Ivanov_lab4

March 5, 2020

Глава 4 (Иванов Илья, группа 3530901/70203)

1 Теория

В контексте обработки сигналов слово “шум” используется в двух значниях:

1) Нежелательный сигнал любого рода. Если два сигнала мешают друг другу, то один из них относительно другого рассматривается как шум.

2) Сигнал, содержащий компоненты с самыми разными частотами, но не имеющий гармонической структуры периодических сигналов.

Далее рассматривается именно второй вариант.

1.1 Некоррелированный шум

“Некоррелированный” означает, что значения сигнала независимые, то есть из одного значения нельзя получить сведения о других.

Если некоррелированный шум будет содержать случайные значения из равномерного распределения, то он будет называться “равномерным некорелированным шумом” и при прослушивании такой сигнал будет звучать как шум, слышимый при перестройке радио между станциями.

О шумовом сигнале или о его спектре необходимо знать как минимум три вещи:

1) Распределение. Распределение случайного сигнала - это набор возможных значений и их вероятностей.

2) Корреляция. Зависит ли одно значение сигнала от прочих или не зависит?

3) Связь между мощностью и частотой.

1.2 Интегральный спектр

Это функция от частоты f, и она показывает полную мощность(квадрат амплитуды) в спектре вплоть до f.

В случае UU-шума связь между частотой и мощностью видна чётче, если смотреть не на обычный спектр, а на интергральный. Интегральный спектр UU-шума - прямая линия, указывающая на то, что мощность на всех частотах в среднем неизменна.

1.3 Броуновский шум

UU-шум не коррелирован, то есть каждое его значение не зависит от остальных. Альтернатива - броуновский шум, в котором каждое значение есть сумма предыдущего значения и некоего случайного шага.

Способ создания броуновского шума: генерировать некоррелированные случайные шаги, а затем суммировать их.

Если спектр броуновского шума вывести в линейном масштабе, он будет выглядет необычно. Почти вся мощность будет сосредоточена на низких частотах; компоненты на высоких частотах будут почти не видны.

Если вывести мощность и частоту в двойном логарифмическом масштабе, можно заметить, что связь между ними размытая, но строго линейная.

Для броуновского шума уклон спектра мощности равен -2, так что можно записать соотношение: `log(P) = k - 2log(f)`, где P - мощность, f - частота, k- постоянная, которая сейчас не важна.

Потенцируем обе части:

`P = K/(f^2)`, где K - это e^k.

Получается, что мощность пропорциональна 1/(f^2), - характерное свойство броуновского шума.

Броуновский шум называют также красным шумом.

1.4 Розовый шум

В общем случае можно синтезировать шум с любой степенью b: `P = K/(f^b)` При b=0 мощность неизменна на всех частотах, и в результате будет белый шум(UU-шум). При b=2 будет красный шум.

Если b находится в пределах от 0 до 2, то результат будет между белым и красным шумом - он называется розовым шумом.

Есть несколько способов получения розового шума. Проще всего генерировать белый шум и пропускать его через фильтр НЧ со скатом, соответствующим нужной степени b.

Как и в случае с броуновским шумом, розовый шум блуждает вверх и вниз, причём заметна корреляция между соседними значениями, но выглядит он “более случайным”.

1.5 Гауссов шум

Когда речь идёт о белом шуме, не всегда имеется в виду UU-шум. Часто в таких случаях подразумевают некоррелированный гауссов шум(UG). В UG-шуме используются случайные значения с гауссовым распределением.

UG-шум многим похож на UU-шум. У спектра одинаковая средняя мощность на всех частотах, поэтому UG-шум также белый. Но спектр UG-шума - также UG-шум. Точнее, действительные и мнимые части спектра - некоррелированные гауссовы значения.

Спектр UU-шума в первом приближении - также UG-шум.

