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:
Д. С. Ковалевский
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?

Загрузим выбранный звук пианино

```
if not os.path.exists('440931__xhale303__piano-loop-1.wav'):
  !wget https://github.com/hotnotHD/Telecom/raw/main/440931__xhale303__piano-
     loop-1.wav
wave = read_wave('440931__xhale303__piano-loop-1.wav')

wave.make_audio()
```

Построим график wave

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

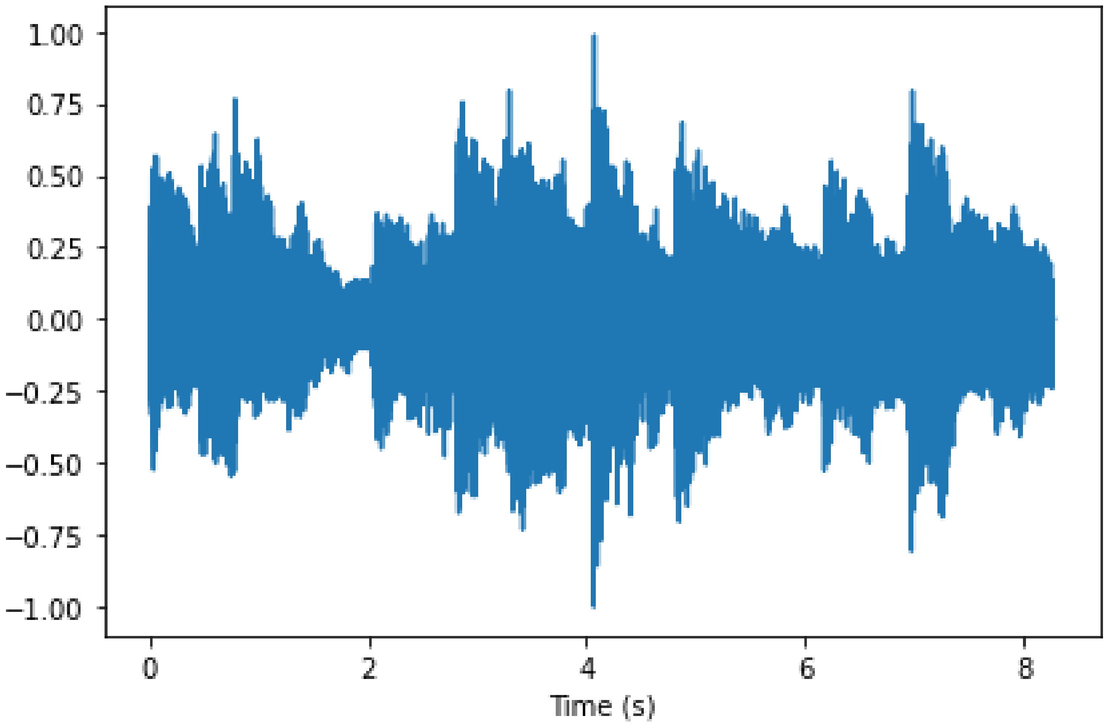

Рисунок 1.1. График всего звука

Выделяем полусекундный сегмент

```
segment = wave.segment(start=3, duration=0.5)
segment.make_audio()

segment.plot()
decorate(xlabel='Time (s)')
```

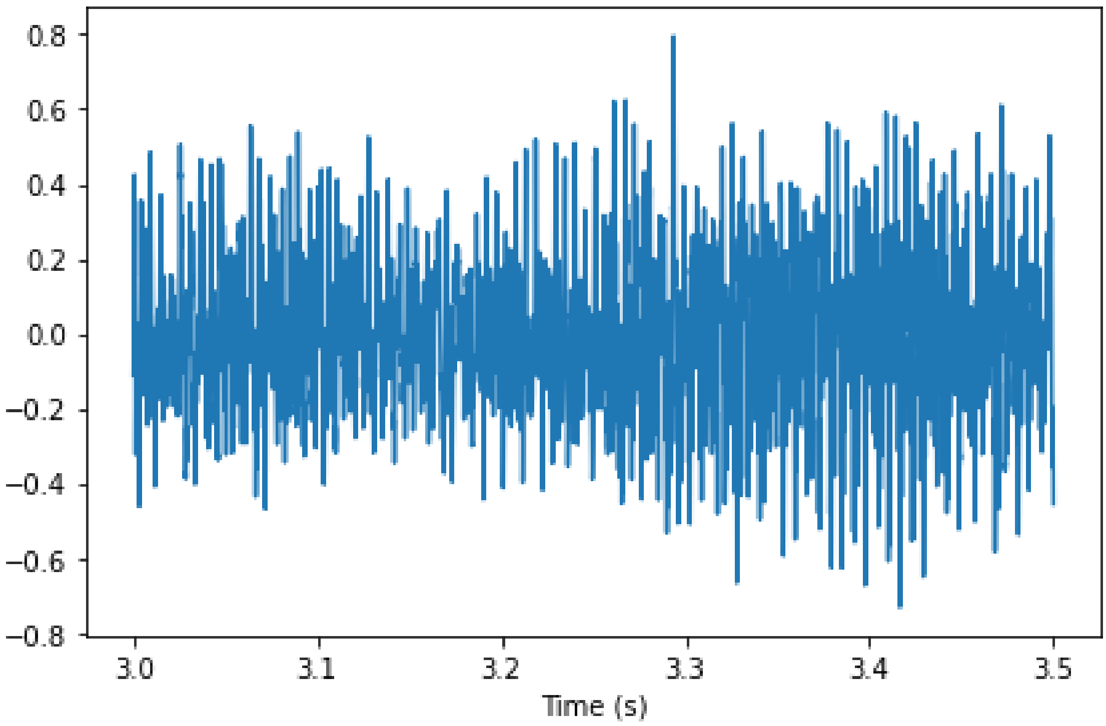

Рисунок 1.2. График сегмента звука

Спектр сегмента

```
spectrum = segment.make_spectrum()
spectrum.plot(high=6000)
decorate(xlabel='Frequency (Hz)')
```

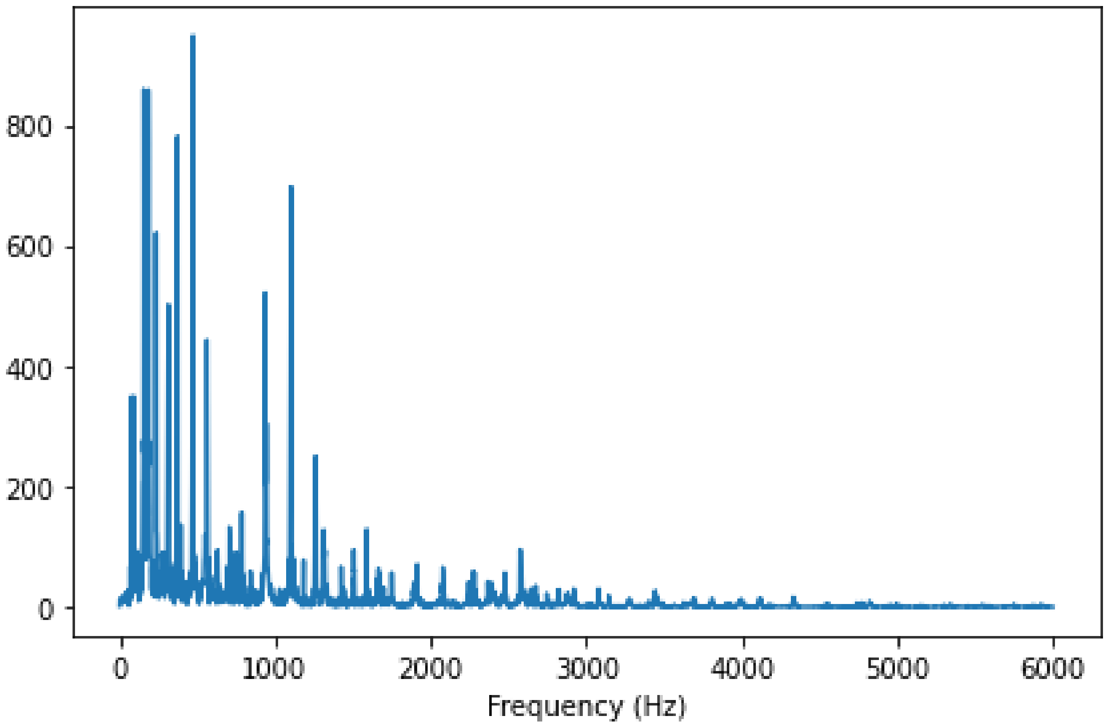

Рисунок 1.3. Спектр сегмента звука

Применим функции фильтрации

Убираем частоты ниже 500

```
spectrum.high_pass(500)
spectrum.plot(high=6000)
decorate(xlabel='Frequency (Hz)')
```

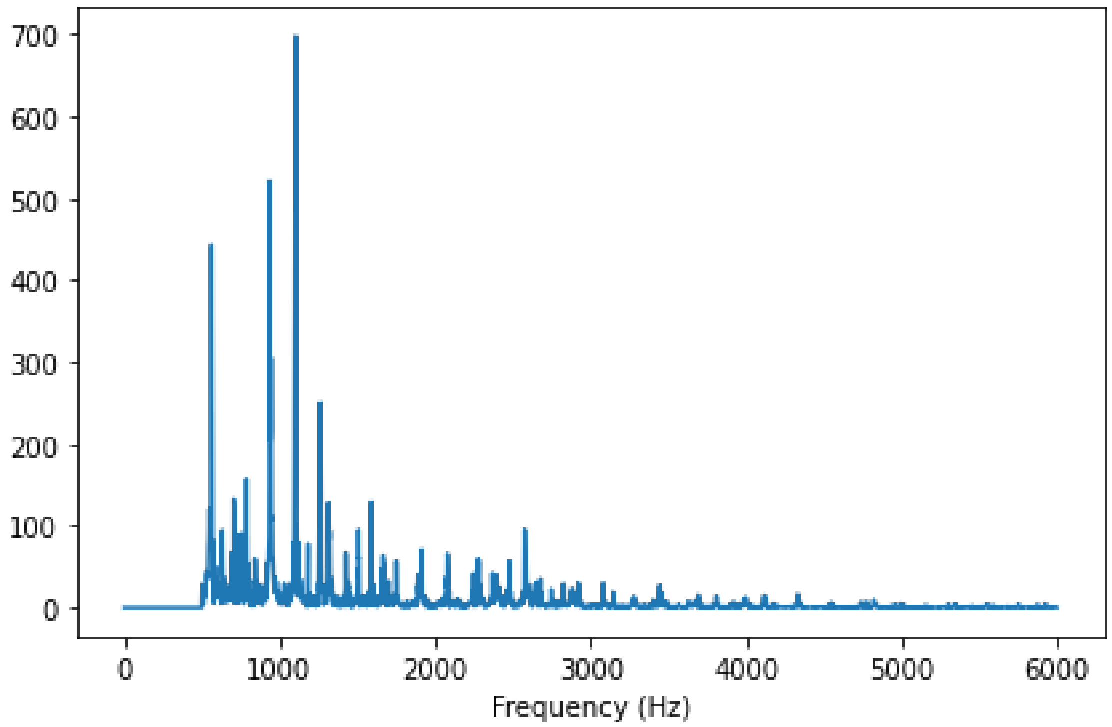

Рисунок 1.4. График частот без тех, что ниже 500

Уберем частоты выше 2000

```
spectrum.low_pass(2000)
spectrum.plot(high=6000)
decorate(xlabel='Frequency (Hz)')
```

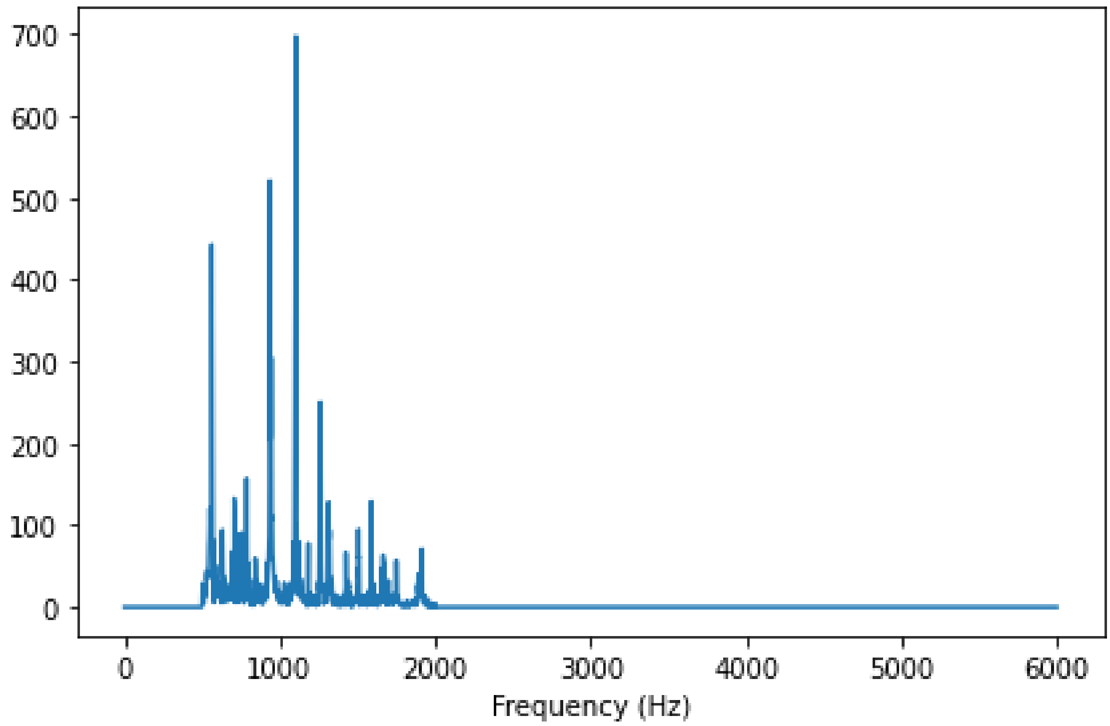

Рисунок 1.5. График частот без тех, что выше 2000

Убираем частоты в срезе

```
spectrum.band_stop(1100, 2500)
spectrum.plot(high=6000)
decorate(xlabel='Frequency (Hz)')
```

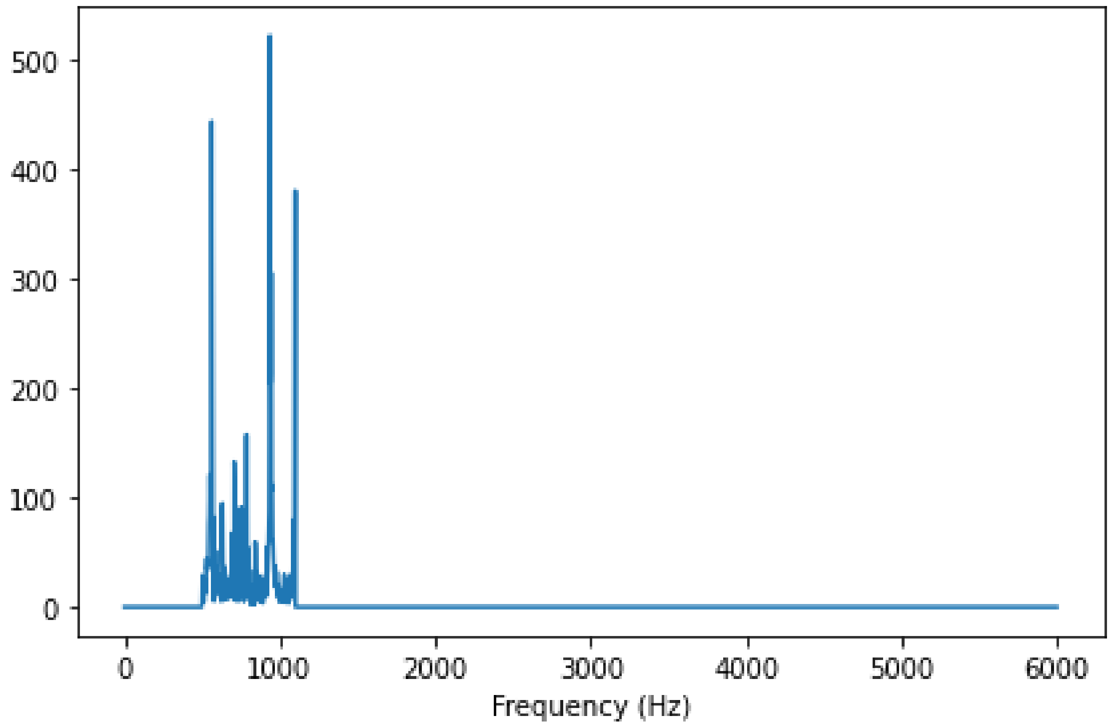

Рисунок 1.6. График частот после применения ФПЗ

Переведем из спектра в волну

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

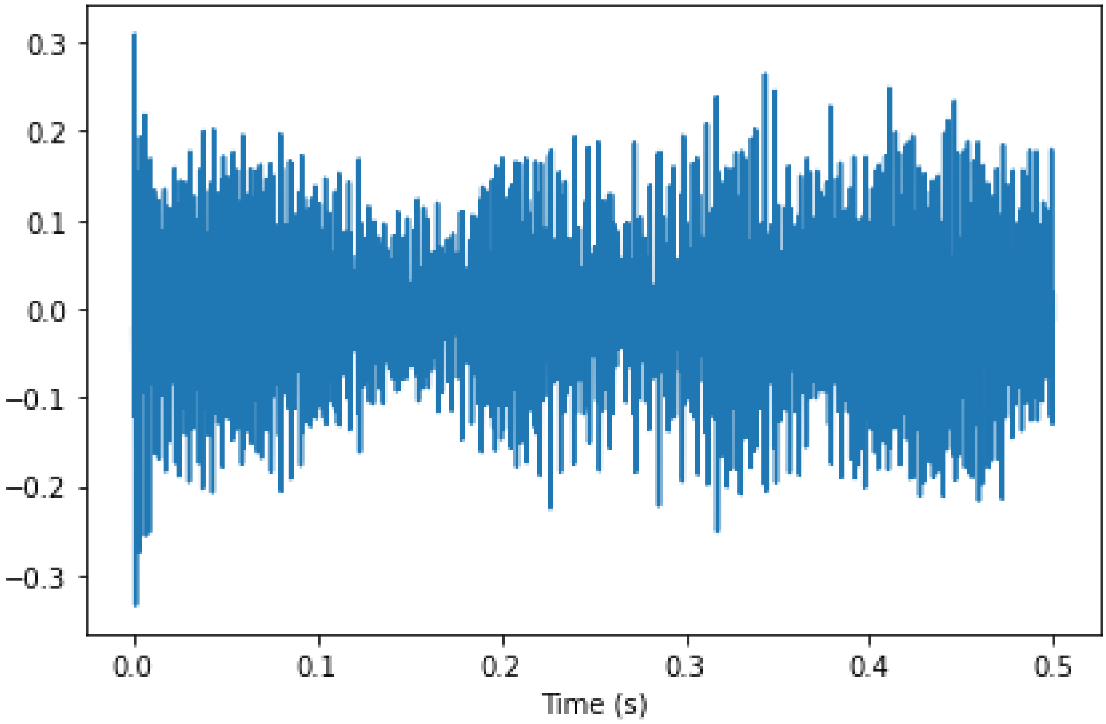

Рисунок 1.7. График преобразованного сигнала

Проведем сравнение изначального среза и отфильтрованного
Изначальный срез

```
segment.make_audio()
```

Отфильтрованный срез

```
wave2.make_audio()
```

В итоге получаем довольно "плоский" звук, лишенный объема.

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 SinSignal и CosSignal

```
cos_sig = CosSignal(freq=103, amp=1.0, offset=0)
sin_sig = SinSignal(freq=206, amp=0.8, offset=0)
cos_sig2 = CosSignal(freq=412, amp=0.3, offset=0)
m_sig = cos_sig + sin_sig + cos_sig2Построимграфиксуммы


\begin{lstlisting}[language=Python]
m_sig.plot()
```

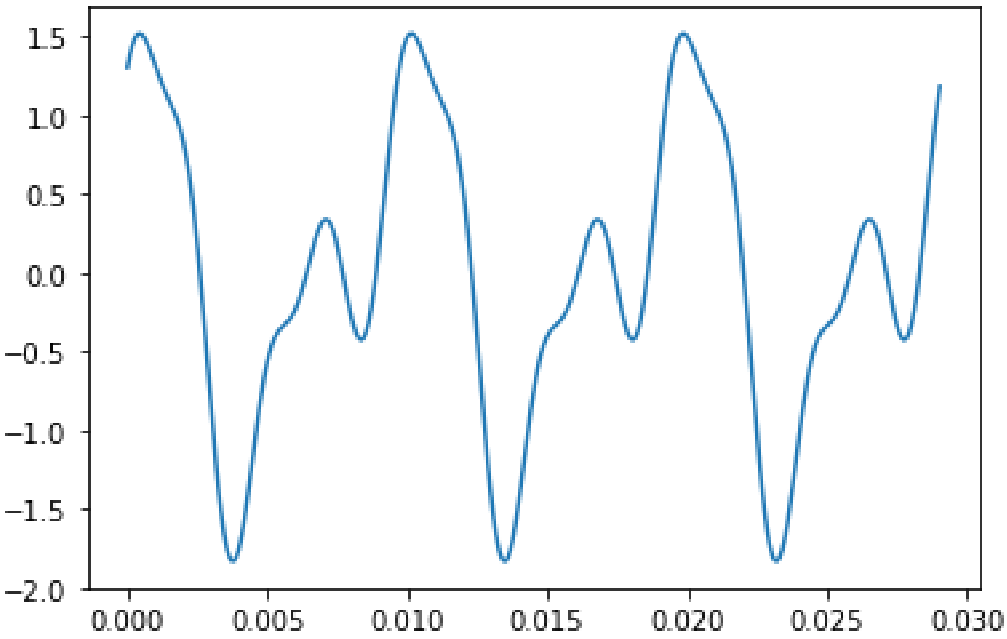

Рисунок 1.8. Графиксуммы сигналов

Сделаем wave и послушаем

```
wave = m_sig.make_wave()

wave.make_audio()
```

Спектр

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

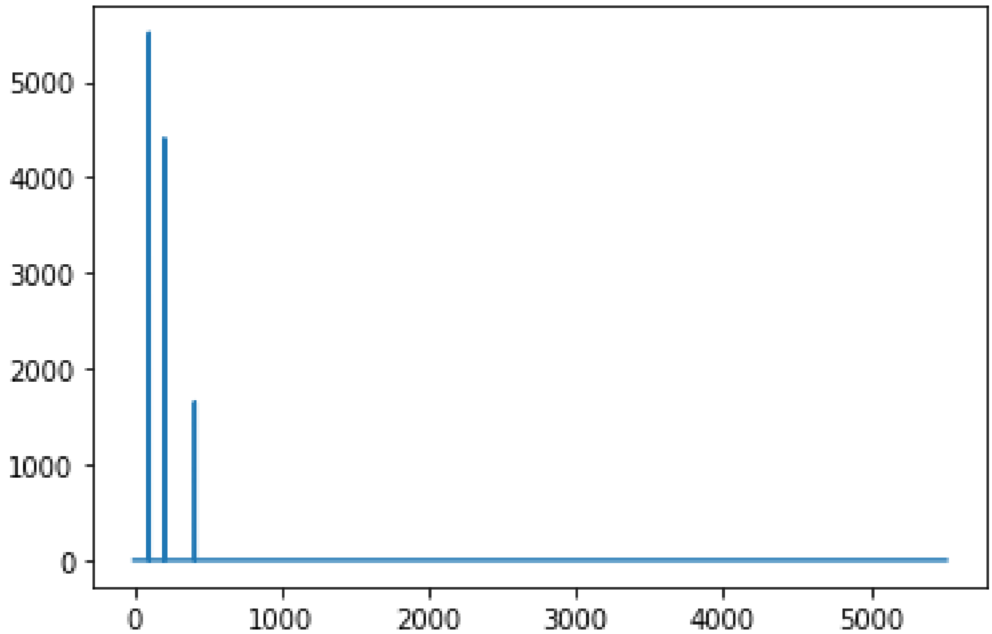

Рисунок 1.9. Спектр сигнала

Добавим к этой волне ещё частотный компонент, не кратный основным

```
wave2 = (m_sig + SinSignal(freq=250, amp=0.5, offset=0)).make_wave()
wave2.make_audio()

wave2.make_spectrum().plot()
```

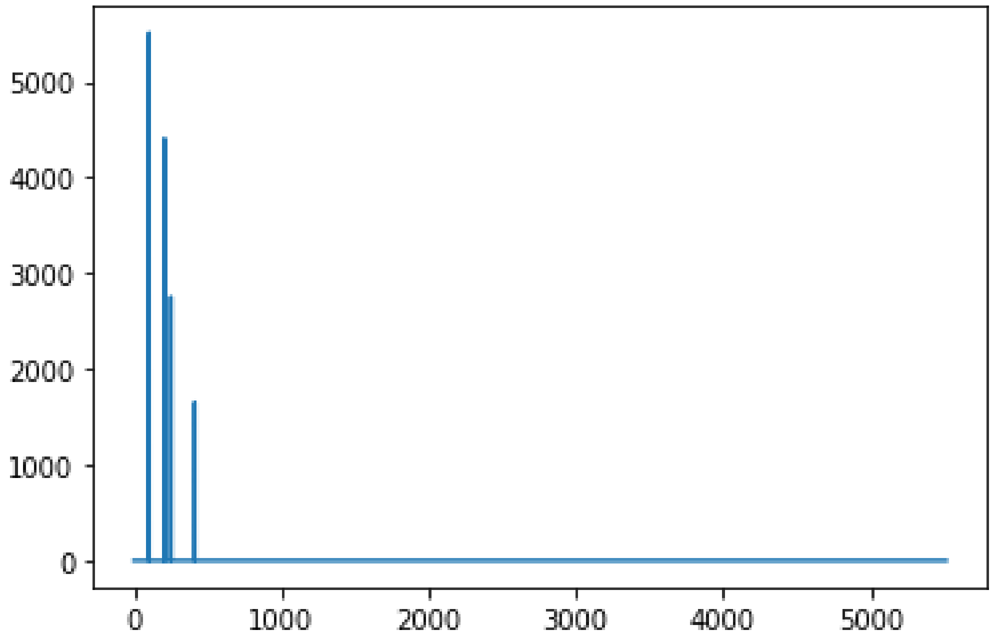

Рисунок 1.10. График после добавления частотного компонента

Звук перестал быть однотонным, появился "дребезг"

1.3. Упражнение 3

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

Создадим функцию stretch, которая будет замедлять или ускорять сигнал в зависимости от коэффициента измнения

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

wave = read_wave('440931__xhale303__piano-loop-1.wav')
wave.make_audio()
```

Проверим функцию

```
wave2 = wave
stretch(wave2, 2)
wave2.make_audio()
```

Звук замедлился в 2 раза

1.4. Вывод

В ходе данной работы было выполнено знакомство с основыми понятиями при работе со звуками и сигналами. При помощи библиотеки thinkDSP открывается множество возможностей по взаимодействию с сингалами, таких как их созданию, так и для их обработки

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 трeугольными с прямоугольными сигналами?

Создадим класс SawtoothSignal:

```
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
```

Сделаем пилообразный сигнал и вычислим спектр

```
saw = SawtoothSignal()
saw.plot()
saw_wave = saw.make_wave(duration=1, framerate=10000)
decorate(xlabel='Time (s)')
```

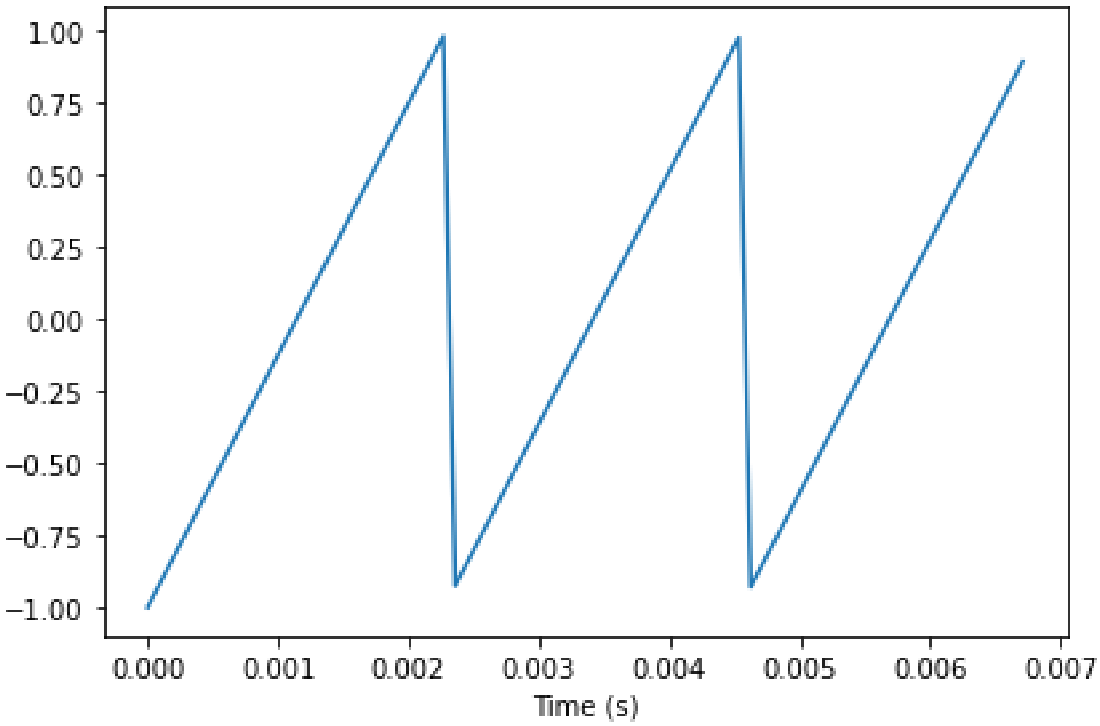

Рисунок 2.1. График пилообразного сигнала

Вычислим спектр:

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

```
decorate(xlabel='Frequency (Hz)')
```

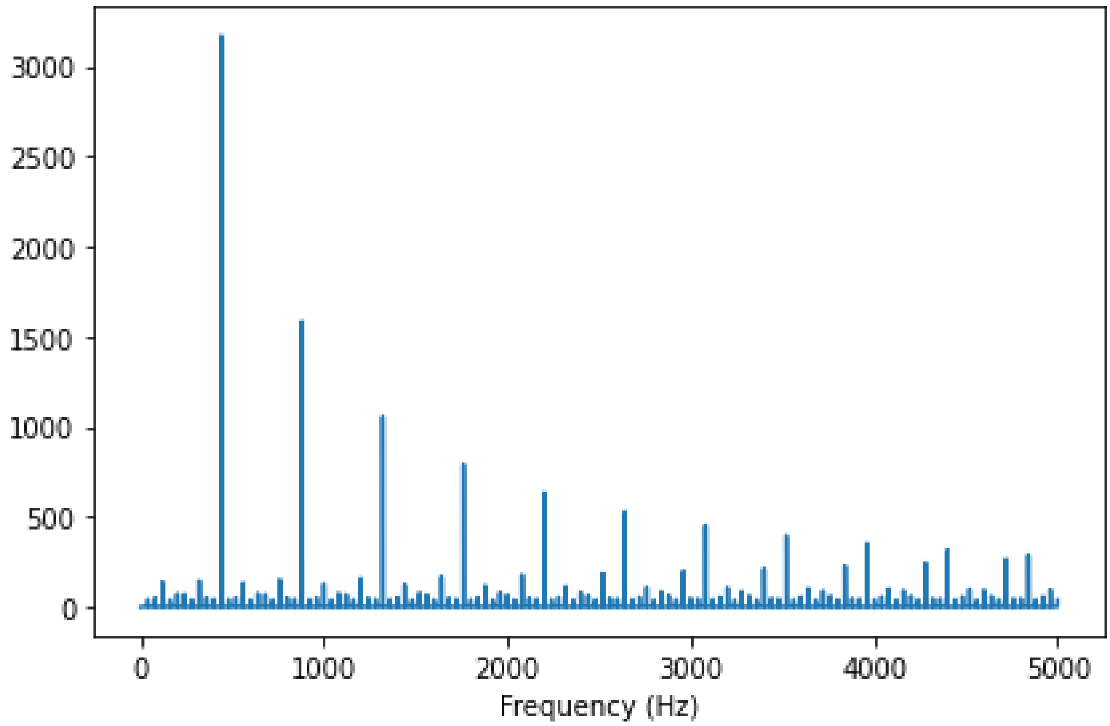

Рисунок 2.2. График спектра пилообразного сигнала

Также, для сравнения, создадим прямоугольный и треугольный сигнал

```
square = SquareSignal()
square.plot()
decorate(xlabel='Time (s)')
```

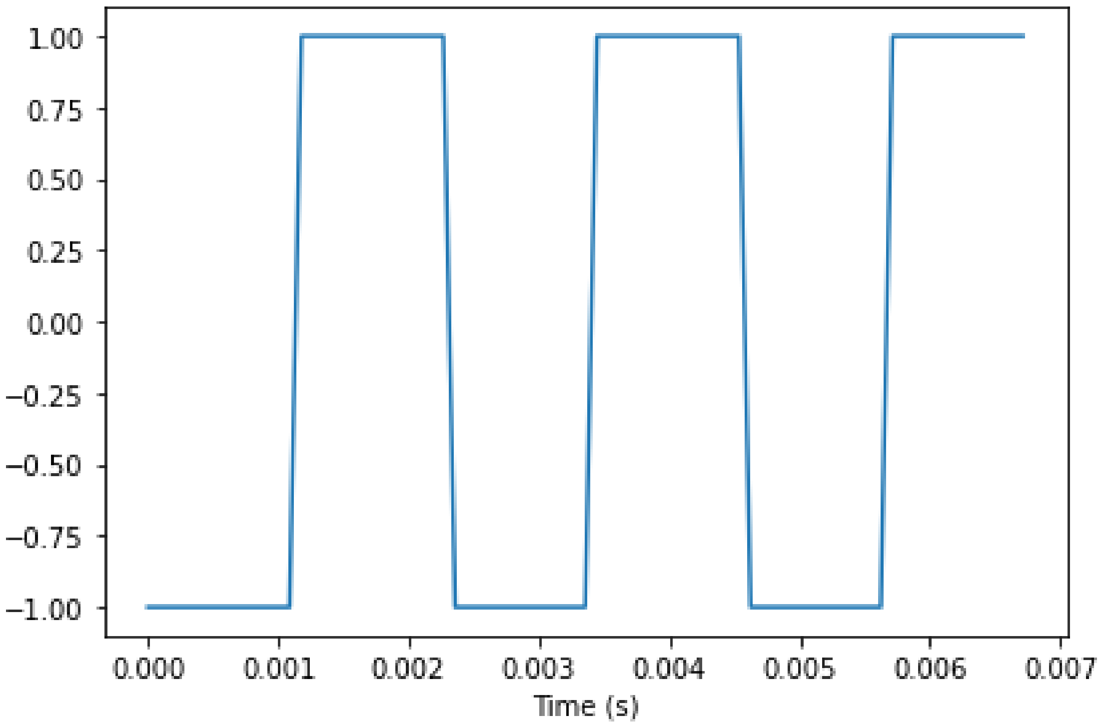

Рисунок 2.3. График прямоугольного сигнала

```
square.make_wave(duration=1, framerate=10000).make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

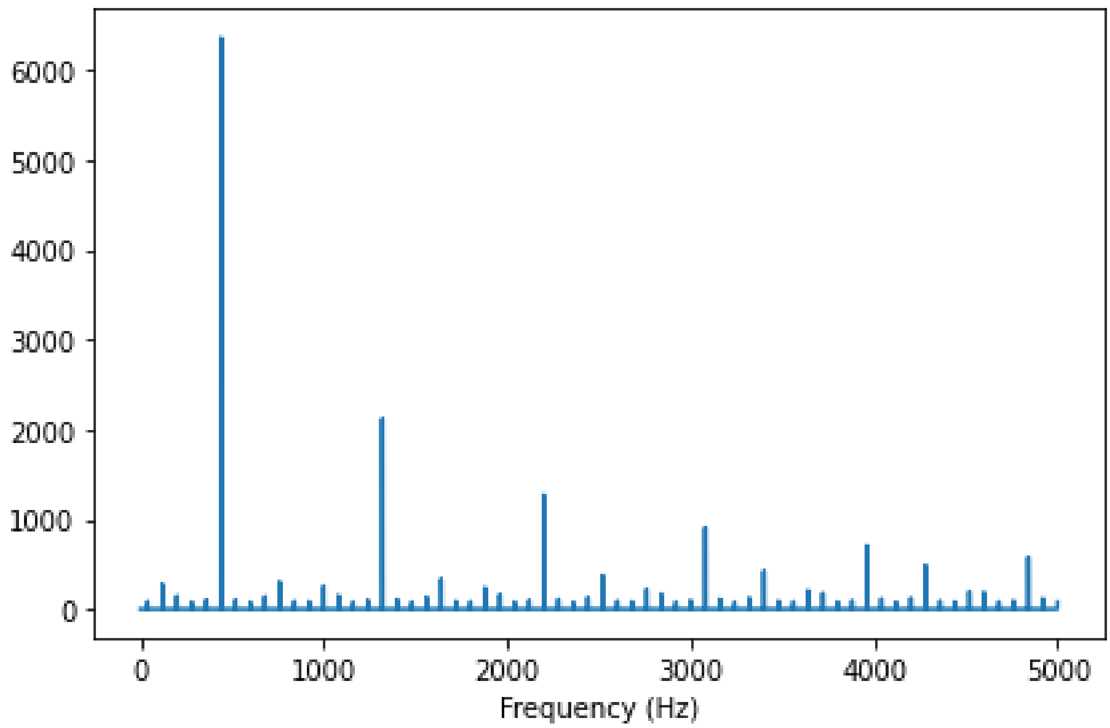

Рисунок 2.4. График спектра прямоугольного сигнала

```
tria = TriangleSignal()
tria.plot()
decorate(xlabel='Time (s)')
```

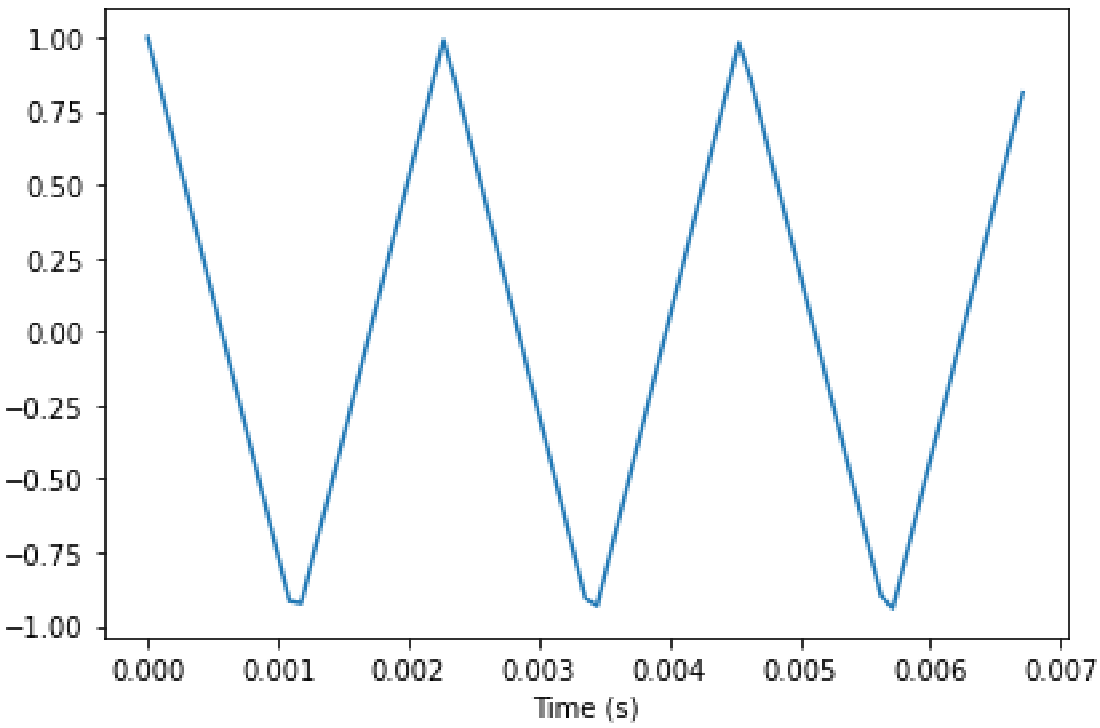

Рисунок 2.5. График треугольного сигнала

```
tria.make_wave(duration=1, framerate=10000).make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

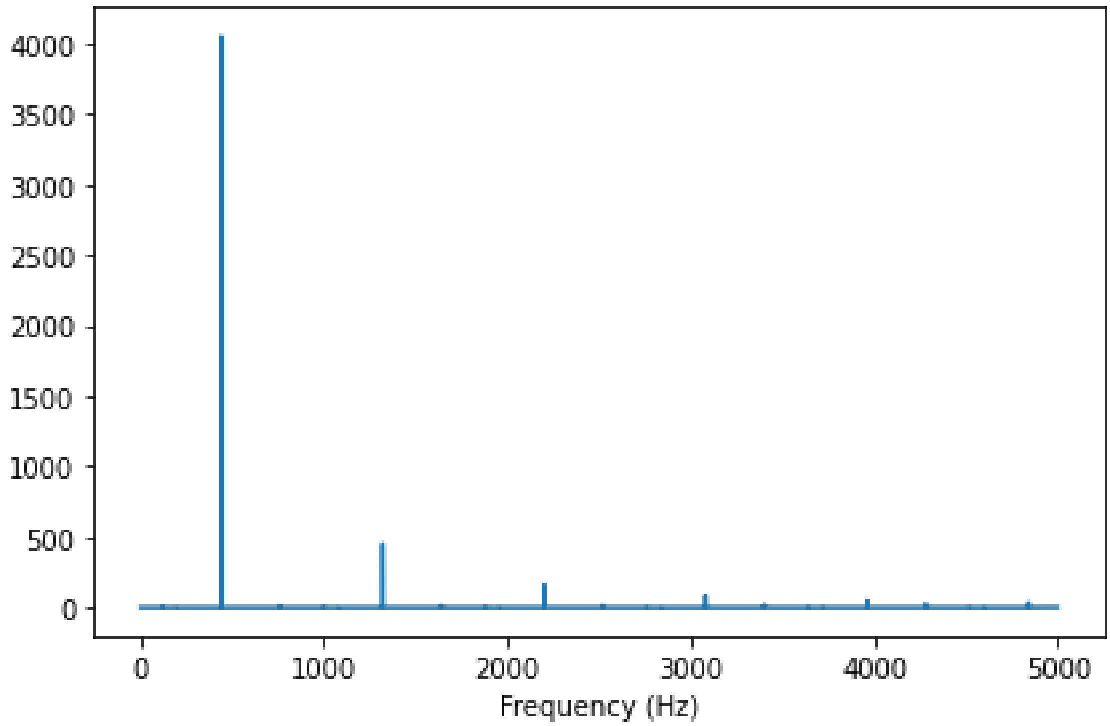

Рисунок 2.6. График спектра треугольного сигнала

У треугольного сигнала нечетный гармоники, а амплитуда спадает пропорцинально квадрату частоты. У прямоугольного сигнала тоже только нечетный гармоники, спад по амплитуде пропорционален частоте. Пилообраный же сигнал падает пропорционально частоте и имеет как четные, так и нечетные гармоники.

2.2. Упражнение 2

Создайте прямоугольный 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?

Создадим прямоугольный сигнал с частотой 1100Гц

