Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа компьютерных технологий и информационных систем

Отчёт по лабораторным работам

Дисциплина: Телекоммуникационные технологии.

Выполнил студент гр. 5130901/10101 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ Н.В. Богач
(подпись)

“11” февраля 2024 г.

Санкт-Петербург
2024

Оглавление

1. Лабораторная работа 1. Сигналы и звуки.

1.1. Упражнение 1.2.

Скачаем с сайта https://freesound.org/ образец звука и различными способами исследуем его. Для удобной работы с сигналами здесь, и в дальнейших работах будем использовать библиотеку thinkdsp.

Откроем скачанный файл, нормализуем и выведем на экран. Код будет выглядеть следующим образом:

```
from thinkdsp import read_wave

wave = read_wave('680840__seth_makes_sounds__homemade.wav')
wave.normalize()
wave.make_audio()
wave.plot()
```

Результат выполнения кода выглядит следующим образом:

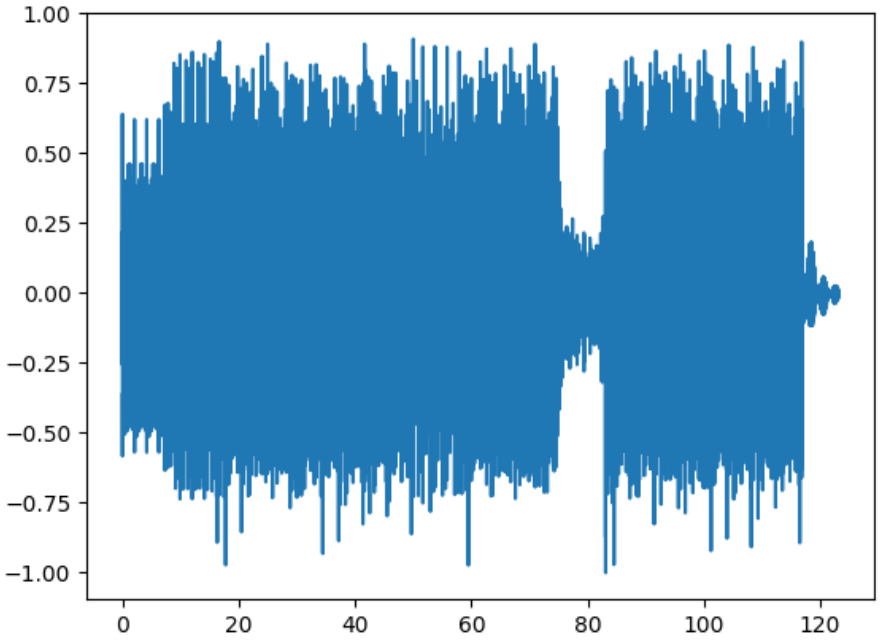

Рис. 1.1. Спектрограмма аудио файла.

Данный отрезок слишком длинный, выделим из него отрезок длинной пол секунды, начиная с 40 секунды аудио файла. Выведем полученный сегмент на экран, используя следующий код:

```
segmet = wave.segment(start=40.0, duration=0.5)
segmet.make_audio()
segmet.plot()
```

Спектрограмма заданного сегмента выглядит следующим образом:

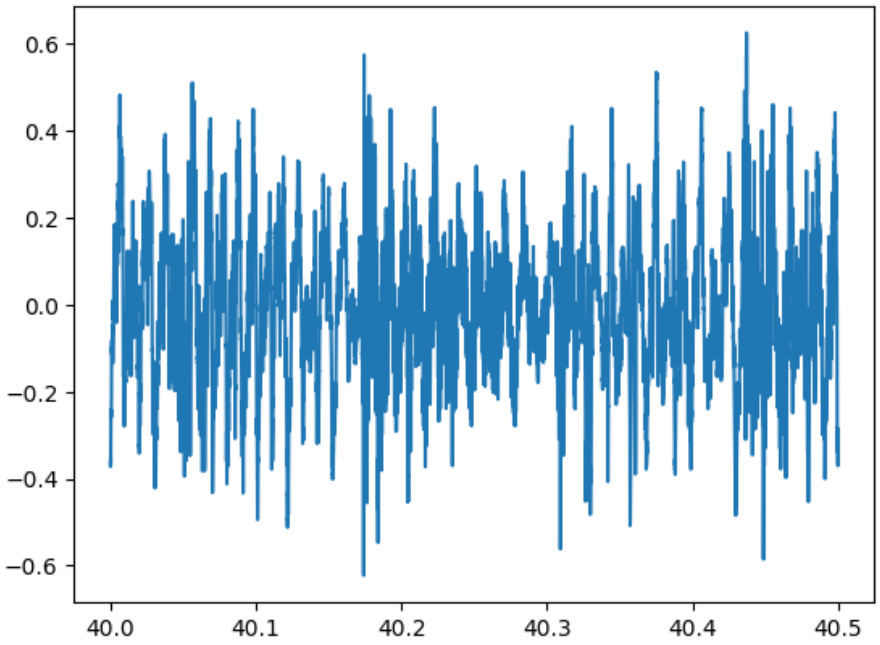

Рис. 1.2. Спектрограмма аудио файла с 40.0 по 40.5 секунды.

Разложим полученный отрезок в спектр и выведем на экран. Код будет выглядеть следующем образом:

```
1  spectrum = segment.make_spectrum()
2  spectrum.plot(high=5000)
```

Этот код выведет спектр до 5000 частоты т.к. далее частоты равны примерно нулю:

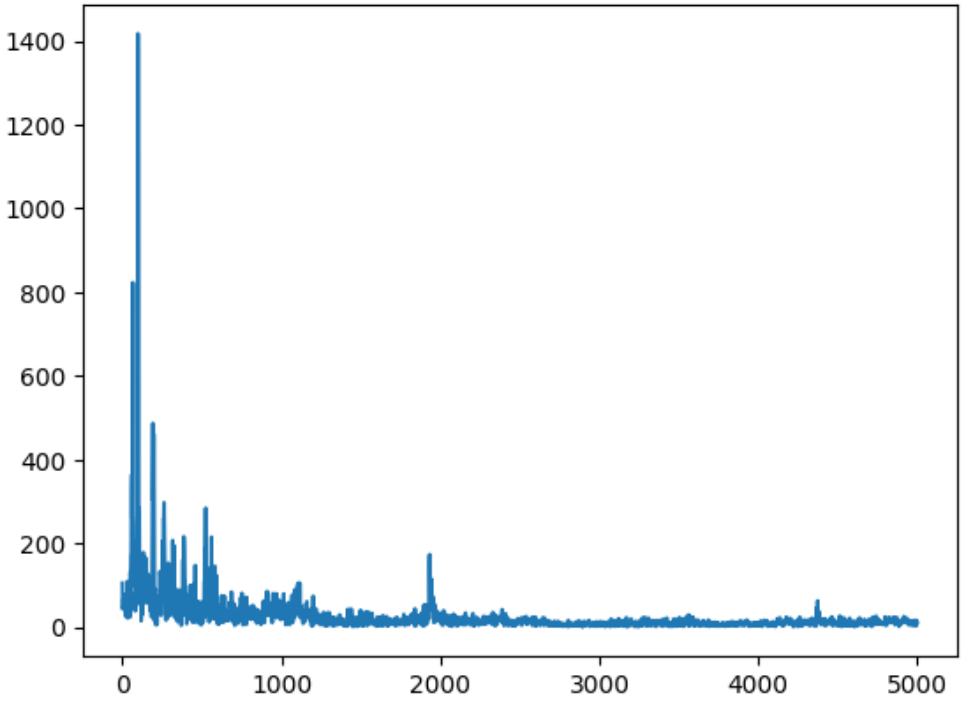

Рис. 1.3. Результат разложения сегмента в спектр.

Доминантной частотой в этом отрывке является 98 Гц.

Теперь поэкспериментируем с функциями high_pass, low_pass и band_stop, которые фильтруют гармоники.

Начнем с low_pass:

```
spectrum.make_wave().make_audio()
spectrum.low_pass(2000)
spectrum.plot(high=5000)
spectrum.make_wave().make_audio()
```

Данный код сохраняет музыкальный фрагмент (для дальнейшего сравнения), после чего применяет функцию low_pass и выводит его спектр на экран, а также опять сохраняет фрагмент. Полученный спектр выглядит следующим образом:

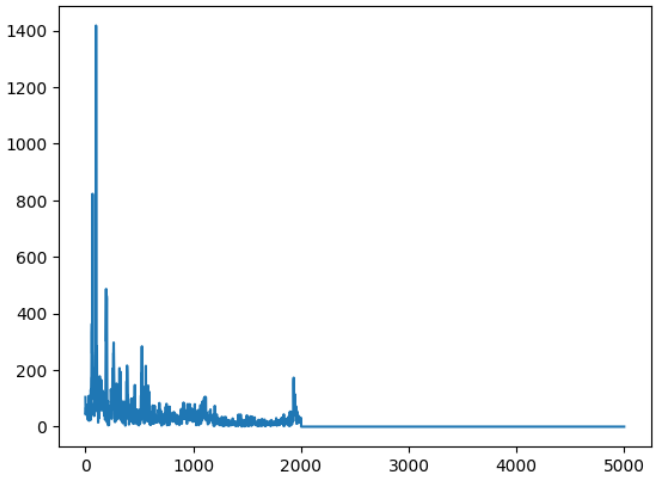

Рис. 1.4. Спектр фрагмента после применения low_pass.

Как видно из рисунка выше, данная функция полностью убрала частоты, выше 2000. Таким образом звук стал более «глухим» и «отдаленным».

Теперь к исходному сегменту применим метод high_pass:

```
spectrum.make_wave().make_audio()
spectrum.high_pass(1000)
spectrum.plot(high=5000)
spectrum.make_wave().make_audio()
```

Полученный спектр имеет следующий вид:

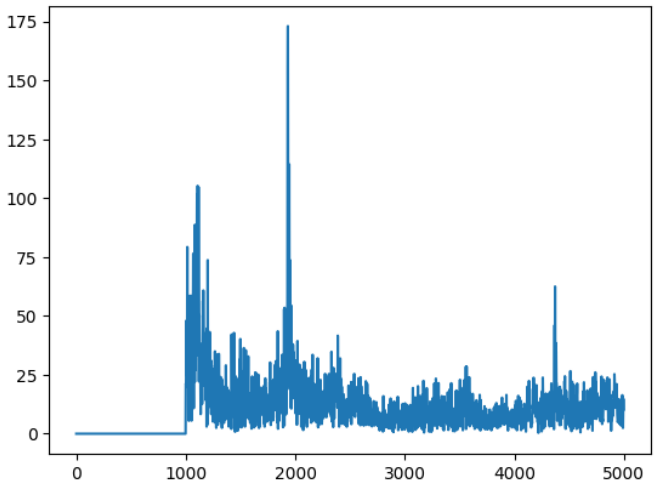

Рис. 1.5. Спектр фрагмента после применения high_pass.

Как видно по спектру, эта функция убирает все частоты ниже заданной. Таким образом звук сильно поменял свое звучание, став более шипящим и менее глубоким.
И последняя функция band_stop:

```
spectrum.make_wave().make_audio()
spectrum.band_stop(low_cutoff=100, high_cutoff=1000)
spectrum.plot(high=5000)
spectrum.make_wave().make_audio()
```

Полученный спектр выглядит следующим образом:

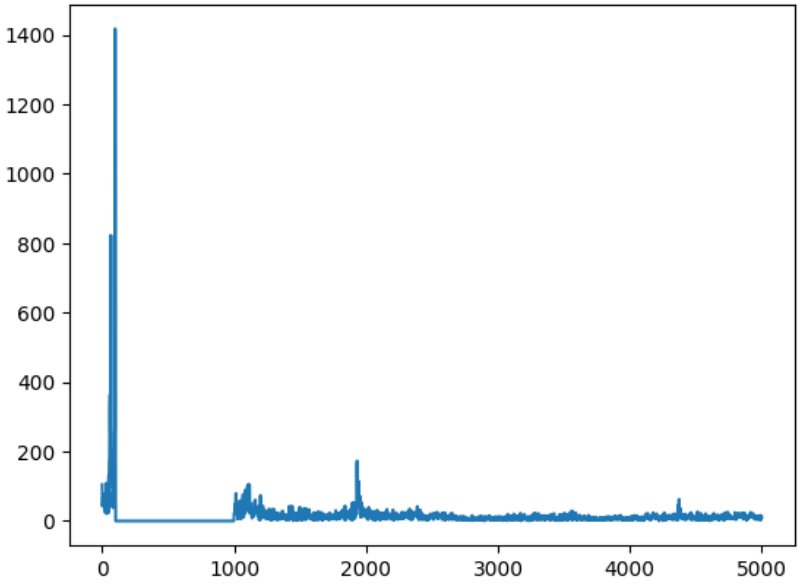

Рис. 1.6. Спектр фрагмента после применения band_stop.

Как мы видим, данная функция убирает частоты из заданного диапазона. Звук фрагмента при удалении частот со 100 Гц до 1000 Гц сильно изменился, в нем практически не слышны ударные.

1.2. Упражнение 1.3.

