Санкт-Петербургский политехнический университет Петра Великого
Кафедра компьютерных систем и программных технологий
Высшая школа интеллектуальный систем и суперкомпьютерных технологий

Телекоммуникационные технологии

Отчёт по лабораторным работам

Работу выполнил:
В. И. Высоцкий
Группа:
3530901/90203
Преподаватель:
Н. В. Богач

Санкт-Петербург
2022

Содержание

1. Звуки и сигналы

1.1. Упражнение 2

Скачайте с сайта http://freesound.org , включающий музыку, речь или иные звуки, имеющие четко выраженную высоту. Выделите примерно полусекундный сегмент, в котором высота постоянна. Вычислите и распечатайте спектр выбранного сегмента. Как связаны тембр звука и гармоническая структура, видимая в спектре?
Используйте high_pass, low_pass, и band_stop для фильтрациитех или иных гармоник. Затем преобразуйте спектры обратно в сигнал и прослушайте его. Как звук соотносится с изменениями, сделанными в спектре?

Загрузим .wav файл, сохраненный в моем репозитории на гитхабе.

```
if not os.path.exists("trumpet.wav"):
  !wget https://github.com/CliffBooth/telecom_labs/raw/main/
  samples/trumpet.wav
wave = read_wave('trumpet.wav')
wave.make_audio()
```

График волны:

```
wave.plot()
```

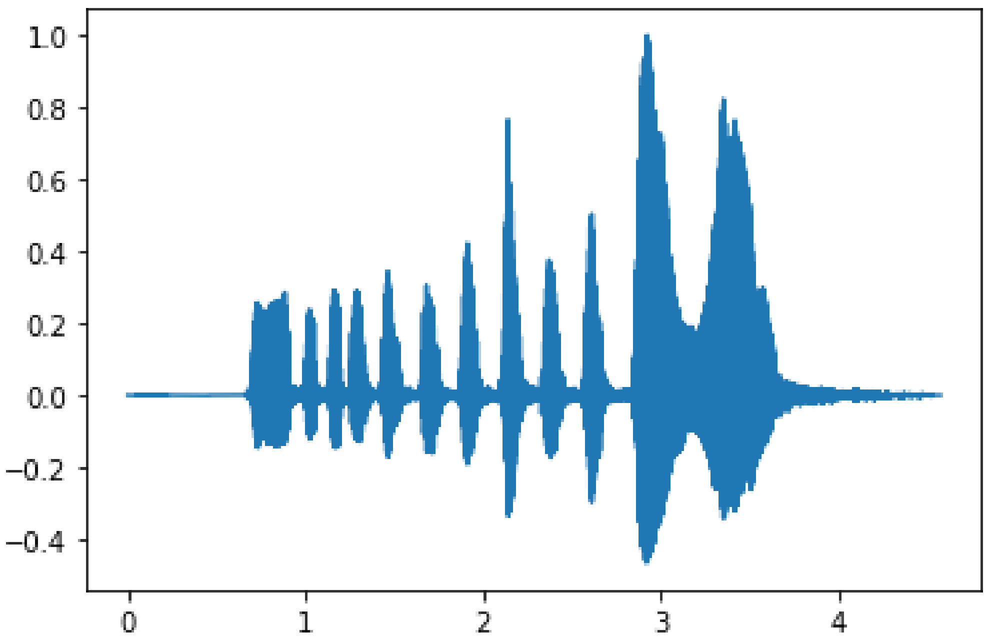

Рисунок 1.1

Выберем полсекундный отрезок с постоянной высотой.

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

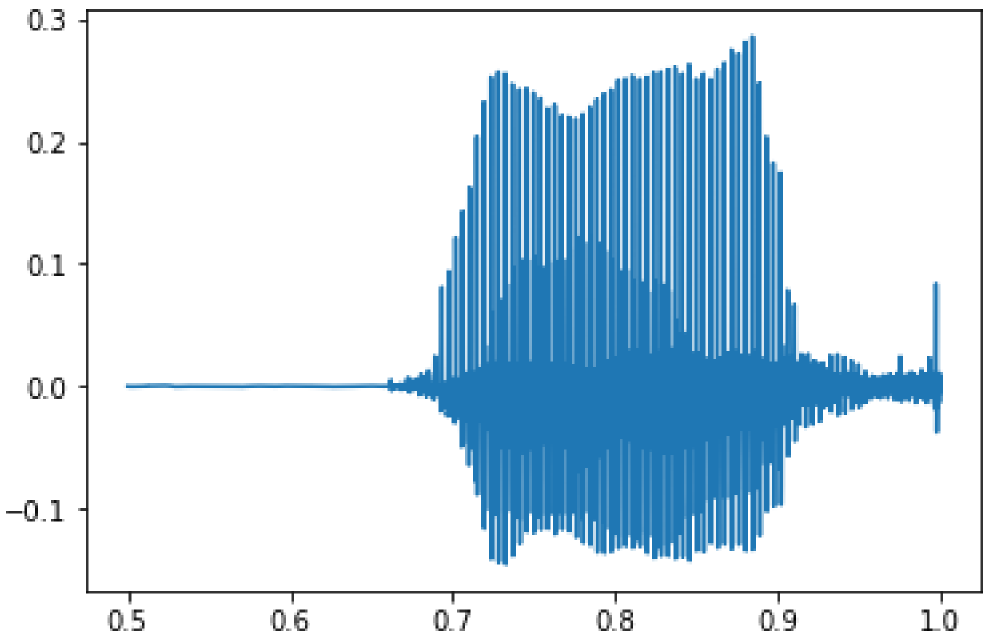

Рисунок 1.2

Построим спектр сегмента до 4000 Гц

```
spectrum = segment.make_spectrum()
spectrum.plot(high=4000)
decorate(xlabel='Frequency␣(Hz)')
```

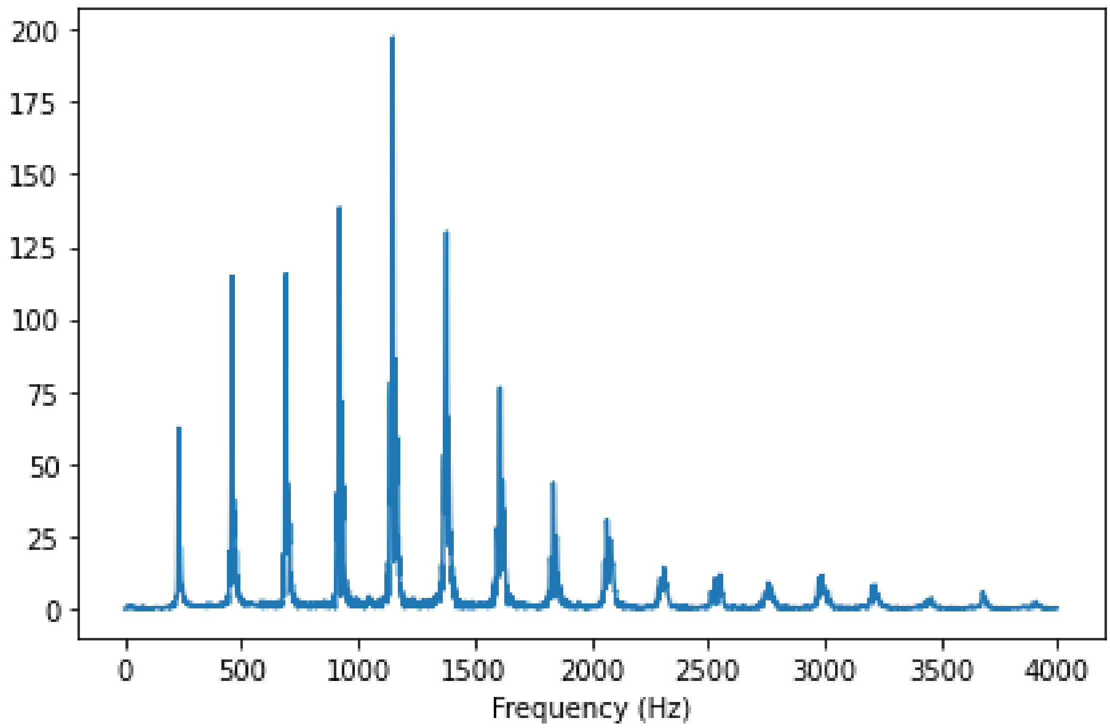

Рисунок 1.3

Посмотрим на пики спектра в порядку убывания:

```
spectrum.peaks()[:10]
```

```
[(197.3273749513319, 1150.0),
 (188.67875341629798, 1148.0),
 (138.45352894416652, 920.0),
 (130.11710283980153, 1380.0),
 (126.24380647951595, 1152.0),
 (122.59195851895944, 1378.0),
 (116.1451982656223, 1146.0),
 (115.70548632497977, 690.0),
 (114.89089691602553, 460.0),
 (114.13059684147524, 918.0)]
```

Доминирующая частота = 1150, основная частота = 230. Высота звука, который мы воспринимаем - это, как правило, основная частота.

Теперь попробуем отфильтровать высокие частоты:

```
spectrum.low_pass(1500)
spectrum.make_wave().make_audio()
spectrum.plot(4000)
```

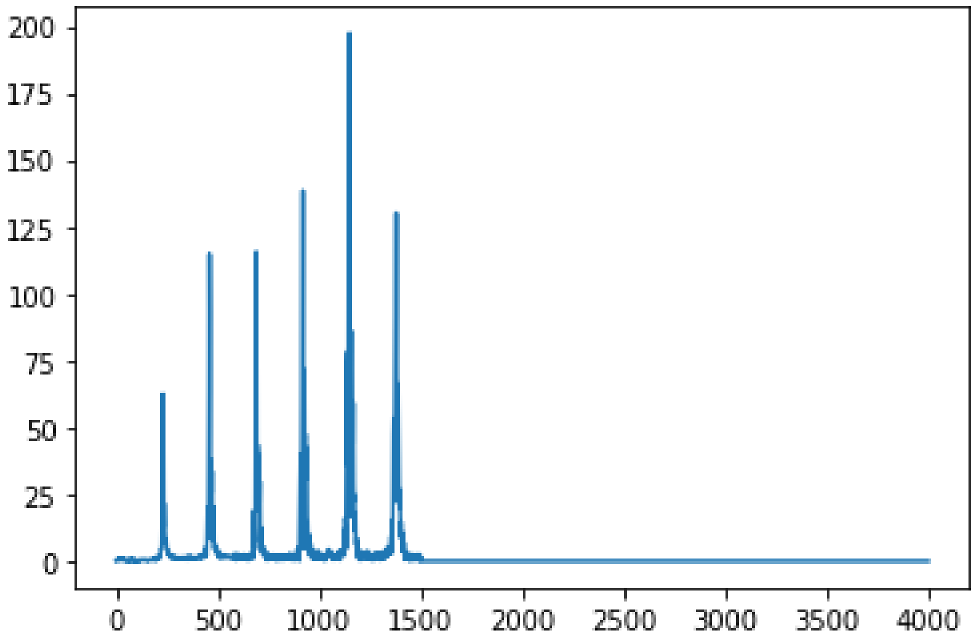

Рисунок 1.4

Звук стал более низким и менее наполненным

Теперь отфильтруем нижние частоты:

```
spectrum = segment.make_spectrum()
spectrum.high_pass(1000)
spectrum.make_wave().make_audio()
spectrum.plot(4000)
```

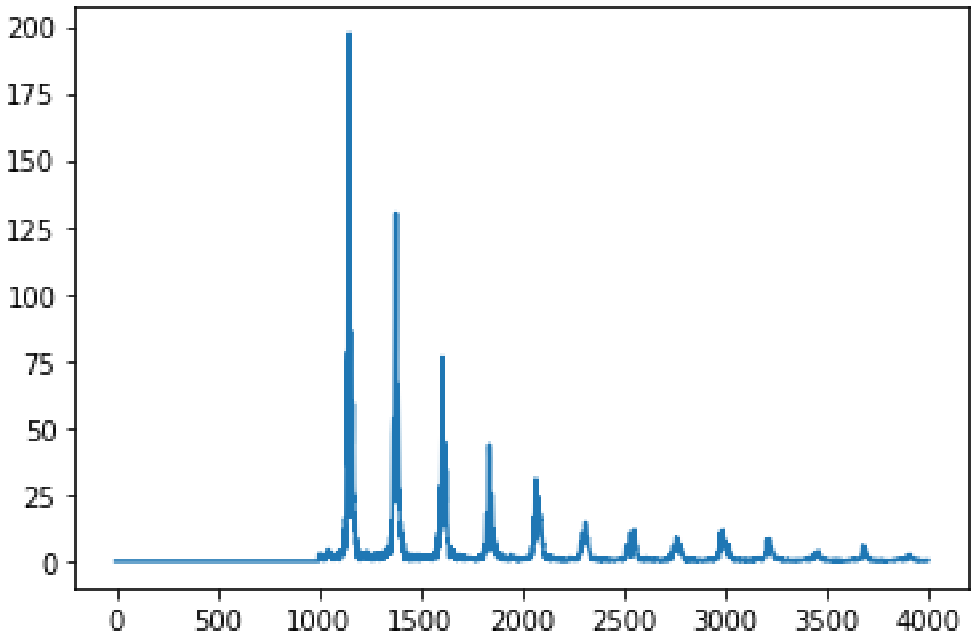

Рисунок 1.5

1.2. Упражнение 3

Создайте сложный сигнал из объектов SinSignal и CosSignal, суммируя их. Обработайте сигнал для получения wave и прослушайте его. Вычислите Spectrum и распечатайте. Что произойдёт при добавлении частотных компонент, не кратных основным?

Построим наш сигнал и пктем сложения синусоидальных сигналов

```
signal = SinSignal(freq=200, amp=1.0) + SinSignal(freq=400, amp=2.0) + SinS
signal.plot()
```

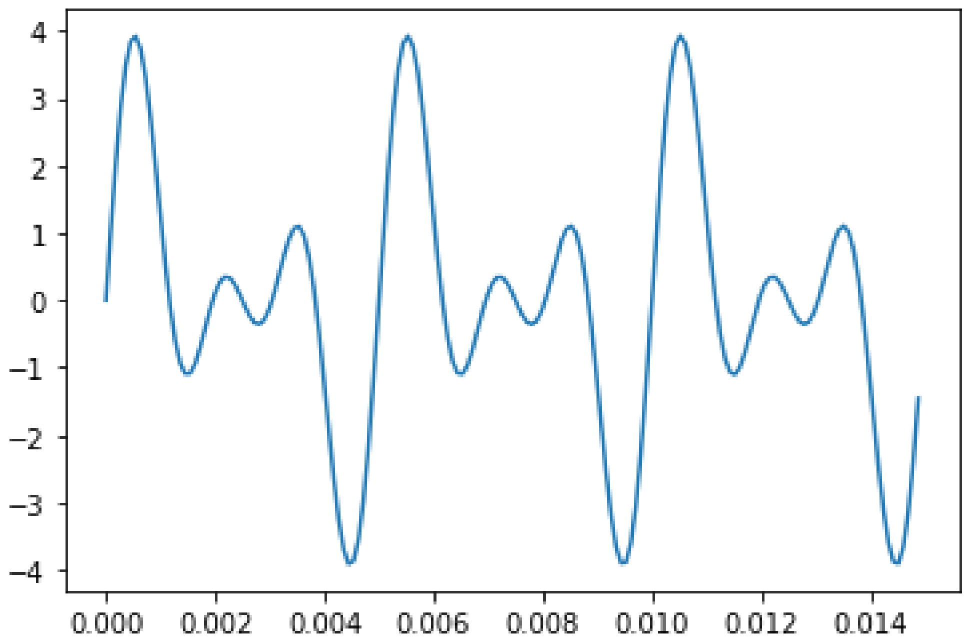

Рисунок 1.6

Сделаем волну из сигнала:

```
wave2 = signal.make_wave(duration=1)
wave2.apodize()
wave2.make_audio()
```

Все частоты всех компонентов сигнала являются кратными 200 (гармониками).

Помтроим спектр:

```
spectrum2 = wave2.make_spectrum()
spectrum2.plot(high = 1000)
```

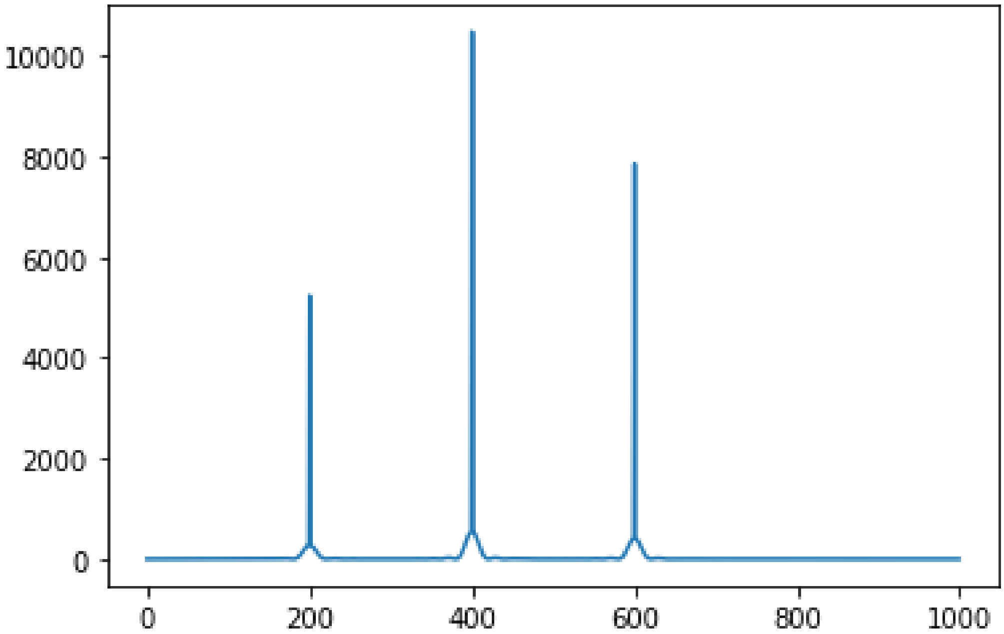

Рисунок 1.7

1.3. Упражнение 4

Напишите функцию strech, берущую wave и коэффицент изменения. Она должна ускорять или замедлять сигнал изменением ts и framerate.

Напиешм функцию и применим ее к волне из нашей аудиозаписи:
ts - отвечает за моменты выборки сигнала framerate - число выборок в единицу времени.
Если умножим ts на k, то интервалы между моментами увеличатся в k раз.
Если framerate поделим на k, то будет меньшее число подвыборок.

```
wave4 = read_wave('trumpet.wav')
def stretch(wave, stretch_factor):
  wave.framerate *= stretch_factor
  wave.ts *= stretch_factor
stretch(wave4, 2)
wave4.make_audio()
```

Построим график получившейся волны:

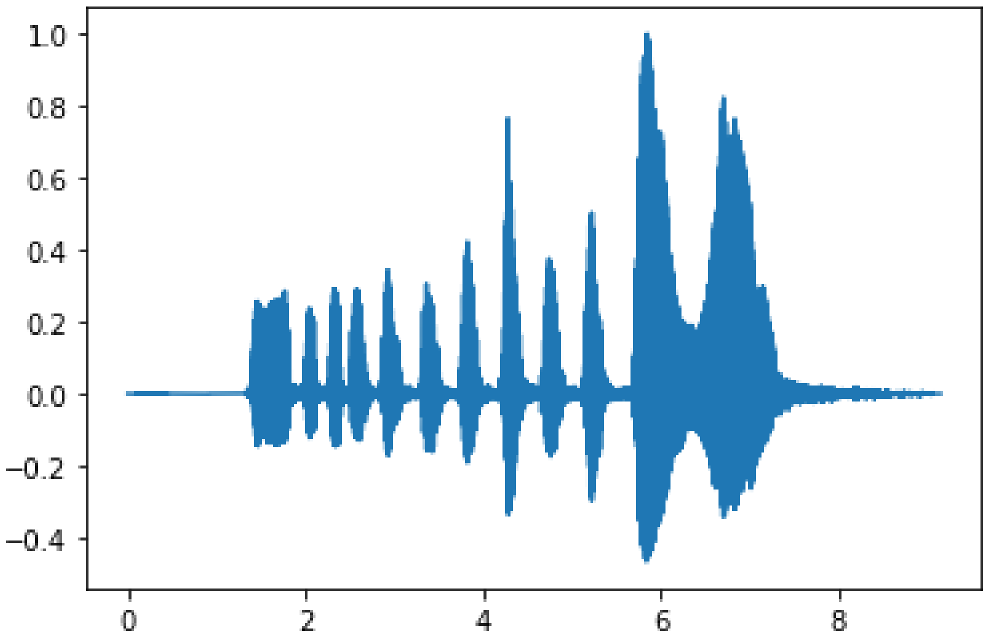

Рисунок 1.8

1.4. Вывод

В ходе данной работы было выполнено знакомство с основыми понятиями при работе со звуками и сигналами. При помощи библиотеки thinkDSP можно создавать сигналы и обрабатывать их.

2. Гармоники

2.1. Упражнение 2

Пилообразный сигнал линейно нарастает от -1 до 1, а затем резко падает до -1 и повторяется.
Напишите класс, называемый SawtoothSignal, расширяющий signal и предоставляющий evaluate для оценки пилообразного сигнала.
Вычислите спектр пилообразного сигнала. Как соотносится его гармоническая структура с тругольными с прямоугольными сигналами?

класс SawToothSignal унаследуем от класса Sinusoid, так как он имеет необходимые характеристики, такие как частота, период, сдвиг:

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

Создадим пилообразный сигнал

```
signal = SawToothSignal(200)
signal.plot()
wave = signal.make_wave()
wave.make_audio()
```

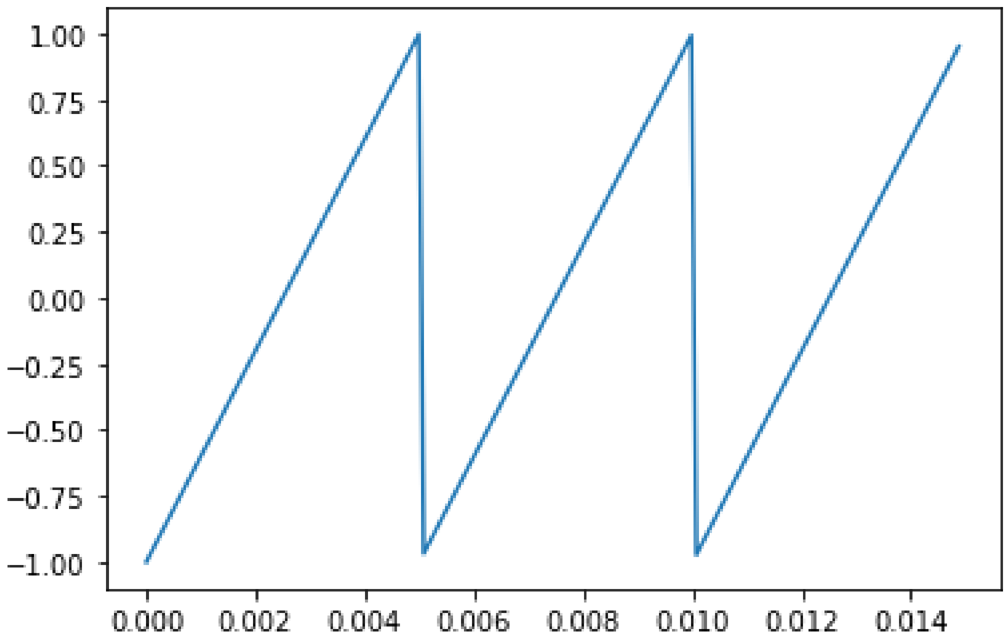

Рисунок 2.1

Посмотрим на спектр этого сигнала

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

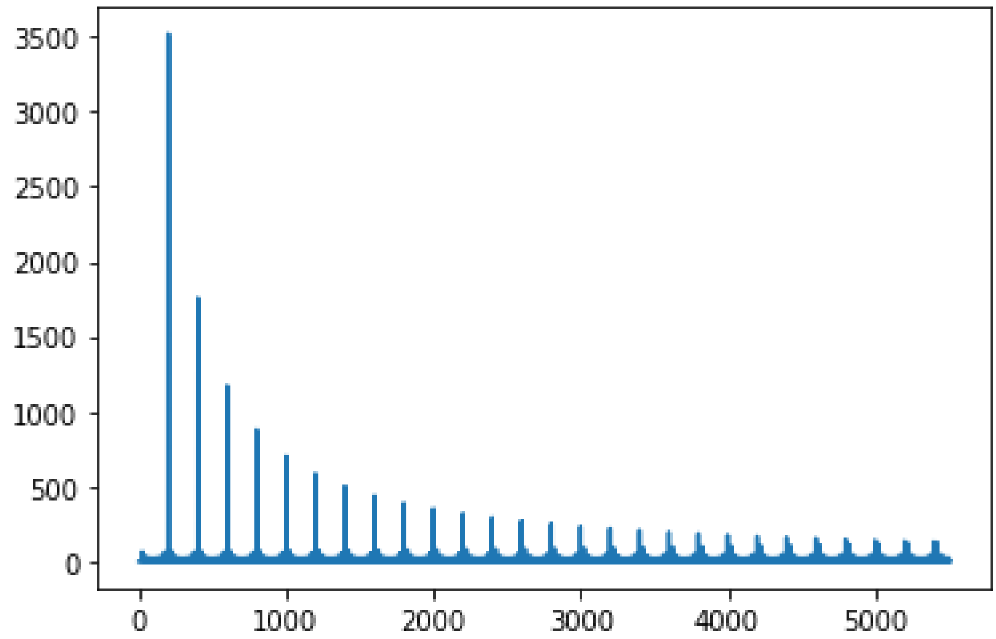

Рисунок 2.2

Сравним спектр пилообразного сигнала со спектрами квадратнгого и треугольного сигналов:

```
triangle = TriangleSignal(200)
triangle.make_wave().make_spectrum().plot()
```

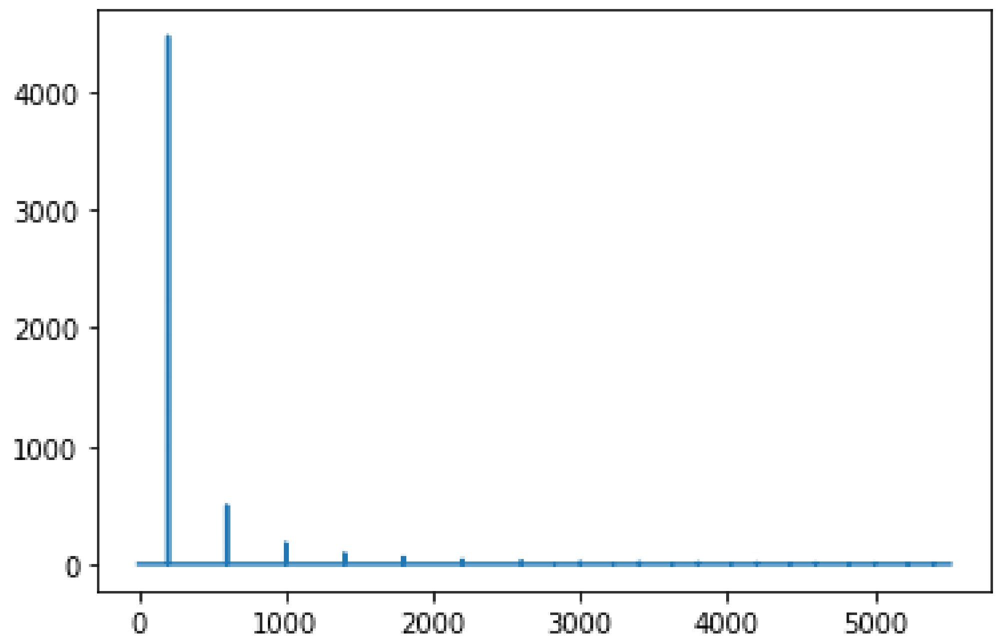

Рисунок 2.3

```
square = SquareSignal(200)
square.make_wave().make_spectrum().plot()
```

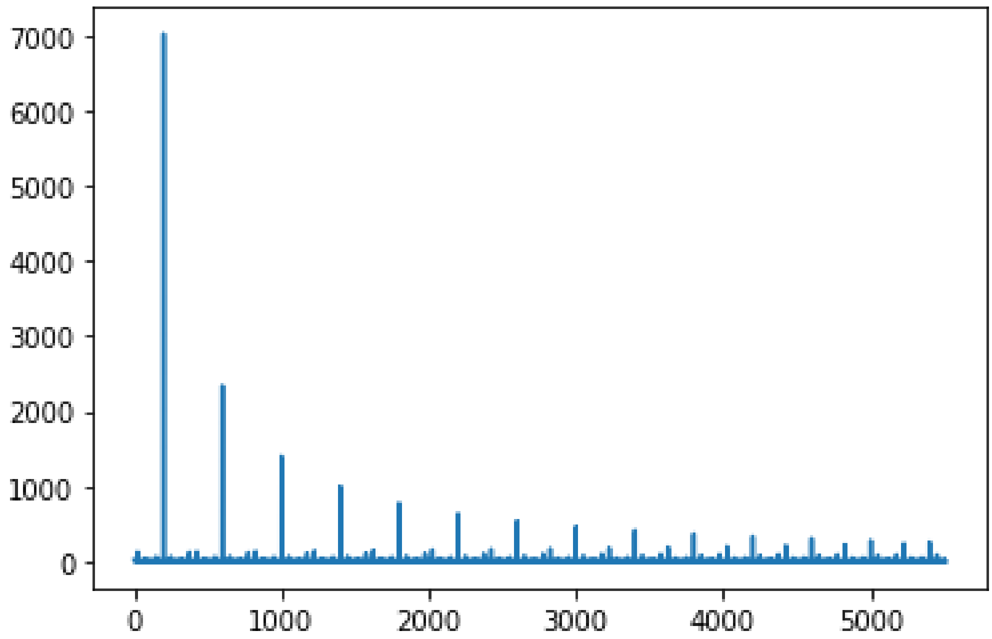

Рисунок 2.4

Можно заметить, что амплитуда пилообразного сигнала падает также как и спектр квадратного сигнала (пропорционально частоте), однако в отличие от квадратного он включает в себя и четные и нечетные гармоники.

Амплитуда треугольного же сигнала убывает в пропорции $1/f^2$, в то время как амплитуда пилообразного сигнала в пропорции $1/f$

2.2. Упражнение 3

Создайте прямугольный сигнал 1100 Гц и вычислите wave с выборками 10 000 кадров в секунду. Постройте спектр и убедитесь, что большинство гармоник "завёрнуты" из-за биений, слышно ли последствия этого при проигрывании?

```
signal = SquareSignal(freq=1100)
wave = signal.make_wave(duration=0.5, framerate=10000)
wave.make_spectrum().plot()
```

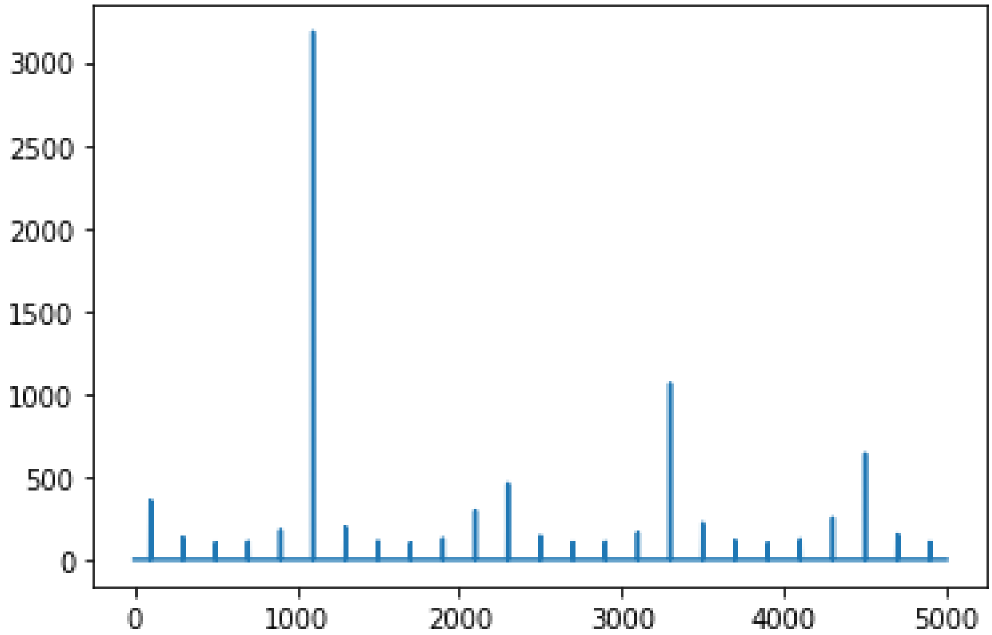

Рисунок 2.5

Базовая частота = 1100 Hz, первая гармоника на частоте 3300 Hz, вторая должна быть на 5500, но из-за наложения она на частоте 4500

2.3. Упражнение 4

Возьмите объект спектра spectrum, и выведите первые несколько значений spectrum.fs, вы увидите, что частоты начинаются с нуля. Итак, «spectrum.hs[0]» — это величина компонента с частотой 0. Но что это значит?
Попробуйте этот эксперимент:

1. Сделать треугольный сигнал с частотой 440 и создать Волну длительностью 0,01 секунды. Постройте форму волны.

2. Создайте объект Spectrum и напечатайте spectrum.hs[0]. Каковы амплитуда и фаза этой составляющей?

3. Установите spectrum.hs[0] = 100. Создайте волну из модифицированного спектра и выведите ее. Как эта операция влияет на форму сигнала?

```
triangle = TriangleSignal(440).make_wave(duration=0.01)
triangle.plot()
```

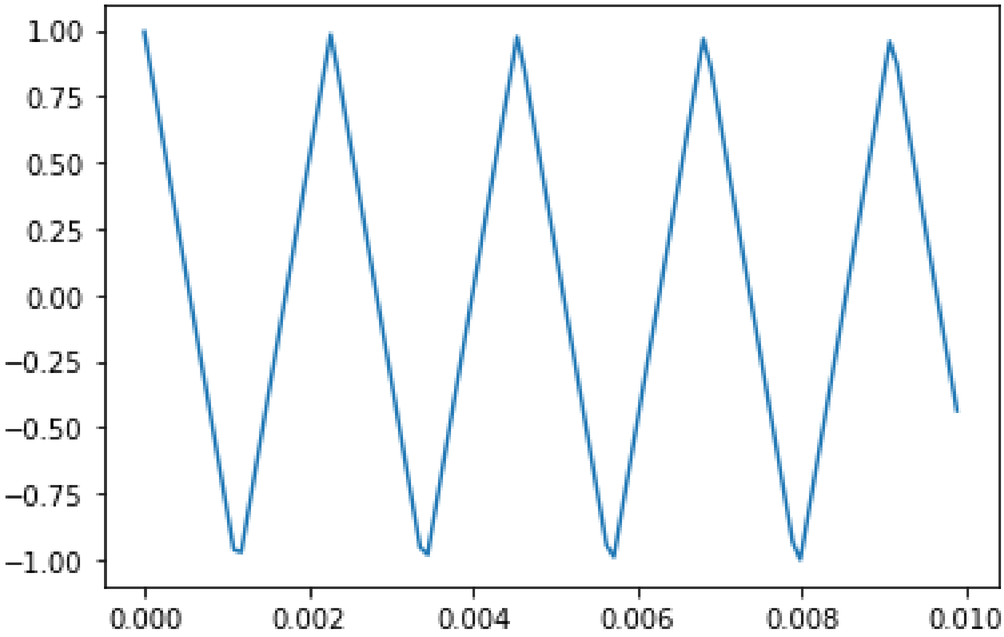

Рисунок 2.6

Проверим что лежит в 0 элементе чисел спекторграммы

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

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

Первый элемент спектра - это комплексное число близкое к 0.

