

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

Телекоммуникационные технологии

Отчёт по лабораторным работам

Работу

выполнил:

Е. В. Бурков

Группа:

3530901/90201

Преподаватель:

Н. В. Богач

Санкт-Петербург
2022

Содержание

1. Звуки и сигналы	4
1.1. Упражнение 1	4
1.2. Упражнение 2	8
1.3. Упражнение 3	10
1.4. Вывод	10
2. Гармоники	11
2.1. Упражнение 1	11
2.2. Упражнение 2	14
2.3. Упражнение 3	16
2.4. Упражнение 4	17
2.5. Упражнение 5	19
2.6. Вывод	21
3. Непериодические сигналы	22
3.1. Упражнение 1	22
3.2. Упражнение 2	25
3.3. Упражнение 3	26
3.4. Упражнение 4	27
3.5. Упражнение 5	28
3.6. Упражнение 6	29
3.7. Вывод	30
4. Шумы	31
4.1. Упражнение 1	31
4.2. Упражнение 2	34
4.3. Упражнение 3	36
4.4. Упражнение 4	38
4.5. Упражнение 5	40
4.6. Вывод	42
5. Автокорреляция	43
5.1. Упражнение 1	43
5.2. Упражнение 2	45
5.3. Упражнение 3	46
5.4. Упражнение 4	48
5.5. Вывод	52
6. Дискретное косинусное преобразование	53
6.1. Упражнение 1	53
6.2. Упражнение 2	54
6.3. Упражнение 3	56
6.4. Вывод	61
7. Дискретное преобразование Фурье	62
7.1. Упражнение 1	62
7.2. Вывод	62

8. Фильтрация и свертка	63
8.1. Упражнение 1	63
8.2. Упражнение 2	65
8.3. Упражнение 3	67
8.4. Вывод	69
9. Дифференциация и интеграция	70
9.1. Упражнение 1	70
9.2. Упражнение 2	72
9.3. Упражнение 3	75
9.4. Упражнение 4	77
9.5. Вывод	80
10. Сигналы и системы	81
10.1. Упражнение 1	81
10.2. Упражнение 2	84
10.3. Вывод	87
11. Модуляция и сэмплирование	88
11.1. Упражнение 1	88
11.2. Вывод	92
12. FSK	93
12.1. Теоритическая основа	93
12.2. Схема в GNU Radio	93
12.3. Тестирование	96
12.4. Вывод	97
Заключение	98
Перечень использованных источников	98

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, после читаем звуки в специальный Wave класс и вырезаем фрагмент.

```
1 if not os.path.exists('164718__bradovic__piano.wav'):
2     !wget https://github.com/wooftown/spbstu-telecom/raw/main/Content/164718
      __bradovic__piano.wav
3 wave = read_wave('164718__bradovic__piano.wav')
4 wave.make_audio()
5 wave = wave.segment(18.3,0.5)
6 wave.make_audio()
7 read_wave('164718__bradovic__piano.wav').plot()
8 wave.plot()
```

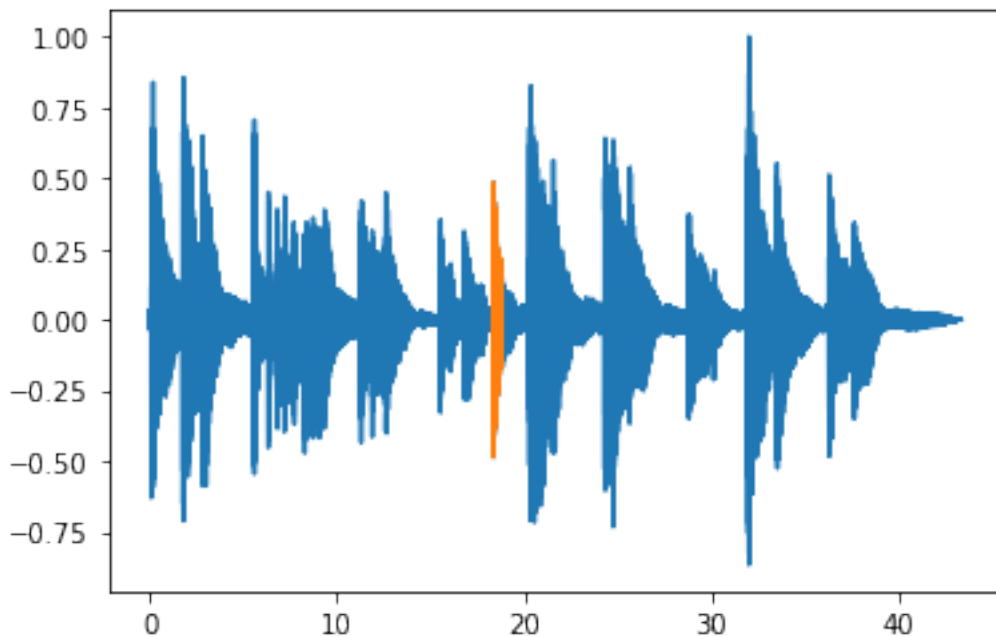


Рисунок 1.1. График фрагмента звука

При помощи метода `make_spectrum` вычислим спектр звука, и для удобства построим график.

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

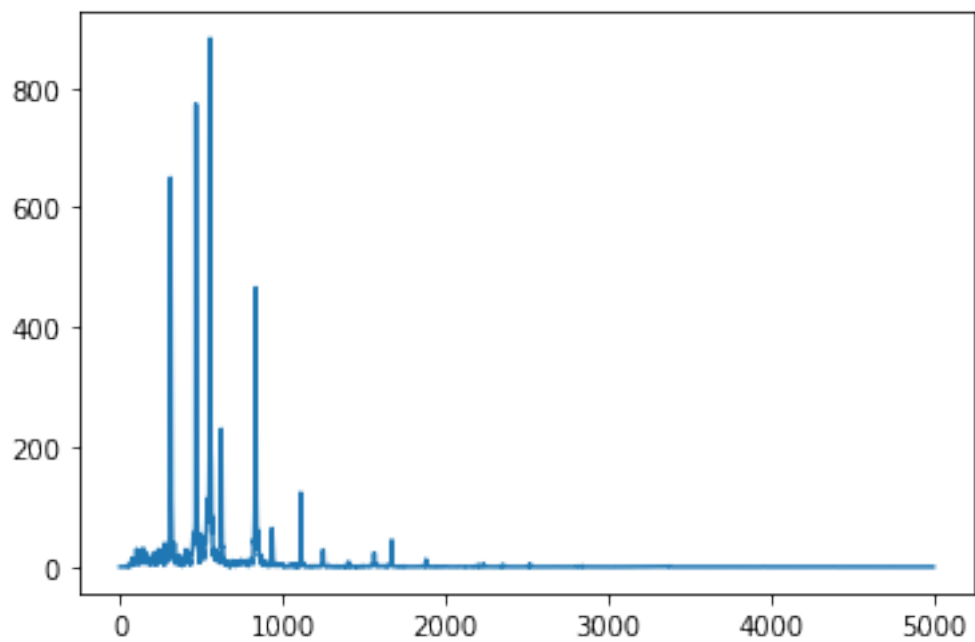


Рисунок 1.2. Спектр звука

Можно задать максимальную частоту для графика:

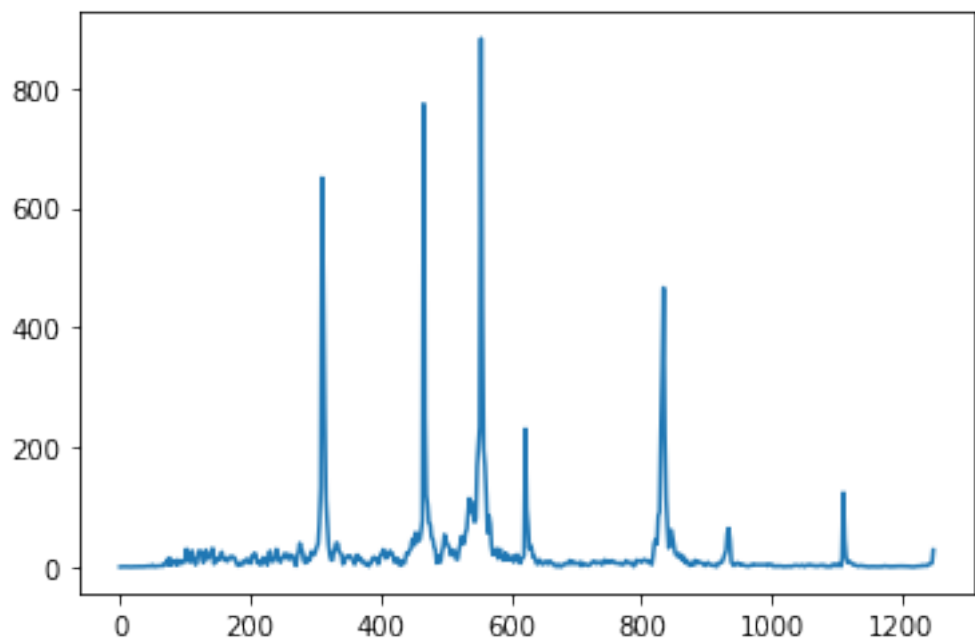


Рисунок 1.3. Спектр звука, частота меньше 1250 Гц

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

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

```
1 [(882.3500533686324, 554.0),
2  (772.8812508117549, 466.0),
3  (649.662314039115, 310.0),
4  (466.0353893832775, 834.0),
```

```

5 (455.88324114785496, 312.0),
6 (435.6400663619303, 556.0),
7 (328.0318016013845, 832.0),
8 (260.99365239113706, 314.0),
9 (251.02745102094198, 468.0),
10 (234.35739758178494, 552.0)]

```

Находим соответствие музыкальных нот и частот из пиков:

- 554.36 Гц - До-диез второй октавы
- 466.16 Гц - Ля-диез первой октавы
- 311.13 Гц - Ре-диез первой октавы
- 830.60 Гц - Соль-диез второй октавы

У До-диез второй октавы самая большая амплитуда, поэтому 554.36 Гц - доминирующая частота. Общая воспринимаемая высота звука зависит от основной частоты, тут о на 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%).

```

1 spectrum2 = wave.make_spectrum()
2 spectrum2.low_pass(540)
3 spectrum2.plot(high=1000)

```

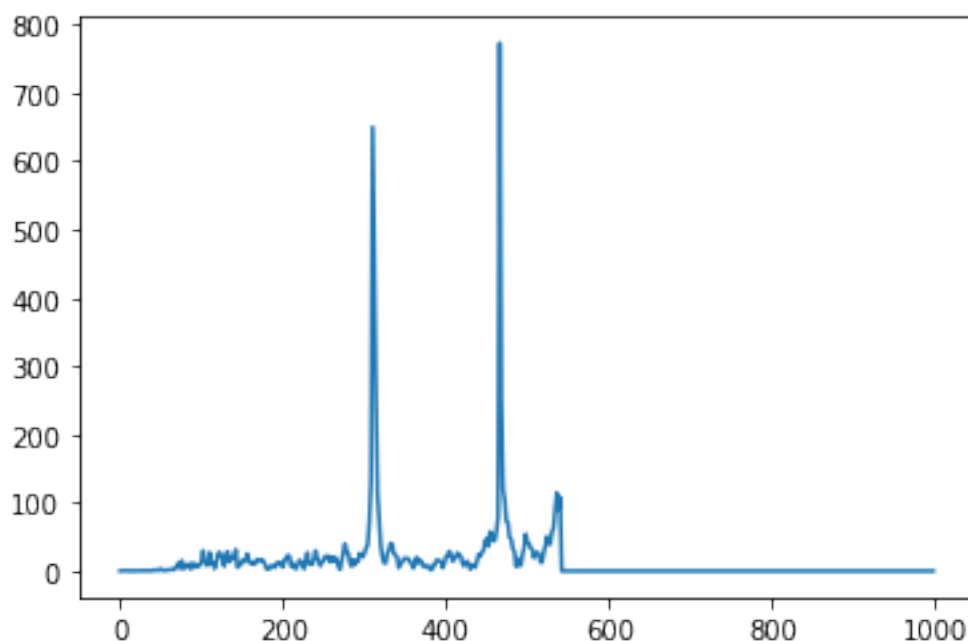


Рисунок 1.4. Спект с фильтром нижних частот

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

```

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

```

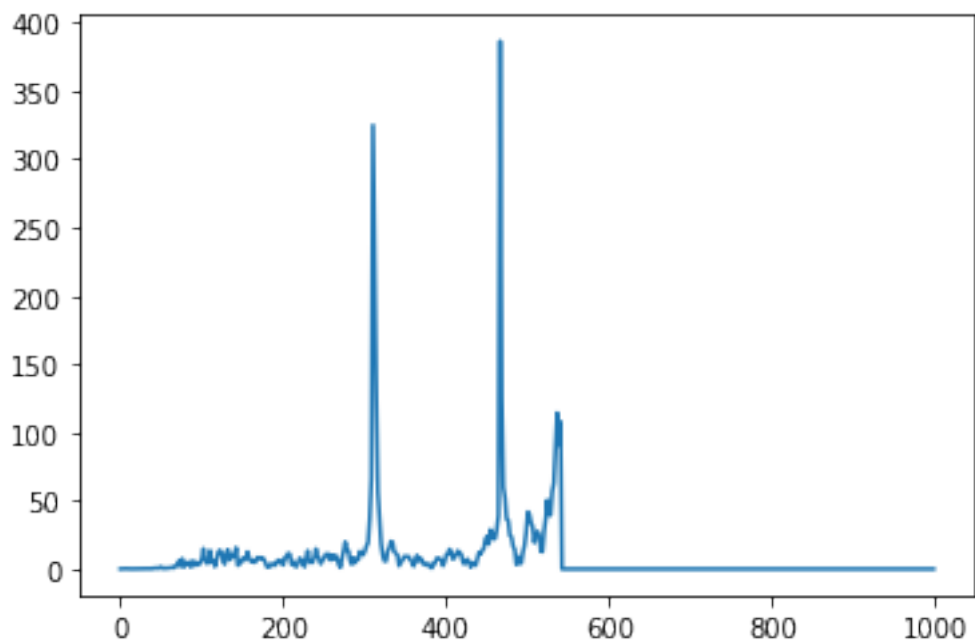


Рисунок 1.5. Спект с фильтром верхних частот

Уберём частоты между Ре и Ля.

