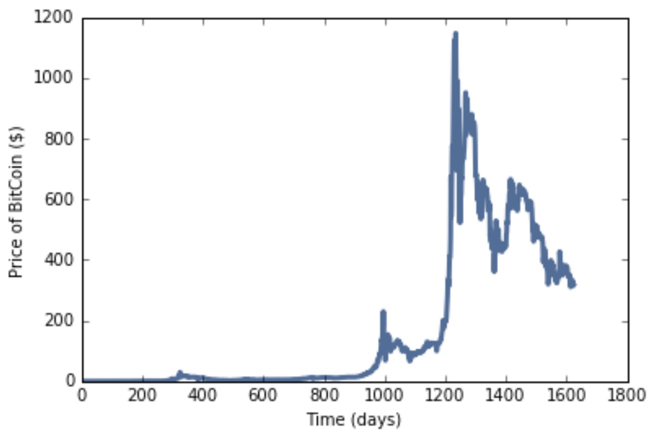
This notebook contains solutions to exercises in Chapter 5: Autocorrelation

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot
import thinkstats2

import numpy as np
import pandas as pd

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

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

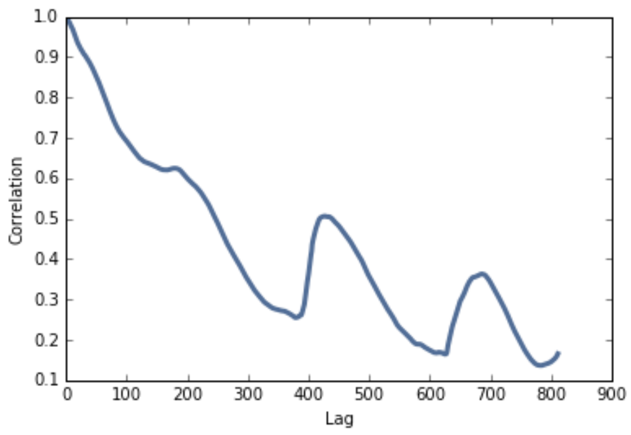
In [2]:
```
df = pd.read_csv('coindesk-bpi-USD-close.csv', nrows=1625, parse_dates=[0])
ys = df.Close.values
```

In [3]:
```
wave = thinkdsp.Wave(ys, framerate=1)
wave.plot()
thinkplot.config(xlabel='Time (days)',
                 ylabel='Price of BitCoin ($)')
```



Here's the autocorrelation function using the statistical definition, which unbiases, normalizes, and standardizes; that is, it shifts the mean to zero, divides through by standard deviation, and divides the sum by N.

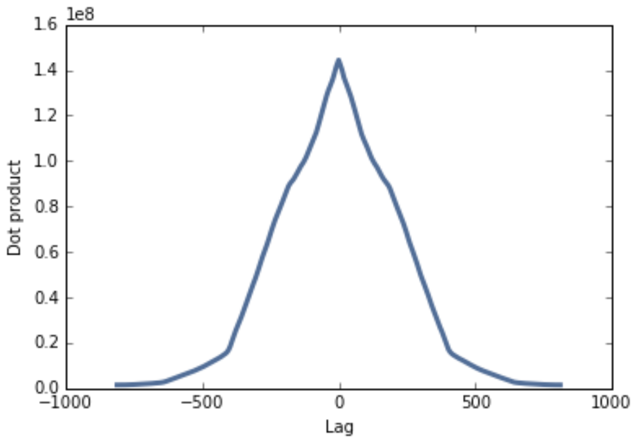
In [4]:
```
from autocorr import autocorr

lags, corrs = autocorr(wave)
thinkplot.plot(lags, corrs)
thinkplot.config(xlabel='Lag',
                 ylabel='Correlation')
```



The ACF drops off slowly as lag increases, suggesting some kind of pink noise. And it looks like there are moderate correlations with lags near 200, 425 and 700 days.

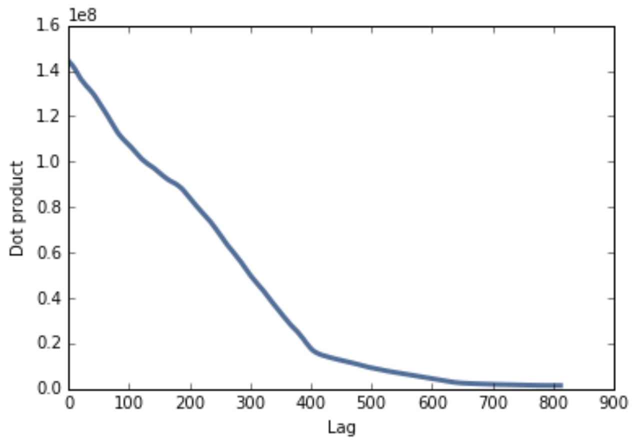
We can compare my implementation of `autocorr` with `np.correlate` , which uses the definition of correlation used in signal processing. It doesn't unbias, normalize, or standardize the wave.

In [5]:
```
N = len(wave)
corrs2 = np.correlate(wave.ys, wave.ys, mode='same')
lags = np.arange(-N//2, N//2)
thinkplot.plot(lags, corrs2)
thinkplot.config(xlabel='Lag',
                 ylabel='Dot product')
```



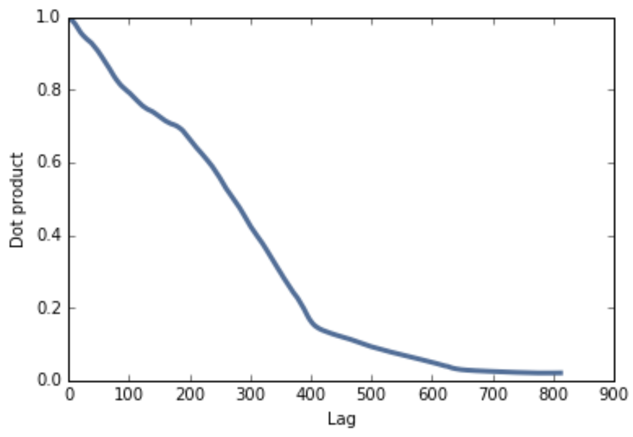
The second half of the result corresponds to positive lags:

In [6]:
```
N = len(corrs2)
half = corrs2[N//2:]
thinkplot.plot(half)
thinkplot.config(xlabel='Lag',
                 ylabel='Dot product')
```



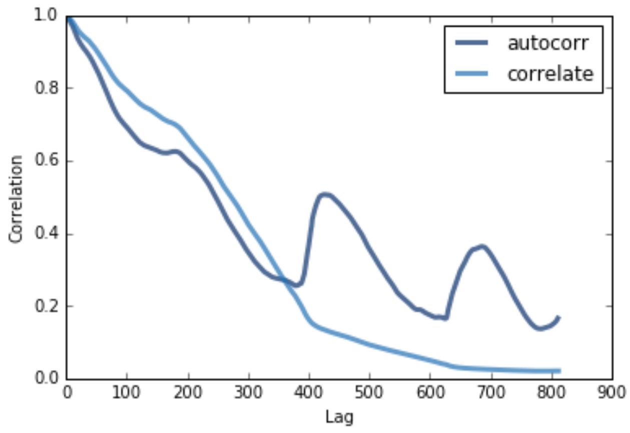
We can standardize the results after the fact by dividing through by `lengths`:

In [7]:
```
lengths = range(N, N//2, -1)
half /= lengths
half /= half[0]
thinkplot.plot(half)
thinkplot.config(xlabel='Lag',
                 ylabel='Dot product')
```



But even after standardizing, the results look very different. In the results from `correlate`, the peak at lag 200 is less apparent, and the other two peaks are obliterated.

In [8]:
```
thinkplot.preplot(2)
thinkplot.plot(corrs, label='autocorr')
thinkplot.plot(half, label='correlate')
thinkplot.config(xlabel='Lag', ylabel='Correlation')
```



I think the reason the results are so different the data look very different in different parts of the range; in particular, the variance changes a lot over time.

For this dataset, the statistical definition of ACF, is probably more appropriate.

Exercise: 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.

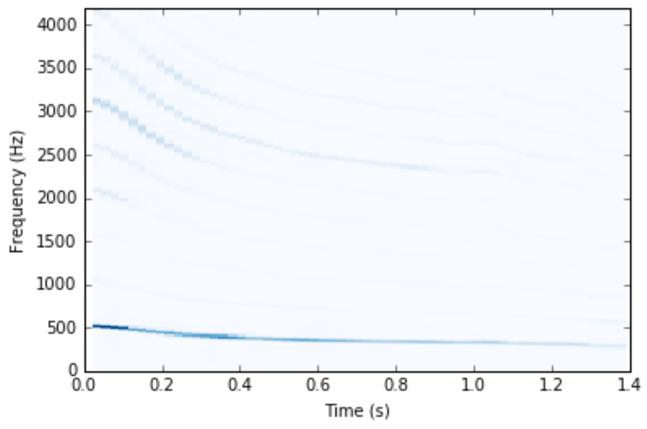
In [9]:
```
wave = thinkdsp.read_wave('28042__bcjordan__voicedownbew.wav')
wave.normalize()
wave.make_audio()
```

Out[9]:

```
0:00:00 / 13:31:36
```

I'll use the same example from `chap05.ipynb`. Here's the spectrogram:

In [10]:
```
wave.make_spectrogram(2048).plot(high=4200)
thinkplot.config(xlabel='Time (s)',
                 ylabel='Frequency (Hz)',
                 xlim=[0, 1.4],
                 ylim=[0, 4200])
```



And here's a function that encapsulates the code from Chapter 5. In general, finding the first, highest peak in the autocorrelation function is tricky. I kept it simple by specifying the range of lags to search.

In [11]:
```
def estimate_fundamental(segment, low=70, high=150):
    lags, corrs = autocorr(segment)
    lag = np.array(corrs[low:high]).argmax() + low
    period = lag / segment.framerate
    frequency = 1 / period
    return frequency
```

Here's an example of how it works.

In [12]:
```
duration = 0.01
segment = wave.segment(start=0.2, duration=duration)
freq = estimate_fundamental(segment)
freq
```

Out[12]:
```
436.63366336633663
```

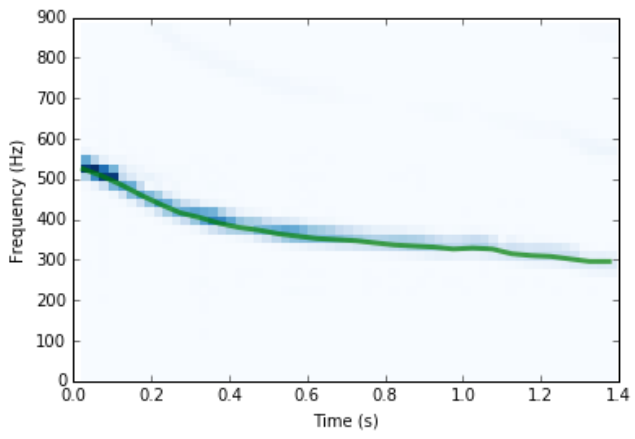
And here's a loop that tracks pitch over the sample.

The `ts` are the mid-points of each segment.

In [13]:
```
step = 0.05
starts = np.arange(0.0, 1.4, step)

ts = []
freqs = []

for start in starts:
    ts.append(start + step/2)
    segment = wave.segment(start=start, duration=duration)
    freq = estimate_fundamental(segment)
    freqs.append(freq)
```

Here's the pitch-tracking curve superimposed on the spectrogram:

In [14]:
```
wave.make_spectrogram(2048).plot(high=900)
thinkplot.plot(ts, freqs, color='green')
thinkplot.config(xlabel='Time (s)',
                 ylabel='Frequency (Hz)',
                 xlim=[0, 1.4],
                 ylim=[0, 900])
```



Looks pretty good!

In [ ]:

ThinkDSP

This notebook contains solutions to exercises in Chapter 6: Discrete Cosine Transform