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

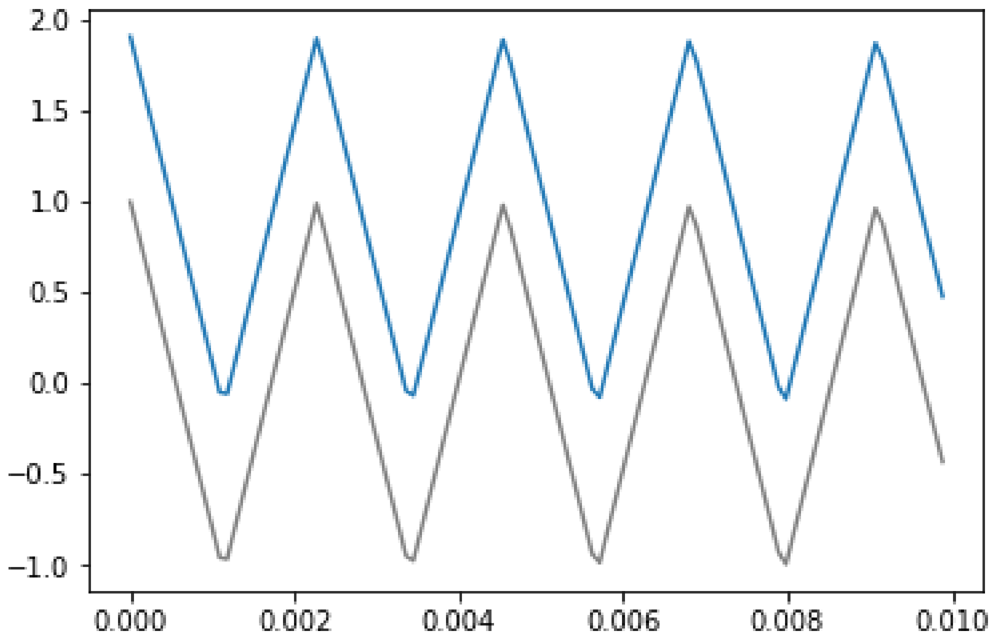

Рисунок 2.7

Можно заметить, что после того как мы зменили первый элемент спектра, весь спектр сместился вверх. Следовательно, от первого элемента зависит смещение

2.4. Упражнение 5

Напишите функцию, которая принимает Spectrum в качестве параметра и модифицирует его, деля каждый элемент hs на соответствующую частоту из fs. Протестируйте свою функцию, используя один из файлов WAV в репозитории или любой объект Wave.

1. Рассчитайте спектр и начертите его.
2. Измените спектр, используя свою функцию, и снова начертите его.
3. Сделать волну из модифицированного Spectrum и прослушать ее. Как эта операция влияет на сигнал?

Исходя из последнего пункта первый элемент очень близок к нулю. Поэтому на него делить не надо, а то получим очень большие значения.

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

square = SquareSignal().make_wave().make_spectrum()
triangle = TriangleSignal().make_wave().make_spectrum()
sawTooth = SawToothSignal().make_wave().make_spectrum()
```

Построим графики сигналов до и после применения к ним написанной функции:

```
square.plot(color='yellow')
divide(square)
square.scale(440)
square.plot()
```

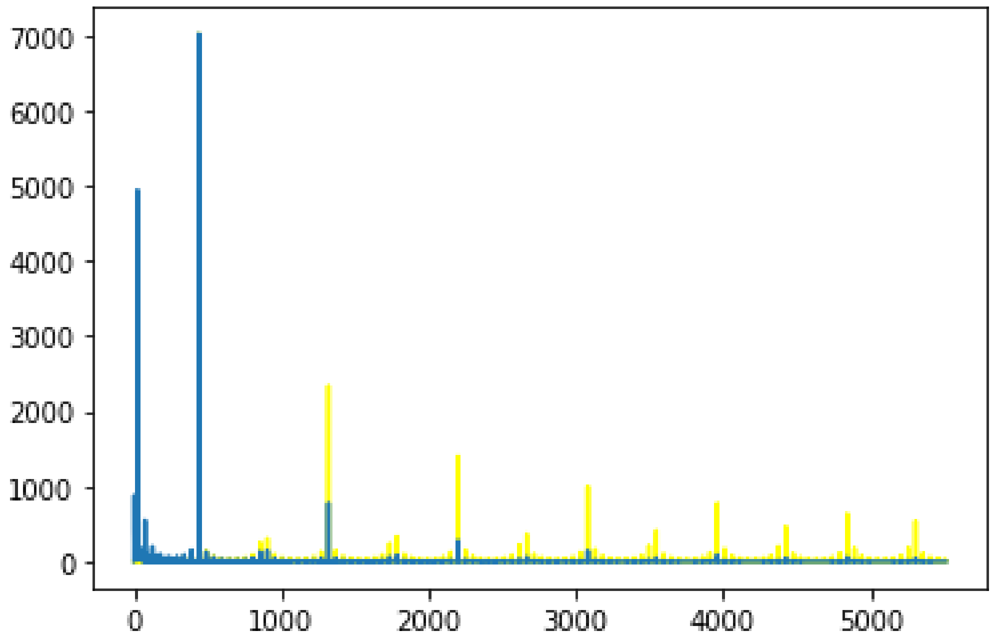

Рисунок 2.8

```
triangle.plot(color='yellow')
divide(triangle)
triangle.scale(440)
triangle.plot()
```

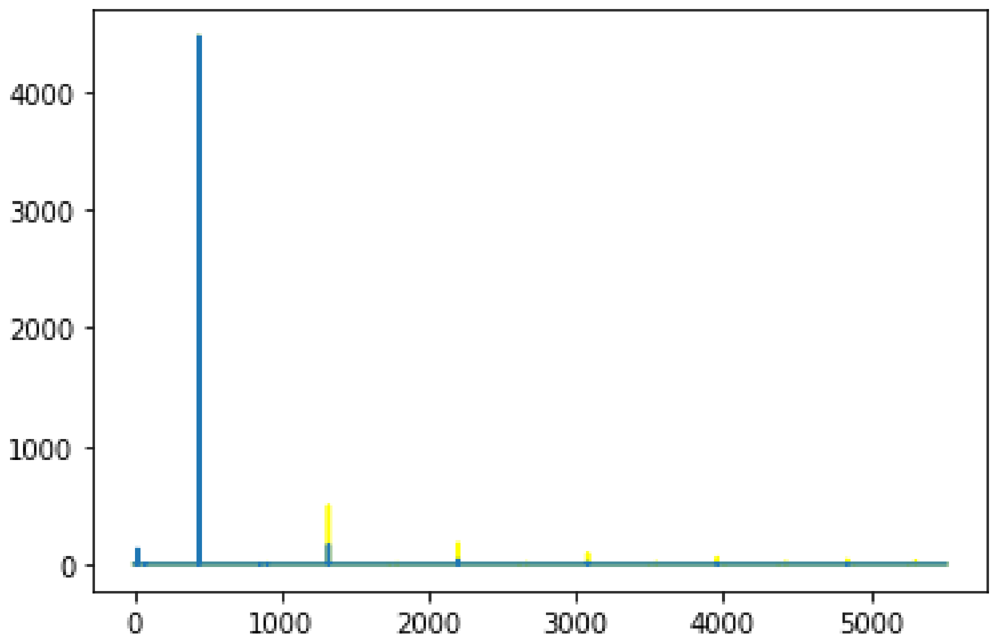

Рисунок 2.9

```
sawTooth.plot(color='yellow')
divide(sawTooth)
```

```
sawTooth.scale(440)
sawTooth.plot()
```

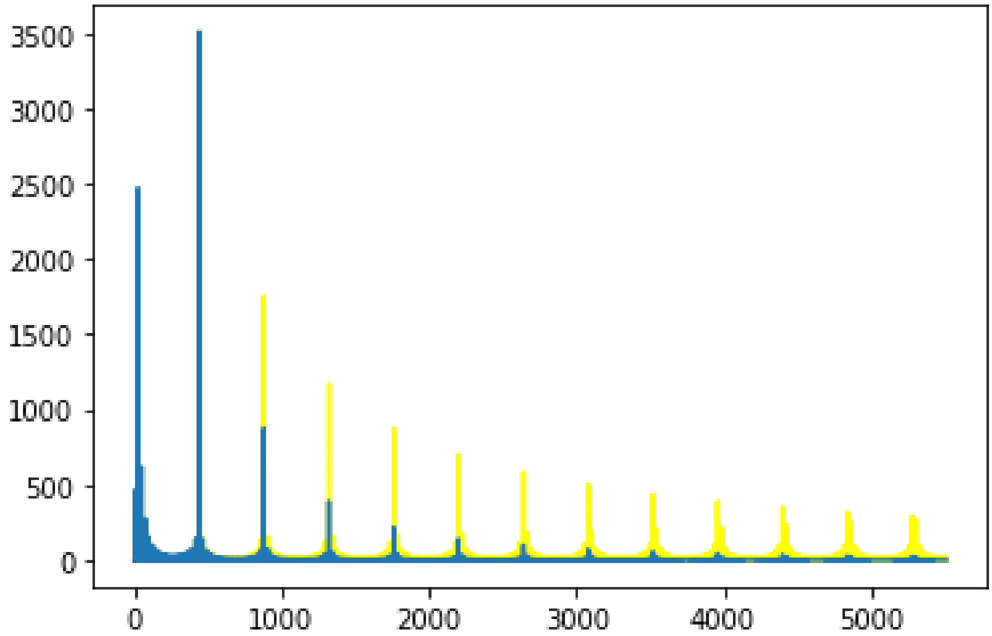

Рисунок 2.10

Видно, что функция уменьшает гормоники, так что можно сказать, что функция оказывает тот же эффект, что и low_pass()

2.5. Упражнение 6

Треугольные и прямоугольные волны имеют только нечетные гармоники; пилообразная волна имеет как четные, так и нечетные гармоники. Гармоники прямоугольной и пилообразной волн затухают пропорционально $1/f$; гармоники треугольной волны затухают как $1/f^2$. Можете ли вы найти форму волны, в которой четные и нечетные гармоники затухают как $1/f^2$?
Подсказка: есть два способа подойти к этому: вы можете построить нужный сигнал путем сложения синусоид, или вы может начаться с сигнала, похожего на то, что вы хотите, и изменить его.
Начнем с построения пилообразного сигнала, у которого есть четные и нечетные гармоники

```
signal = SawtoothSignal(freq=500)
wave = signal.make_wave(duration=0.5, framerate=20000)
spectrum = wave.make_spectrum()
spectrum.plot()
```

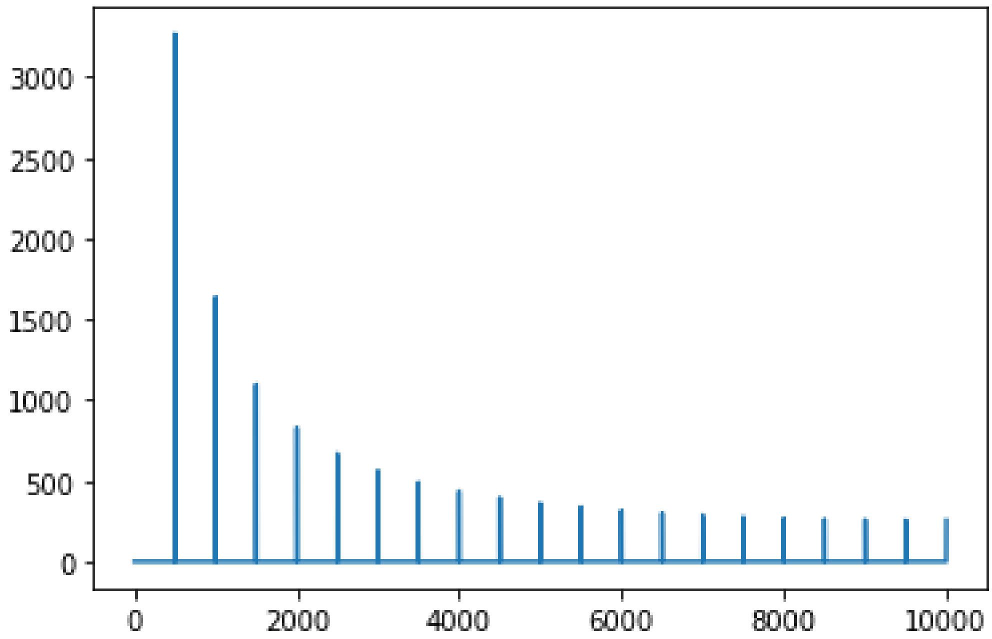

Рисунок 2.11

На графике спектра амплитуда падает пропорционально частоте $1/f$

Если к спектру применить фильтр из предыдущего задания, то амплитуда будет падать пропорционально $1/f^2$

```
divide(spectrum)
spectrum.plot()
spectrum.make_wave().make_audio()
```

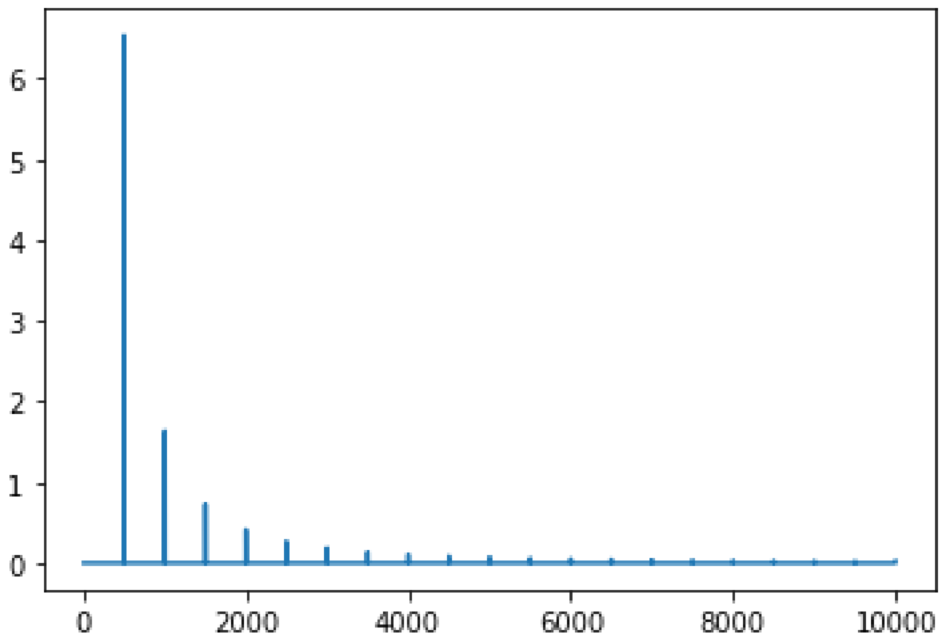

Рисунок 2.12

2.6. Вывод

В данной работе были исследованы некоторые виды сигналов. Были рассмотрены спектры и гармонические структуры сигналов. Также в одном из пунктов были замечены биения и мы проверили их действие на звук.

3. Непериодические сигналы

3.1. Упражнение 1

Запустите и прослушайте примеры в файле chap03.ipynb. В примере с утечкой попробуйте заменить окно Хэмминга одним из других окон, предоставляемых NumPy, и посмотрите, как они влияют на утечку.

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

spectrum.plot(high=880)
```

Спектр непериодической волны:

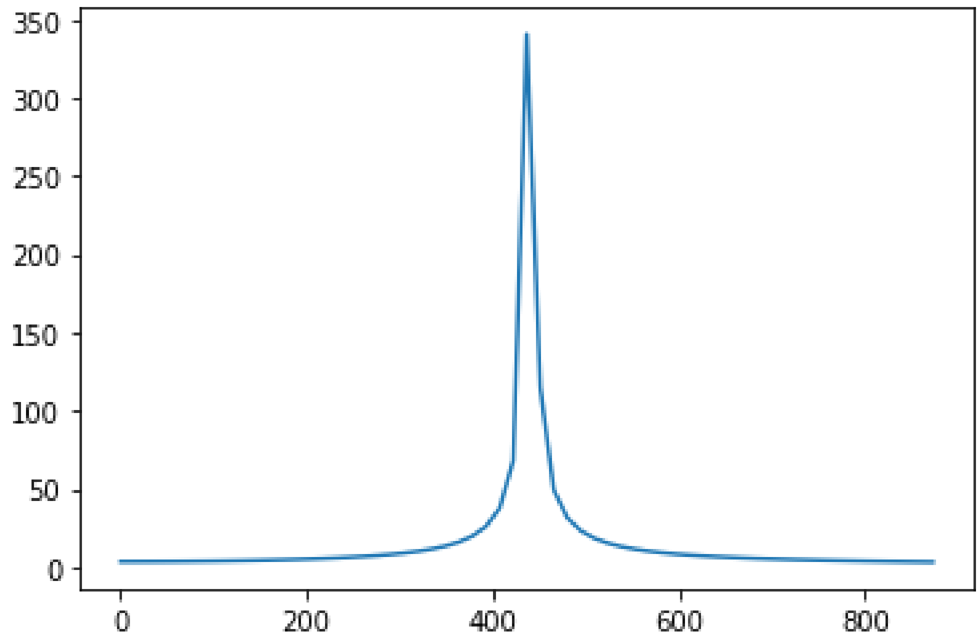

Рисунок 3.1

Спектр это волны с применением окна Хэмминга:

```
wave.hamming()
wave.make_spectrum().plot(high=800)
```

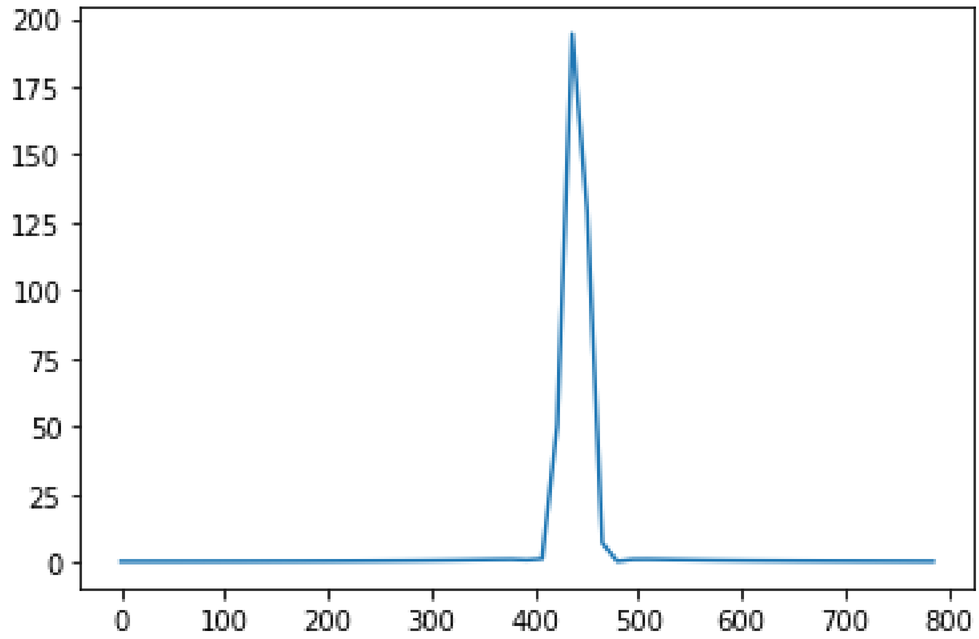

Рисунок 3.2

Заменим окно Хэмминга на окно Бартлетта:

```
wave = signal.make_wave(duration)
wave.ys *= np.bartlett(len(wave))
wave.make_spectrum().plot(high=800)
```

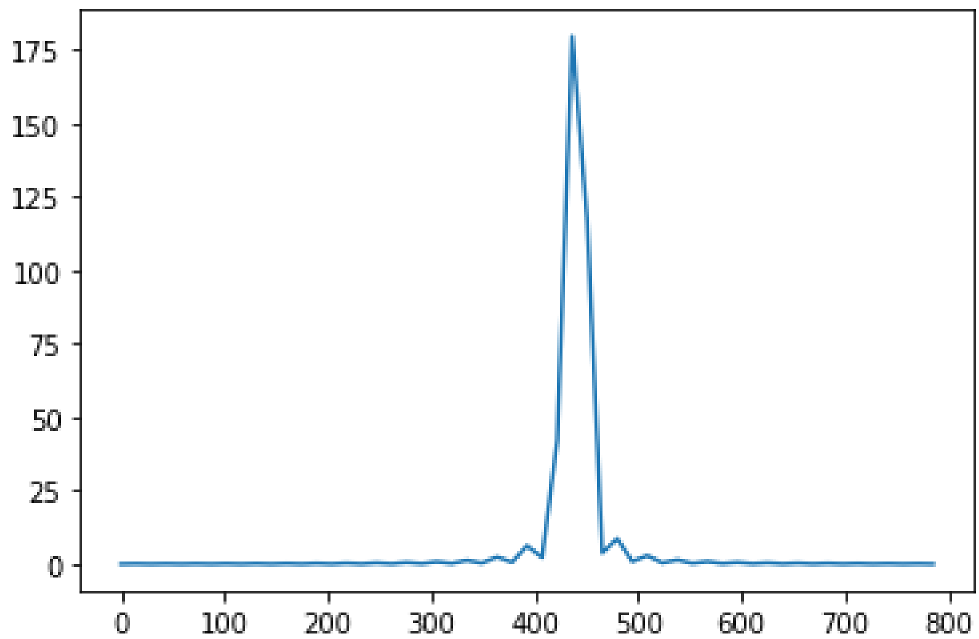

Рисунок 3.3

Теперь на частотах, близких к пику, график выглядит как ломанная линия

Окно Блэкмена:

```
wave = signal.make_wave(duration)
wave.ys *= np.blackman(len(wave))
wave.make_spectrum().plot(high=800)
```

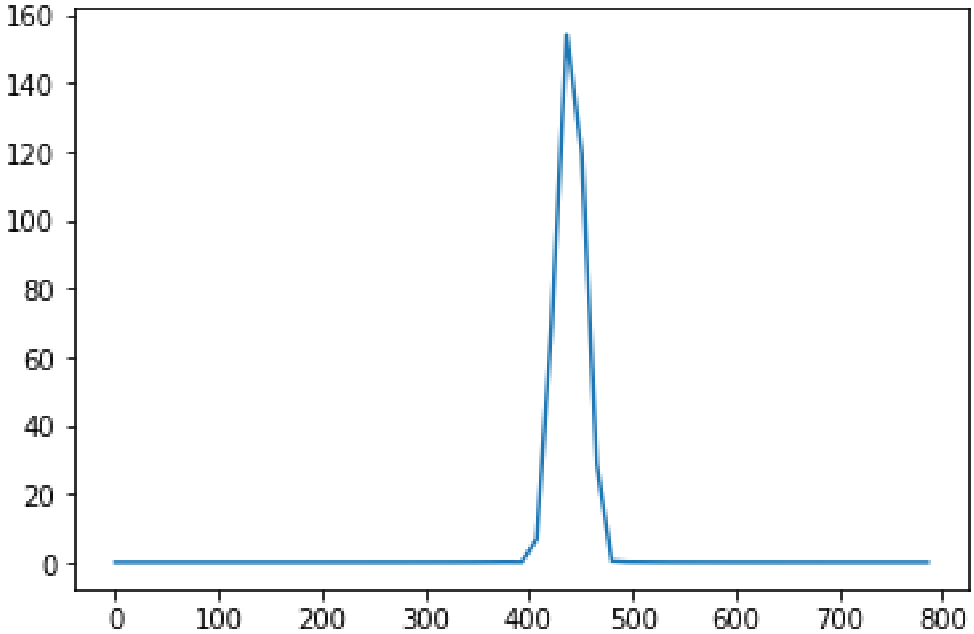

Рисунок 3.4

Этот график очень похож на получекнный с помощью окна Хэмминга

Окно Хэннинга:

```
wave = signal.make_wave(duration)
wave.ys *= np.hanning(len(wave))
wave.make_spectrum().plot(high=800)
```

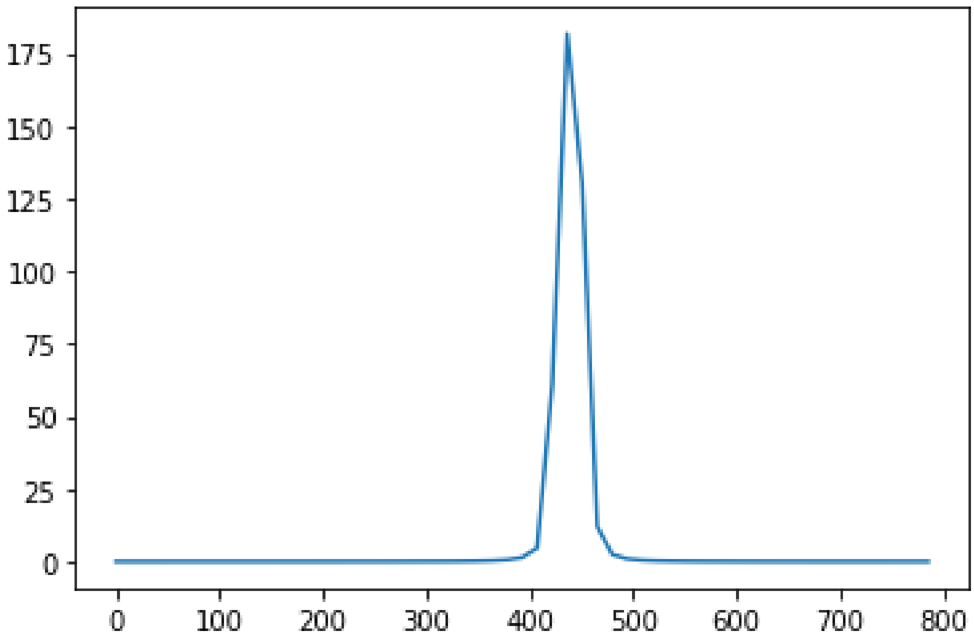

Рисунок 3.5

Как видно из графиков, все функции окон хорошо справляются со своей задачей - борьбой с растеканием спектра.

3.2. Упражнение 2

Напишите класс SawtoothChirp, расширяющий Chirp и переопределяющий evaluate для генерации пилообразного сигнала с линейно увеличивающейся частотой.
Нарисуйте эскиз спектограммы этого сигнала, затем распечатайте её. Эффект биение должен быть очевиден, а если сигнал внимательно прослушать, то биения можно и услышать.

```
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
```

Спектограмма сигнала:

```
signal = SawtoothChirp(start=220, end=880)
wave = signal.make_wave(duration=1, framerate=4000)
sp = wave.make_spectrogram(256)
sp.plot()
```

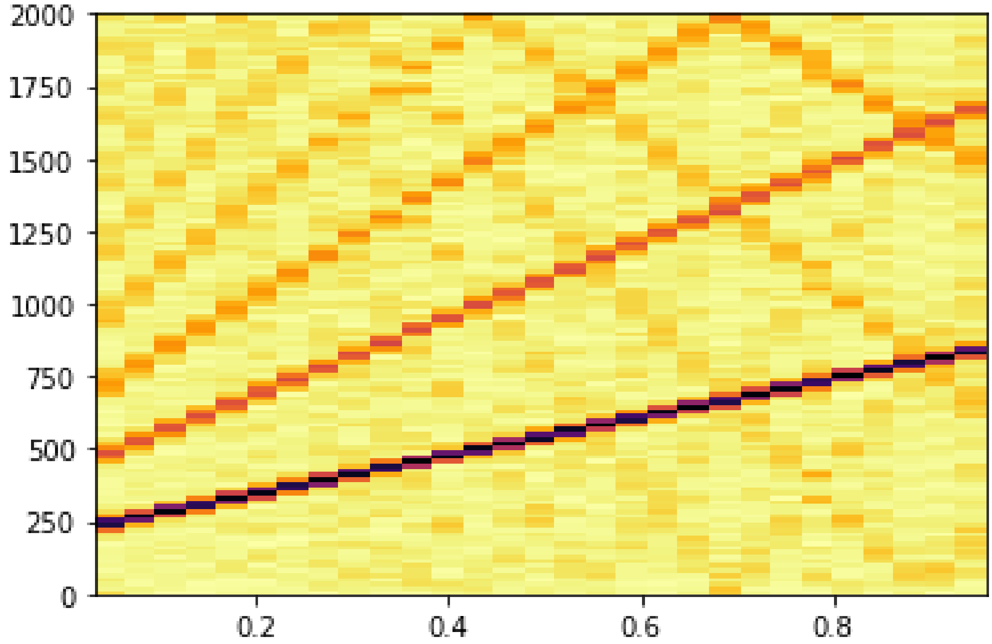

Рисунок 3.6

Жирная черная линия - это наша основная частота, красные линии - это частоты с биением, которые "отскакивают" от частоты завароты (folding frequency)

3.3. Упражнение 3

Создайте пилообразный чирп, меняющийся от 2500 до 3000 Гц, и на его основе сгенерируйте сигнал длительностью 1 с и частотой кадоров 20 кГц. Нарисуйте, каким примерно будет Spectrum. Затем распечатайте Spectrum и посмотрите, правы ли вы.

```
signal = SawtoothChirp(start=2500, end=3000)
wave = signal.make_wave(duration=1, framerate=20_000)
sp = wave.make_spectrum()
sp.plot()
```

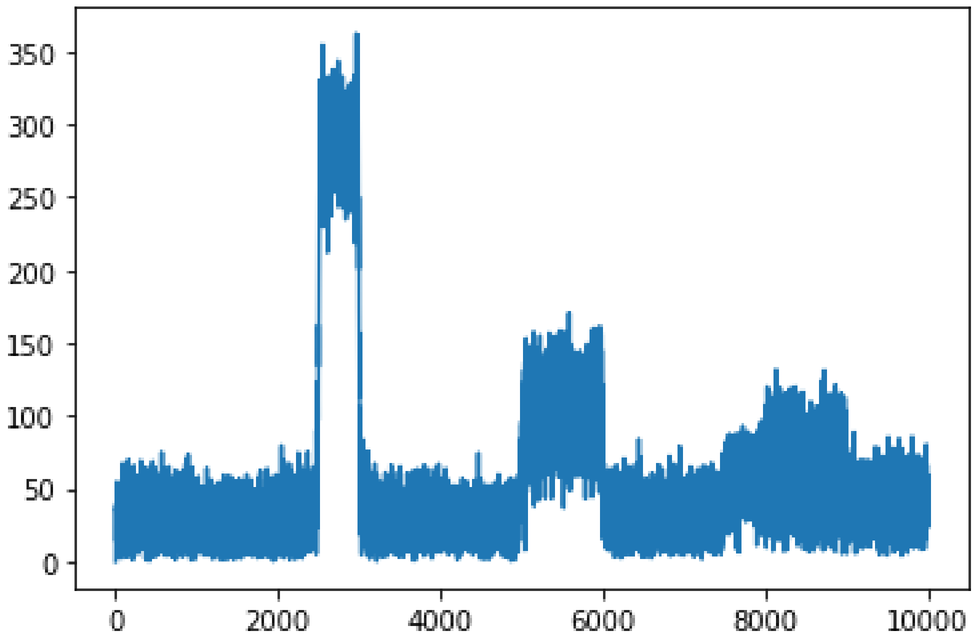

Рисунок 3.7

Между частотами 2500 и 3000 можно увидеть "глаз Саурона". Последующие гармоники (от 5000 до 6000 и от 7500 до 9000) имеют меньший дипазон амплитуды

3.4. Упражнение 4

В музыкальной терминологии «глиссандо» — это нота, которая скользит от одной высоты тона к другой, поэтому она похожа на чириканье. Найдите или сделайте запись глиссандо и постройте его спектрограмму.

Возьмём звук из репозитория учебника:

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

wave = read_wave('72475__rockwehrmann__glissup02.wav')
wave.make_audio()
wave.make_spectrogram(1024).plot(high=5000)
```

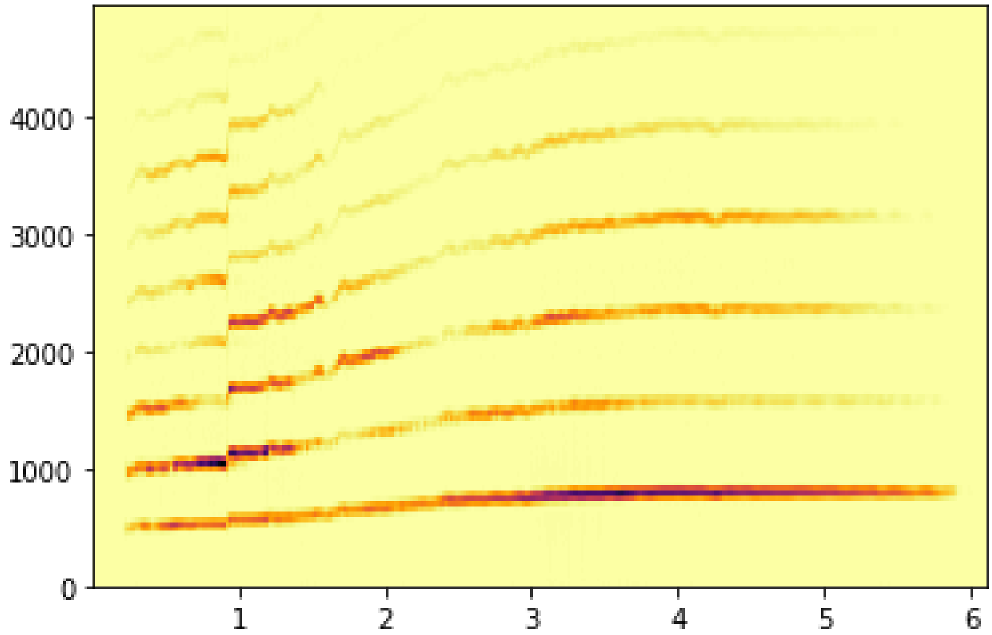

Рисунок 3.8

Спектограмма похожа на пилообразный чирп.

3.5. Упражнение 5

Тромбонист может играть глиссандо, выдвигая слайд тромбона и непрерывно дуя. По мере выдвижения ползуна общая длина трубки увеличивается, а результирующий шаг обратно пропорционален длине. Предполагая, что игрок перемещает слайд с постоянной скоростью, как меняется ли частота со временем?
Напишите класс TromboneGliss, расширяющий класс Chirp и предоставляет evaluate. Создайте волну, имитирующую тромбон глиссандо от F3 вниз до C3 и обратно до F3. C3 — 262 Гц; F3 есть 349 Гц.