Создадим сигнал, состоящий из синусов, разной частоты, однако кратных одному числу, например 200:

```
from thinkdsp import SinSignal

signal = (SinSignal(freq=400, amp=1.0) +
          SinSignal(freq=600, amp=1.0) +
          SinSignal(freq=800, amp=1.0))
signal.plot()
```

Полученный сигнал имеет следующий вид:

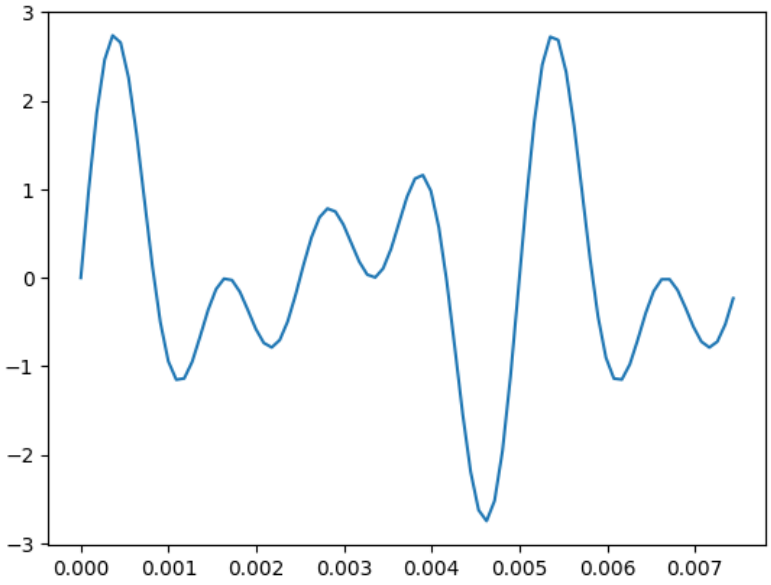

Рис. 1.7. Сигнал, полученный суммой синусов разной частоты.

Создадим файл для прослушивания этого звука, длинной 1 секунда:

```
wave = signal.make_wave(duration=1)
wave.apodize()
wave.make_audio()
```

Полученный звуковой файл является однотонным писком, похожим на звук гудка, но монотонного.

Выведем спектр полученного сигнала:

```
spectrum = wave.make_spectrum()
spectrum.plot(high=2000)
```

Результат выглядит следующим образом:

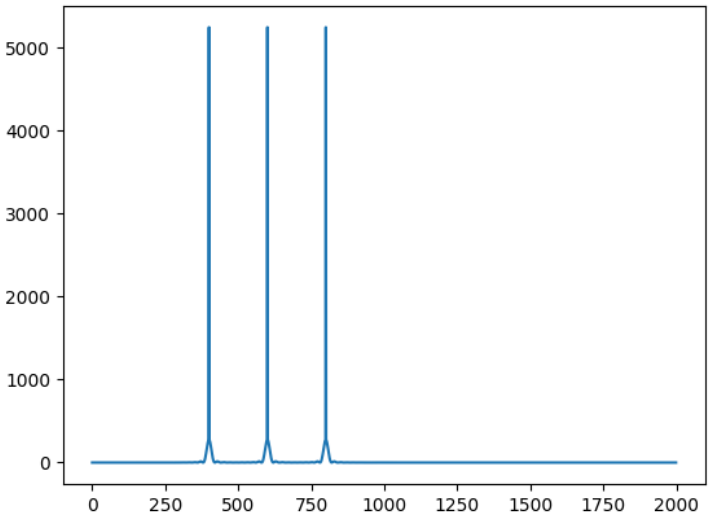

Рис. 1.8. Спектр сигнала, полученного суммой синусов разной частоты.

Как видим, спектр полностью соответствует ожидания, на нем пики находятся именно в тех частотах, которые мы указывали при создании.

Теперь изменим наш сигнал, добавив частоту, не кратную 200:

```
signal += SinSignal(freq=450, amp=1.0)
signal.plot()
signal.make_wave().make_audio()
```

Полученный сигнал имеет следующий вид:

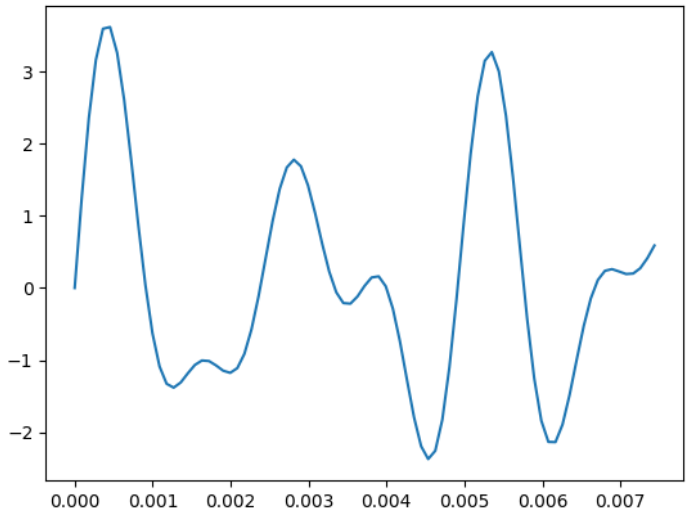

Рис. 1.9. Сигнал, после добавления синуса не кратной частоты.

Полученный сигнал сильно отличается от того, который был ранее. Так же аудио файл тоже чуть-чуть отличается. В монотонном звуке гудка различим какой-то посторонний периодический сигнал.

Выведем спектр полученного сигнала:

```
wave = signal.make_wave(duration=1)
wave.apodize()
spectrum = wave.make_spectrum()
spectrum.plot(high=2000)
```

Полученный спектр имеет следующий вид:

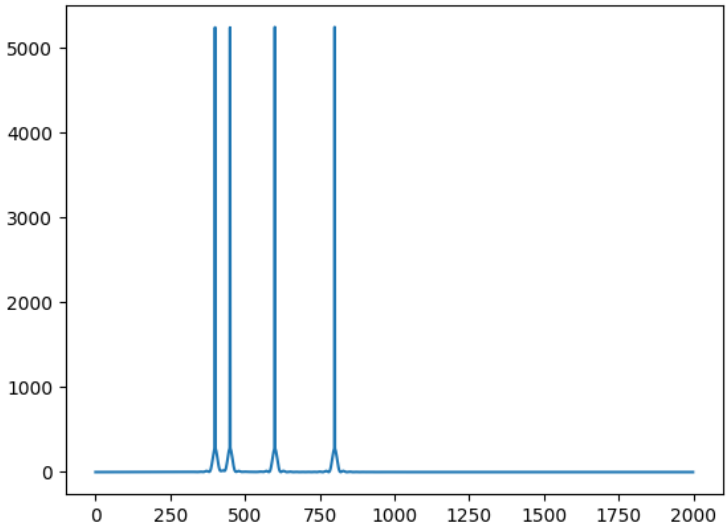

Рис. 1.10. Спектр сигнала, после добавления синуса не кратной частоты.

Как и ожидалось, в спектре появился добавленный ранее сигнал.

1.3. Упражнение 1.4.

Напишем функцию для ускорения и замедления аудио. Для начала прочитаем аудио фрагмент и выведем его на экран:

```
from thinkdsp import read_wave

wave = read_wave('680840__seth_makes_sounds__homemade.wav')
wave.normalize()
wave.plot()
wave.make_audio()
```

Спектрограмма будет выглядеть следующим образом:

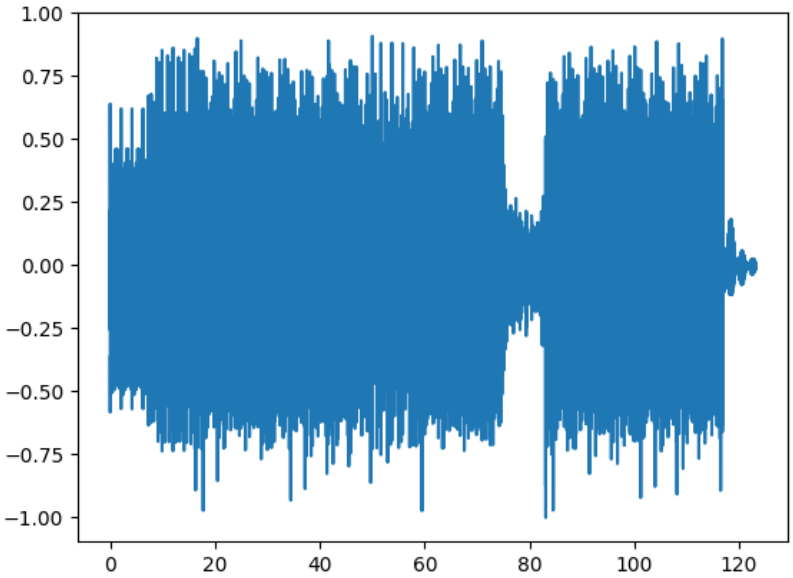

Рис. 1.11. Спектрограмма аудио файла.

Функция для ускорения будет выглядеть следующим образом:

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

Она изменяет ts (которое используется для корректного отображения временной шкалы в plot) и framerate, что, собственно, и ускоряет произведение.

Передадим функции значение 0.5, что эквивалентно ускорению в 2 раза:

```
stretch(wave, 0.5)
wave.plot()
wave.make_audio()
```

После выполнения мы получили аудио файл, который ускорен в 2 раза, как и ожидалось.

Посмотрим на полученную спектрограмму:

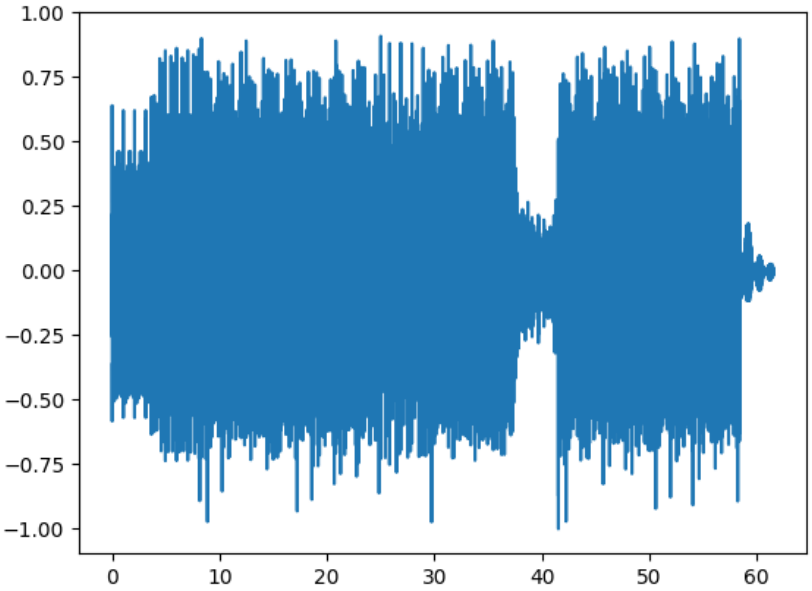

Рис. 1.12. Спектрограмма аудио файла после ускорения.

Как мы видим, полученная спектрограмма не отличается от исходной ничем, кроме длительности аудио фрагмента, он меньше в 2 раза.

2. Лабораторная работа 2. Гармоники.

2.1. Упражнение 2.2.

Разработаем класс, который бы наследовался от Sinusoid из thinkdsp, который позволял бы строить пилообразный сигнал (нарастает от -1 до 1, а затем резко падает до -1). Переопределить необходимо только функцию evaluate:

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

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
```

Здесь:

cycles – число циклов со времени старта.

frac – дробная часть, растущая от 0 до 1 за период.

unbias – сдвигает frac так, что он растет от -0.5 до 0.5.

normalize – нормализует функцию, чтоб она росла от $-self.amp$ до *self.amp*.

Создадим экземпляр этого класса и сразу же получим из него Wave, после чего выведем его часть на экран, дабы проверить, что функция реализованная корректно:

```
sawtooth = SawtoothSignal(200).make_wave(duration=0.5, framerate=40000)
sawtooth.segment(start=0, duration=0.005 * 4).plot()
```

Результат запуска выглядит следующим образом:

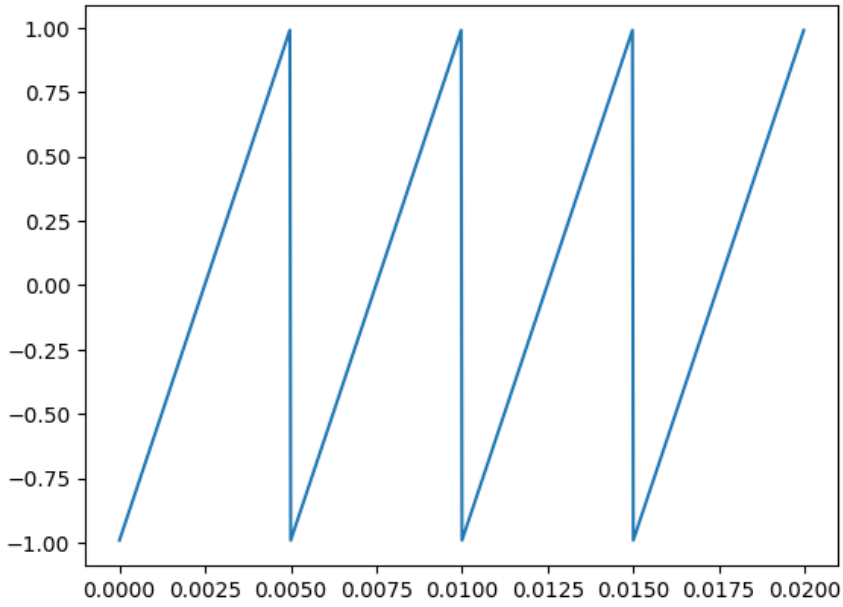

Рис. 2.1. Пилообразный сигнал.

Создадим спектр этого сигнала, выведем на экран, а также посмотрим наибольшие 10 пиков, дабы изучить каким образом частота зависит от амплитуды:

```
spectrum = sawtooth.make_spectrum()
spectrum.plot(high=10000)
spectrum.peaks()[:10]
```

График будет выглядеть таким образом:

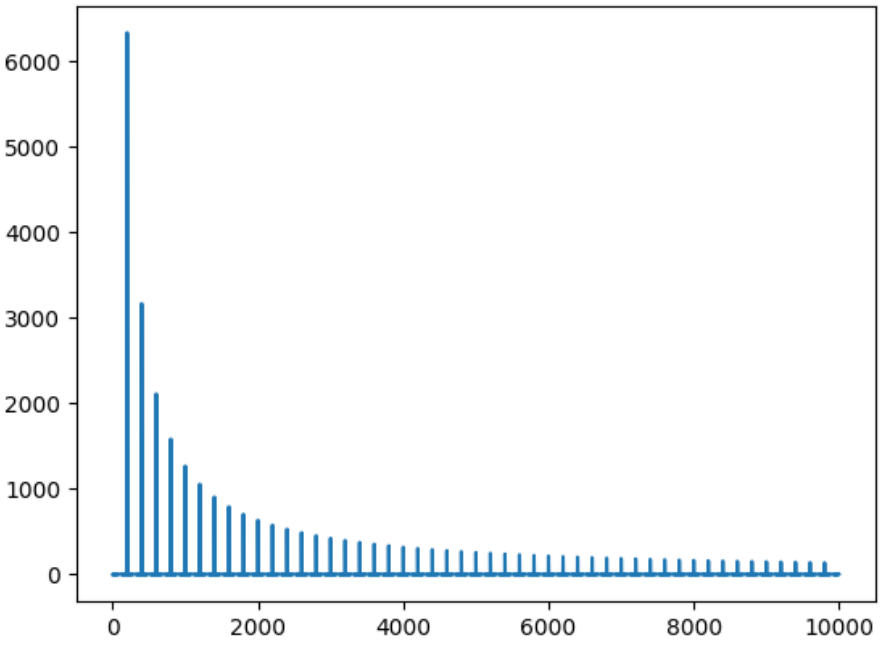

Рис. 2.2. Спектр пилообразного сигнала.

А также мы получаем следующий массив пиков:

```
[(6336.586158412468, 200.0),
 (3168.547531644226, 400.0),
 (2112.647887262727, 600.0),
 (1584.783147437211, 800.0),
 (1268.132547579567, 1000.0),
 (1057.089208734752, 1200.0),
 (906.3930765164864, 1400.0),
 (793.4141558611690, 1600.0),
 (705.5802559636873, 1800.0),
 (635.3480882678591, 2000.0)]
```

Как можно заметить, сигнал содержит как четные, так и нечетные гармоники, а также они уменьшаются пропорционально $1/f$.

Сравним полученный спектр пилообразного сигнала с прямоугольным:

```python
from thinkdsp import SquareSignal

sawtooth.make_spectrum().plot(high=10000, color='gray')
square = SquareSignal(freq=200, amp=0.5).make_wave(duration=0.5, framerate=40000)
sqere_spectrum = square.make_spectrum()
sqere_spectrum.plot(high=10000)
sqere_spectrum.peaks()[:10]
```

Стоит отметить, что прямоугольный сигнал создается с *amp=0.5*, чтоб выровнять спектрограмму для сравнения её с пилообразным сигналом.

Полученный график выглядит так:

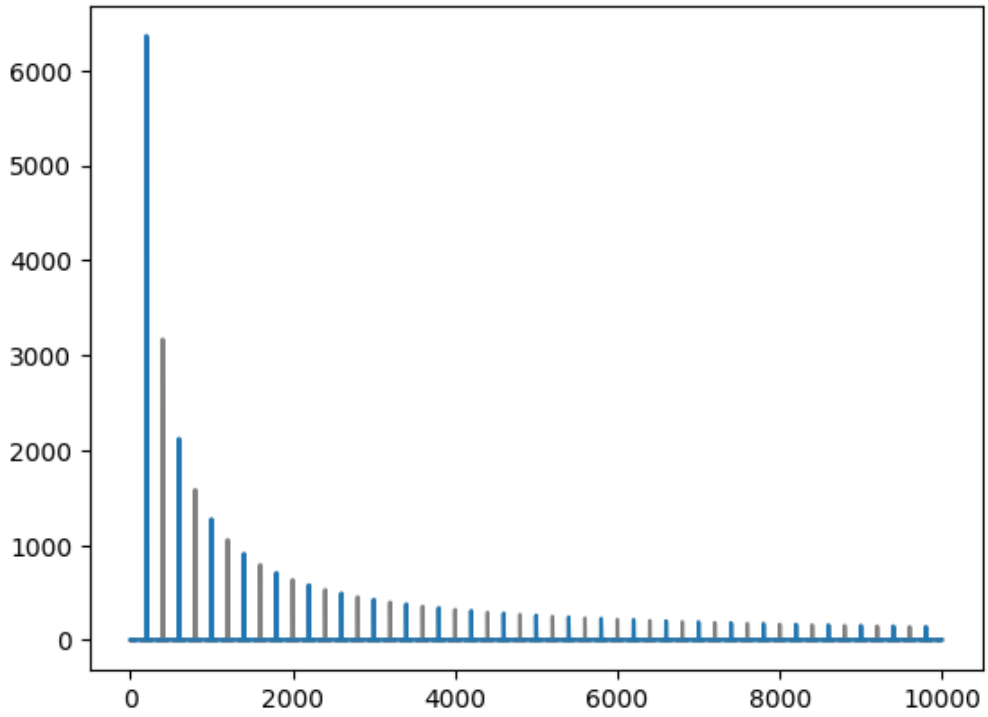

Рис. 2.3. Спектрограммы пилообразного и прямоугольного сигналов.

Для удобства сравнения так же проанализируем первые 10 пиков прямоугольного сигнала:

```
[(6366.41311530820, 200.0),
 (2122.71230545574, 600.0),
 (1274.31761659952, 1000.0),
 (910.967679013610, 1400.0),
 (709.300517302879, 1800.0),
 (581.126830719483, 2200.0),
 (492.527938505399, 2600.0),
 (427.675369582061, 3000.0),
 (378.189565961787, 3400.0),
 (339.219405574489, 3800.0)]
```

Стоит обратить внимание, что в отличии от пилообразного сигнала, прямоугольный сигнал имеет только нечетные гармоники, а вот зависимость падения от частоты сохраняется и пропорционально $1/f$.

Так же выполним аналогичное сравнение с треугольным сигналом:

```
from thinkdsp import TriangleSignal

sawtooth.make_spectrum().plot(high=10000, color='gray')
triangle = TriangleSignal(freq=200, amp=0.78).make_wave(duration=0.5, framerate=40000)
triangle_spectrum = triangle.make_spectrum()
triangle_spectrum.plot(high=10000)
triangle_spectrum.peaks()[:10]
```

Аналогично прямоугольному сигналу необходимо изменить *amp*, однако для треугольного сигнала это значение равно 0.78.

График выглядит следующим образом:

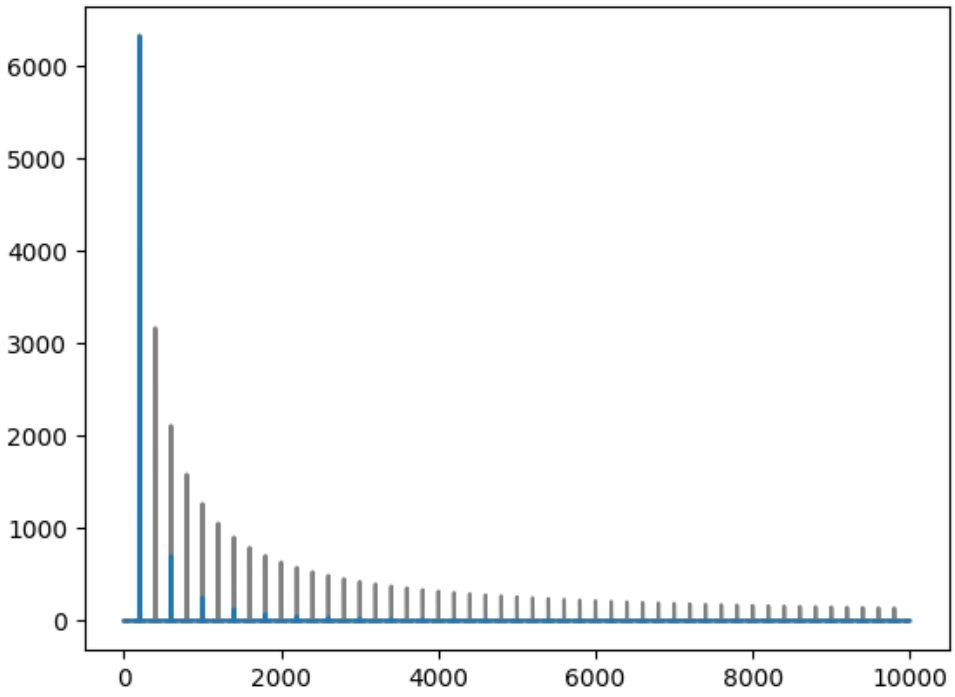

Рис. 2.4. Спектрограммы пилообразного и треугольного сигналов.

Для удобства сравнения так же проанализируем первые 10 пиков треугольного сигнала:

```
[(6322.961884943854, 200.0),
 (703.0137709503831, 600.0),
 (253.4183165242379, 1000.0),
 (129.5506855043607, 1400.0),
 (78.57692291999746, 1800.0),
 (52.77470536760347, 2200.0),
 (37.93526415250758, 2600.0),
 (28.62556659255658, 3000.0),
 (22.40446225716876, 3400.0),
 (18.04308559845682, 3800.0)]
```

Как мы видим, этот сигнал ведет себя совершенно отлично, от пилообразного. В первую очередь мы видим, что в нем присутствуют только нечетные гармоники, а также зависимость падения от частоты пропорциональна $1/f^2$.

2.2. Упражнение 2.3.

Создадим прямоугольный сигнал с частотой 1100 Гц и выборкой 10000 кадров в секунду. Отобразим получившийся спектр, а также первые 10 пиков:

```python
from thinkdsp import SquareSignal

square = SquareSignal(1100).make_wave(duration=0.5, framerate=10000)
square_spectrum = square.make_spectrum()
square_spectrum.plot()
square_spectrum.peaks()[:10]
```

Получившийся спектр выглядит таким образом:

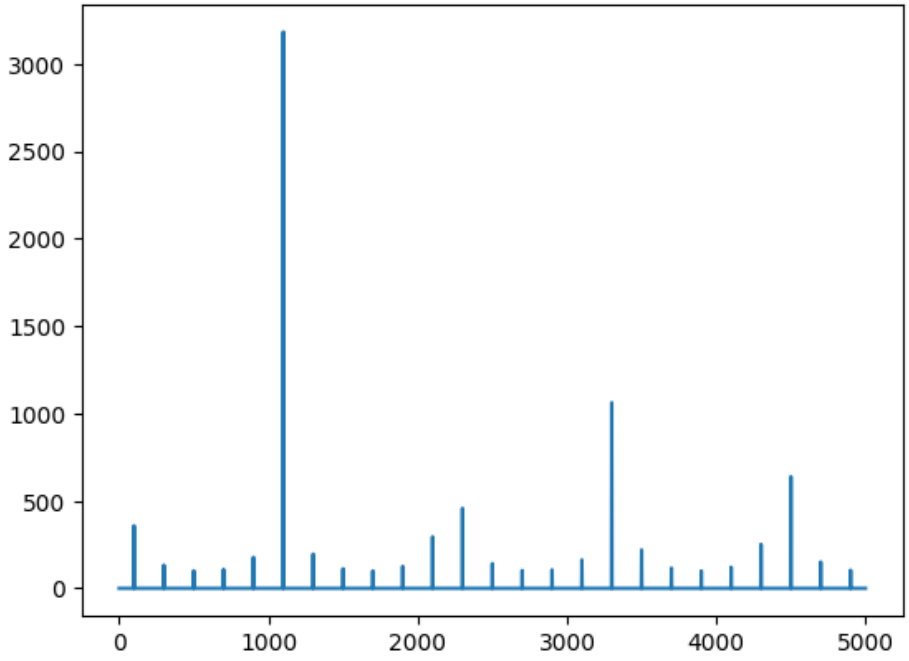

Рис. 2.5. Спектрограмма прямоугольного сигнала с частотой 1100 Гц и выборкой 10000 кадров в секунду.

А также для удобства анализа приведем старшие 10 пиков:

```
[(3183.622520909762, 1100.0),
 (1062.605379628311, 3300.0),
 (639.2453221499661, 4500.0),
 (458.4143857027373, 2300.0),
 (358.4343652372162, 100.0),
 (295.2134792809340, 2100.0),
 (251.7953698310349, 4300.0),
 (220.2689264585266, 3500.0),
 (196.4476698867248, 1300.0),
 (177.9095485479867, 900.0)]
```

Как мы помним, прямоугольный сигнал имеет только нечетные гармоники, а зависимость падения амплитуды от частоты пропорциональна $1/f$.

Ожидается, что гармоники будут на 3300, 5500, 7700 и 9900 Гц. Как мы видим, пики есть на 1100 и 3300 Гц, однако дальше наблюдается эффект биения, поэтому вместо 5500 мы получаем лишь 4500 (10000 – 5500), а следующая гармоника вместо 7700 получается 2300 (10000 – 7700). Из-за этого сигнал звучит совершенно по-другому, а именно появляются лишние низкие частоты (например, 100), а также посторонние не кратные частоты (2300), которые сильно выбиваются.

Продемонстрируем это, создав аудио файл, исходного прямоугольного сигнала, а также две синусоиды на 2300 и 100 Гц и убедимся, что они достаточно заметны:

```
square.make_audio()

from thinkdsp import SinSignal

SinSignal(2300).make_wave(duration=0.5, framerate=10000).make_audio()
SinSignal(100).make_wave(duration=0.5, framerate=10000).make_audio()
```

При прослушивании этих записей действительно заметно, что они сильно выбиваются из исходного сигнала.

2.3. Упражнение 2.4.

Создадим треугольный сигнал и выведем его график на экран:

```
from thinkdsp import TriangleSignal

triangle = TriangleSignal().make_wave(duration=0.01)
triangle.plot()
```

График выглядит следующим образом:

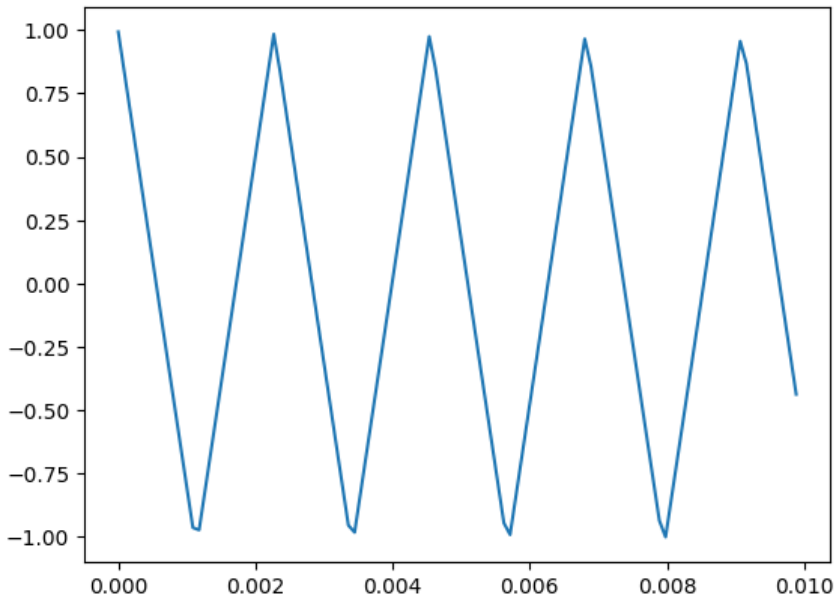

Рис. 2.6. График треугольного сигнала.

Теперь получим спектр этого сигнала и выведем первый элемент массива hs, который является результатом БПФ:

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

Результат примерно равен нулю:

```
(1.0436096431476471e-14+0j)
```

Изменим его значение на 100 и посмотрим на результат:

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

В результате получаем следующий график:

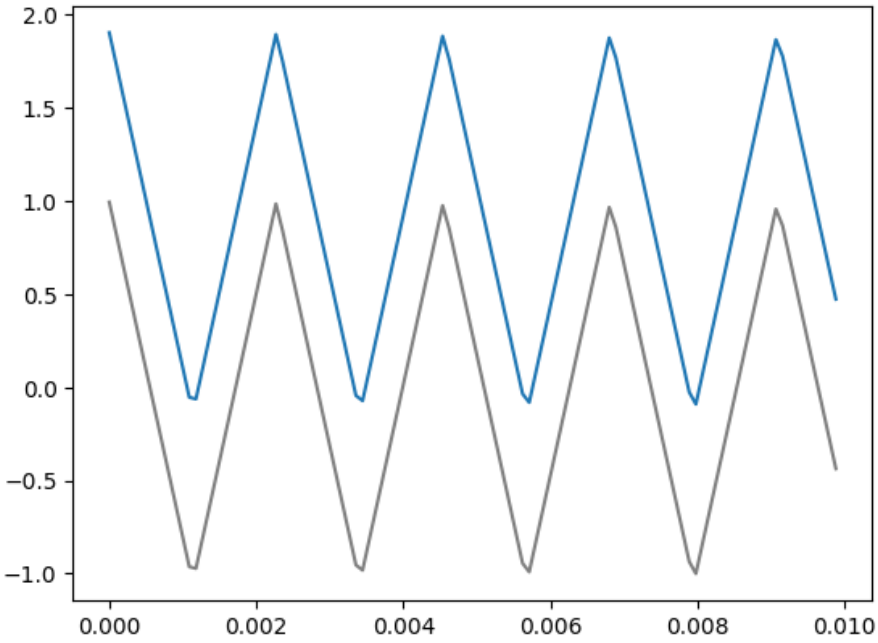

Рис. 2.7. График треугольного сигнала, после изменения hs.

Как можно заметить, получившийся сигнал отличается от исходного только вертикального смещения.

2.4. Упражнение 2.5.

Напишем функцию *filter_spectrum*, которая принимает спектр и изменяет его, выполняя деление каждый элемент *hs*, на соответствующую частоты из *fs*:

```
def filter_spectrum(spectrum):
    spectrum.hs[1:] /= spectrum.fs[1:]
    spectrum.hs[0] = 0
```

Создадим треугольный сигнал, выведем его в виде аудио для дальнейшего сравнения, после чего вызовем нашу функцию фильтрации. Выведем спектрограмму до и после, а также аудио файл после использования функции:

```
wave = TriangleSignal(freq=440).make_wave(duration=0.5)
wave.make_audio()
spectrum = wave.make_spectrum()
spectrum.plot(high=10000, color='gray')
filter_spectrum(spectrum)
spectrum.scale(440)
spectrum.plot(high=10000)
filtered = spectrum.make_wave()
filtered.make_audio()
```

Получившийся график выглядит следующим образом:

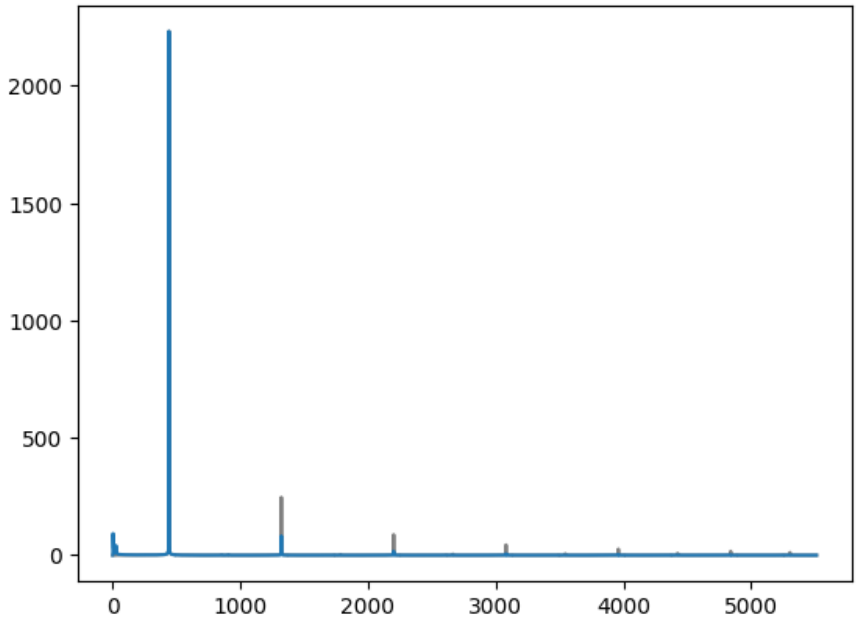

Рис. 2.8. Спектрограмма до и после использования функции.

Как можно заметить, чем больше частота, тем меньше становится пик после использования функции. Это логично, ведь деление происходит именно на частоту. Результат похож на low_pass фильтр.

Так же при прослушивании полученных аудио, мы получаем схожий результат.

2.5. Упражнение 2.6.

Создадим сигнал, в котором есть как четные, так и нечетные гармоники, которые спадают пропорционально $1/f^2$. Сигнал будем собирать, используя несколько синусоид:

```
from thinkdsp import SinSignal
import numpy as np

freqs = np.arange(500, 9500, 500)
amps = (1 / freqs**2) * 10 ** 5
signal = sum(SinSignal(freq, amp) for freq, amp in zip(freqs, amps))
wave = signal.make_wave(duration=0.5, framerate=20000)
wave.segment(duration=0.01).plot()
wave.make_audio()
```

Как мы видим. Частоты будут изменяться от 500 до 9000 с шагом 500. Амплитуда вычисляется, путем деления $1/f^2$, как это требует задание.

График сигнала будет выглядеть следующим образом:

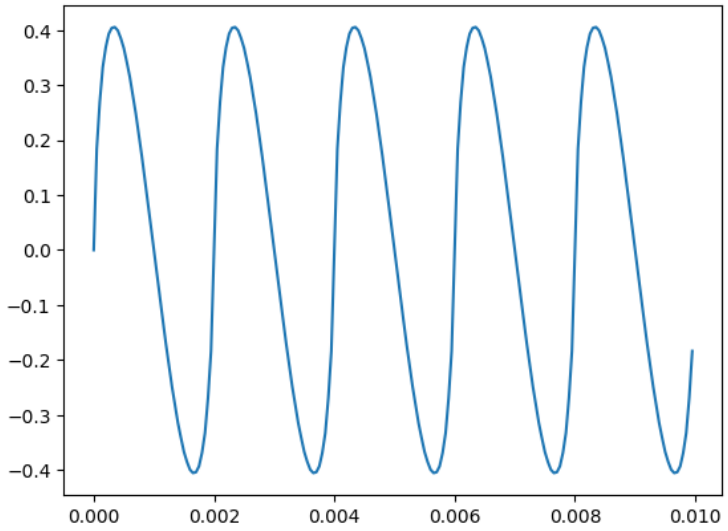

Рис. 2.9. Сигнал с четными и нечетными гармониками и амплитудой пропорциональной $1/f^2$

Убедимся, что полученный сигнал соответствует требованиям, выведем его спектрограмму:

```
spectrum = wave.make_spectrum()
spectrum.plot()
spectrum.peaks()[:10]
```

График выглядит следующим образом:

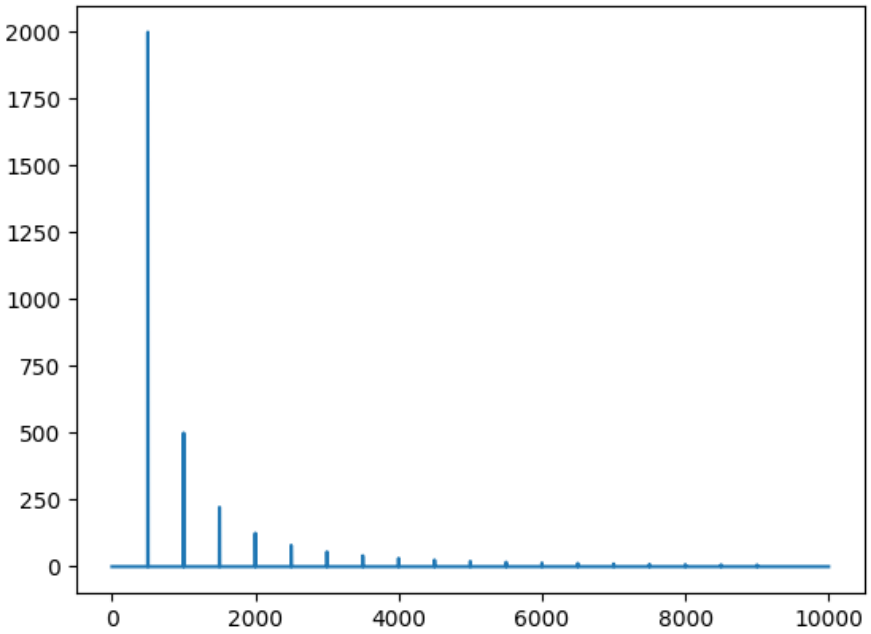

Рис. 2.10. Спектрограмма сигнала.

Для удобства анализа посмотрим на первые 10 пиков сигнала:

```
[(2000.0, 500.0),
 (500.00, 1000.0),
 (222.22, 1500.0),
 (125.00, 2000.0),
 (80.000, 2500.0),
 (55.555, 3000.0),
 (40.816, 3500.0),
 (31.250, 4000.0),
 (24.691, 4500.0),
 (20.000, 5000.0)]
```

Как мы видим, спектрограмма соответствует требованиям задания.

3. Лабораторная работа 3. Апериодические сигналы.

3.1. Упражнение 3.1.

Выполним сравнение различных оконных функций, а именно стандартной hamming и bartlett, blackman, hanning.

Создадим сначала синусоидальный сигнал, с частотой 440 и сделаем, чтоб сигнал начинался с 0, а заканчивался в 1. Выведем его спектрограмму:

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

signal = SinSignal(freq=440)
duration = signal.period * 30.25
wave = signal.make_wave(duration)
spectrum = wave.make_spectrum()
spectrum.plot(high=880)
```

Результат выглядит следующим образом:

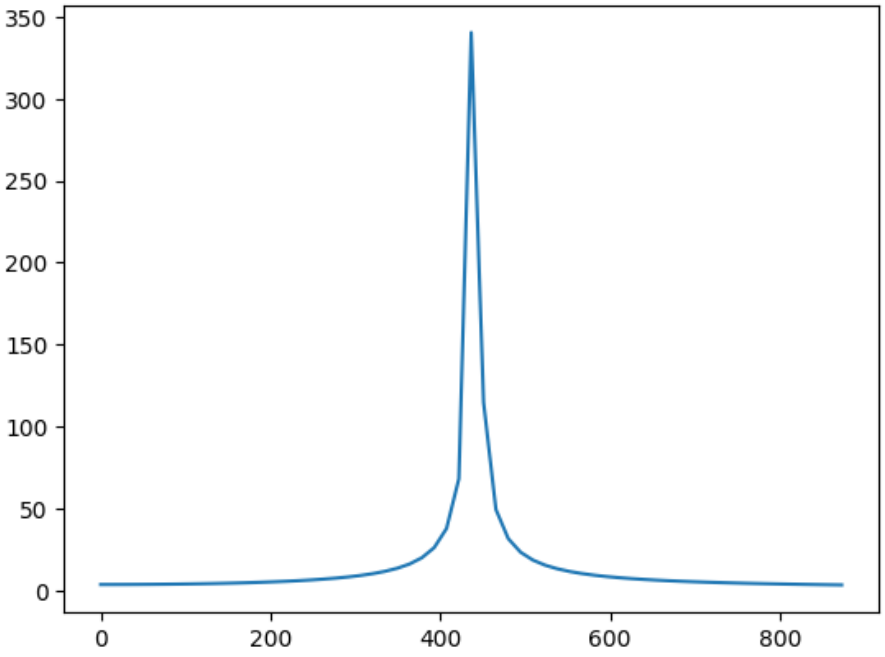

Рис. 3.1. Спектрограмма сигнала.

