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

Телекоммуникационные технологии

Отчёт по лабораторным работам

Работу выполнил:
Е. К. Борисов
Группа:
3530901/90201
Преподаватель:
Н. В. Богач

Санкт-Петербург
2022

Содержание

1. Звуки и сигналы

1.1. Упражнение 1

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?

Был выбран звук пианино, загружаем его, прослушиваем, затем вырезаем полусекундный фрагмент выбранного звука и строим wave график.

```
if not os.path.exists('440931__xhale303__piano-loop-1.wav'):
    !wget https://github.com/Eugenepolyt/ThinkDSP/raw/master/440931__xhale3
wave = read_wave('440931__xhale303__piano-loop-1.wav')
wave.make_audio()
segment = wave.segment(start=1.5, duration=0.5)
segment.make_audio()
segment.plot()
```

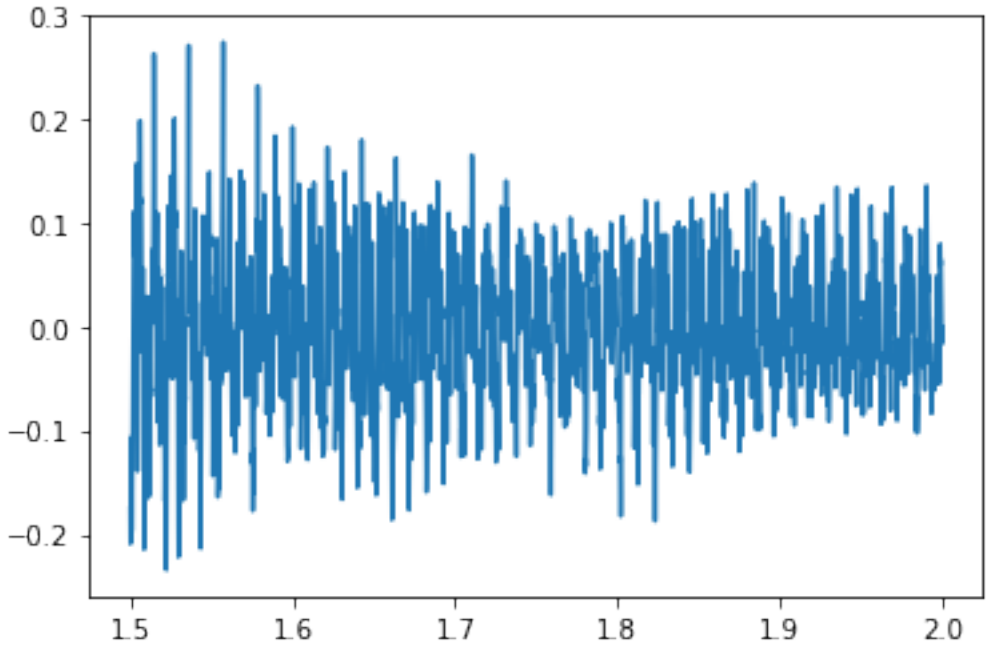

Рисунок 1.1. График выделенного сегмента

Вычислим спектр выделенного сегмента и построим график.

```
spectrum = segment.make_spectrum()
spectrum.plot(high=3000)
decorate(xlabel='Frequency')
```

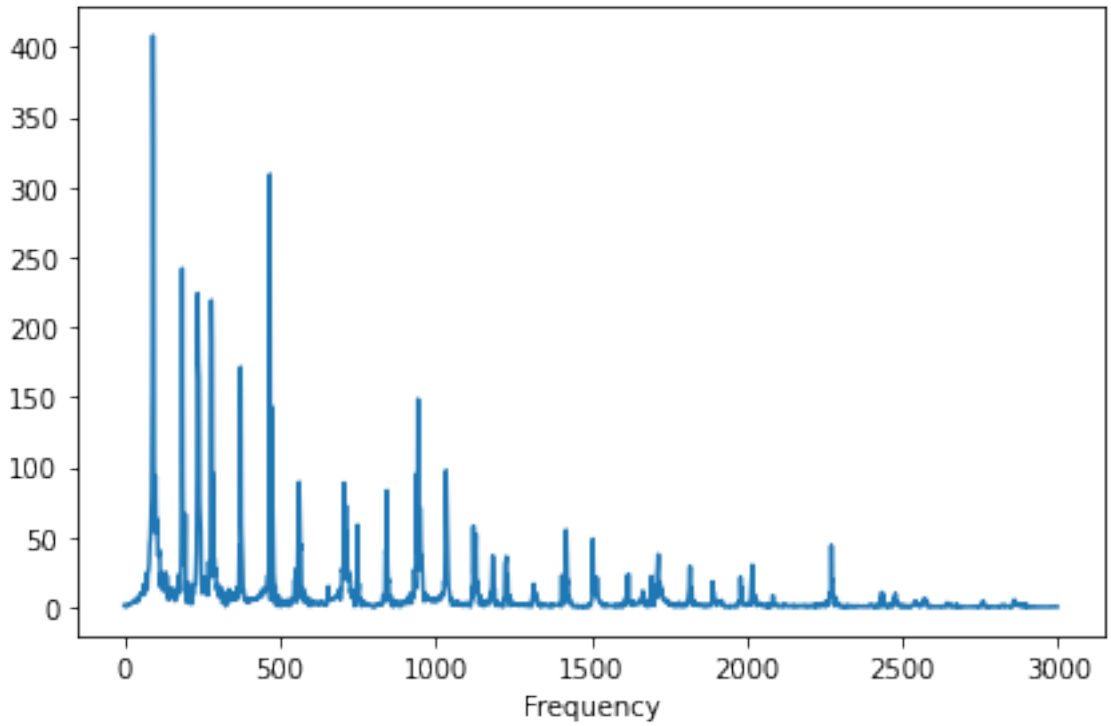

Рисунок 1.2. Спектр звука фрагмента

В данном примере основная частота является доминирующей, воспринимаемая высота звука сильно зависит от основной частоты. Используем функции для фильтрации для данного спектра.

Уберем частоты выше 1500.

```
spectrum.low_pass(1500)
spectrum.plot(high=3000)
decorate(xlabel='Frequency')
```

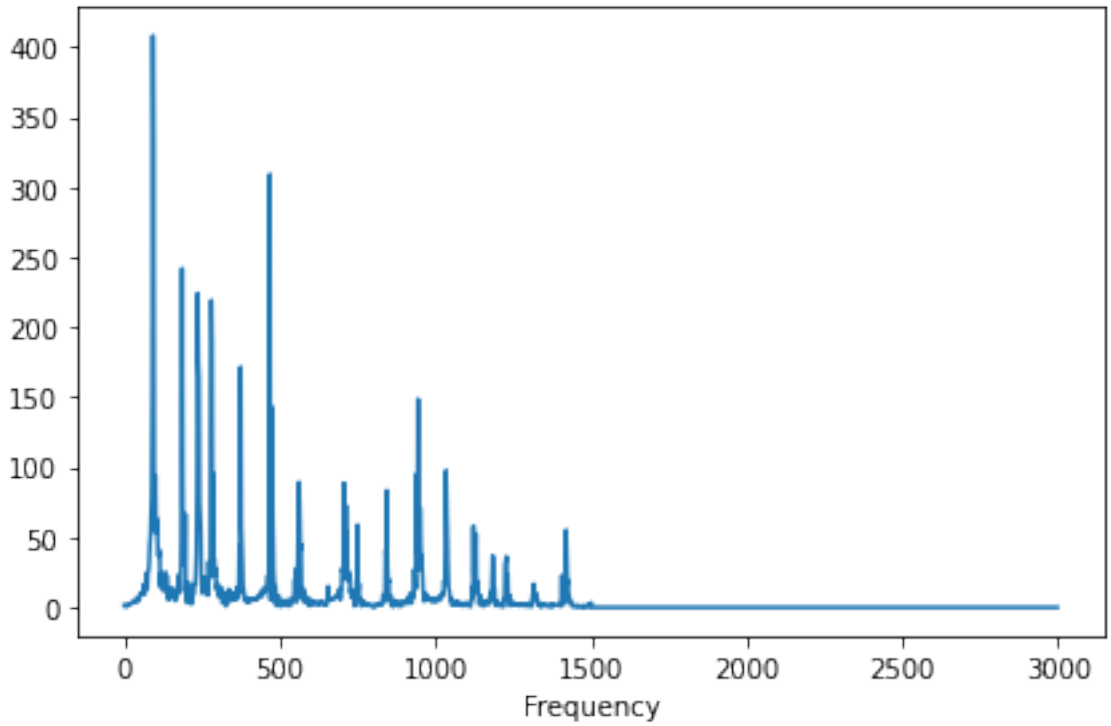

Рисунок 1.3. Спектр звука с урезанными частотами

Уберем частоты ниже 400, тем самым изменим доминирующую частоту.

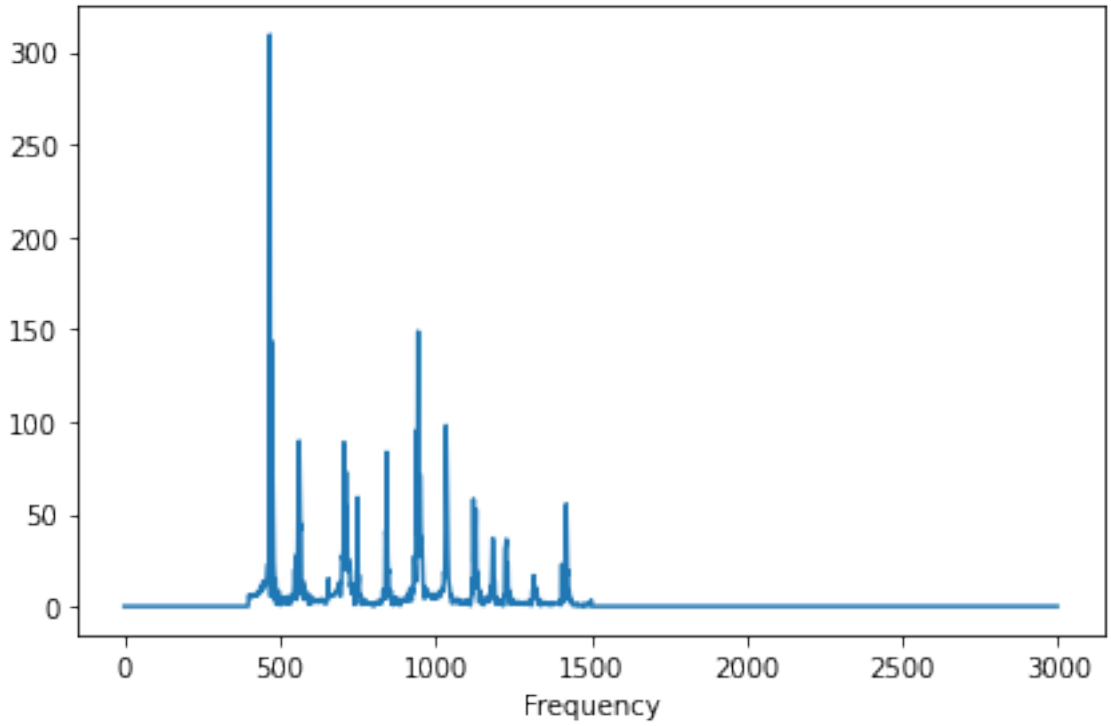

Рисунок 1.4. Спектр звука с частотами выше 400 и ниже 1500

Применим ФПЗ.

```
spectrum.band_stop(500, 800)
```

```
spectrum.plot(high=3000)
decorate(xlabel='Frequency')
```

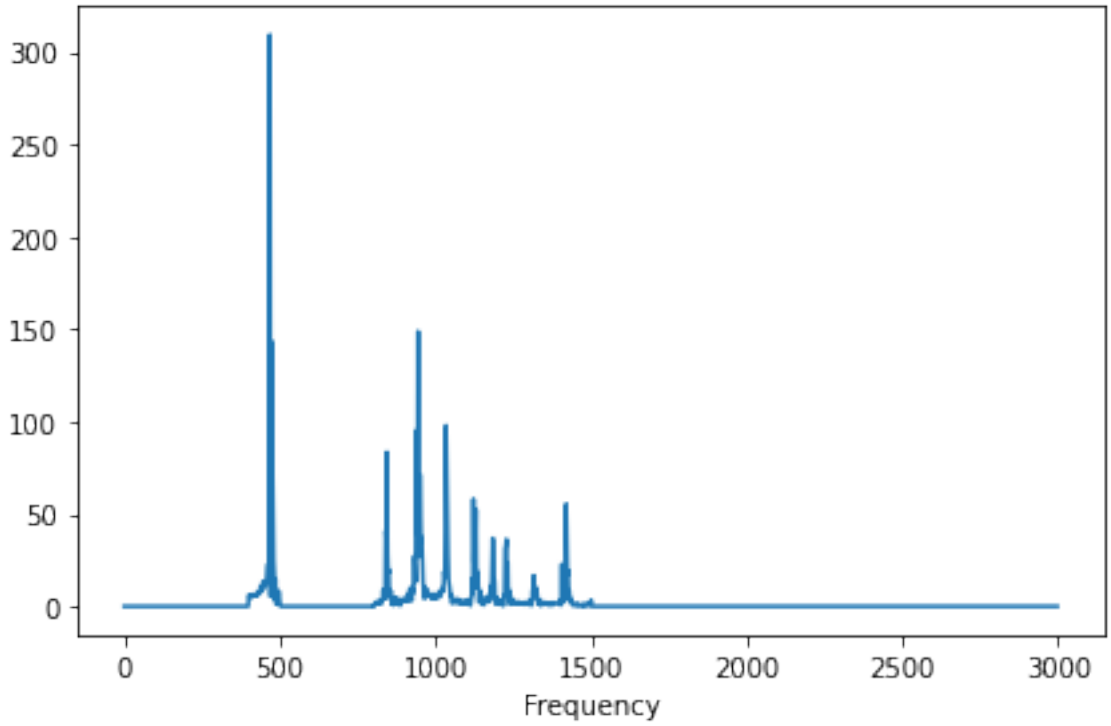

Рисунок 1.5. Спектр звука после ФПЗ

Преобразуем спектр обратно в сигнал.

```
filtered = spectrum.make_wave()
filtered.plot()
decorate(xlabel='Time')
```

Сравниваем первоначальный сигнал с отфильтрованным фрагментом.

```
segment.make_audio()
filtered.make_audio()
```

После обработки звук звучит уже не так объемно, и напоминает гудок, а не пианино

1.2. Упражнение 2

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?

Создадим сложный сигнал из CosSignal и SinSignal и построим график.

```
cos_sig = CosSignal(freq=50, amp=0.8, offset=0)
sin_sig = SinSignal(freq=200, amp=0.4, offset=0)
cos2_sig = SinSignal(freq=600, amp=0.6, offset=0)
sin2_sig = SinSignal(freq=1000, amp=0.2, offset=0)
m_sig = cos_sig + sin_sig + sin2_sig + cos2_sig
```

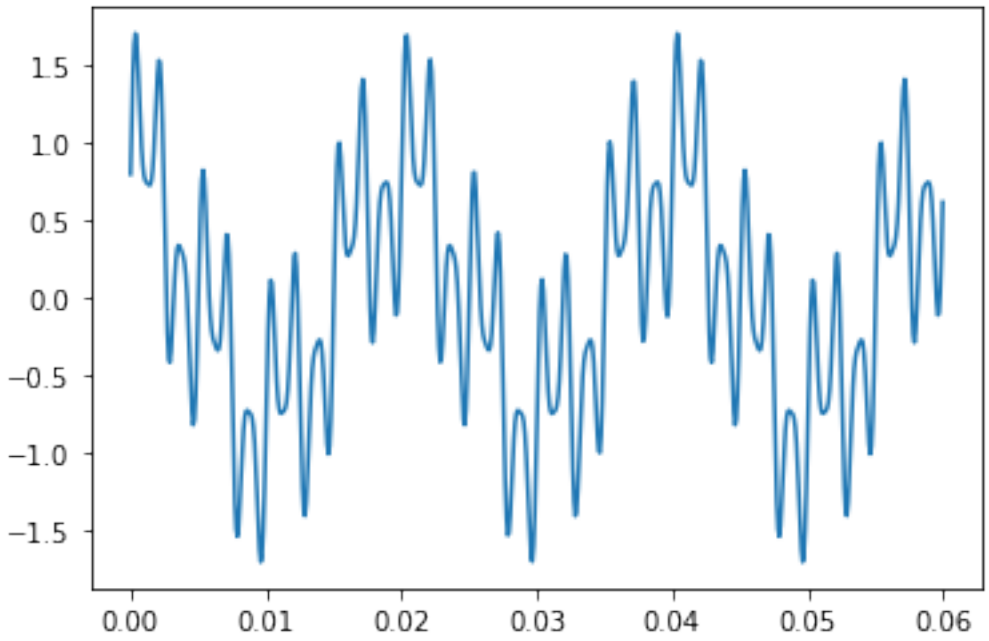

Рисунок 1.6. Граффик после суммирования сигналов

Создадим wave и вычислим спектр.

```
wave = m_sig.make_wave()
wave.make_audio()
spectrum = wave.make_spectrum()
spectrum.plot()
```

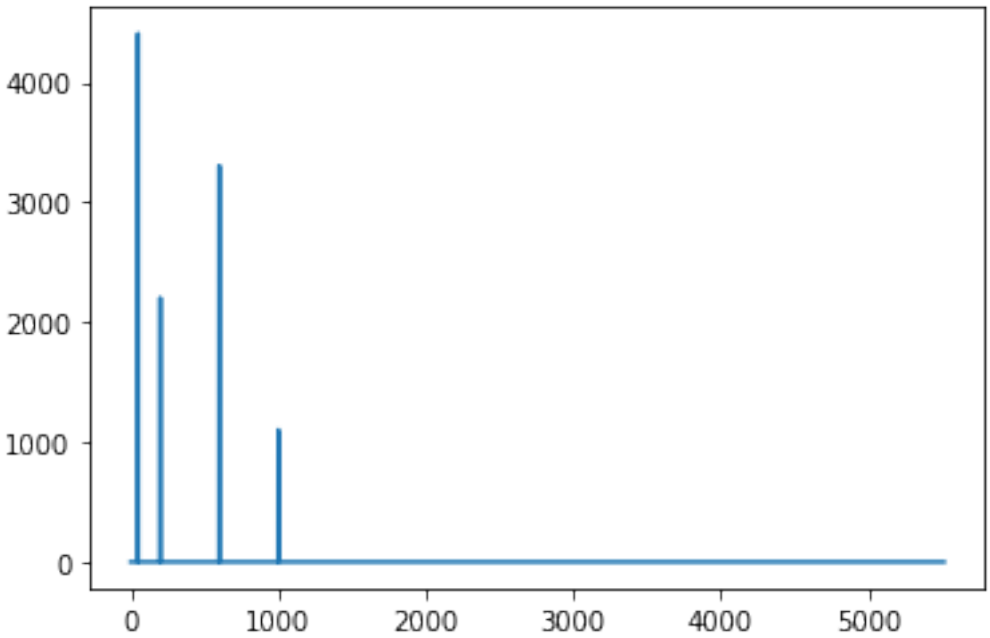

Рисунок 1.7. Спектр сигнала

Добавим частотный компонент, не кратный основным.

```
secondWave = (m_sig + CosSignal(freq=440, amp=0.75, offset=0)).make_wave()
secondWave.make_audio()
spectr = secondWave.make_spectrum()
spectr.plot()
```

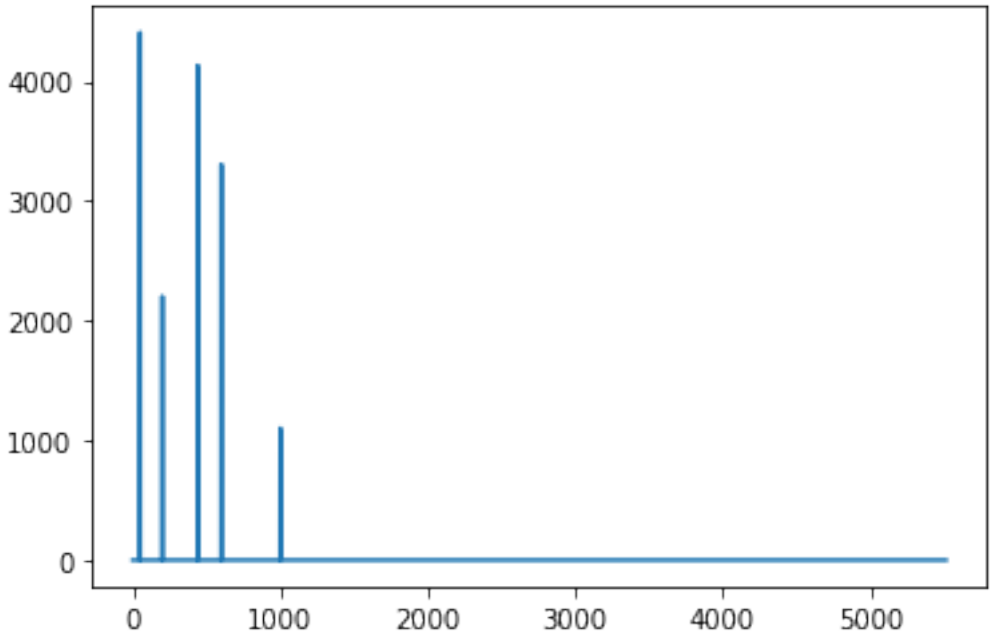

Рисунок 1.8. Спектр после изменения

После изменения звук стал не таким однотонным, каким был.

1.3. Упражнение 3

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

Напишем необходимую функцию.

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

Рисунок 1.9. Длинна изначального звука пианино

Попробуем ускорить и замедлить звук пианино.

```
fasterX2 = wave
stretch(fasterX2, 0.5)
fasterX2.make_audio()
```

```
slowX2 = read_wave('440931__xhale303__piano-loop-1.wav')
stretch(slowX2, 2)
slowX2.make_audio()
```

Рисунок 1.10. Длинна ускоренного звука

Рисунок 1.11. Длинна замедленного звука

По рисункам можно заметить , что функция работает весьма успешно, в случае ускорения длинна дорожки сокращается в два раза, в случае замедленния, увеличивается в два раза.

1.4. Вывод

В данной лабораторной работе, были изучены основные понятия при работе со звуком и проведена работа с библиотекой thinkDSP, которая позваляет производить различные действия с сигналами.

2. Гармоники

2.1. Упражнение 1

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?

Для создания пилообразного сигнала создадим класс SawtoothSignal. В методе класса evaluate опишем число циклов и с помощью библиотеки numpy разделим число циклов. unbias - смещает сигнал а normalize масштабирует его до заданной амплитуды.

```
import thinkdsp

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

Отобразим график пилообразного сигнала.

```
saw = SawtoothSignal()
saw.plot()
saw_wave = saw.make_wave(duration=3, framerate=10000)
```

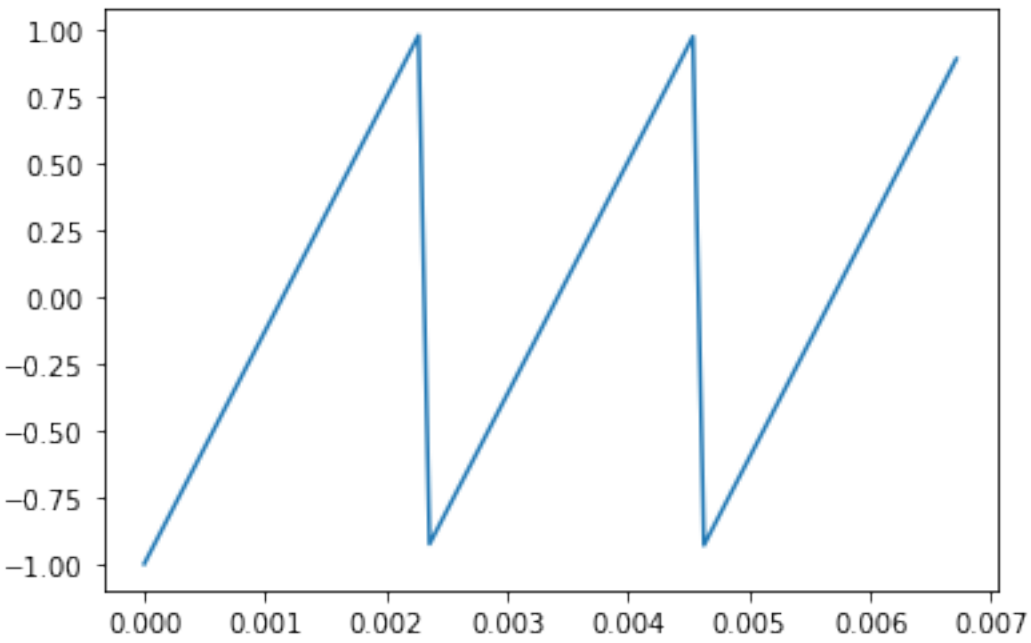

Рисунок 2.1. График пилообразного сигнала

Вычислим спектр.

```
spectr = saw_wave.make_spectrum()
spectr.plot()
```

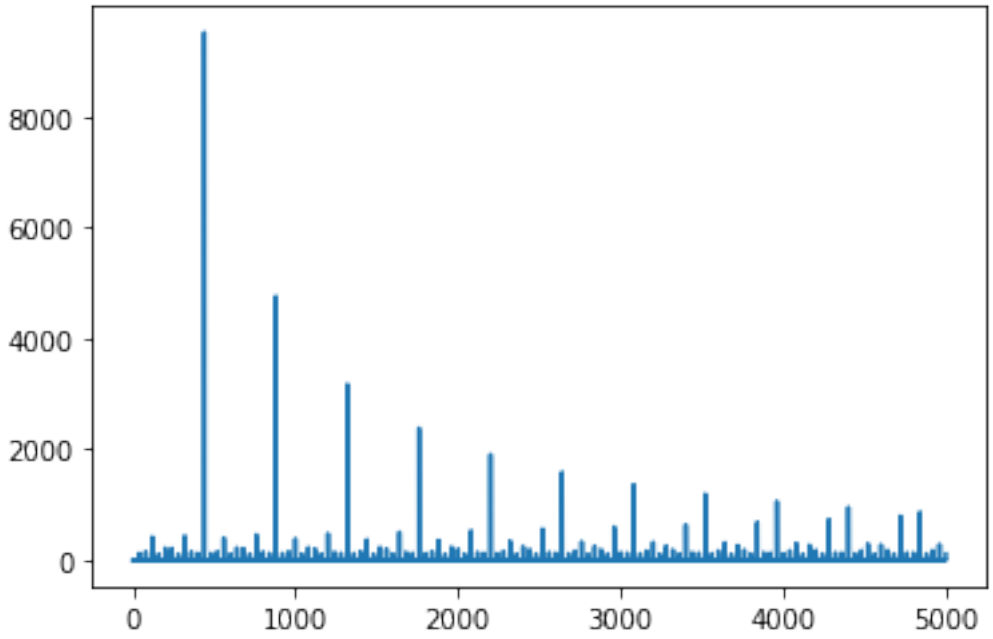

Рисунок 2.2. Спектр пилообразного сигнала

Теперь сравним полученный спектр с треугольными и прямоугольными сигналами.

```
triangle = TriangleSignal()
triangle.make_wave(duration=3, framerate=10000).make_spectrum().plot()
```

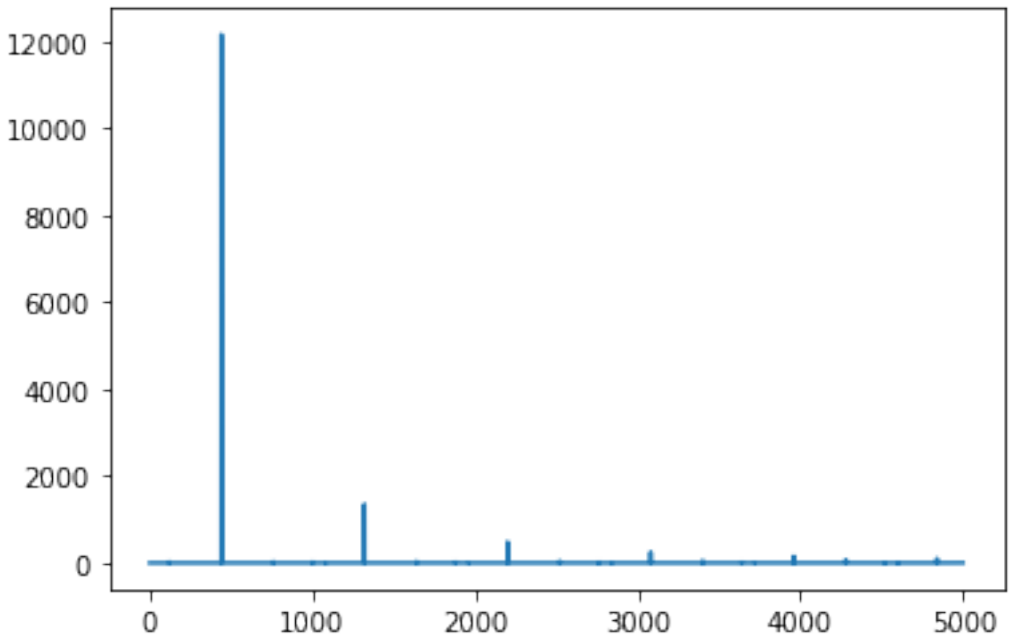

Рисунок 2.3. Спектр треугольного сигнала

```
square = SquareSignal()
square.make_wave(duration=3, framerate=10000).make_spectrum().plot()
```

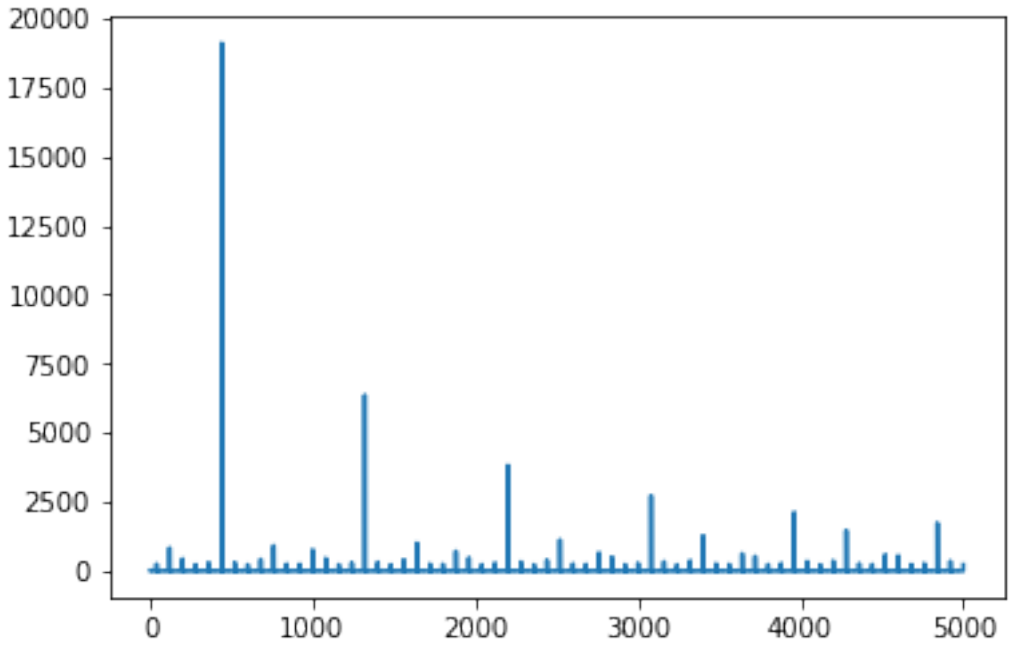

Рисунок 2.4. Спектр прямоугольного сигнала

Можно заметить, в сравнении с треугольным сигналом, его амплитуда падает пропорционально квадрату частоты, а у пилообразного пропорционально частоте, ровно также как и у квадратного сигнала, однако пилообразный включает в себя как четные, так и нечетные гармоники, когда у квадратного только нечетные.

2.2. Упражнение 2

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?

