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:
С. В. Пименов
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 фильтрации тех или иных гармоник. Затем преобразуйте спектры обратно в сигнал и прослушайте его. Как звук соотносится с изменениями, сделанными в спектре?

Загружаем звуки игры на пианино, взятые на сайте freesound.org и загруженные на мой репозиторий. Добавляем звуки в класс Wave. Поделим исходную запись на фрагменты и вырежем одну часть.

```
if not os.path.exists('1647_piano.wav'):
    !wget https://github.com/pimenov01/telecom/raw/
    main/files/1647_piano.wav
wave = read_wave('1647_piano.wav')
wave.make_audio()
wave = wave.segment(18.3,0.5)
wave.make_audio()
read_wave('1647_piano.wav').plot().plot()
wave.plot()
```

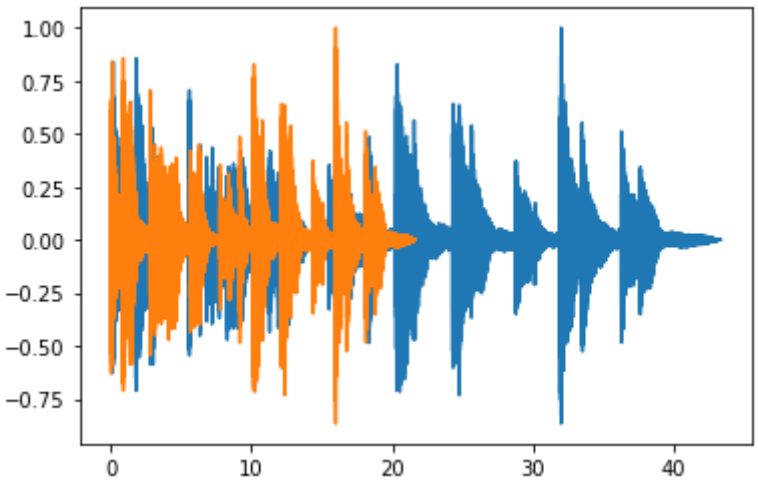

Рисунок 1.1. График фрагмента звука

Далее узнаем спектр звука при помощи метода make_spectrum и построим график для наглядности.

```
spectrum = wave.make_spectrum()
spectrum.plot(high=5000)
```

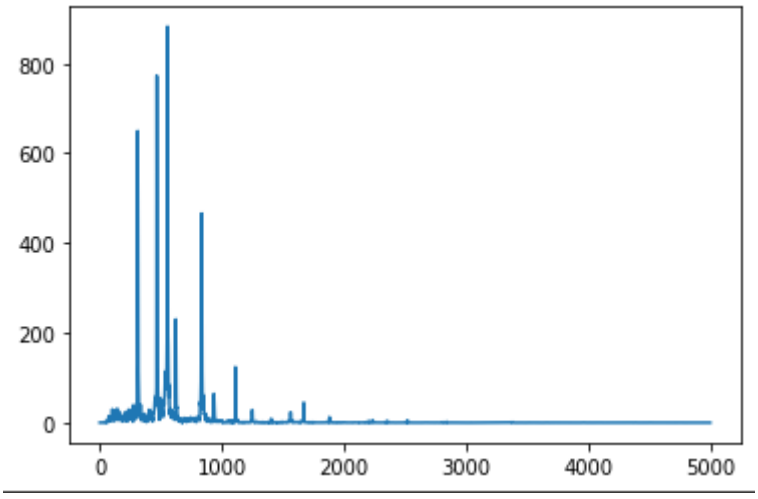

Рисунок 1.2. Спектр звука

Мы видим, что данные расположились до 2Кгц, а график получился не информативный. Обрежем частоты выше 1250Гц, т.к. до них хранится вся важная информация - пики. На остальное смотреть не будем.

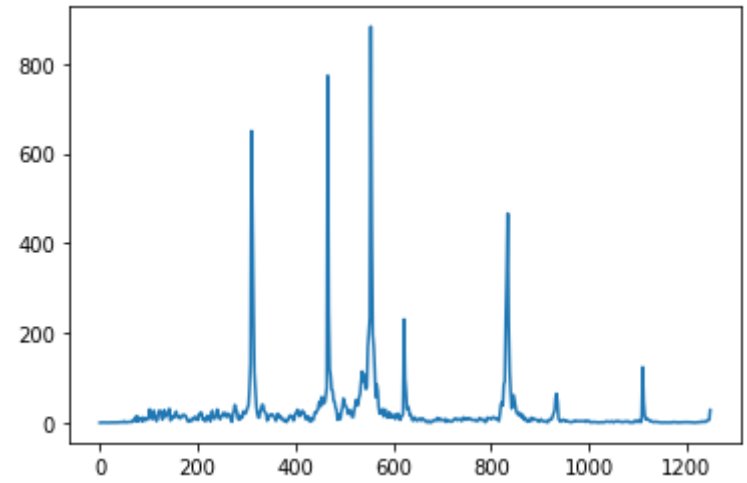

Рисунок 1.3. Спектр звука, частота меньше 1250 ГЦ

Для точного понимания какие ноты сыграны выведем список пиков частот в спектре:

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

```
[(882.3500533686324, 554.0),
 (772.8812508117549, 466.0),
 (649.662314039115, 310.0),
 (466.03538938327 75, 834.0),
 (455.88324114785496, 312.0),
 (435.6400663619303, 556.0),
 (328.0318016013845, 832.0),
 (260.99365239113706, 314.0),
 (251.02745102094198, 468.0),
 (234.35739758178494, 552.0)]
```

Находим соответствие музыкальных нот и частот из пиков:

- 554.36 Гц - C# II
- 466.16 Гц - A# I
- 311.13 Гц - D# I
- 830.60 Гц - G# II

Где # обозначает "диез" или же черную ноту, римские цифра обозначают октаву ноты. У C# II самая большая амплитуда, поэтому 554.36 Гц - доминирующая частота. Общая воспринимаемая высота звука зависит от основной частоты, тут она 311.13 Гц.

Добавим фильтр нижних частот. Все компоненты выше 540 ГЦ будут удалены. (На самом деле можно выбирать на сколько их ослаблять, но я решил на 100%).

```
spectrum2 = wave.make_spectrum()
spectrum2.low_pass(540)
spectrum2.plot(high=1000)
```

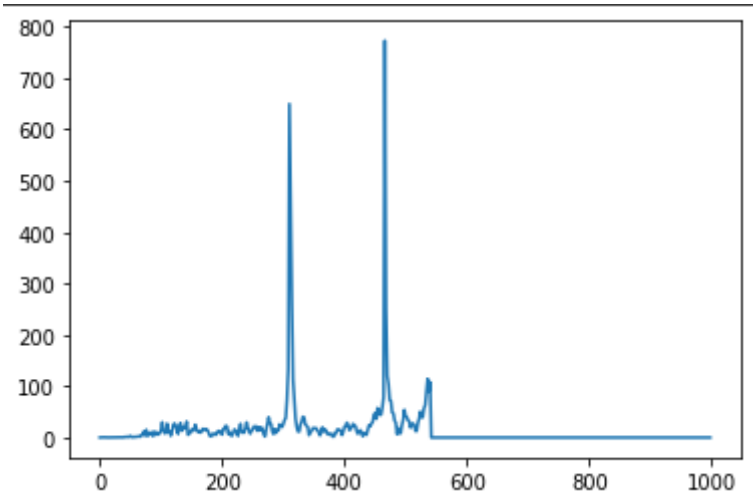

Рисунок 1.4. Спект с фильтром нижних частот

Добавим фильтр верхних частот, и ослабим на половину компоненты до 500 Гц.

```
spectrum2.high_pass(500,0.5)
spectrum2.plot(high=1000)
```

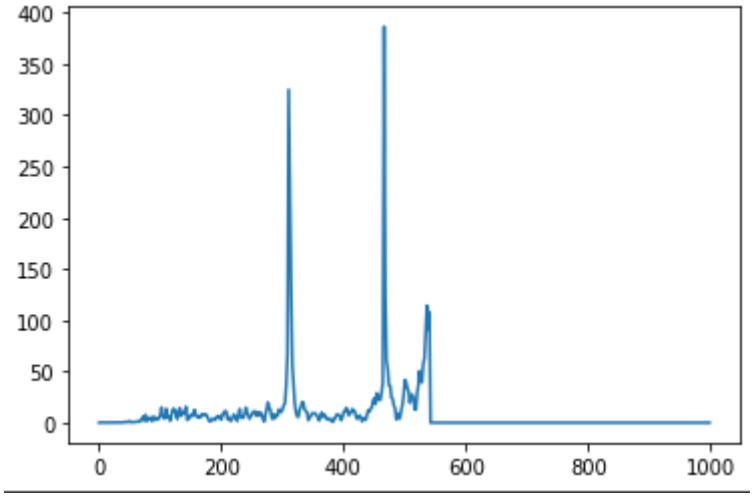

Рисунок 1.5. Спект с фильтром верхних частот

Уберём частоты между Ре и Ля.

```
spectrum2.band_stop(320,450)
spectrum2.plot(high=1000)
```

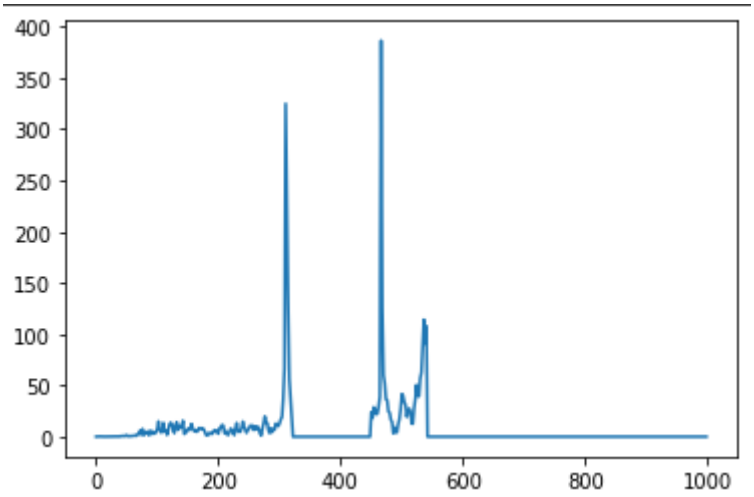

Рисунок 1.6. Получившийся спектр

Звучание заметно изменилось из-за изменения доиминирующей частоты (её ампилитуда была уменьшена в 2 раза), но напоминает изначальный отрезок.

```
wave = read_wave('1647_piano.wav')
wave = wave.segment(18.3,0.5)
spectrum2 = wave.make_spectrum()
spectrum2.high_pass(500)
spectrum2.plot(high=1000)
```

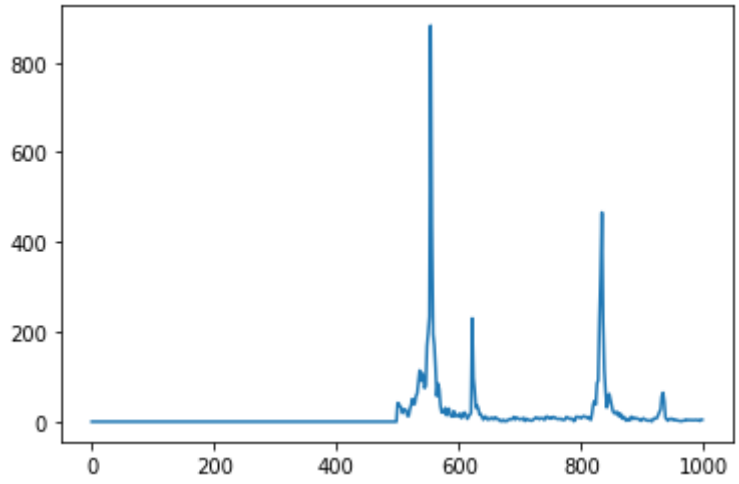

Рисунок 1.7. Отфильтрованные нижние компоненты

Отфильтровав нижние частоты звук стал более высоким. Это логично, ведь ннизкие частоты придают более гулкое звучание, а на получившейся дорожке убраны все частоты, ниже 500.

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?

Берём два сигнала с частотой одной октавы. В итоге мы должны получить классическую синусоиду - волновой сигнал с очень высоким звучанием, однако из него при желании можно сделать бас, если опустить на несколько октав ниже.

```
from thinkdsp import SinSignal, CosSignal
# https://nch-nch.ru/apps/frequency/
cos_sig1 = CosSignal(freq=784.00,amp=1,offset=0)
sin_sig2 = CosSignal(freq=392.00,amp=0.5,offset=0)
mix = cos_sig1 + sin_sig2
wave = mix.make_wave(duration=1)
wave.make_audio()
```

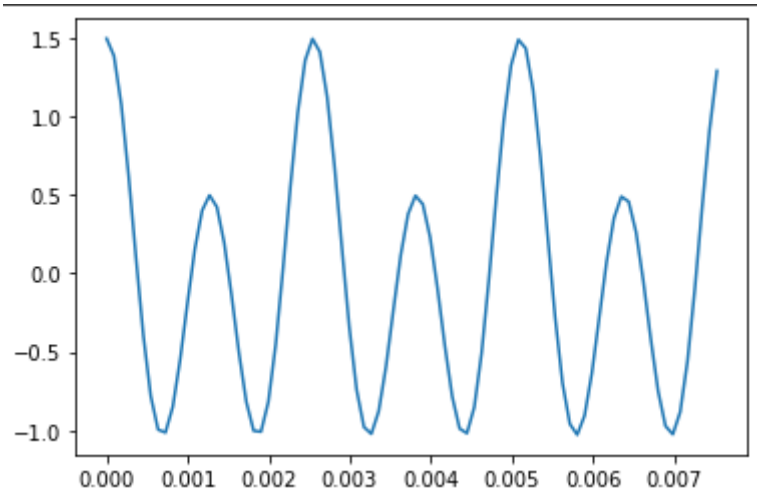

Рисунок 1.8. Суммированные сигналы

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

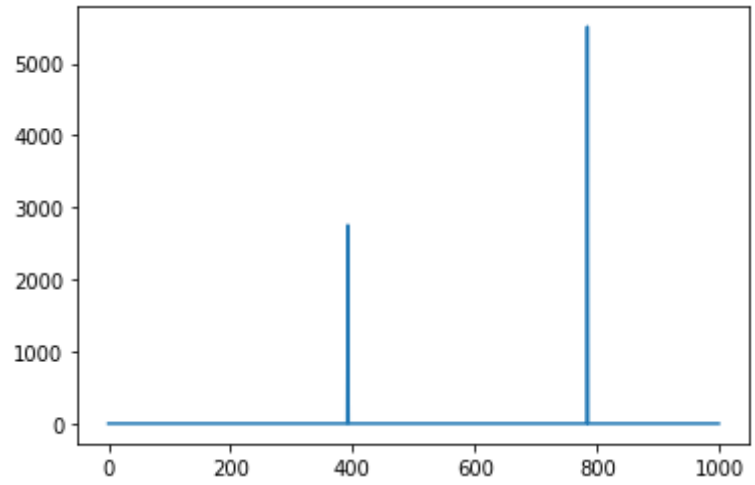

Рисунок 1.9. Спектр сигнала

Добавим частоту из другой октавы. Должно получится ужасно для ушей.

```
cos_signal3 = CosSignal(freq = 500, amp=0.25,offset = 0)
mix = mix + cos_signal3
wave = mix.make_wave(duration=1)
wave.make_audio()
```

На графике за 7мс не видно цикла.

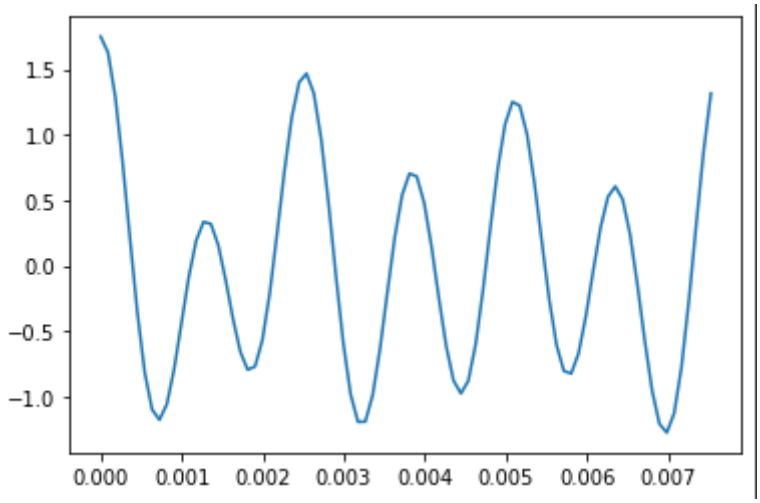

Рисунок 1.10. Получившийся сигнал

Полученный звук не очень приятный.

1.3. Упражнение 3

Напишите функцию strech, берущую wave и коэффицент изменения. Она должна ускорять или замедлять сигнал изменением ts и framerate.
ts - моменты выборки сигнала, framerate - число выборок в единицу времени
если умножим ts на k - интервалы между моментами увеличатся в k раз
если framerate поделим на k - будет меньшее число подвыборок

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

1.4. Вывод

В ходе первой ЛР я познакомился с основными понятиями при работе со звуками и сигналами в частности. Библиотека thinkDSP дает огромный потенциал для взаимодействия со звуками.

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
```

Далее отобразим график пилообразного сигнала.

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

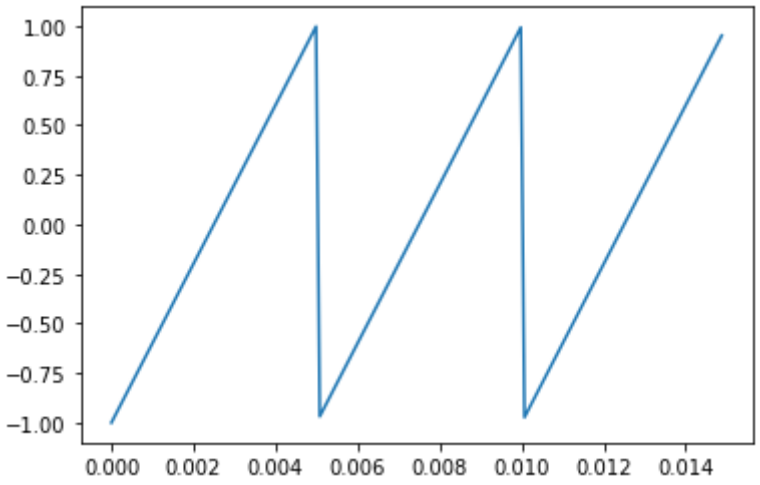

Рисунок 2.1. График пилообразного сигнала

Мы действительно видим пилообразный сигнал. Далее сделаем экземпляр класса Wave для построения спектра сигнала.

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

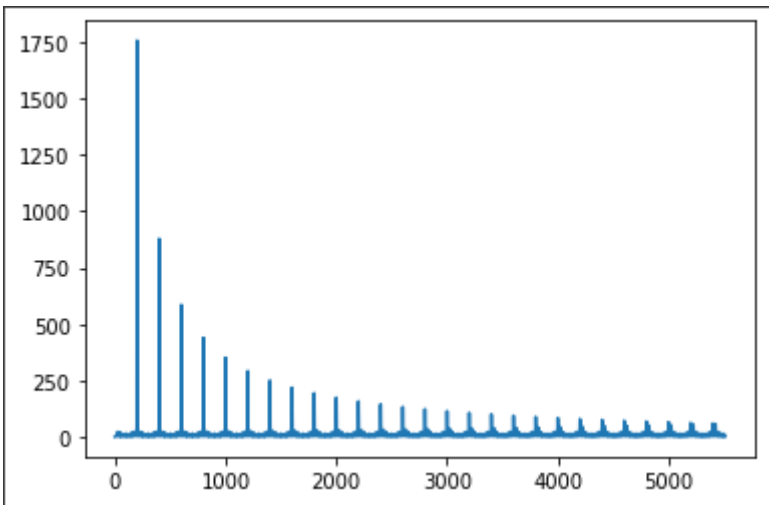

Рисунок 2.2. Спектр пилообразного сигнала

Теперь сравним полученный спектр с треугольными и прямоугольными сигналами.

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

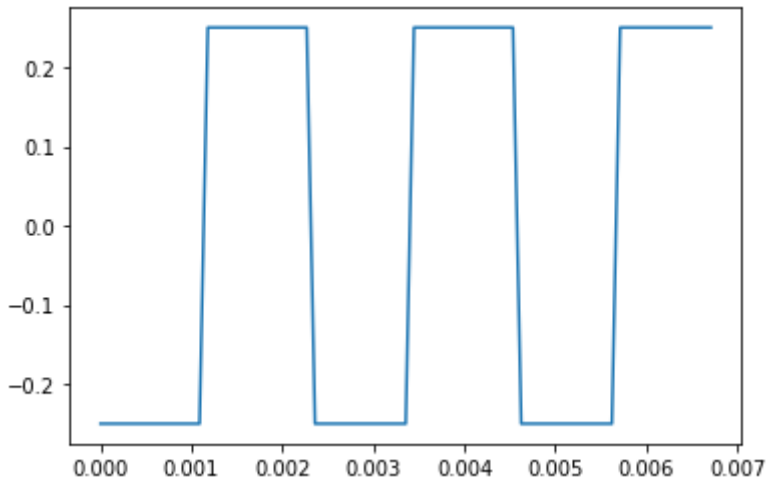

Рисунок 2.3. Спектр треугольного сигнала

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

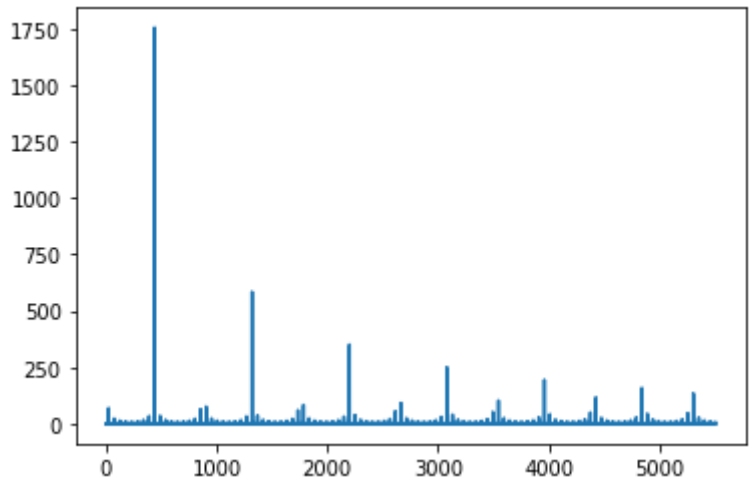

Рисунок 2.4. Спектр прямоугольного сигнала

По сравнению с квадратным сигналом, пилообразный включает в себя четные и нечётные гармоники. Но оба сигнала снижают амплитуду обратно пропорциально частоте. По сравнению с изначальным сигналом, треугольный сигнал падает $1/f^2$, а пилообразный $1/f$.

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()
```