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot
import thinkstats2

import numpy as np
import scipy.fftpack

import warnings
warnings.filterwarnings('ignore')

import dct

%matplotlib inline
```

Exercise: In this chapter I claim that `analyze1` takes time proportional to n^3 and `analyze2` takes time proportional to n^2. To see if that's true, run them on a range of input sizes and time them. In IPython, you can use the magic command `%timeit`.

If you plot run time versus input size on a log-log scale, you should get a straight line with slope 3 for `analyze1` and slope 2 for `analyze2`. You also might want to test `dct_iv` and `scipy.fftpack.dct`.

I'll start with a noise signal and an array of power-of-two sizes

In [2]:
```
signal = thinkdsp.UncorrelatedGaussianNoise()
noise = signal.make_wave(duration=1.0, framerate=16384)
noise.ys.shape
```

Out[2]:
```
(16384,)
```

In [3]:
```
ns = 2 ** np.arange(6, 15)
ns
```

Out[3]:
```
array([   64,   128,   256,   512,  1024,  2048,  4096,  8192, 16384])
```

The following function takes an array of results from a timing experiment, plots the results, and fits a straight line.

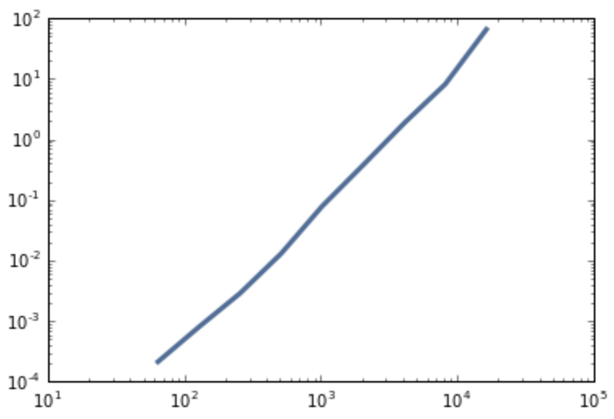
In [4]:
```
def plot_bests(bests):
    thinkplot.plot(ns, bests)
    thinkplot.config(xscale='log', yscale='log', legend=False)

    x = np.log(ns)
    y = np.log(bests)
    t = scipy.stats.linregress(x,y)
    slope = t[0]

    return slope
```

Here are the results for `analyze1` .

In [5]:
```
results = []
for N in ns:
    print(N)
    ts = (0.5 + np.arange(N)) / N
    freqs = (0.5 + np.arange(N)) / 2
    ys = noise.ys[:N]
    result = %timeit -r1 -o dct.analyze1(ys, freqs, ts)
    results.append(result)

bests = [result.best for result in results]
plot_bests(bests)
```

```
64
The slowest run took 20.00 times longer than the fastest. This could mean tha
t an intermediate result is being cached
1000 loops, best of 1: 215 µs per loop
128
The slowest run took 29.29 times longer than the fastest. This could mean tha
t an intermediate result is being cached
1000 loops, best of 1: 807 µs per loop
256
100 loops, best of 1: 2.89 ms per loop
512
100 loops, best of 1: 13.1 ms per loop
1024
10 loops, best of 1: 79.2 ms per loop
2048
1 loops, best of 1: 380 ms per loop
4096
1 loops, best of 1: 1.87 s per loop
8192
1 loops, best of 1: 8.21 s per loop
16384
1 loops, best of 1: 1min 5s per loop
```

Out[5]:
```
2.2729126725573052
```



The estimated slope is close to 2, not 3, as expected. One possibility is that the performance of `np.linalg.solve` is nearly quadratic in this range of array sizes.

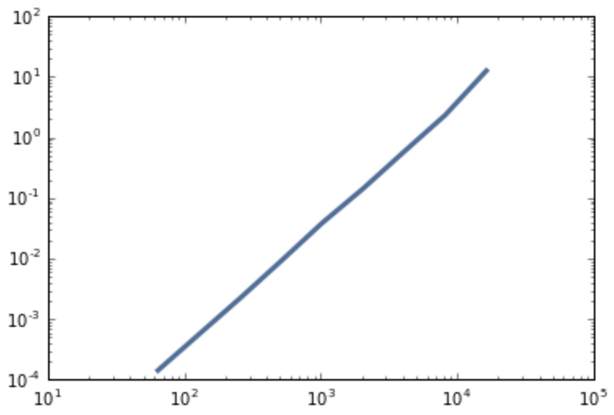
The line is curved, which suggests that we have not reached the array size where the runtime shows cubic growth. With larger array sizes, the estimated slope increases, so maybe it eventually converges on 3.

Here are the results for `analyze2` :

In [6]:
```
results = []
for N in ns:
    ts = (0.5 + np.arange(N)) / N
    freqs = (0.5 + np.arange(N)) / 2
    ys = noise.ys[:N]
    result = %timeit -r1 -o dct.analyze2(ys, freqs, ts)
    results.append(result)

bests2 = [result.best for result in results]
plot_bests(bests2)
```

```
1000 loops, best of 1: 143 µs per loop
1000 loops, best of 1: 565 µs per loop
100 loops, best of 1: 2.22 ms per loop
100 loops, best of 1: 9.16 ms per loop
10 loops, best of 1: 38.8 ms per loop
10 loops, best of 1: 144 ms per loop
1 loops, best of 1: 588 ms per loop
1 loops, best of 1: 2.35 s per loop
1 loops, best of 1: 12.7 s per loop
```

Out[6]:
```
2.0312154911232745
```



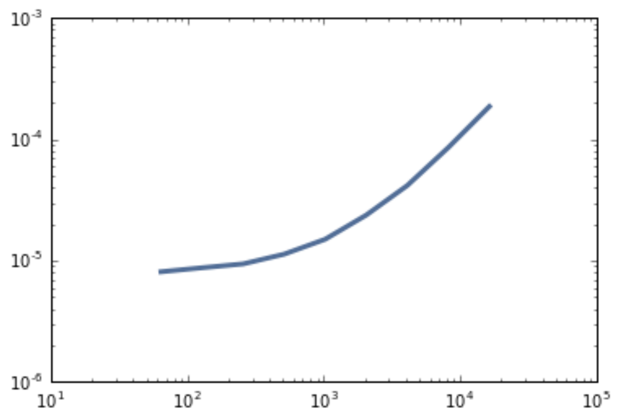
The results for `analyze2` fall in a straight line with the estimated slope close to 2, as expected.

Here are the results for the `scipy.fftpack.dct`

In [7]:
```
results = []
for N in ns:
    ys = noise.ys[:N]
    result = %timeit -o scipy.fftpack.dct(ys, type=3)
    results.append(result)

bests3 = [result.best for result in results]
plot_bests(bests3)
```

```
The slowest run took 12267.38 times longer than the fastest. This could mean
that an intermediate result is being cached
100000 loops, best of 3: 8.11 µs per loop
100000 loops, best of 3: 8.75 µs per loop
The slowest run took 4.03 times longer than the fastest. This could mean that
an intermediate result is being cached
100000 loops, best of 3: 9.42 µs per loop
The slowest run took 5.12 times longer than the fastest. This could mean that
an intermediate result is being cached
100000 loops, best of 3: 11.3 µs per loop
The slowest run took 4.82 times longer than the fastest. This could mean that
an intermediate result is being cached
100000 loops, best of 3: 15 µs per loop
The slowest run took 4.72 times longer than the fastest. This could mean that
an intermediate result is being cached
10000 loops, best of 3: 23.8 µs per loop
The slowest run took 4.62 times longer than the fastest. This could mean that
an intermediate result is being cached
10000 loops, best of 3: 41.7 µs per loop
The slowest run took 4.87 times longer than the fastest. This could mean that
an intermediate result is being cached
10000 loops, best of 3: 85.7 µs per loop
10000 loops, best of 3: 187 µs per loop
```

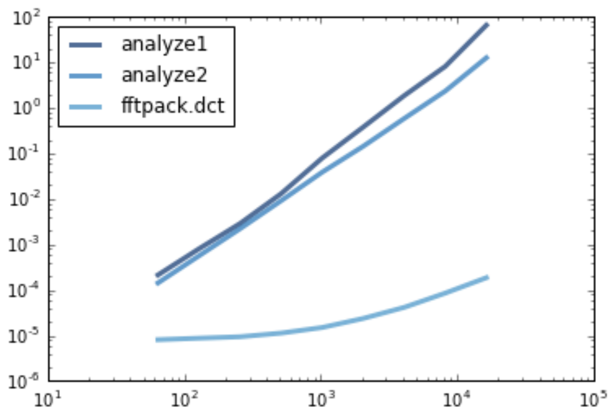
Out[7]:
```
0.55556666552707867
```



This implementation of dct is even faster. The line is curved, which means either we haven't seen the asymptotic behavior yet, or the asymptotic behavior is not a simple exponent of n. In fact, as we'll see soon, the run time is proportional to $n \log n$.

The following figure shows all three curves on the same axes.

In [8]:
```
thinkplot.preplot(3)
thinkplot.plot(ns, bests, label='analyze1')
thinkplot.plot(ns, bests2, label='analyze2')
thinkplot.plot(ns, bests3, label='fftpack.dct')
thinkplot.config(xscale='log', yscale='log', legend=True, loc='upper left')
```



In [ ]:

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

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

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

`thinkdsp` provides a class, `Dct` that is similar to a `Spectrum`, but which uses DCT instead of FFT.

As an example, I'll use a recording of a saxophone:

In [2]:
```
wave = thinkdsp.read_wave('100475__iluppai__saxophone-weep.wav')
wave.make_audio()
```

Out[2]:
```
0:00:00 / 13:31:36
```

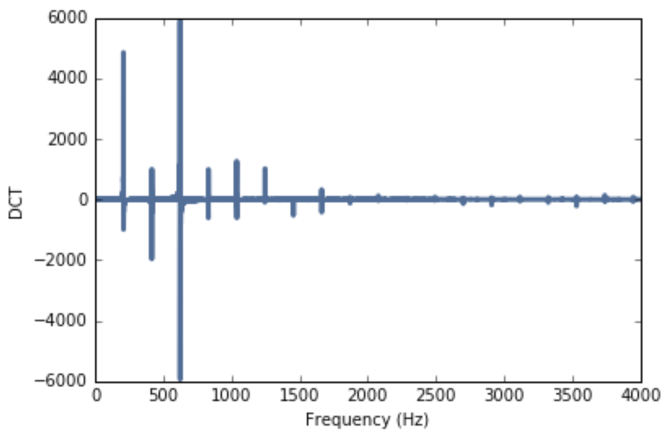
Here's a short segment:

In [3]:
```
segment = wave.segment(start=1.2, duration=0.5)
segment.normalize()
segment.make_audio()
```

Out[3]:
```
0:00:00 / 13:31:36
```

And here's the DCT of that segment:

```
In [4]: seg_dct = segment.make_dct()
        seg_dct.plot(high=4000)
        thinkplot.config(xlabel='Frequency (Hz)', ylabel='DCT')
```



There are only a few harmonics with substantial amplitude, and many entries near zero.

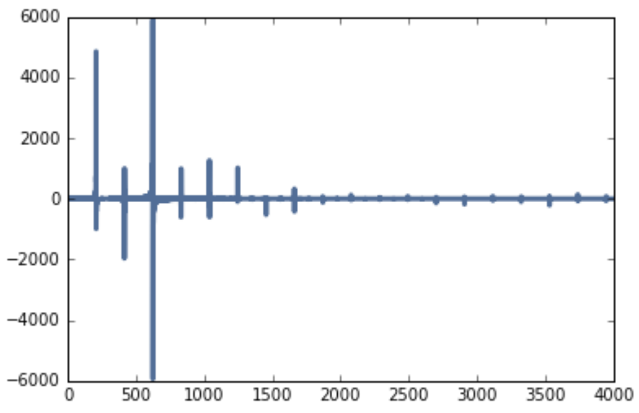
The following function takes a DCT and sets elements below `thresh` to 0.

```
In [5]: def compress(dct, thresh=1):
            count = 0
            for i, amp in enumerate(dct.amps):
                if abs(amp) < thresh:
                    dct.hs[i] = 0
                    count += 1

            n = len(dct.amps)
            print(count, n, 100 * count / n, sep='\t')
```

If we apply it to the segment, we can eliminate more than 90% of the elements:

```
In [6]: seg_dct = segment.make_dct()
        compress(seg_dct, thresh=10)
        seg_dct.plot(high=4000)
```

```
20457   22050   92.77551020408163
```



And the result sounds the same (at least to me):

In [7]:
```
seg2 = seg_dct.make_wave()
seg2.make_audio()
```

Out[7]:

```
0:00:00 / 13:31:36
```

To compress a longer segment, we can make a DCT spectrogram. The following function is similar to `wave.make_spectrogram` except that it uses the DCT.

In [8]:
```
def make_dct_spectrogram(wave, seg_length):
    """Computes the DCT spectrogram of the wave.

    seg_length: number of samples in each segment

    returns: Spectrogram
    """
    window = np.hamming(seg_length)
    i, j = 0, seg_length
    step = seg_length / 2

    # map from time to Spectrum
    spec_map = {}

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

        # the nominal time for this segment is the midpoint
        t = (segment.start + segment.end) / 2
        spec_map[t] = segment.make_dct()

        i += step
        j += step

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

Now we can make a DCT spectrogram and apply `compress` to each segment:

In [9]:
```
spectro = make_dct_spectrogram(wave, seg_length=1024)
for t, dct in sorted(spectro.spec_map.items()):
    compress(dct, thresh=0.2)
```

```
1018    1024    99.4140625
1016    1024    99.21875
1014    1024    99.0234375
1017    1024    99.31640625
1016    1024    99.21875
1017    1024    99.31640625
1016    1024    99.21875
1020    1024    99.609375
1014    1024    99.0234375
1005    1024    98.14453125
1009    1024    98.53515625
1015    1024    99.12109375
1015    1024    99.12109375
1016    1024    99.21875
1016    1024    99.21875
1015    1024    99.12109375
1017    1024    99.31640625
1020    1024    99.609375
1013    1024    98.92578125
1017    1024    99.31640625
1013    1024    98.92578125
1017    1024    99.31640625
1018    1024    99.4140625
1015    1024    99.12109375
1013    1024    98.92578125
794     1024    77.5390625
785     1024    76.66015625
955     1024    93.26171875
995     1024    97.16796875
992     1024    96.875
976     1024    95.3125
925     1024    90.33203125
802     1024    78.3203125
836     1024    81.640625
850     1024    83.0078125
882     1024    86.1328125
883     1024    86.23046875
891     1024    87.01171875
901     1024    87.98828125
902     1024    88.0859375
900     1024    87.890625
900     1024    87.890625
894     1024    87.3046875
904     1024    88.28125
901     1024    87.98828125
915     1024    89.35546875
913     1024    89.16015625
899     1024    87.79296875
905     1024    88.37890625
905     1024    88.37890625
888     1024    86.71875
898     1024    87.6953125
879     1024    85.83984375
893     1024    87.20703125
893     1024    87.20703125
882     1024    86.1328125
874     1024    85.3515625
876     1024    85.546875
864     1024    84.375
879     1024    85.83984375
869     1024    84.86328125
872     1024    85.15625
871     1024    85.05859375
878     1024    85.7421875
872     1024    85.15625
859     1024    83.88671875
879     1024    85.83984375
889     1024    86.81640625
```

In most segments, the compression is 75-80%.

To hear what it sounds like, we can convert the spectrogram back to a wave and play it.

In [10]:
```
wave2 = spectro.make_wave()
wave2.make_audio()
```

Out[10]:

0:00:00 / 13:31:36

And here's the original again for comparison.

In [11]:
```
wave.make_audio()
```

Out[11]:

0:00:00 / 13:31:36

As an experiment, you might try increasing `thresh` to see when the effect of compression becomes audible (to you).

Also, you might try compressing a signal with some noisy elements, like cymbals.