```
square = thinkdsp.SquareSignal(1100)
segment = square.make_wave(duration=1, framerate=10000)
spectr = segment.make_spectrum()
spectr.plot()
```

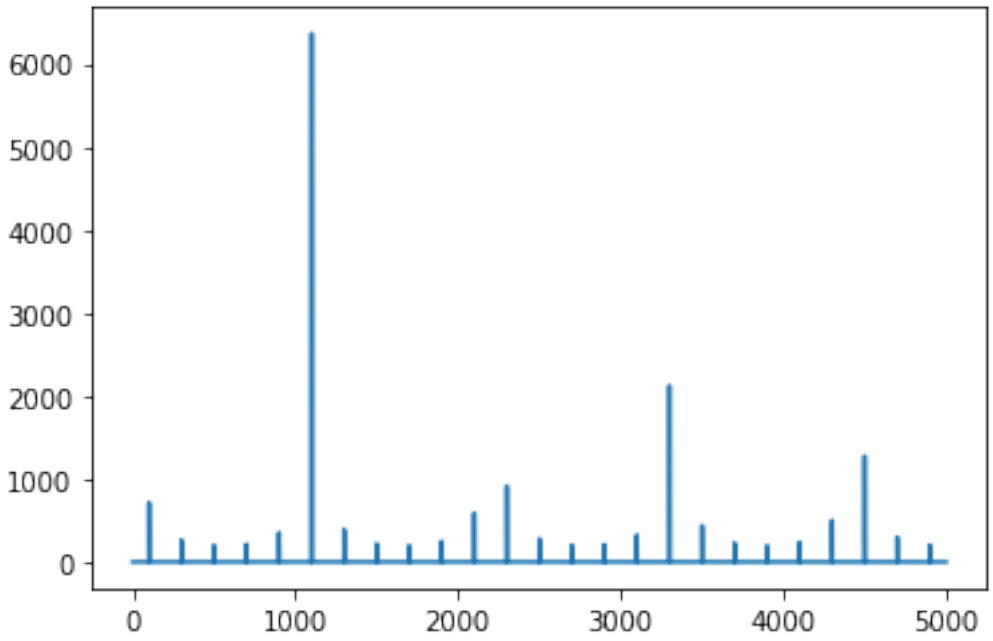

Рисунок 2.5. Спектр сигнала с биениями

На спектре видно, что происходит "завернутость" биений, гармоники 1100 и 3300 выстроены верно, однако видно, что третий и четвертый пик находятся на 4500 и 2300 соответственно, они неотличимы от 5500 и 7700

2.3. Упражнение 3

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?

Создадим треугольный сигнал с частотой 440Hz и длительностью 0,01 сек, построим его график, распечатаем сигнал и распечатаем Spectrum.hs[0].

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

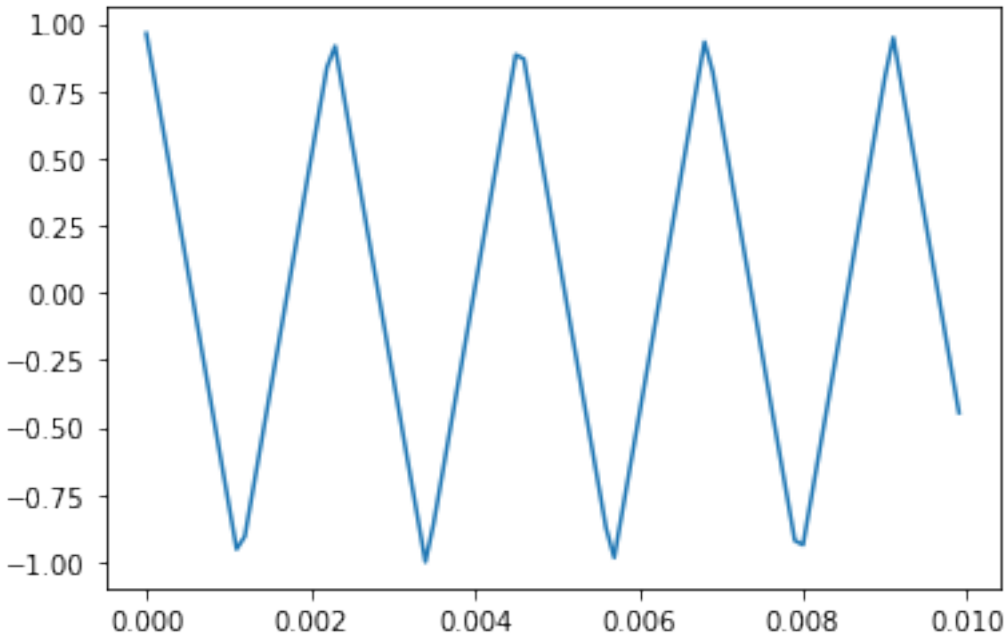

Рисунок 2.6. График сигнала

Проверим что лежит в 0 элементе.

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

```
(3.375077994860476e-14+0j)
```

Каждый элмент массива hs объекта Spectrum представлет собой комплексное число. Они соотвествуют частотной компоненте: размах пропоцианален амплитуде соответствующей компоненты, а угол - это фаза.
Видно, что первый элемент массива hs - это комплексное число близкое к нулю. Установим этому элементу значение 100 и посмотрим на результат

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

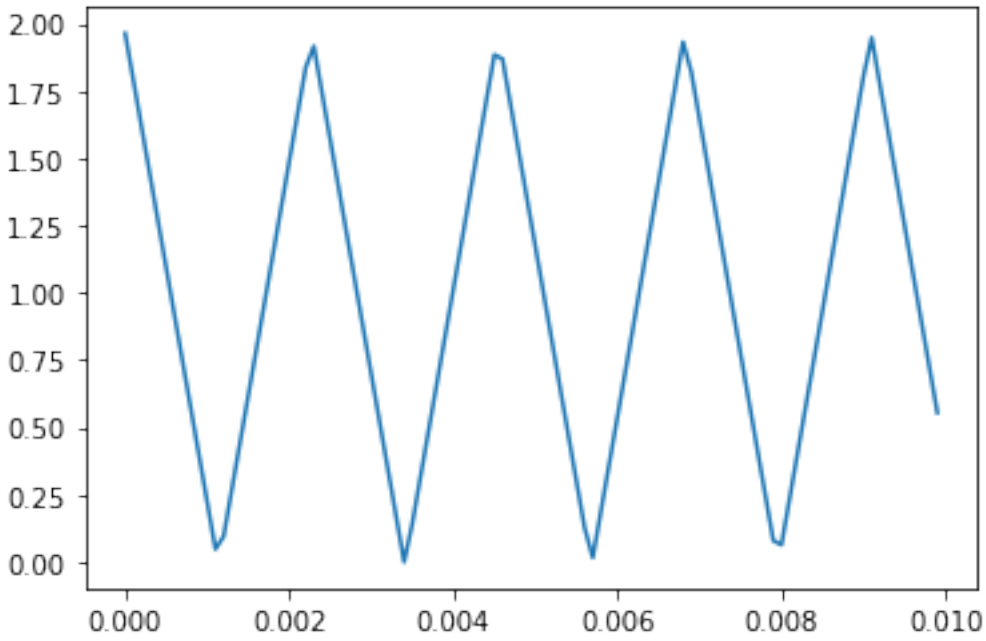

Рисунок 2.7. График сигнала с изменённым нулевым числом спекторграммы

Из полученного графика видно смещение сигнала по вертикали, таким образом можно сделать вывод о том, что первый элемент отвечает за смещение по вертикали, в случае, если он близок к нулю, то сигнал не смещается.

2.4. Упражнение 4

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?

Напишем функцию которая принимает на вход Spectrum и изменяет его его делением каждого элемента hs на соответствующую частоту fs и проверим эту функцию на треугольном сигнале

```
def spec_div(spec):
  spec.hs[1:] /= spec.fs[1:]
  spec.hs[0] = 0
  spec.plot()

triangle = TriangleSignal()
wave = triangle.make_wave(duration=0.5, framerate=10000)
wave.make_audio()

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

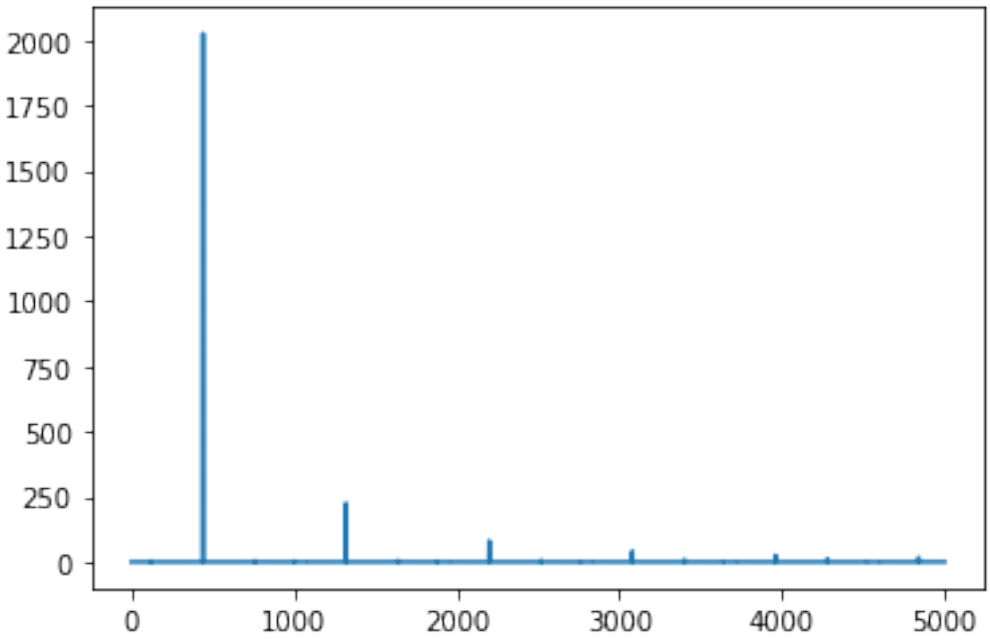

Рисунок 2.8. Спектр треугольного сигнала

Применим написанную функцию.

```
spec_div(spectr)
```

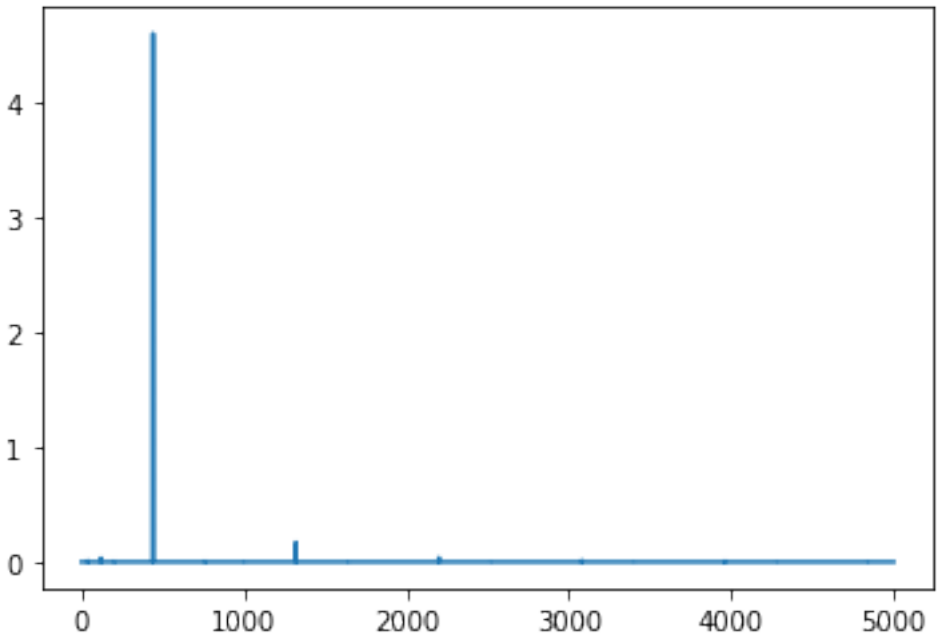

Рисунок 2.9. Спектр изменённого сигнала

```
spectr.make_wave().make_audio()
```

Если прослушать полученный звук, можно услышать что он стал тише и чище, это связано с работой функции, которая ослабляет низкие частоты на некоторую величину.

2.5. Упражнение 5

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.
Создадим сигнал, состоящий из четных и нечетных гармоник, при этом, чтобы они падали пропорцианально квадрату частоты. Возьмем пилообразный сигнал, который имет четные и нечетные гармоники, а далее скорректируем его спад при помощи функции из предыдущего упражнения.

```
saw_sign = SawtoothSignal(400)
saw_w = saw_sign.make_wave(duration=0.5, framerate=20000)
spectr = saw_w.make_spectrum()
spectr.plot()
decorate(xlabel='Frequency (Hz)')
```

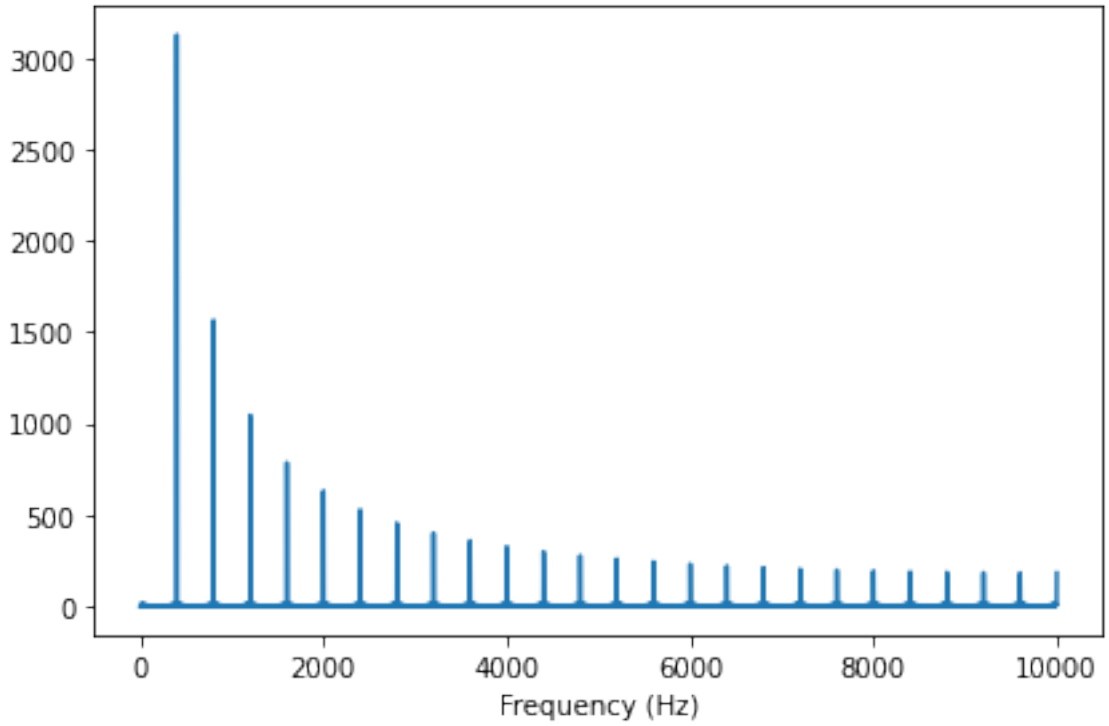

Рисунок 2.10. Спектр пилообразного сигнала

Применим функцию для изменения амплитуды спада

```
spec_div(spectr)
spectr.scale(400)
spectr.plot()
```

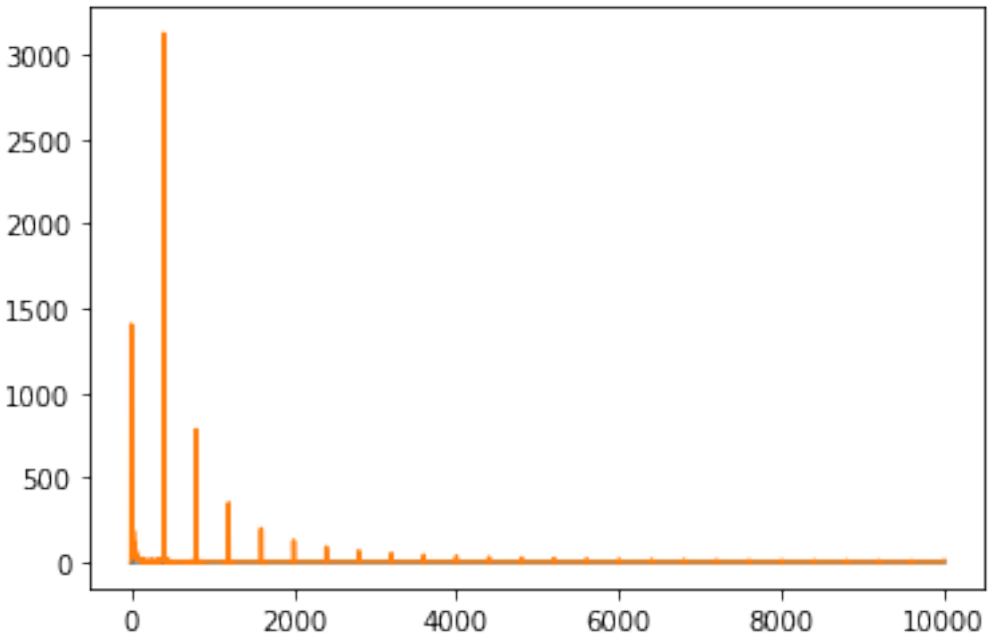

Рисунок 2.11. Спектр пилообразного сигнала после функции

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

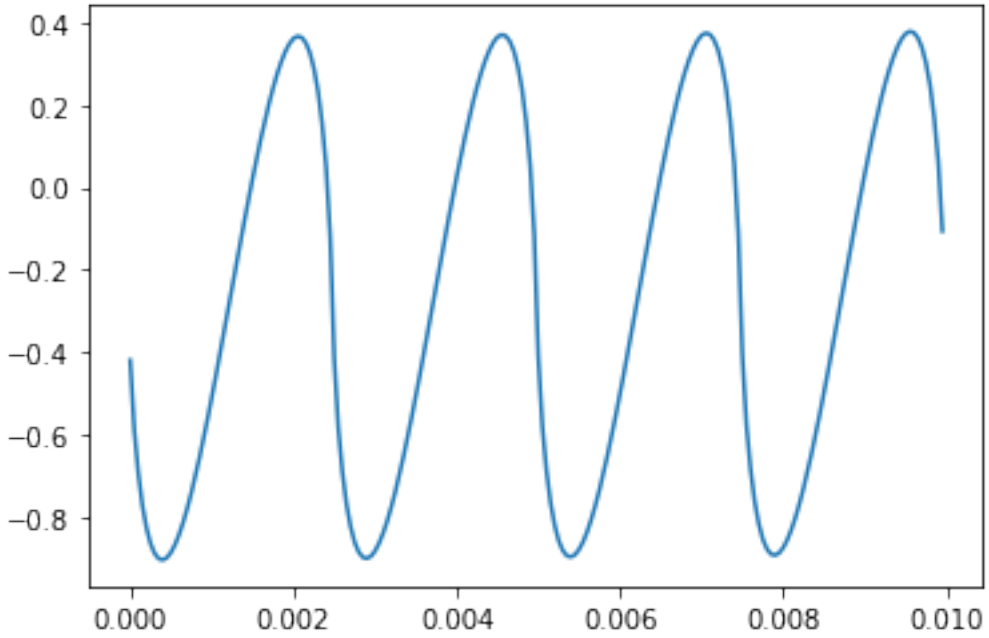

Рисунок 2.12. График сигнала

Видно, что спектр спадает пропорционально квадрату частоты и при этом имеет четные и нечетные гармоники

2.6. Вывод

В данной работе были произведены различные действия с разными видами сигналов, были рассмотрены спектры и гармонические структуры и биения.

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)
```

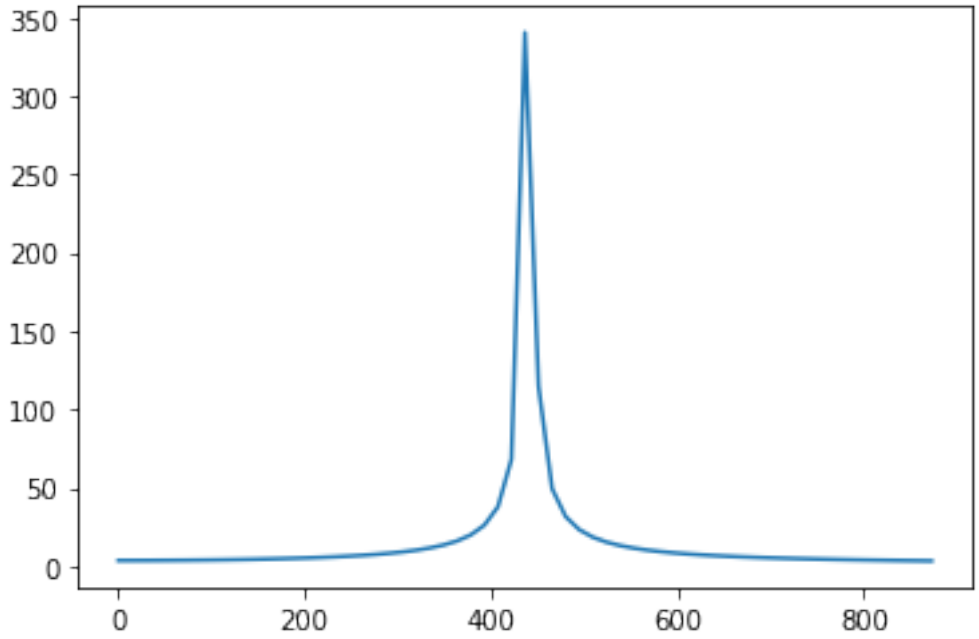

Рисунок 3.1. Рассматриваемый сигнал

Посмотрим как выглядит спектограмма с использованием окна Хэмминга:

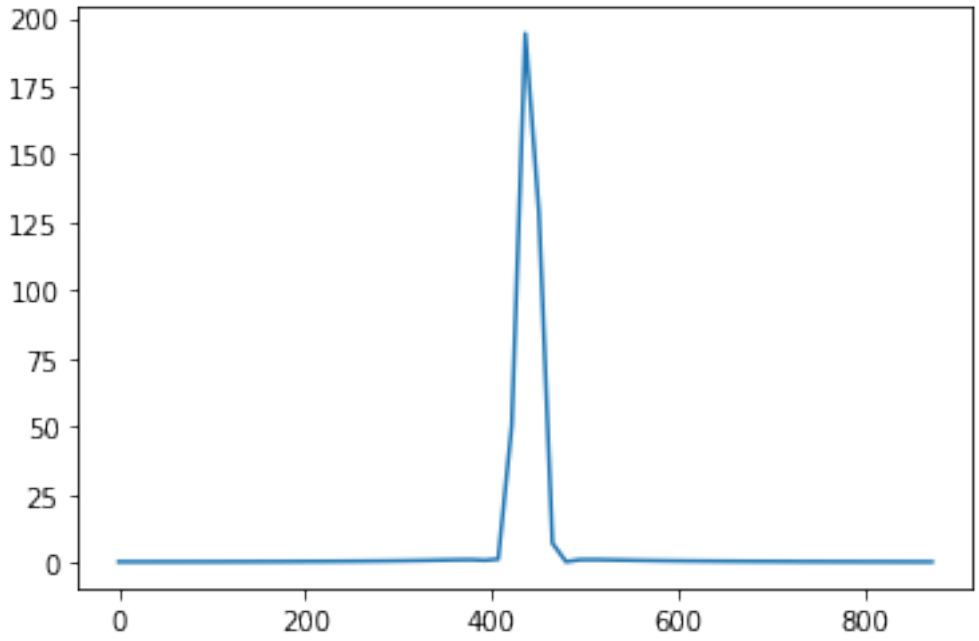

Рисунок 3.2. Сигнал с использованием окна Хэмминга

Используем другое окно а именно Барлетта

```
wave = signal.make_wave(duration)
wave.ys *= np.bartlett(len(wave.ys))
spectrum = wave.make_spectrum()
spectrum.plot(high=880)
```

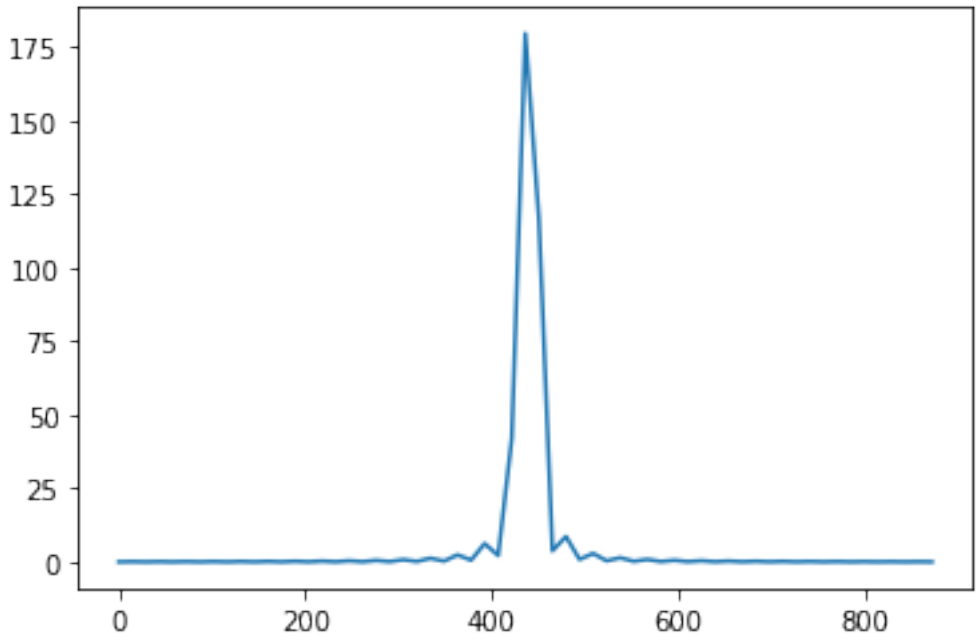

Рисунок 3.3. Сигнал с использованием окна Барлетта

Можно заметить. что низкие амплитуды стали ломанными линиями.

3.2. Упражнение 2

Напишите класс SawtoothChirp, расширяющий Chirp и переопределяющий evaluate для генерации пилообразного сигнала с линейно увеличивающейся частотой.

```
import thinkdsp
from thinkdsp import normalize, unbias
from math import pi

class SawtoothChirp(Chirp):

    def evaluate(self, ts):
        freqs = np.linspace(self.start, self.end, len(ts))
        dts = np.diff(ts, prepend=0)
        dphis = (2 * np.pi) * freqs * dts
        phases = np.cumsum(dphis)
        cycles = phases / (2 * np.pi)
        frac, _ = np.modf(cycles)
        ys =  normalize(unbias(frac), self.amp)
        return ys
```

Создадим пилообразный сигнал от второй до пятой октавы "ля" 880 - 7040.

```
signal = SawtoothChirp(start=880, end=7040)
wave = signal.make_wave(duration=3, framerate=4000)
wave.apodize()
wave.make_audio()
sp = wave.make_spectrogram(seg_length=512)
sp.plot(high=5000)
```

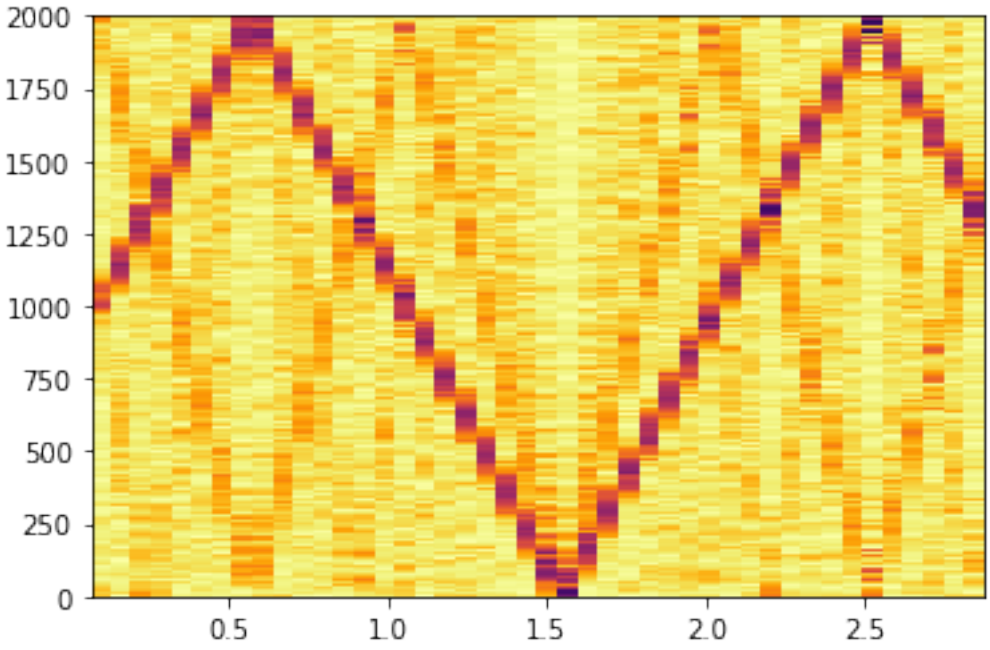

Рисунок 3.4. Спектр сигнала

По звуку и эскизу виден эффект биения.

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.

```
sawC = SawtoothChirp(start=2500, end=3000)
wave = sawC.make_wave(duration=1, framerate=20000)
wave.make_spectrum().plot()
```

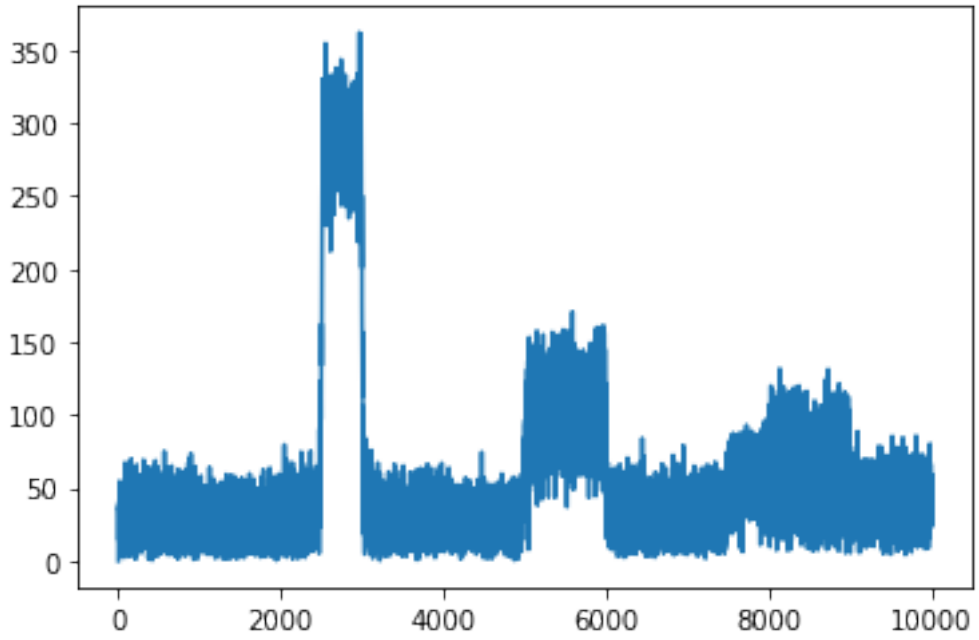

Рисунок 3.5. Спектр сигнала

По графику видно, что ожидаемо базовая частота находится в пределах от 2500Hz до 3500 Hz, следующие же гармоники отличаются от базовой и находятся на частотах от 5000 до 6000 и 7500 до 9000 герц, остальные гармоники наложены друг на друга

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.

Для вывода соответствующей спектограммы, возьмем нужный нам звук из репозитория учебника:

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

from thinkdsp import read_wave
wave = read_wave('72475__rockwehrmann__glissup02.wav')
wave.make_audio()
wave.make_spectrogram(512).plot(high=5000)
```

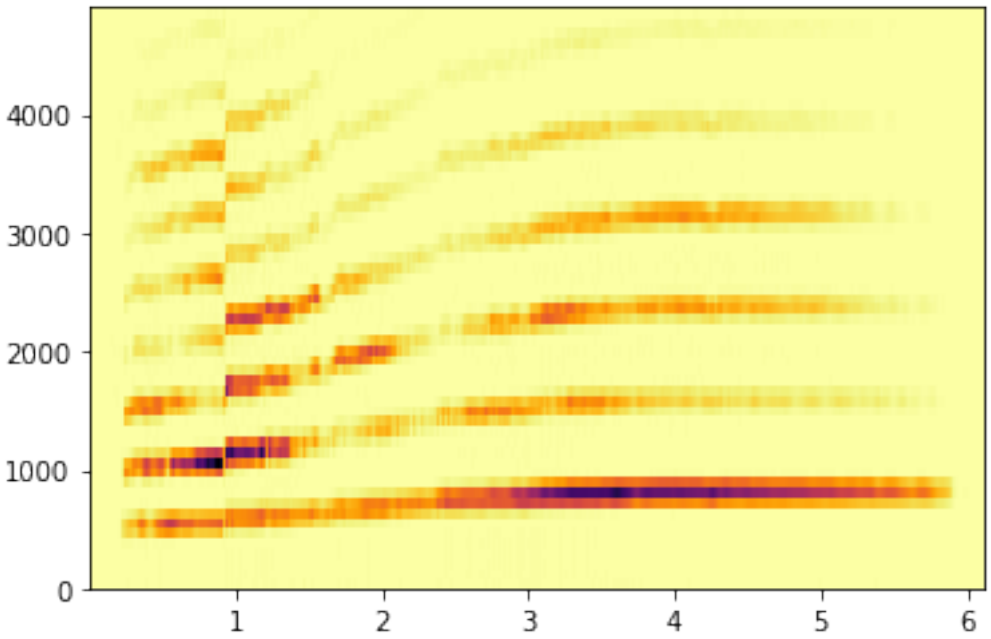

Рисунок 3.6. Спектрограмма сигнала

Видим, что спектограмма очень похожа на наш 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):
        lengths = np.linspace(1.0 / self.start, 1.0 / self.end, len(ts))
        freqs = 1 / lengths
        dts = np.diff(ts, prepend=0)
        dphis = np.pi * 2 * freqs * dts
        phases = np.cumsum(dphis)
        ys = self.amp * np.cos(phases)
        return ys
```

Создадим два сигнала-глиссандо от C3 до F3 и от F3 к C3 и соеденим эти два сигнала.