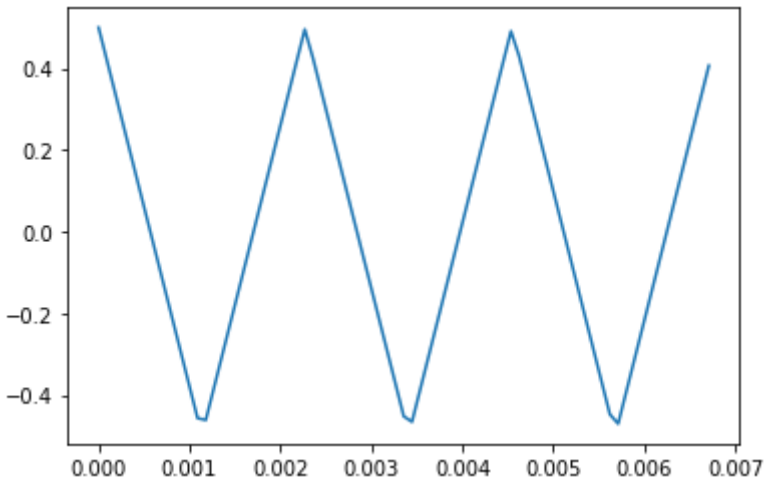

Рисунок 2.5. Спектр сигнала с биениями

Из графика можно увидеть, что из-за фреймрейта, раввного 10.000 сигналы больших частот закольцовываются около 0Гц и 5Кгц - это называется биением. Когда мы слушаем получившийся звук - мы слышим основную частоту на 5КГц.

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()
```

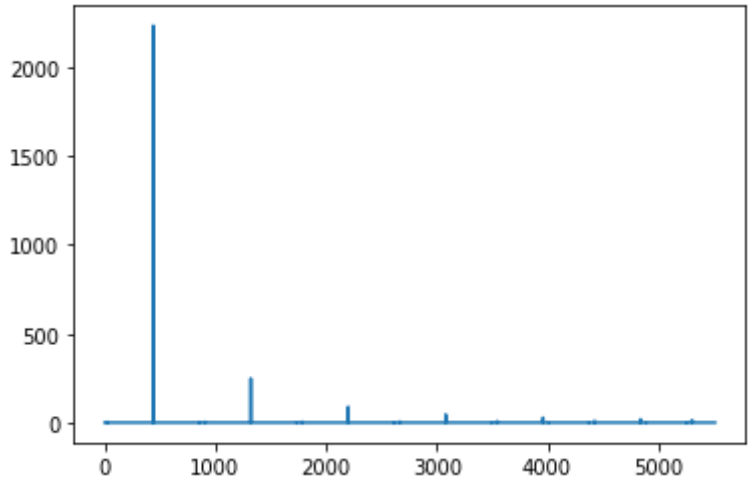

Рисунок 2.6. График сигнала

Проверим что лежит в 0 элементе чисел спекторграммы.

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

```
(1.375077994860476e-14+0j)
```

Видим комплексное число, с 0 мнимой частью. Сам элемент очень близок к нулю. Каждый элмент массива hs объекта Spectrum представялет собой комплексное число и соответствует частотоной компоненте: размах пропорционален амплитуде соответствующей компоненты, а угол - это фаза. Как видно из результатов выполнения кода, первый элемент массива hs - комплексное число, близкое к нулю, мнимая часть равна нулю.
Присвоим первый элемент 100 и посмотрим, что из этого выйдет.

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

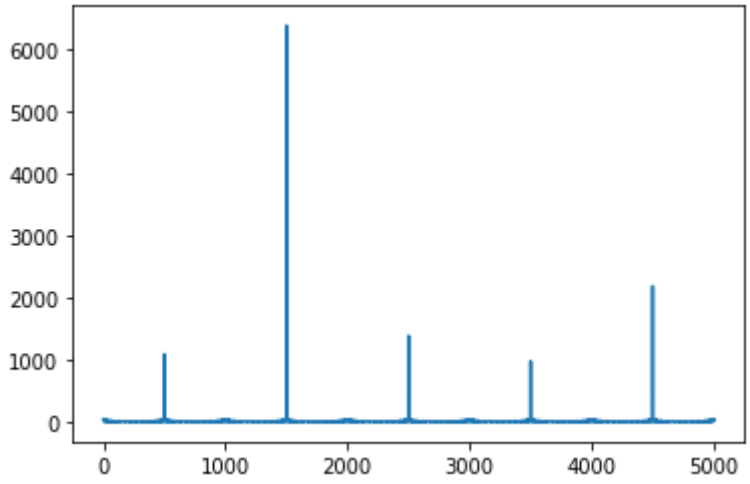

Рисунок 2.7. График сигнала с изменённым нулевым числом спекторграммы

Как видно из полученного графика, сигнал сместился по вертикали. То есть первый элемент массив hs отвечает за смещение сигнали относительно вертикали. Если элемент близок или равен нулю, сигнал не смещенный.

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?

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

```
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()
```

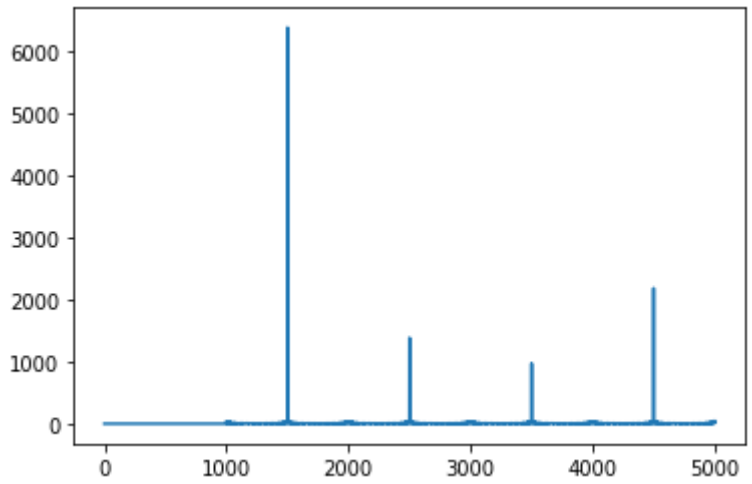

Рисунок 2.8. Спектр треугольного сигнала

Применим написанную функцию.

```
spec_div(spectr)
```

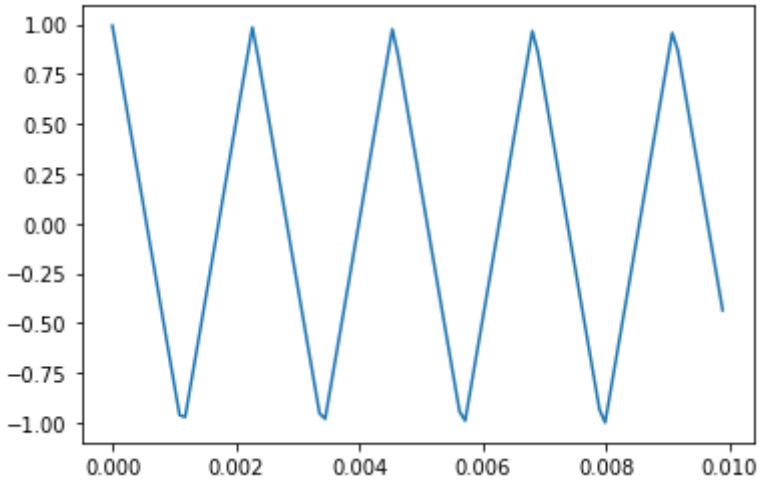

Рисунок 2.9. Спектр изменённого сигнала

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

Видим. что ампилутуда очень поменялась, а частоты стоящие ближе к 0 Гц стали больше, чем следующие, что понятно. Если сравнить полученные графики, можно сделать вывод, что функция действует как фильтр низких частот: частоты ослабляются на некоторую величину. Если сравнить две полученных звука на слух, то второй звучит более чисто, из-за отсутсвия низких часот, похож на звук синусоидального сигнала.

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.
Не зря мы писали предыдущую функцию, поэтому возьмём пилообразный сигнал который имеет и четные и нечётные гармоники, а потом применим нашу функцию.

```
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)')
```

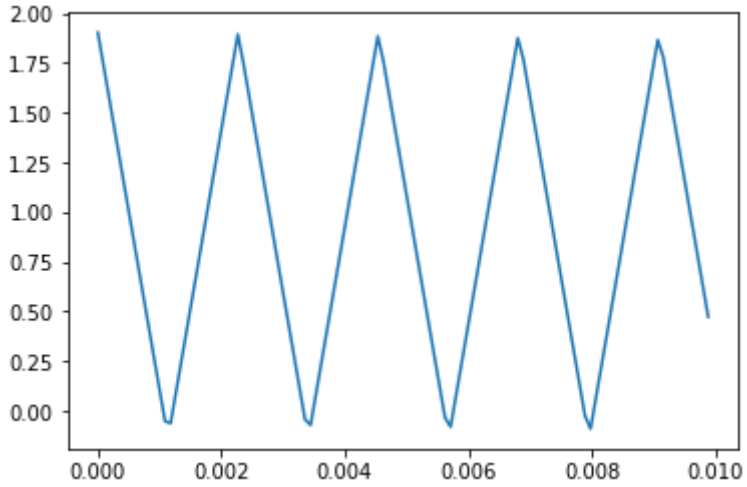

Рисунок 2.10. Спектр пилообразного сигнала

Применим функцию для изменения амплитуды спада

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

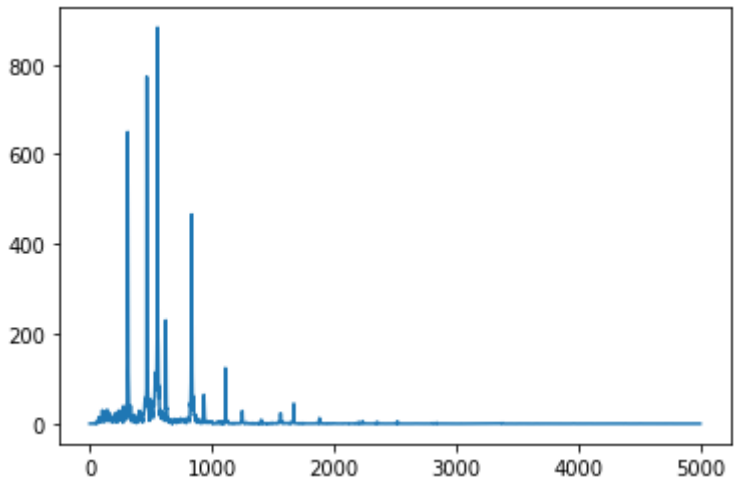

Рисунок 2.11. Спектр пилообразного сигнала после функции

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

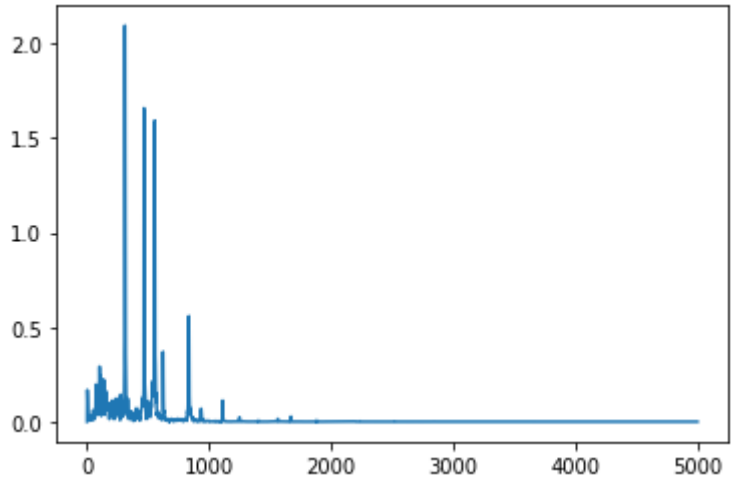

Рисунок 2.12. График сигнала

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

2.6. Вывод

В ходе данной ладораторной работы были произведены различные действия с разными видами сигналов, а также 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)
```

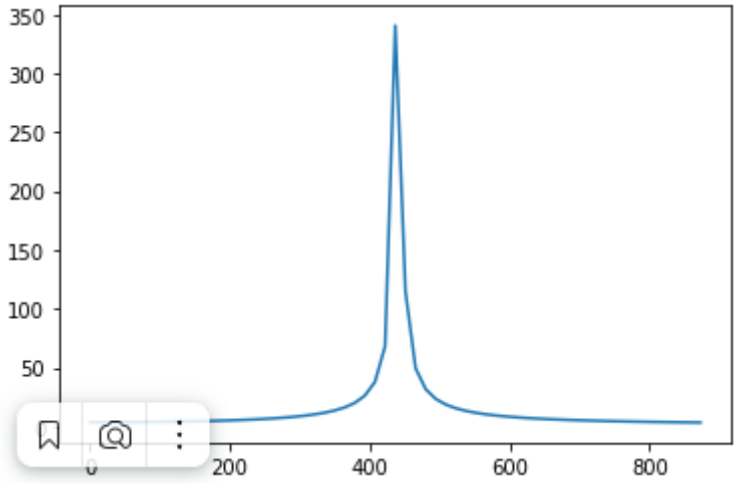

Рисунок 3.1. Рассматриваемый сигнал

Посмотрим как выглядит спектограмма с использованием окна Хэмминга:

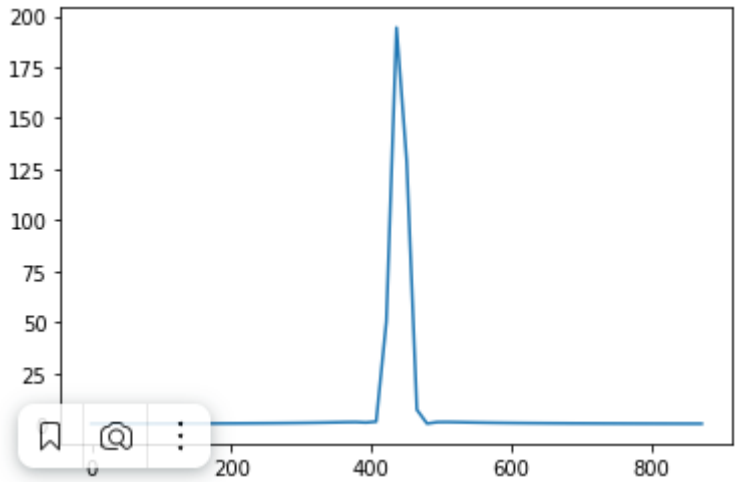

Рисунок 3.2. Сигнал с использованием окна Хэмминга

Посмотрим остальные окна. Окно Бартлетта:

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

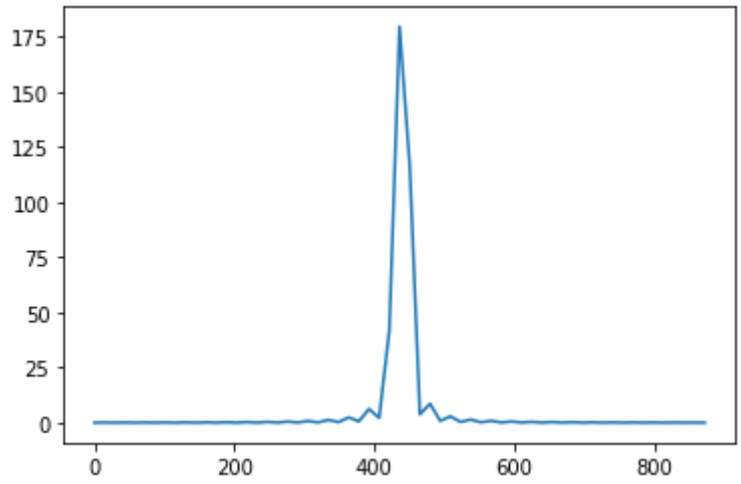

Рисунок 3.3. Сигнал с использованием окна Барлетта

Можно заметить, что низкие амплитуды стали ломанными линиями.

Окно Блэкмена:

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

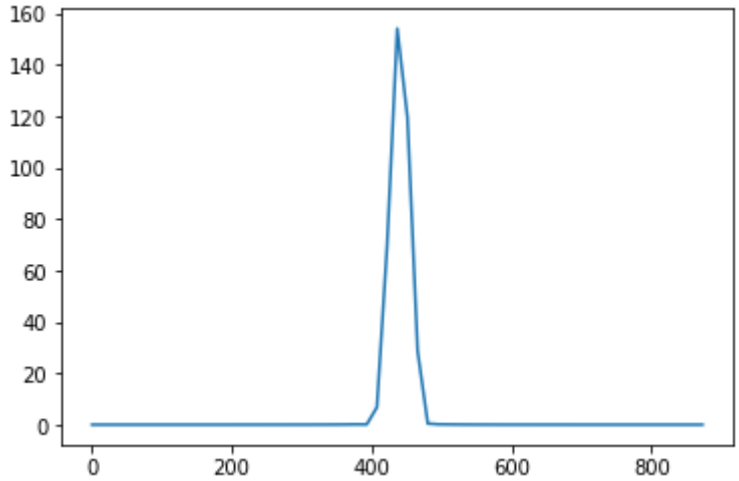

Рисунок 3.4. Сигнал с использованием окна Блэкмена

Видим, что утечка стала линейно переходить к нужной частоте.

Окно Хэннинга:

```
wave = signal.make_wave(duration)
wave.ys *= np.hanning(len(wave.ys))
```

```
spectrum = wave.make_spectrum()
spectrum.plot(high=880)
```

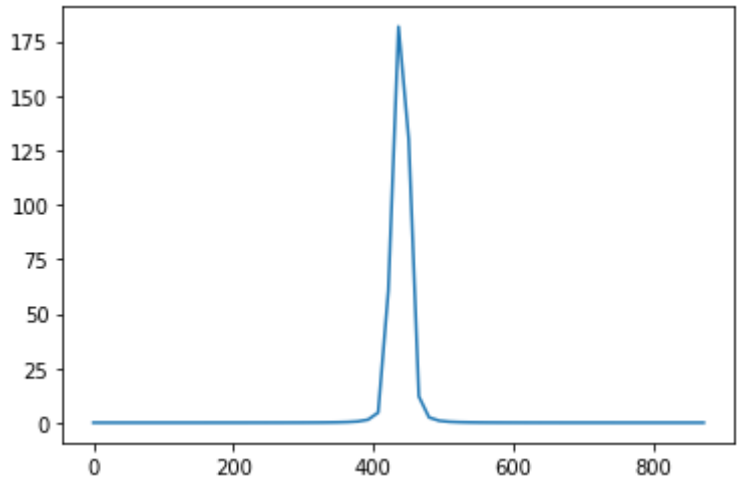

Рисунок 3.5. Сигнал с использованием окна Хэннинга

Все оконные функции хорошо справились со своими задачами.