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

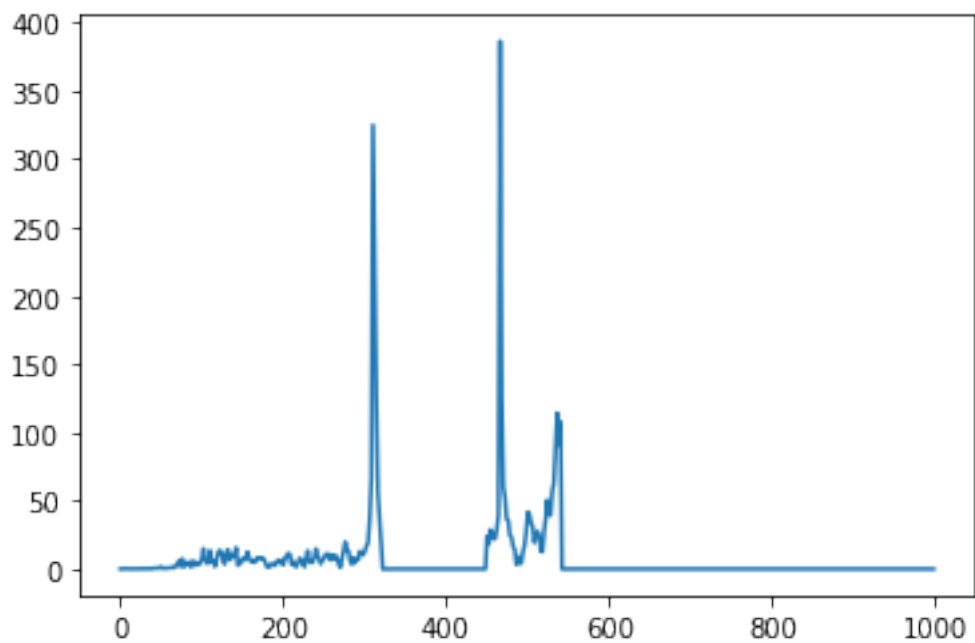


Рисунок 1.6. Получившийся спектр

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

```
1 wave = read_wave('164718__bradovic__piano.wav')
2 wave = wave.segment(18.3,0.5)
```

```

3 spectrum2 = wave.make_spectrum()
4 spectrum2.high_pass(500)
5 spectrum2.plot(high=1000)

```

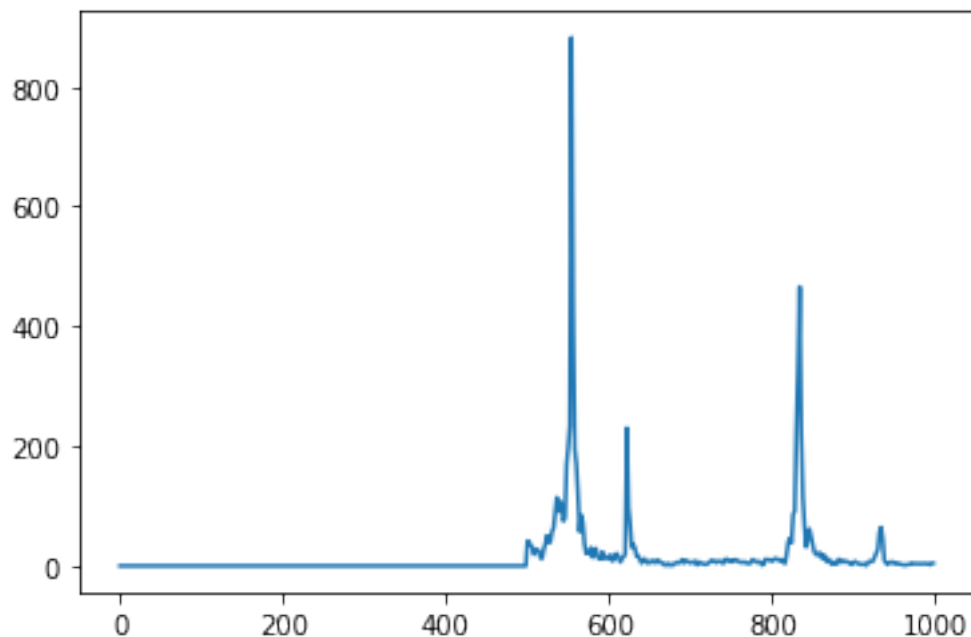


Рисунок 1.7. Отфильтрованные нижние компоненты

Отфильтровав нижние частоты звук стал более высоким.

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.

```

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

```

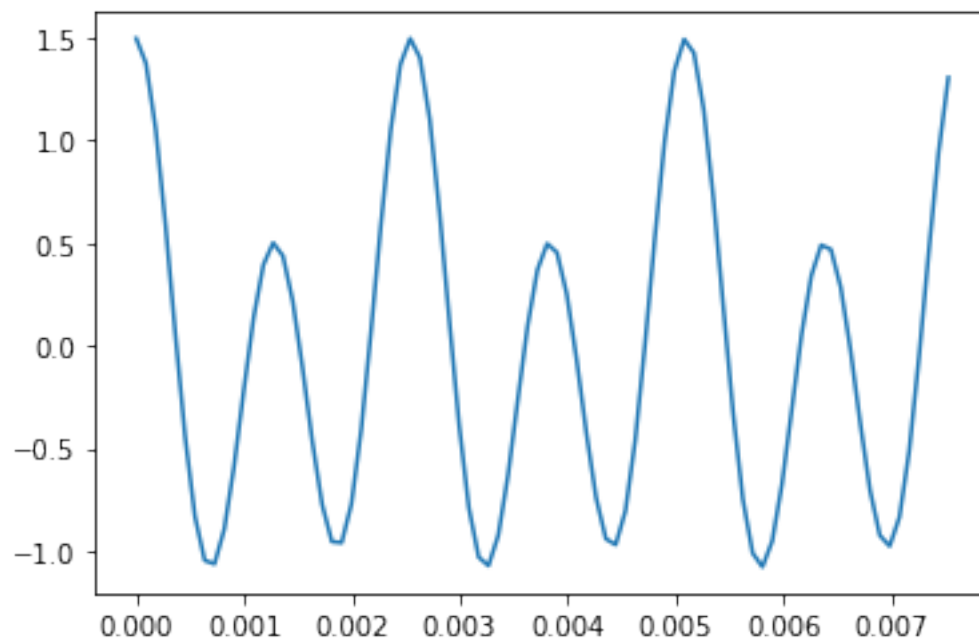



Рисунок 1.8. Суммированные сигналы

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

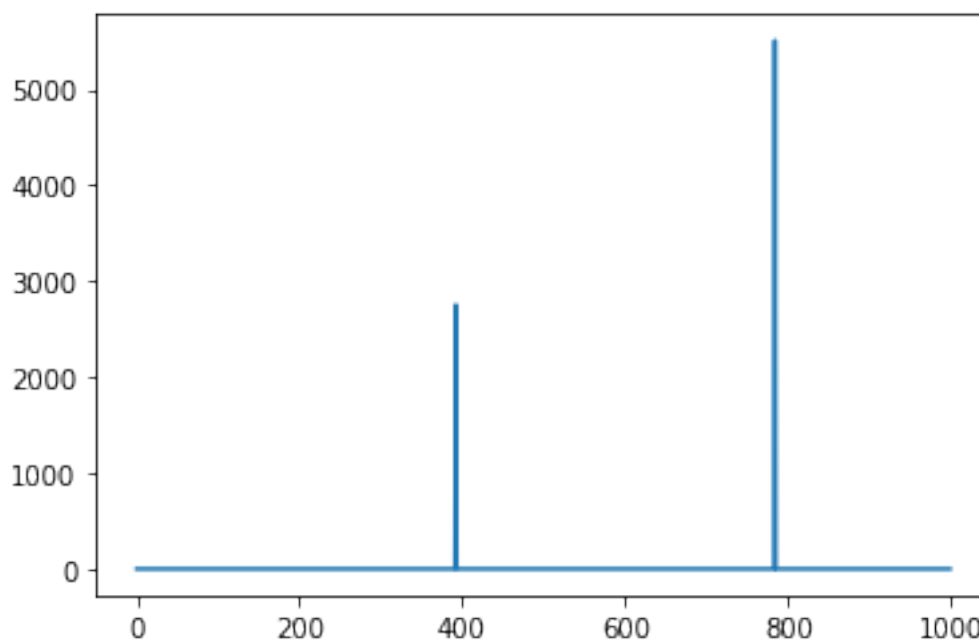


Рисунок 1.9. Спектр сигнала

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

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

На графике за 7мс не видно цикла.

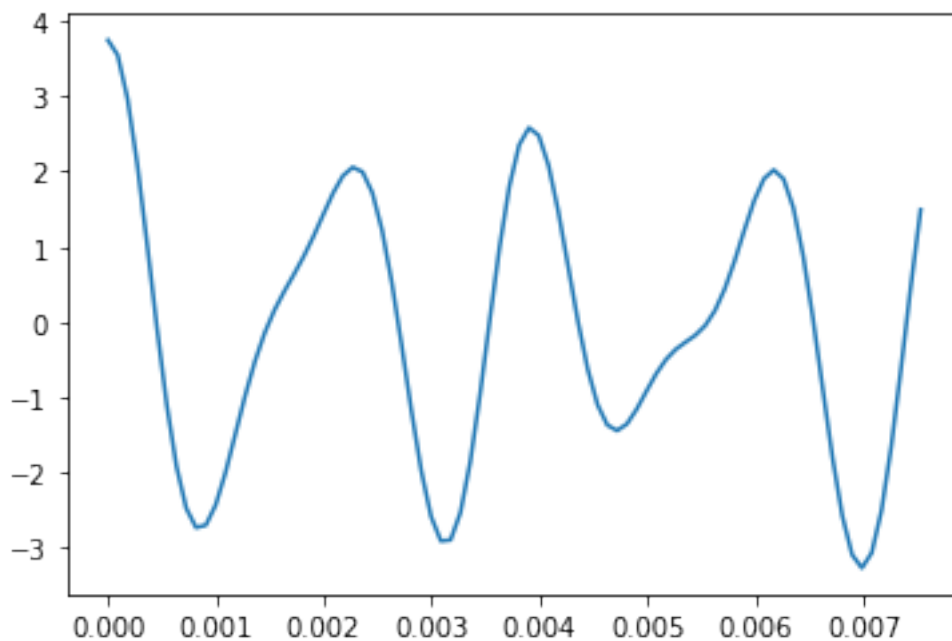


Рисунок 1.10. Получившийся сигнал

Действительно, звук очень неприятный.

1.3. Упражнение 3

Напишите функцию `stretch`, берущую `wave` и коэффициент изменения. Она должна ускорять или замедлять сигнал изменением `ts` и `framerate`.

`ts` - отвечает за моменты выборки сигнала `framerate` - число выборок в единицу времени.

Если умножим `ts` на `k`, то интервалы между моментами увеличатся в `k` раз.

Если `framerate` поделим на `k`, то будет меньшее число подвыборок.

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

1.4. Вывод

В ходе данной работы было выполнено знакомство с основными понятиями при работе со звуками и сигналами. При помощи библиотеки `thinkDSP` можно делать обширный круг взаимодействий с сигналами, как для их создания, 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 треугольными и прямоугольными сигналами?