```
class TromboneGliss(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 2 сигнала и объединим их вместе:

```
c3 = 262
f3 = 349
signal =  TromboneGliss(f3, c3)
wave = signal.make_wave(duration=1)
sp = wave.make_spectrogram(1024)
```

```
signal = TromboneGliss(c3, f3)
wave2 = signal.make_wave(duration=1)
sp2 = wave.make_spectrogram(1024)

result = wave | wave2
sp3 = result.make_spectrogram(1024)
sp3.plot(high = 1000)
```

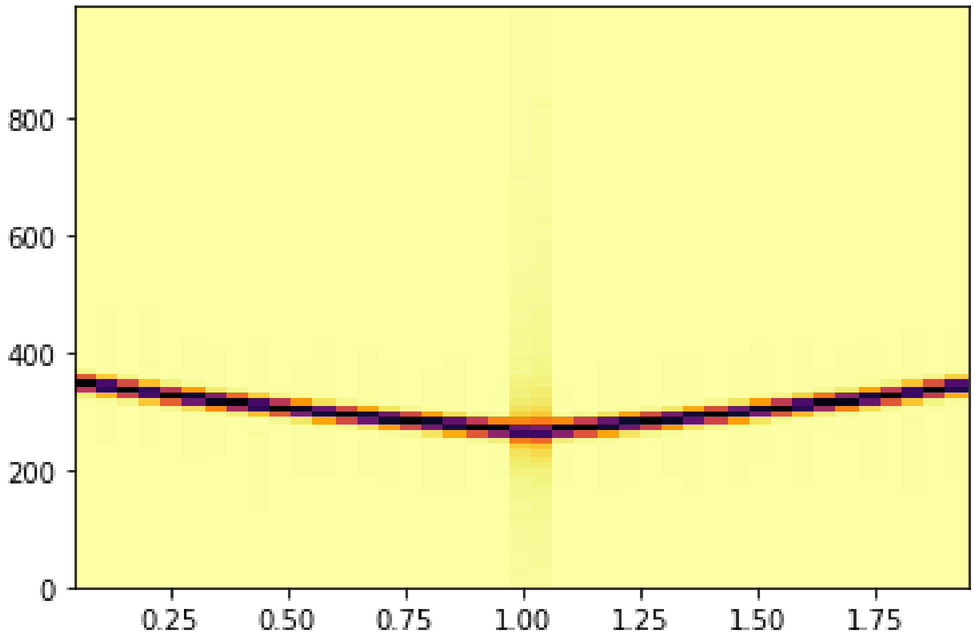

Рисунок 3.9

Отчетливо слышен переход медждy нотами

3.6. Упражнение 6

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

Возьмем пример звукоа из учебного репозитория:

```
if not os.path.exists('87778__marcgascon7__vocals.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/87778__m
wave = read_wave('87778__marcgascon7__vocals.wav')
wave.make_audio()
wave.make_spectrogram(1024).plot(high=1000)
```

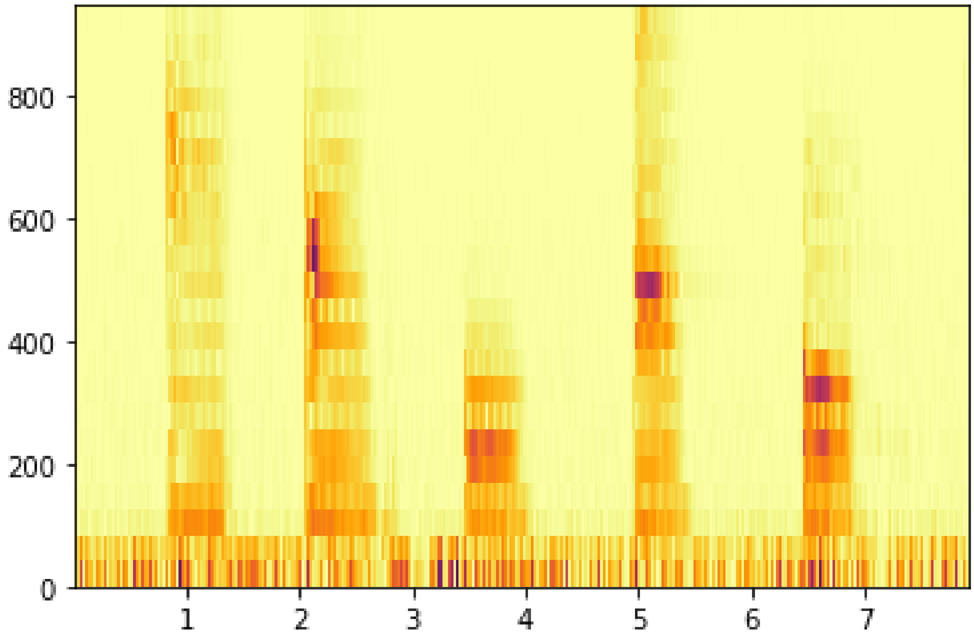

Рисунок 3.10

Пики на спектограмме являются гласными звуками, полосы снизу - это фоновый шум

3.7. Вывод

В этой работе были расмотрены апериодические сигналы, частотные компоненты которых изменяются во времени. Также в этой главе были рассмотрены спектрограммы - способ визуализации апериодичных сигналов.

4. Шумы

4.1. Упражнение 1

«A Soft Murmur» — это веб-сайт, на котором можно послушать множество естественных источников шума, включая дождь, волны, ветер и т. д.
На http://asoftmurmur.com/about/ вы можете найти их список записей, большинство из которых находится на http://freesound.org.
Загрузите несколько таких файлов и вычислите спектр каждого сигнала. Спектр мощности похож на белый шум, розовый шум, или броуновский шум? Как изменяется спектр во времени?

Скачаем файл со звуком дождя и грозы и выберем небольшой фрагмент:

```
wave = read_wave('thunderstorm.wav')
wave.make_audio()
segment = wave.segment(start=1.5, duration=1.0)
segment.make_audio()

spectrum = segment.make_spectrum()
spectrum.plot_power(high=5000)
decorate(xlabel='Frequency (Hz)', ylabel='Power')
```

Спектр:

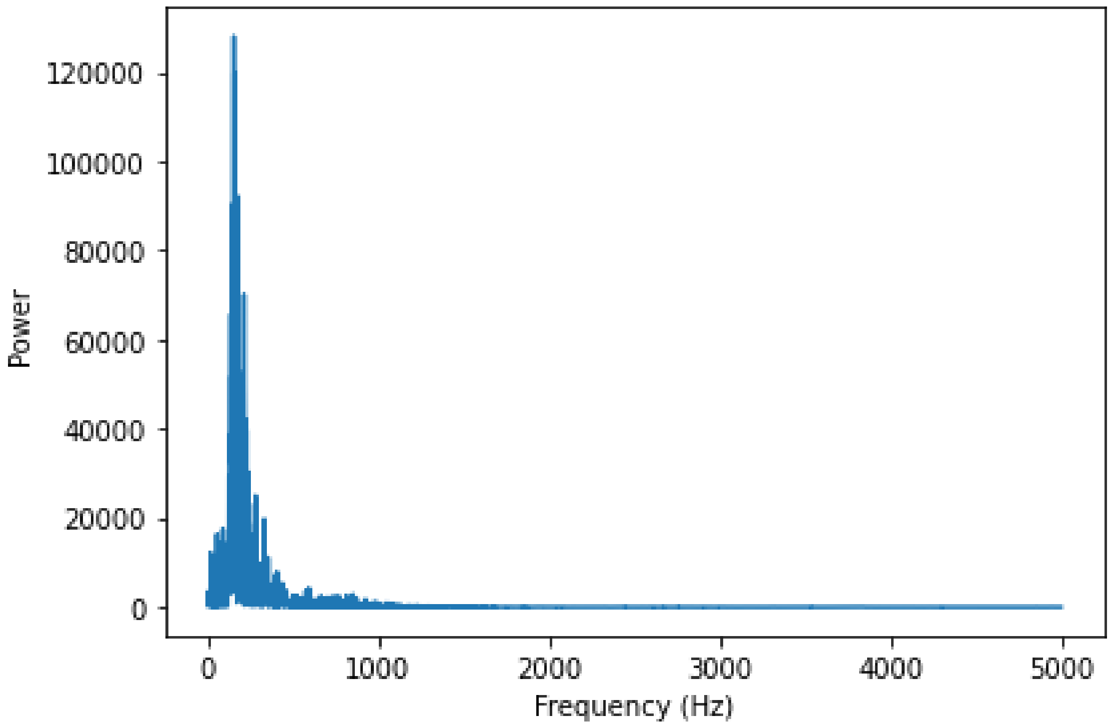

Рисунок 4.1

С увиличением частоты амплитуда резко падает, что похоже на розовый шум. Построим график в логарифмическом масштабе:

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

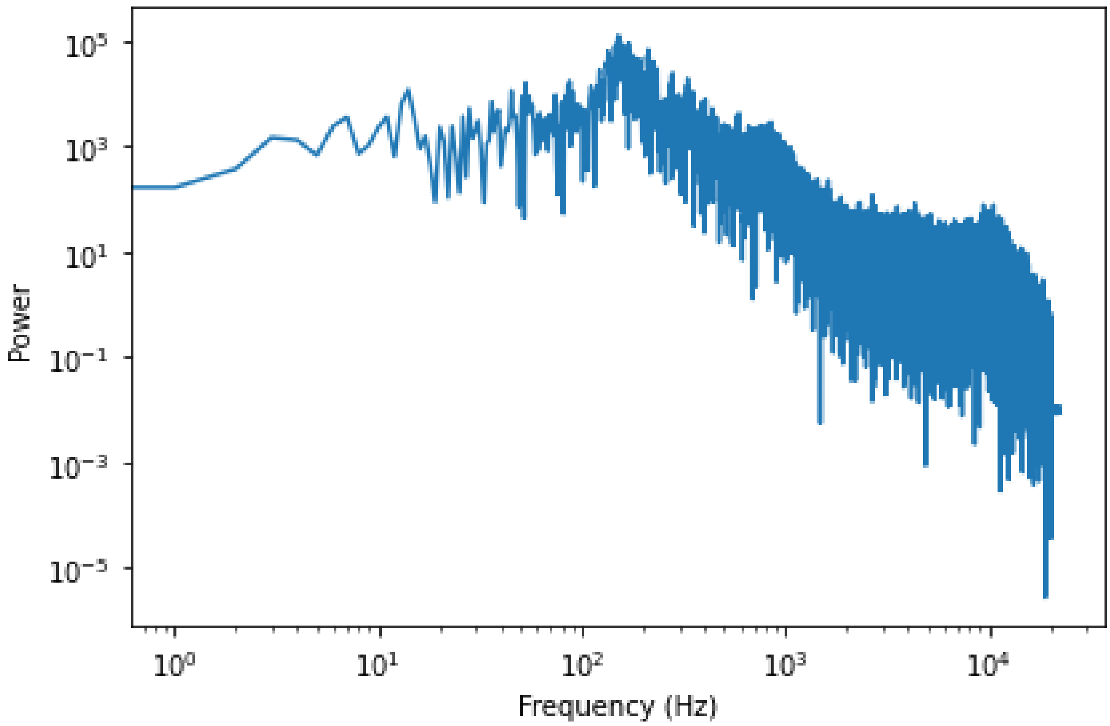

Рисунок 4.2

Чтобы увидеть, как меняется спектр с течением времени, выберем другой сегмент и построим графики обоих сегментов:

```
segment2 = wave.segment(start=2.5, duration=1.0)
segment2.make_audio()
spectrum2 = segment2.make_spectrum()
spectrum.plot_power(alpha=0.5, high=3000)
spectrum2.plot_power(alpha=0.5, high=3000)
decorate(xlabel='Frequency (Hz)', ylabel='Power')
```

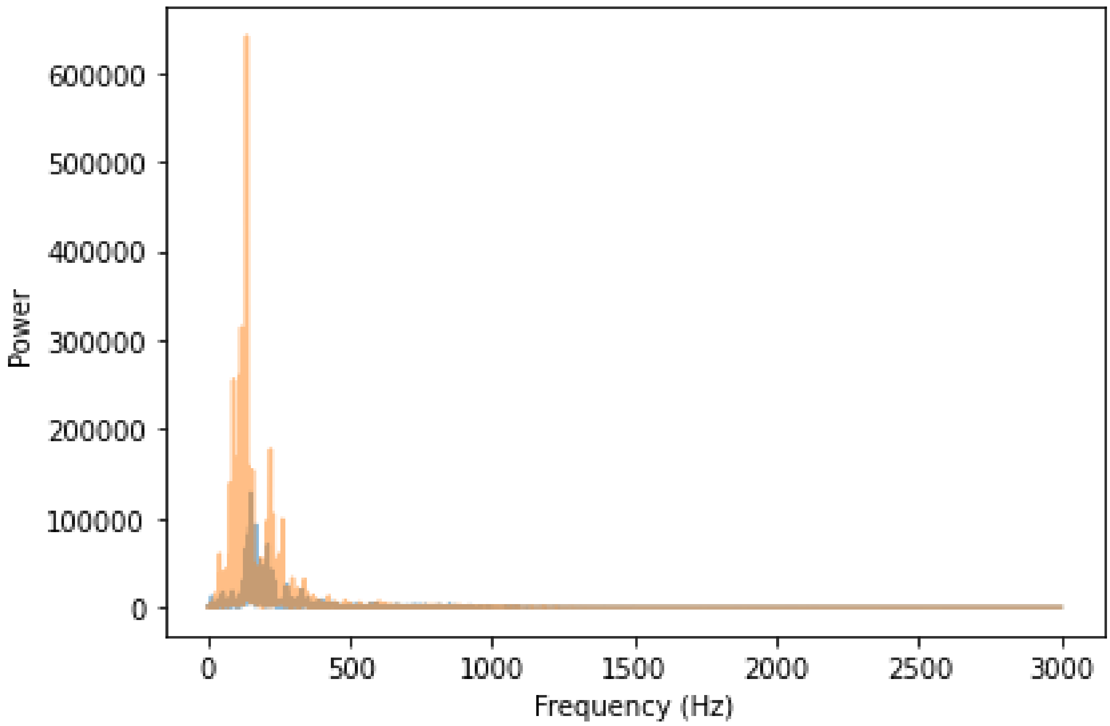

Рисунок 4.3

В логарифмическом масштабе:

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

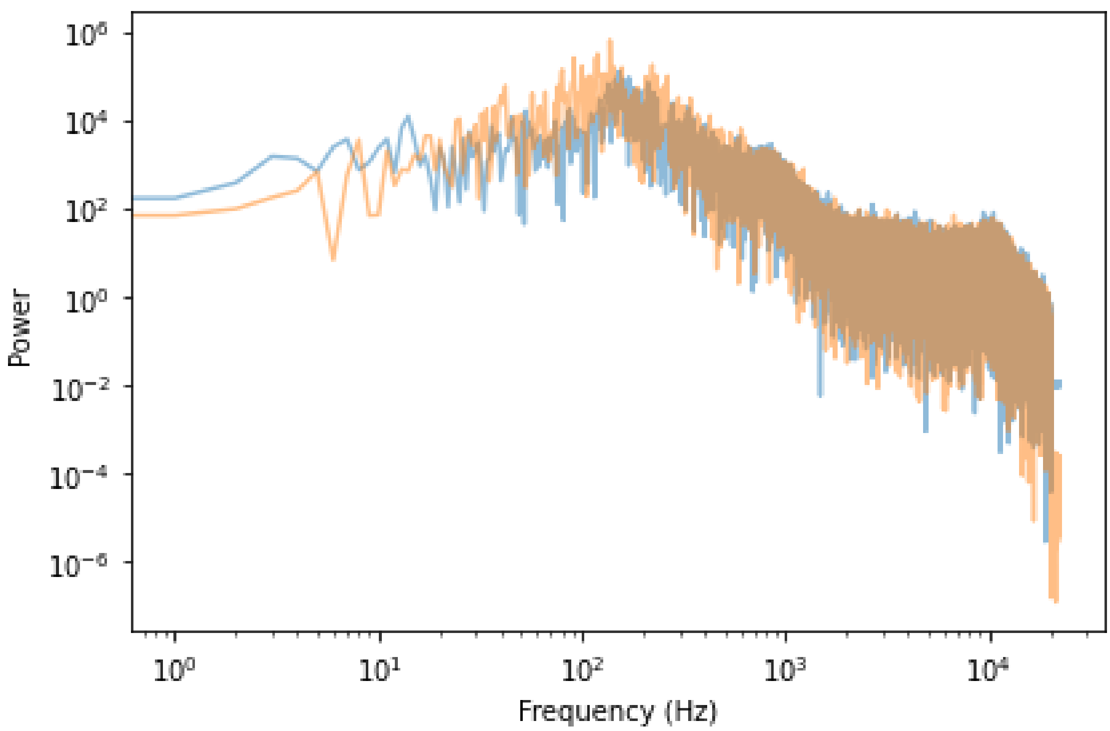

Рисунок 4.4

Можно сделать вывод, что форма графика остается одинаковой с течением времени.

4.2. Упражнение 2

Реализуйте метод Бартлетта[**barlett**] и используйте его для оценки спектра мощности шумового сигнала. Подсказка: посмотрите на реализацую make_spectrogram.

Нужно разложить сигнал на сегменты, после этого для каждого сегмента вычислить квадрат спектра, сложить их все, разделить на количество и взять корень.

```
def bartlett_method(wave, seg_length=512, win_flag=True):
    spectrogram = wave.make_spectrogram(seg_length, win_flag)
    spectrums = spectrogram.spec_map.values()
    psds = [spectrum.power for spectrum in spectrums]
    hs = np.sqrt(sum(psds) / len(psds))
    fs = next(iter(spectrums)).fs
    return Spectrum(hs, fs, wave.framerate)
```

```
bartlett1 = bartlett_method(segment, 1024)
bartlett2 = bartlett_method(segment2, 1024)
bartlett1.plot_power(high=3000)
bartlett2.plot_power(high=3000)
decorate(xlabel='Frequency (Hz)', ylabel='Power', **loglog)
```

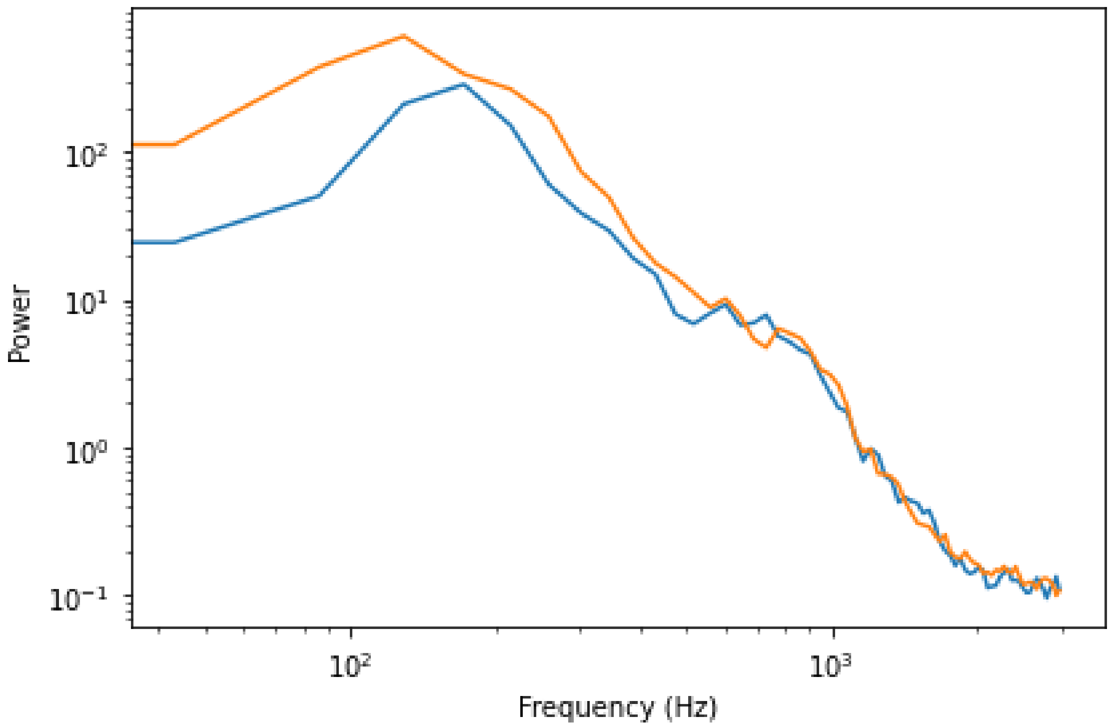

Рисунок 4.5

Видно, что отношение между квадратом амплитуды и частотой сохраняется на разных сегментах.

4.3. Упражнение 3

Загрузите в виде CSV-файла исторические данные о ежедневной цене BitCoin. Откройте этот файл и вычислите спектр цен BitCoin как функцию времени. Похоже ли это на белый, розовый или броуновский шум?

Загрузим csv файл с ценами на биткоин за каждые 3 дня с 2015 по 2022 года:

```
if not os.path.exists('bitcoin.csv'):
    !wget https://github.com/CliffBooth/telecom_labs/raw/main/
    samples/bitcoin.csv

import pandas as pd
csv = pd.read_csv('bitcoin.csv', parse_dates=[0])

ys = csv['market-price']
ts = csv.index
wave = Wave(ys, ts, framerate=1)
wave.plot()
```

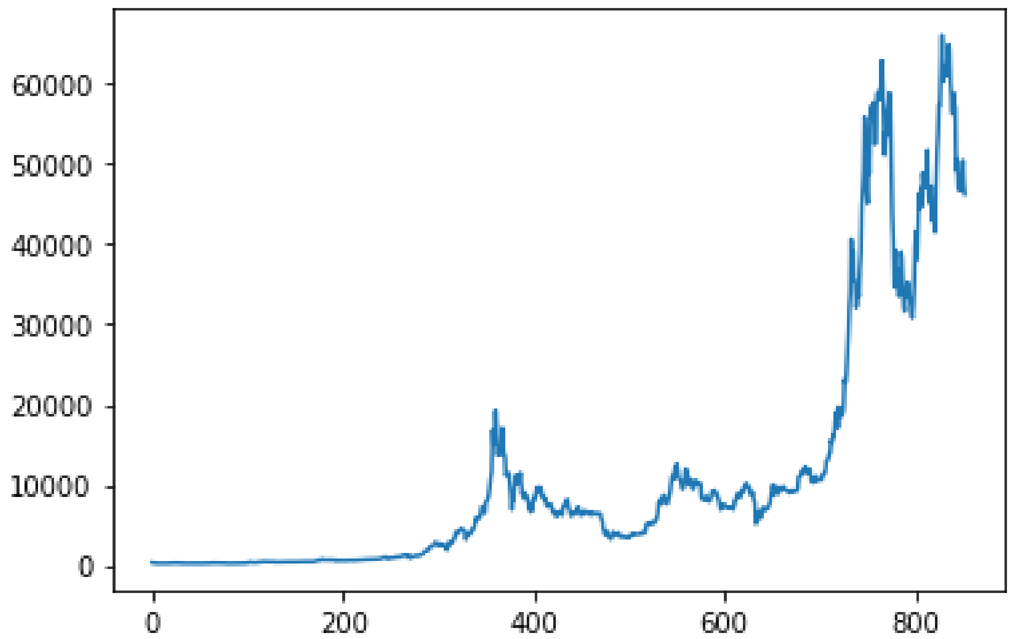

Рисунок 4.6

```
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Frequency', ylabel='Power', **loglog)
```

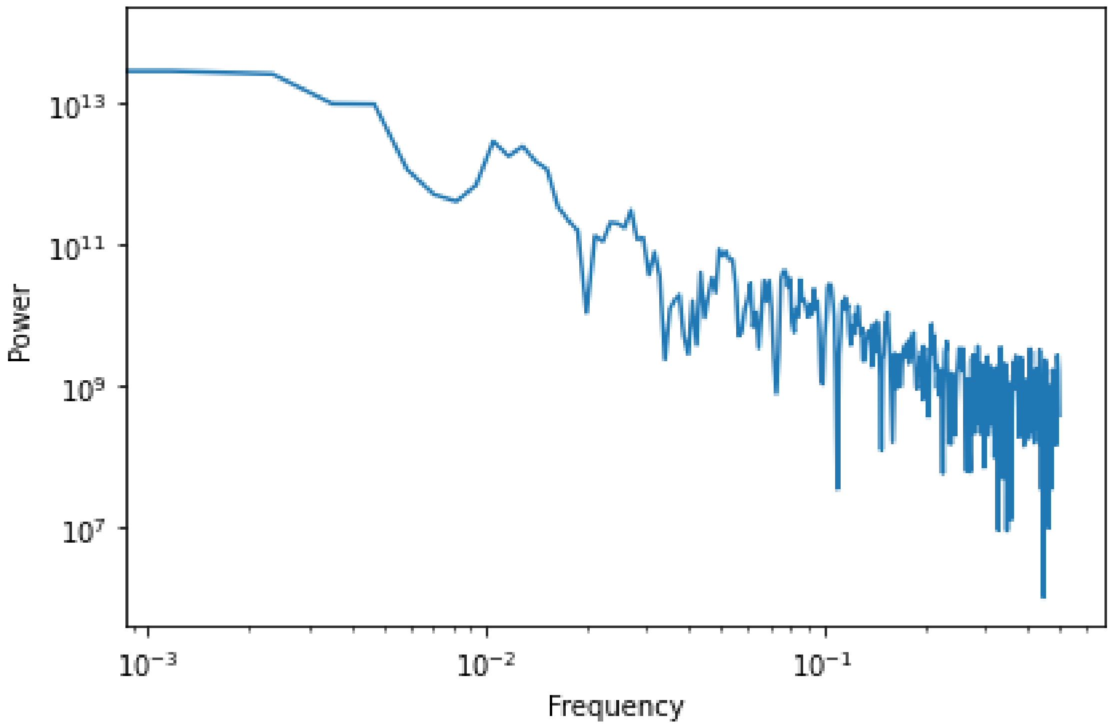

Рисунок 4.7

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

```
-1.8707327981737178
```

Уклон спектра мощности равен примерно -1.87, так что это очень близко к красному 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.

Создайте около секунды шума UP и послушайте его. Для низких значений «ампер», например 0,001, это должно звучать как счетчик Гейгера. Для более высоких значений это должно звучать как белый шум. Вычислите и начертите спектр мощности, чтобы увидеть, похож ли он на белый шум.

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

```
    return ys
```

Посмотрим на график волны:

```
amp = 0.001
framerate = 10_000
duration = 1
signal = UncorrelatedPoissonNoise(amp=amp)
wave = signal.make_wave(duration=duration, framerate=framerate)
wave.plot()
```

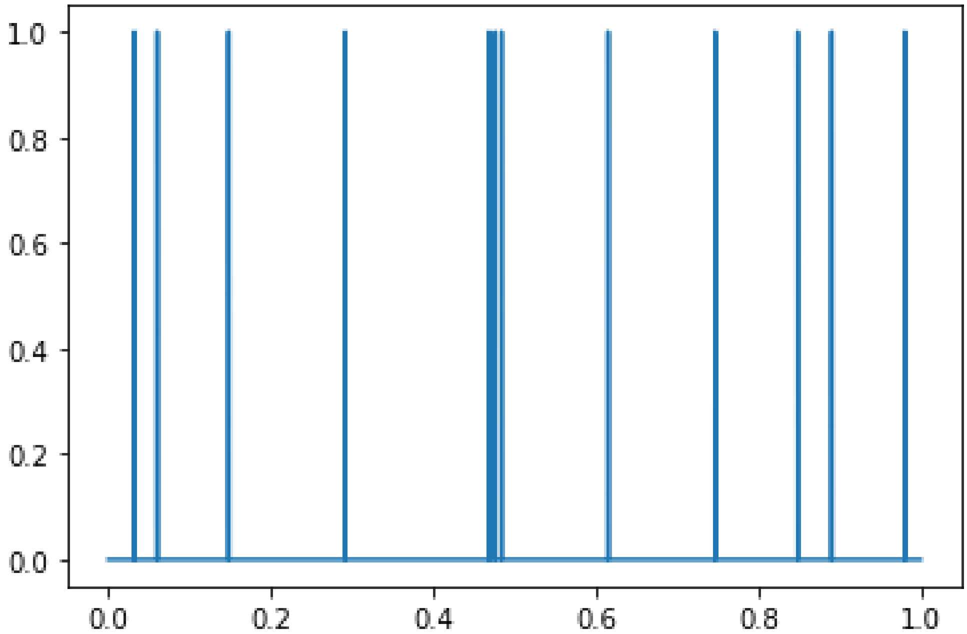

Рисунок 4.8

График спектра в логарифмическом масштабе:

```
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Frequency␣(Hz)', ylabel='Power', **loglog)
```

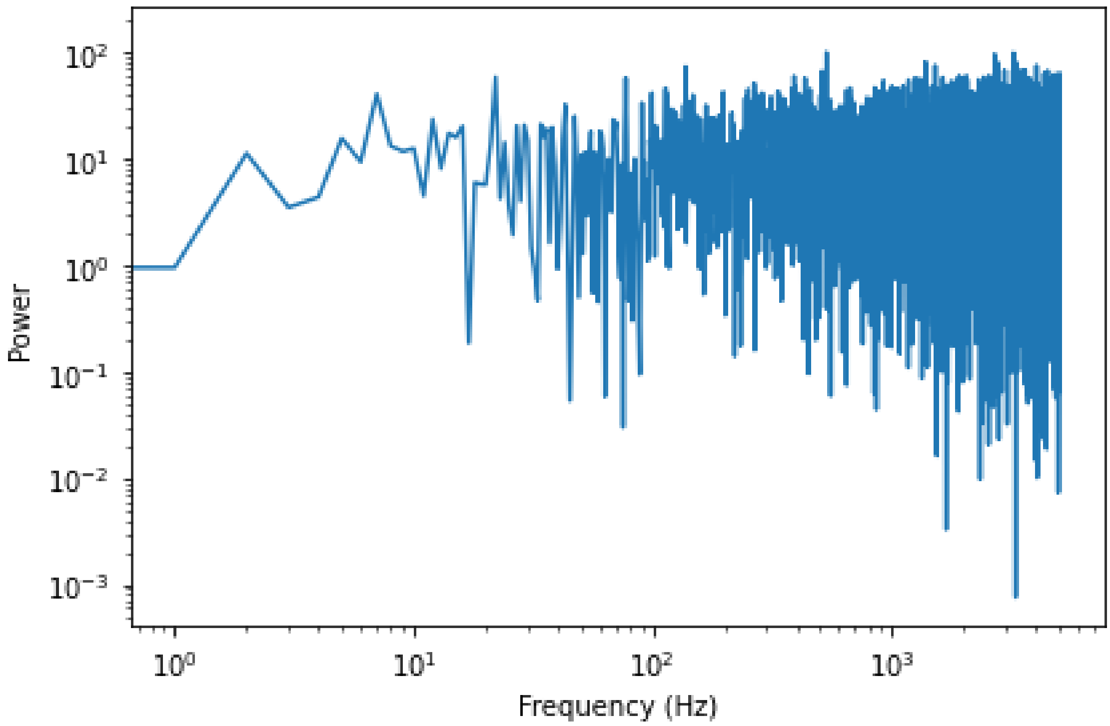

Рисунок 4.9

При увеличении апмлитуды, сигнал будет звучать как белый шум:

```
amp = 1
framerate = 10000
duration = 1

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

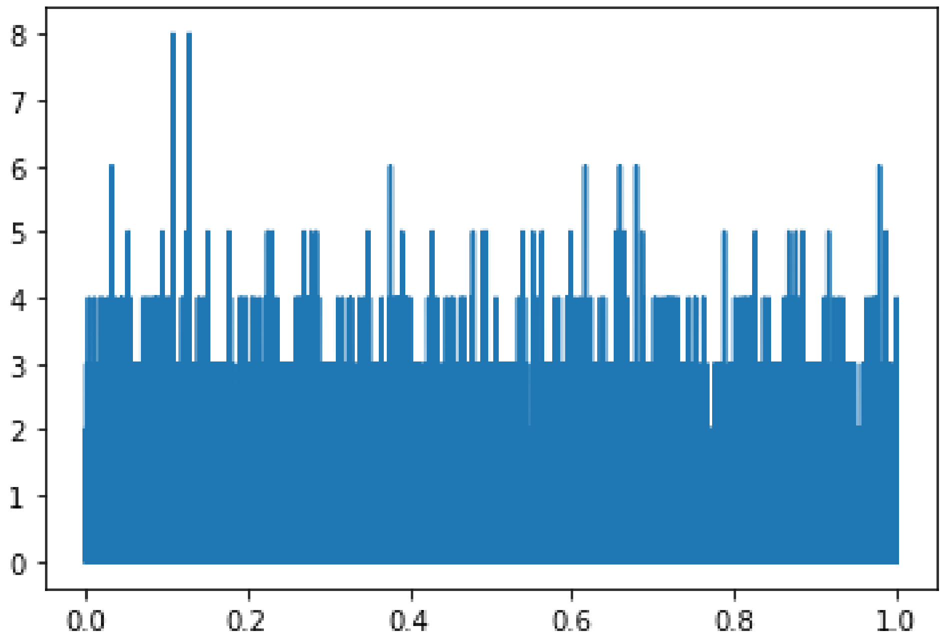

Рисунок 4.10

4.5. Упражнение 5

В этой главе описан алгоритм генерации розового шума. Концептуально простой, но вычислительно затратный. Есть более эффективные альтернативы, такие как алгоритм Восса-Маккартни.
Исследуйте этот метод, реализуйте его, вычислите спектр и подтвердите, что он имеет желаемое отношение между мощностью и частотой.