3.2. Упражнение 2

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

```
class SawtoothChirp(Chirp):

    def evaluate(self, ts):

        freqs = np.linspace(self.start, self.end, len(ts))
        dts = np.diff(ts, prepend=0)
        dphis = 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
```

Для начала создадим звук:

```
signal = SawtoothChirp(start = 440, end = 880)
wave = signal.make_wave(duration=1, framerate=4000)
wave.make_audio()
```

Слышно, как частота постепенно увеливается.

Выполним кратковременное преобразование Фурье и представим результат в виде спектограммы. По оси OY будет частота, по оси OX - время.

```
sp = wave.make_spectrogram(256)
sp.plot()
```

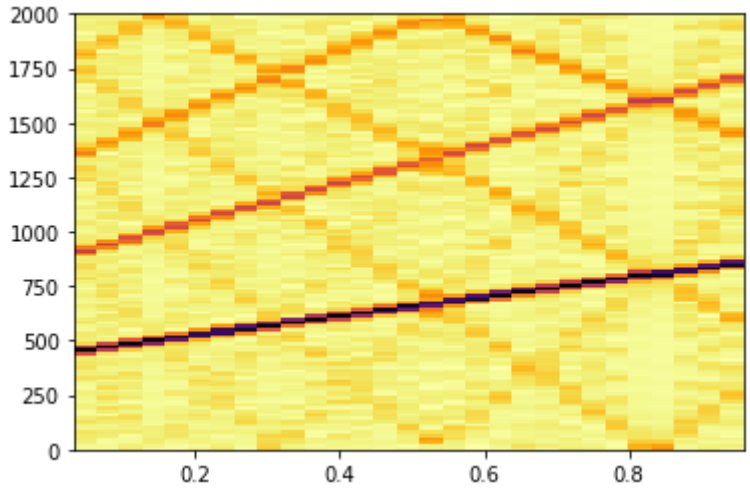

Рисунок 3.6. КПФ сигнала

Черной линией обозначена наша основная частота, остальные частоты "отпрыгивают" от рамок координат и слышны на заднем плане.

3.3. Упражнение 3

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

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

wave.make_spectrum().plot()
```

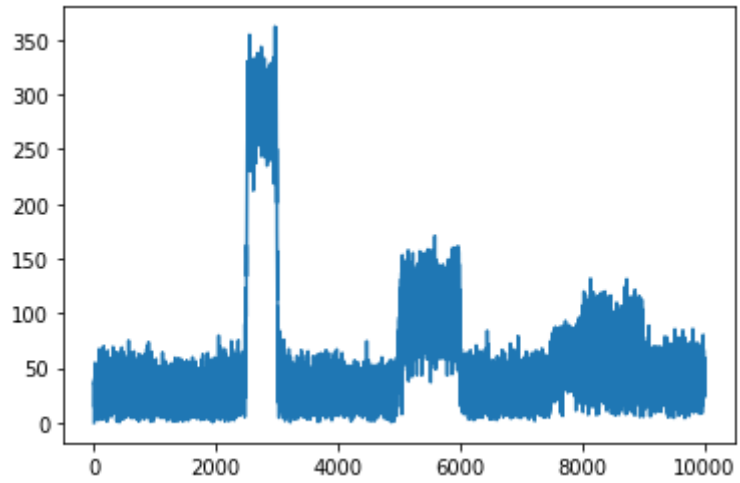

Рисунок 3.7. Спектр сигнала

Видим, что гармоники накладываются друг на друга, и заметно что на частоте 2500 появляется возвышенность и на частоте около 5000 тоже. Это связано с тем, что в данном диапозоне равное изменение частоты занимает равное время.

3.4. Упражнение 4

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

Возьмём звук из репозитория учебника:

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

```
wave = read_wave('72475__rockwehrmann__glissup02.wav')
```

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

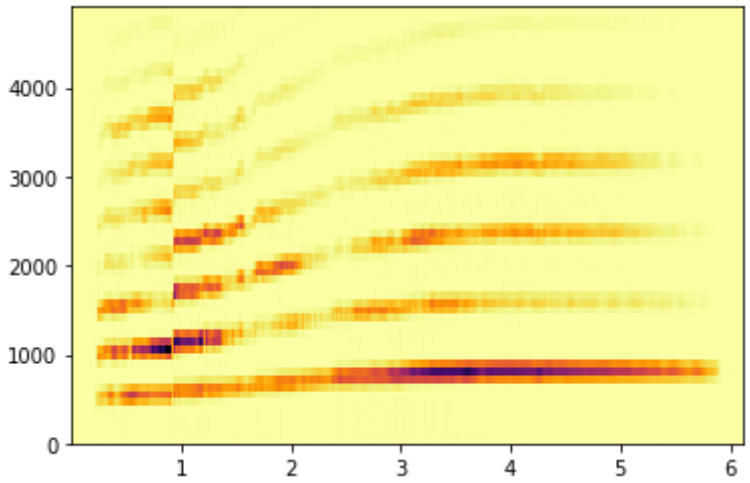

Рисунок 3.8. Спектрограмма сигнала

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. Напишем класс TromboneGliss расширяющий класс Chirp и переопределяющий метод evaluate

```
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
```

Соединим сигналы:

```
signal1 = TromboneGliss(262, 349)
wave1 = signal.make_wave(duration=1)

signal2 = TromboneGliss(349, 262)
wave2 = signal2.make_wave(duration=1)

result = wave1 | wave2
sp = result.make_spectrogram(1024)
sp.plot(high=1000)
```

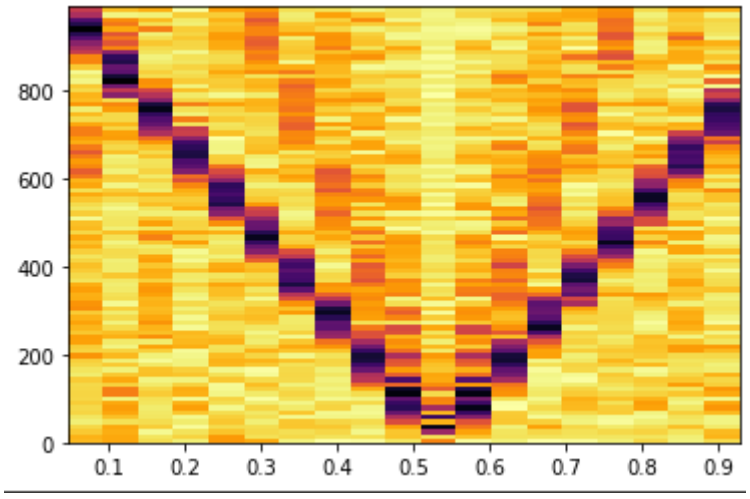

Рисунок 3.9. Спектрограмма сигнала

Отчётливо слышно, как друг за другом идут 2 части.

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 гласных:

```
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_spectrogram(1024).plot(1000)
```

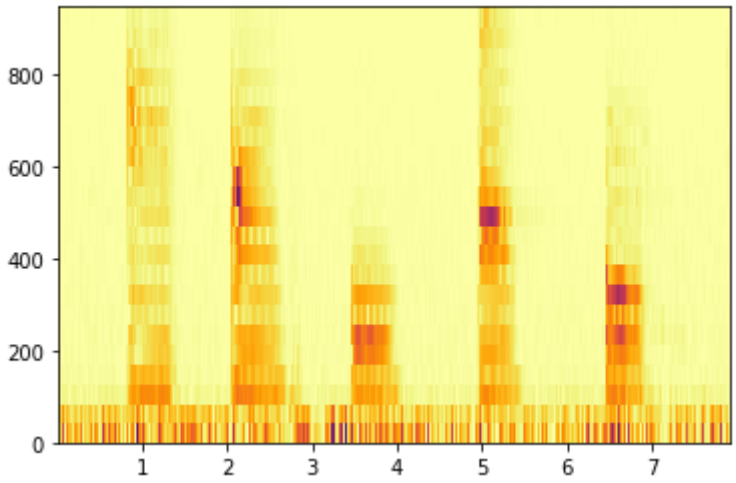

Рисунок 3.10. Спектрограмма гласных звуков

На спектограмме видны пики, они и будут нашими гласными.

3.7. Вывод

В ходе данной ЛР были рассмотрены сигналы, частотные компоненты которых изменяются со временем. Такие сигналы называются апериодическими. Кроме того были рассмотрены спектрограммы - способ визуализации вышеприведенных 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?

Возьмем звук моря и выделим два сегмента.

```
if not os.path.exists('13793__soarer__north-sea.wav'):
    !wget https://drive.google.com/file/d/1OJcFGqEs128g5T8pQmgaERBHX8cyFSsT
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()
```

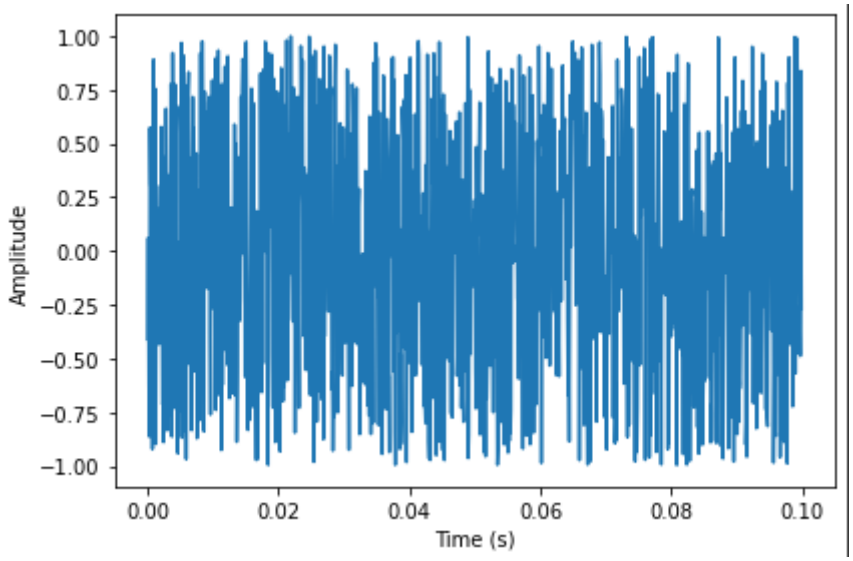

Рисунок 4.1. График сигнала

Определим характеристики шума.

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

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

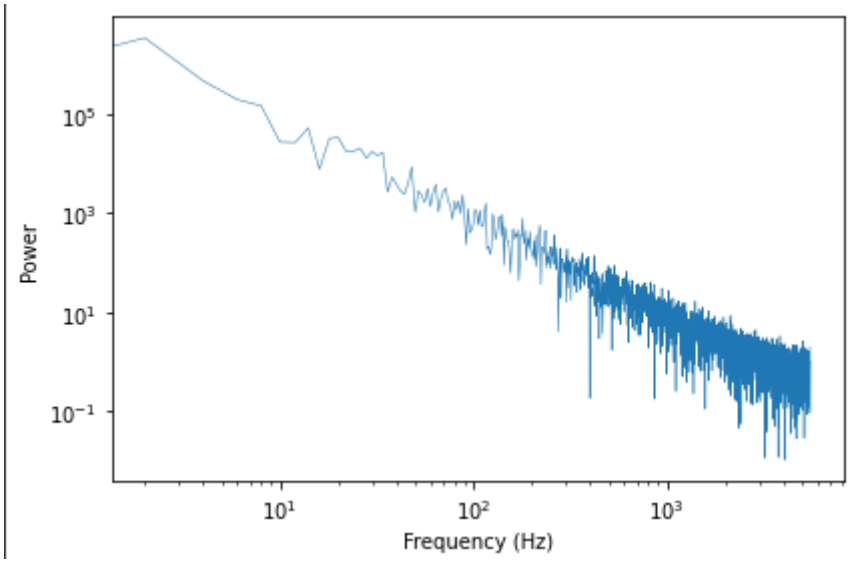

Рисунок 4.2. Спектр в логорифмическом масштабе

График напоминает белый шум. Далее возьмем 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)
```

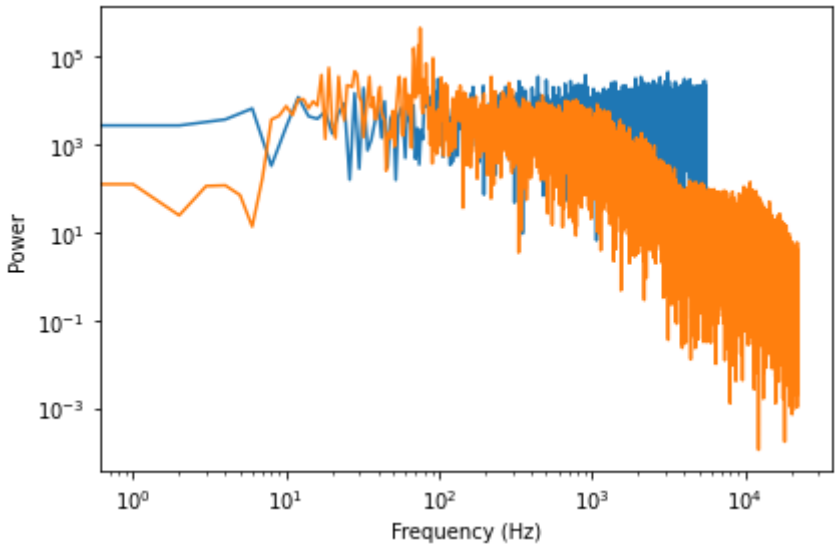

Рисунок 4.3. Сравнение спектров в логорифмическом масштабе

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)
```

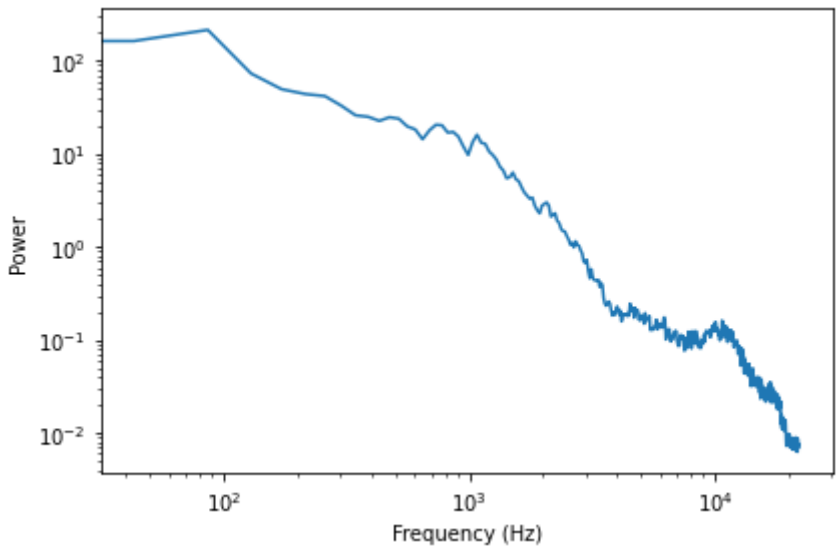

Рисунок 4.4. Результат работы функции

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/pimenov01/telecom/raw/main/files/market-price.
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)
```

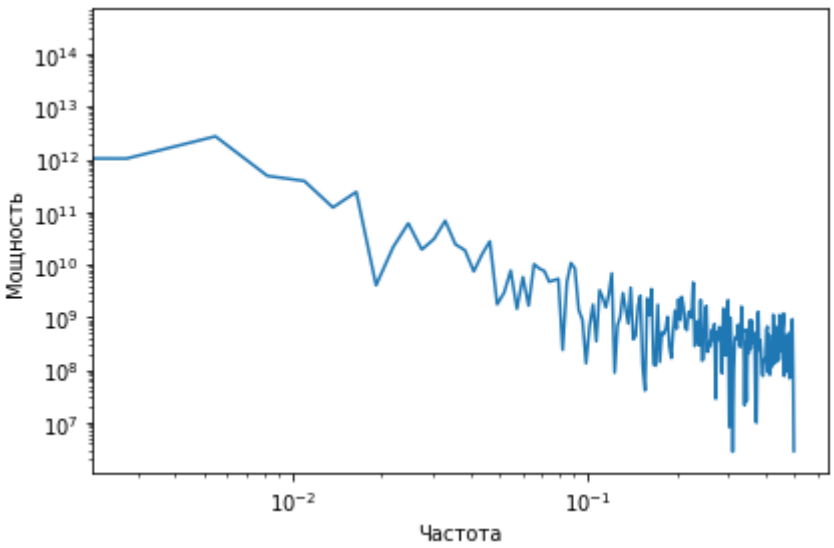

Рисунок 4.5. Спектрограмма цен BitCoin в логорифмическом формате

Больше всего 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
```

Сгенирируем 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()
```

Далее 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)
```

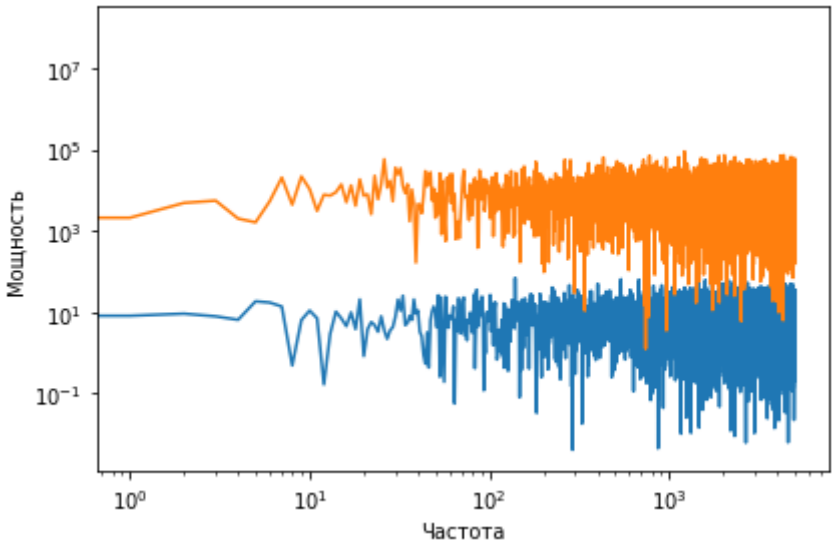

Рисунок 4.6. Сравнение спектров

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 Восса-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()
```

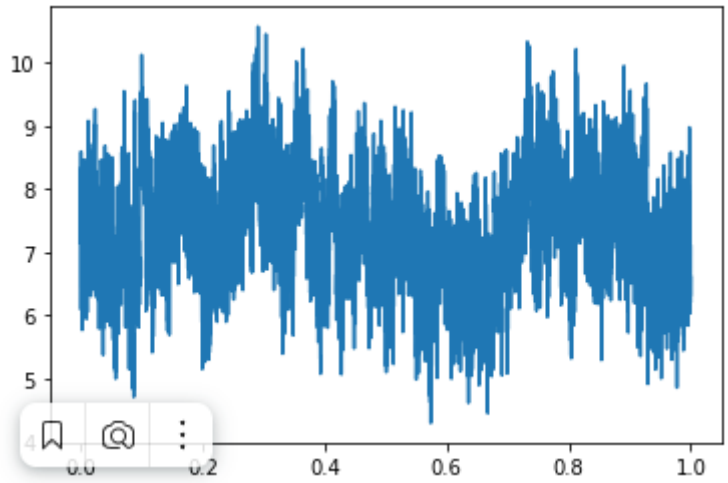

Рисунок 4.7. Сгенерированный сигнал

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

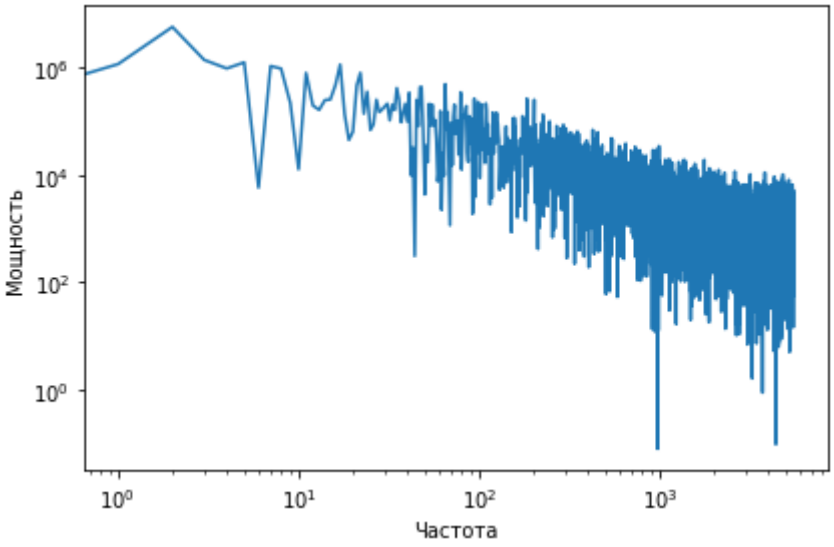

Рисунок 4.8. Спектр сигнала

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

```
-0.9809177359665783
```

Видим, что в итоге был получен сигнал розового шума.