2 Упражнения

[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
```

2.1 Упражнение 4.1.

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

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

[2]:
```
wave = thinkdsp.read_wave('13793__soarer__north-sea.wav')
wave.make_audio()
```

[2]:
```
<IPython.lib.display.Audio object>
```

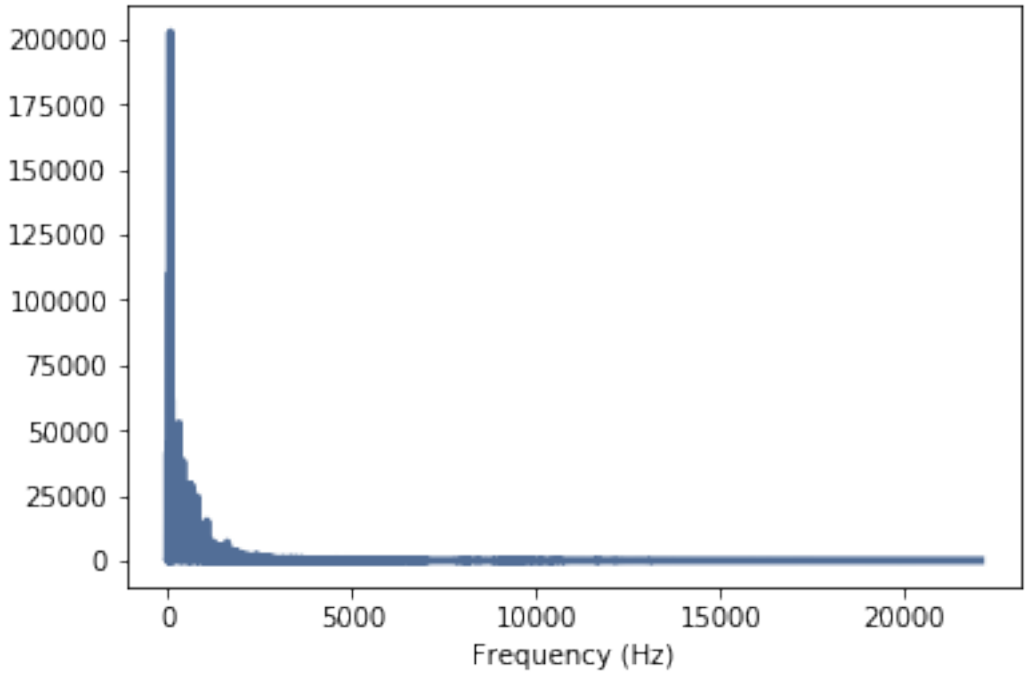
Я выбрал звук северного моря. Выделим небольшой отрезок из этой записи:

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

[3]:
```
<IPython.lib.display.Audio object>
```

Получим спектр сигнала:

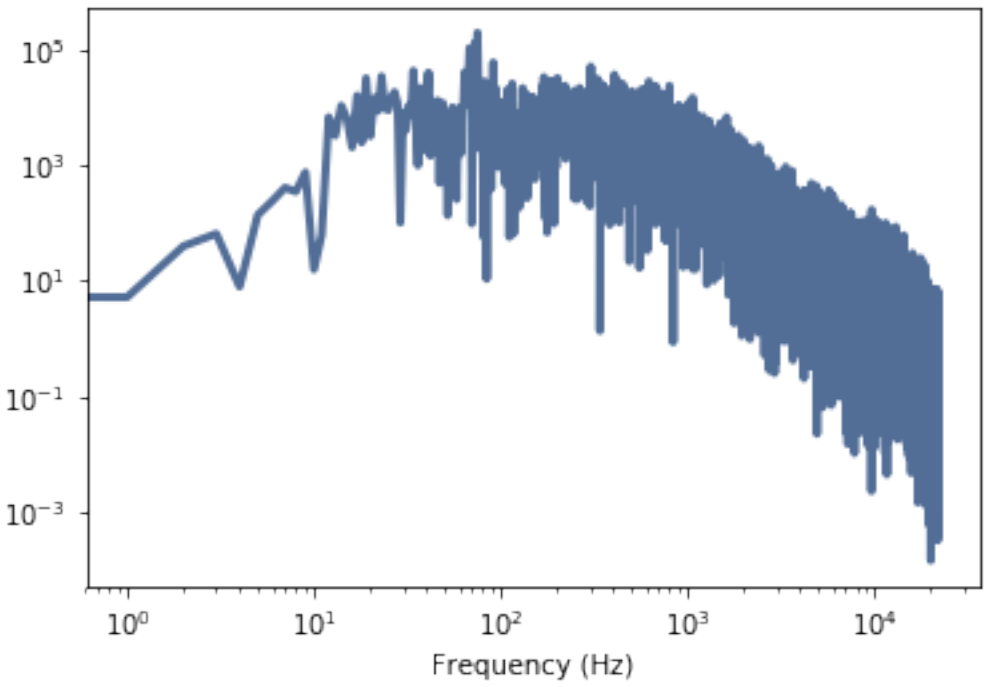
[4]:
```
spectrum = segment.make_spectrum()
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (Hz)')
```



Амплитуда уменьшается с увеличением частоты, следовательно, это может быть красный или розовый шум.

Выведем мощность и частоту в двойном логарифмическом масштабе.

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



После f = 10^(2,5) уменьшение мощности выглядит линейным.

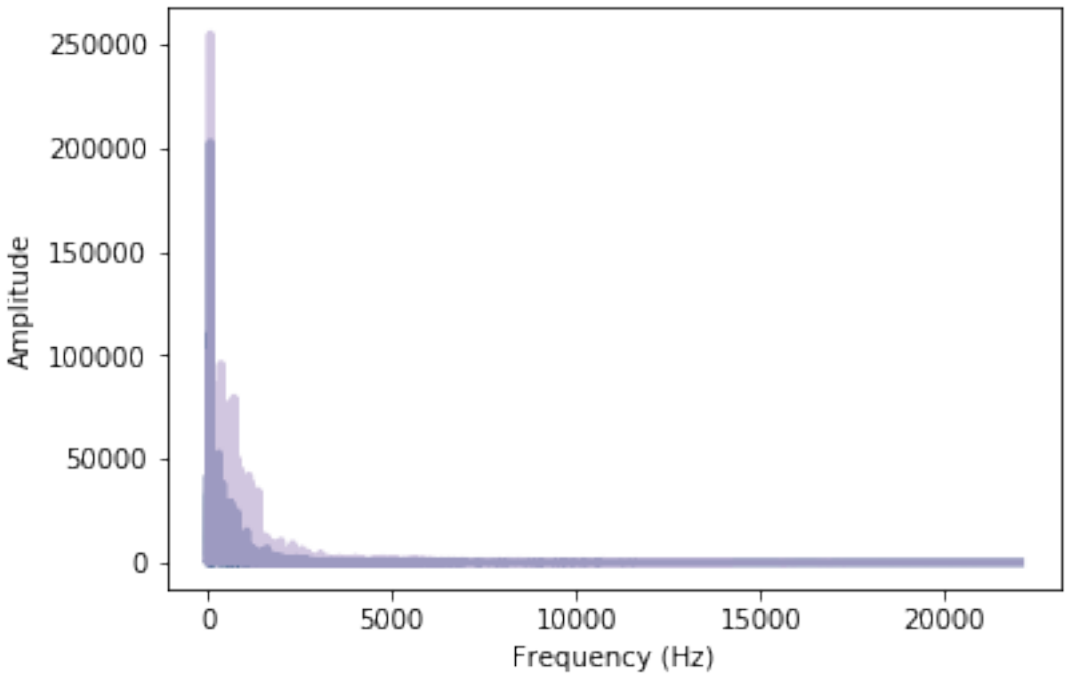
Выделим ещё один фрагмент на записи, чтобы увидеть, как спектр изменяется с течением времени:

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

[6]:
```