Напишем класс пилообразного сигнала:

```
1 class SawtoothSignal(Sinusoid):
2
3     def evaluate(self, ts):
4         cycles = self.freq * ts + self.offset / (pi / 2)
5         frac, _ = np.modf(cycles)
6         u = unbias(frac)
7         high, low = abs(max(u)), abs(min(u))
8         ys = self.amp * u / max(high, low)
9         return ys
```

Проверим график:

```
1 saw_signal = SawtoothSignal(200)
2 saw_signal.plot()
```

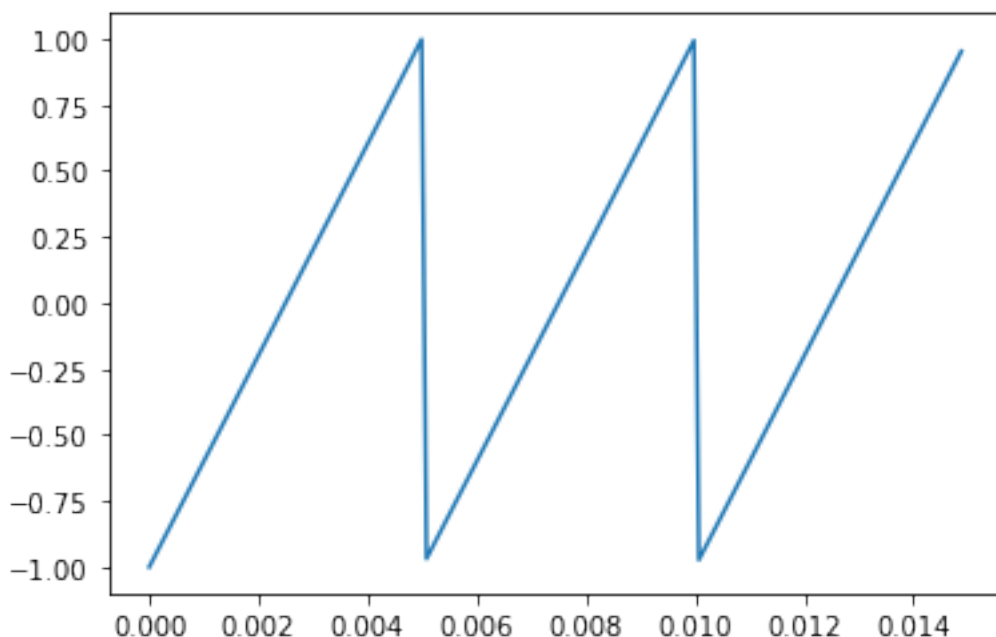


Рисунок 2.1. График пилообразного сигнала

Сделаем экземпляр класса `Wave` для построения спектра сигнала.

```
1 saw_spectrum = saw_signal.make_wave(duration = 0.5).make_spectrum()
2 saw_spectrum.plot()
```

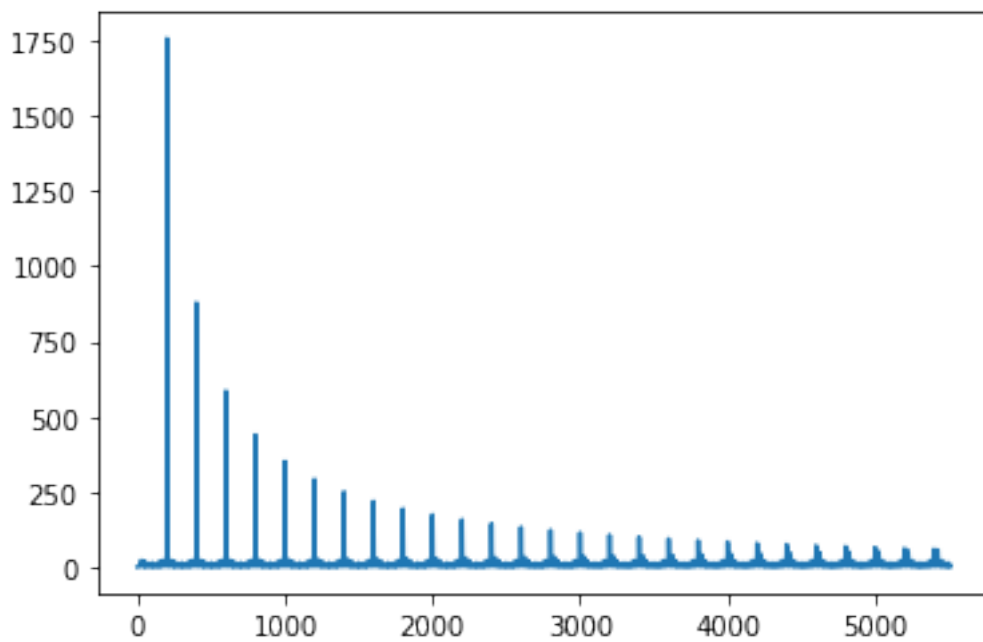


Рисунок 2.2. Спектр пилообразного сигнала

Добавим прямоугольный сигнал:

```
1 from thinkdsp import SquareSignal
2
3 squar_signal = SquareSignal(amp = 0.25)
4 squar_signal.plot()
```

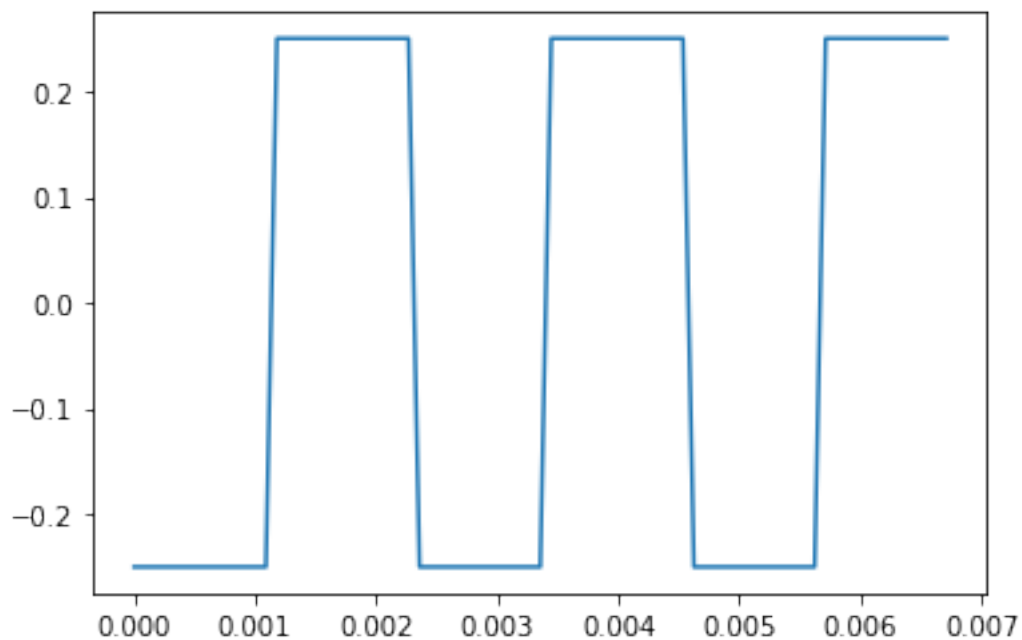


Рисунок 2.3. График прямоугольного сигнала

```
1 squar_spectrum = squar_signal.make_wave().make_spectrum()
2 squar_spectrum.plot()
```

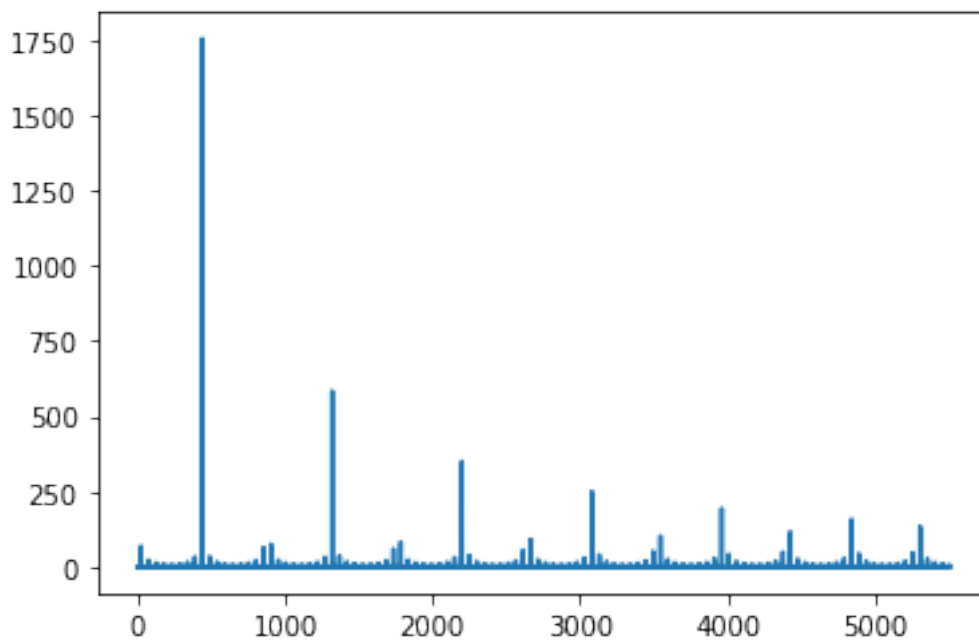


Рисунок 2.4. Спектр прямоугольного сигнала

Также добавим треугольный сигнал:

```
1 from thinkdsp import TriangleSignal
2
3 tri_signal = TriangleSignal(amp = 0.5)
4 tri_signal.plot()
```

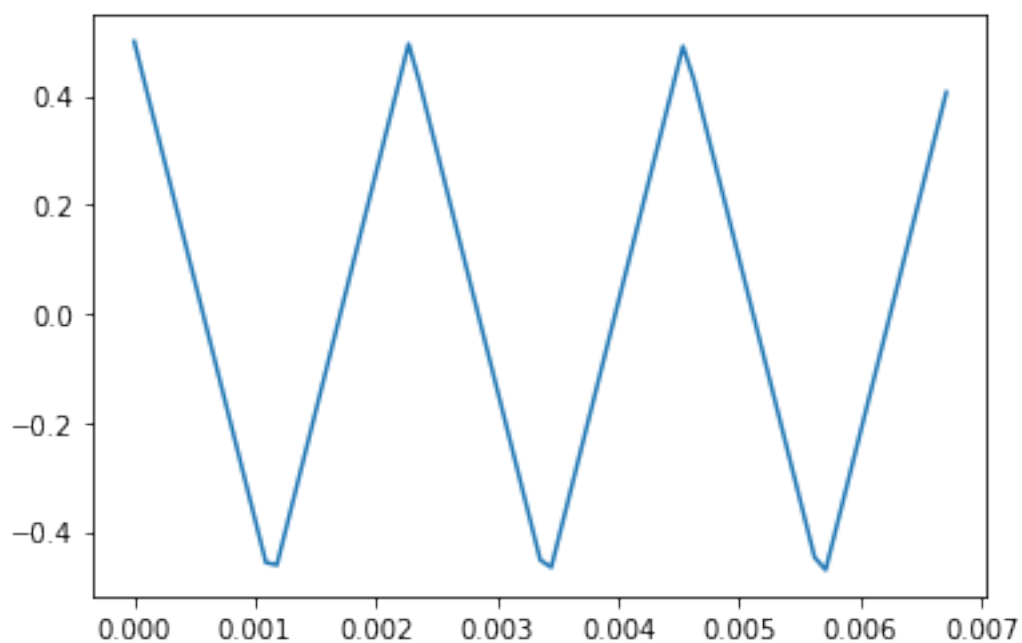


Рисунок 2.5. График треугольного сигнала

```
1 tri_spectrum = tri_signal.make_wave().make_spectrum()
2 tri_spectrum.plot()
```

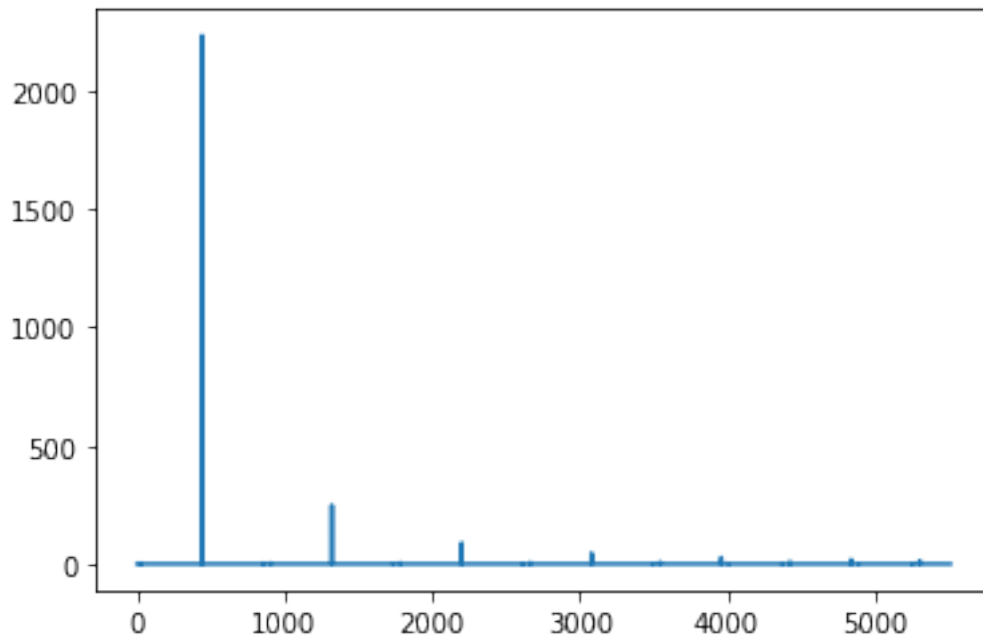


Рисунок 2.6. Спектр треугольного сигнала

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

2.2. Упражнение 2

Создайте прямоугольный сигнал 1500 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?

```

1 square_signal = SquareSignal(freq=1500)
2 square_wave = square_signal.make_wave(duration = 1, framerate = 10000)
3 square_wave.make_spectrum().plot()

