Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальный систем и суперкомпьютерных технологий

Телекоммуникационные технологии

Отчёт по лабораторным работам

Работу выполнил:
И. А. Сергеев
Группа:
3530901/90101
Преподаватель:
Н. В. Богач

Санкт-Петербург
2022

Содержание

1. Звуки и сигналы **4**
- 1.1. Упражнение 1 . . . 4
- 1.2. Упражнение 2 . . . 4
- 1.3. Упражнение 3 . . . 8
- 1.4. Упражнение 4 . . . 10
- 1.5. Вывод . . . 12

2. Гармоники **13**
- 2.1. Упражнение 1 . . . 13
- 2.2. Упражнение 2 . . . 13
- 2.3. Упражнение 3 . . . 15
- 2.4. Упражнение 4 . . . 16
- 2.5. Упражнение 5 . . . 17
- 2.6. Упражнение 6 . . . 18

3. Непериодические сигналы **21**
- 3.1. Упражнение 1 . . . 21
- 3.2. Упражнение 2 . . . 21
- 3.3. Упражнение 3 . . . 22
- 3.4. Упражнение 4 . . . 23
- 3.5. Упражнение 5 . . . 24
- 3.6. Упражнение 6 . . . 25

4. Шумы **29**
- 4.1. Упражнение 1 . . . 29
- 4.2. Упражнение 2 . . . 32
- 4.3. Упражнение 3 . . . 33
- 4.4. Упражнение 4 . . . 35
- 4.5. Упражнение 5 . . . 37

5. Автокорреляция **40**
- 5.1. Упражнение 1 . . . 40
- 5.2. Упражнение 2 . . . 41
- 5.3. Упражнение 3 . . . 43
- 5.4. Упражнение 4 . . . 46

6. Дискретное косинусное преобразование **47**
- 6.1. Упражнение 1 . . . 47
- 6.2. Упражнение 2 . . . 51
- 6.3. Упражнение 3 . . . 53

7. Дискретное преобразование Фурье **54**
- 7.1. Упражнение 1 . . . 54
- 7.2. Упражнение 2 . . . 54

8. Фильтрация и свертка **56**
8.1. Упражнение 1 . . . 56
8.2. Упражнение 2 . . . 56
8.3. Упражнение 3 . . . 59

9. Дифференциация и интеграция **62**
9.1. Упражнение 1 . . . 62
9.2. Упражнение 2 . . . 62
9.3. Упражнение 3 . . . 64
9.4. Упражнение 4 . . . 66
9.5. Упражнение 5 . . . 69

10. Сигналы и системы **73**
10.1. Упражнение 1 . . . 73
10.2. Упражнение 2 . . . 76

11. Модуляция и сэмплирование **81**
11.1. Упражнение 1 . . . 81
11.2. Упражнение 2 . . . 81
11.3. Упражнение 3 . . . 81

12. FSK **87**
12.1. Теория . . . 87
12.2. Тестирование . . . 90

1. Звуки и сигналы

1.1. Упражнение 1

Прочитаны объяснения и запущены примеры chap01.ipynb

1.2. Упражнение 2

Выберем в качестве рабочего варианта звуковой файл мелодии музыкальной шкатулки из репозитория ThinkDSP, предварительно загрузив сам репозиторий.

```
import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/
    master/code/thinkdsp.py
```

Затем скачаем с сайта freesound.org мелодию музыкальной шкатулки

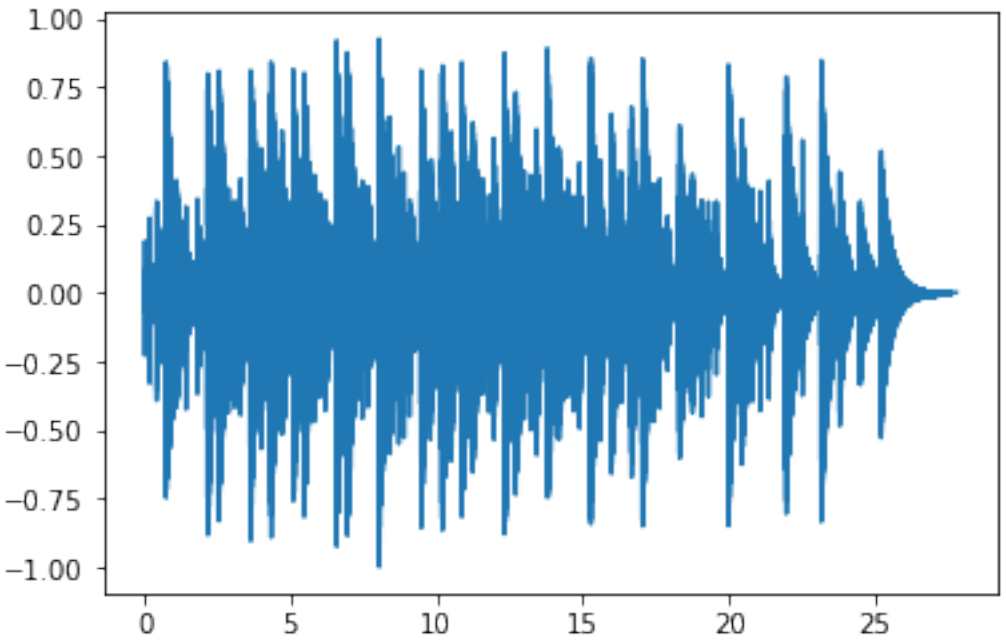
```
if not os.path.exists('369148__flying-deer-fx__music-box-the-flea-waltz.wav
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/369148__flying-deer-fx__music-box-the-flea-waltz.wav
```

Прочитаем и запустим аудио файл

```
from thinkdsp import read_wave

wave = read_wave('369148__flying-deer-fx__music-box-the-flea-waltz.wav')
wave.normalize()
wave.make_audio()
```

Построим график спектра

```
wave.plot()
```

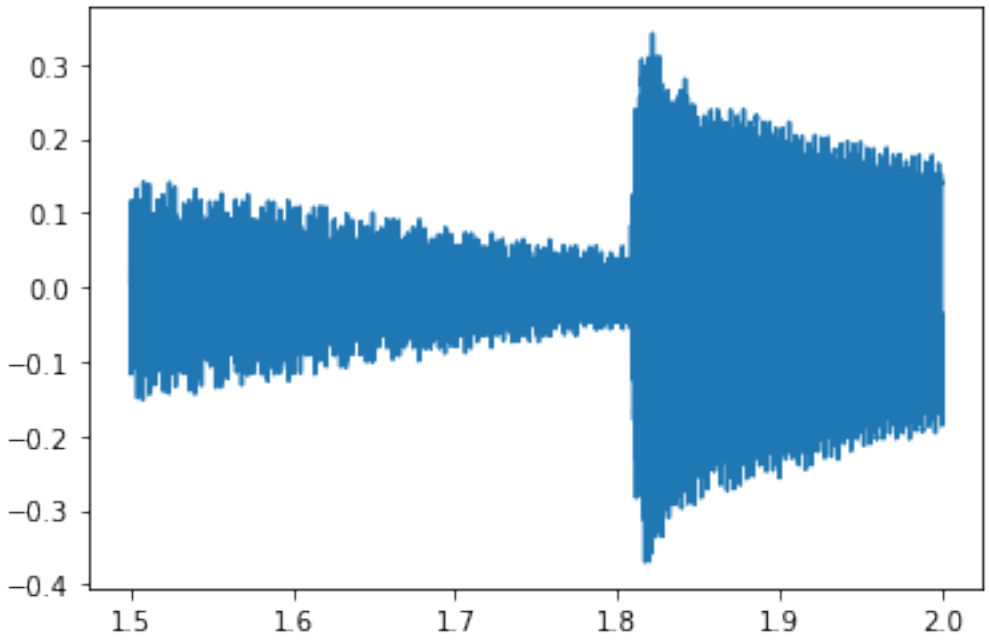


Теперь вырежем полусекундный отрывок

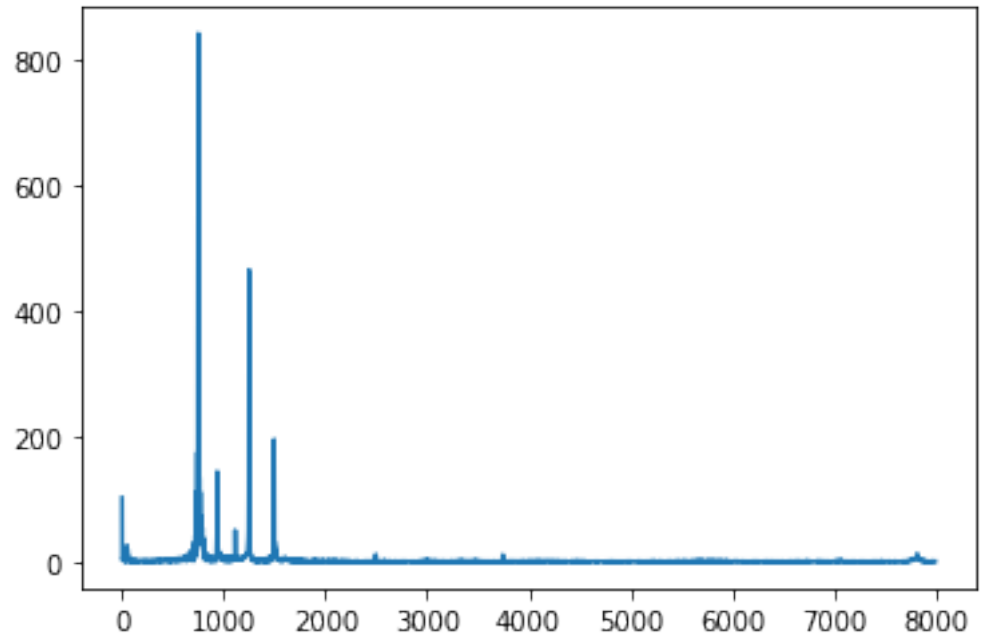
```
segment = wave.segment(start=1.5, duration=0.5)
segment.make_audio()
```

Посмотрим на график спектра

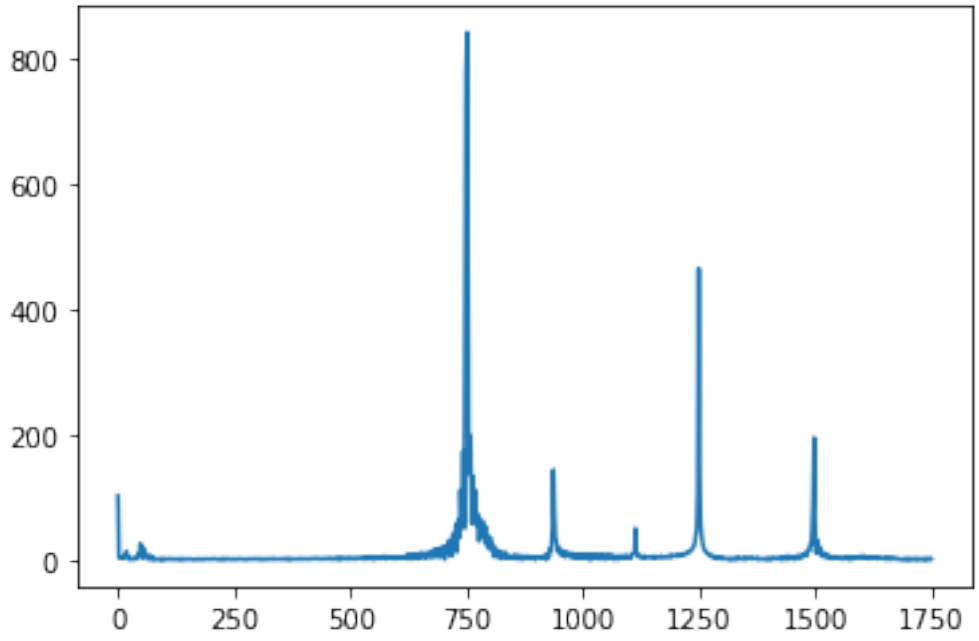
```
segment.plot()
```




```
spectrum = segment.make_spectrum()
spectrum.plot(high=8000)
```



```
spectrum = segment.make_spectrum()
spectrum.plot(high=1750)
```



Заметим, что доминирующая частота равна 750 Гц, что приблизительно соответствует ноте Фа-диез второй октавы с частотой 739 Гц.
Попробуем обработать наш звук. Для начала удалим высокочастотные звуки, из-за чего звук должен станет звучать "ниже". Удалим частоты выше 5000 Гц.

```
spectrum.low_pass(5000)
```

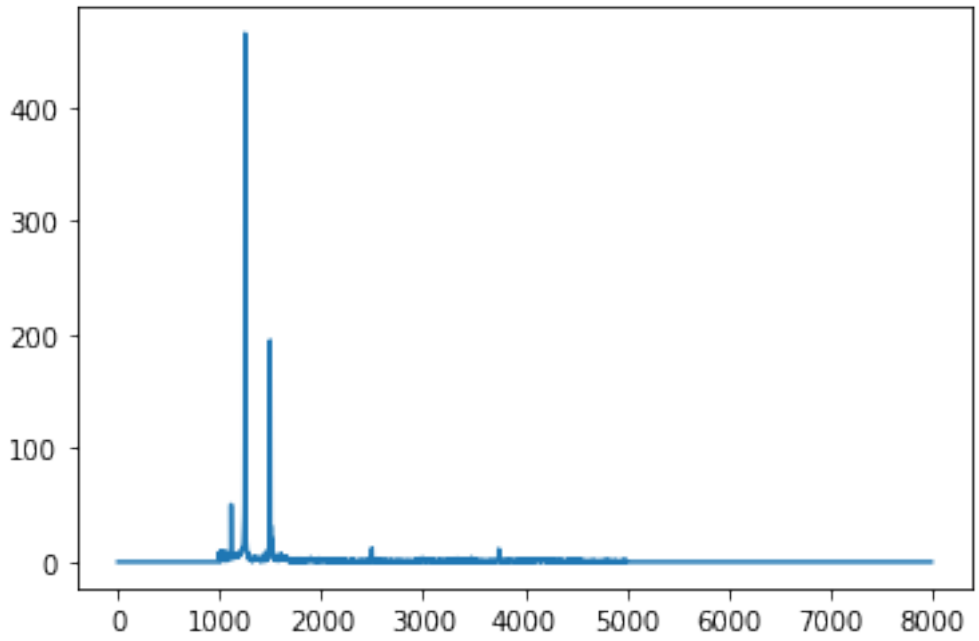
```
spectrum.make_wave().make_audio()
```

Также удалим низкочастотные звуки ниже 1000 Гц.

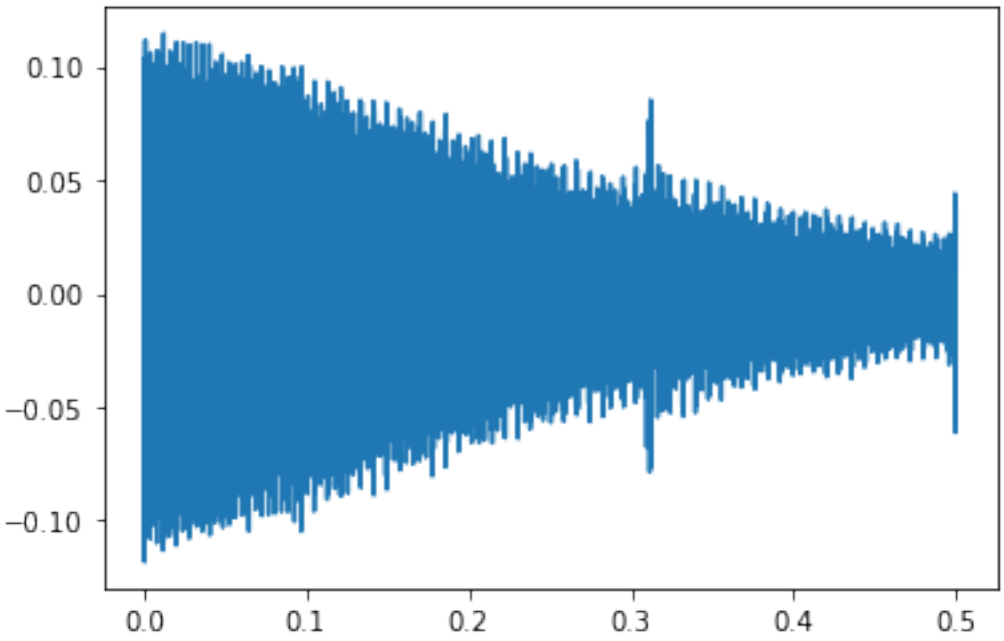
```
spectrum.high_pass(1000)
```

```
spectrum.make_wave().make_audio()
```

```
spectrum.plot(8000)
```




```
filtered = spectrum.make_wave()
filtered.plot()
filtered.make_audio()
```

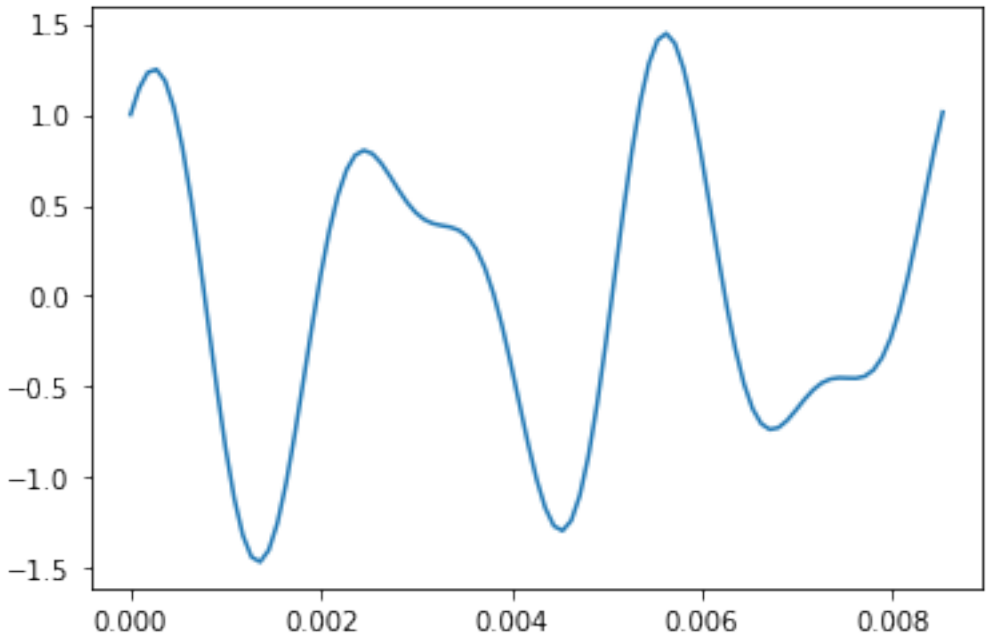


Как можно увидеть из графика спектра, в следствие применения низкочастотного фильтра домонирующая частота стала выше и теперь равна примерно 1250 Гц. Также стало меньше и высоких частот. В итоге, второй звук звучит бледнее, потерялся тембр мелодии. А как мы знаем, тембр отвечает за окраску звука, полноту всех частот. Так как мы вырезали некоторые частоты, то потеряли и тембр.

1.3. Упражнение 3

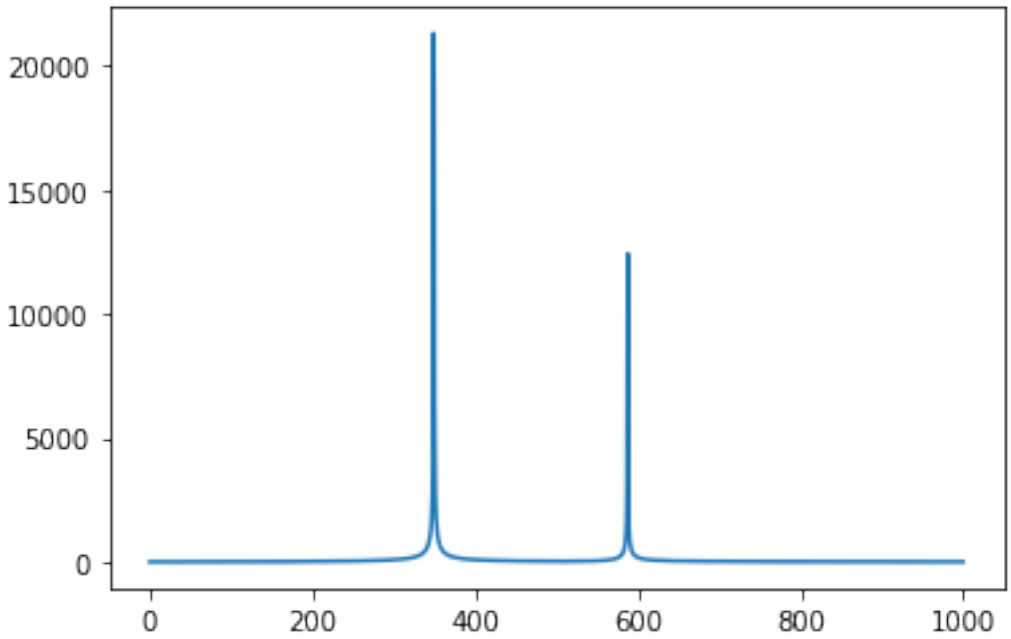
Возьмём в качестве синусоидального сигнала ноту Ре второй откавы, а в качестве косиноидального ноту Фа первой октавы. Сложим сигналы и получим график.

```
import thinkdsp

sin_signal = thinkdsp.SinSignal(freq=587.32, amp=0.5, offset=0)
cos_signal = thinkdsp.CosSignal(freq=349.23, amp=1.0, offset=0)
mix = cos_signal + sin_signal
mix.plot()
```



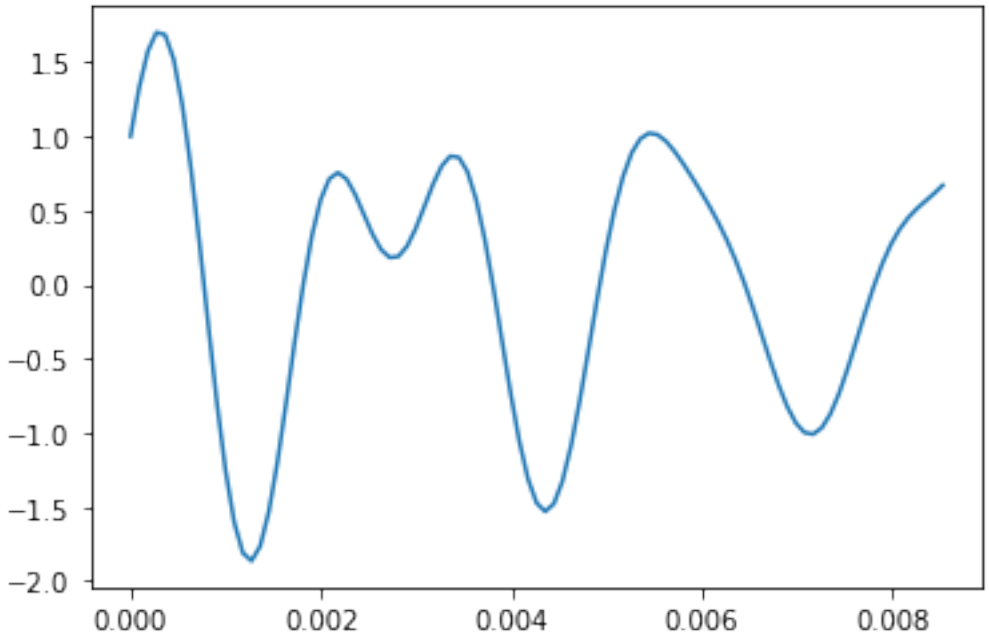
Воспроизведём звук из двух сложенных сигналов

```
wave = mix.make_wave(duration=2, start=0, framerate=31025)
wave.make_audio()

spectrum = wave.make_spectrum()
spectrum.plot(1000)
```

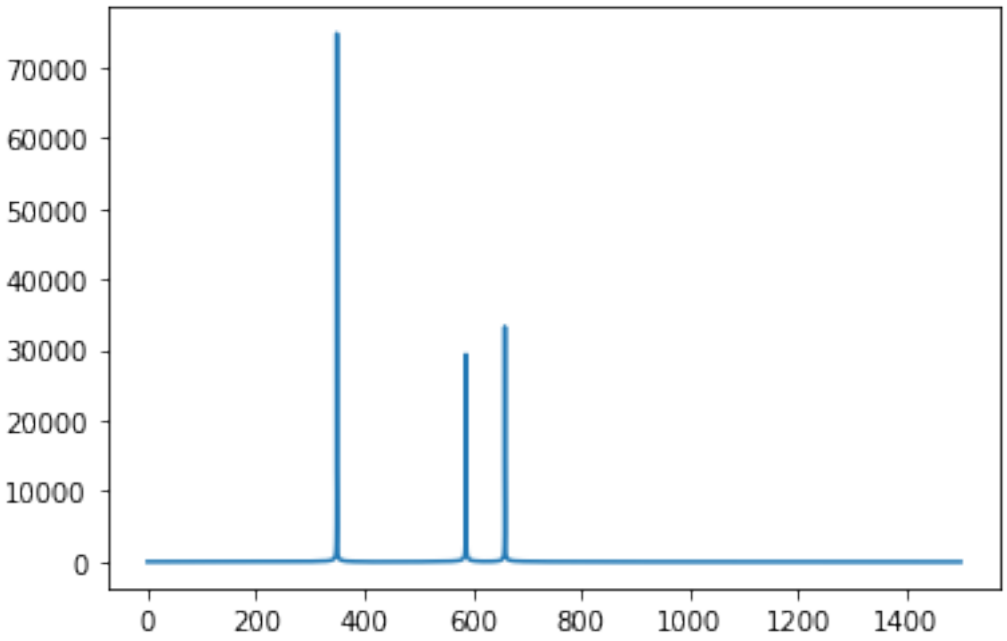


Теперь добавим частоту, не кратную остальным. Это будет нота Ми второй октавы, интервал которой с Ре второй октавы составляет малую секунду.

```
sin_signal2 = thinkdsp.SinSignal(freq=659.26, amp=0.5, offset=0)
mix += sin_signal2
mix.plot()
```




```
wave = mix.make_wave(duration=5, start=0, framerate=31025)
wave.make_audio()

spectrum = wave.make_spectrum()
spectrum.plot(1500)
```

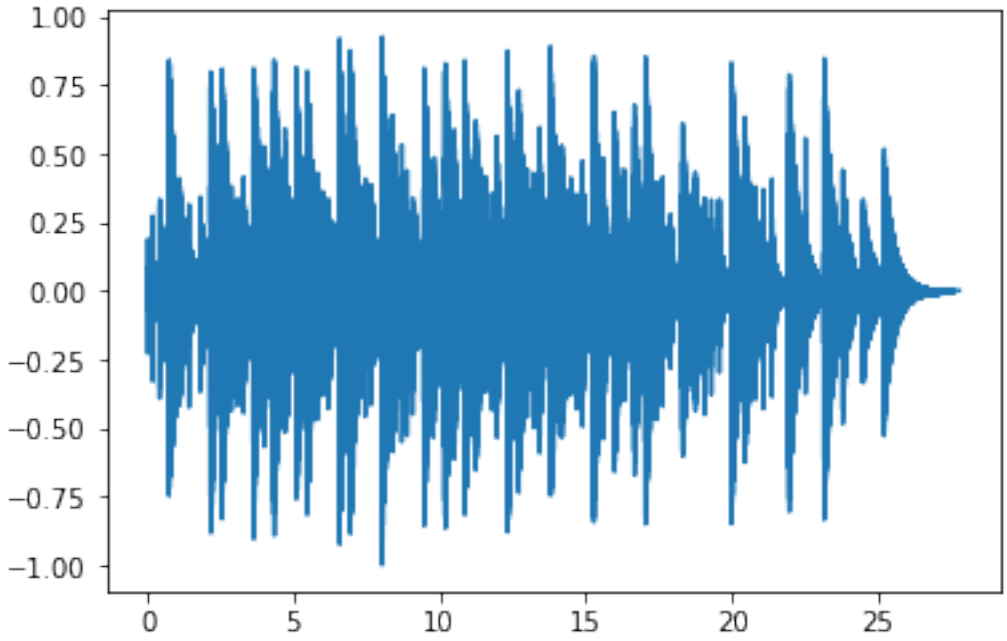


В итоге получили режущий слух звук.

1.4. Упражнение 4

Напишем функцию stretch, которая будет ускорять или замедлять сигнал изменением ts и framerate.
Для примера возьмём изначальную мелодию музыкальной шкатулки.

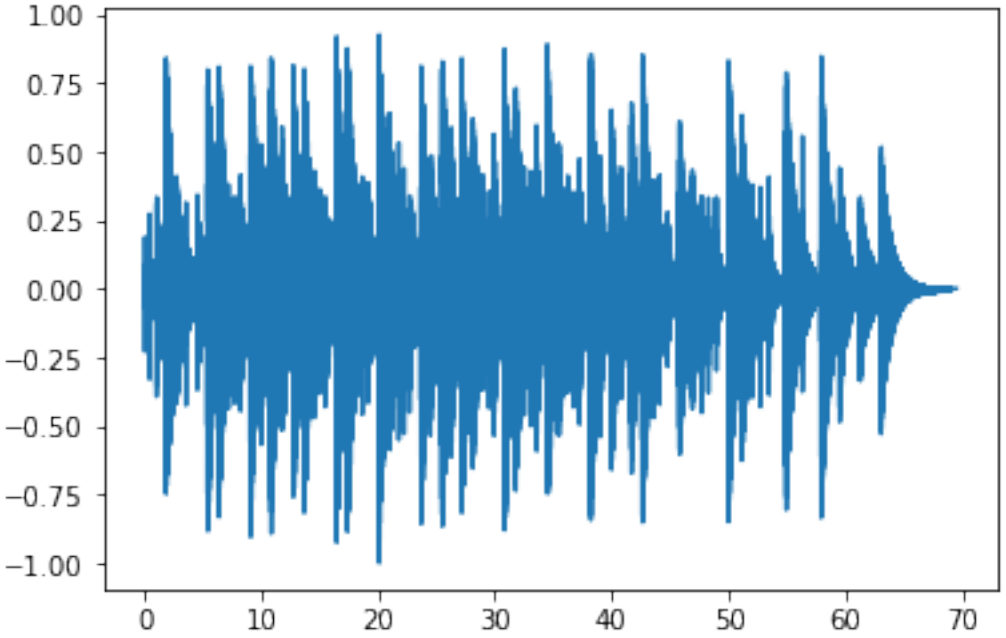
```
wave = read_wave('369148__flying-deer-fx__music-box-the-flea-waltz.wav')
fast = wave
slow = wave

wave.plot()
```



```
def stretch(wave, factor):
    wave.ts *= factor
    wave.framerate *= factor
```

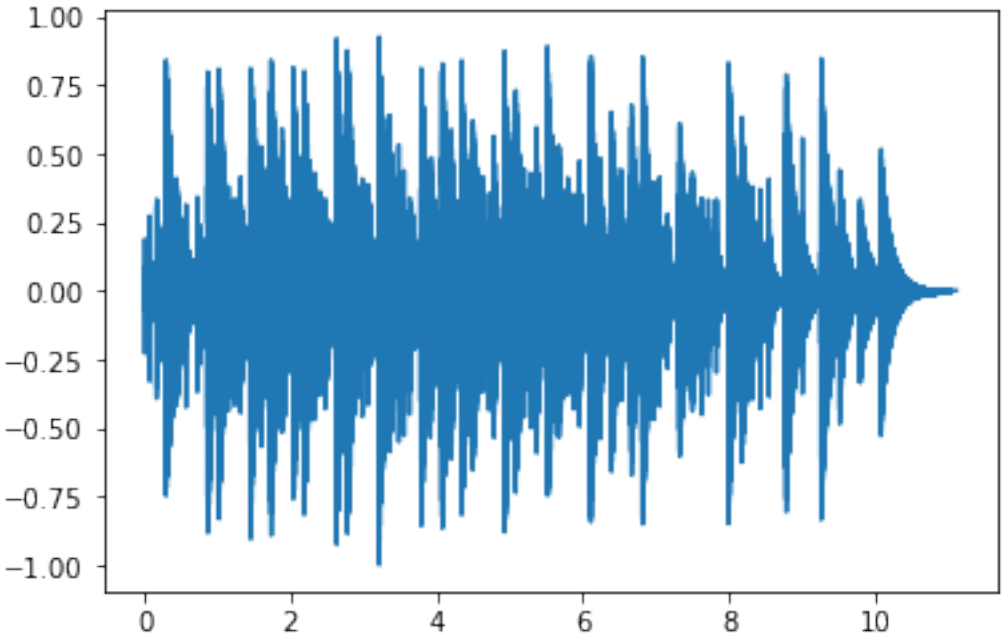
Ускорим в 2.5 раза

```
stretch(fast, 2.5)
fast.make_audio()

fast.plot()
```



Замедлим в 2.5 раза

```
stretch(slow, 0.4)
slow.make_audio()

slow.plot()
```



1.5. Вывод

В ходе данной работы было выполнено знакомство с основыми понятиями при работе со звуками и сигналами. При помощи библиотеки thinkDSP можно делать обширный круг взаимодействий с сигналами, как для их создания, так и для их обработки.