<IPython.lib.display.Audio object>
```

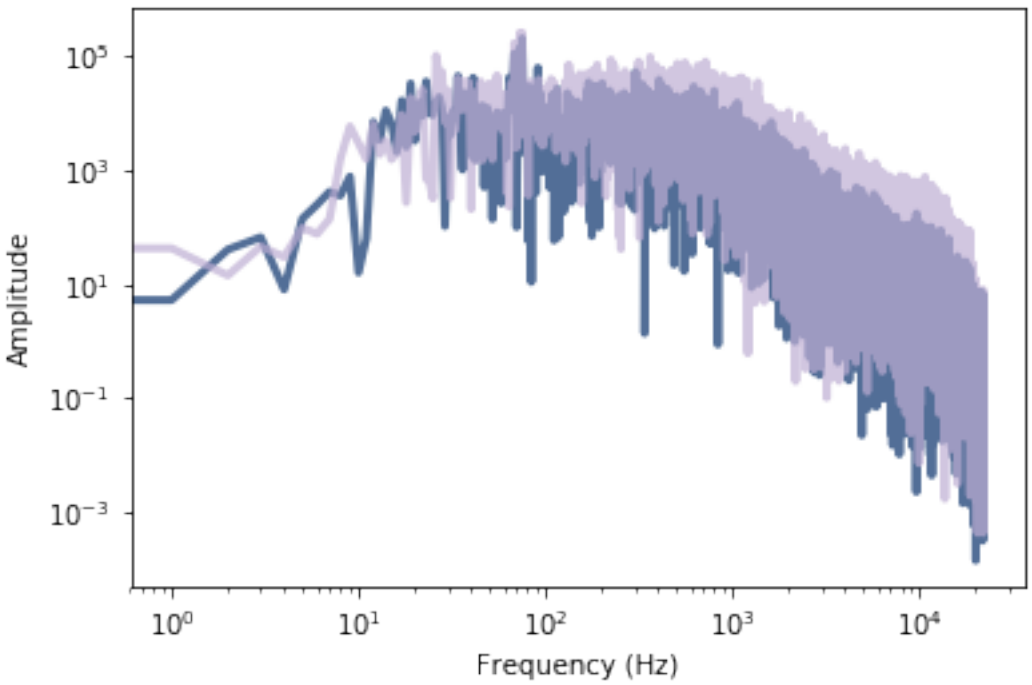
Выведем спектры обоих сегментов на один график:

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



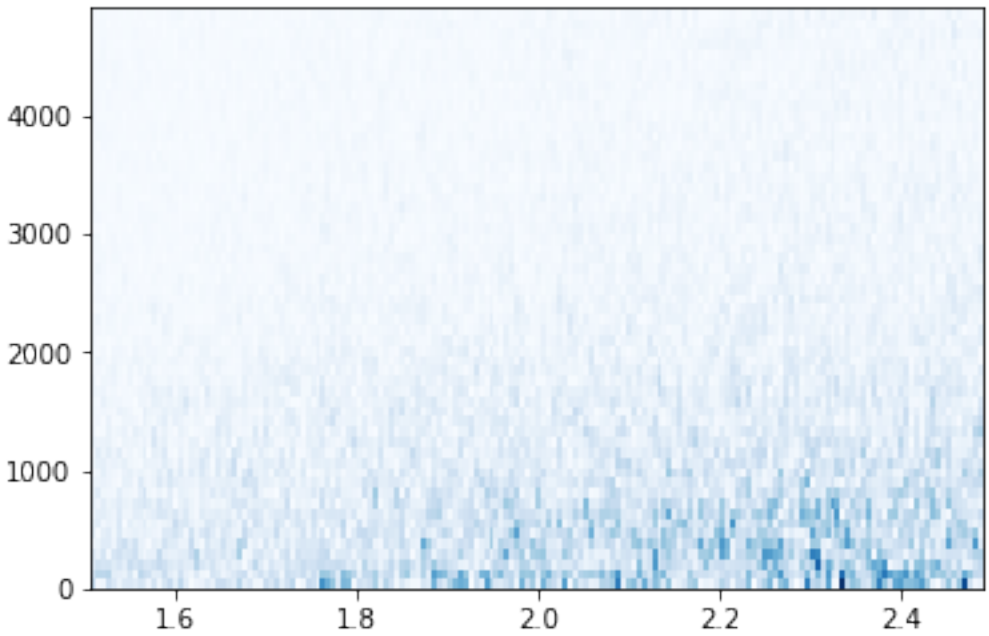
В двойном логарифмическом масштабе:

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



Структура сигнала не изменяется с течением времени. Получим спектрограмму:

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



В рамках данного сегмента общая амплитуда увеличивается.

2.2 Упражнение 4.2.

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

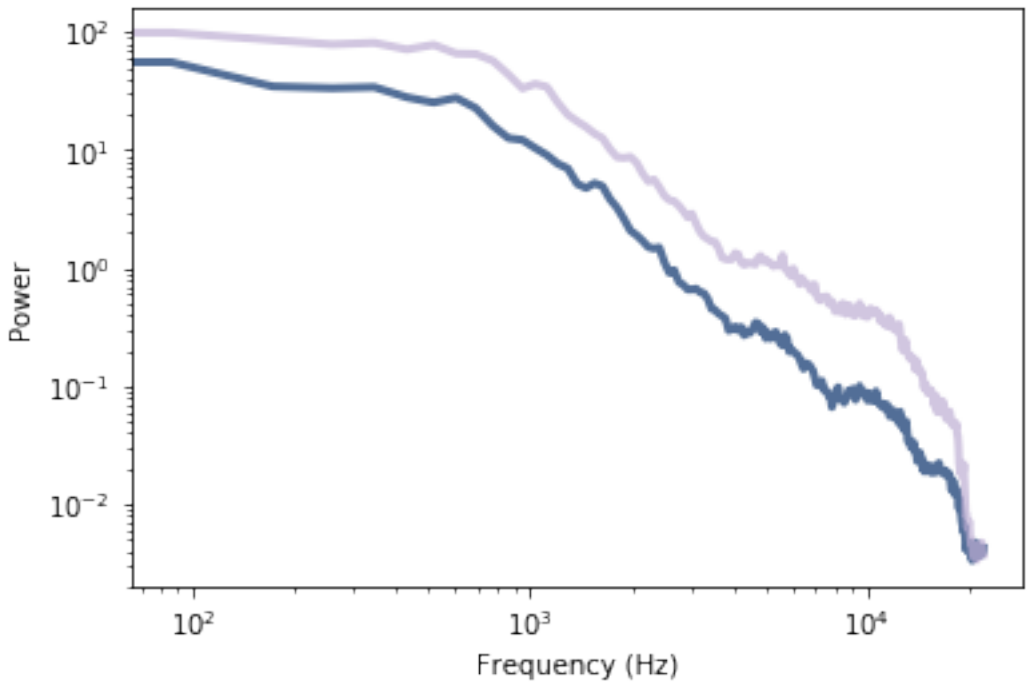
To estimate the long-term average power at each frequency, we can break a long signal into segments, compute the power spectrum for each segment, and then compute the average across the segments. You can read more about this algorithm at http://en.wikipedia.org/wiki/Bartlett's_method.

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

Реализация метода Бартлетта:

[10]:
```
def bartlett_method(wave, seg_length=512, win_flag=True):
    # Получение спектрограммы и извлечение спектров
    spectro = wave.make_spectrogram(seg_length, win_flag)
    spectrums = spectro.spec_map.values()