4.6. Вывод

В ходе данной ЛР были изучены различные виды шумов: белый, розовый, красный и т. д. Шум - это сишнал, представленный компонентами с разными частотами, но не

имеющий гармонической структура периодических 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__bcjordan__voicedownbew.wav
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()
```

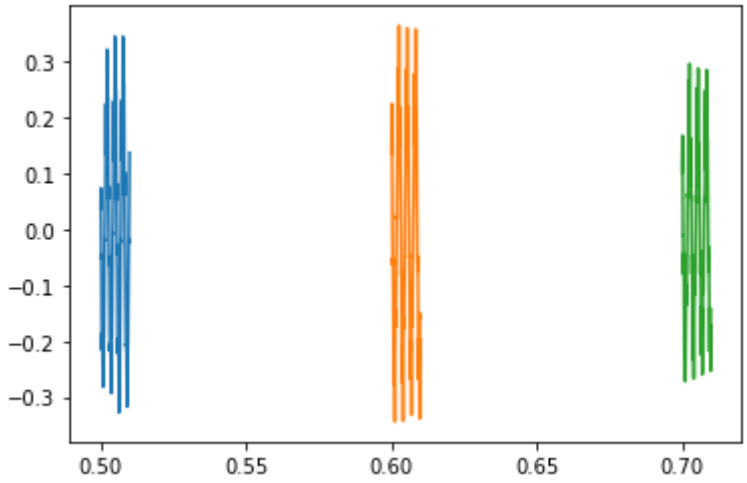

Рисунок 5.1. График сегментов

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])
```

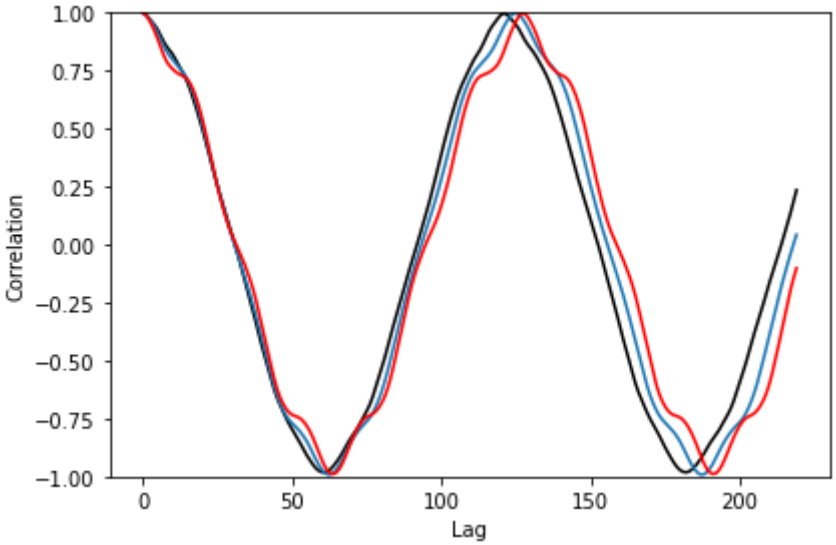

Рисунок 5.2. Автокорреляция сигналов

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.

Для тестирования возьмем звук из прошлого номера.

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

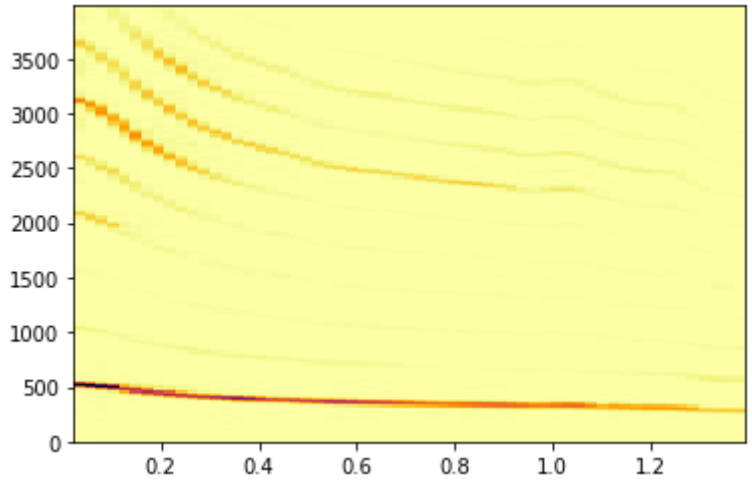

Рисунок 5.3. Спектрограмма звука

Объединим 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)')
```

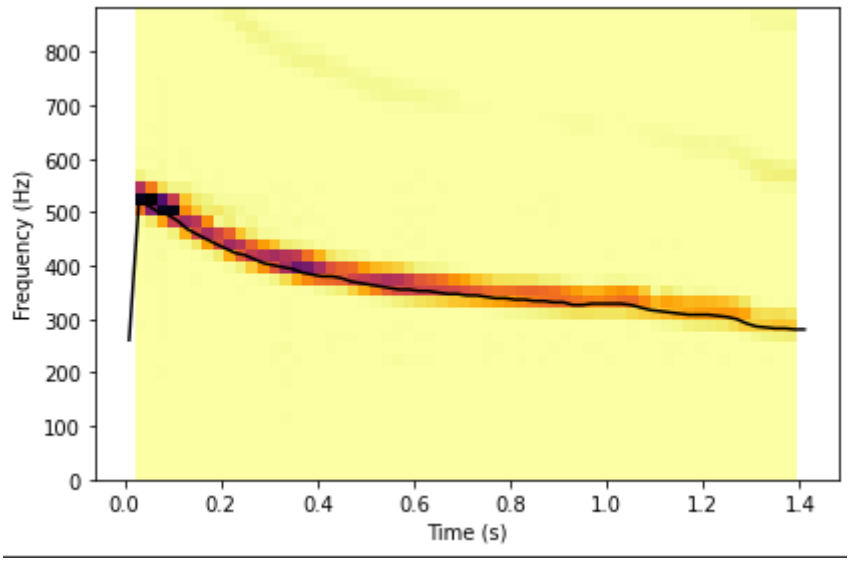

Рисунок 5.4. Результат оценки

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'):
    if not os.path.exists('market-pric.csv'):
    !wget https://github.com/pimenov01/telecom/raw/main/files/data.csv

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()
```

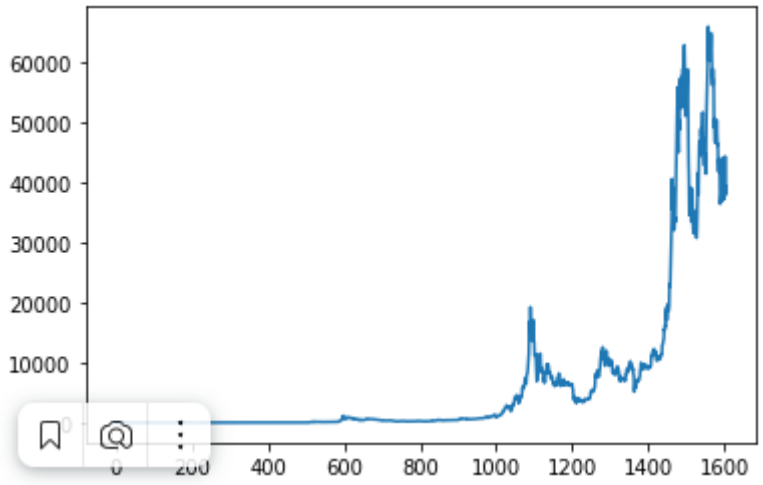

Рисунок 5.5. График цены на BitCoin

Вычислим автокорреляцию:

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

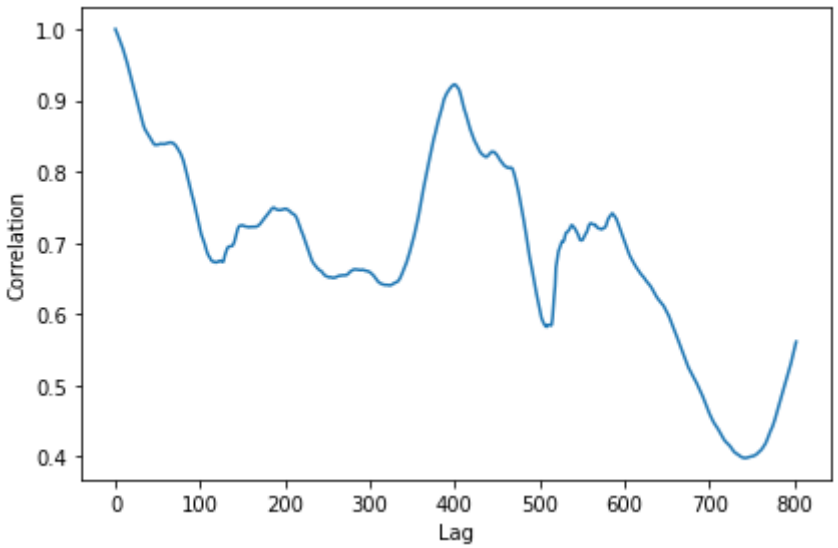

Рисунок 5.6. Автокорреляция функции цены на BitCoin

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)')
```

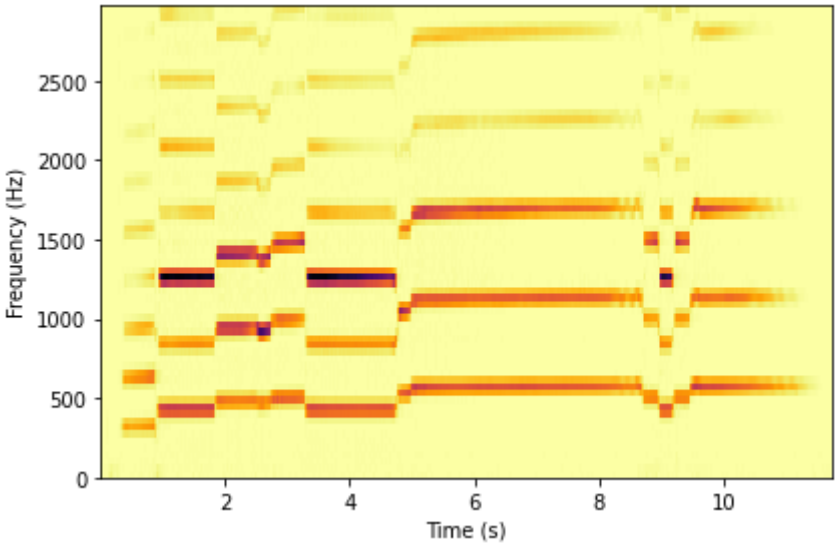

Рисунок 5.7. Спектрограмма сегмента

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')
```

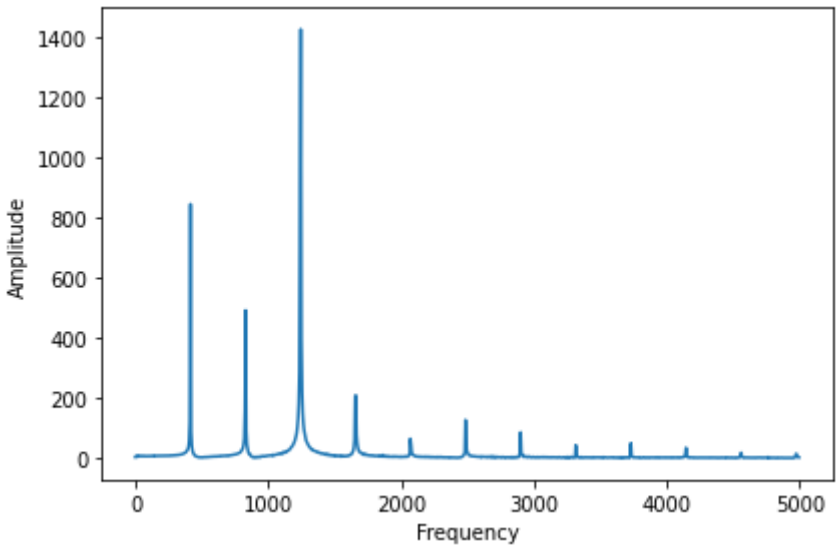

Рисунок 5.8. Спектр другого сегмента

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])
```

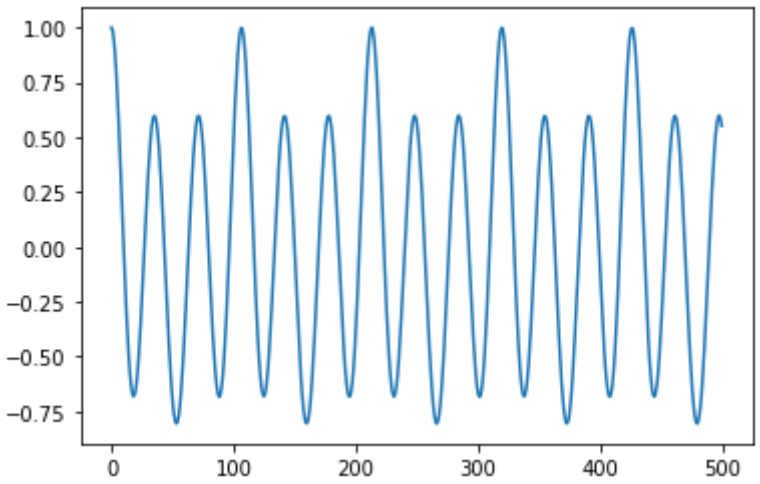

Рисунок 5.9. График после АКФ

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')
```

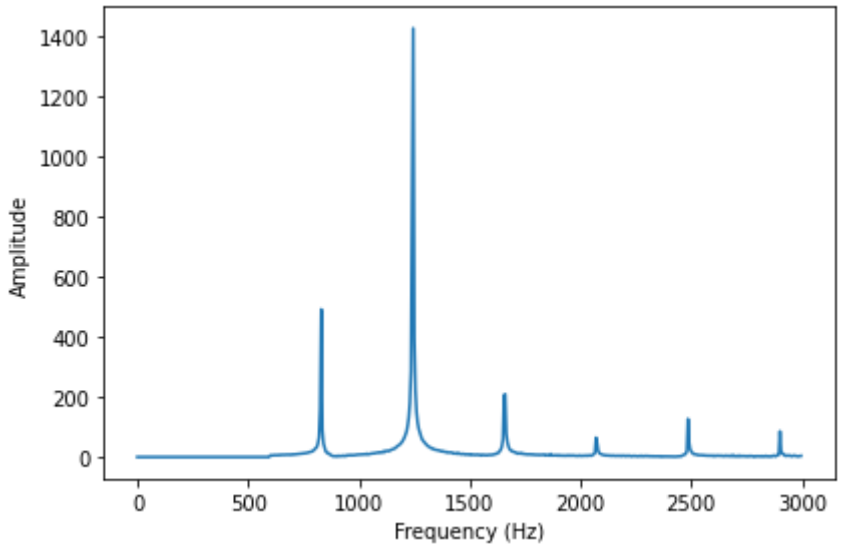

Рисунок 5.10. Спектр после ФНЧ

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])
```

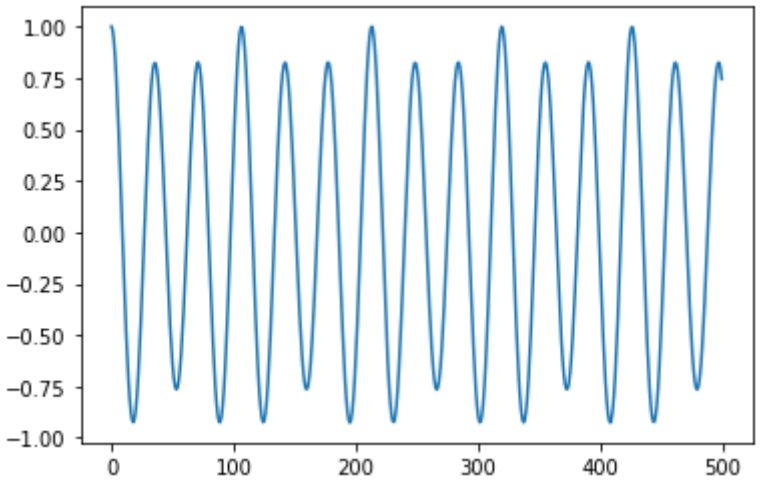

Рисунок 5.11. Полученный график

```
estimate_fundamental(segment)
```

```
416.0377358490566
```

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

5.5. Вывод

В ходе данной ЛР мы изучали корреляцию и ее влияние на сигналы. Также был написан код для сигнала с "отсутствующим фундаментом".

6. Дискретное косинусное преобразование

6.1. Упражнение 1

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

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

Зазадим размер массива степенями двойки.

```
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)
```

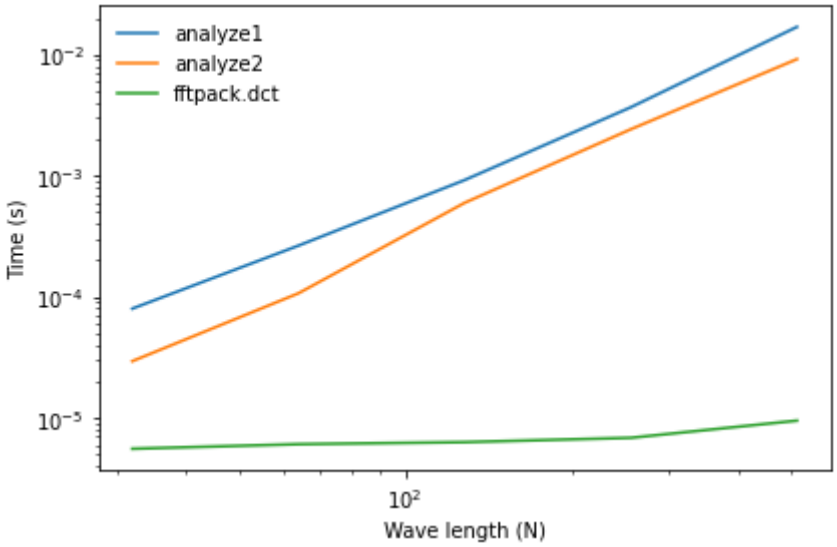

Рисунок 6.1. Время работы различных методов ДКП

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

6.2. Упражнение 2

Реализовать алгоритм сжатия для музыки или речи.

Выберем звук для сжатия:

```
if not os.path.exists('1647_piano.wav'):
    !wget https://github.com/pimenov01/telecom/raw/main/files/1647_piano.wa
    wave = read_wave('1647_piano.wav')
```

Для начала возьмём небольшой сегмент:

```
segment = wave.segment(start = 1.7,duration = 1.0)
segment.normalize()
segment.make_audio()
```

Используем DCT вместо DFT.

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

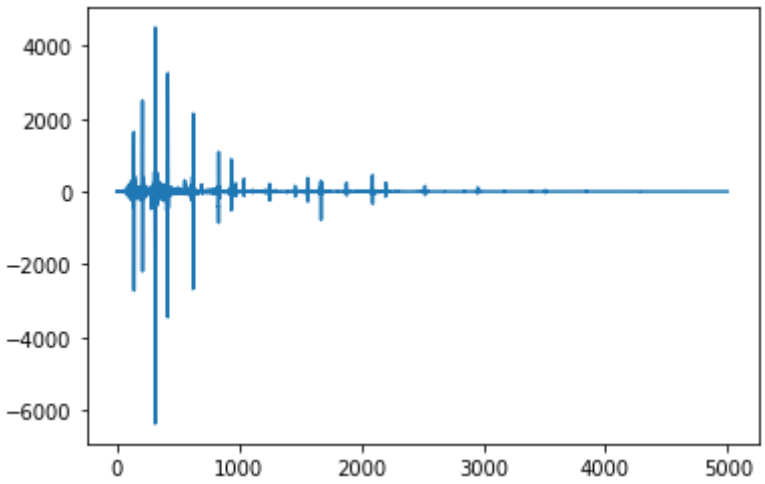

Рисунок 6.2. Спектр сигнала полученный при помощи ДКП