```
firstSignal = TromboneGliss(262, 349)
firstWave = firstSignal.make_wave(duration=1)
spectr = firstWave.make_spectrogram(1024)
spectr.plot(high=1000)
```

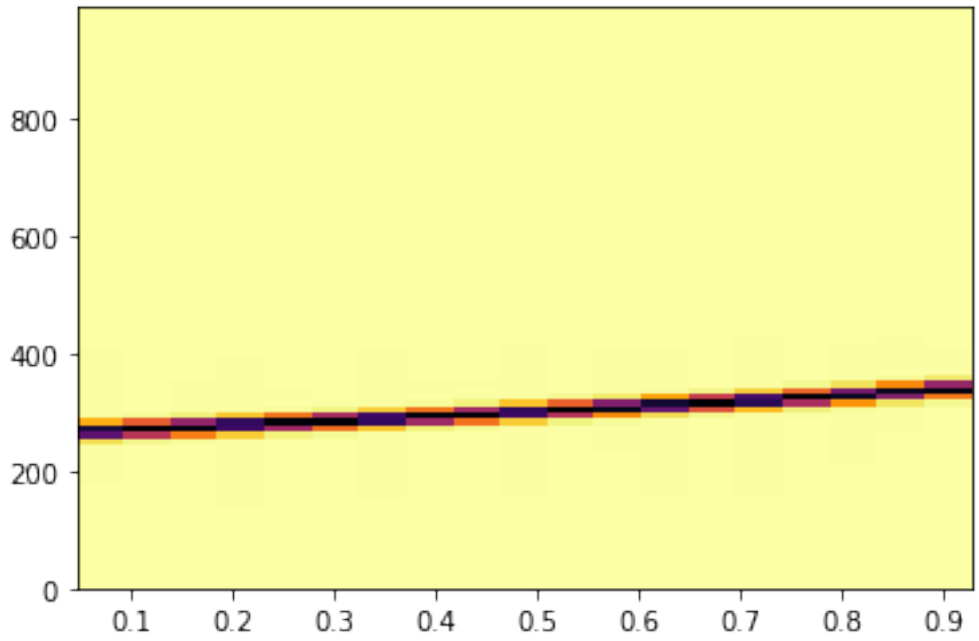

Рисунок 3.7. Спектрограмма первого сигнала

```
secondSignal = TromboneGliss(349, 262)
secondWave = secondSignal.make_wave(duration=1)
secondWave.make_audio()
spectr = secondWave.make_spectrogram(1024)
spectr.plot(high=1000)
```

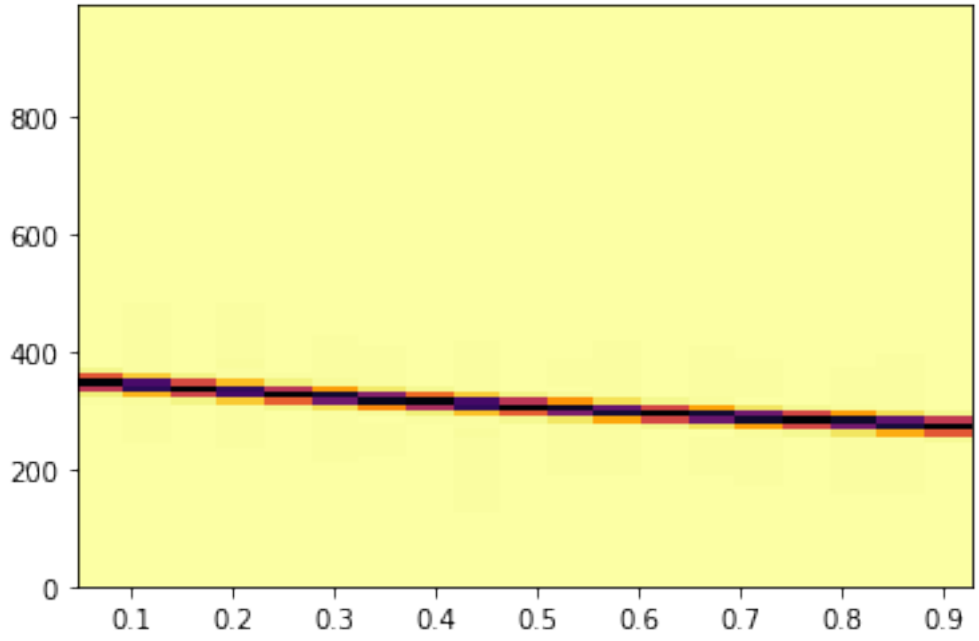

Рисунок 3.8. Спектрограмма второго сигнала

Объеденим сигналы

```
result = firstWave | secondWave
result.make_audio()
spectr = result.make_spectrogram(1024)
spectr.plot(high=1000)
```

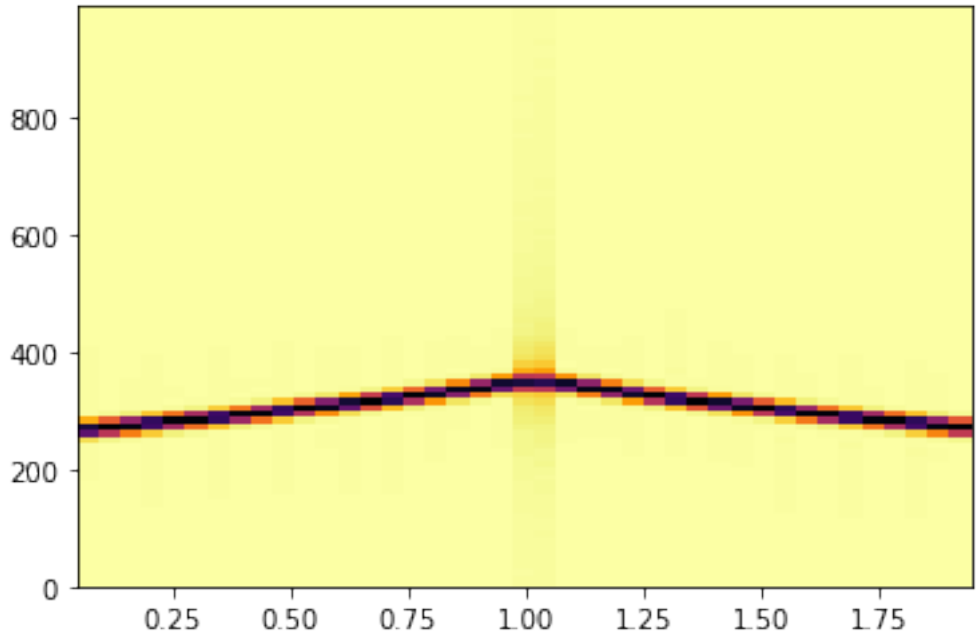

Рисунок 3.9. Спектрограмма объеденненного сигнала

Слышно, как две части идут друг за другом, также это хорошо видно на графике

3.6. Упражнение 6

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

Снова воспользуемся репозиторием учебника и возьмем оттуда звуки глассных.

```
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(1000)
```

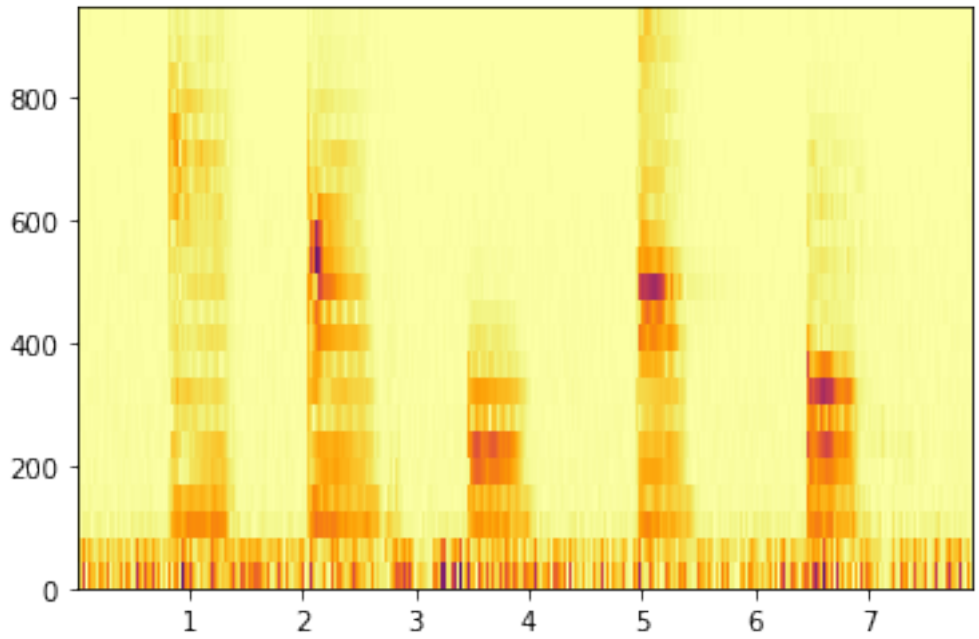

Рисунок 3.10. Спектрограмма гласных звуков

На записи глассных есть 5 звуков и по спектограмме отчетливо видны эти глассные звуки в виде пиков.

3.7. Вывод

В данной работе были рассмотрены частотные компоненты и апериодические сигналы.

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?

Возьмем звук моря и выделим два сегмента.

```
if not os.path.exists('13793__soarer__north-sea.wav'):
    !wget https://github.com/Eugenepolyt/telecomunaction/raw/main/13793__so
from thinkdsp import read_wave
wave = read_wave('13793__soarer__north-sea.wav')
wave.make_audio()
segment = wave.segment(start=15, duration=1.0)
segment.make_audio()
segment.plot()
```

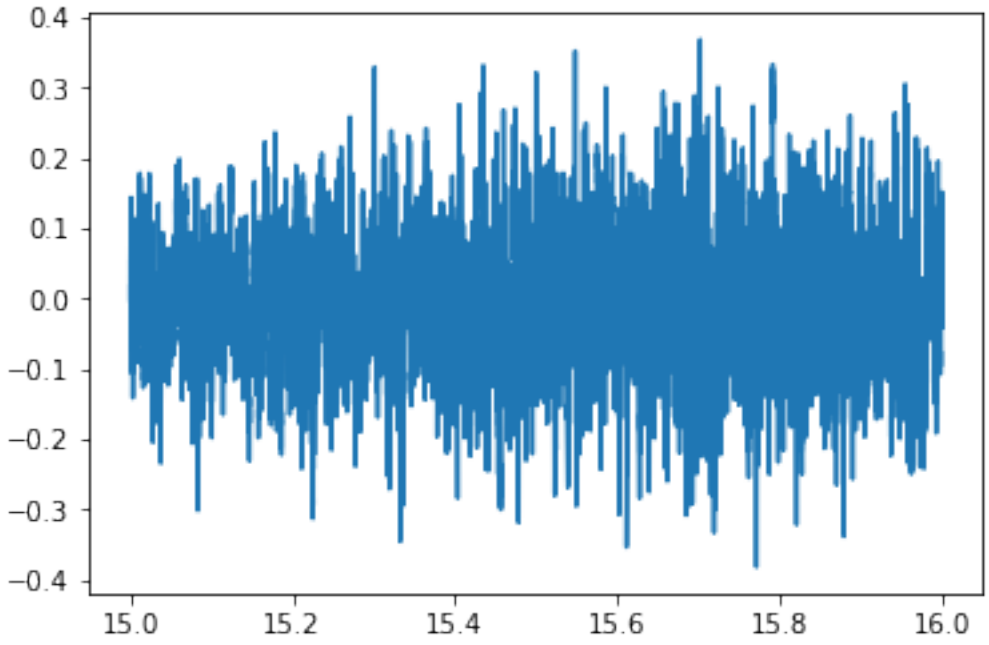

Рисунок 4.1. График сигнала

Определим характеристики шума.

```
from thinkdsp import decorate
spectrum.plot_power()

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

```
            **loglog)
```

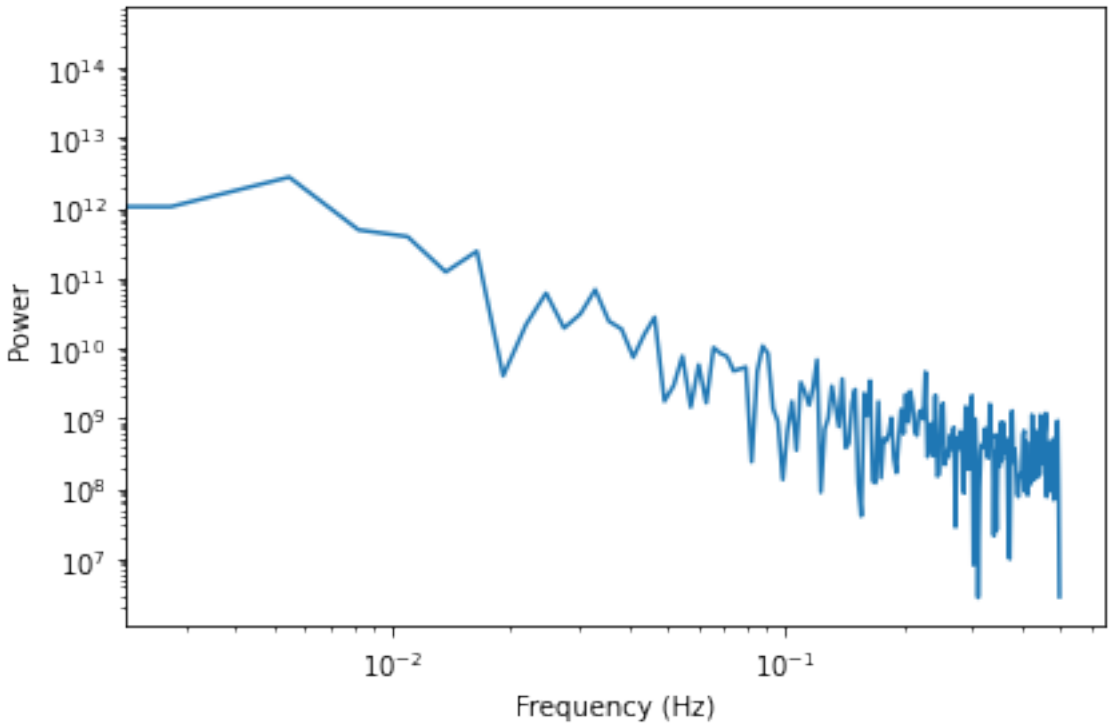

Рисунок 4.2. Спектр в логорифмическом масштабе

График напоминает белый шум, возьмем теперь идущий за ним другой сегмент.

```
segmentNext = wave.segment(start=16, duration=1.0)
segmentNext.make_audio()

spectrumNext = segmentNext.make_spectrum()
spectrum.plot_power()
spectrumNext.plot_power()

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

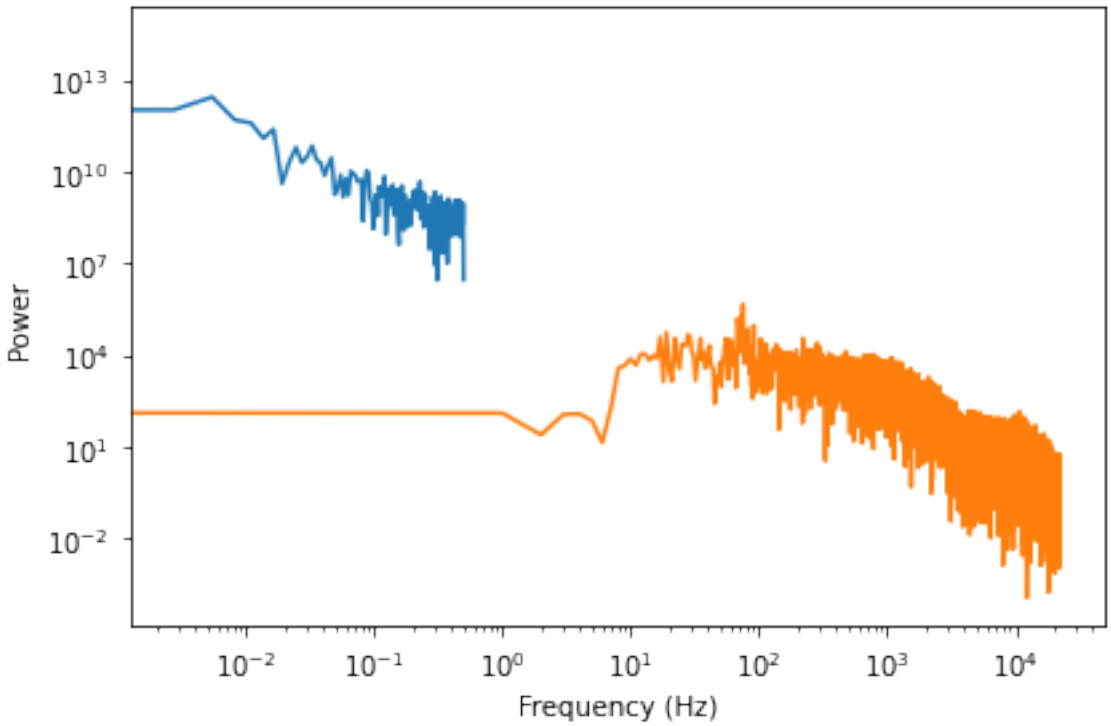

Рисунок 4.3. Сравнение спектров в логорифмическом масштабе

4.2. Упражнение 2

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

Реализуем метод Бартлетта для оценки спектра мощности шумового сигнала. Данный метод будет разделять сигнал на сегменты, вычислять для них разложение Фурье, вычислять сумму квадратов, находить среднее и вычислять корень.

```
from thinkdsp import Spectrum

def make_barlett(wave, N, flag=True):
  spectrogram = wave.make_spectrogram(N, flag)
  spec_mac = spectrogram.spec_map.values()

  powers = []
  for spectrum in spec_mac:
    powers.append(spectrum.power)

  hs = np.sqrt(sum(powers)/ len(powers))
  fs = next(iter(spec_mac)).fs

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

Проведем тестирование на сигнале с предыдущего задания.

```
barlett = make_barlett(segmentNext,1024)
barlett.plot_power()
decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
```

```
        **loglog)
```

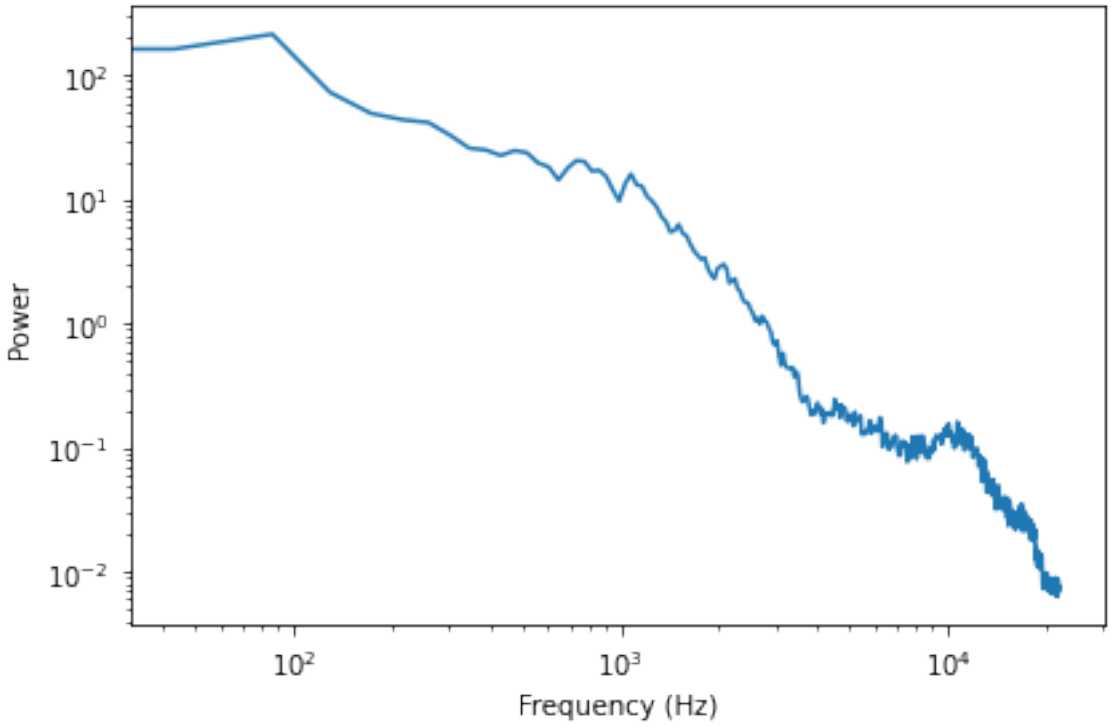

Рисунок 4.4. Результат работы функции

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?

Скачаем csv файл с ценами на биткоин

```
if not os.path.exists('market-price.csv'):
    !wget https://github.com/Eugenepolyt/telecomunaction/raw/main/market-pr
import csv
worth = []
with open('market-price.csv') as File:
  reader = csv.reader(File, delimiter=',', quotechar=',',
                      quoting=csv.QUOTE_MINIMAL)
  for row in reader:
    worth.append(row[1])
worth = worth [1:]
days = a=np.arange(0,len(worth))

from thinkdsp import Wave
wave = Wave(worth,days,1)
spectrum = wave.make_spectrum()
spectrum.plot_power()
decorate(xlabel='Частота',
         ylabel='Мощность',
         **loglog)
```

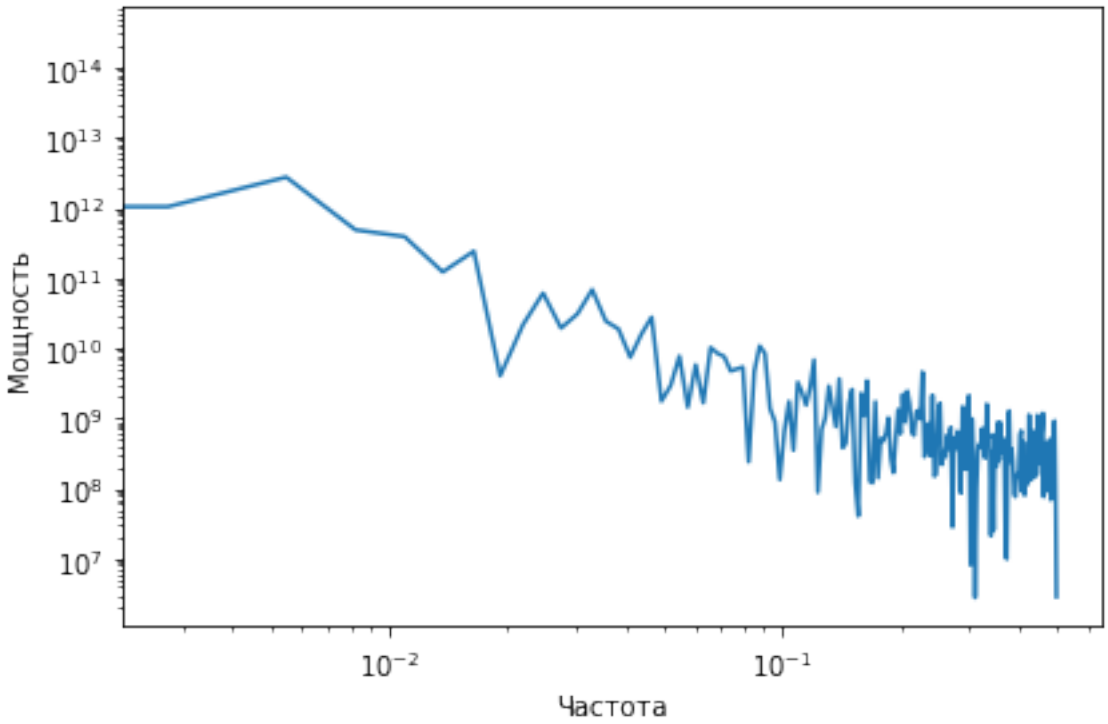

Рисунок 4.5. Спектрограмма цен BitCoin в логорифмическом формате

График напоминает красный шум.

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.

Напишем класс UncorrelatedPoissonNoise, который наследуется от класса thinkdsp Noise и который моделирует некоррелированный пуассонвский шум (UP

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

Сгенерируем сигнал с маленькой амплитудой, звук должен быть похож на счетчик Гейгера

```
firstSignal = UncorrelatedPoissonNoise(amp=0.001)
firstWave = firstSignal.make_wave(duration=1, framerate=10000)
firstWave.make_audio()
```

Теперь сгенерируем сигнал с большой амплитудой

```
secondSignal = UncorrelatedPoissonNoise(1)
secondWave = secondSignal.make_wave(duration=1,framerate = 10000)
secondWave.make_audio()
```

Посмотрим на характеристики данных сигналов

```
spectrum1 = firstWave.make_spectrum()
spectrum2 = secondWave.make_spectrum()

spectrum1.plot_power()
spectrum2.plot_power()

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

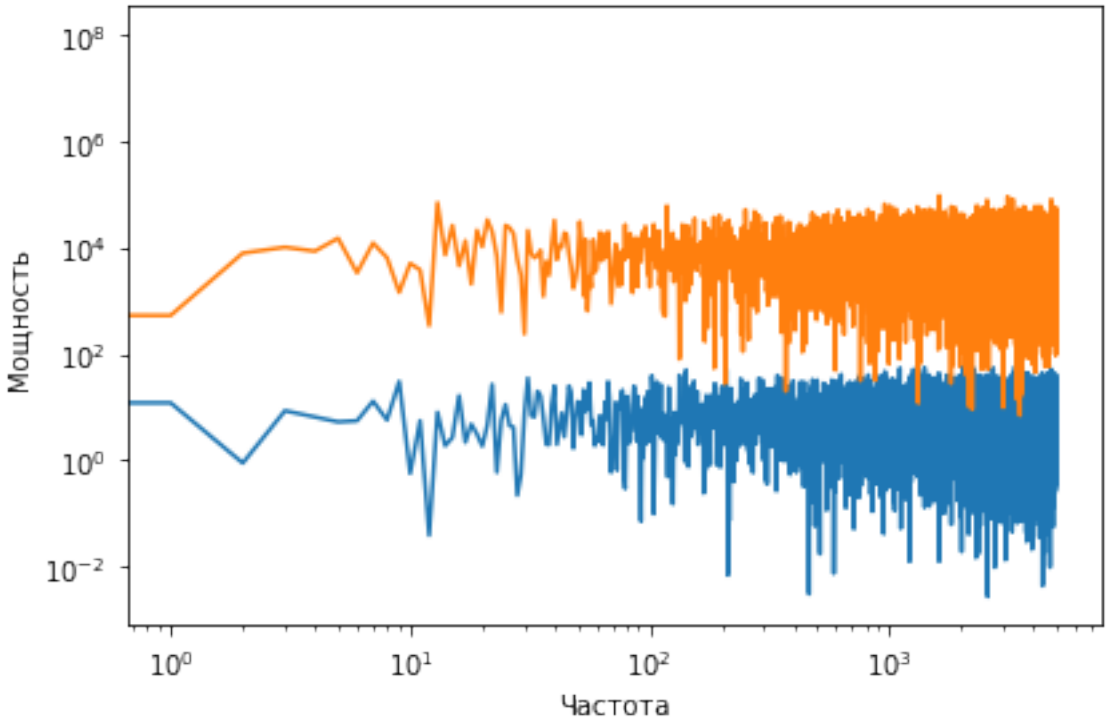

Рисунок 4.6. Сравнение спектров

Видно, что при увеличении амплитуды звук больше похож на белый шум

4.5. Упражнение 5

В этой главе описан алгоритм генерации розового шума. Концептуально простой, но вычислительно затратный. Есть более эффективные альтернативы, такие как алгоритм

Босса-Маккартни.

Используем алгоритм Voss-McCartney для генерации розового шума

```
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

ys = voss(11025,16)
wave = Wave(ys)
wave.plot()
```

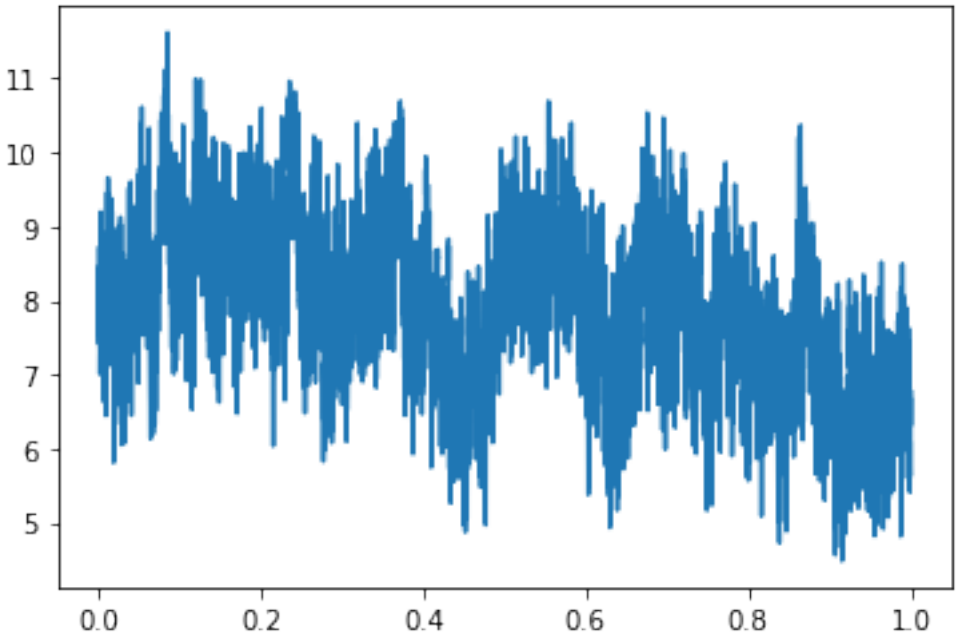

Рисунок 4.7. Сгенерированный сигнал

```
spectrum = wave.make_spectrum()
spectrum.hs[0] = 0
spectrum.plot_power()
decorate(xlabel='Частота',
```

```
    ylabel='Мощность',
    **loglog)
```

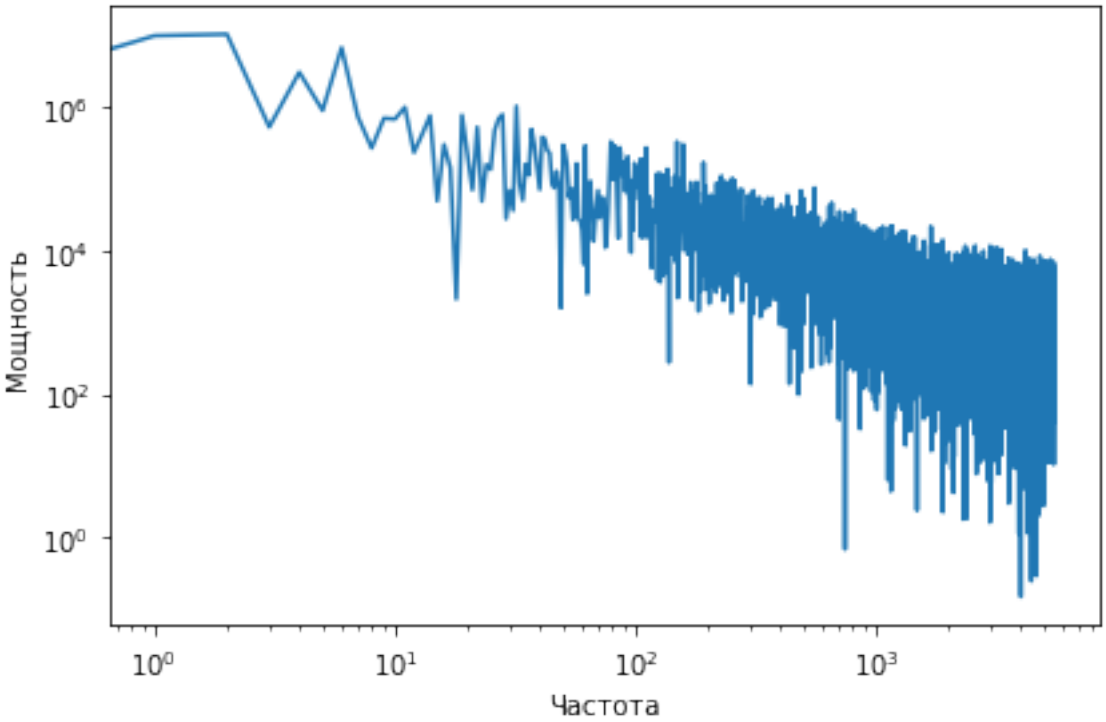

Рисунок 4.8. Спектр сигнала

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

```
    -1.03581503502058
```

Видим, что в итоге был получен сигнал розового шума

4.6. Вывод

В данной работе были рассмотрены различные виды шумов. Шум представляет собой сигнал содержащий компоненты с различными частотами, но не имеющий гармонической структуры переодических сигналов.

5. Автокорреляция

5.1. Упражнение 1

Оцените высоты тона вокального чирпа для нескольких времён начала сегмента.

```
if not os.path.exists('28042__bcjordan__voicedownbew.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/28042__bc
from thinkdsp import read_wave
wave = read_wave('28042__bcjordan__voicedownbew.wav')
wave.normalize()
wave.make_audio()

duration = 0.01
segment1 = wave.segment(start=0.5, duration=duration)
segment1.plot()
segment2 = wave.segment(start=0.6, duration=duration)
segment2.plot()
segment3 = wave.segment(start=0.7, duration=duration)
segment3.plot()
```

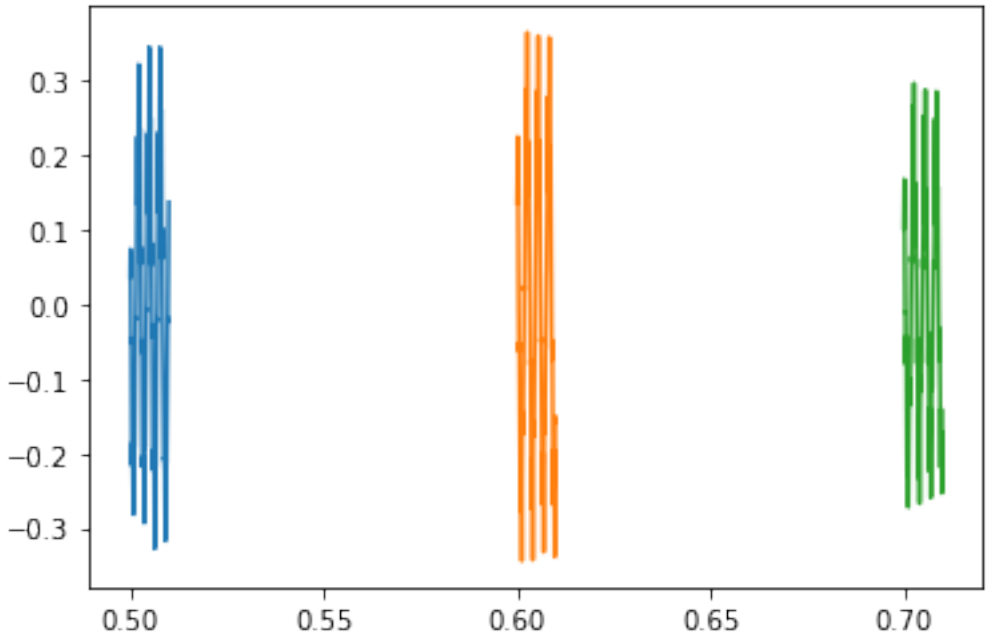

Рисунок 5.1. График сегментов

Используем автокореляцию.