```
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
```

Используя Voss-McCartney алгоритм, сгенерируем 11025 значений и создадим волну:

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

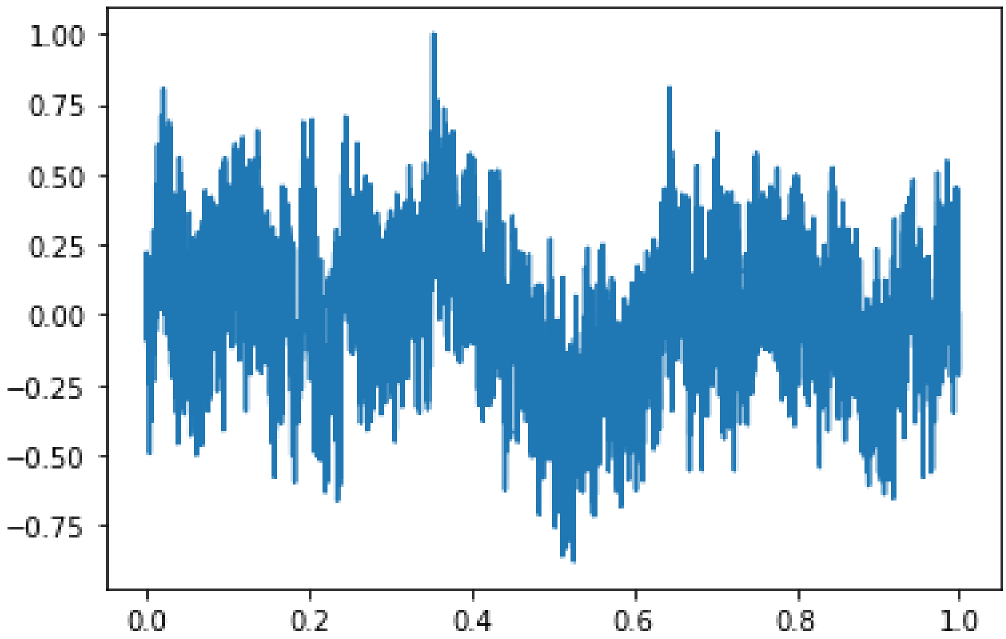

Рисунок 4.11

Спектр волны:

```
spectrum = wave.make_spectrum()
spectrum.hs[0] = 0
spectrum.plot_power()
decorate(xlabel='Frequency␣(Hz)', ylabel='Power', **loglog)
```

z

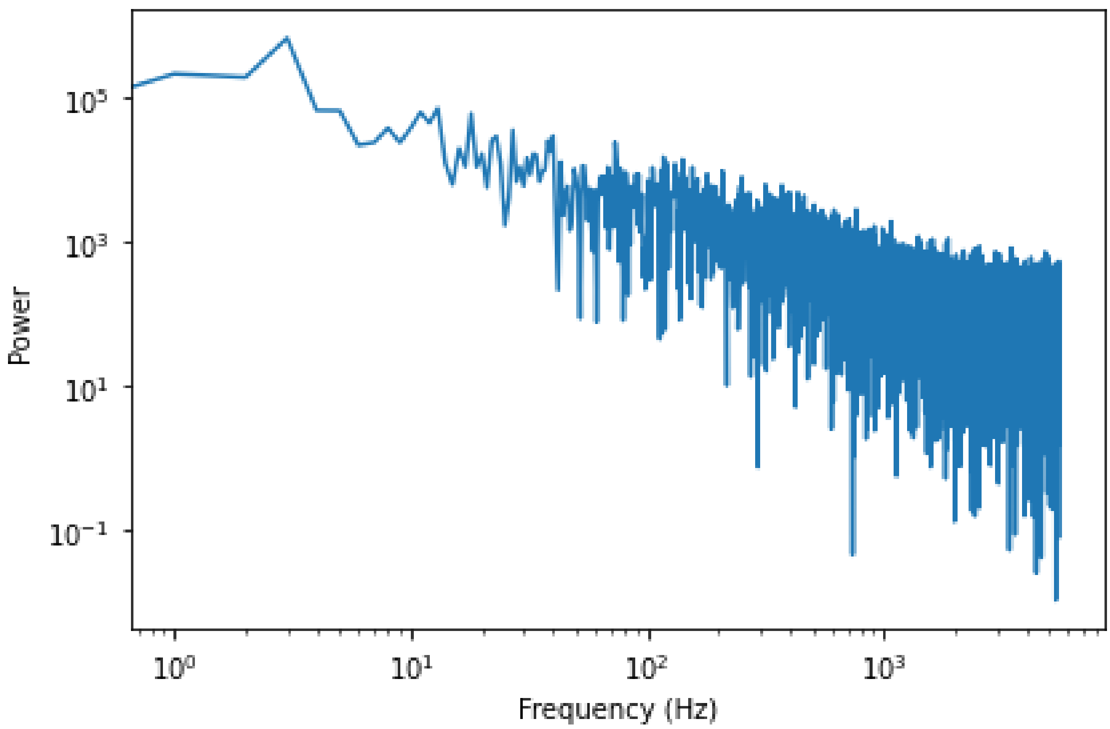

Рисунок 4.12

4.6. Вывод

В этой работе был рассмотрен шум. Шум - сигнал, содержащий компоненты с самыми разными частотами, но не имеющий гармонической структуры периодических сигналов, рассмотреных в предыдущих работах.

5. Автокорреляция

5.1. Упражнение 1

Оцените высоты тона вокального чирпа для нескольких времён начала сегмента.

Скачаем чирп и создадим 3 сегмента с разными моментами старта::

```
if not os.path.exists('28042__bcjordan__voicedownbew.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/28042__b
wave = read_wave('28042__bcjordan__voicedownbew.wav')
wave.normalize()
wave.make_audio()
duration = 0.01
segment1 = wave.segment(start=0.0, duration=duration)
segment2 = wave.segment(start=0.1, duration=duration)
segment3 = wave.segment(start=0.2, duration=duration)
```

Используем автокорреляцию для определения высоты тона:

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

lags1, corrs1 = autocorr(segment1)
plt.plot(lags1, corrs1)
lags2, corrs2 = autocorr(segment2)
plt.plot(lags2, corrs2)
lags3, corrs3 = autocorr(segment3)
plt.plot(lags3, corrs3)
decorate(xlabel='Lag (index)', ylabel='Correlation')
```

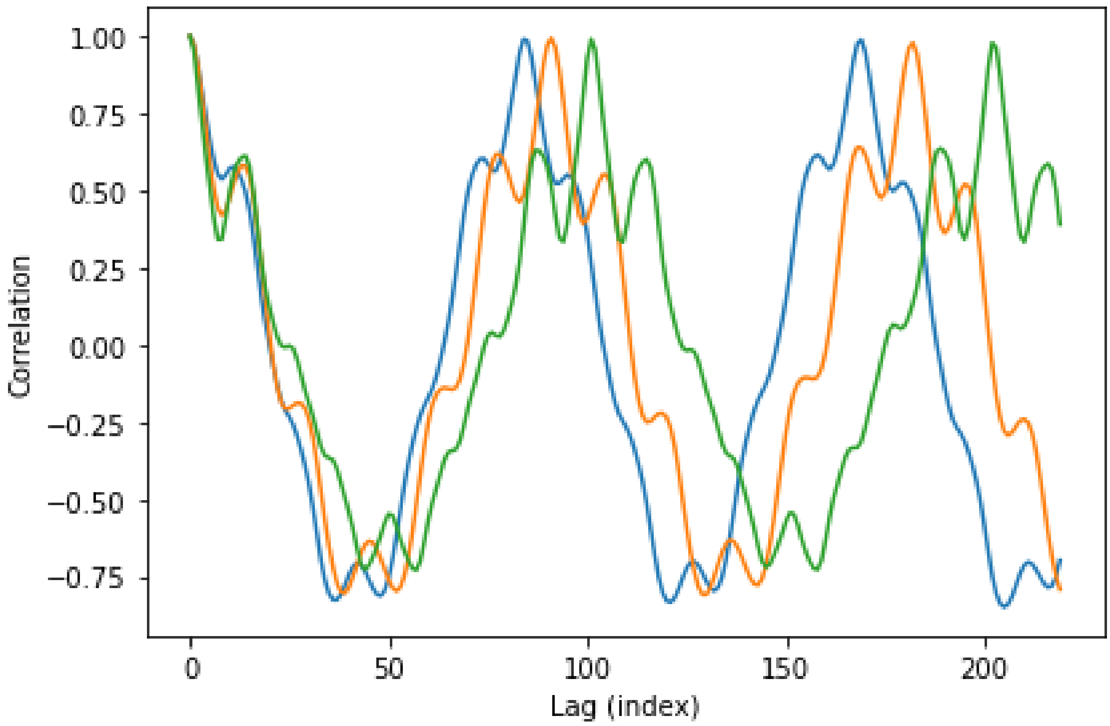

Рисунок 5.1

Узнаем значения первого пика для каждого сегмента:

```
low, high = 50, 140
lag1 = np.array(corrs1[low:high]).argmax() + low
lag2 = np.array(corrs2[low:high]).argmax() + low
lag3 = np.array(corrs3[low:high]).argmax() + low
lag1, lag2, lag3
```

```
(84, 91, 101)
```

Найдем частоты пиков:

```
period1 = lag1 / segment1.framerate
period2 = lag2 / segment2.framerate
period3 = lag3 / segment3.framerate
frequency1 = 1 / period1
frequency2 = 1 / period2
frequency3 = 1 / period3
frequency1, frequency2, frequency3
```

```
(525.0, 484.6153846153846, 436.63366336633663)
```

5.2. Упражнение 2

Инкапсулировать код автокорреляции для оценки основной частоты периодического сигнала в функцию, названную estimate_fundamental, и исользуйте её для отслеживания высоты тона записанного звука.

Посмотрим на спектограмму вокального чирпа:

```
wave.make_spectrogram(1024).plot(4000)
```

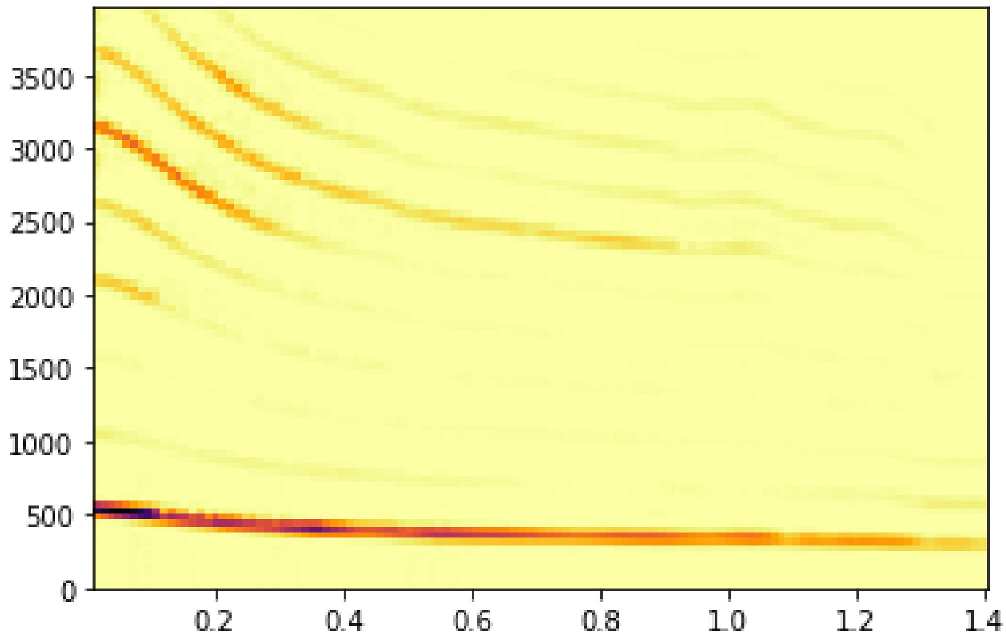

Рисунок 5.2

Напишем функцию estimate_fundamental, которая будет находить фундаментальную частоту с помощью автокорреляции.

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

Значение базовой частоты для всей записи:

```
estimate_fundamental(wave)
```

```
436.63366336633663
```

Для большей точности, разделим запись на небольшие сегменты, для каждого вычислим значение фундаментальной частоты и отобразим их на графике поверх спектограммы, полученной выше. Жирная нижняя линия (базовая частота) должна совпасть с графиком.

```
duration = wave.duration
step = 0.01
start = 0
time = []
freq = []
while start + step < duration:
  time.append(start + step)
  freq.append(estimate_fundamental(wave.segment(start=start, duration=step)
  start += step
plt.plot(time, freq)
wave.make_spectrogram(2048).plot(900)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

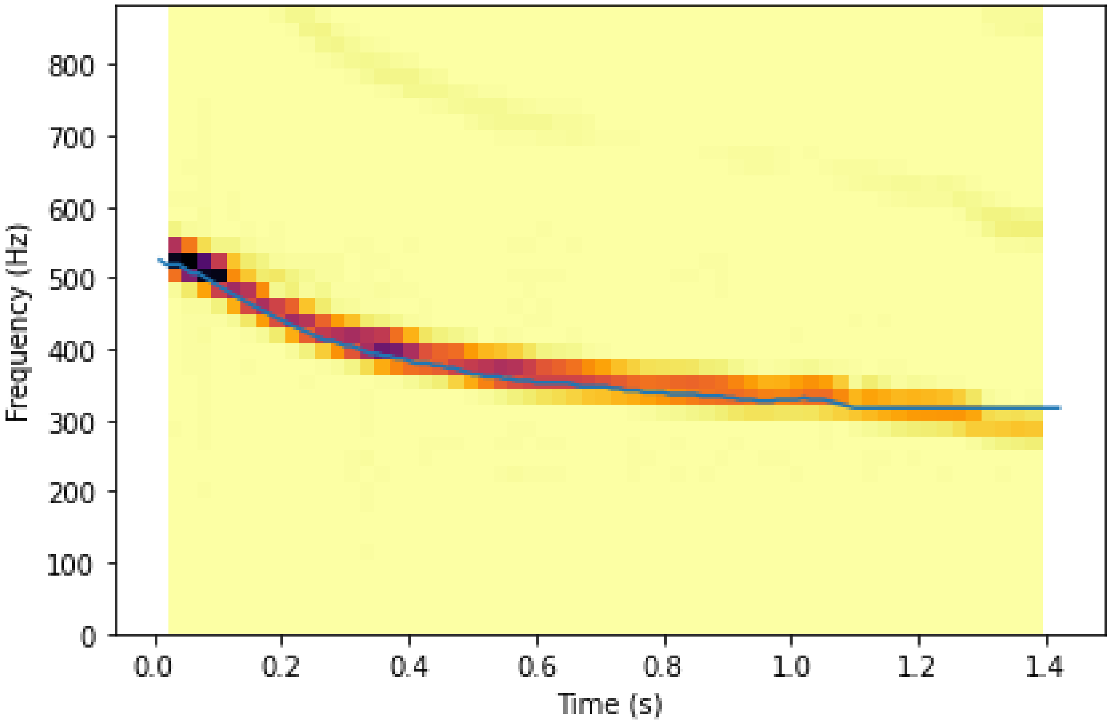

Рисунок 5.3

5.3. Упражнение 3

Вычислить автокорреляцию цен в платёжной системе Bitcoin. Оценить автокореляцию и проверить на признаки переодичности процесса.

Загрузим csv файл:

```
if not os.path.exists('bitcoin.csv'):
    !wget https://github.com/CliffBooth/telecom_labs/raw/main/
    samples/bitcoin.csv

import pandas as pd
csv = pd.read_csv('bitcoin.csv', parse_dates=[0])
ys = csv['market-price']
ts = csv.index
wave = Wave(ys, ts, framerate=1)
wave.plot()
```

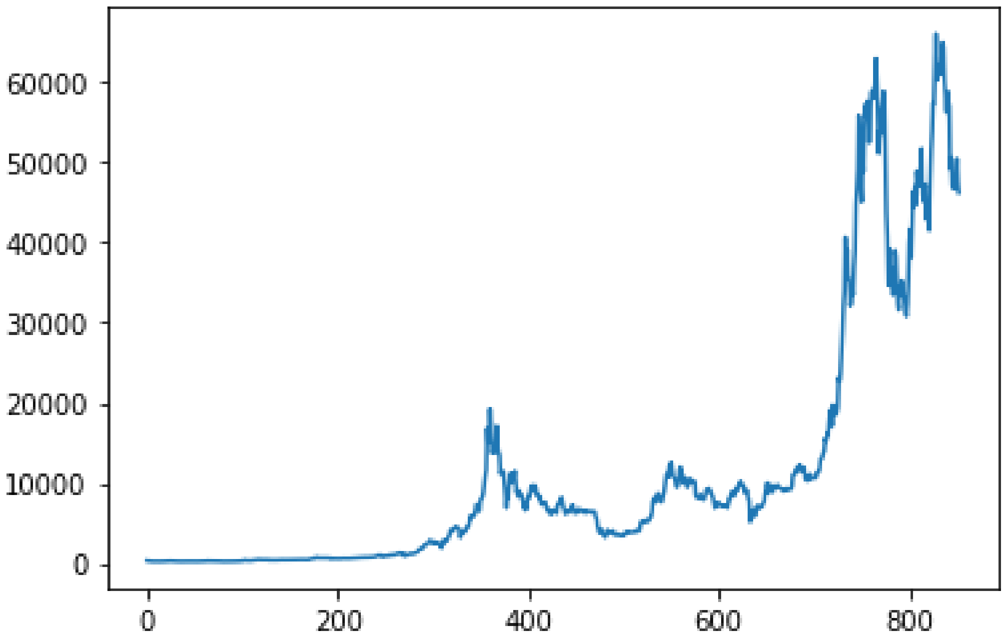

Рисунок 5.4

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

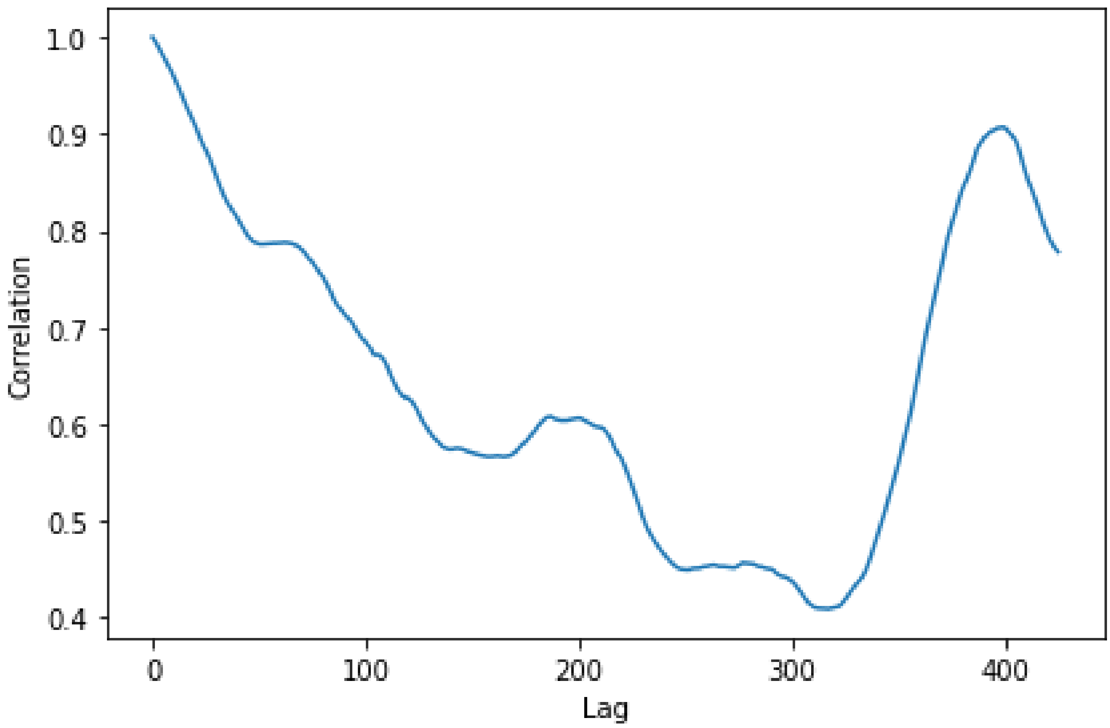

Рисунок 5.5

Функця автокорреляции довольно резко падает, а затем повышается, никаких признаков периодичности нет.

5.4. Упражнение 4

В репозитории этой книги есть блокнот Jupyter под названием saxophone.ipynb, в котором исследуются автокорреляция, восприятие высоты тона и явление, называемое подавленная основная. Прочтите этот блокнот и «погоняйте» примеры. Выберите другой сегмент записи и вновь поработайте с примерами.

Скачаем запись саксофона:

```
if not os.path.exists('100475__iluppai__saxophone-weep.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/100475__i
wave = read_wave('100475__iluppai__saxophone-weep.wav')
wave.normalize()
wave.make_audio()
```

Расммотрим феномен "Отсутсвющая фундаментальная частота":

```
gram = wave.make_spectrogram(seg_length=1024)
gram.plot(high=3000)
decorate(xlabel='Time␣(s)', ylabel='Frequency␣(Hz)')
```

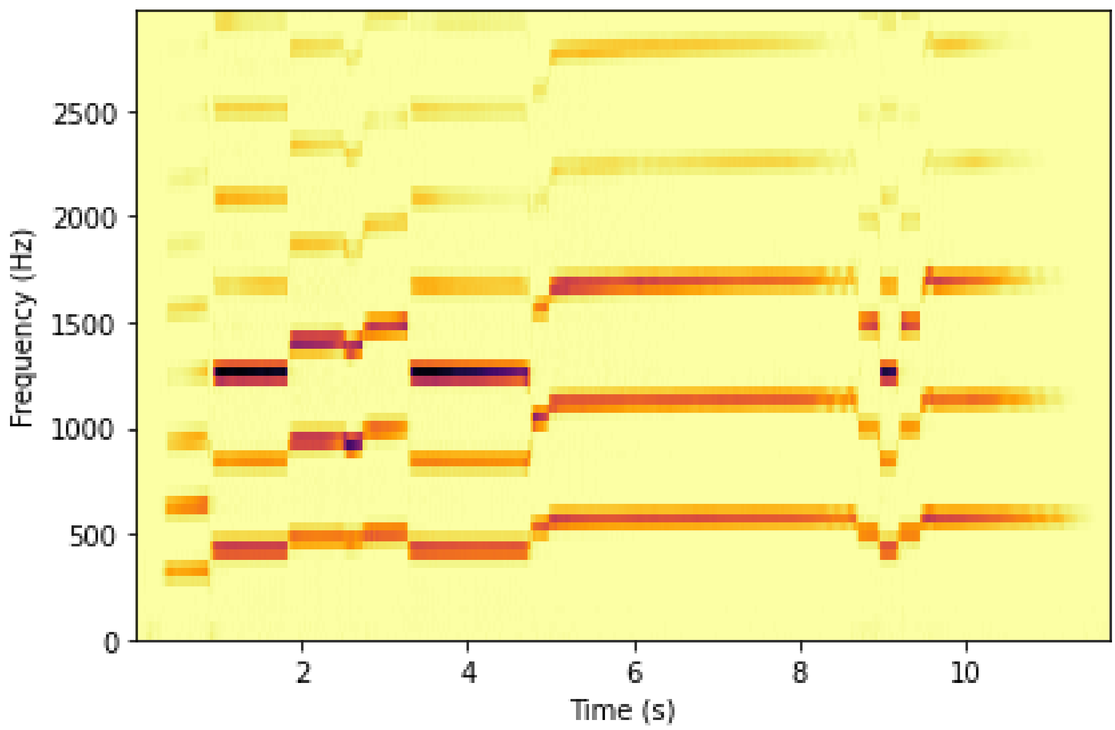

Рисунок 5.6

Высчитаем спектр сегмента около 2 секунд:

```
start = 2.0
duration = 0.5
segment = wave.segment(start=start, duration=duration)
segment.make_audio()
spectrum = segment.make_spectrum()
spectrum.plot(high=3000)
decorate(xlabel='Frequency␣(Hz)', ylabel='Amplitude')
spectrum.peaks()[:10]
```

```
[(2054.0622639591206, 1392.0),
 (1850.8544230639036, 928.0),
 (1684.8468845494765, 1394.0),
 (1332.8150506072802, 930.0),
 (1249.1774991462646, 464.0),
 (1177.6718910227576, 1396.0),
 (857.3729096557305, 932.0),
 (742.841588837269, 1398.0),
 (515.1804113061312, 934.0),
 (513.7226300908811, 466.0)]
```

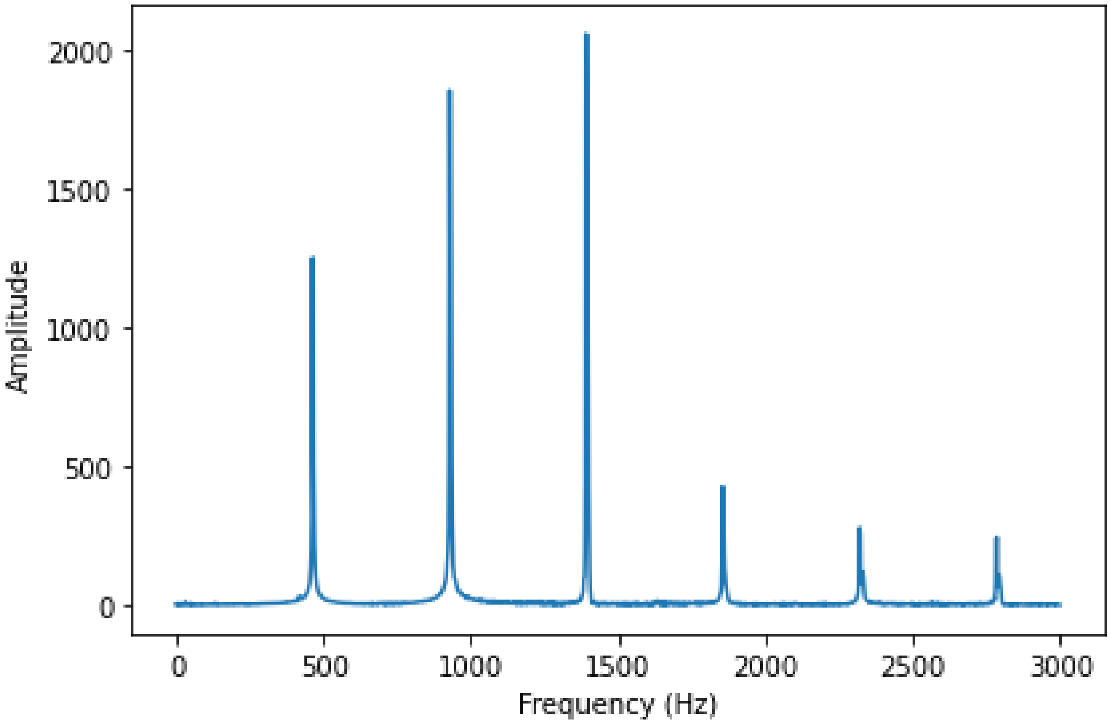

Рисунок 5.7

Пики спектра - 1392б 928 и 464 Гц. При проигрывании, мы слышим фундаментальную частоту - 464 Гц, несмотря на то, что она не доменантная.

```
TriangleSignal(freq=464).make_wave(duration=0.5).make_audio()
```

Попробуем понять, почему мы воспринимаем фундаментальную частоту, несмотря на то, что она не доменантная. Для этого посмотрим на функцию автокорреляции:

```
def autocorr(segment):
    corrs = np.correlate(segment.ys, segment.ys, mode='same')
    N = len(corrs)
    lengths = range(N, N//2, -1)

    half = corrs[N//2:].copy()
    half /= lengths
    half /= half[0]
    return half
```

```
corrs = autocorr(segment)
plt.plot(corrs[:200])
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1.05, 1.05])
```

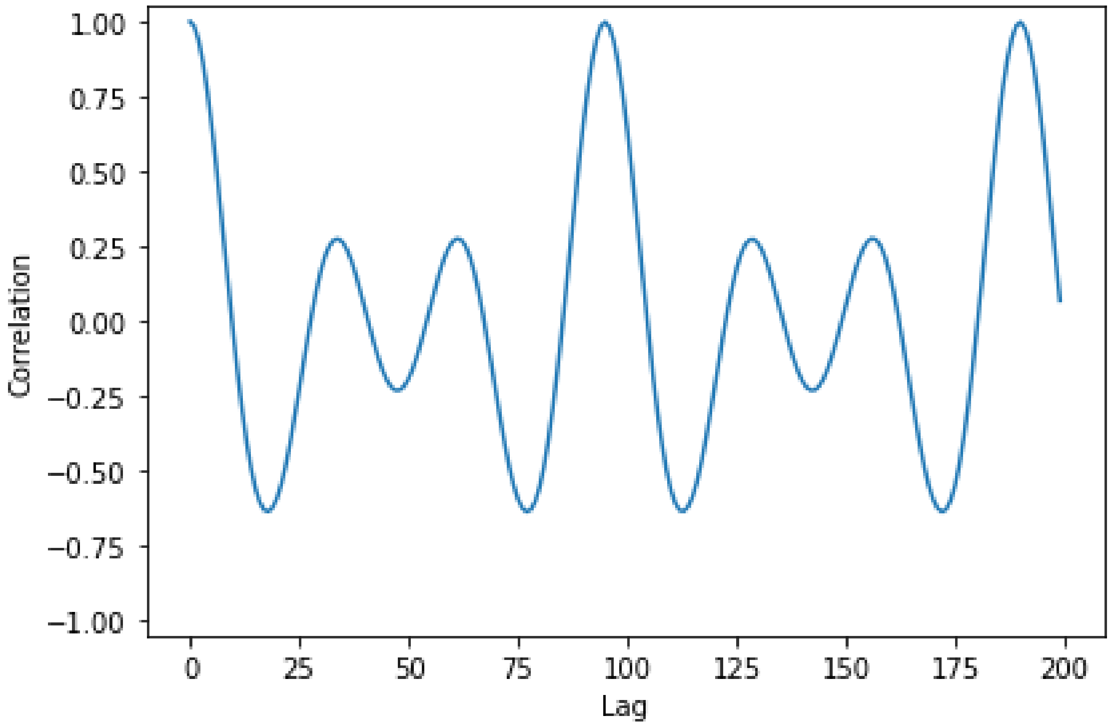

Рисунок 5.8

Первый пик видим около lag = 100

Функция, которая находит самое высокое значение корреляции в заданном диапазоне и возвращает значение чатсоты:

```
def find_frequency(corrs, low, high):
    lag = np.array(corrs[low:high]).argmax() + low
    print(lag)
    period = lag / segment.framerate
    frequency = 1 / period
    return frequency

find_frequency(corrs, 80, 100)
```

```
95
464.2105263157895
```

Получается, что мы слышем ту частоту, которая соответсвует пику в функции корреляции.

Высота звука не изменится, даже если мы уберем фундаментальную частоту:

```
spectrum2 = segment.make_spectrum()
spectrum2.high_pass(600)
spectrum2.plot(high=3000)
decorate(xlabel='Frequency␣(Hz)', ylabel='Amplitude')
segment2 = spectrum2.make_wave()
```

```
segment2.make_audio()
```

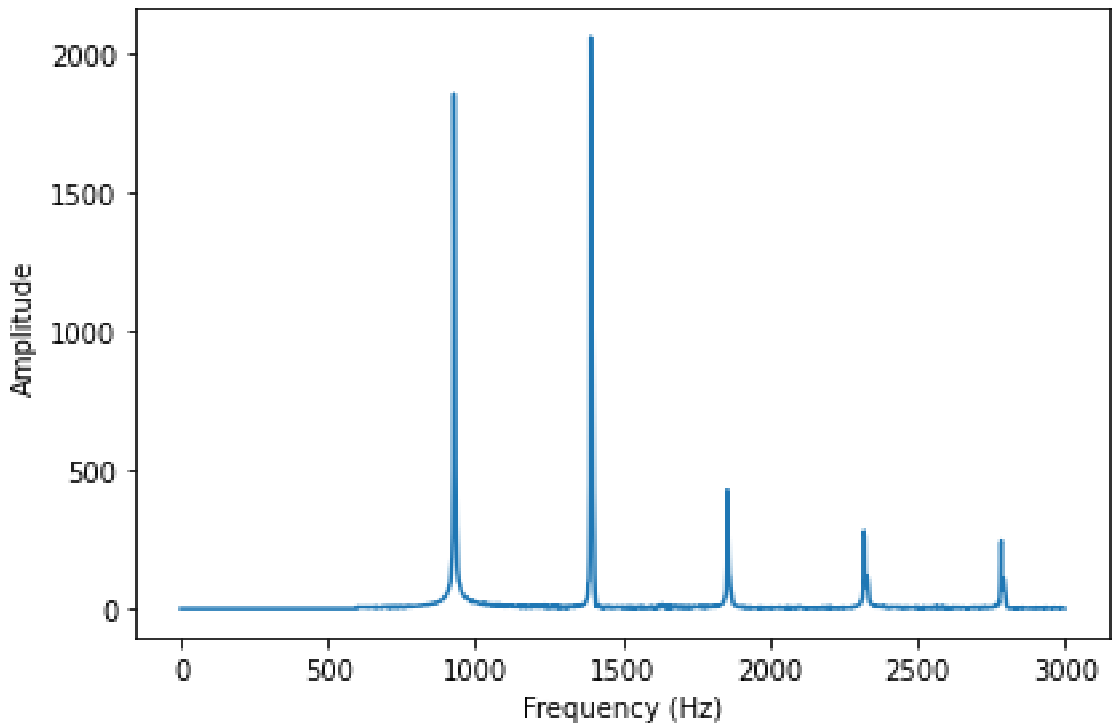

Рисунок 5.9

Это явление и есть "missing fundamental"

Чтобы понять, почему мы слышим частоту, которой даже нет в сигнале, обратимся к функции корреляции:

```
corrs = autocorr(segment2)
plt.plot(corrs[:200])
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1.05, 1.05])
```

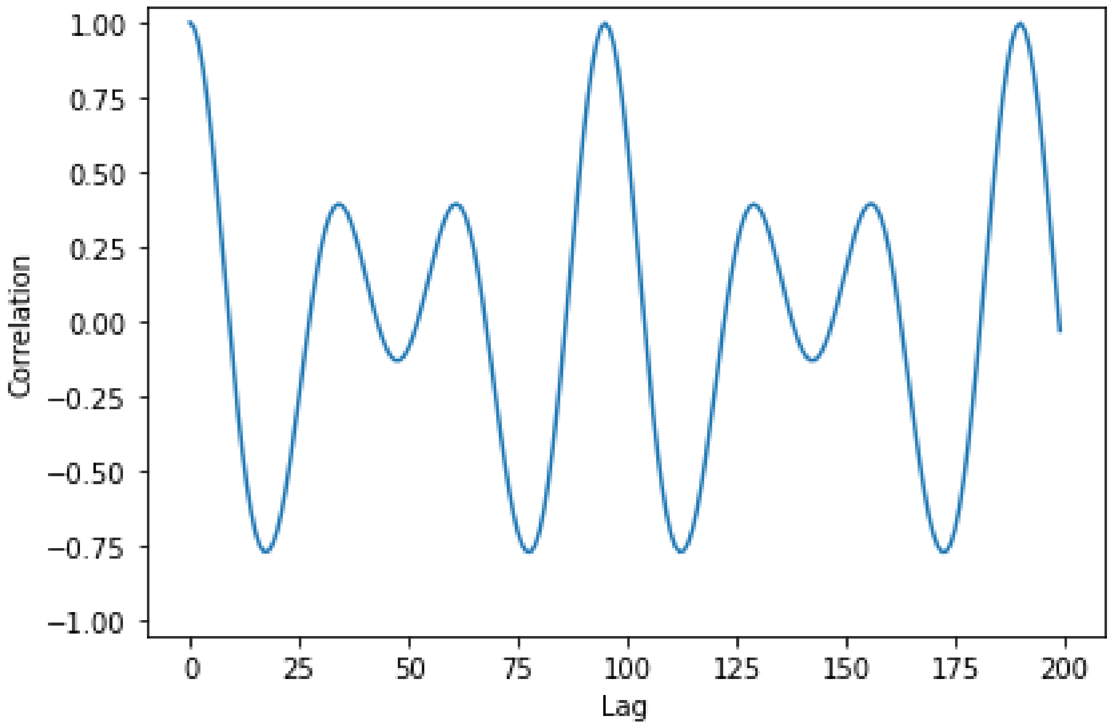

Рисунок 5.10

Все еще присутсвует пик, соответсвующий частоте 464 Гц, но так же есть и другие пики с частотами 1297 и 722