```
def filtering(dct,limit = 0):
  for i, amp in enumerate(dct.amps):
    if np.abs(amp) < limit:
      dct.hs[i] = 0

filtering(dct,1000)
dct.plot(high = 5000)
```

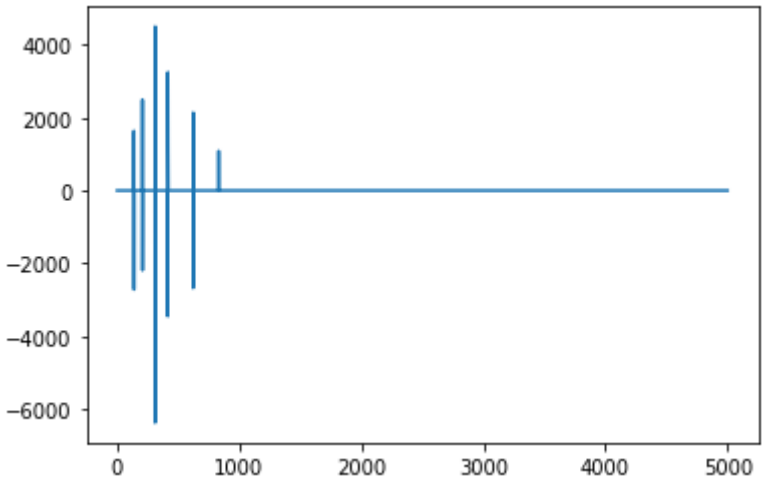

Рисунок 6.3. ДКП после фильтрации

Теперь подберём значение, чтобы результат звучал как исходник. Для эффективного хранения данных можно использовать разряженные массивы.

6.3. Управжнение 3

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

```
signal = SawtoothSignal(freq=500, offset=0)
wave = signal.make_wave(duration=0.5, framerate=40000)
wave.segment(start=0.005,duration=0.01).plot()
```

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

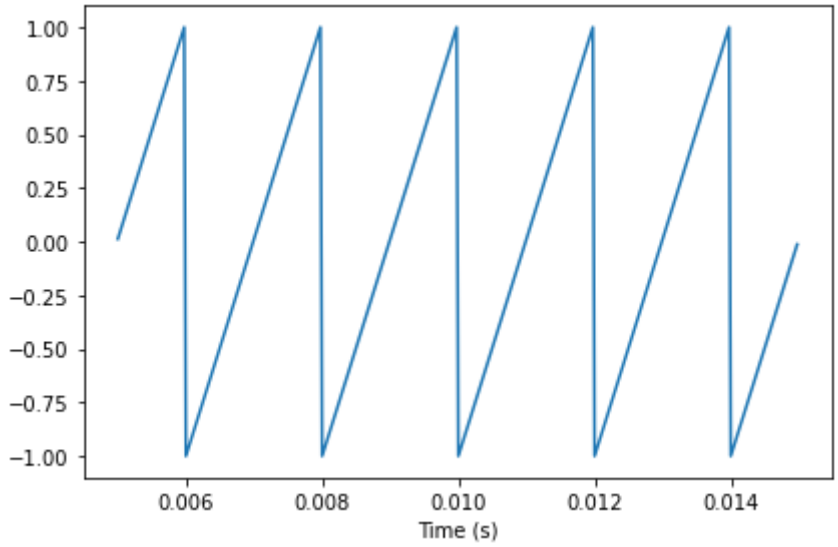

Рисунок 6.4. Выбранный сегмент

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

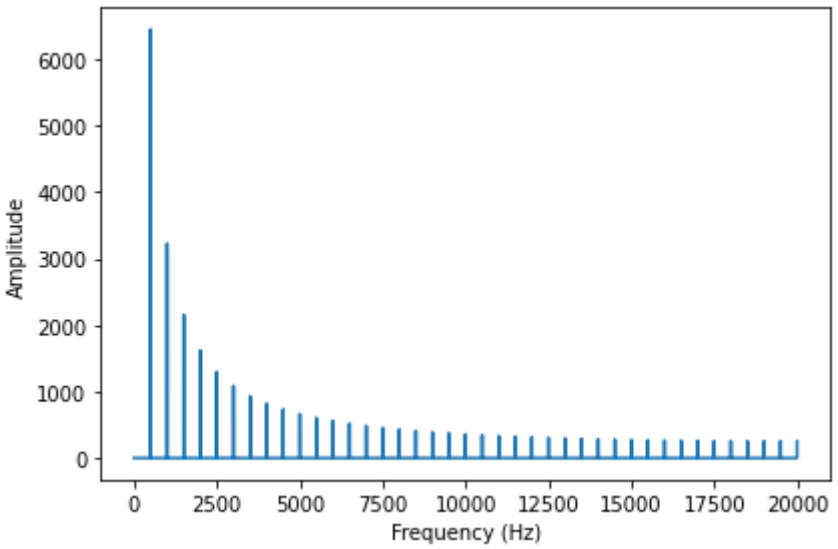

Рисунок 6.5. Спектр сегемента

```
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)')
```

```
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())

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

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

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

plot_three(spectrum)
```

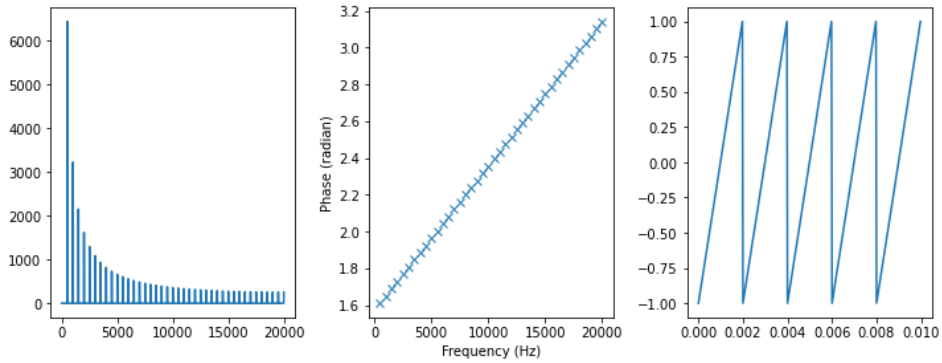

Рисунок 6.6. Получившиеся графики

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

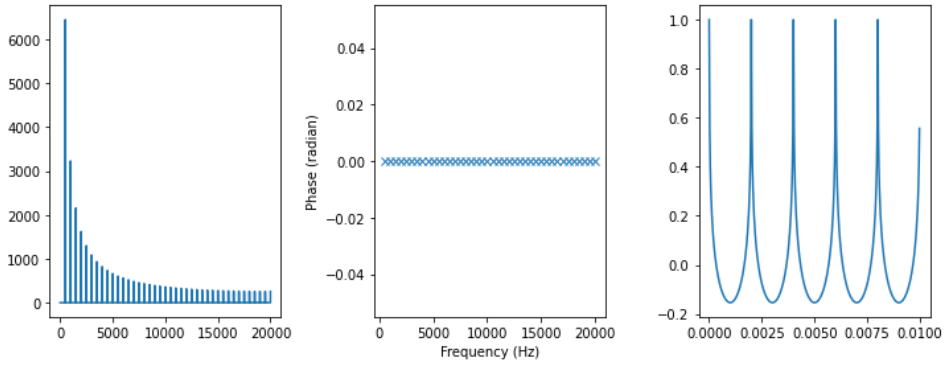

Рисунок 6.7. Получившиеся графики

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

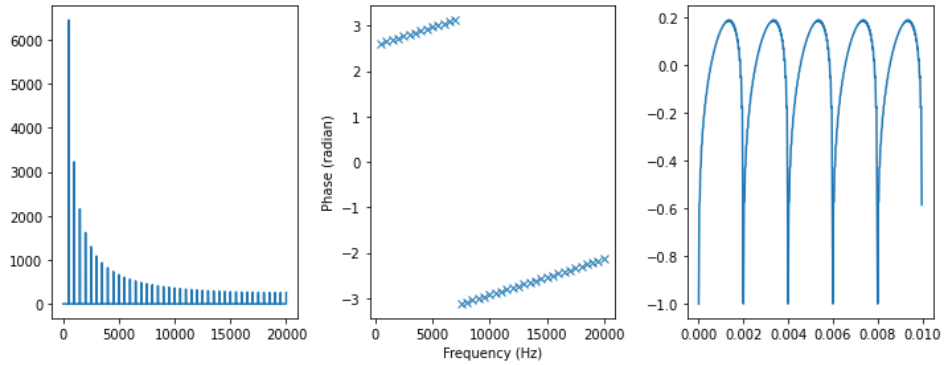

Рисунок 6.8. Получившиеся графики

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

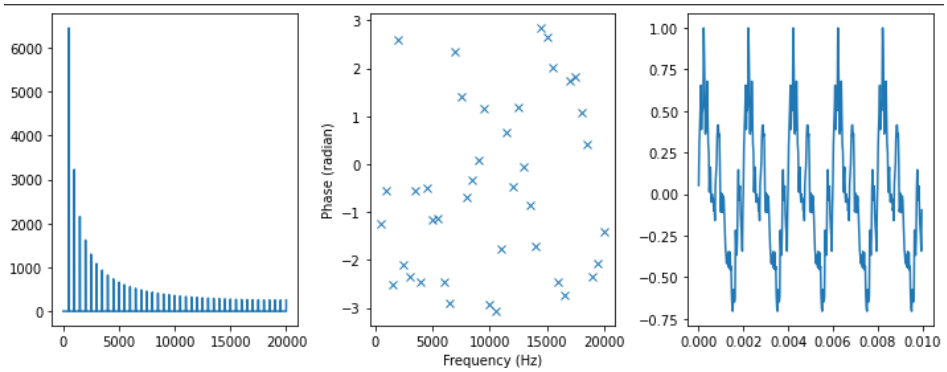

Рисунок 6.9. Получившиеся графики

Теперь возьмём другой звук и сделаем всё тоже самое:

```
if not os.path.exists('120994__thirsk__120-oboe.wav'):
    !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/120994__t
wave = read_wave('120994__thirsk__120-oboe.wav')
segment = wave.segment(start=0.1, duration=0.5)
spectrum = segment.make_spectrum()

plot_three(spectrum)
```

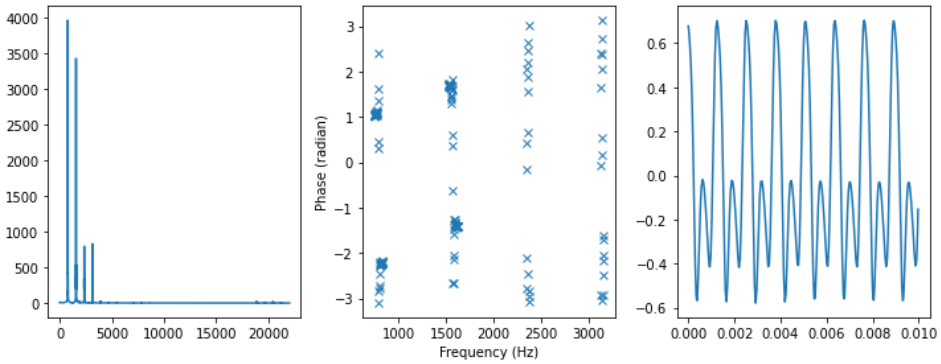

Рисунок 6.10. Получившиеся графики

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

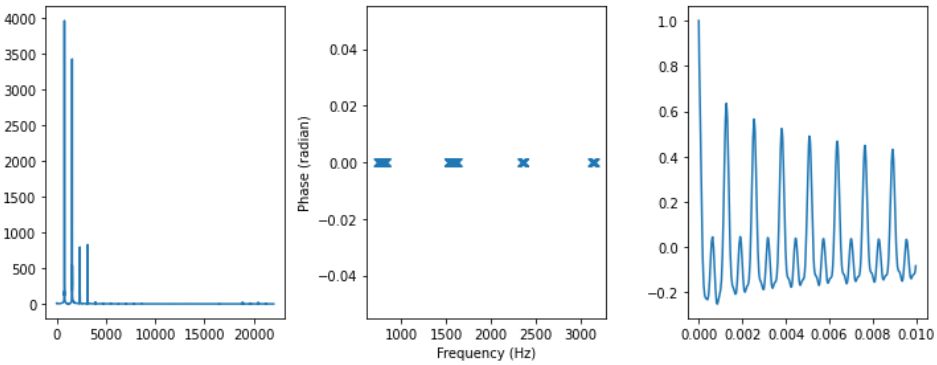

Рисунок 6.11. Получившиеся графики

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

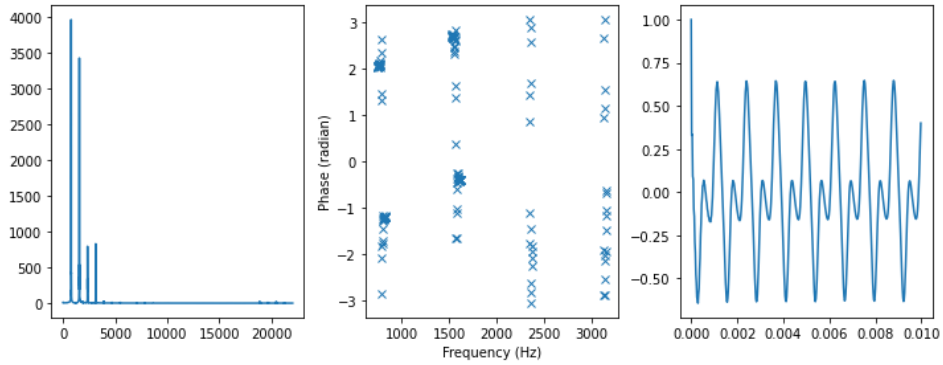

Рисунок 6.12. Получившиеся графики

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

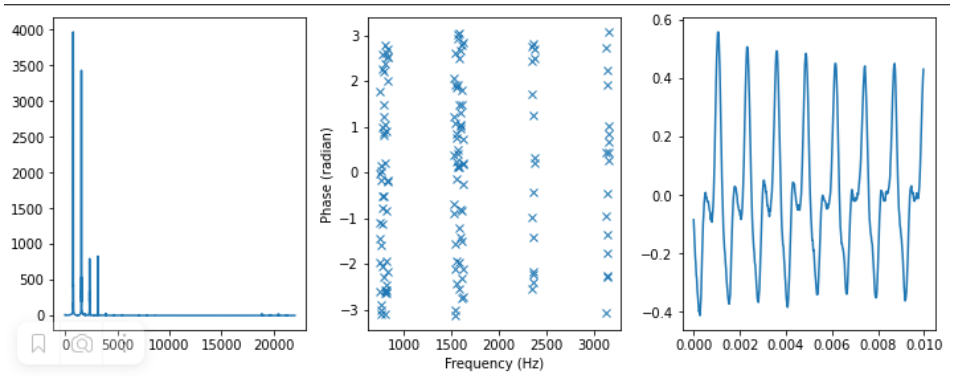

Рисунок 6.13. Получившиеся графики

Звук при прослушивании остался таким же, хотя мы изменили сигнал достаточно сильно. Если гармоническая структура звука неизменна - то для звуков с простой структурой мы не услышим изменений в фазовой структуре.

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

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 и протестирован алгоритм БПФ. Он 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);
```

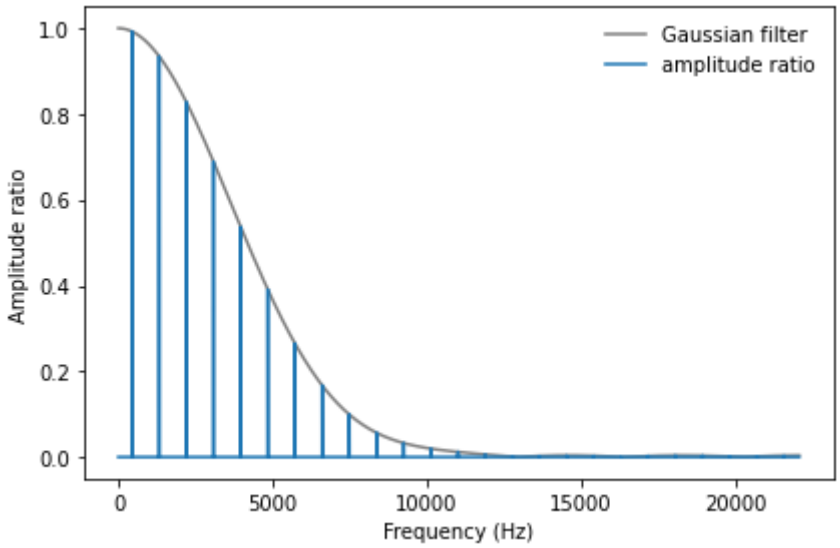

Рисунок 8.1. Гауссово окно для фильтрации

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

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

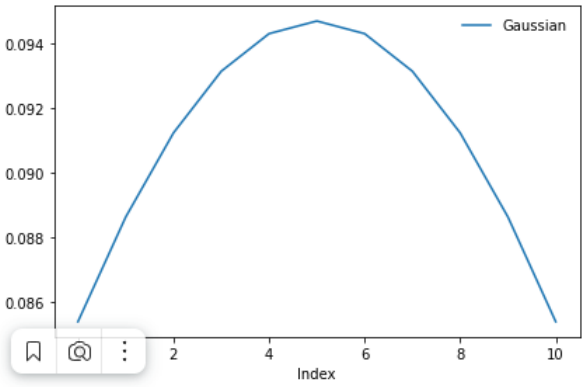

Рисунок 8.2. Гауссово окно

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

```
gaussian /= sum(gaussian)

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

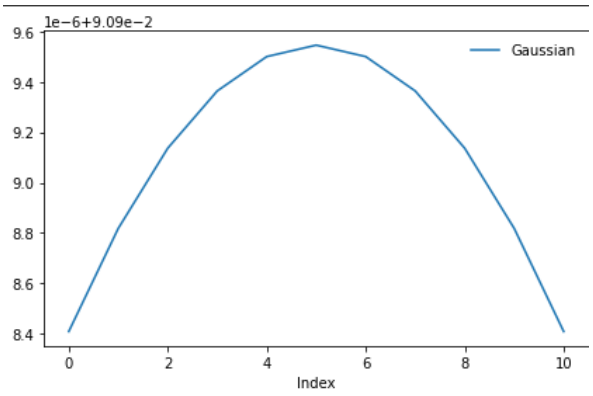

Рисунок 8.3. Гауссово окно

Исходя из графиков видно, что БПФ стала меньше, а кривая - шире.

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')
```

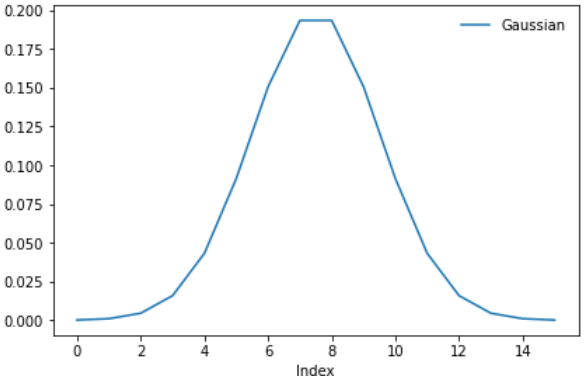

Рисунок 8.4. Гауссово окно

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

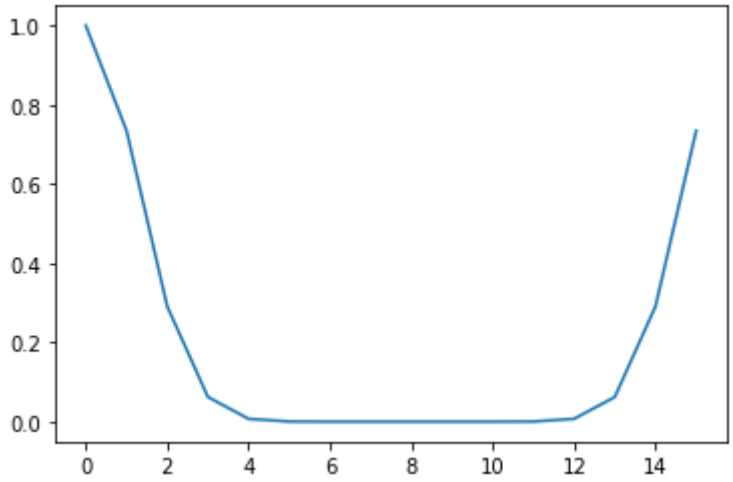

Рисунок 8.5. FTT применённое на окно

Сделаем свертку отрицательных частот влево.