```
square = SquareSignal(1100)
sq_wave = square.make_wave(1, 0, 10000)
sq_wave.make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

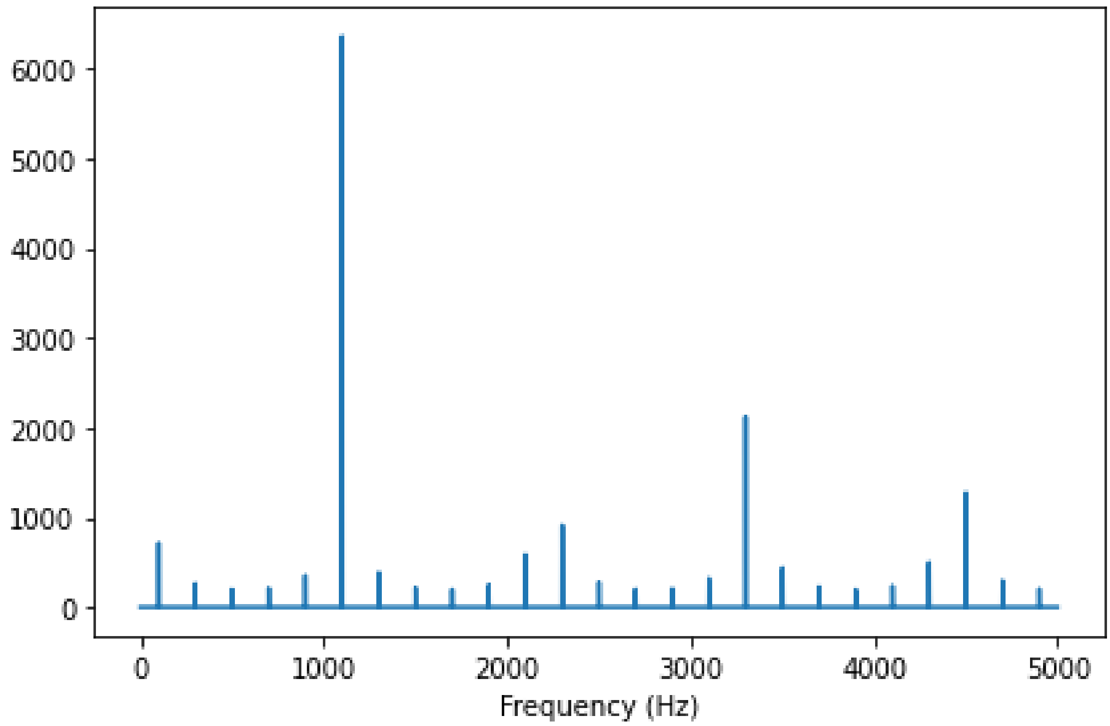

Рисунок 2.7. График прямоугольного сигнала с частотой 1100 Hz

Видим заернутось гармоний из-за биения

```
sq_wave.make_audio()
```

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, возьмем спектр и распечатаем значения spectrum.fs

```
tr = TriangleSignal(440)
tr_wave = tr.make_wave(duration=0.01)
tr_wave.plot()
decorate(xlabel='Time (s)')
```

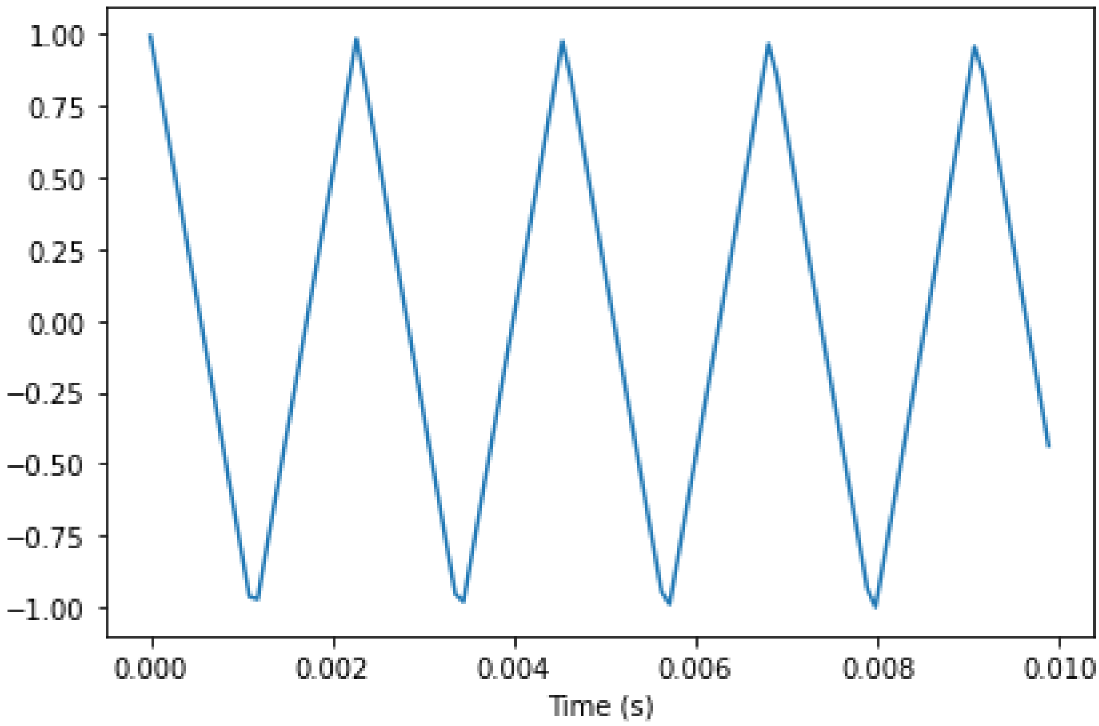

Рисунок 2.8. График треугольного сигнала с частотой 440 Hz

Убедимся, что нулевой элемент равен 0

```
tr_sp = tr_wave.make_spectrum()
tr_sp.hs[0]

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

Изменим его на значение 100

```
tr_sp.hs[0] = 100
tr_sp.make_wave().plot()
decorate(xlabel='Time (s)')
```

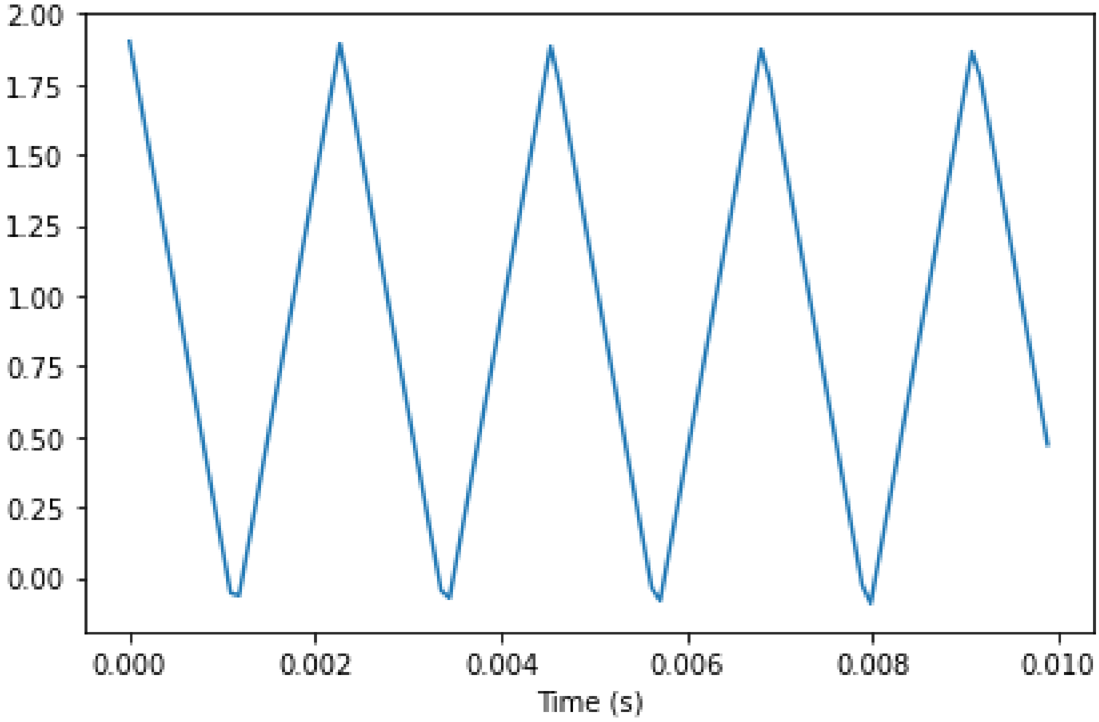

Рисунок 2.9. Обновленный график сигнала

Получили смещение относительно вертикальной оси.

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?

Создадим функцию для деления spectrum.hs на соответствующую spectrum.fs

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

Проверим на пилообразном сигнале

```
saw = SawtoothSignal()
saw_wave = saw.make_wave(duration=0.2, framerate=10000)
saw_wave.make_audio()

saw_sp = saw_wave.make_spectrum()
saw_sp.plot()
decorate(xlabel='Frequency (Hz)')
```

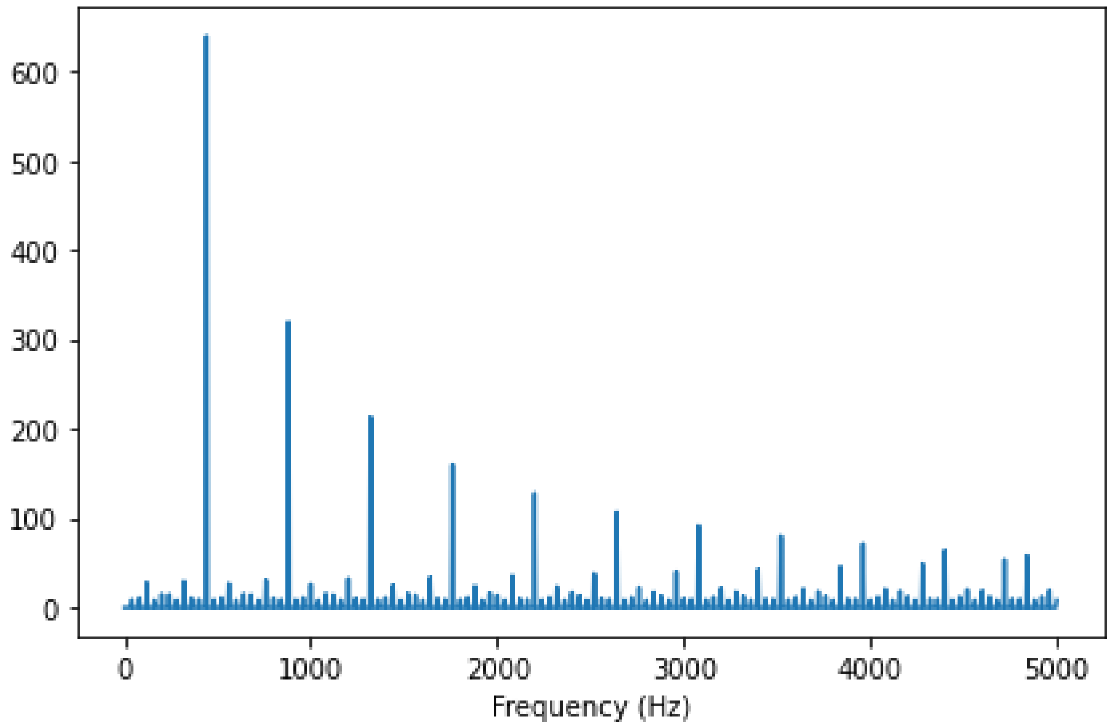

Рисунок 2.10. Спектр сигнала

Применим функцию

```
spec_div(saw_sp)
saw_sp.plot()
decorate(xlabel='Frequency (Hz)')
```

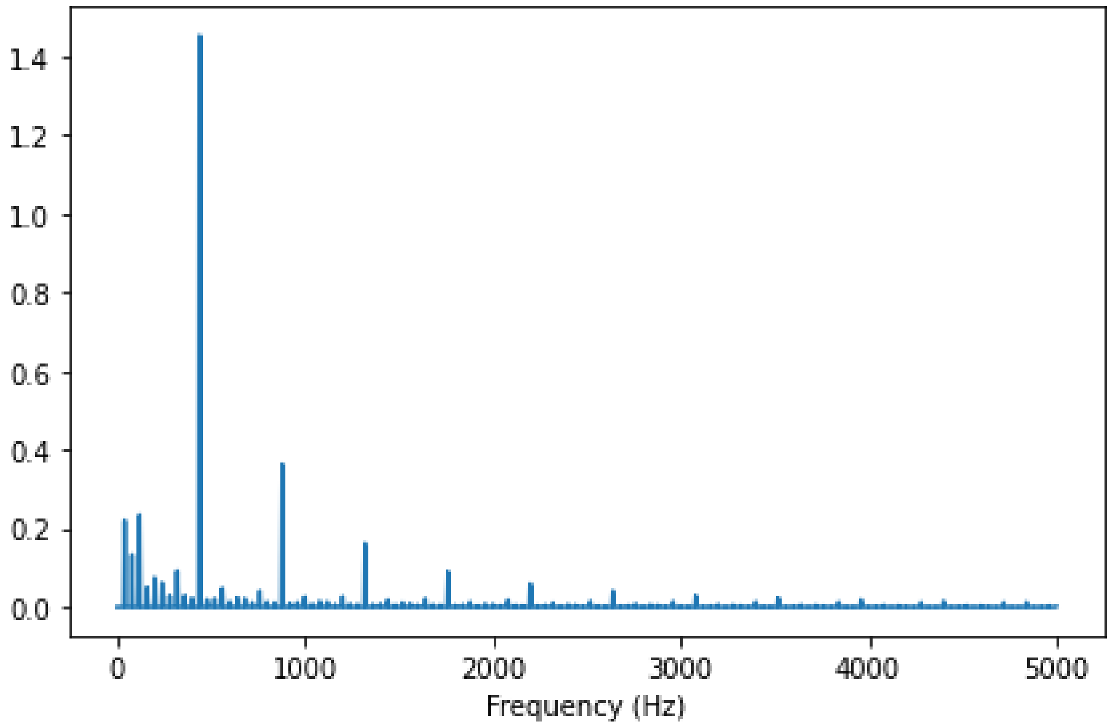

Рисунок 2.11. Спектр измененного сигнала

```
saw_sp.make_wave().make_audio()
```

Звук стал ниже

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, который состоит из четных и нечетных гармоник, а также эти гармоники падают пропорционально квадрату частоты.

Возьмем пилообраный сигнал, так как у него четные и нечетные гармоники, надо его свести лишь к уменьшению гамони кропорционально квадрату

```
saw = SawtoothSignal()
saw_wave = saw.make_wave(duration=0.5, framerate=20000)
saw_sp = saw_wave.make_spectrum()
saw_sp.plot()
decorate(xlabel='Frequency (Hz)')
```

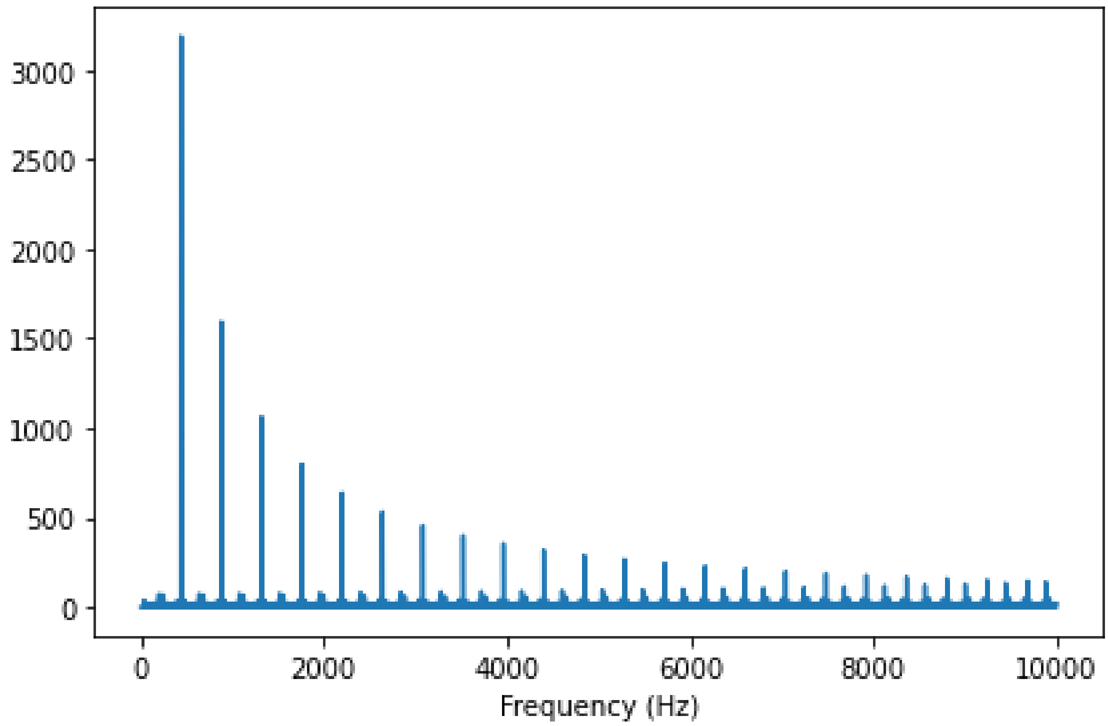

Рисунок 2.12. Спектр пилообразного сигнала

Применяем функцию из прошлого упражнения

```
spec_div(saw_sp)
saw_sp.plot()
decorate(xlabel='Frequency (Hz)')
```

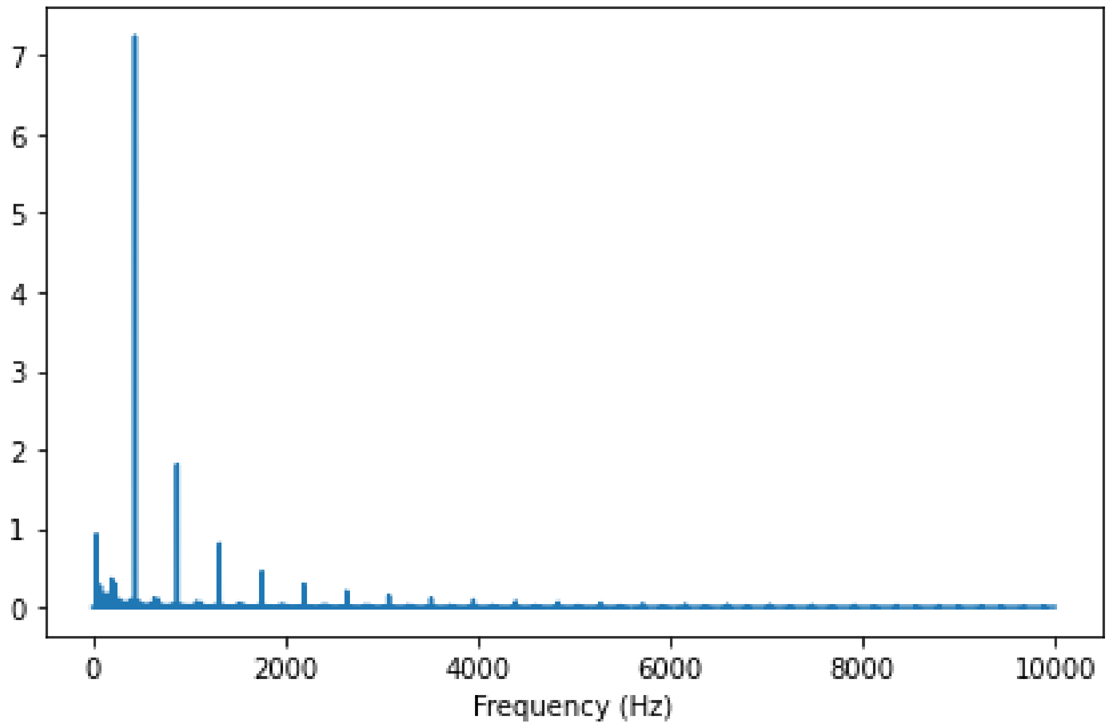

Рисунок 2.13. Спектр измененного сигнала

Получаем, что амплитуда спадает пропорцинально квадрату частоты, имеются четные и нечетные гармоники.

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.

Если длительность кратна периоду, то начало и конец отрезка совпадают, и мы получаем минимальную утечку.

```
from thinkdsp import SinSignal

signal = SinSignal(freq=440)
duration = signal.period * 30
wave = signal.make_wave(duration)
wave.plot()
decorate(xlabel='Time (s)')

spectrum.plot(high=880)
```

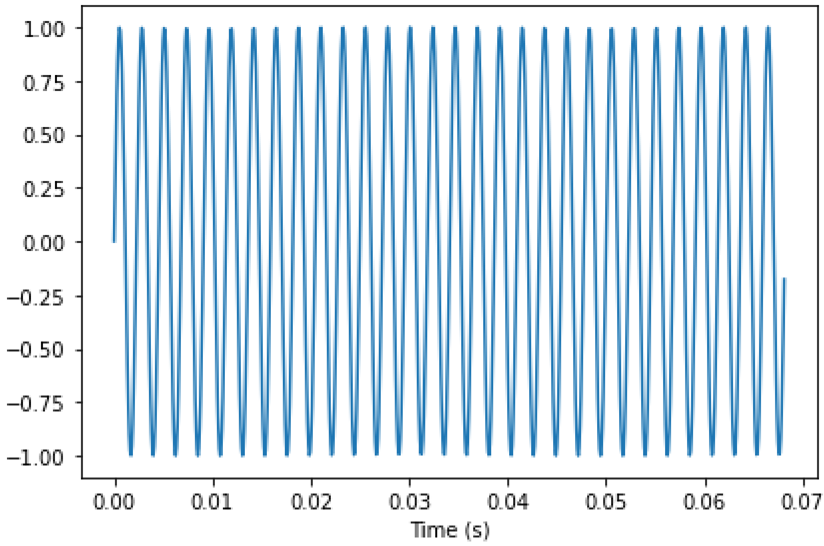

Рисунок 3.1. Рассматриваемый сигнал

```
spectrum = wave.make_spectrum()
spectrum.plot(high=880)
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

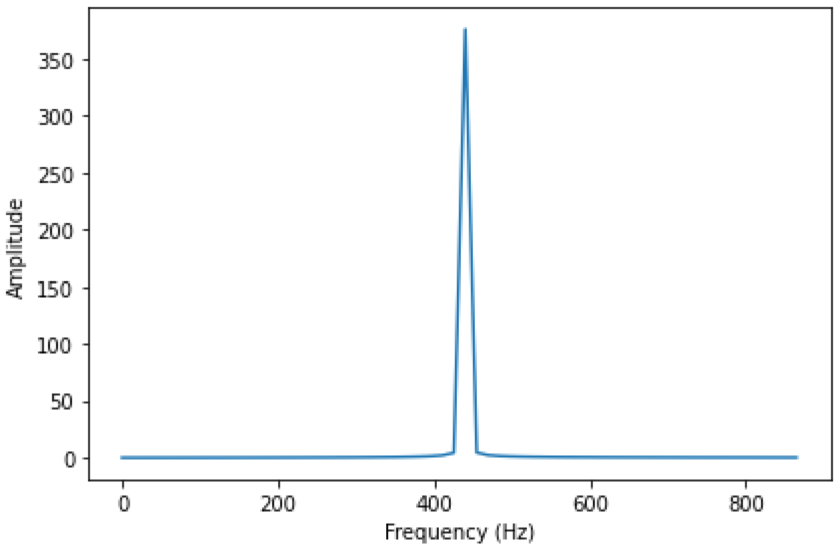

Рисунок 3.2. Спектр рассматриваемого сигнала

Если продолжительность не кратна периоду, утечка довольно плохая.

```
duration = signal.period * 30.25
wave = signal.make_wave(duration)
wave.plot()
decorate(xlabel='Time (s)')
```

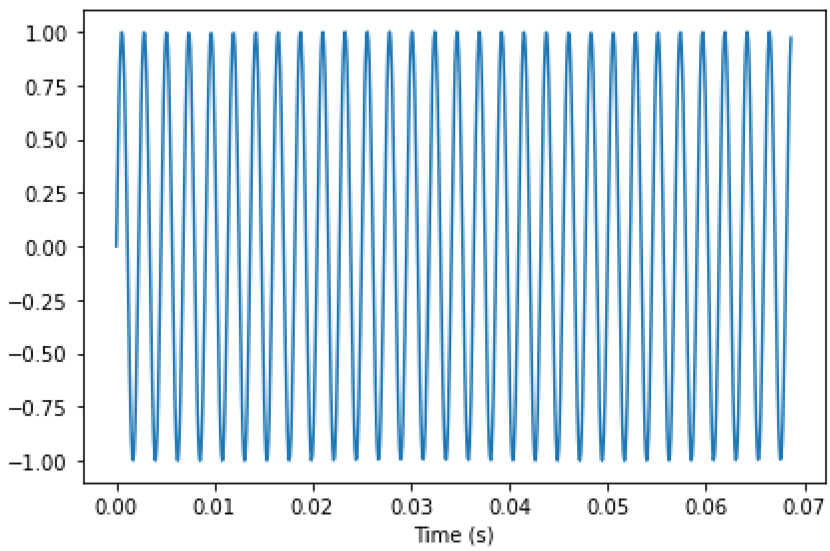

Рисунок 3.3. Рассматриваемый сигнал

```
spectrum = wave.make_spectrum()
spectrum.plot(high=880)
decorate(xlabel='Frequency (Hz)')
```

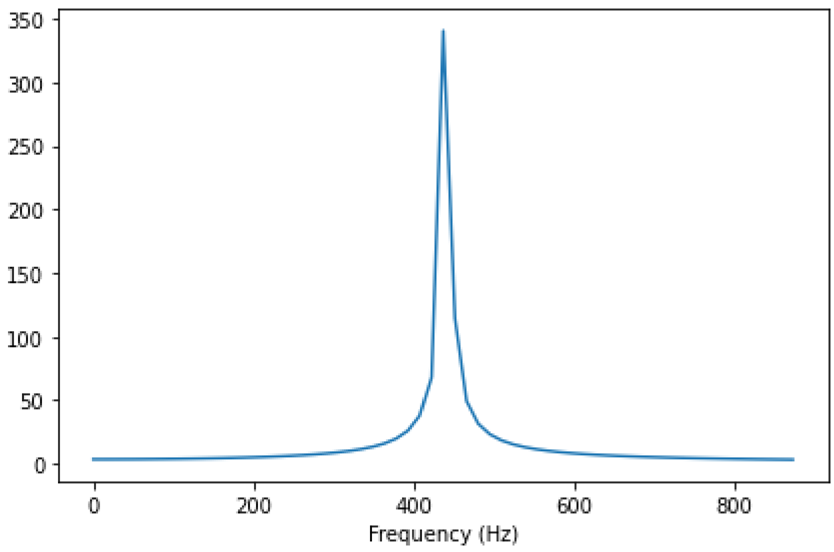

Рисунок 3.4. Спектр рассматриваемого сигнала

Работа с окнами помогает (но обратите внимание, что она снижает общую энергию).

```
wave.hamming()
spectrum = wave.make_spectrum()
spectrum.plot(high=880)
decorate(xlabel='Frequency (Hz)')
```

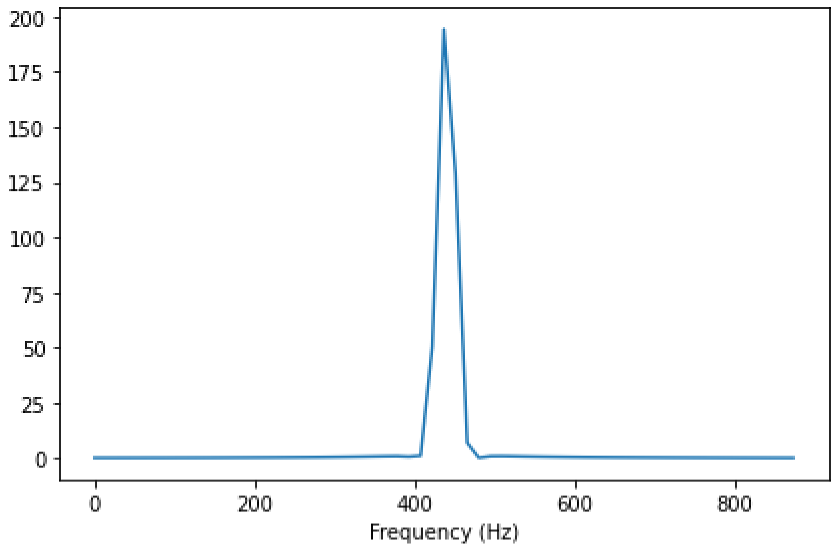

Рисунок 3.5. Спектр сигнала с применение окна Хээминга

Если вы вслепую вычислите ДПФ непериодического сегмента, вы получите «размытие движения».

```
signal = Chirp(start=220, end=440)
wave = signal.make_wave(duration=1)
spectrum = wave.make_spectrum()
spectrum.plot(high=700)
decorate(xlabel='Frequency (Hz)')
```

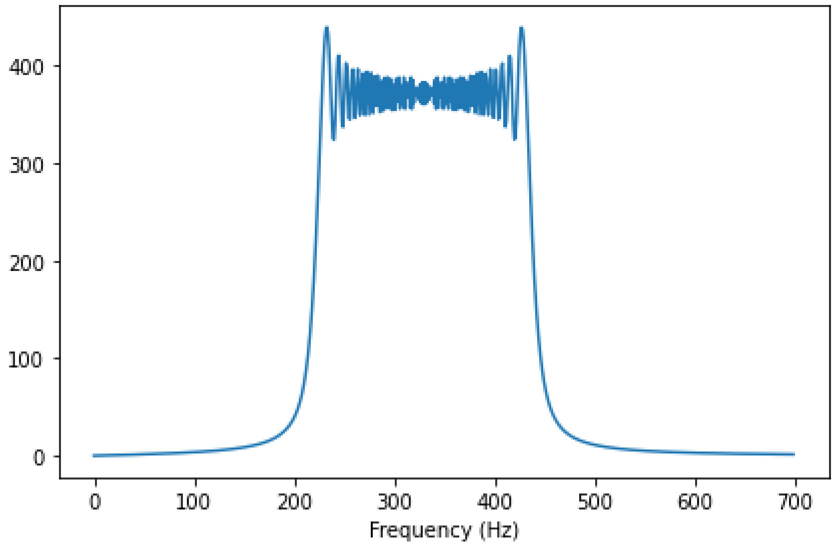

Рисунок 3.6. Спектр сигнала

Спектрограмма — это визуализация кратковременного ДПФ, позволяющая увидеть, как спектр меняется во времени.

```
def plot_spectrogram(wave, seg_length):
    """
    """
    spectrogram = wave.make_spectrogram(seg_length)
    print('Time resolution (s)', spectrogram.time_res)
    print('Frequency resolution (Hz)', spectrogram.freq_res)
    spectrogram.plot(high=700)
    decorate(xlabel='Time(s)', ylabel='Frequency (Hz)')

signal = Chirp(start=220, end=440)
wave = signal.make_wave(duration=1, framerate=11025)
plot_spectrogram(wave, 512)

Time resolution (s) 0.046439909297052155
Frequency resolution (Hz) 21.533203125
```

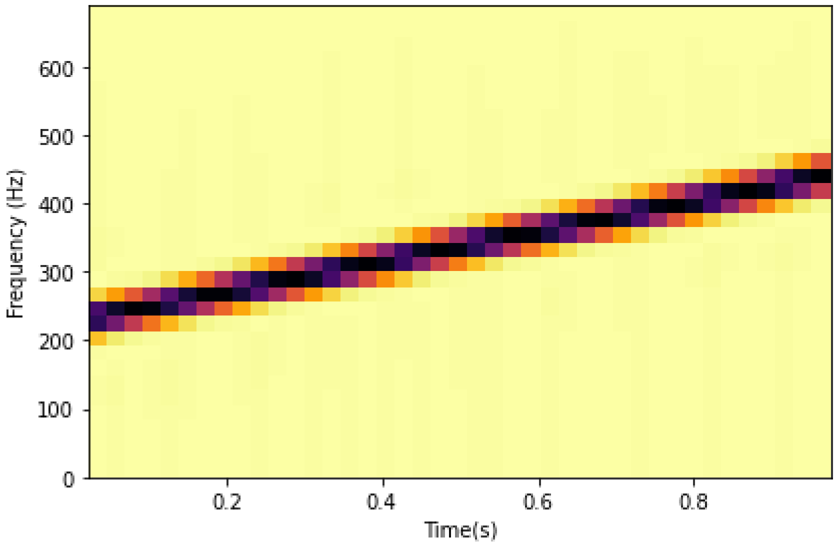

Рисунок 3.7. КПФ сигнала

Если вы увеличите длину сегмента, вы получите лучшее разрешение по частоте, худшее разрешение по времени.

```
plot_spectrogram(wave, 1024)

Time resolution (s) 0.09287981859410431
Frequency resolution (Hz) 10.7666015625
```

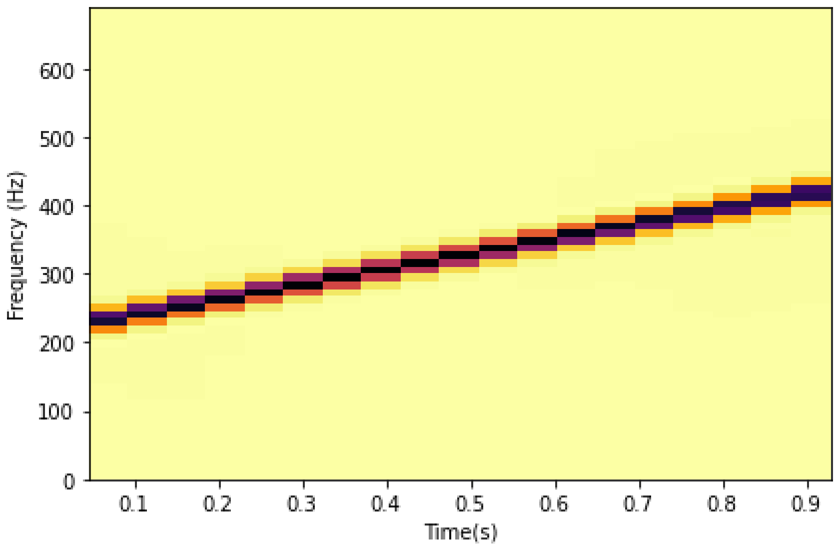

Рисунок 3.8. КПФ измененного сигнала

Если вы уменьшите длину сегмента, вы получите лучшее временное разрешение, худшее разрешение по частоте.

```
plot_spectrogram(wave, 256)

Time resolution (s) 0.023219954648526078
Frequency resolution (Hz) 43.06640625
```

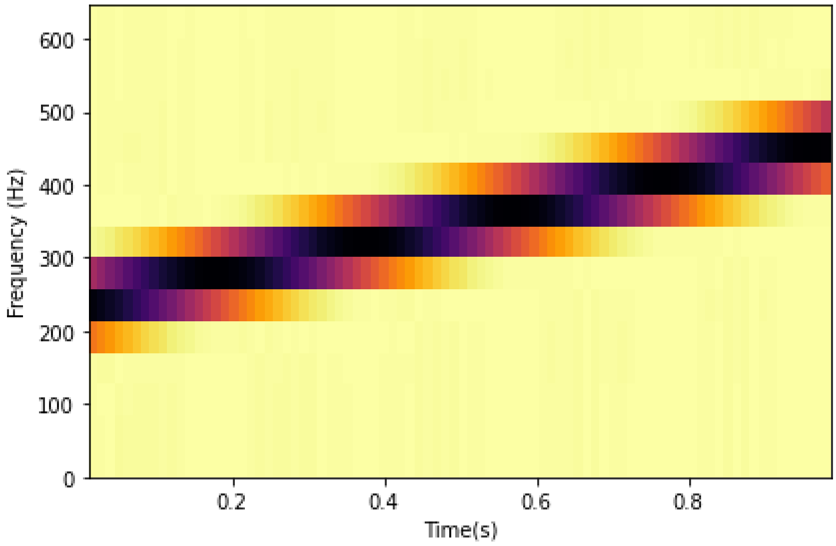

Рисунок 3.9. КПФ измененного сигнала

3.2. Упражнение 2

Напишем класс SawtoothChirp

```
from thinkdsp import Chirp
from thinkdsp import normalize, unbias

class SawtoothChirp(Chirp):

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

Создадим сигнал

```
sig = SawtoothChirp(100, 2000)
w = sig.make_wave(duration = 6, framerate = 10000)
w.apodize()
w.make_audio()
```

Можем услышать биение, напечатаем эскиз спектрограммы

```
sp = w.make_spectrogram(seg_length = 1000)
sp.plot(high = 7000)
```

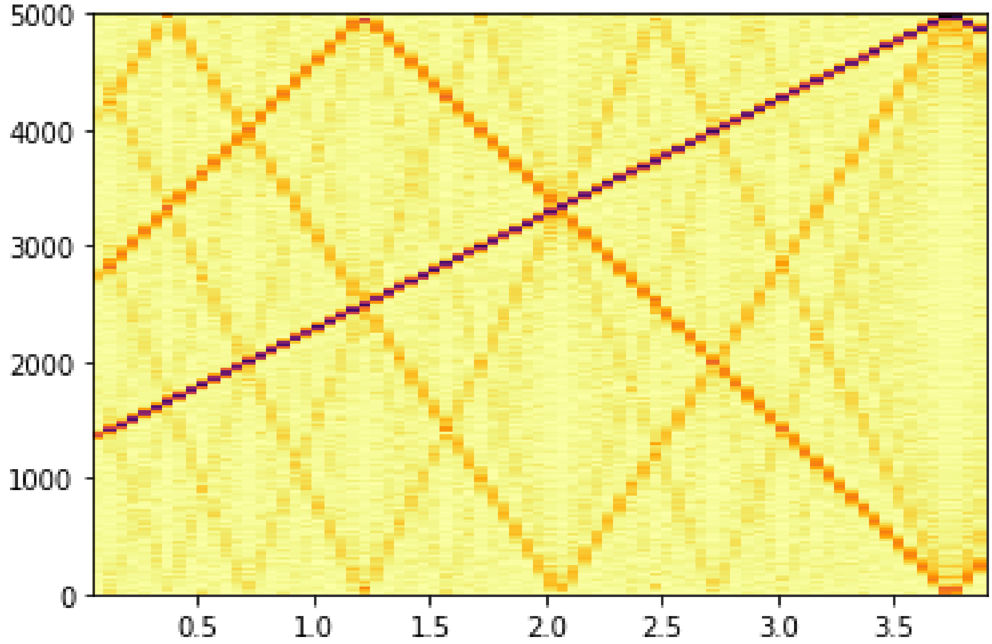

Рисунок 3.10. КПФ сигнала

3.3. Упражнение 3

Создаем пилообразный чирп с изменением от 2500 до 3000 Гц и посмотрим спектограму

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

wave.make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

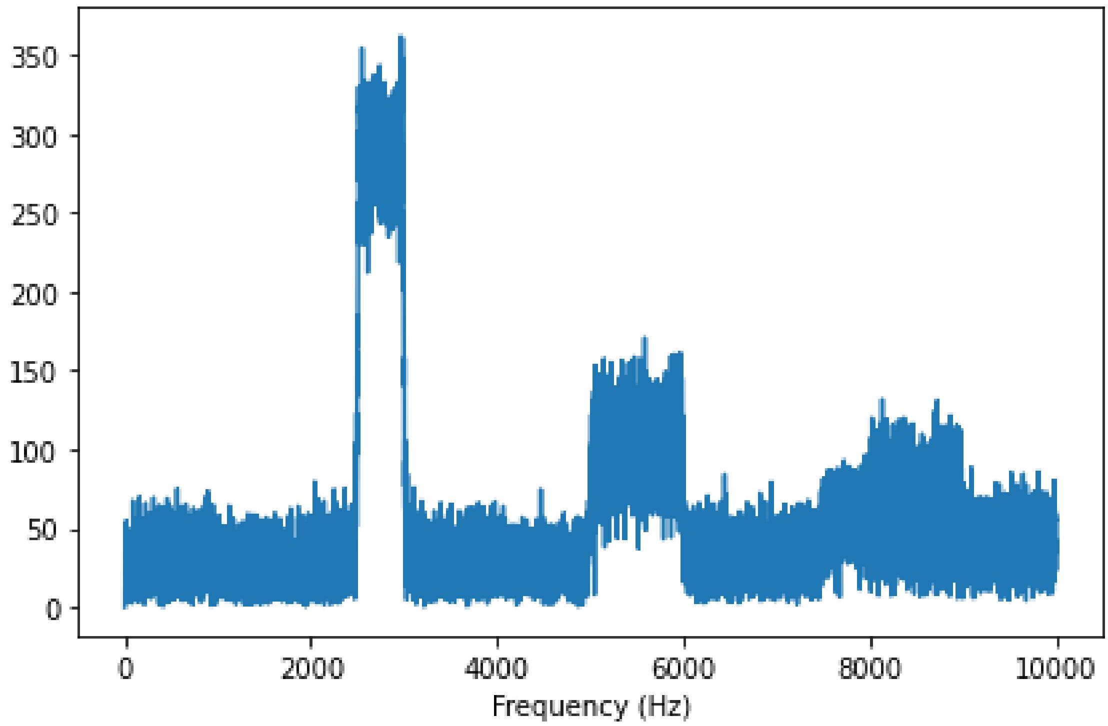

Рисунок 3.11. Спектр сигнала

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.

Загрузим глиссандо и посмотрим спектограмму

```
if not os.path.exists('411728__inspectorj__violin-glissando-ascending-a-h1.wav')
    :
    !wget https://github.com/hotnotHD/Telecom/raw/main/411728
        __inspectorj__violin-glissando-ascending-a-h1.wav

from thinkdsp import read_wave

wave = read_wave('411728__inspectorj__violin-glissando-ascending-a-h1.wav')
wave.make_audio()

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

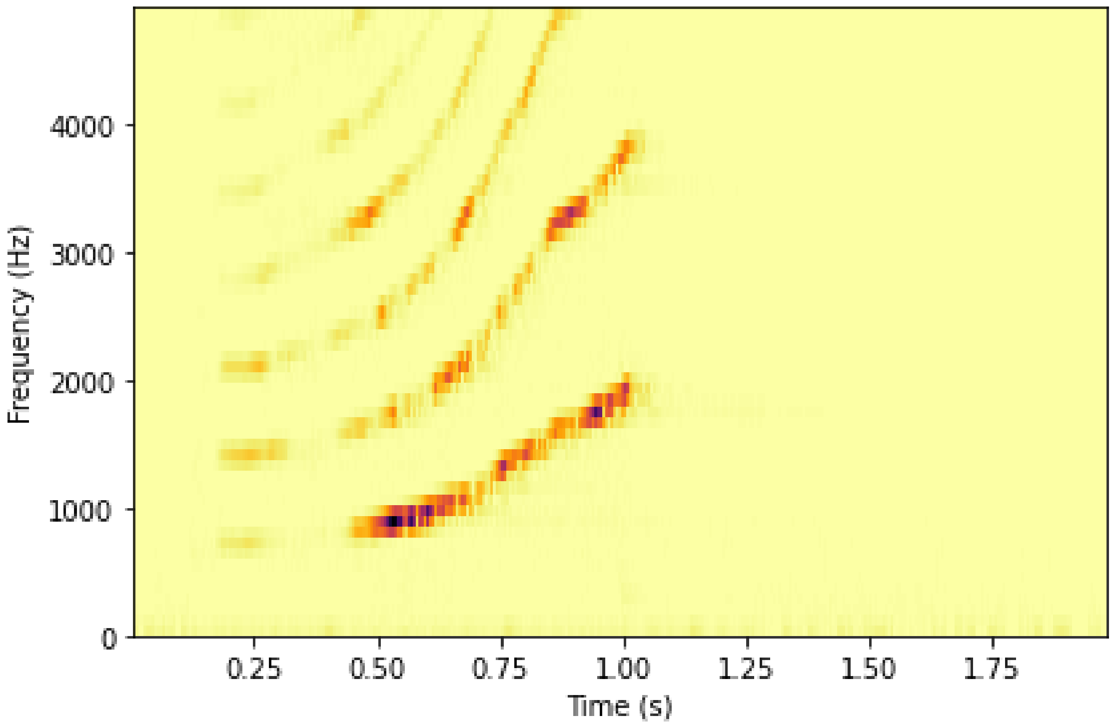

Рисунок 3.12. Спектрограмма сигнала

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.