```
find_frequency(corrs, 80, 100)
find_frequency(corrs, 20, 50)
find_frequency(corrs, 50, 80)
```

```
95
464.2105263157895
34
1297.0588235294117
61
722.9508196721312
```

Почему же мы не воспринимаем эти частоты? Потому, что компоненты с более высокими частотами в сигнале являются гормониками 464 Гц, но не 722 Гц и 1297 Гц.

В итоге, это показывает, что восприятие тона основано не только на на спектральном анализе, но и еще на таких вещах как автокорреляция.

5.5. Вывод

В данной главе была изучена корреляция и её роль в сигналах. Также на пратике был обработан сигнал с "missing fundamental". Когда мы убирали основной тон, всё равно звук звучал также.

6. Дискретное косинусное преобразование

6.1. Упражнение 1

Показать на графике время работы analyze1 и analyze2 в логорифмическом масштабе. Сравнить с scipy.fftpack.dct.

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

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

Создадим сгнал, на котором будем тестировать функции:

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

Протестируем фунуции:

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

results = []
for N in ns:
    ts = (0.5 + np.arange(N)) / N
    freqs = (0.5 + np.arange(N)) / 2
    ys = noise.ys[:N]
    result = %timeit -r1 -o analyze1(ys, freqs, ts)
    results.append(result)
best_analyze1 = [result.best for result in results]
```

```
The slowest run took 6.39 times longer than the fastest. This could mean th
1000 loops, best of 1: 344 µs per loop
1000 loops, best of 1: 1.18 ms per loop
100 loops, best of 1: 7.36 ms per loop
10 loops, best of 1: 24.1 ms per loop
10 loops, best of 1: 91.6 ms per loop
1 loop, best of 1: 422 ms per loop
1 loop, best of 1: 2.33 s per loop
10000 loops, best of 1: 133 µs per loop
1000 loops, best of 1: 770 µs per loop
100 loops, best of 1: 4.7 ms per loop
100 loops, best of 1: 13.3 ms per loop
10 loops, best of 1: 46.3 ms per loop
10 loops, best of 1: 160 ms per loop
1 loop, best of 1: 491 ms per loop
```

```
results = []
for N in ns:
  ts = (0.5 + np.arange(N)) / N
  freqs = (0.5 + np.arange(N)) / 2
  ys = noise.ys[:N]
  result = %timeit -r1 -o analyze2(ys, freqs, ts)
  results.append(result)
best_analyze2 = [result.best for result in results]
```

```
10000 loops, best of 1: 133 µs per loop
1000 loops, best of 1: 768 µs per loop
100 loops, best of 1: 4.8 ms per loop
100 loops, best of 1: 13.8 ms per loop
10 loops, best of 1: 47.8 ms per loop
10 loops, best of 1: 159 ms per loop
1 loop, best of 1: 486 ms per loop
```

```
import scipy.fftpack
```

```
results = []
for N in ns:
  ys = noise.ys[:N]
  result = %timeit -r1 -o scipy.fftpack.dct(ys, type=3)
  results.append(result)
best3 = [result.best for result in results]
```

```
The slowest run took 450.01 times longer than the fastest. This could mean
100000 loops, best of 1: 6.72 µs per loop
The slowest run took 7.68 times longer than the fastest. This could mean th
100000 loops, best of 1: 6.9 µs per loop
The slowest run took 6.54 times longer than the fastest. This could mean th
100000 loops, best of 1: 7.91 µs per loop
The slowest run took 73.43 times longer than the fastest. This could mean t
100000 loops, best of 1: 10.4 µs per loop
The slowest run took 44.78 times longer than the fastest. This could mean t
100000 loops, best of 1: 14.4 µs per loop
The slowest run took 30.66 times longer than the fastest. This could mean t
10000 loops, best of 1: 24.8 µs per loop
The slowest run took 18.71 times longer than the fastest. This could mean t
10000 loops, best of 1: 47.6 µs per loop
```

```
plt.plot(ns, best_analyze1, label='analyze1')
plt.plot(ns, best_analyze2, label='analyze2')
plt.plot(ns, best3, label='scipy.fftpack.dct')
loglog = dict(xscale='log', yscale='log')
decorate(xlabel='Wave length (N)', ylabel='Time (s)', **loglog)
```

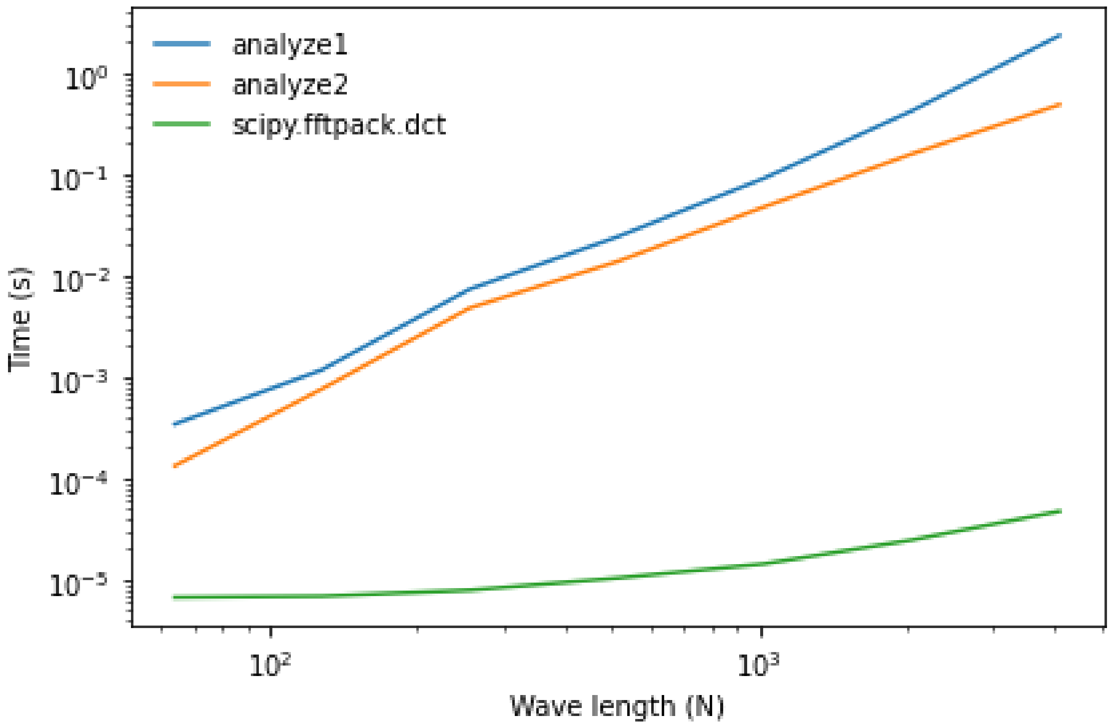

Рисунок 6.1

Реализация scipy.fftpack.dct оказалась самой быстрой.

Время исполнения analyze1 ближе к n^2 чем к ожидаемому n^3. Возможно, это объясняется малым размером массива ns.

6.2. Упражнение 2

Реализовать алгоритм сжатия для музыки или речи.

Загрузим аудиозапись:

```
if not os.path.exists("trumpet.wav"):
  !wget https://github.com/CliffBooth/telecom_labs/raw/main/
  samples/trumpet.wav
wave = read_wave('trumpet.wav')
```

возьмем короткий сегмент:

```
segment = wave.segment(0.5, 0.5)
segment.make_audio()
```

Построим ДКП этого сегмента:

```
seg_dct = segment.make_dct()
seg_dct.plot(high=2000)
decorate(xlabel='Frequency␣(Hz)', ylabel='DCT')
```

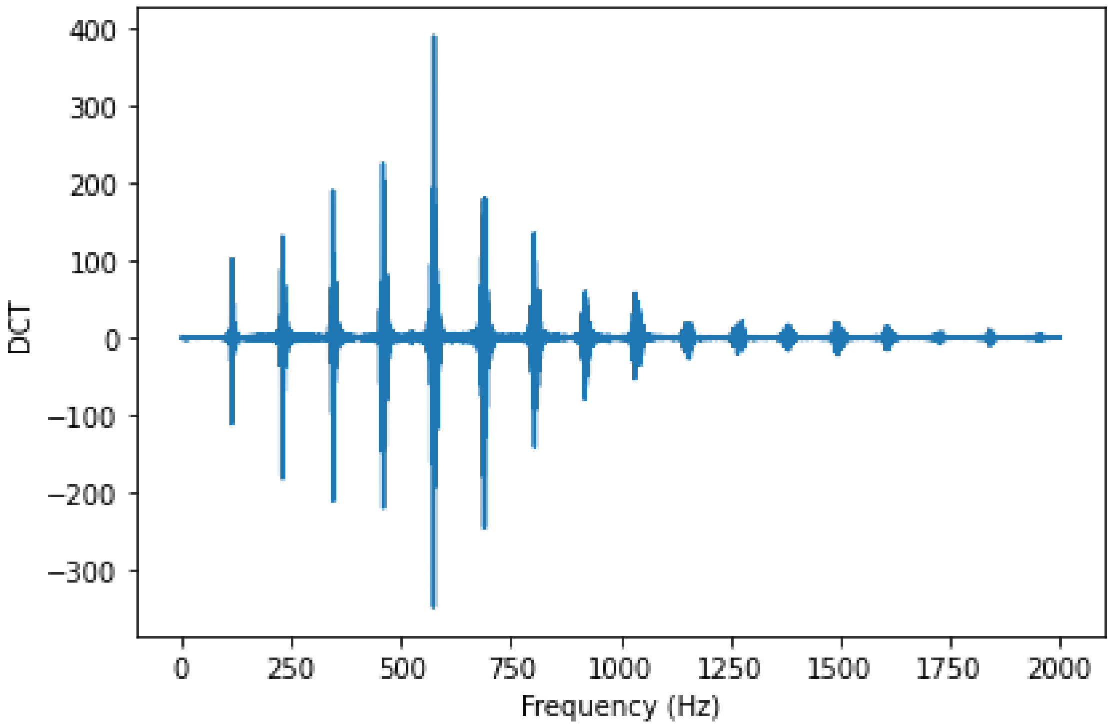

Рисунок 6.2

Напишем функцию сжатия, ктораякоторая будет обнулять элементы ниже порогового значения

```
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')
```

После применения этой функции к сегменту, у нас исчезают более чем 90% всех элементов

```
seg_dct = segment.make_dct()
compress(seg_dct, thresh=2)
seg_dct.plot(high=2000)
```

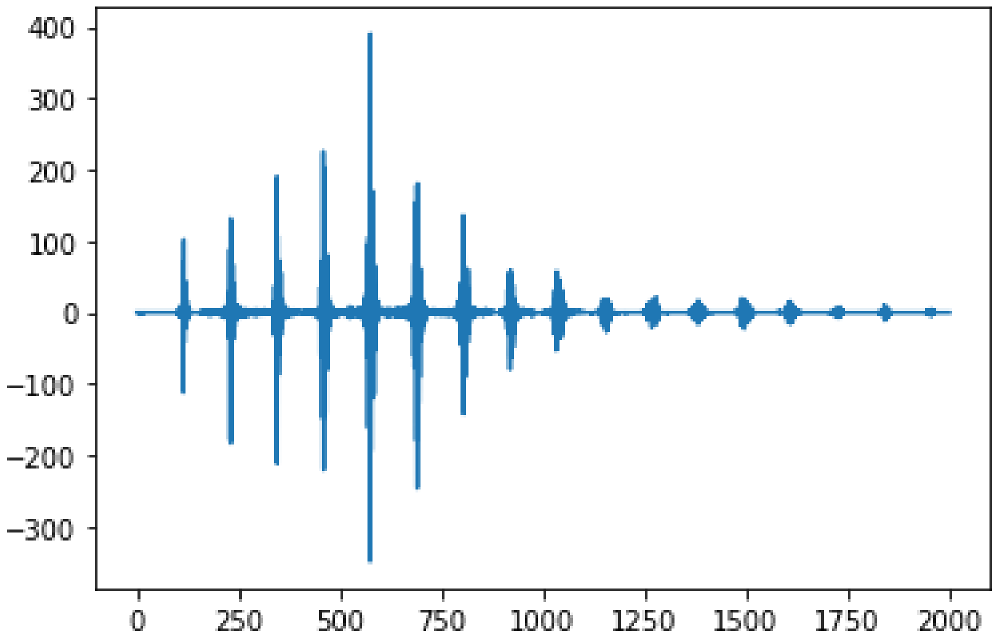

Рисунок 6.3

При этом звук практически не изменился

6.3. Управжнение 3

В блокноте phase.ipynb взять другой сегмент звука и повторить эксперименты.

```
from thinkdsp import SawtoothSignal

signal = SawtoothSignal(freq=500, offset=0)
wave = signal.make_wave(duration=0.5, framerate=40000)

wave.segment(start=0.107, duration=0.01).plot()
decorate(xlabel='Time (s)')
```

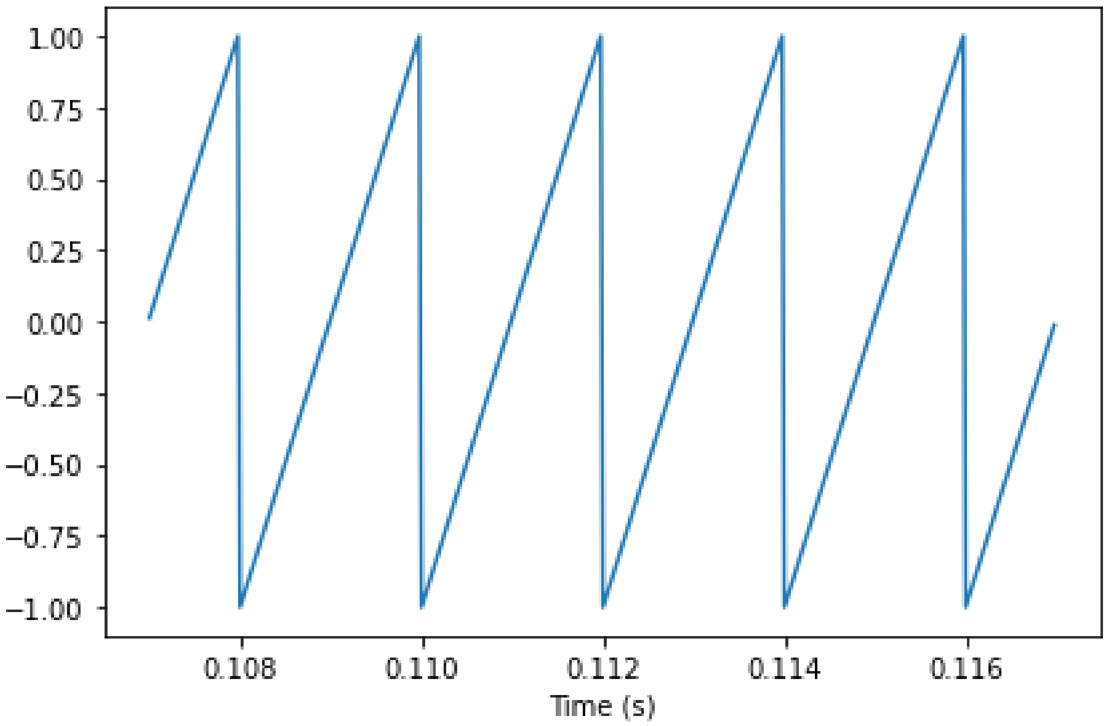

Рисунок 6.4

```
spectrum = wave.make_spectrum()
spectrum.plot()
decorate(xlabel='Frequency␣(Hz)',
         ylabel='Amplitude')
```

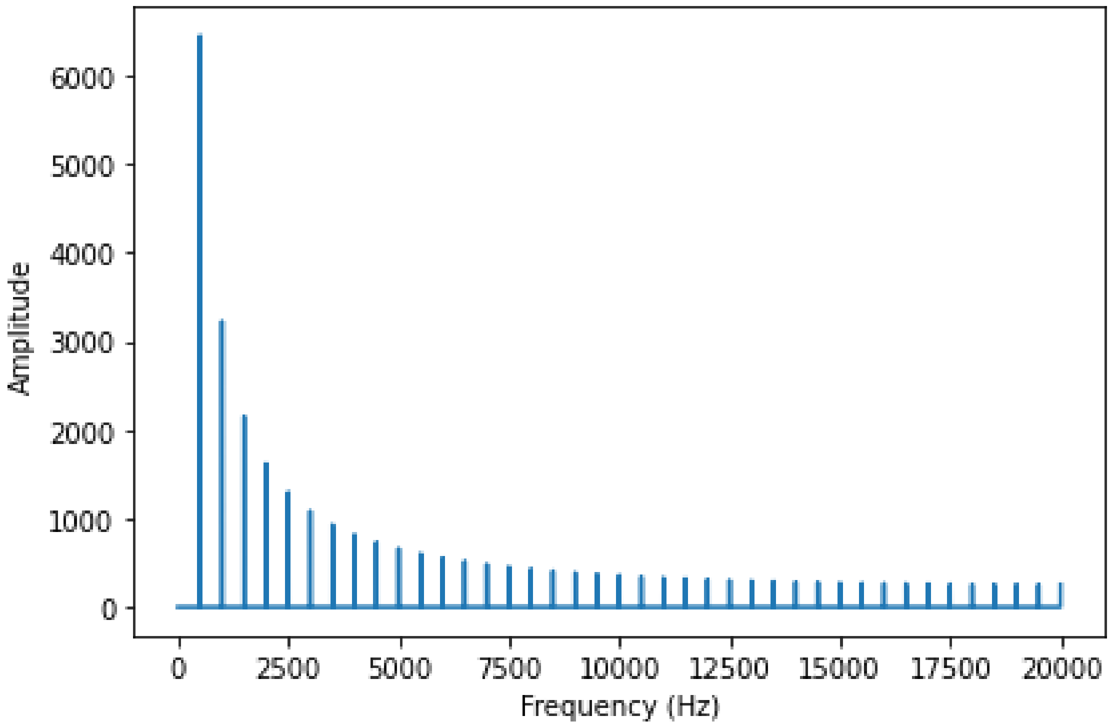

Рисунок 6.5

```
def plot_angle(spectrum, thresh=1):
    angles = spectrum.angles
    angles[spectrum.amps < thresh] = np.nan
    plt.plot(spectrum.fs, angles, 'x')
    decorate(xlabel='Frequency␣(Hz)',
             ylabel='Phase␣(radian)')

plot_angle(spectrum, thresh=0)
```

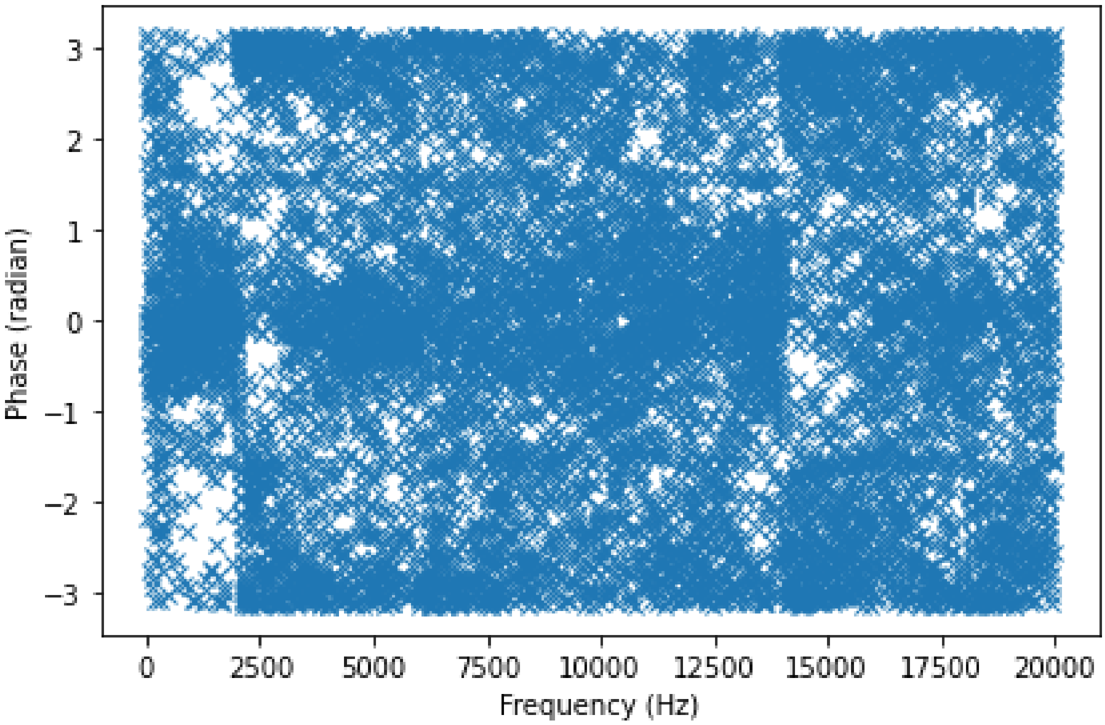

Рисунок 6.6

Если мы выберем только те частоты, у которых значения амплитуды превышают порог, мы увидим определенную структуру

```
plot_angle(spectrum, thresh=1)
```

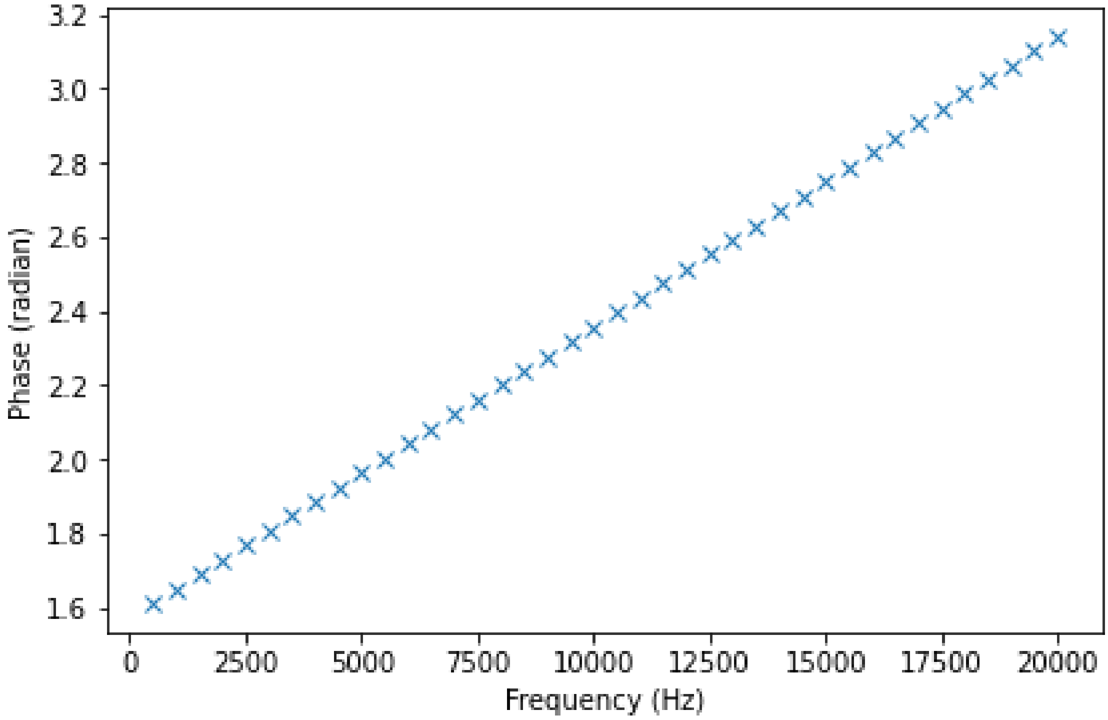

Рисунок 6.7

Функция, которая строит амплитуды, углы и волну:

```
def plot_three(spectrum, thresh=1):
    plt.figure(figsize=(10, 4))
    plt.subplot(1,3,1)
    spectrum.plot()
    plt.subplot(1,3,2)
    plot_angle(spectrum, thresh=thresh)
    plt.subplot(1,3,3)
    wave = spectrum.make_wave()
    wave.unbias()
    wave.normalize()
    wave.segment(duration=0.01).plot()
    display(wave.make_audio())

plot_three(spectrum)
```

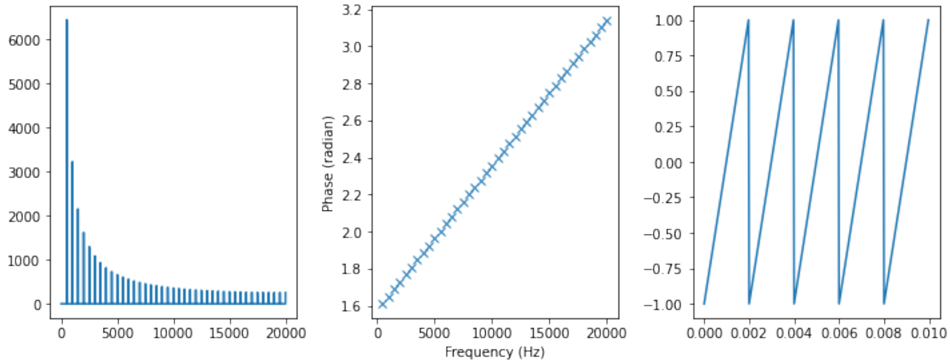

Рисунок 6.8

Зададим все углы равными 0:

```
def zero_angle(spectrum):
    res = spectrum.copy()
    res.hs = res.amps
    return res

spectrum2 = zero_angle(spectrum)
plot_three(spectrum2)
```

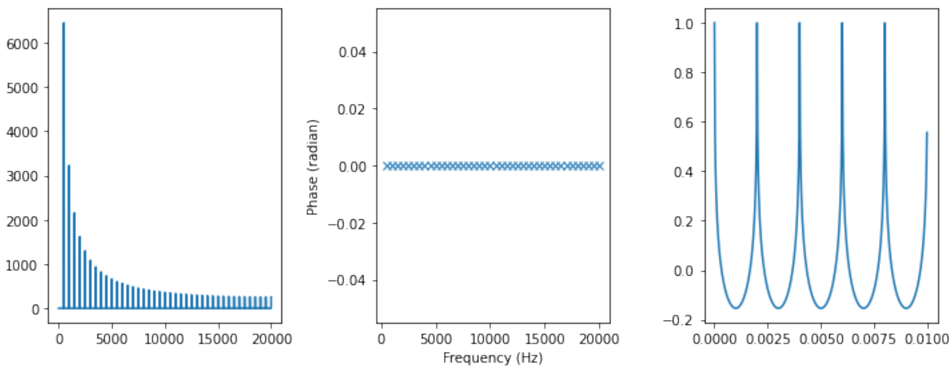

Рисунок 6.9

Амлитуды не поменялись, все углы равны 0, волна теперь выглядит совсем по другому, но звучит также

```
def rotate_angle(spectrum, offset):
        res = spectrum.copy()
        res.hs *= np.exp(1j * offset)
        return res

spectrum3 = rotate_angle(spectrum, 1)
plot_three(spectrum3)
```

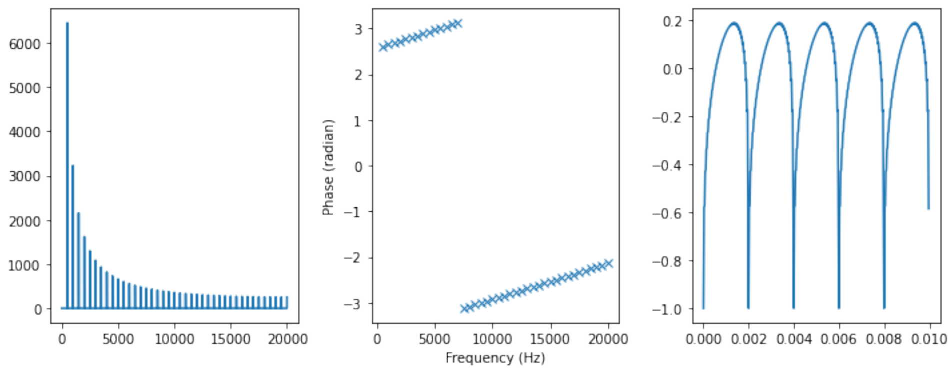

Рисунок 6.10

При повороте угла, мы опять видим, что волна изменилась, но звучит также

Посмотрим, что произайдет, если значениям углов присвоить случайные значения

```
PI2 = np.pi * 2

def random_angle(spectrum):
    res = spectrum.copy()
    angles = np.random.uniform(0, PI2, len(spectrum))
    res.hs *= np.exp(1j * angles)
    return res

spectrum4 = random_angle(spectrum)
plot_three(spectrum4)
```

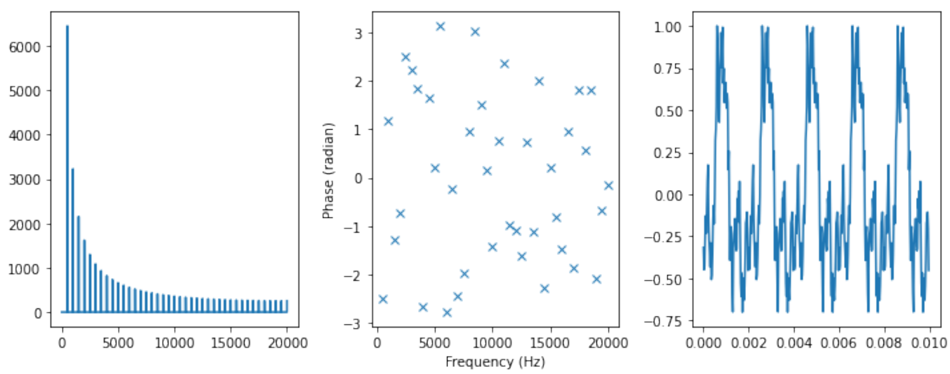

Рисунок 6.11

Опять же, форма волны сильно изменилась, но звучание остается как прежде.

Выводы: для звуков с гармонической структурой мы не замечаем измений в фазавой структуре, при условии, что гармоническая структура не меняется.

6.4. Вывод

ДКП применяется в MP3 и соответвующих форматах сжатия музыки, в JPEG, MPEG и так далее. ДКП похоже на ДПФ, использованное в спектральном анализе. Также при помощи ДКП были исследованы свойства звуков с разной структурой.

7. Дискретное преобразование Фурье

7.1. Упражнение 1

Реализовать алгоритм БПФ.

Библиотечный алгоритм fft:

```
ys = [0.4, 0.2, 0.5, -0.3]
hs = np.fft.fft(ys)
print(hs)
```

```
[ 0.8+0.j  -0.1-0.5j  1. +0.j  -0.1+0.5j]
```

Реализация DFT из учебника:

```
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
```

Увидим, что получаем такой же результат, что и от библиотечной функции (не считая погрешность)

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

```
5.777195988230628e-16
```

Напишем нерекурсивную функцию, которая разделит входной массив на подмассивы четных и нечетных элементов и использует библиотечную функцию для расчета БПФ подмассивов.

```
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)
```

Результат, как видно, остается таким же:

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

```
1.3714655826180364e-16
```

Теперь, напишем рекурсивное решение:

```
def fft(ys):
    N = len(ys)
    if N == 1:
        return ys
    He = fft(ys[::2])
```

```
    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)
```

Убедимся, что получаем такой же результат:

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

```
2.7984961504774455e-16
```

7.2. Вывод

Дискретное преобразование Фурье — это одно из преобразований Фурье, широко применяемых в алгоритмах цифровой обработки сигналов , а также в других областях, связанных с анализом частот в дискретномсигнале. Дискретное преобразование Фурье требует в качестве входа дискретную функцию. Такие функции часто создаются путём дискретизации. В качестве упражнения была написана одна из реализаций БПФ.