In [ ]:

ThinkDSP

This notebook contains solutions to exercises in Chapter 7: Discrete Fourier Transform

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot

import numpy as np

import warnings
warnings.filterwarnings('ignore')

PI2 = 2 * np.pi

np.set_printoptions(precision=3, suppress=True)
%matplotlib inline
```

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

The key to the FFT is the Danielson-Lanczos lemma:

$$DFT(y)[n] = DFT(e)[n] + exp(-2\pi in/N) DFT(o)[n]$$

Where $DFT(y)[n]$ is the nth element of the DFT of y; e is a wave array containing the even elements of y, and o contains the odd elements of y.

This lemma suggests a recursive algorithm for the DFT:

1. Given a wave array, y, split it into its even elements, e, and its odd elements, o.
2. Compute the DFT of e and o by making recursive calls.
3. Compute $DFT(y)$ for each value of n using the Danielson-Lanczos lemma.

For the base case of this recursion, you could wait until the length of y is 1. In that case, $DFT(y) = y$. Or if the length of y is sufficiently small, you could compute its DFT by matrix multiplication, possibly using a precomputed matrix.

Hint: I suggest you implement this algorithm incrementally by starting with a version that is not truly recursive. In Step 2, instead of making a recursive call, use `dft` or `np.fft.fft`. Get Step 3 working, and confirm that the results are consistent with the other implementations. Then add a base case and confirm that it works. Finally, replace Step 2 with recursive calls.

One more hint: Remember that the DFT is periodic; you might find `np.tile` useful.

You can read more about the FFT at https://en.wikipedia.org/wiki/Fast_Fourier_transform (https://en.wikipedia.org/wiki/Fast_Fourier_transform).

As the test case, I'll start with a small real signal and compute its FFT:

In [2]:
```
ys = [-0.5, 0.1, 0.7, -0.1]
hs = np.fft.fft(ys)
print(hs)
```

```
[ 0.2+0.j  -1.2-0.2j  0.2+0.j  -1.2+0.2j]
```

Here's my implementation of DFT from the book:

In [3]:
```
def dft(ys):
    N = len(ys)
    ts = np.arange(N) / N
    freqs = np.arange(N)
    args = np.outer(ts, freqs)
    M = np.exp(1j * PI2 * args)
    amps = M.conj().transpose().dot(ys)
    return amps
```

We can confirm that this implementation gets the same result.

In [4]:
```
hs2 = dft(ys)
print(sum(abs(hs - hs2)))
```

```
5.86477584677e-16
```

As a step toward making a recursive FFT, I'll start with a version that splits the input array and uses np.fft.fft to compute the FFT of the halves.

In [5]:
```
def fft_norec(ys):
    N = len(ys)
    He = np.fft.fft(ys[::2])
    Ho = np.fft.fft(ys[1::2])

    ns = np.arange(N)
    W = np.exp(-1j * PI2 * ns / N)

    return np.tile(He, 2) + W * np.tile(Ho, 2)
```

And we get the same results:

In [6]:
```
hs3 = fft_norec(ys)
print(sum(abs(hs - hs3)))
```

```
0.0
```

Finally, we can replace `np.fft.fft` with recursive calls, and add a base case:

In [7]:
```
def fft(ys):
    N = len(ys)
    if N == 1:
        return ys

    He = fft(ys[::2])
    Ho = fft(ys[1::2])

    ns = np.arange(N)
    W = np.exp(-1j * PI2 * ns / N)

    return np.tile(He, 2) + W * np.tile(Ho, 2)
```

And we get the same results:

In [8]:
```
hs4 = fft(ys)
print(sum(abs(hs - hs4)))
```

```
1.66533453694e-16
```

This implementation of FFT takes time proportional to $n \log n$. It also takes space proportional to $n \log n$, and it wastes some time making and copying arrays. It can be improved to run "in place"; in that case, it requires no additional space, and spends less time on overhead.

In [ ]:
```
```

ThinkDSP

This notebook contains solutions to exercises in Chapter 8: Filtering and Convolution

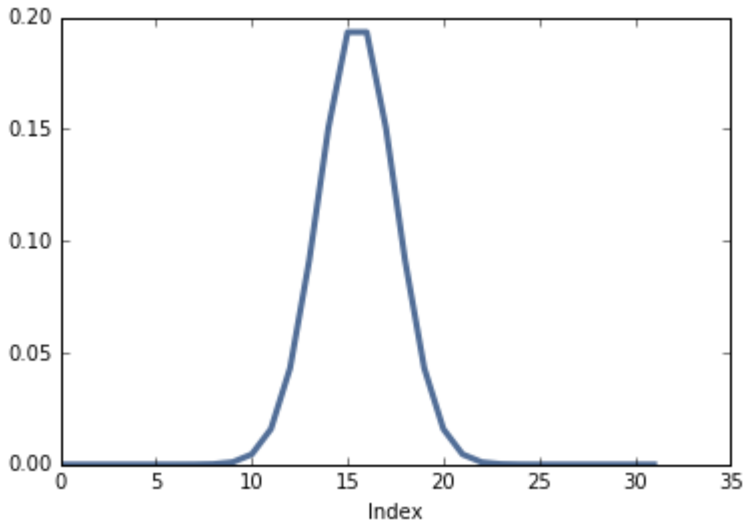
In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot

import numpy as np
import scipy.signal

import warnings
warnings.filterwarnings('ignore')

PI2 = 2 * np.pi

np.set_printoptions(precision=3, suppress=True)
%matplotlib inline
```

Exercise: In this chapter I claimed that the Fourier transform of a Gaussian curve is also a Gaussian curve. For discrete Fourier transforms, this relationship is approximately true.

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

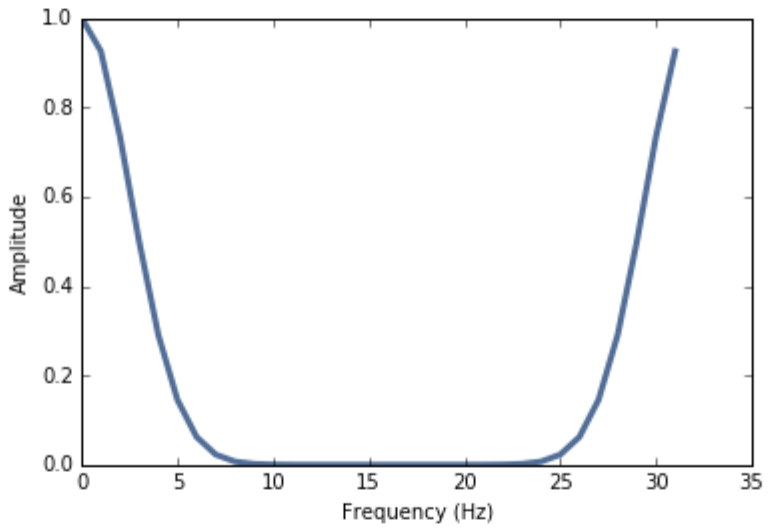
Solution: I'll start with a Gaussian similar to the example in the book.

In [2]:
```
gaussian = scipy.signal.gaussian(M=32, std=2)
gaussian /= sum(gaussian)
thinkplot.plot(gaussian)
thinkplot.config(xlabel='Index')
```



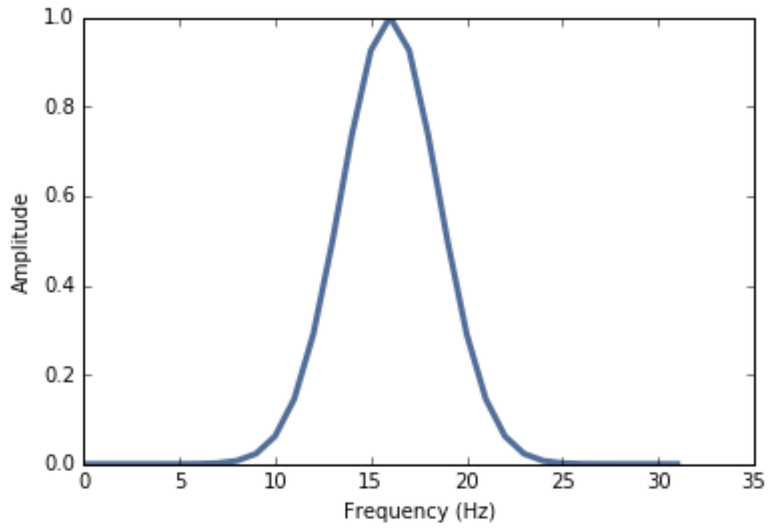
Here's what the FFT looks like:

In [3]:
```
fft_gaussian = np.fft.fft(gaussian)
thinkplot.plot(abs(fft_gaussian))
thinkplot.config(xlabel='Frequency (Hz)', ylabel='Amplitude
')
```



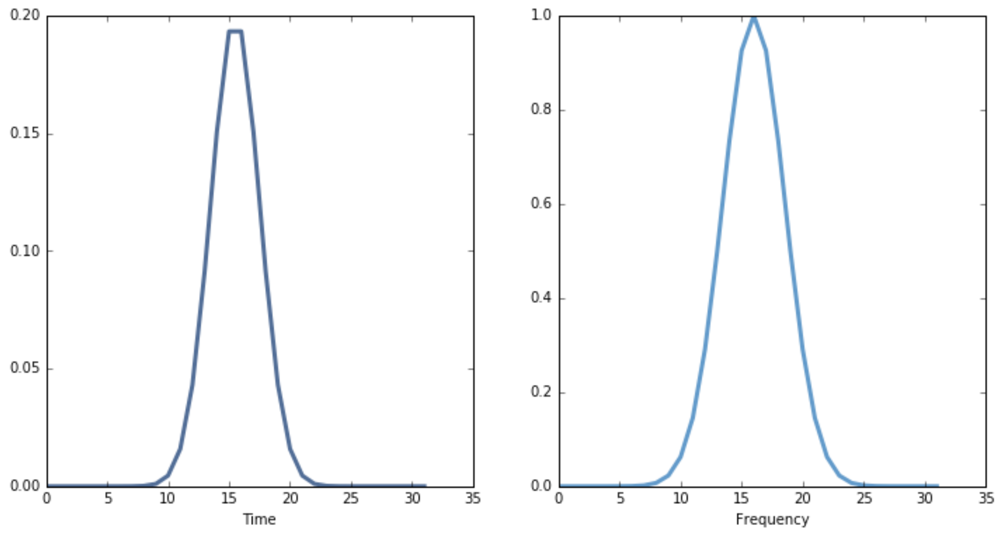
If we roll the negative frequencies around to the left, we can see more clearly that it is Gaussian, at least approximately.

In [4]:
```
N = len(gaussian)
fft_rolled = np.roll(fft_gaussian, N//2)
thinkplot.plot(abs(fft_rolled))
thinkplot.config(xlabel='Frequency (Hz)', ylabel='Amplitude
')
```



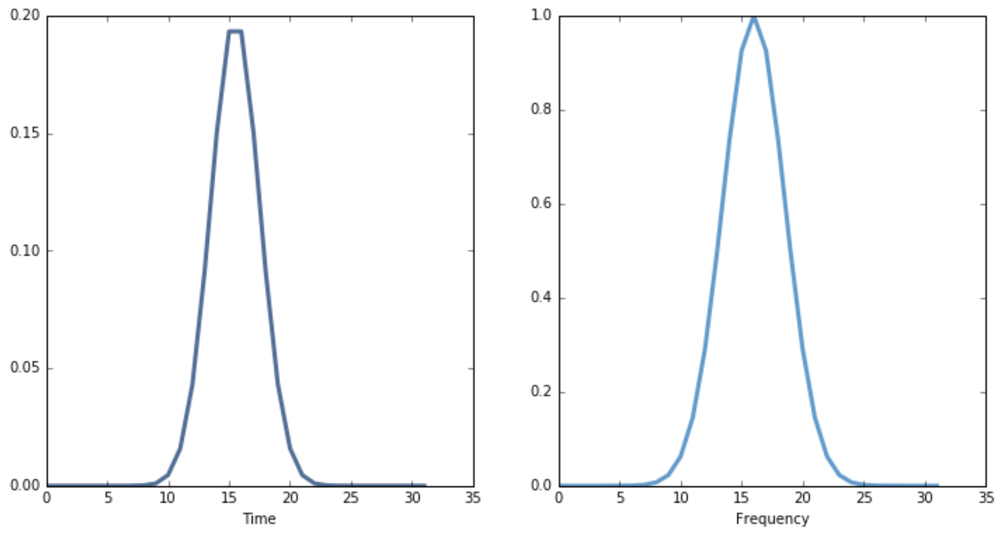
This function plots the Gaussian window and its FFT side-by-side.

In [5]:
```
def plot_gaussian(std):
    M = 32
    gaussian = scipy.signal.gaussian(M=M, std=std)
    gaussian /= sum(gaussian)

    thinkplot.preplot(num=2, cols=2)
    thinkplot.plot(gaussian)
    thinkplot.config(xlabel='Time', legend=False)

    fft_gaussian = np.fft.fft(gaussian)
    fft_rolled = np.roll(fft_gaussian, M//2)

    thinkplot.subplot(2)
    thinkplot.plot(abs(fft_rolled))
    thinkplot.config(xlabel='Frequency')


plot_gaussian(2)
```



Now we can make an interaction that shows what happens as `std` varies.

In [7]:
```
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets

slider = widgets.FloatSlider(min=0.1, max=10, value=2)
interact(plot_gaussian, std=slider);
```



As `std` increases, the Gaussian gets wider and its FFT gets narrower.

In terms of continuous mathematics, if

$$f(x) = e^{-ax^2}$$

which is a Gaussian with mean 0 and standard deviation $1/a$, its Fourier transform is

$$F(k) = \sqrt{\frac{\pi}{a}} e^{-\pi^2 k^2 / a}$$

which is a Gaussian with standard deviation a/π^2. So there is an inverse relationship between the standard deviations of f and F.

For the proof, see http://mathworld.wolfram.com/FourierTransformGaussian.html (http://mathworld.wolfram.com/FourierTransformGaussian.html)

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

In addition to the Gaussian window we used in this window, create a Hamming window with the same size. Zero pad the windows and plot their DFTs. Which window acts as a better low-pass filter? You might find it useful to plot the DFTs on a log-y scale.