Напишем класс TromboneGliss.

```
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 = np.pi * 2 * freqs * dts
        phases = np.cumsum(dphis)
        ys = self.amp * np.cos(phases)
        return ys
```

Создадим сигнал, имитирующий глиссандо с 262 до 349 гц и обратно.

```
sig1 = TromboneGliss(262, 349)
w1 = sig1.make_wave(duration=1)
sig2 = TromboneGliss(349, 262)
w2 = sig2.make_wave(duration=1)
w = w1 | w2
```

```
w.make_audio()

sp = w.make_spectrogram(1024)
sp.plot(high=1000)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

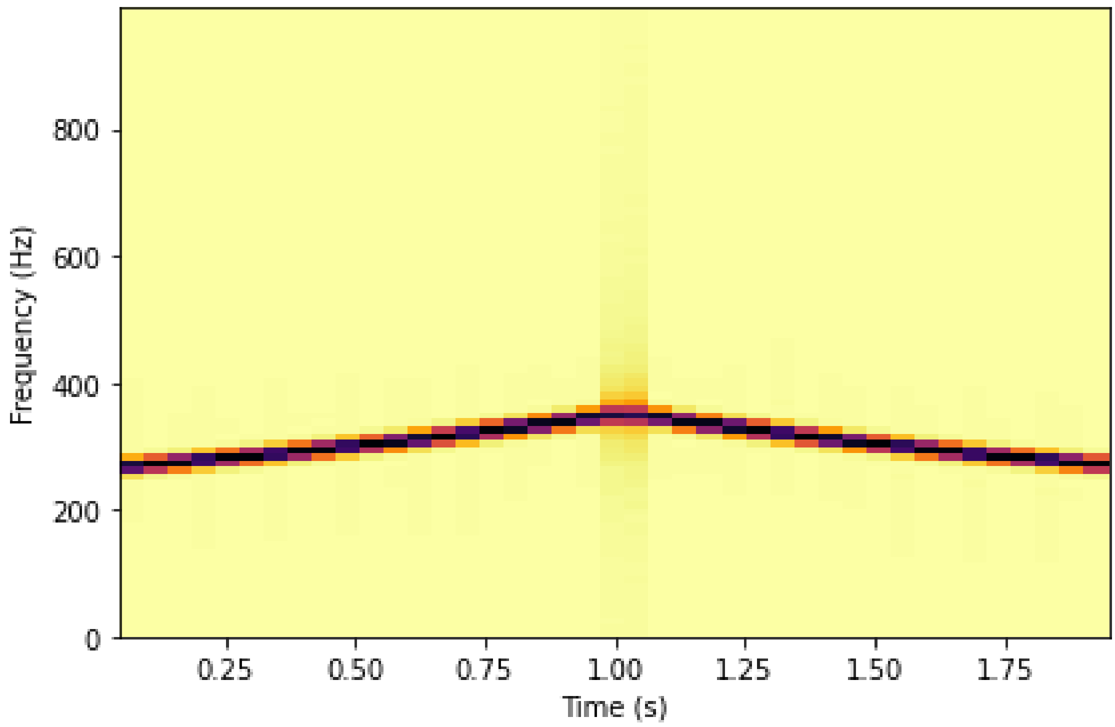

Рисунок 3.13. Спектрограмма сигнала

3.6. Упражнение 6

```
if not os.path.exists('67713__tim-kahn__b.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/67713__tim-kahn__b.wav

wave1 = read_wave('67713__tim-kahn__b.wav')
wave.make_audio()

if not os.path.exists('67714__tim-kahn__c.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/67714__tim-kahn__c.wav

wave2 = read_wave('67714__tim-kahn__c.wav')
wave2.make_audio()

wave.make_spectrogram(1024).plot(high=1000)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

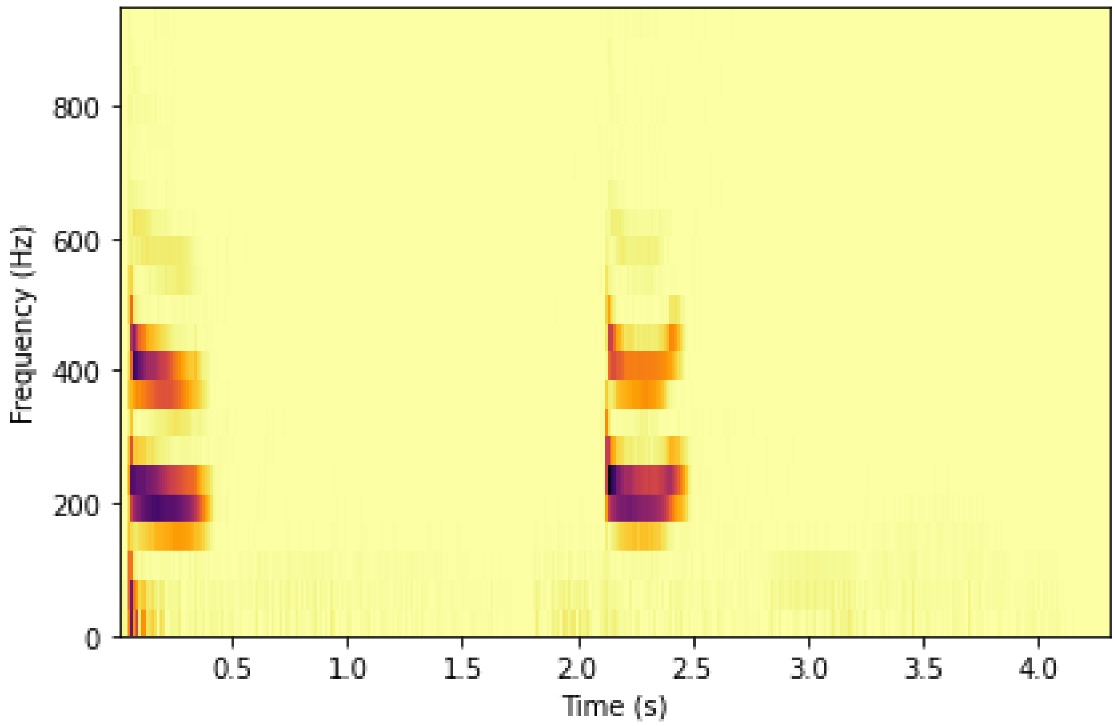

Рисунок 3.14. Спектрограмма гласных звуков

Частоты звука "b"

```
seg = wave.segment(start=0, duration=1)
seg.make_spectrum().plot(high=1000)
decorate(xlabel='Frequency (Hz)')
```

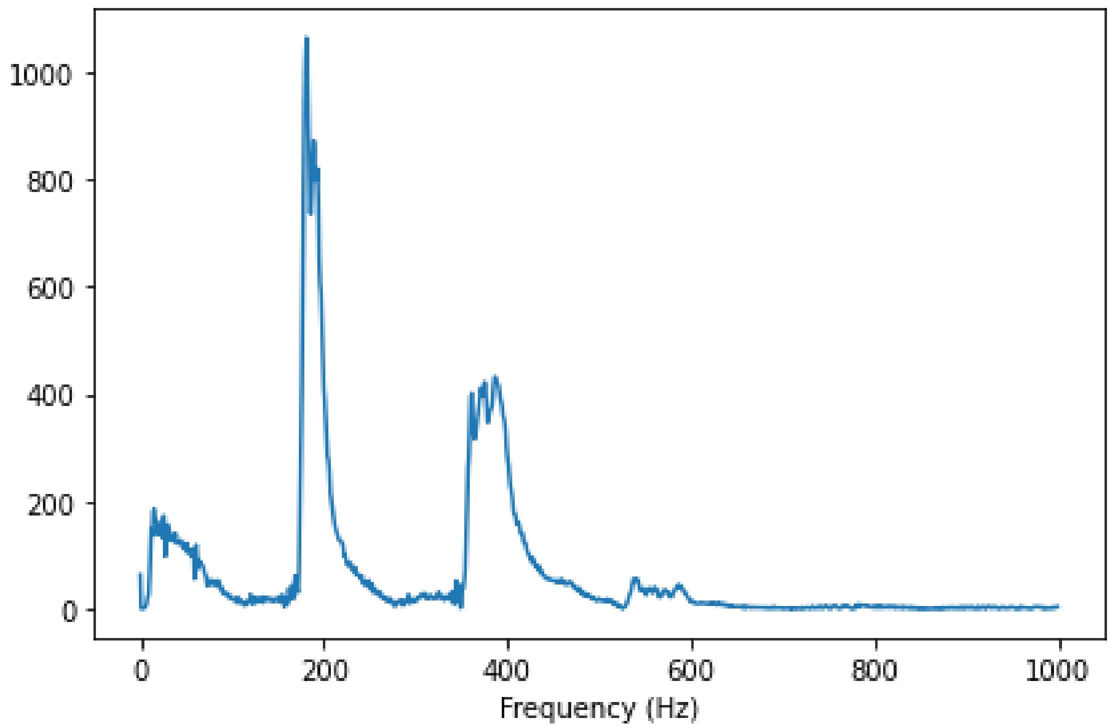

Рисунок 3.15. График частот для звука "b"

Частоты звука "с"

```
seg2 = wave.segment(start=2, duration=3)
seg2.make_spectrum().plot(high=1000)
decorate(xlabel='Frequency (Hz)')
```

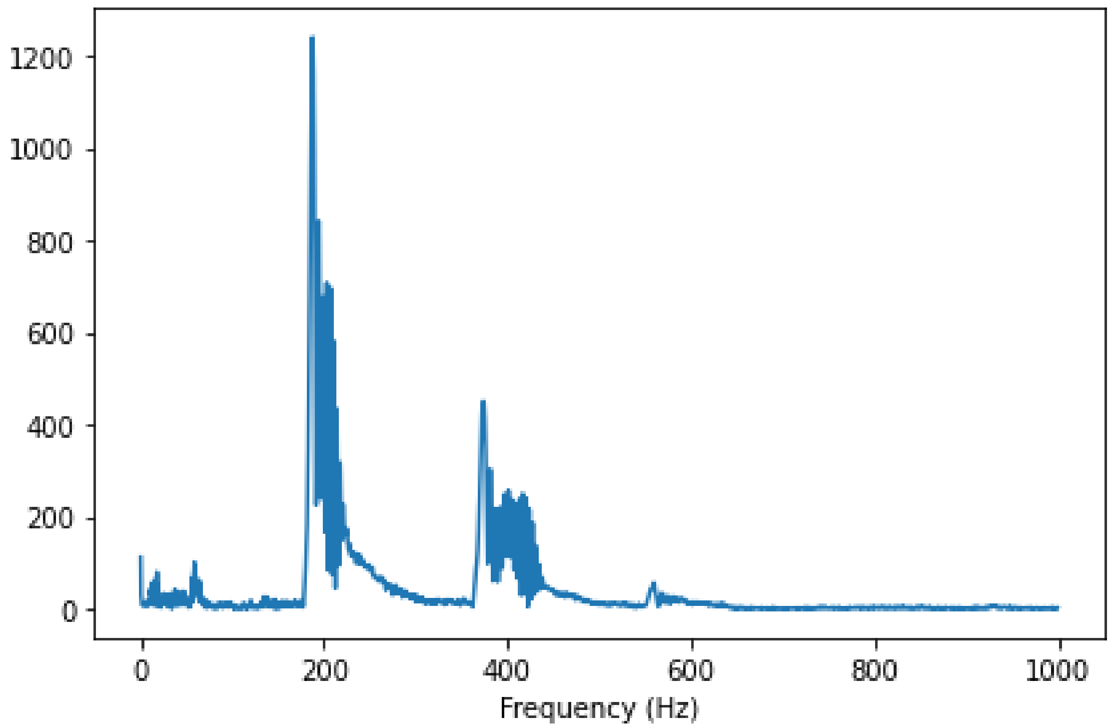

Рисунок 3.16. График частот для звука "с"

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 некоторые звуки шумов и рассмотрим

```

if not os.path.exists('457318__stek59__autumn-wind-and-dry-leaves.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/457318__stek59__autumn-wind-and-dry-leaves.wav
if not os.path.exists('518863__idomusics__rain.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/518863__idomusics__rain.wav
```

Взяли звуки шума дождя и ветра

```
wave_wind = read_wave('457318__stek59__autumn-wind-and-dry-leaves.wav')
wave_wind = wave_wind.segment(start = 1, duration = 5)
wave_wind.make_audio()

wave_rain = read_wave('518863__idomusics__rain.wav')
wave_rain = wave_rain.segment(start = 0, duration = 4)
wave_rain.make_audio()


spec = wave_wind.make_spectrum()
spec.plot_power(high = 200)
decorate(xlabel='Frequency (Hz)', ylabel='Power')
```

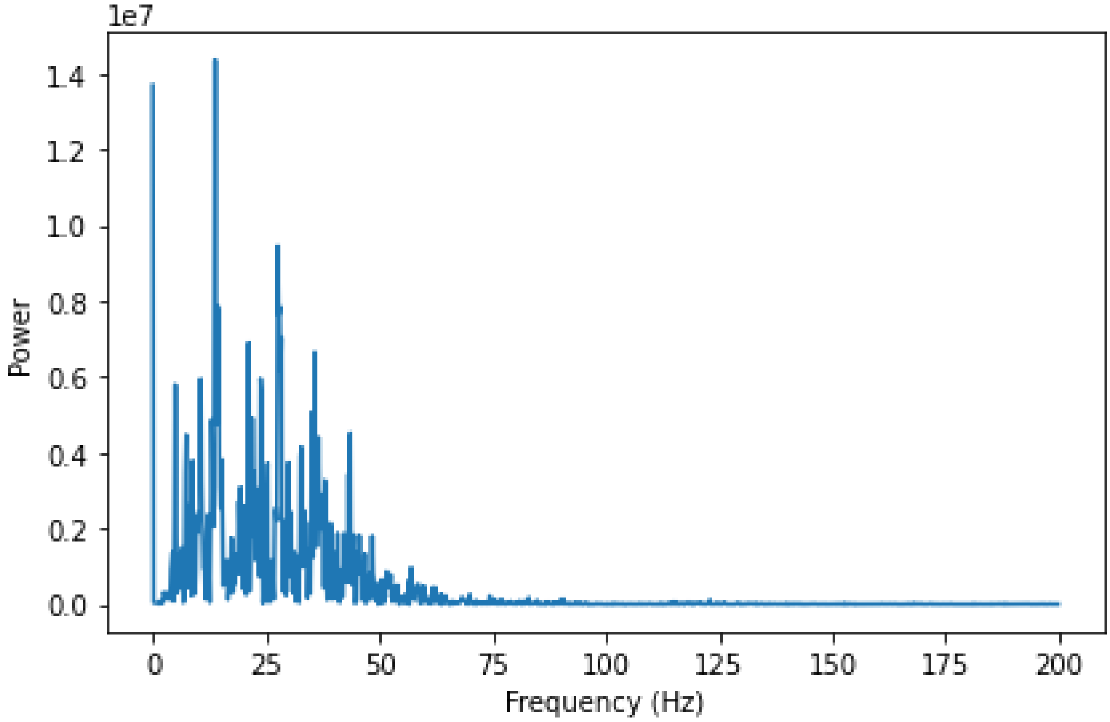

Рисунок 4.1. График сигнала

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

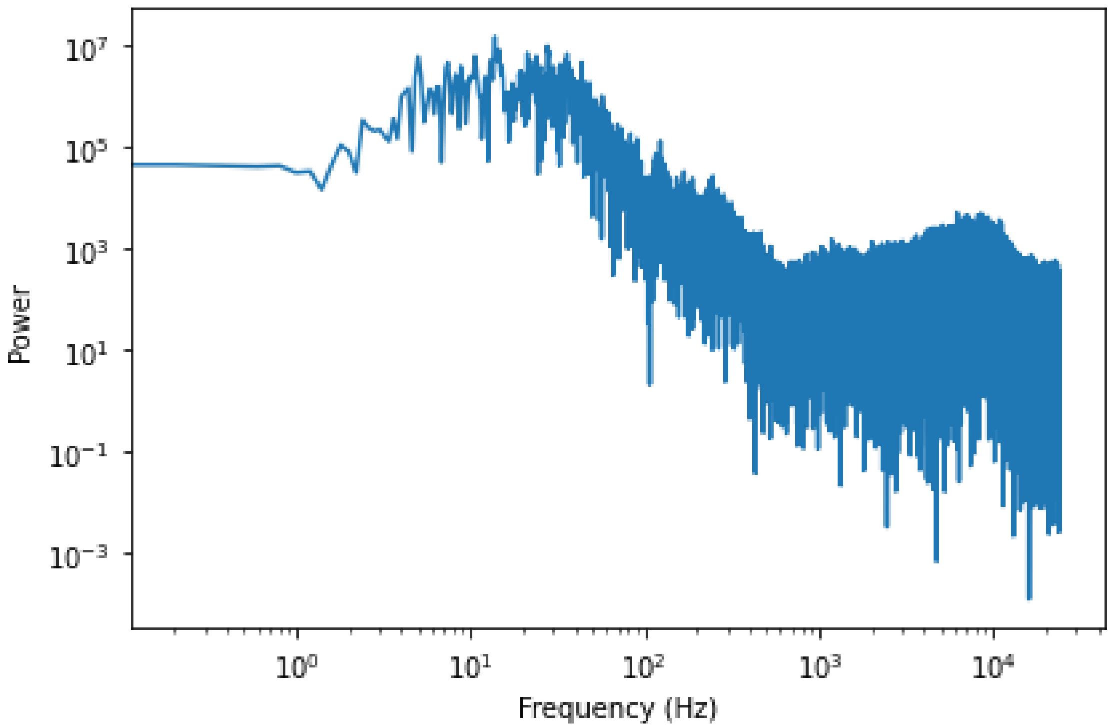

Рисунок 4.2. Спектр в логорифмическом масштабе

Похоже на Броуновский шум

```
spec = wave_rain.make_spectrum()
spec.plot_power(high = 200)
decorate(xlabel='Frequency (Hz)', ylabel='Power')
```

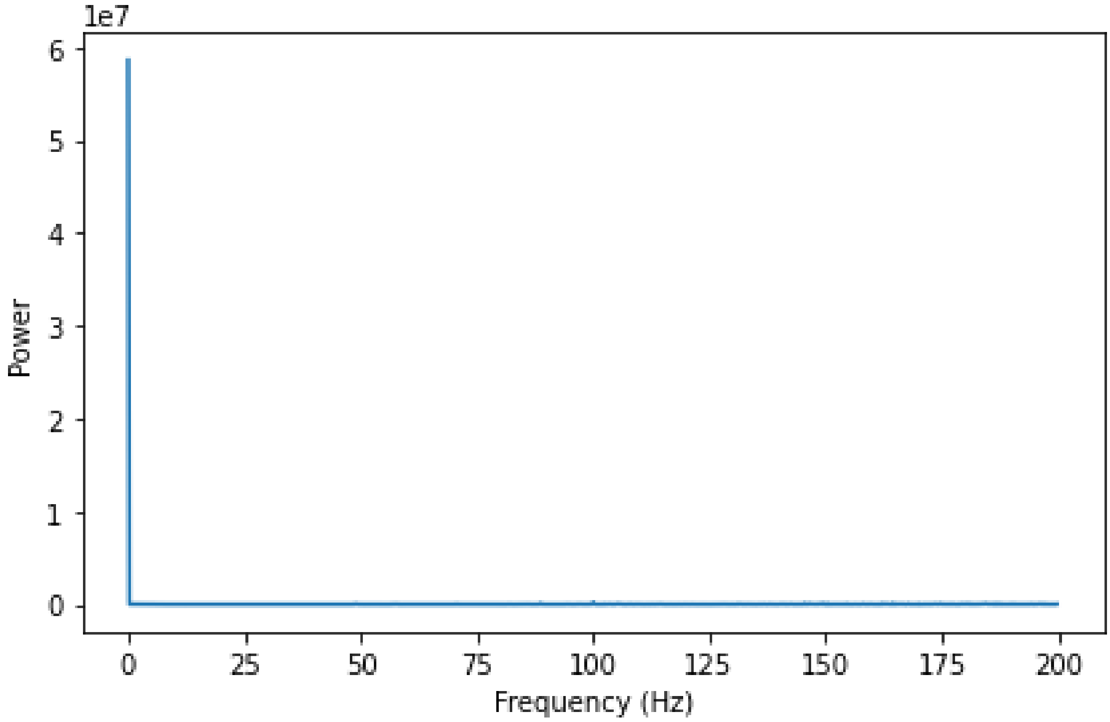

Рисунок 4.3. График сигнала

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

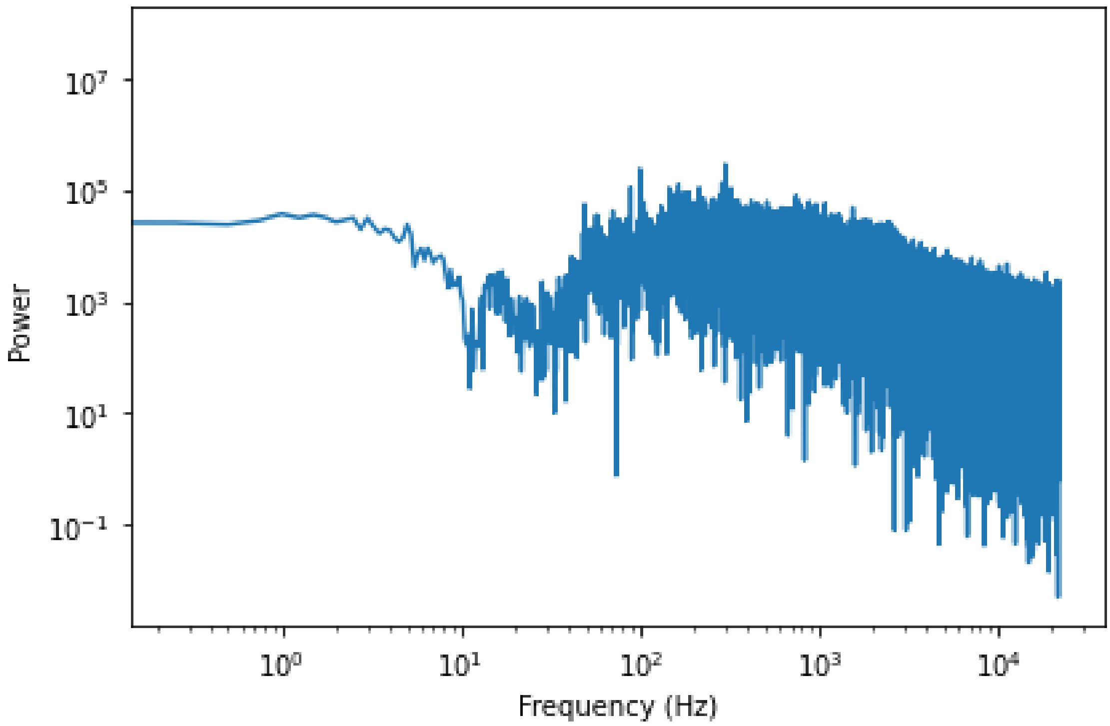

Рисунок 4.4. Спектр в логорифмическом масштабе

Шум дождя тоже

4.2. Упражнение 2

Необходимо создать функцию, которая реализует метод Бартлетта

```
def bar_method(wave, seg_length = 512, win_flag = True):
    spectogramm = wave.make_spectrogram(seg_length, win_flag)
    spec_vals = spectogramm.spec_map.values()

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

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

Протестируем

```
psd = bar_method(wave_rain)
psd.plot_power()
log_test_1 = dict(xscale='log', yscale='log')
decorate(xlabel='Frequency (Hz)', ylabel='Power', **log_test_1)
```

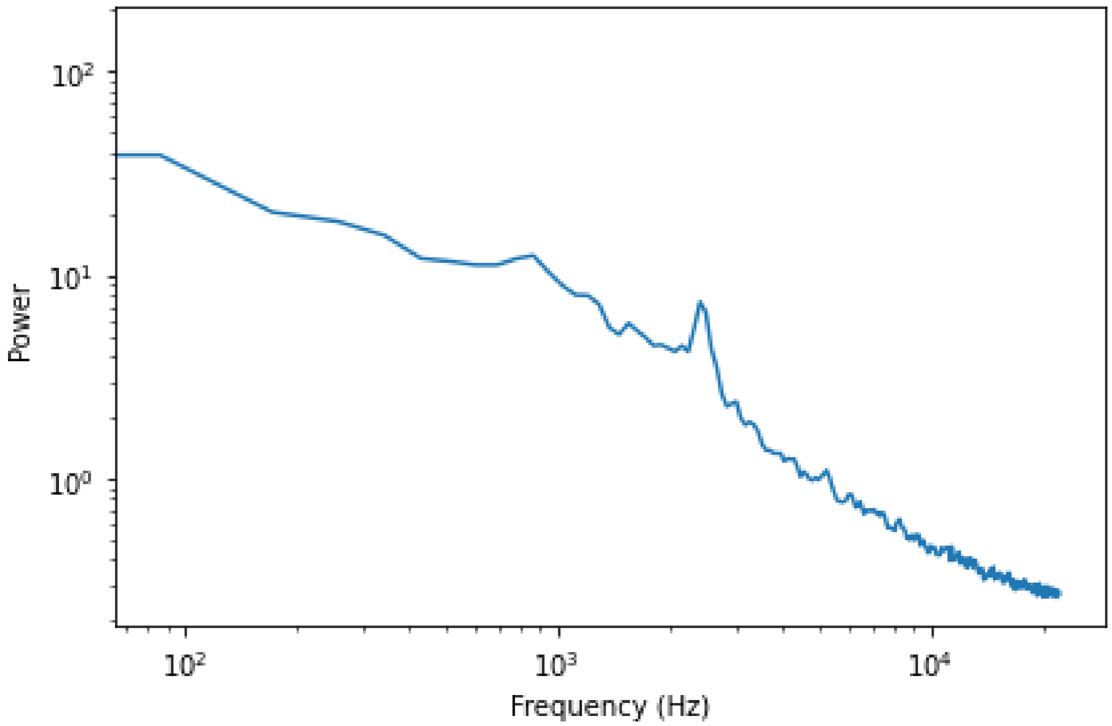

Рисунок 4.5. Результаты применения функции

```
psd = bar_method(wave_wind)
psd.plot_power()
log_test_2 = dict(xscale='log', yscale='log')
decorate(xlabel='Frequency (Hz)', ylabel='Power', **log_test_2)
```

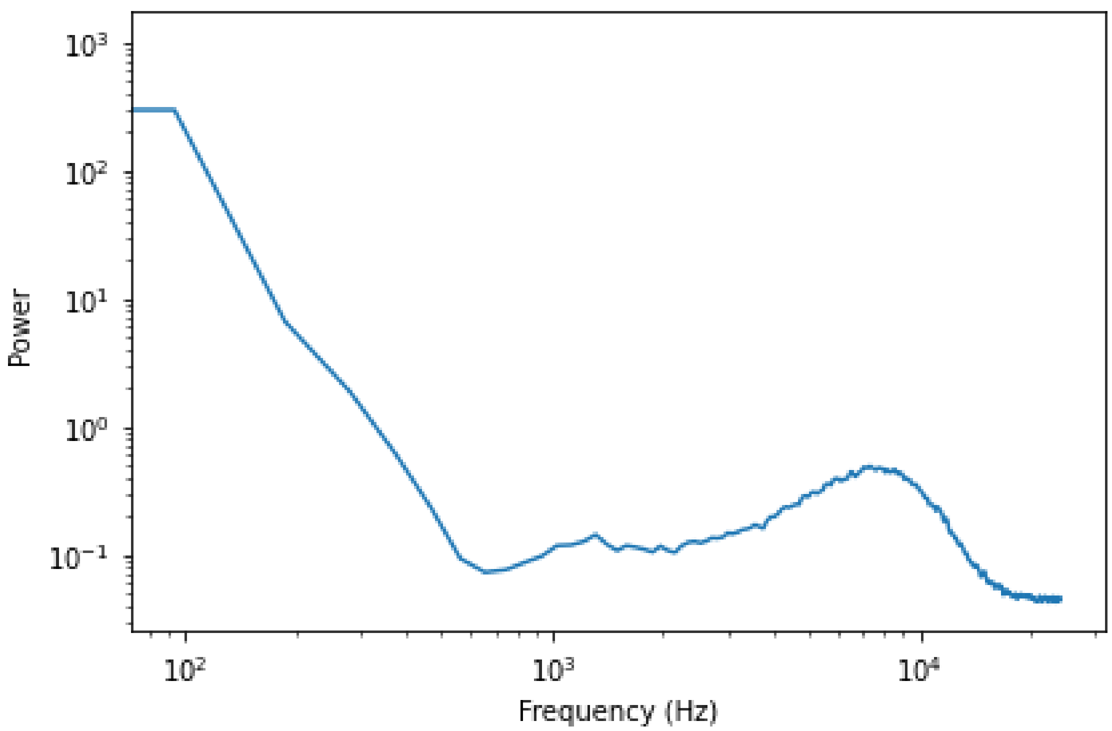

Рисунок 4.6. Результаты применения функции

4.3. Упражнение 3

Скачаем данные по цене биткоина и вычислим его спектр

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

import pandas as pd

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

	Currency	Date	Closing Price (USD)	24h Open (USD)	24h High (USD)	24h Low
0	BTC	2013-10-01	123.654990	124.304660	124.751660	122.[illegible]
1	BTC	2013-10-02	125.455000	123.654990	125.758500	123.[illegible]
2	BTC	2013-10-03	108.584830	125.455000	125.665660	83.[illegible]
3	BTC	2013-10-04	118.674660	108.584830	118.675000	107.[illegible]
4	BTC	2013-10-05	121.338660	118.674660	121.936330	118.[illegible]
...	...	...	...	...	...	
2354	BTC	2020-03-22	5884.340133	6187.042146	6431.873162	5802.[illegible]
2355	BTC	2020-03-23	6455.454688	5829.352511	6620.858253	5694.[illegible]
2356	BTC	2020-03-24	6784.318011	6455.450650	6863.602196	6406.[illegible]
2357	BTC	2020-03-25	6706.985089	6784.325204	6981.720386	6488.[illegible]
2358	BTC	2020-03-26	6721.495392	6697.948320	6796.053701	6537.[illegible]

2359 rows × 6 columns

Рисунок 4.7. Таблица значений

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

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

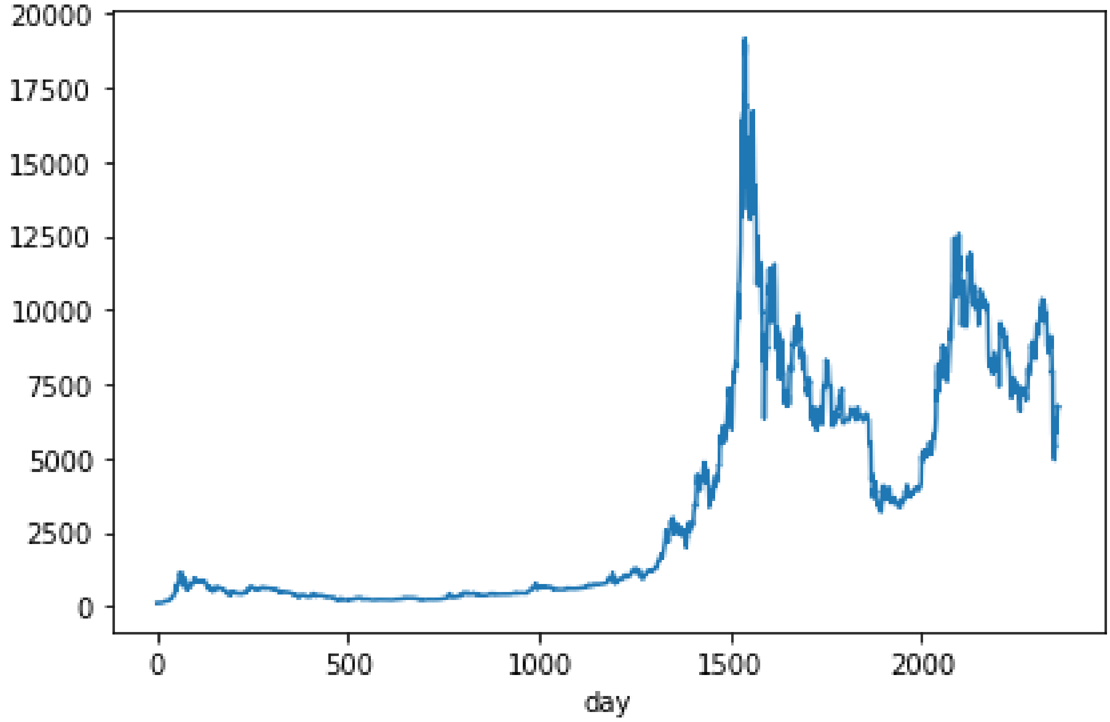

Рисунок 4.8. График цен BitCoin

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

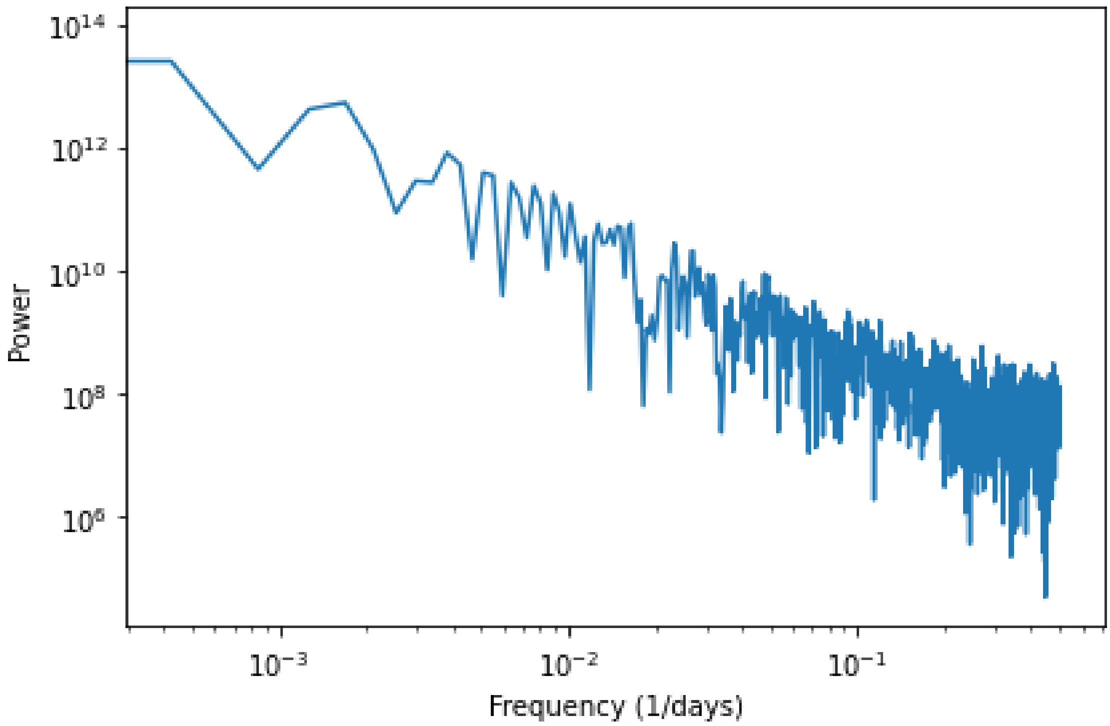

Рисунок 4.9. Спектрограмма цен BitCoin в логорифмическом формате

```
spec.estimate_slope()[0]

-1.7986681316517856
```

Похоже на розовый шум, так как наклон красного -2

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.

Напишем класс UncorrelatedPoissonNoise с наследованием от $_Noise$

```
from thinkdsp import Noise

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

Проверим работу класса на значениях малой и большой амплитуды

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

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

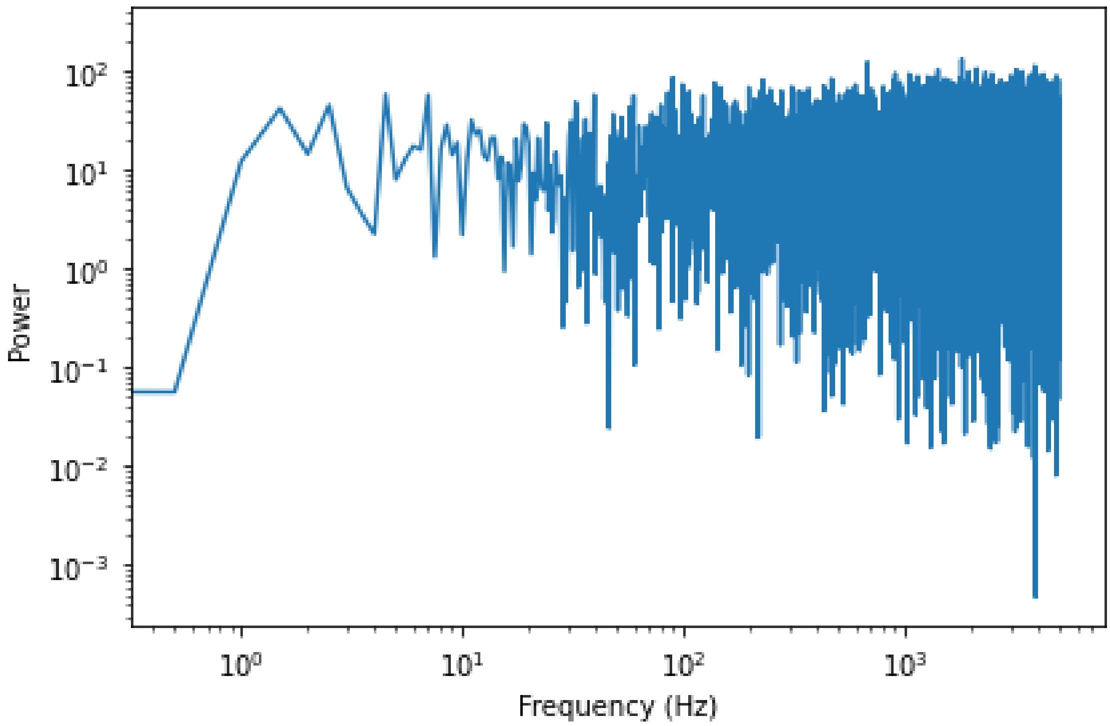

Рисунок 4.10. Получившийся спектр сигнала

Теперь создадим такой же сигнал, но с большей амплитудой

```
signal = UncorrelatedPoissonNoise(amp = 0.1)
wave = signal.make_wave(duration = 2, framerate = 10000)
wave.make_audio()

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

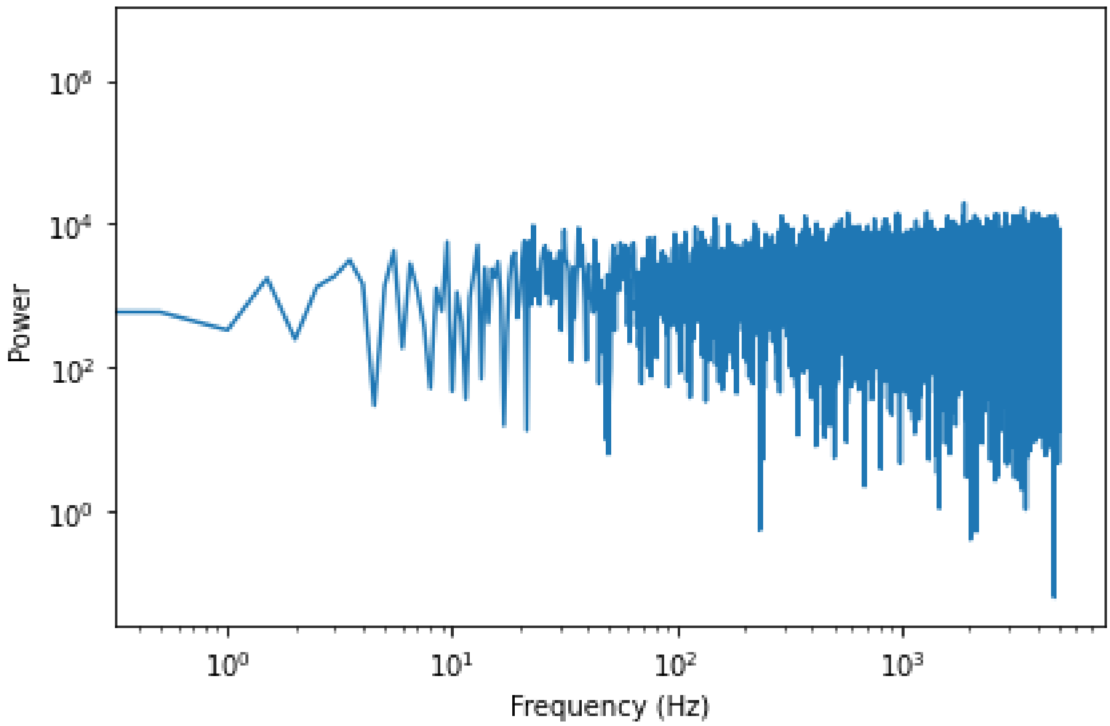

Рисунок 4.11. Получившийся спектр сигнала

При малой амплитуде слышим треск как счетчик Гейгера, при большой, он становится 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 Восса-Маккартни.
Исследуйте этот метод, реализуйте его, вычислите спектр и подтвердите, что он имеет желаемое отношение между мощностью и частотой.

Создадим функцию voss.