8. Фильтрация и свертка

8.1. Упражнение 2

Что происходит с преобразованием Фурье, если меняется std гауссовой кривой?

```
import scipy.signal

gaussian = scipy.signal.gaussian(M=32, std=2)
gaussian /= sum(gaussian)
plt.plot(gaussian)
decorate(xlabel='Index')
```

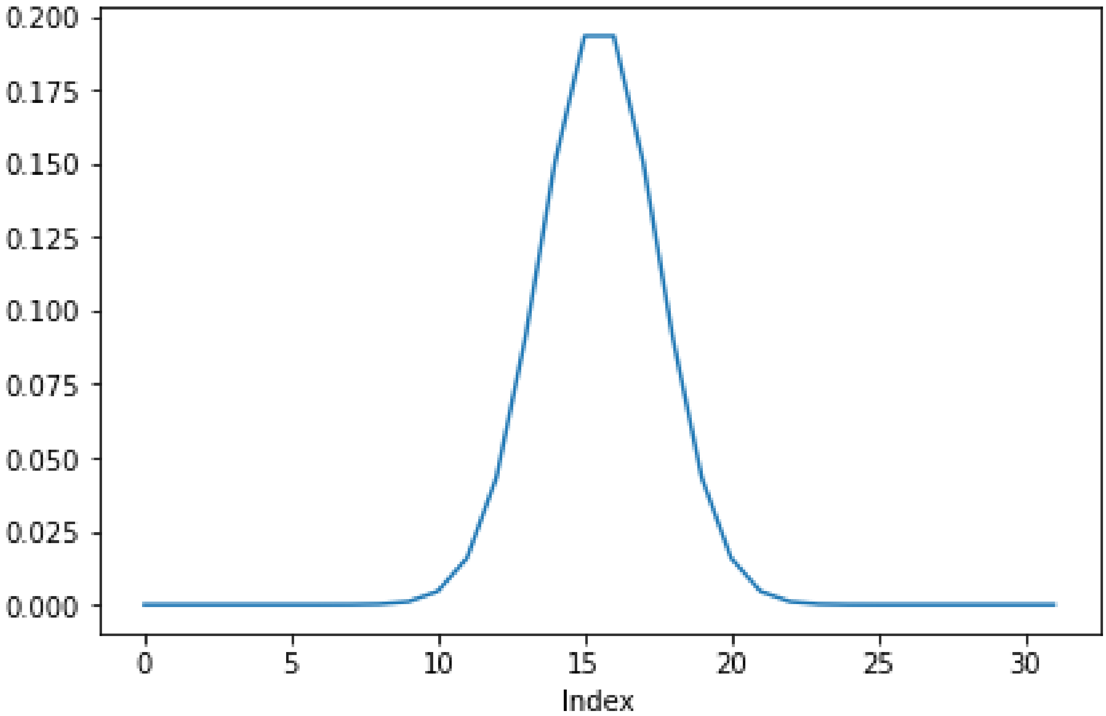

Рисунок 8.1

FFT:

```
fft_gaussian = np.fft.fft(gaussian)
plt.plot(abs(fft_gaussian))
decorate(xlabel='Frequency␣(Hz)', ylabel='Amplitude')
```

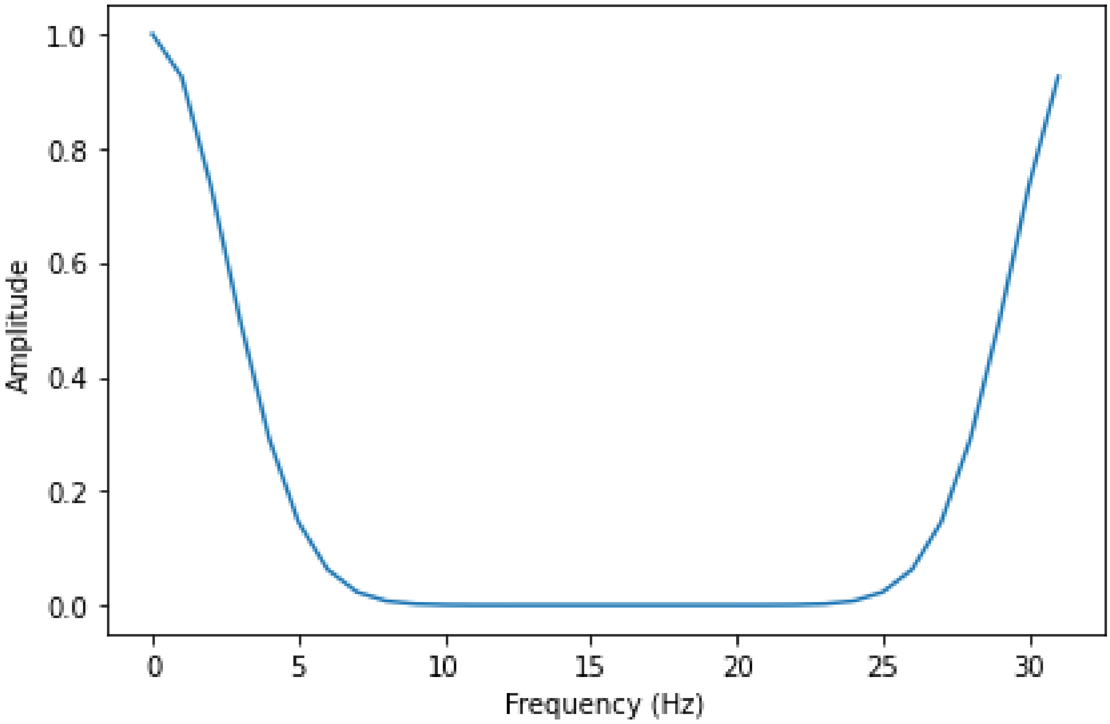

Рисунок 8.2

При сдвиге влево, видно, что это гауссова кривая.

```
N = len(gaussian)
fft_rolled = np.roll(fft_gaussian, N//2)
plt.plot(abs(fft_rolled))
decorate(xlabel='Frequency␣(Hz)', ylabel='Amplitude')
```

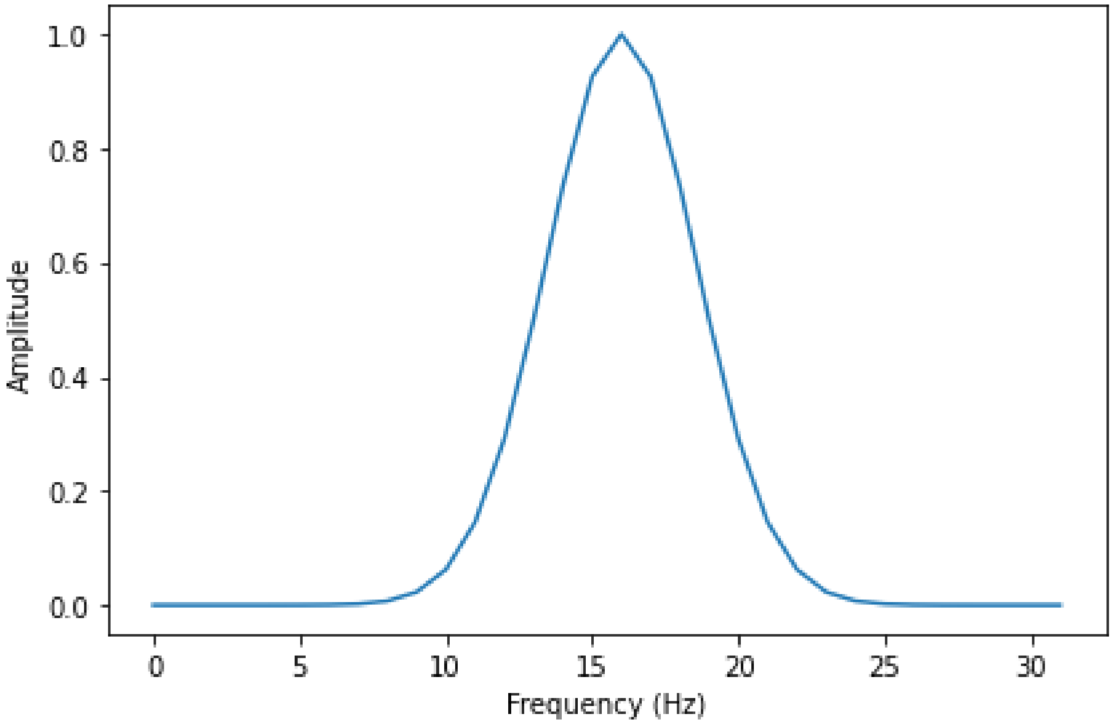

Рисунок 8.3

Построим графики гаусовой кривой и БПФ рядом, и посмотрим как на них влияет изменение std с помощью виджета

```
def plot_gaussian(std):
    M = 32
    gaussian = scipy.signal.gaussian(M=M, std=std)
    gaussian /= sum(gaussian)
    plt.subplot(1, 2, 1)
    plt.plot(gaussian)
    decorate(xlabel='Time')
    fft_gaussian = np.fft.fft(gaussian)
    fft_rolled = np.roll(fft_gaussian, M//2)
    plt.subplot(1, 2, 2)
    plt.plot(np.abs(fft_rolled))
    decorate(xlabel='Frequency')
    plt.show()

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);
```

При увеличении std гауссовой кривой, преобразование Фурье становится уже.

8.2. Упражнение 3

Поработать с разными окнами. Какое из них лучше подходит для филтра НЧ?

Создадим несколько окон.

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

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)

decorate(xlabel='Index')
```

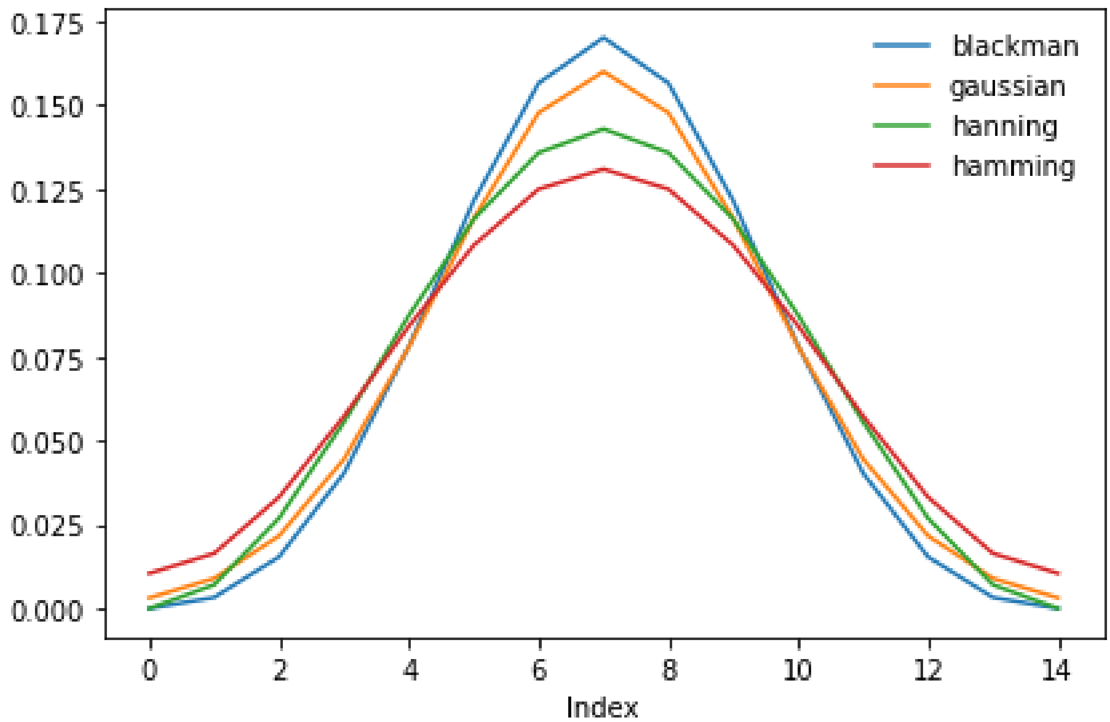

Рисунок 8.4

Посмотрим на БПФ:

```
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='Frequency (Hz)')
```

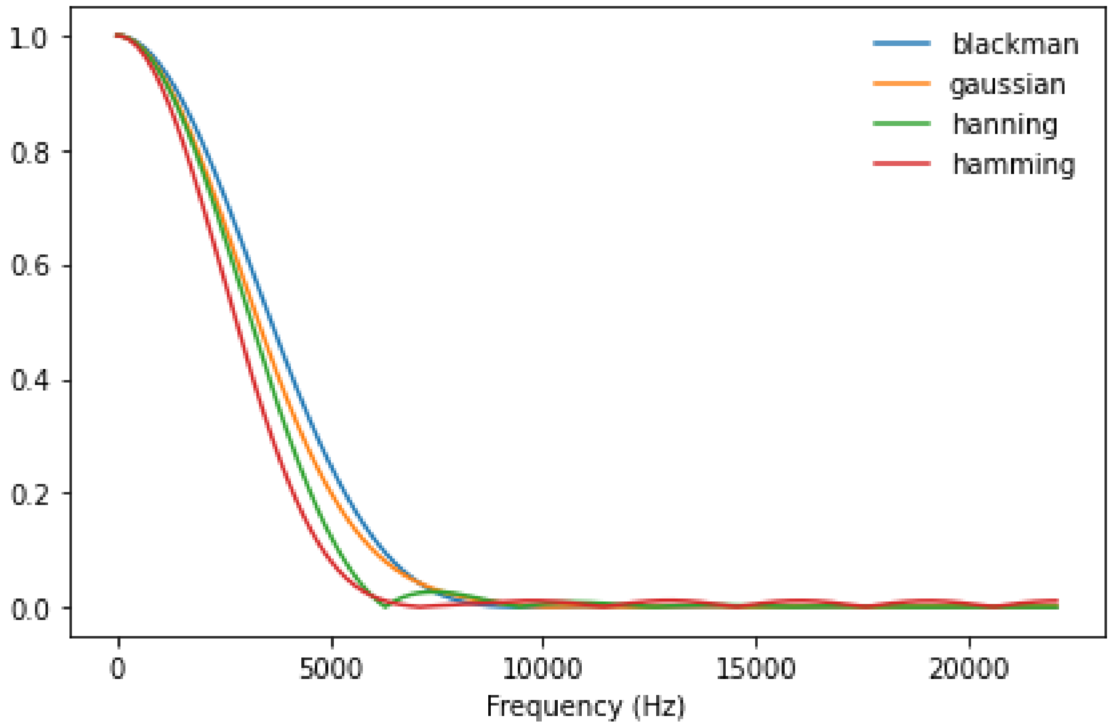

Рисунок 8.5

Вот как это выглядит в логарифмическом масштабе:

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

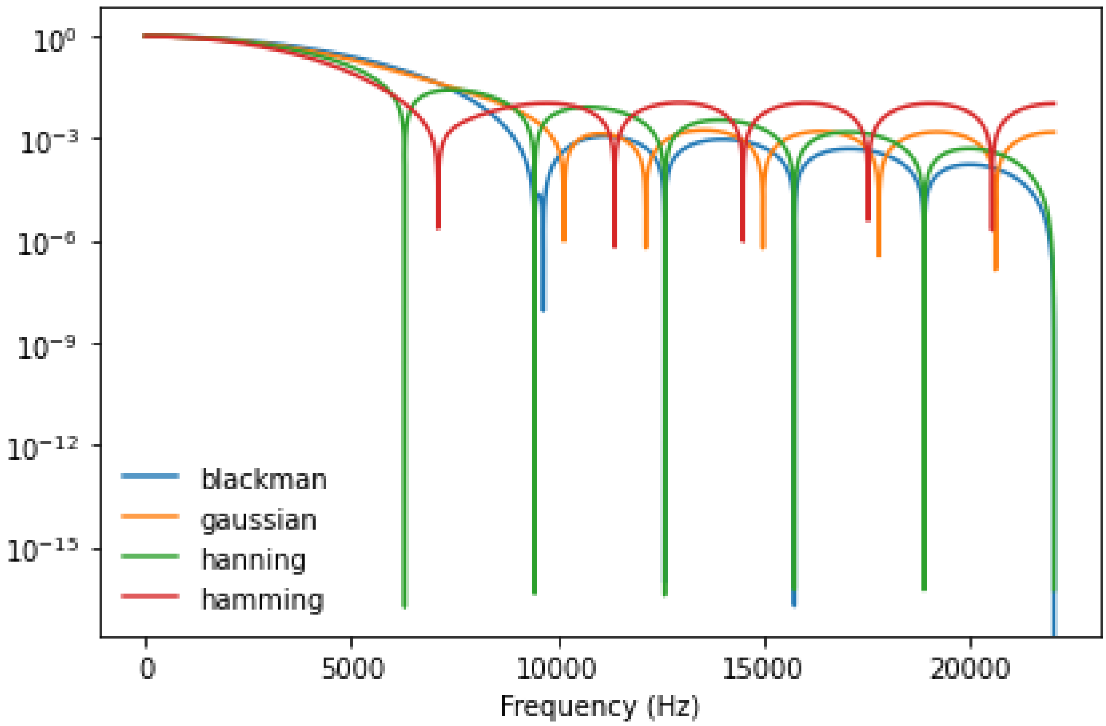

Рисунок 8.6

Из графиков видно, что окно Хэннинга спадает быстрее, чем все остальные. Окно Хэннинга подойдет лучше всех для фильтрации низких частот.

8.3. Вывод

В данной работе были рассмотрены фильтрации, свёртки, сглаживания. Сглаживание - операция удаляющая быстрые изменения сигнала для выявления общих особенностей. Свёртка - применение оконной функции к перекрывающимся сигментам сигнала. В упражнениях были исследованы различные свойства данных явлений.

9. Дифференциация и интеграция

9.1. Упражнение 1

Создайте треугольный сигнал и напечатайте его. Примените diff к сигналу и напечатайте результат. Вычислите спектр треугольного сигнала, примените differentiate и напечатайте результат. Преобразуйте спектр обратно в сигнал и напечатайте его. Есть ли различия в воздействии diff и differentiate на этот сигнал?

Треугольный сигнал:

```
wave = TriangleSignal(freq=100).make_wave(duration=0.1, framerate=44100)
wave.plot()
decorate(xlabel='Time, s')
```

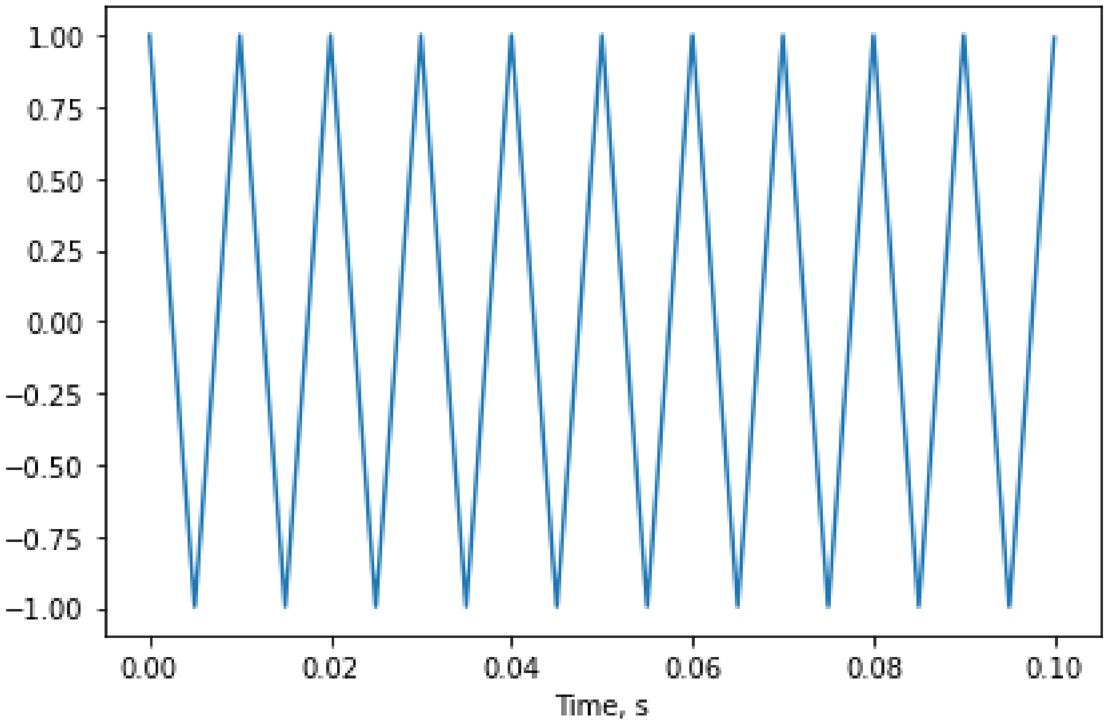

Рисунок 9.1

Применим diff к сигналу:

```
diff = wave.diff()
diff.plot()
decorate(xlabel='Tiime, s')
```

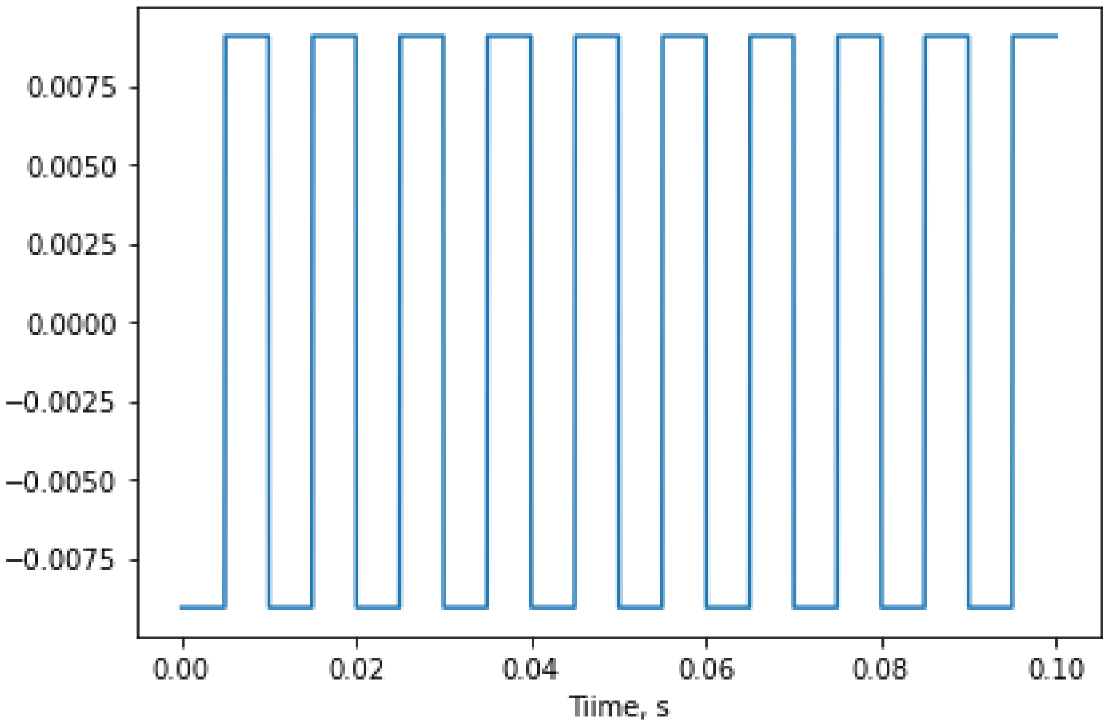

Рисунок 9.2

Получили прямоугольный сигнал.

Когда мы берем спектральную производную, мы получаем «звон» вокруг разрывов:

```
differentiate = wave.make_spectrum().differentiate().make_wave()
differentiate.plot()
decorate(xlabel='Time, s')
```

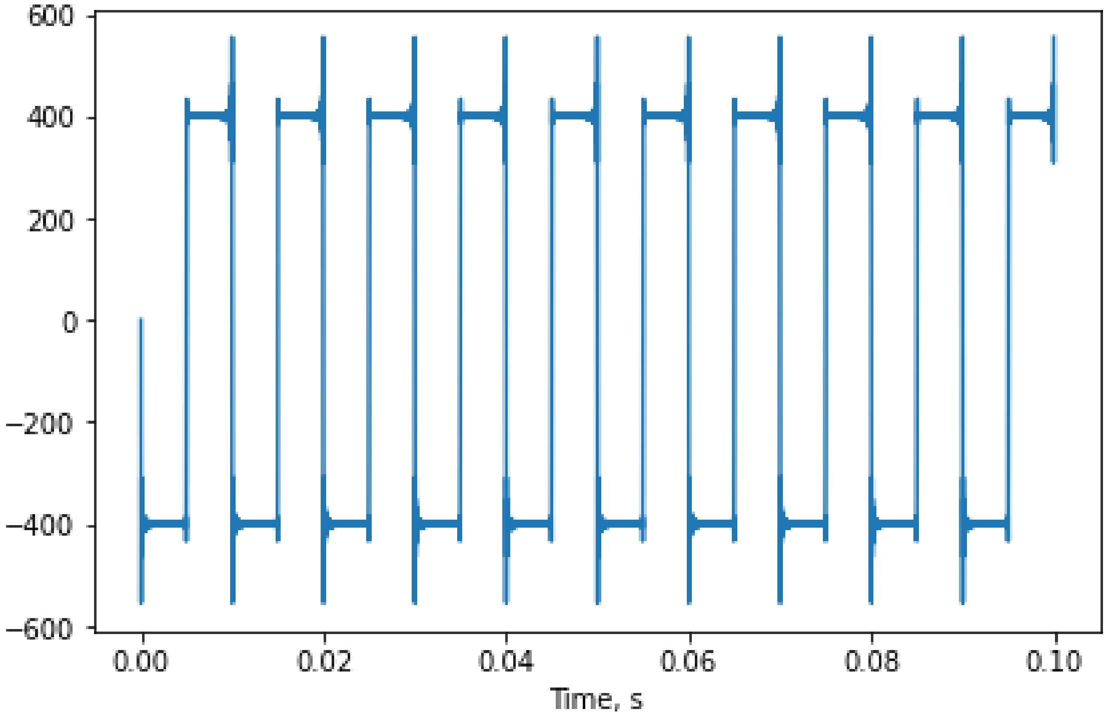

Рисунок 9.3

Это происходит потому, что производная не определена в местах разрывов.

9.2. Упражнение 2

Создайте прямоугольный сигнал и напечатайте его. Примените cumsum и напечатайте результат. Вычислите спектр прямогоульного сигнала, примените integrate и напечатайте результат. Преобразуйте спектр обратно в сигнал и напечайте его. Есть различия в воздействии cumsum и integrate на этот сигнал?

Квадратный сигнал:

```
square = SquareSignal(freq=100).make_wave(duration=0.1, framerate=44100)
square.plot()
decorate(xlabel='Time,␣s')
```

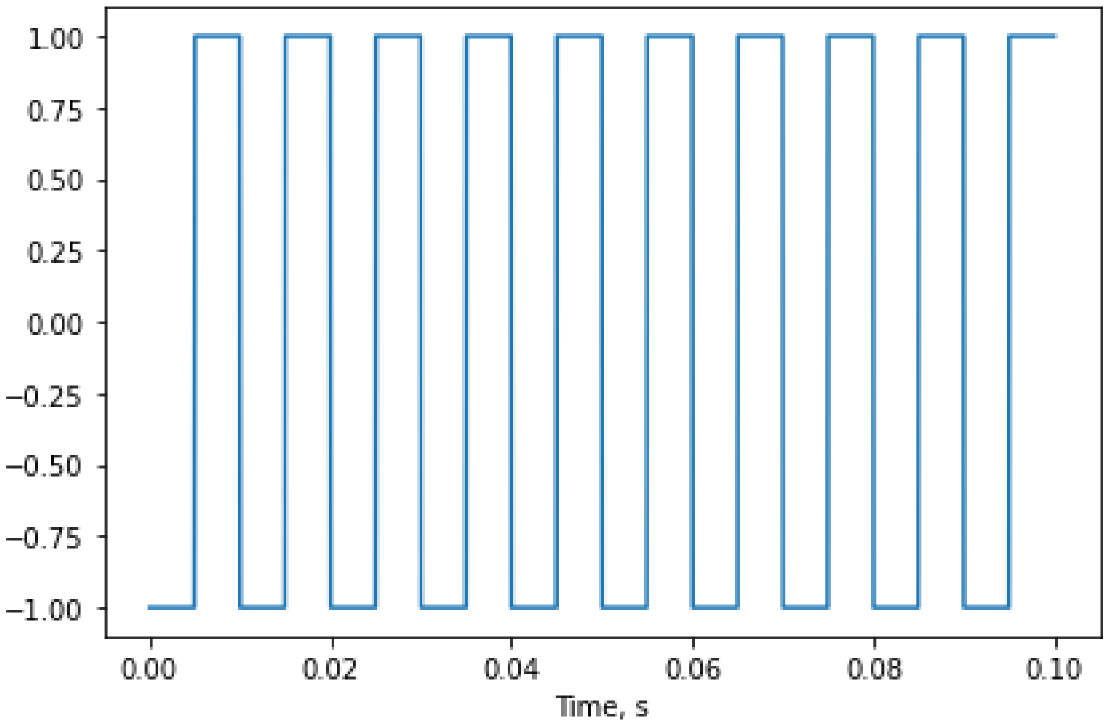

Рисунок 9.4

Применим cumsum:

```
out1 = square.cumsum()
out1.plot()
decorate(xlabel='Time, s')
```

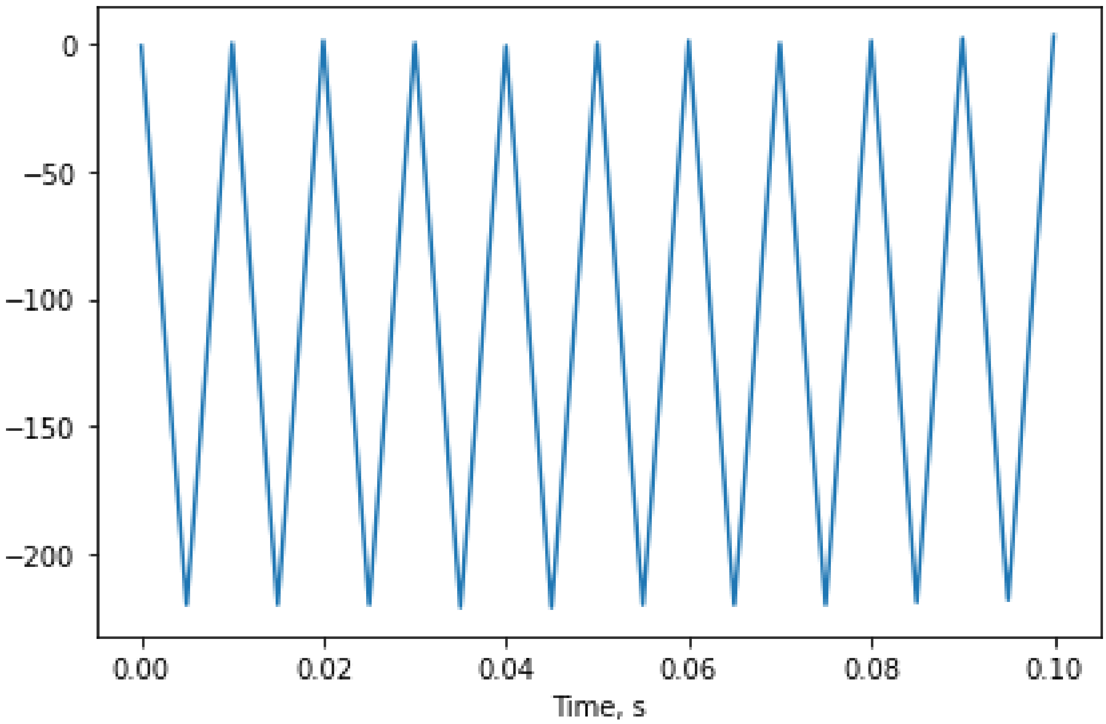

Рисунок 9.5

Нарастающая сумма квадратного сигнала - это треугольный сигнал.

```
spectrum = square.make_spectrum().integrate()
spectrum.hs[0] = 0
out2 = spectrum.make_wave()
out2.plot()
decorate(xlabel='Time, s')
```

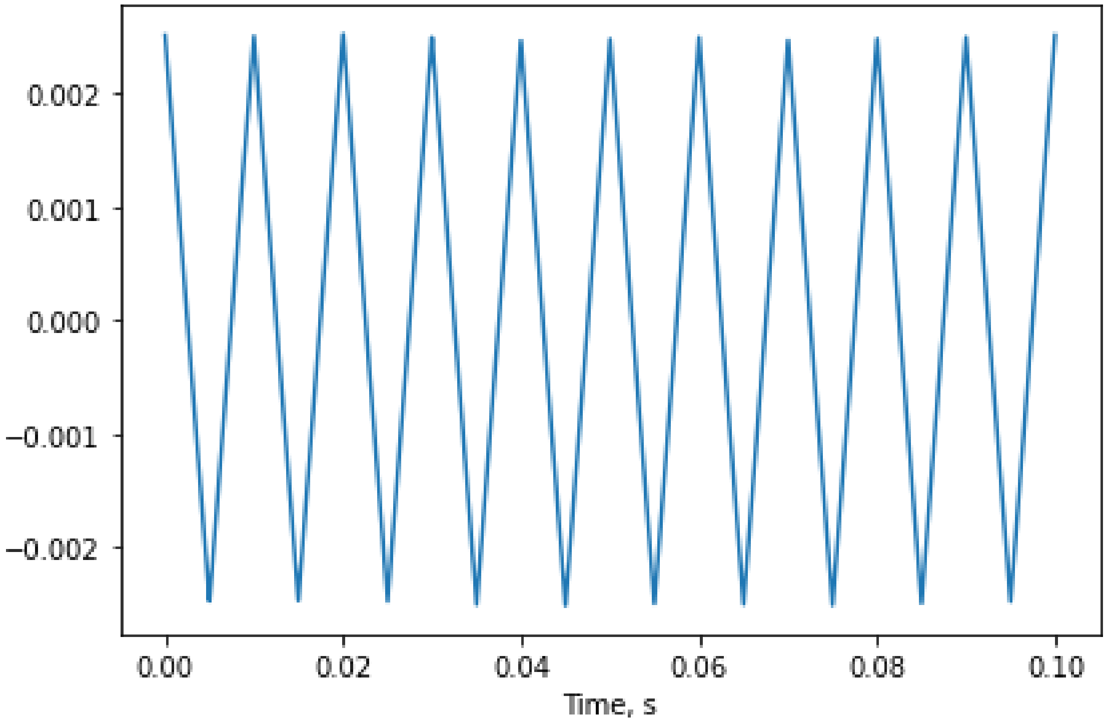

Рисунок 9.6

Интеграл спектра также является треугольным сигналом.

Если нормализировать эти две волны, можно увидеть, что между ними разницы практичеси нет.