    # Извлечение массива мощностей для каждого спектра
    psds = [spectrum.power for spectrum in spectrums]

    # Вычисление среднего квадратического (Амплитуды)
    hs = np.sqrt(sum(psds) / len(psds))
    fs = next(iter(spectrums)).fs

    #Результирующий спектр
    spectrum = thinkdsp.Spectrum(hs, fs, wave.framerate)
    return spectrum
```

[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')
```



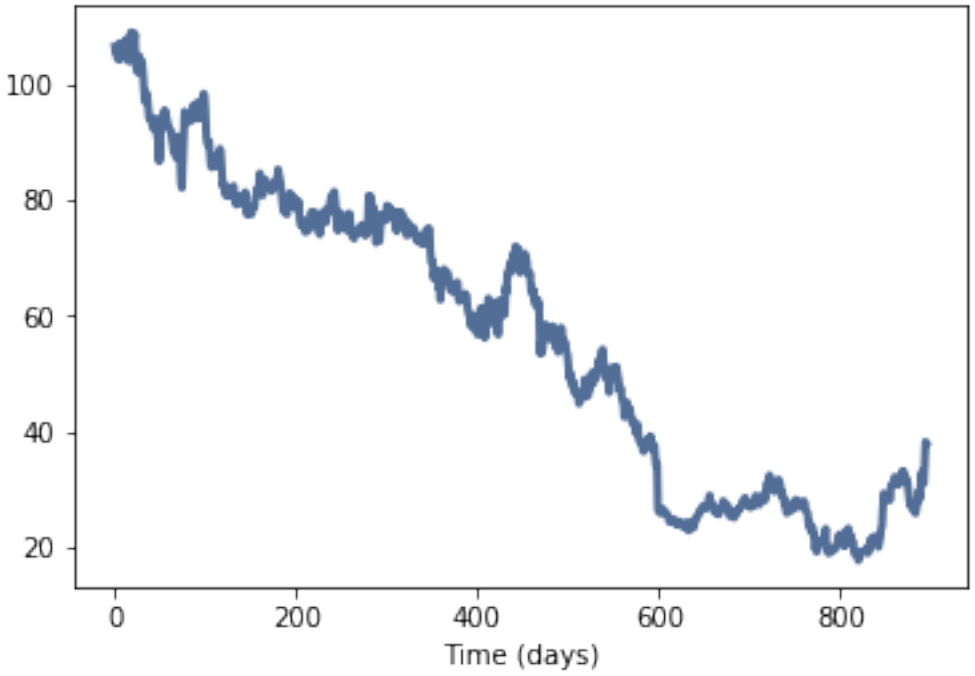
При использовании метода Бартлетта зависимость между частотой и мощностью видна куда лучше. Это не простая линейная зависимость, но она одинакова для разных сегментов даже в достаточно мелких деталях.

2.3 Упражнение 4.3.

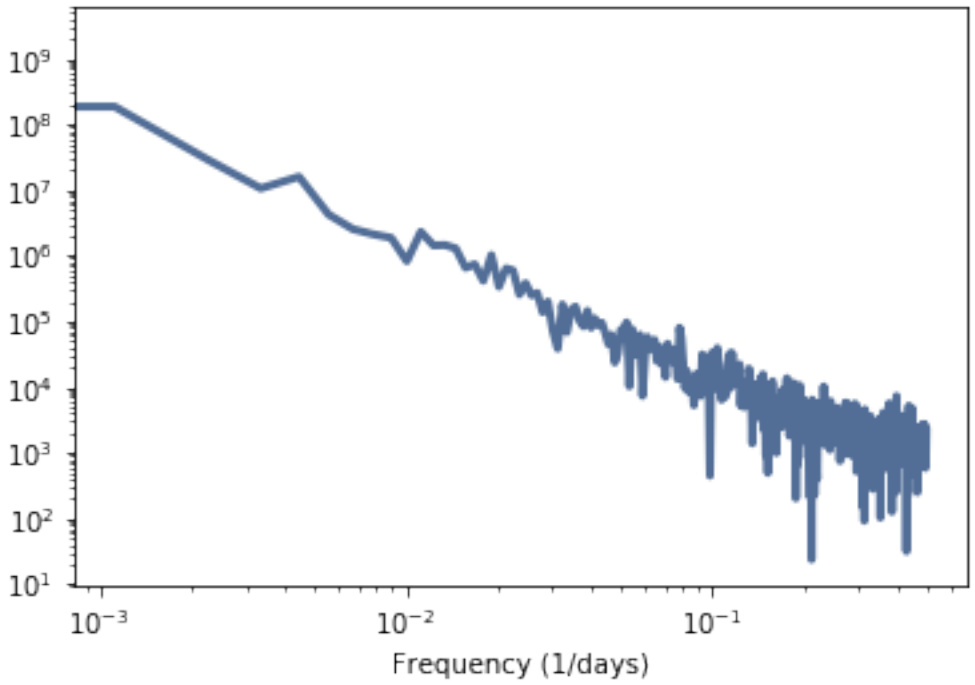
At http://www.coindesk.com you can download the daily price of a BitCoin as a CSV file. Read this file and compute the spectrum of BitCoin prices as a function of time. Does it resemble white, pink, or Brownian noise?