А теперь применим к этой спектрограмме различные оконные функции:

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

    spectrum = wave.make_spectrum()
    spectrum.plot(high=880, label=window_func.__name__)
plt.legend(loc='best')
plt.xlabel('Frequency (Hz)')
```

Результат выглядит следующим образом:

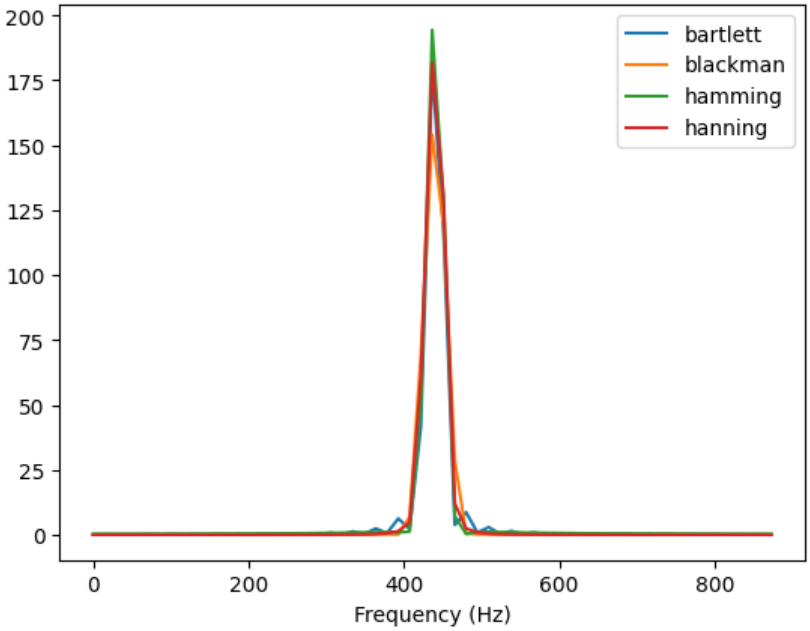

Рис. 3.2. Спектрограммы после применения оконных функций.

Как мы видим, функция Хемминга показывает себя лучше всего, именно поэтому она выбрана как универсальный вариант.

3.2. Упражнение 3.2.

Создадим класс, расширяющий Chirp для создания увеличивающегося пилообразного сигнала:

```
from thinkdsp import Chirp
from thinkdsp import normalize, unbias, PI2, decorate
import numpy as np

class SawtoothChirp(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 =  normalize(unbias(frac), self.amp)
        return ys
```

Эта функция является совмещением функции, созданной в 2.1 и функции для создания Chirp. Создадим экземпляр этого класса с начальной частотой 220 и конечной 880:

```
signal = SawtoothChirp(start=220, end=880)
wave = signal.make_wave(duration=1, framerate=4000)
wave.apodize()
wave.segment(duration=0.05).plot()
wave.make_audio()
```

Убедимся в том, что полученная запись соответствует ожиданиям:

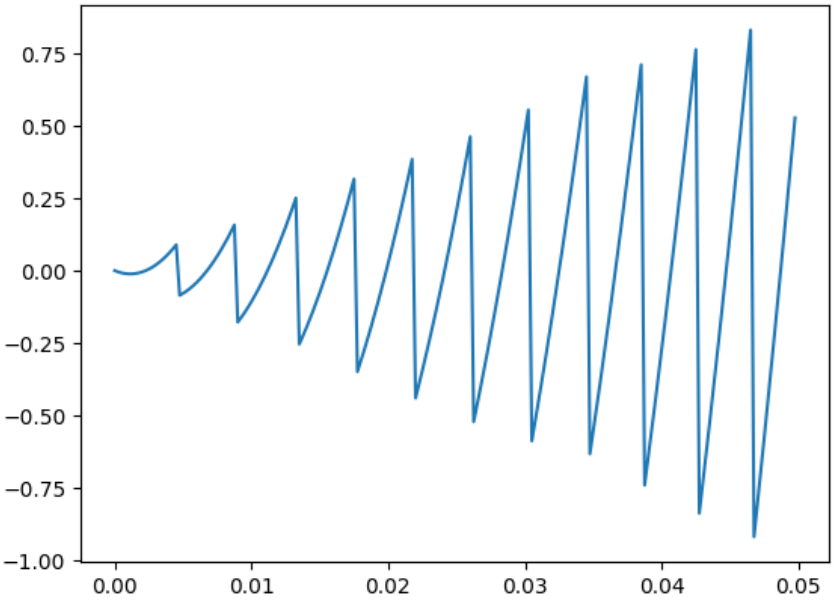

Рис. 3.3. Пилообразный чирп.

Как мы видим, это действительно пилообразный чирп, так же это слышно по аудио записи.
Создадим спектрограмму заданного сигнала:

```
sp = wave.make_spectrogram(256)
sp.plot()
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

Получившаяся спектрограмма выглядит следующим образом:

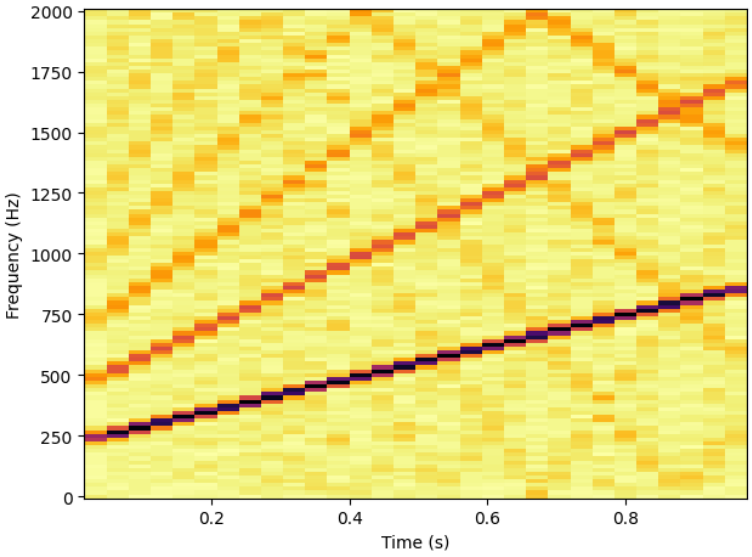

Рис. 3.4. Спектрограмма пилообразного сигнала.

Здесь стоит отметить, что в связи с маленьким фреймрейтом мы получаем достаточно шумный спектр из-за заворота (биения), этот эффект хорошо заметен, как «отскакивающие» от верхней границы темные участки.

3.3. Упражнение 3.3.

Попытаемся предположить, какой будет иметь вид спектр пилообразного чирпа с начальной частотой 2500, конечной 3000 и фреймрейтом 20000.

Поскольку пилообразный сигнал содержит как четные, так и нечетные гармоники, а также они уменьшаются пропорционально $^1/_f$ можно предположить, что первый пик будет размазан между начальной и конечной частотой, второй будет в 2 раза меньше на частотах с 5000 до 6000, а третий будет на частотах с 7500 до 9000 Гц и будет в 3 раза меньше пика с 2500 до 3000. Убедимся в этом, создав необходимый чирп:

```
from lab_3_2 import SawtoothChirp

signal = SawtoothChirp(start=2500, end=3000)
wave = signal.make_wave(duration=1, framerate=20000)
wave.make_spectrum().plot(scalex=500)
wave.make_audio()
```

Получившаяся спектрограмма выглядит следующим образом:

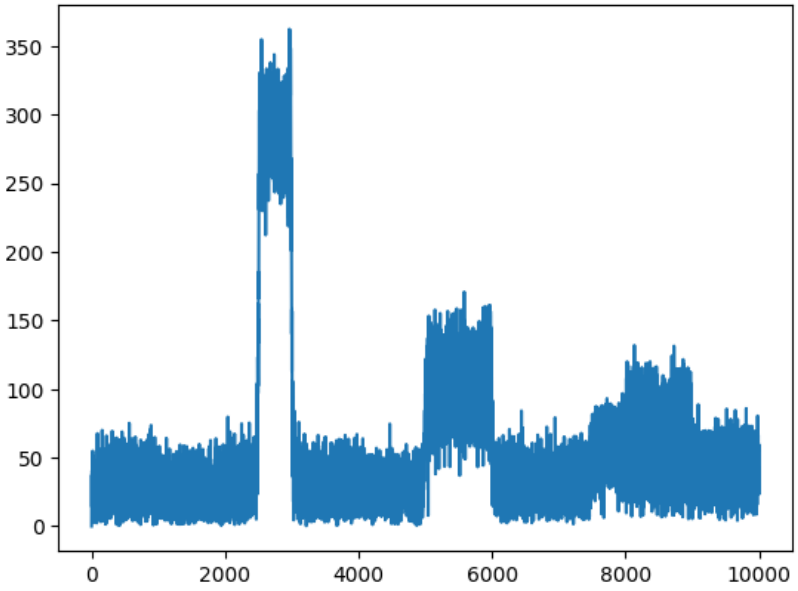

Рис. 3.5. Спектрограмма пилообразного чирпа.

Как мы и ожидали спектрограмма полностью соответствует ожиданиям.

3.4. Упражнение 3.4.

Скачаем глиссандо (нота, меняющаяся от одной высоты к другой) и создадим его спектрограмму:

```
from thinkdsp import read_wave, decorate
wave = read_wave('code_72475__rockwehrmann__glissup02.wav')
wave.make_audio()
wave.make_spectrogram(512).plot(high=5000)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

Результирующая спектрограмма выглядит следующим образом:

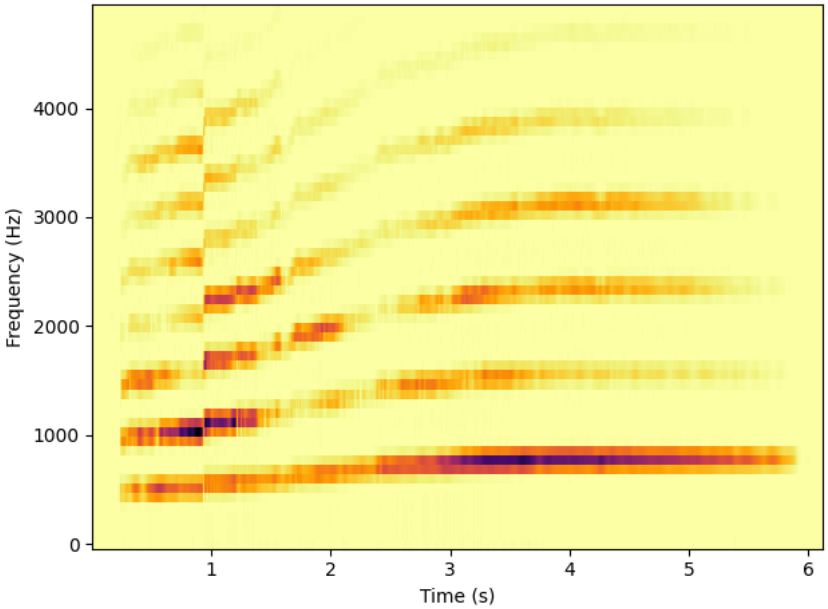

Рис. 3.6. Спектрограмма глиссандо.

Как можно заметить, частота меняется вместе с изменением высоты ноты (под конец в произведении нота почти не менялась, поэтому и спектрограмма сохраняла свое состояние).

3.5. Упражнение 3.5.

Создадим класс, который будет имитировать глиссандо на тромбоне, при постоянной скорости изменения трубы, если частота звука обратно пропорциональна длине:

```
1  from thinkdsp import Chirp, decorate, PI2
2  import numpy as np
3
4  class TromboneGliss(Chirp):
5
6      def evaluate(self, ts):
7          l1, l2 = 1.0 / self.start, 1.0 / self.end
8          lengths = np.linspace(l1, l2, len(ts))
9          freqs = 1 / lengths
10
11         dts = np.diff(ts, prepend=0)
12         dphis = PI2 * freqs * dts
13         phases = np.cumsum(dphis)
14         ys = self.amp * np.cos(phases)
15         return ys
```

В строках с 7 по 9 имитируется изменение длины кулисы тромбона, а далее стандартное объявление функции evaluate.