```
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
```

Протестируем

```
ys = voss(10000)
ys

array([8.05587101, 6.94436007, 6.57274844, ..., 7.95820399, 8.83511144,
       8.77457504])

wave = thinkdsp.Wave(ys)
wave.unbias()
wave.normalize()
wave.plot()
```

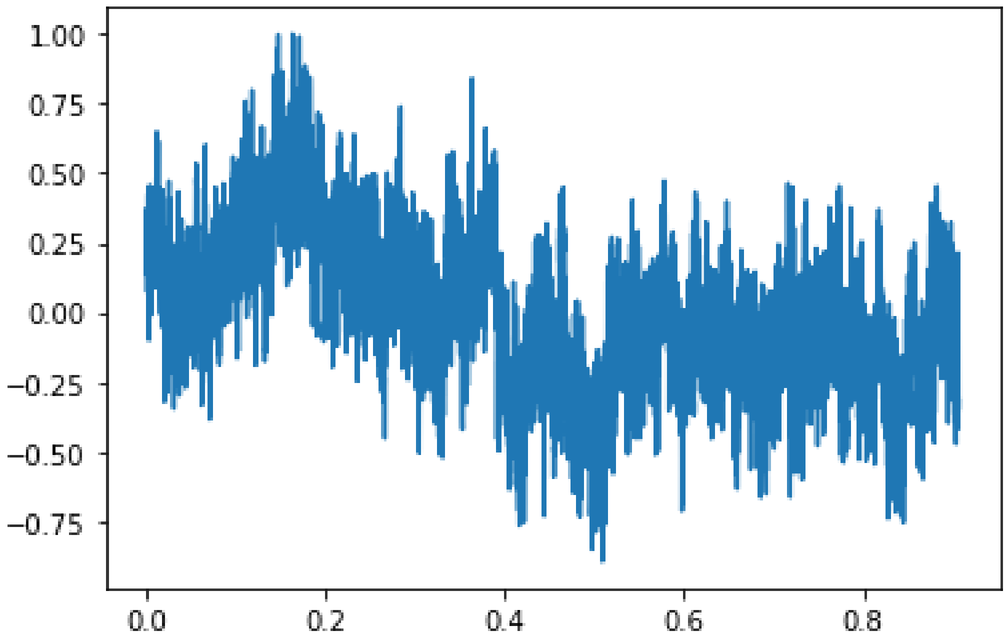

Рисунок 4.12. Сгенерированный сигнал

Получаем какой-то случайный шум

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

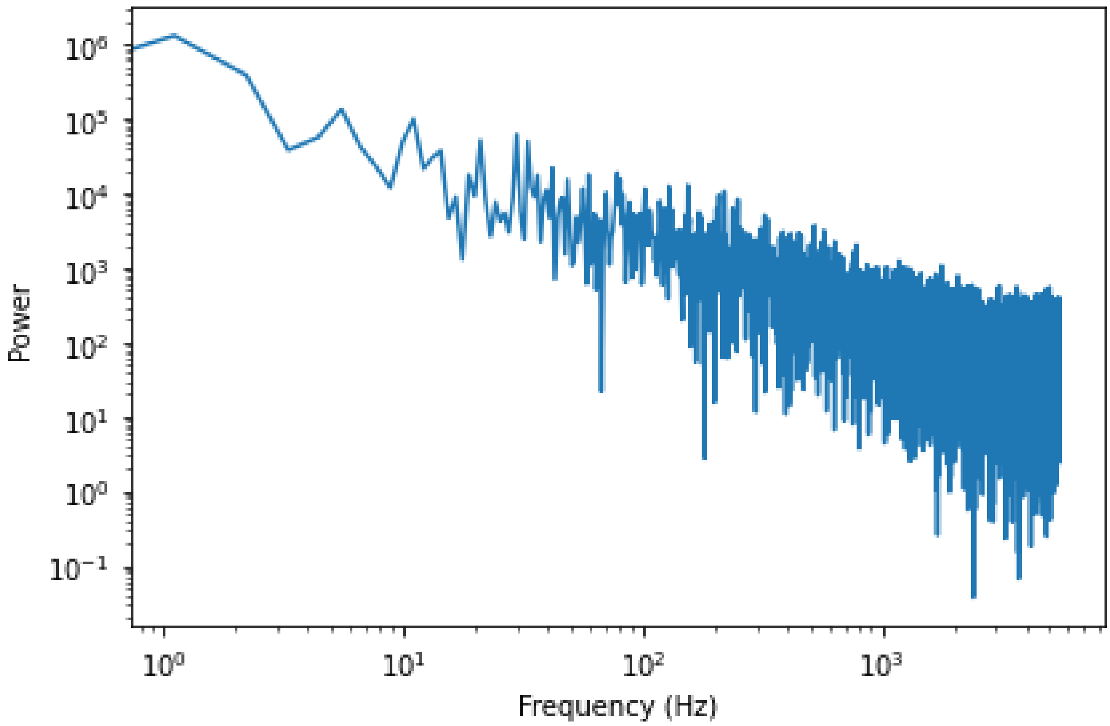

Рисунок 4.13. Спектр полученного шума

```
spec.estimate_slope().slope

-1.0346367430959535
```

Расчетный наклон близок к -1. Проверим на методе Бартлетта

```
spec_2 = bar_method(wave, seg_length=8000, win_flag=False)
spec_2.hs[0] = 0
spec_2.plot_power()
decorate(xlabel='Frequency (Hz)', ylabel='Power', **loglog)
```

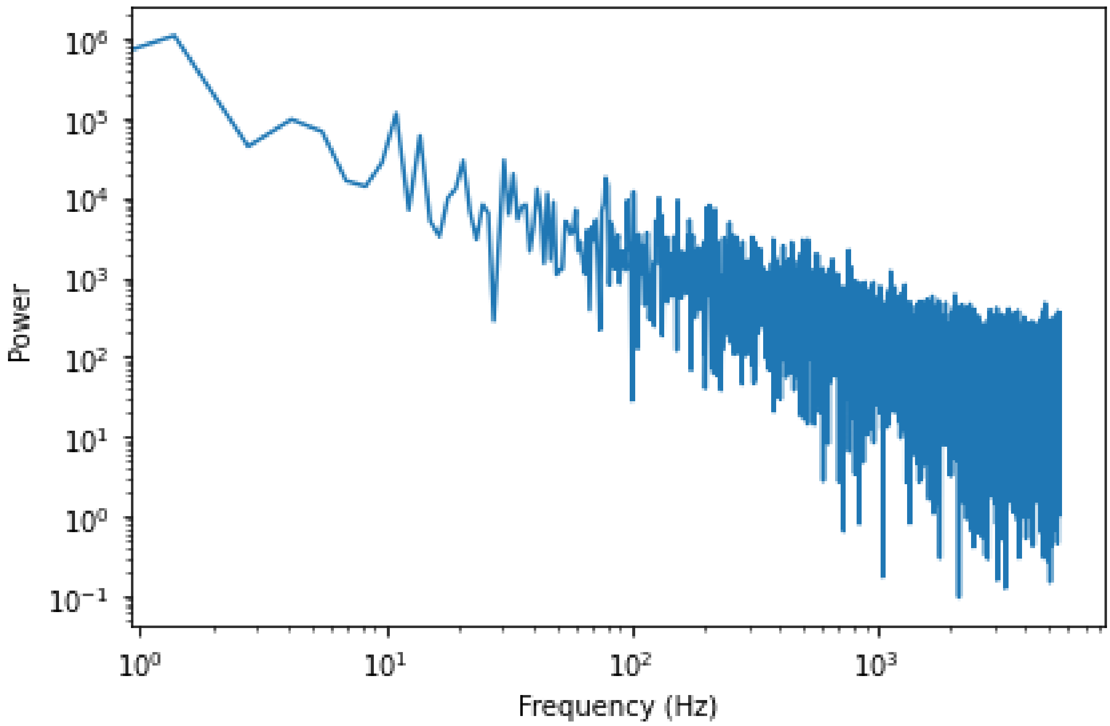

Рисунок 4.14. Спектр средней мощности шума

```
spec_2.estimate_slope().slope

-1.0262915371386319
```

Получаем значение чуть ближе к -1

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

Оценка вокального чирпа для нескольких времён начала сегмента. Возьмем пение птицы

```python
if not os.path.exists('456440__inspectorj__bird-whistling-robin-single-13.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/456440__inspectorj__bird-
        whistling-robin-single-13.wav

wave = read_wave('456440__inspectorj__bird-whistling-robin-single-13.wav')

wave.normalize()
wave.make_audio()

duration = 0.01
segment1 = wave.segment(start=0.24, duration=duration)
segment1.plot()
segment2 = wave.segment(start=0.26, duration=duration)
segment2.plot()
segment3 = wave.segment(start=0.28, duration=duration)
segment3.plot()
```

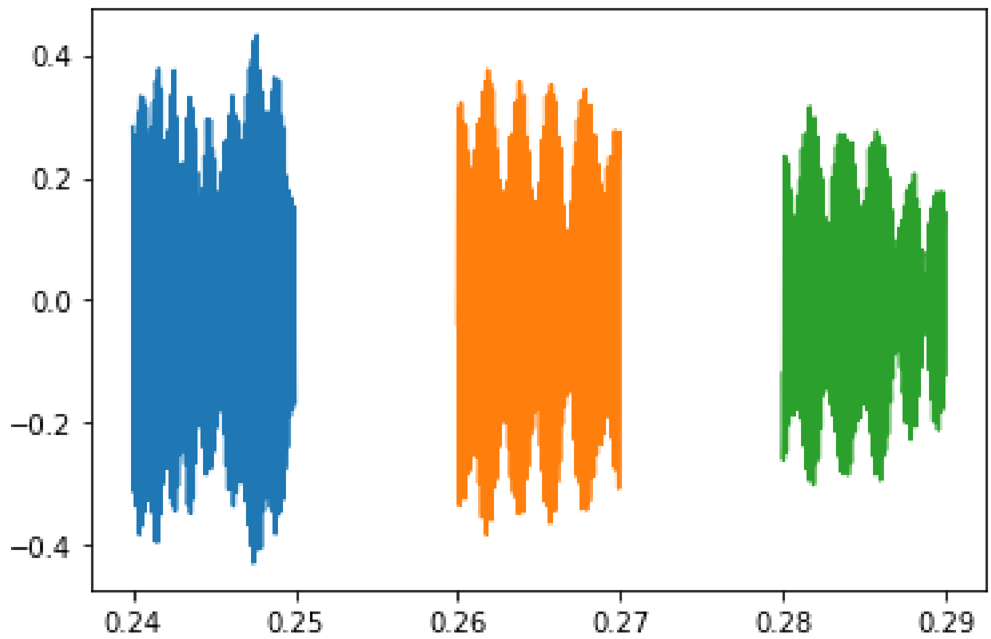

Рисунок 5.1. График выбранных сегментов

Используем автокорреляцию для поиска высоты тона

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

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

```

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

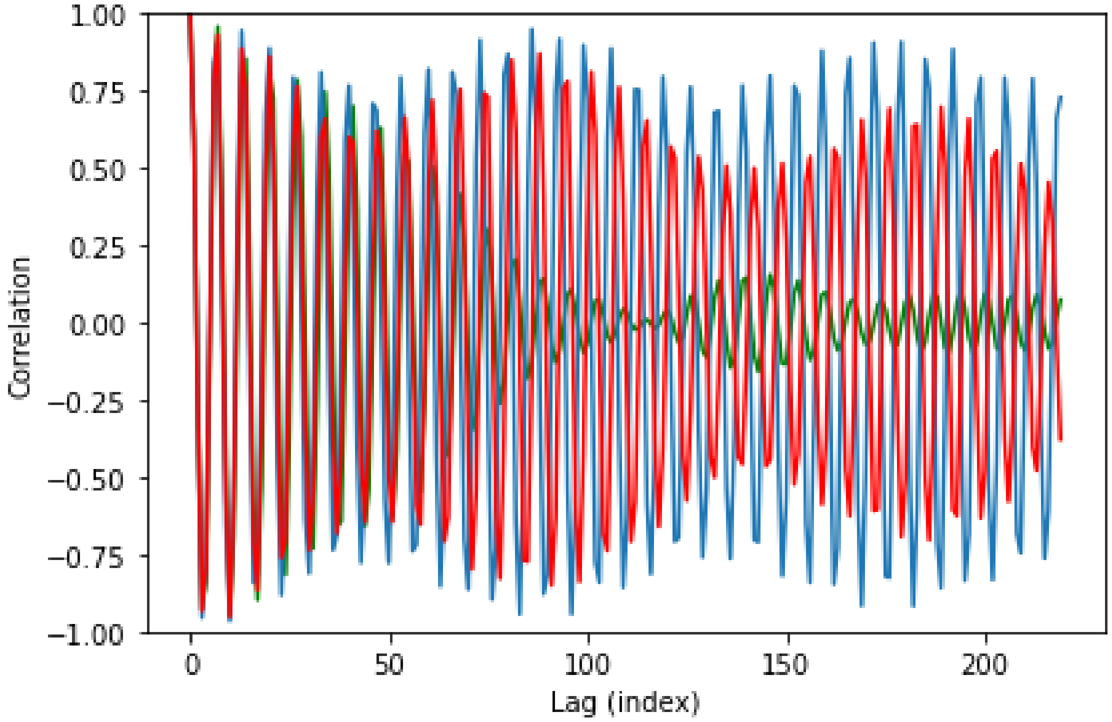

Рисунок 5.2. Автокорреляция сигналов

Вычислим значения Lags, периоды и Fmax

```
low, high = 50, 200
lag1 = np.array(corrs1[low:high]).argmax() + low

low, high = 50, 200
lag2 = np.array(corrs2[low:high]).argmax() + low

low, high = 50, 200
lag3 = np.array(corrs3[low:high]).argmax() + low

lag1, lag2, lag3

(54, 86, 88)

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

(0.0012244897959183673, 0.0019501133786848073, 0.00199546485260771)

frequency1 = 1 / period1
frequency2 = 1 / period2
frequency3 = 1 / period3
frequency1, frequency2, frequency3

(816.6666666666667, 512.7906976744185, 501.1363636363636)
```

5.2. Упражнение 2

Пример кода в chap05.ipynb показывает, как использовать автокорреляцию для оценки основной частоты периодического сигнала. Инкапсулируйте этот код в функцию estimate$_{d}undamental$, .

Инкапсулируем код для оценки основной частоты периодического сигнала из chap05.ipynb в функцию estimate$_{d}undamental$

```
wave.make_audio()
```

Построим спектограмму

```
wave.make_spectrogram(2048).plot(high = 4200)
```

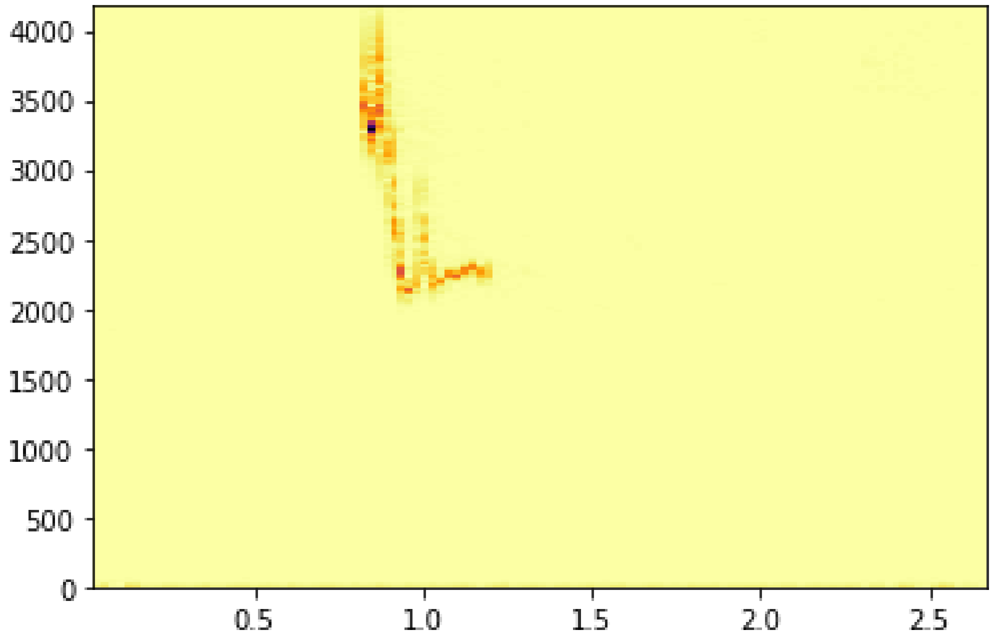

Рисунок 5.3. Спектрограмма записи

Используем функцию estimate$_{f}undamental$5.70200

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

duration = 0.01
segment = wave.segment(start=0.1, duration=duration)
freq = estimate_fundamental(segment)
freq

383.4782608695652
```

В цикле отследим пик по всему звуку. ts - это середина каждого сегмента

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

ts = []
```

```
freqs = []

for start in starts:
    ts.append(start + step/2)
    segment = wave.segment(start=start, duration=duration)
    freq = estimate_fundamental(segment)
    freqs.append(freq)
```

Синяя линия на графике наложена на спектограмму и показывает отслеживание высоты тона

```
wave.make_spectrogram(2048).plot(high = 900)
plt.plot(ts, freqs, color='blue')
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

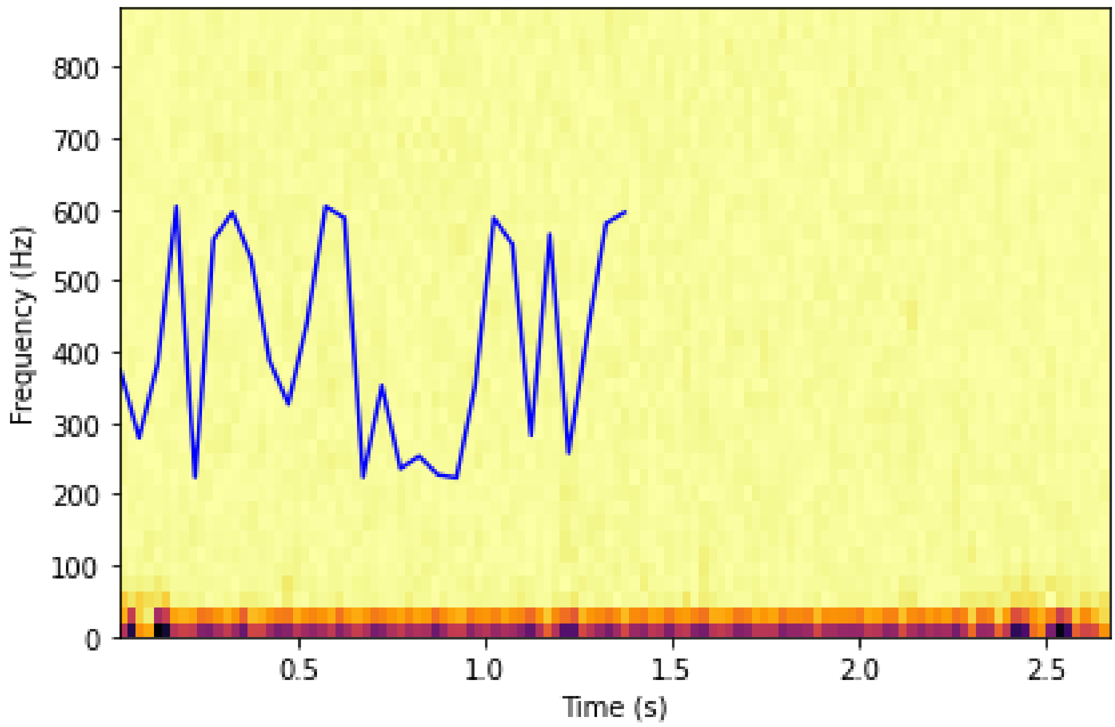

Рисунок 5.4. Результат

5.3. Упражнение 3

Возьмем данные о BitCoin из прошлой лабораторной работы и вычислим для этих данных автокорреляцию цен

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

import pandas as pd

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

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

Рисунок 5.5. Таблица цены на BitCoin

Вычислим автокорреляцию:

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

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

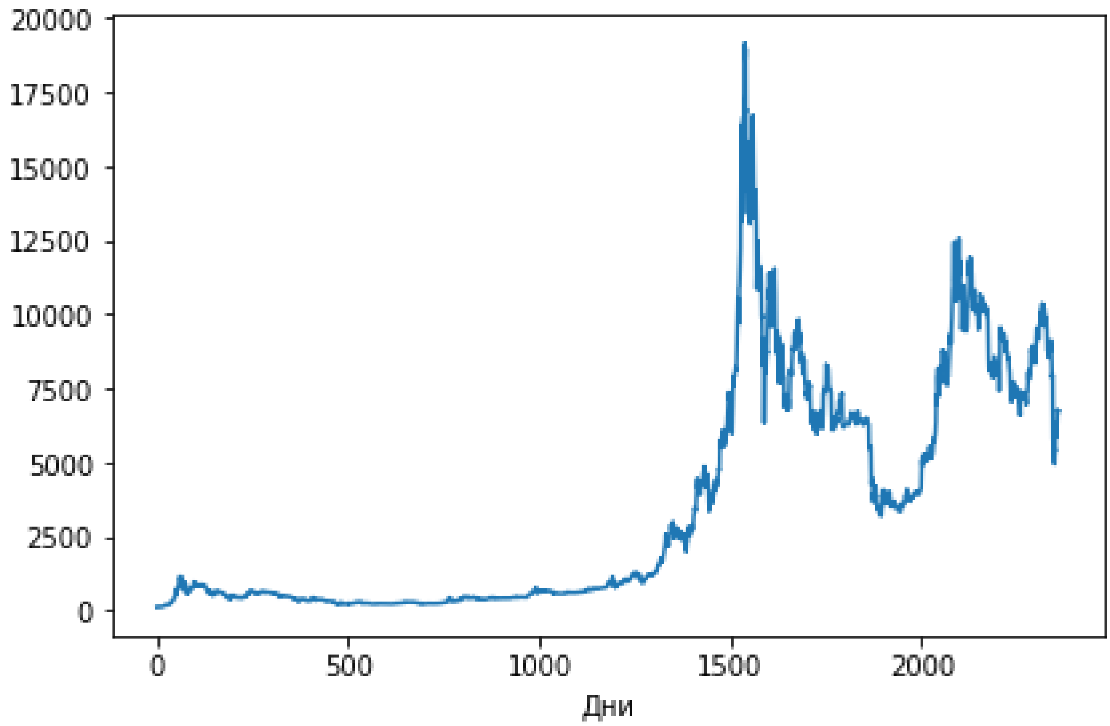

Рисунок 5.6. График цен на BitCoin

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

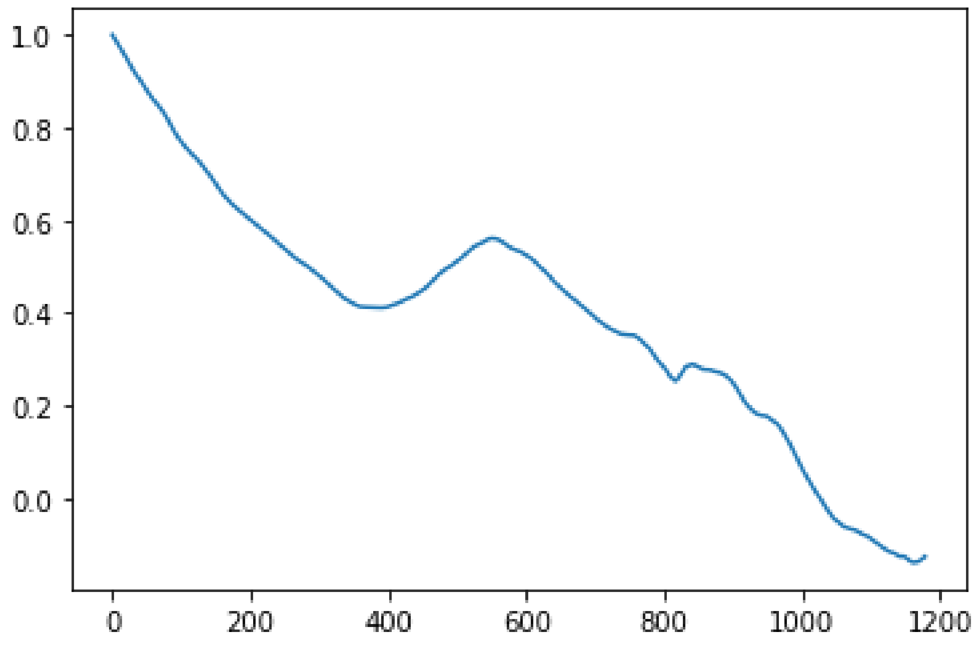

Рисунок 5.7. Автокорреляция функции цены на BitCoin

Исходя из графика видно, что он постепенно снижается и похож на розовый шум. Также присутствует умеренная корреляция на 550 дне и 820. Теперь вычислим корреляцию на основе функции np.correlate, она не смещает и нормализует волну.

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

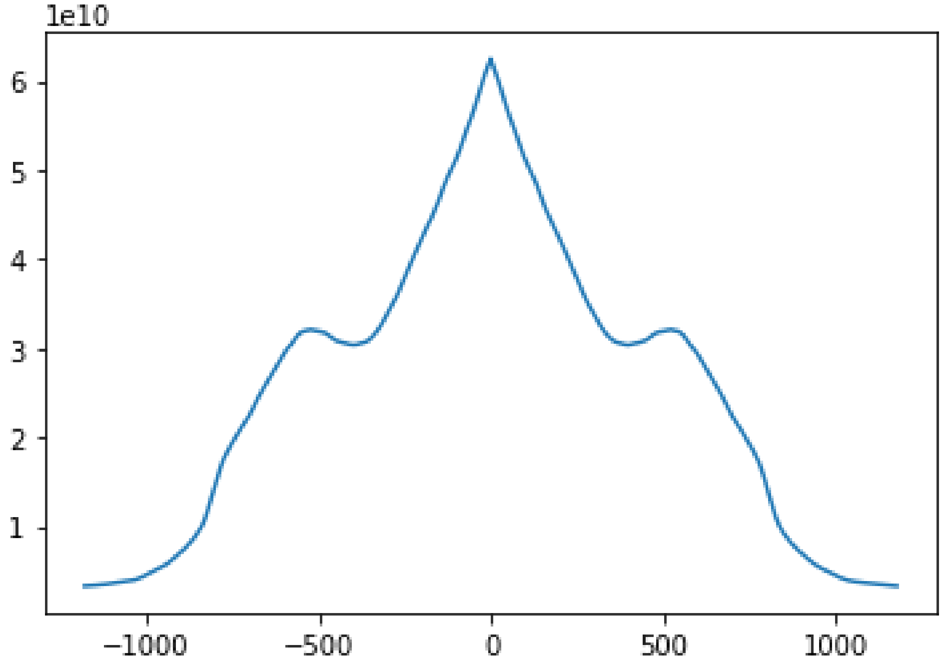

Рисунок 5.8. Автокорреляция функции цены на BitCoin при помощи np.correlate

Вторая часть результатов (правая) соответствует положительными интервалам lags

```
N = len(corrs2)
half = corrs[N//4:]
plt.plot(half)
```

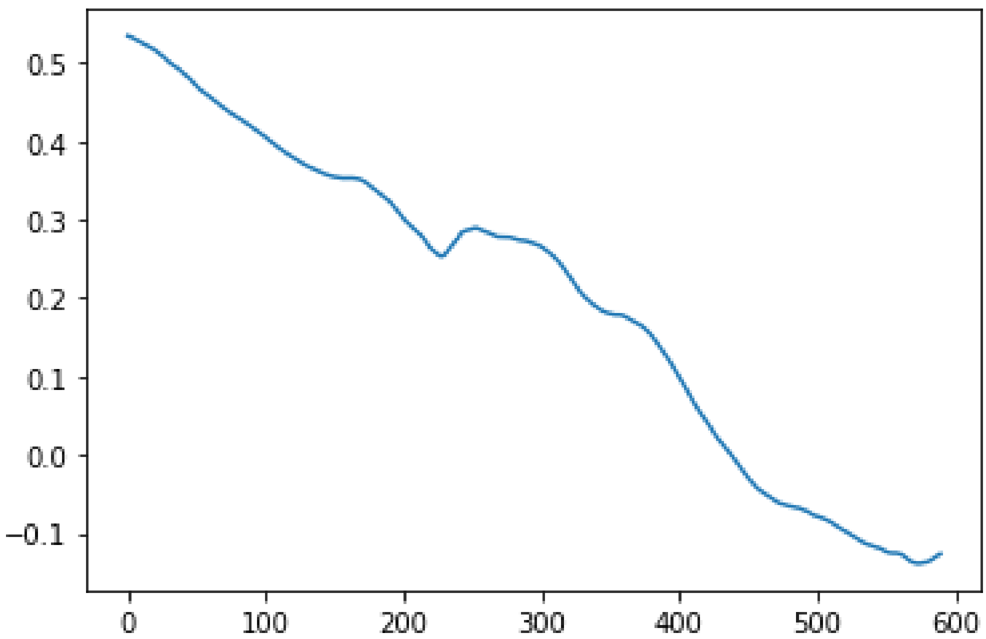

Рисунок 5.9. Правая часть результатов

5.4. Упражнение 4

В репозитории Jupyter есть блокнот saxophone.inypb, в котором исследуется автокорреляция, восприятие высоты тона и явление, называемое подавленная основная. Прочтите этот блокнто 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
        __iluppai__saxophone-weep.wav

wave = read_wave('100475__iluppai__saxophone-weep.wav')
wave.normalize()
wave.make_audio()
```

Построим спектограмму

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

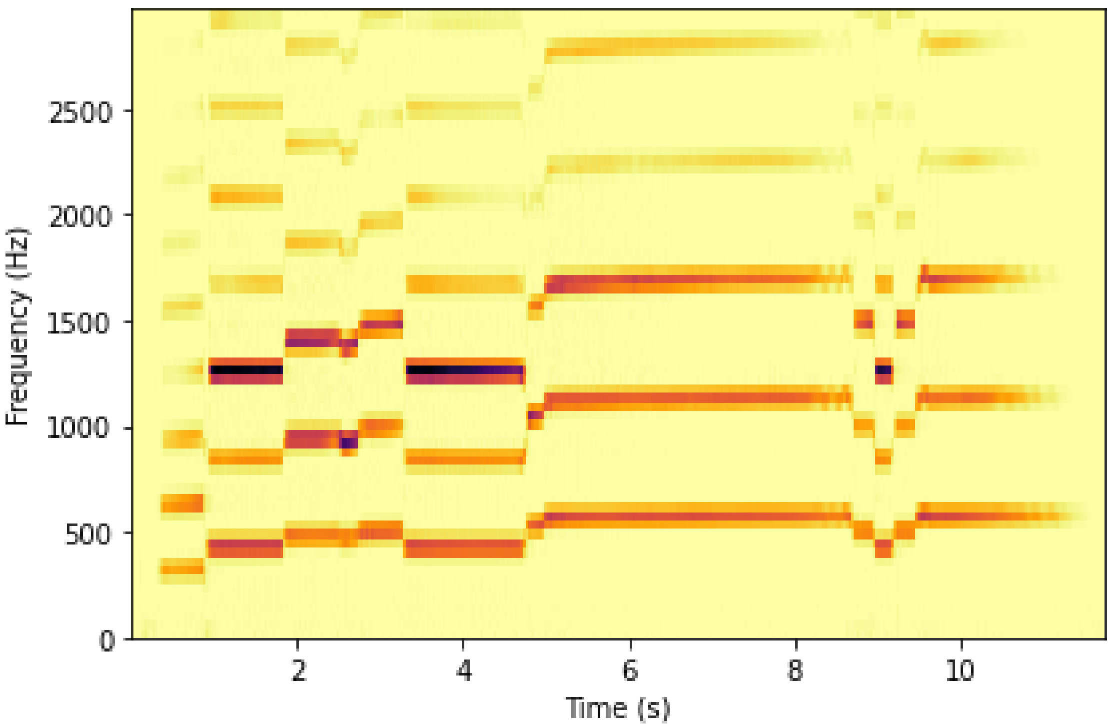

Рисунок 5.10. Спектрограмма звука

На графике видна гармоническая структура во времени

Теперь возьмем некоторый сегмент и "прогоним" его через функции из блокнота

```
segment = wave.segment(start=1, duration=0.2)
segment.make_audio()

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

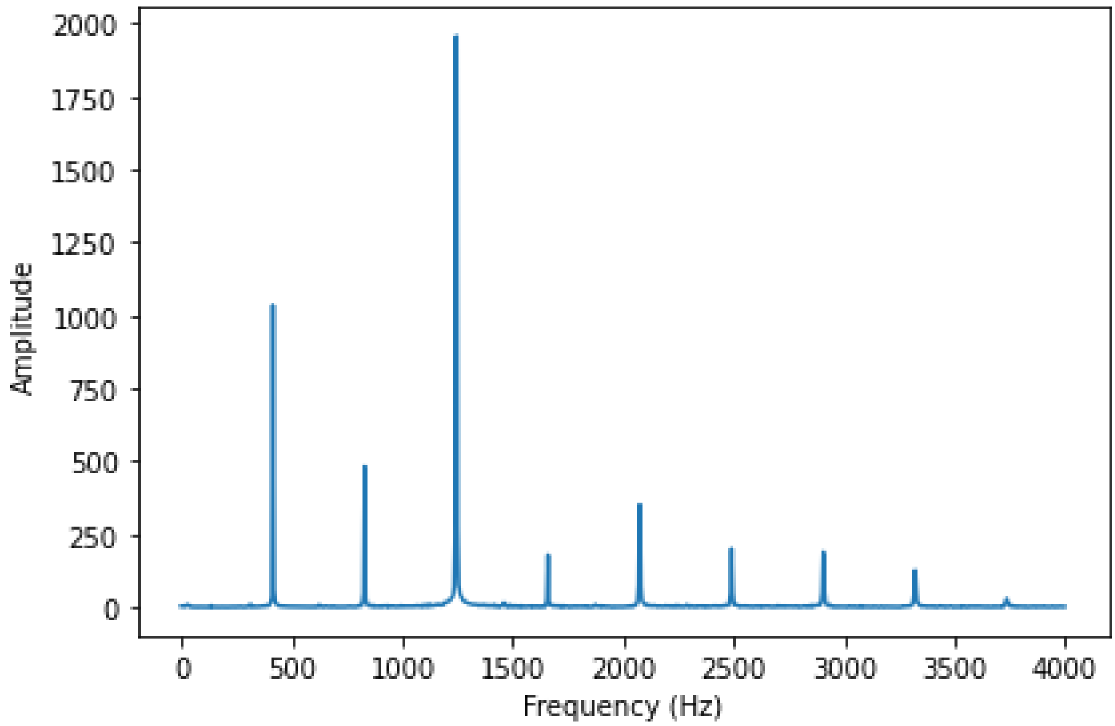

Рисунок 5.11. Спектр звука

Данный спектр похож на спектр квадратного сигнала. Пики находятся на 1245, 415 и 830 Гц

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

[(1956.703292712428, 1245.0),
 (1034.570299739157, 415.0),
 (481.52480153206335, 830.0),
 (351.0776583833776, 2075.0),
 (275.4196733573752, 1250.0),
 (264.72288947314, 1240.0),
 (200.51232139688983, 2490.0),
 (188.9737445345148, 2905.0),
 (179.39558697128783, 1660.0),
 (126.34825373317715, 3320.0)]
```

Теперь сравним наш сигнал с треугольным, у которого такая же низкая частота пика

```
from thinkdsp import TriangleSignal

TriangleSignal(freq=415).make_wave(duration=0.2).make_audio()
```

Воспользуемся автокорреляцией для понимания основной частоты

```
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])
```