2. Гармоники

2.1. Упражнение 1

Просмотрены примеры из chap2.ipynb

2.2. Упражнение 2

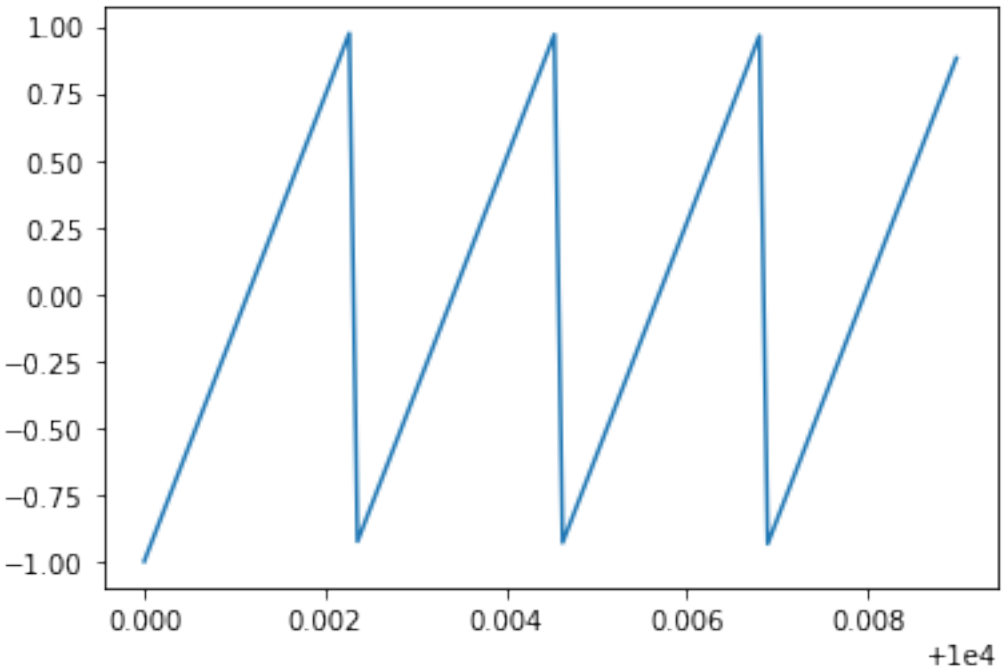
```
import os

if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master
    /code/thinkdsp.py
```

Создадим класс SawtoothSignal, который будет расширять signal и предоставлять evaluate для оценки пилообразного сигнала

```
import numpy as np
import thinkdsp
from thinkdsp import Sinusoid
from thinkdsp import normalize, unbias

class SawtoothSignal(Sinusoid):

    def evaluate(self, ts):
        cycles = self.freq * ts + self.offset / np.pi / 2
        frac, _ = np.modf(cycles)
        ys = normalize(unbias(frac), self.amp)
        return ys
```

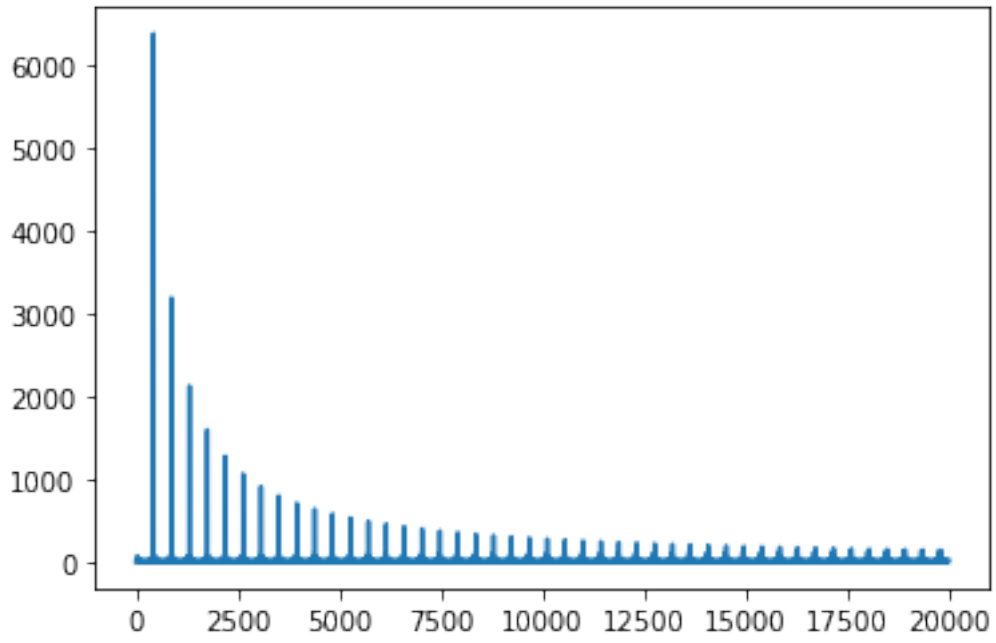
Cycles — число циклов со времени начала np.modf разделяет число циклов на дробную часть frac и целую часть, которая не используется.
frac — последовательность, растущая от 0 до 1 с заданной частотой
unbias — смещает сигнал так, что он центрируется относительно 0. normalize уже масштабирует его до заданной амплитуды amp.

```
signal = SawtoothSignal()
duration = signal.period*4
segment = signal.make_wave(duration, 10000)
segment.plot()
```



Так будет выглядеть спектр:

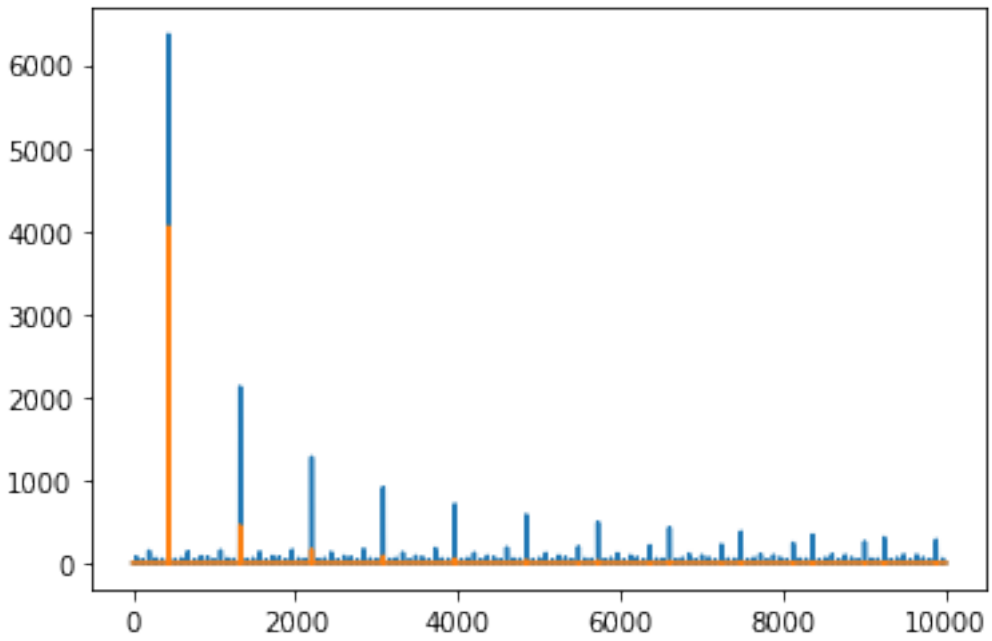
```
spec = SawtoothSignal().make_wave(duration=0.5,framerate=40000).make_spectru
spec.plot()
```



Сравним гармоническую структуру с треугольным (жёлтый) и прямоугольным (синий) сигналами

```
from thinkdsp import SquareSignal, TriangleSignal

square = SquareSignal().make_wave(duration=0.5, framerate=20000)
square.make_spectrum().plot()
```

```
triangle = TriangleSignal().make_wave(duration=0.5, framerate=20000)
triangle.make_spectrum().plot()
```



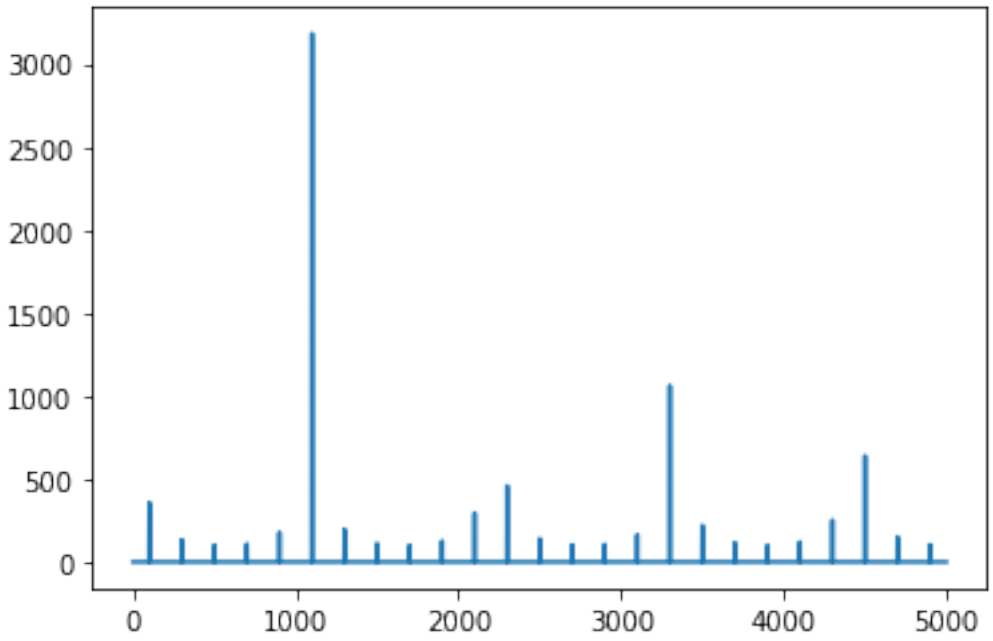
Сравнивая гармонические структуры квадратого и пилообразного сигнала, можно увидеть, что и пилообразный, и квадратный сигналы падают пропорционально 1/f , но пилообразный сигнал включает в себя четные и нечетные гармоники, в отличие от квадратного.

Сравнивая гармоническую структуру пилообразного сигнала и треугольного, видим, что треугольный сигнал падает $1/f^2$, а пилообразный $1/f$.

2.3. Упражнение 3

Создадим прямоугольный сигнал частотой 1100 Гц и вычислим wave с выборками 10000 кадров в секунду.

```
signal = thinkdsp.SquareSignal(1100)
segment = signal.make_wave(duration=0.5, framerate=10000)
spec = segment.make_spectrum()
spec.plot()
segment.make_audio()
```

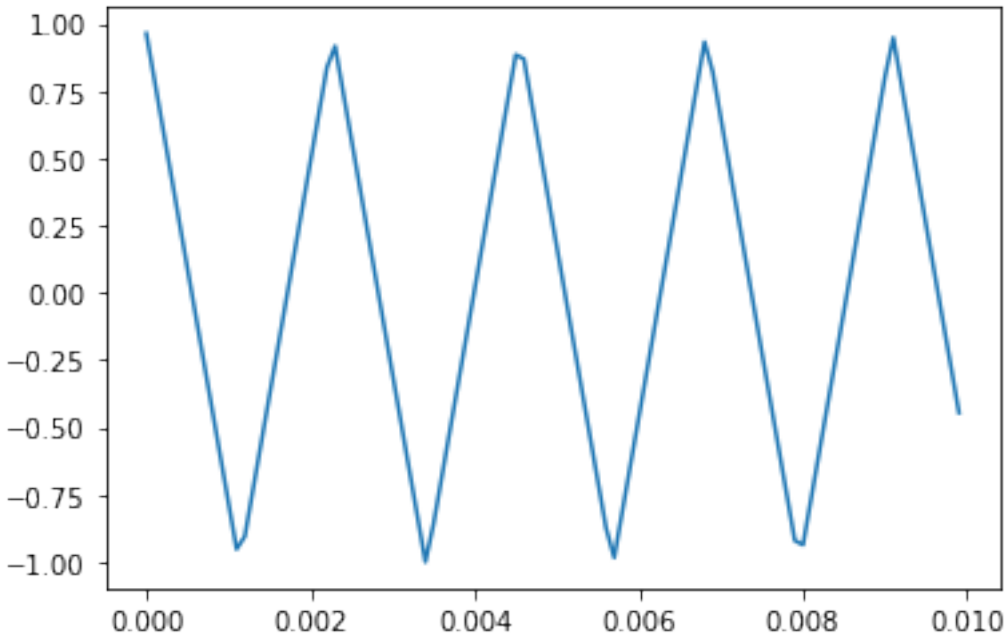


Заметим, что первая гармоника находится находится в нужном месте, тогда как вторая 5500 Гц совмещена с гармоникой 4500 Гц. Следующая гармоника совмещена с гармоникой на 2300 Гц.

2.4. Упражнение 4

Проведём эксперимент посредством создания треугольного сигнала частотой 440 Гц длительностью 0.01 сек., построением графика, распечатыванием Spectrum.hs[0] и в итоге узнаем, как повлияет на сигнал установка Spectrum.hs[0] = 100.

```
signal = thinkdsp.TriangleSignal(440)
segment = signal.make_wave(0.01, framerate=10000)
segment.plot()
```



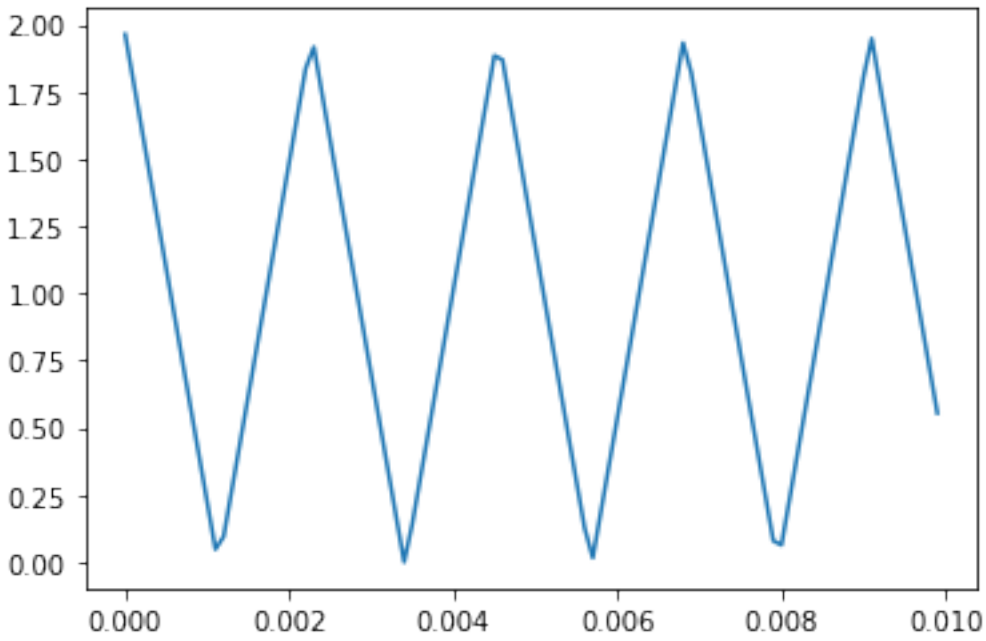
```
spec = segment.make_spectrum()
spec.hs[0]
```

(3.375077994860476e-14+0j)

Данное комплексное число - частотный компонент. Размах соответствует амплитуде компонента, а угол - фазе.

присвоим в качестве первого элемента массива 100 и посмотрим, что из этого выйдет.

```
spec.hs[0] = 100
spec.make_wave().plot()
```

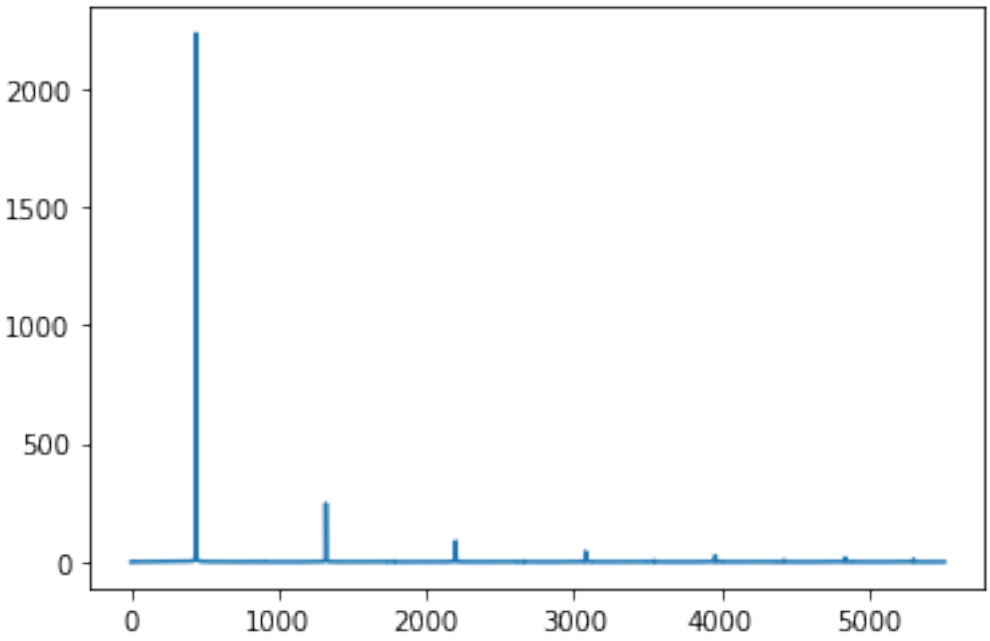


По графику видно, что сигнал сместился по вертикали. Из этого делаем вывод, что первый элемент массива hs отвечает за смещение сигнала относительно вертикали. Если элемент близок или равен нулю, то сигнал не смещенный.

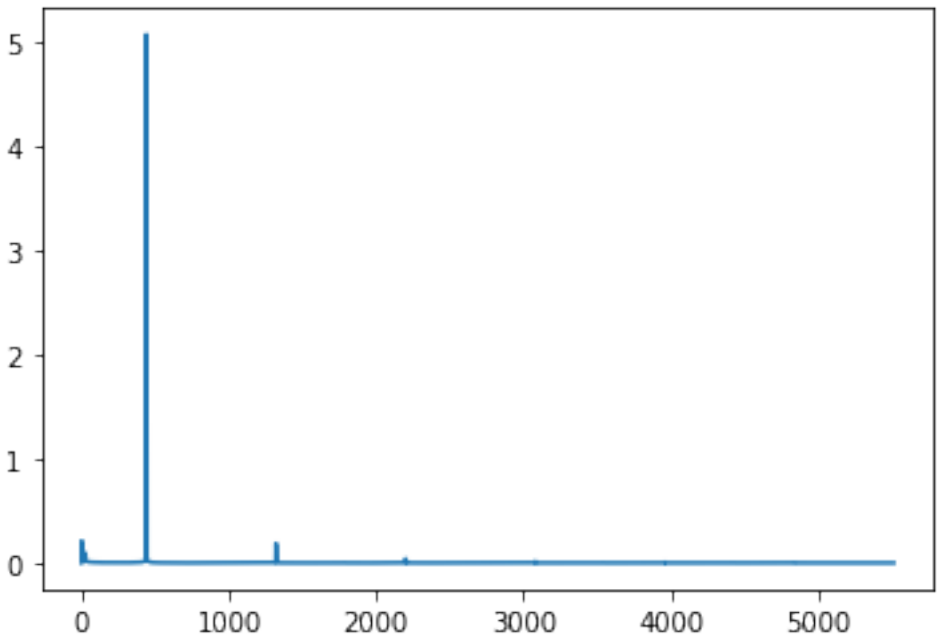
2.5. Упражнение 5

Напишем функцию, принимающую Spectrum как параметр и изменяющую его делением каждого элемента hs на соответствующую частоту из fs. Проверим функцию на треугольном сигнале.

```
def spectrum_divider(spectrum):

    spectrum.hs[1:] /= spectrum.fs[1:]
    spectrum.hs[0] = 0
    spectrum.plot()

signal = thinkdsp.TriangleSignal(freq=440).make_wave(duration=0.5)
spectrum = signal.make_spectrum()
spectrum.plot()
signal.make_audio()
```




```
spectrum_divider(spectrum)
```




```
spectrum.make_wave().make_audio()
```

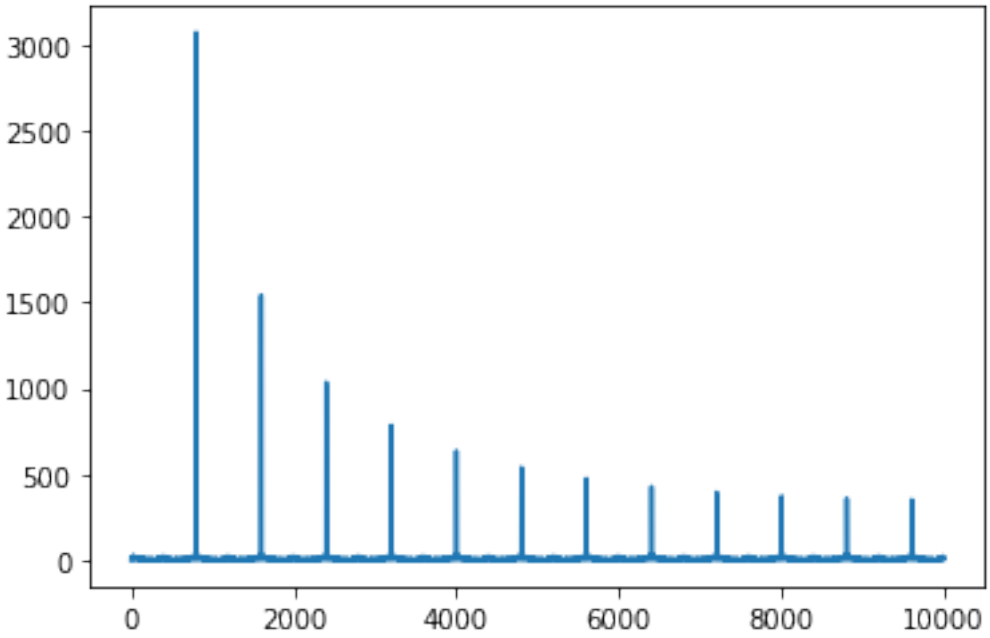
Сравнивая полученные графики, сделаем вывод о том, что функция действует как фильтр низких частот: частоты ослабляются на некоторую величину. В результате этого звук стал более глухим.

2.6. Упражнение 6

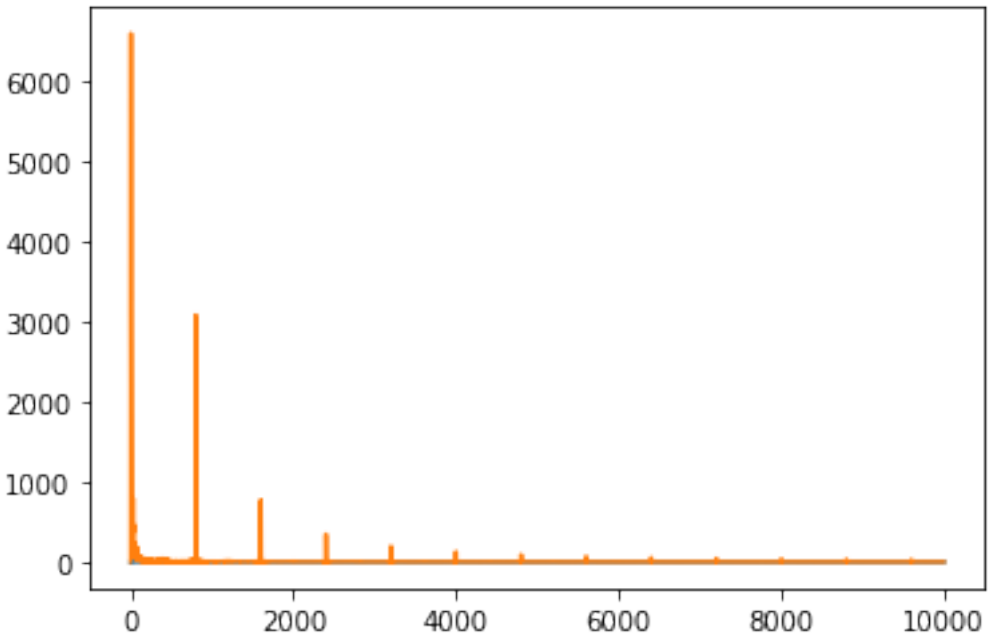
Создадим сигнал, состоящий из четных и нечетных гармоник, которые при этом будут падать пропорционально $1/f2$. Для этого воспользуемся одним из способов: создадим

пилообразный сигнал, которые имеет четные и нечетные гармоники, но при этом спадает не пропорционально $1/f2$, и применим к нему функцию из предыдущего упражнения $\text{spectrum}_d ivider 1/f^2$.

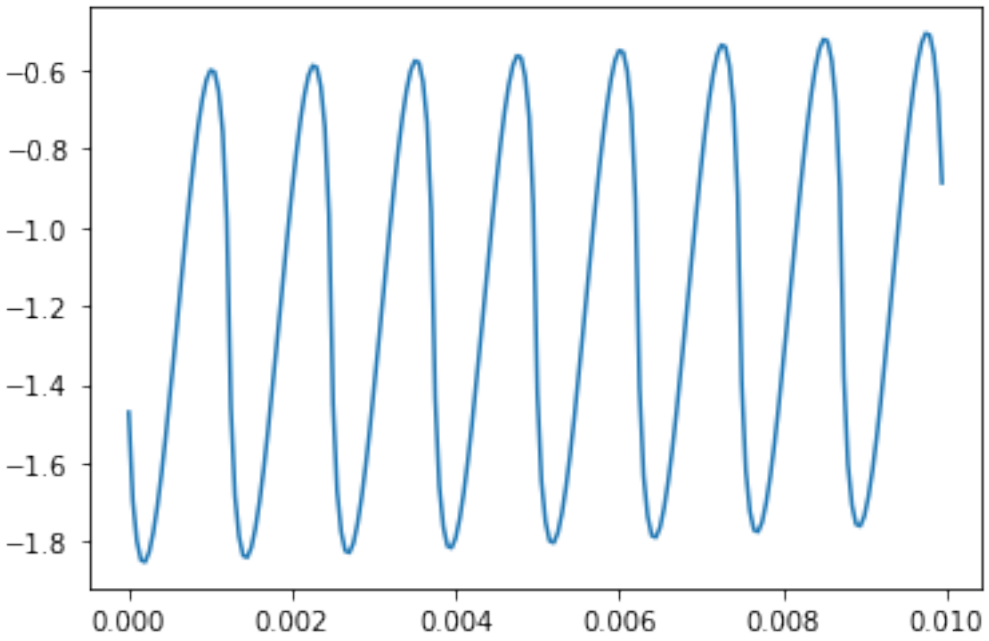
```
signal = thinkdsp.SawtoothSignal(800)
wave = signal.make_wave(duration=0.5, framerate=20000)
spectrum = wave.make_spectrum()
spectrum.plot()
```




```
spectrum_divider(spectrum)
spectrum.scale(800)
spectrum.plot()
```



```
spectrum.make_wave().segment(duration=0.01).plot()
```



Как видно из графиков, полученных после обработки сигнала, спектр имеет четные и нечетные гармоники, а также спадает пропорционально 1/f2. На последнем графике сигнал перестал быть пилообразным, однако и не стал синосуидальным.

3. Непериодические сигналы

3.1. Упражнение 1

Запущены и прослушаны примеры из блокнота chap03.ipynb.

3.2. Упражнение 2

Создадим коасс SawtoothChirp, расширяющий Chirp и переопределяющий evaluate для генерации пилообразного сигнала с линейно увеличивающейся/уменьшающейся частотой. Для этого нужно переопределить частный метод evaluate. np.diff вычисляет разницу между соседними элементами ts, возвращая длину каждого интервала в секундах. dphis — содержит изменение фазы. Благодаря этому, можно рассчитать полную фазу phases на каждом отрезке времени, суммируя изменения. np.cumsum вычисляет нарастающую сумму. Но т. к. он это делает не с нуля, то np.insert добавляет 0 в начало. frac — последовательность, растущая от 0 до 1 с заданной частотой, а unbias смещает сигнал так, что он центрируется к 0.

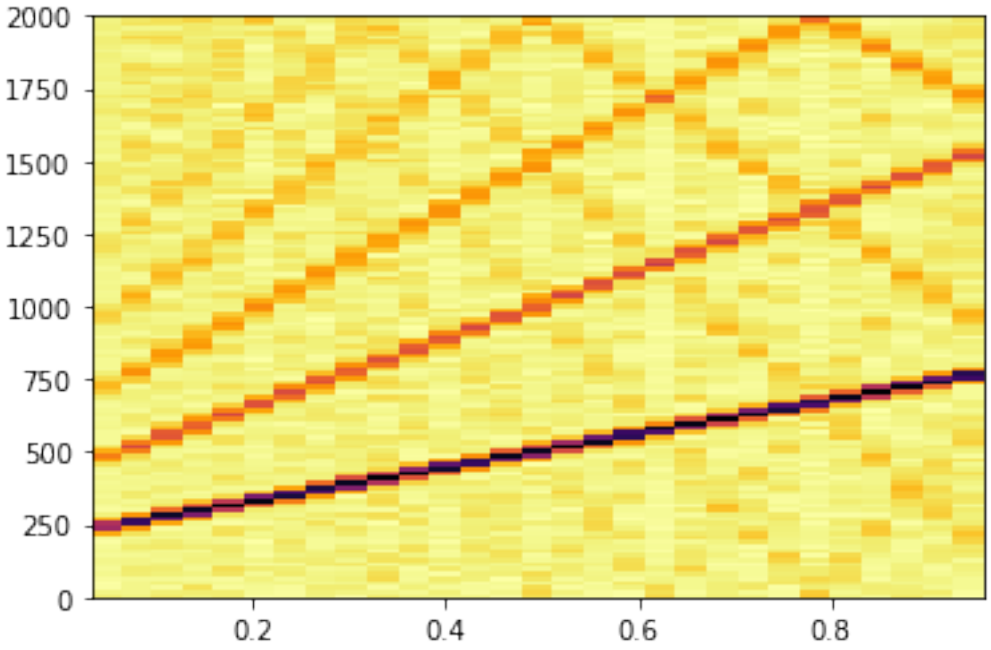
```
import os

if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master
    /code/thinkdsp.py

from __future__ import print_function, division
%matplotlib inline
import matplotlib.pyplot as plt
import os
import thinkdsp
import numpy as np
import math
PI2 = 2 * math.pi
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets

if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master/code/thinkdsp.p

class SawtoothChirp(thinkdsp.Chirp):

    def evaluate(self, ts):

        freqs = np.linspace(self.start, self.end, len(ts))
        dts = np.diff(ts, prepend=0)
        dphis = PI2 * freqs * dts
        phases = np.cumsum(dphis)
        cycles = phases / PI2
        frac, _ = np.modf(cycles)
        ys =  thinkdsp.normalize(thinkdsp.unbias(frac), self.amp)
        return ys
```

Прослушаем

```
sig = SawtoothChirp(start=220, end=800)
w = sig.make_wave(duration=1, framerate=4000)
w.apodize()
w.make_audio()
```

Спектограмма сигнала:

```
sp = w.make_spectrogram(seg_length=256)
sp.plot()
```

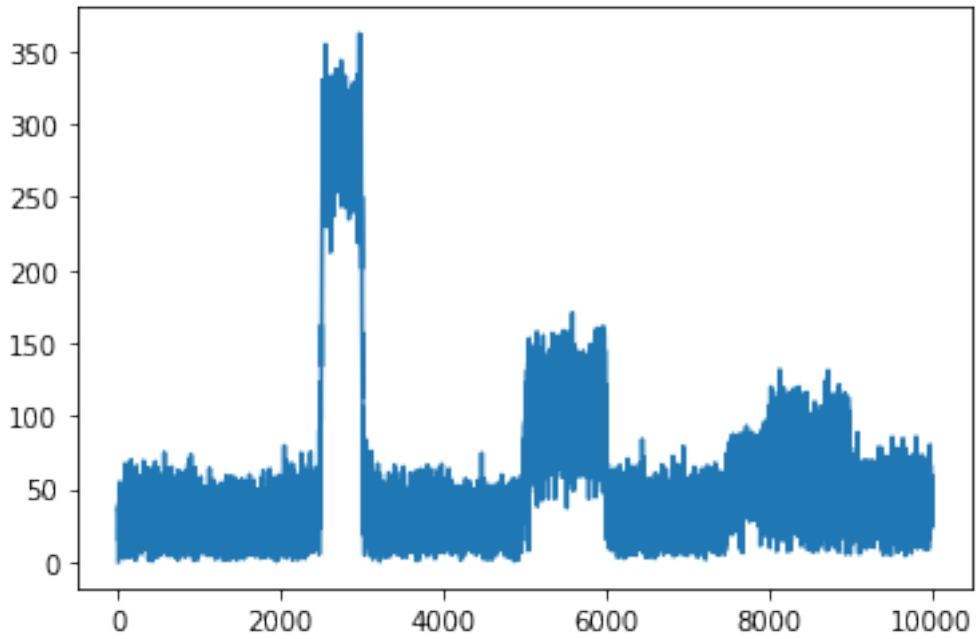


При относительно низкой частоте кадров мы можем видеть искаженные гармоники, отражающиеся от частоты сворачивания. И мы можем услышать их как фоновое шипение. Если увеличить частоту кадров, они исчезают.

3.3. Упражнение 3

Создадим пилообразный чирп, меняющийся с частотой от 2500 Гц до 3000 Гц, и на его основе создадим сигнал длительностью 1 секунду и частотой кадров 20 кГц. Распечатаем его спектр.

```
s = SawtoothChirp(start=2500, end=3000)
w = s.make_wave(duration=1, framerate=20000)
w.make_spectrum().plot()
w.make_audio()
```



В диапазоне от 2500 до 3000 Гц мы видим Око Саурона. Первая гармоника колеблется от 5000 до 6000 Гц, так что мы видим более короткую башню в этом диапазоне, как у флигеля Саурона. Вторая гармоника колеблется от 7500 до 9000 Гц, так что мы видим что-то еще более короткое в этом диапазоне, как Патио Саурона.
Другие гармоники повсюду накладываются друг на друга, поэтому мы видим некоторую энергию на всех других частотах. Эта распределенная энергия создает некоторые интересные звуки.