```

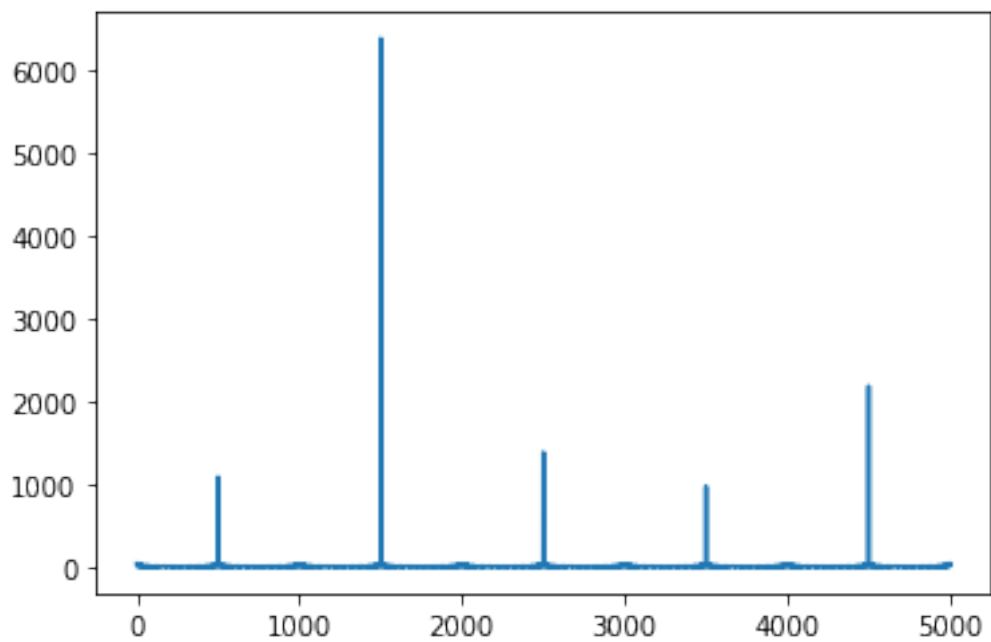


Рисунок 2.7. Спектр сигнала с биениями

По спекторграмме видим, что из-за выбранного фреймрейта 10000 у нас начинаются биения. Сигналы больших частот закольцовываются вокруг 5000Гц и 0Гц.

Когда мы слушаем получившийся звук, мы слышим основную частоту на 500Гц.

```

1 s1 = square_wave.make_spectrum()
2 s1.high_pass(1000)
3 s1.plot()

```

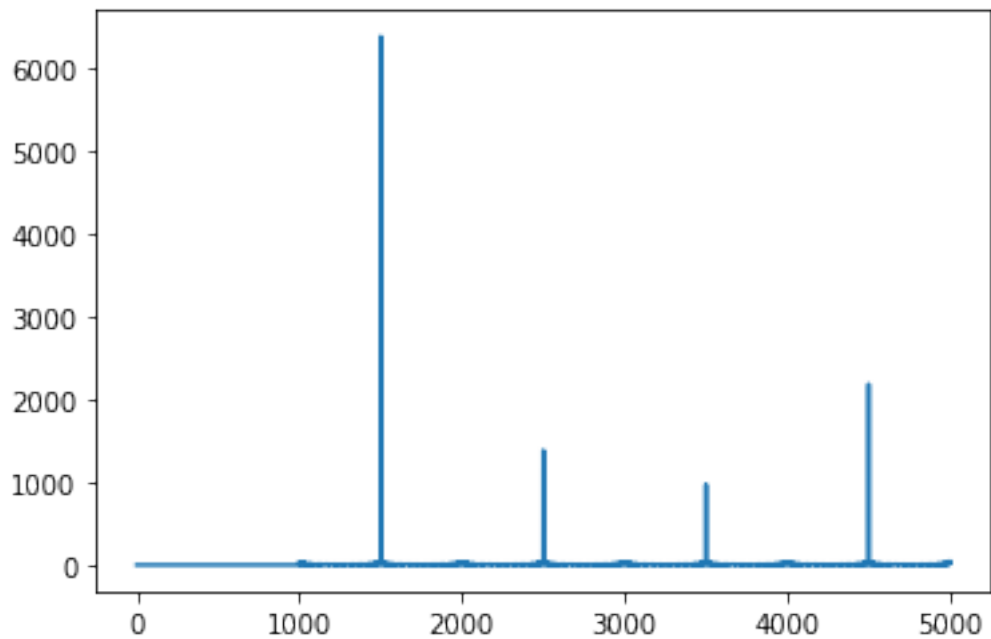


Рисунок 2.8. Спектр сигнала с фильтром

Звук отличается, значит действительно, мы слышим звук частотой 500Гц.

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?

```
1 trian_signal = TriangleSignal(freq=440)
2 trian_wave = trian_signal.make_wave(duration = 0.01)
3 trian_wave.plot()
```

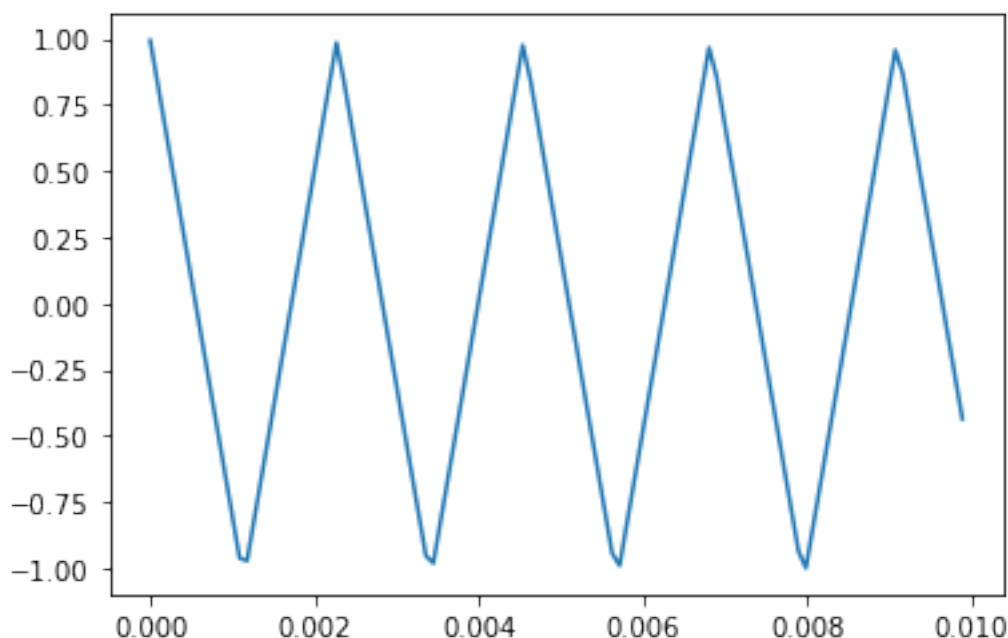


Рисунок 2.9. График сигнала

Проверим что лежит в 0 элементе чисел спекторграммы

```
1 trian_spectrum = trian_wave.make_spectrum()
2 trian_spectrum.hs[0]
3 (1.0436096431476471e-14+0j)
```

Видим комплексное число, с 0 мнимой частью. Сам элемент очень близок к нулю.

```
1 trian_spectrum.hs[0] = 100
2 trian_wave = trian_spectrum.make_wave()
3 trian_wave.plot()
```

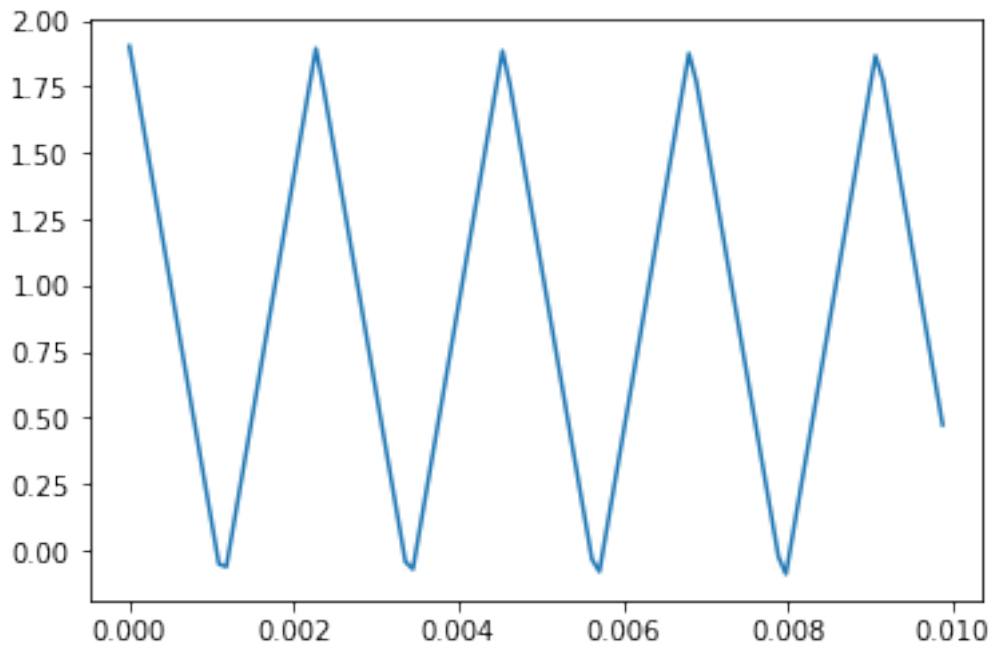



Рисунок 2.10. График сигнала с изменённым нулевым числом спекторграммы

Можно заметить, что сигнал сместился по вертикали вверх. Следовательно от первого элемента зависит смещение сигнала. Т.к. сначала элемент был близок к нулю, то нулевой элемент это сигнал без смещения.

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 (на самом деле я понял это после запуска программы из-за ошибок при делении на ноль).

```

1 def spectrum_divider(spectrum):
2     spectrum.hs[1:] /= spectrum.fs[1:]
3     spectrum.hs[0] = 0

1 if not os.path.exists('164718__bradovic__piano.wav'):
2     !wget https://github.com/wooftown/spbstu-telecom/raw/main/Content/164718
      __bradovic__piano.wav
3
4 wave = read_wave('164718__bradovic__piano.wav').segment(18.3, 0.5)
5 wave.make_audio()
6 wave.make_spectrum().plot(high = 5000)

```

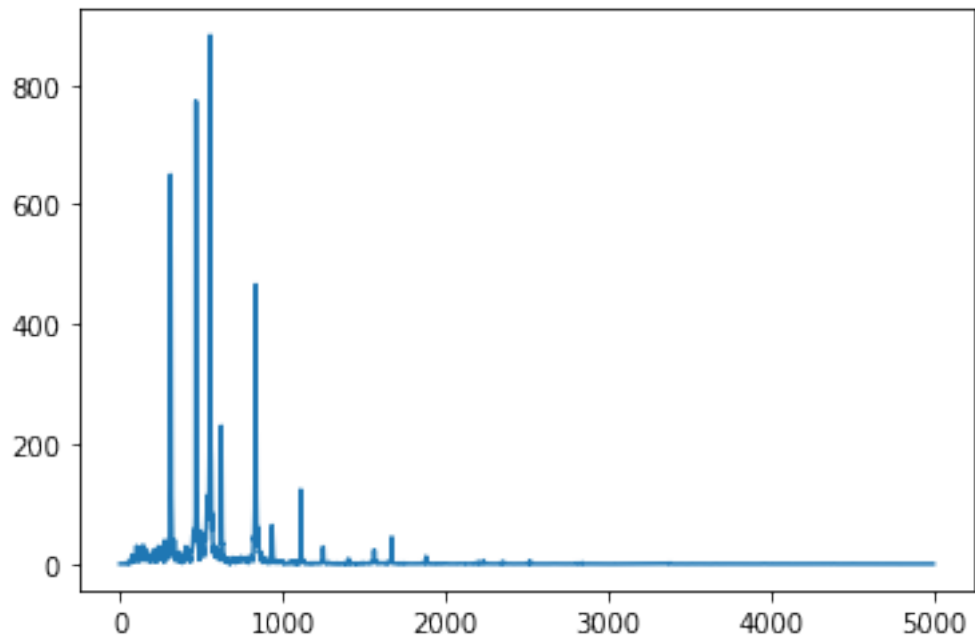


Рисунок 2.11. Спектр сигнала

```

1 sp = wave.make_spectrum()
2 spectrum_divider(sp)
3 sp.plot(high = 5000)