```
out1.unbias()
out1.normalize()
out2.normalize()
out1.plot()
out2.plot()
```

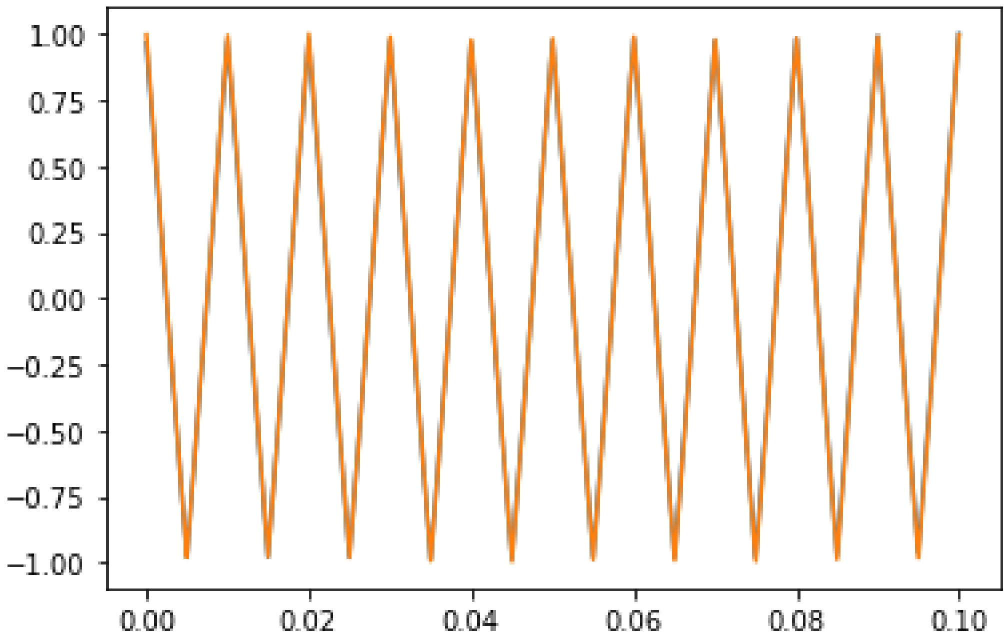

Рисунок 9.7

9.3. Упражнение 3

Создайте пилообразный сигнал, вычислите его спектр, а затем дважды примените integrate. Напечатайте результирующий сигнал и его спектр. Какова математическая форма сигнала? Почему он напоминает синусоиду?

Создадим пилообразный сигнал:

```
wave = SawtoothSignal(freq=100).make_wave(duration=0.1, framerate=44100)
wave.plot()
decorate(xlabel='Time, s')
```

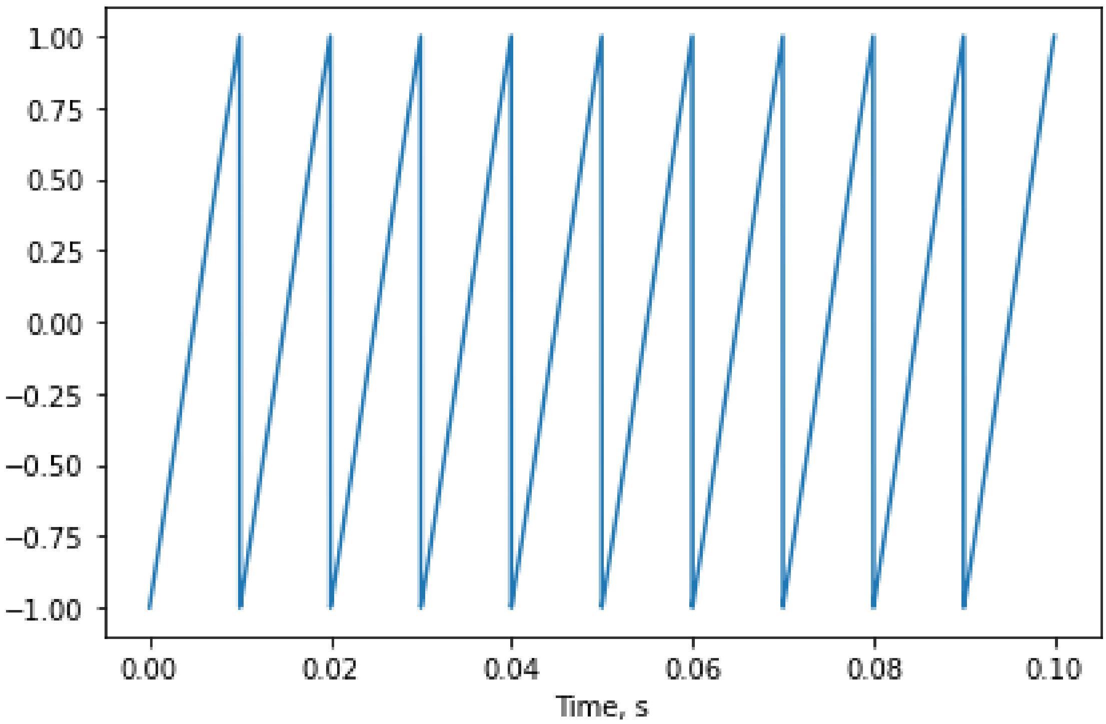

Рисунок 9.8

Дважды интегрируем спектр:

```
spectrum = wave.make_spectrum().integrate().integrate()
spectrum.hs[0] = 0

out1 = spectrum.make_wave()
out1.plot()
decorate(xlabel='Time, s')
```

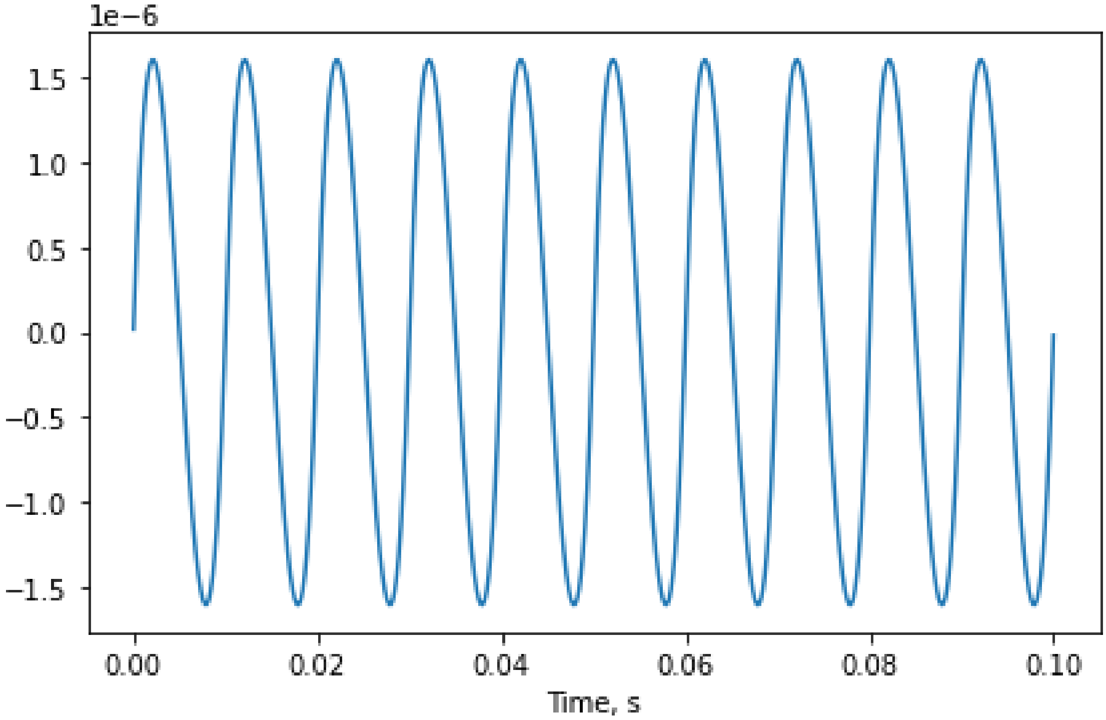

Рисунок 9.9

Результат похож на синусоиду. Причина этого в том, что интегрирование является фильтром низких частот, и мы отфильтровали почти все частоты, кроме фундаментальной.

```
out1.make_spectrum().plot(high=1000)
```

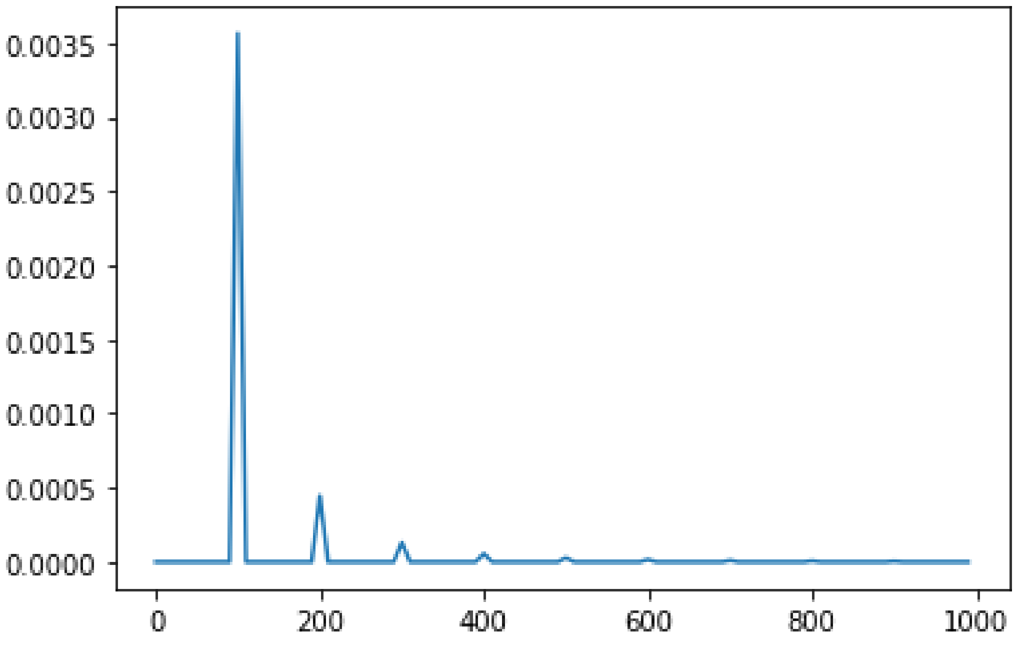

Рисунок 9.10

9.4. Упражнение 4

Создайте CubicSignal, определённый в thinkdsp. Вычислите вторую разность, дважды применив diff. Как выглядит результат? Вычислите вторую разность, дважды применив differentiate к спектру. Похожи ли результаты? Распечатйте фильтры, соответсвующие второй разнице и второй производной. Сравните их.

Создадим CubicSignal:

```
wave = CubicSignal(freq=100).make_wave(duration=0.1, framerate=44100)
wave.plot()
```

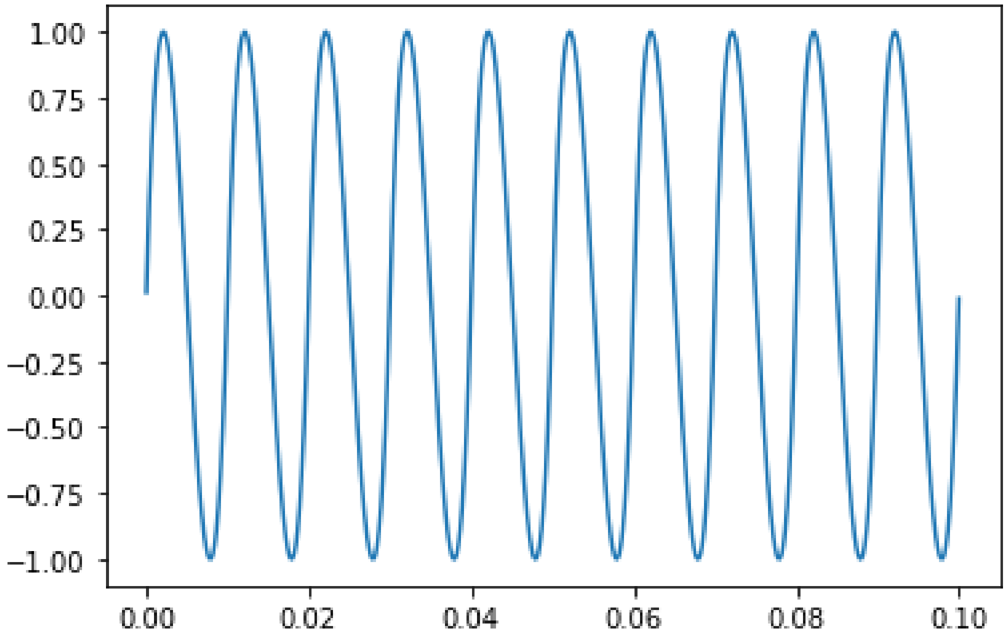

Рисунок 9.11

Вторая разность - пилообразный сигнал:

```
wave1= wave.diff().diff()
wave1.plot()
```

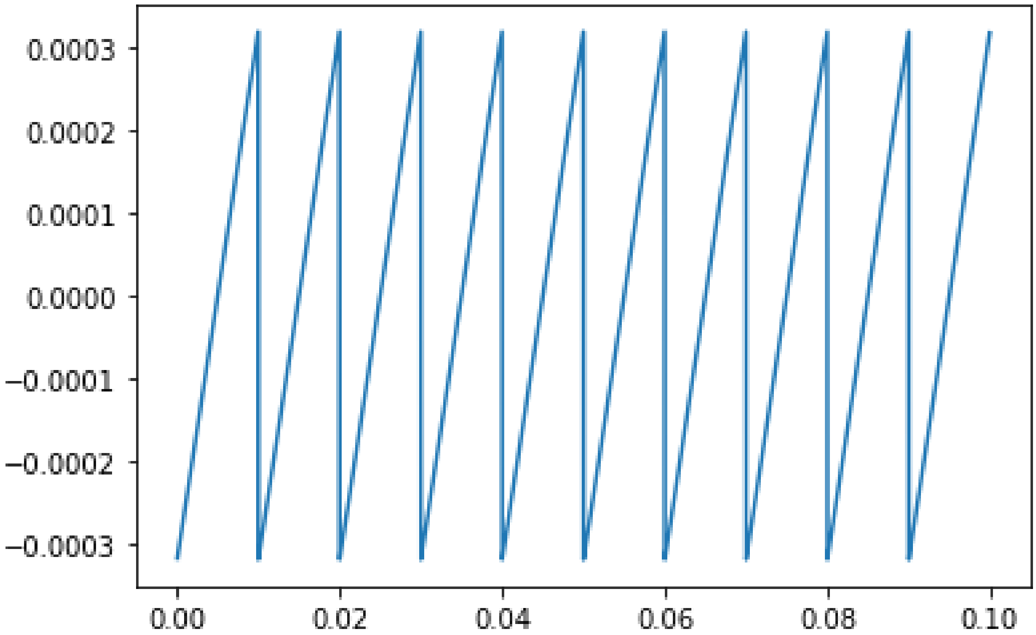

Рисунок 9.12

Вторая производная:

```
spectrum = wave.make_spectrum().differentiate().differentiate()
differentiated = spectrum.make_wave()
differentiated.plot()
```

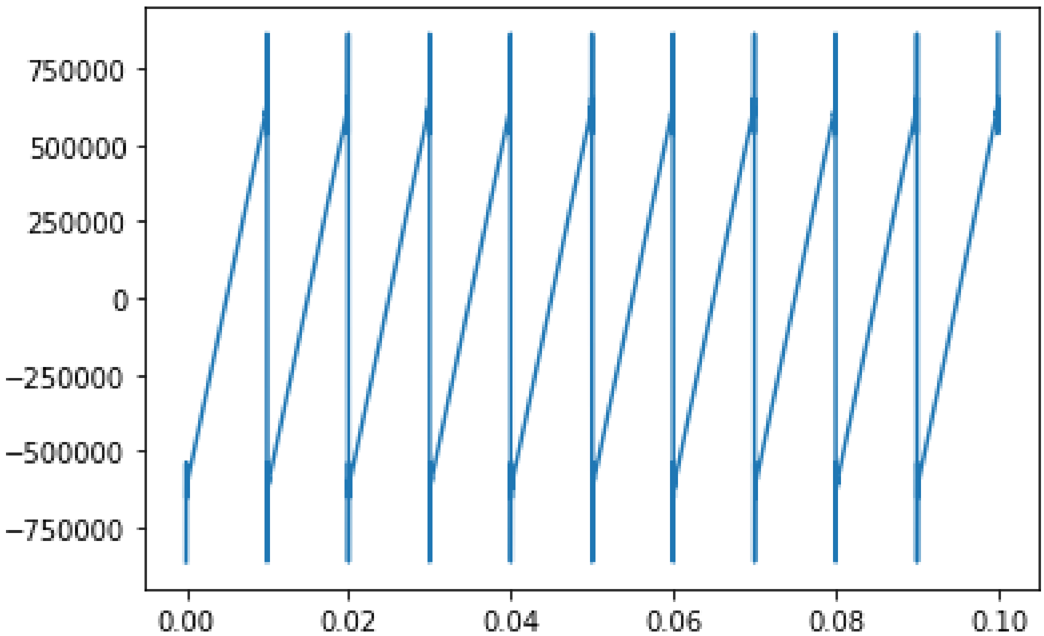

Рисунок 9.13

При двойном дифференцировании получаем пилообразный сигнал со звоном. Проблема, опять же, в том, что производная в этих точках не определена.

Фильтры, соответствующие второй разнице производной:

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

decorate(xlabel='Frequency (Hz)',
         ylabel='Amplitude ratio')
```

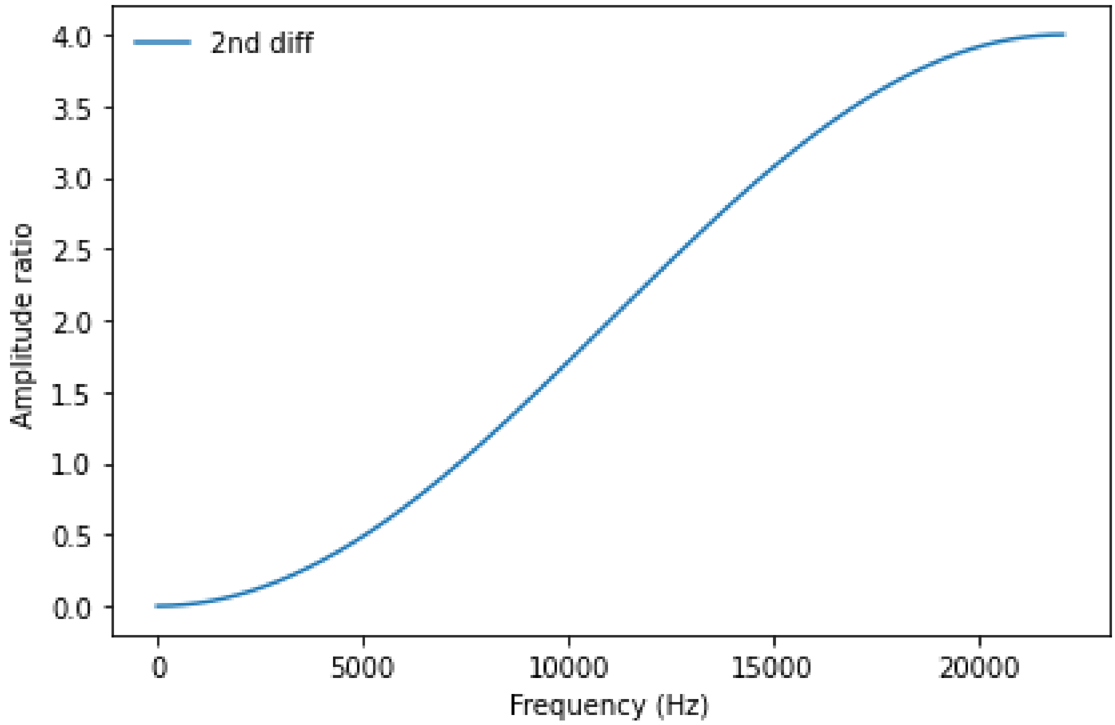

Рисунок 9.14

```
deriv_filter = wave.make_spectrum()
deriv_filter.hs = (PI2 * 1j * deriv_filter.fs)**2
deriv_filter.plot()
```

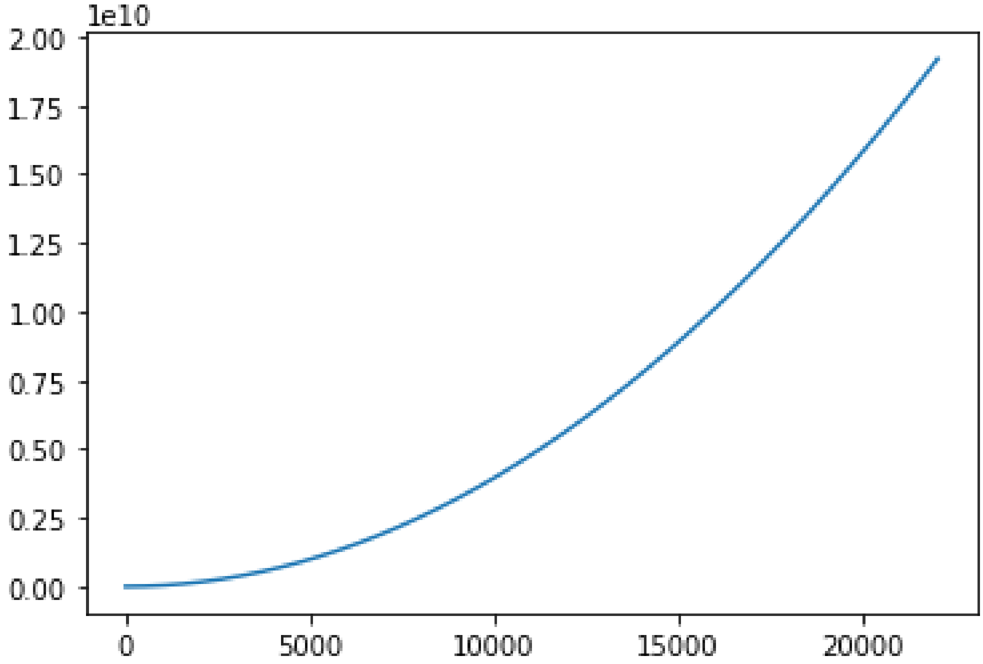

Рисунок 9.15

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

9.5. Вывод

В данной работе были рассмотрены соотношения между окнами во временной области и фильтрами в частотной. Были рассмотрены конечные разности, аппроксимирующее дифференцирование и накапливающие суммы с аппроксимирующим интегрированием.

10. Сигналы и системы

10.1. Упражнение 1

Измените пример в chap10.ipynb и убедитесь, что дополнение нулями устраняет лишнюю ноту в начале фрагмента:

Добавим 0 в начало обоих сигналов.

```
if not os.path.exists('180960__kleeb__gunshot.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/180960__k
response = read_wave('180960__kleeb__gunshot.wav')

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

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

response.normalize()
response.plot()
decorate(xlabel='Time␣(s)')
```

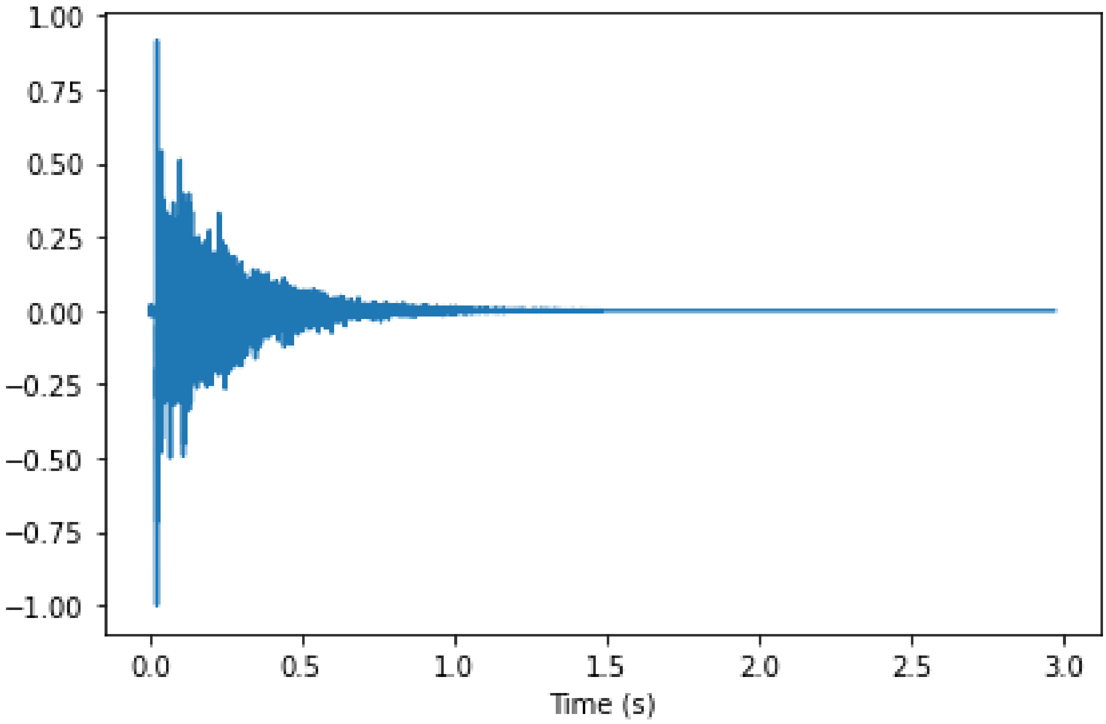

Рисунок 10.1

```
transfer = response.make_spectrum()
transfer.plot()
decorate(xlabel='Frequency␣(Hz)', ylabel='Amplitude')
```

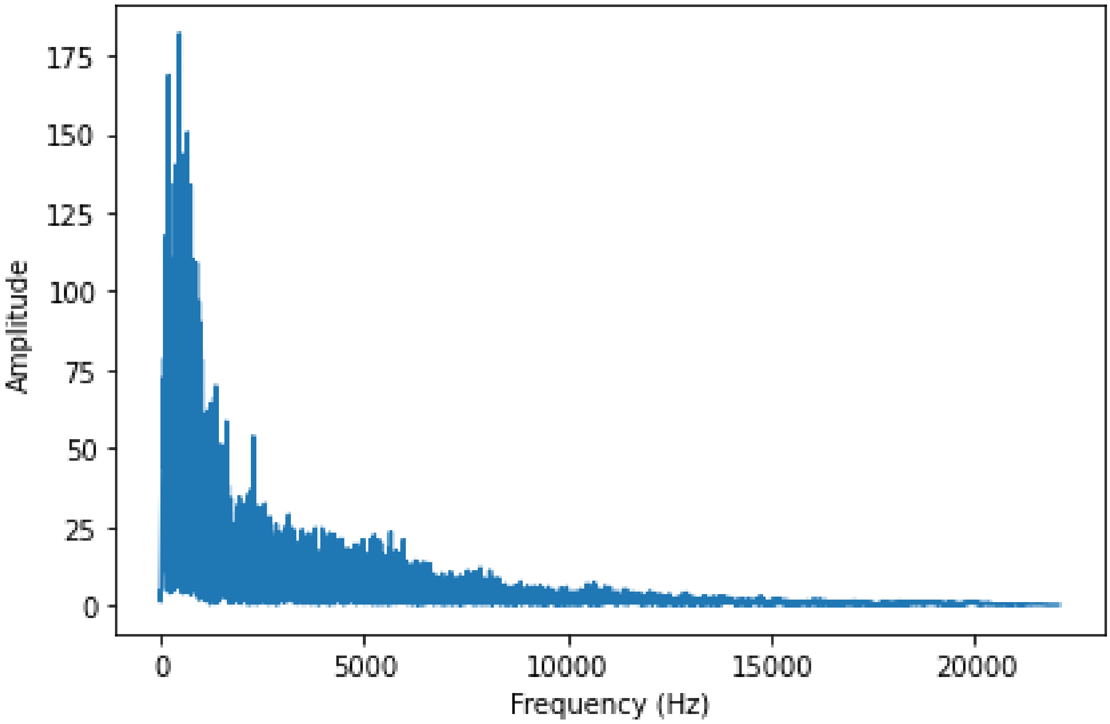

Рисунок 10.2

Теперь перейдём к самой записе:

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

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()
decorate(xlabel='Time (s)')
```

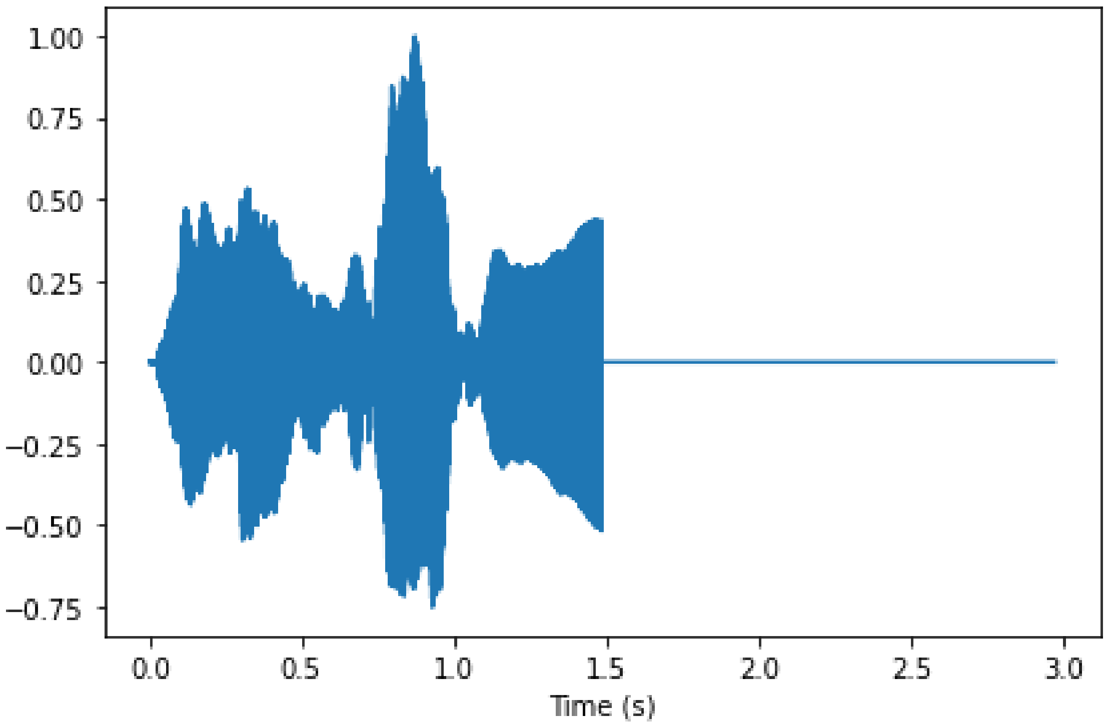

Рисунок 10.3

Произведём умножения спектров и посмотрим на результат:

```
spectrum = violin.make_spectrum()
wave = (spectrum * transfer).make_wave()
wave.normalize()
wave.plot()
```

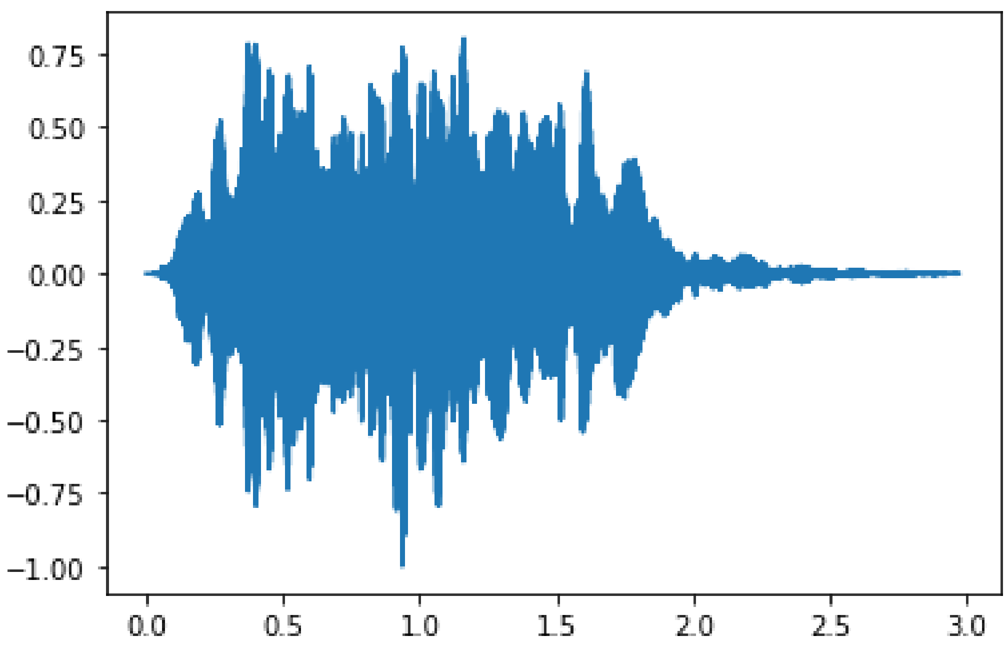

Рисунок 10.4

Лишней ноты в начале больше нет.

10.2. Упражнение 2

Смоделируйте двумя способами звучание записи в том пространстве, где была измерена импульсная харпактеристика, как свёрткой самой записи с импульсной характеристикой, так и умножением ДПФ записи на вычисленный фильтр, соотвествующий импульсной характеристики.

Загрузим звук - результат импульса в пространстве.

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

response = read_wave(filename)
start = 0
duration = 4
response = response.segment(duration=duration)
response.shift(-start)

response.normalize()
response.plot()
decorate(xlabel='Time␣(s)')
```

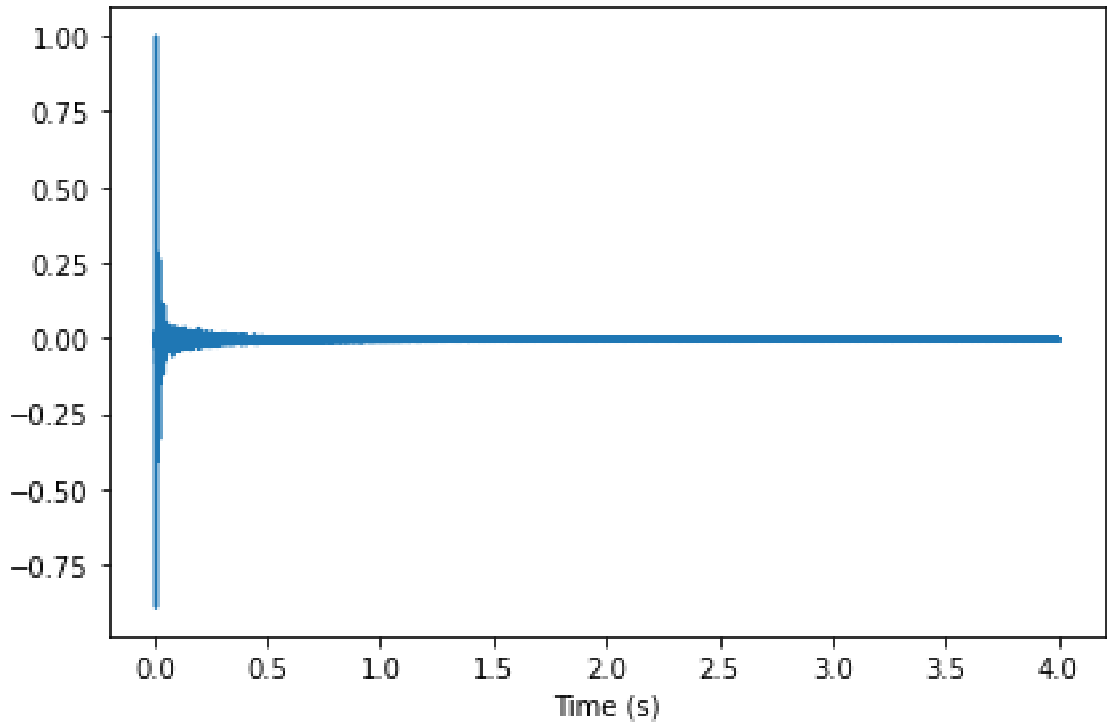

Рисунок 10.5

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

```
decorate(xlabel='Frequency␣(Hz)', ylabel='Amplitude')
```