```
low = 262
high = 349
signal = TromboneGliss(high, low)
wave1 = signal.make_wave(duration=1)
wave1.apodize()
signal = TromboneGliss(low, high)
wave2 = signal.make_wave(duration=1)
wave2.apodize()
wave = wave1 | wave2
sp = wave.make_spectrogram(1024)
sp.plot(high=500)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
wave.make_audio()
```

Получившаяся спектрограмма выглядит следующим образом:

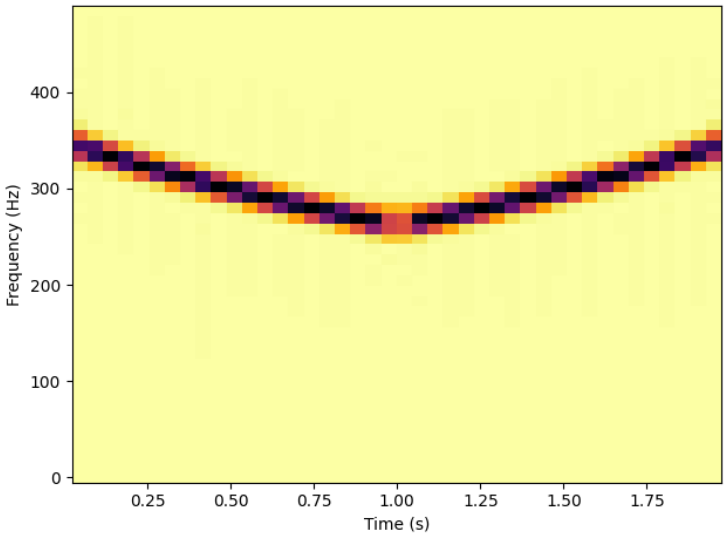

Рис. 3.7. Спектрограмма звука тромбона от C3 до F3 и обратно.

Кажется, что этот сигнал ближе к линейному, однако увеличим разброс до с 1000 Гц до 263:

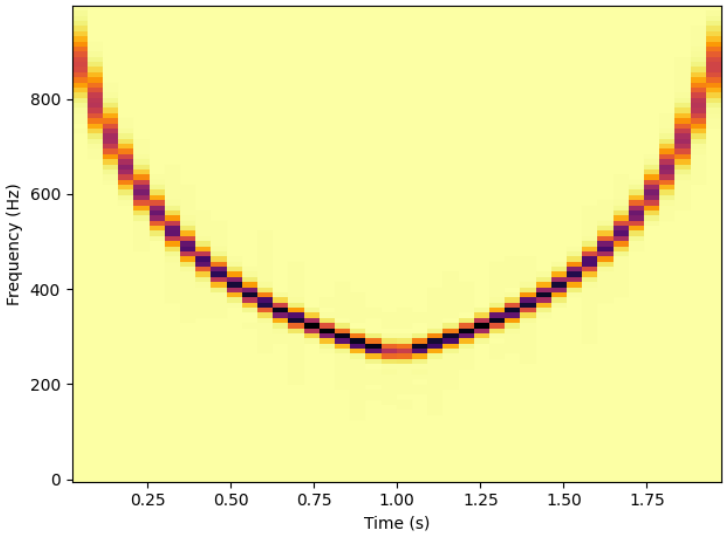

Рис. 3.8. Спектрограмма звука тромбона от 1000 до F3 и обратно.

Как мы видим, теперь сигнал ближе к экспоненциальному, нежели линейному т.к. функция обратно пропорциональна длине, а эта функция степенная, а не линейная.

3.6. Упражнение 3.6.

Далее возьмем аудио с гласными звуками и посмотрим на их спектрограммы, попробуем понять, как они отличаются.

Для начала загрузим аудио и посмотрим на полную спектрограмму:

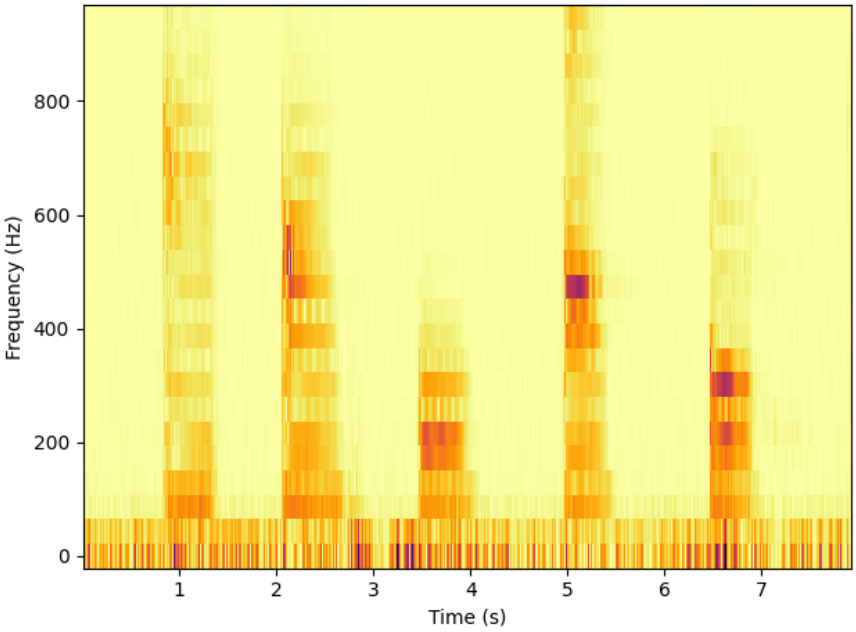

Рис. 3.9. Спектрограмма гласных звуков.

Теперь обрежем первый звук ‘а’ и посмотрим на его спектрограмму:

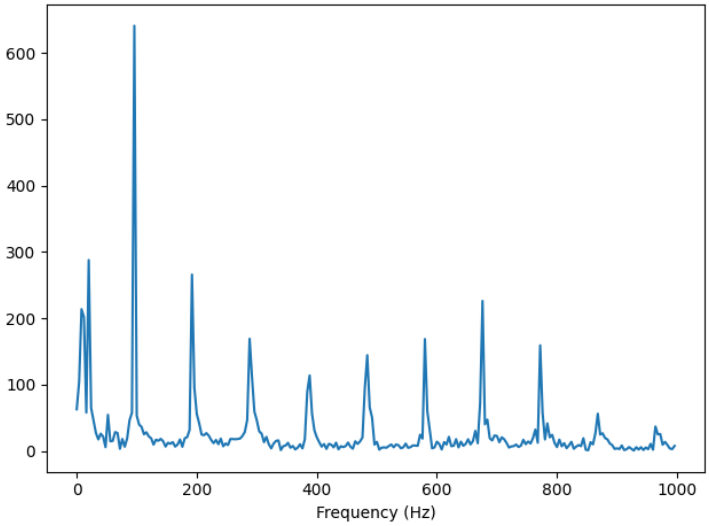

Рис. 3.10. Спектрограмма звука 'а'.

Теперь обрежем второй звук ‘э’ и посмотрим на его спектрограмму:

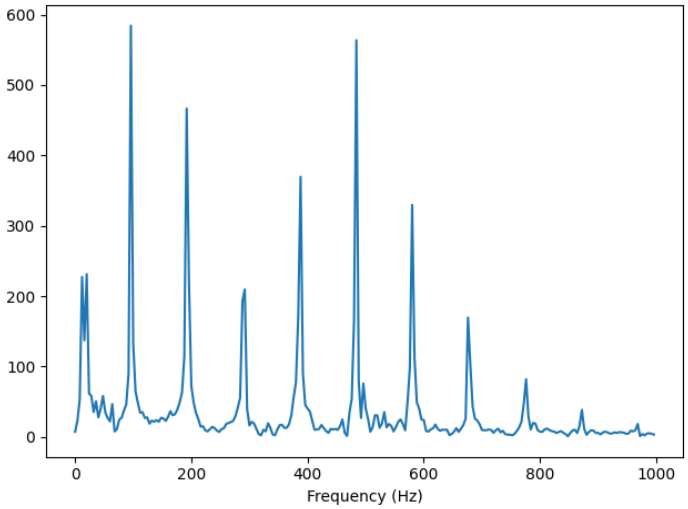

Рис. 3.11. Спектрограмма звука 'э'.

Теперь обрежем второй звук ‘и’ и посмотрим на его спектрограмму:

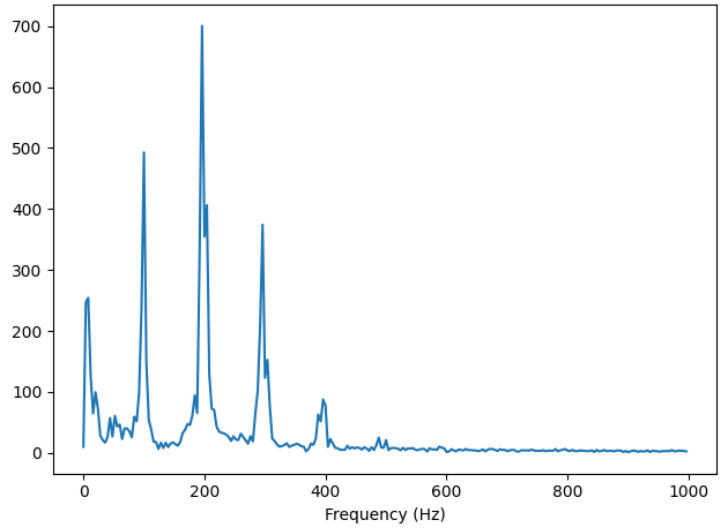

Рис. 3.12. Спектрограмма звука 'и'.

Теперь обрежем второй звук ‘о’ и посмотрим на его спектрограмму:

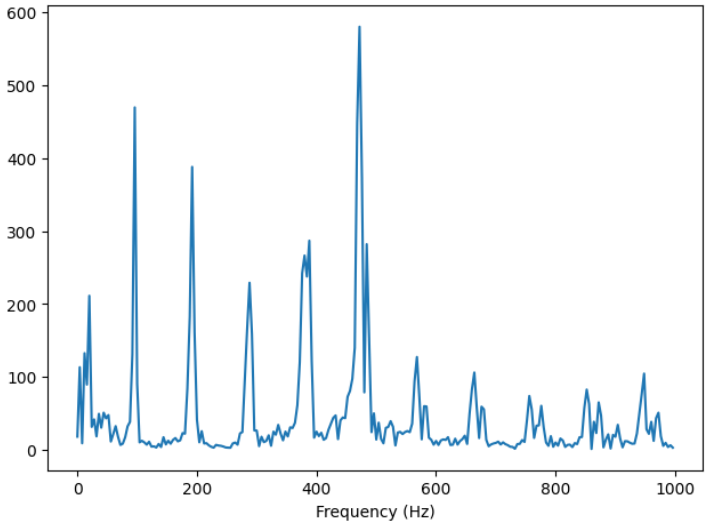

Рис. 3.13. Спектрограмма звука 'о'.

Теперь обрежем второй звук ‘у’ и посмотрим на его спектрограмму:

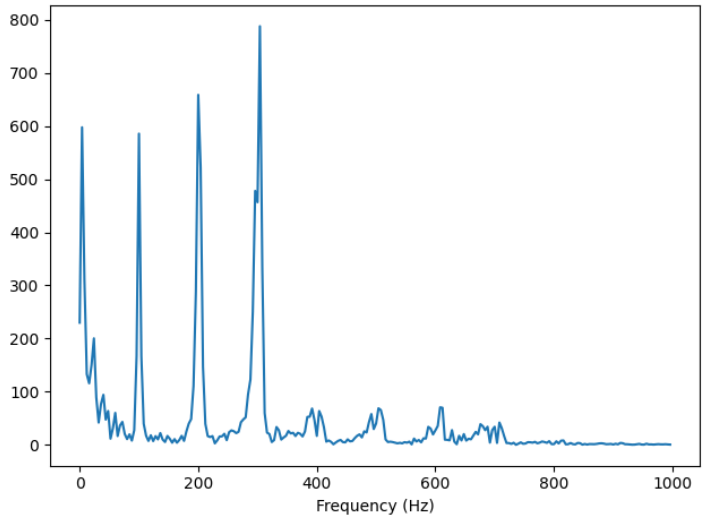

Рис. 3.14. Спектрограмма звука 'у'

Как можно заметить несмотря на то, что звуки звучат достаточно похоже их спектрограммы сильно отличаются и дают возможность понять, какой действительно звук был произнесен.

Вероятно, именно этот метод позволяет распознавать речь и отдельные звуки в современных голосовых помощниках.

4. Лабораторная работа 4. Шум.

4.1. Упражнение 4.1.

На сайте http://asoftmurmur.com/about/ выполним скачивание шумов природы, например звук северного моря. Обрежем его первые 1.5 секунды и выведем его спектр:

```
from thinkdsp import read_wave, decorate

wave = read_wave('13793__soarer__north-sea.wav')
segment = wave.segment(start=0.0, duration=1.5)
spectrum = segment.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Frequency (Hz)',
         ylabel='Power')
segment.make_audio()
```

В результате получим следующий спектр:

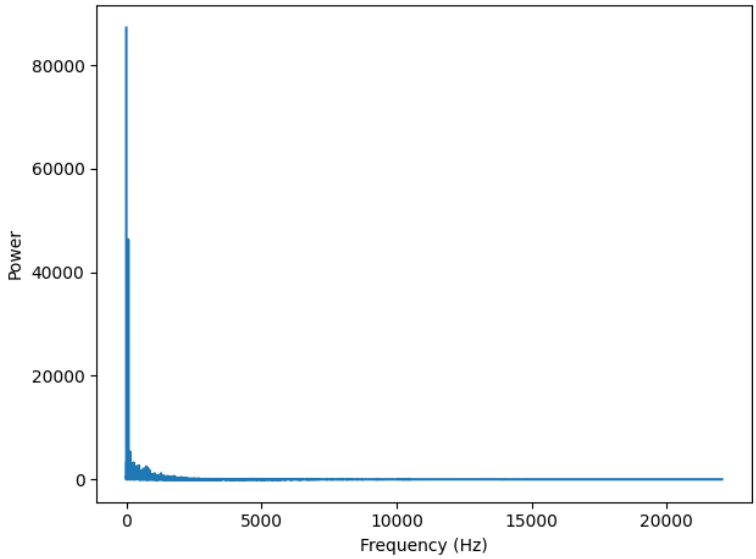

Рис. 4.1. Спектр шума Северного океана

Как мы видим, у нас сильный пик в районе низких частот, а остальные частоты не столь заметны. Этот шум сильно похож на розовый или красный. Проверим это, взглянув на спектр мощности в логарифмическом масштабе:

```
spectrum.plot_power()

loglog = dict(xscale='log', yscale='log')
decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         **loglog)
```

В результате получаем следующую логарифмическую зависимость:

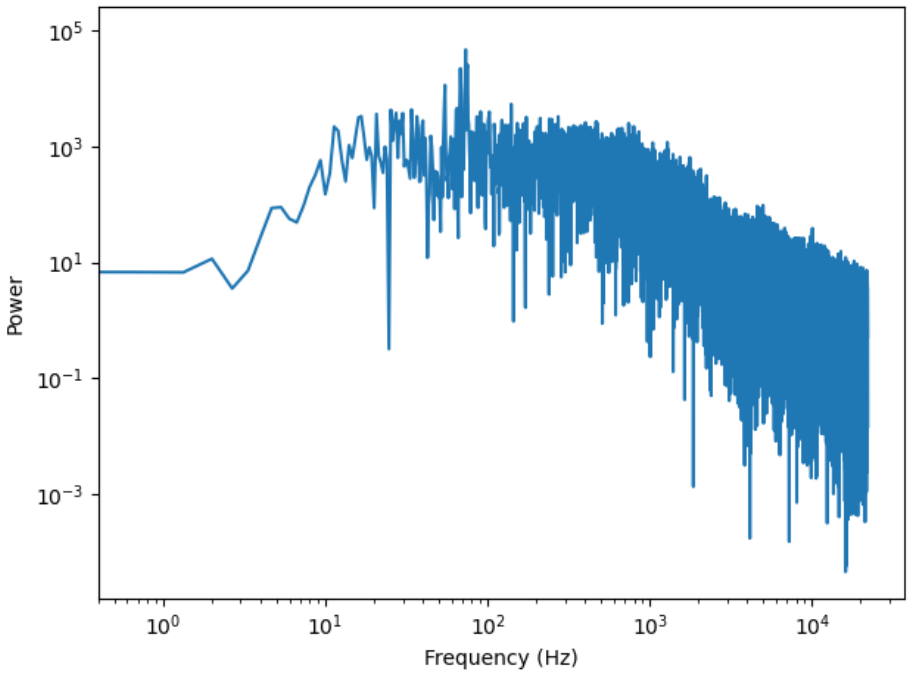

Рис. 4.2. Логарифмическая зависимость мощности от частоты.

Рассматриваемая зависимость необычна т.к. все зависимости, рассмотренные ранее, не возрастали, как эта.

Рассмотрим, как спектрограмма изменяется с течением времени, для этого выберем другой звуковой фрагмент:

```
segment2 = wave.segment(start=2.5, duration=1.5)
spectrum2 = segment2.make_spectrum()

spectrum.plot_power(alpha=0.5)
spectrum2.plot_power(alpha=0.5)
decorate(xlabel='Frequency (Hz)',
         ylabel='Power')
```

Получившийся спектр имеет следующий вид:

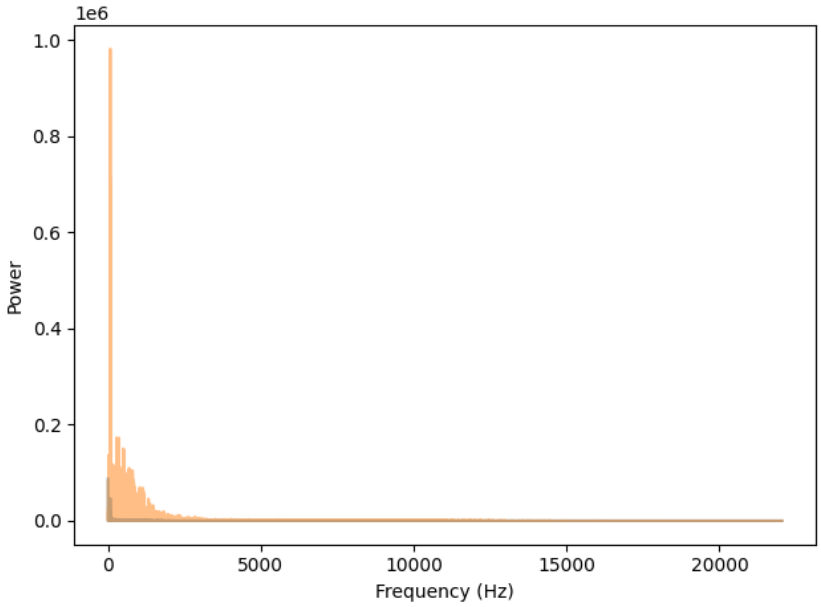

Рис. 4.3. Спектр двух звуковых отрезков.

Как мы видим, мощность второго сигнала имеет совершенно другой порядок, однако и в ней заметно, что преобладают именно низкочастотные сигналы.
Теперь выведем эти сигналы в логарифмической системе:

```
spectrum.plot_power(alpha=0.5)
spectrum2.plot_power(alpha=0.5)
decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         **loglog)
```

Получившаяся зависимость приведена ниже:

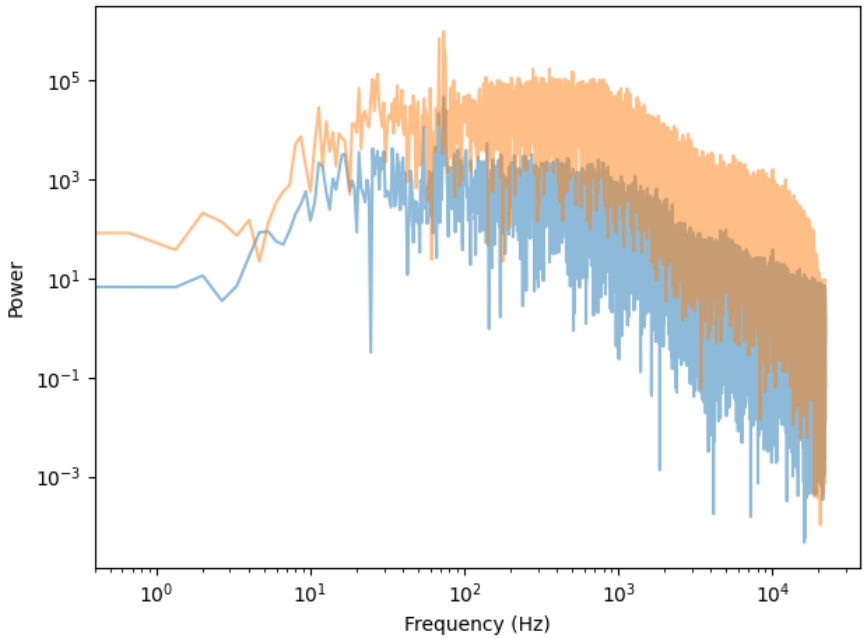

Рис. 4.4. Спектр двух отрезков в логарифмической шкале.

Как мы видим сигналы ведут себя схожим образом. Вероятно их отличие связано с тем, что первый отрывок брался с самого начала, где какое-то время тишина, а лишь потом слышна волна.
Для иллюстрации выведем спектрограмму первого и второго сегментов:

```
segment.make_spectrogram(512).plot(high=5000)
decorate(xlabel='Time(s)', ylabel='Frequency (Hz)')

segment2.make_spectrogram(512).plot(high=5000)
decorate(xlabel='Time(s)', ylabel='Frequency (Hz)')
```

Получившиеся спектрограммы приведены ниже:

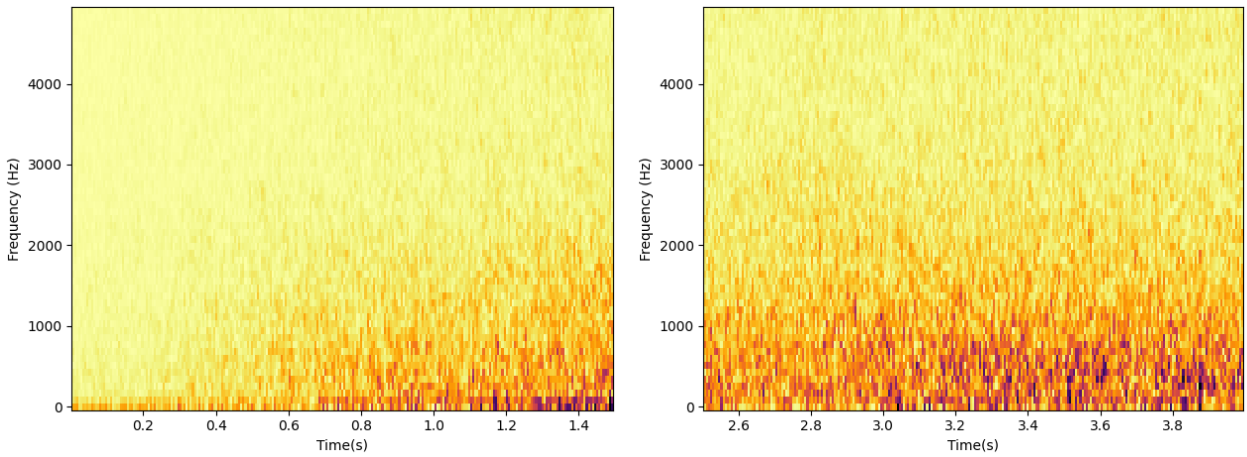

Рис. 4.5. Спектрограмма первого и второго сегментов.

4.2. Упражнение 4.2.

Реализуем метод Бартлетта, который позволит лучше проанализировать спектры мощностей для созданных ранее сегментов:

```python
def bartlett_method(wave, seg_length=512, win_flag=True):
    # make a spectrogram and extract the spectrums
    spectro = wave.make_spectrogram(seg_length, win_flag)
    spectrums = spectro.spec_map.values()

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

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

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

Воспользуемся созданной функцией:

```python
psd = bartlett_method(segment)
psd2 = bartlett_method(segment2)

psd.plot_power()
psd2.plot_power()

decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         **loglog)
```

После выполнения кода получим следующий результат:

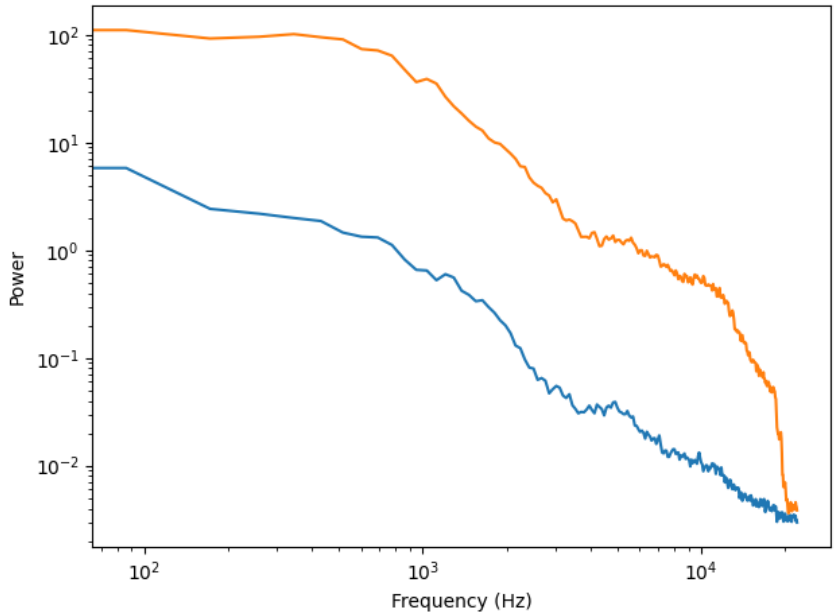

Рис. 4.6. Зависимость мощности от частоты после метода Бартлетта.

Как можно заметить эти два графика не похожи друг на друга, как ожидалось. Как и было сказано ранее, вероятнее всего это связано с тем, что в первом сегменте большую часть занимает лишь накатывающая волна, в отличии от второго.

4.3. Упражнение 4.3.