Experiment with a few different windows and a few different sizes.

Solution: Following the examples from the chapter, I'll create a 1-second wave sampled at 44.1 kHz.

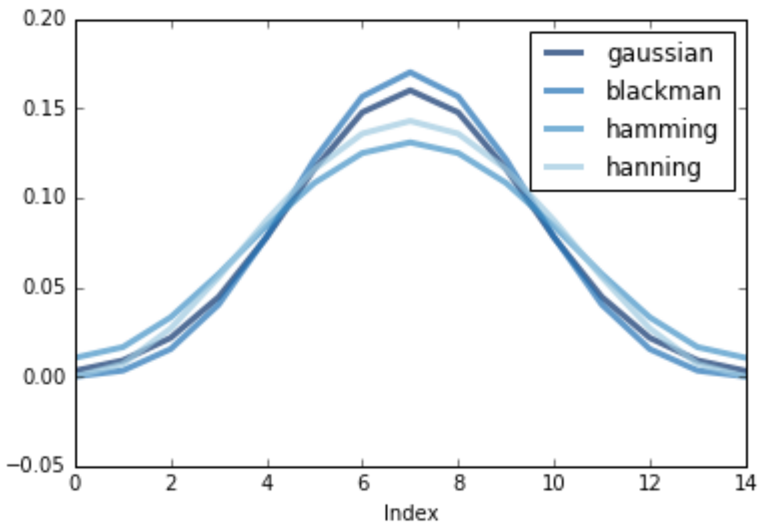
In [8]:
```
signal = thinkdsp.SquareSignal(freq=440)
wave = signal.make_wave(duration=1.0, framerate=44100)
```

And I'll create a few windows. I chose the standard deviation of the Gaussian window to make it similar to the others.

In [9]:
```
M = 15
std = 2.5

gaussian = scipy.signal.gaussian(M=M, std=std)
bartlett = np.bartlett(M)
blackman = np.blackman(M)
hamming = np.hamming(M)
hanning = np.hanning(M)

windows = [gaussian, blackman, hamming, hanning]
names = ['gaussian', 'blackman', 'hamming', 'hanning']

for window in windows:
    window /= sum(window)
```

Let's see what the windows look like.

```
In [11]: thinkplot.preplot(4)
         for window, name in zip(windows, names):
             thinkplot.plot(window, label=name)

         thinkplot.config(xlabel='Index', legend=True, loc='center b
         ottom')
```

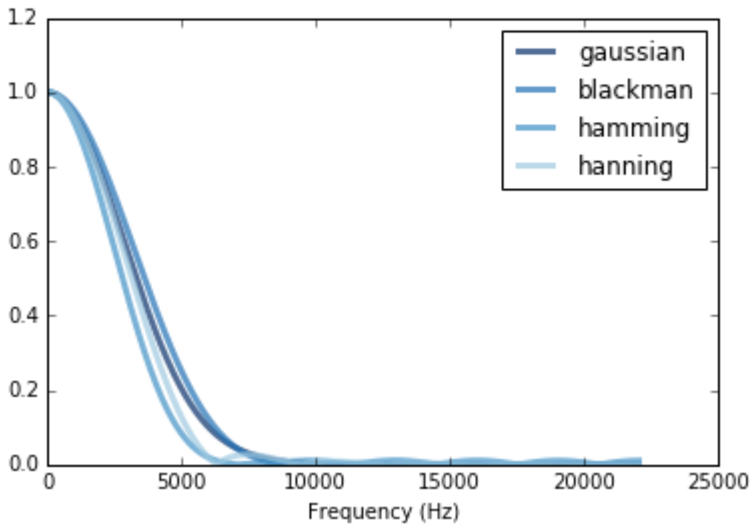


They are pretty similar. Let's see what their DFTs look like:

```
In [12]: def plot_window_dfts(windows, names):
             thinkplot.preplot(5)

             for window, name in zip(windows, names):
                 padded = thinkdsp.zero_pad(window, len(wave))
                 dft_window = np.fft.rfft(padded)
                 thinkplot.plot(abs(dft_window), label=name)
```

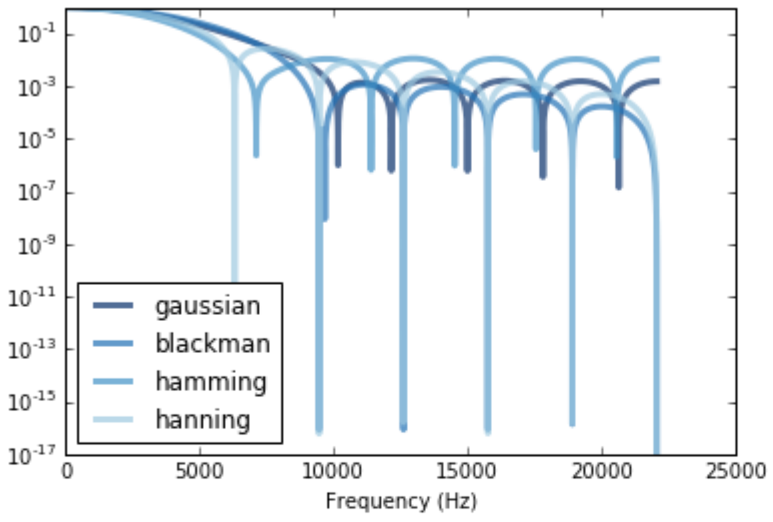
Also pretty similar, but it looks like Hamming drops off the fastest, Blackman the slowest, and Hanning has the most visible sidelobes.

In [13]:
```
plot_window_dfts(windows, names)
thinkplot.config(xlabel='Frequency (Hz)', loc='upper right
')
```



On a log scale we can see that the Hamming and Hanning drop off faster than the other two at first. And the Hamming and Gaussian windows seem to have the most persistent sidelobes. The Hanning window seems to have the best combination of fast drop off and minimal sidelobes.

In [14]:
```
plot_window_dfts(windows, names)
thinkplot.config(xlabel='Frequency (Hz)', yscale='log',
                 loc='lower left')
```



In [ ]:
```
```

ThinkDSP

This notebook contains solutions to exercises in Chapter 9: Differentiation and Integration

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot
import thinkstats2

import numpy as np
import pandas as pd
import scipy.signal

import warnings
warnings.filterwarnings('ignore')

PI2 = 2 * np.pi
GRAY = '0.7'

np.set_printoptions(precision=3, suppress=True)
%matplotlib inline
```

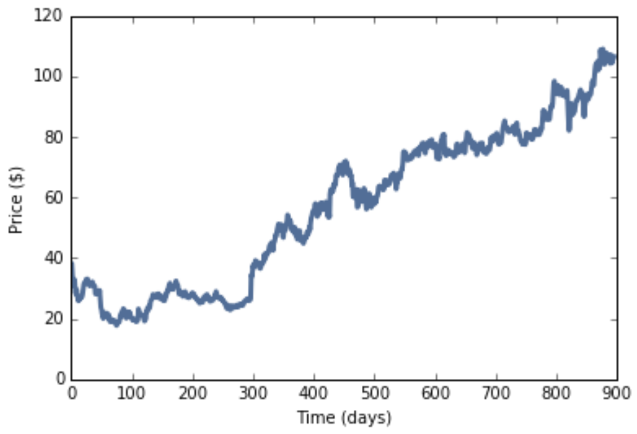
Exercise: In the section on cumulative sum, I mentioned that some of the examples don't work with non-periodic signals. Try replacing the sawtooth wave, which is periodic, with the Facebook data, which is not, and see what goes wrong.

Solution: I'll start by loading the Facebook data again.

In [2]:
```
names = ['date', 'open', 'high', 'low', 'close', 'volume']
df = pd.read_csv('fb.csv', header=0, names=names, parse_dates=[0])
ys = df.close.values[::-1]
```

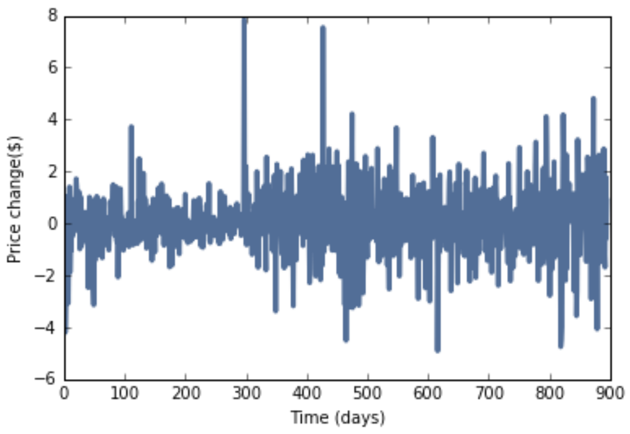
And making a Wave

In [3]:
```
close = thinkdsp.Wave(ys, framerate=1)
close.plot()
thinkplot.config(xlabel='Time (days)', ylabel='Price ($)')
```



I'll compute the daily changes using `Wave.diff` :

In [4]:
```
change = close.diff()
change.plot()
thinkplot.config(xlabel='Time (days)', ylabel='Price change($)')
```

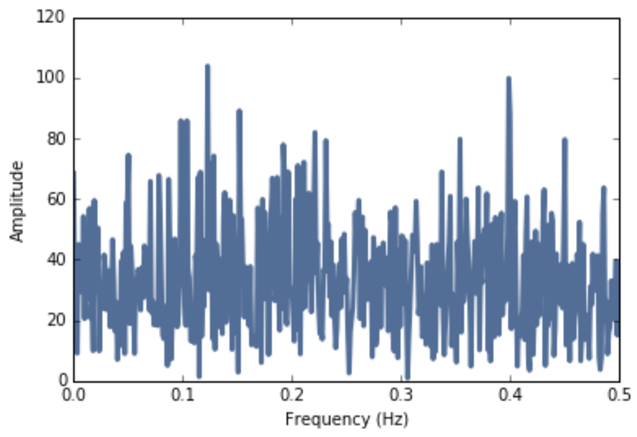


Now I'll run the `cumsum` example using `change` as the input wave:

In [5]:
```
in_wave = change
```

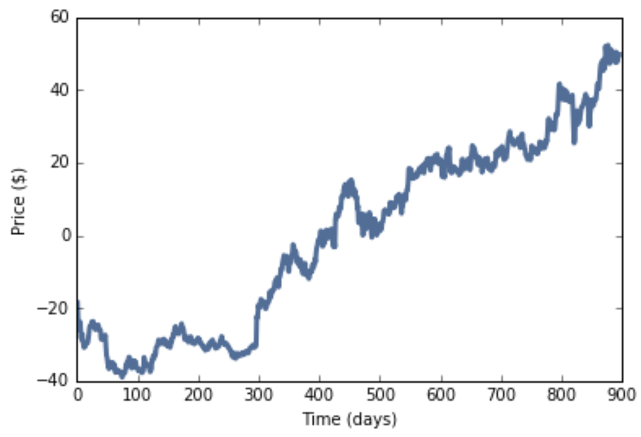
Here's the spectrum before the cumulative sum:

In [6]:
```
in_spectrum = in_wave.make_spectrum()
in_spectrum.plot()
thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude')
```



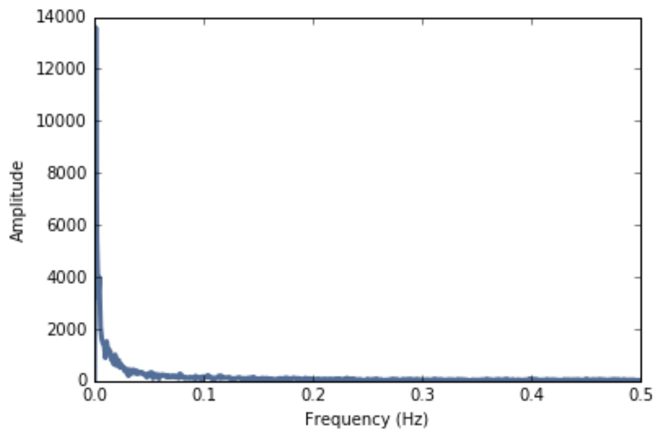
The output wave is the cumulative sum of the input

In [7]:
```
out_wave = in_wave.cumsum()
out_wave.unbias()
out_wave.plot()
thinkplot.config(xlabel='Time (days)', ylabel='Price ($)')
```



And here's its spectrum

In [8]:
```
out_spectrum = out_wave.make_spectrum()
out_spectrum.plot()
thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude')
```



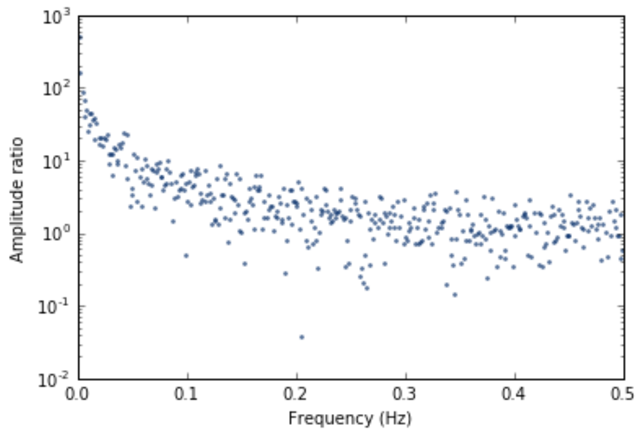
Now we compute the ratio of the output to the input:

In [9]:
```
sum(in_spectrum.amps < 10), len(in_spectrum)
```

Out[9]:
```
(37, 448)
```

In between the harmonics, the input componenents are small, so I set those ratios to NaN.

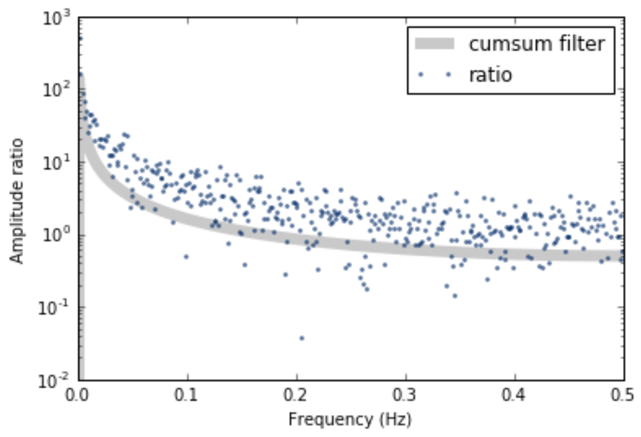
In [10]:
```
ratio_spectrum = out_spectrum.ratio(in_spectrum, thresh=10)
ratio_spectrum.hs[0] = 0
ratio_spectrum.plot(style='.', markersize=4)

thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude ratio',
                 yscale='log')
```



Instead of falling in a neat line, the ratios are pretty noisy.

We can compare them with the cumsum filter:

In [11]:
```
diff_window = np.array([1.0, -1.0])
padded = thinkdsp.zero_pad(diff_window, len(in_wave))
diff_wave = thinkdsp.Wave(padded, framerate=in_wave.framerate)
diff_filter = diff_wave.make_spectrum()

cumsum_filter = diff_filter.copy()
cumsum_filter.hs = 1 / cumsum_filter.hs
cumsum_filter.hs[0] = 0
cumsum_filter.plot(label='cumsum filter', color=GRAY, linewidth=7)

ratio_spectrum.plot(label='ratio', style='.', markersize=4)
thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude ratio',
                 yscale='log', legend=True)
```



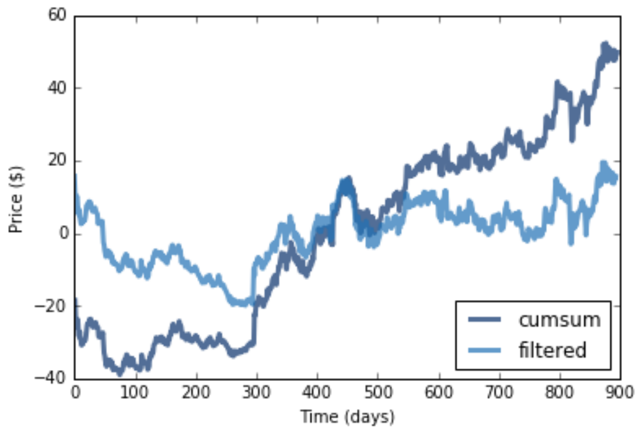
The ratios follow the general shape of the filter, but they are not in accord.

Now we can compute the output wave using the convolution theorem, and compare the results:

In [12]:
```
len(in_spectrum), len(cumsum_filter)
```

Out[12]:
```
(448, 448)
```

In [13]:
```
thinkplot.preplot(2)

out_wave.plot(label='cumsum')

in_spectrum = in_wave.make_spectrum()
out_wave2 = (in_spectrum * cumsum_filter).make_wave()
out_wave2.plot(label='filtered')

thinkplot.config(legend=True, loc='lower right')
thinkplot.config(xlabel='Time (days)', ylabel='Price ($)')
```

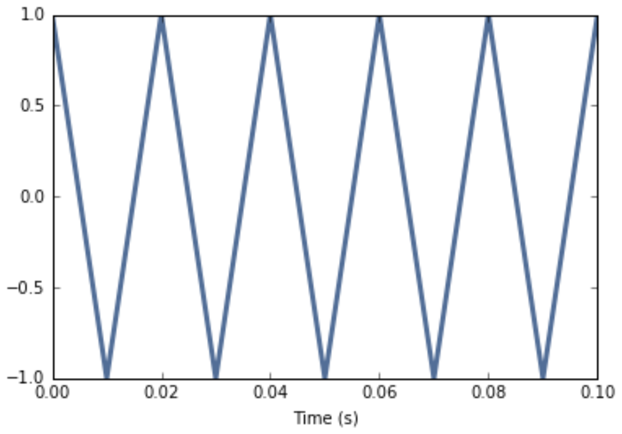


They are clearly different.

Exercise: The goal of this exercise is to explore the effect of `diff` and `differentiate` on a signal. Create a triangle wave and plot it. Apply the `diff` operator and plot the result. Compute the spectrum of the triangle wave, apply `differentiate`, and plot the result. Convert the spectrum back to a wave and plot it. Are there differences between the effect of `diff` and `differentiate` for this wave?

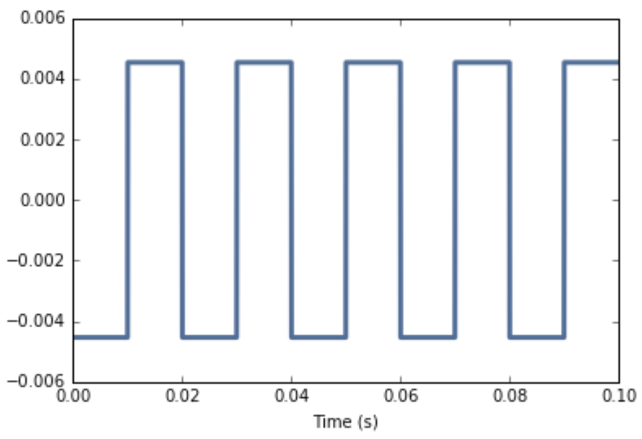
Solution: Here's the triangle wave.

In [14]:
```
in_wave = thinkdsp.TriangleSignal(freq=50).make_wave(duration=0.1, framerat
e=44100)
in_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



The diff of a triangle wave is a square wave, which explains why the harmonics in a square wave drop off like $1/f$, compared to the triangle wave, which drops off like $1/f^2$.

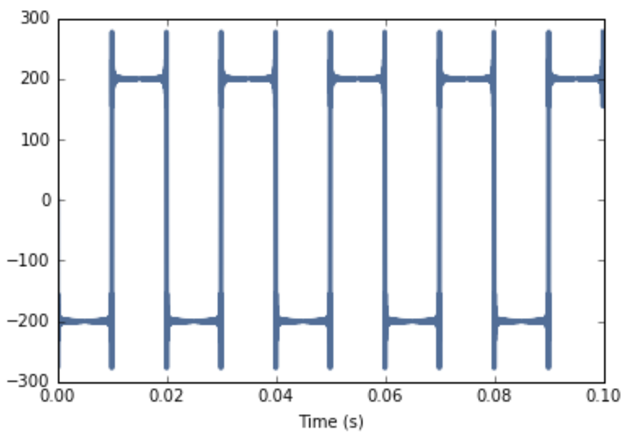
In [15]:
```
out_wave = in_wave.diff()
out_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



When we take the spectral derivative, we get "ringing" around the discontinuities: https://en.wikipedia.org/wiki/Ringing_(signal (https://en.wikipedia.org/wiki/Ringing_(signal))

Mathematically speaking, the problem is that the derivative of the triangle wave is undefined at the points of the triangle.

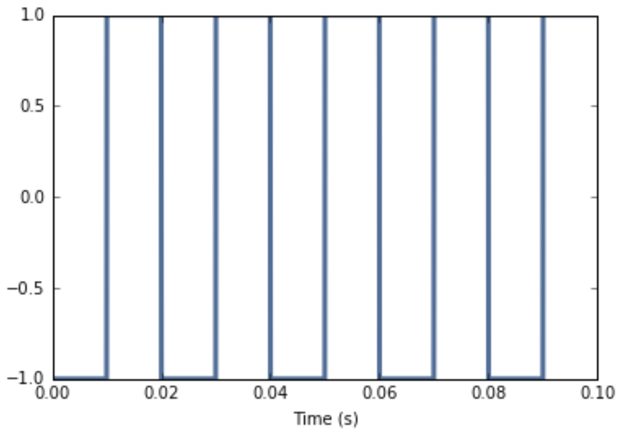
In [16]:
```
out_wave2 = in_wave.make_spectrum().differentiate().make_wave()
out_wave2.plot()
thinkplot.config(xlabel='Time (s)')
```



Exercise: The goal of this exercise is to explore the effect of `cumsum` and `integrate` on a signal. Create a square wave and plot it. Apply the `cumsum` operator and plot the result. Compute the spectrum of the square wave, apply `integrate`, and plot the result. Convert the spectrum back to a wave and plot it. Are there differences between the effect of `cumsum` and `integrate` for this wave?

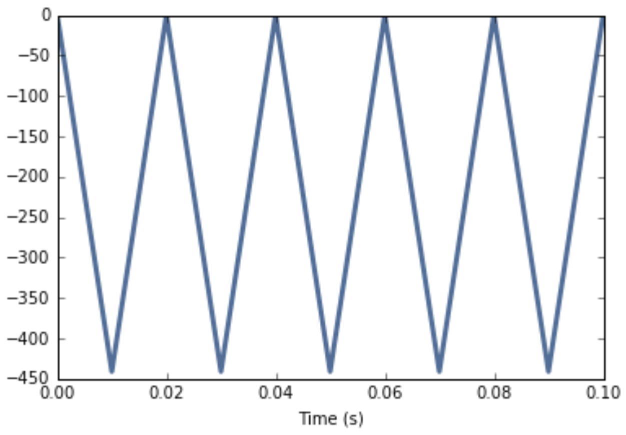
Solution: Here's the square wave.

In [17]:
```
in_wave = thinkdsp.SquareSignal(freq=50).make_wave(duration=0.1, framerate=4
4100)
in_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



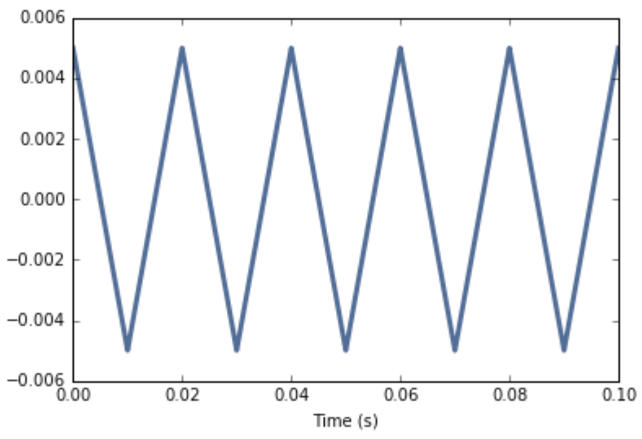
The cumulative sum of a square wave is a triangle wave. After the previous exercise, that should come as no surprise.

In [18]:
```
out_wave = in_wave.cumsum()
out_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



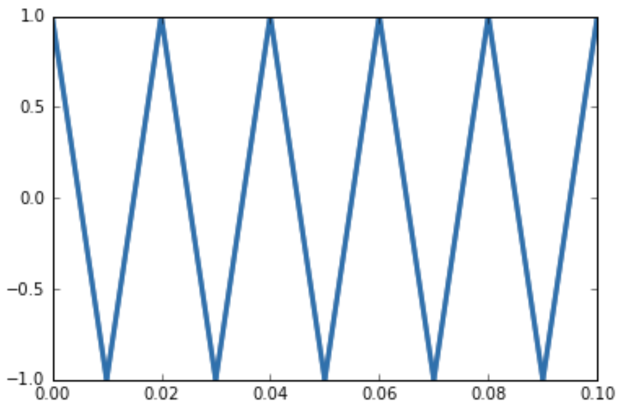
The spectral integral is also a triangle wave, although the amplitude is very different.

```
In [19]: spectrum = in_wave.make_spectrum().integrate()
         spectrum.hs[0] = 0
         out_wave2 = spectrum.make_wave()
         out_wave2.plot()
         thinkplot.config(xlabel='Time (s)')
```



If we unbias and normalize the two waves, they are visually similar.

```
In [20]: out_wave.unbias()
         out_wave.normalize()
         out_wave2.normalize()
         out_wave.plot()
         out_wave2.plot()
```



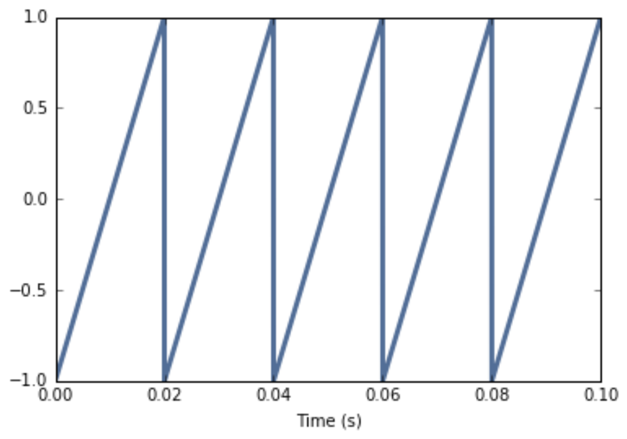
And they are numerically similar, but with only about 3 digits of precision.