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

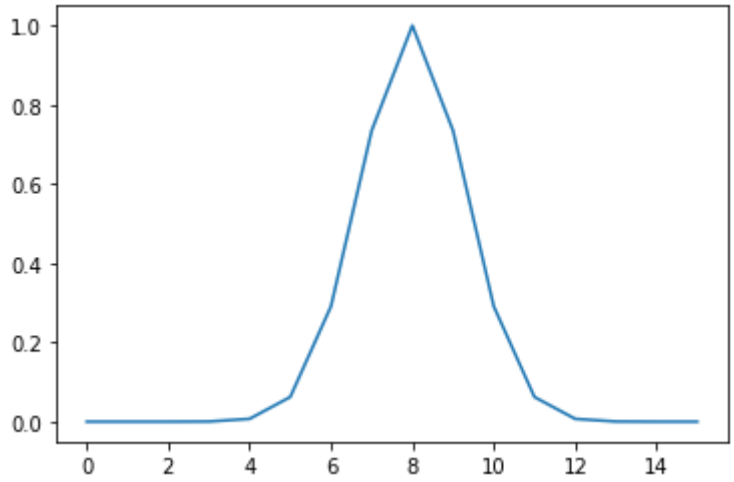

Рисунок 8.6. Результат свертки

Можно отметить, что 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 равной 1 секунде и протестируем.

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

sig.plot()
```

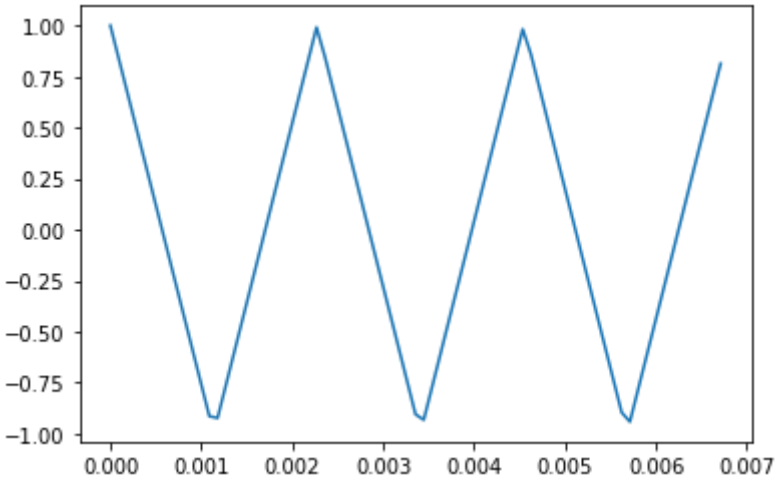

Рисунок 8.7. Пилообразный сигнал

Далее 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()
```

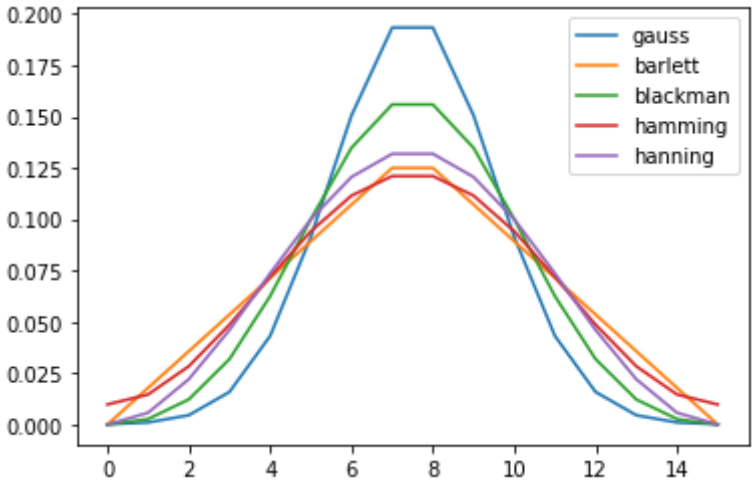

Рисунок 8.8. Применение различных окон на выбранный сигнал

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()
```

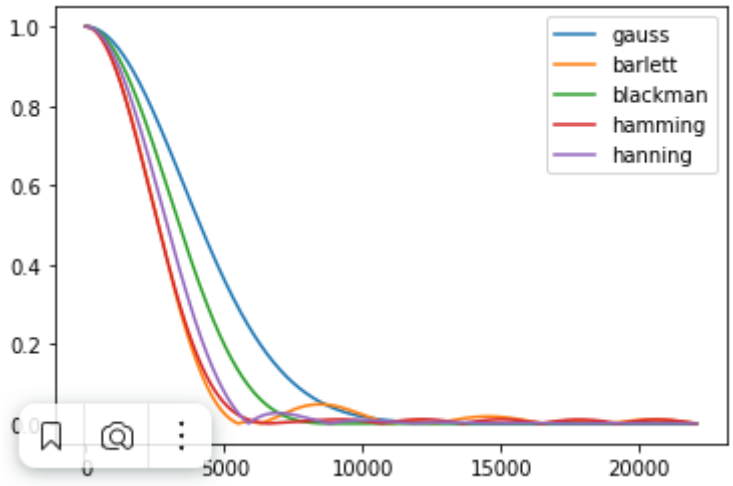

Рисунок 8.9. Применение различных окон на выбранный сигнал

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')
```

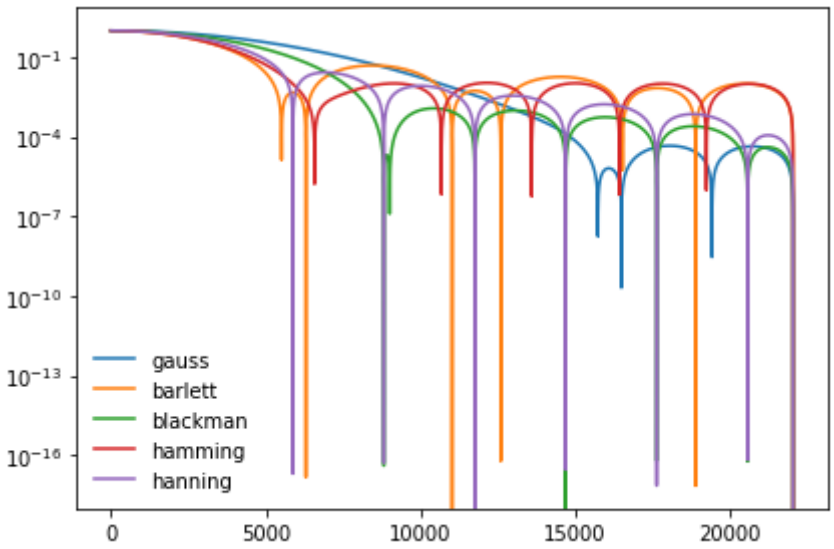

Рисунок 8.10. Логорифмический масштаб

Исходя из результатов можно предположить, что Хэнинг лучше всего подходит для фильтрации низких частот.

8.4. Вывод

В ходе данной ЛР были рассмотренны операции фильтрации, сглаживания и свертки. Каждая из этих операций может быть полезной для какой-либо определенной задачи. Например, сглаживание 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?

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

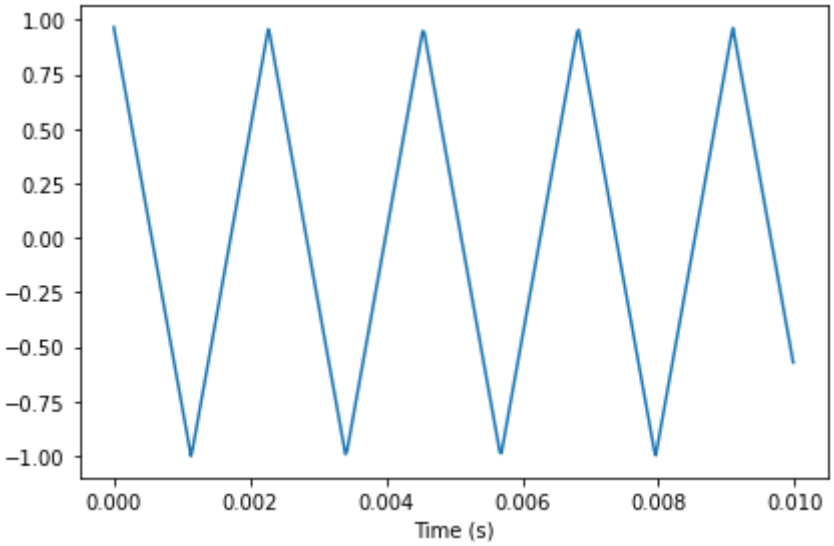

Рисунок 9.1. График сигнала

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

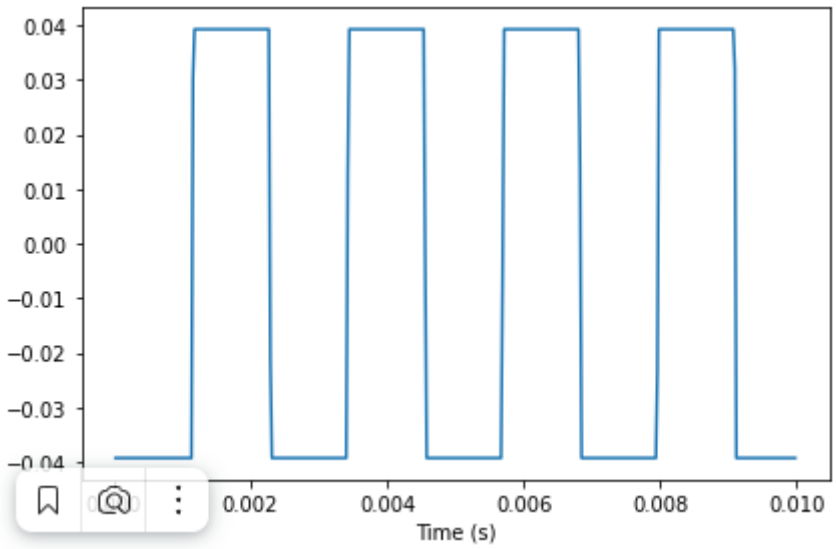

Рисунок 9.2. Сигнал после применения diff

В итоге получили прямоугольный сигнал с такой же частотой.

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

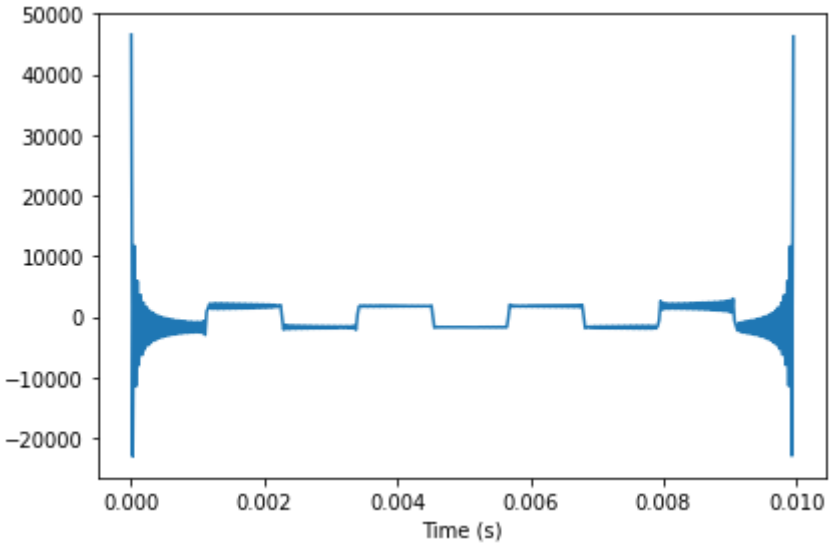

Рисунок 9.3. Сигнал после применения differentiate

Из графика видно, что начало и конец интервала сильно зашумлены. Возможно, это связано с невозможностью определения производной.

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?

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

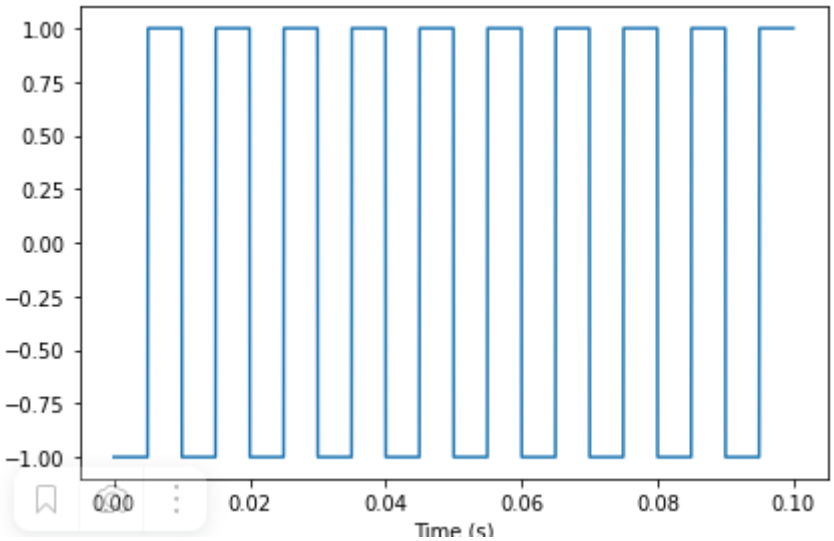

Рисунок 9.4. Рассматриваемый сигнал

Кумулятивная сумма:

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

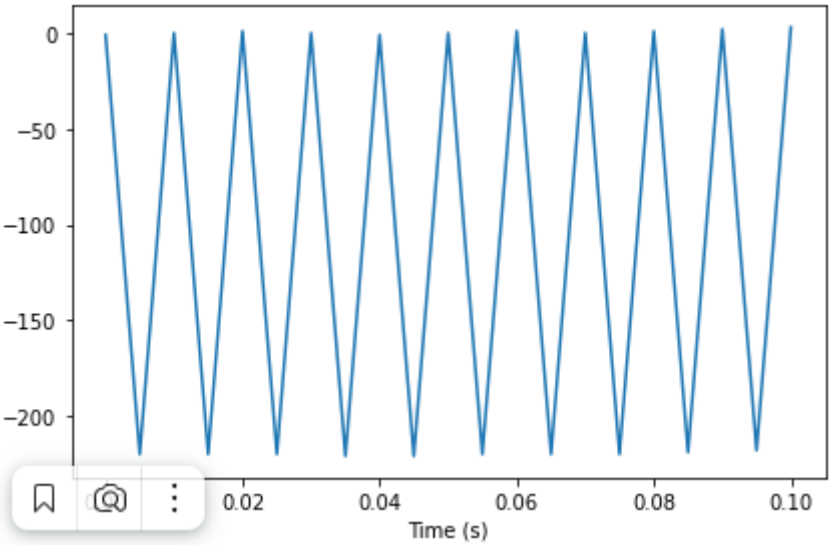

Рисунок 9.5. Рассматриваемый сигнал после применения cumsum

Получили 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)')
```

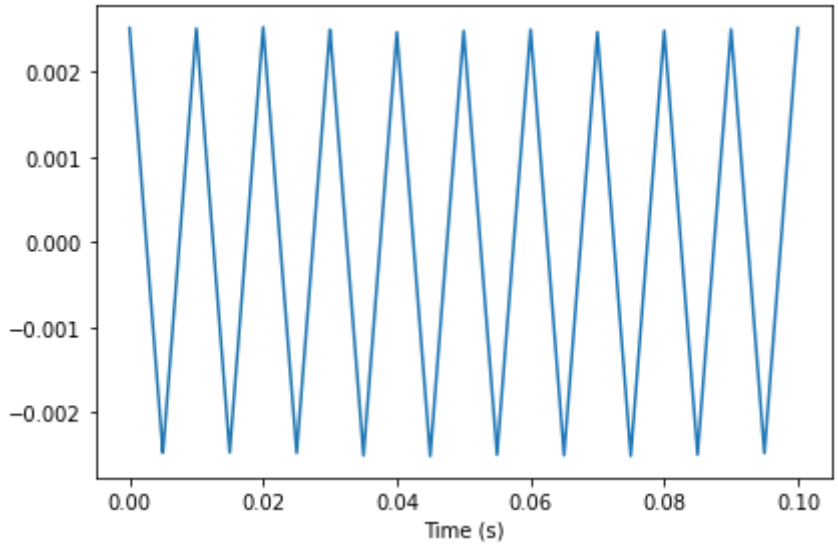

Рисунок 9.6. Рассматриваемый сигнал после применения integrate

Воспользуемся кодом автора и "сложим" два графика с изменением масштаба.

```
cumsum_wave.unbias()
cumsum_wave.normalize()
int_wave.normalize()
cumsum_wave.plot()
int_wave.plot()
```

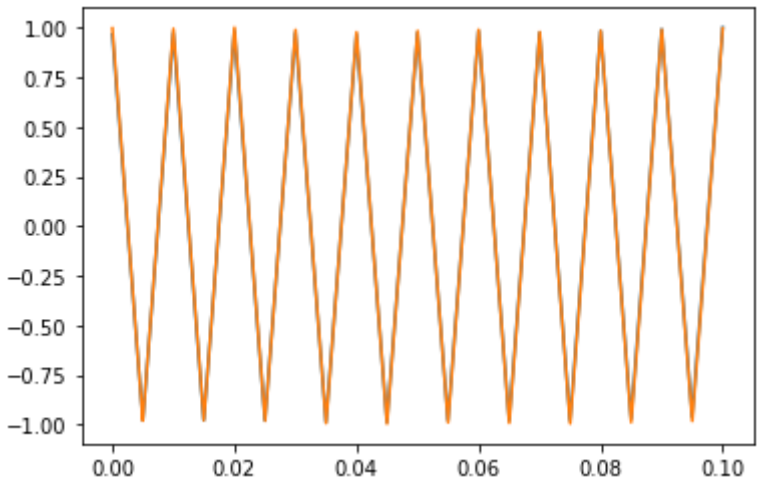

Рисунок 9.7. Сравнение полученных графиков

Видим, что графики практически идентичны. Следовательно разные у них лишь амплитуды.

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?

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

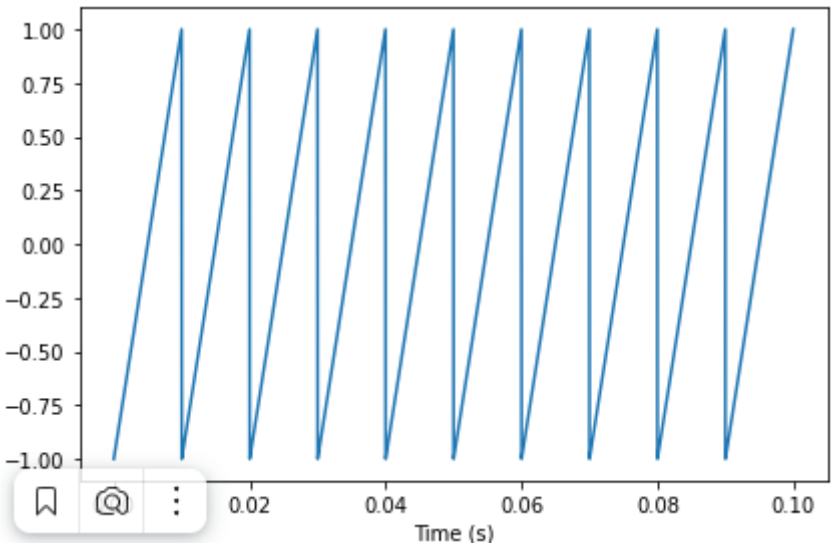

Рисунок 9.8. Пилообразный сигнал

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

wave1 = spectrum.make_wave()
wave1.plot()
decorate(xlabel='Time (s)')
```

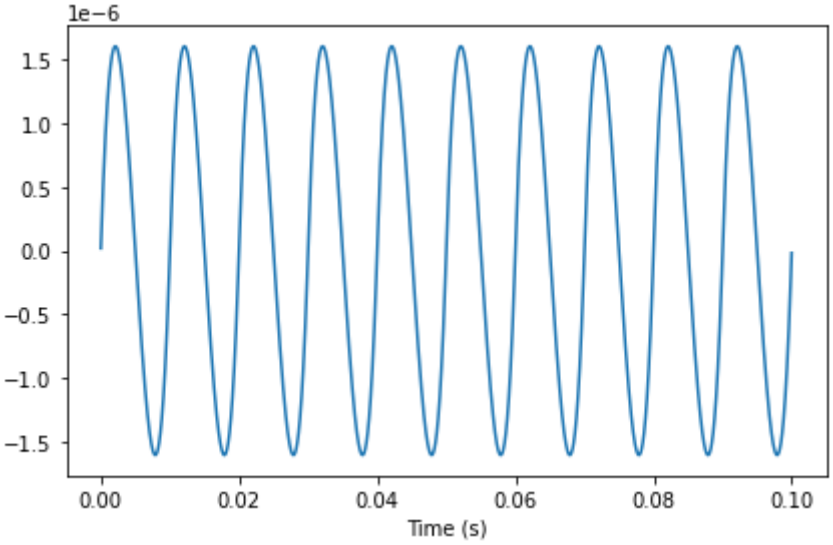

Рисунок 9.9. Изменённый сигнал

Из графика видно, что сигнал действительно напоминает синусоиду. Причиной этому является фильтрация низких чистот, за исключением основной.