```
lags1, corrs1 = autocorr(segment1)
plt.plot(lags1, corrs1, color='black')
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1, 1])

lags2, corrs2 = autocorr(segment2)
plt.plot(lags2, corrs2)
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1, 1])
```

```
lags3, corrs3 = autocorr(segment3)
plt.plot(lags3, corrs3, color='red')
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1, 1])
```

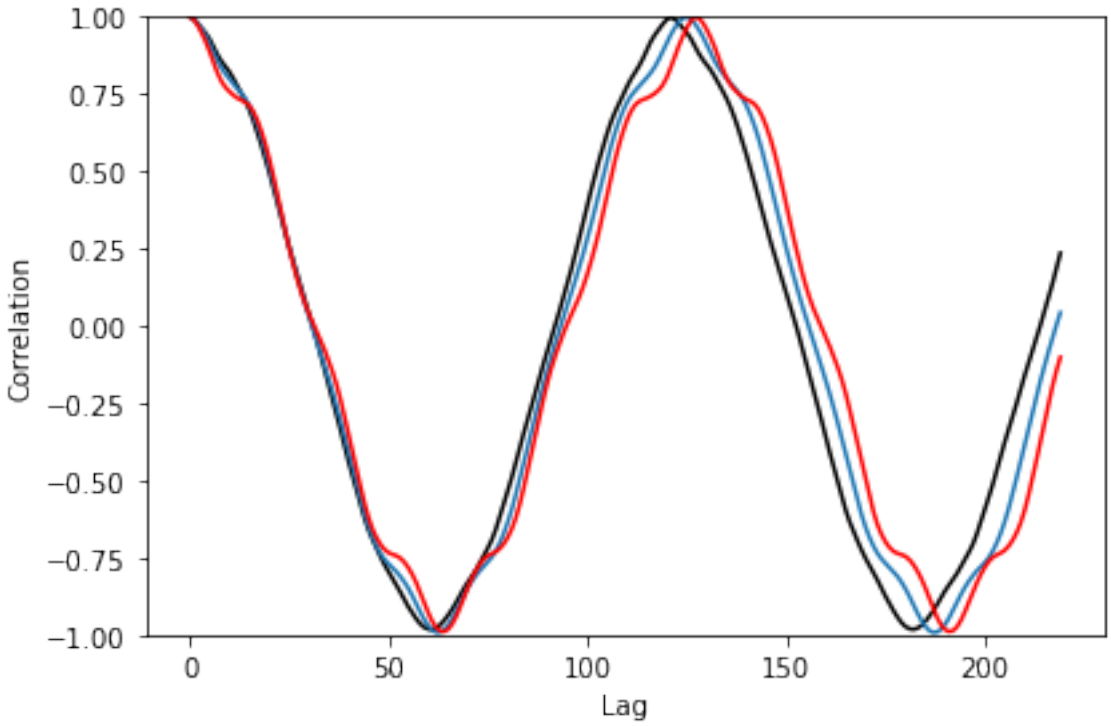

Рисунок 5.2. Автокорреляция сигналов

Вычислим lag

```
low = 50
high = 200
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
```

```
(121, 125, 127)
```

Вычислим периоды

```
period1 = lag1 / segment1.framerate
period2 = lag2 / segment2.framerate
period3 = lag3 / segment3.framerate
period1, period2, period3
```

```
(0.0027437641723356007, 0.002834467120181406, 0.0028798185941043084)
```

Соответствующие периодам частоты

```
frequency1 = 1 / period1
frequency2 = 1 / period2
```

```
frequency3 = 1 / period3
frequency1, frequency2, frequency3
```

```
(364.4628099173554, 352.8, 347.244094488189)
```

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.

Используем звук из предыдущего упражнения.

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

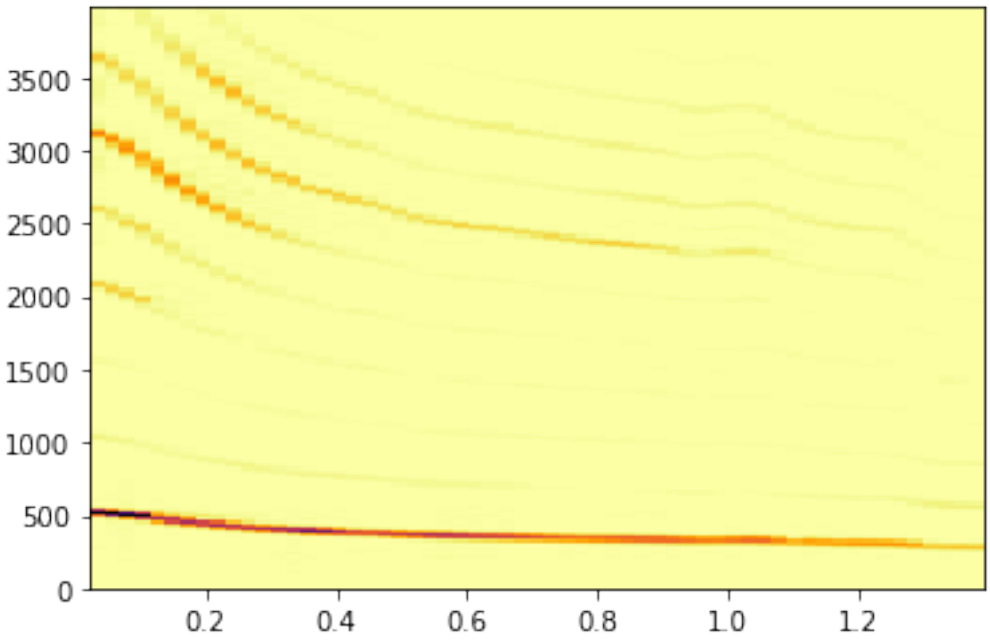

Рисунок 5.3. Спектрограмма звука

Объеденим весь код из предыдущего пункта в одну функцию

```
def estimate_fundamental(segment, low=50, high=200):
    lags, corrs = autocorr(segment)
    lag = np.array(corrs[low:high]).argmax() + low
    period = lag / segment.framerate
    frequency = 1 / period
    return frequency
```

```
estimate_fundamental(segment1)
```

```
364.4628099173554
```

Сделаем оценку высоты тона, применяя разделение на сегменты

```
duration = wave.duration
step = 0.02
start = 0
```

```
time = []
frequencys = []
while start + step < duration:
  time.append(start + step/2)
  frequencys.append(estimate_fundamental(wave.segment(start=start ,duration=
  start += step
wave.make_spectrogram(2048).plot(high=900)
plt.plot(time, frequencys, color='black')
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

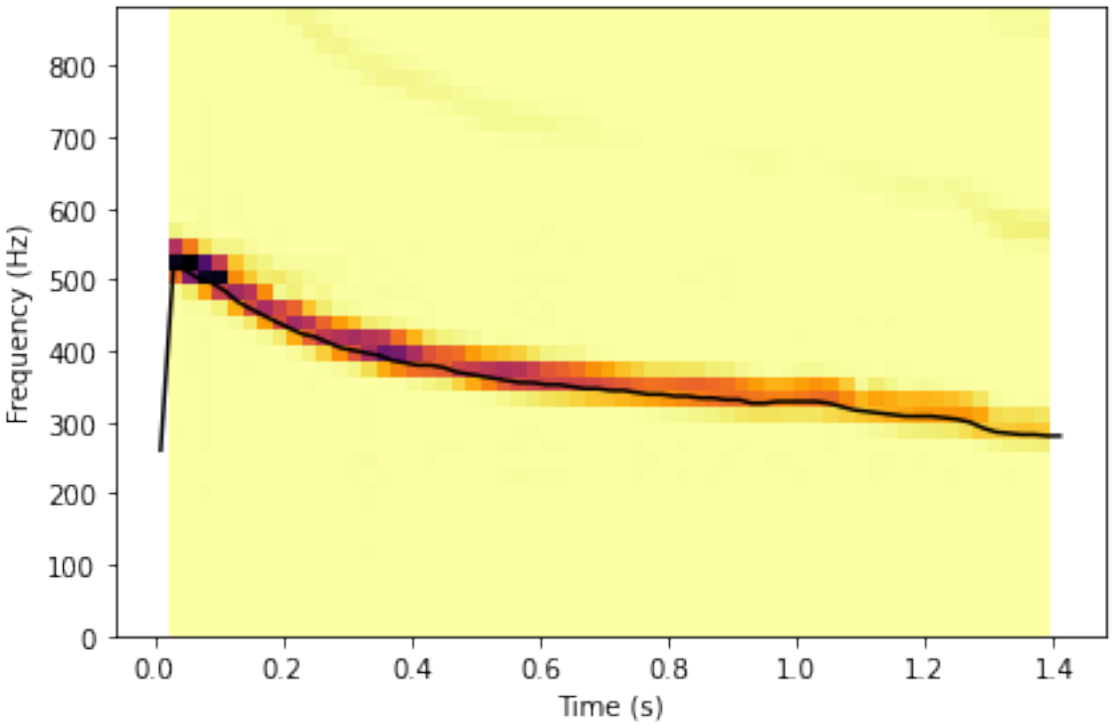

Рисунок 5.4. Результат оценки

5.3. Упражнение 3

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

```
if not os.path.exists('market-pric.csv'):
    !wget https://github.com/Eugenepolyt/telecomunaction/raw/main/market-pr

import pandas as pd
from thinkdsp import Wave

df = pd.read_csv('market-pric.csv', parse_dates=[0])
ys = df['market-price']
ts = df.index

w = Wave(ys, framerate=1)
w.plot()
```

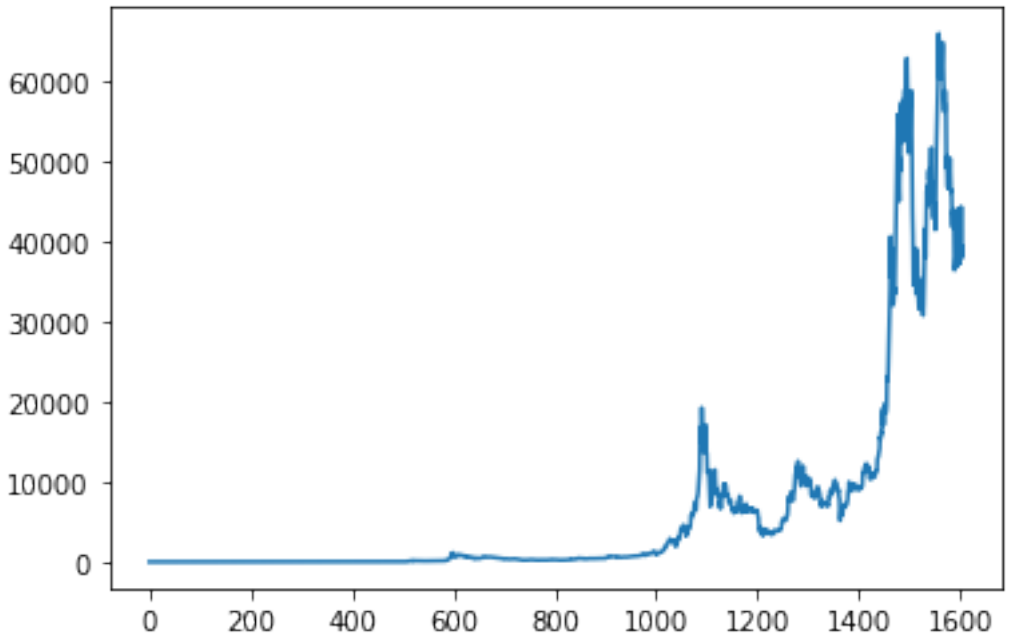

Рисунок 5.5. График цены на BitCoin

Вычислим автокорреляцию:

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

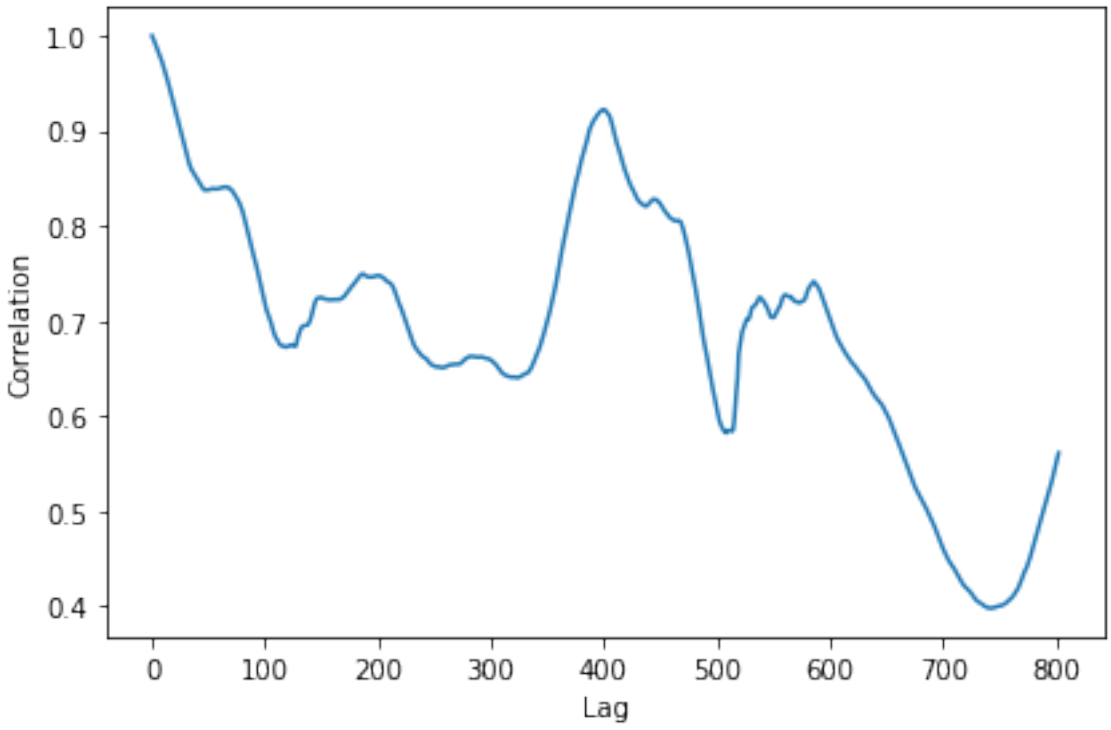

Рисунок 5.6. Автокорреляция функции цены на BitCoin

По графику видно, что есть резкие спады и повышения. Процесс может напоминать переодичность.

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.

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

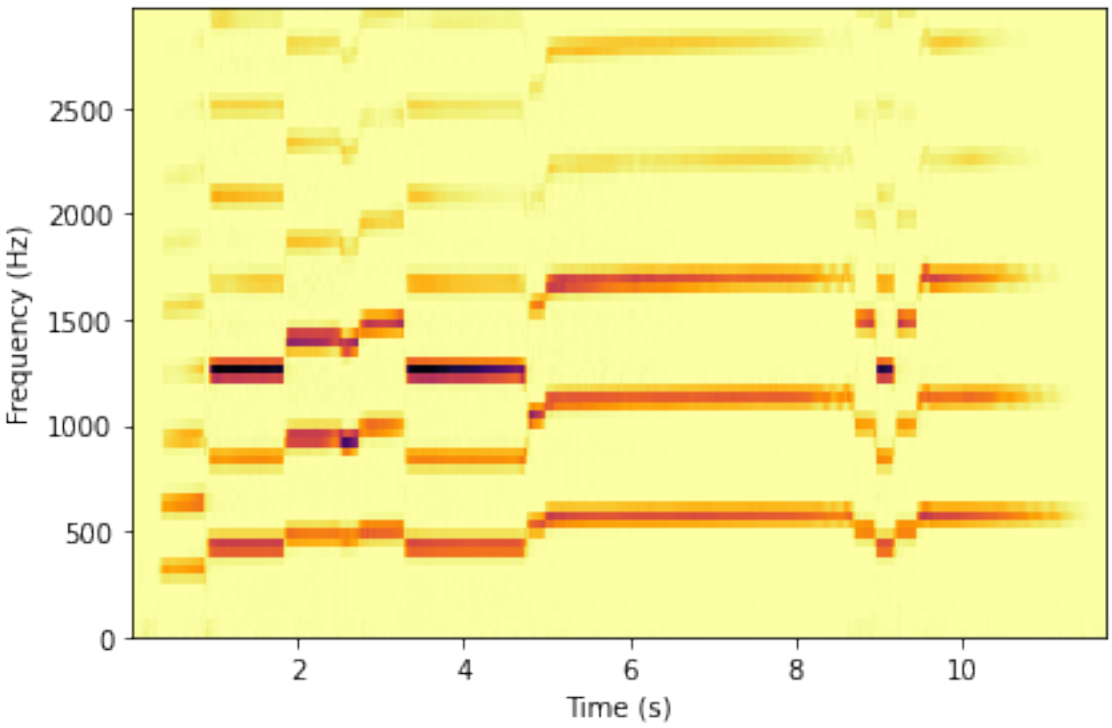

Рисунок 5.7. Спектрограмма сегмента

Используем функции из блокнтота из репозитория и используем их для другого сегмента

```
segment = wave.segment(start=4.0, duration=0.2)
segment.make_audio()
spectrum = segment.make_spectrum()
spectrum.plot(high=5000)
decorate(xlabel='Frequency', ylabel='Amplitude')
```

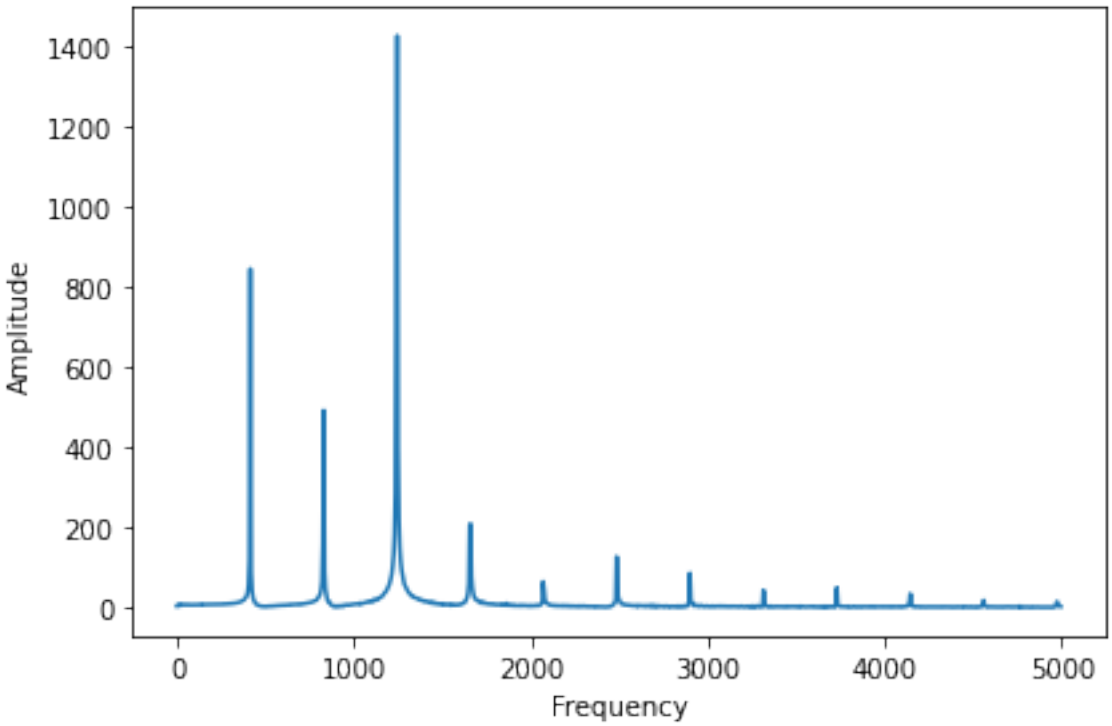

Рисунок 5.8. Спектр другого сегмента

Пики в спектре находятся на 1245, 415 и 830 Гц

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

```
[(1425.371205417228, 1245.0),
 (844.1565084866448, 415.0),
 (810.3146734198679, 1240.0),
 (491.1468807713408, 830.0),
 (395.0157320768441, 1250.0),
 (285.5428668623747, 1235.0),
 (220.80813321938248, 1255.0),
 (208.75420107735613, 1660.0),
 (205.643155157793, 1655.0),
 (180.59606616391875, 1230.0)]
```

Сравним наш сегмент с треугольным сигналом

```
from thinkdsp import TriangleSignal
TriangleSignal(freq=415).make_wave(duration=0.2).make_audio()
segment.make_audio()
```

У сигналов одинаковая воспринимаемая частота звука. Для понимания процесса восприятия основной частоты испольщуем АКФ.

```
def autocorr2(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 = autocorr2(segment)
plt.plot(corrs[:500])
```

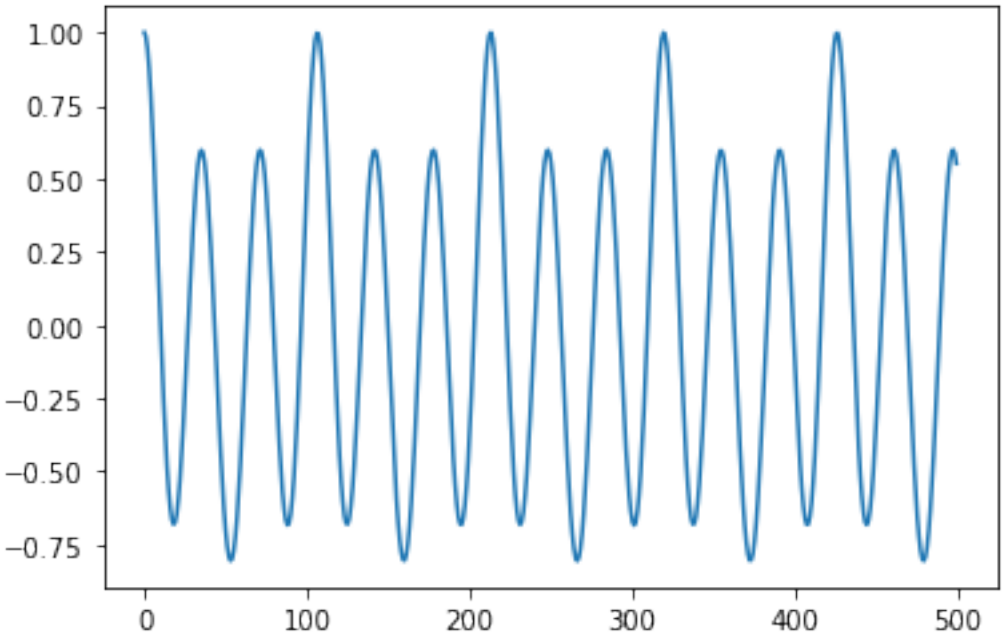

Рисунок 5.9. График после АКФ

Первый пик находился возле lag 100

Найдём основную частоту при помощи написанной ранее функции.

```
estimate_fundamental(segment)
```

```
416.0377358490566
```

Воспринимаемая высота тона не изменится, если мы полностью удалим основной тон. Вот как выглядит спектр, если мы используем фильтр верхних частот, чтобы стереть основные частоты

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

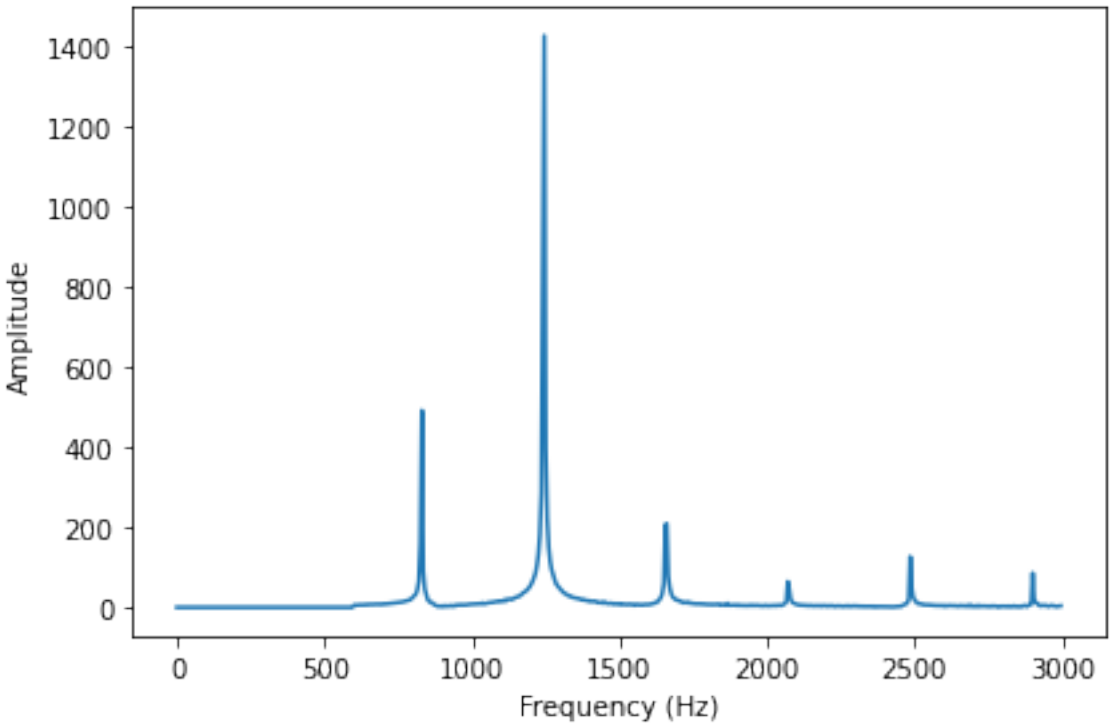

Рисунок 5.10. Спектр после ФНЧ

Это явление называется "отсутствующим фундаментом" Чтобы понять, почему мы слышим частоту, которой нет в сигнале, полезно взглянуть на функцию автокорреляции (ACF).

```
corrs = autocorr2(segment2)
plt.plot(corrs[:500])
```

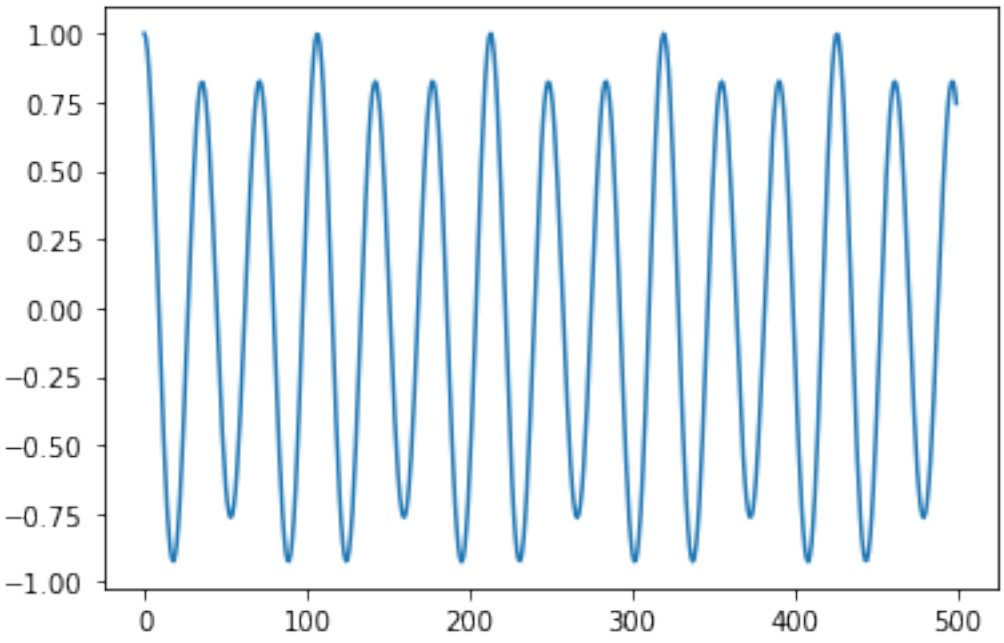

Рисунок 5.11. Полученный график

```
estimate_fundamental(segment)
```

```
416.0377358490566
```

Таким образом, эти эксперименты показывают, что восприятие высоты тона основано не только на спектральном анализе, но и на чем-то вроде автокорреляции.

5.5. Вывод

В данной работе мы изучили корреляцию и как она влияет на сигналы, также был написан код для сигнала с "отсутствующим фунтаментом".

6. Дискретное косинусное преобразование

6.1. Упражнение 1

Убедимся, что analyze1 требует времени пропорционально $n^3, analyze2 n^2 c$

```
import numpy as np
PI2 = np.pi * 2
def analyze1(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.linalg.solve(M, ys)
    return amps
def analyze2(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = M.dot(ys) / 2
    return amps
```

Возьмём размеры массива как степени 2.

```
ns = 2 ** np.arange(5,10)

best_analyze1 = []
for n in ns:
    ts = (0.5 + np.arange(n)) / n
    freqs = (0.5 + np.arange(n)) / 2
    ys = wave.ys[:n]
    best = %timeit -r1 -o analyze1(ys,freqs,ts)
    best_analyze1.append(best.best)
best_analyze2 = []
for n in ns:
    ts = (0.5 + np.arange(n)) / n
    freqs = (0.5 + np.arange(n)) / 2
    ys = wave.ys[:n]
    best = %timeit -r1 -o analyze2(ys,freqs,ts)
    best_analyze2.append(best.best)
best_dct = []
for n in ns:
    ys = wave.ys[:n]
    best = %timeit -r1 -o scipy.fftpack.dct(ys, type=3)
    best_dct.append(best.best)
plt.plot(ns, best_analyze1, label='analyze1')
plt.plot(ns, best_analyze2, label='analyze2')
plt.plot(ns, best_dct, label='fftpack.dct')
loglog = dict(xscale='log', yscale='log')
decorate(xlabel='Wave length (N)', ylabel='Time (s)', **loglog)
```

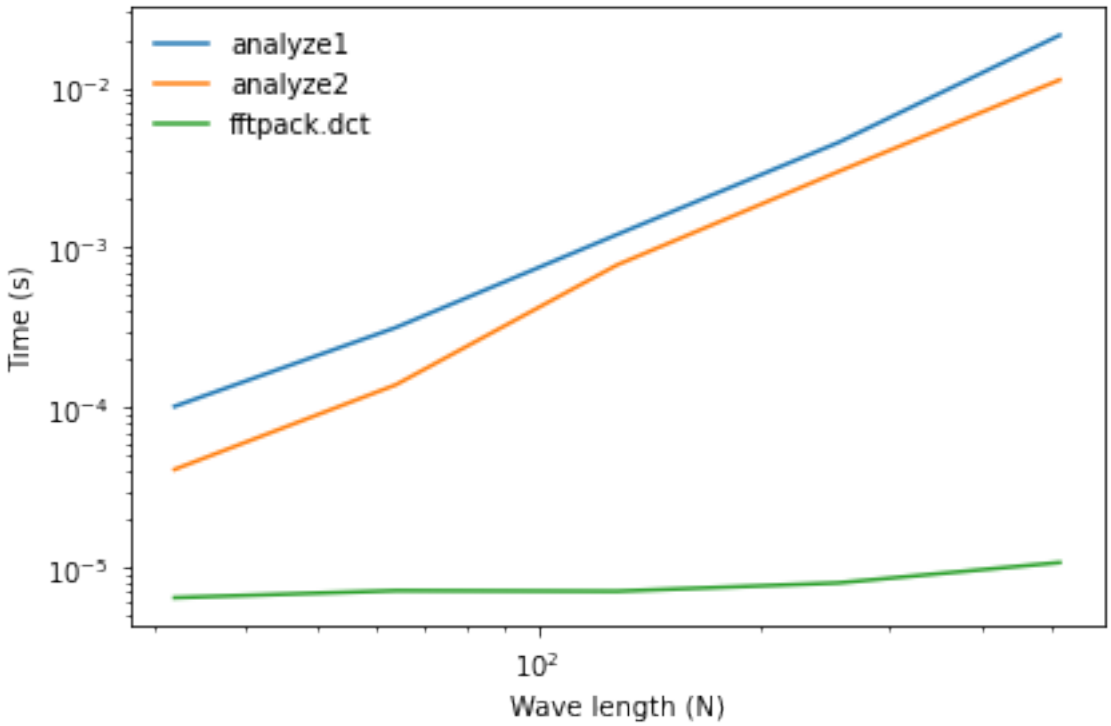

Рисунок 6.1. Время работы различных методов ДКП

Не смотря на теоритическое время исполнения, время analyze1 получилсось пропорциональным n2 .

6.2. Упражнение 2

Реализуем алгоритм ДКП и применим его для записи звуков или речи.