```
In [21]: max(abs(out_wave.ys - out_wave2.ys))

Out[21]: 0.0045351473922902175
```

Exercise: The goal of this exercise is the explore the effect of integrating twice. Create a sawtooth wave, compute its spectrum, then apply `integrate` twice. Plot the resulting wave and its spectrum. What is the mathematical form of the wave? Why does it resemble a sinusoid?

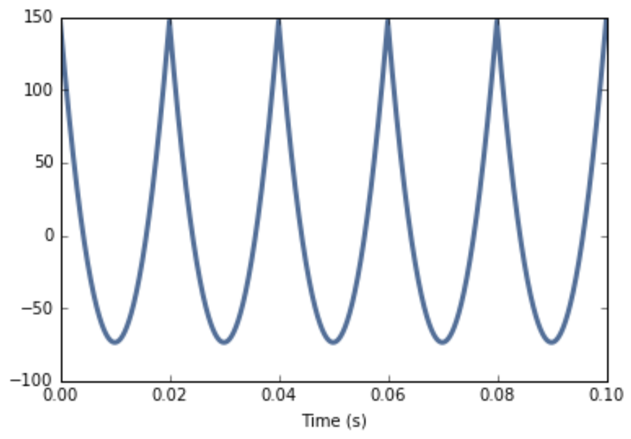
Here's the sawtooth.

In [22]:
```
in_wave = thinkdsp.SawtoothSignal(freq=50).make_wave(duration=0.1, framerat
e=44100)
in_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



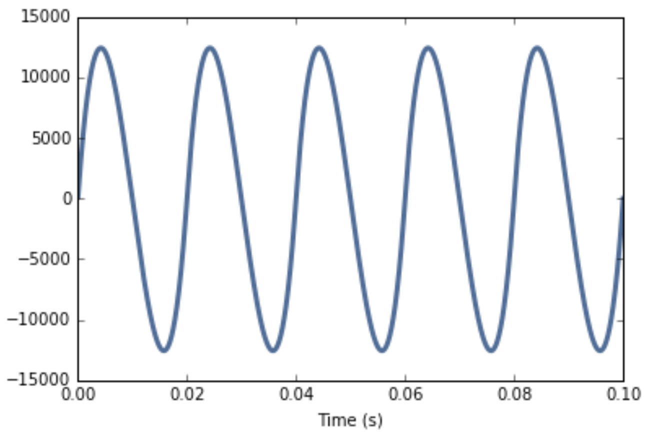
The first cumulative sum of a sawtooth is a parabola:

In [23]:
```
out_wave = in_wave.cumsum()
out_wave.unbias()
out_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



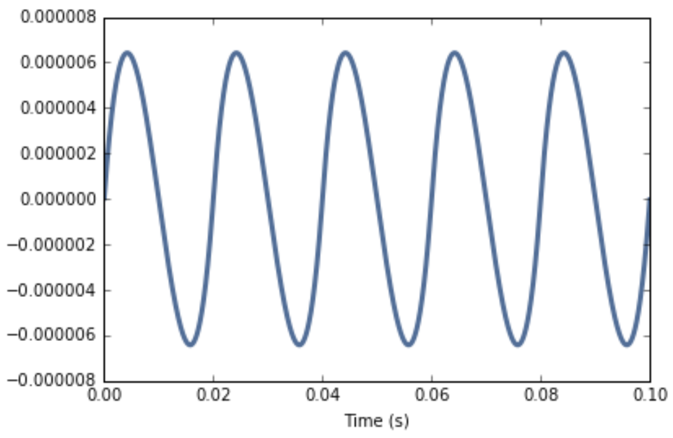
The second cumulative sum is a cubic curve:

In [24]:
```
out_wave = out_wave.cumsum()
out_wave.plot()
thinkplot.config(xlabel='Time (s)')
```



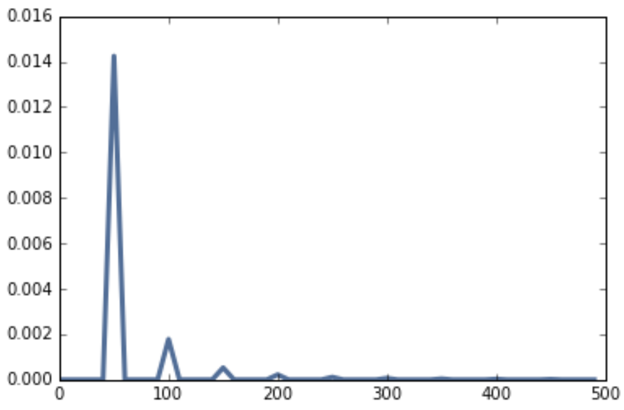
Integrating twice also yields a cubic curve.

In [25]:
```
spectrum = in_wave.make_spectrum().integrate().integrate()
spectrum.hs[0] = 0
out_wave2 = spectrum.make_wave()
out_wave2.plot()
thinkplot.config(xlabel='Time (s)')
```



At this point, the result looks more and more like a sinusoid. The reason is that integration acts like a low pass filter. At this point we have filtered out almost everything except the fundamental, as shown in the spectrum below:

In [26]:
```
out_wave2.make_spectrum().plot(high=500)
```

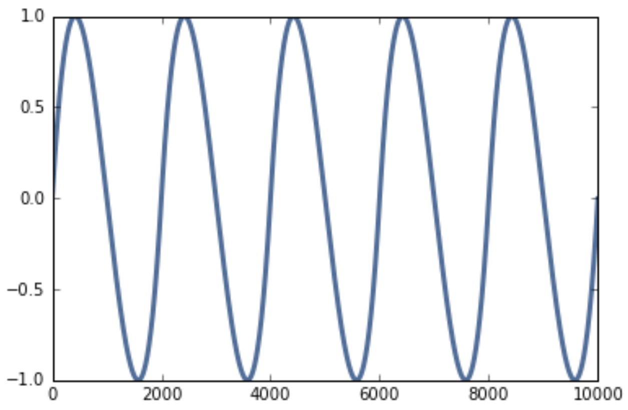


Exercise: The goal of this exercise is to explore the effect of the 2nd difference and 2nd derivative. Create a `CubicSignal`, which is defined in `thinkdsp`. Compute the second difference by applying `diff` twice. What does the result look like. Compute the second derivative by applying `differentiate` twice. Does the result look the same?

Plot the filters that corresponds to the 2nd difference and the 2nd derivative and compare them. Hint: In order to get the filters on the same scale, use a wave with framerate 1.

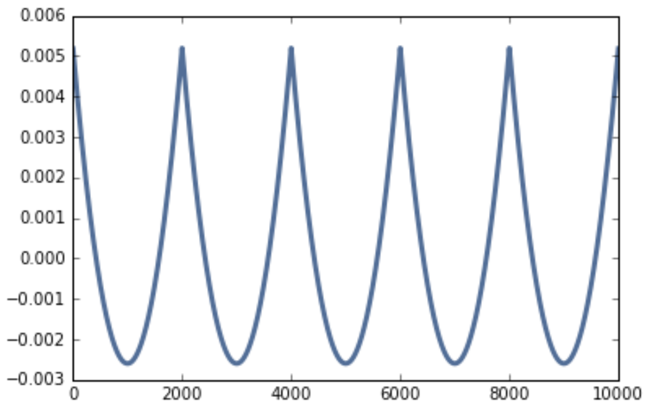
Solution: Here's the cubic signal

In [27]:
```
in_wave = thinkdsp.CubicSignal(freq=0.0005).make_wave(duration=10000, framer
ate=1)
in_wave.plot()
```

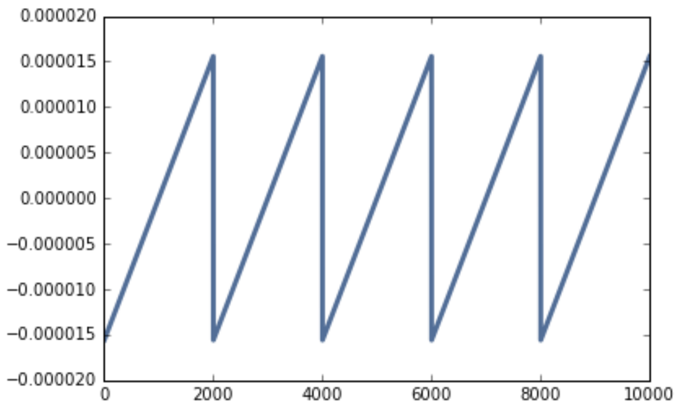


The first difference is a parabola and the second difference is a sawtooth wave (no surprises so far):

In [28]:
```
out_wave = in_wave.diff()
out_wave.plot()
```

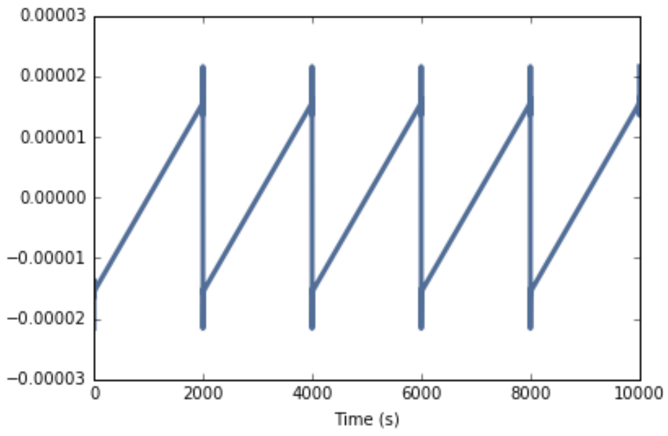


In [29]:
```
out_wave = out_wave.diff()
out_wave.plot()
```



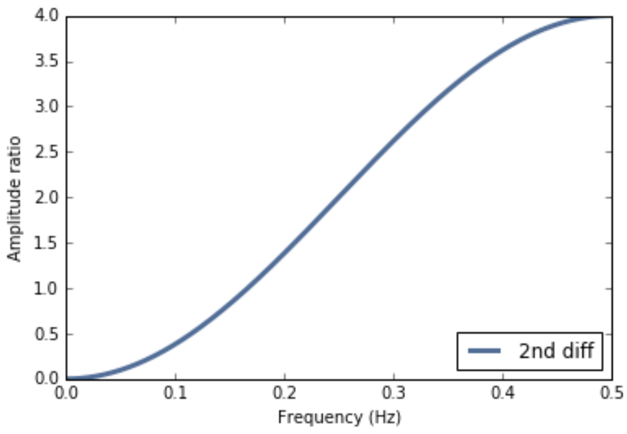
When we differentiate twice, we get a sawtooth with some ringing. Again, the problem is that the deriviative of the parabolic signal is undefined at the points.

In [30]:
```
spectrum = in_wave.make_spectrum().differentiate().differentiate()
out_wave2 = spectrum.make_wave()
out_wave2.plot()
thinkplot.config(xlabel='Time (s)')
```



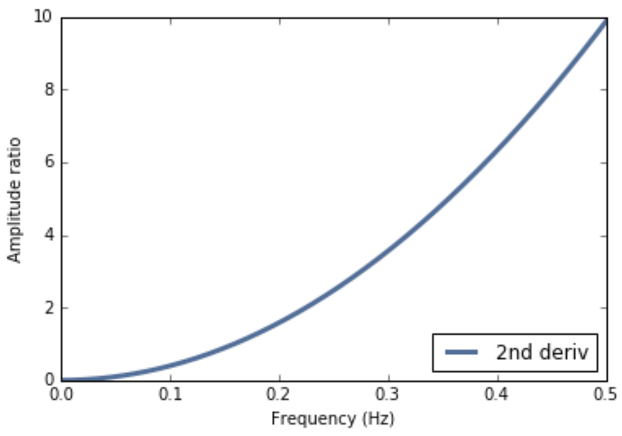
The window of the second difference is -1, 2, -1. By computing the DFT of the window, we can find the corresponding filter.

In [31]:
```
diff_window = np.array([-1.0, 2.0, -1.0])
padded = thinkdsp.zero_pad(diff_window, len(in_wave))
diff_wave = thinkdsp.Wave(padded, framerate=in_wave.framerate)
diff_filter = diff_wave.make_spectrum()
diff_filter.plot(label='2nd diff')

thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude ratio',
                 legend=True, loc='lower right')
```



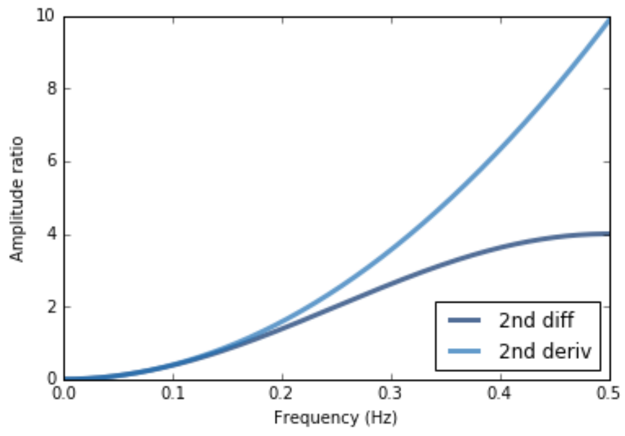
And for the second derivative, we can find the corresponding filter by computing the filter of the first derivative and squaring it.

In [32]:
```
deriv_filter = in_wave.make_spectrum()
deriv_filter.hs = (PI2 * 1j * deriv_filter.fs)**2
deriv_filter.plot(label='2nd deriv')

thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude ratio',
                 legend=True, loc='lower right')
```



Here's what the two filters look like on the same scale:

In [33]:
```
diff_filter.plot(label='2nd diff')
deriv_filter.plot(label='2nd deriv')

thinkplot.config(xlabel='Frequency (Hz)',
                 ylabel='Amplitude ratio',
                 legend=True, loc='lower right')
```



Both are high pass filters that amplify the highest frequency components. The 2nd derivative is parabolic, so it amplifies the highest frequencies the most. The 2nd difference is a good approximation of the 2nd derivative only at the lowest frequencies, then the deviates substantially.

In [ ]:
```
```

ThinkDSP

This notebook contains solutions to exercises in Chapter 10: Signals and Systems

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot

import numpy as np
import pandas as pd

import warnings
warnings.filterwarnings('ignore')

PI2 = 2 * np.pi

np.set_printoptions(precision=3, suppress=True)
%matplotlib inline
```

Exercise: In this chapter I describe convolution as the sum of shifted, scaled copies of a signal. Strictly speaking, this operation is *linear* convolution, which does not assume that the signal is periodic.

But when we multiply the DFT of the signal by the transfer function, that operation corresponds to *circular* convolution, which assumes that the signal is periodic. As a result, you might notice that the output contains an extra note at the beginning, which wraps around from the end.