[12]:
```
df = pd.read_csv('BitCoin.csv', nrows=1625, parse_dates=[0])
ys = df.Close.values
ts = np.arange(len(ys))
```

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



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



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

[15]:
```
-1.857980433494222
```

Уклон спектра мощности приблизительно равен -1.85, из чего следует, что шум близок к красному шуму, уклон которого равен -2.

2.4 Упражнение 4.4.

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

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

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

Класс, реализующий некоррелированный пуассоновский шум(UP):

[16]:
```
class UncorrelatedPoissonNoise(thinkdsp._Noise):
    def evaluate(self, ts):
        ys = np.random.poisson(self.amp, len(ts))
        return ys
```

Прослушаем сигнал при маленькой амплитуде (низком уровне радиации):

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

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

[17]:
```
<IPython.lib.display.Audio object>
```

Для проверки сравним ожидаемое число частиц и реальное:

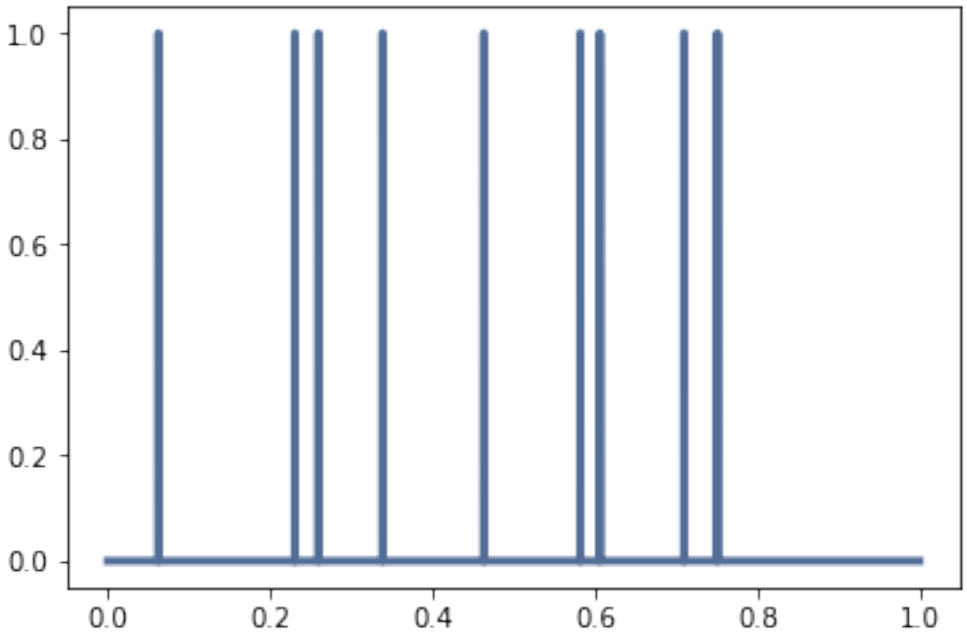
[18]:
```
expected = amp * framerate * duration
actual = sum(wave.ys)
```

```
print(expected, actual)
```

```
10.0 9
```

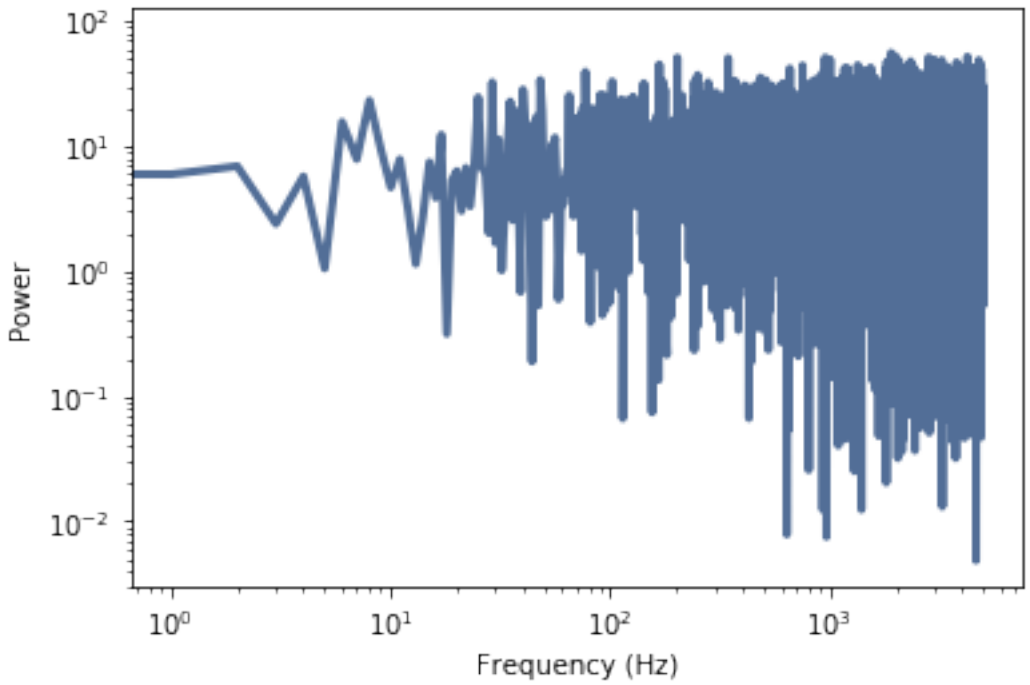
Сигнал, при этом, выглядит следующим образом:

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



Его интегральный спектр в двойном логарифмическом масштабе:

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



Выглядит очень похоже на спектр белого шума

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

[21]:
```
0.008747296245786103
```

О схожести с белым шумом говорит и уклон спектра мощности, близкий к нулю

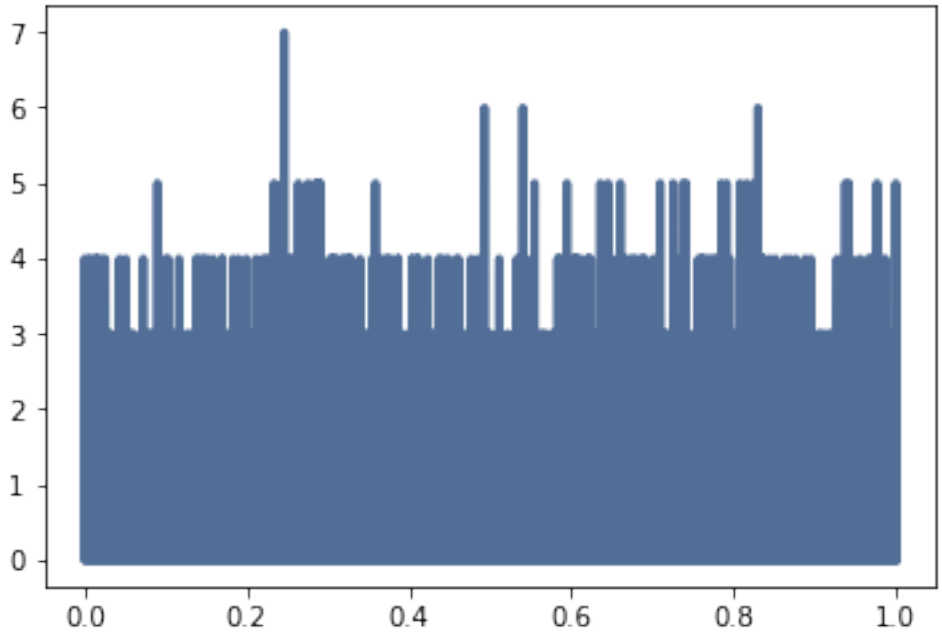
При больших значениях амплитуды сигнал звучит как белый шум:

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

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

[22]:
```
<IPython.lib.display.Audio object>
```

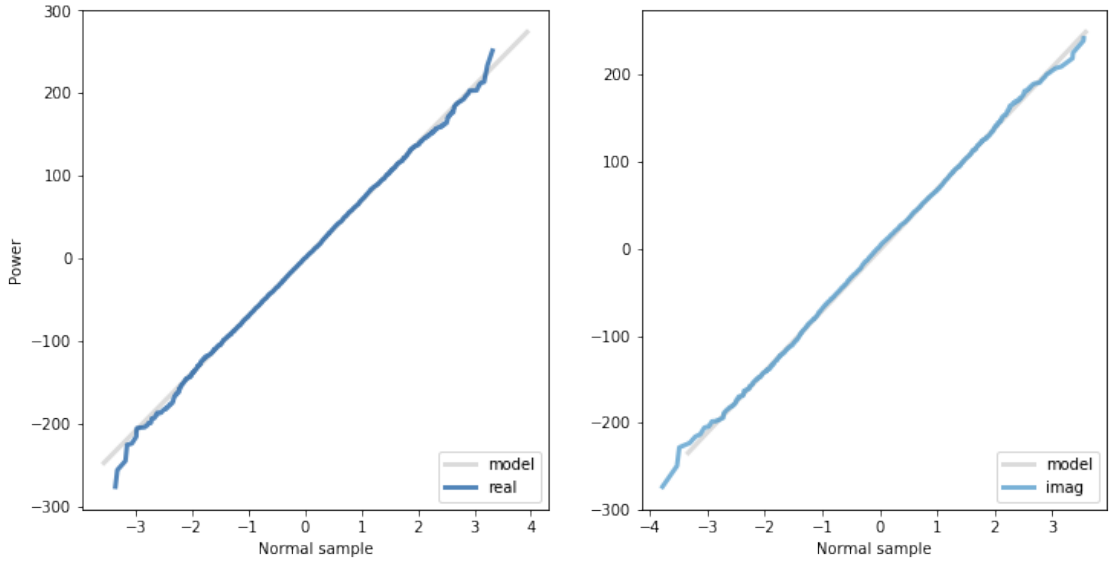
Сам сигнал выглядит следующим образом:

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



Спектр сигнала является Гауссовским шумом:

[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')
```



2.5 Упражнение 4.5.

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

Основная идея этого алгоритма состоит в сложении нескольких последовательностей случайных чисел, которые обновляются с различной частотой дискретизации. Первый источник должен обновляться на каждом временном шаге; второй источник каждый второй шаг; третий источник каждый четвертый шаг и т.д.

Реализация начинается с массива с одной строкой для каждого временного шага и одним столбцом для каждого из источников белого шума. Первоначально первая строка и первый столбец являются случайными, а остальная часть массива - NaN.

[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]
```

[25]:
```
array([[0.68893742, 0.70022763, 0.99454899, 0.97486277, 0.60380594],
       [0.21734525,        nan,        nan,        nan,        nan],
       [0.06976497,        nan,        nan,        nan,        nan],
       [0.99834583,        nan,        nan,        nan,        nan],
```

```
       [0.45632409,        nan,        nan,        nan,        nan],
       [0.72780473,        nan,        nan,        nan,        nan]])
```

Следующим шагом является выбор мест, в которых будут меняться случайные числа. Если число строк равно , количество изменений в первом столбце равно , во втором столбце в среднем равно / 2, в третьем столбце в среднем равно / 4 и т.д.

Таким образом, общее количество изменений в матрице составляет в среднем 2, поскольку из них находятся в первом столбце и ещё находятся в остальной части матрицы.

Чтобы разместить оставшиеся изменений(не считая первого столбца), мы генерируем случайные столбцы с помощью геометрического распределения с = 0.5. Если мы генерируем значение за пределами границ,то устанавливаем его равным 0 (первый столбец получает дополнительные значения).