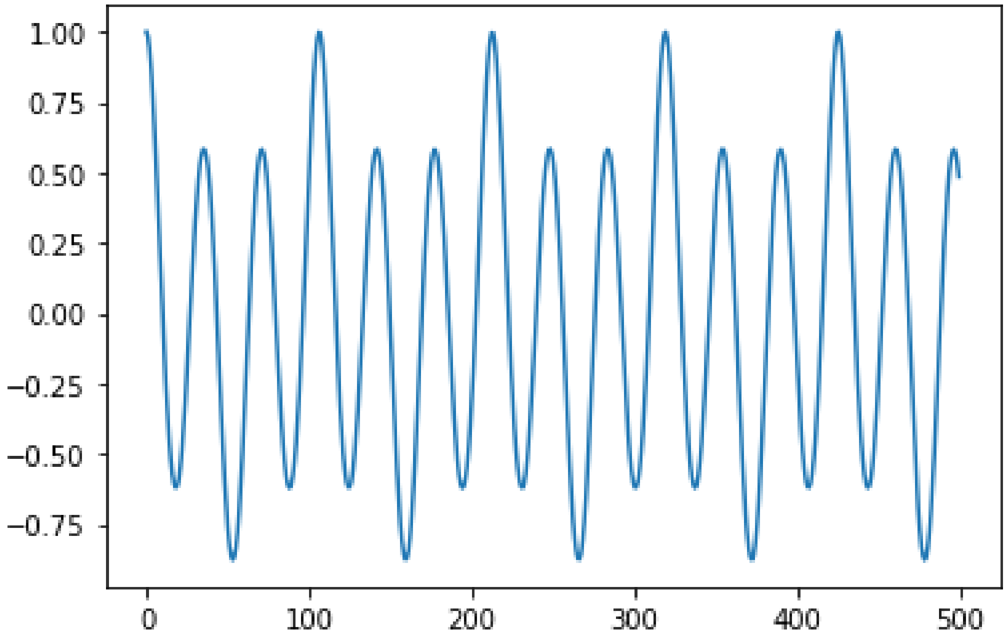

Рисунок 5.12. Автокорреляция

Исходя из графика видны пики рядом с lag 100
Теперь найдем основную частоту

```
estimate_fundamental(segment)

416.0377358490566
```

Попробуем убрать основной тон, что лучше воспринимать звук

```
spec2 = segment.make_spectrum()
spec2.high_pass(600)
spec2.plot(high=5000)
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

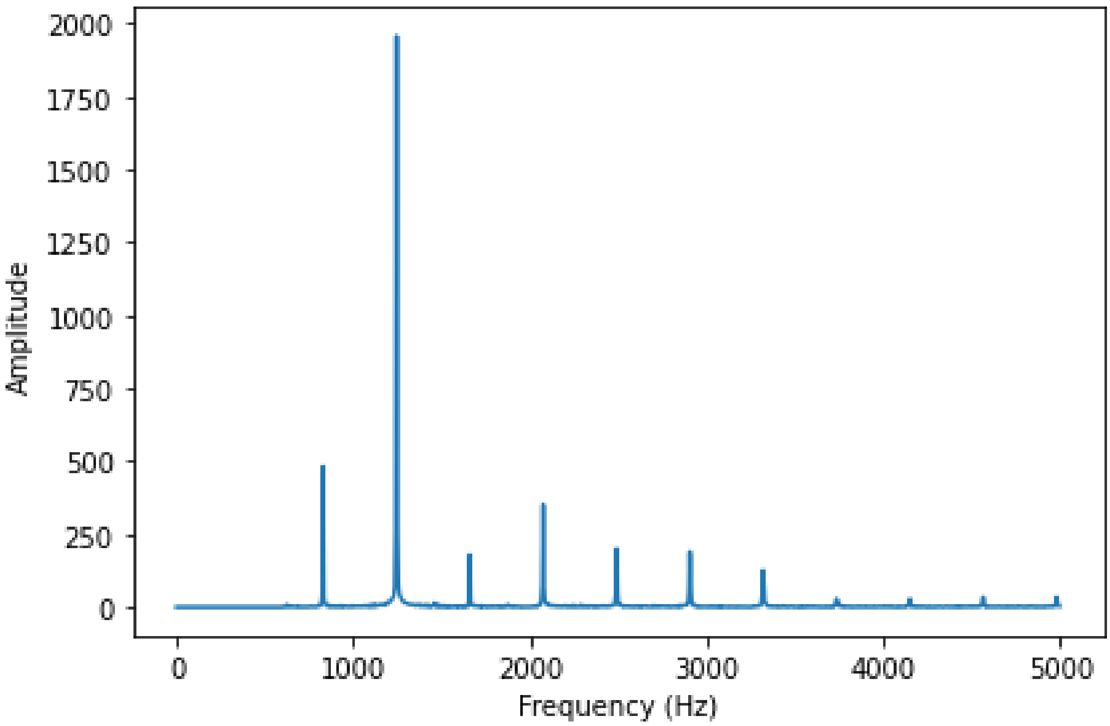

Рисунок 5.13. Спектр сигнала

```
segment2 = spec2.make_wave()
segment2.make_audio()
```

Звук воспринимается также

Это явление называется missing fundamental. Чтобы понять то, что мы слышим частоту, которой нет, можно снова использовать автокорреляцию.

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

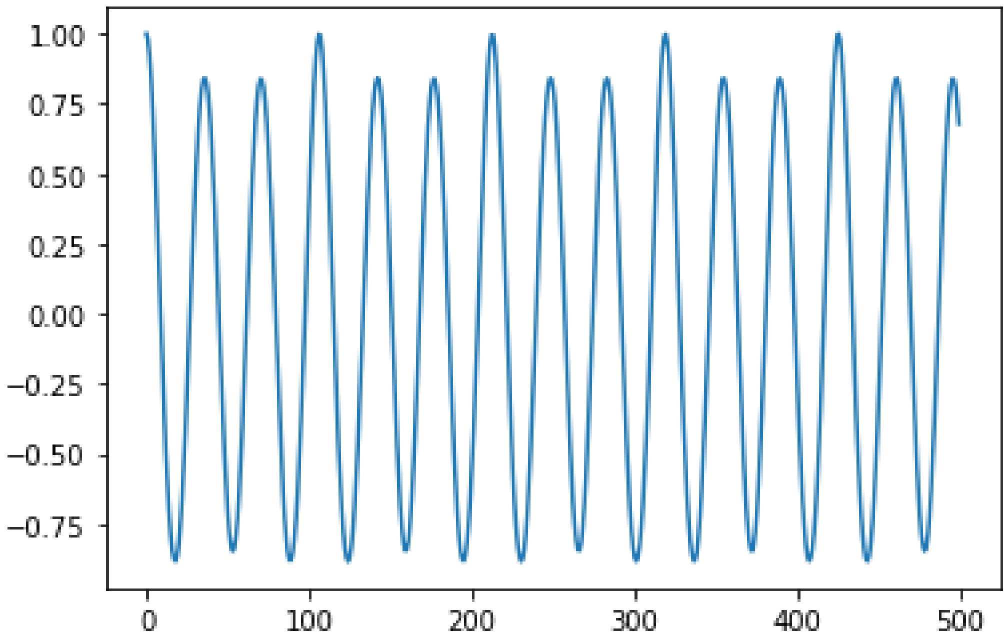

Рисунок 5.14. Автокорреляция

```
estimate_fundamental(segment2)

416.0377358490566
```

Получились одинаковые значения, так как более высокие компоненты сигнала являются гармониками 416Гц

Исходя из проведенных опытов можно сделать вывод, что восприятие выосты тона основано не только на спектральном анализе, но и на вычислении АКФ.

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

Убедимся в том, что analyze1 требует времени пропорционально n^3, $analyze2n^2$.

```
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
```

Возьмем сигнал шума и массив, состоящий из степеней двойки

```
from thinkdsp import UncorrelatedGaussianNoise

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

(8192,)
```

```
ns = 2 ** np.arange(6, 14)
ns

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

Напишем функцию plot$_b ests$,

```
from scipy.stats import linregress

def plot_bests(bests):
    plt.plot(ns, bests)
    loglog = dict(xscale='log', yscale='log')
    decorate(xlabel='Wave length (N)', ylabel='Time (s)', **loglog)
    x = np.log(ns)
    y = np.log(bests)
    t = linregress(x, y)
    slope = t[0]

    return slope
```

Вычисдим результат для analyze1

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

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

```
plot_bests(bests)

64
The slowest run took 4.21 times longer than the fastest. This could mean that an
    intermediate result is being cached.
1000 loops, best of 1: 243 µs per loop
128
1000 loops, best of 1: 818 µs per loop
256
100 loops, best of 1: 3.57 ms per loop
512
100 loops, best of 1: 18.1 ms per loop
1024
10 loops, best of 1: 73.7 ms per loop
2048
1 loop, best of 1: 549 ms per loop
4096
1 loop, best of 1: 3.68 s per loop
8192
1 loop, best of 1: 24.4 s per loop
2.3906127505491934
```

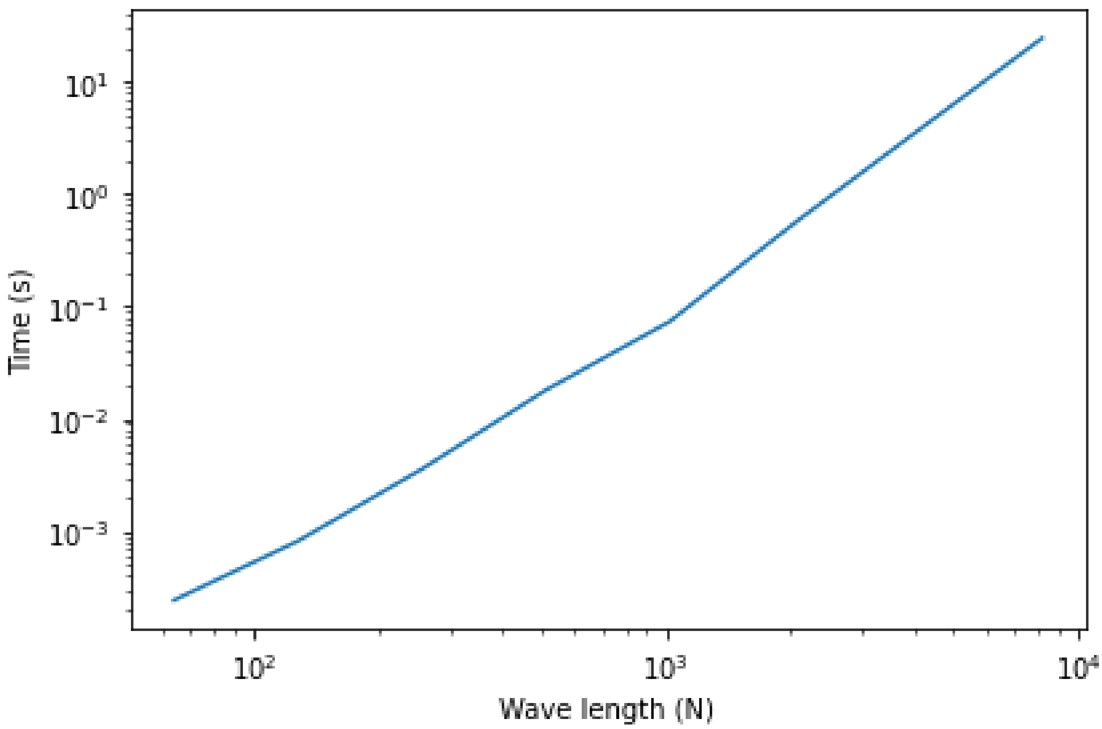

Рисунок 6.1. Время работы метода ДКП analyze1

Исходя из графика видно, что расчетный наклон близок к 2, а не 3, как ожидалось. Также на графике видно, что в конце линия немного изогнута, что говорит о том, что размер массива не достигнут, где analyze1 будет пропорционально n^3.,, $analyze1n^3, n^2$

Теперь протестируем analyze2

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

(8192,)
```

```

ns = 2 ** np.arange(6, 14)
ns

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

bests2 = [result.best for result in results]
plot_bests(bests2)
array([  64,  128,  256,  512, 1024, 2048, 4096, 8192])

64
10000 loops, best of 1: 99.1 µs per loop
128
1000 loops, best of 1: 558 µs per loop
256
100 loops, best of 1: 8.5 ms per loop
512
100 loops, best of 1: 14.8 ms per loop
1024
10 loops, best of 1: 41.4 ms per loop
2048
10 loops, best of 1: 134 ms per loop
4096
1 loop, best of 1: 401 ms per loop
8192
1 loop, best of 1: 1.49 s per loop
1.8809074973380902
```

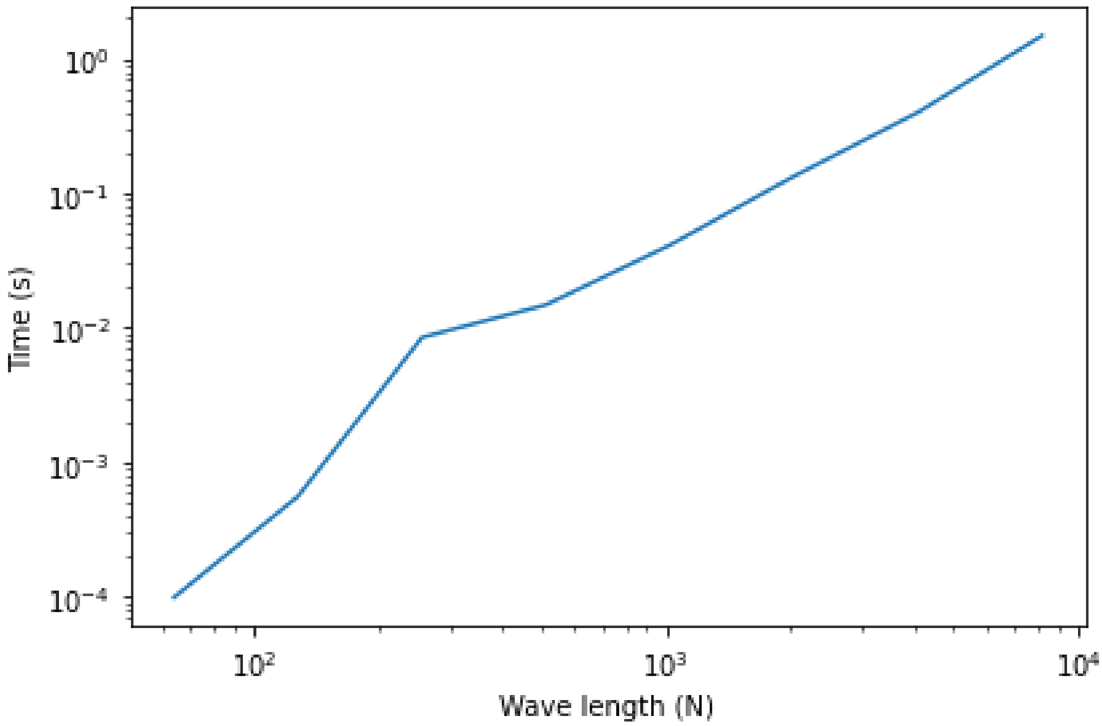

Рисунок 6.2. Время работы метода ДКП analyze2

Исходя из графика видно, что analyze2 растет пропорционально n^2,

Теперь проведем такой же эксперимент с использованием scipy.fftpack.dct

```
from scipy.fftpack import dct

results = []
for N in ns:
    print(N)
    ts = (0.5 + np.arange(N)) / N
    freqs = (0.5 + np.arange(N)) / 2
    ys = noise.ys[:N]
    result = %timeit -r1 -o dct(ys, type = 3)
    results.append(result)

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

64
The slowest run took 503.63 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 1: 5.73 µs per loop
128
The slowest run took 22.67 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 1: 6.07 µs per loop
256
The slowest run took 8.18 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 1: 6.78 µs per loop
512
The slowest run took 8.95 times longer than the fastest. This could mean that an intermediate result is being cached.
100000 loops, best of 1: 8.55 µs per loop
```

```
1024
The slowest run took 22.77 times longer than the fastest. This could mean that
    an intermediate result is being cached.
100000 loops, best of 1: 11.3 µs per loop
2048
The slowest run took 33.98 times longer than the fastest. This could mean that
    an intermediate result is being cached.
100000 loops, best of 1: 18.2 µs per loop
4096
The slowest run took 4.52 times longer than the fastest. This could mean that an
     intermediate result is being cached.
10000 loops, best of 1: 34.3 µs per loop
8192
The slowest run took 4.07 times longer than the fastest. This could mean that an
     intermediate result is being cached.
10000 loops, best of 1: 69.9 µs per loop
0.5049032811234534
```

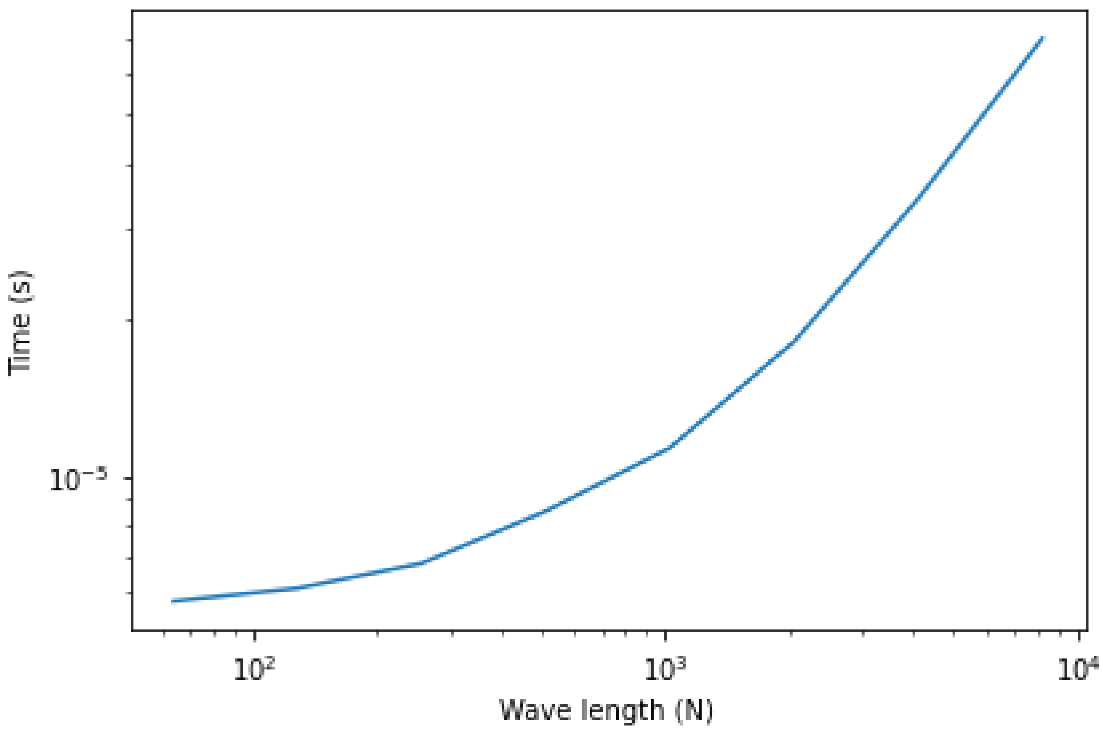

Рисунок 6.3. Время работы метода ДКП scipy.fftpack.dct

Данная реализация довольно быстрая, также она пропорциональна n*log(n)
Построим для наглядности все 3 графика на одном

```
plt.plot(ns, bests, label='analyze1')
plt.plot(ns, bests2, label='analyze2')
plt.plot(ns, bests3, label='fftpack.dct')
loglog = dict(xscale='log', yscale='log')
decorate(xlabel='Wave length (N)', ylabel='Time (s)', **loglog)
```

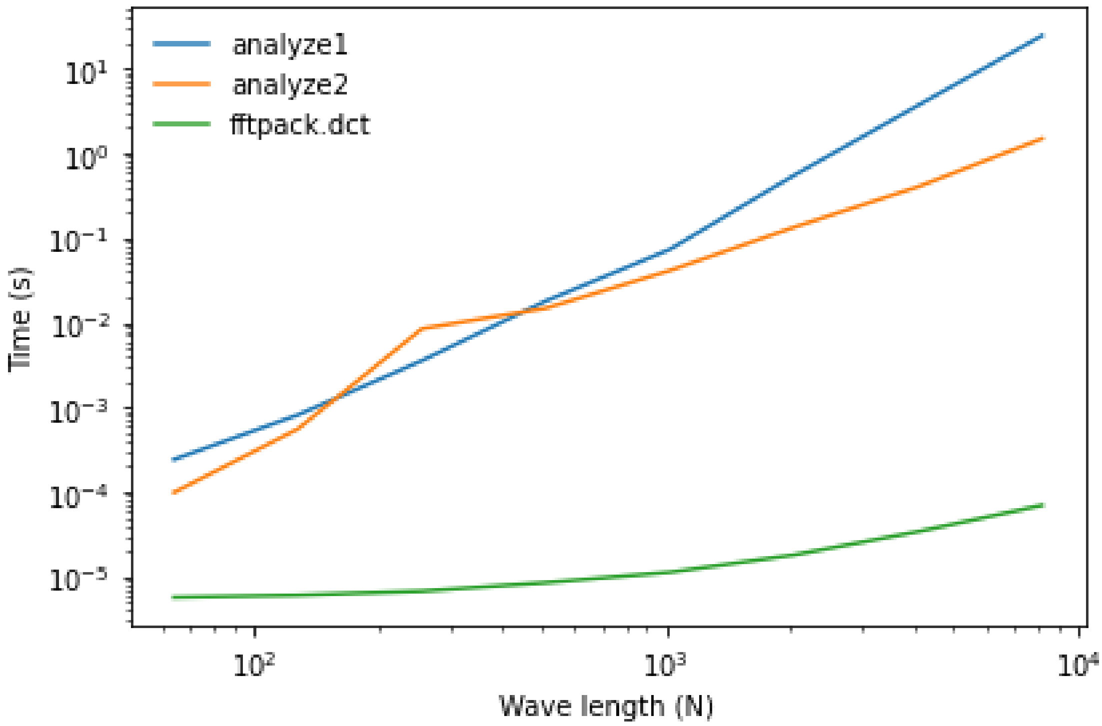

Рисунок 6.4. Время работы различных методов ДКП

6.2. Упражнение 2

Реализуем аглоритм ДКП, который предназначен для сжатия звука и изображений. Возьмем звук гитары и выделим из него короткий сегмент

```
if not os.path.exists('469283__matt141141__cm7-dm7-115bpm-loop.wav'):
    !wget https://github.com/sergeyfedorov02/Telecom/raw/main/469283
        __matt141141__cm7-dm7-115bpm-loop.wav

wave = read_wave('469283__matt141141__cm7-dm7-115bpm-loop.wav')

wave.make_audio()

segment = wave.segment(start = 2, duration = 0.8)
segment.normalize()
segment.make_audio()
```

Построим DCT график для данного сегмента

```
segment_dct = segment.make_dct()
segment_dct.plot(high = 4000)
decorate(xlabel='Frequency (Hz)', ylabel='DCT')
```

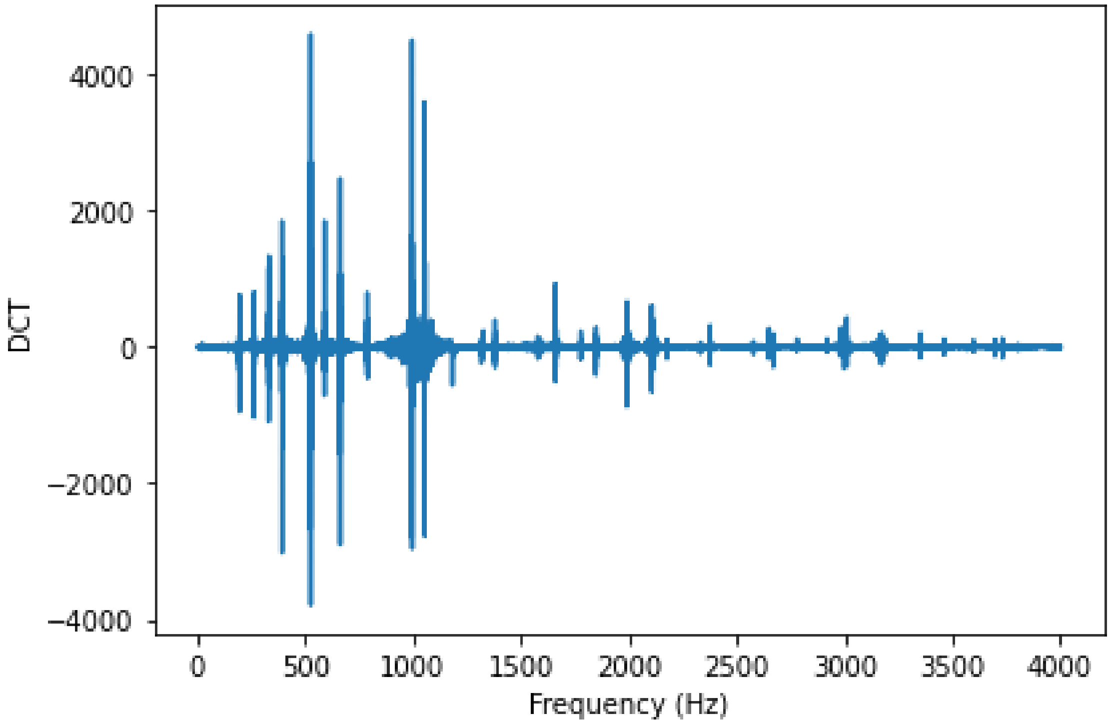

Рисунок 6.5. Спектр сигнала полученный при помощи DCT

Исходя из графика видно, что есть частоты с большой амплитудой. Далее воспользуемся функцией compress, которая берет DCT и режет элементы, которые ниже аргумента thresh и применим её.

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

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

```
segment_dct = segment.make_dct()
compress(segment_dct, thresh = 200)
segment_dct.plot(high = 4000)
```

65803 66150 99.47543461829176

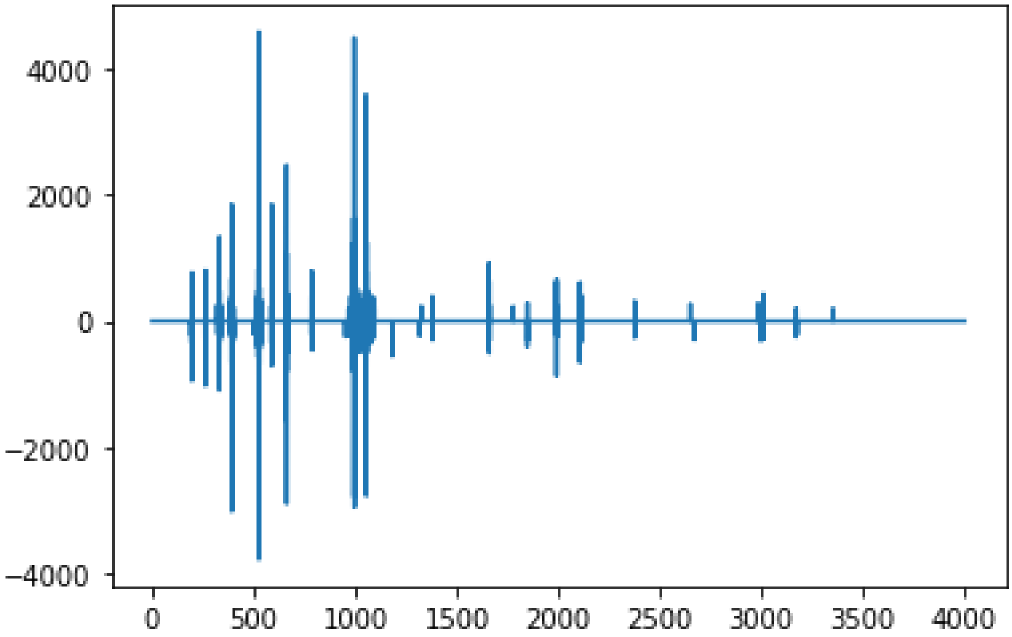

Рисунок 6.6. ДКП после фильтрации

Звучание обратного сигнала

```
segment2 = segment_dct.make_wave()
segment2.make_audio()
```

6.3. Управжнение 3

Воспользуемся блокнотом phase.ipynb, возьмем оттуда некоторый сегмент звука и повторим эксперименты.

```
from thinkdsp import SquareSignal

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

wave.segment(start=0.005,duration=0.01).plot()
decorate(xlabel='Time (s)')
```

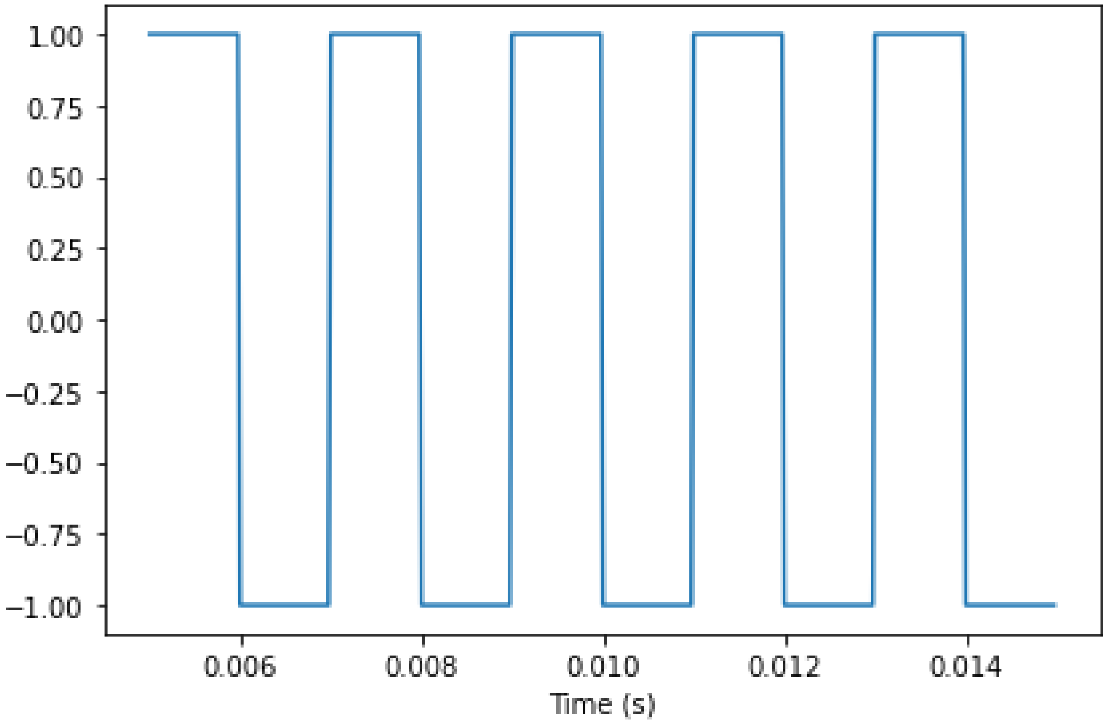

Рисунок 6.7. Выбранный сегмент

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

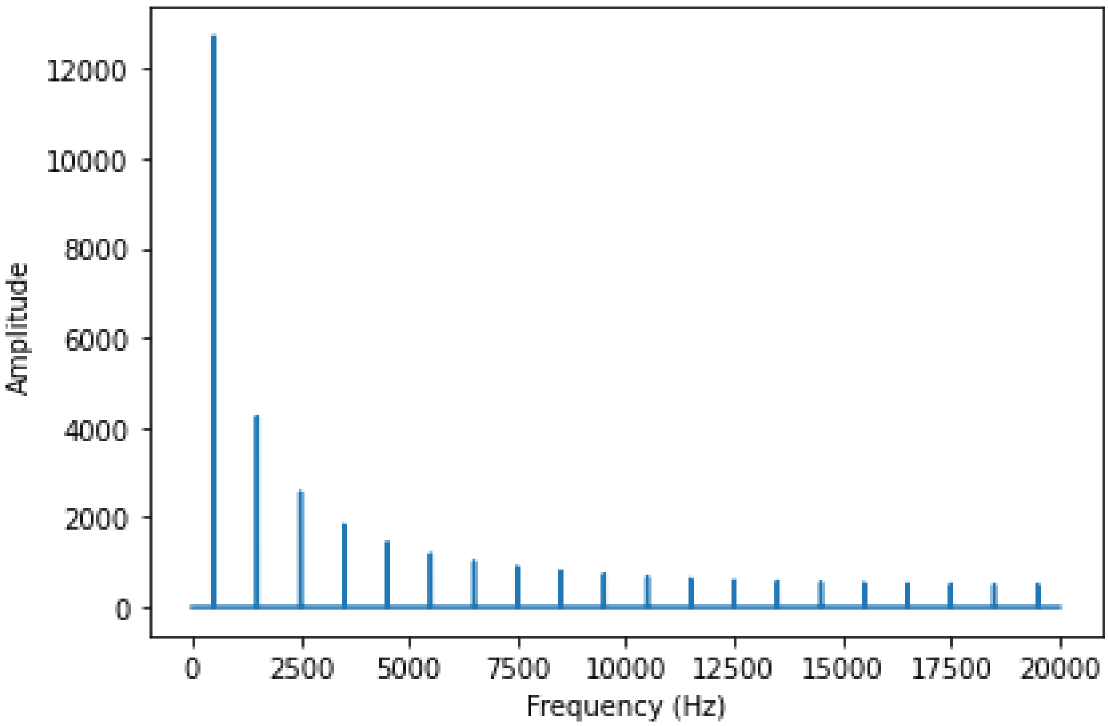

Рисунок 6.8. Спектр сегемента

```
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(spect, thresh=0)
```

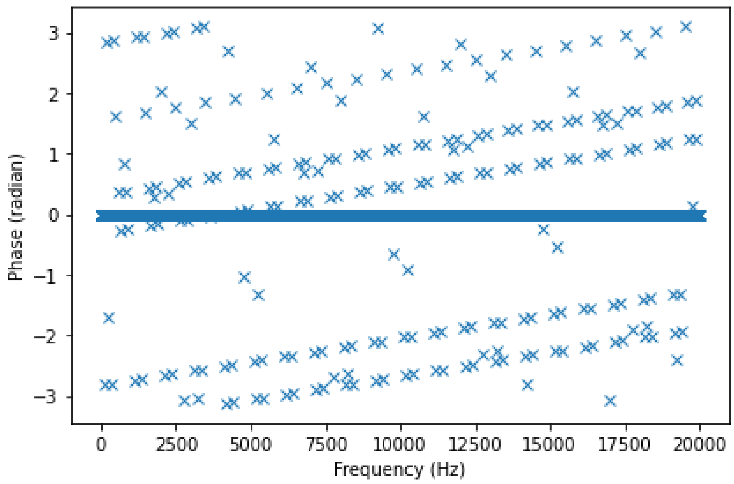

Рисунок 6.9. Получившиеся графики

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

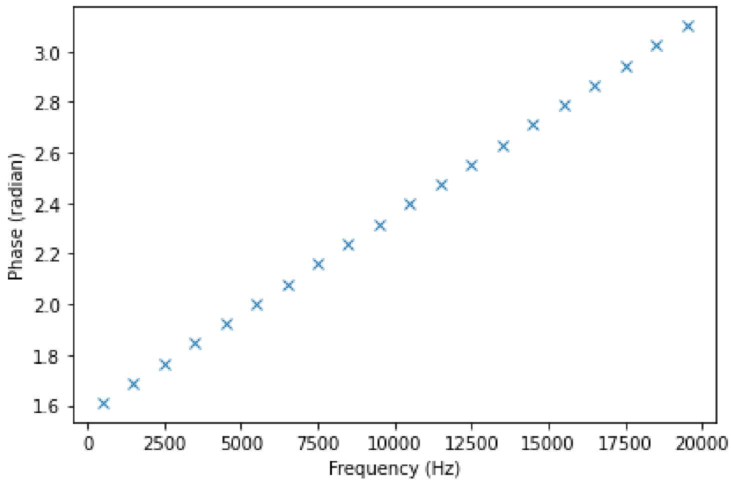

Рисунок 6.10. Получившиеся графики

```
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(spect)
```

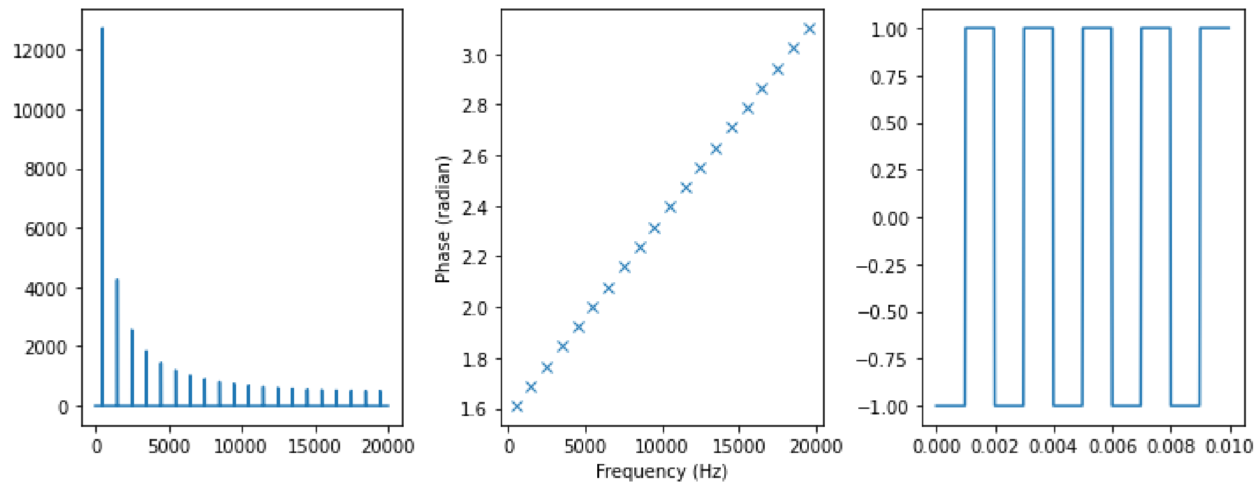

Рисунок 6.11. Получившиеся графики

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

spect2 = zero_angle(spect)
plot_three(spect2)
```

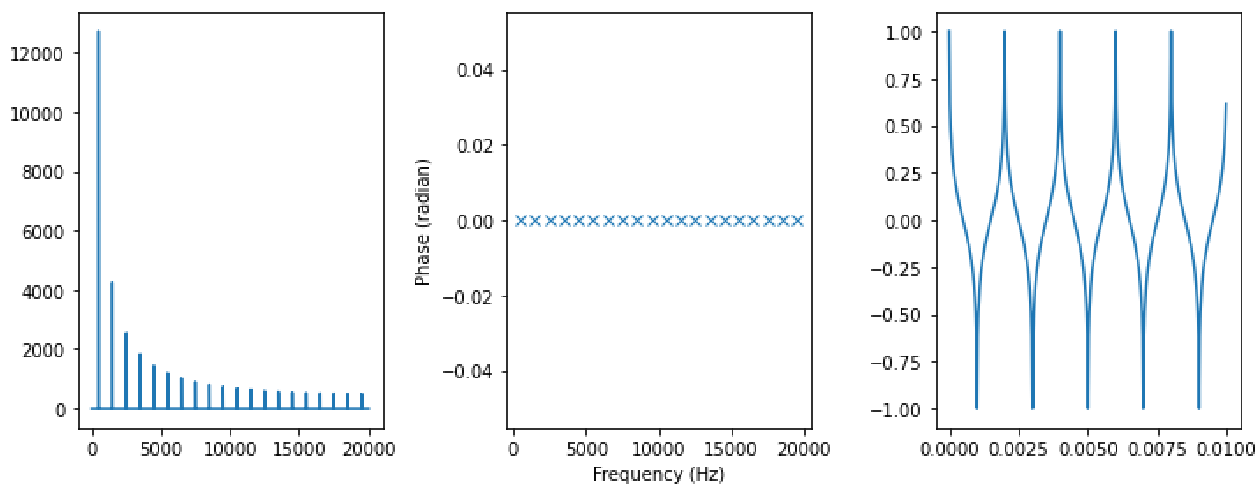

Рисунок 6.12. Получившиеся графики

```
def rotate_angle(spectrum, offset):
    res = spectrum.copy()
```

```
    res.hs *= np.exp(1j * offset)
    return res

spect3 = rotate_angle(spect, 1)
plot_three(spect3)
```

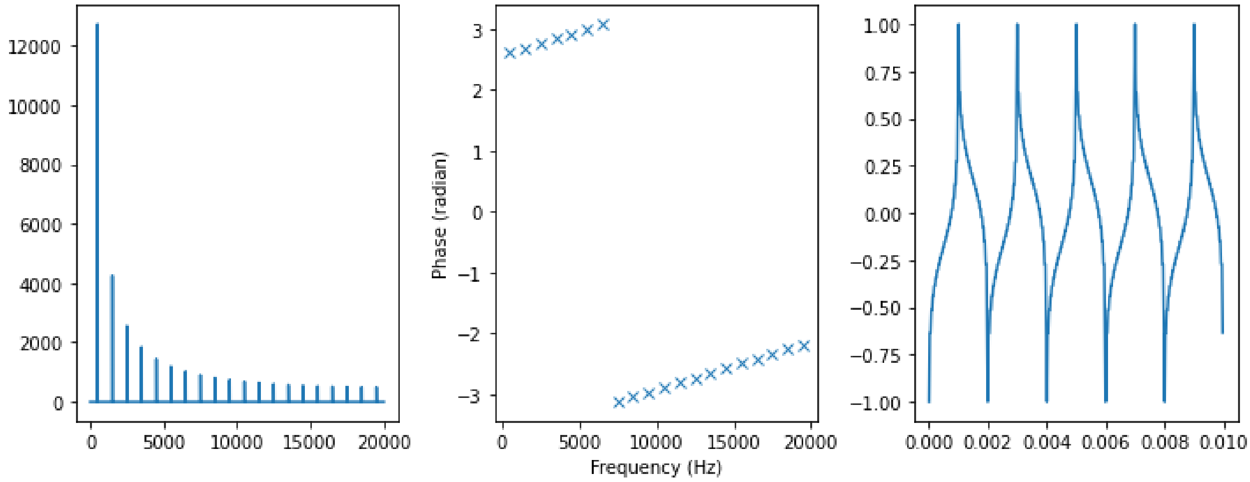

Рисунок 6.13. Получившиеся графики

Исходя из графиков видно, что сигнал довольно сильно изменился, но звучание осталось прежним