```

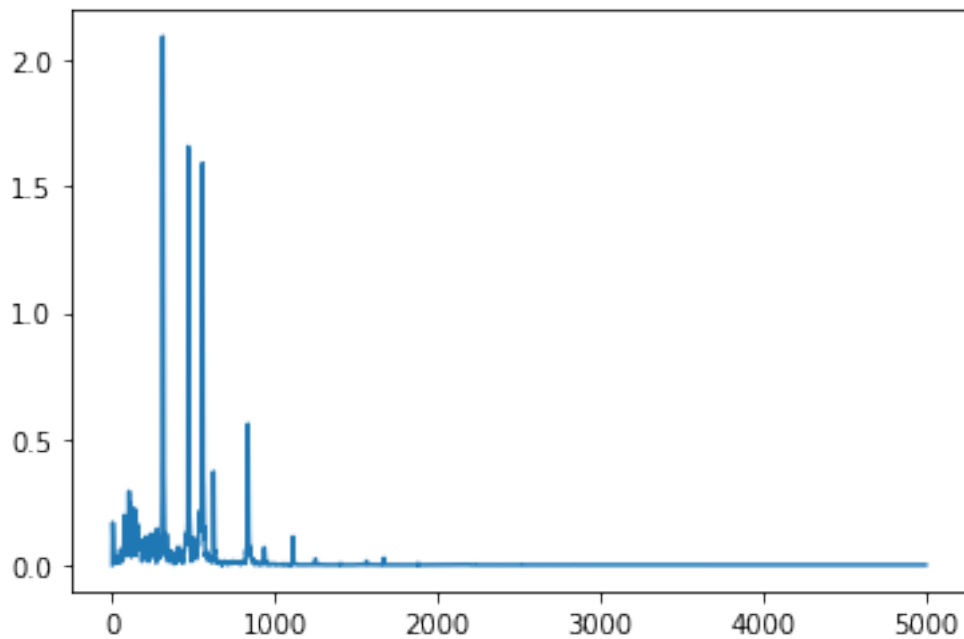


Рисунок 2.12. Спектр изменённого сигнала

Видим, что амплитуда очень поменялась, а частоты стоящие ближе к 0 Гц стали больше, чем следующие, что понятно. На выходе получилось, что полученный звук звучит более чисто, из-за фильтрации высоких частот.

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 можете начать 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.

```
1 saw_signal = SawtoothSignal(500)
2 saw_spectrum = saw_signal.make_wave().make_spectrum()
3 saw_spectrum.plot()
```

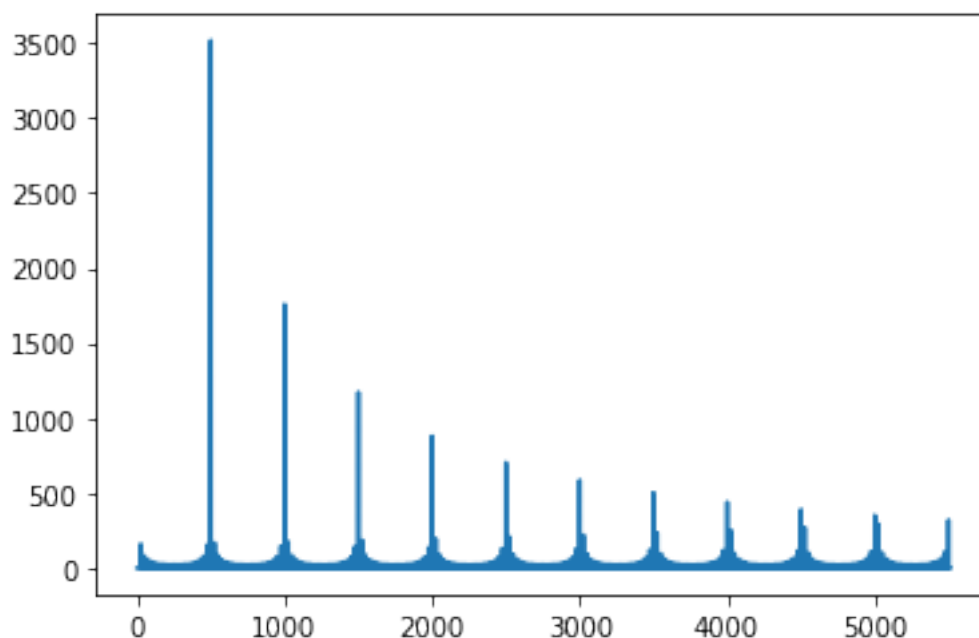


Рисунок 2.13. Спектр пилообразного сигнала

```
1 saw_spectrum1 = saw_signal.make_wave().make_spectrum()
2 spectrum_divider(saw_spectrum1)
3 saw_spectrum1.plot()
```

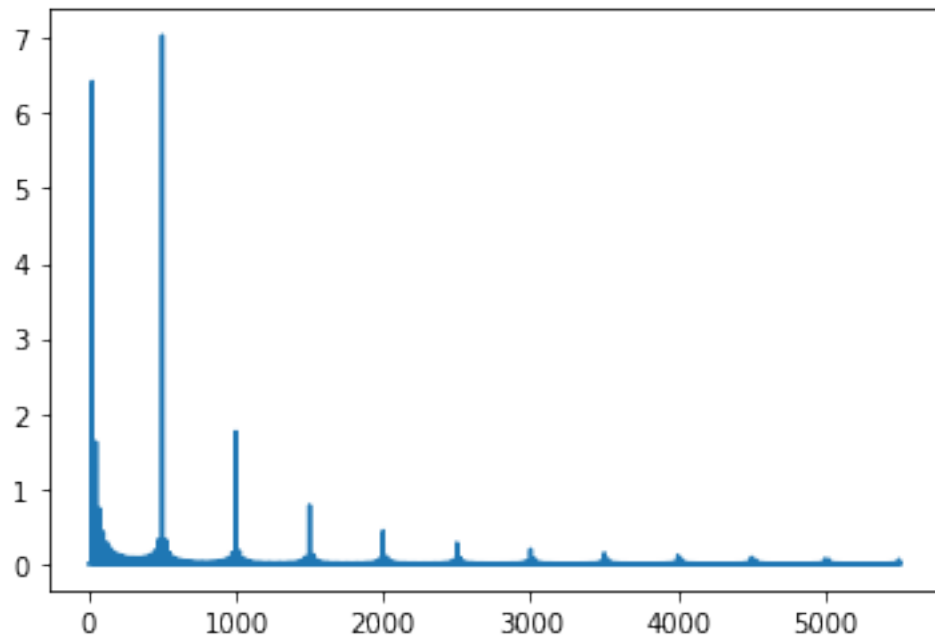


Рисунок 2.14. Спектр пилообразного сигнала

Тут получилось, что амплитуда у 0 слишком большая, исправим изменив параметры.

```

1 saw_signal = SawtoothSignal(freq=freq)
2 saw_wave = saw_signal.make_wave(duration=1, framerate=10000)
3 saw_spectrum = saw_wave.make_spectrum()
4 saw_spectrum.plot()
5 saw_spectrum1 = saw_wave.make_spectrum()
6 spectrum_divider(saw_spectrum1)
7 saw_spectrum1.plot()

```

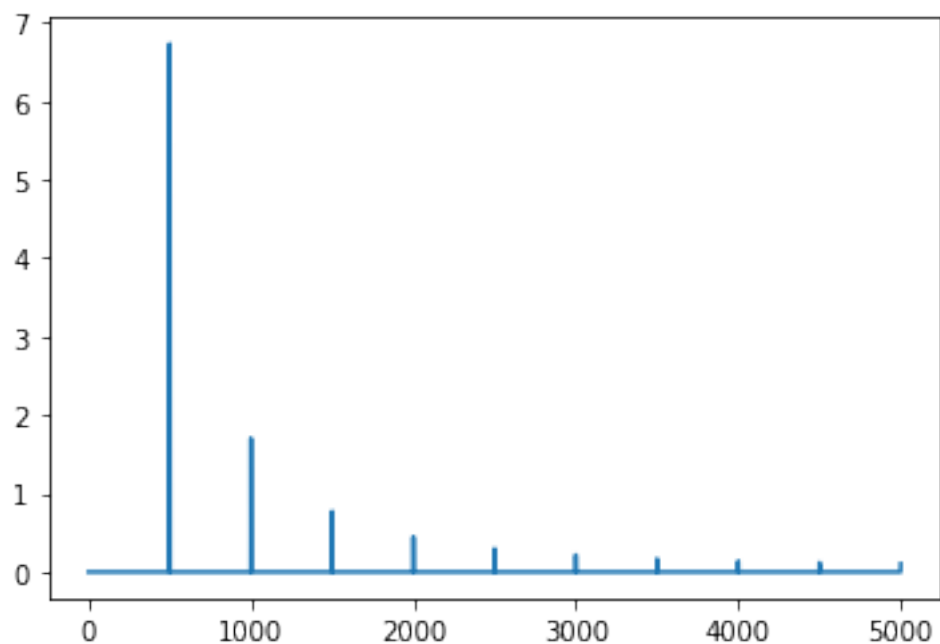


Рисунок 2.15. Спектр необходимого сигнала

Теперь нам интересно какой получился график:

```
1 saw_spectrum1.make_wave().segment(duration = 0.005).plot()
```

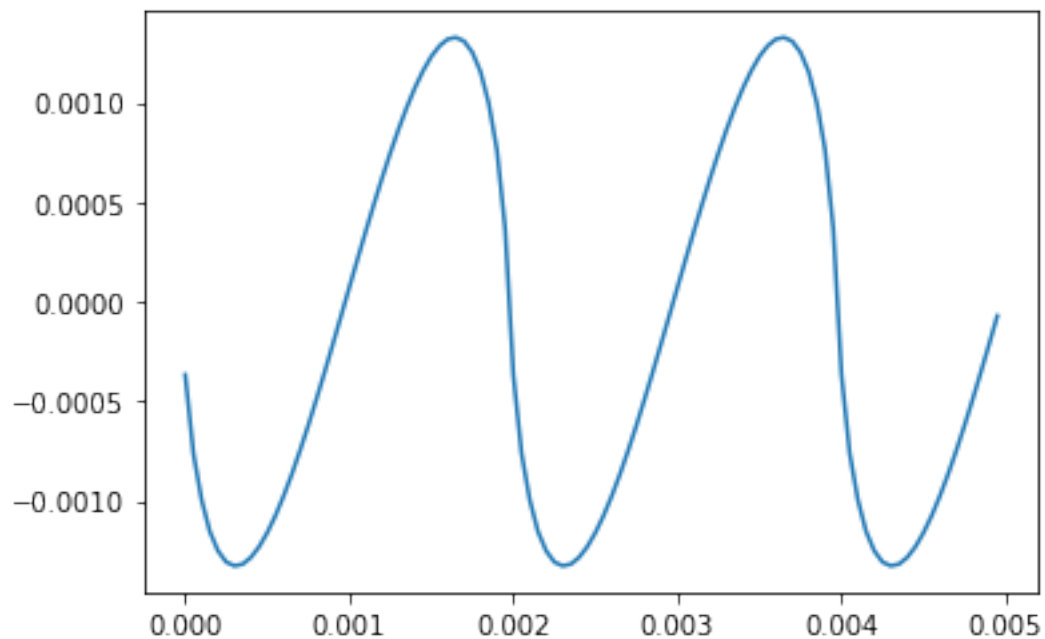


Рисунок 2.16. График необходимого сигнала

Сигнал немного напоминает синусоиду, но она как-будто немного наклонена.

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.