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

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

Для каждого столбца выберем случайную строку из с помощью равномерного распределения:

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

[27]:
```
array([99, 95, 28, 38,  3, 65, 79,  1,  5,  4, 12, 62,  2, 99, 17, 75, 65,
       21,  2, 90, 89, 61, 95, 72, 48, 69, 49, 58, 12, 83, 57, 42, 45, 33,
       70, 22, 36, 74, 29, 19, 60, 15, 90, 61, 64,  1,  9, 65, 42, 66, 45,
       35, 33, 63, 25, 41, 78, 91, 45, 55,  4, 56, 16, 44, 29, 15, 59, 67,
        5, 56, 37, 12, 44, 39, 25, 64, 12, 59, 54, 55, 68, 82,  8, 56, 22,
        9, 18, 55, 29, 26, 40, 79, 66, 65, 25, 70, 39, 32, 85, 11])
```

Поместим случайные значения в выбранные ячейки:

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

[28]:
```
array([[0.68893742, 0.70022763, 0.99454899, 0.97486277, 0.60380594],
       [0.21734525, 0.41629885,        nan,        nan,        nan],
       [0.06976497, 0.39192704, 0.82440457,        nan,        nan],
       [0.99834583,        nan,        nan, 0.80357965,        nan],
       [0.45632409, 0.9642679 , 0.40235156,        nan,        nan],
```

```
       [0.72780473,        nan, 0.1275656 ,        nan,        nan]])
```

С помощью `fillna` заполним NaN-ы:

[29]:
```
df = pd.DataFrame(array)
filled = df.fillna(method='ffill', axis=0)
filled.head()
```

[29]:

	0	1	2	3	4
0	0.688937	0.700228	0.994549	0.974863	0.603806
1	0.217345	0.416299	0.994549	0.974863	0.603806
2	0.069765	0.391927	0.824405	0.974863	0.603806
3	0.998346	0.391927	0.824405	0.803580	0.603806
4	0.456324	0.964268	0.402352	0.803580	0.603806

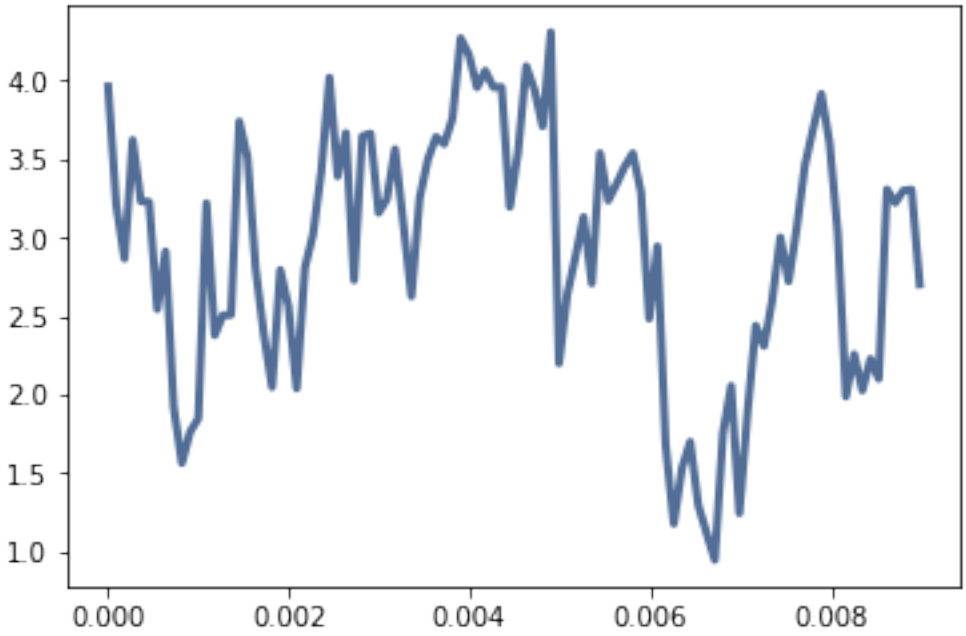
Сложим строки:

[30]:
```
total = filled.sum(axis=1)
total.head()
```

[30]:
```
0    3.962383
1    3.206862
2    2.864765
3    3.622063
4    3.230329
dtype: float64
```

Представим результат в виде wave:

[31]:
```
wave = thinkdsp.Wave(total.values)
wave.plot()
```



Объединим все предыдущие шаги в функцию:

[32]:
```
def voss(nrows, ncols=16):
    array = np.empty((nrows, ncols))
    array.fill(np.nan)
    array[0, :] = np.random.random(ncols)
    array[:, 0] = np.random.random(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
```

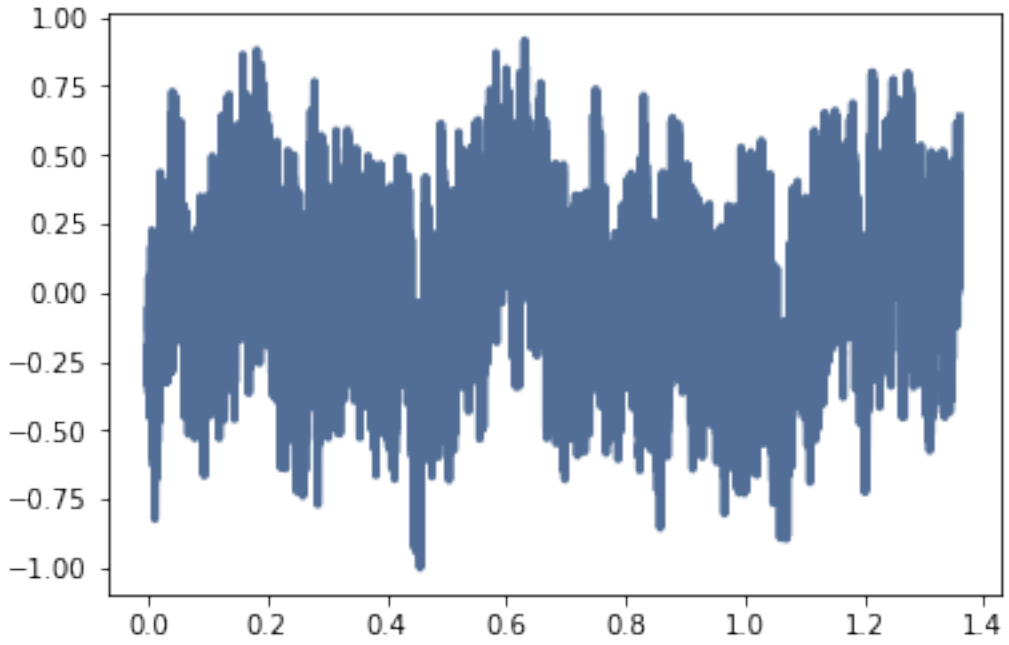
Сгенерируем 15000 значений:

[33]:
```
ys = voss(15000)
ys
```

[33]:
```
array([ 8.02013698,  7.49594438,  7.28863225, ...,  9.83200149,
        9.89747579, 10.41021809])
```

Превратим их в wave:

[34]:
```
wave = thinkdsp.Wave(ys)
wave.unbias()
wave.normalize()
wave.plot()
```



Сигнал, ожидаемо, выглядит менее случайно, чем белый шум, но более случайно, чем красный.