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

spect4 = random_angle(spect)
plot_three(spect4)
```

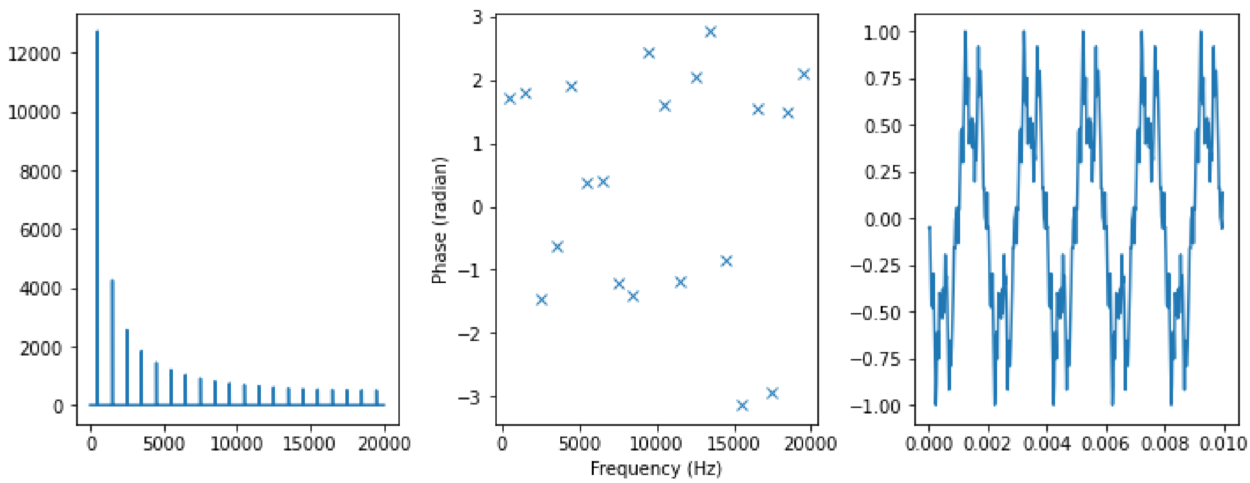

Рисунок 6.14. Получившиеся графики

Теперь загрузим собственный звук и проведем тестирование

```
if not os.path.exists('186942__lemoncreme__piano-melody.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/186942__lemoncreme__piano
        -melody.wav
wave = read_wave('186942__lemoncreme__piano-melody.wav')

wave.make_audio()

segment = wave.segment(start=1.1, duration=0.5)

spect = segment.make_spectrum()
plot_three(spect, thresh=50)
```

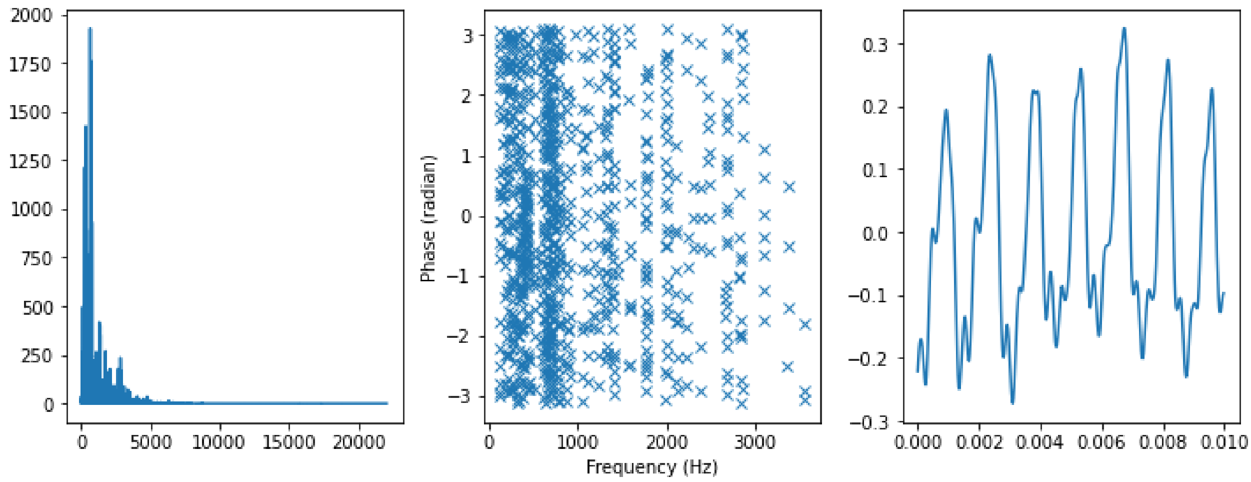

Рисунок 6.15. Получившиеся графики

```
spect2 = zero_angle(spect)
plot_three(spect2, thresh=50)
```

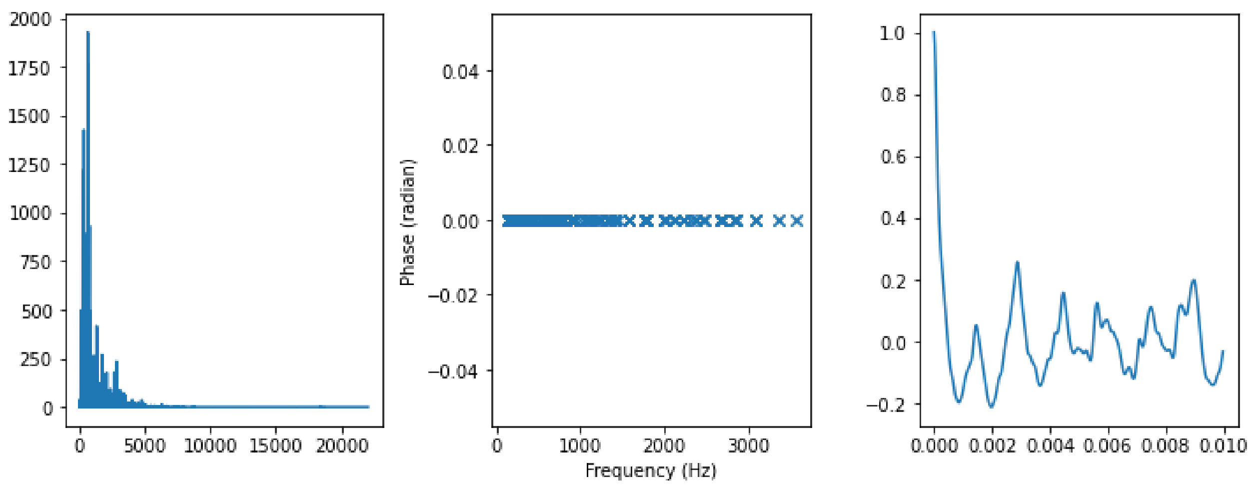

Рисунок 6.16. Получившиеся графики

Звучит как в реверсе

```
spect3 = rotate_angle(spect, 1)
plot_three(spect3, thresh=50)
```

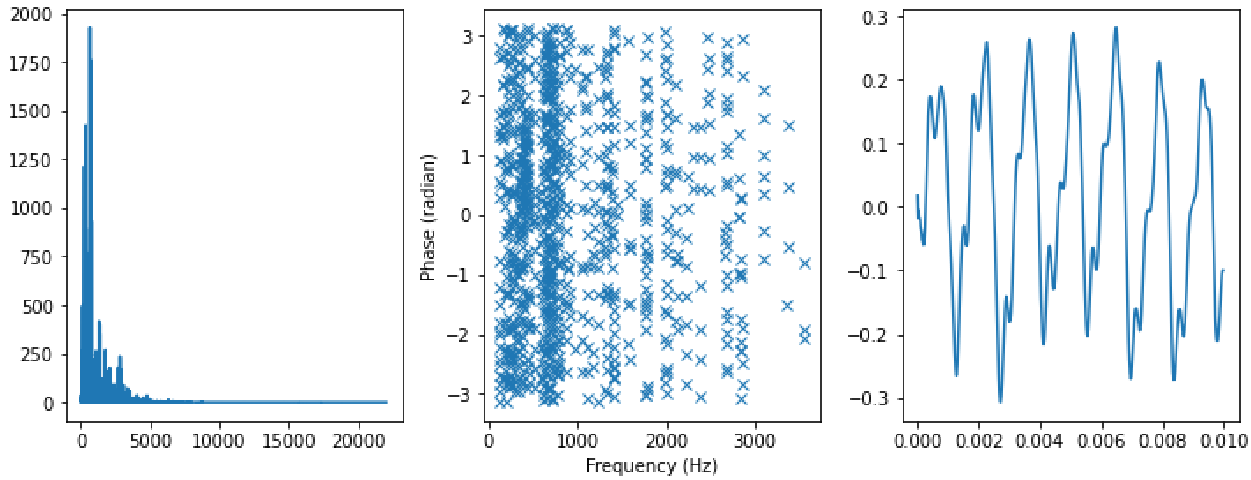

Рисунок 6.17. Получившиеся графики

```
spect4 = random_angle(spect)
plot_three(spect4, thresh=50)
```

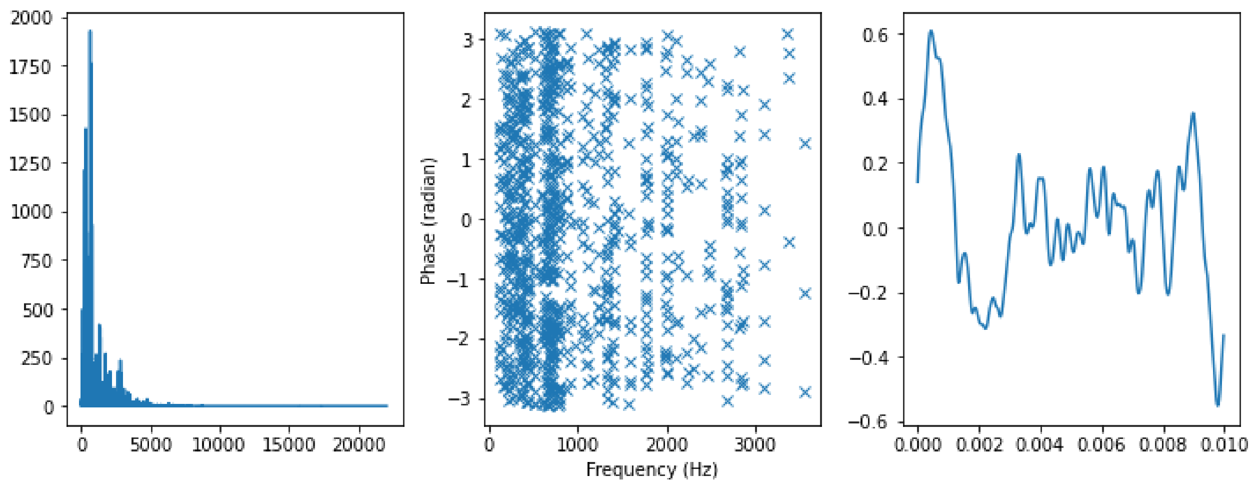

Рисунок 6.18. Получившиеся графики

Для звуков с простой гармонической структурой мы не слышим измнения в фазовой структуре, при условии что гармоническая структура неизменна.

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

В этом упражнении требуется реализовать алгоритм Быстрого преобразования Фурье, его время работы пропорционально n logn

```
import numpy as np
PI2 = 2 * np.pi
```

Возьмем простой массив

```
ys = [0.2, 0.3, -0.7, -0.2]
hs = np.fft.fft(ys)
hs

array([-0.4+0.j ,  0.9-0.5j, -0.6+0.j ,  0.9+0.5j])
```

Дискретное преобразование Фурье

```
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
```

```
dft(ys)

array([-0.4+0.j ,  0.9-0.5j, -0.6-0.j ,  0.9+0.5j])
```

Функция для рекурсивного быстрого преобразования Фурье

```
def fft_1(ys):
    N = len(ys)
    e_arr = np.fft.fft(ys[::2])
    o_arr = np.fft.fft(ys[1::2])

    ns = np.arange(N)
    W = np.exp(-1j * PI2 * ns / N)

    return np.tile(e_arr, 2) + W * np.tile(o_arr, 2)

fft_1(ys)

array([-0.4+0.j ,  0.9-0.5j, -0.6-0.j ,  0.9+0.5j])
```

Теперь применим рекурсию для np.fft.fft

```
def fft_2(ys):
    if len(ys) == 1:
      return ys

    e_arr = fft_1(ys[::2])
    o_arr = fft_1(ys[1::2])

    ns = np.arange(len(ys))
    W = np.exp(-1j * PI2 * ns / len(ys))

```

```
    return np.tile(e_arr, 2) + W * np.tile(o_arr, 2)

fft_2(ys)

array([-0.4+0.j ,  0.9-0.5j, -0.6-0.j ,  0.9+0.5j])
```

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 1

Выясним, что случится, при увеличении ширины Гауссова окна std не увеличивая ширину окна m

Функция расширения массива нулями

```
from thinkdsp import SquareSignal
from thinkdsp import decorate

def zero_pad(array, n):
    """Extends an array with zeros.

    array: NumPy array
    n: length of result

    returns: new NumPy array
    """
    res = np.zeros(n)
    res[:len(array)] = array
    return res


def plot_filter(M=11, std=2):
    signal = SquareSignal(freq=440)
    wave = signal.make_wave(duration=1, framerate=44100)
    spectrum = wave.make_spectrum()

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

    ys = np.convolve(wave.ys, gaussian, mode='same')
    smooth =  Wave(ys, framerate=wave.framerate)
    spectrum2 = smooth.make_spectrum()

    # plot the ratio of the original and smoothed spectrum
    amps = spectrum.amps
    amps2 = spectrum2.amps
    ratio = amps2 / amps
    ratio[amps<560] = 0

    # plot the same ratio along with the FFT of the window
    padded =  zero_pad(gaussian, len(wave))
    dft_gaussian = np.fft.rfft(padded)

    plt.plot(np.abs(dft_gaussian), color='gray', label='Gaussian filter')
    plt.plot(ratio, label='amplitude ratio')

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

```
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets
from thinkdsp import Wave
import scipy.signal

slider = widgets.IntSlider(min=2, max=100, value=11)
```

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

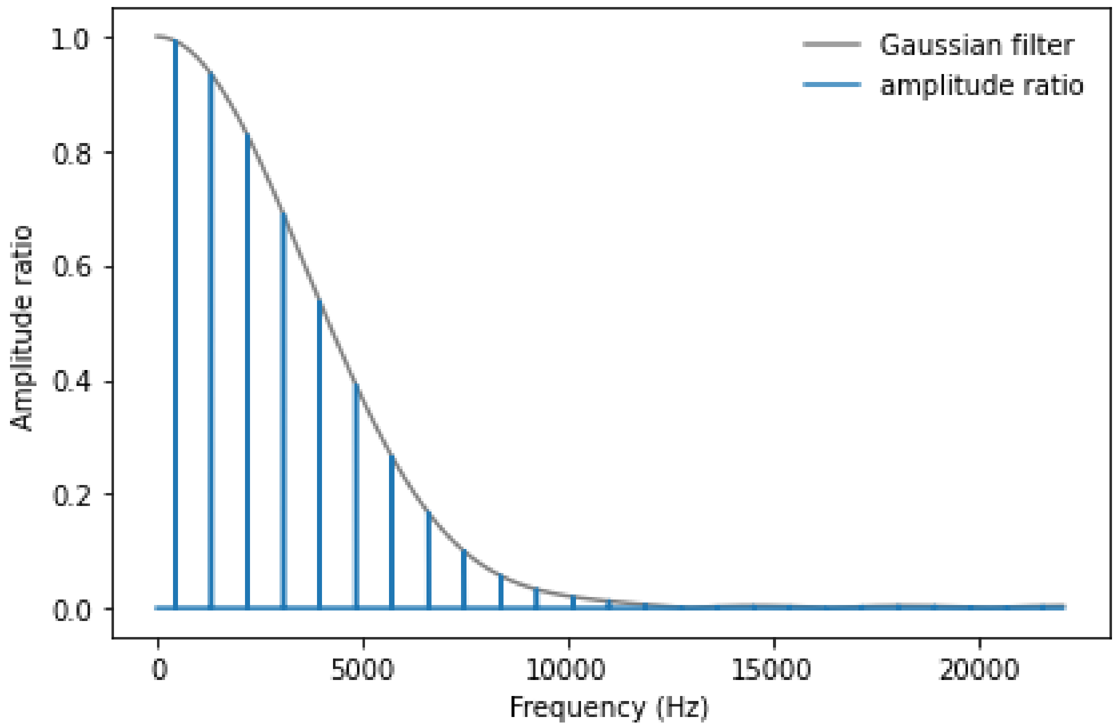

Рисунок 8.1. Гауссово окно для фильтрации

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

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

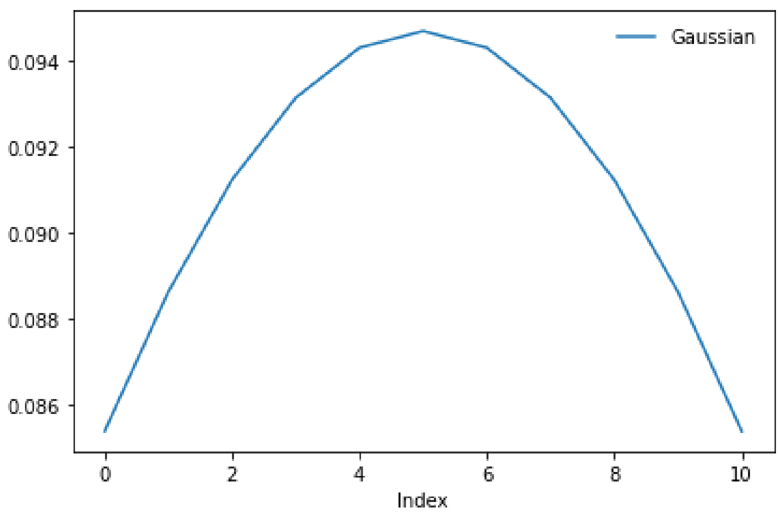

Рисунок 8.2. Гауссово окно

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

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

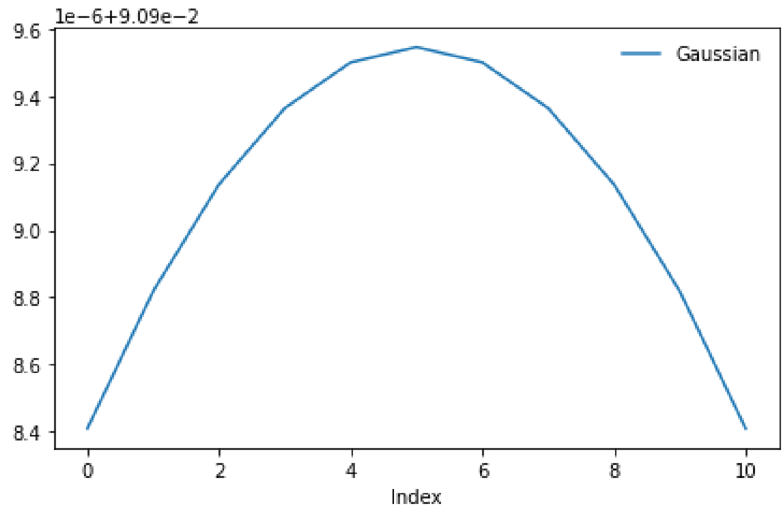

Рисунок 8.3. Гауссово окно

Исходя из результатов видно, что при увеличении std -> кривая становится шире, а сам БПФ меньше (уже).

8.2. Упражнение 2

Протестируем, является ли преобразование Фурье гауссовой кривой - также гауссовой кривой.

Кривая Гаусса

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

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

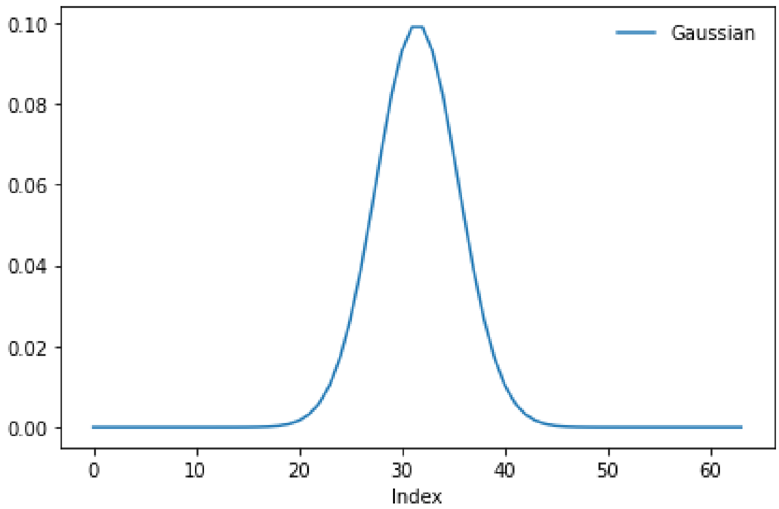

Рисунок 8.4. Гауссово окно

Применим БПФ

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

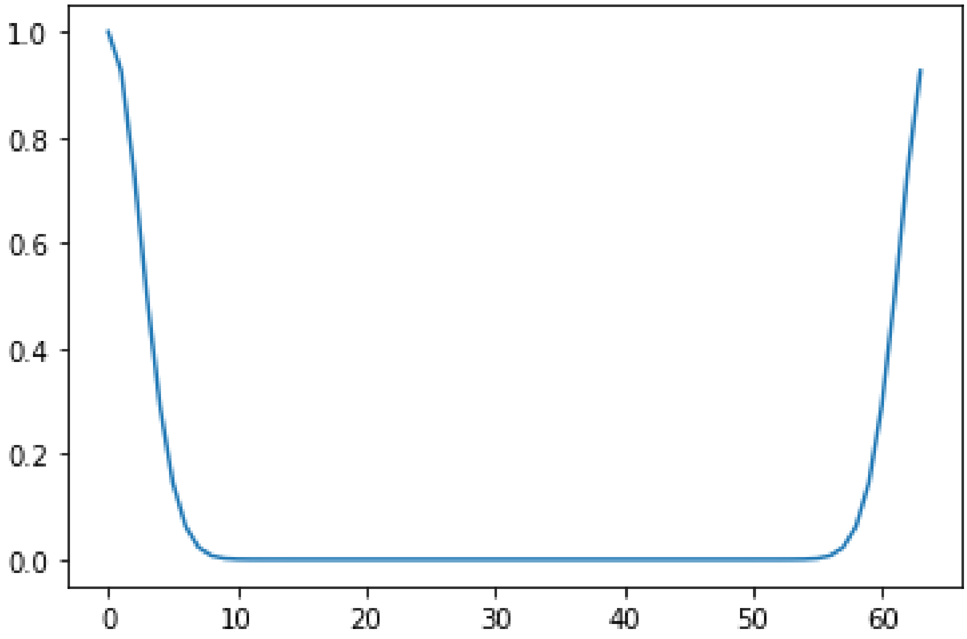

Рисунок 8.5. FTT применённое на окно

Произведем свертку отрицательных частот влево.

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

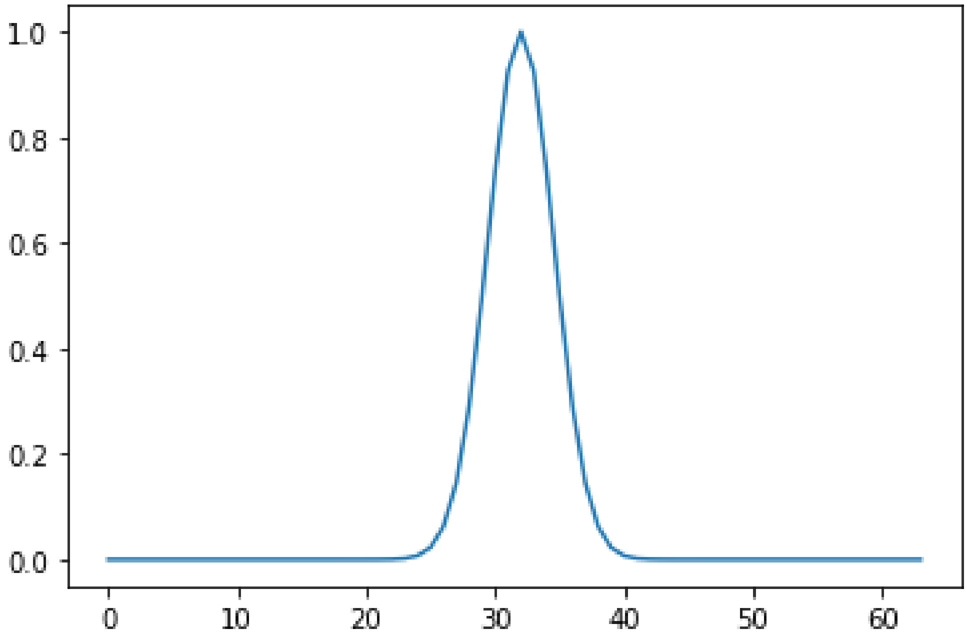

Рисунок 8.6. Результат

Преобразование Фурье гауссовой кривой приблизительно похоже на гауссову кривую

8.3. Упражнение 3

Поэксперементтируем с окнами, поищем подходящее для НЧ

```
signal = SquareSignal(freq=220)
wave = signal.make_wave(duration=1, framerate=44100)

M = 8
std = 2

gaussian = scipy.signal.gaussian(M=M, std=std)
blackman = np.blackman(M)
bartlett = np.bartlett(M)
hanning = np.hanning(M)
hamming= np.hamming(M)

windows  = [gaussian, blackman, bartlett, hanning, hamming]
labels = ['gaussian','blackman','bartlett','hanning','hamming']

for element, label in zip(windows,labels):
    element /= sum(element)
    plt.plot(element,label=label)

plt.legend()
```

Построим графики для нескольких окон

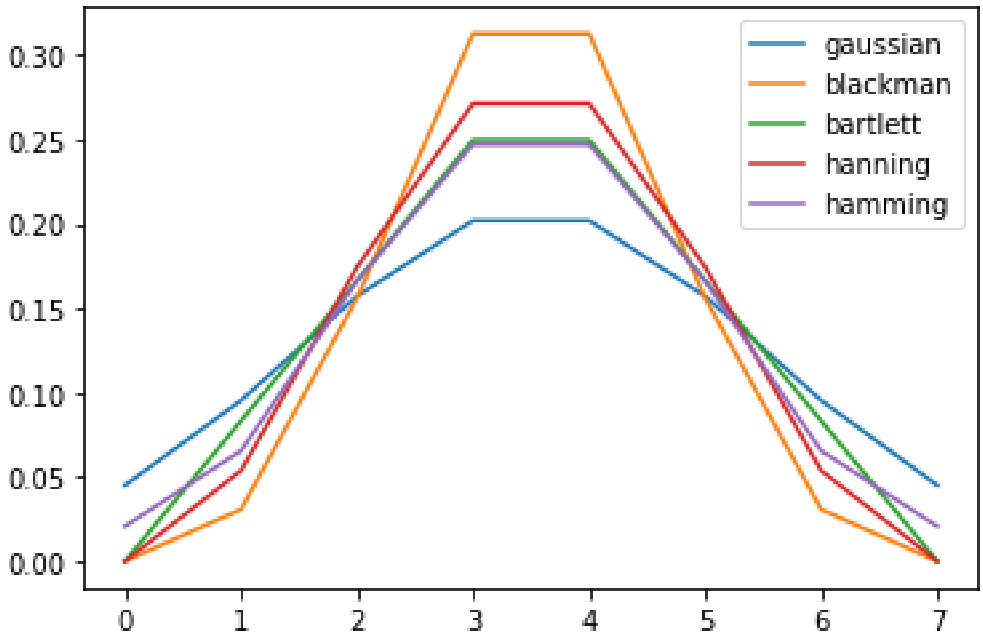

Рисунок 8.7. Применение различных окон на выбранный сигнал

Графики дикретного преобразования Фурье

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

plt.legend()
```

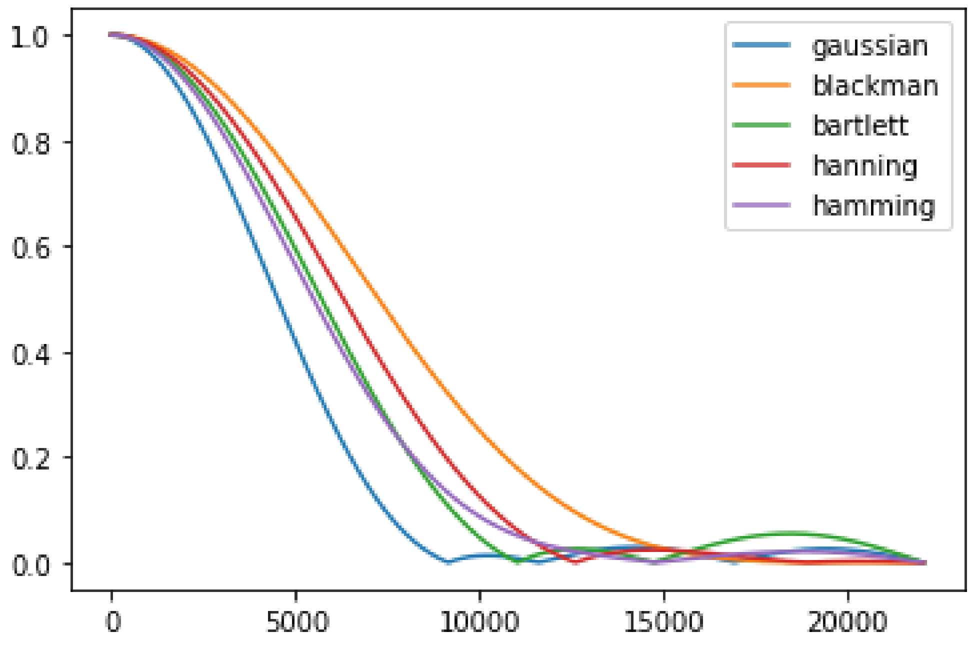

Рисунок 8.8. Применение различных окон на выбранный сигнал

```python
for element, label in zip(windows, labels):
    padded =  zero_pad(element, len(wave))
    dft_window = np.fft.rfft(padded)
    plt.plot(abs(dft_window), label=label)

plt.legend()
decorate(yscale='log')
```

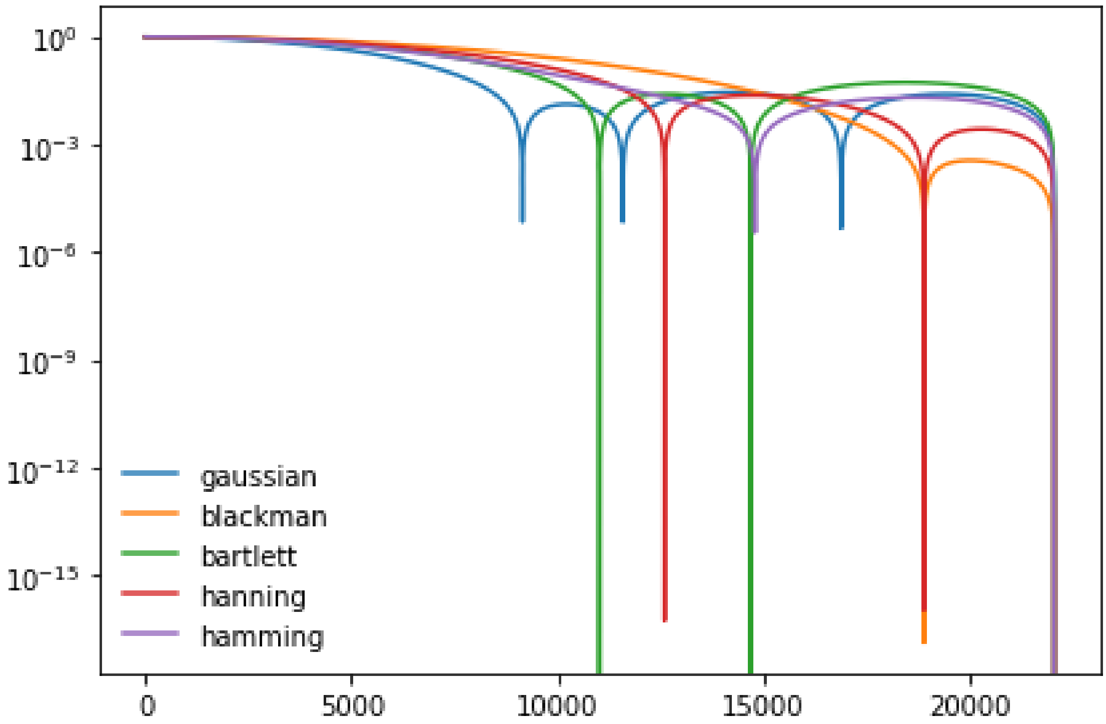

Рисунок 8.9. Применение различных окон на выбранный сигнал в лагорифмическом масштабе

Исходя из этого и предыдущих графиков можно сделать вывод, что Хэнинг лучше всего подоёдет для фильтрации низких частот за счет быстрого спада и минимальных боковых лепестков.

8.4. 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

Изучим влияние diff и differentiate на сигнал. Создадим треугольный сигнал.

```
from thinkdsp import TriangleSignal

wave = TriangleSignal(freq=40).make_wave(duration=0.2, framerate=44100)
wave.plot()
decorate(xlabel='Time (s)')
```

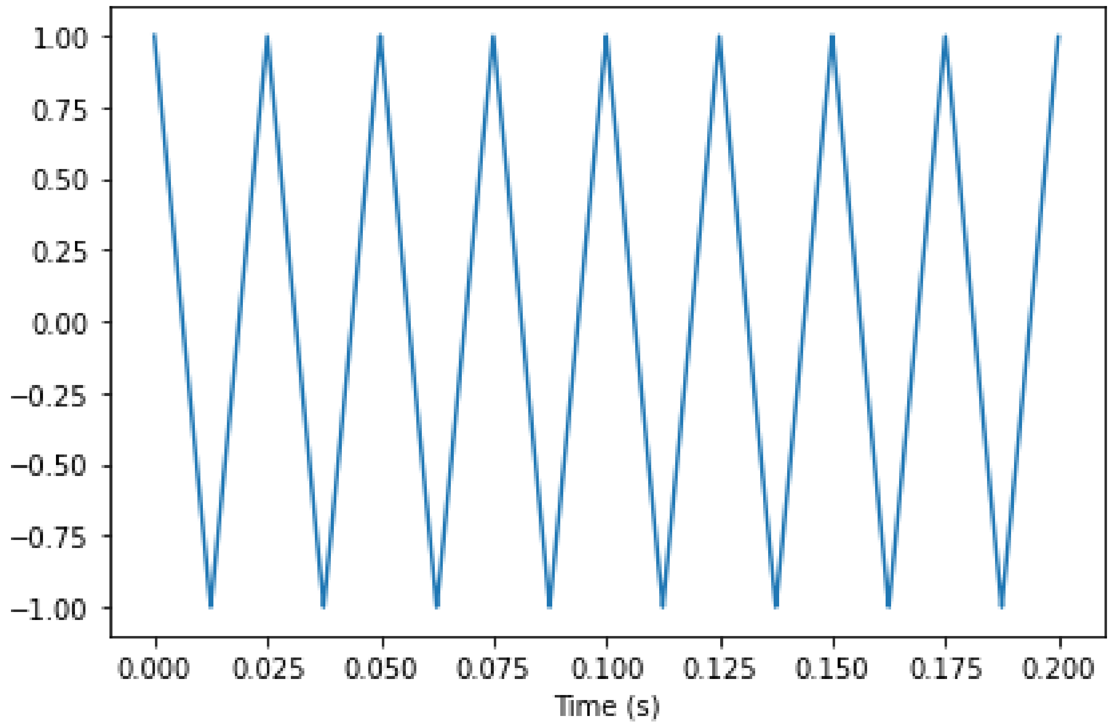

Рисунок 9.1. График сигнала

Функция diff

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

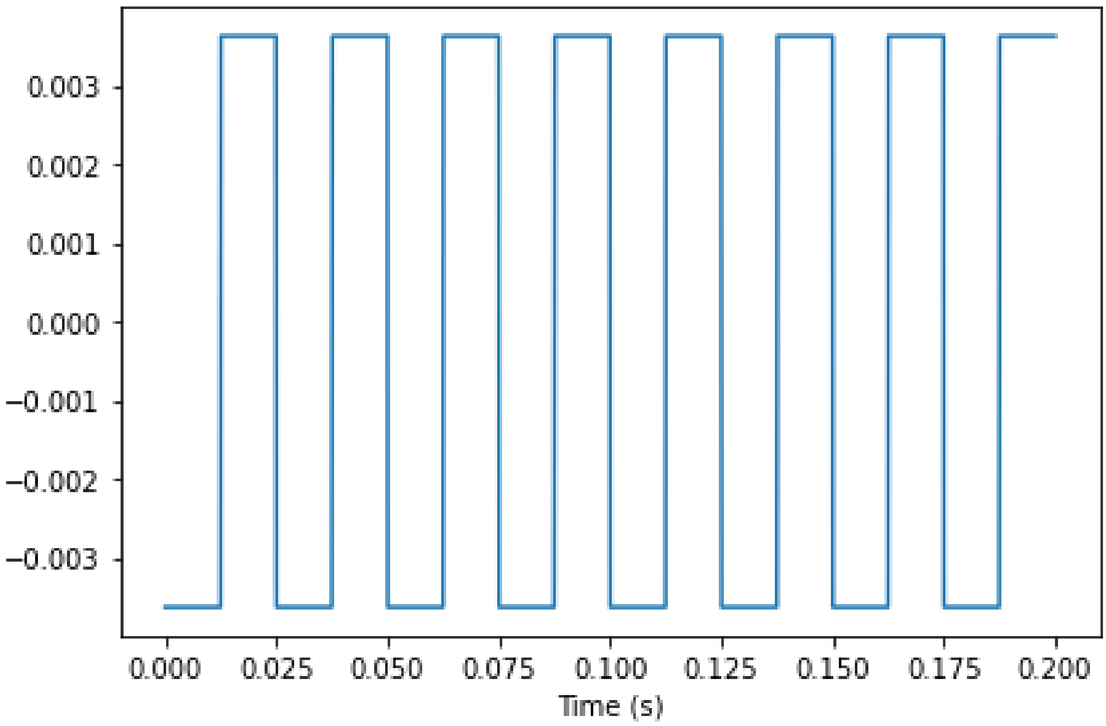

Рисунок 9.2. Сигнал после применения diff

Функция differentiate

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

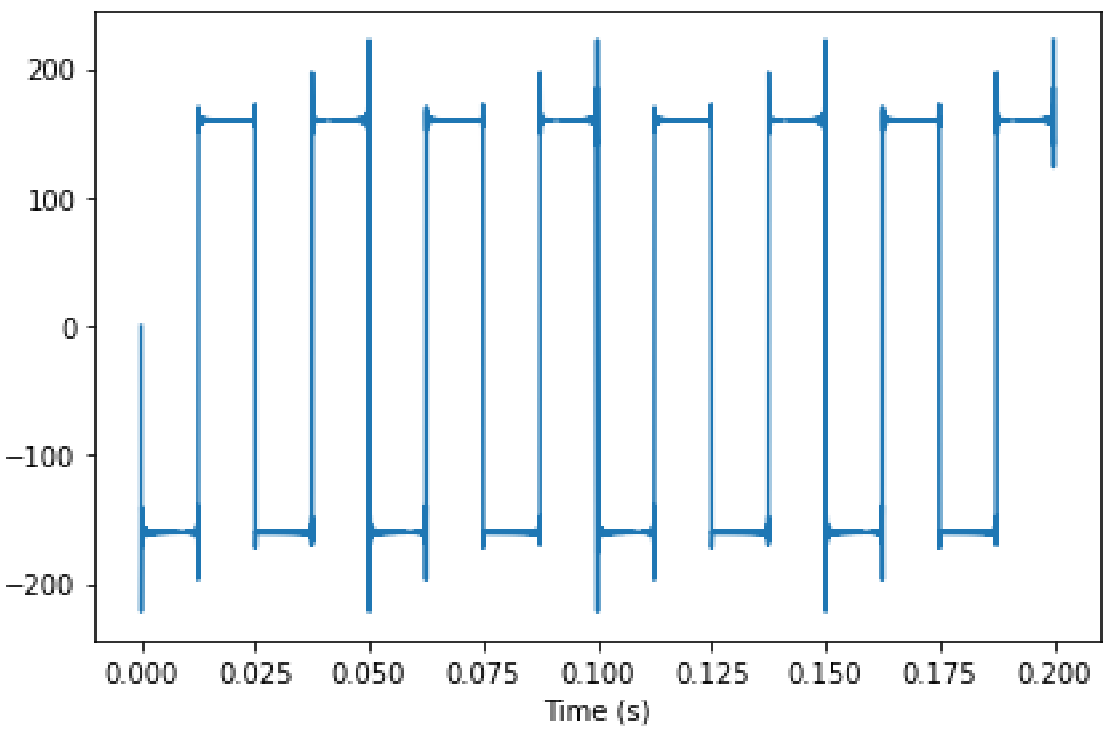

Рисунок 9.3. Сигнал после применения differentiate

Видны различия в местах перегиба

9.2. Упражнение 2

Испытаем cumsum и integrate на прямоугольном сигнале