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

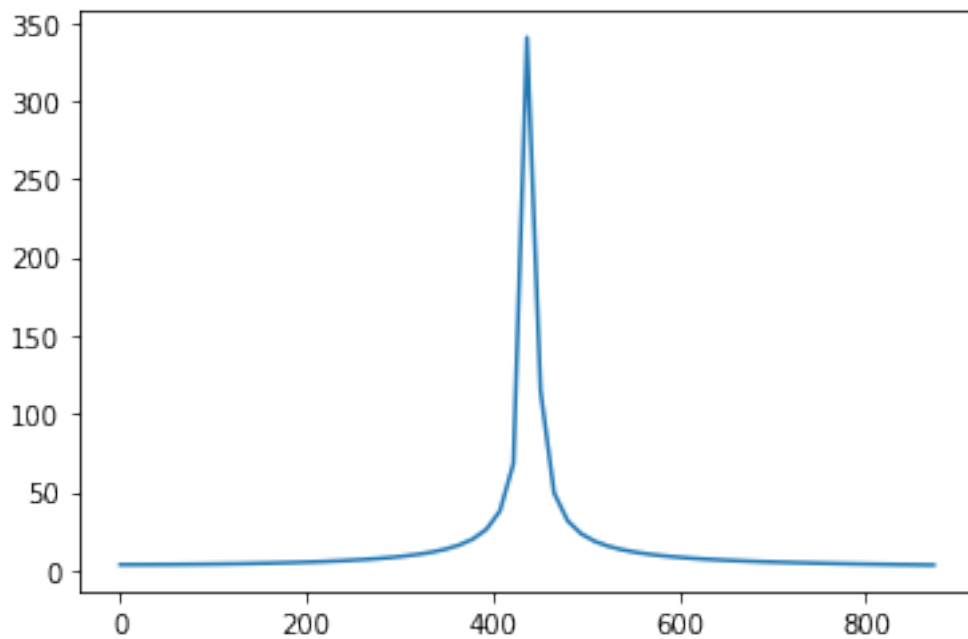


Рисунок 3.1. Рассматриваемый сигнал

Посмотрим как выглядит спектограмма с использованием окна Хэмминга:

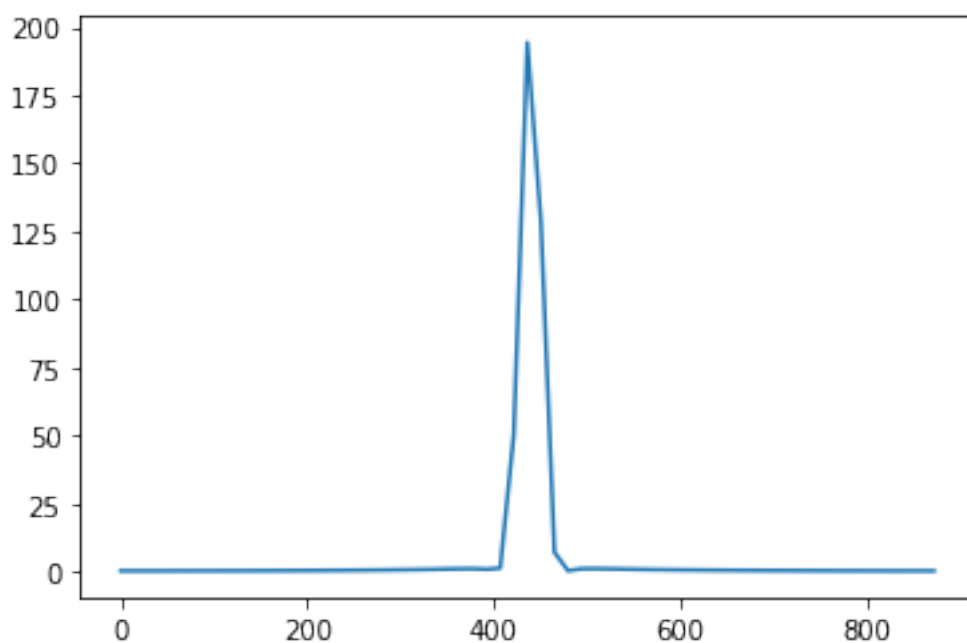


Рисунок 3.2. Сигнал с использованием окна Хэмминга

Посмотрим остальные окна

Окно Бартлетта:

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

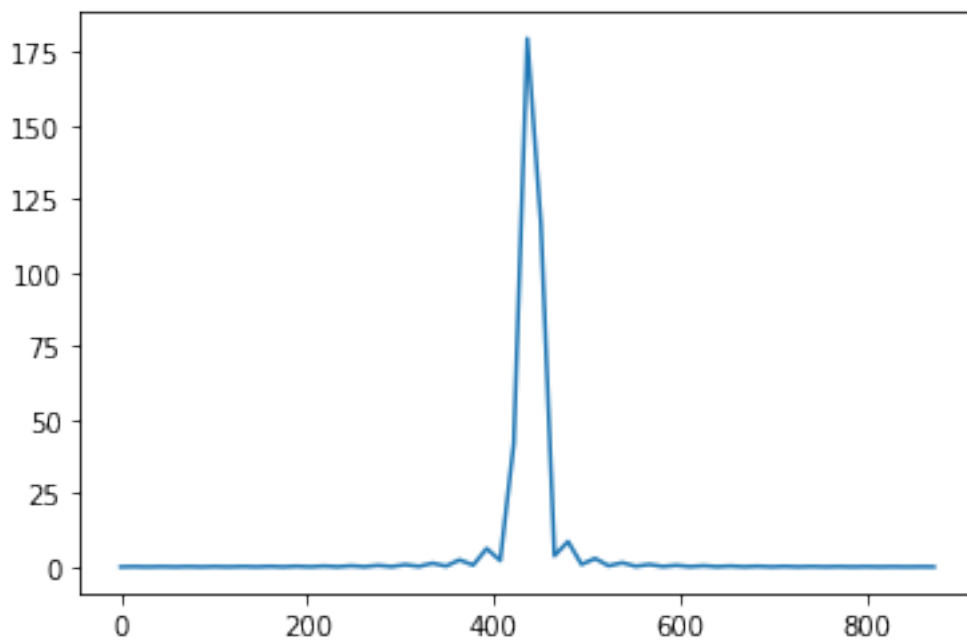


Рисунок 3.3. Сигнал с использованием окна Барлетта

Можно заметить, что низкие амплитуды стали ломанными линиями.

Окно Блэкмена:

```

1 wave = signal.make_wave(duration)
2 wave.ys *= np.blackman(len(wave.ys))
3 spectrum = wave.make_spectrum()
4 spectrum.plot(high=880)

```

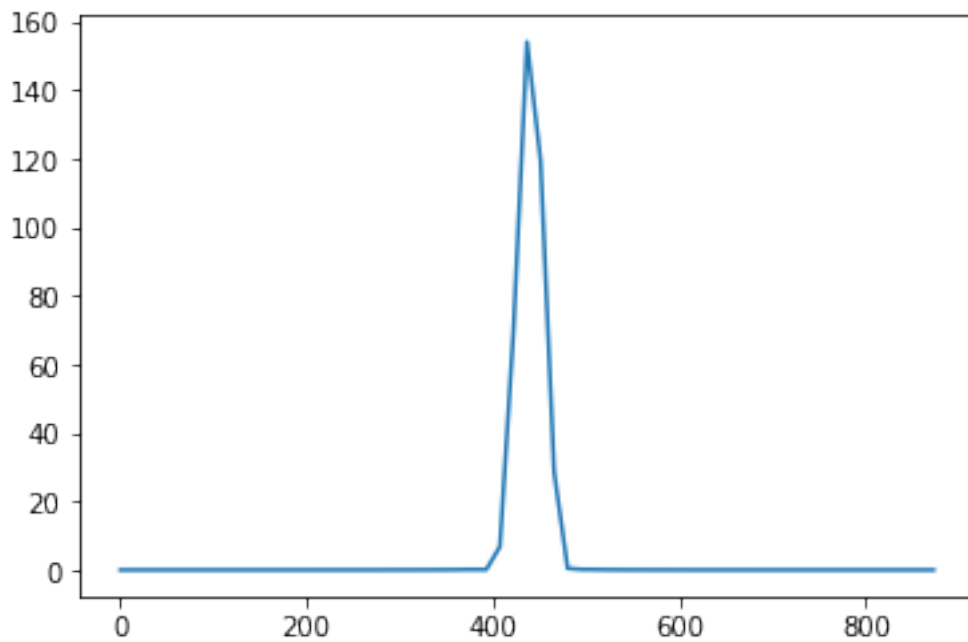


Рисунок 3.4. Сигнал с использованием окна Блэкмена

Видим, что утечка стала линейно переходить к нужной частоте.

Окно Хэннинга:

```

1 wave = signal.make_wave(duration)
2 wave.ys *= np.hanning(len(wave.ys))
3 spectrum = wave.make_spectrum()
4 spectrum.plot(high=880)

```

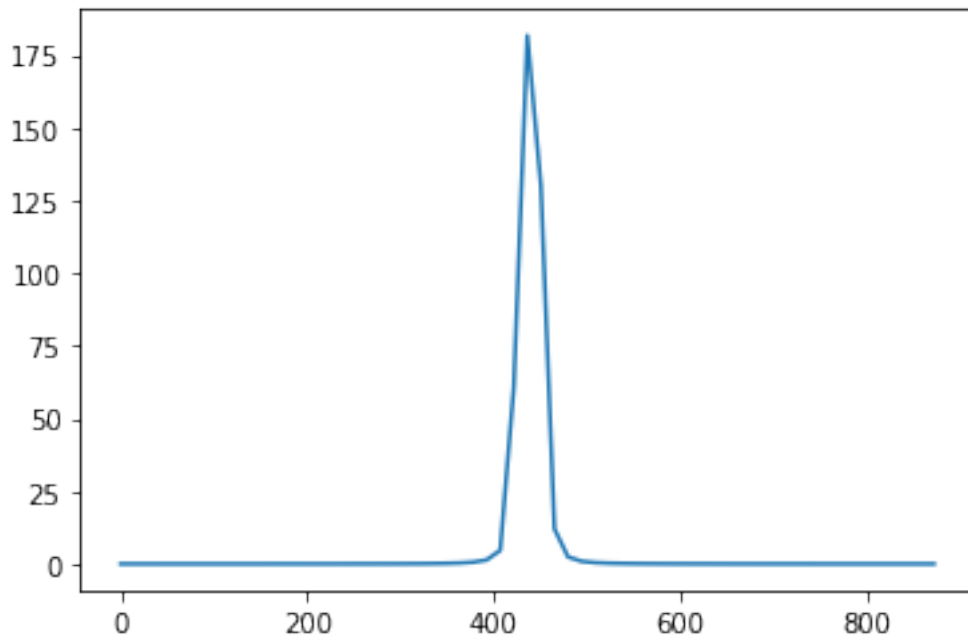



Рисунок 3.5. Сигнал с использованием окна Хэннинга

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 биения должен быть очевиден, а если сигнал внимательно прослушать, то биения можно и услышать.

```

1 class SawtoothChirp(Chirp):
2
3     def evaluate(self, ts):
4
5         freqs = np.linspace(self.start, self.end, len(ts))
6         dts = np.diff(ts, prepend=0)
7         dphis = 2 * np.pi * freqs * dts
8         phases = np.cumsum(dphis)
9         cycles = phases / (2 * np.pi)
10        frac, _ = np.modf(cycles)
11        ys = normalize(unbias(frac), self.amp)
12        return ys

```

Создадим звук:

```

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

```

Слышно, как частота постепенно увеличивается.

Выполним кратковременное преобразование Фурье и представим результат в виде спектограммы. По оси ОУ будет частота, по оси ОХ время.

```

1 sp = wave.make_spectrogram(256)
2 sp.plot()

```

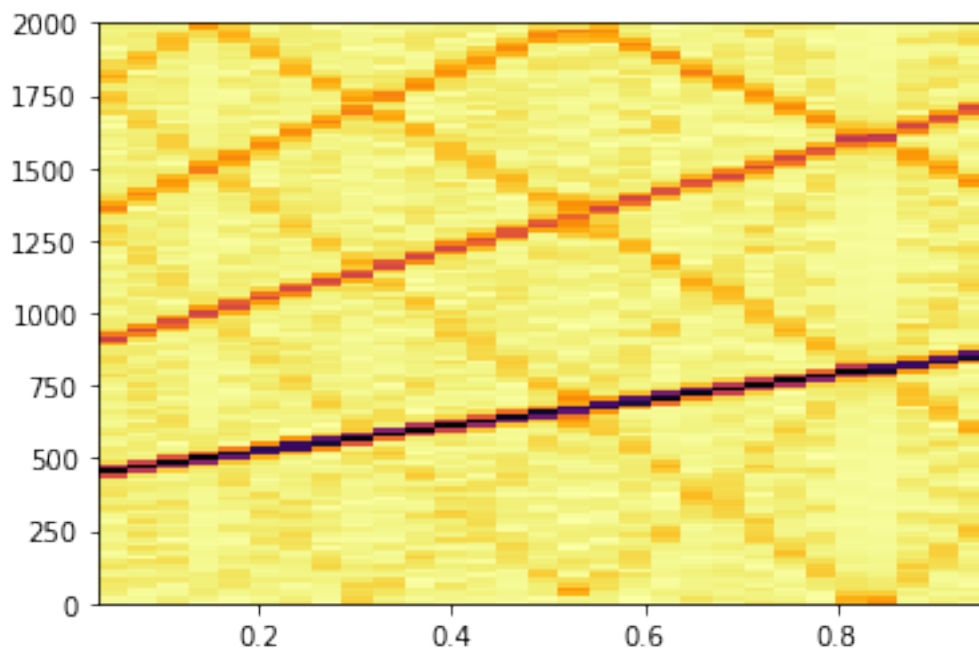


Рисунок 3.6. КПФ сигнала

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 кадров 20 кГц. Нарисуйте, каким примерно будет Spectrum. Затем распечатайте Spectrum и посмотрите, правы ли вы.

```

1 signal = SawtoothChirp(start=2500, end=3000)
2 wave = signal.make_wave(duration=1, framerate=20000)
3
4 wave.make_spectrum().plot()
```