Возьмем звук пианино из первой лабораторной работы

```
if not os.path.exists('jcveliz__violin-origional.wav'):
    !wget https://github.com/Eugenepolyt/telecomunaction/raw/main/jcveliz__
```

Выделим сегмент:

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

DCT график для полученного сегмента

```
dct = segment.make_dct()
dct.plot(high = 5000)
```

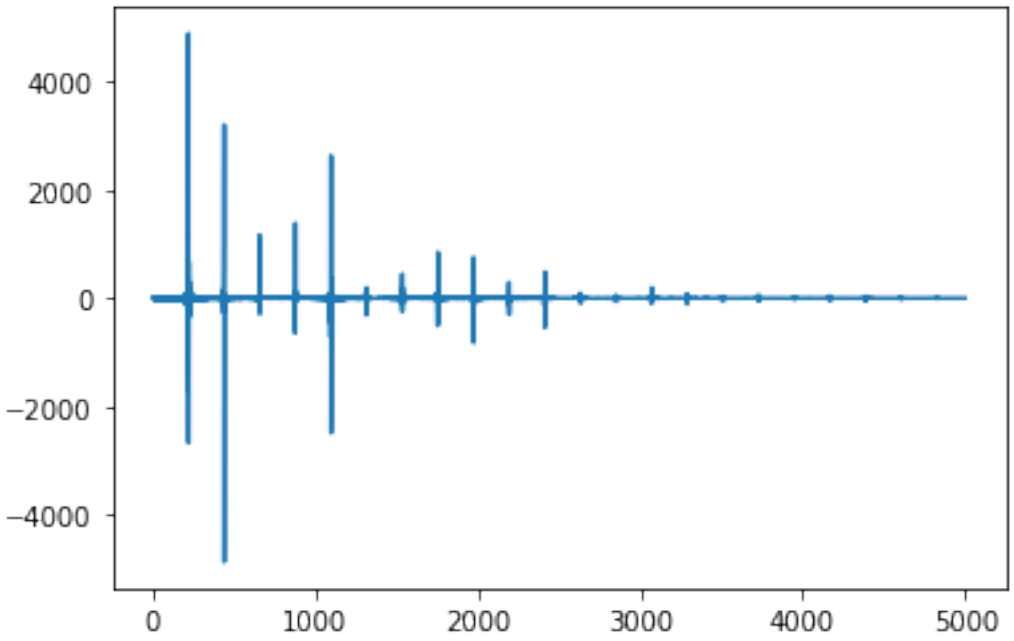

Рисунок 6.2. DCT график

Следующая функция принимает dct и устанавливает все элементы ниже порога "trash" в ноль

```
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')
```

Применим функцию для нашего сегмента

```
seg_dct = segment.make_dct()
compress(seg_dct, thresh=100)
seg_dct.plot(high=4000)
```

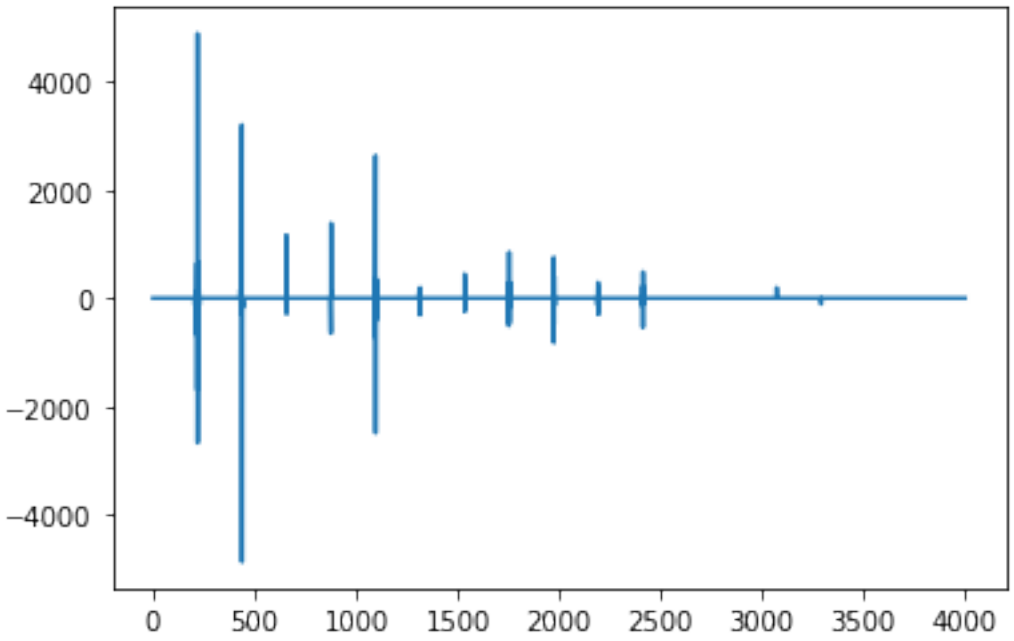

Рисунок 6.3. DCT после фильтрации

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.segment(start=0.02,duration=0.01).plot()
decorate(xlabel='Time (s)')
```

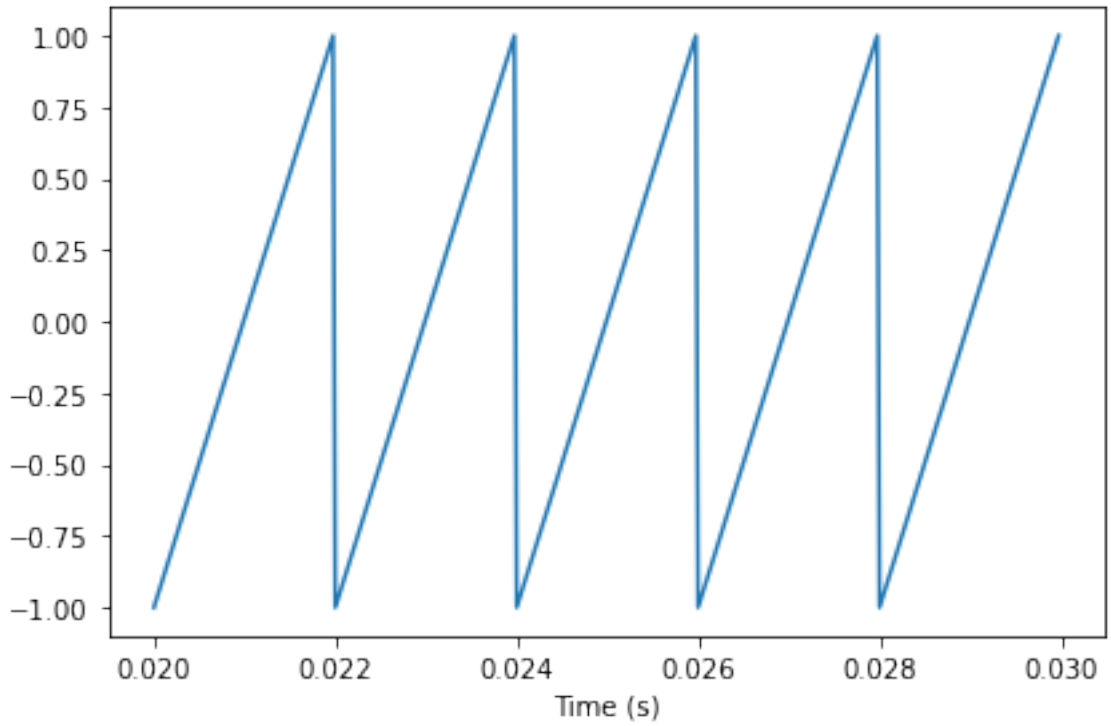

Рисунок 6.4. График сегмента

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

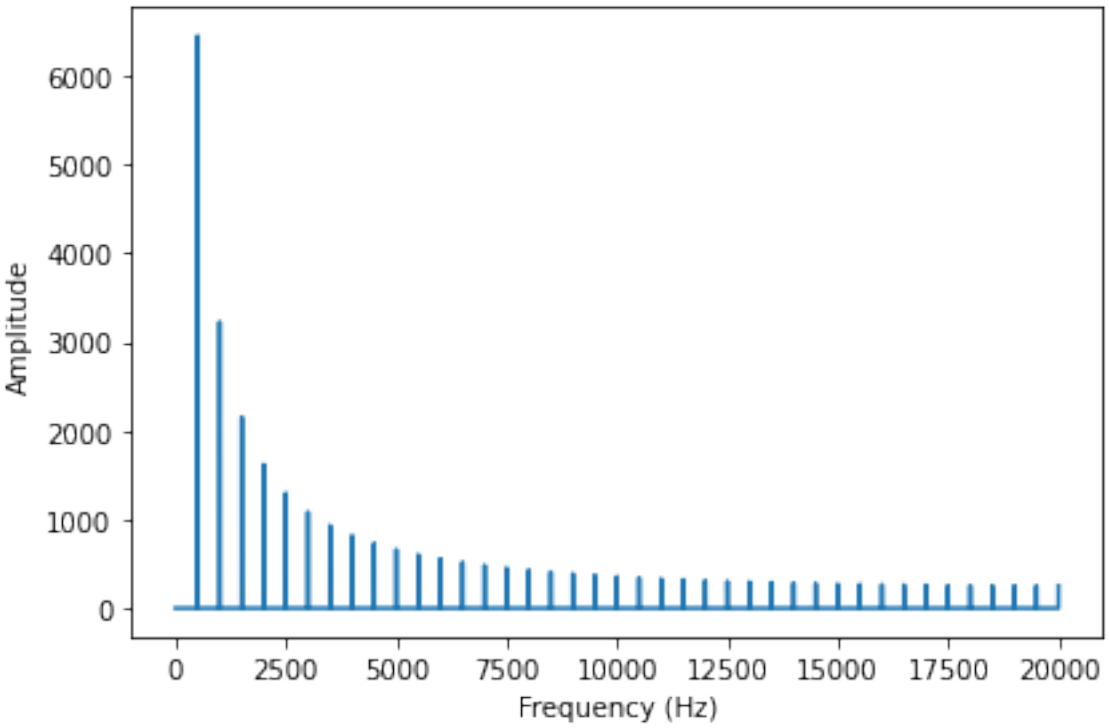

Рисунок 6.5. Спектр сегмента

```
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)
plot_angle(spectrum, thresh=1)
def plot_three(spectrum, thresh=1):
    """Plot amplitude, phase, and waveform.

    spectrum: Spectrum object
    thresh: threshold passed to plot_angle
    """
    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)
```

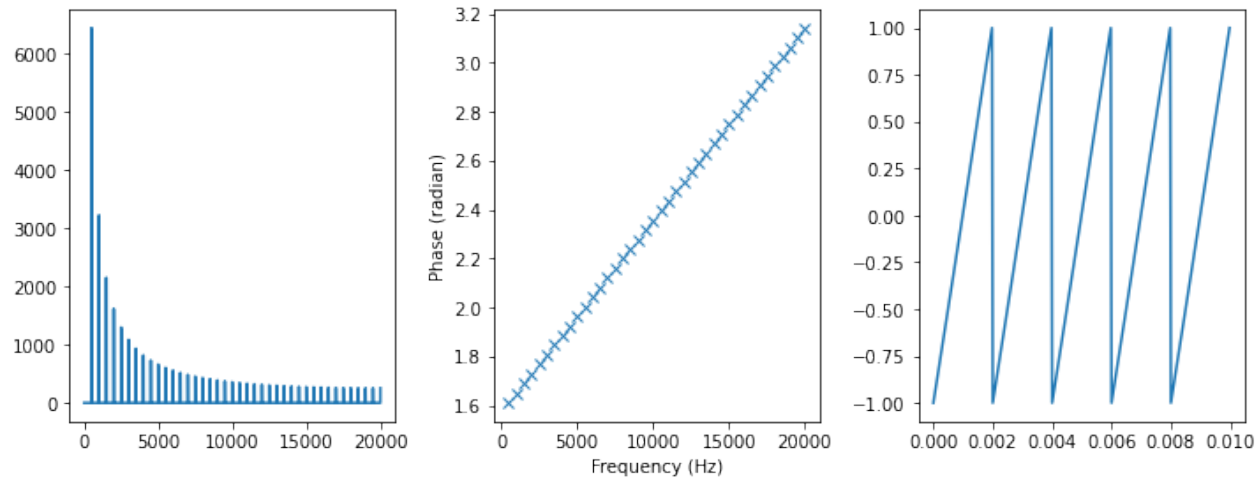

Рисунок 6.6. Результат работы функции

```
def zero_angle(spectrum):
    res = spectrum.copy()
    res.hs = res.amps
    return res
spectrum2 = zero_angle(spectrum)
plot_three(spectrum2)
```

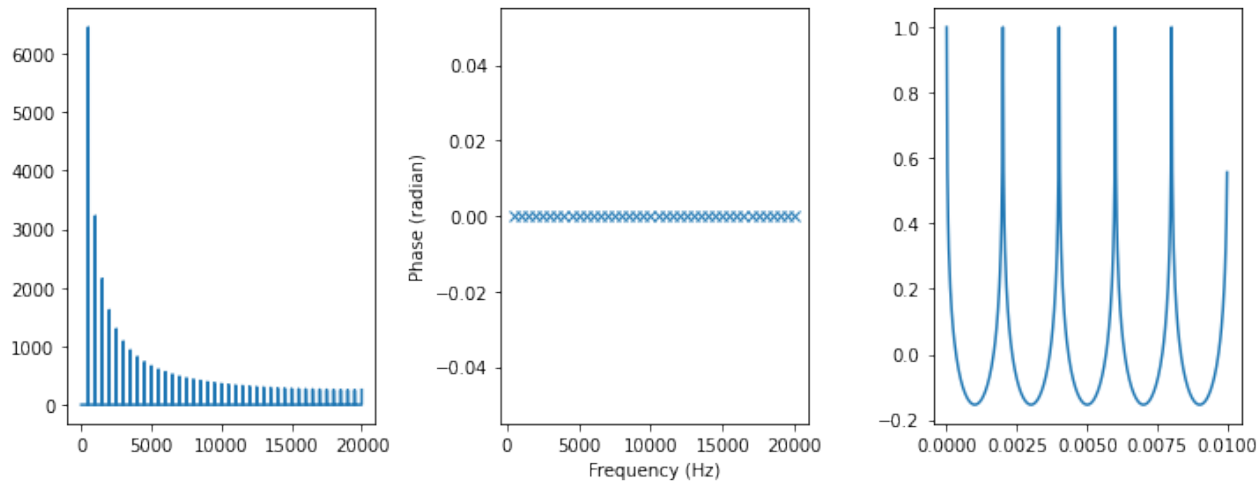

Рисунок 6.7. Результат работы функции

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

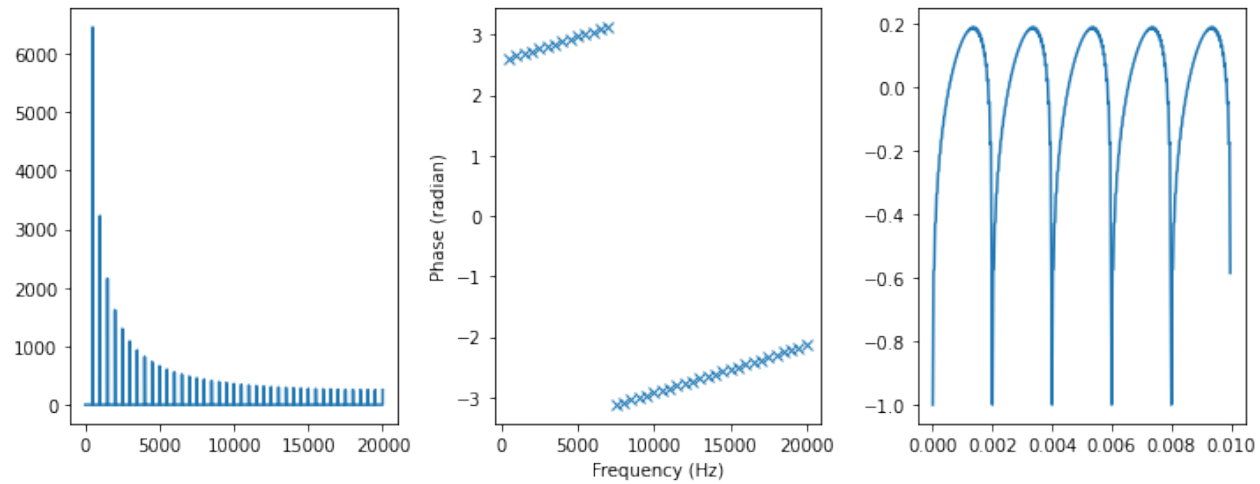

Рисунок 6.8. Результат работы функции

```
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)
```

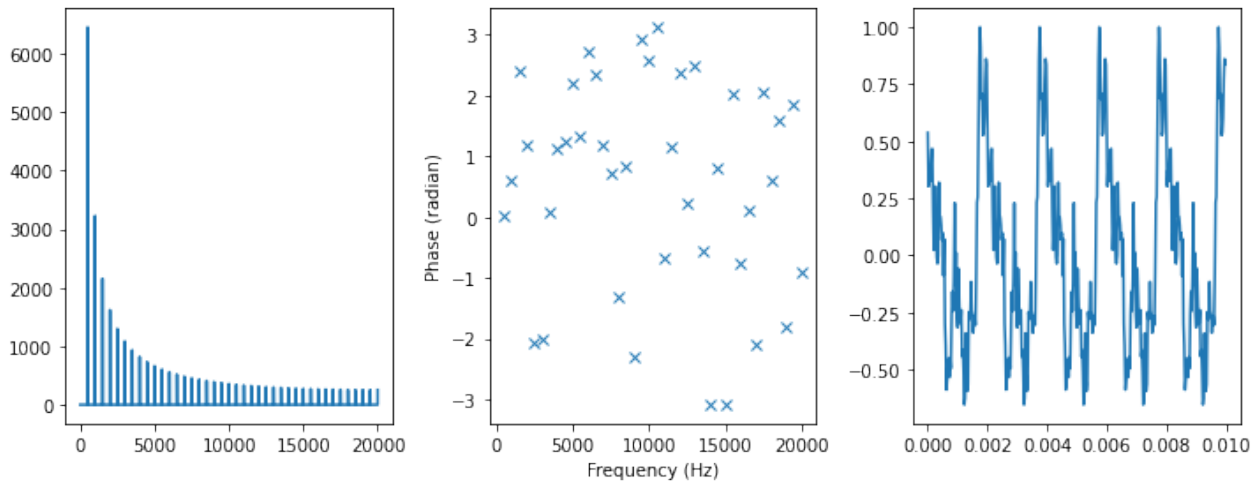

Рисунок 6.9. Результат работы функции

Звук звучит также, как первоначальный, возьмем другой звук.

```
if not os.path.exists('120994__thirsk__120-oboe.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/120994__t
from thinkdsp import read_wave
wave = read_wave('120994__thirsk__120-oboe.wav')
wave.make_audio()
segment = wave.segment(start=0.2, duration=0.5)
spectrum = segment.make_spectrum()

plot_three(spectrum)
```

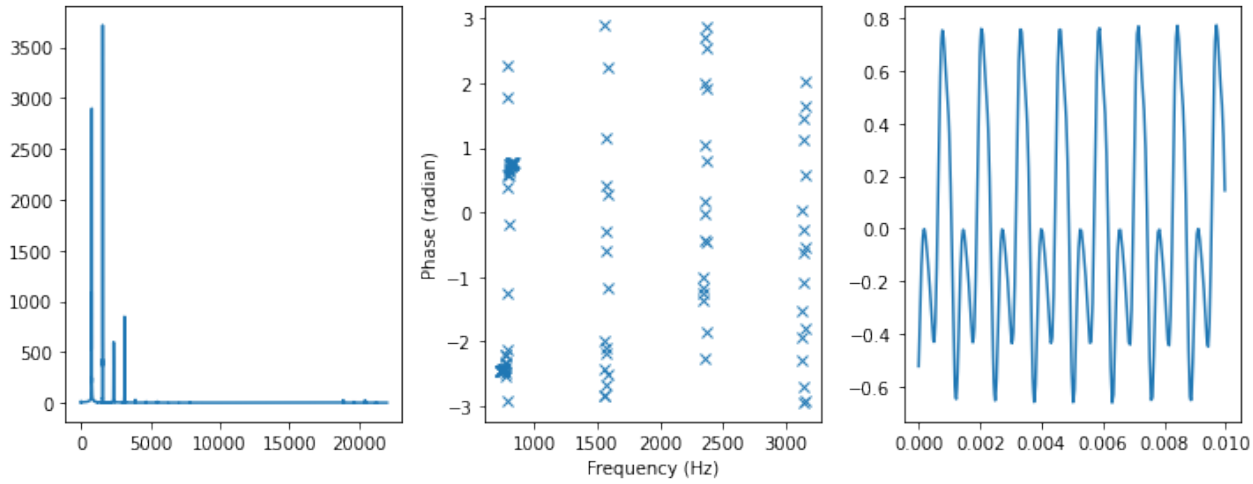

Рисунок 6.10. Результат работы функции

```
spectrum2 = zero_angle(spectrum)
plot_three(spectrum2, thresh=50)
```

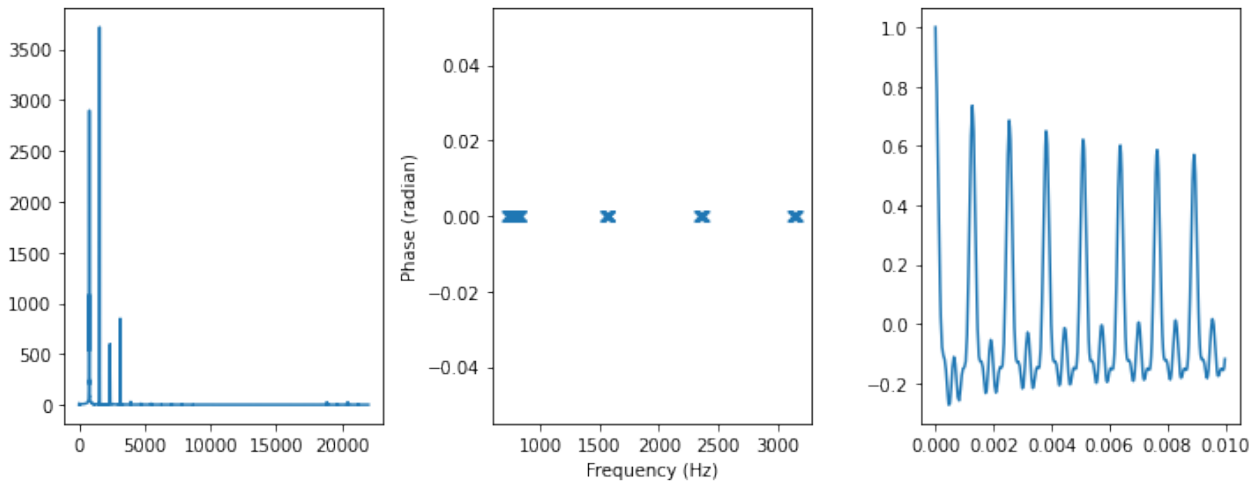

Рисунок 6.11. Результат работы функции

Звук стал более глухой

```
spectrum3 = rotate_angle(spectrum, 1)
plot_three(spectrum3, thresh=50)
```

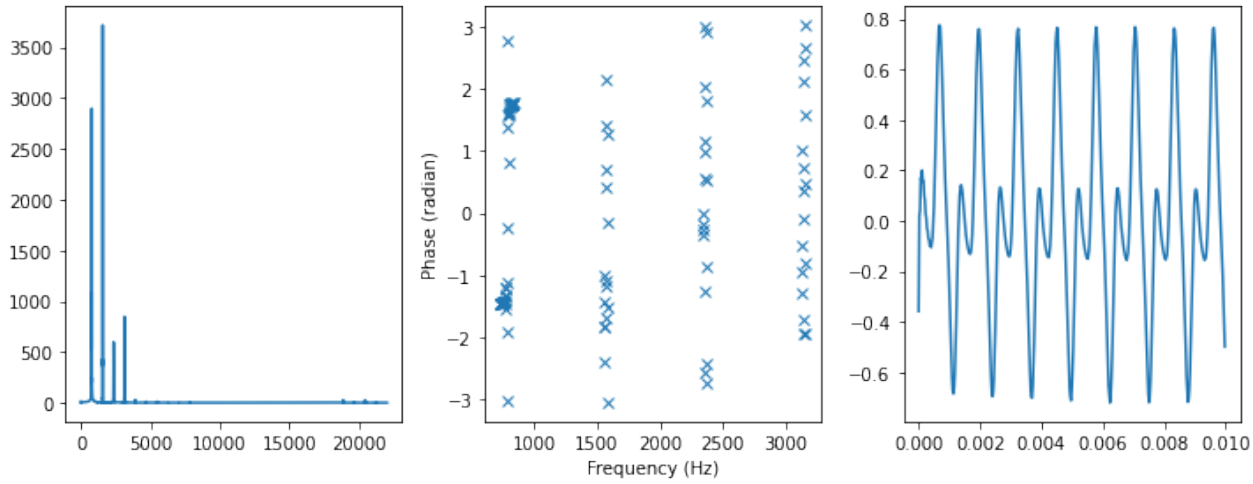

Рисунок 6.12. Результат работы функции

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

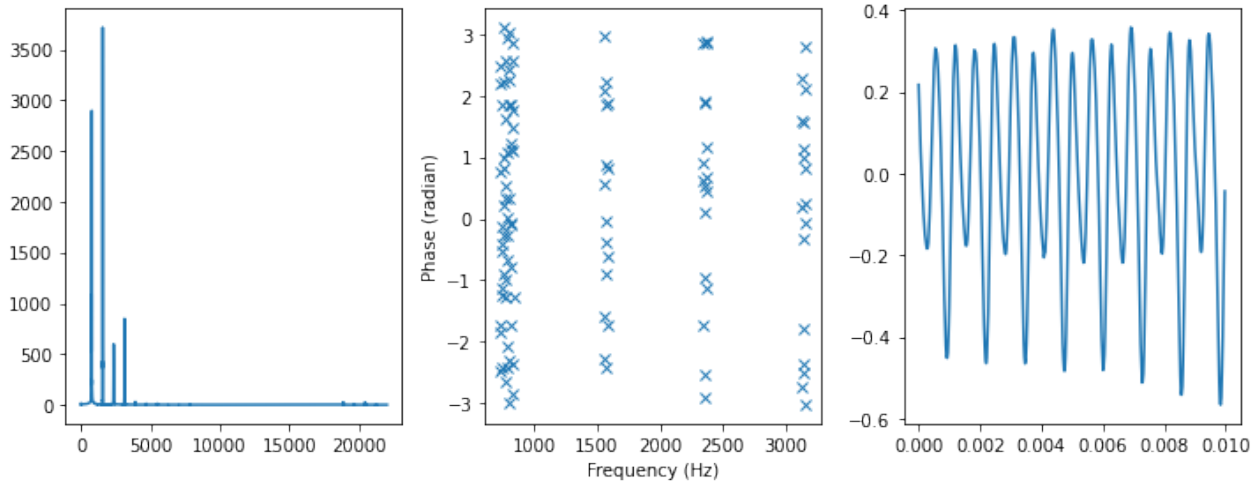

Рисунок 6.13. Результат работы функции

В данном случае звук уже отличается от изначального, но не сильно. Мы не слышим особо изменения в фазовой структуре, если гармоническая структура не изменялась.

6.4. Вывод

В данной лабораторной работе был рассмотрен ДКП , который применяется в различных форматах сжатия музыки. С помощью ДКП были рассмотрены свойства звуков с разной структурой

7. Дискретное преобразование Фурье

7.1. Упражнение 1

Необходимо реализовать алгоритм БПФ. Для этого разделим массив сигнала на четные и нечетные элементы и затем вычислить ДФТ для обоих групп. Также используем лемму Дэниеолсона-Ланцоша.

Для тестирования возьмем небольшой массив сигнала

```
ys = [0.8, 0.7, -0.5, 0.5]
hs = np.fft.fft(ys)
hs
```

```
array([ 1.5+0.j ,  1.3-0.2j, -0.9+0.j ,  1.3+0.2j])
```

Применим ДФТ функцию, которая представлена в предыдущем пункте, то есть в блокноте к главе книги.

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

```
7.249538831146999e-16
```

Теперь реализуем БПФ без рекурсии при помощи разделения элементов.

```
def fft(ys):
    He = dft(ys[::2])
    Ho = dft(ys[1::2])
    ns = np.arange(len(ys))
    W = np.exp(-1j * PI2 * ns / len(ys))
    return np.tile(He, 2) + W * np.tile(Ho, 2)

fft(ys)
```

```
array([ 1.5+0.j ,  1.3-0.2j, -0.9-0.j ,  1.3+0.2j])
```

Теперь реализуем вариант алгоритма с рекурсией

```
def fft_rec(ys):
    if len(ys) == 1:
        return ys
    He = fft_rec(ys[::2])
    Ho = fft_rec(ys[1::2])
    ns = np.arange(len(ys))
    W = np.exp(-1j * PI2 * ns / len(ys))
    return np.tile(He, 2) + W * np.tile(Ho, 2)

fft_rec(ys)
```

```
array([ 1.5+0.j ,  1.3-0.2j, -0.9-0.j ,  1.3+0.2j])
```

7.2. Вывод

В данной лабораторной работе был реализован алгоритм БПФ и протестирован, алгоритм работает в точности также, как и библиотечная функция.

8. Фильтрация и свертка

8.1. Упражнение 1

Что случится, если при увеличении std не менять M

```
slider = widgets.IntSlider(min=2, max=100, value=11)
slider2 = widgets.FloatSlider(min=0, max=20, value=2)
interact(plot_filter, M=slider, std=slider2);
```

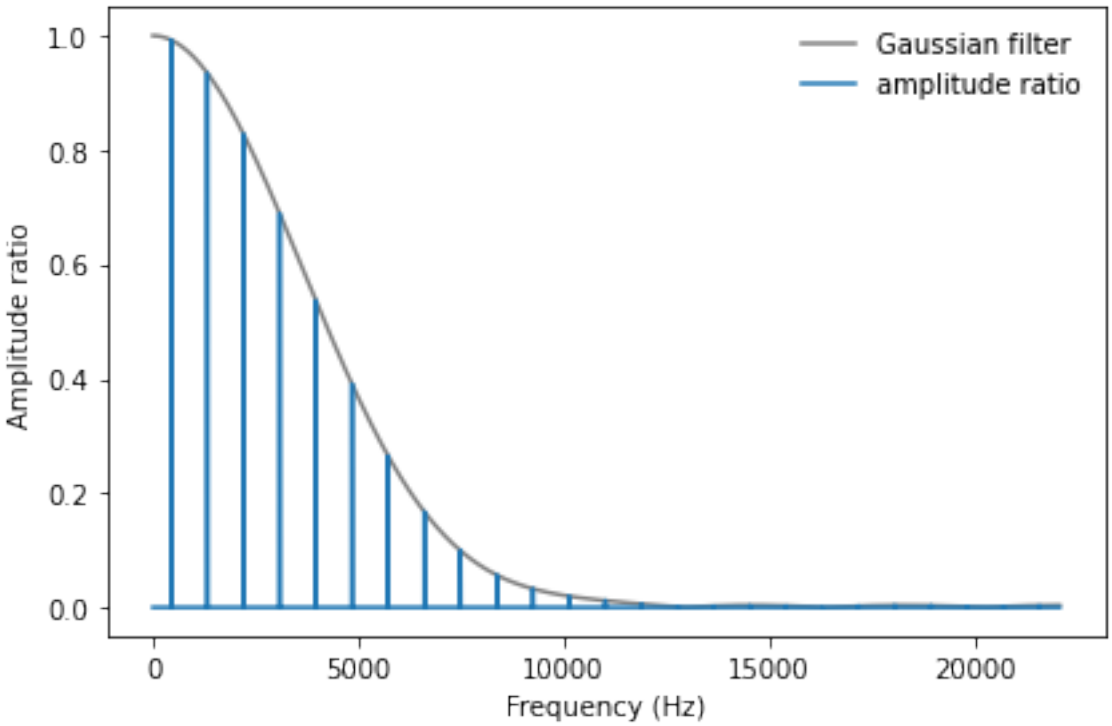

Рисунок 8.1. Гауссово окно для фильтрации

```
gaussian = scipy.signal.gaussian(M=11, std=11)
gaussian /= sum(gaussian)

plt.plot(gaussian, label='Gaussian')
decorate(xlabel='Index')
```

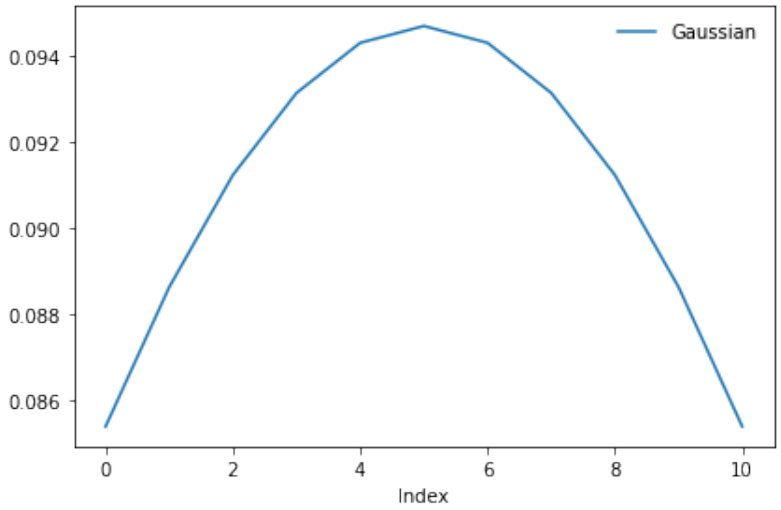

Рисунок 8.2. Гауссово окно

```
gaussian = scipy.signal.gaussian(M=11, std=1000)
gaussian /= sum(gaussian)

plt.plot(gaussian, label='Gaussian')
decorate(xlabel='Index')
```

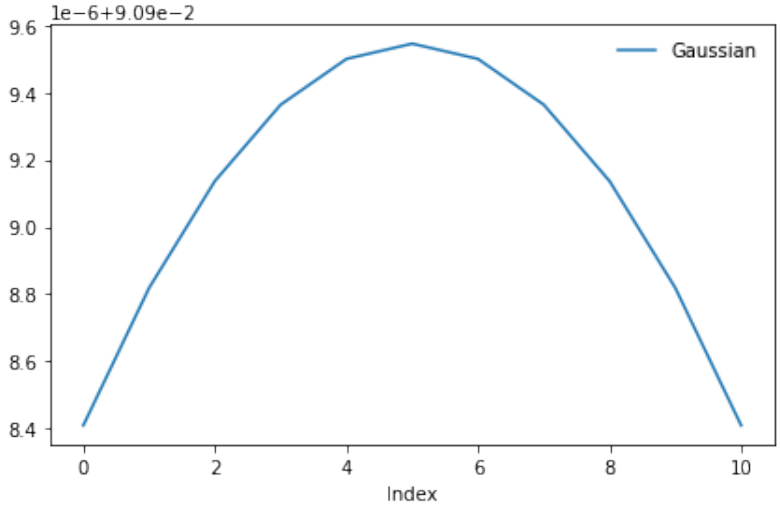

Рисунок 8.3. Гауссово окно

По графикам видно что кривая стала шире а БПФ меньше.

8.2. Упражнение 2

Выясним что происходит с преобразованием Фурье, если меняется std.

```
gaussian = scipy.signal.gaussian(M=16, std=2)
gaussian /= sum(gaussian)

plt.plot(gaussian, label='Gaussian')
```

```
decorate(xlabel='Index')
```

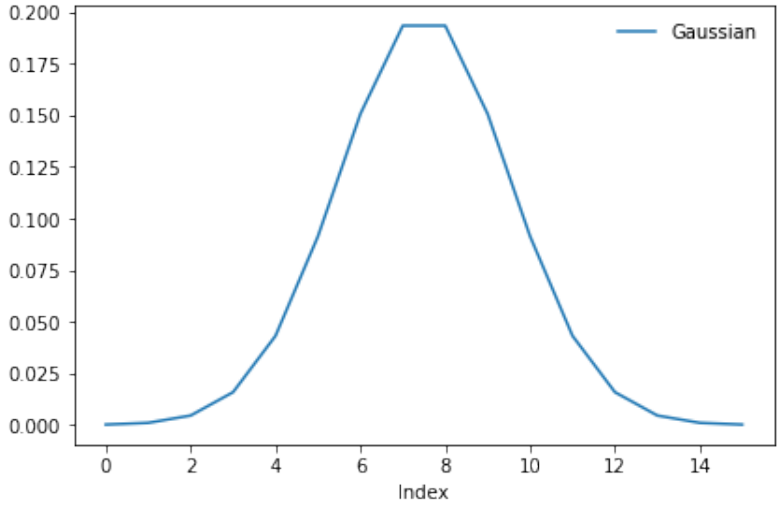

Рисунок 8.4. Гауссово окно

```
gaussian_fft = np.fft.fft(gaussian)
plt.plot(abs(gaussian_fft), label='Gaussian')
```

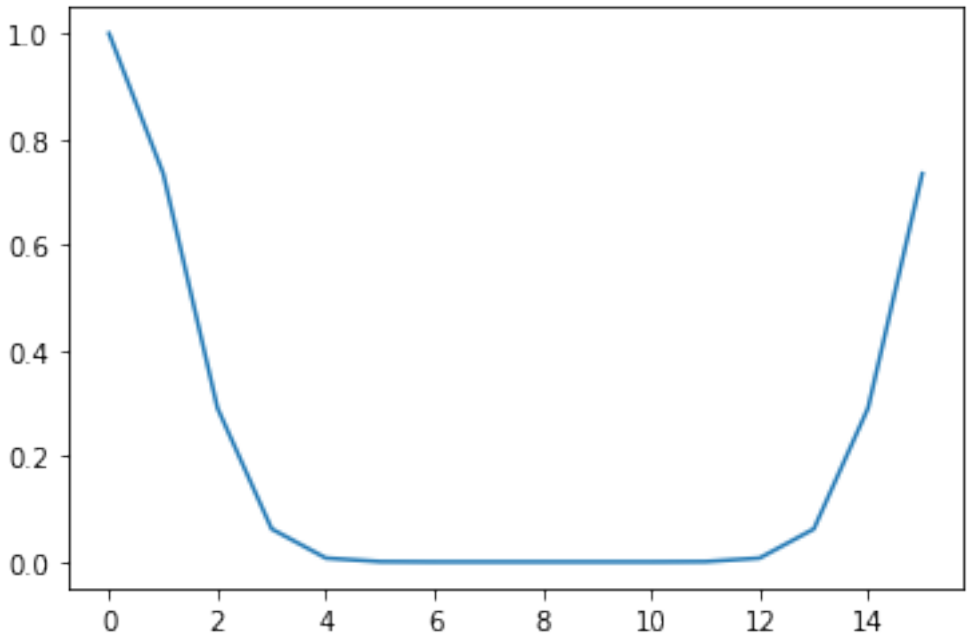

Рисунок 8.5. FTT применённое на окно

Сделаем свертку отрицательных частот влево.