```
from thinkdsp import SquareSignal

wave = SquareSignal(freq=40).make_wave(duration=0.2, framerate=44100)
wave.plot()
decorate(xlabel='Time (s)')
```

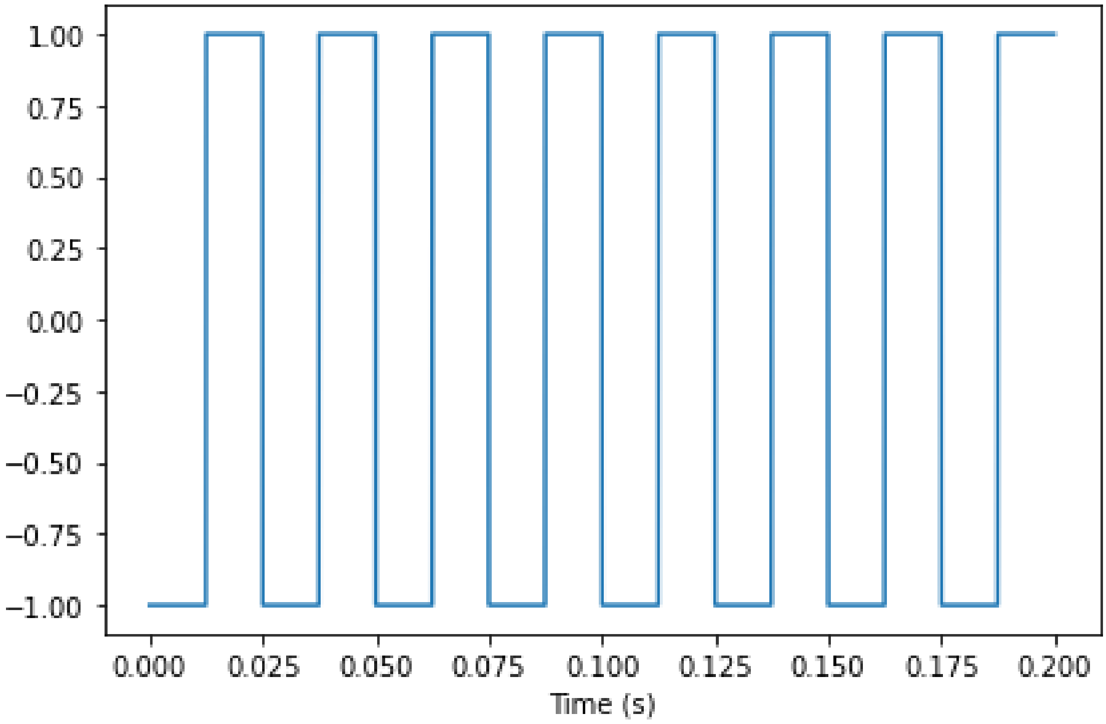

Рисунок 9.4. Рассматриваемый сигнал

cumsum

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

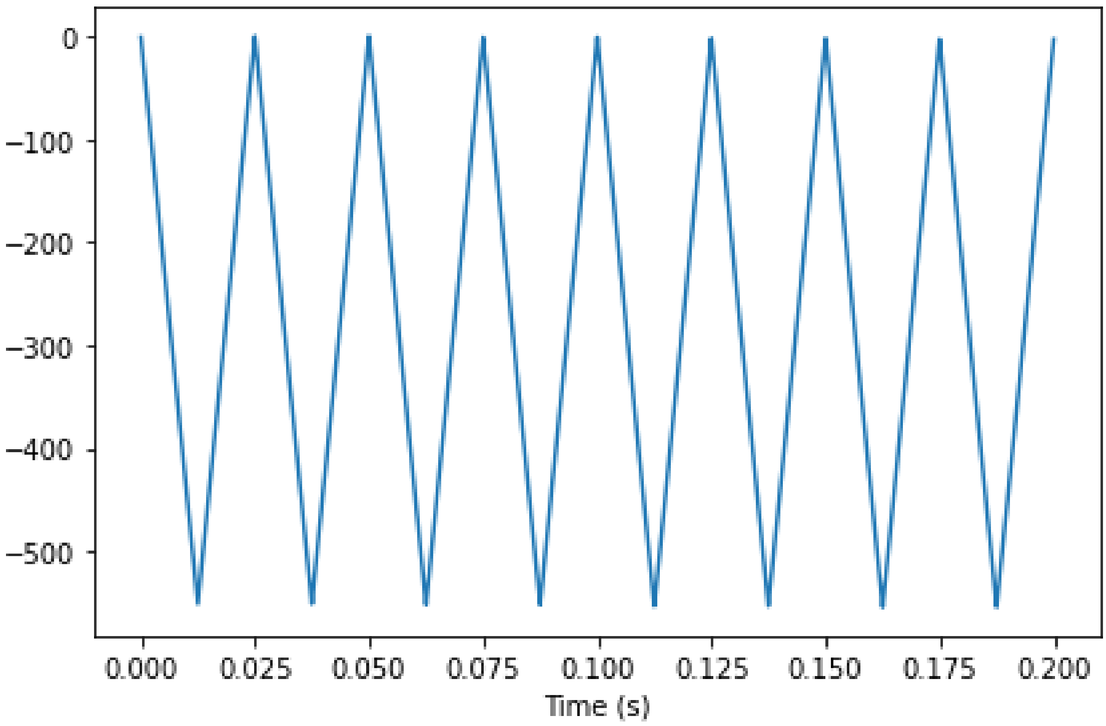

Рисунок 9.5. Рассматриваемый сигнал после применения cumsum

Получился треугольный сигнал, так как мы аппроксимировали интегрирование. Используем integrate на спектре и обратно перобразуем в волну

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

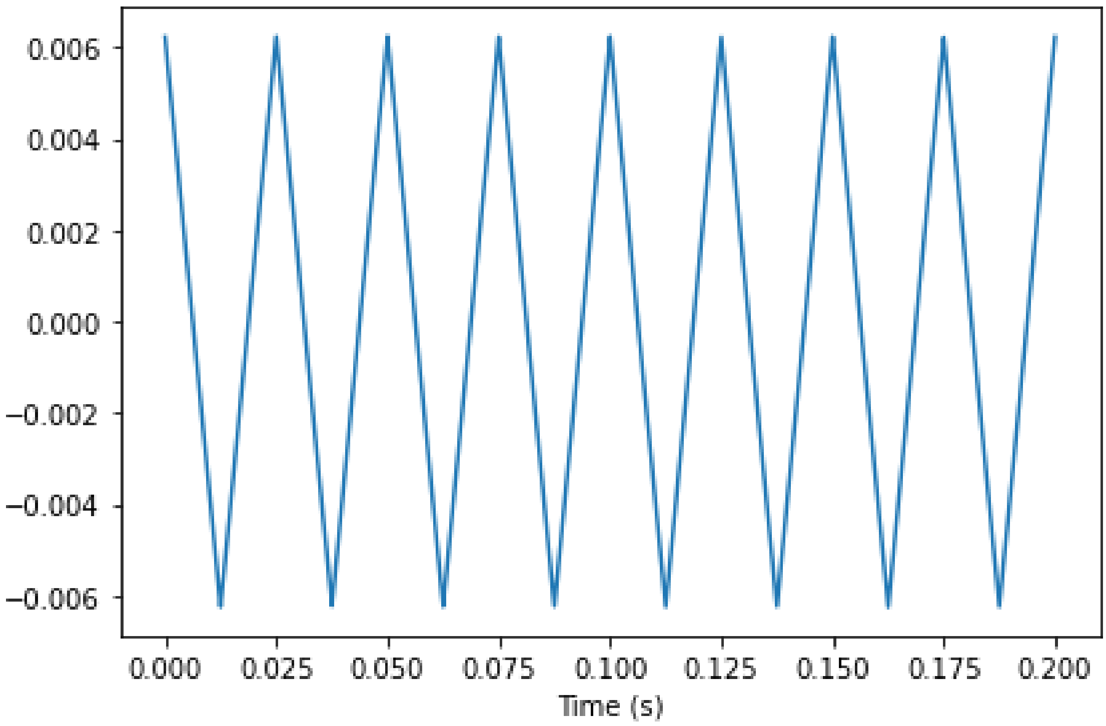

Рисунок 9.6. Рассматриваемый сигнал после применения integrate

Отличия только в амплитуде

9.3. Упражнение 3

Посмотрим на двойное интегрирование пилообразного сигнала

```
from thinkdsp import SawtoothSignal

wave = SawtoothSignal(freq=40).make_wave(duration=0.2, framerate=44100)
wave.plot()
decorate(xlabel='Time (s)')
```

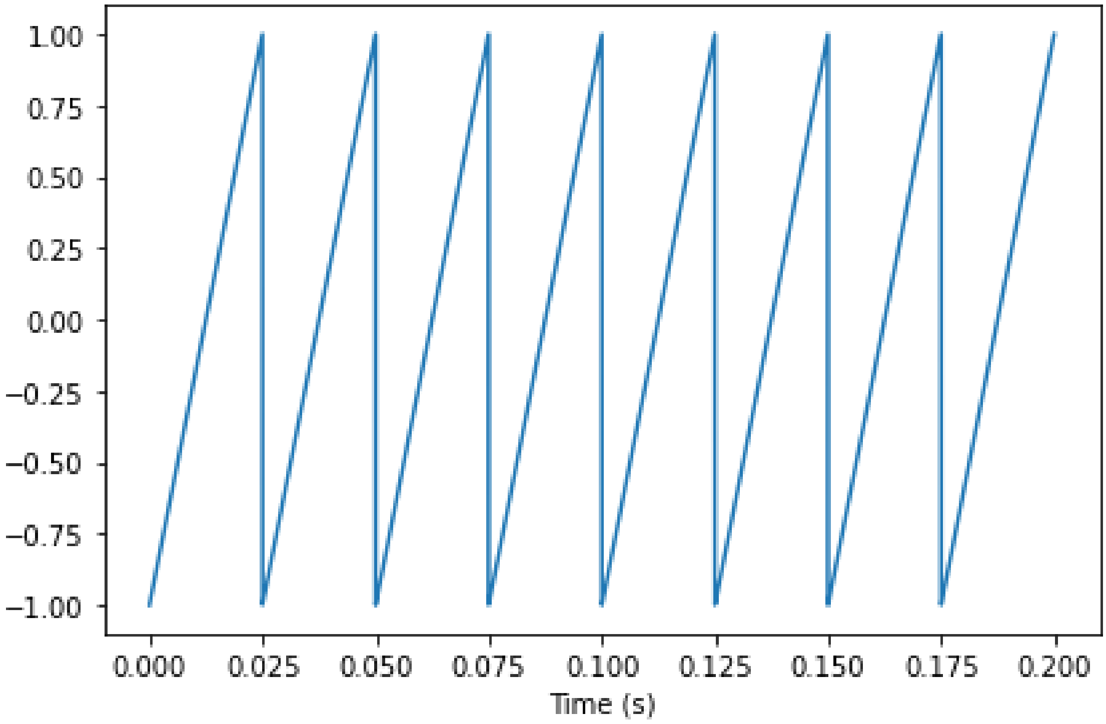

Рисунок 9.7. Пилообразный сигнал

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

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

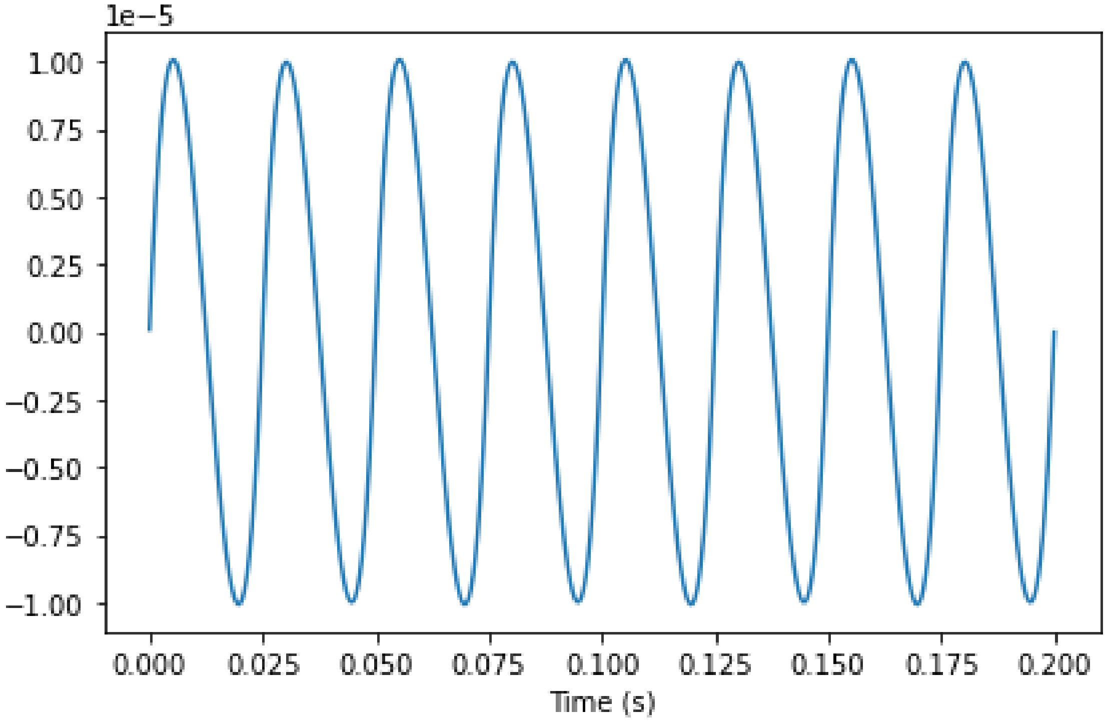

Рисунок 9.8. Двойное интегрирование

Получили кубическую кривую, который похож на синусоиду, двойное интегрирование действует как фильтр нижних частот.

```
wave_out.make_spectrum().plot(high=500)
```

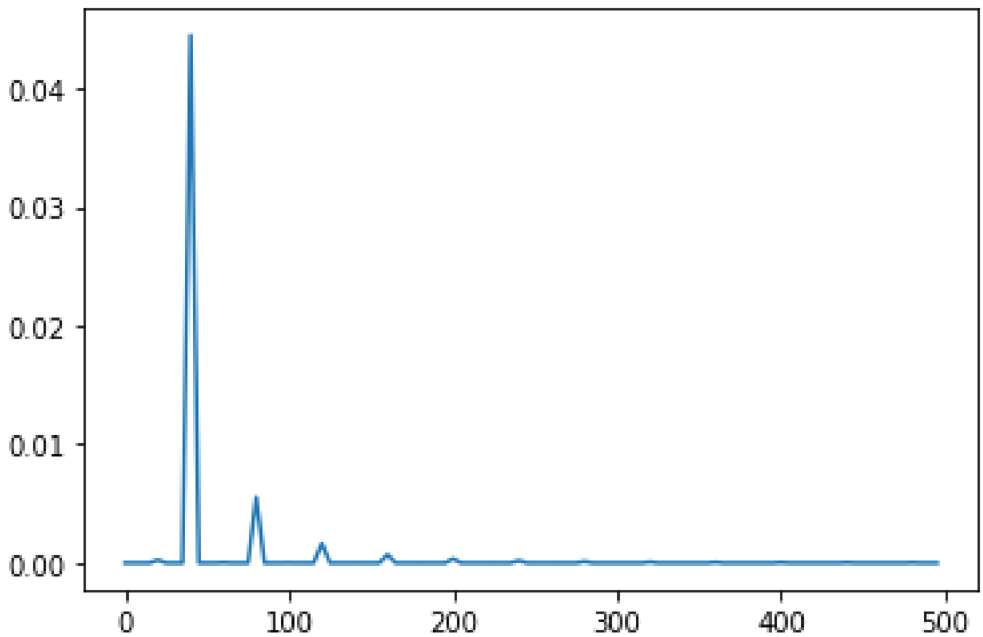

Рисунок 9.9. График

9.4. Упражнение 4

Проверм влияние второй разности и второй производной на CubicSignal

```
from thinkdsp import CubicSignal

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

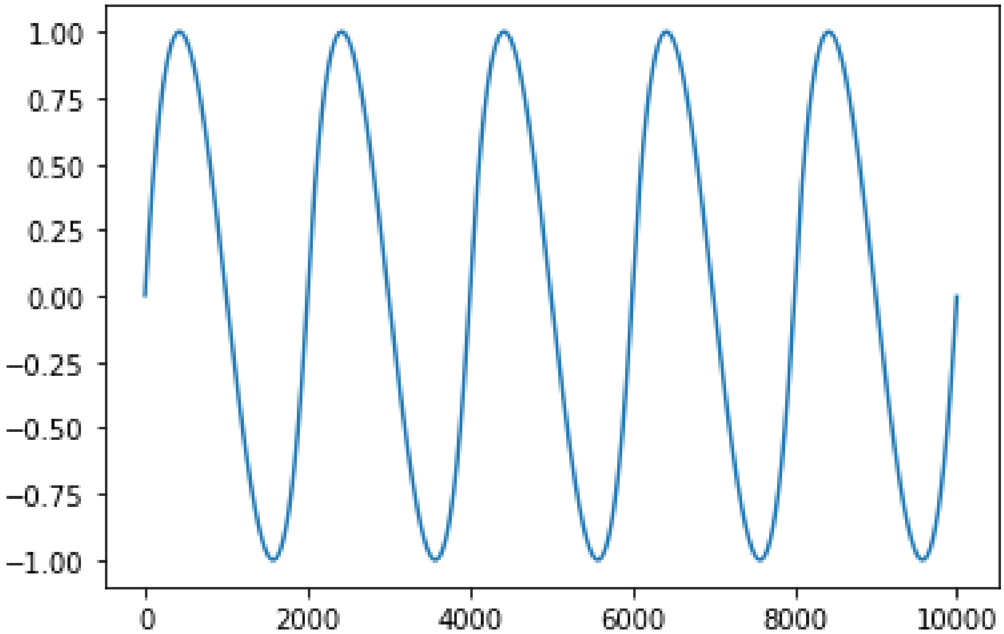

Рисунок 9.10. Кубический сигнал

```
wave_diff_1 = wave.diff()
wave_diff_1.plot()
```

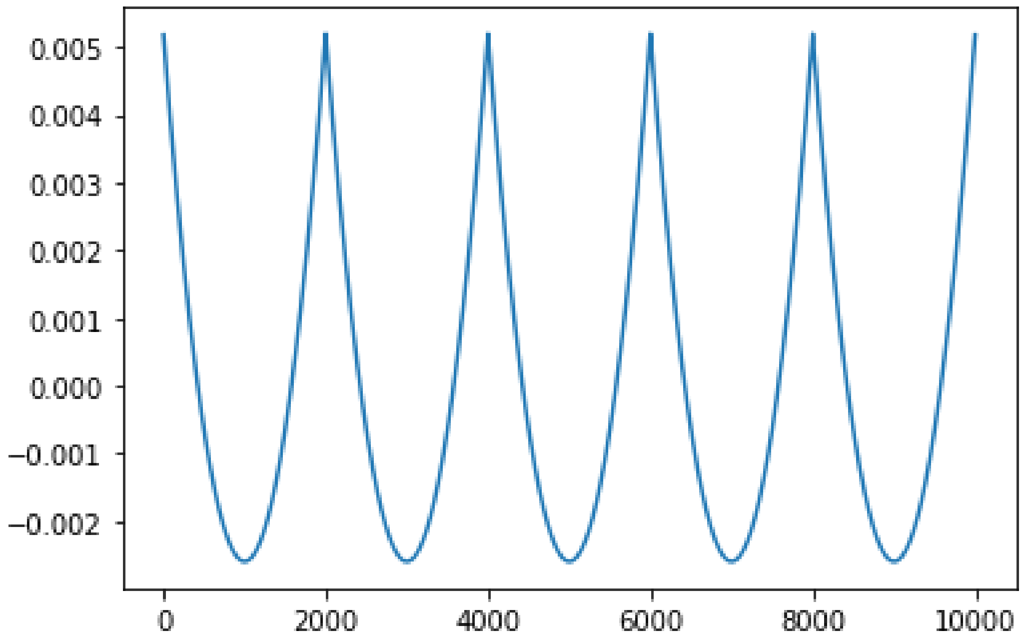

Рисунок 9.11. Первая разность

```
wave_diff_2 = wave_diff_1.diff()
wave_diff_2.plot()
```

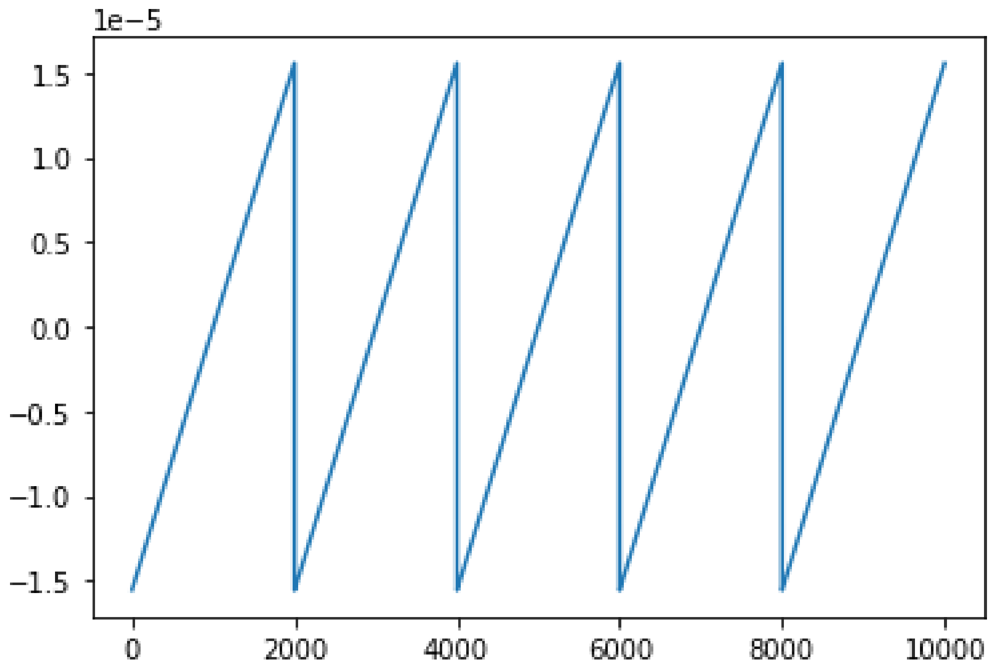

Рисунок 9.12. Вторая разность

Первая разность - это парабола, а вторая разность - пилообразный сигнал

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

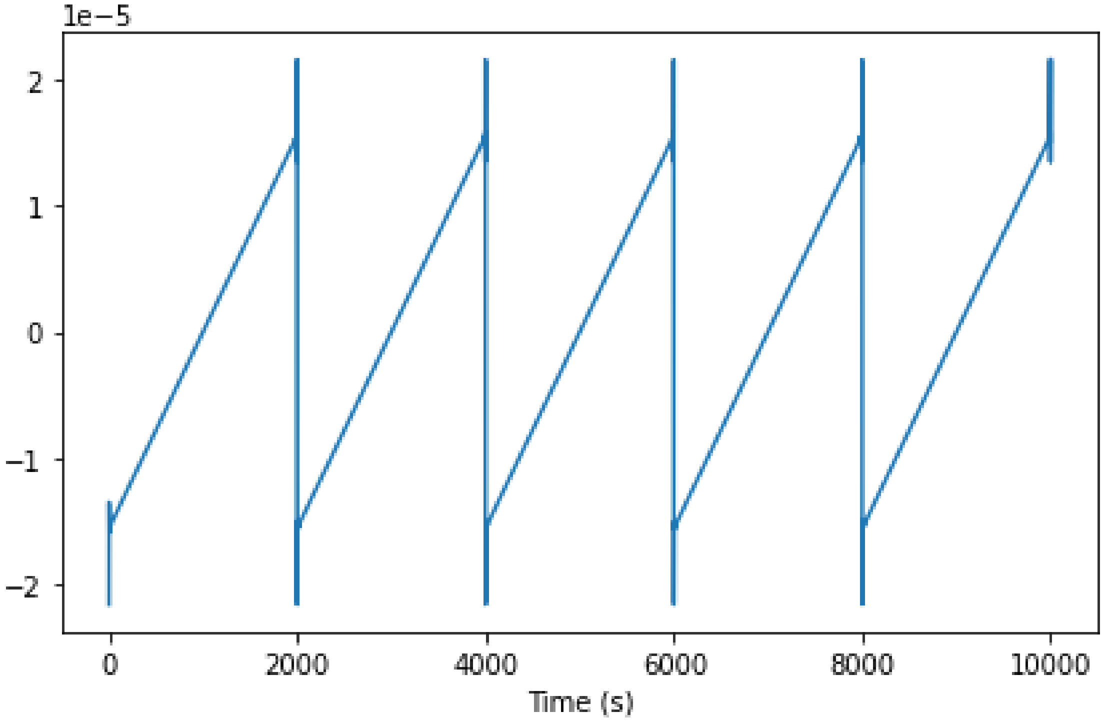

Рисунок 9.13. Полученный сигнал со звоном

После двойного дифференцирования differentiate на графике виден пилообразный сигнал с звоном, из-за того, что производная параболического сигнала в некоторых точках не определена.

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

```
from thinkdsp import zero_pad
from thinkdsp import Wave

diff_window = np.array([-1.0, 2.0, -1.0])
padded = zero_pad(diff_window, len(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')
```

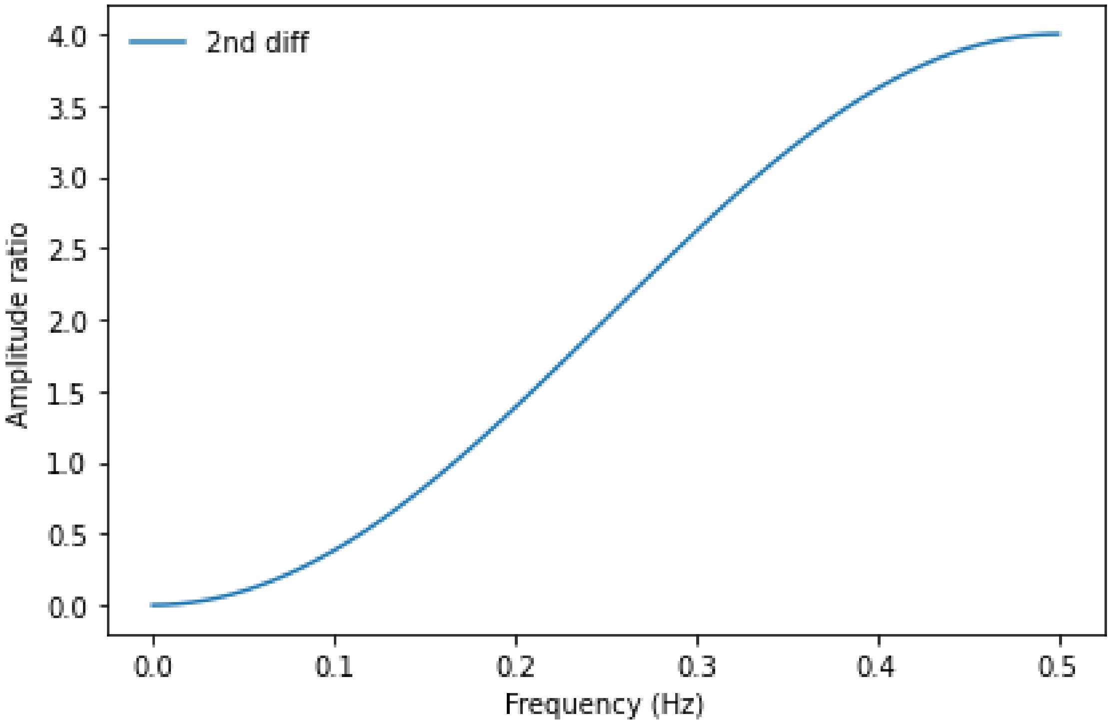

Рисунок 9.14. Первый фильтр

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

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

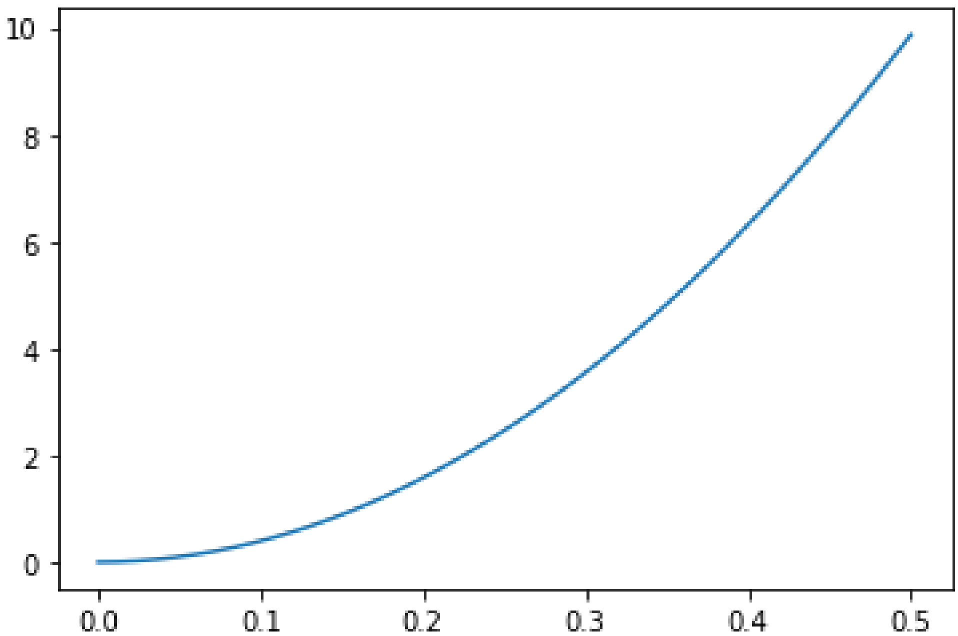

Рисунок 9.15. Второй фильтр

Сравним эти два графика

```
diff_filter.plot(label='2nd diff')
deriv_filter.plot(label='2nd deriv')

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

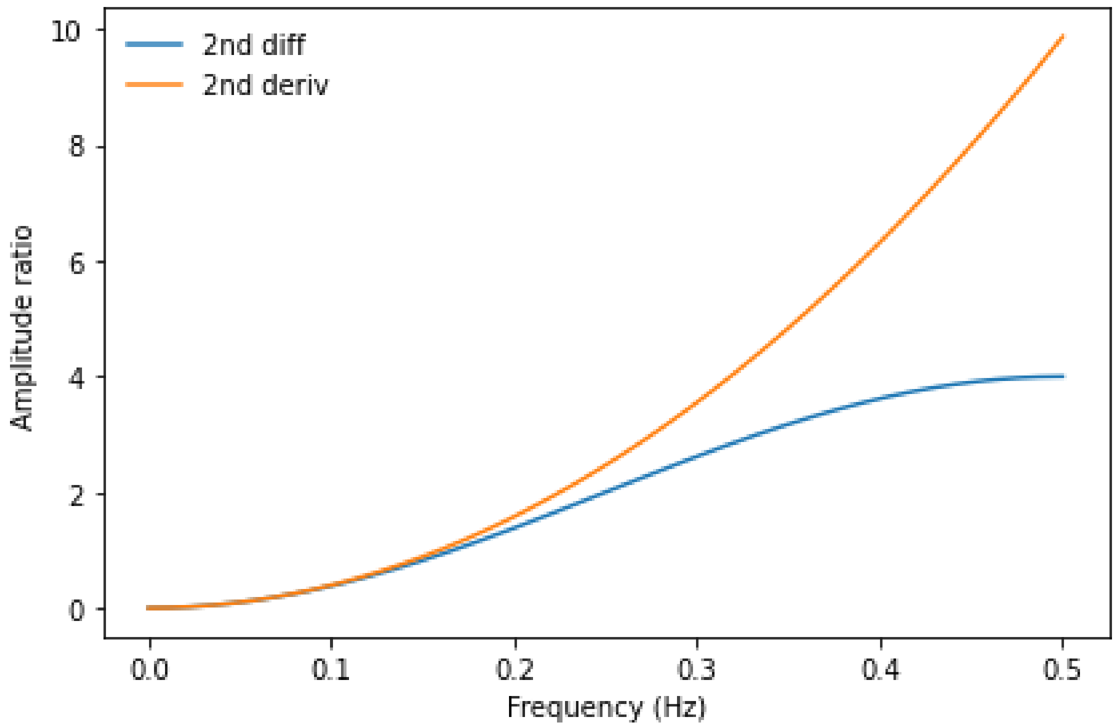

Рисунок 9.16. Сравнение фильтров

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

Изменим пример из chap10.ipynb и удостоверимся, что дополнение нулями устраняет лишнюю ноту. Урежем оба сигнала до $2^{1}62^{1}7$.

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

from thinkdsp import read_wave

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)')
```

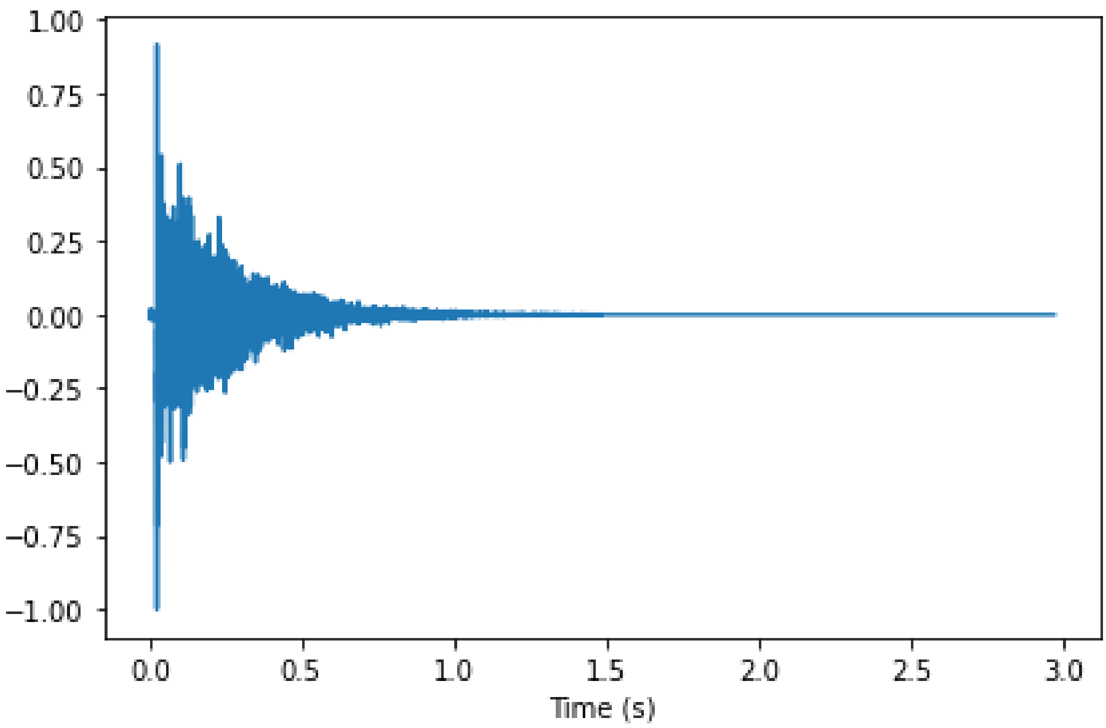

Рисунок 10.1. Сигнал

Вычислим спектр:

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

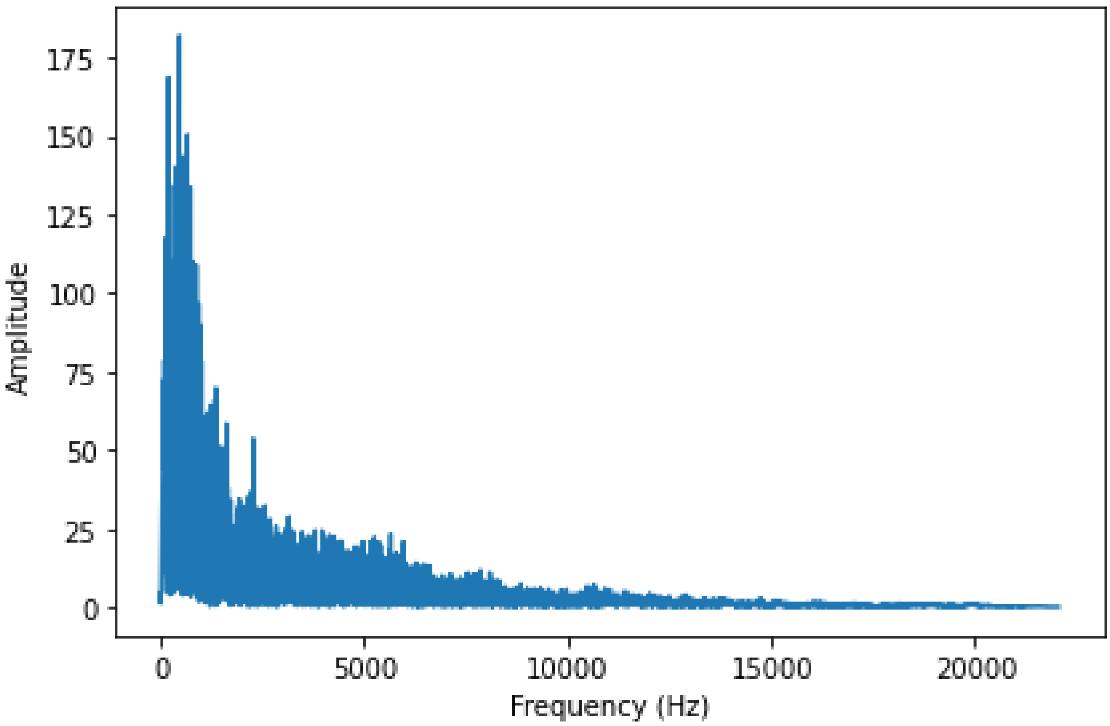

Рисунок 10.2. Спектр сигнала

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__jcveliz__violin-origional.wav

violin = read_wave('92002__jcveliz__violin-origional.wav')

start = 0.11
violin = violin.segment(start=start)
violin.shift(-start)

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

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

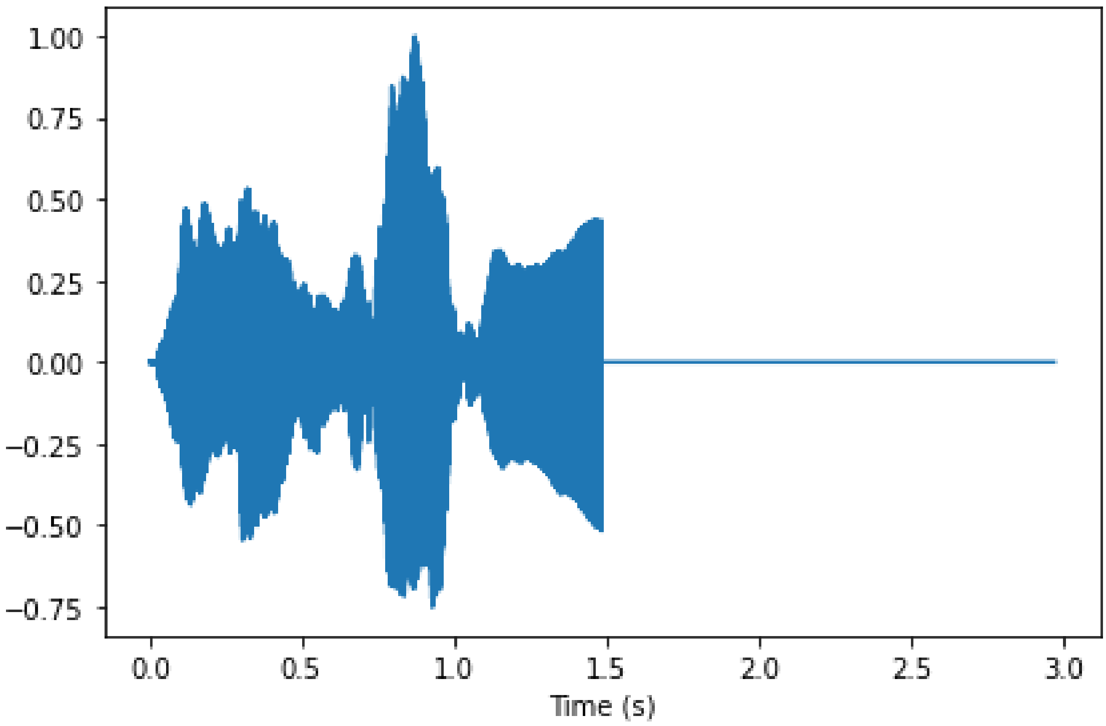

Рисунок 10.3. График сигнала

Вычислим спектр:

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

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

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

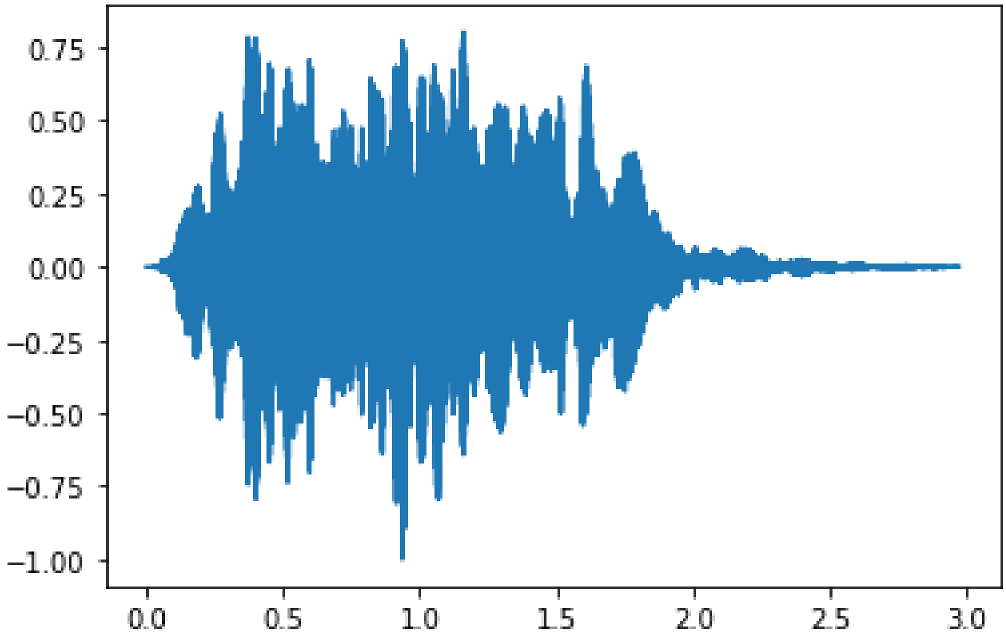

Рисунок 10.4. График сигнала

Исходя из результатов видно, что проблему с лишней нотой удалось решить.