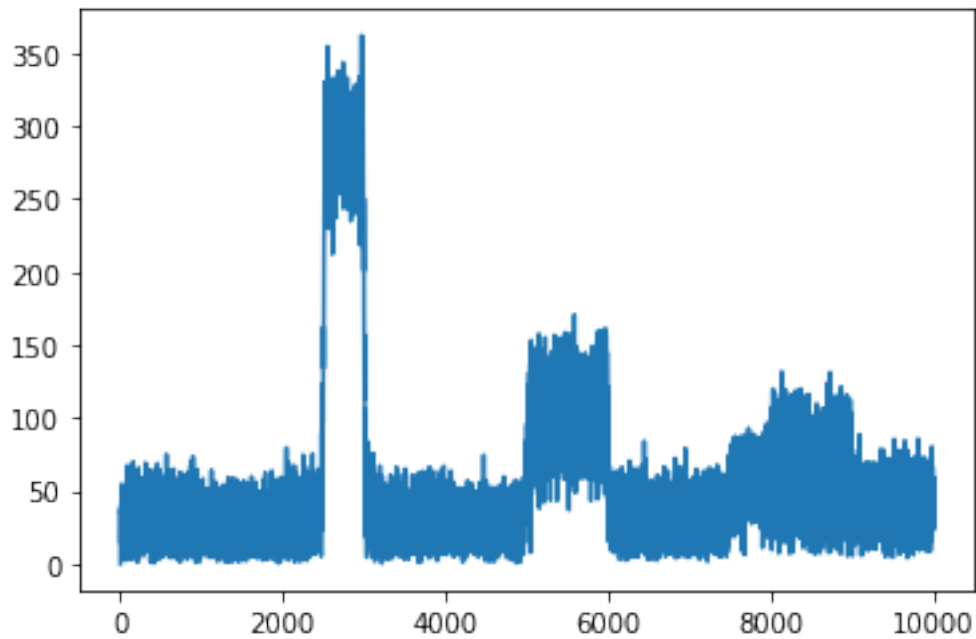


Рисунок 3.7. Спектр сигнала

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 диапазоне 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:

```

1 if not os.path.exists('72475__rockwehrmann__glissup02.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/72475
      __rockwehrmann__glissup02.wav
3
4
5 wave = read_wave('72475__rockwehrmann__glissup02.wav')
6
7 wave.make_spectrogram(512).plot(high=5000)

```

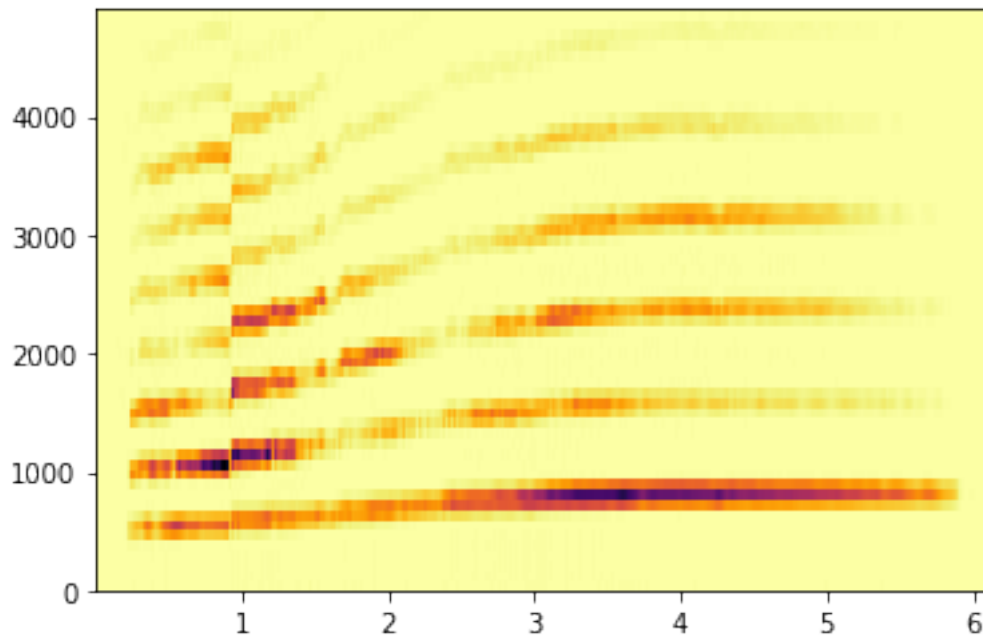


Рисунок 3.8. Спектрограмма сигнала

Видим, что спектрограмма очень похожа на наш чирп.

3.5. Упражнение 5

Тромбонист может играть глиссандо, выдвигая слайд тромбона и непрерывно дую. 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.

```

1 class TromboneGliss(Chirp):
2
3
4     def evaluate(self, ts):
5         lengths = np.linspace(1.0 / self.start, 1.0 / self.end, len(ts))
6         freqs = 1 / lengths
7         dts = np.diff(ts, prepend=0)
8         dphis = np.pi * 2 * freqs * dts
9         phases = np.cumsum(dphis)
10        ys = self.amp * np.cos(phases)
11        return ys

```

Соединим сигналы:

```

1 signal1 = TromboneGliss(262, 349)
2 wave1 = signal1.make_wave(duration=1)
3
4 signal2 = TromboneGliss(349, 262)
5 wave2 = signal2.make_wave(duration=1)
6
7 result = wave1 | wave2
8 sp = result.make_spectrogram(1024)

```

```
9 sp.plot(high=1000)
```

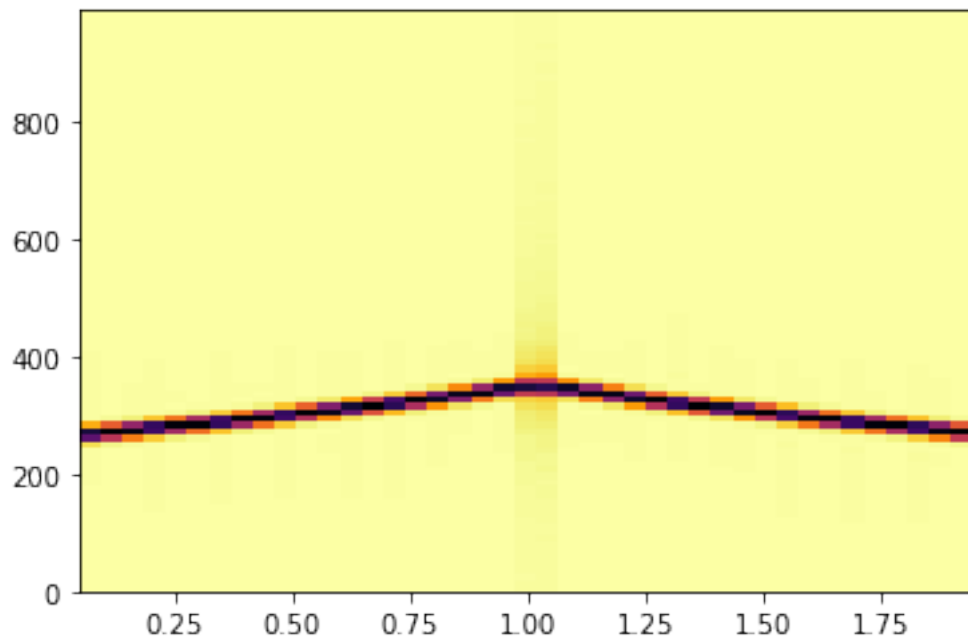


Рисунок 3.9. Спектрограмма сигнала

Отчётливо слышно, как идут друг за другом 2 части.

3.6. Упражнение 6

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

Возьмём гласные из репозитория учебника:

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

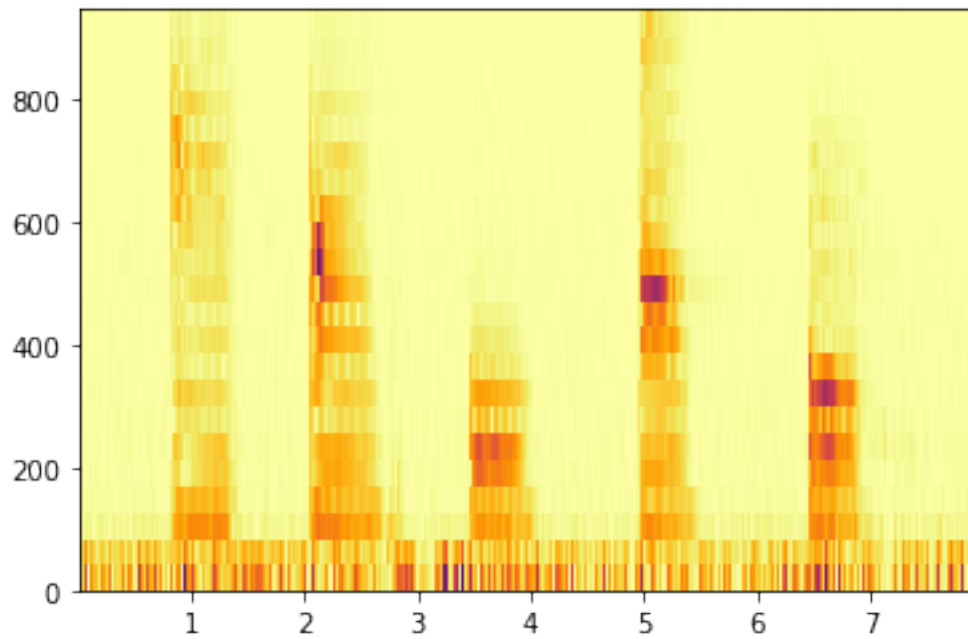


Рисунок 3.10. Спектрограмма гласных звуков

На спектограмме видны пики, они и будут нашими гласными.

3.7. Вывод

В этой работе были рассмотрены аperiodические сигналы, частотные компоненты которых изменяются во времени. Также в этой главе были рассмотрены спектрограммы - способ визуализации аperiodичных 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?

Возьмём звук свёрчков и вырежем сегмент. Мне кажется шум будет скоррелирован.

```
1 wave = read_wave('22604__martypinso__dmp010037-crickets-texas.wav')
2 segment = wave.segment(start=10, duration=1.0)
3 segment.plot()
```

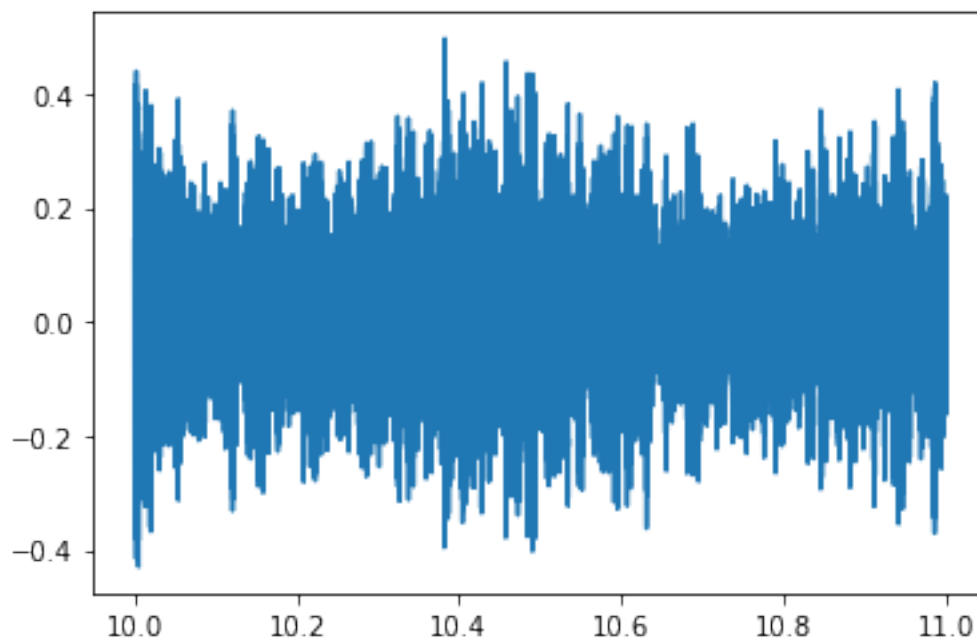


Рисунок 4.1. График сигнала

Для определения характеристик шума используем код из пособия по построению графика спектра в логорифмическом масштабе.

```
1 from thinkdsp import decorate
2 spectrum.plot_power()
3
4 loglog = dict(xscale='log', yscale='log')
5 decorate(xlabel='Frequency (Hz)',
6          ylabel='Power',
7          **loglog)
```