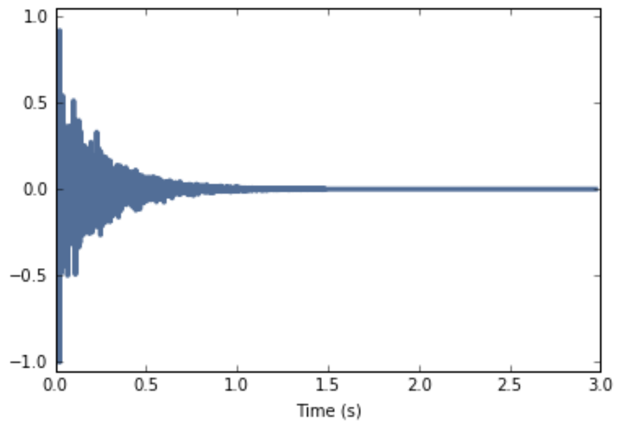
Fortunately, there is a standard solution to this problem. If you add enough zeros to the end of the signal before computing the DFT, you can avoid wrap-around and compute a linear convolution.

Modify the example in `chap10soln.ipynb` and confirm that zero-padding eliminates the extra note at the beginning of the output.

Solution: I'll truncate both signals to 2^{16} elements, then zero-pad them to 2^{17}. Using powers of two makes the FFT algorithm most efficient.

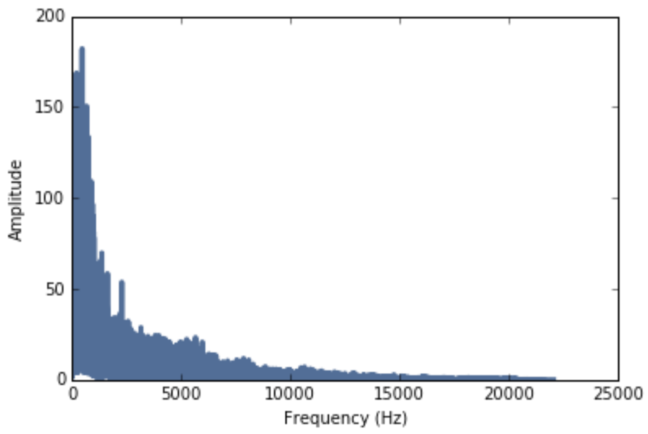
Here's the impulse response:

In [2]:
```
response = thinkdsp.read_wave('180960__kleeb__gunshot.wav')

start = 0.12
response = response.segment(start=start)
response.shift(-start)

response.truncate(2**16)
response.zero_pad(2**17)

response.normalize()
response.plot()
thinkplot.config(xlabel='Time (s)', ylim=[-1.05, 1.05])
```



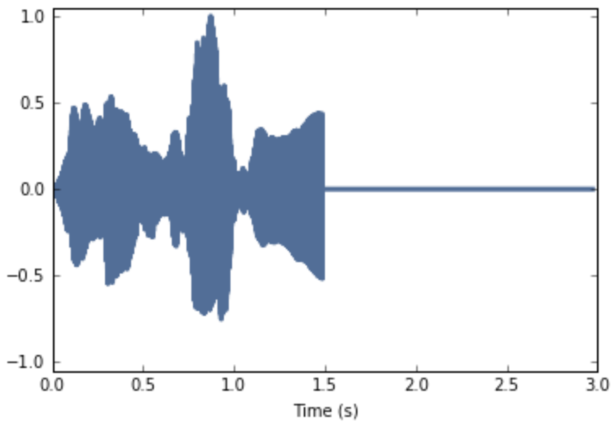
And its spectrum:

In [3]:
```
transfer = response.make_spectrum()
transfer.plot()
thinkplot.config(xlabel='Frequency (Hz)', ylabel='Amplitude')
```



Here's the signal:

In [4]:
```
violin = thinkdsp.read_wave('92002__jcveliz__violin-origional.wav')

start = 0.11
violin = violin.segment(start=start)
violin.shift(-start)

violin.truncate(2**16)
violin.zero_pad(2**17)

violin.normalize()
violin.plot()
thinkplot.config(xlabel='Time (s)', ylim=[-1.05, 1.05])
```



And its spectrum:

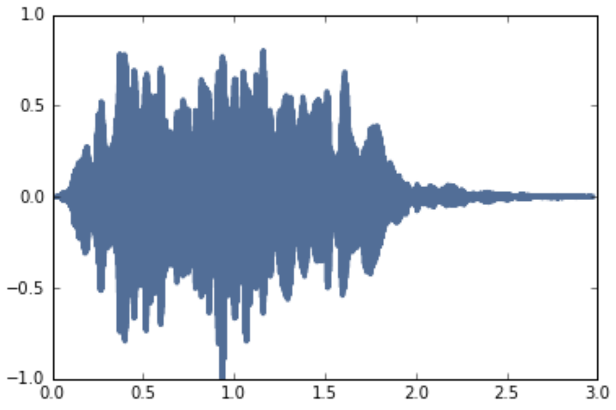
In [5]:
```
spectrum = violin.make_spectrum()
```

Now we can multiply the DFT of the signal by the transfer function, and convert back to a wave:

In [6]:
```
output = (spectrum * transfer).make_wave()
output.normalize()
```

The result doesn't look like it wraps around:

In [7]:
```
output.plot()
```



And we don't hear the extra note at the beginning:

```
In [8]: output.make_audio()
```

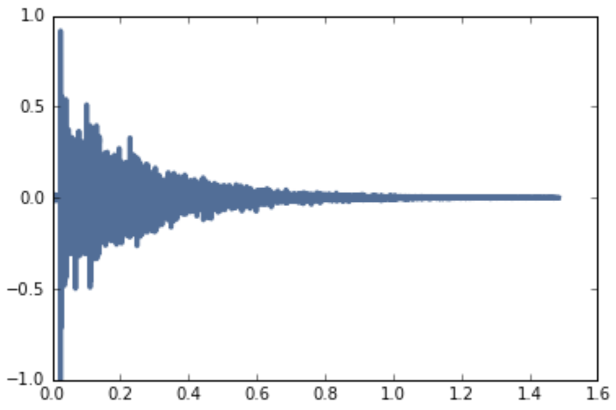
Out[8]:

0:00:00 / 13:31:36

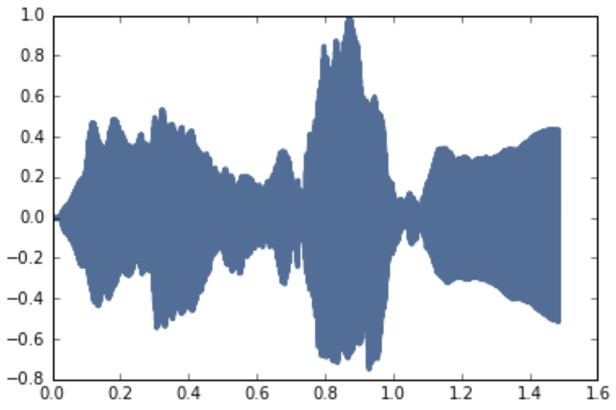
We should get the same results from np.convolve and scipy.signal.fftconvolve.

First I'll get rid of the zero padding:

```
In [9]: response.truncate(2**16)
        response.plot()
```



```
In [10]: violin.truncate(2**16)
         violin.plot()
```

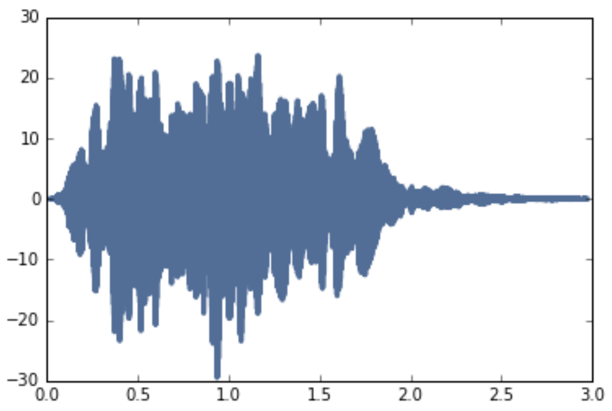


Now we can compare to `np.convolve` :

```
In [11]: output2 = violin.convolve(response)
```

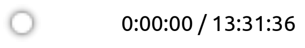
The results are similar:

```
In [12]: output2.plot()
```



And sound the same:

```
In [13]: output2.make_audio()
Out[13]:
```


0:00:00 / 13:31:36

But the results are not exactly the same length:

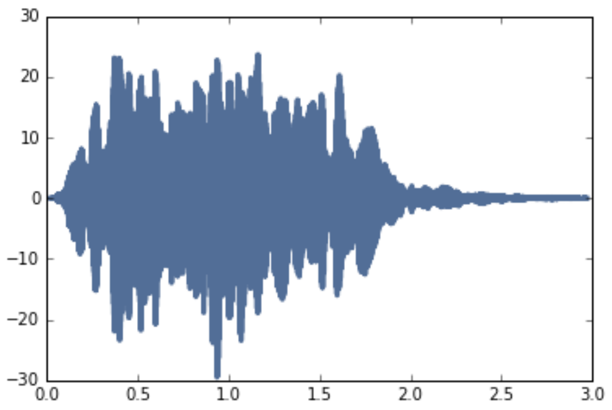
```
In [14]: len(output), len(output2)
Out[14]: (131072, 131071)
```

`scipy.signal.fftconvolve` does the same thing, but as the name suggests, it uses the FFT, so it is substantially faster:

```
In [15]: import scipy.signal
         ys = scipy.signal.fftconvolve(violin.ys, response.ys)
         output3 = thinkdsp.Wave(ys, framerate=violin.framerate)
```

The results look the same.

```
In [16]: output3.plot()
```



And sound the same:

```
In [17]: output3.make_audio()
```

```
Out[17]:
0:00:00 / 13:31:36
```

And within floating point error, they are the same:

```
In [18]: output2.max_diff(output3)
```

```
Out[18]: 5.8841820305133297e-14
```

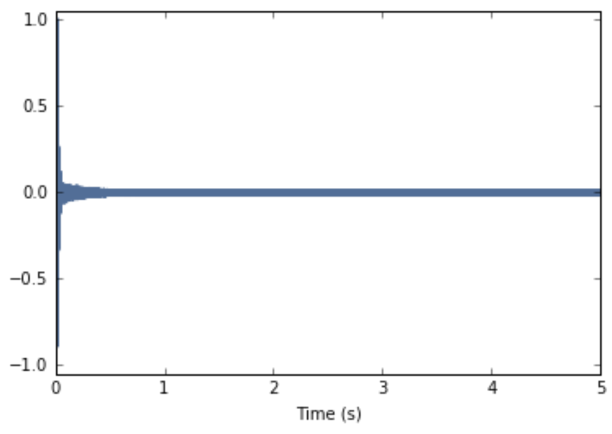
Exercise: The Open AIR library provides a ``centralized... on-line resource for anyone interested in auralization and acoustical impulse response data'' (http://www.openairlib.net (http://www.openairlib.net)). Browse their collection of impulse response data and download one that sounds interesting. Find a short recording that has the same sample rate as the impulse response you downloaded.

Simulate the sound of your recording in the space where the impulse response was measured, computed two way: by convolving the recording with the impulse response and by computing the filter that corresponds to the impulse response and multiplying by the DFT of the recording.

Solution: I downloaded the impulse response of the Lady Chapel at St Albans Cathedral http://www.openairlib.net/auralizationdb/content/lady-chapel-st-albans-cathedral (http://www.openairlib.net/auralizationdb/content/lady-chapel-st-albans-cathedral)

Thanks to Audiolab, University of York: Marcin Gorzel, Gavin Kearney, Aglaia Foteinou, Sorrel Hoare, Simon Shelley.

```
In [19]: response = thinkdsp.read_wave('stalbans_a_mono.wav')

         start = 0
         duration = 5
         response = response.segment(duration=duration)
         response.shift(-start)

         response.normalize()
         response.plot()
         thinkplot.config(xlabel='Time (s)', ylim=[-1.05, 1.05])
```



Here's what it sounds like:

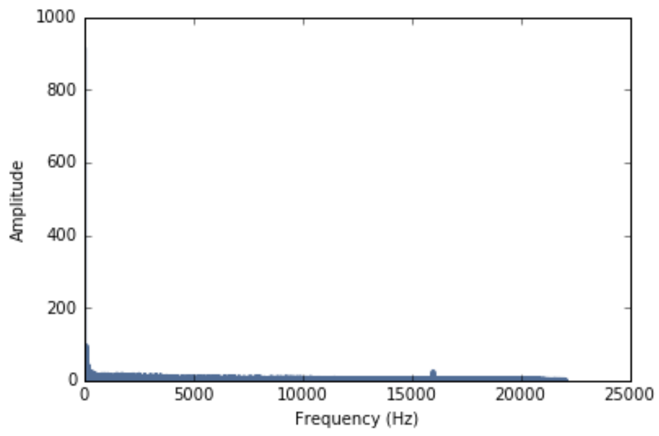
In [20]:
```
response.make_audio()
```

Out[20]:

```
0:00:00 / 13:31:36
```

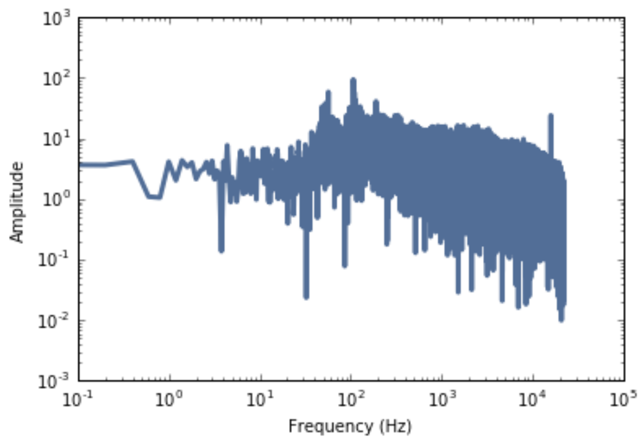
The DFT of the impulse response is the transfer function:

In [21]:
```
transfer = response.make_spectrum()
transfer.plot()
thinkplot.config(xlabel='Frequency (Hz)', ylabel='Amplitude')
```



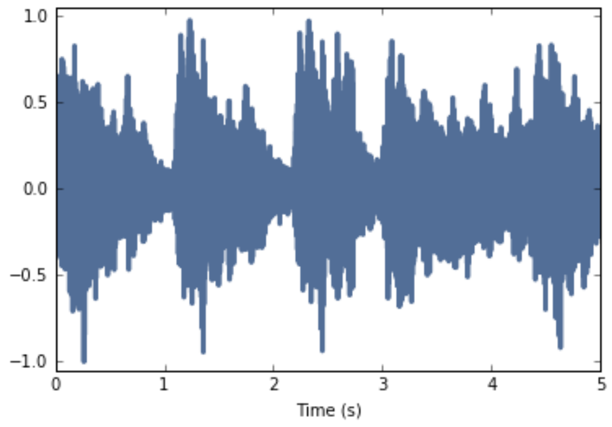
Here's the transfer function on a log-log scale:

In [22]:
```
transfer.plot()
thinkplot.config(xlabel='Frequency (Hz)', ylabel='Amplitude',
                 xscale='log', yscale='log')
```



Now we can simulate what a recording would sound like if it were played in the same room and recorded in the same way. Here's the violin recording we have used before:

In [23]:
```
wave = thinkdsp.read_wave('170255__dublie__trumpet.wav')

start = 0.0
wave = wave.segment(start=start)
wave.shift(-start)

wave.truncate(len(response))
wave.normalize()
wave.plot()
thinkplot.config(xlabel='Time (s)', ylim=[-1.05, 1.05])
```



Here's what it sounds like before transformation:

In [24]:
```
wave.make_audio()
```

Out[24]:
```
0:00:00 / 13:31:36
```

Now we compute the DFT of the violin recording.