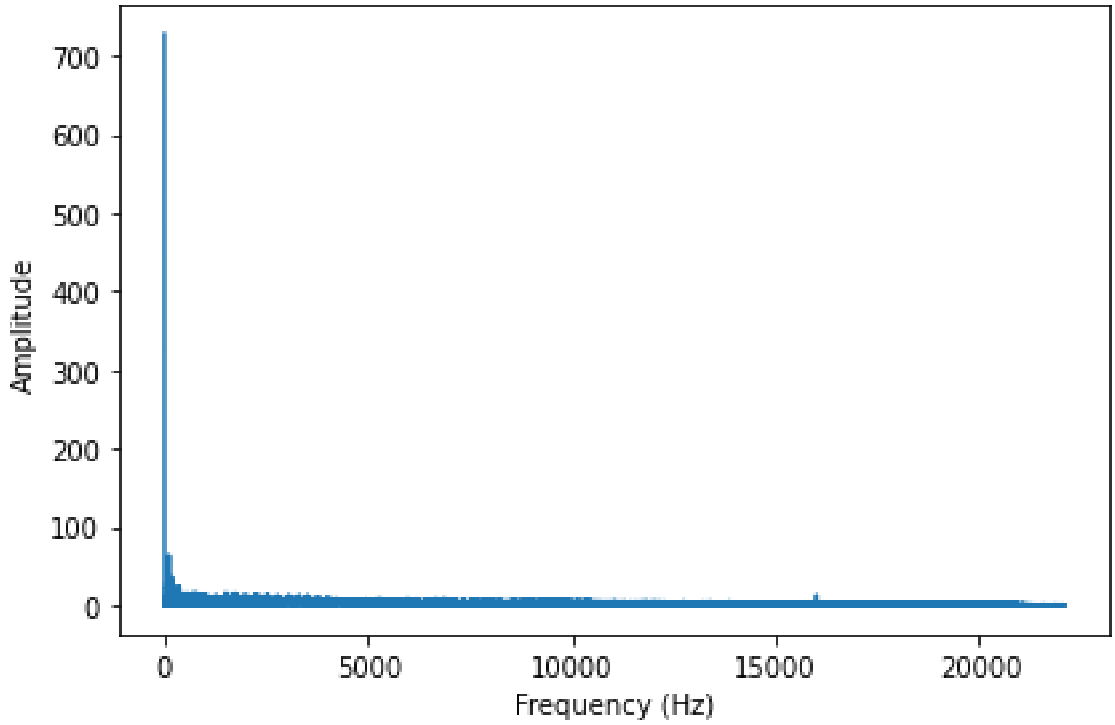

Рисунок 10.6

Промоделируем запись в пространстве.

```
if not os.path.exists('trumpet.wav'):
    !wget https://github.com/CliffBooth/telecom_labs/raw/main/
    samples/trumpet.wav

wave = read_wave('trumpet.wav')

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

wave.truncate(len(response))
wave.normalize()
wave.plot()
decorate(xlabel='Time␣(s)')
```

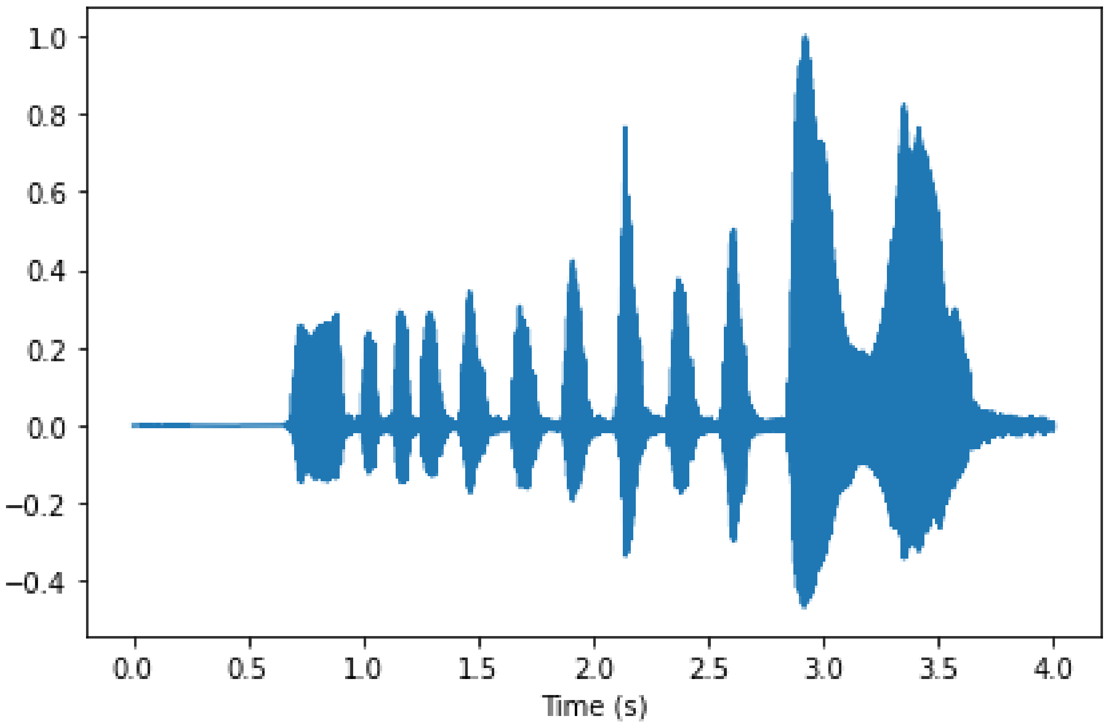

Рисунок 10.7

После трансформации:

```
spectrum = wave.make_spectrum()

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

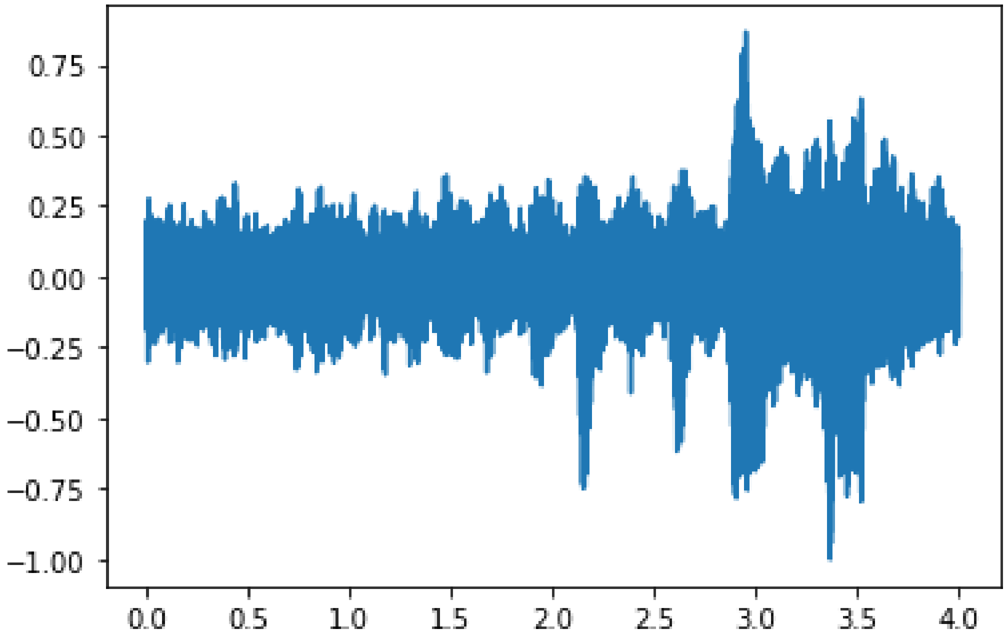

Рисунок 10.8

Добьемся такого же эффекта, используя свертку.

```
convolved2 = wave.convolve(response)
convolved2.normalize()
convolved2.make_audio()
```

10.3. Вывод

В данной работе были рассмотренны основные позиции из теории сигналов и систем. Как примеры - музыкальная акустика. При описании линейных стационарных систем используется теорема о свёртке.

11. Модуляция и сэмплирование

11.1. Упражнение 3

При взятии выборок из сигнала при слишком низкой чистоте кадров составляющие, большие частоты заворота дадут биения. В таком случаее эти компоненты не отфильтруешь, посколько они неотличимы от более низких частот. Полезно отфильтровать эти частоты до выборки: фильтр НЧ, используемый для этой цели, называется фильтром сглаживания. Вернитесь к примеру "Соло на барабане", примените фильтр НЧ до выборки, а затем, опять с помощью фильтра НЧ, удалите спектральные копии, вызванные выборкой. Результат должен быть идентицент отфильтрованному сигналу.

Загрузим "соло на барабане"

```
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

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

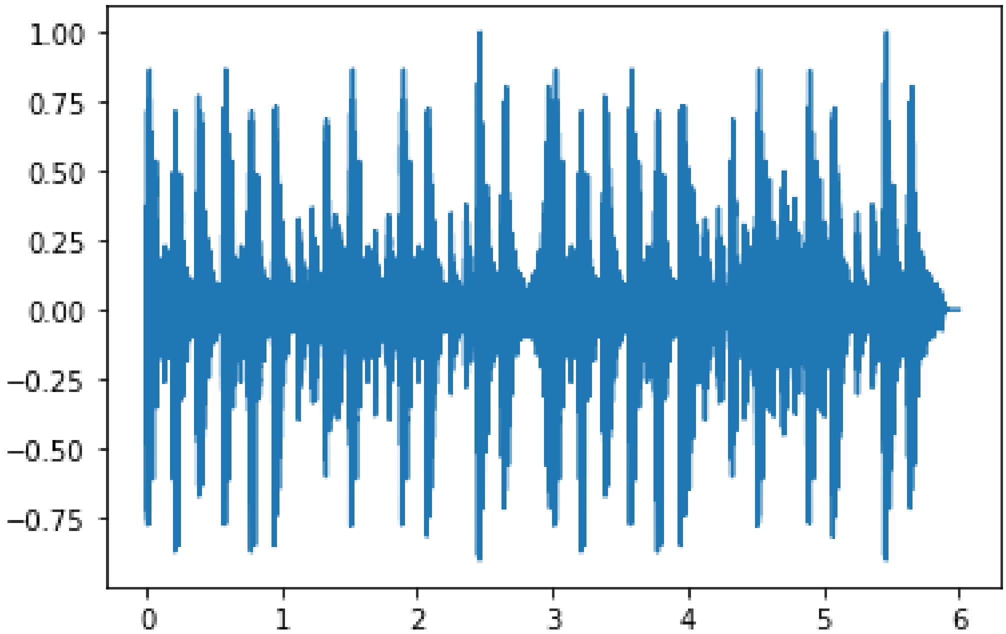

Рисунок 11.1

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

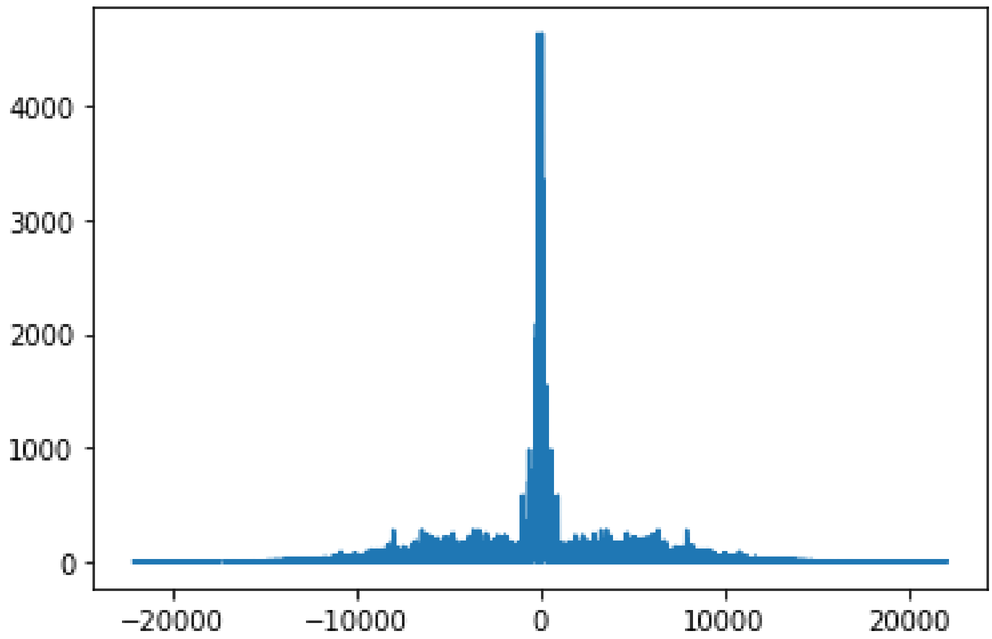

Рисунок 11.2

Уменьшим частоту сэмплирования в 5 раз.

```
factor = 5
framerate = wave.framerate / factor
cutoff = framerate / 2 - 1

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

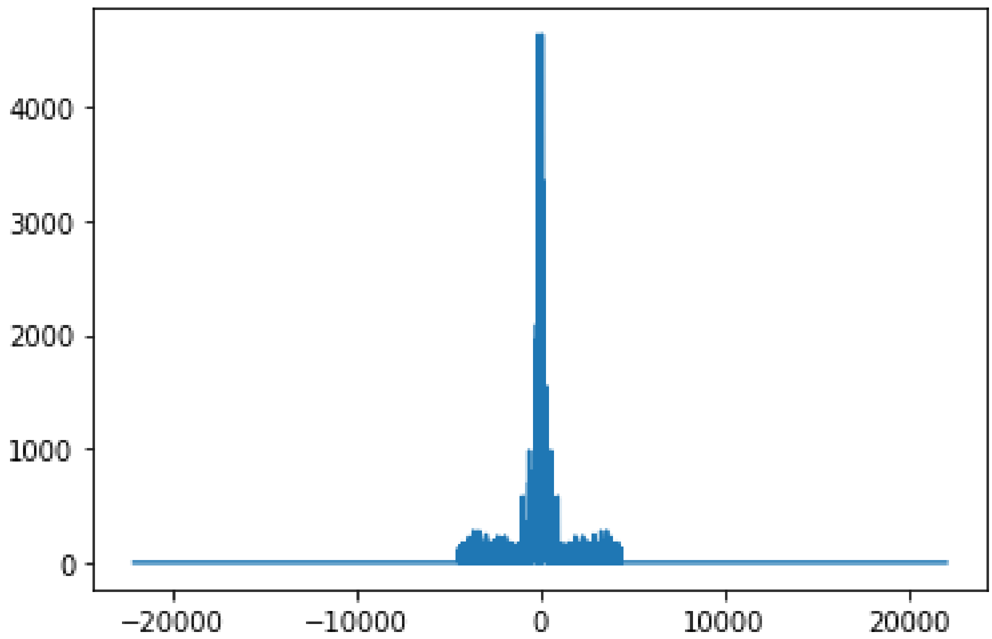

Рисунок 11.3

Функция, имитирующая процесс сэмплирования

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

Запись теперь звучит по другому, и на спектральной диаграмме появились спектральные копии.

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

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

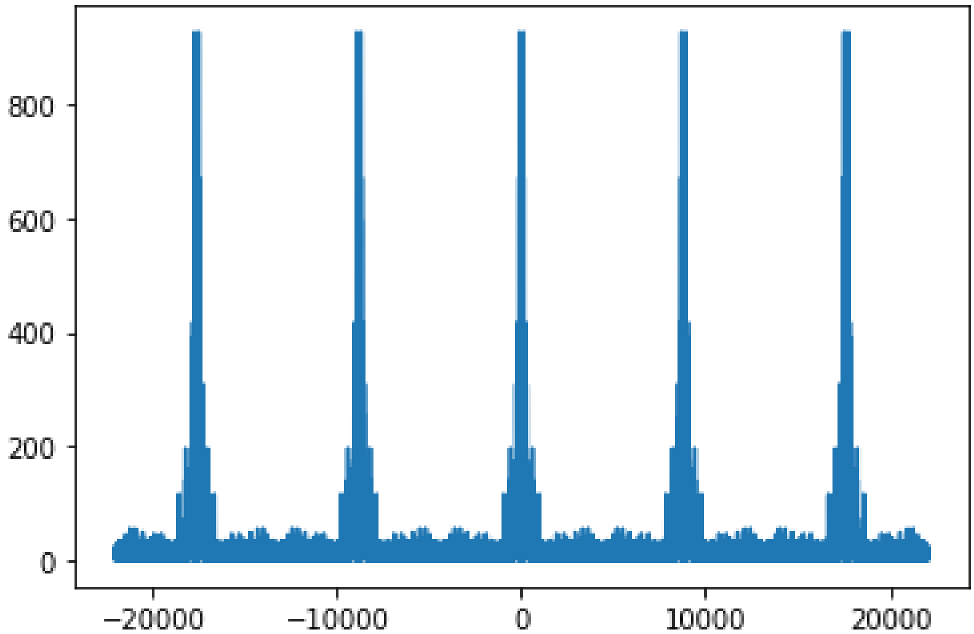

Рисунок 11.4

Избавимся от спектральных копий:

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

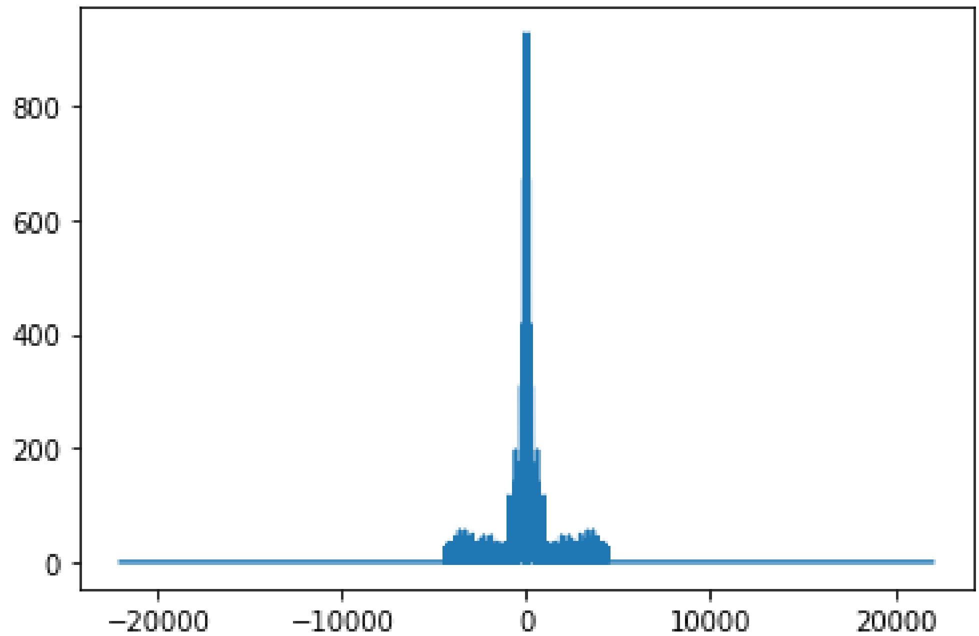

Рисунок 11.5

Увеличим амплитуду в 5 раз.

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

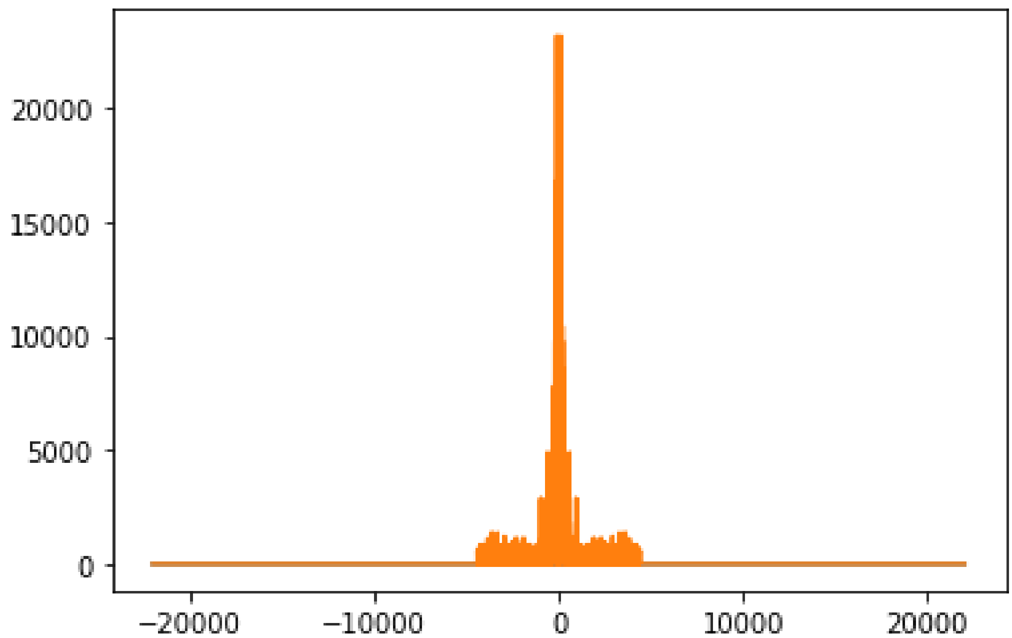

Рисунок 11.6

Разницы между волнами почти нет:

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

```
5.6681505884047e-16
```

11.2. Вывод

В данной работе были проверены свойства выборок и прояснены биения и заворот частот.

12. FSK

12.1. Теоритическая основа

Частотная манипуляция (Frequency-shift keying) — это схема частотной модуляции, в которой цифровая информация передается посредством дискретных изменений частоты несущего сигнала. Эта технология используется для систем связи, таких как телеметрия, радиозонды метеозондов, идентификация вызывающего абонента, устройства открывания гаражных ворот и низкочастотная радиопередача в диапазонах VLF и ELF.

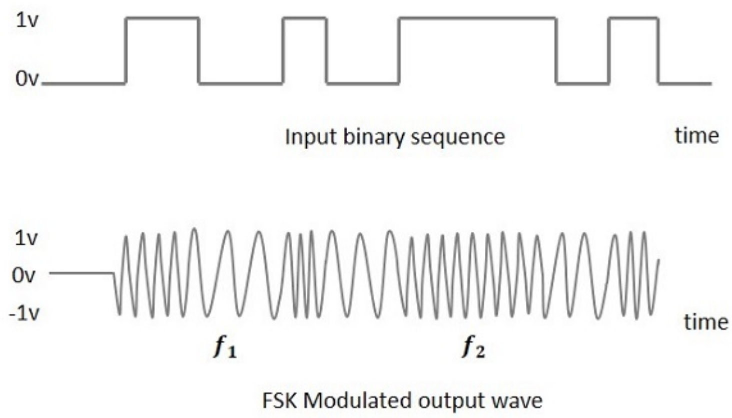

Рисунок 12.1. Пример FSK

12.2. GNU Radio

GNU Radio [1] — это бесплатный набор инструментов для разработки программного обеспечения, который предоставляет блоки обработки сигналов для реализации программно-определяемых радиостанций и систем обработки сигналов.

Для изучения процесса FSK в GNU Radio необходимо построить следующую блок схему 12.2:

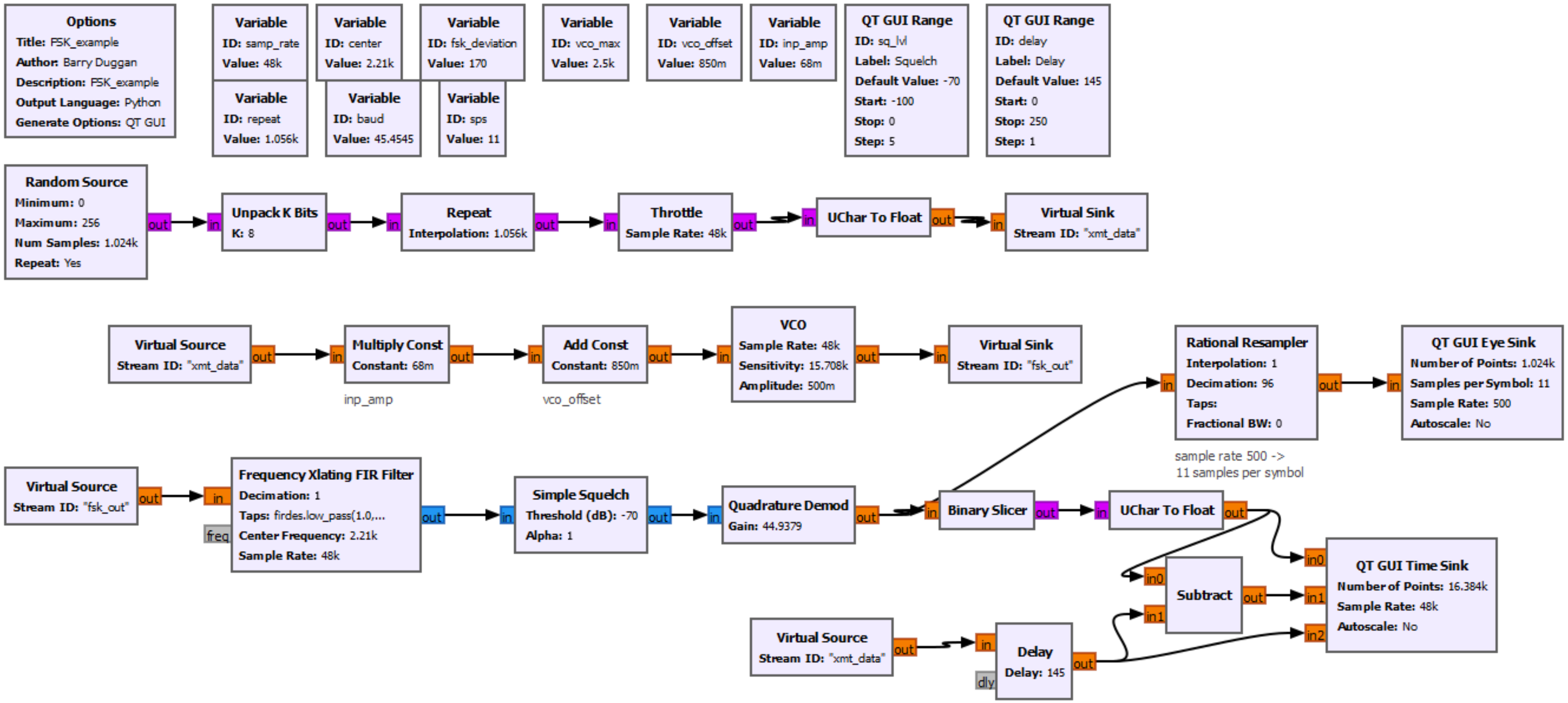

Рисунок 12.2. Схема FSK

Случайный источник генерирует значения байтов от 0 до 255. Байт распоковывается в каждый бит становится байтом со значащим младшим разрядом. Поскольку аппаратное обеспечение не задействовано, для ограничения потока через систему используется блок дроссельной заслонки (Throttle).

Приёмник при помощи фильтра смещает принимаемый сигнал так, чтобы он был сосредоточен вокруг центральной частоты - между частотами Mark и Space. Шумоподавитель добавлен для реального приёма сигналов. Блок Quadrature Demod производит сигнал, который является положительным для входных частот выше нуля и отрицательным для частот ниже нуля. Когда данные доходят до Binary Slicer, то на выходе получает биты, это и есть наша полученная информация.

Описание основных блоков на схеме 12.2:

- Variable - блок адресующий в уникальной переменной. При помощи ID можно передавать информацию через другие блоки.
- Random Source - генератор случайных чисел.
- Unpack K bits - преобразуем байт с k релевантными битами в k выходных байтов по одному биту в каждом.
- Throttle - дросселировать поток таким образом, чтобы средняя скорость не превышала удельную скорость.
- Virtual Sink - сохраняет поток в вектор.
- Virtual Source - источник данных, который передаёт элементы на основе входного вектора.
- VCO - генератор, управляемый напряжением. Создает синусойду на основе входной ампилтуды.
- Frequency Xlating FIR Filter - этот блок выполняет преобразование частоты сигнала, 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.
- Simple Squelch - простой блок шумоподавления на основе средней мощности сигнала и порога в дБ.
- Quadrature Demod - квадратурная модуляция.
- Binary Slicer - слайсы от значения с плавающей запятой, производя 1-битный вывод. Положительный ввод производит двоичную 1, а отрицательный ввод производит двоичный ноль.
- QT GUI Sink - выводы необходимой infomрации в графическом интерфейсе.

После запуска моделирования был получен следующий результат:

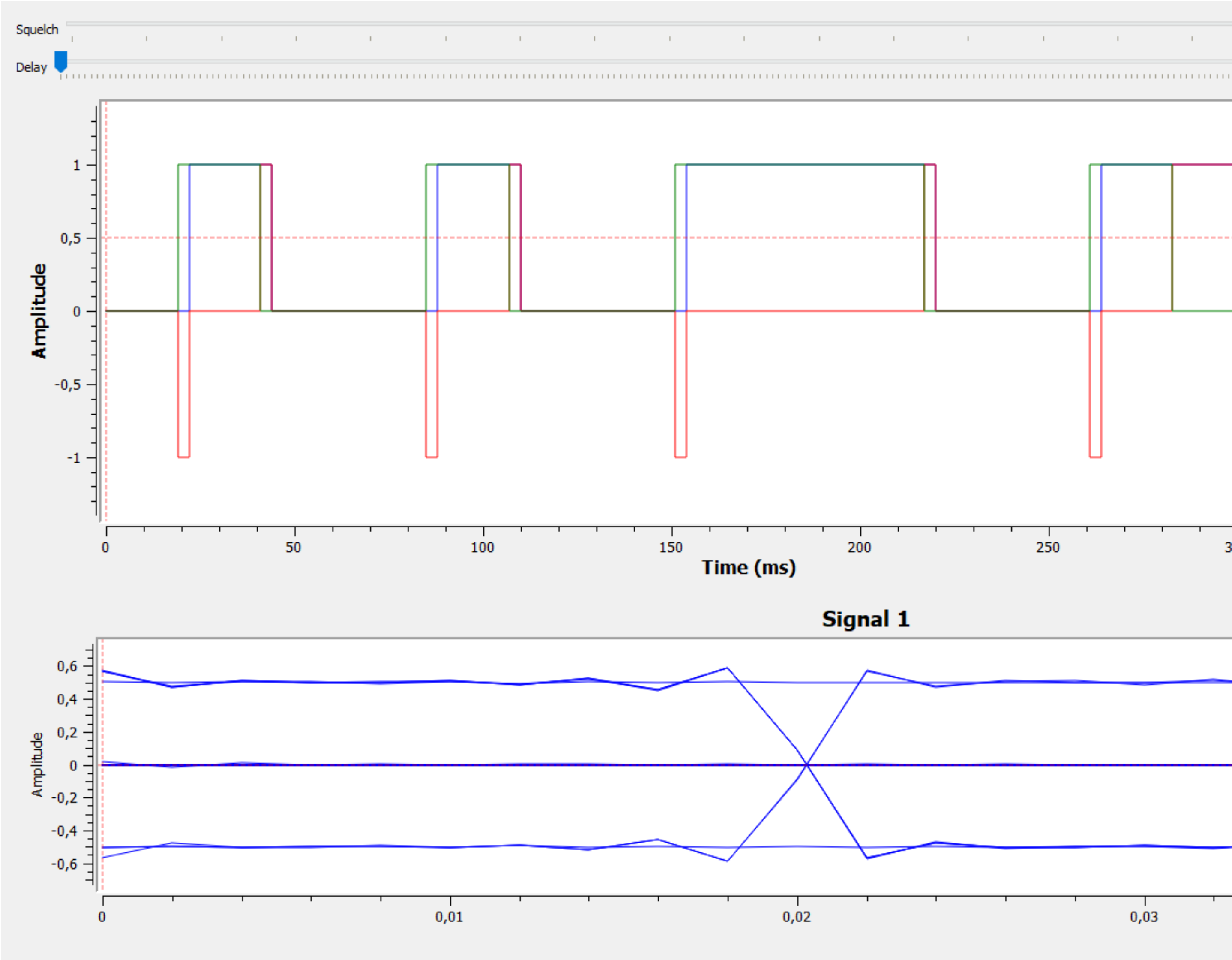

Рисунок 12.3. Результат тестирования

На графике есть 3 сигнала. Синий сигнал - данные полученные приёмником. Зелёный сигнал - данные переданные передатчиком. Красный сигнал - разница между двумя предыдущими.

Видно, что полученный сигнал отстает на некоторое количество битов, потому что цепочка передатчика и приемника имеет много блоков и фильтров, которые задерживают сигнал. Чтобы компенсировать это, мы должны задержать передаваемые биты на ту же величину, используя блок Delay.

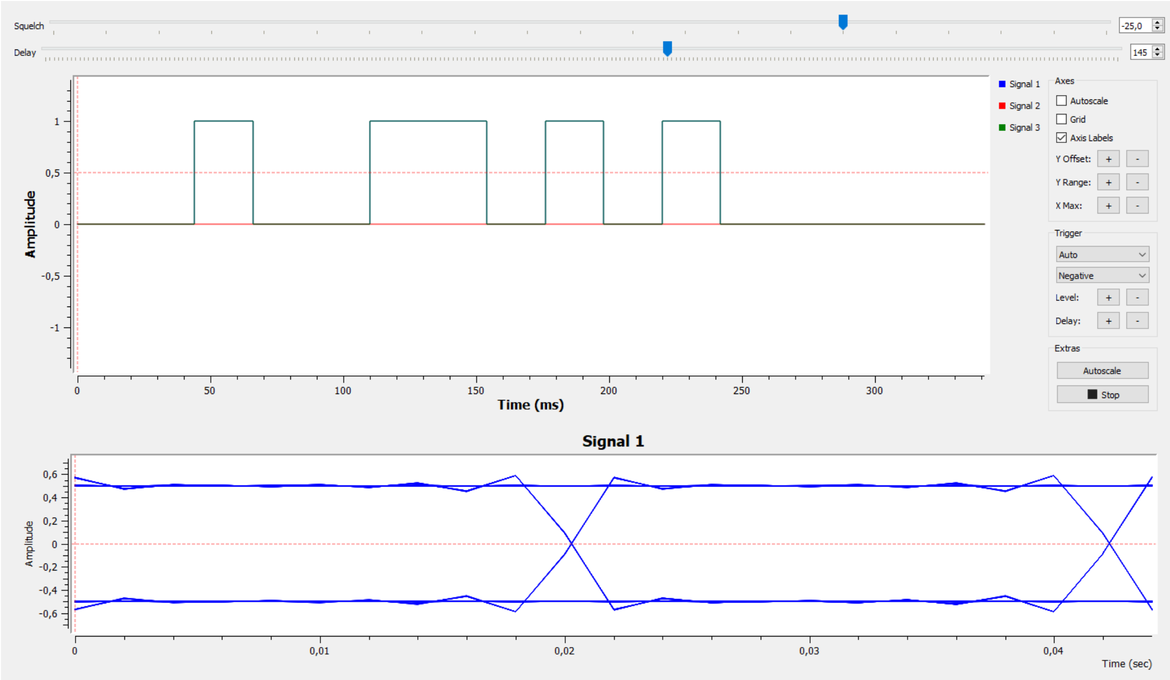

Рисунок 12.4. Результат тестирования с правильной задержкой

Правильная задержка 145.

12.3. Вывод

В данной работе был изучен новый способ модуляции. Его особенность в том, что информация передаётся при помощи изменений частоты, а не амплитуды, что делает его довольно шумоустойчивым.

При помощи среды Radio GNU была создана модель схемы и проверена на корректность.

Перечень использованных источников

1. GNU Radio official page. — URL: https://www.gnuradio.org/.