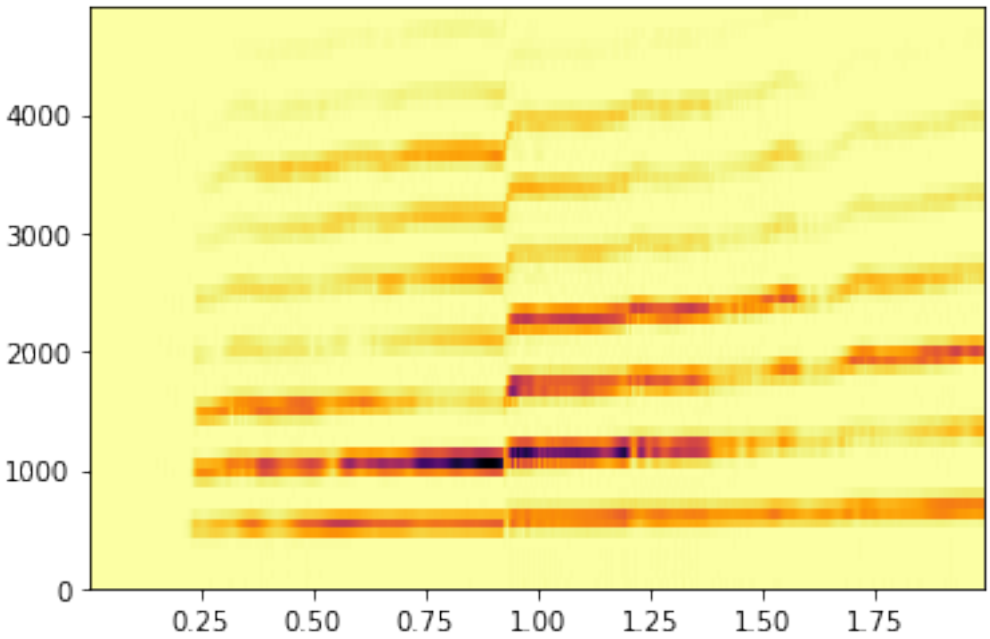
3.4. Упражнение 4

Для этого упражнения был найден звук глиссандо на кларнете. Проанализируем этот звук, построим его спектрограмму.

```
if not os.path.exists('72475__rockwehrmann__glissup02.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/72475__ro
```

```
from thinkdsp import read_wave

wave = read_wave('72475__rockwehrmann__glissup02.wav')
start = 0
duration = 2
segment = wave.segment(start, duration)
segment.make_audio()
```

```
segment.make_spectrogram(512).plot(high=5000)
```



3.5. Упражнение 5

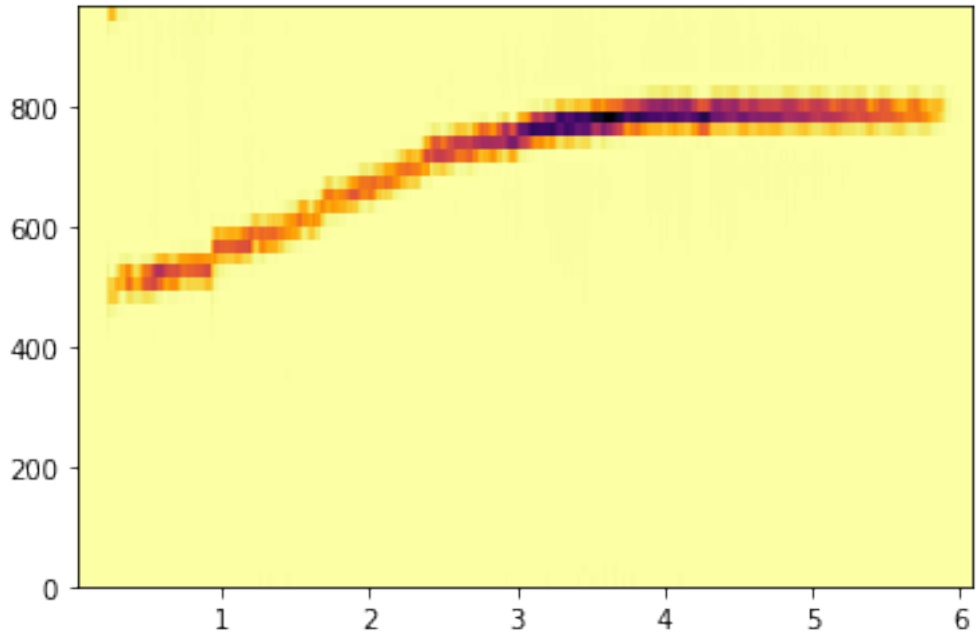
Напишем класс TrmboneGliss, расширяющий класс Chirp и переопределяющий метод evaluate. Для этого для вычисления частоты используется функция np.linspace, но в аргументы функции передаётся не просто начальная и конечная частоты, а единица, деленная на эти частоты.

```
class TromboneGliss(thinkdsp.Chirp):
  def evaluate(self, ts):
    l1, l2 = 1.0 / self.start, 1.0 / self.end
    lengths = np.linspace(l1, l2, len(ts))
    freqs = 1 / lengths

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

Создадим два сигнала-глиссандо от C3 до F3 и обратно, воспроизведем эти звуки, соединим два сигнала в один и построим спектрограмму.

```
sig = TromboneGliss(262, 349)
w1 = sig.make_wave(duration=1)
w1.make_audio()

sig = TromboneGliss(349, 262)
w2 = sig.make_wave(duration=1)
w2.make_audio()

w = w1 | w2
w.make_audio()
```

```
spec_gram = wave.make_spectrogram(2048)
spec_gram.plot(high=1000)
```

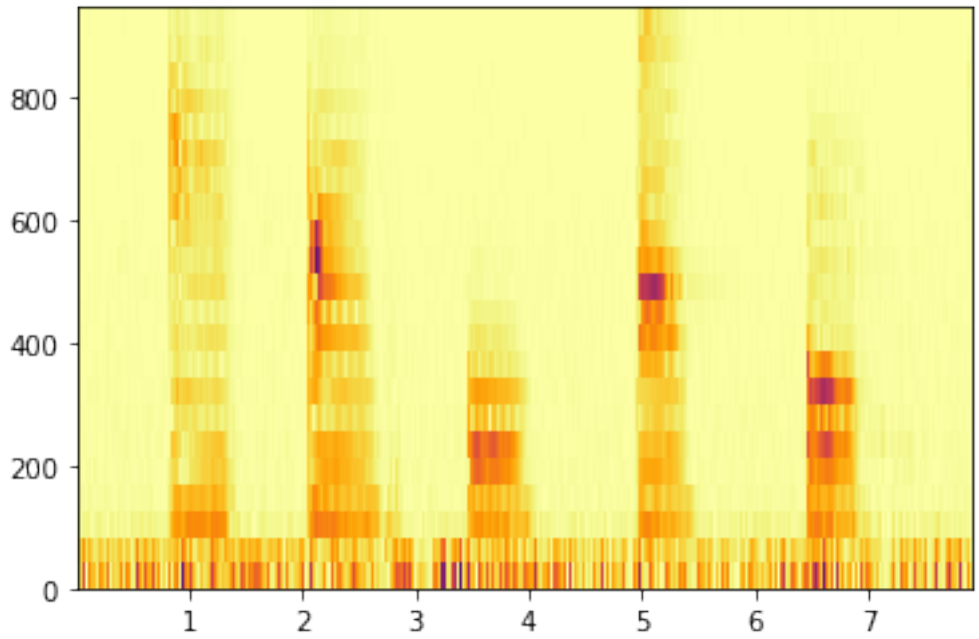


Из полученного графика, заметим, что частота сначала возрастает, а потом снова уменьшается до начальной. Следовательно, написанный класс работает корректно. Однако, из-за маленькой разницы в начальной и конечной частотах невозможно определить на что похоже глиссандо на тромбоне — на линейный или экспоненциальный чирп.

3.6. Упражнение 6

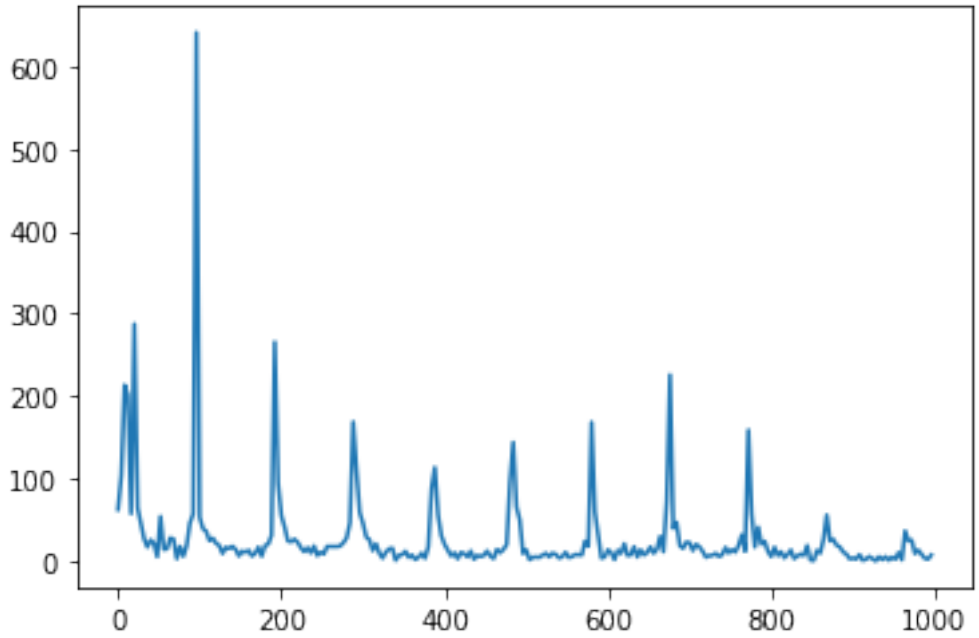
Создайте или найдите запись серии гласных звуков и посмотрите на спектрограмму. Сможете ли вы различить разные гласные?

```
if not os.path.exists('87778__marcgascon7__vocals.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master
    /code/87778__marcgascon7__vocals.wav

wave = read_wave('87778__marcgascon7__vocals.wav')
wave.make_audio()

wave.make_spectrogram(1024).plot(high=1000)
```



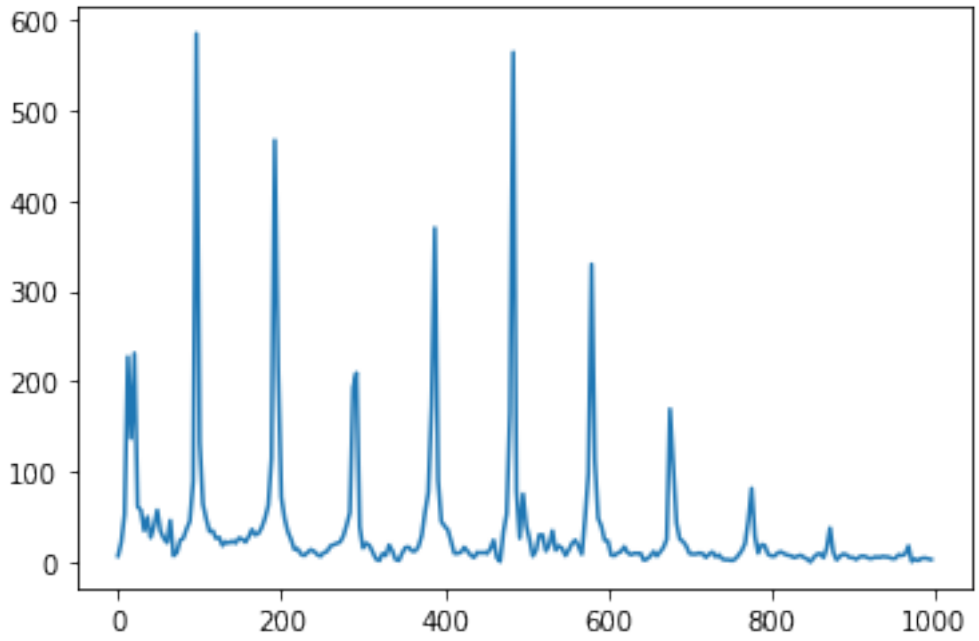
На спектограмме можно различить разные гласные по частоте соответствующего ей звука. Полоса внизу, вероятно, является фоновым шумом. Пики на спектрограмме называются формантами. Мы можем увидеть форматы более четко, выбрав сегмент во время «а».

```
high = 1000

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



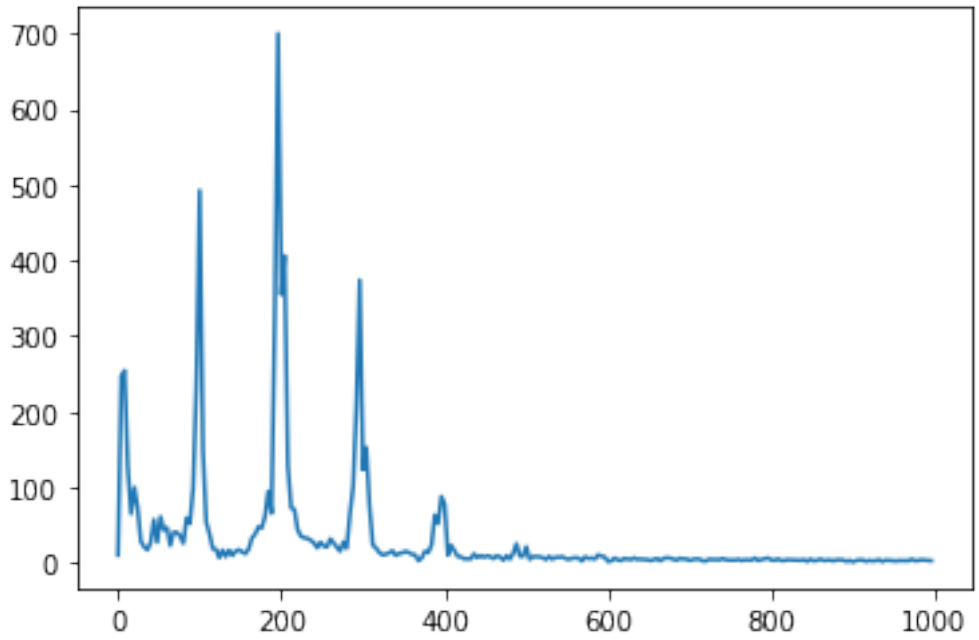
Основная частота около 100 Гц. Следующие самые высокие пики находятся на частотах 200 Гц и 700 Гц.
Сегмент «э» имеет высокоамплитудную форманту около 500 Гц.

```
segment = wave.segment(start=2.2, duration=0.25)
segment.make_spectrum().plot(high=high)
```



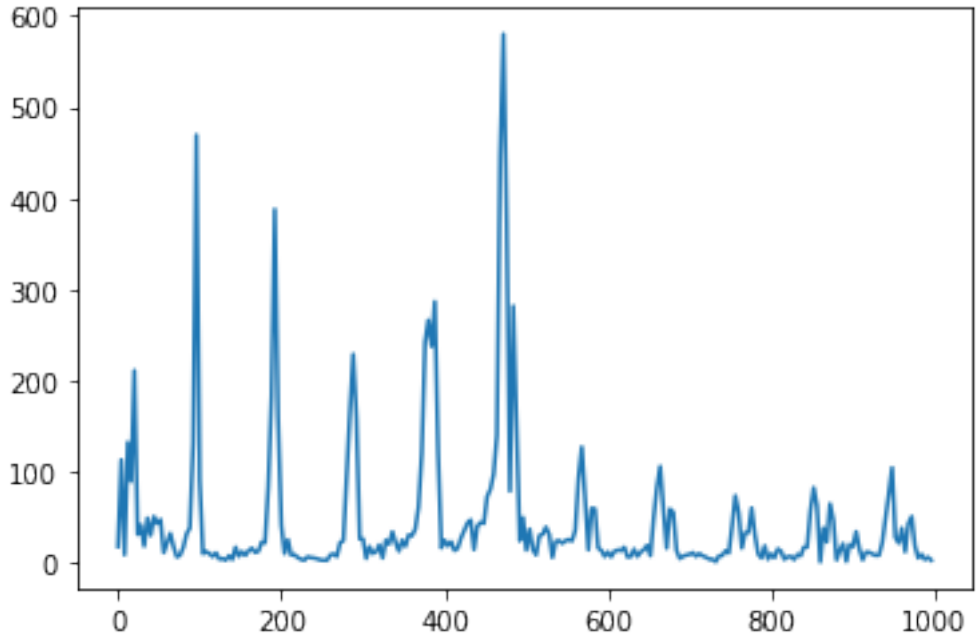
Сегмент «и» не имеет высокочастотных компонентов.

```
segment = wave.segment(start=3.5, duration=0.25)
segment.make_spectrum().plot(high=high)
```



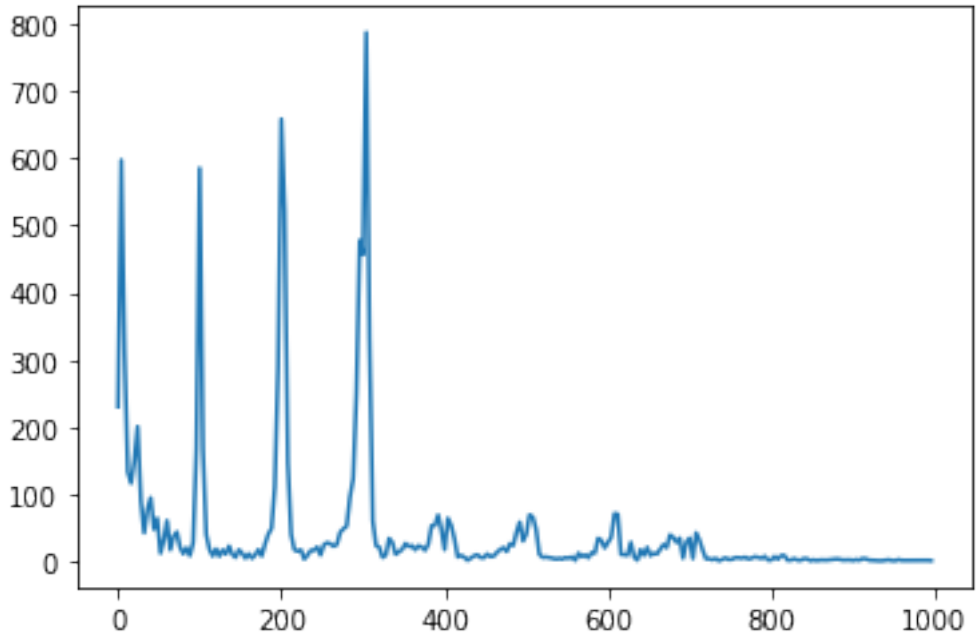
Сегмент «о» имеет высокоамплитудную форманту около 500 Гц, даже выше основной частоты.

```
segment = wave.segment(start=5.1, duration=0.25)
segment.make_spectrum().plot(high=high)
```



Сегмент «y» имеет высокоамплитудную форманту около 300 Гц и не содержит высокочастотных составляющих.

```
segment = wave.segment(start=6.5, duration=0.25)
segment.make_spectrum().
plot(high=high)
```



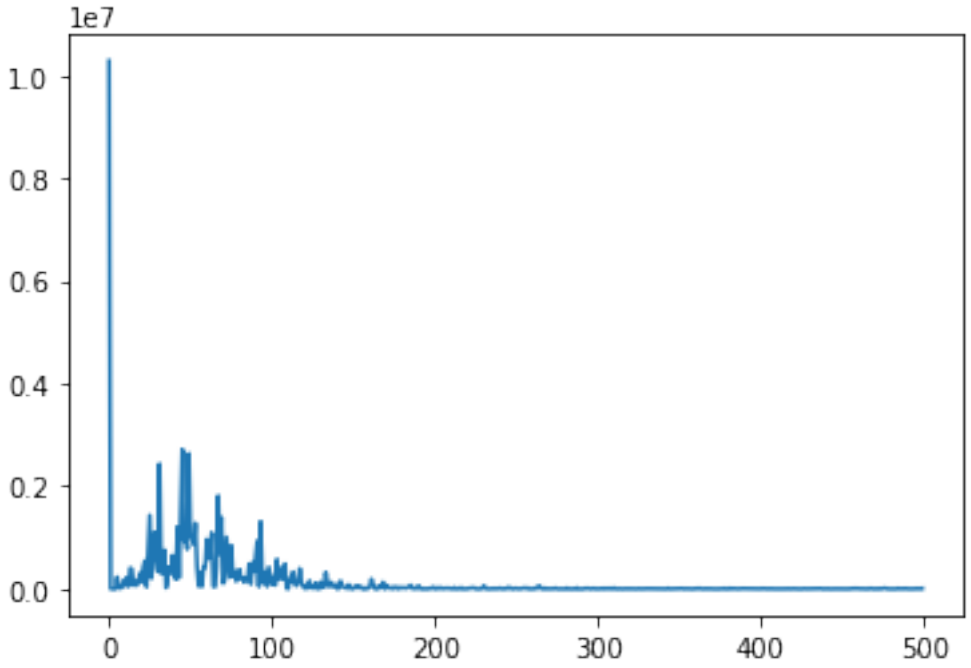
4. Шумы

4.1. Упражнение 1

```
import os

if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master
    /code/thinkdsp.py

import numpy as np
import matplotlib.pyplot as plt
import thinkdsp
import matplotlib.pyplot as plt
from thinkdsp import decorate
```

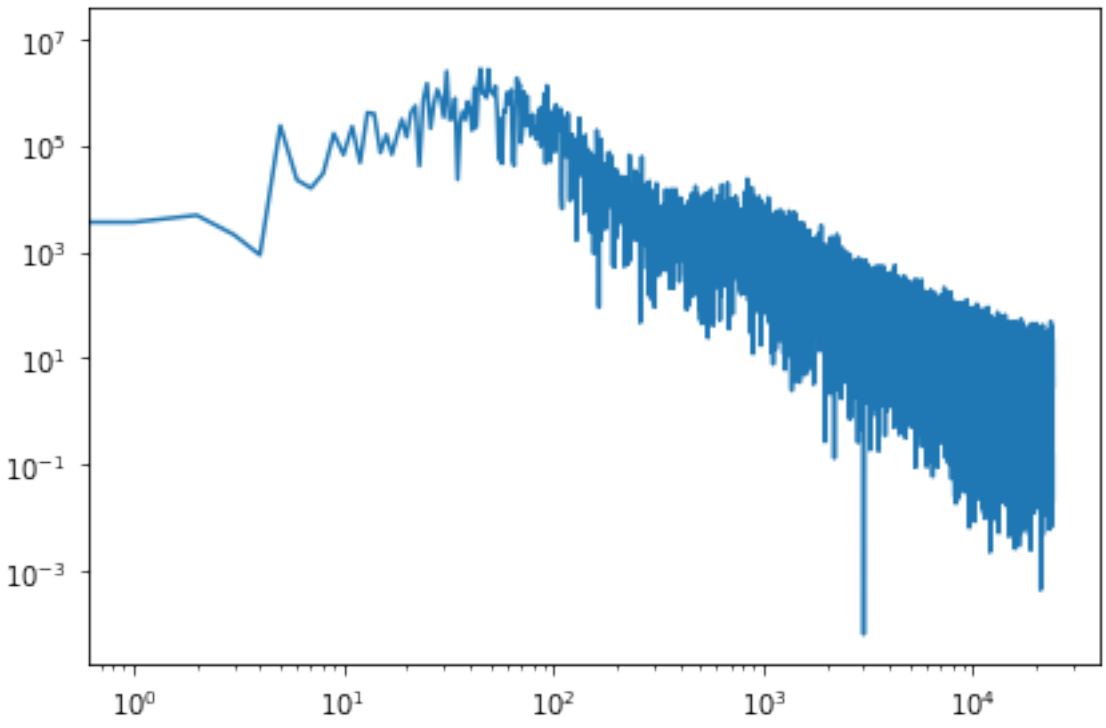
Скачайте некоторые из аудио файлов природных источников шума и вычислите спектры каждого сигнала. Похож ли их спектр мощности на белый, розовый или броуновский шум? Как спектр меняется во времени?
Возьмём звуки морских волн и ветра. Выделим из звуков короткие сегменты и построем спектры полученных сигналов.

```
if not os.path.exists('wind.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master/code/wind.wav
if not os.path.exists('waves.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master/code/waves.wav

wind_wave = thinkdsp.read_wave('wind.wav')
wind_wave.make_audio()

wind_wave = wind_wave.segment(start = 5, duration = 1)
wind_wave.make_audio()

spec = wind_wave.make_spectrum()
spec.plot_power(high = 500)
```



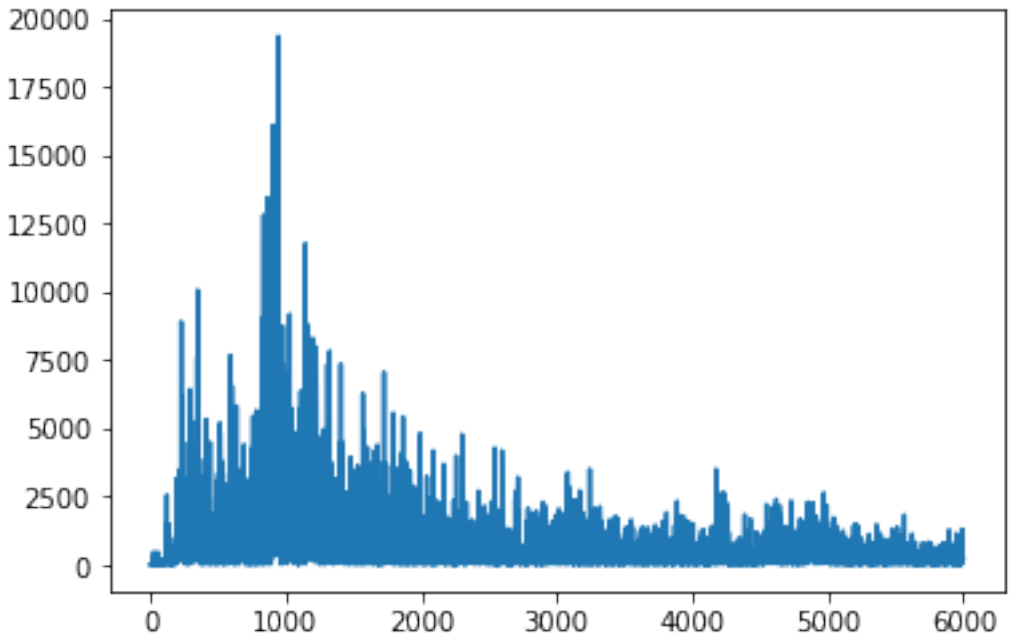
Зависимость падения амплитуды от частоты удаленно напоминает розовый или белый шум. Взглянем на спектр мощности в логарифмическом масштабе.

```
spec.plot_power()

loglog = dict(xscale='log', yscale='log')
decorate(**loglog)
```



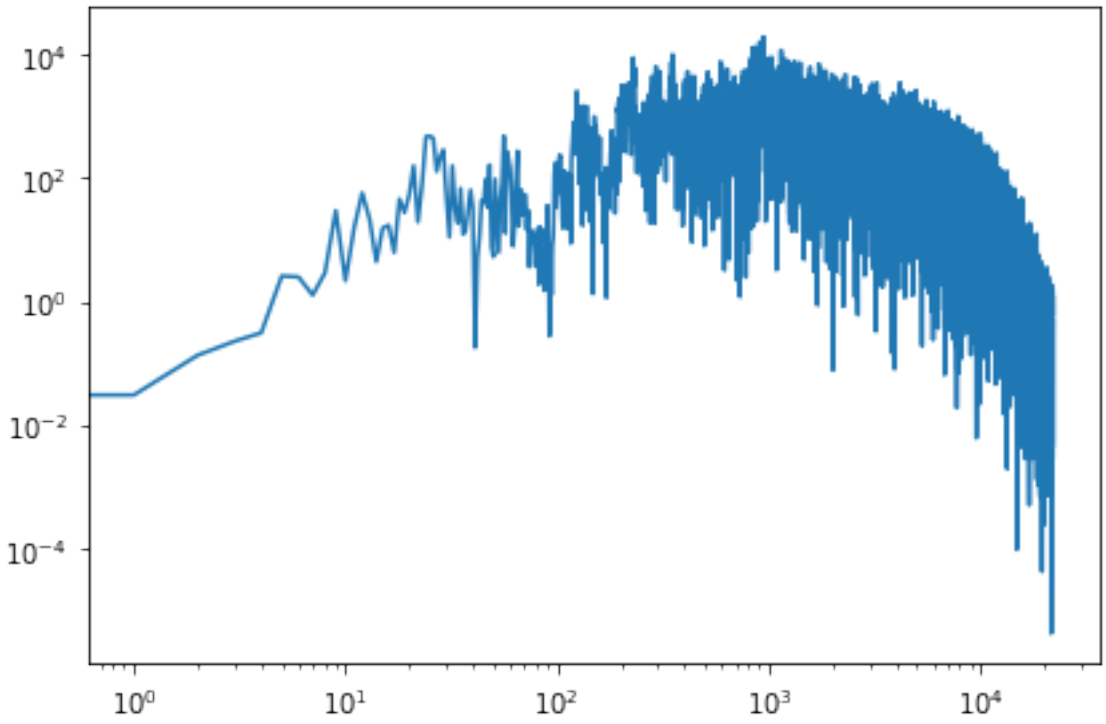
В начале график возрастает, а потом снова уменьшается.

```
sea_wave = thinkdsp.read_wave('waves.wav')
sea_wave.make_audio()

sea_wave = sea_wave.segment(start = 2, duration = 1)
sea_wave.make_audio()

spec2 = sea_wave.make_spectrum()
spec2.plot_power(high=6000)
```



Полученный график похож на график из предыдущего примера. Но зависимость падения более похожа на линейную. Как и в предыдущем примере, выведем график в логарифмическом масштабе.

```
spec2.plot_power()

loglog = dict(xscale='log', yscale='log')
decorate(**loglog)
```



4.2. Упражнение 2

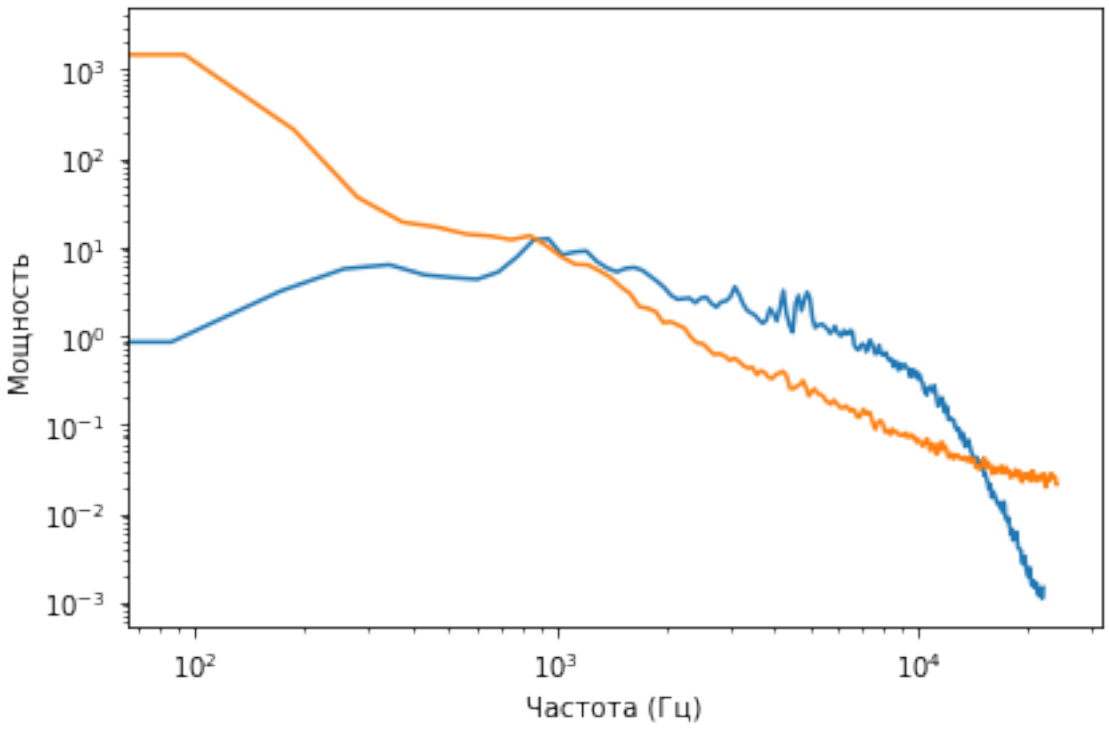
В шумовом сигнале частостный состав меняется во времени. На большой интервале мощность на всех частотах одинакова, а на коротком мощность на каждой частоте случайна. Реализуйте метод Бартлетта и исользуйте его для оценки спектра мощности шумового сигнала.
Создадим метод bartlett, который будет брать сигнал, разделять его на сегменты и вычислять спектр мощности для каждого сегмента и находить среднее по сегментам.

```
from thinkdsp import Spectrum

def bartlett_method(wave, seg_length=512, win_flag=True):

    sp = wave.make_spectrogram(seg_length, win_flag)
    specs = sp.spec_map.values()

    psds = [spectrum.power for spectrum in specs]
    hs = np.sqrt(sum(psds) / len(psds))
    fs = next(iter(specs)).fs

    spectrum = Spectrum(hs, fs, wave.framerate)
    return spectrum
```

Протестируем функцию на сигналах из первого задания.

```
sea = bartlett_method(sea_wave)
wind = bartlett_method(wind_wave)

sea.plot_power()
wind.plot_power()
```

```
decorate(xlabel='Частота␣Гц()',
         ylabel='Мощность',
         **loglog)
```



Исходя из графика, можно сказать, что второй сигнал (звуки моря) более похож на линейную зависимость, чем первый.