```
gaussian_fft_rolled = np.roll(gaussian_fft, len(gaussian) // 2)
plt.plot(abs(gaussian_fft_rolled), label='Gaussian')
```

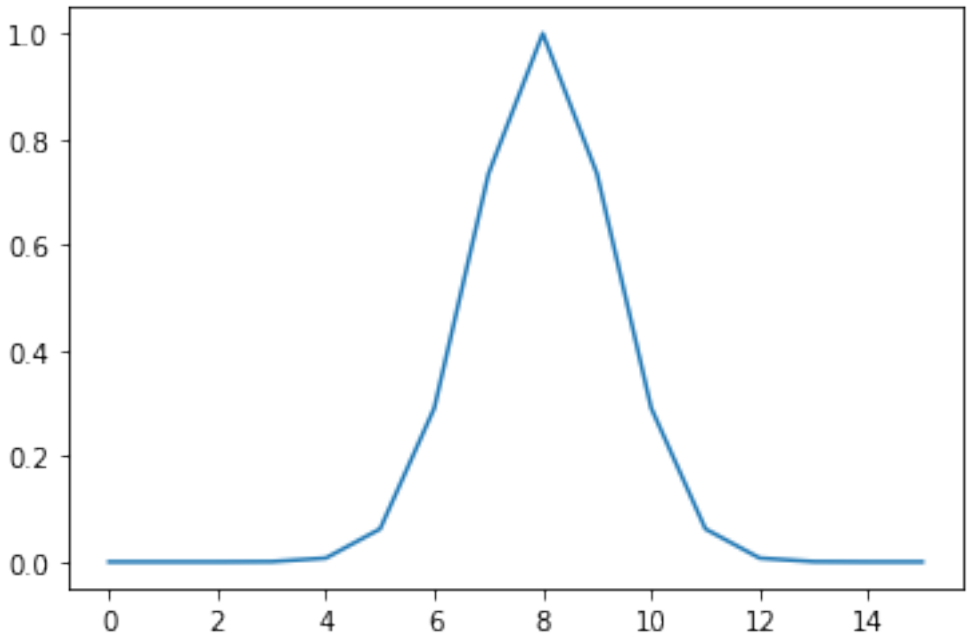

Рисунок 8.6. Результат свертки

При увеличении std гауссовой кривой, преобразование Фурье становится уже.

8.3. Упражнение 3

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

Создадим сигнал длительностью 1 секунда для тестирования.

```
import thinkdsp
sig = thinkdsp.TriangleSignal(freq=440)
wave = sig.make_wave(duration=1.0, framerate=44100)
wave.make_audio()

sig.plot()
```

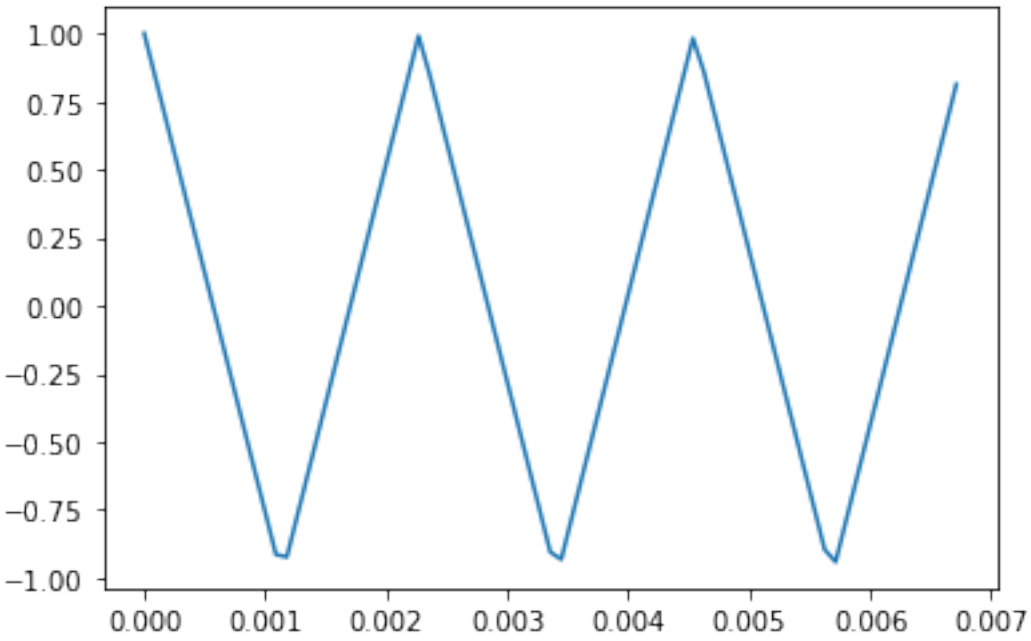

Рисунок 8.7. Пилообразный сигнал

Создадим различные окна

```
M = 16
std = 2

g = scipy.signal.gaussian(M, std)
br = np.bartlett(M)
bl = np.blackman(M)
hm = np.hamming(M)
hn = np.hanning(M)

array = [gaussian, bartlett, blackman, hamming, hanning]
labels = ['gauss', 'barlett', 'blackman', 'hamming', 'hanning']

for elem, label in zip(array, labels):
    elem /= sum(elem)
    plt.plot(elem, label=label)
plt.legend()
```

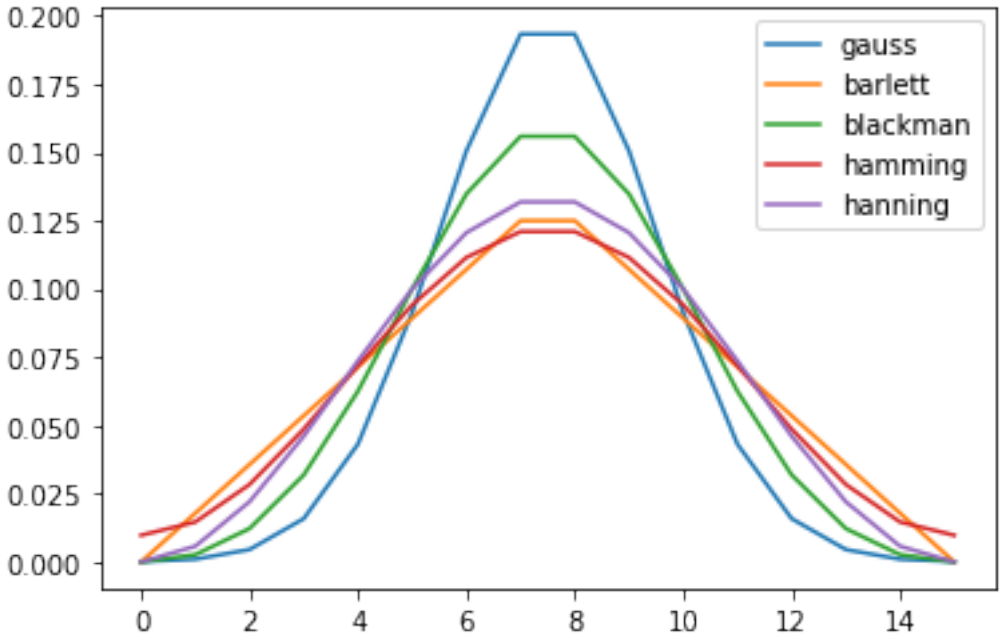

Рисунок 8.8. Применение различных окон на выбранный сигнал

Дополним окна нулями и выведем ДПФ:

```
for elem, label in zip(array, labels):
    padded = zero_pad(elem, len(wave))
    dft_window = np.fft.rfft(padded)
    plt.plot(abs(dft_window), label=label)
plt.legend()
```

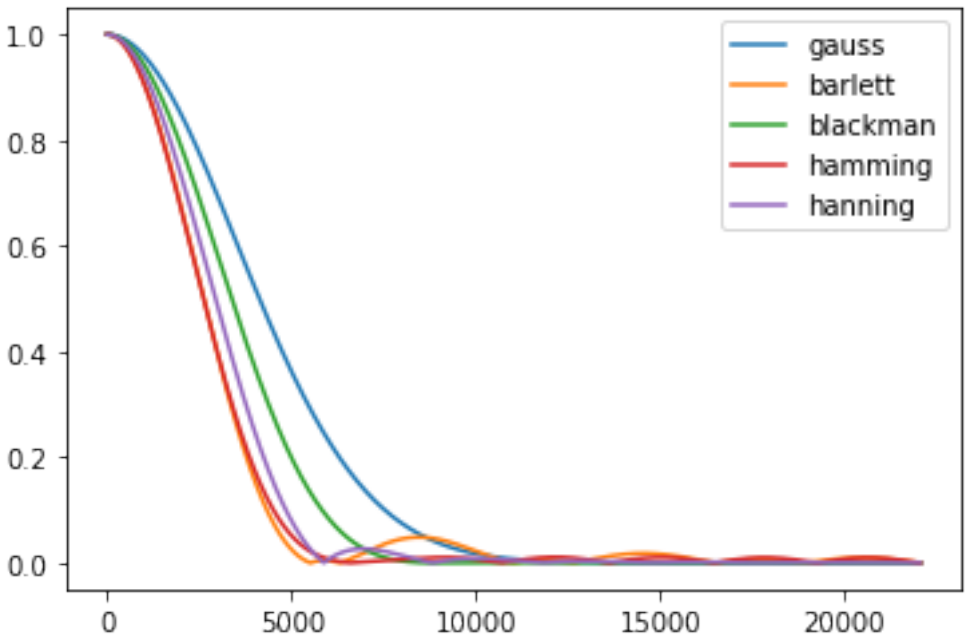

Рисунок 8.9. Применение различных окон на выбранный сигнал

Изменим маштаб.

```
for elem, label in zip(array, labels):
    padded =  zero_pad(elem, len(wave))
    dft_window = np.fft.rfft(padded)
    plt.plot(abs(dft_window), label=label)
plt.legend()
decorate(yscale='log')
```

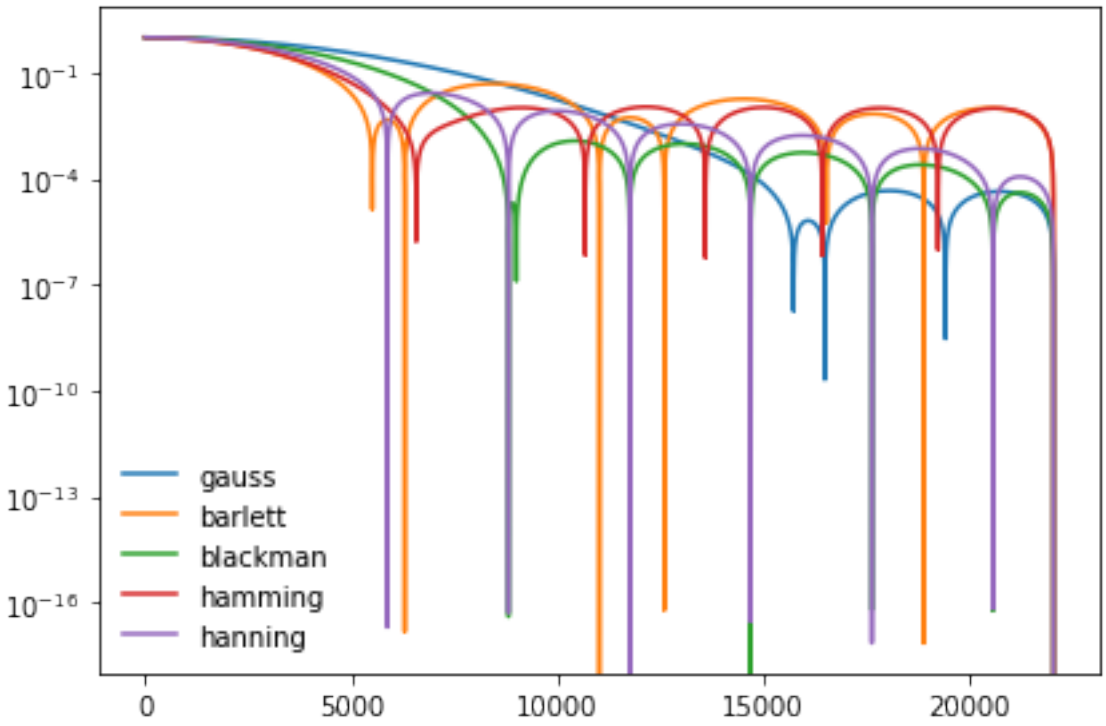

Рисунок 8.10. Логорифмический масштаб

Кажется, что хэнинг лучше всего подоёдет для фильтрации НЧ.

8.4. Вывод

В данной работе были рассмотренны операции фильстрации , сглаживания и свертки. Каждая функция может быть полезной для какой-либо определенной задачи, сглаживание, например, удаляет быстрые изменения сигнала для выявления общих особенностей.

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?

```
from thinkdsp import TriangleSignal

wave = TriangleSignal(freq=440).make_wave(duration=0.01, framerate=44100)
wave.plot()
decorate(xlabel='Time (s)')
```

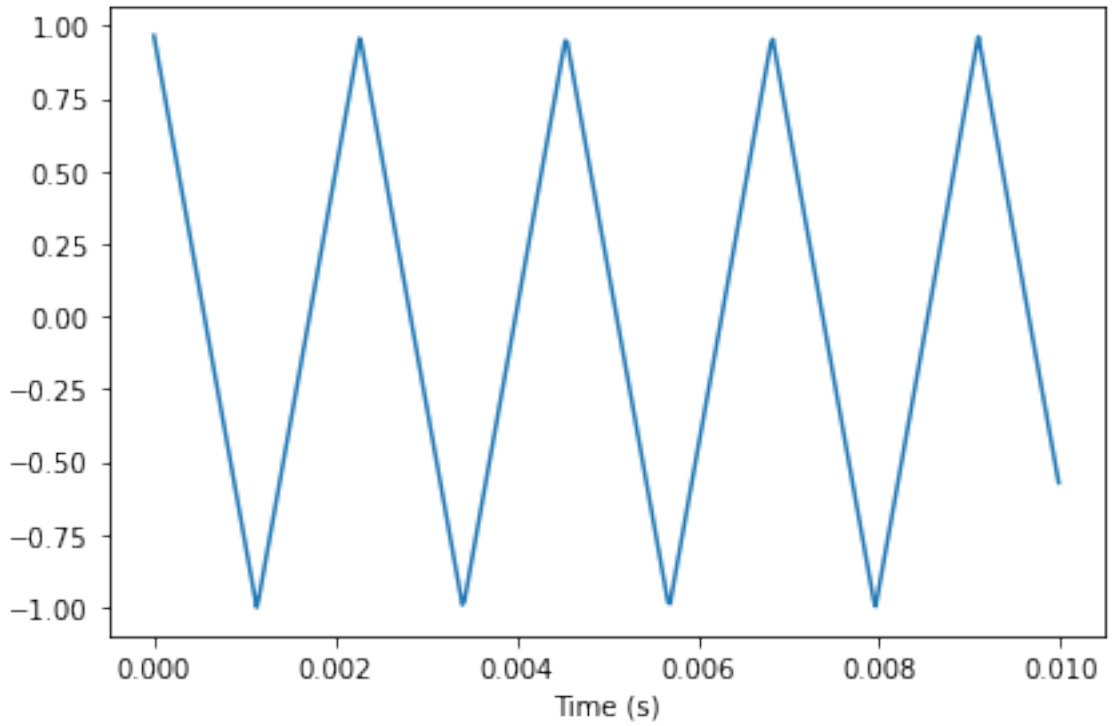

Рисунок 9.1. График треугольного сигнала

```
diff_wave = wave.diff()
diff_wave.plot()
decorate(xlabel='Time (s)')
```

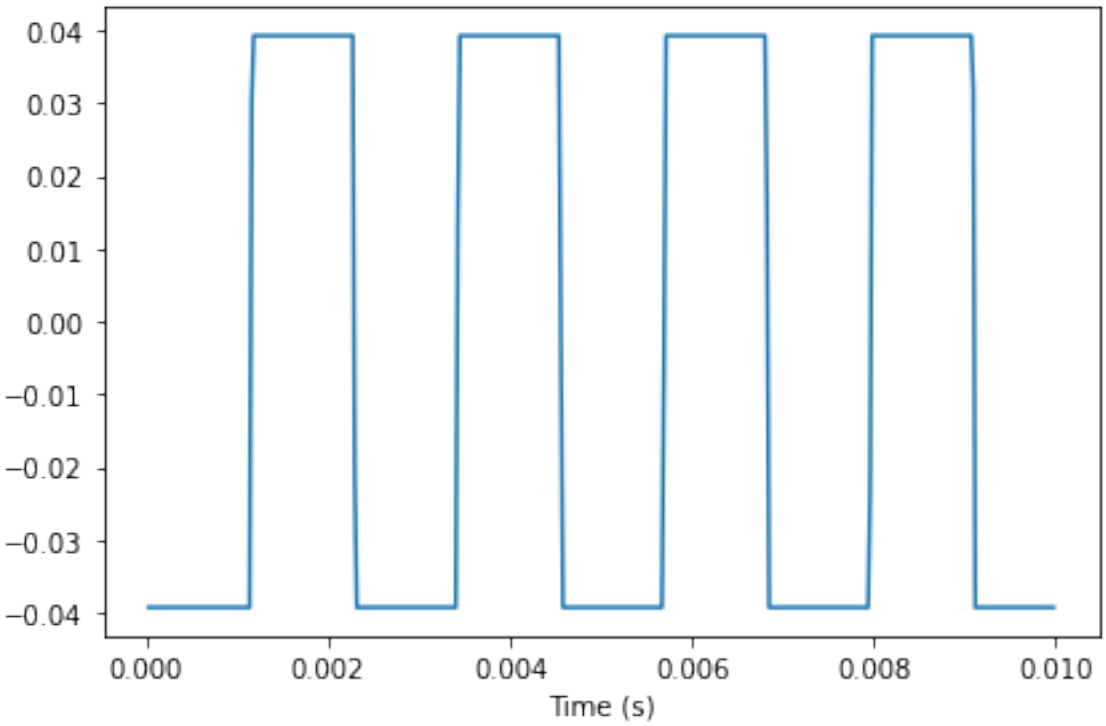

Рисунок 9.2. Сигнал после применения diff

Получился прямоугольны сигнал с такой же частотой

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

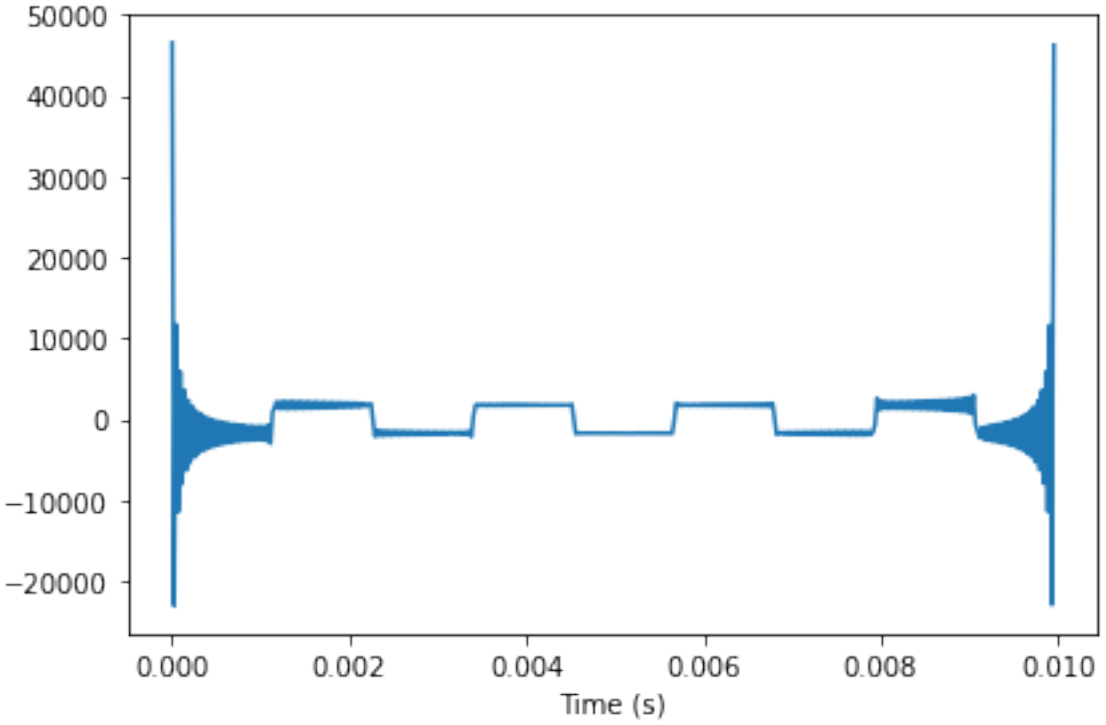

Рисунок 9.3. Сигнал после применения differentiate

Вначале и в конце шум, вероятнее всего это связано с тем, что невозможно вычислить производную.

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?

Изучим влияние cumsum и integrate на прямоугольный сигнал.

```
from thinkdsp import SquareSignal

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

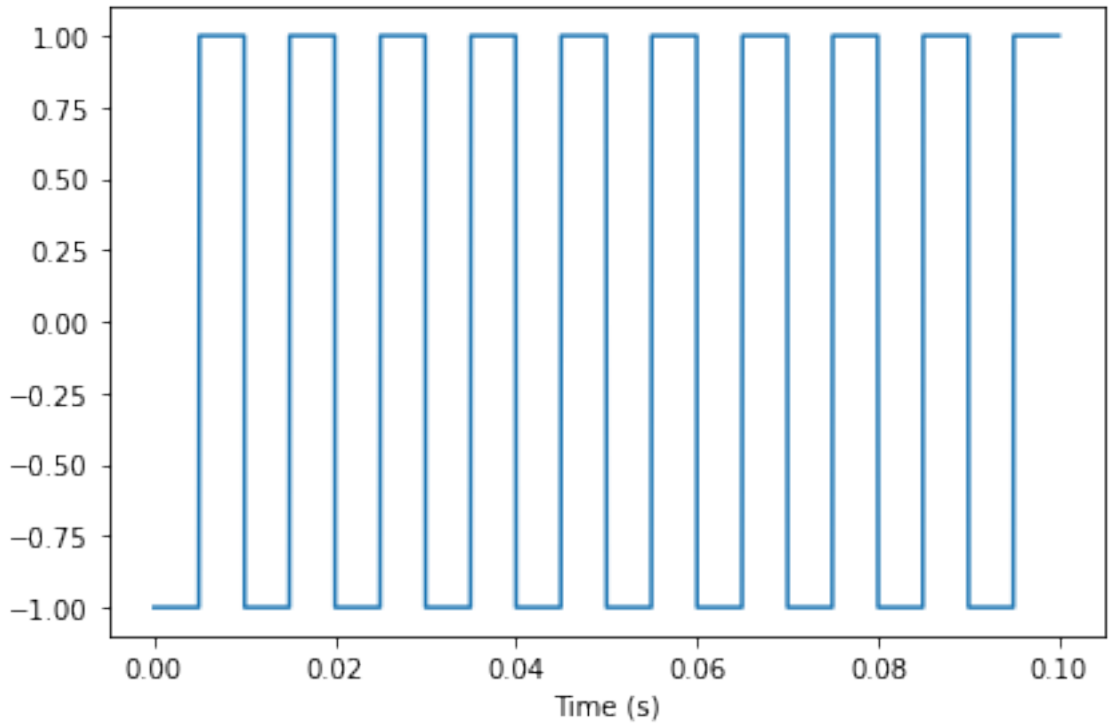

Рисунок 9.4. Рассматриваемый сигнал

Применим cumsum:

```
cumsum_wave = wave.cumsum()
cumsum_wave.plot()
decorate(xlabel='Time (s)')
```

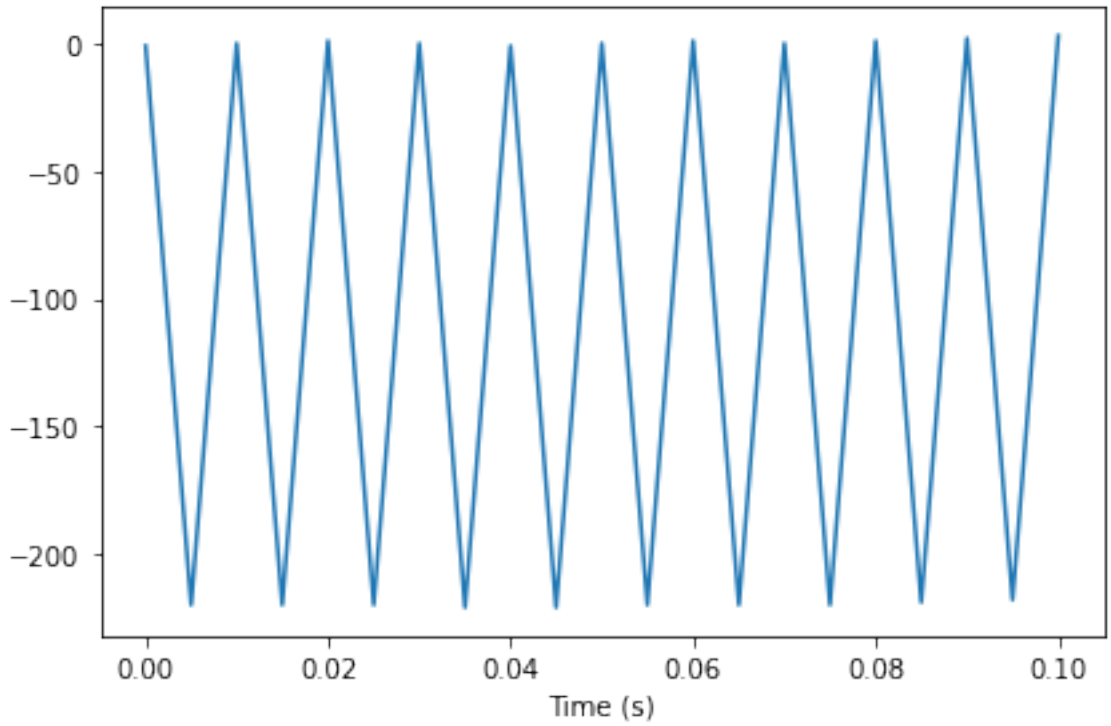

Рисунок 9.5. Рассматриваемый сигнал после применения cumsum

Получился треугольный сигнал

Теперь интеграл спектра:

```
int_spec = wave.make_spectrum().integrate()
int_spec.hs[0] = 0
int_wave = int_spec.make_wave()
int_wave.plot()
decorate(xlabel='Time (s)')
```

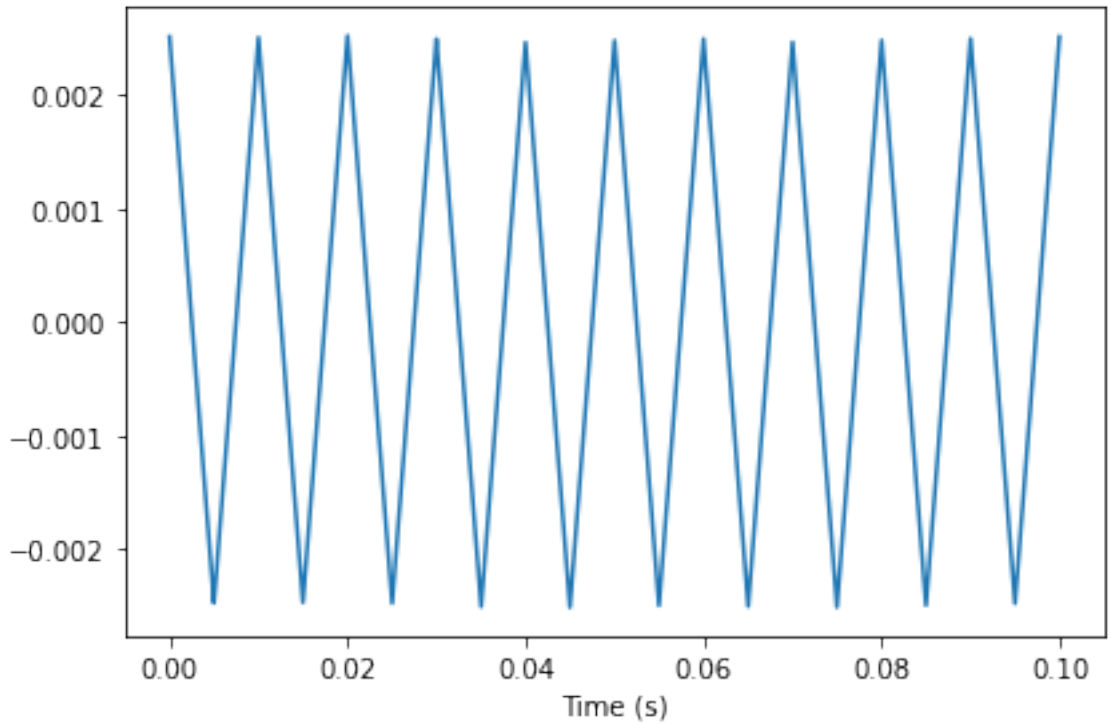

Рисунок 9.6. Рассматриваемый сигнал после применения integrate

Спектральный интеграл также получлся треугольным, однако с другой амплитудой

9.3. Упражнение 3

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

Изучим влияние двойного интегрирования на пилообразный сигнал.

```
from thinkdsp import SawtoothSignal

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

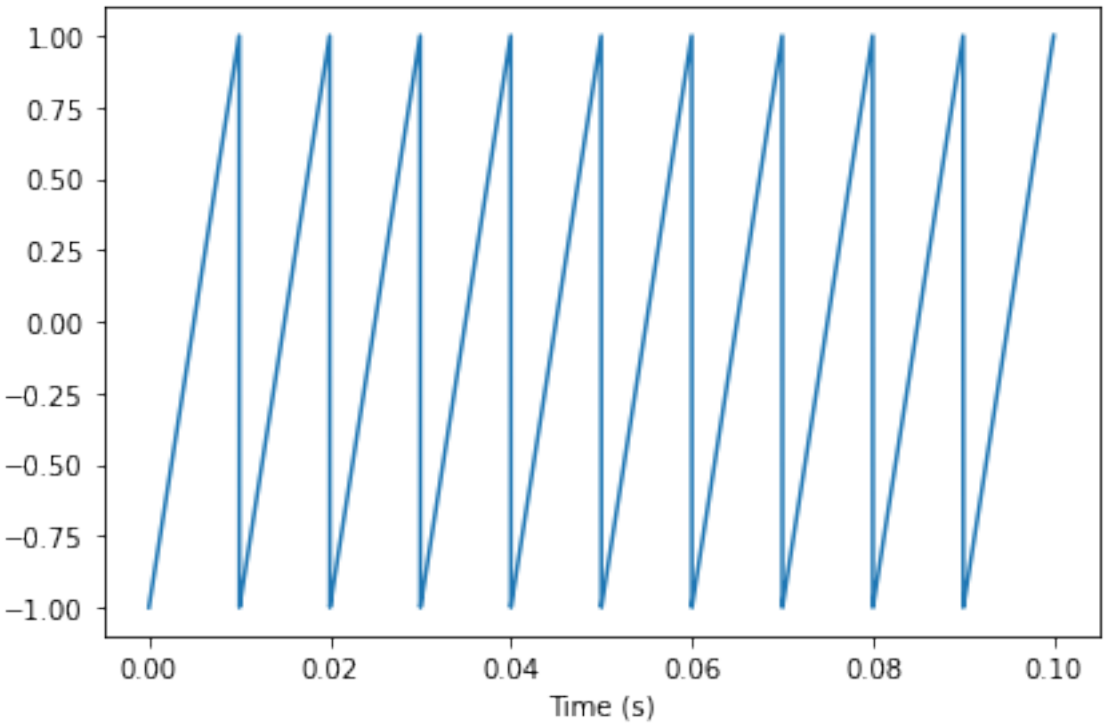

Рисунок 9.7. Пилообразный сигнал