Для упражнения скачаем csv таблицу с данными о ежедневной цене криптовалюты bitcoin (на сайте coindesk кнопки для получения этих данных найдено не было, поэтому csv файл взят из методички) и откроем их в python:

```
import pandas as pd

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

В результате получаем таблицу с необходимыми данными, где есть цена в начале торгов, в конце и наивысшая за этот период.

Построим график закрывающей цены за период, приведенный в таблице:

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

from thinkdsp import Wave, decorate

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

Мы получаем значение цены биткоина, при закрытии торгов, а также число измерений. Все это передаем Wave, тем самым создав волну. Получившийся рисунок выглядит следующим образом:

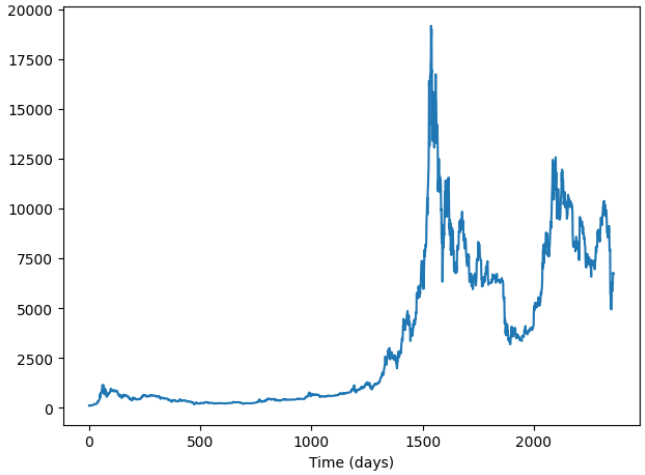

Рис. 4.7. Колебания цены биткоина.

Создадим спектр данного графика и выведем его график мощности на логарифмической шкале:

```
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Frequency (1/days)',
         ylabel='Power',
         xscale='log',
         yscale='log')
```

Получившийся график имеет следующий вид:

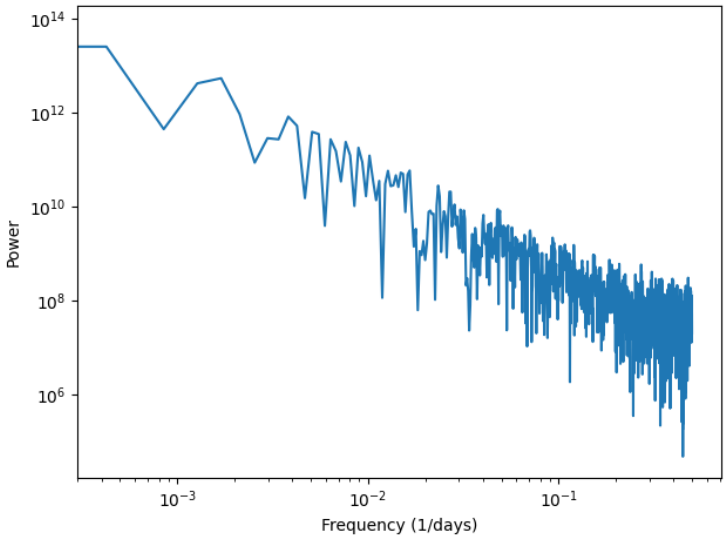

Рис. 4.8. Спектр биткоина.

Данный спектр похож как на красный, так и на розовый. Для ясности выведем значение наклона:

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

Полученное значение равно:

```
-1.7332540936758942
```

Это значение близко к значению красного шума (-2), но все же меньше его. Можно сказать, что этот шум все же является розовым.

4.4. Упражнение 4.4.

Выполним моделирование счетчика Гейгера, используя некоррелированный пуассоновый шум. Для этого создадим новый класс, наследующийся от Noise:

```
from thinkdsp import Noise
import numpy as np

class UncorrelatedPoissonNoise(Noise):

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

Далее создадим экземпляр этого класса, задав amp = 0.001, а частоту 10 кГц (получится 10 «щелчков» в минуту, что будет сильно похоже на звук счетчика Гейгера):

```
amp = 0.001
framerate = 10000
duration = 30

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

Данный звук действительно схож с звуком счетчика.
Посмотрим на график этого звука:

```
wave.plot()
plt.show()
wave.segment(start=0, duration=1).plot()
plt.show()
```

Полученные графики выглядят следующим образом:

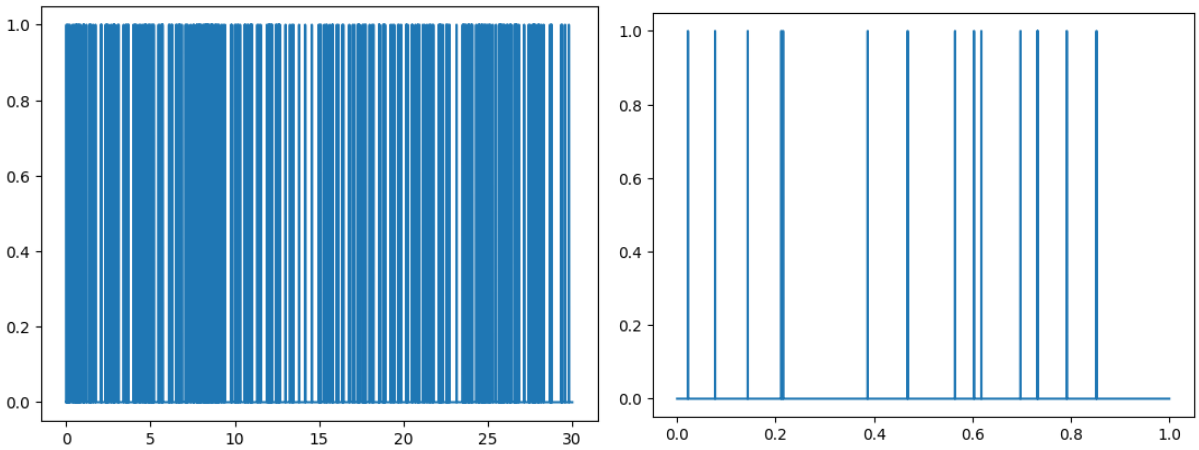

Рис. 4.9. Графики получившегося звука (30 сек и 1 сек).

Посмотрим на спектр мощности получившегося звука в логарифмической шкале:

```
from thinkdsp import decorate

spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         xscale='log',
         yscale='log')
```

Полученный график выглядит следующим образом:

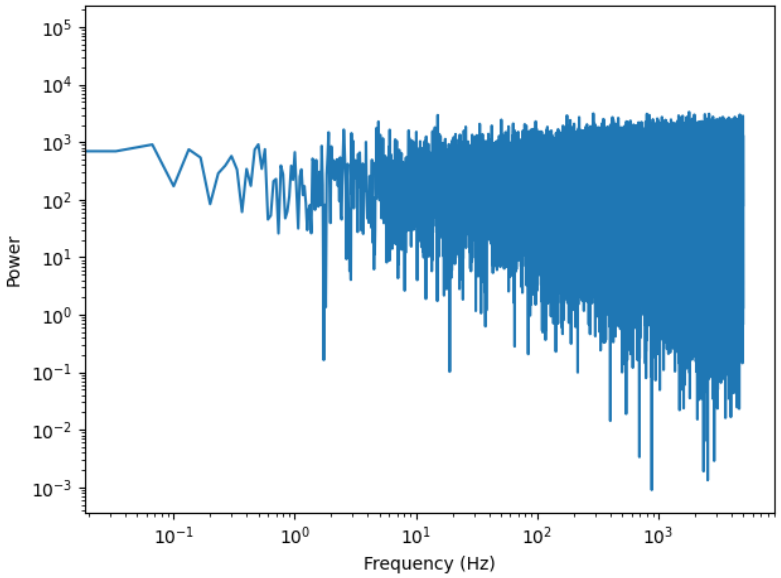

Рис. 4.10. Спектр мощности полученного звука.

Как видно по графику, он сильно похож на график белового шума, убедимся в этом, выведя значение наклона:

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

Оно равно:

```
0.00289475039409401
```

Как можно заметить, полученное число близко к нулю, что свидетельствует о том, что перед нами белый шум.

4.5. Упражнение 4.5.

Реализуем на языке python алгоритм Voss-McCartney, который предлагает более эффективный способ генерации розового шума. Его главная идея состоит в суммировании нескольких последовательностей случайных чисел, которые обновляются с разной частотой дискретизации. Первый источник должен обновляться на каждом временном шаге; второй источник - на каждом втором временном шаге, третий источник - на каждом четвертом шаге и так далее. Есть вариант со случайно распределенной частотой обновлений (именно этот вариант и будет реализован нами).

Первым этапом алгоритма будет генерация двумерного массива. Первое измерение – количество точек, для которых будет создан итоговый розовый шум, второе – количество источников для итогового суммирования. Заполним первый столбец и строку случайными числами:

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

Далее необходимо сгенерировать в каких столбцах произойдут обновления. Для этого очень удобно использовать функцию, генерирующую n-ое число испытаний Бернулли.
Результаты, которые превышают число столбцов – обнуляем:

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

Далее создадим строки, в которых будут соответствующие обновления и обновим значения в соответствующих парах строк-столбцов, записав туда случайное число от 0 до 1:

```
rows = np.random.randint(nrows, size=n)
array[rows, cols] = np.random.random(n)
```

Далее необходимо избавиться от оставшихся nan, записав вместо них предыдущее не nan значение в соответствующем столбце. Для этого воспользуемся библиотекой pandas и функцией fill:

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

После чего выполним суммирование в соответствующей строке и получим итоговый розовый шум:

```
total = df.sum(axis=1)

return total.values
```

Итоговая функция для генерации имеет следующий вид:

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

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

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

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

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

    return total.values
```

Создадим волну, используя эту функцию и проверим, действительно ли получился розовый шум, как требовалось:

```python
ys = voss(11025)
wave = Wave(ys)
wave.unbias()
wave.normalize()
wave.plot()
```

Получился следующий график:

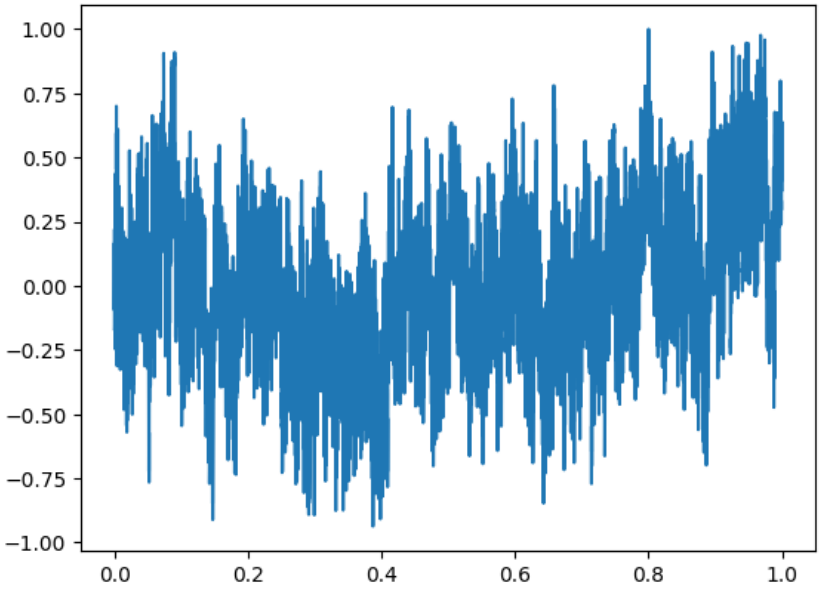

Рис. 4.11. График сгенерированного сигнала.

По этому графику видно, что зависимость какая-то присутствует, однако хорошо она не проглядывается. Лучше будет заметно по спектру мощности в логарифмической шкале:

```
spectrum = wave.make_spectrum()
spectrum.hs[0] = 0
spectrum.plot_power()
decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         xscale='log',
         yscale='log')
```

Получившийся спектр выглядит следующим образом:

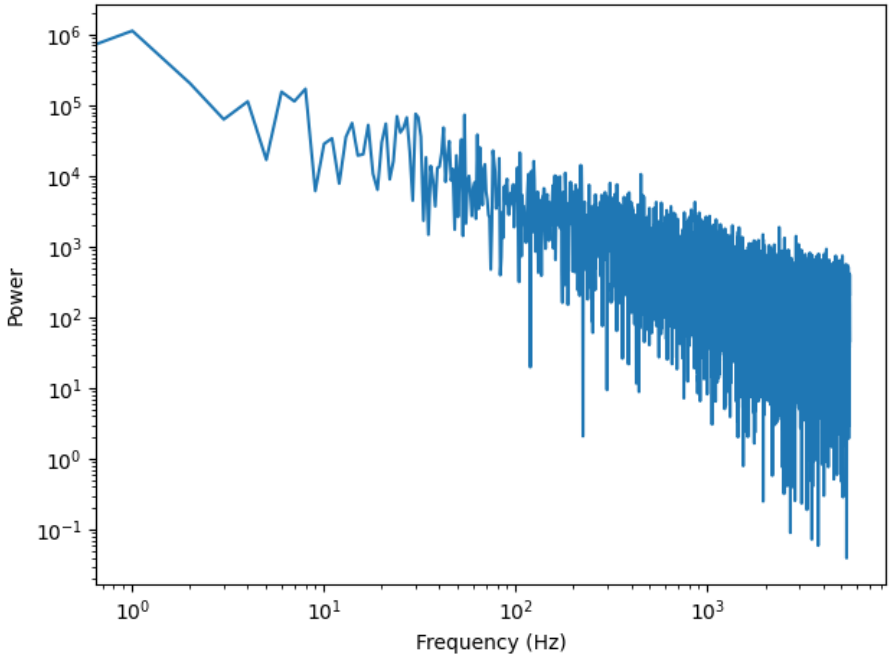

Рис. 4.12. Спектр сгенерированного шума.

Как видим, спектр сильно похож на аналогичный у розового шума, однако, чтоб быть уверенным наверняка, выведем наклон:

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

Получено следующее значение:

```
-1.0042795267007751
```

Это значение очень близко к 1, можно с большой уверенностью сказать, что получившийся шум действительно является розовым, как и требовалось.

5. Приложение:

Ссылка на репозиторий с исходными кодами: https://github.com/DafterT/telecom_labs