4.3. Упражнение 3

BTC.csv файл содержит исторические данные о ежедневной цене биткоина за последние пол-года. Откроем этот файл и вычислим спектр цен как функцию от времени.

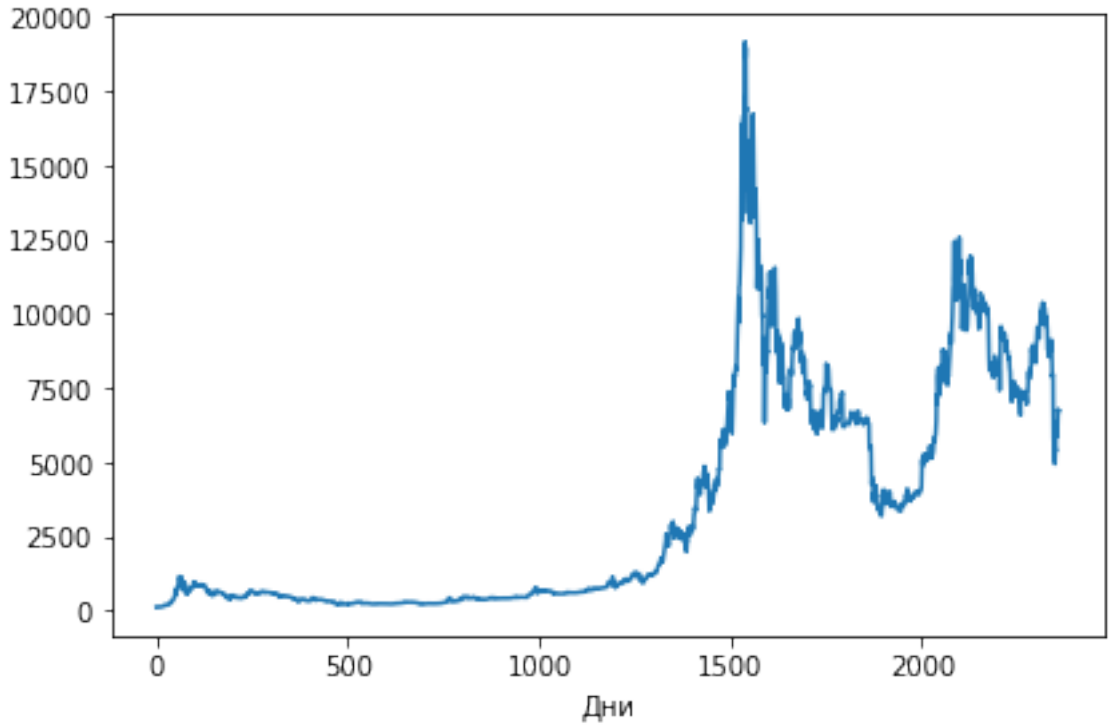
```
if not os.path.exists('BTC_USD_2013-10-01_2020-03-26-CoinDesk.csv'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/BTC_USD_2013-10-01_2020-03-26-CoinDesk.csv
```

```
import pandas as pd

df = pd.read_csv('BTC_USD_2013-10-01_2020-03-26-CoinDesk.csv',
                 parse_dates=[0])
df
```

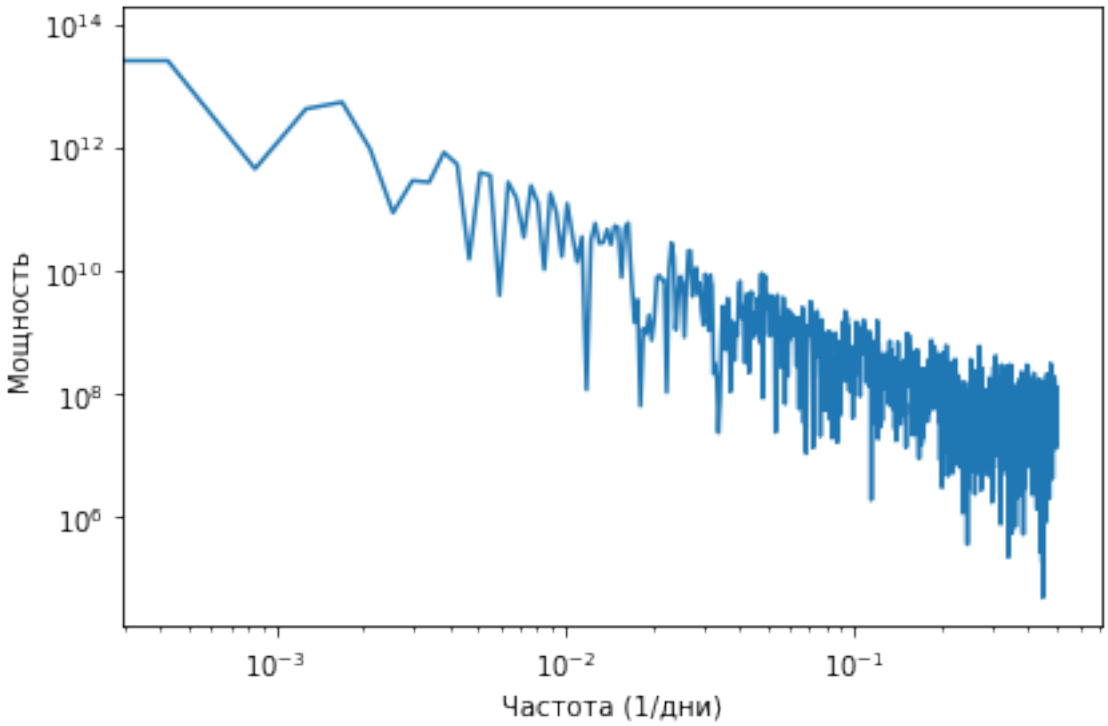
	Currency	Date	Closing Price (USD)	24h Open (USD)	24h High (USD)	24h Low (USD)
0	BTC	2013-10-01	123.654990	124.304660	124.751660	122.563490
1	BTC	2013-10-02	125.455000	123.654990	125.758500	123.633830
2	BTC	2013-10-03	108.584830	125.455000	125.665660	83.328330
3	BTC	2013-10-04	118.674660	108.584830	118.675000	107.058160
4	BTC	2013-10-05	121.338660	118.674660	121.936330	118.005660
...	...	...	...	...	...	...
2354	BTC	2020-03-22	5884.340133	6187.042146	6431.873162	5802.553402
2355	BTC	2020-03-23	6455.454688	5829.352511	6620.858253	5694.198299
2356	BTC	2020-03-24	6784.318011	6455.450650	6863.602196	6406.037439
2357	BTC	2020-03-25	6706.985089	6784.325204	6981.720386	6488.111885
2358	BTC	2020-03-26	6721.495392	6697.948320	6796.053701	6537.856462

2359 rows × 6 columns

```
from thinkdsp import Wave

ys = df['Closing Price (USD)']
ts = df.index

wave = thinkdsp.Wave(ys, ts, framerate=1)
wave.plot()
decorate(xlabel='Дни')
```




```
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Частота дни(1/)',
         ylabel='Мощность',
         **loglog)
```




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

```
-1.7332540936758951
```

Уклон спектра мощности составляет приблизительно -2, что близко к красному, броуновскому, шуму.

4.4. Упражнение 4

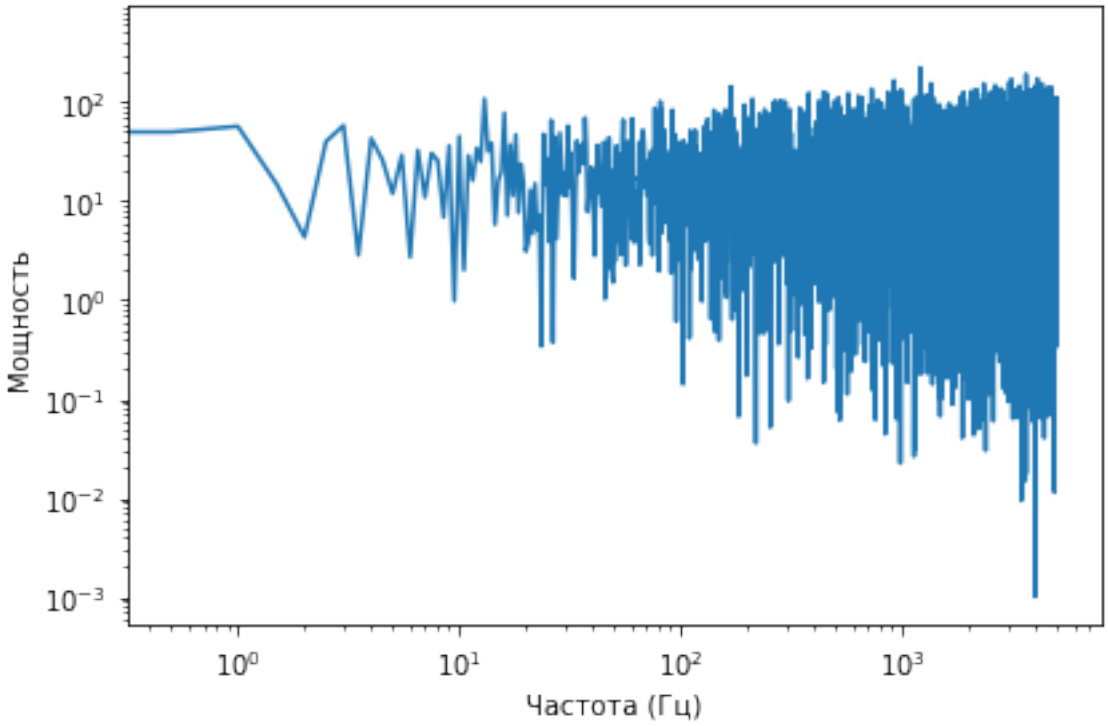
Счетчик Гейгера — это прибор, который регистрирует радиацию. Когда ионизирующая частица попадает на детектор, он генерирует всплеск тока. Общий вывод в определенный момент времени можно смоделировать как некоррелированный шум Пуассона (UP), где каждая выборка представляет собой случайную величину из распределения Пуассона, которая соответствует количеству частиц, обнаруженных в течение интервала.
Напишите класс с именем UncorrelatedPoissonNoise, который наследуется от Noise и предоставляет evaluate. Он должен использовать np.random.poisson для генерации случайных значений из распределения Пуассона. Параметр этой функции, lam, представляет собой среднее число частиц в течение каждого интервала. Вы можете использовать атрибут amp, чтобы указать lam. Например, если частота кадров равна 10 кГц, а amp равно 0,001, мы ожидаем около 10 «кликов» в секунду.

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

Сгенерируем сигнал с маленькой амплитудой (0.001) на основе этого класса. Ожидается услышать звук, как у счетчика Гейгера.

```
signal = UncorrelatedPoissonNoise(amp=0.001)
wave = signal.make_wave(duration=2, framerate=10000)
wave.make_audio()
```

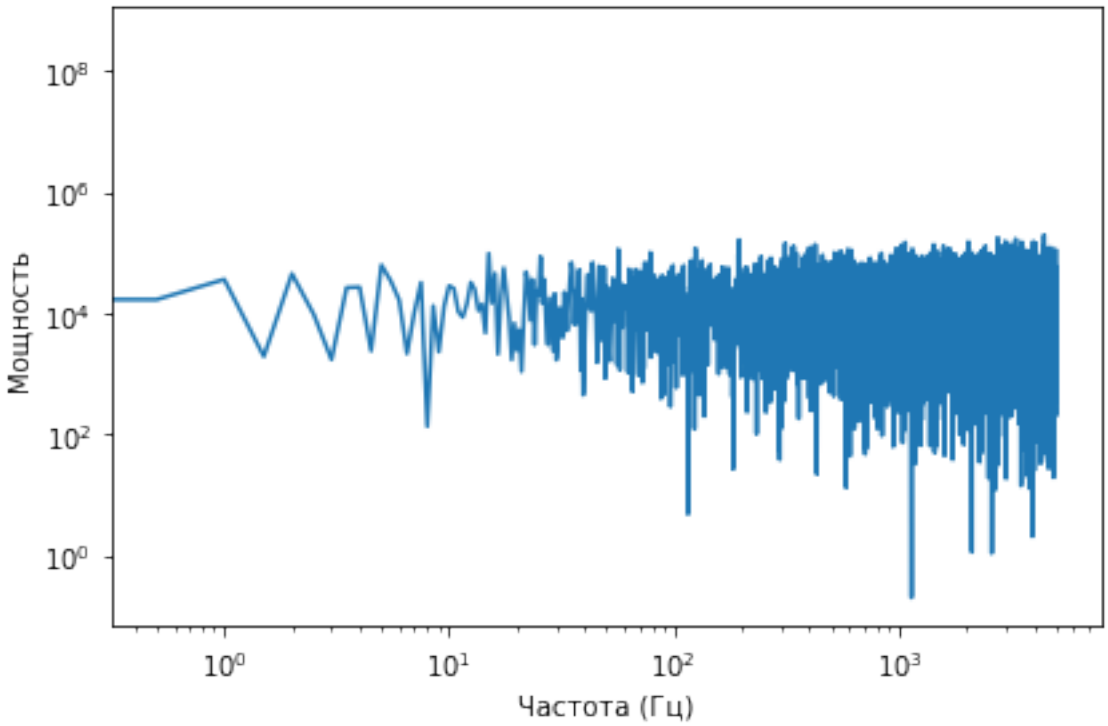
```
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Частота␣Гц()',
         ylabel='Мощность',
         **loglog)
```



Создадим такой же сигнал, но с большей амплитудой

```
signal = UncorrelatedPoissonNoise(amp=1)
wave = signal.make_wave(duration=2, framerate=10000)
wave.make_audio()
```

```
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Частота␣Гц()',
         ylabel='Мощность',
         **loglog)
```

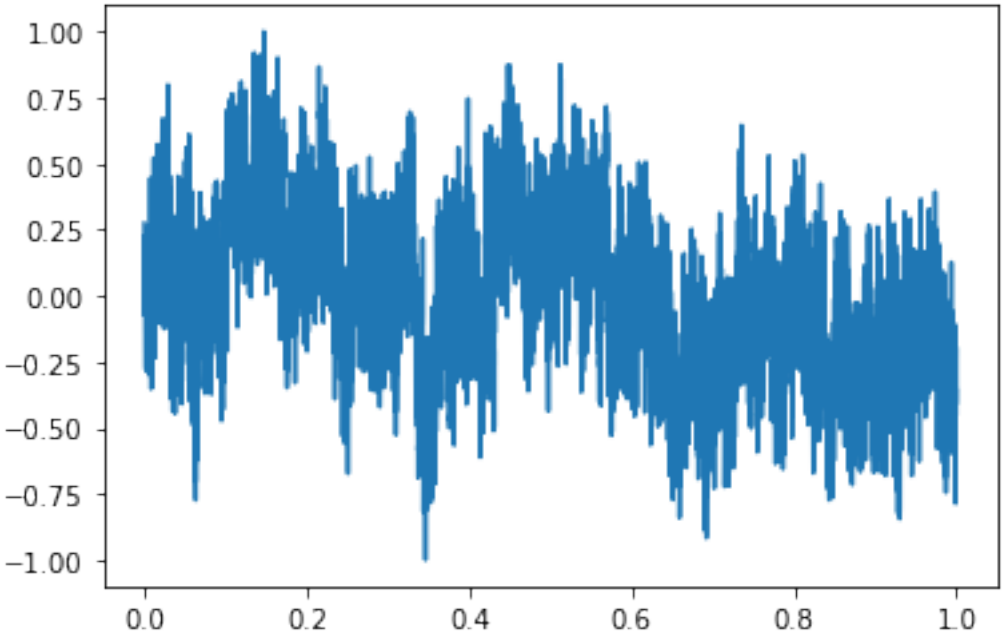


4.5. Упражнение 5

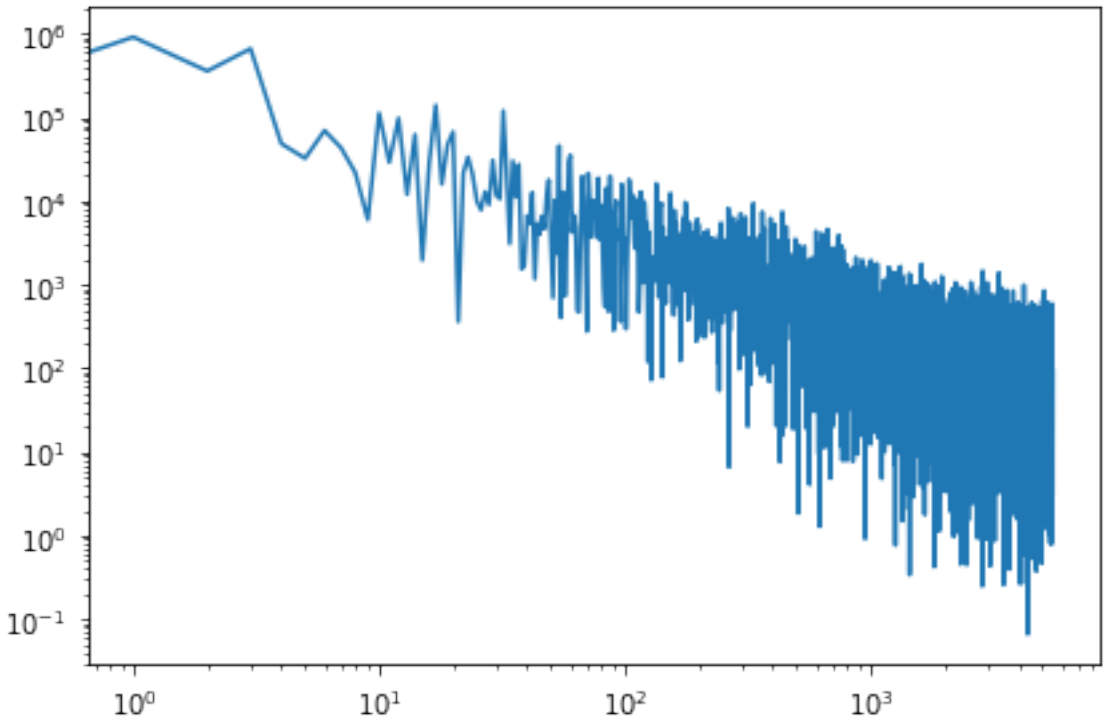
В этой главе алгоритм для генерации розового шума концептуально простой, но затратный. Существуют более эффективные варианты, например алгоритм Voss-McCartney. Изучите этот способ, реализуейте его, вычислите спектр результата и убедитесь, что соотношение между мощностью и частотой соответствующее.

```
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

wave = Wave(voss(11025))
wave.unbias()
wave.normalize()
wave.make_audio()
```

```
wave.plot()
```




```
spectrum = wave.make_spectrum()
spectrum.hs[0] = 0
spectrum.plot_power()
decorate(xscale='log', yscale='log')
```



Как можно видеть, соотношение между мощностью и частотой соответвствует розовому шуму.

```
spectrum.estimate_slope().slope
```

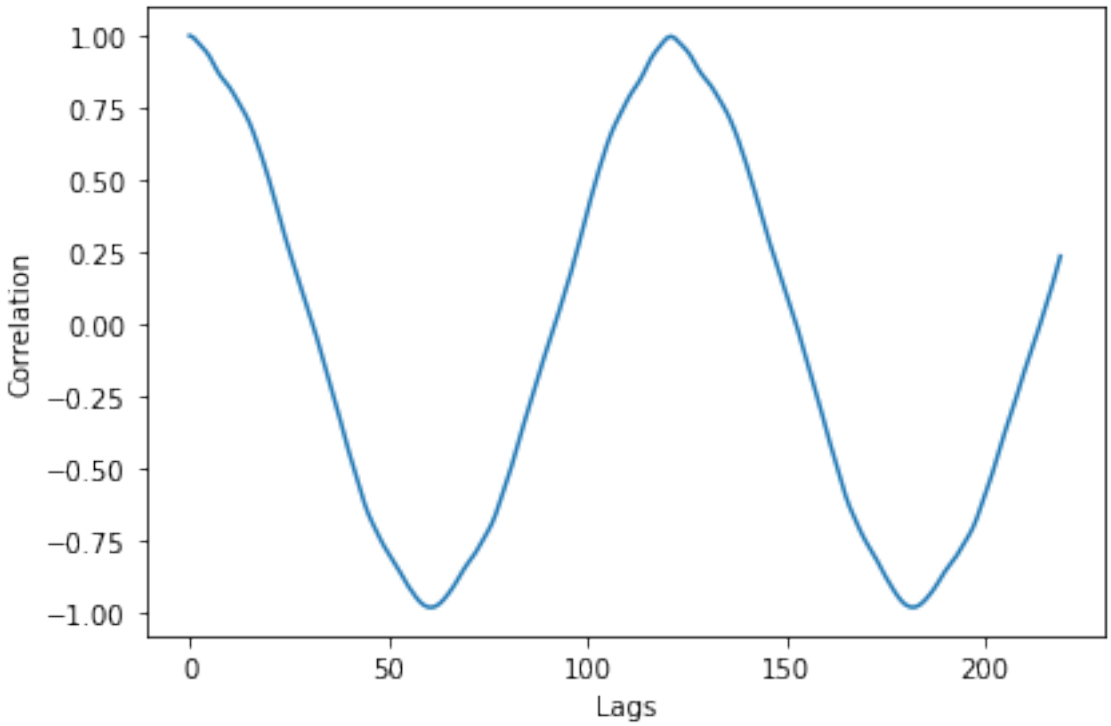
-0.9825046893659338

Расчетный наклон близок к -1.

5. Автокорреляция

5.1. Упражнение 1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from thinkdsp import decorate
from thinkdsp import read_wave

import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/thinkdsp.py

if not os.path.exists('28042__bcjordan__voicedownbew.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master
    /code/28042__bcjordan__voicedownbew.wav

def serial_corr(wave, lag=1):
    N = len(wave)
    y1 = wave.ys[lag:]
    y2 = wave.ys[:N-lag]
    corr = np.corrcoef(y1, y2)[0, 1]
    return corr

def autocorr(wave):
    """Computes and plots the autocorrelation function.

    wave: Wave

    returns: tuple of sequences (lags, corrs)
    """
    lags = np.arange(len(wave.ys)//2)
    corrs = [serial_corr(wave, lag) for lag in lags]
    return lags, corrs

wave = read_wave('28042__bcjordan__voicedownbew.wav')
wave.normalize()
wave.make_audio()

segment = wave.segment(start = 0.5, duration = 0.01)
lags, corrs = autocorr(segment)
plt.plot(lags, corrs)
decorate(xlabel = 'Lags', ylabel = 'Correlation')
```



Пик находится между lag = 100 и lag = 150:

```
np.array(corrs[100:150]).argmax() + 100
```

121

```
period = 121 / segment.framerate
frequency = 1 / period
frequency
```

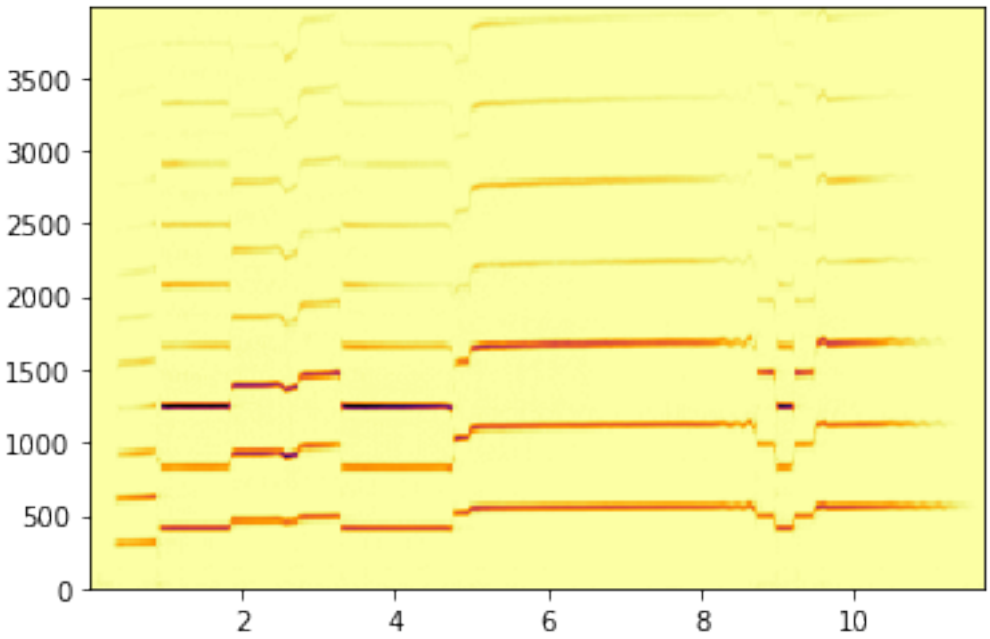
364.4628099173554

Высота тона вокального чирпа оказалась равной примерно 364 Гц.

5.2. Упражнение 2

Из примеров кода chap05.ipynb показано как использовать автокорреляцию для оценки основной частоты периодического сигнала. Инкапсулируем код в функцию estimateFundamental и используем ее для отслеживания высоты тона записанного звука. Для теста возьмем звук саксофона.

```
if not os.path.exists('sax.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master/code/sax.wav

import thinkdsp

wave = thinkdsp.read_wave('sax.wav')
wave.normalize()
wave.make_audio()

wave.make_spectrogram(2048).plot(high=4000)
```



Функция estimateFundamental из главы 5. Самый высоки пик в функции автокорреляции был отслежен с помощью выставления диапозана лагов для поиска от 70 до 200.

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

def serial_corr(wave, lag=1):
    n = len(wave)
    y1 = wave.ys[lag:]
    y2 = wave.ys[:n-lag]
    corr_mat = np.corrcoef(y1, y2)
    return corr_mat[0, 1]

def estimate_fundamental(segment, low=70, high=200):
    lags, corrs = autocorr(segment)
    lag = np.array(corrs[low:high]).argmax() + low
    period = lag / segment.framerate
    frequency = 1 / period
    return frequency

duration = 0.01
segment = wave.segment(start=0.2, duration=duration)
freq = estimate_fundamental(segment)
freq
```

518.8235294117648

В цикле отслеживается пик по всему звуку. Здесь ts — это середина каждого сегмента.

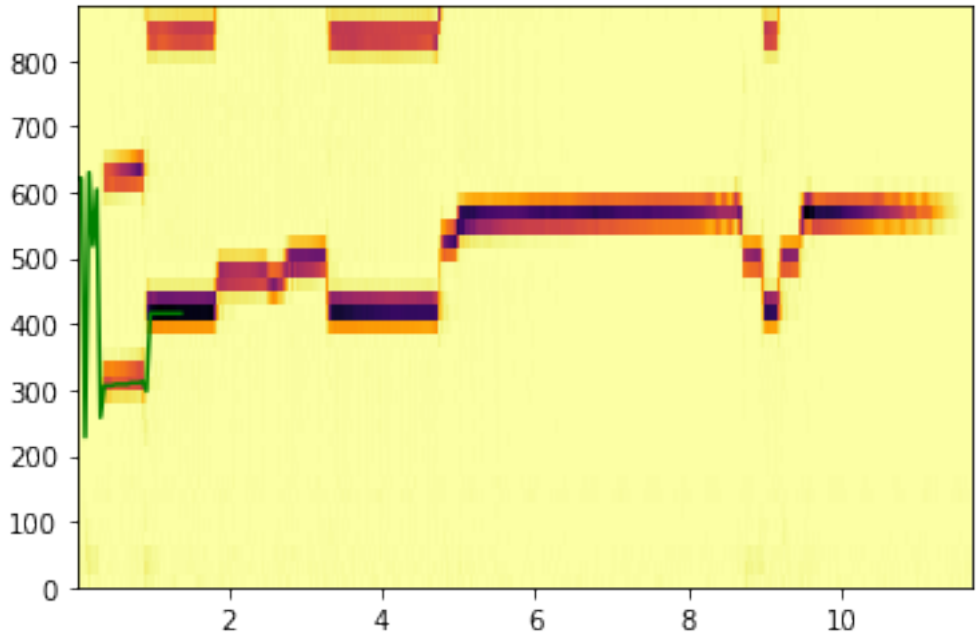
```
step = 0.05
starts = np.arange(0.0, 1.4, step)
```

```
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)
```

Зеленая линия на графике показвает отслеживание высоты тона, она наложена на спектрограмму.

```
wave.make_spectrogram(2048).plot(high=900)
plt.plot(ts, freqs, color='green')
```

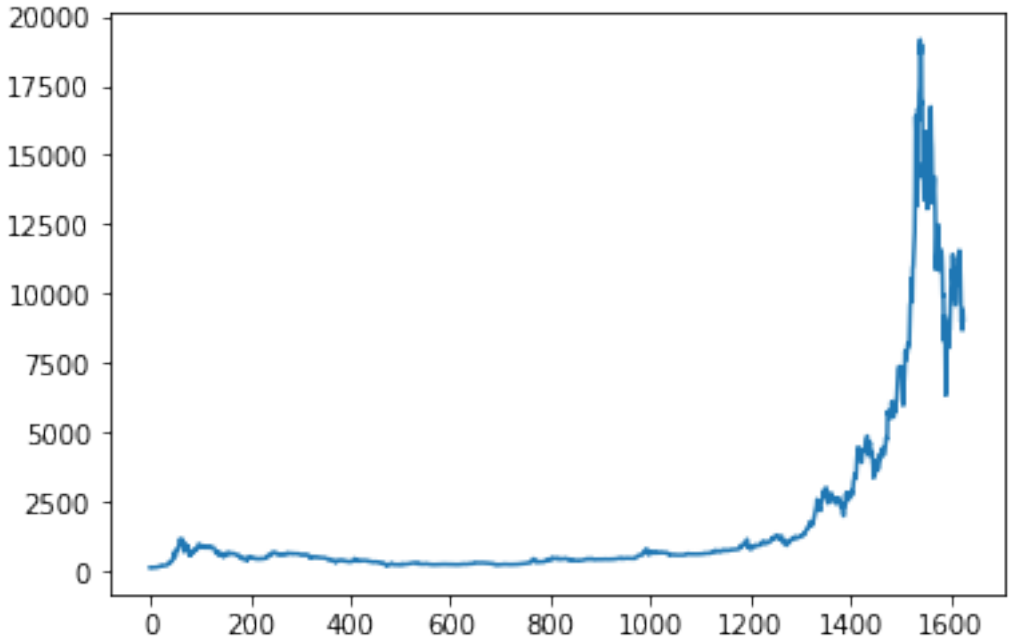
[<matplotlib.lines.Line2D at 0x7f5413a74850>]



5.3. Упражнение 3

Возьмем исторические данные Bitcoin за прошедшие полгода из прошлой лабораторной работы. Для этих данных вычислим автокорреляцию цен.

```
if not os.path.exists('BTC_USD_2013-10-01_2020-03-26-CoinDesk.csv'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master/code/BTC_USD_20

df = pd.read_csv('BTC_USD_2013-10-01_2020-03-26-CoinDesk.csv', nrows=1625,
ys = df['Closing Price (USD)']
ts = df.index
wave = thinkdsp.Wave(ys, ts, framerate=1)
wave.plot()
```




```
lags, corrs = autocorr(wave)
plt.plot(lags, corrs)
```

[<matplotlib.lines.Line2D at 0x7f540f512590>]

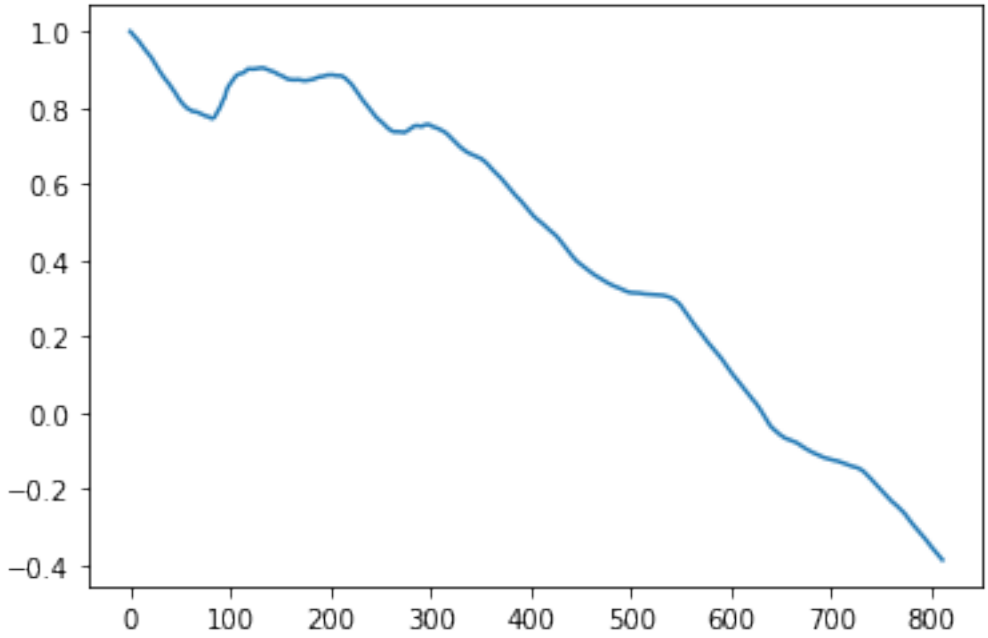
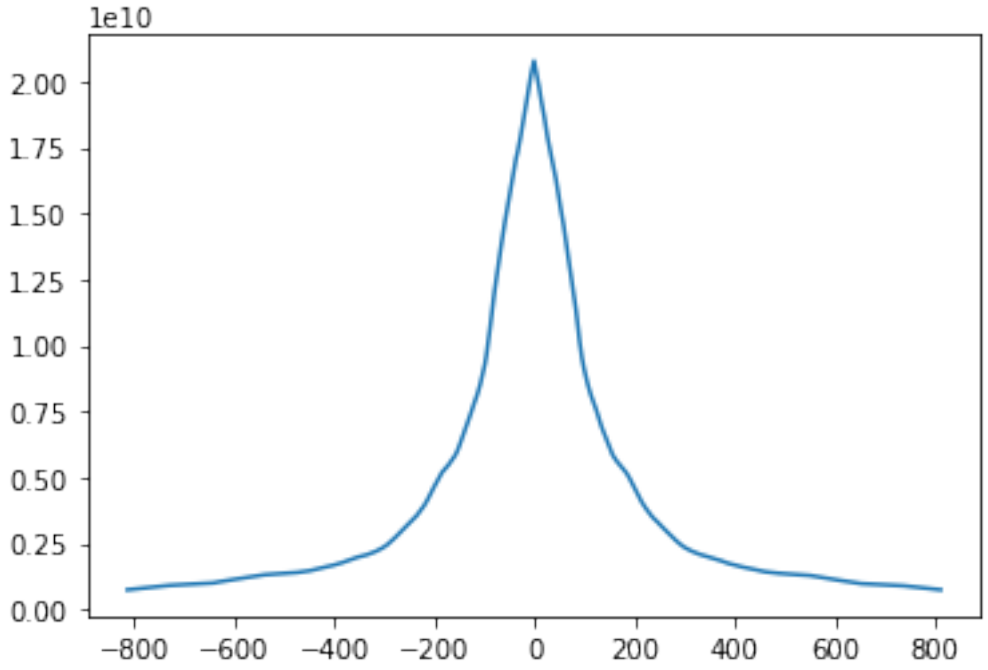


График медлено снажается, похож на розовый шум. Есть умеренная корреляция на 55 дне. Теперь вычислим корреляцию на основе функции np.correlate. Она не смещает и не нормализует волну.

```
N = len(wave)
corrs2 = np.correlate(wave.ys, wave.ys, mode='same')
lags = np.arange(-N//2, N//2)
plt.plot(lags, corrs2)
```

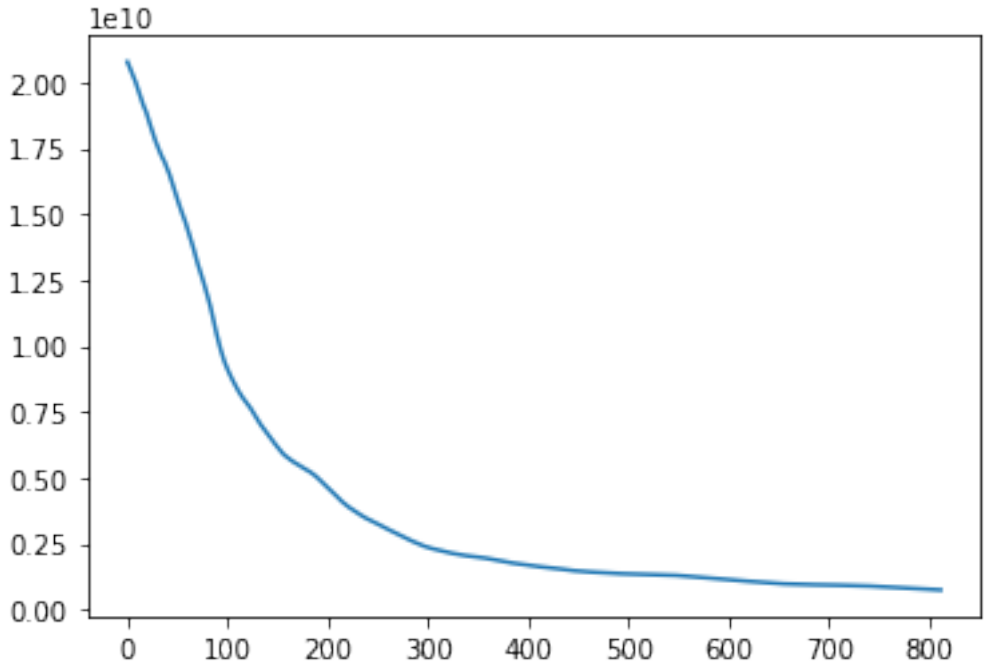
[<matplotlib.lines.Line2D at 0x7f540f48c710>]



Вторая половина результата соответствует положительным интервалам lags.