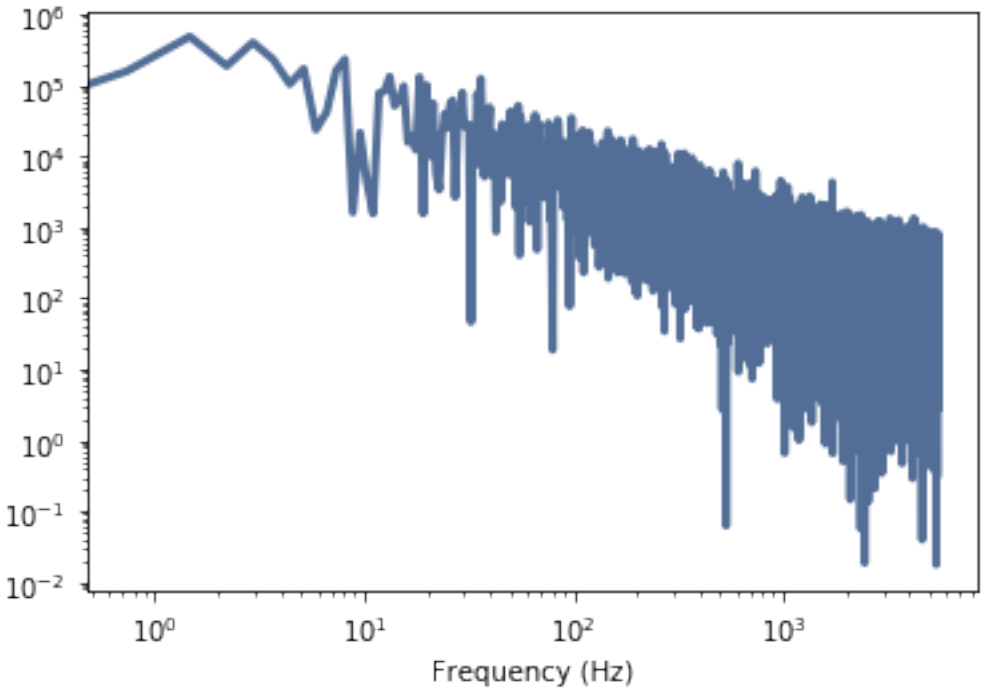
Прослушаем получившийся сигнал:

[35]:
```
wave.make_audio()
```

[35]:
```
<IPython.lib.display.Audio object>
```

Получим интегральный спектр:

[36]:
```
spectrum = wave.make_spectrum()
spectrum.hs[0] = 0
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (Hz)', xscale='log', yscale='log')
```



Уклон спектра мощности приблизительно равен -1:

[37]:
```
spectrum.estimate_slope().slope
```

[37]:
```
-0.9701226492039562
```

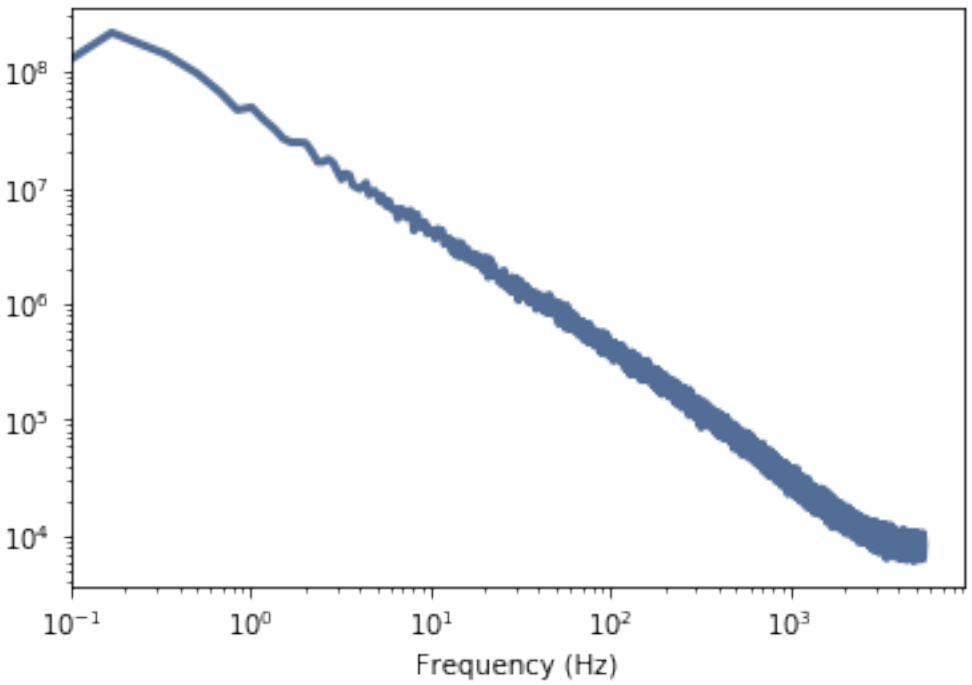
Создадим более длинный сигнал:

[38]:
```
seg_length = 64 * 1024
iters = 100
wave = thinkdsp.Wave(voss(seg_length * iters))
len(wave)
```

[38]:
```
6553600
```

Получим спектр, используя метод Бартлетта:

[39]:
```
spectrum = bartlett_method(wave, seg_length=seg_length, win_flag=False)
spectrum.hs[0] = 0
spectrum.plot_power()
thinkplot.config(xlabel='Frequency (Hz)', xscale='log', yscale='log')
```



[40]:
```
spectrum.estimate_slope().slope
```

[40]:
```
-1.0012805698724079
```

Уклон спектра мощности остался близким к -1

3 Вывод

В ходе выполнения данной работы были получены навыки работы с различными шумами (белыми, красными, розовыми). Кроме того, был рассмотрен интегральный спектр и его применение.