In [25]:
```
spectrum = wave.make_spectrum()
```

I trimmed the violin recording to the same length as the impulse response:

In [26]:
```
len(spectrum.hs), len(transfer.hs)
```

Out[26]:
```
(110251, 110251)
```

In [27]:
```
spectrum.fs
```

Out[27]:
```
array([     0. ,      0.2,      0.4, ...,  22049.6,  22049.8,  22050. ])
```

In [28]:
```
transfer.fs
```

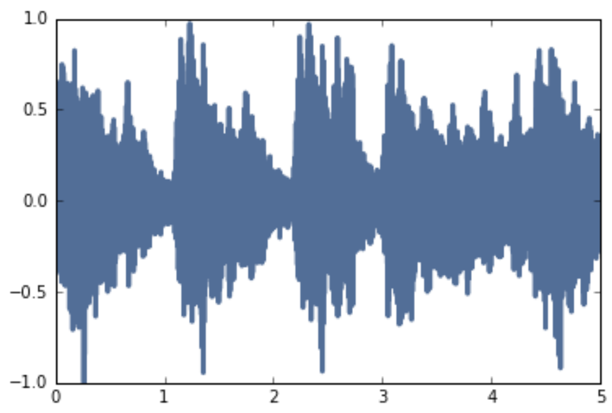
Out[28]:
```
array([     0. ,      0.2,      0.4, ...,  22049.6,  22049.8,  22050. ])
```

We we can multiply in the frequency domain and the transform back to the time domain.

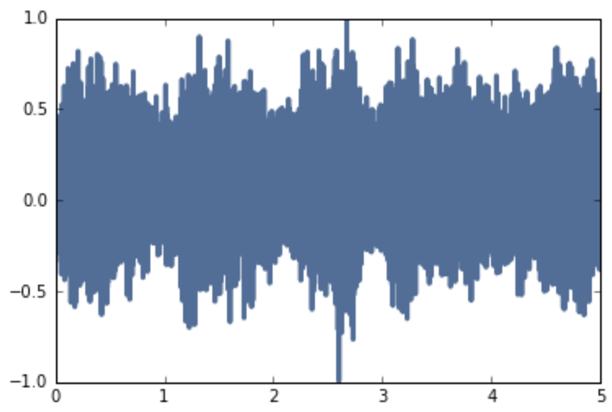
In [29]:
```
output = (spectrum * transfer).make_wave()
output.normalize()
```

Here's a comparison of the original and transformed recordings:

In [30]:
```
wave.plot()
```



In [31]:
```
output.plot()
```



And here's what it sounds like:

In [32]:
```
output.make_audio()
```

Out[32]:

0:00:00 / 13:31:36

Now that we recognize this operation as convolution, we can compute it using the convolve method:

In [33]:
```
convolved2 = wave.convolve(response)
convolved2.normalize()
convolved2.make_audio()
```

Out[33]:

0:00:00 / 13:31:36

In [ ]:

ThinkDSP

This notebook contains solutions to exercises in Chapter 11: Modulation and sampling

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot

import numpy as np

import warnings
warnings.filterwarnings('ignore')

PI2 = 2 * np.pi

np.set_printoptions(precision=3, suppress=True)
%matplotlib inline
```

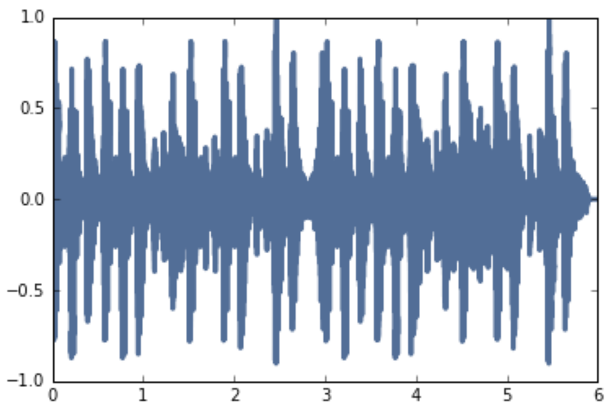
Exercise: As we have seen, if you sample a signal at too low a framerate, frequencies above the folding frequency get aliased. Once that happens, it is no longer possible to filter out these components, because they are indistinguishable from lower frequencies.

It is a good idea to filter out these frequencies *before* sampling; a low-pass filter used for this purpose is called an ``anti-aliasing filter''.

Returning to the drum solo example, apply a low-pass filter before sampling, then apply the low-pass filter again to remove the spectral copies introduced by sampling. The result should be identical to the filtered signal.

Solution: I'll load the drum solo again.

In [2]:
```
wave = thinkdsp.read_wave('263868__kevcio__amen-break-a-160-bpm.wav')
wave.normalize()
wave.plot()
```



This signal is sampled at 44100 Hz. Here's what it sounds like.

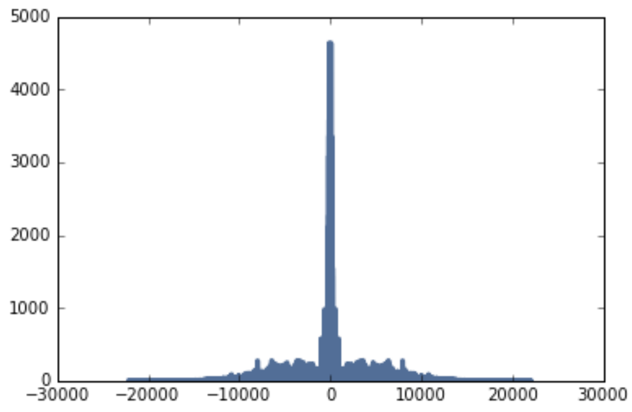
In [3]:
```
wave.make_audio()
```

Out[3]:

0:00:00 / 13:31:36

And here's the spectrum:

In [4]:
```
spectrum = wave.make_spectrum(full=True)
spectrum.plot()
```

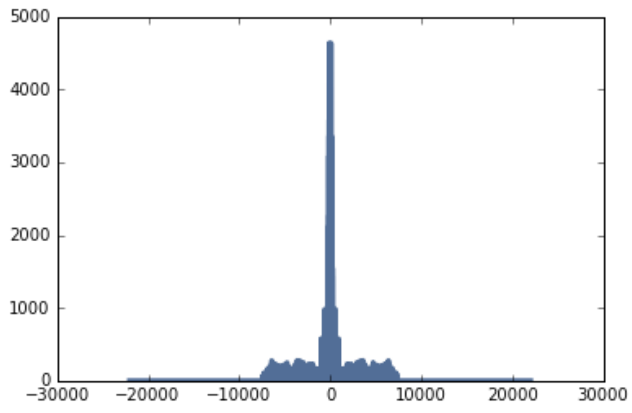


I'll reduce the sampling rate by a factor of 3 (but you can change this to try other values):

In [5]:
```
factor = 3
framerate = wave.framerate / factor
cutoff = framerate / 2 - 1
```

Before sampling we apply an anti-aliasing filter to remove frequencies above the new folding frequency, which is `framerate/2` :

In [6]:
```
spectrum.low_pass(cutoff)
spectrum.plot()
```



Here's what it sounds like after filtering (still pretty good).

```
In [7]: filtered = spectrum.make_wave()
        filtered.make_audio()
```

```
Out[7]:
        0:00:00 / 13:31:36
```

Here's the function that simulates the sampling process:

```
In [8]: def sample(wave, factor):
            """Simulates sampling of a wave.

            wave: Wave object
            factor: ratio of the new framerate to the original
            """
            ys = np.zeros(len(wave))
            ys[::factor] = wave.ys[::factor]
            return thinkdsp.Wave(ys, framerate=wave.framerate)
```

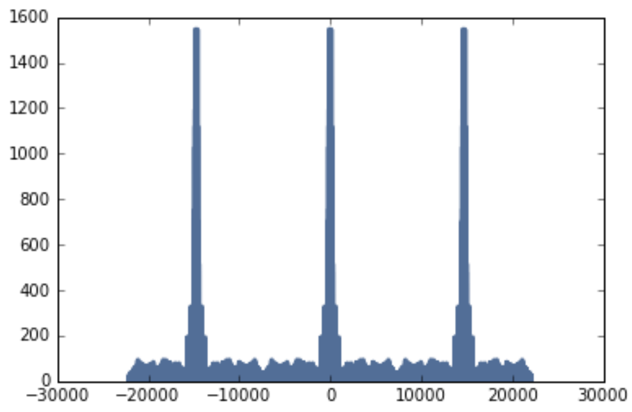
The result contains copies of the spectrum near 20 kHz; they are not very noticeable:

```
In [9]: sampled = sample(filtered, factor)
        sampled.make_audio()
```

```
Out[9]:
        0:00:00 / 13:31:36
```

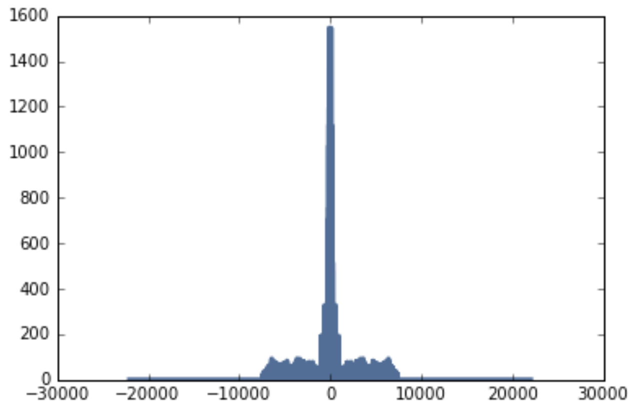
But they show up when we plot the spectrum:

```
In [10]: sampled_spectrum = sampled.make_spectrum(full=True)
         sampled_spectrum.plot()
```



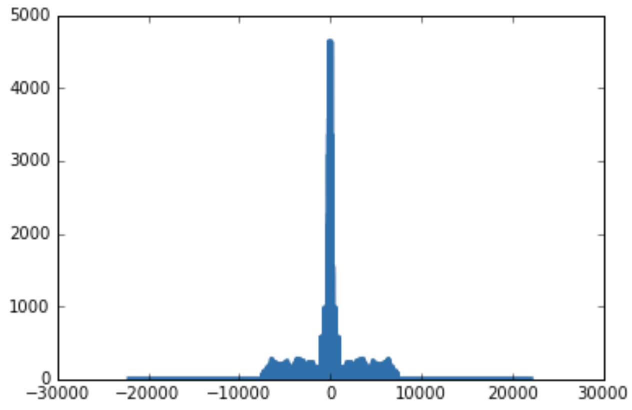
We can get rid of the spectral copies by applying the anti-aliasing filter again:

```
In [11]: sampled_spectrum.low_pass(cutoff)
         sampled_spectrum.plot()
```



We just lost half the energy in the spectrum, but we can scale the result to get it back:

```
In [12]: sampled_spectrum.scale(factor)
         spectrum.plot()
         sampled_spectrum.plot()
```



Now the difference between the spectrum before and after sampling should be small.

```
In [13]: spectrum.max_diff(sampled_spectrum)
Out[13]: 9.094947017729282e-12
```

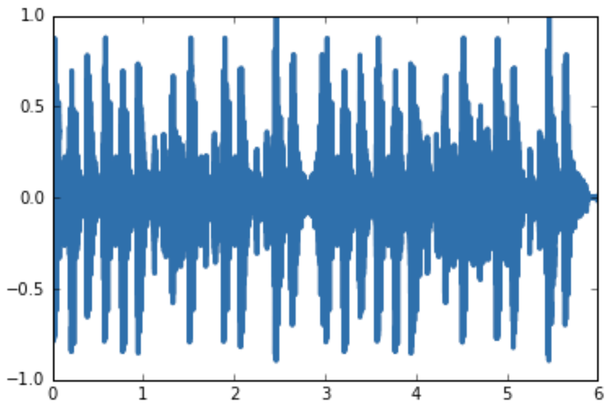
After filtering and scaling, we can convert back to a wave:

```
In [14]: interpolated = sampled_spectrum.make_wave()
         interpolated.make_audio()
Out[14]:
```

0:00:00 / 13:31:36

And the difference between the interpolated wave and the filtered wave should be small.

```
In [15]: filtered.plot()
         interpolated.plot()
```



Multiplying by `impulses` makes 4 shifted copies of the original spectrum. One of them wraps around from the negative end of the spectrum to the positive, which is why there are 5 peaks in the spectrum off the sampled wave.

```
In [16]: filtered.max_diff(interpolated)
Out[16]: 3.663750800998542e-15
```

```
In [ ]:
```

```
In [ ]:
```