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

[<matplotlib.lines.Line2D at 0x7f540f4077d0>]



5.4. Упражнение 4

Прогнаны примеры из блокнота saxophone.ipynb.

6. Дискретное косинусное преобразование

6.1. Упражнение 1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/thinkdsp.py

from thinkdsp import decorate
```

Убедимся в том, что analyze1 требует времени пропорционально n^3 , а analyze2 пропорционально n^2 . Для этого будем запускать их с несколькими разными массивами и засекать время работы с помощью команды timeit.
Возьмем шумовой сигнал и массив со степенями двойки от 6 до 15.

```
from thinkdsp import UncorrelatedGaussianNoise

signal = UncorrelatedGaussianNoise()
noise = signal.make_wave(duration=1.0, framerate=8192)
noise.ys.shape
```

(8192,)

```
from scipy.stats import linregress

loglog = dict(xscale='log', yscale='log')

def plot_bests(ns, bests):
    plt.plot(ns, bests)
    decorate(**loglog)

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

    return slope

PI2 = np.pi * 2

def analyze1(ys, fs, ts):

    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.linalg.solve(M, ys)
    return amps
```

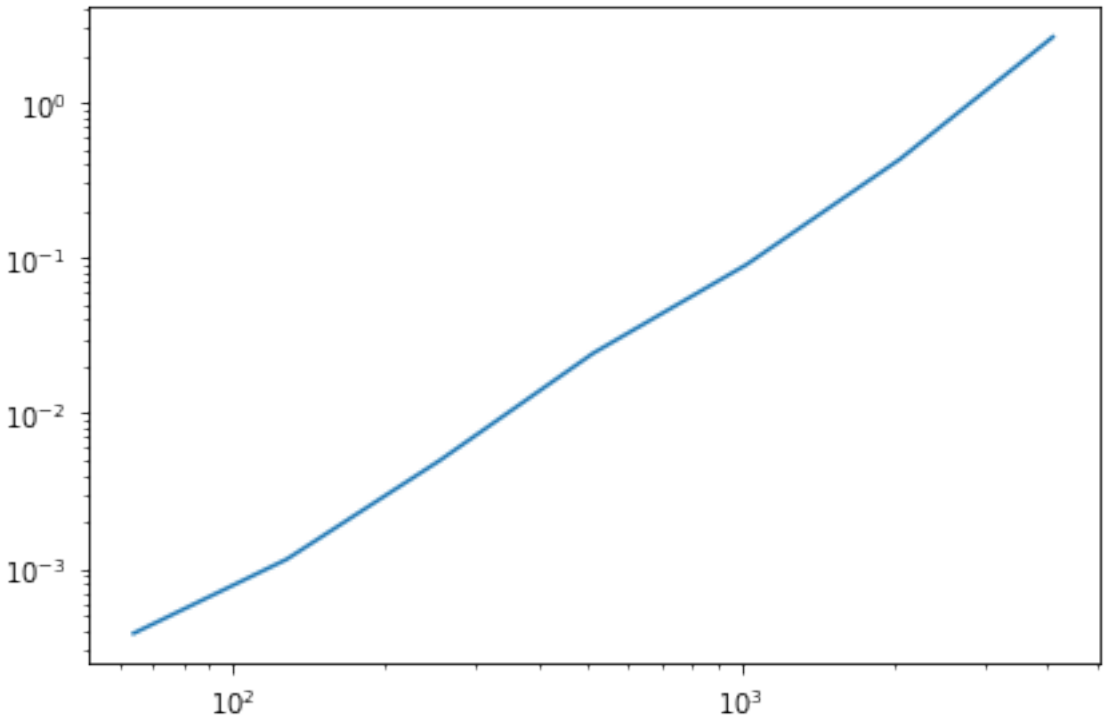
Результаты для analyze1.

```
ns = 2 ** np.arange(6, 13)
ns
```

```
array([  64,  128,  256,  512, 1024, 2048, 4096])
```

```
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 analyze1(ys, freqs, ts)
  results.append(result)

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

```
64
384 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
128
1.15 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
256
4.98 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
512
24.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
1024
90.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2048
430 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
4096
2.63 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
2.1252061541805842
```



Уклон выходит чуть больше 2, что не сходится с ожидаемым в пособии, в котором говорится об уклоне 3, что, вероятно, связано с тем, что массив слишком мал для того, чтобы требовалось пропорциональное n^3 время.
Проведем такой же эксперимент для analyze2

```
def analyze2(ys, fs, ts):

    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.dot(M, ys) / 2
    return amps
```

```
signal = UncorrelatedGaussianNoise()
noise = signal.make_wave(duration=1.0, framerate=16384)
noise.ys.shape
```

```
ns = 2 ** np.arange(6, 15)
ns
```

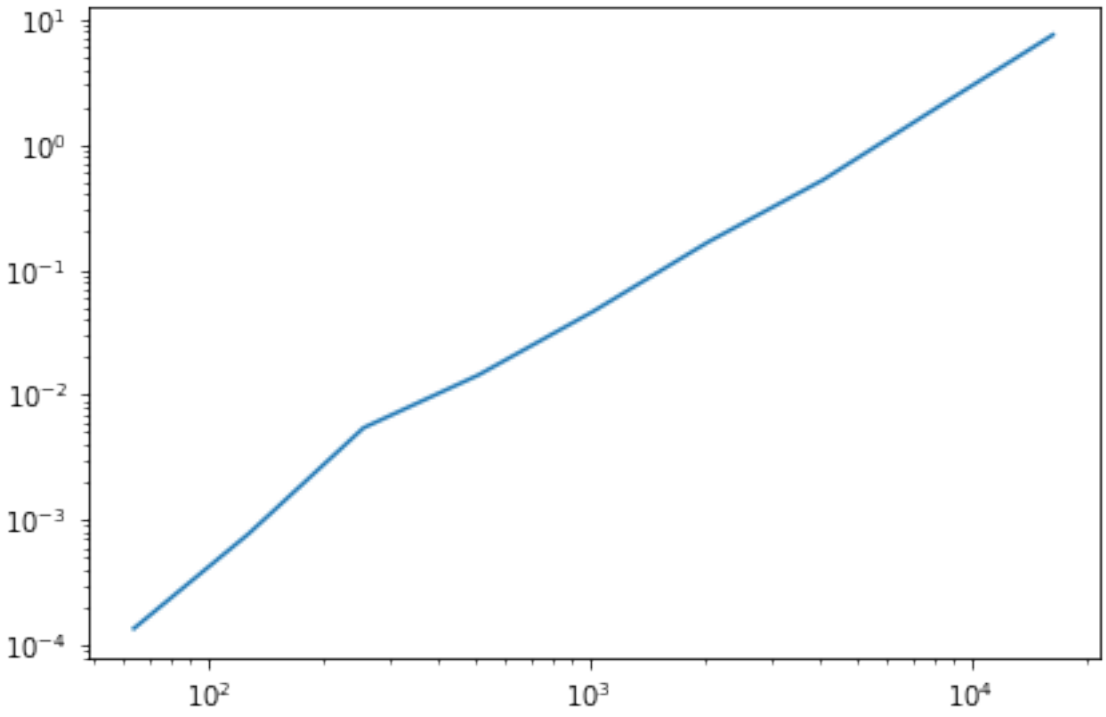
```
array([ 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384])
```

```
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 analyze2(ys, freqs, ts)
  results.append(result)
```

```
bests2 = [result.best for result in results]
```

```
plot_bests(ns, bests2)
```

```
64
135 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
128
774 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
256
5.44 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
512
14.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
1024
46.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2048
166 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
4096
522 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
8192
2 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
16384
7.53 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
1.8962250290107265
```

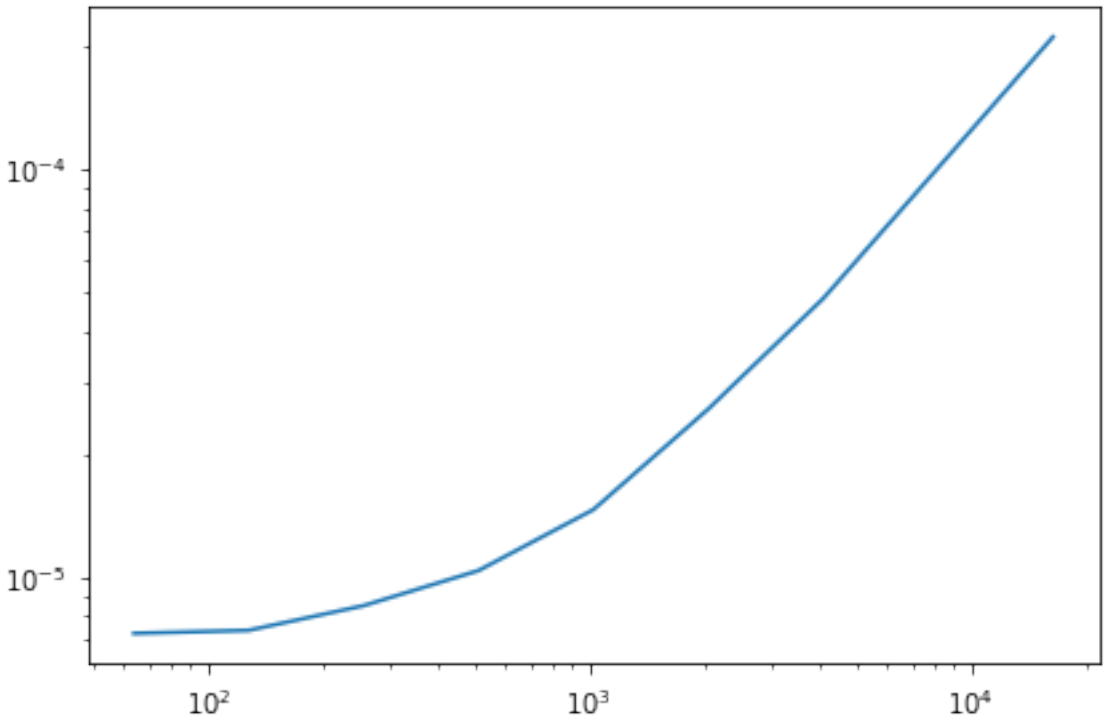


Как и ожидалось, результаты для analyze2 укладываются в прямую линию с расчетным наклоном, близким к 2.
Проведем такой же эксперимент scipy.fftpack.dct.

```
import scipy.fftpack

def scipy_dct(ys, freqs, ts):
    return scipy.fftpack.dct(ys, type=3)
```

```
results = []
for N in ns:
  ys = noise.ys[:N]
  result = %timeit -r1 -o scipy.fftpack.dct(ys, type=3)
  results.append(result)

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

```
7.27 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
7.39 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
8.49 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
10.4 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
14.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
25.7 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
47.9 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
100 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
209 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
0.6164230151411354
```



dct реализация намного быстрее, она пропорциональна nlogn , но на графике этого не видно.

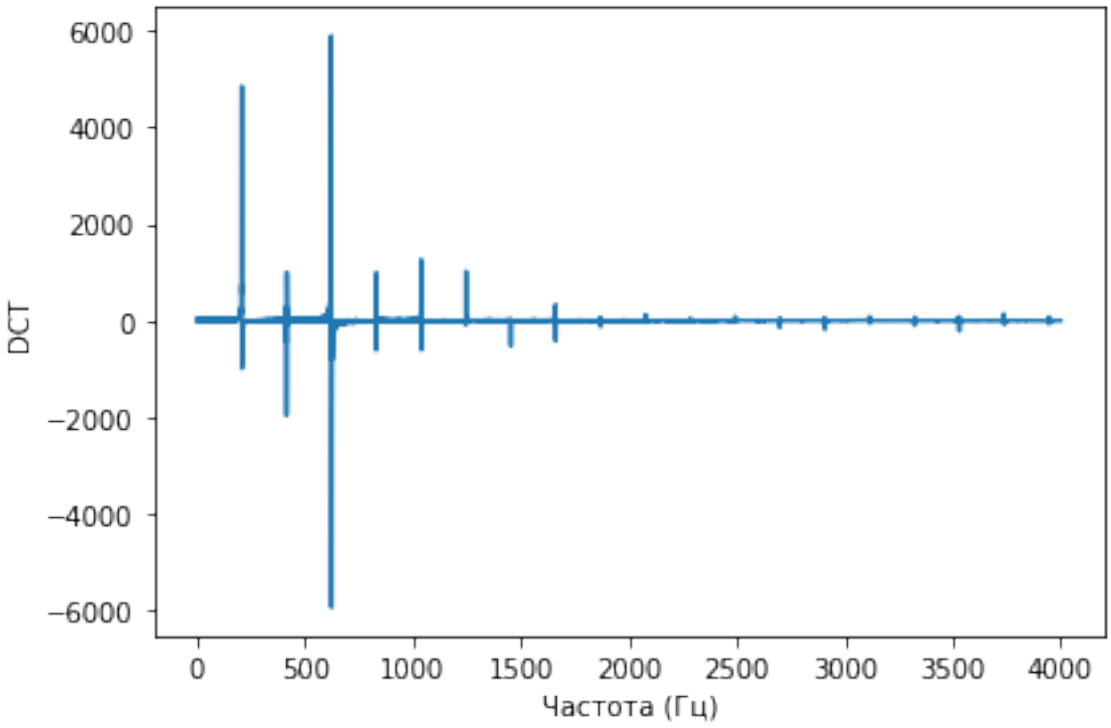
6.2. Упражнение 2

Необходимо реализовать алгоритм ДКП для сжатия звука и изображений. В качестве сжимаемого файла возьмем звук трубы. Выделим из него короткий сегмент.

```
if not os.path.exists('saxop.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master/code/saxop.wav

from thinkdsp import read_wave

wave = read_wave('saxop.wav')
wave.make_audio()

segment = wave.segment(start=1.2, duration=0.5)
segment.normalize()
segment.make_audio()
```

DCT график для полученного сегмента:

```
seg_dct = segment.make_dct()
seg_dct.plot(high=4000)
decorate(xlabel='Частота␣Гц()', ylabel='DCT')
```



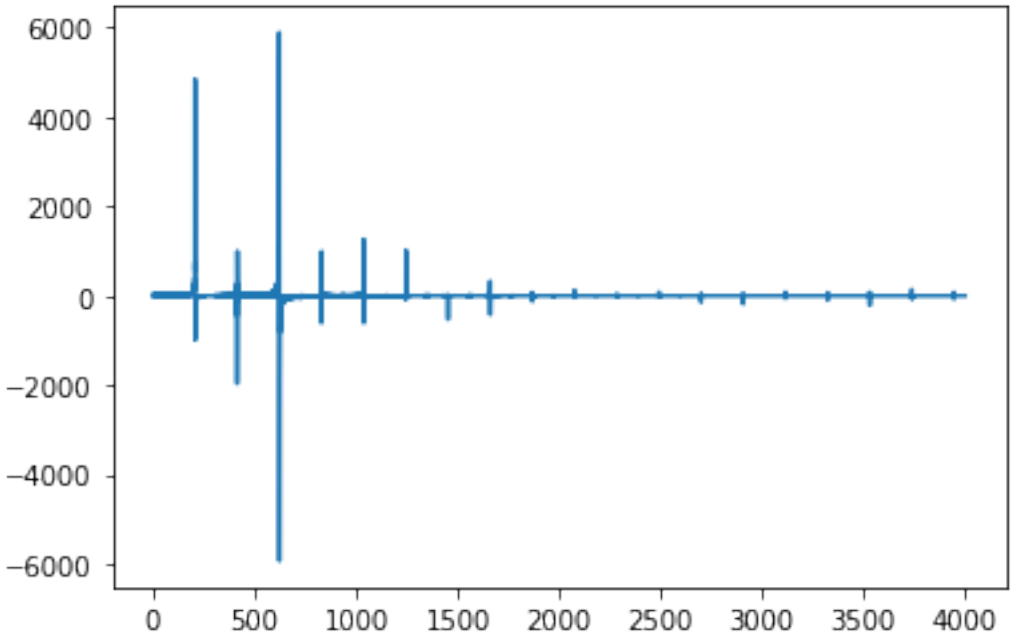
Из графика видно, что есть немного частот с большой амплитудой. Следующая функция compress и режет элемент, которые ниже аргумента thresh.

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

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

Теперь применим функцию для нашего сегмента:

```
seg_dct = segment.make_dct()
compress(seg_dct, thresh=10)
seg_dct.plot(high=4000)
```

20457 22050 92.77551020408163



Звучание обработанного сигнала:

```
seg2 = seg_dct.make_wave()
seg2.make_audio()
```

6.3. Упражнение 3

Прогнаны примеры из блокнота phase.ipynb.

7. Дискретное преобразование Фурье

7.1. Упражнение 1

```
import os

if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/thinkdsp.

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

PI2 = np.pi * 2
```

Прогнаны примеры из блокнота chap07.ipynb.

7.2. Упражнение 2

Для реализации быстрого преобразования Фурье нужно разделить массив на чётные (e) и нечётные элементы (o); вычислить DFT e и o, делая рекурсивные вызовы; вычислить DFT для каждого значения n, используя Лемму Дэниеолсона-Ланцоша.
Для теста начнем с небольшого реального сигнала и вычислим его БПФ:

```
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]
```

Реализация ДПФ из книги:

```
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
```

Можем подтвердить, что эта реализация дает тот же результат.

```
hs2 = dft(ys)
np.sum(np.abs(hs - hs2))
```

```
5.864775846765962e-16
```

В качестве шага к созданию рекурсивного БПФ начнем с версии, которая разбивает входной массив и использует np.fft.fft для вычисления БПФ половин

```
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)
```

Получаем те же результаты:

```
hs3 = fft_norec(ys)
np.sum(np.abs(hs - hs3))
```

0.0

Теперь мы можем заменить np.fft.fft рекурсивными вызовами и добавить базовый вариант:

```
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)
```

Получаем те же результаты:

```
hs4 = fft(ys)
np.sum(np.abs(hs - hs4))
```

1.6653345369377348e-16

8. Фильтрация и свертка

8.1. Упражнение 1

```
import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/thinkdsp.py

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.signal


from thinkdsp import decorate
```

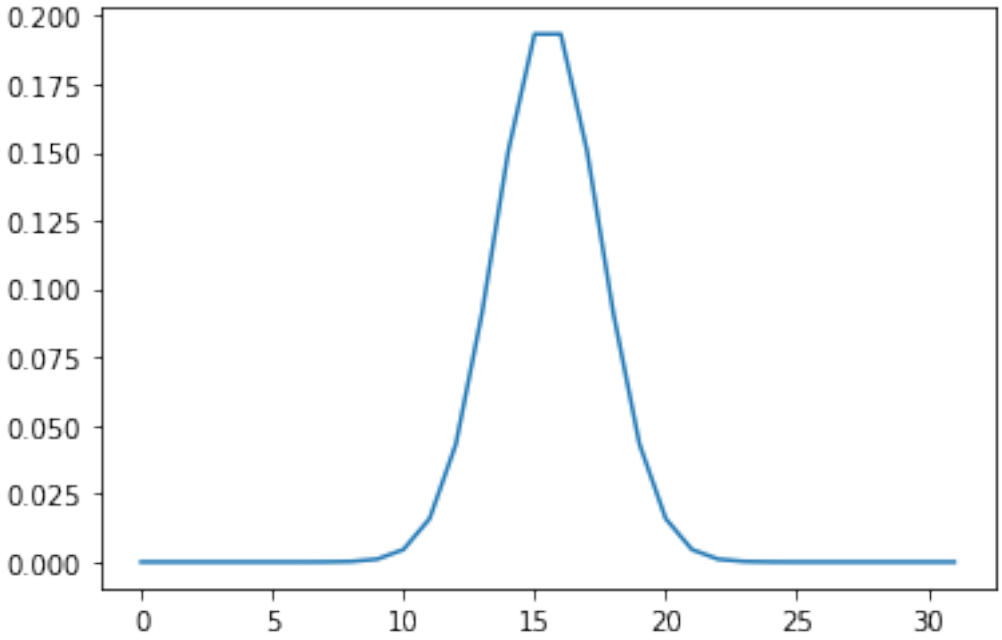
Прогнаны примеры из блокнота chap08.ipynb. В интерактивном виджете есои при увеличении ширины гауссового окна std не увеличивать число элементов в окне M, то на графике возникают лепестки.

8.2. Упражнение 2

В главе утверждается, что преобразование Фурье гауссовой кривой - тоже гауссова кривая. Опробуем это соотношение на нескольких примерах. Начнем с кривой Гауса:

```
g = scipy.signal.gaussian(M=32, std=2)
g /= sum(g)
plt.plot(g)
```

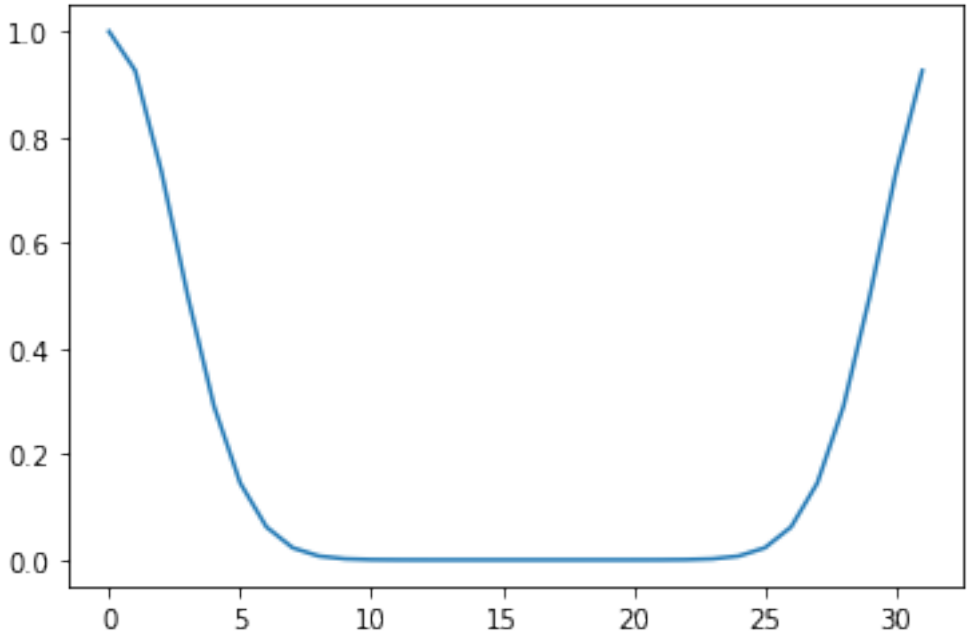
[<matplotlib.lines.Line2D at 0x7f8a4191e290>]



Быстрое преобразование Фурье для гауссовой кривой:

```
fft_g = np.fft.fft(g)
plt.plot(abs(fft_g))
```

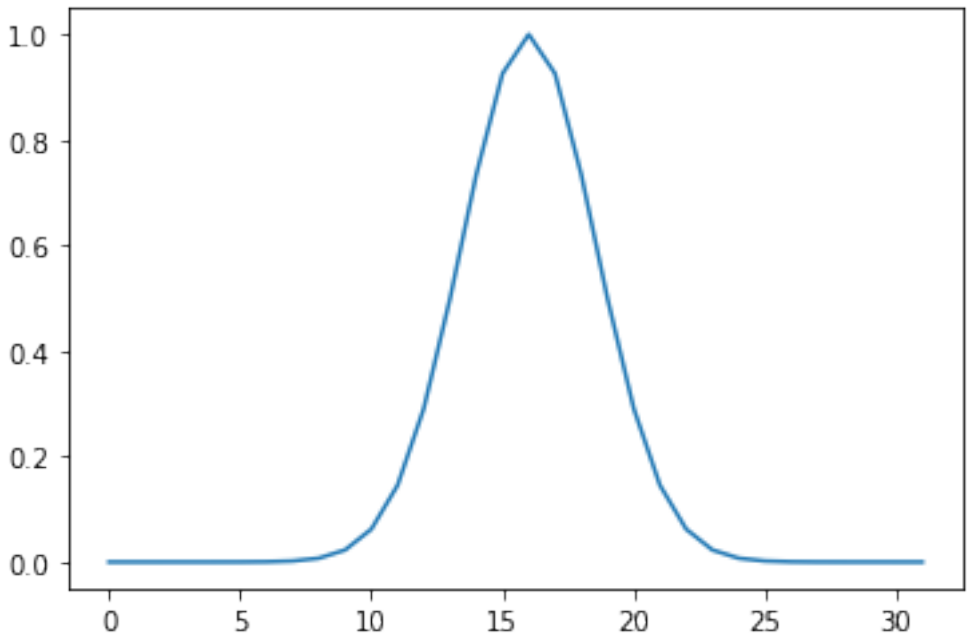
[<matplotlib.lines.Line2D at 0x7f8a41ac0750>]



Произведем свертку отрицательных частов влево. Результат приблизительно похож на гауссову кривую.

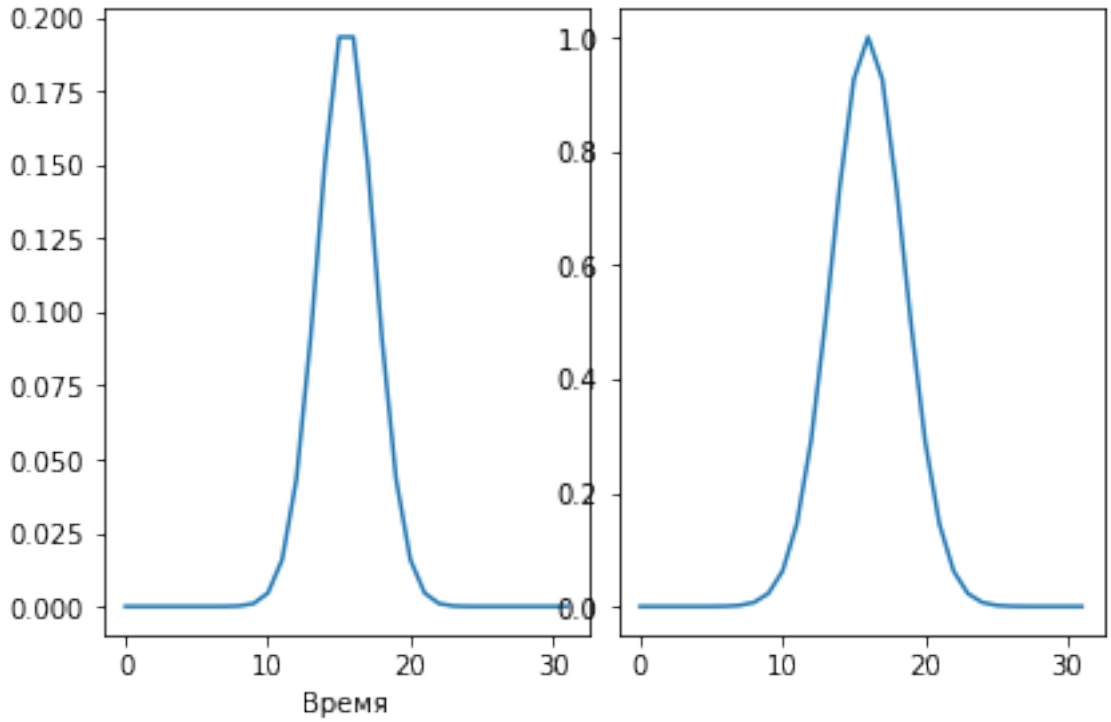
```
N = len(g)
fft_rolled = np.roll(fft_g, N//2)
plt.plot(abs(fft_rolled))
```

[<matplotlib.lines.Line2D at 0x7f8a419ccd10>]



Это функция была взята из chap08sol, она строит гауссово окно и БПФ рядом.

```
def plot_gaussian(std):
    M = 32
    gaussian = scipy.signal.gaussian(M=M, std=std)
    gaussian /= sum(g)

    plt.subplot(1, 2, 1)
    plt.plot(g)
    decorate(xlabel='Время')

    fft_g = np.fft.fft(g)
    fft_rolled = np.roll(fft_g, M//2)

    plt.subplot(1, 2, 2)
    plt.plot(np.abs(fft_rolled))
    plt.show()

plot_gaussian(2)
```




```
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);
```

interactive(children=(FloatSlider(value=2.0, description='std', max=10.0, min=0.1), Output()), domClasses=('...

С увеличением std, гауссова кривая становится шире, а БПФ - уже.

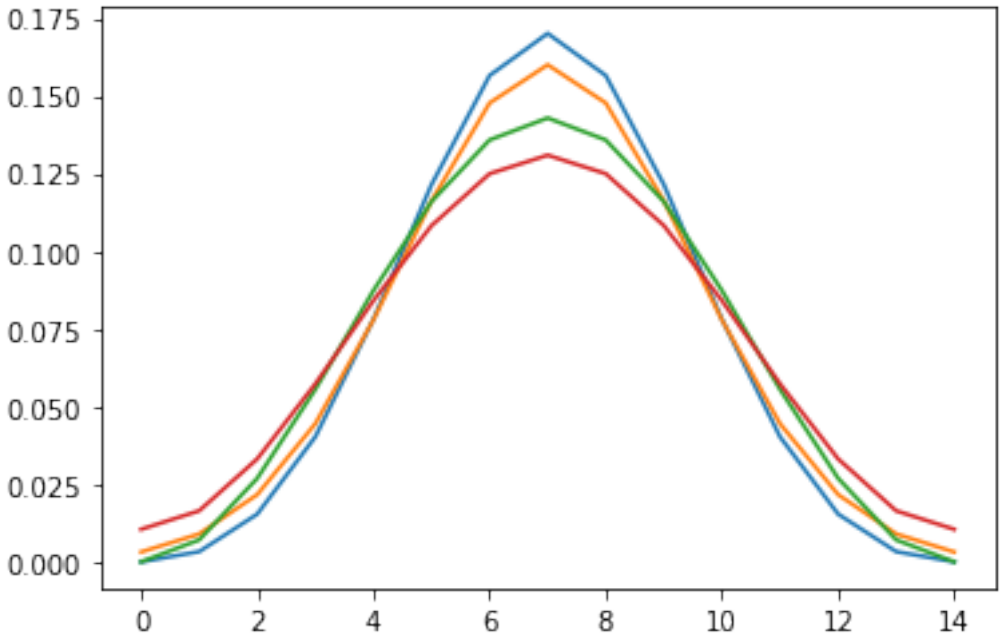
8.3. Упражнение 3

По примерам из главы, создадим 1-секундную волну с частотой дискретизации 44,1 кГц.