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

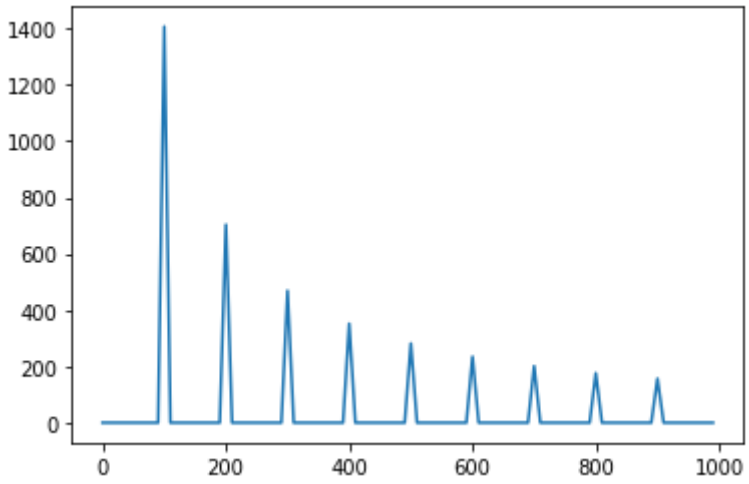

Рисунок 9.10. Спектр исходного сигнала

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

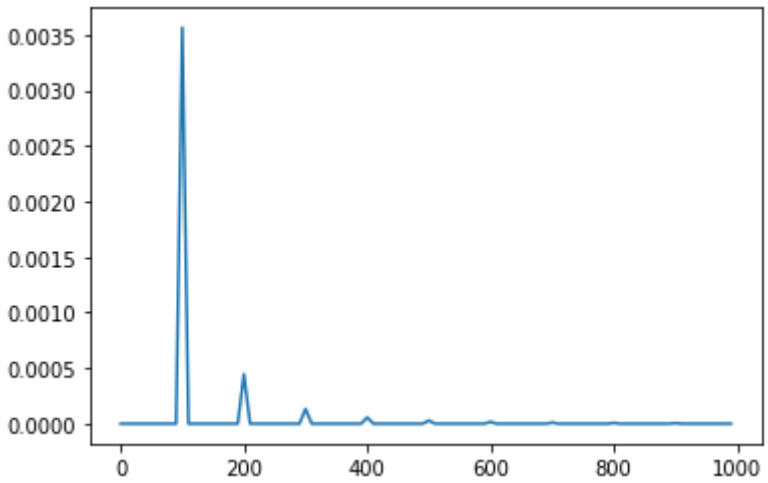

Рисунок 9.11. Спектр нового сигнала

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.

Тут надо точно подобрать параметры, чтобы сигнал красиво отображался при таком маленьком frametime.

```
wave = CubicSignal(freq=0.01).make_wave(duration=1000, framerate=1)
wave.plot()
```

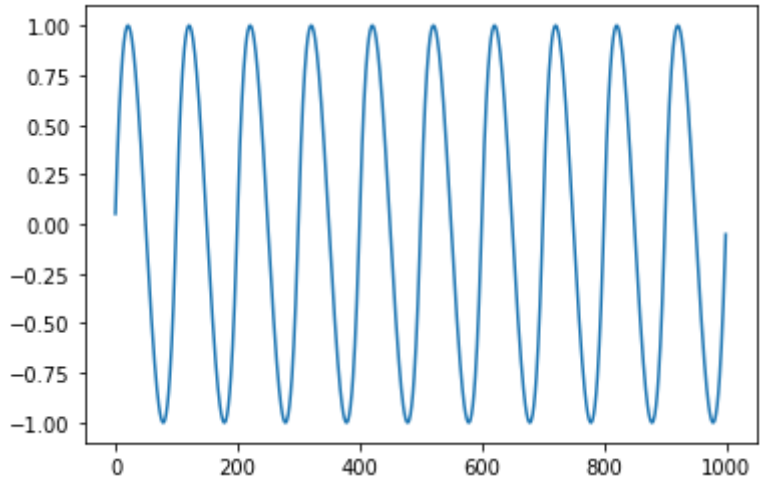

Рисунок 9.12. Кубический сигнал

Первая разность - параболы.

```
d1_wave = wave.diff()
d1_wave.plot()
```

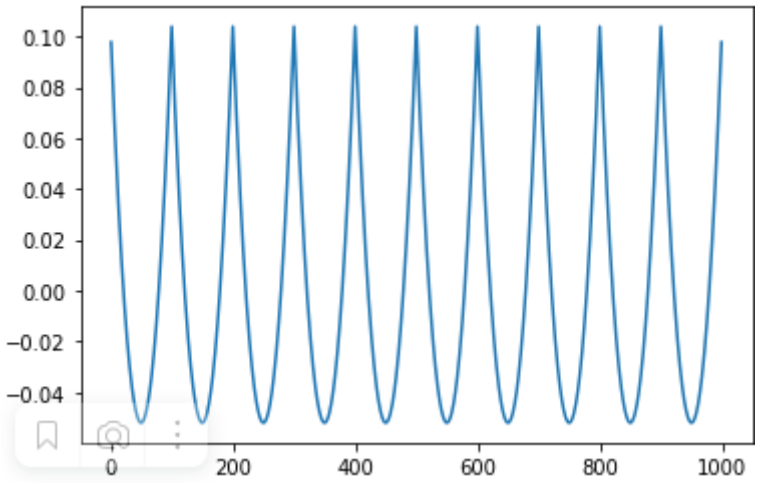

Рисунок 9.13. Первая разность

Вторая разность - пилообразный сигнал.

```
d2_wave = d1_wave.diff()
d2_wave.plot()
```

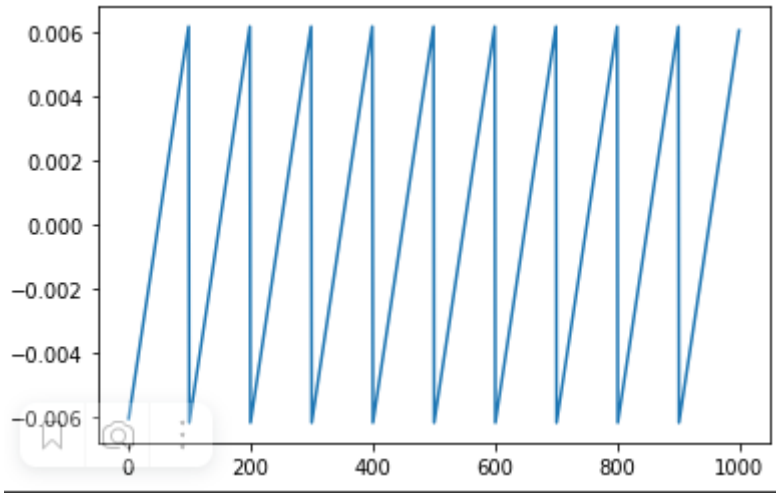

Рисунок 9.14. Вторая разность

При двойном дифференцировании получаем звон в пилообразном сигнале, звон связан со сложностями в вычислении производной, как и ранее.

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

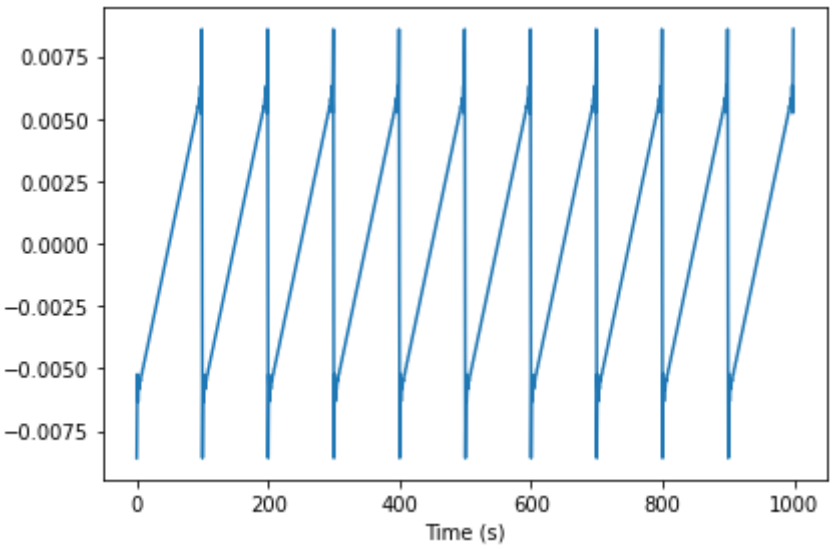

Рисунок 9.15. Полученный сигнал со звоном

Фильтры:

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

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

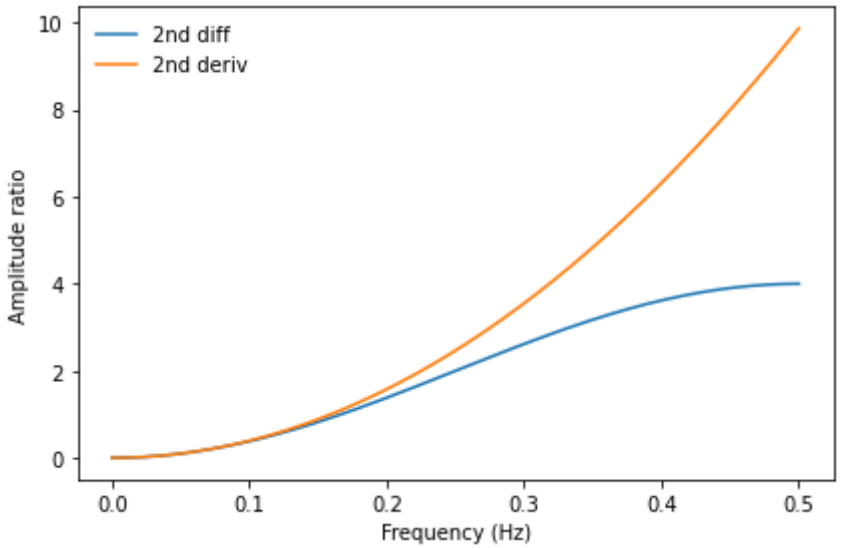

Рисунок 9.16. Полученные фильтры

Мы получили фильтры для усиления высокочастотных компонент. Производная является параболой, поэтому она усиливает сильней. Разность хорошо аппроксимирует только на низких частотах, а далее получаем существенное отклонение.

9.5. Вывод

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

10. Сигналы и системы

10.1. Упражнение 1

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

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)')
```

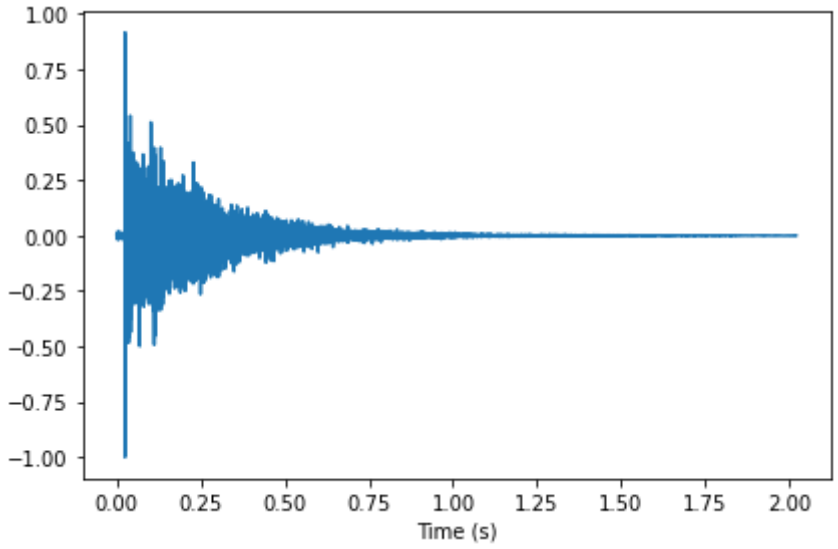

Рисунок 10.1. Сигнал

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

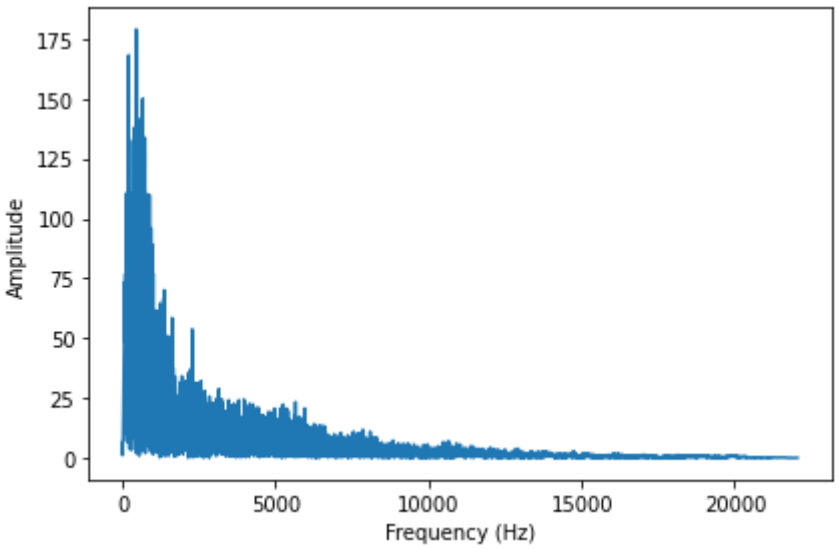

Рисунок 10.2. Спектр сигнала

Теперь перейдём к самой записи:

```
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)')
```

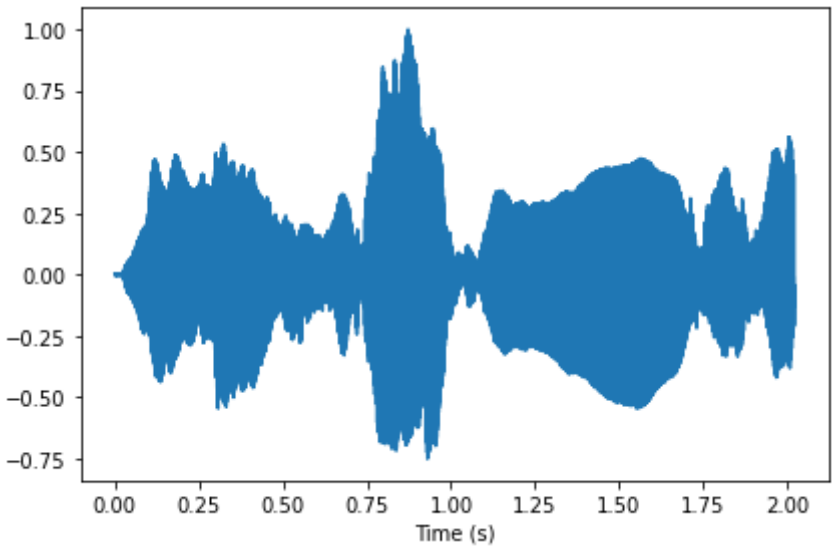

Рисунок 10.3. График сигнала

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

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

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

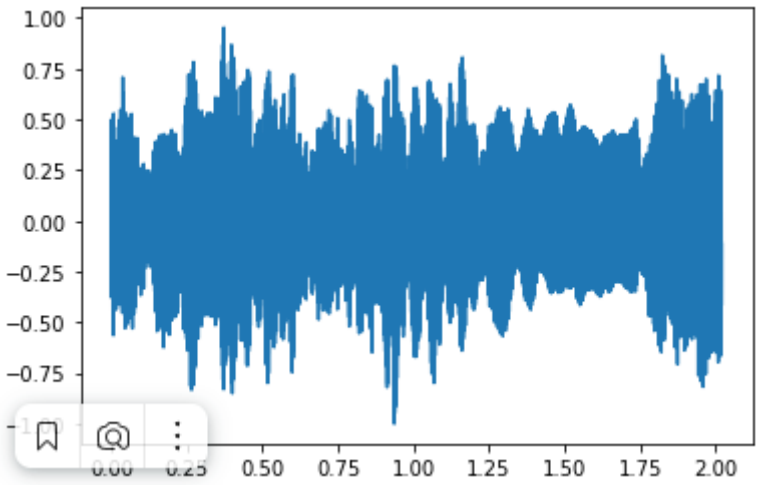

Рисунок 10.4. График получившегося сигнала

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)')
```

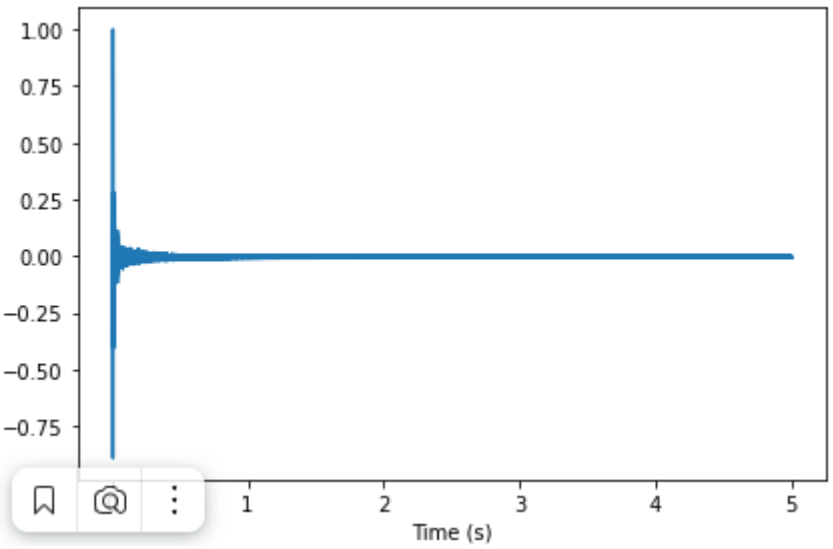

Рисунок 10.5. График загруженного сигнала

ДПФ:

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

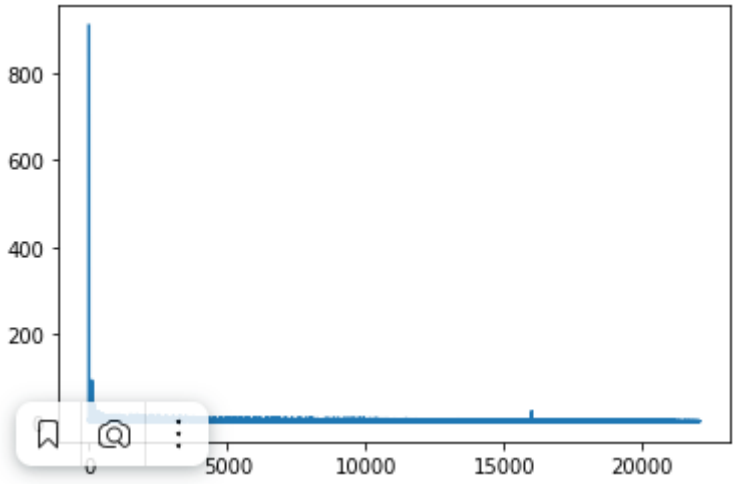

Рисунок 10.6. ДПФ импульсной характеристики

Будем использовать звук скрипки из учебника и смоделируем ее запись в пространстве.

```
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()
```

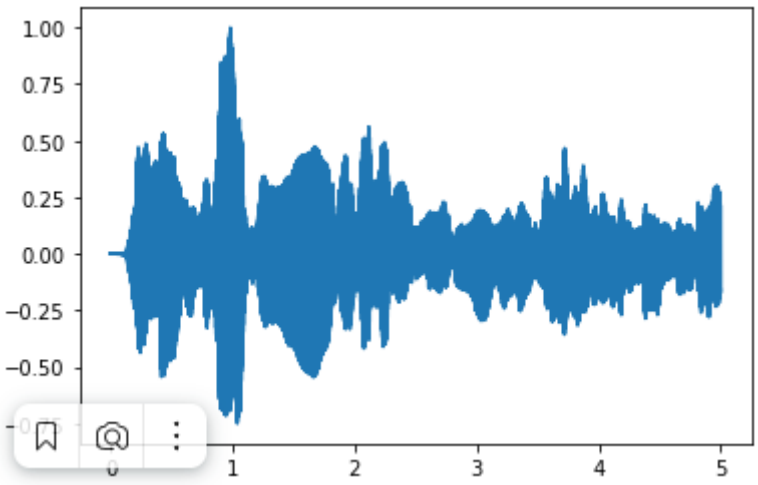

Рисунок 10.7. Сигнал звука скрипки

```
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()
```

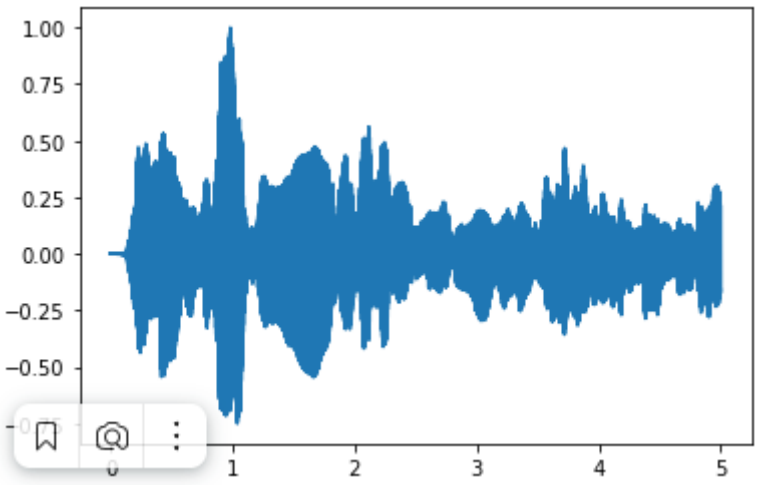

Рисунок 10.8. Полученный график

Оба звучания были успешно смоделированы разными способами

10.3. Вывод

В ходе данной ЛР 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.