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.

Воспользуемся характеристикой из учебного пособия, так как при взятии звуков с импульсной характеристикой с ресурса Open Air получается сильный шум.

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

response = read_wave('stalbans_a_mono.wav')

start = 0
duration = 3
response = response.segment(duration=duration)
response.shift(-start)

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

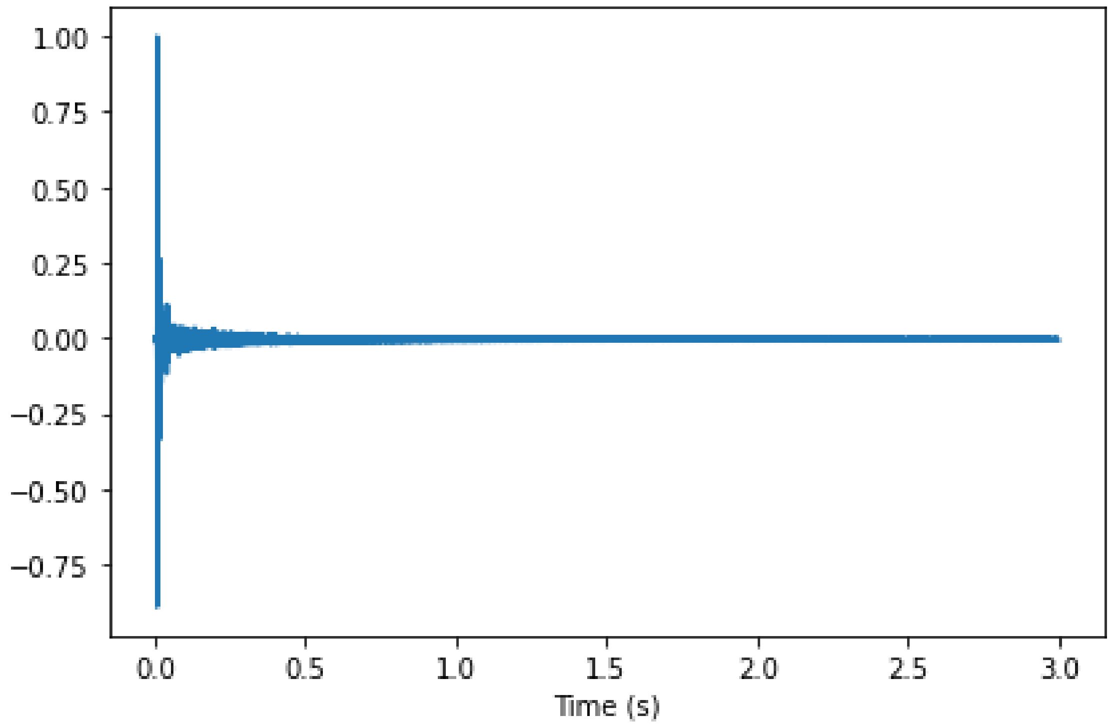

Рисунок 10.5. График загруженного сигнала

ДПФ импульсной характеристики:

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

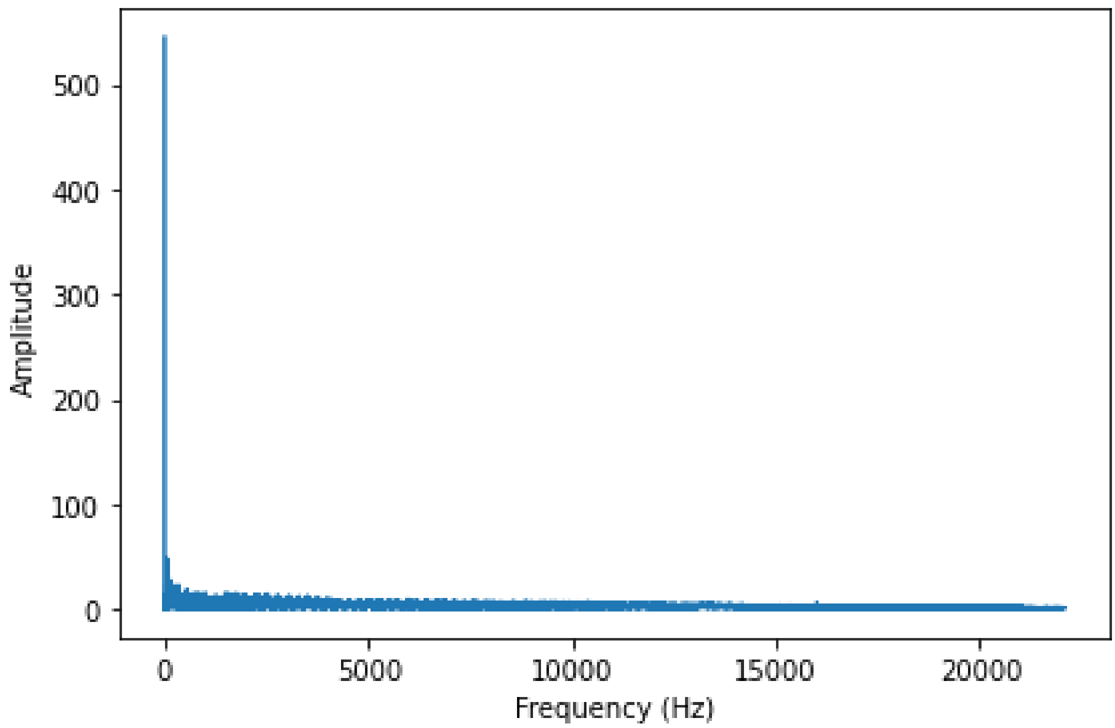

Рисунок 10.6. ДПФ импульсной характеристики

В лагорифмическом масштабе:

```
transfer.plot()
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude',
         xscale='log', yscale='log')
```

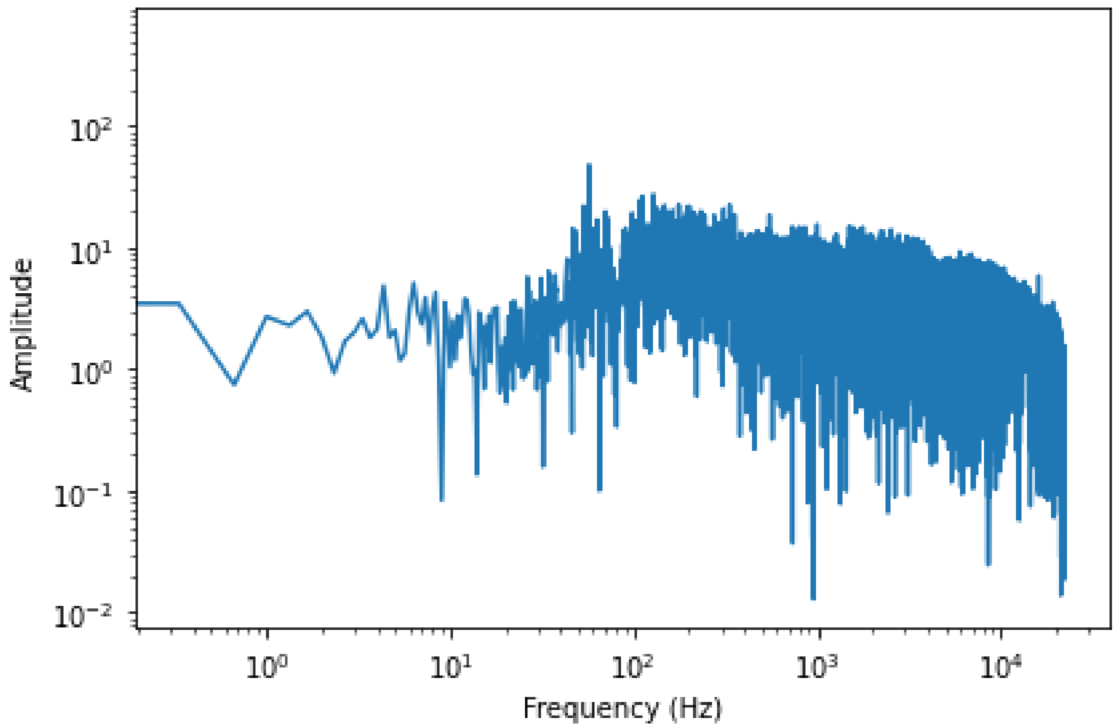

Рисунок 10.7. ДПФ импульсной характеристики в лагорифмическом масштабе

Возьмем запись для преобразования

```
if not os.path.exists('440931__xhale303__piano-loop-1.wav'):
    !wget https://github.com/hotnotHD/Telecom/raw/main/440931__xhale303__piano-
        loop-1.wav

wave = read_wave('440931__xhale303__piano-loop-1.wav')

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

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

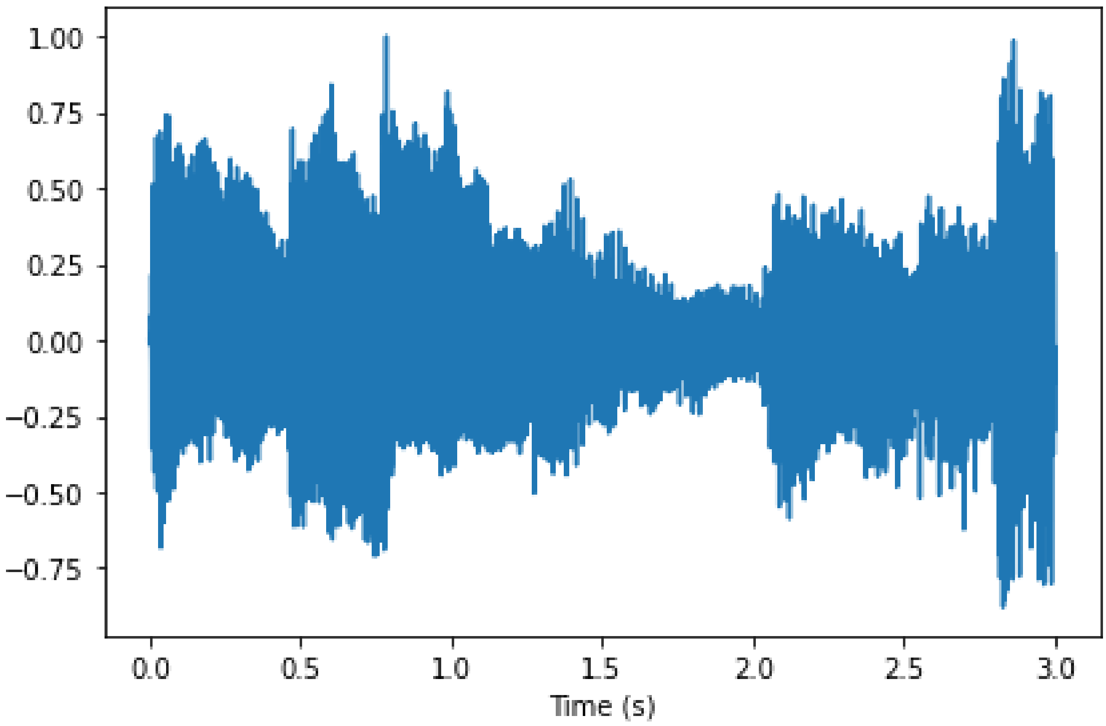

Рисунок 10.8. Сигнал звука пианино

```
wave.framerate

44100
```

Теперь вычислим ДПФ преобразование записи и урежем запись до той же длины, что и импульсная характеристика

```
spectrum = wave.make_spectrum()

len(spectrum.hs), len(transfer.hs)
(66151, 66151)

spectrum.fs
array([0.00000000e+00, 3.33333333e-01, 6.66666667e-01, ...,
       2.20493333e+04, 2.20496667e+04, 2.20500000e+04])

transfer.fs
array([0.00000000e+00, 3.33333333e-01, 6.66666667e-01, ...,
       2.20493333e+04, 2.20496667e+04, 2.20500000e+04])
```

С использованием свертки:

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

Через умножение:

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

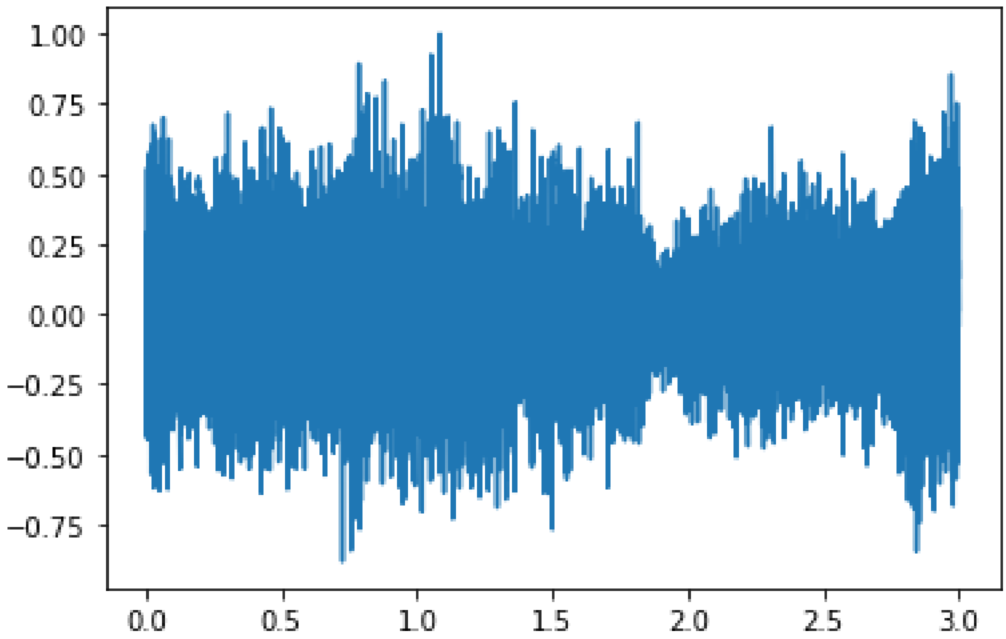

Рисунок 10.9. Полученный ДПФ

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 1

Вернемся к примеру "Соло на барабане", применим фильтр НЧ до выборки, а затем, опять с помощью фильтра НЧ, удалим спектральные копии, вызванные выборкой. Результат должен быть идентицент отфильтрованному сигналу.

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

from thinkdsp import read_wave

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

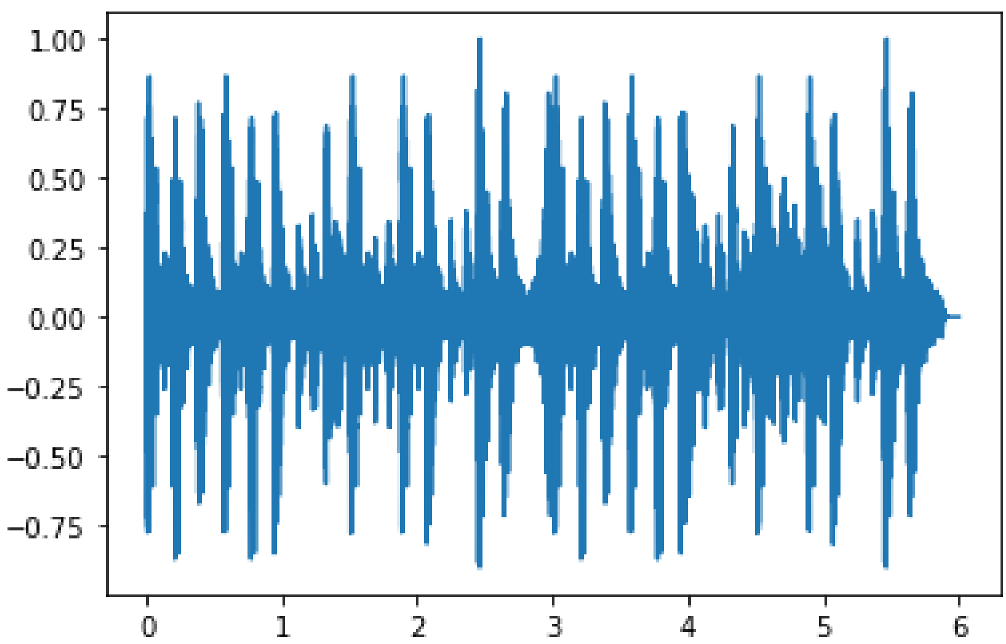

Рисунок 11.1. График сигнала игры на барабанах

```
wave.framerate

44100
```

Видно, что сигнал дискретизируется с частотой 44100 hz

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

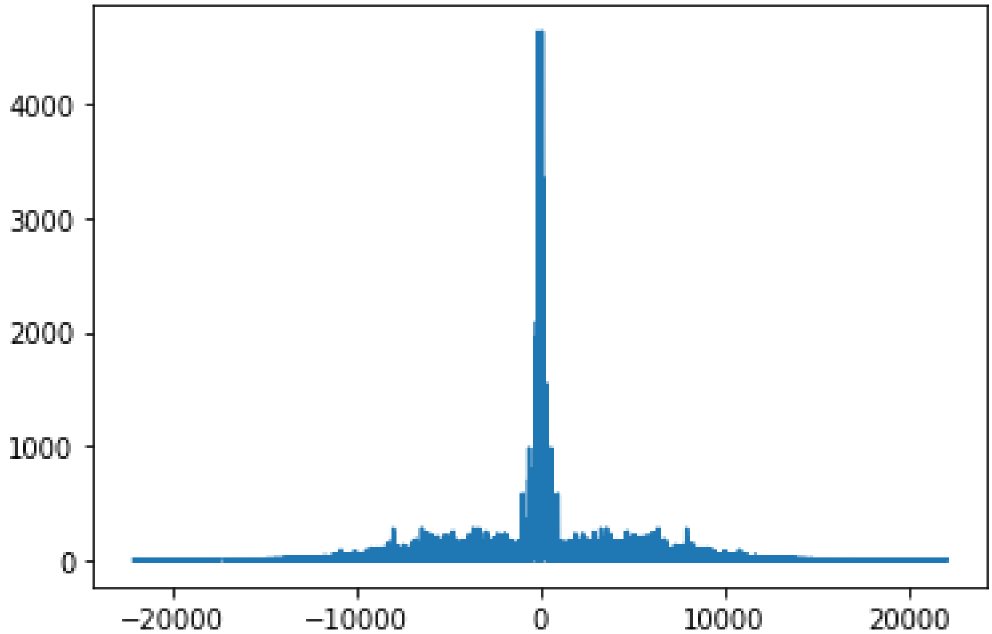

Рисунок 11.2. Спектр сигнала

Применим фильтр НЧ

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

Применим фильтр сглаживания для удаления частот выше новой частоты сворачивания, которая равна framerate / 2

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

Рисунок 11.3. Отфильтрованный сигнал

Функция, которая имитирует процесс выборки:

```
from thinkdsp import Wave

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

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

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

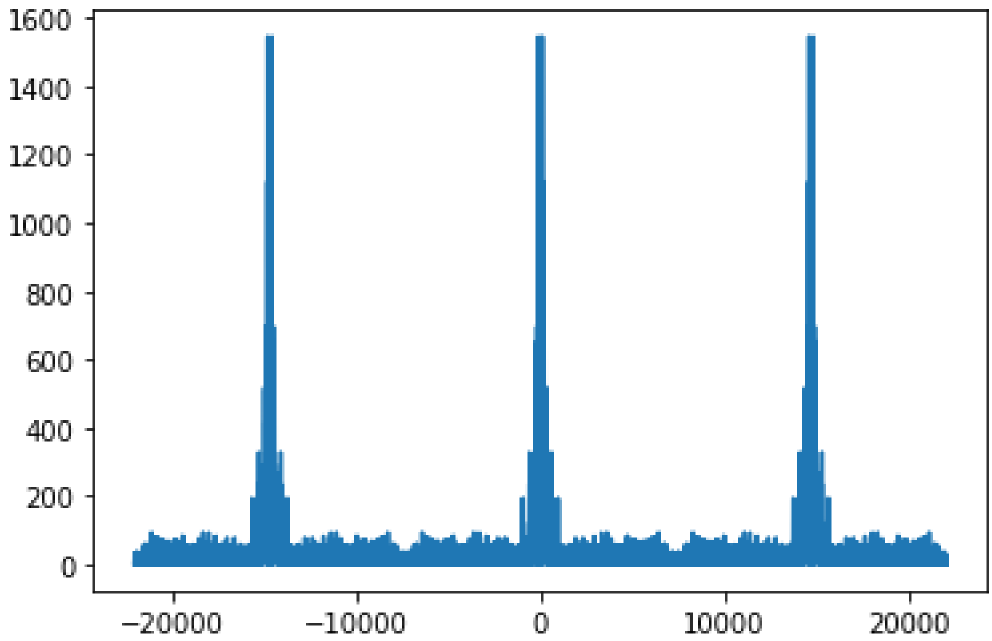

Рисунок 11.4. Получившийся спектр

Видно, что появляются копии спектра
Ещё раз применив фильтр НЧ избавились от них

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

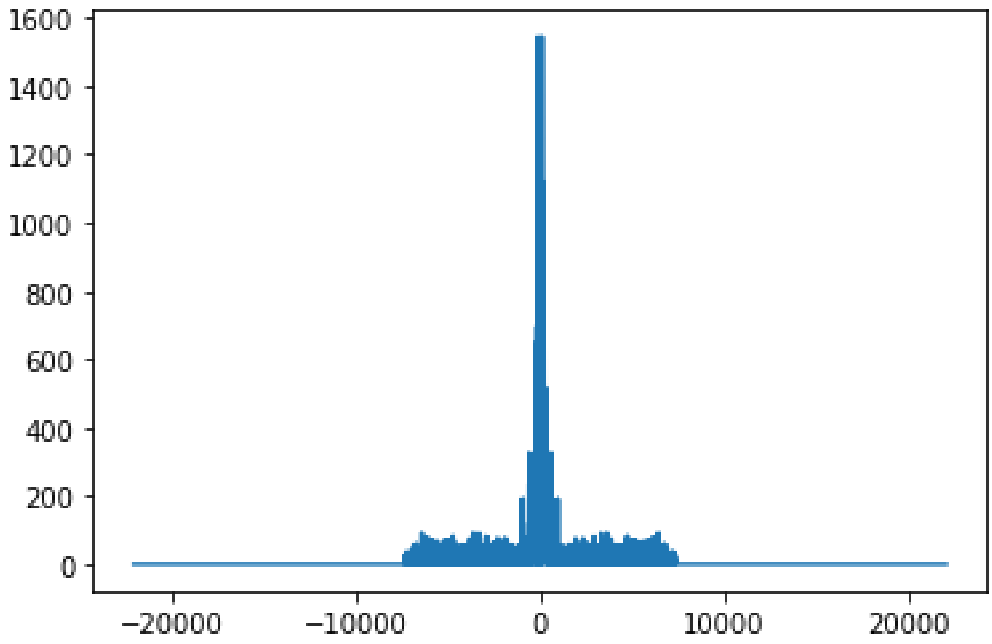

Рисунок 11.5. Результат избавления от копий

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

```

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

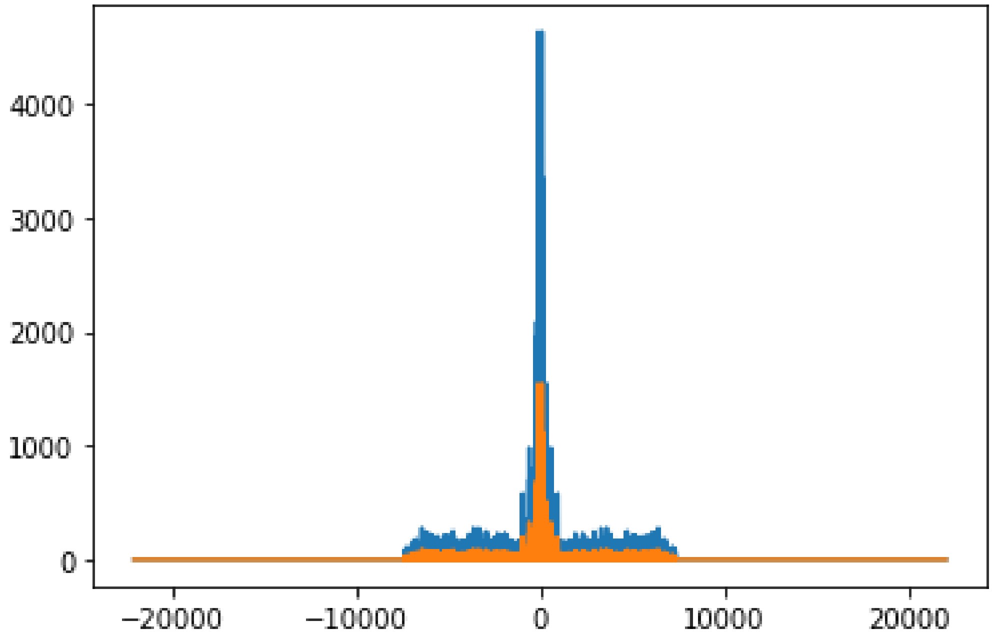

Рисунок 11.6. Сравнение спектров

Увеличим амплитуду в 3 раза

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

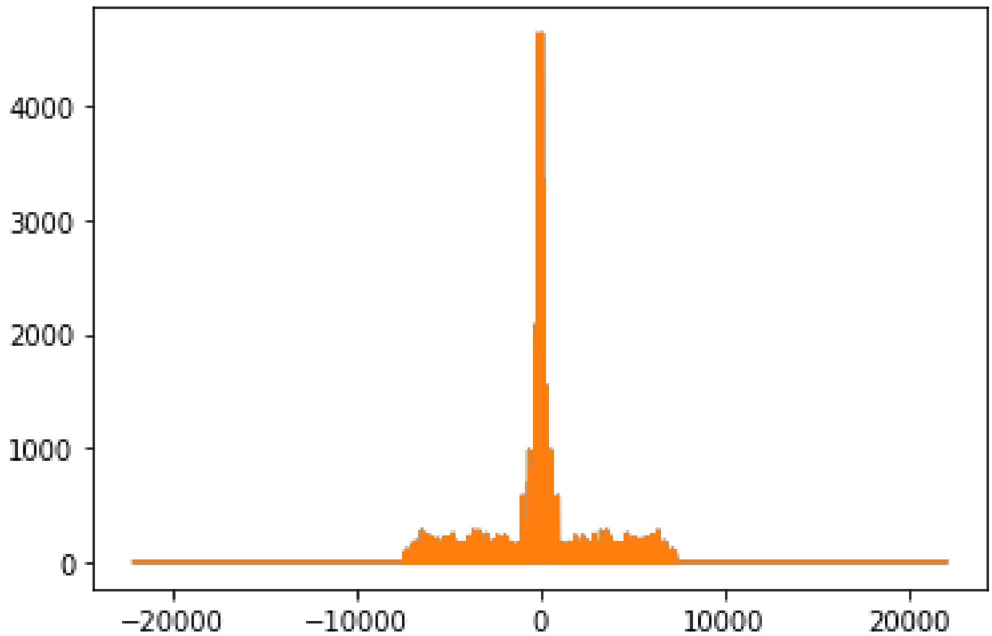

Рисунок 11.7. Сравнение спектров

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

Разница едва заметна

11.2. Вывод

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

12. FSK

12.1. Теоритическая основа

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.

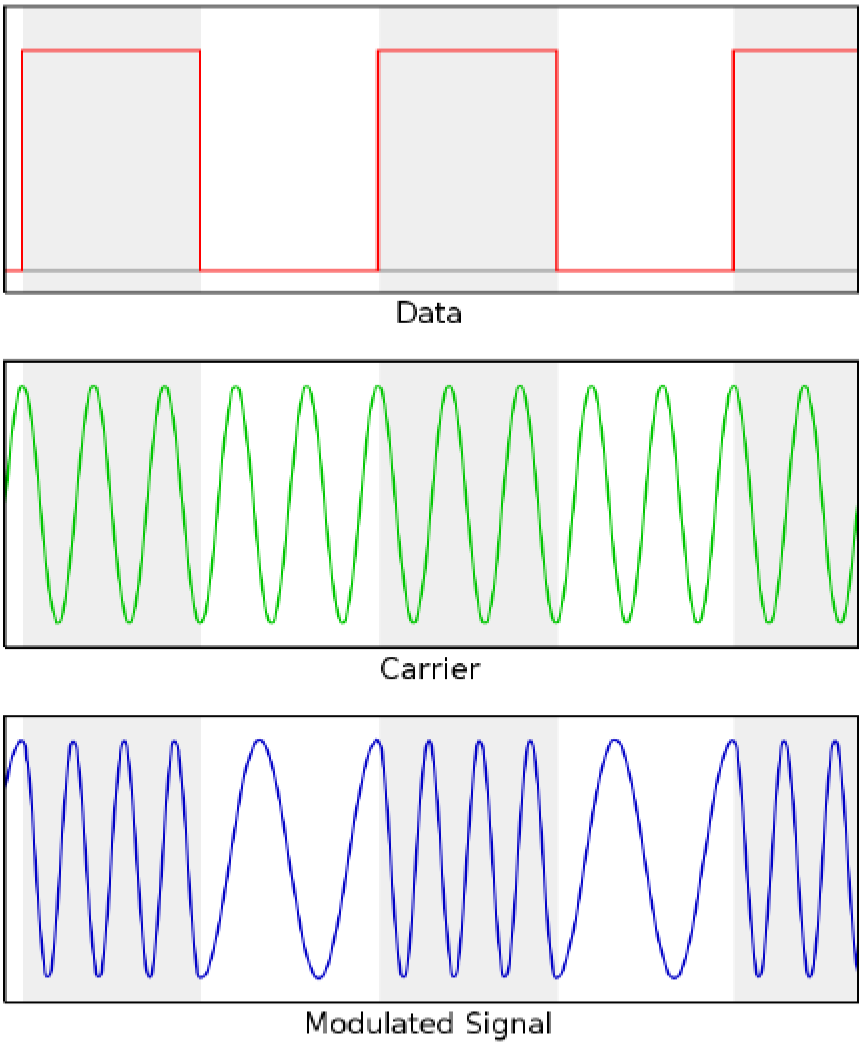

Рисунок 12.1. Пример FSK с двоичными данными

12.2. Схема в GNU Radio

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

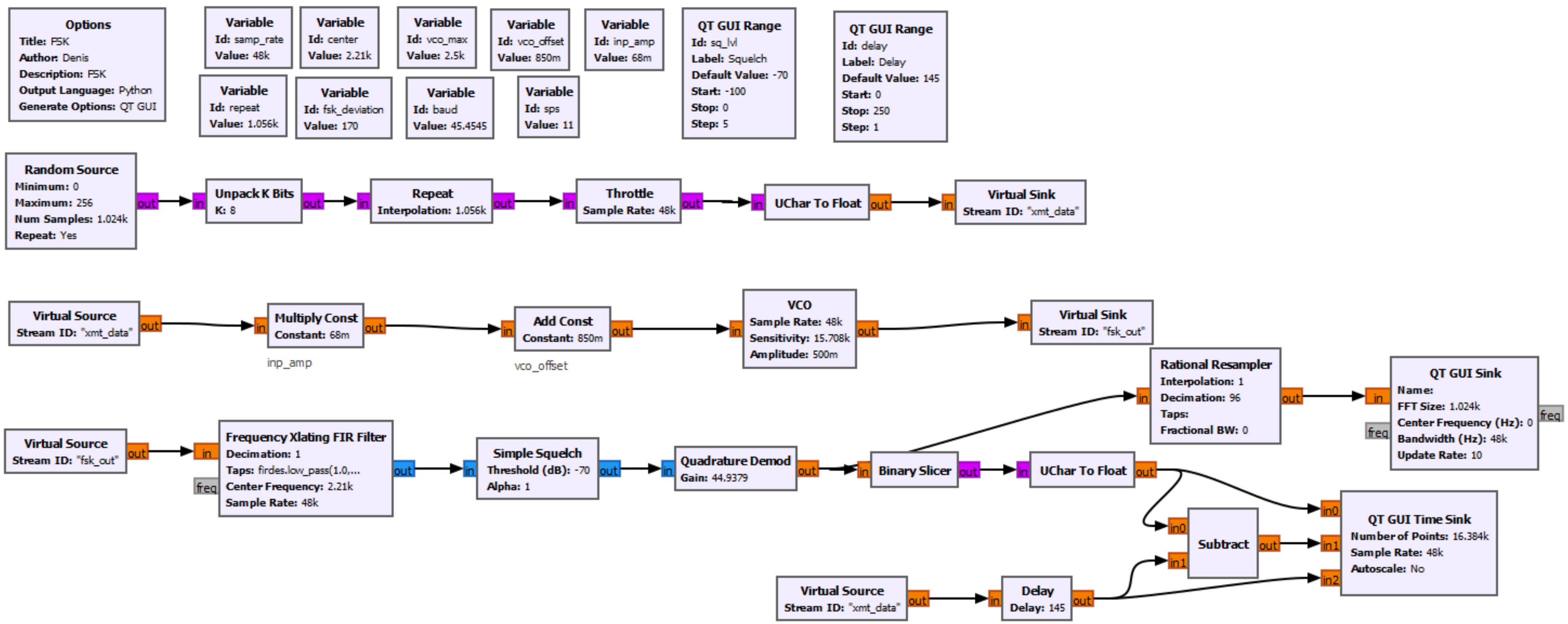

Рисунок 12.2. Схема FSK

Описание используемых блоков:

- Variable - блок адресующий в уникальной переменной. При помощи ID можно передавать информацию через другие блоки.
- QT GUI Range - графический интерфейс для изменения заднной переменной.
- Random Source - генератор случайных чисел.
- Unpack K bits - преобразуем байт с k релевантными битами в k выходных байтов по одному биту в каждом.
- Repeat - количество повторенний ввода, деуйствующее как коэффицент интерполяции.
- Throttle - дросселировать поток таким образом, чтобы средняя скорость не превышала удельную скорость.
- Uchar To Float - конвертация байта в Float.
- Virtual Sink - сохраняет поток в вектор, что полезно, если нам нунжо иметь данные за эксперимент.
- Virtual Source - источник данных, который передаёт элементы на основе входного вектора.
- Multiply Const - умножает входной поток на скаляр или вектор.
- Add Const - прибавляет к потоку скаляр или вектор.
- VCO - генератор, управляемый напрямжением. 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 инфомрации в графическом интерфейсе.

Алгоритм работы: Источник генерирует случайные байты (от 0 до 255). Далее этот байт распоковывается в каждый бит становится байтом со значащим младшим разрядом. Для ограничения потока использует 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.

12.3. Тестирование

Запустим моделирование.

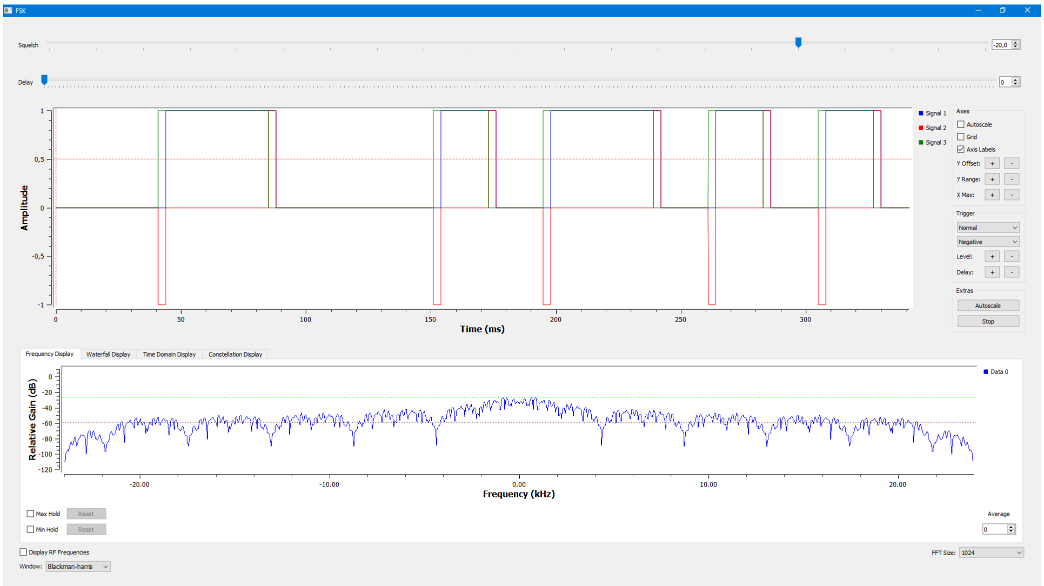

Рисунок 12.3. Тестирование без задержки с наложением шума

Исходя из Рисунка 12.3 видно, что у нас присутствует 3 сигнала. Синий сигнал - данные полученные приёмником. Зелёный сигнал - данные переданные передатчиком. Красный сигнал - разница между двумя предыдущими. Если всё передаётся верно, то красный сигнал должен быть равен 0. Исходя из результатов видно, что переднная и полученная информация разная. Дело в том, что всё блоки передатчика и приёмника не работают с бесконечно малой задержкой. Поэтому надо ввести задержку между приёмом и выдачей данных на диаграмму. Делается это при помощи блока Delay. Установил задержку 145.

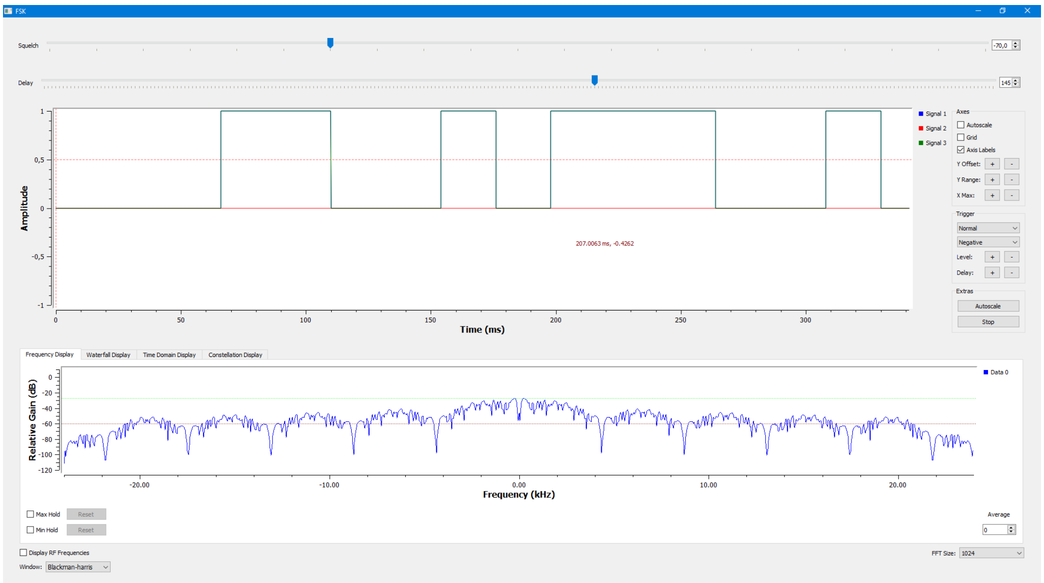

Рисунок 12.4. Тестирование с установленной задержкой

На рисунке видно, что мы подверги сигнал шумам, но из-за фильтра это не помешало нам получить информацию.

12.4. Вывод

В данной работе был изучен новый способ модуляции. Как говорилось ранее, он довольно шумоустойчив из-за того, что информация передаётся при помощи изменений частоты, а не амплитуды. При помощи среды Radio GNU была создана модель и проверена на корректность.