```
from thinkdsp import SquareSignal

sig = SquareSignal(freq=440)
wave = sig.make_wave(duration=1.0, framerate=44100)
```

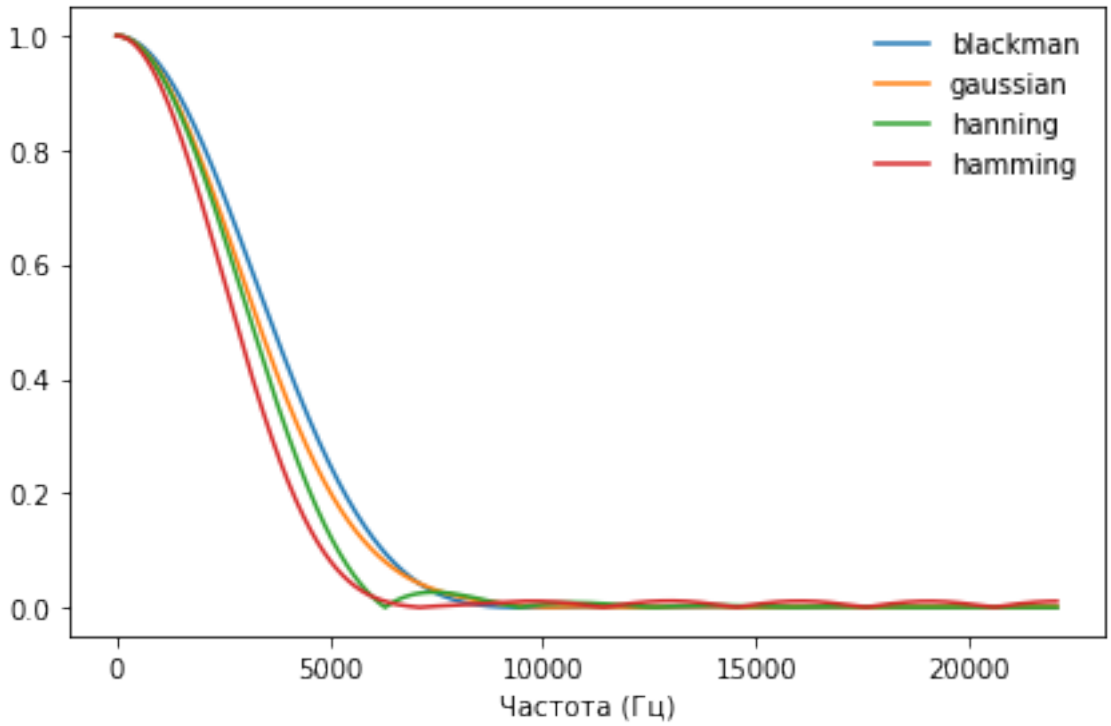
Создадим несколько окон. Выбираем стандартное отклонение окна Гаусса, чтобы сделать его похожим на другие.

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

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

for window, name in zip(windows, names):
    plt.plot(window, label=name)
```



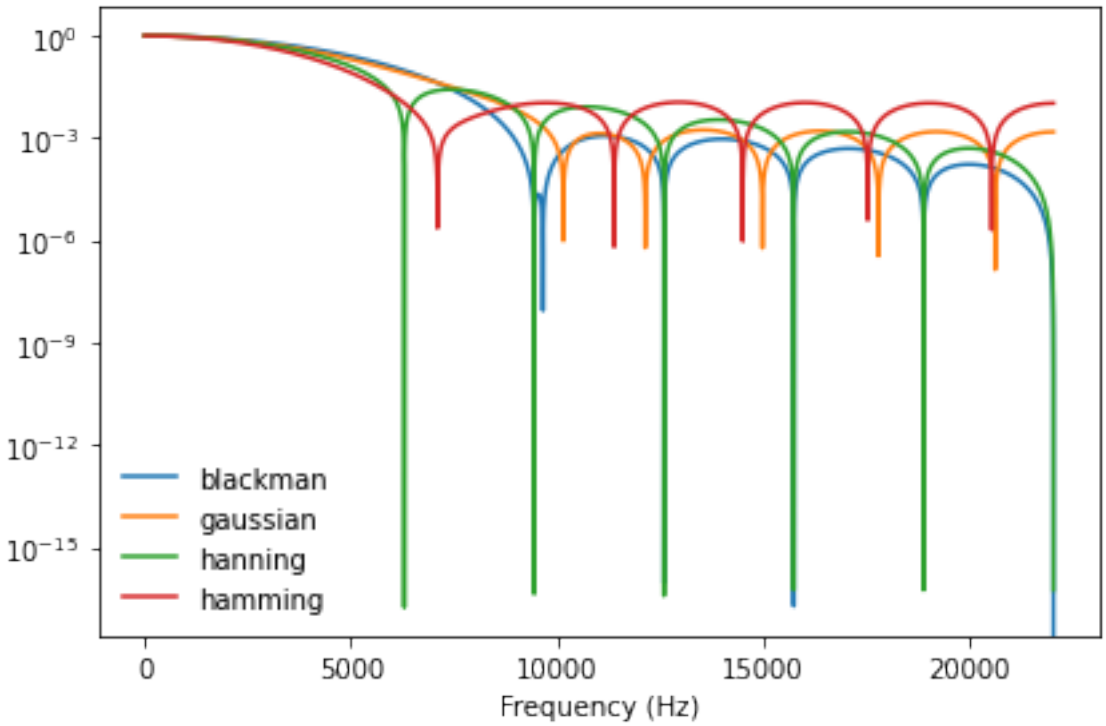
Графики выглядят очень похоже. Построим графики дискретного преобразования Фурье.

```
def zero_pad(array, n):

    res = np.zeros(n)
    res[:len(array)] = array
    return res

def plot_window_dfts(windows, names):

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

plot_window_dfts(windows, names)
decorate(xlabel='Частота␣Гц()')
```



Хэмминг падает быстрее всего, Блэкман — медленнее, а Хэннинг имеет самые заметные боковые лепестки.

```
plot_window_dfts(windows, names)
decorate(xlabel='Frequency␣(Hz)', yscale='log')
```



На логарифмической шкале мы видим, что показатели Хэмминга и Ханнинга поначалу снижаются быстрее, чем у двух других. И окна Хэмминга и Гаусса, кажется, имеют самые стойкие боковые лепестки. Окно Ханнинга может иметь наилучшее сочетание быстрого спада и минимальных боковых лепестков.

9. Дифференциация и интеграция

9.1. Упражнение 1

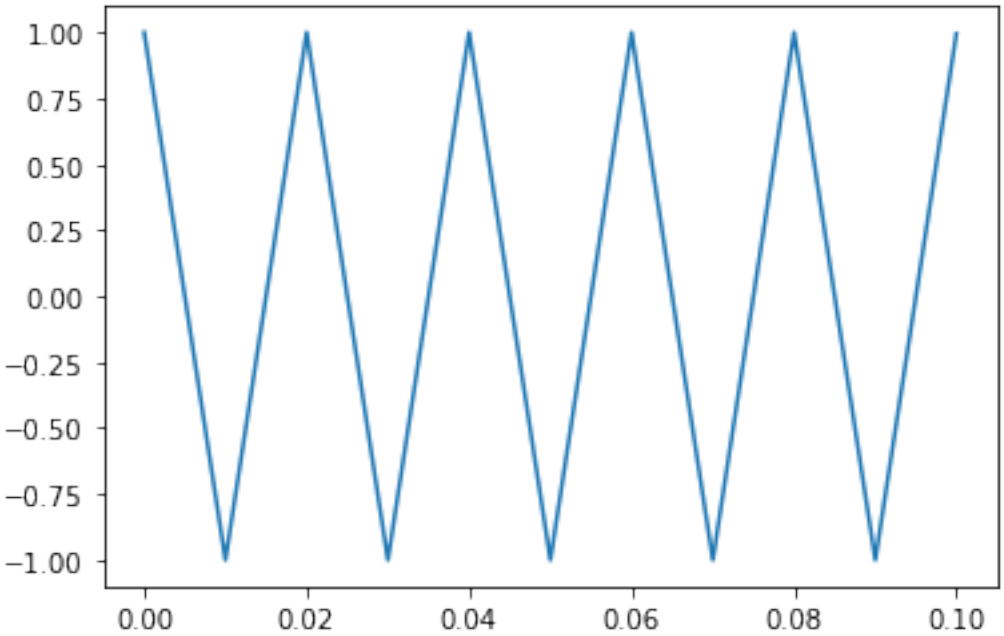
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

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

import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/thinkdsp.py
from thinkdsp import decorate
```

Прогнаны примеры из блокнота chap09.ipynb.

9.2. Упражнение 2

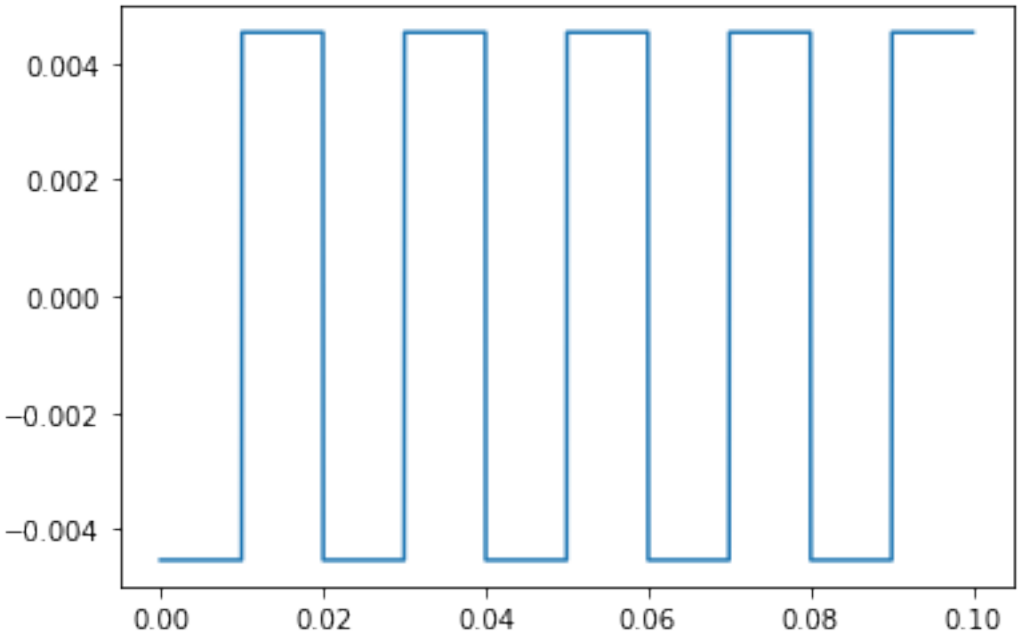
Создадим треугольный сигнал, напечатаем его, применим функцию diff к сигналу и проанализируем результат. Вычислим спектр изначального сигнала, применим differentiate, проанализируем результат. Преобразуем спектр обратно в сигнал, проанализируем результат и различия воздействия diff и differentiate на исходный сигнал.

```
from thinkdsp import TriangleSignal

w = TriangleSignal(freq=50).make_wave(duration=0.1, framerate=44100)
w.plot()
```



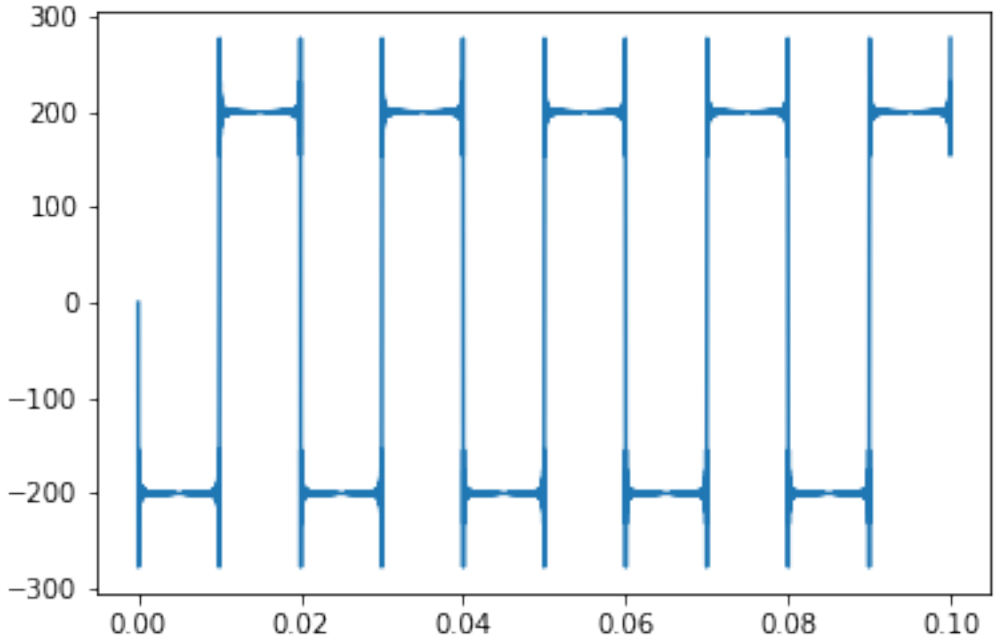
Применение функции diff:

```
out_w = w.diff()
out_w.plot()
```



Когда мы берем спектральную производную, мы получаем «звон» вокруг разрывов. Проблема в том, что производная треугольной волны не определена в точках треугольника. Применим дифференцирующий фильтр differentiate:

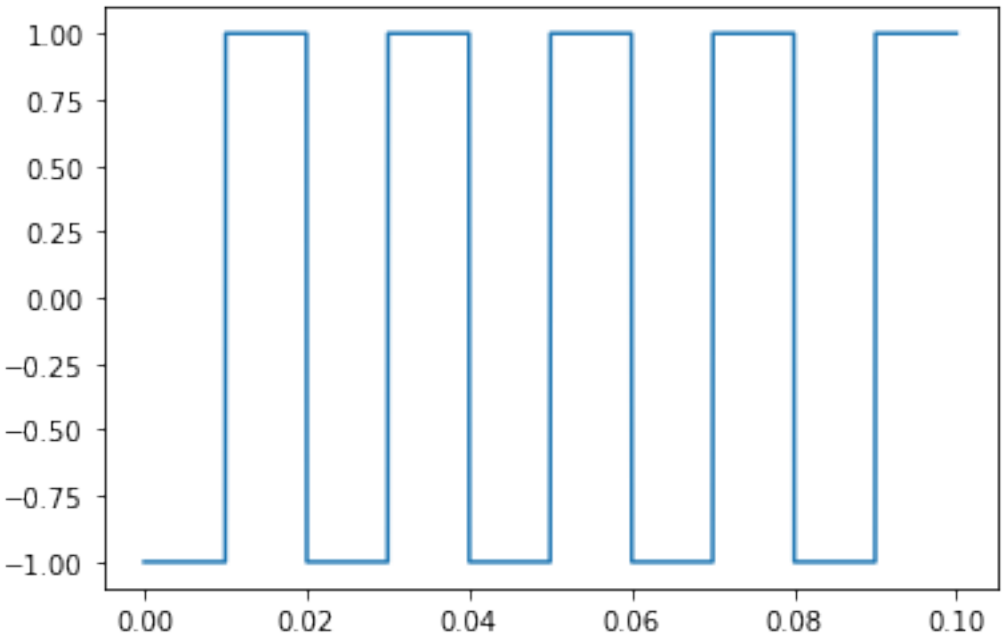
```
out_w2 = w.make_spectrum().differentiate().make_wave()
out_w2.plot()
```



9.3. Упражнение 3

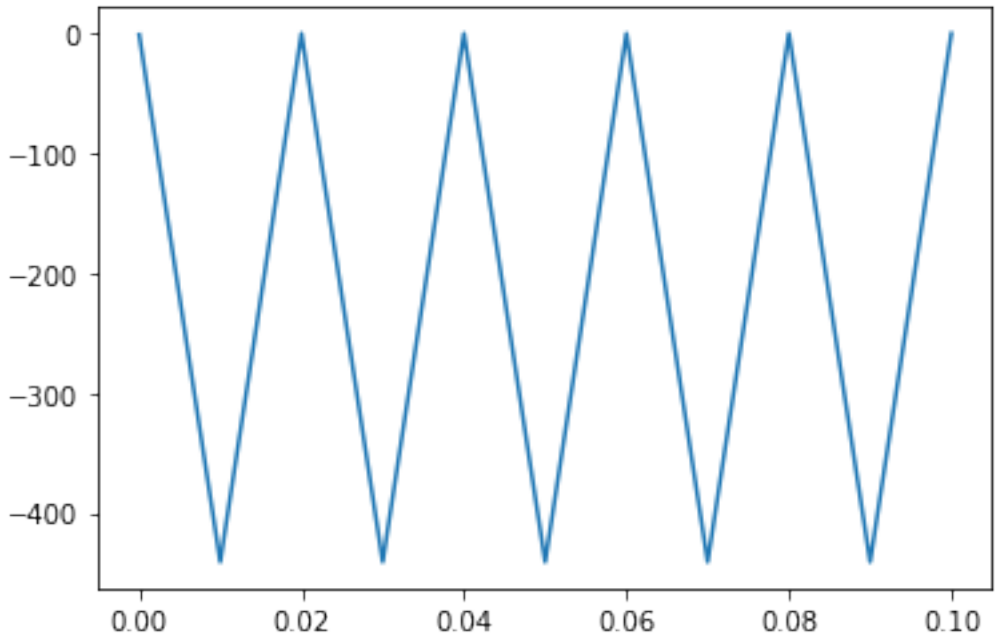
Изучим влияние cumsum и integrate на прямоугольный сигнал. Создадим прямоугольный сигнал, напечатаем его и применим cumsum. Вычислим спектр исходного сигнала, применим integrate. Преобразуем спектр в изначальный сигнал, проанализируем результаты и различия воздействия cumsum и integrate на прямоугольный сигнал.

```
from thinkdsp import SquareSignal

w = SquareSignal(freq=50).make_wave(duration=0.1, framerate=44100)
w.plot()
```



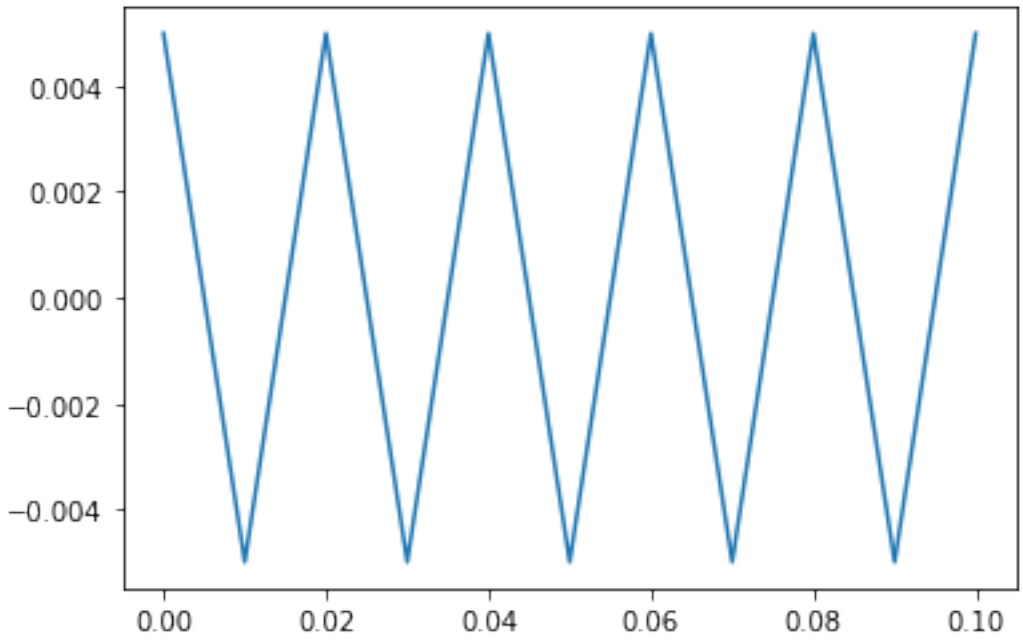
Совокупная сумма прямоугольной волны представляет собой треугольную волну.

```
out = w.cumsum()
out.plot()
```



Спектральный интеграл также представляет собой треугольную волну, хотя амплитуда сильно отличается.

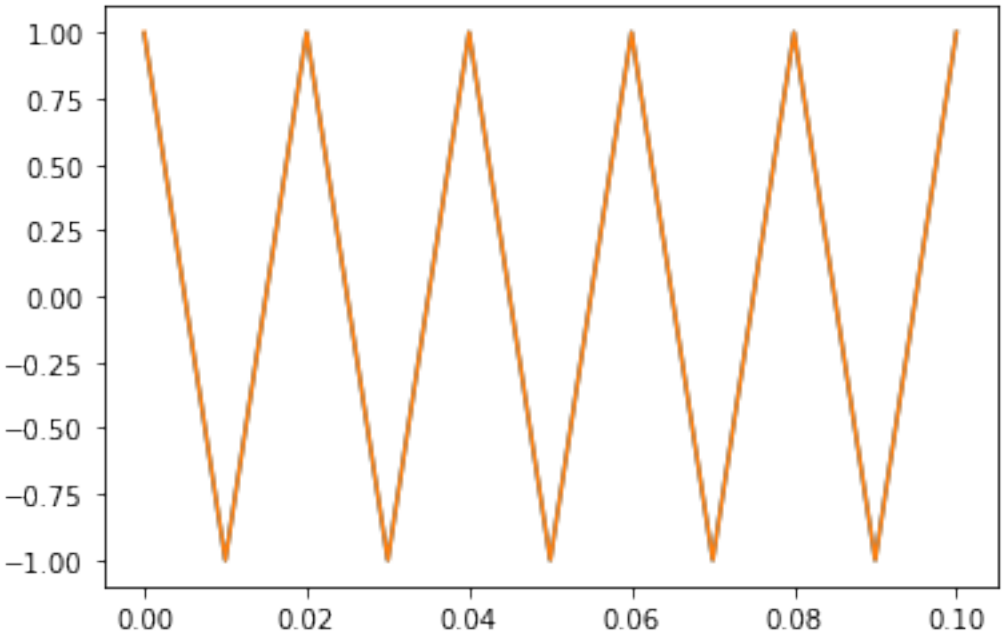
```
spectrum = w.make_spectrum().integrate()
spectrum.hs[0] = 0
out2 = spectrum.make_wave()
out2.plot()
```



Если мы снимем смещение и нормализуем две волны, они будут визуально похожи.

```
out.unbias()
out.normalize()
out2.normalize()
```

```
out.plot()
out2.plot()
```



Они численно похожи, но с точностью всего около 3 десятков.

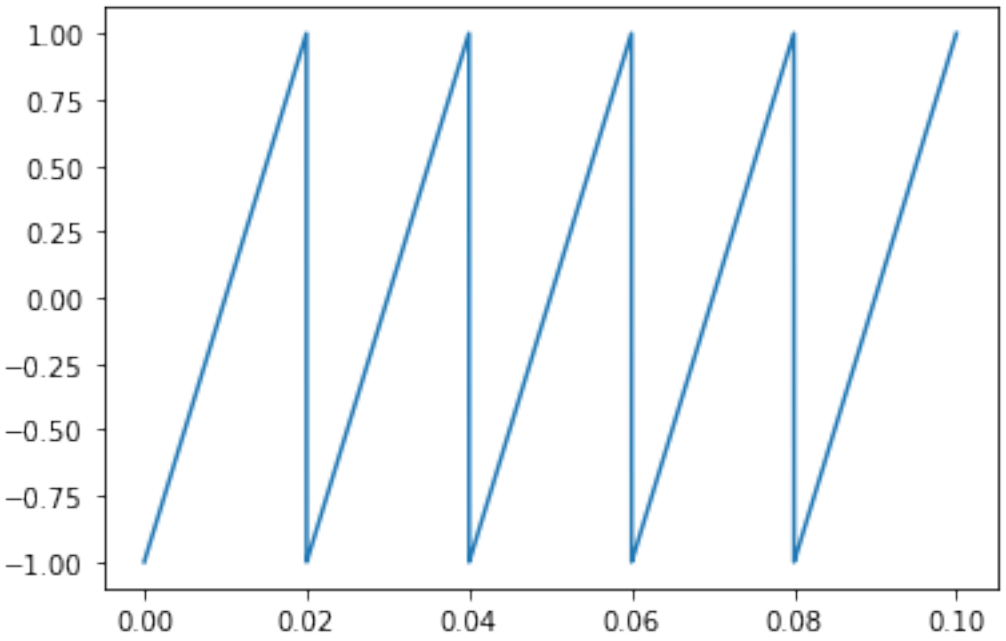
```
out.max_diff(out2)
```

0.0045351473922902175

9.4. Упражнение 4

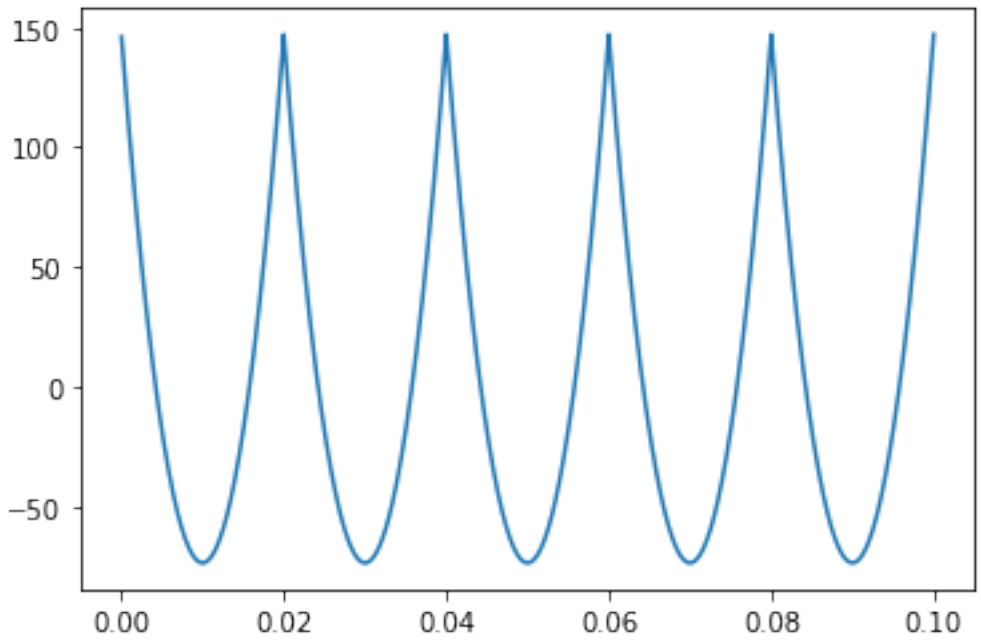
Изучим влияние двойного интегрирования на пилообразный сигнал. Вычислим спектр исходного сигнала, а затем дважды применим integrate.

```
from thinkdsp import SawtoothSignal

w = SawtoothSignal(freq=50).make_wave(duration=0.1, framerate=44100)
w.plot()
```



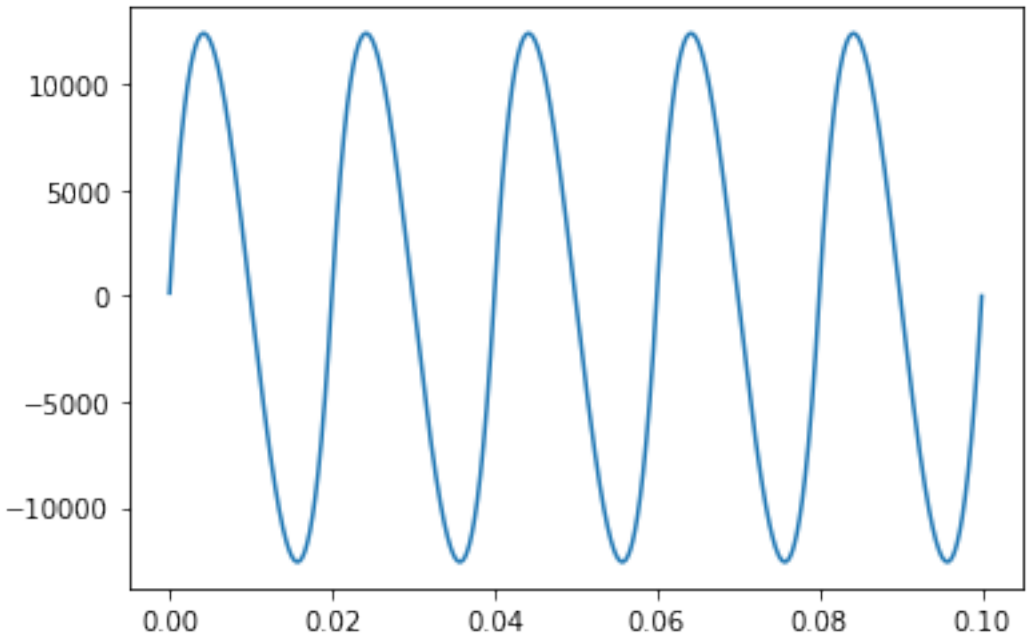
Первая нарастающая сумма — это парабола:

```
out = w.cumsum()
out.unbias()
out.plot()
```



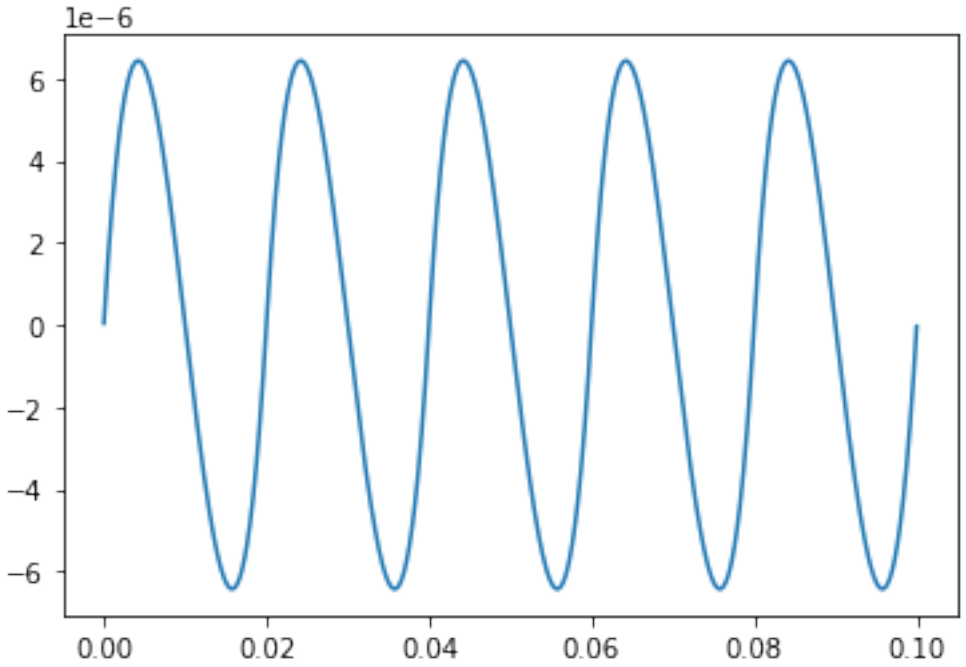
Вторая нарастающая сумма — это кубическая кривая:

```
out = out.cumsum()
out.plot()
```



Дважды интегрируем и получаем также кубическую кривую:

```
spec = w.make_spectrum().integrate().integrate()
spec.hs[0] = 0
out2 = spec.make_wave()
out2.plot()
```

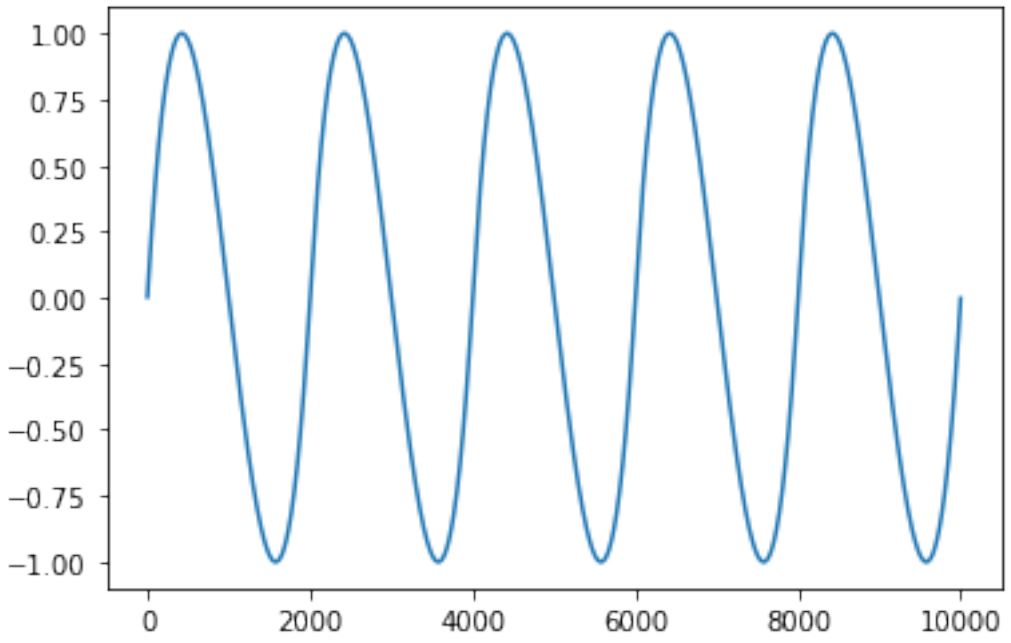


Полученный график похож на синусоиду, двойная интеграция действует как фильтр нижних частот.

9.5. Упражнение 5

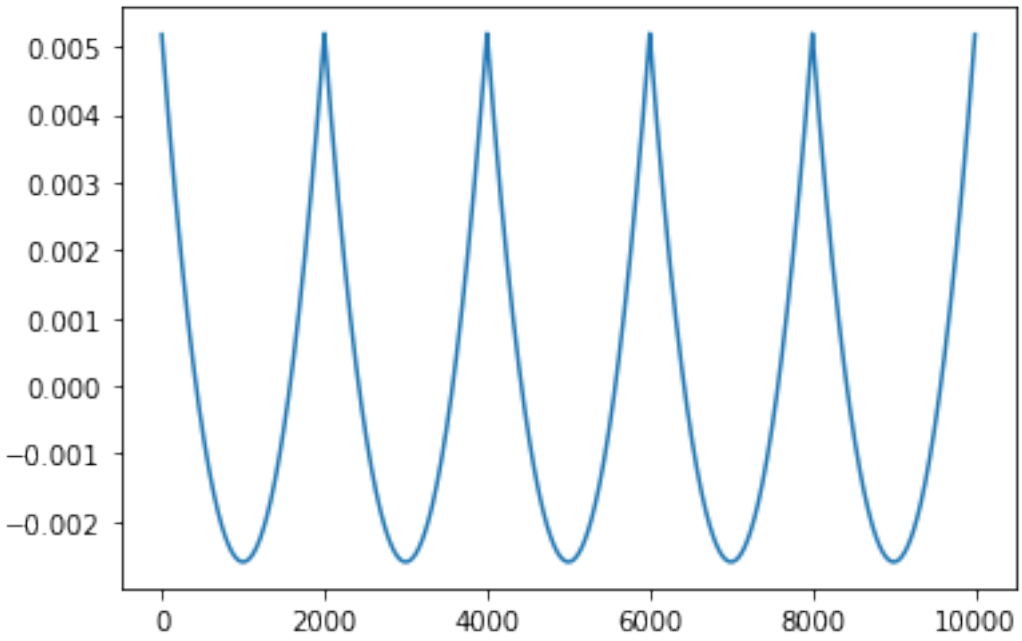
Изучим влияние второй разности и второй производной на CubicSignal сигнале, который определен в thinkdsp. Вычислим вторую разность, дважды применив diff. Вычислим вторую производную, дважды применив differentiate к спектру. Проанализируем получившиеся результаты.