```
spectrum = wave.make_spectrum().integrate().integrate()
spectrum.hs[0] = 0
wave1 = spectrum.make_wave()
wave1.plot()
decorate(xlabel='Time (s)')
```

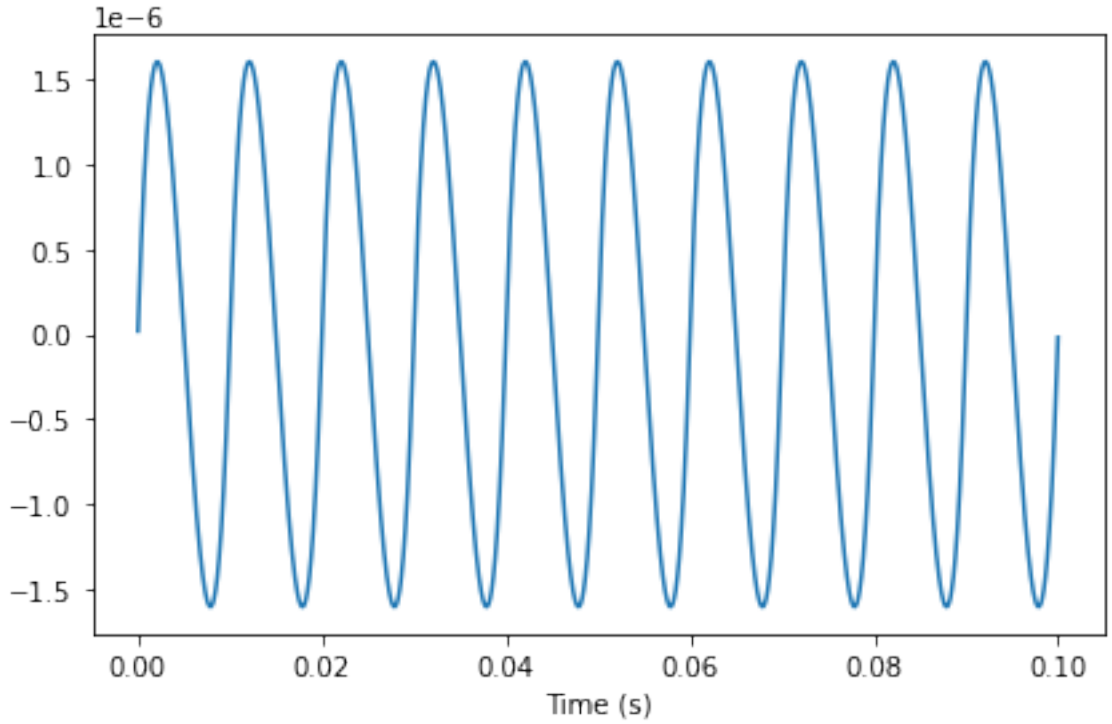

Рисунок 9.8. Изменённый сигнал

Сигнал напоминает синусойду, это связано с фильтрацией низких частот кроме основной.

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

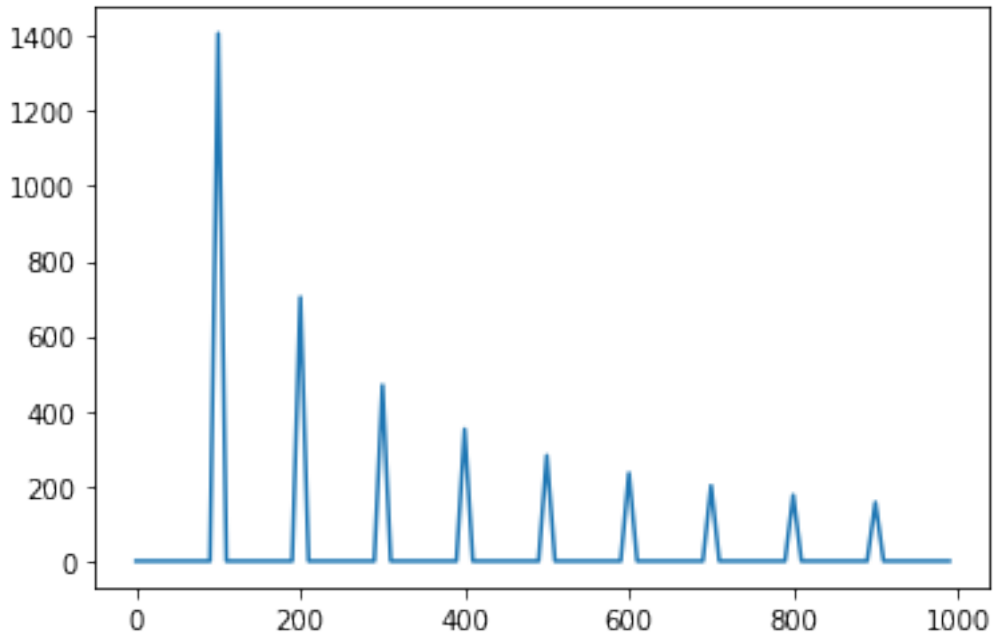

Рисунок 9.9. Спектр исходного сигнала

```
wave1.make_spectrum().plot(high=1000)
```

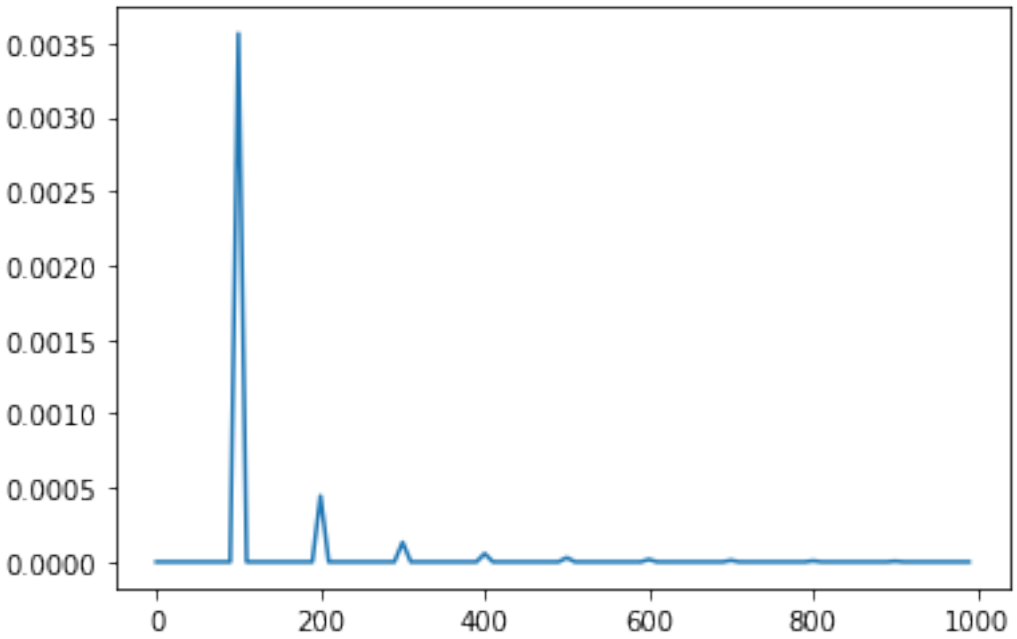

Рисунок 9.10. Спектр нового сигнала

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.

Изучим влияние второй разности и второй производной на CubicSignal сигнале.

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

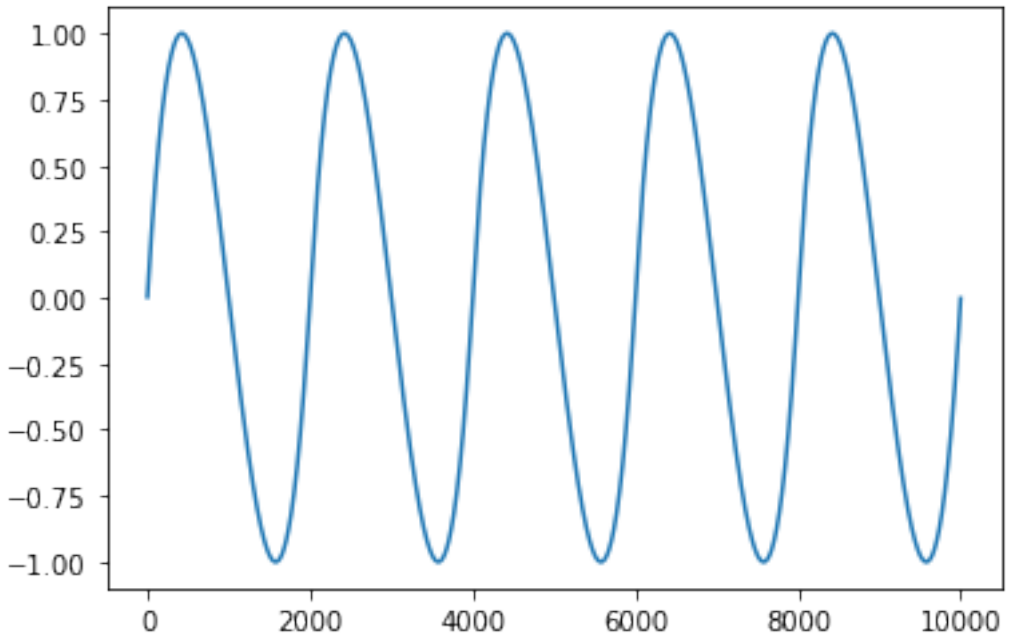

Рисунок 9.11. Кубический сигнал

Первая разность это параболы

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

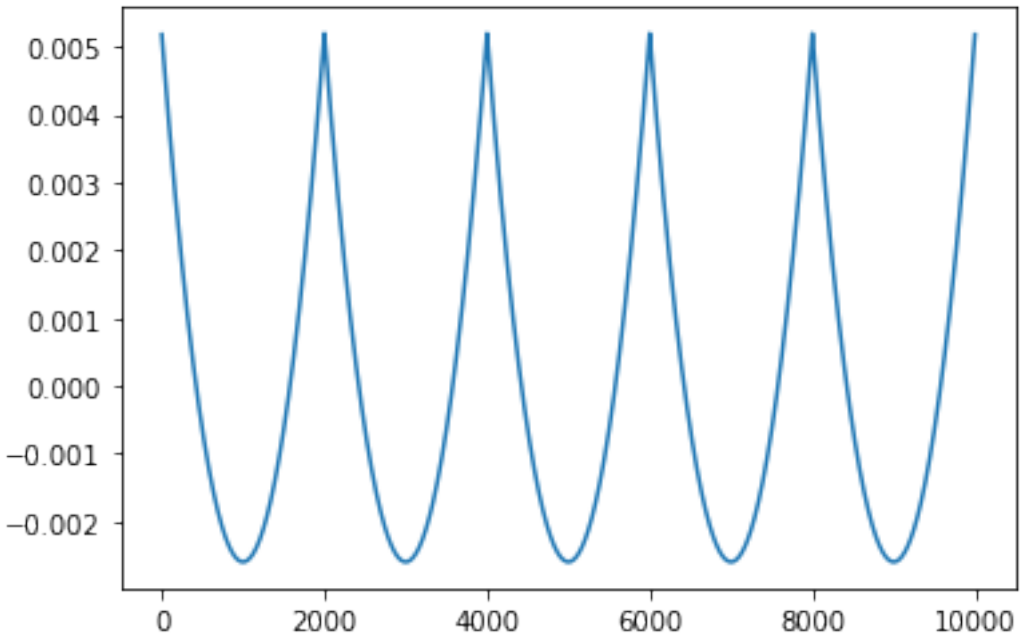

Рисунок 9.12. Первая разность

Вторая разность это пилообразный сигнал.

```
second = first.diff()
second.plot()
```

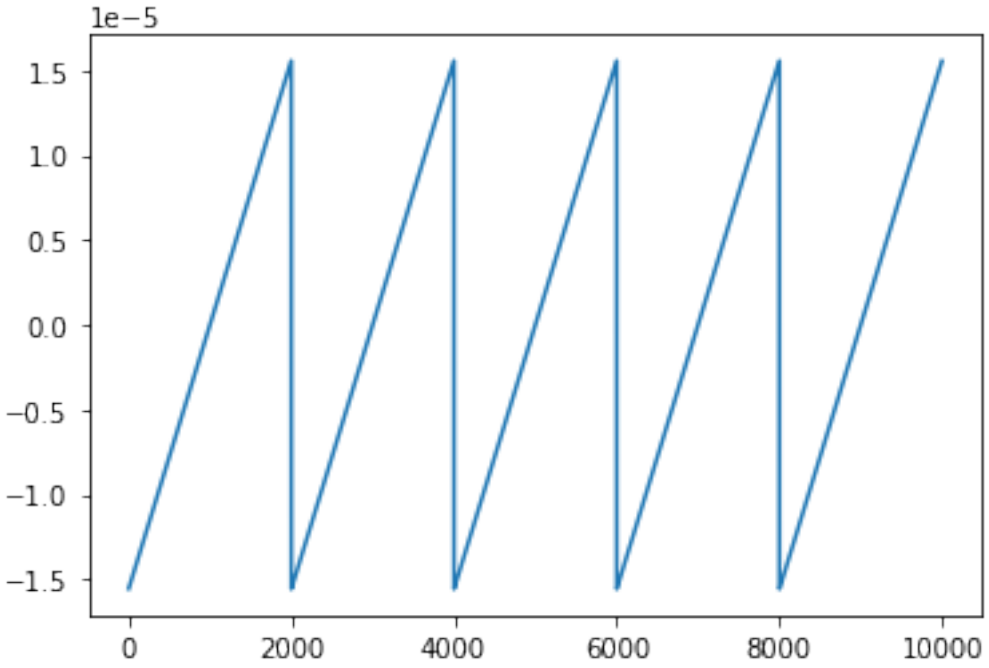

Рисунок 9.13. Вторая разность

Сделаем двойное дифференцирование.

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

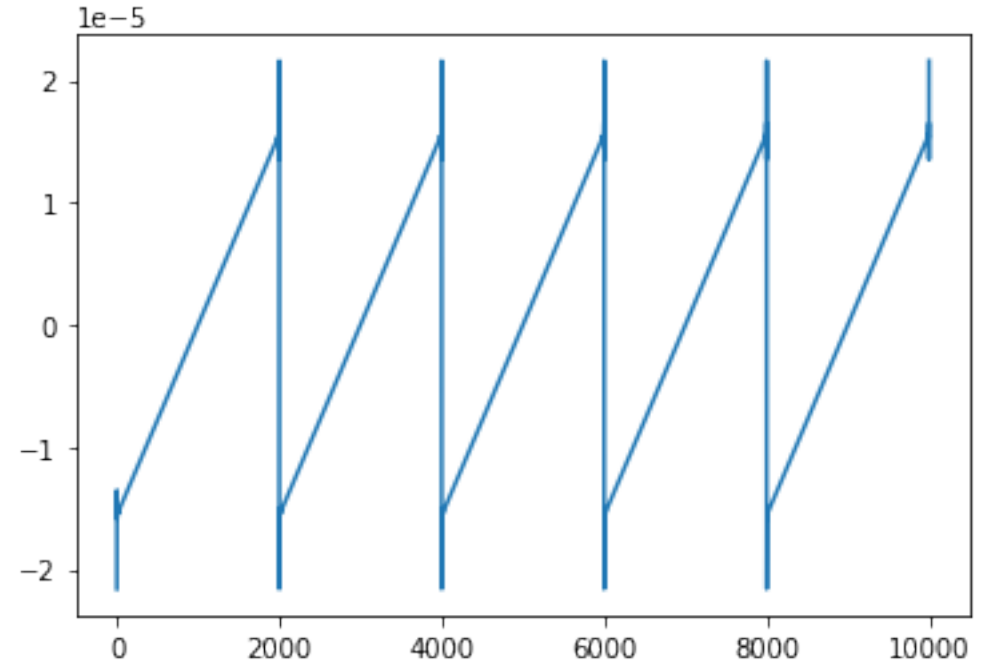

Рисунок 9.14. Полученный сигнал со звоном

В связи с тем что производная не определена в некоторых точках, на графике присутствует звон, используем фильтры:

```
from thinkdsp import zero_pad
from thinkdsp import Wave

diff_window = np.array([-1.0, 2.0, -1.0])
padded = zero_pad(diff_window, len(wave))
diff_wave = Wave(padded, framerate=wave.framerate)
diff_filter = diff_wave.make_spectrum()
diff_filter.plot()
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude ratio')
```

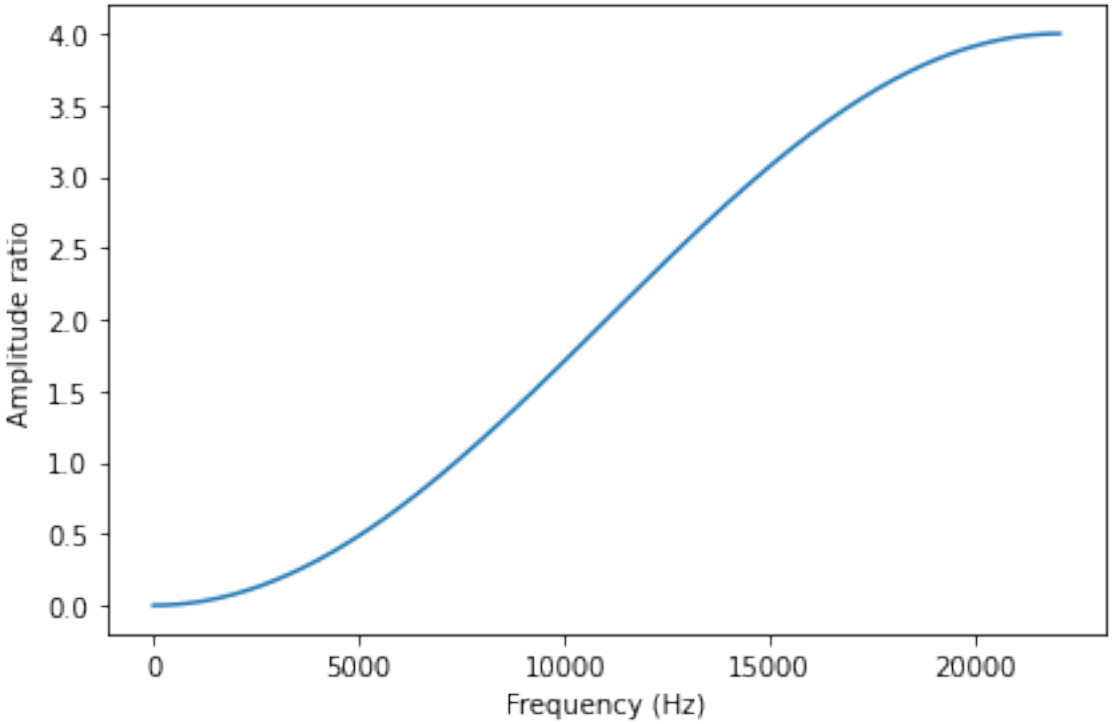

Рисунок 9.15. Полученные фильтры

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

```
deriv_filter = w.make_spectrum()
deriv_filter.hs = (2 * np.pi * 1j * deriv_filter.fs)**2
deriv_filter.plot()
```

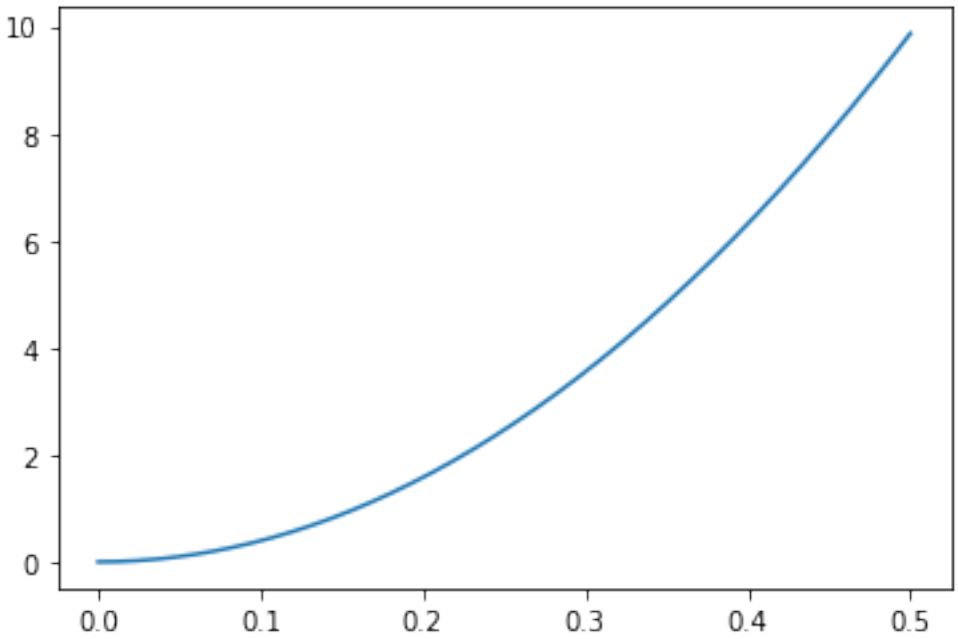

Рисунок 9.16. Полученные фильтры

9.5. Вывод

В данной работе были рассмотрены соотношения между окнами во временной области и фильтрами в частотной. Также рассмотрели конечные разности, cumsum - накапливающие суммы с апроксимирующим 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:

Устраним проблему с лишней нотой путем добавления нулей в конец сигнала.

```
from thinkdsp import read_wave

response = read_wave('180960__kleeb__gunshot.wav')
start = 0.12
response = response.segment(start=start)
response.shift(-start)
response.normalize()
response.plot()
decorate(xlabel='Time (s)')
```

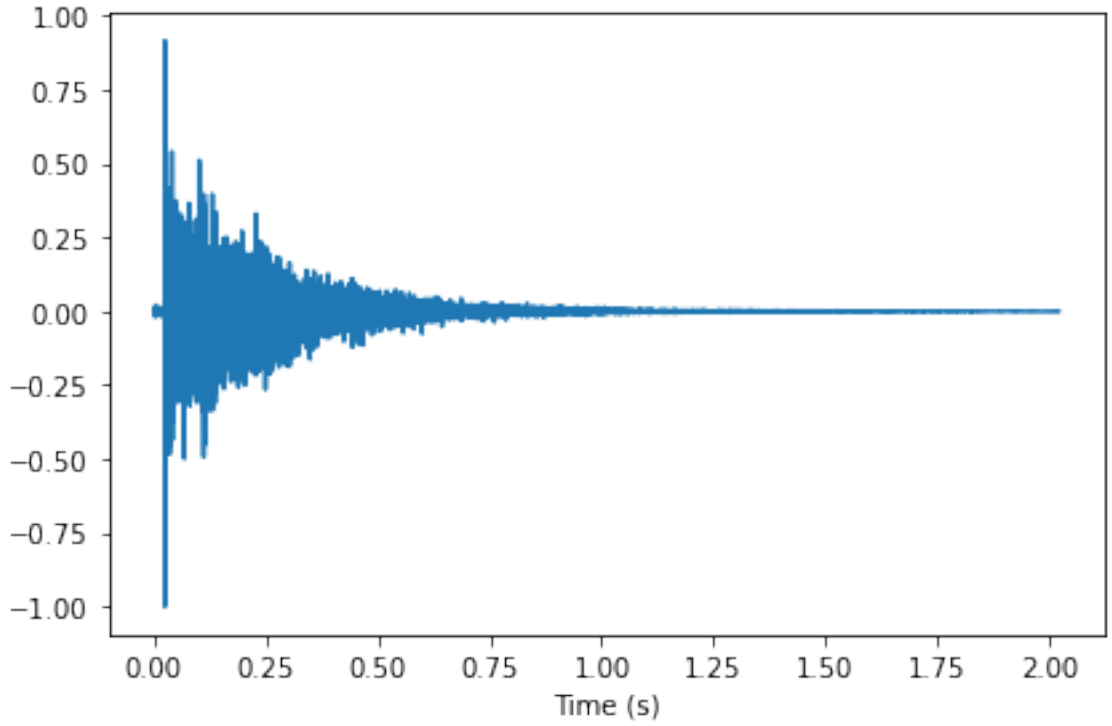

Рисунок 10.1. Сигнал

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

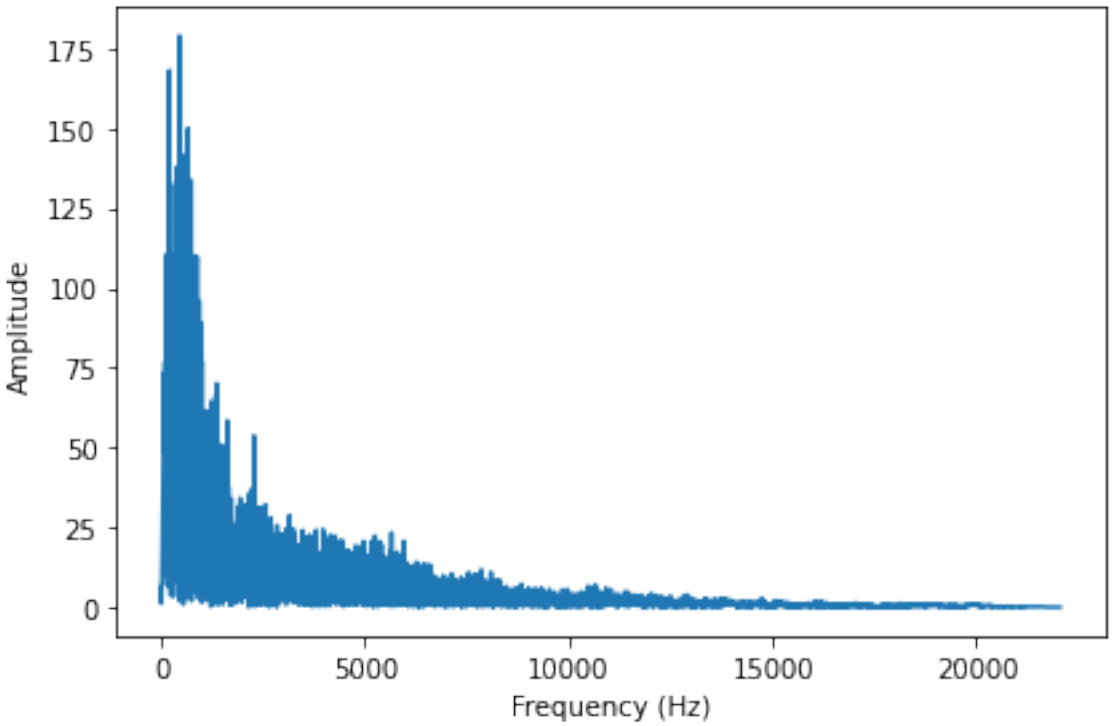

Рисунок 10.2. Спектр сигнала

Теперь перейдём к самой записе:

```
violin = read_wave('92002__jcveliz__violin-origional.wav')
start = 0.11
violin = violin.segment(start=start)
violin.shift(-start)
violin.truncate(len(response))
violin.normalize()
violin.plot()
decorate(xlabel='Time (s)')
```

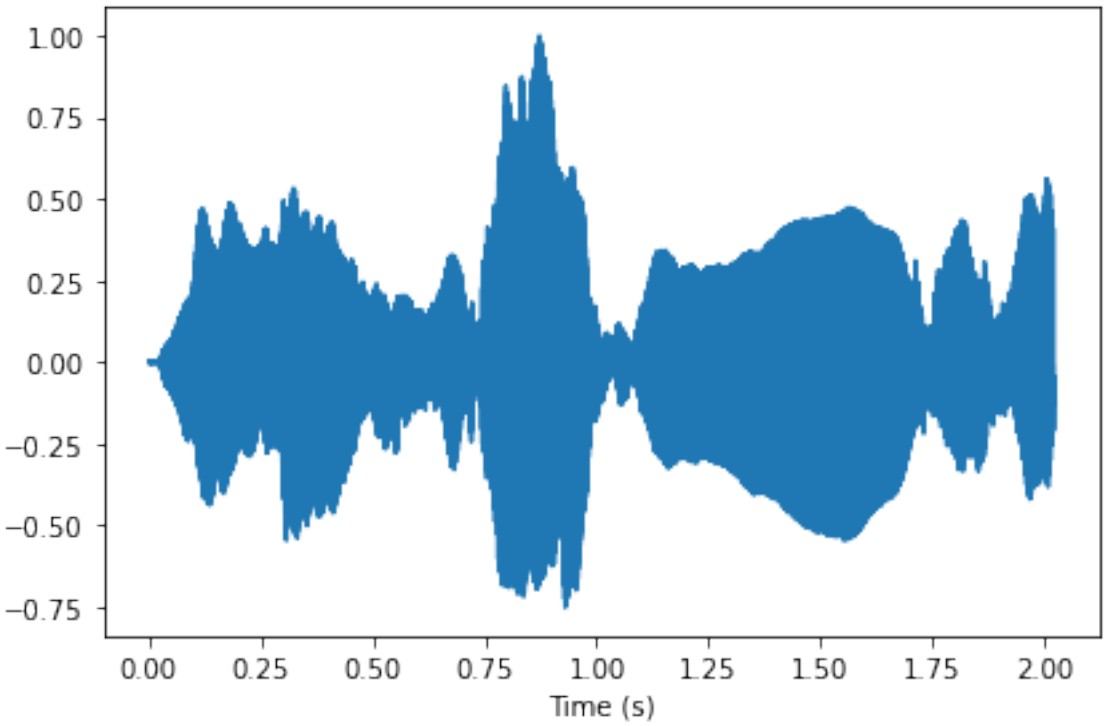

Рисунок 10.3. График сигнала

```
spec2 = violin.make_spectrum()
```

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

```
wave = (spec * spec2).make_wave()
wave.normalize()
wave.plot()
```

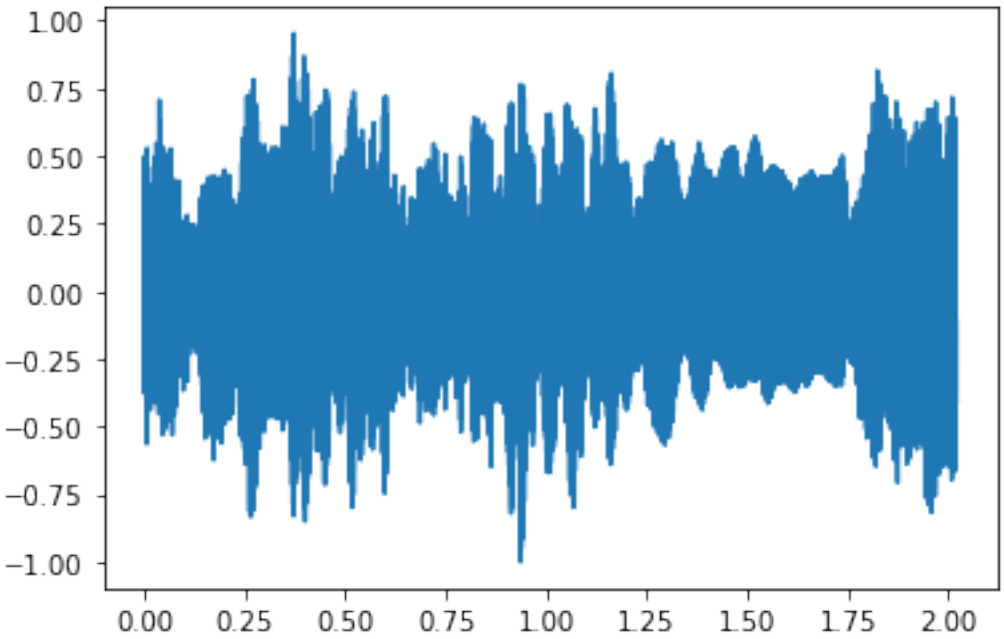

Рисунок 10.4. График получившегося сигнала

Проблему удалось решить.

10.2. Упражнение 2

Необходимо смоделировать 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. Характеристику возьмем из учебника.

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

response = read_wave('stalbans_a_mono.wav')

start = 0
duration = 5
response = response.segment(duration=duration)
response.shift(-start)
response.normalize()
response.plot()
decorate(xlabel='Time (s)')
decorate(xlabel='Time (s)')
```

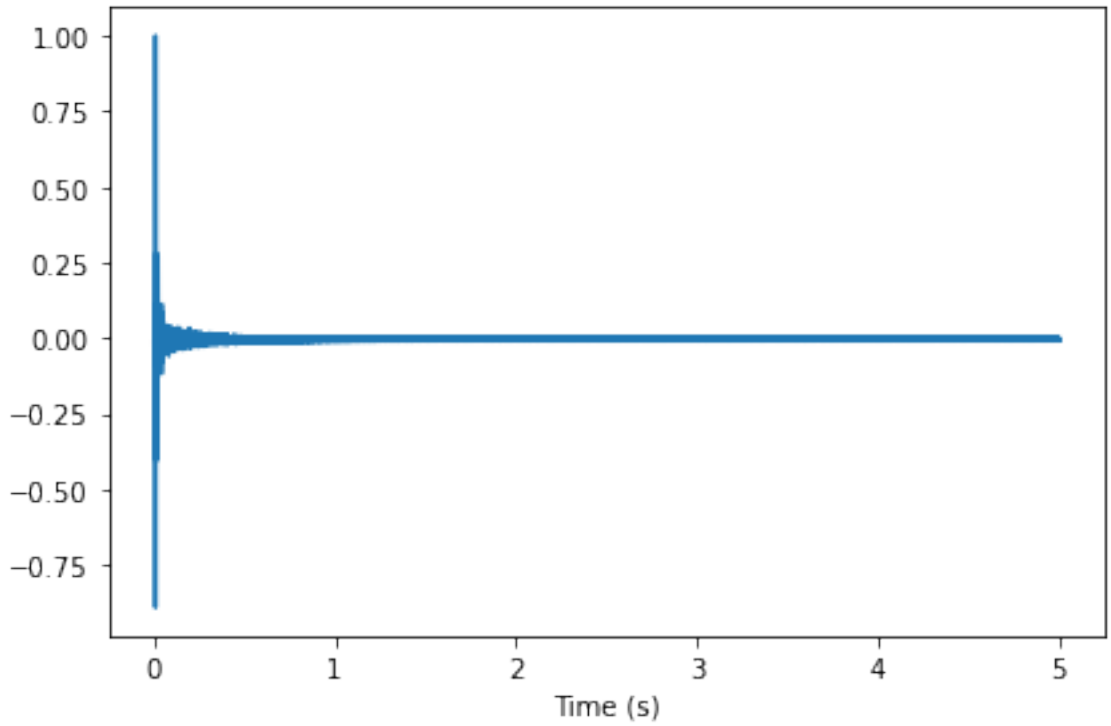

Рисунок 10.5. График загруженного сигнала

ДПФ:

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

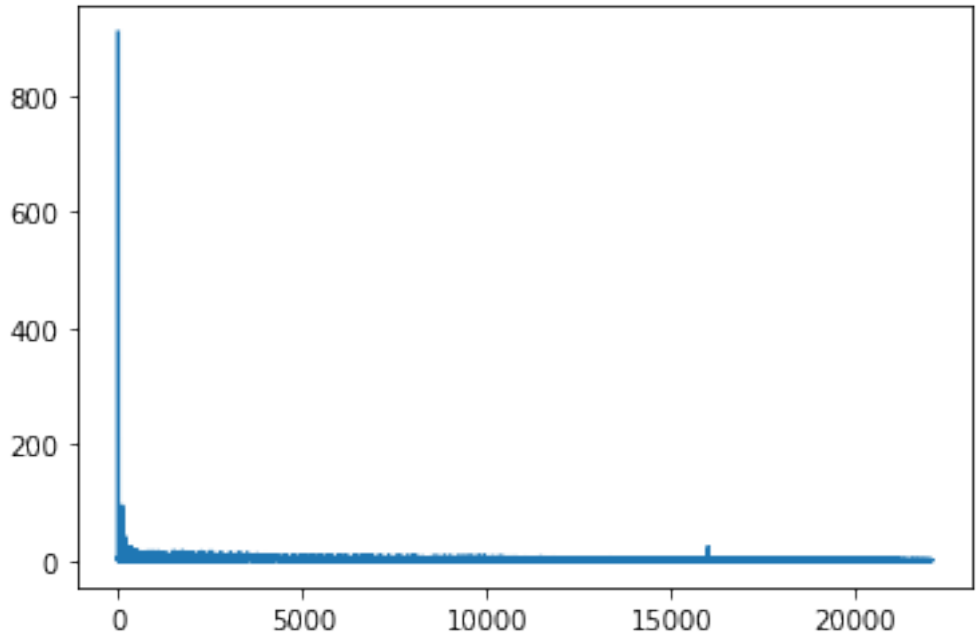

Рисунок 10.6. ДПФ импульсной характеристики

Промоделируем запись в пространстве, будем также использовать звук из учебника - скрипку

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

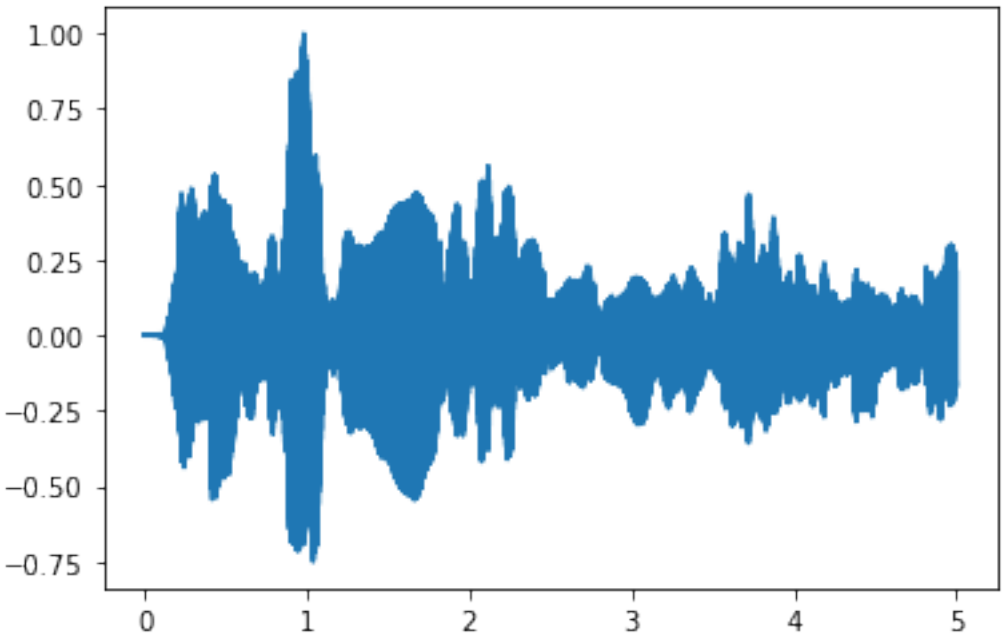

Рисунок 10.7. Сигнал звука скрипки

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

```
(110251, 110251)
```

```
spectrum.fs, transfer.fs
```

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

Используем свертку.