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

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

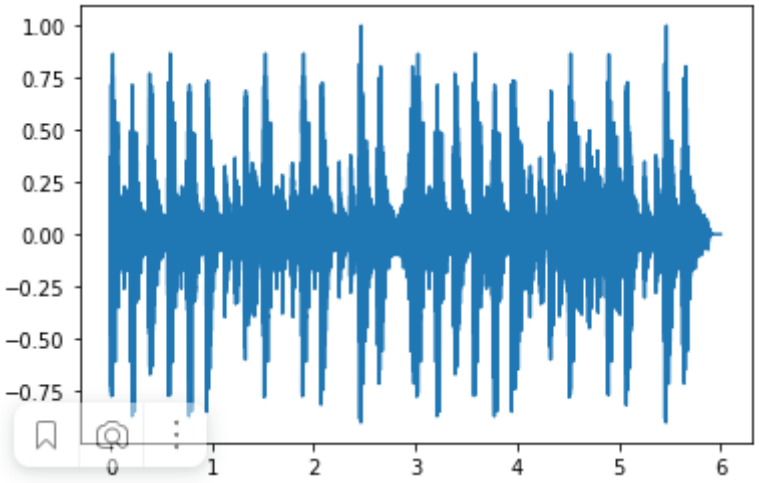

Рисунок 11.1. График "Соло на барабане"

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

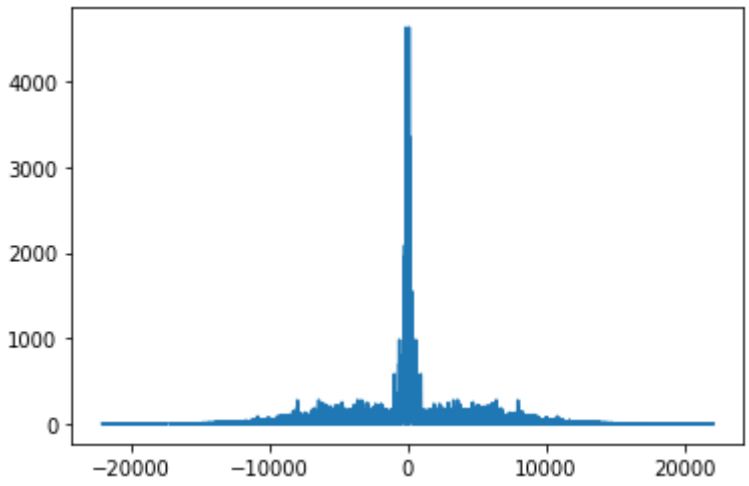

Рисунок 11.2. Спектр сигнала

Применим фильтр низких частот:

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

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

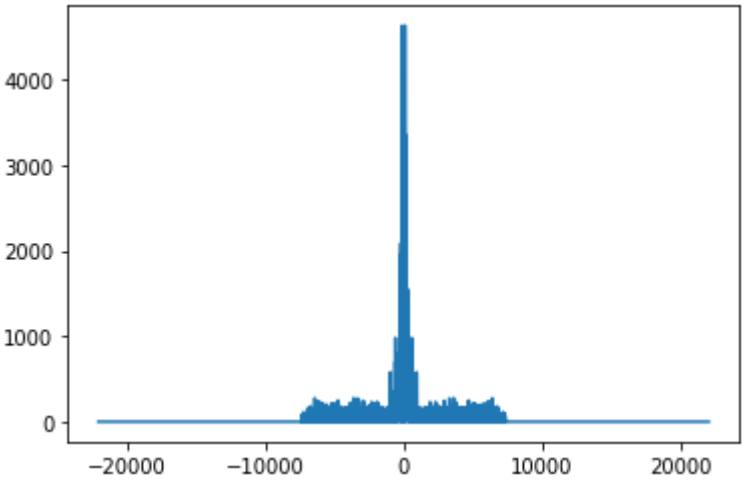

Рисунок 11.3. Отфильтрованный сигнал

Функция сэмплирования:

```
def sample(wave, factor):
    """Simulates sampling of a wave.

    wave: Wave object
    factor: ratio of the new framerate to the original
    """
    ys = np.zeros(len(wave))
```

```
    ys[::factor] = wave.ys[::factor]
    return Wave(ys, framerate=wave.framerate)

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

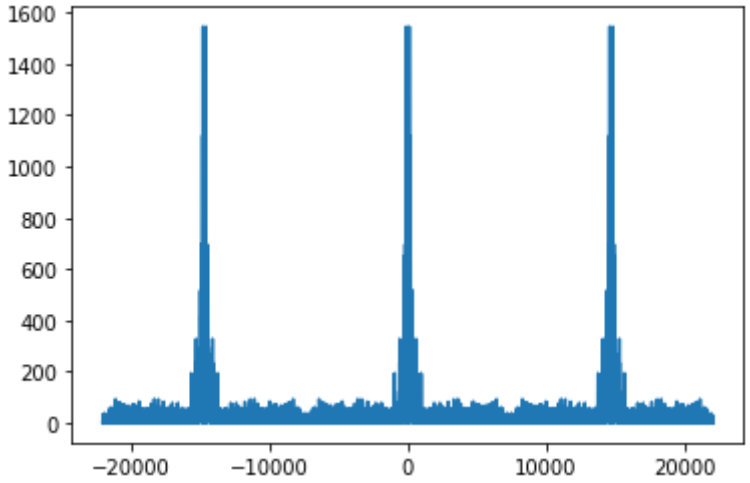

Рисунок 11.4. Получившийся спектр

Далее избавимся от спектральных копий:

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

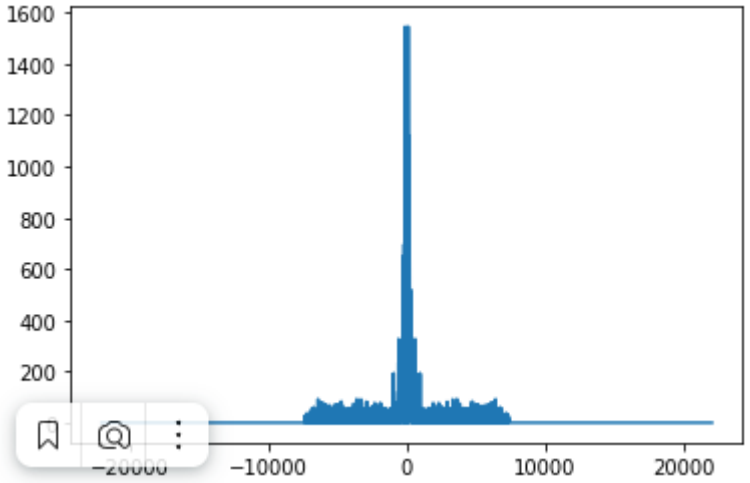

Рисунок 11.5. Результат избавления от копий

Звуки отличаются, чтобы понять в чём проблема сравним спектры:

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

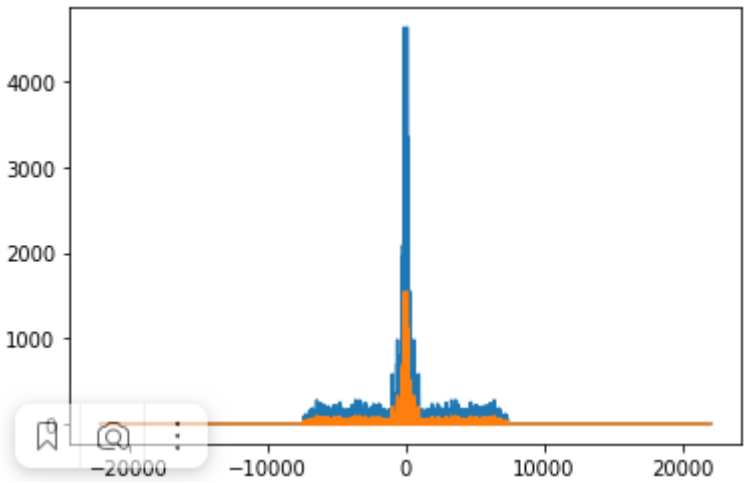

Рисунок 11.6. Сравнение спектров

Графики не совпадают. Чтобы добиться совпадения - надо увеличить ампилитуду в 3 раза:

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

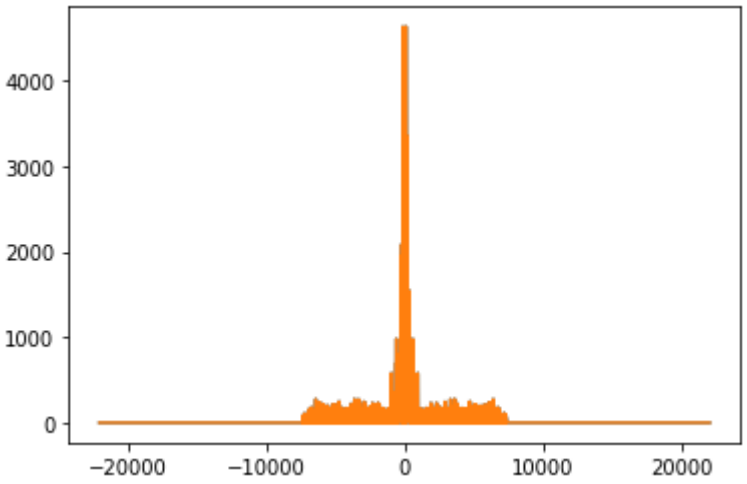

Рисунок 11.7. Сравнение спектров

После всех проделанных действий разница действительно есть, однако - она невелика.

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

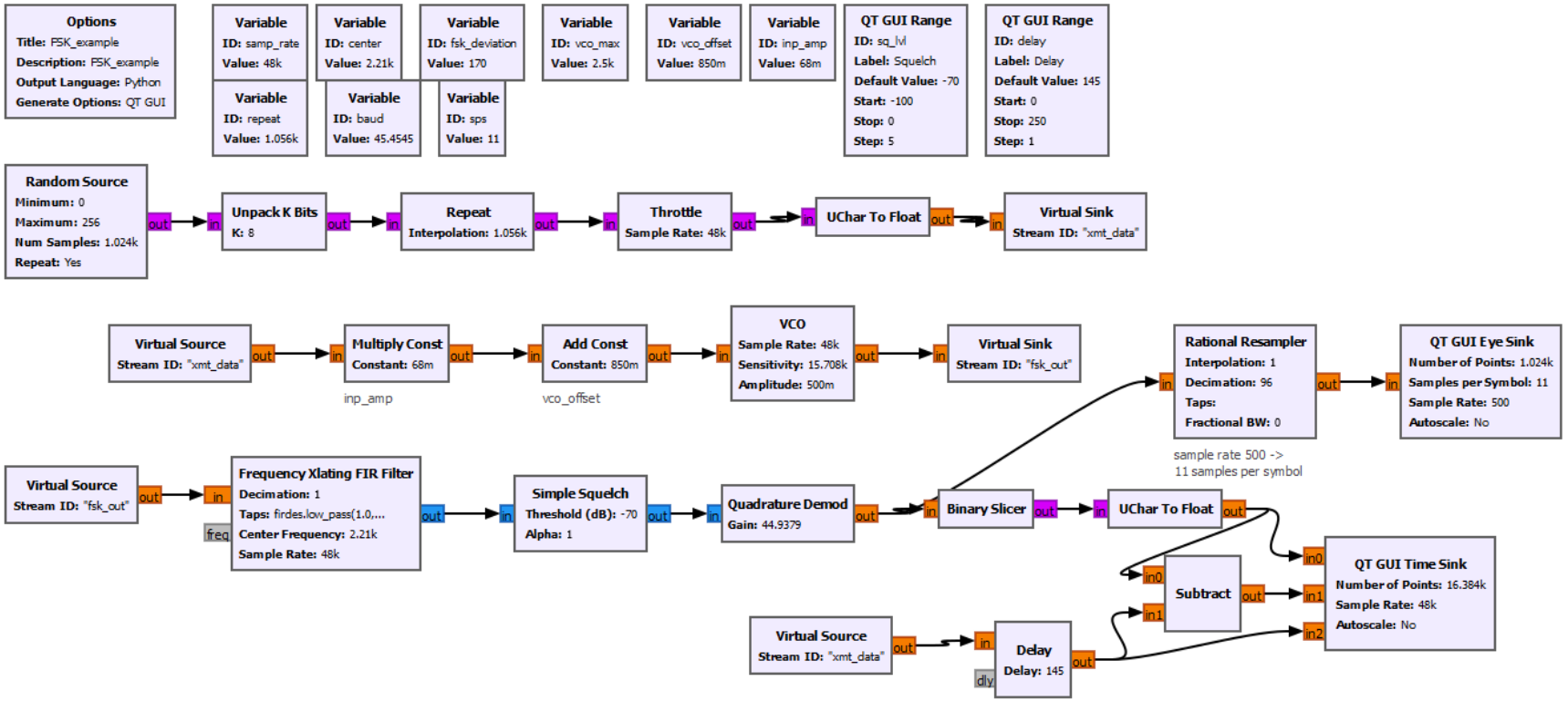

Рисунок 12.1. Схема FSK

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 этом примере используется Baudot Radioteletype, следовательно битовое время = 22 миллисекунды. Получаем скорость передачи 45,4545. Коэффицент повторения равен samp_rate * 0,022.

В VCO генерируются сигналы 2295 Гц (отметка = 1) и 2125 Гц (отметка = 0). 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. 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)'.

Теперь посмотрим, что у нас с данными: 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.

12.2. Тестирование

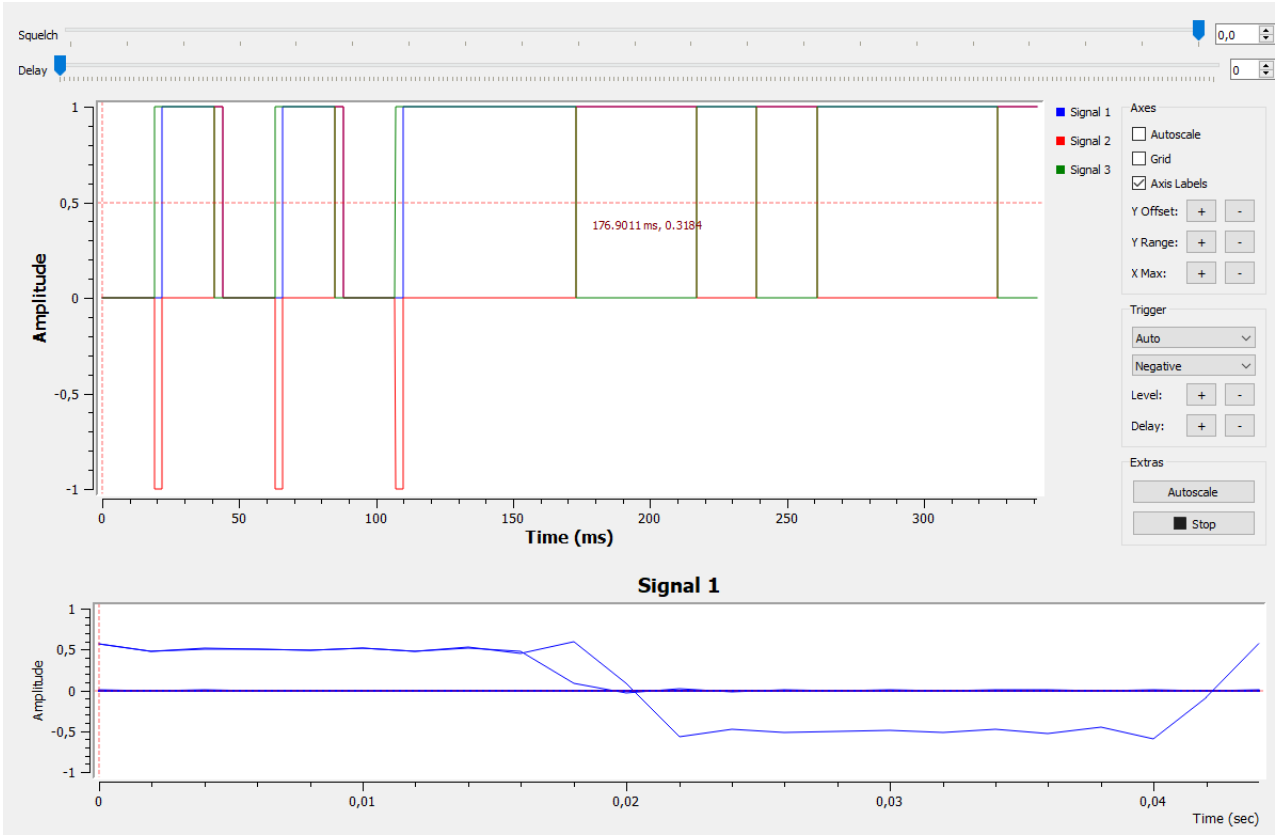

Рисунок 12.2. Модуляция без задержки и шума

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.

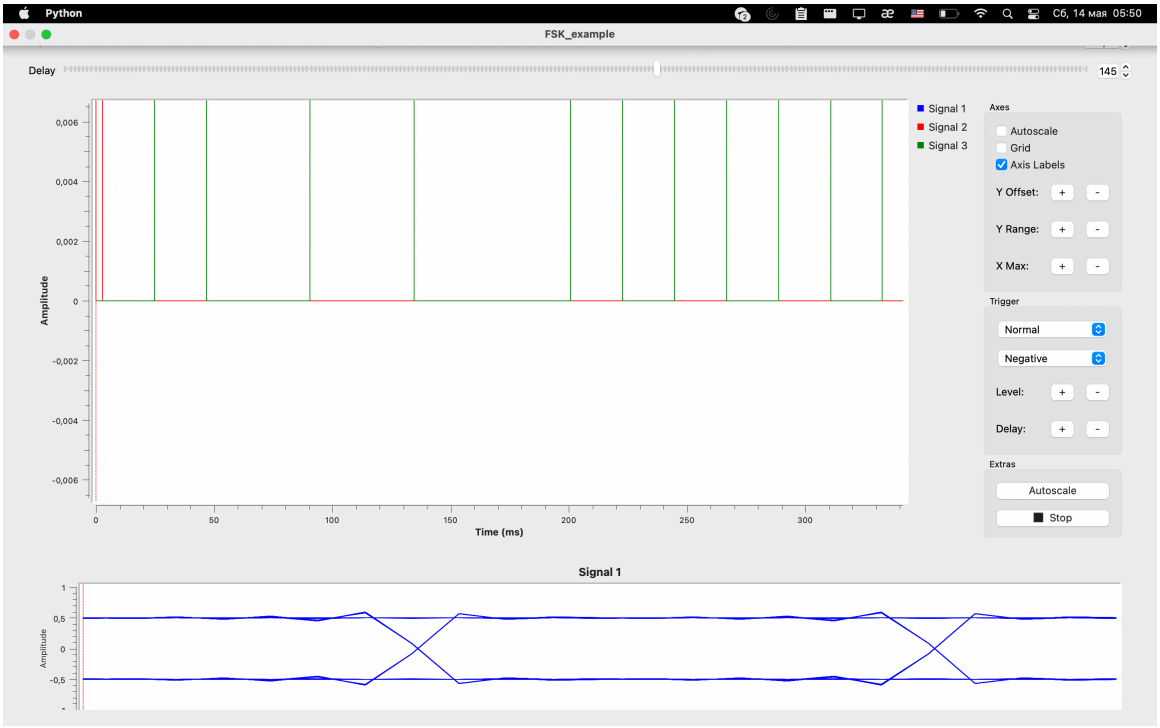

Рисунок 12.3. Модуляция с правильной задержкой

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 и протестирована необходимая модель .