```
from thinkdsp import CubicSignal

w = CubicSignal(freq=0.0005).make_wave(duration=10000, framerate=1)
w.plot()
```

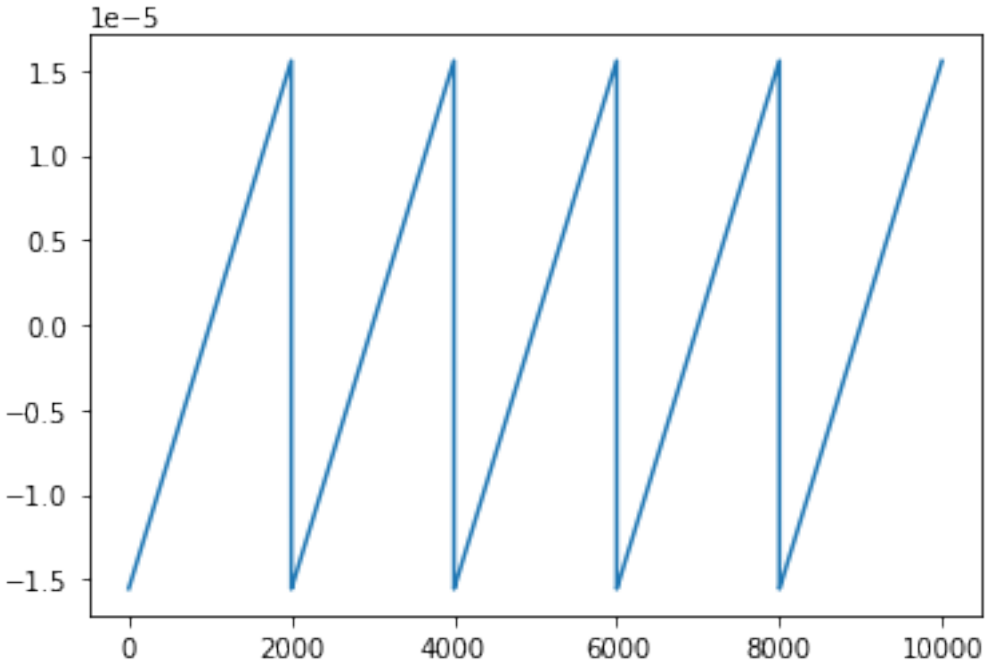


Первая разность — это парабола, а вторая разность — пилообразный сигнал.

```
out = w.diff()
out.plot()
```

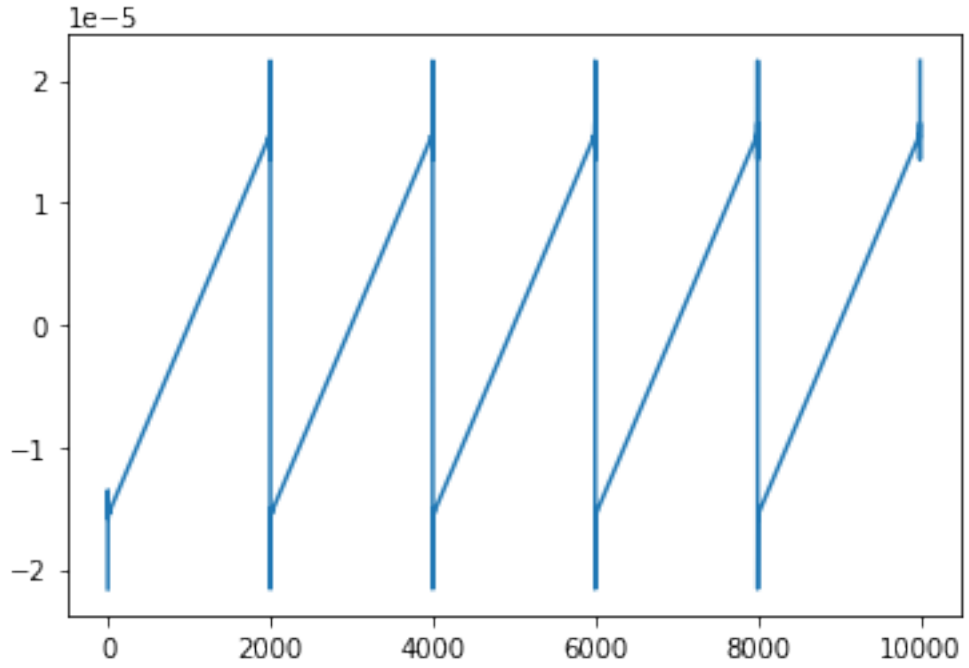



```
out = out.diff()
out.plot()
```



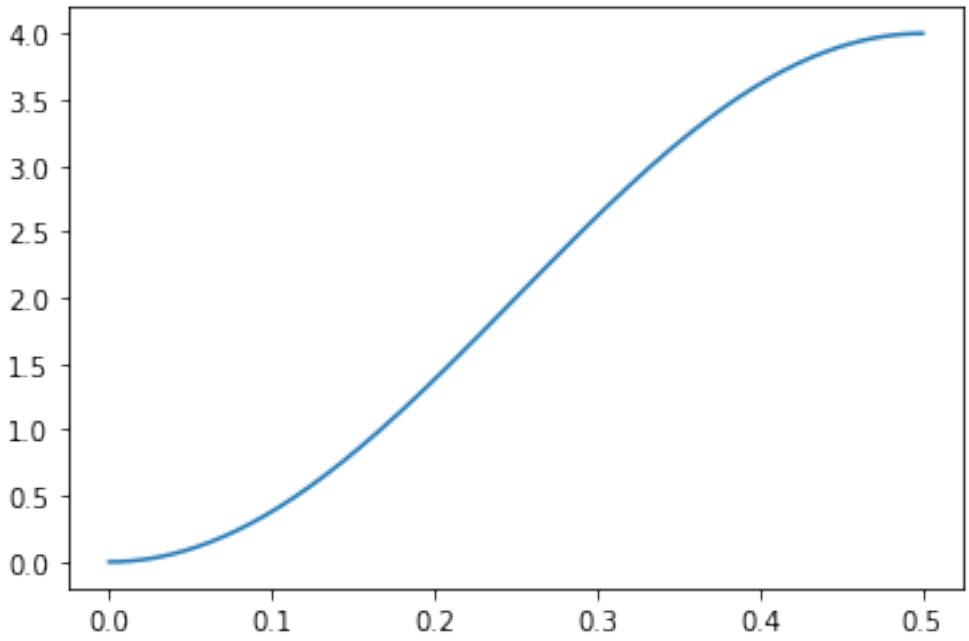
Когда мы дифференцируем дважды, мы получаем пилообразную форму с некоторым звоном. Опять же, проблема в том, что производная параболического сигнала не определена в точках.

```
spec = w.make_spectrum().differentiate().differentiate()
out2 = spec.make_wave()
out2.plot()
```



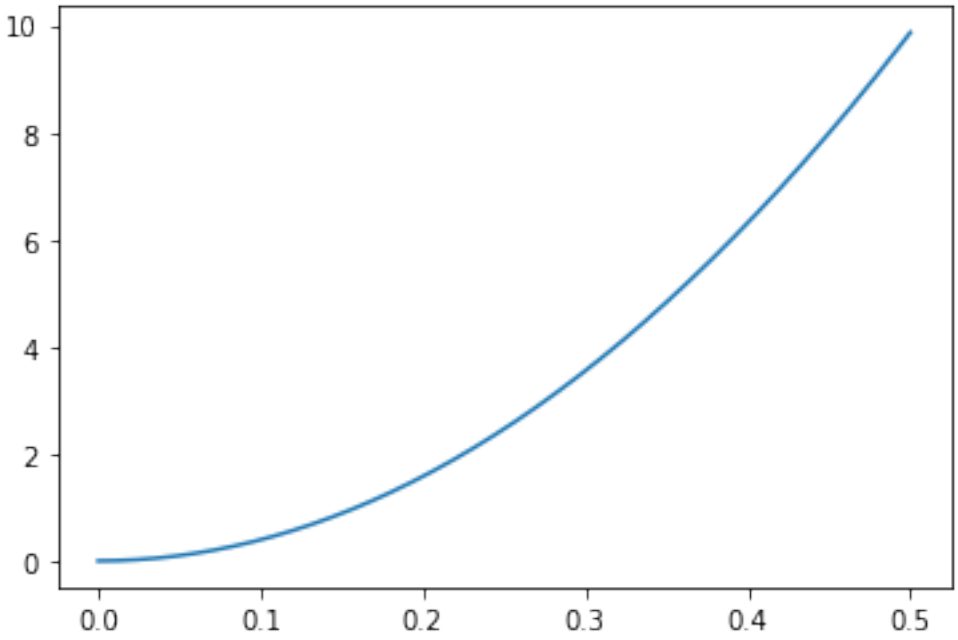
Окно второй разности это -1, 2, -1. Вычисляя ДПФ, можно найти соответствующий фильтр.

```
from thinkdsp import zero_pad
from thinkdsp import Wave

diff_window = np.array([-1.0, 2.0, -1.0])
padded = zero_pad(diff_window, len(w))
diff_wave = Wave(padded, framerate=w.framerate)
diff_filter = diff_wave.make_spectrum()
diff_filter.plot()
```



Для второй производной можно найти соответствующий фильтр, рассчитав фильтр первой прозводной и возведя его в квадрат.

```
derived_filter = w.make_spectrum()
derived_filter.hs = (PI2 * 1j * derived_filter.fs)**2
derived_filter.plot()
```



Оба являются фильтрами верхних частот, которые усиливают высокочастотные компоненты. Вторая производная является параболической, поэтому она больше всего усиливает самые высокие частоты. 2-я разность является хорошей аппроксимацией 2-й производной только на самых низких частотах, далее она существенно отклоняется.

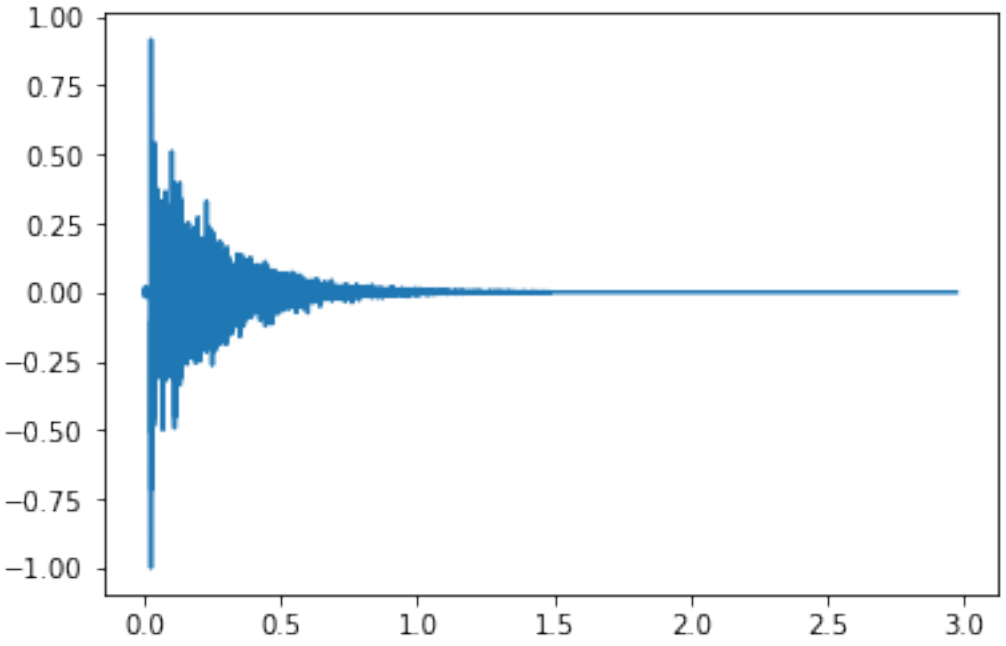
10. Сигналы и системы

10.1. Упражнение 1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

PI2 = 2 * np.pi

import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/thinkdsp.py
from thinkdsp import decorate
```

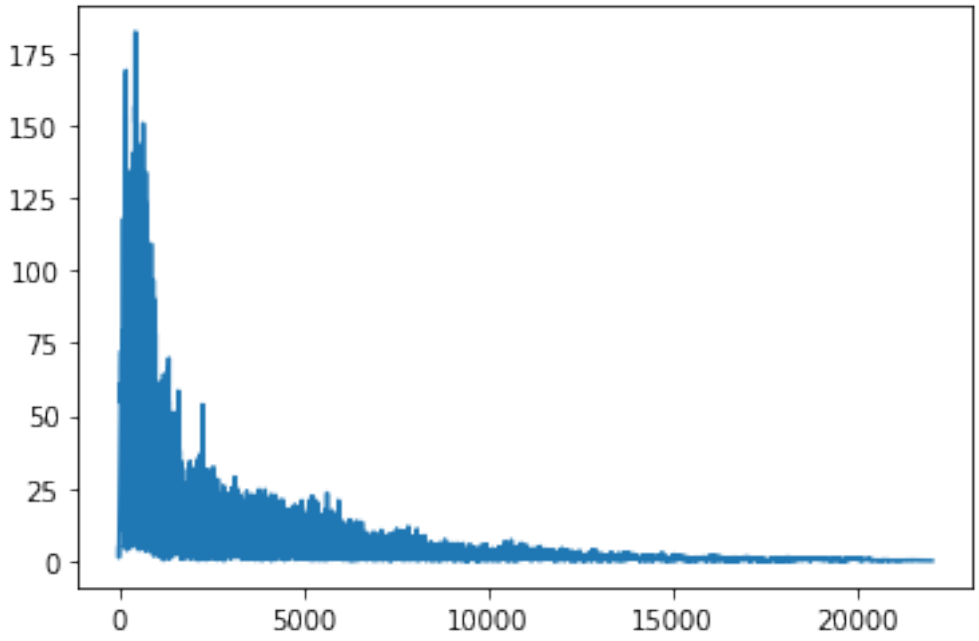
В разделе "Системы и свертка" на стр. 131 свертка описана как сумма сдвинутых и масштабированных копий сигнала.
А в разделе "Акустическая характеристика" на стр. 128 умножение ДПФ сигнала на передаточную функцию соответствует круговой свертке, но в предположении периодичности сигнала. В результате можно заметить, что на выходе, в начале фрагмента, слышна лишняя нота, "затекшая" из конца этого фрагмента.
К счастью, есть стандартное решение этой проблемы. Если перед вычислением ДПФ добавить достаточно нулей в конец сигнала, эффекта "заворота" можно избежать.
Измените пример в chap10.ipynb и убедитесь, что дополнение нулями устраняет лишнюю ноту в начале фрагмента.
Урежем оба сигнала до 2^16 элементов, а затем дополним их нулями до 2^17.

```
from thinkdsp import read_wave

if not os.path.exists('180960__kleeb__gunshot.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/180960__kleeb__gunshot.wav
res = read_wave('180960__kleeb__gunshot.wav')
start = 0.12
res = res.segment(start=start)
res.shift(-start)
res.truncate(2**16)
res.zero_pad(2**17)
res.normalize()
res.plot()
```



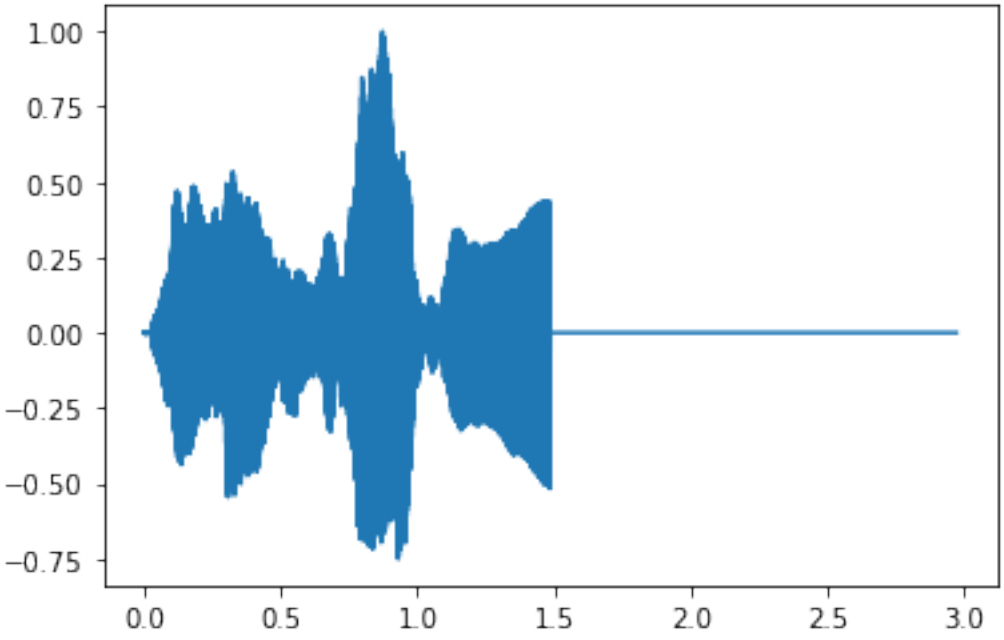
Получим спектр

```
spec = res.make_spectrum()
spec.plot()
```



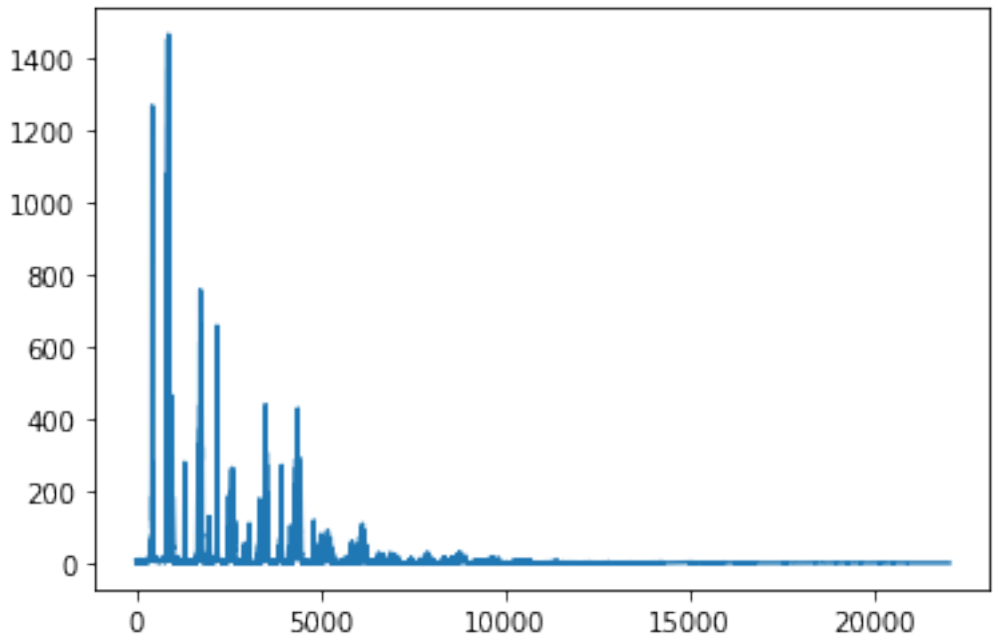

```
if not os.path.exists('92002__jcveliz__violin-origional.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/92002__jcveliz__violin-origional.wav

violin = 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()
```



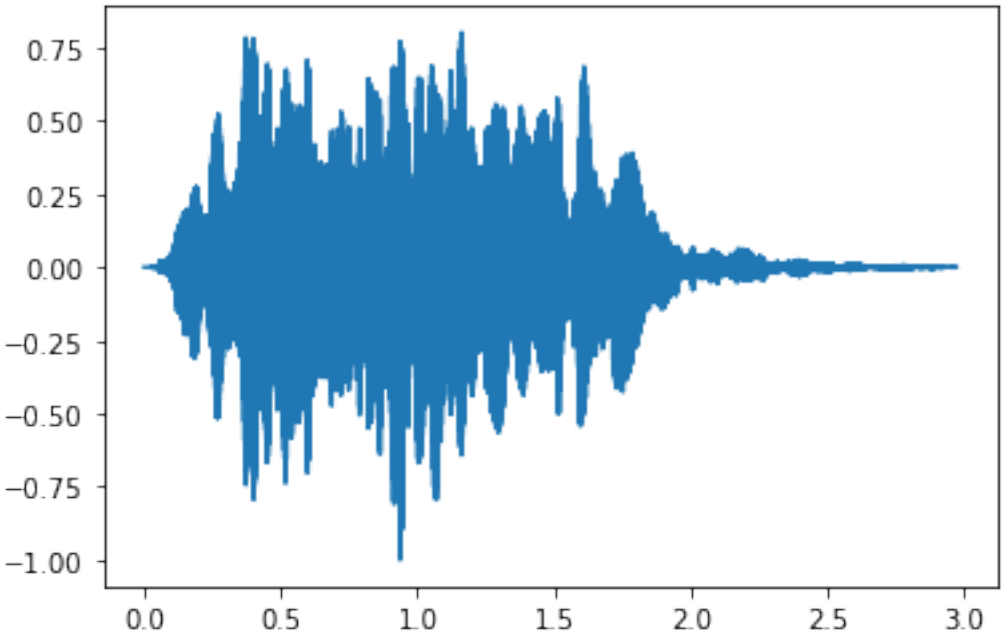
Получим спектр

```
spectrum = violin.make_spectrum()
spectrum.plot()
```



Теперь умножим ДПФ сигнала на передаточную функцию и преобразуем обратно в волну.

```
output = (spectrum * spec).make_wave()
output.normalize()
output.plot()
output.make_audio()
```

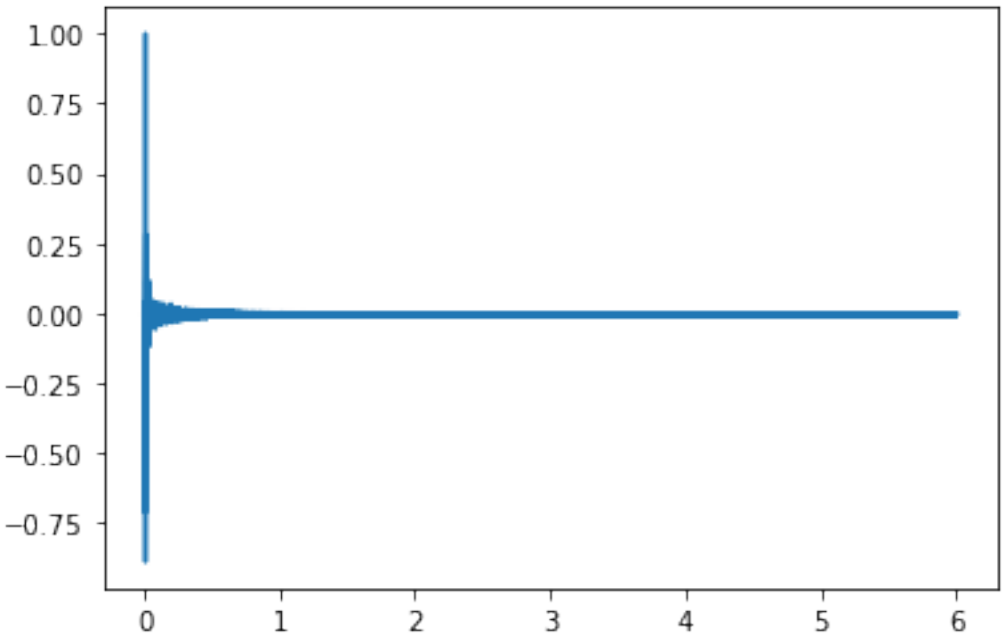


Теперь можно услышать, что лишней ноты в начале нет.

10.2. Упражнение 2

Возьмём один из звуков с ресура Open Air с испульсной харакеристикой. Дальше, найдём короткие записи с той же частотой дискретизации, что и у скачанной импульсной характеристики. Смоделируем двумя способами звучание записи в том пространстве, где была измерена импульсная характеристика, как сверткой самой записи с импульсной характеристикой, так и умножением ДПФ записи на вычисленный фильтр, соответствующий импульсной характеристике.

```
if not os.path.exists('impulse_task10.wav'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/impulse_task10.wav

response = read_wave('impulse_task10.wav')
start = 0
duration = 6
response = response.segment(duration=duration)
response.shift(-start)
response.normalize()
response.plot()
```

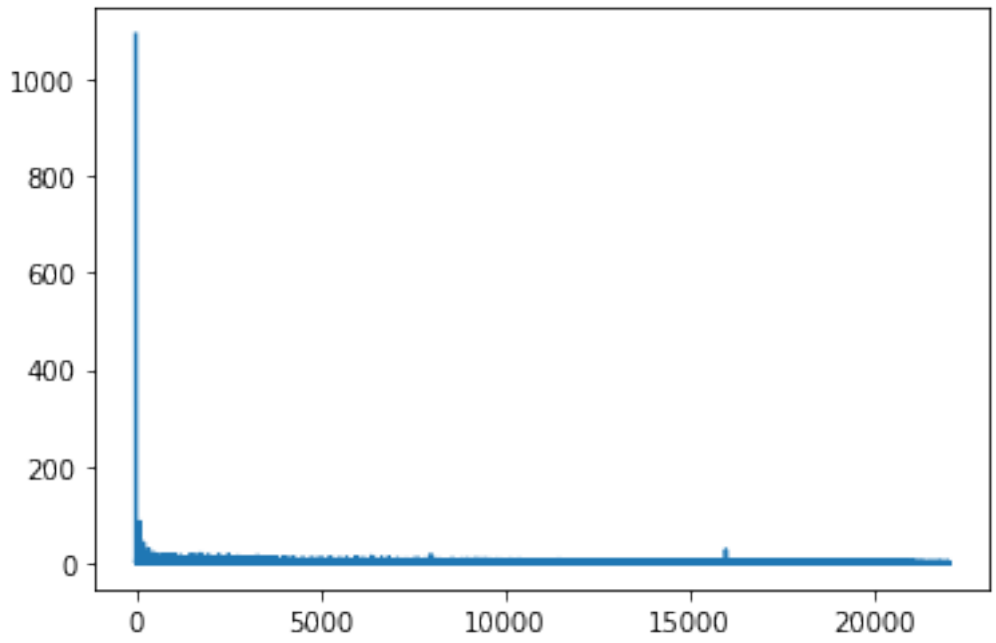


Прослушаем

```
response.make_audio()
```

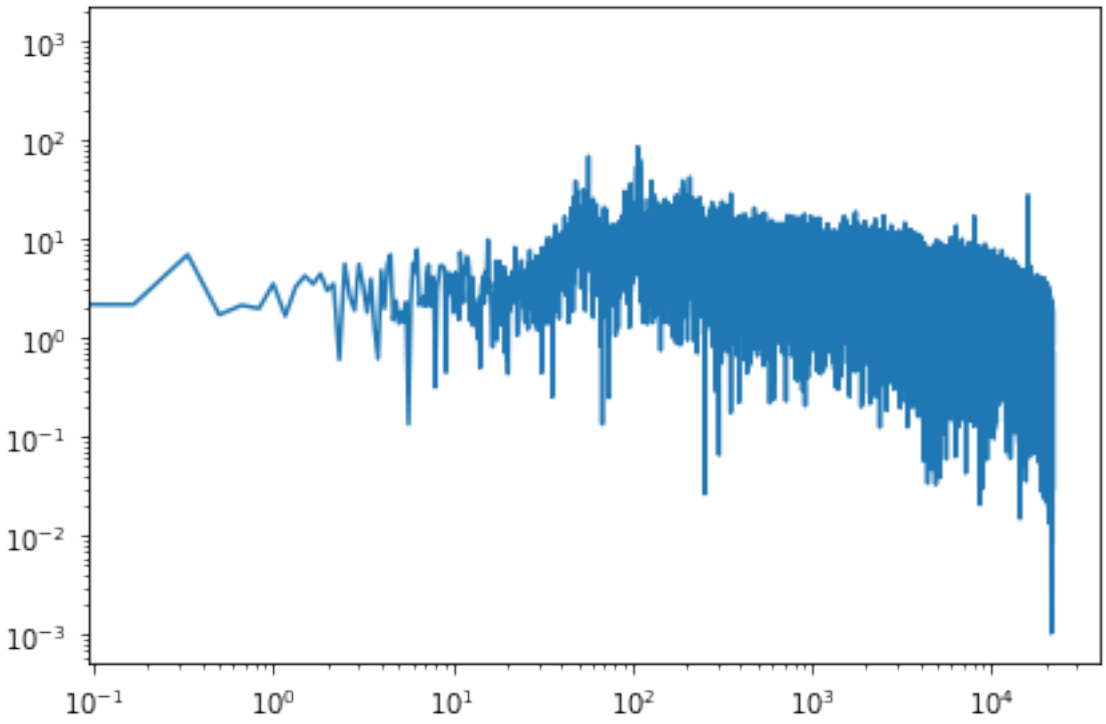
ДПФ импульсной характеристики:

```
transfer = response.make_spectrum()
transfer.plot()
```



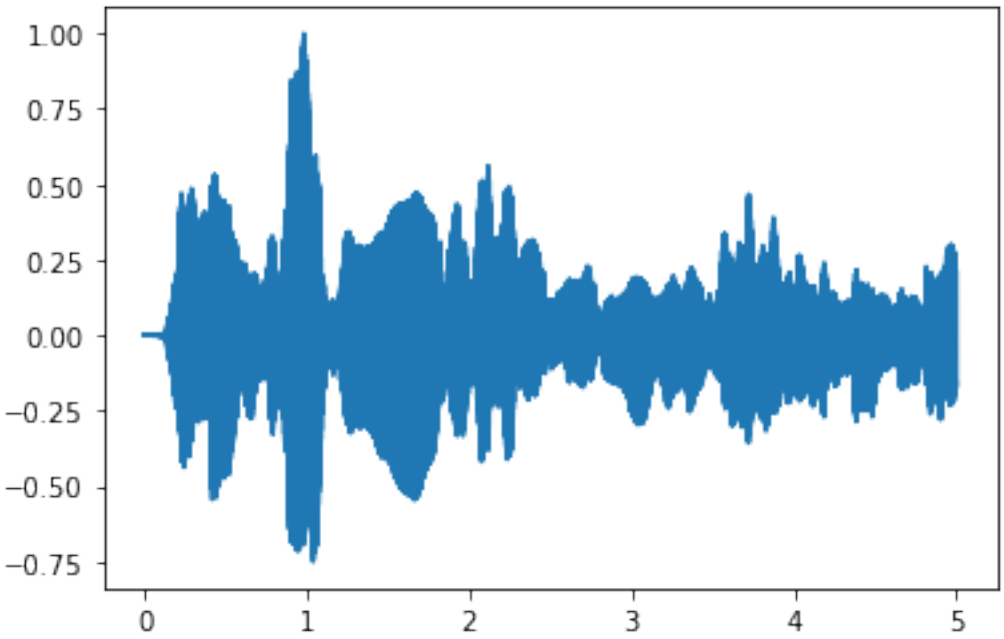
Представим в логарифмическом масштабе:

```
transfer.plot()
decorate(xscale='log', yscale='log')
```