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

Используем умножение:

```
result = (spectrum * transfer).make_wave()
result.normalize()
result.plot()
```

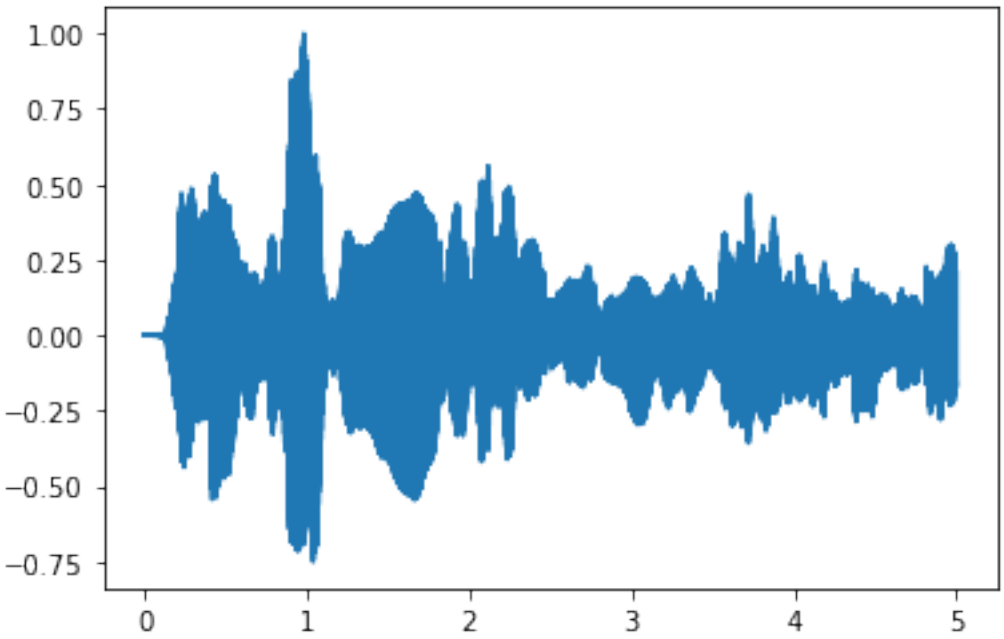

Рисунок 10.8. Полученный график

10.3. Вывод

В данной работе мы рассмотрели основные позиции из теории сигналов и систем, например музыкальную акустику. При описании линейных стационарных систем используется теорема о свёртке.

11. Модуляция и сэмплирование

11.1. Упражнение 1

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.

Возьмем звук барабанов, применим фильтр НЧ до выборки, а затем, опять с помощью фильтра НЧ удалим спектральные копии, вызванные выборкой.

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

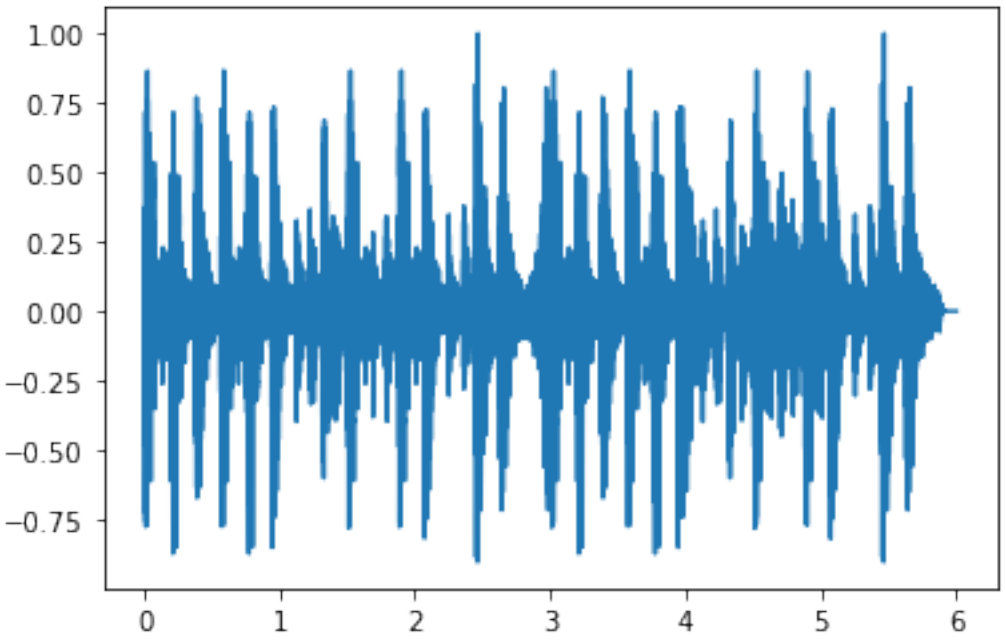

Рисунок 11.1. График звука барабанов

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

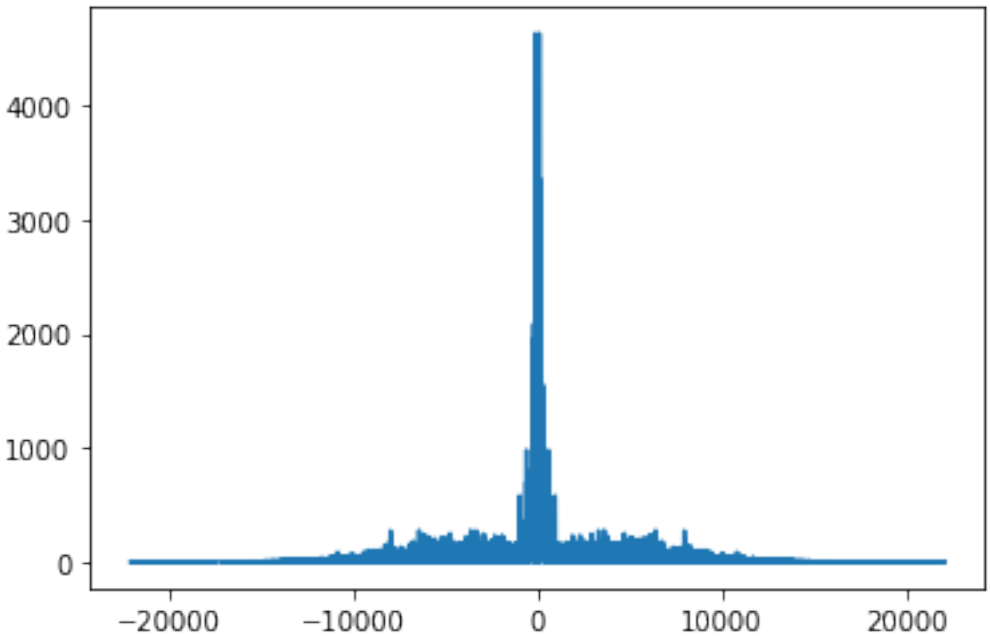

Рисунок 11.2. Спектр сигнала

Применим фильтр низких частот:

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

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

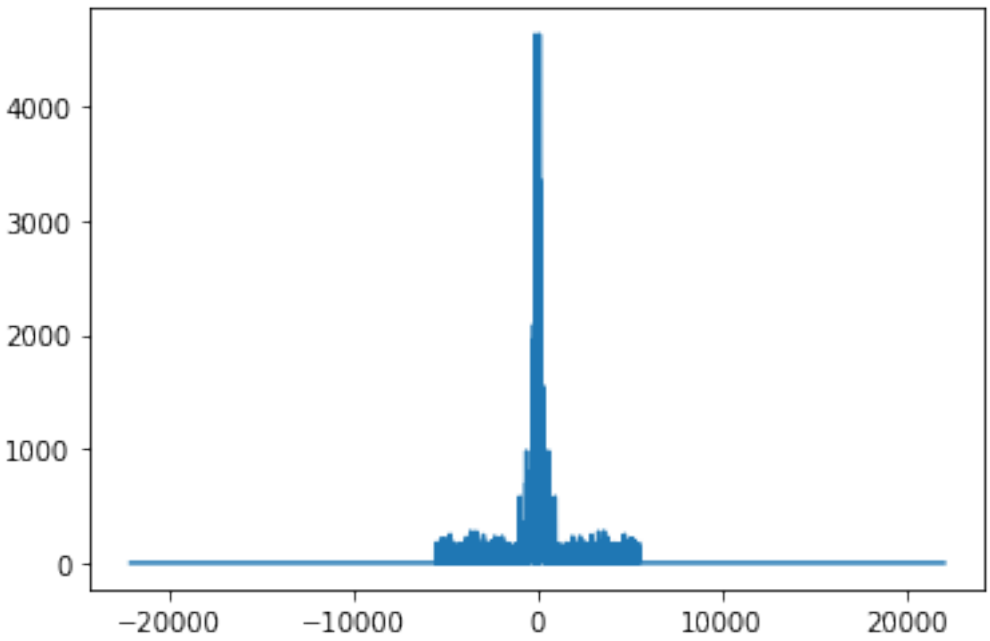

Рисунок 11.3. Отфильтрованный сигнал

Применим функцию из учебника, которая имитирует процесс выборки.

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

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

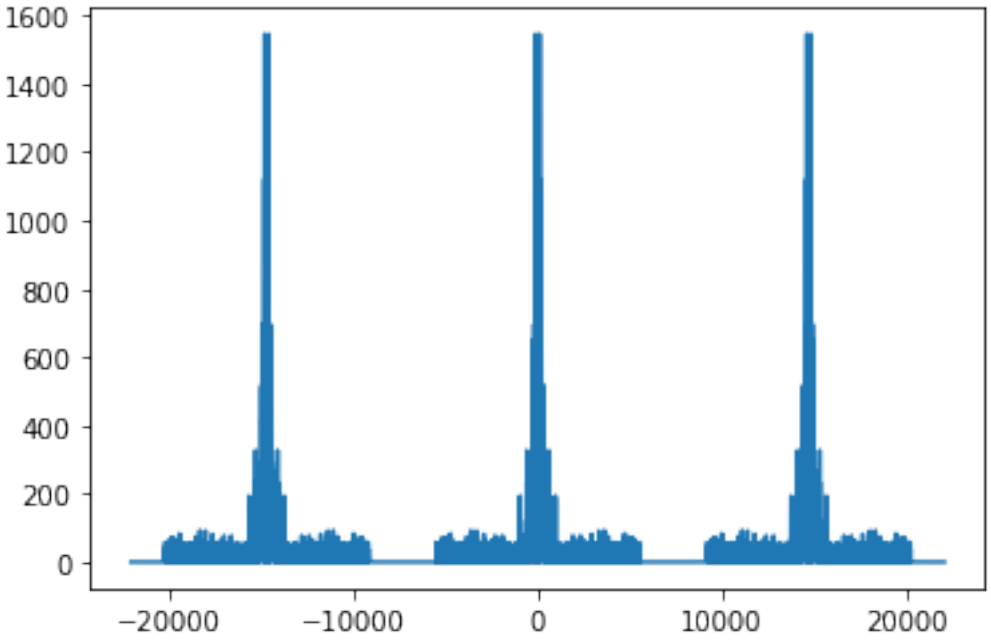

Рисунок 11.4. Получившийся спектр

Теперь удалим спектральные копии:

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

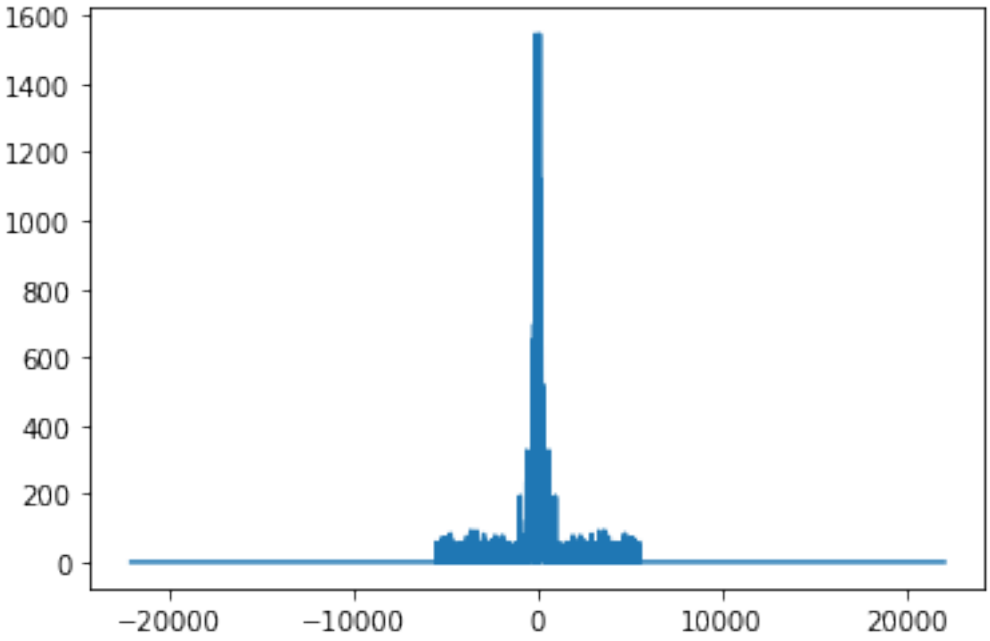

Рисунок 11.5. Результат избавления от копий

Сравним звуки

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

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

spectrum.plot()
sampled_spectrum.plot()
```

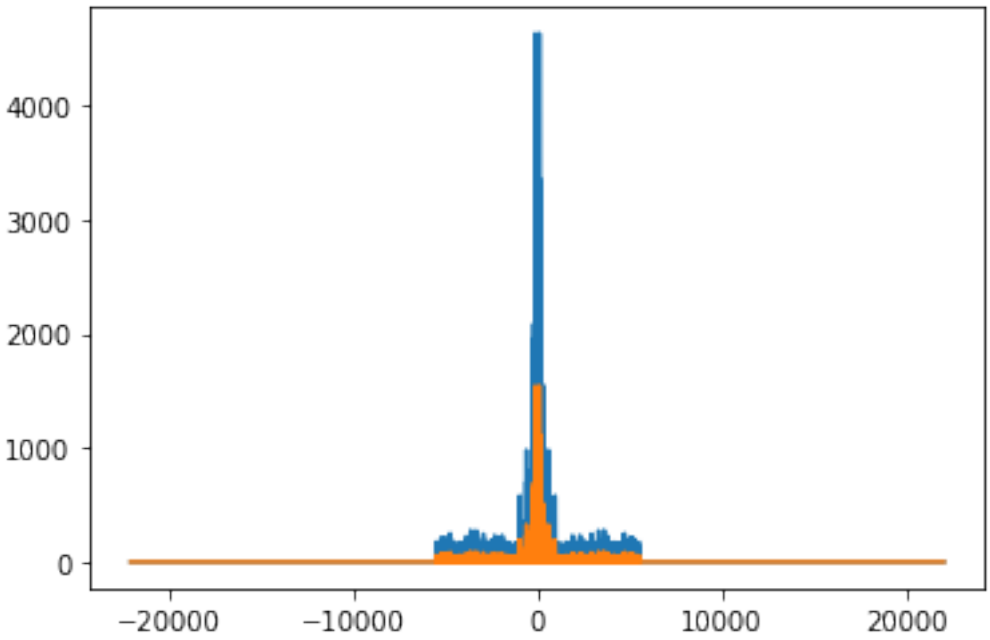

Рисунок 11.6. Сравнение спектров

Звуки отличаются , увеличим амлитуду в три раза:

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

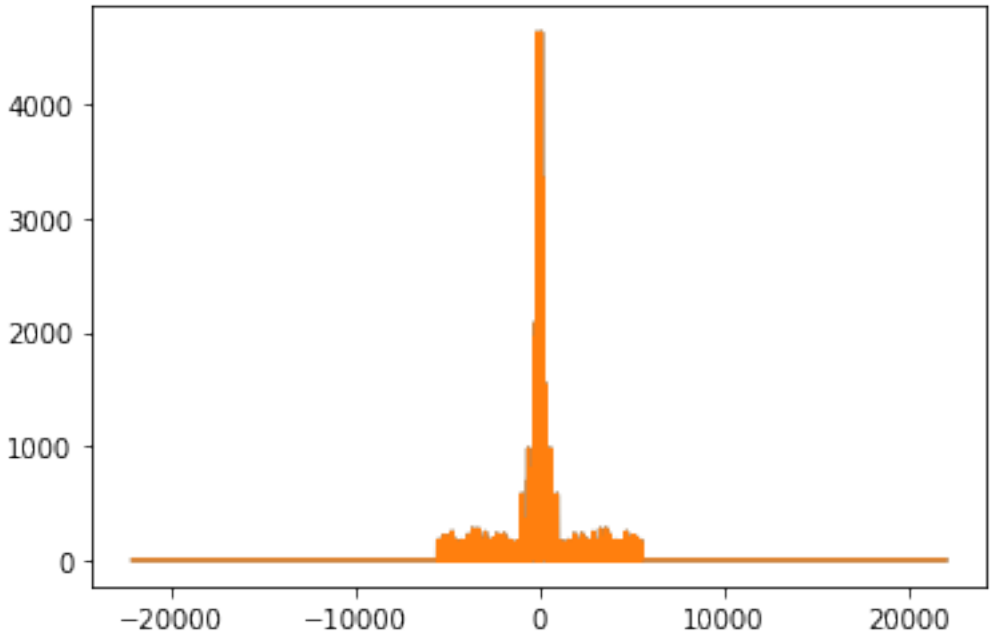

Рисунок 11.7. Сравнение спектров

В итоге разница между интерполированной волной и фильтрованной волной есть, но она едва заметная.

11.2. Вывод

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

12. FSK

12.1. Описание работы FSK

Frequency Shift Key - вид модуляции, при которой скачкообразно изменяется частота несущего сигнала в зависимости от значений символов информационной последовательности. Частотная модуляция весьма помехоустойчива, так как помехи искажают в основном амплитуду, а не частоту сигнала.

Для модулирования и тестирования данного процесса, необходимо построить следующую схему в графическом интерфейсе GNU Radio

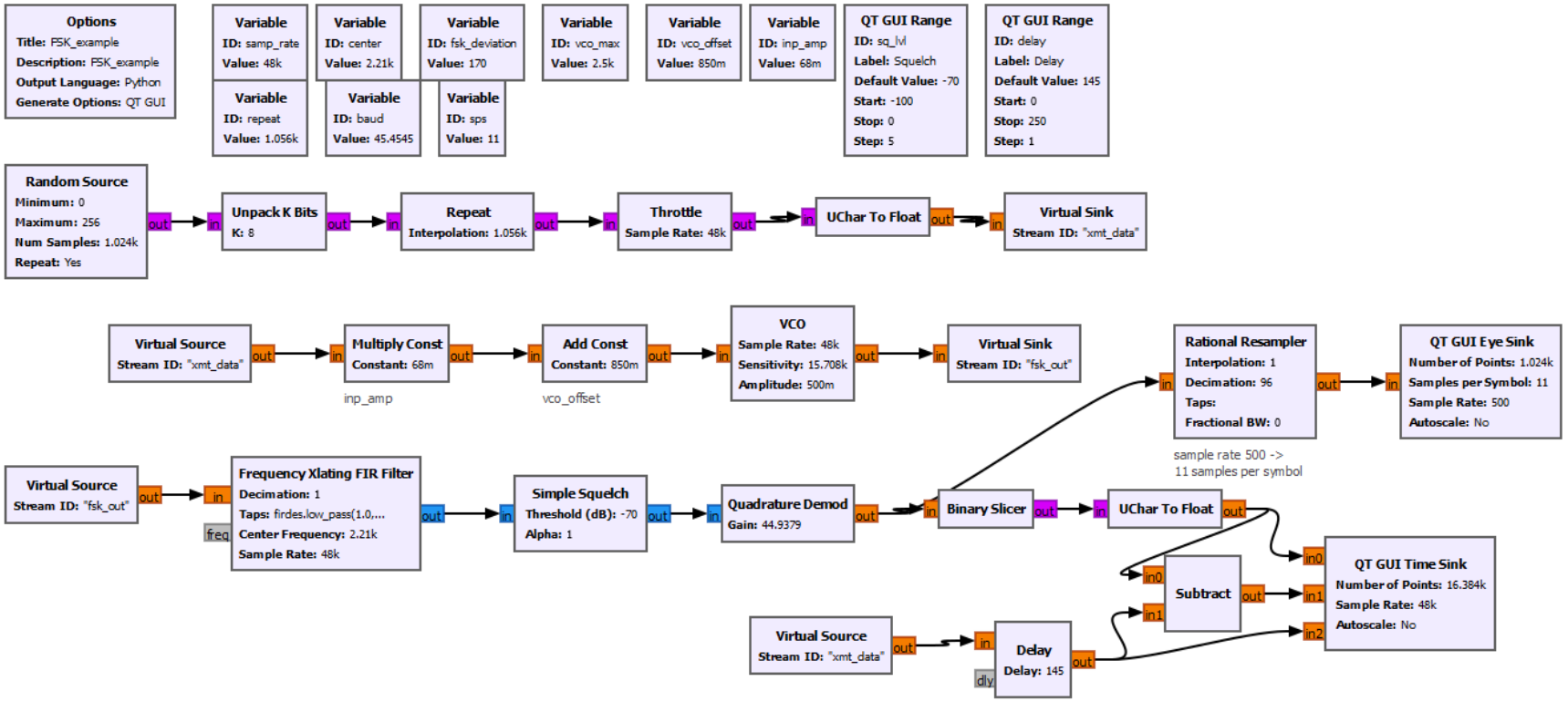

Рисунок 12.1. Схема FSK

В данной схеме используются следующие блоки:

- Frequency Xlating FIR Filter - Этот блок выполняет частотный перевод сигнала, а также понижает разрешение сигнала, запуская на нем децимирующий FIR-фильтр.
- Simple Squelch - Простой блок шумоподавления на основе средней мощности сигнала и порога в дБ.
- Quadrature Demod - Этот блок вычисляет произведение одновыборочного отложенного и сопряженного входного сигнала и нераскрытого сигнала, а затем вычисляет аргумент (также известный как угол, в радианах) результирующего комплексного числа.
- Binary Slicer - Нарезает числа с плавающей запятой, производя 1-битный вывод. Положительный ввод производит двоичную 1, а отрицательный ввод производит двоичный ноль.
- QT GUI Sink - Выводы необходимой инфомрации в графическом интерфейсе.
- Options - Этот блок устанавливает некоторые общие параметры графа потоков. Такие как название проекта, авторство и другие.
- Variable - Этот блок сопоставляет значение с уникальной переменной. Есть возможность использования переменной в другом блоке, благодаря идентификатору (id) блока переменных.
- Multiply Const - Умножает входной поток на скалярную или векторную константу.
- Add Const - Прибавляет к входному потоку скалярную или векторную константу.
- QT GUI Range - Этот блок создает переменную с выбором виджетов. Переменной может быть присвоено значение по умолчанию, и ее значение может быть изменено во время выполнения в указанном диапазоне.
- Random Source - Генератор случайных чисел.
- Unpack K bits - Преобразует байт с k релевантных битов в k выходных байтов с 1 битом каждый.
- Repeat - Количество раз для повторения входных данных, выступающих в качестве коэффициента интерполяции.
- Throttle - Этот блок служит для того, чтобы дросселировать поток так, чтобы средняя скорость не превышала удельную скорость.
- Uchar To Float - Преобразует unsigned chars в поток float.
- Virtual Sink - Служит для сохранения потока в вектор.
- Virtual Source - Работает в паре с Virtual Sink блоком. Источник данных, который передаёт элементы на основе входного вектора.
- VCO - Генератор с регулируемым напряжением. Создает синусоиду частоты на основе амплитуды входного сигнала.

В данном примере используется скорость передачи данных для радиотелетипа Baudot, для данного типа битовое время составляет 22 миллисекунды, поэтому скорость передачи данных установлена на 1/0,022, что дает 45,4545. Коэффициент повторения (int)(samp_rate*0,022). Для этого флоуграфа блок VCO генерирует сигналы 2295 Гц (отметка = 1) и 2125 Гц (отметка = 0). Расчеты для этого следующие:

- При выборе полной шкалы частоты 2500 Гц (vco_max) для входа +1 чувствительность VCO = (2 * math.pi * 2500 / 1) = 15708. Можно использовать любую частоту выше 2295 Гц. 2500 Гц — хорошее круглое число.
- Глядя на вывод виртуального источника «xmt_data», Mark = +1.0 и Space = 0.0.
- Диапазон частон 2125 Гц создается при помощи vco_offset = (2125 / 2500) = 0,850 + (0,0 * 0,068)
- Частота отметки 2295 Гц создается вектором inp_amp = (1,0 * 0,068) + vco_offset = 0,918, что равно (2295/2500). Параметр отводов частотного Xlating FIR Filter равен 'firdes.low_pass(1.0,samp_rate,1000,400)'.

Случайный источник (Random Source) генерирует байтовые значения от 0 до 255. Далее при помощи блока Unpack K Bits распаковывает каждый бит входных данных в виде отдельного байта со значением в младшем разряде. Для ограничения потока используется Throttle блок.

На стороне приема, Frequency Xlating FIR Filter блок смещает принимаемый сигнал так, чтобы он был сосредоточен вокруг центральной частоты - между частотами Mark и Space. Блок Quadrature Demod выдает сигнал, который является положительным для входных частот выше нуля и отрицательным для частот ниже нуля. Этот сигнал подается в блок Binary Slicer, выходные данные представляют собой байт 1 или 0, что и является полученной информацией.

12.2. Тестирование

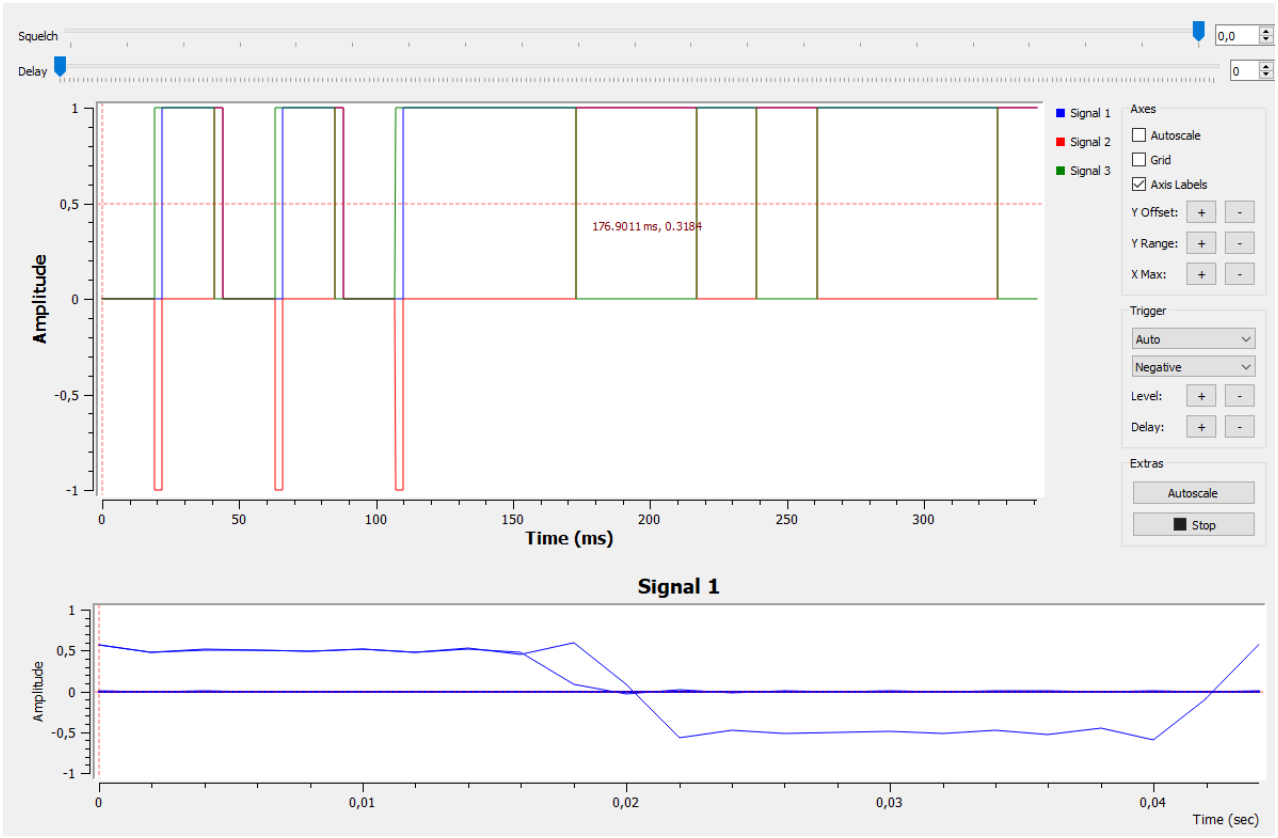

Рисунок 12.2. Модуляция без задержки и шума

На графике можно видеть 3 сигнала разного цвета. Зеленный показывает данные которые были переданы передатчиком. Синий - это данные полученные приемником. А красный отвечает за разницу между зеленым и синим. Красный сигнал должен быть равен нулю, это сигнализирует о том, что данные передаются верно, однако на диаграмме сверху видно, что это не так и выходит так, что полученная информация различается от той, которую передавали. Принятый сигнал находится на некоторое количество бит позади, потому что цепочка передатчика и приемника имеет много блоков и фильтров, которые задерживают сигнал. Для того чтобы это исправить необходимо сделать задержку между приемом и передачей данных, что и обеспечивает блок Delay. Правильном задержкой является 145.

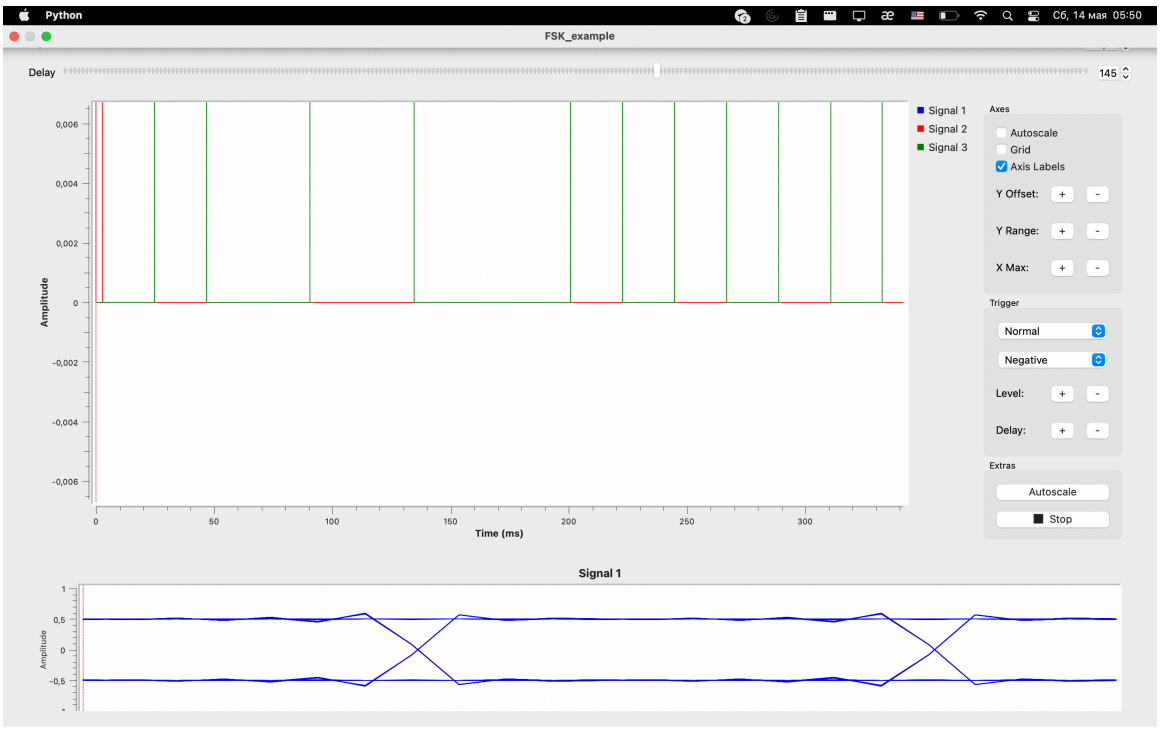

Рисунок 12.3. Модуляция с правильной задержкой

Теперь как можно заметить все хорошо и данные передаются и получаются корректно.

12.3. Вывод

В данной лабораторной работе был рассмотрен один из способов модуляции. При помощи GNU Radio была создана необходимая модель и протестирована.