Теперь мы можем смоделировать, как будет звучать запись, если она будет воспроизведена в той же комнате и записана таким же образом. Вот запись скрипки, которую мы использовали раньше:

```
wave = read_wave('92002__jcveliz__violin-origional.wav')
start = 0.0
wave = wave.segment(start=start)
wave.shift(-start)
wave.truncate(len(response))
wave.normalize()
wave.plot()
```



Так запись звучит до трансформации:

```
wave.make_audio()
```

Теперь вычислим ДПФ преобразования записи. Урежем запись до той же длины, что и импульсная характеристика.

```
spectrum = wave.make_spectrum()
len(spectrum.hs), len(transfer.hs)
```

(110251, 132301)

```
spectrum.fs
```

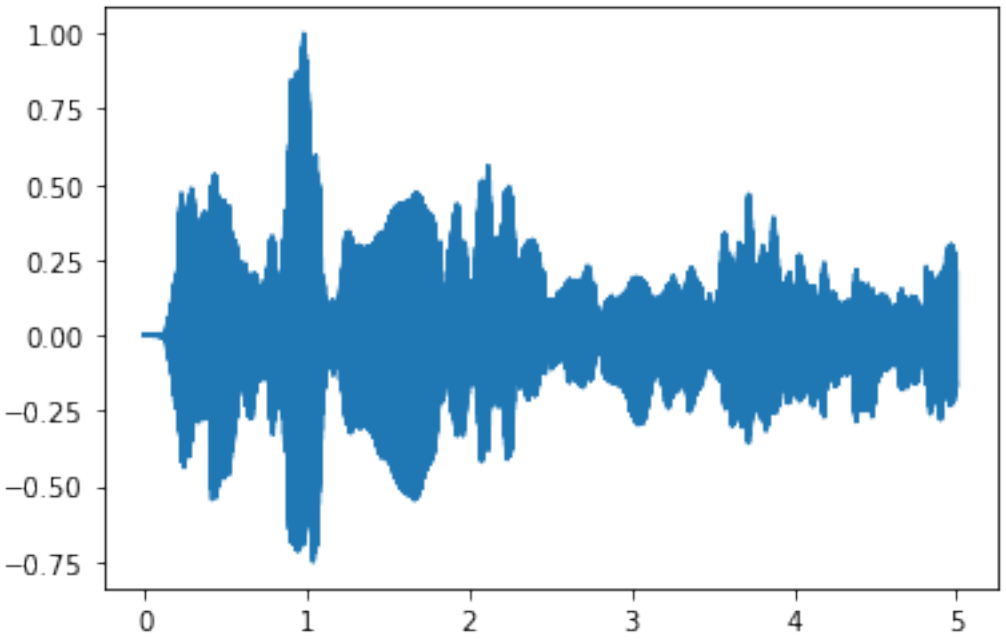
array([0.00000e+00, 2.00000e-01, 4.00000e-01, ..., 2.20496e+04, 2.20498e+04, 2.20500e+04])

```
transfer.fs
```

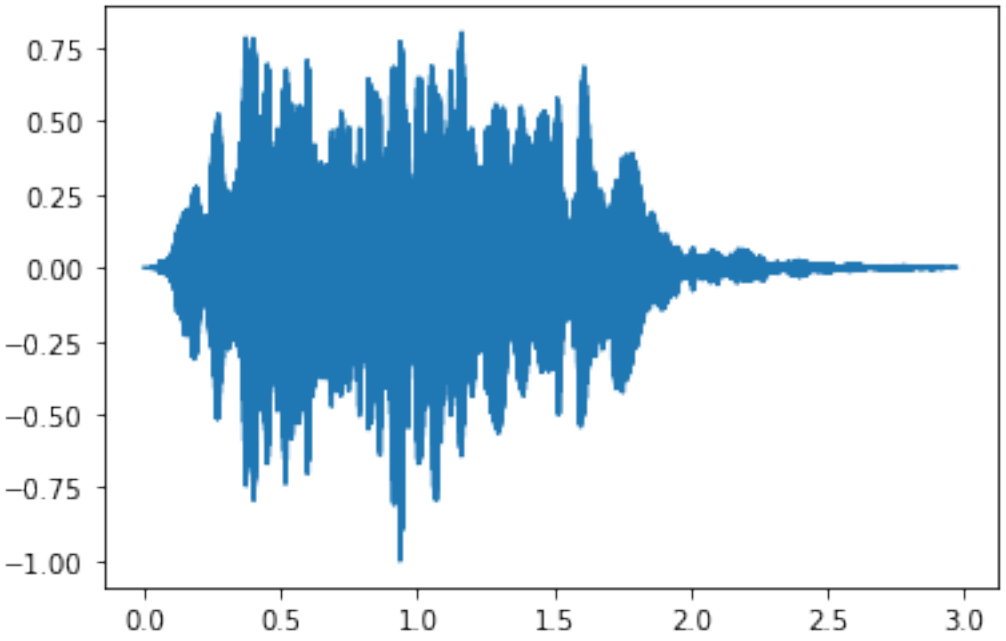
array([0.00000000e+00, 1.66666667e-01, 3.33333333e-01, ..., 2.20496667e+04, 2.20498333e+04, 2.20500000e+04])

Умножим в частотной области и преобразуем обратно во временную область.

```
#output = (spectrum * transfer).make_wave()
output.normalize()
wave.plot()
```




```
output.plot()
```



Получим звучание

```
output.make_audio()
```

11. Модуляция и сэмплирование

11.1. Упражнение 1

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import os
if not os.path.exists('thinkdsp.py'):
    !wget https://github.com/archer-man/ThinkDSP/raw/master
    /code/thinkdsp.py
from thinkdsp import decorate
```

Прогнаны примеры из блокнота chap11.ipynb.

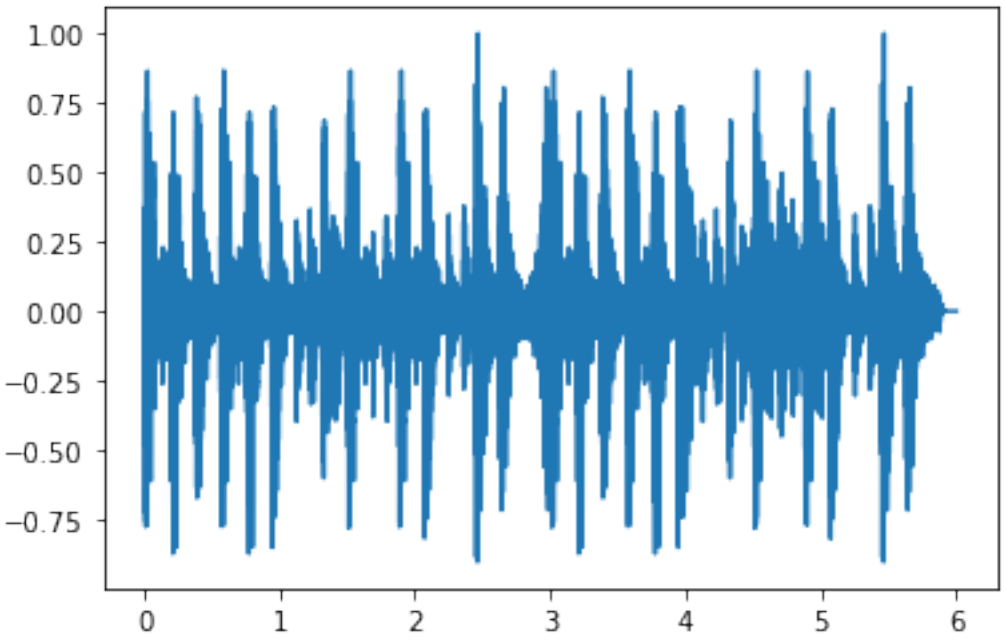
11.2. Упражнение 2

Просмотрен видеоролик от Криса "Монти" Монтгомери.

11.3. Упражнение 3

В этом задании возьмём пример с барабанным соло.В задании нужно применить фильтр низких частот до выборки, затем применить фильтр низких частот для удаления спектральных копий, вызванные выборкой. Результат должен быть идентичен отфильтрованному сигналу.

```
if not os.path.exists('263868__kevcio__amen-break-a-160-bpm.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master
    /code/263868__kevcio__amen-break-a-160-bpm.wav

from thinkdsp import read_wave

wave = read_wave('263868__kevcio__amen-break-a-160-bpm.wav')
wave.normalize()
wave.plot()
```

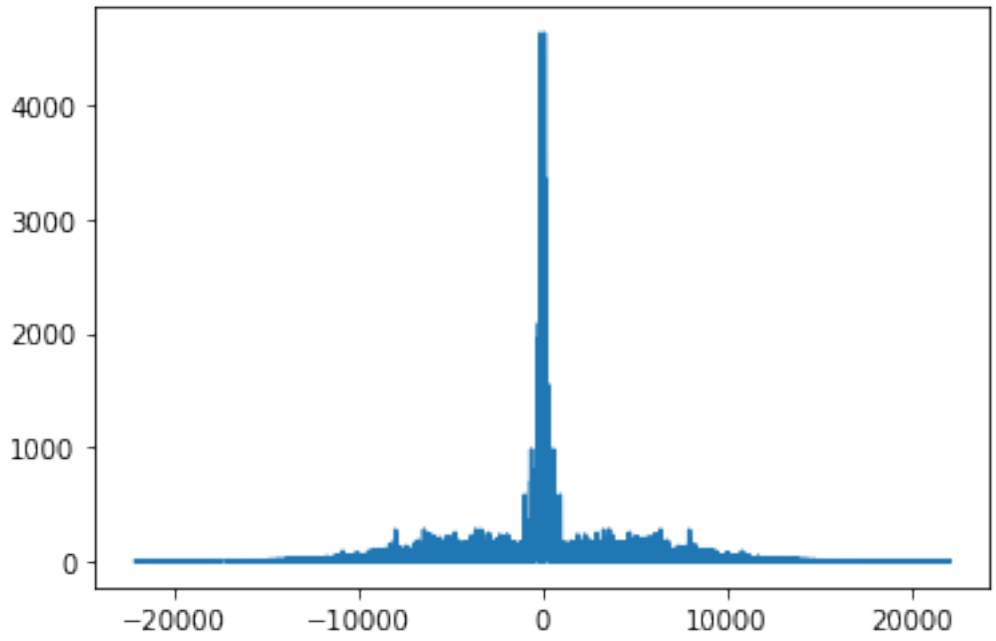


Данный сигнал дискретизируется на частоте 44 100 Гц.
Прослушаем

```
wave.make_audio()
```

Получим спектр

```
spectrum = wave.make_spectrum(full=True)
spectrum.plot()
```

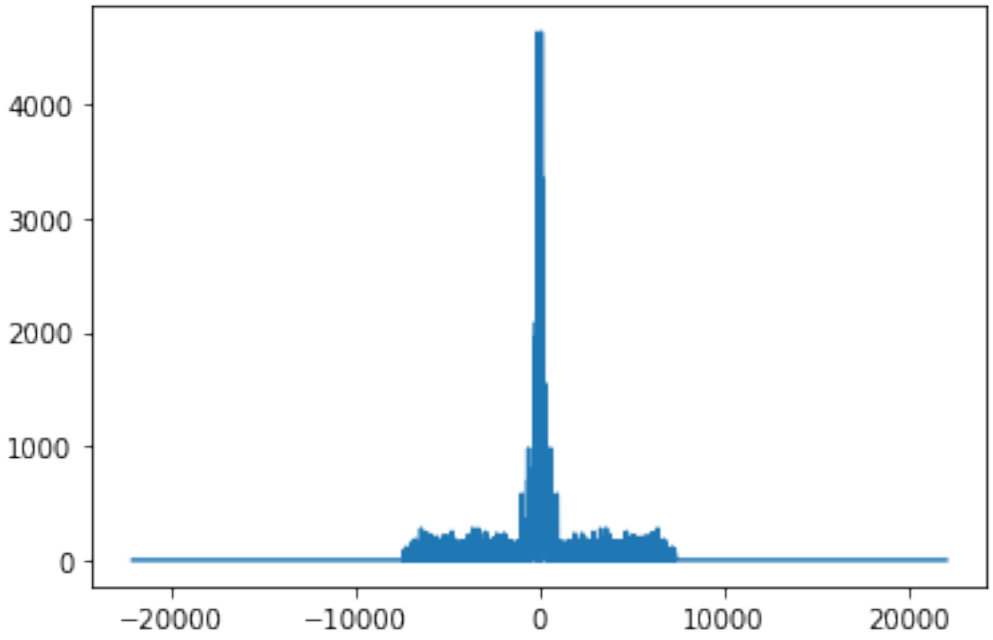


Уменьшим частоту дискретизации в 3 раза:

```
factor = 3
framerate = wave.framerate / factor
cutoff = framerate / 2 - 1
```

Перед сэмплированием мы применяем фильтр сглаживания, чтобы удалить частоты выше новой частоты сворачивания, которая равна частоте кадров деленной на два

```
spectrum.low_pass(cutoff)
spectrum.plot()
```



Вот как это звучит после фильтрации:

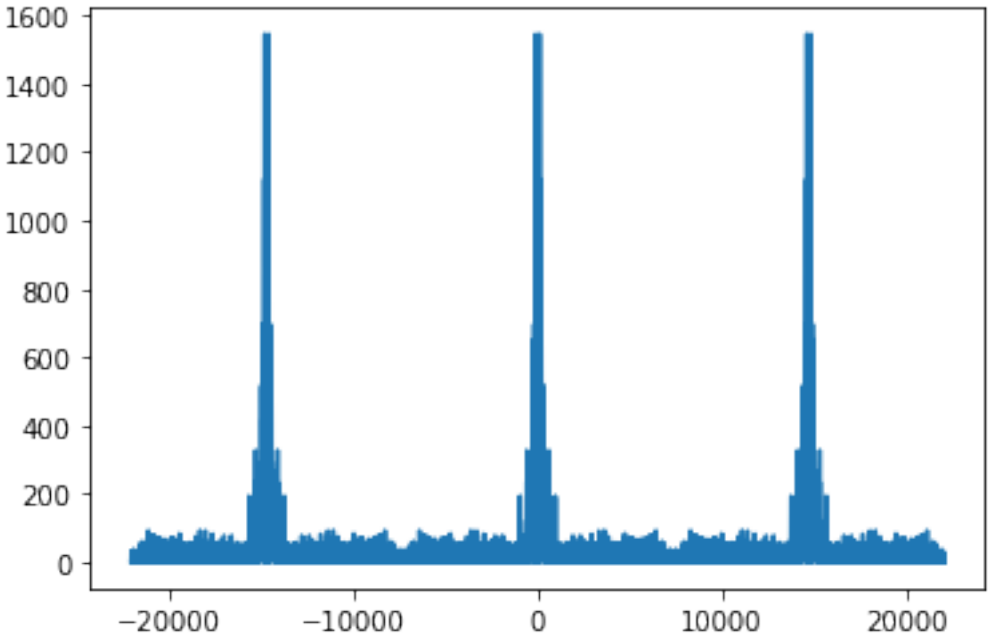
```
filtered = spectrum.make_wave()
filtered.make_audio()
```

Функция, которая симулирует процесс сэмплирования:

```
from thinkdsp import Wave

def sample(wave, factor):

    ys = np.zeros(len(wave))
    ys[::factor] = np.real(wave.ys[::factor])
    return Wave(ys, framerate=wave.framerate)
```

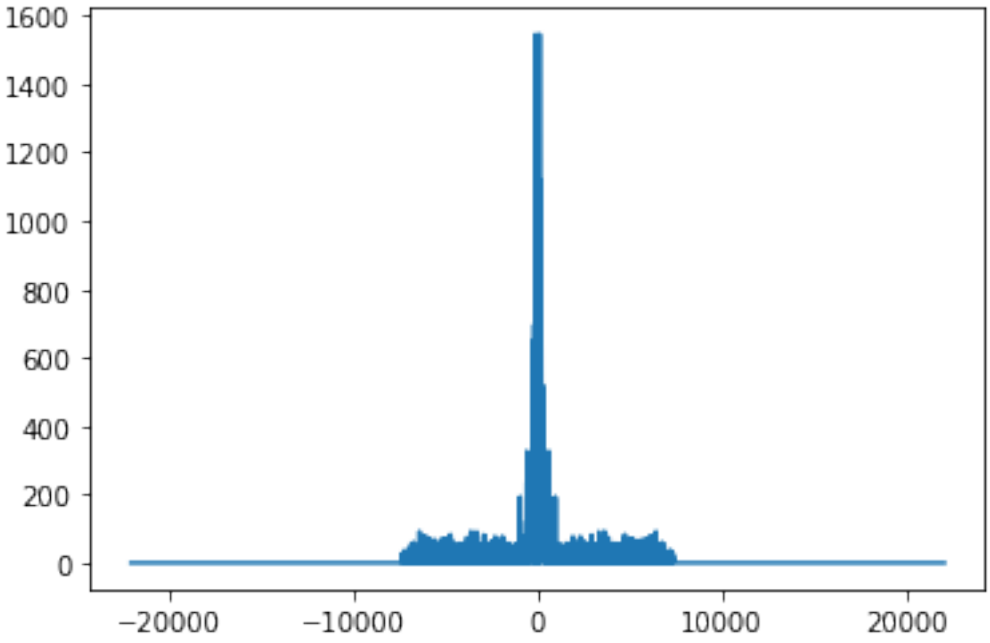
Результат содержит копии спектра около 20 кГц.

```
sampled = sample(filtered, factor)
sampled.make_audio()

sampled_spectrum = sampled.make_spectrum(full=True)
sampled_spectrum.plot()
```



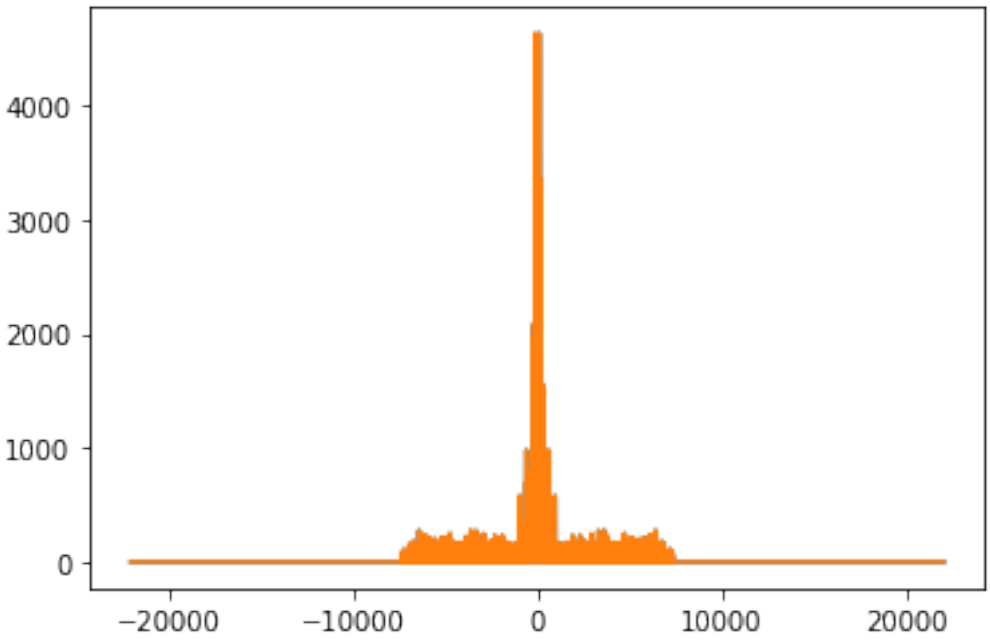
Теперь мы можем избавиться от спектральных копий, снова применив фильтр сглаживания:

```
sampled_spectrum.low_pass(cutoff)
sampled_spectrum.plot()
```



Мы потеряли половину энергии в спектре, но мы можем масштабировать результат, чтобы вернуть его обратно:

```
sampled_spectrum.scale(factor)
spectrum.plot()
sampled_spectrum.plot()
```



Теперь разница между спектром до и после дискретизации должна быть небольшой.

```
spectrum.max_diff(sampled_spectrum)
```

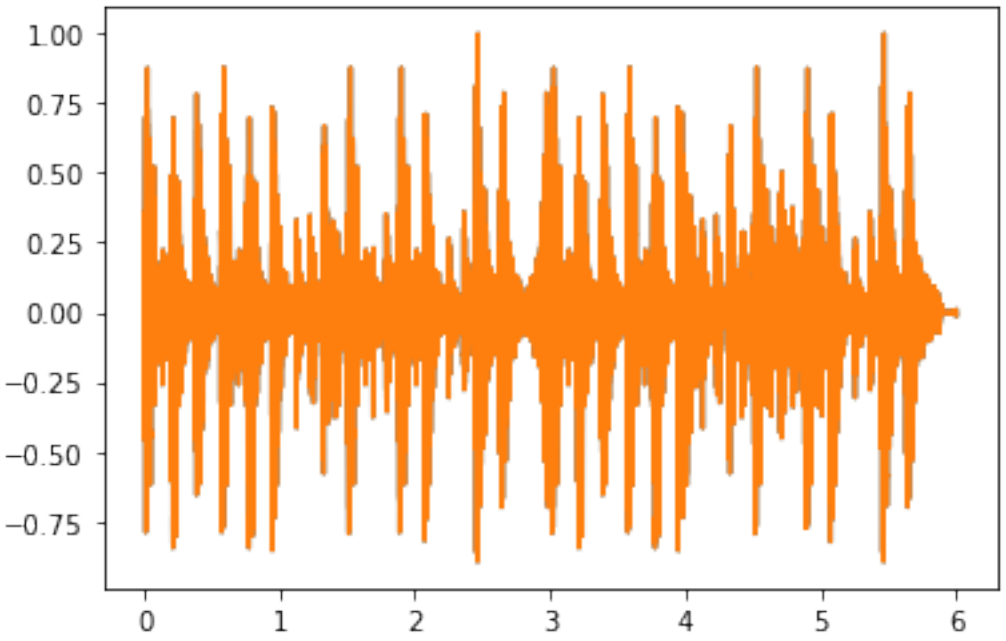
1.8189894035458565e-12

После фильтрации и масштабирования мы можем выполнить преобразование обратно в волну:

```
interpolated = sampled_spectrum.make_wave()
interpolated.make_audio()
```

Разница между интерполированной и отфильтрованной волной должна быть небольшой.

```
filtered.plot()
interpolated.plot()
```




```
filtered.max_diff(interpolated)
```

```
5.56290642113787e-16
```

12. FSK

12.1. Теория

requency Shift Key — вид модуляции, при которой скачкообразно изменяется частота несущего сигнала в зависимости от значений символов информационной последовательности. Частотная модуляция весьма помехоустойчива, так как помехи искажают в основном амплитуду, а не частоту сигнала.
Схема для демонстрации этой технологии приведена ниже. Она состоит из передатчика, генерирующего случайные значения в диапазоне 0–255 байт, и приёмника, в котором с помощью КИХ-фильтра принятый сигнал смещается и центрируется вокруг несущей частоты.

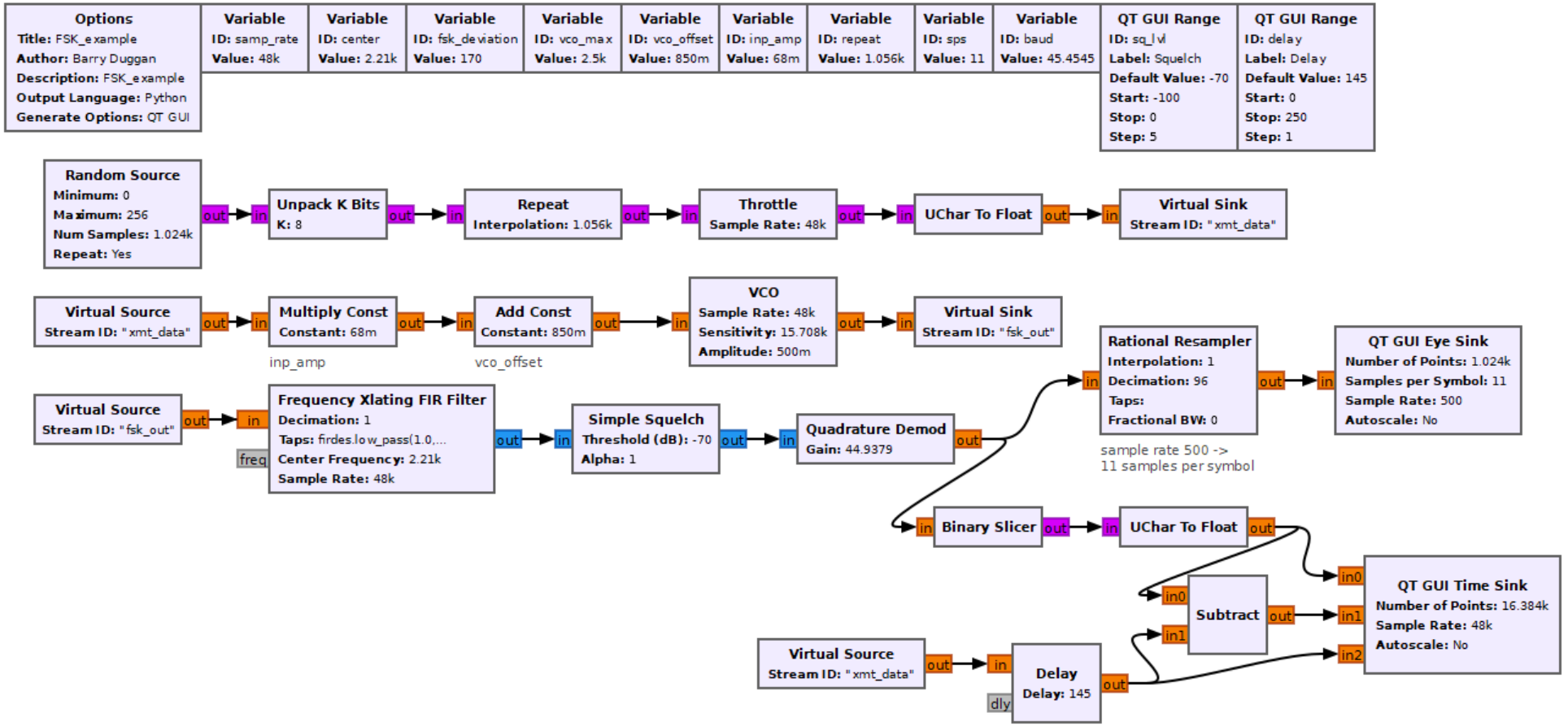


Рисунок 12.1. Схема

Блоки, которые мы используем:

- 1) Variable — блок адресующий в уникальной переменной. При помощи ID можно передавать информацию через другие блоки.
- 2) QT GUI Range - графический интерфейс для изменения заданной переменной.
- 3) Random Source — генератор случайных чисел.
- 4) Unpack K bits — преобразуем байт с k релевантными битами в k выходных байтов по одному биту в каждом.
- 5) Repeat — количество повторений ввода, действующее как коэффициент интерполяции.
- 6) Throttle — дросселировать поток таким образом, чтобы средняя скорость не превышала удельную скорость.
- 7) Uchar To Float — конвертация байта в Float.
- 8) Virtual Sink — сохраняет поток в вектор, что полезно, если нам нужно иметь данные за эксперимент.
- 9) Virtual Source — источник данных, который передаёт элементы на основе входного вектора.
- 10) Multiply Const — умножает входной поток на скаляр или вектор.
- 11) Add Const — прибавляет к потоку скаляр или вектор.
- 12) VCO — генератор, управляемый напряжением. Создает синусоиду на основе входной амплитуды.
- 13) Frequency Xlating FIR Filter — этот блок выполняет преобразование частоты сигна—ла, а также понижает дискретизацию сигнала, запуская на нем прореживающий КИХ—фильтр. Его можно использовать в качестве канализатора для выделения узкополосной части широкополосного сигнала без центрирования этой узкополосной части по частоте.
- 14) Simple Squelch — простой блок шумоподавления на основе средней мощности сигнала и порога в дБ.
- 15) Quadrature Demod — квадратурная модуляция.
- 16) Binary Slicer — слайсы от значения с плавающей запятой, производя 1—битный вывод. Положительный ввод производит двоичную 1, а отрицательный ввод производит двоичный ноль.
- 17) QT GUI Sink — выводы необходимой информации в графическом интерфейсе.

Также приведем значения используемых переменных

Variables

baud	45.45454545454546
center	2210
delay	145
fsk_deviation	170
inp_amp	0.06800000000000006
repeat	1056
samp_rate	48000
sps	11
sq_lvl	-70
vco_max	2500
vco_offset	0.85

Рисунок 12.2. Значения переменных

12.2. Тестирование

Проведем тестирорвание. В первом случае принятый сигнал отстаёт из-за того, что между приёмником и передатчиком много других блоков и фильтров. Для компенсации этого нужно поменять задержку передаваемого сигнала на оптимальную, равную 145.

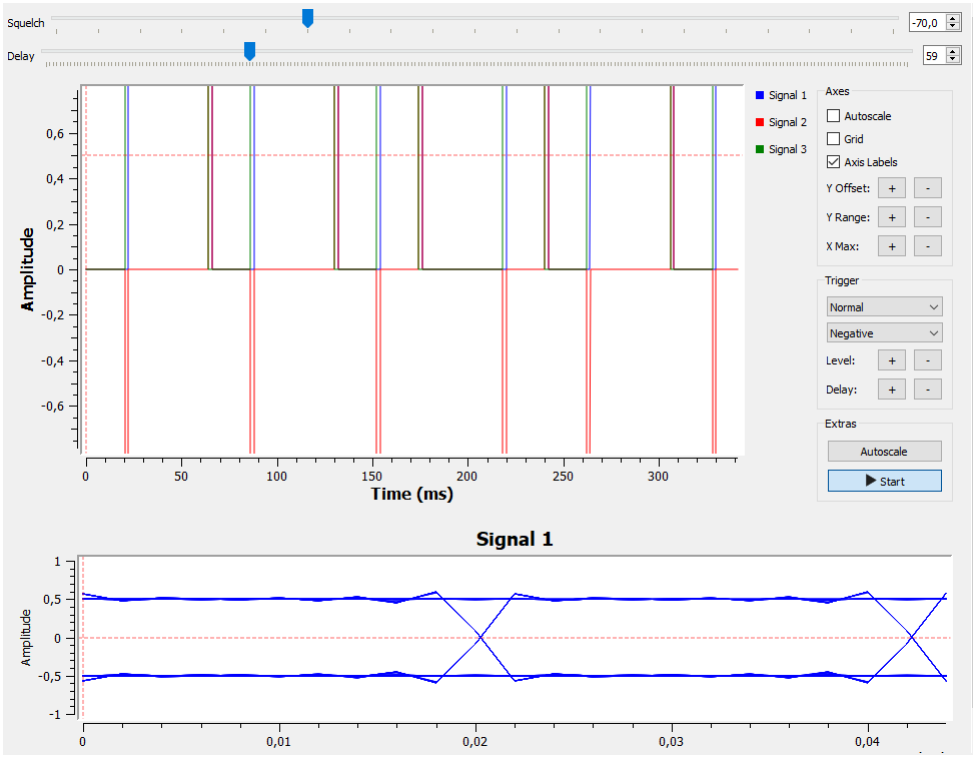


Рисунок 12.3. Задержка = 59

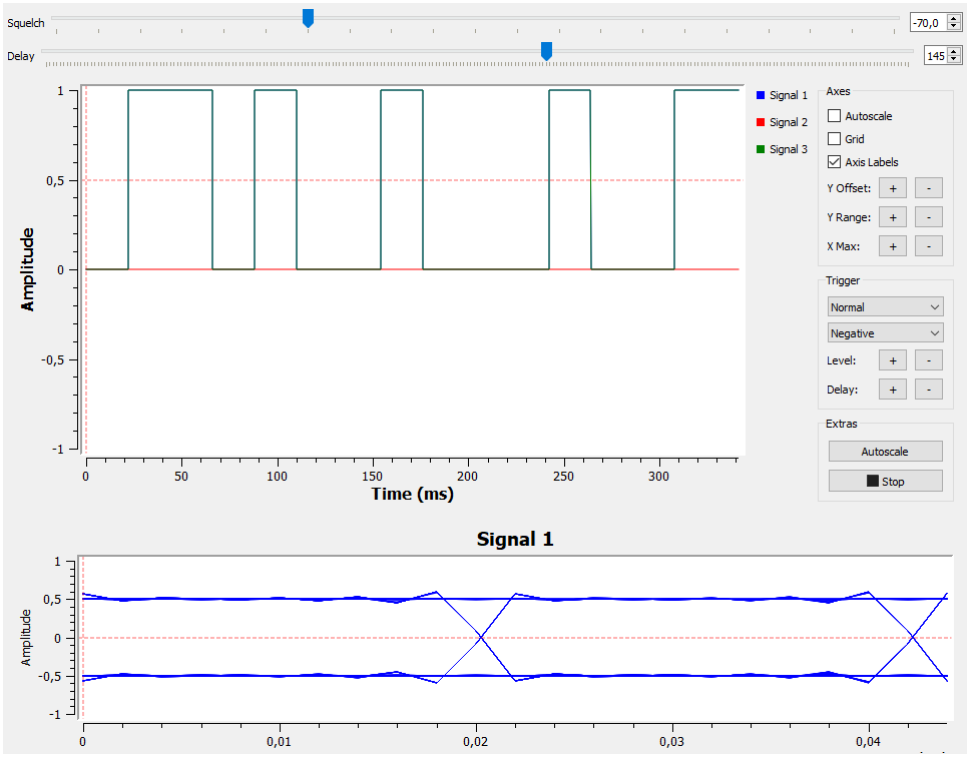


Рисунок 12.4. Задержка = 145