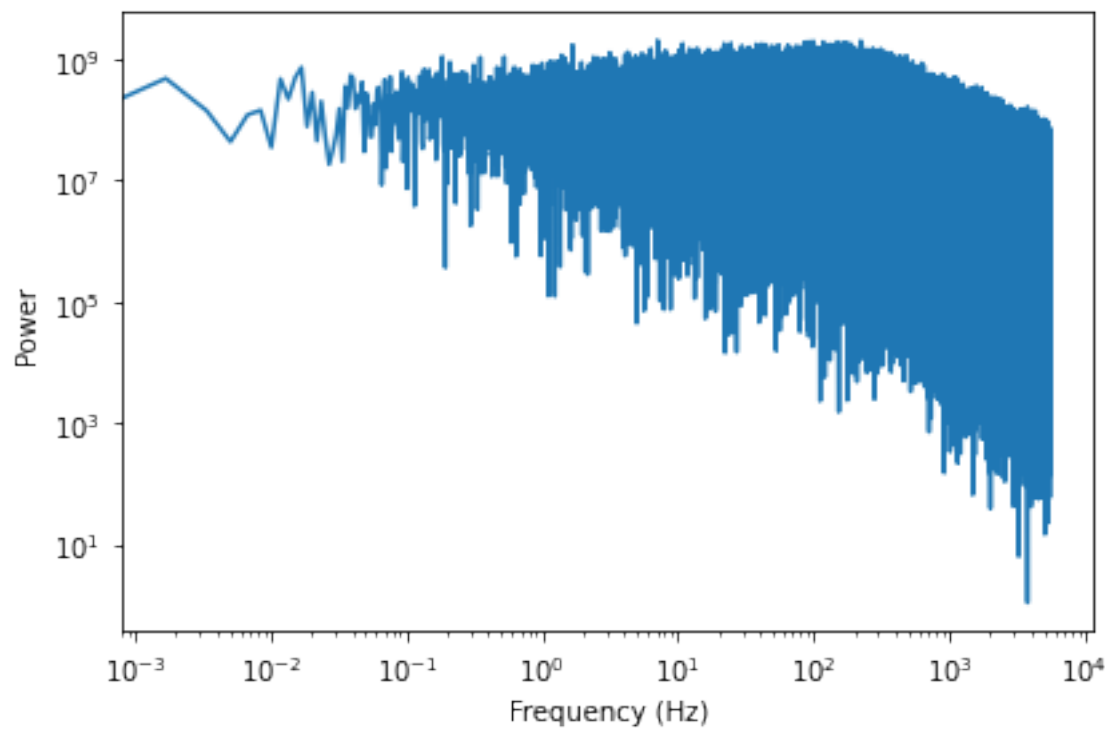


Рисунок 4.2. Спектр в логорифмическом масштабе

Давайте возьмём следующий сегмент для наглядности.

```

1 segment1 = wave.segment(start=11, duration=1.0)
2 spectrum1 = segment1.make_spectrum()
3 spectrum.plot_power()
4 spectrum1.plot_power()
5
6
7 loglog = dict(xscale='log', yscale='log')
8 decorate(xlabel='Frequency (Hz)',
9          ylabel='Power',
10          **loglog)

```

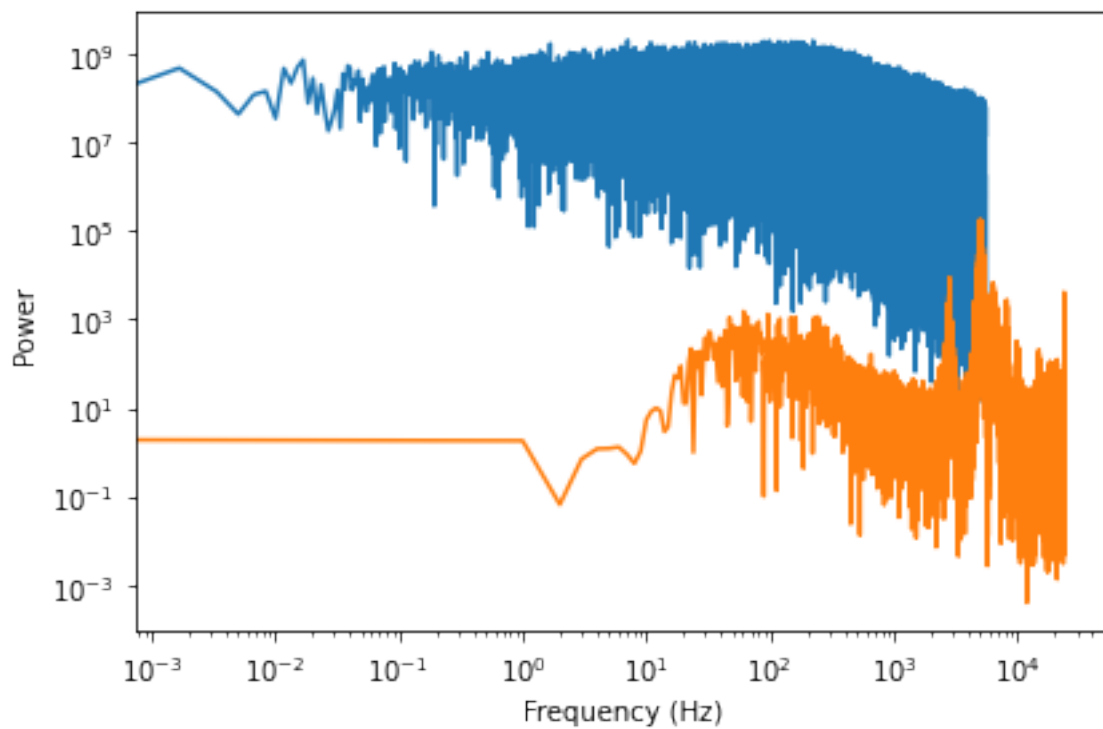



Рисунок 4.3. Сравнение спектров в логорифмическом масштабе

```

1 noise = UncorrelatedGaussianNoise()
2 wave2 = noise.make_wave(duration=1)
3 spectrum2 = wave2.make_spectrum()
4 spectrum.plot_power()
5 spectrum2.plot_power()
6
7
8 loglog = dict(xscale='log', yscale='log')
9 decorate(xlabel='Frequency (Hz)',
10         ylabel='Power',
11         **loglog)

```

Оранжевым обозначен Гауссов шум.

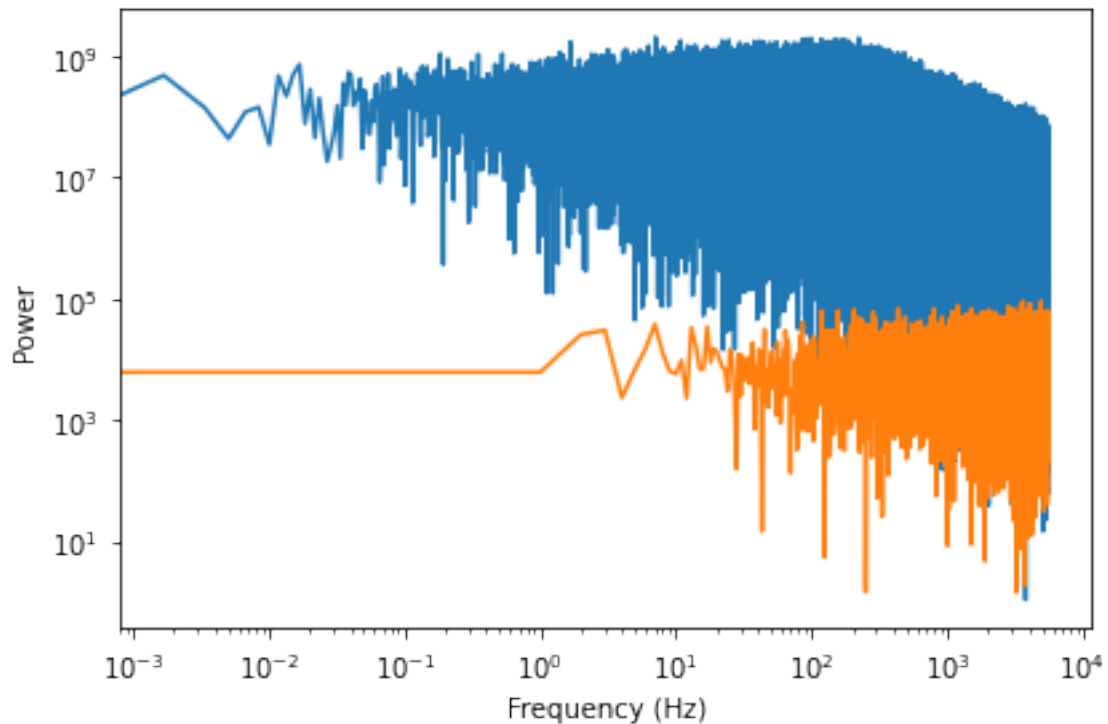


Рисунок 4.4. Сравнение нашего звука с Гауссовым шумом

4.2. Упражнение 2

Реализуйте метод Бартлетта[2] и используйте его для оценки спектра мощности шумового сигнала. Подсказка: посмотрите на реализацию `make_spectrogram`.

Исходя из статьи на википедии надо поделить данные на сегменты, потом для каждого сегмента сделать разложение Фурье, вычислить сумму квадратов и разделить на количество элементов и взять от этого всего корень.

Посмотрим с помощью чего и как создаются спектограммы:

```

1  def make_spectrogram(self, seg_length, win_flag=True):
2      """Computes the spectrogram of the wave.
3      seg_length: number of samples in each segment
4      win_flag: boolean, whether to apply hamming window to each segment
5      returns: Spectrogram
6      """
7      if win_flag:
8          window = np.hamming(seg_length)
9          i, j = 0, seg_length
10         step = int(seg_length // 2)
11
12         # map from time to Spectrum
13         spec_map = {}
14
15         while j < len(self.ys):
16             segment = self.slice(i, j)
17             if win_flag:
18                 segment.window(window)
19
20             # the nominal time for this segment is the midpoint
21             t = (segment.start + segment.end) / 2
22             spec_map[t] = segment.make_spectrum()

```

```

23         i += step
24         j += step
25
26
27     return Spectrogram(spec_map, seg_length)

```

```

1     """Initializes a spectrum.
2     hs: array of amplitudes (real or complex)
3     fs: array of frequencies
4     framerate: frames per second
5     full: boolean to indicate full or real FFT
6     """
7     self.hs = np.asanyarray(hs)
8     self.fs = np.asanyarray(fs)
9     self.framerate = framerate
10    self.full = full

```

Для создания своей функции надо изменять словарь:

```

1 def make_barlett(wave, N, flag=True):
2     spectrogram = wave.make_spectrogram(N, flag)
3     spec_mac = spectrogram.spec_map.values()
4
5     powers = []
6     for spectrum in spec_mac:
7         powers.append(spectrum.power)
8
9     hs = np.sqrt(sum(powers)/ len(powers))
10    fs = next(iter(spec_mac)).fs
11
12    return Spectrogram(hs, fs, wave.framerate)

```

```

1 barlett = make_barlett(segment, 1024)
2 barlett.plot_power()
3
4 decorate(xlabel='Frequency (Hz)',
5          ylabel='Power',
6          **loglog)

```

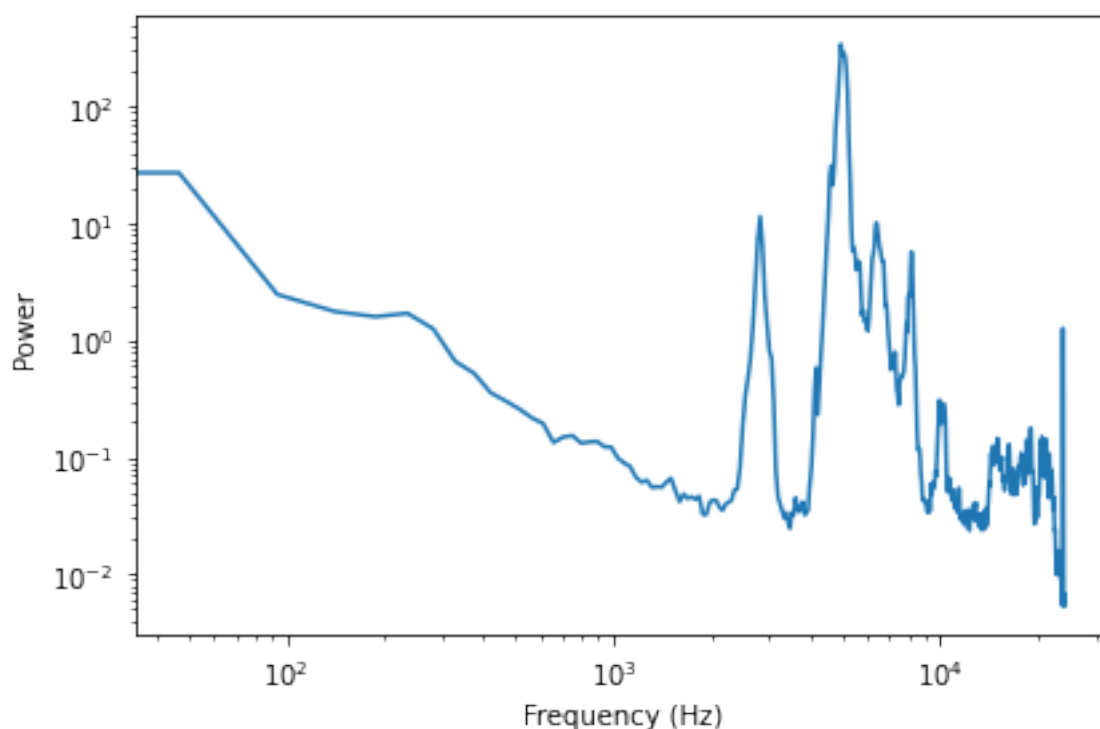


Рисунок 4.5. Результат

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?

```

1 if not os.path.exists('market-price.csv'):
2     !wget https://github.com/wooftown/spbstu-telecom/raw/main/Content/market-
      price.csv

1 import csv
2
3 worth = []
4
5 with open('market-price.csv') as File:
6     reader = csv.reader(File, delimiter=',', quotechar='\"',
7                          quoting=csv.QUOTE_MINIMAL)
8     worth = [row[1] for row in reader]
9
10 worth = worth[1:]
11 days = a=np.arange(0, len(worth))

1 wave = Wave(worth, days, 1)
2 wave.plot()
3 decorate(xlabel='Time (days)')

```

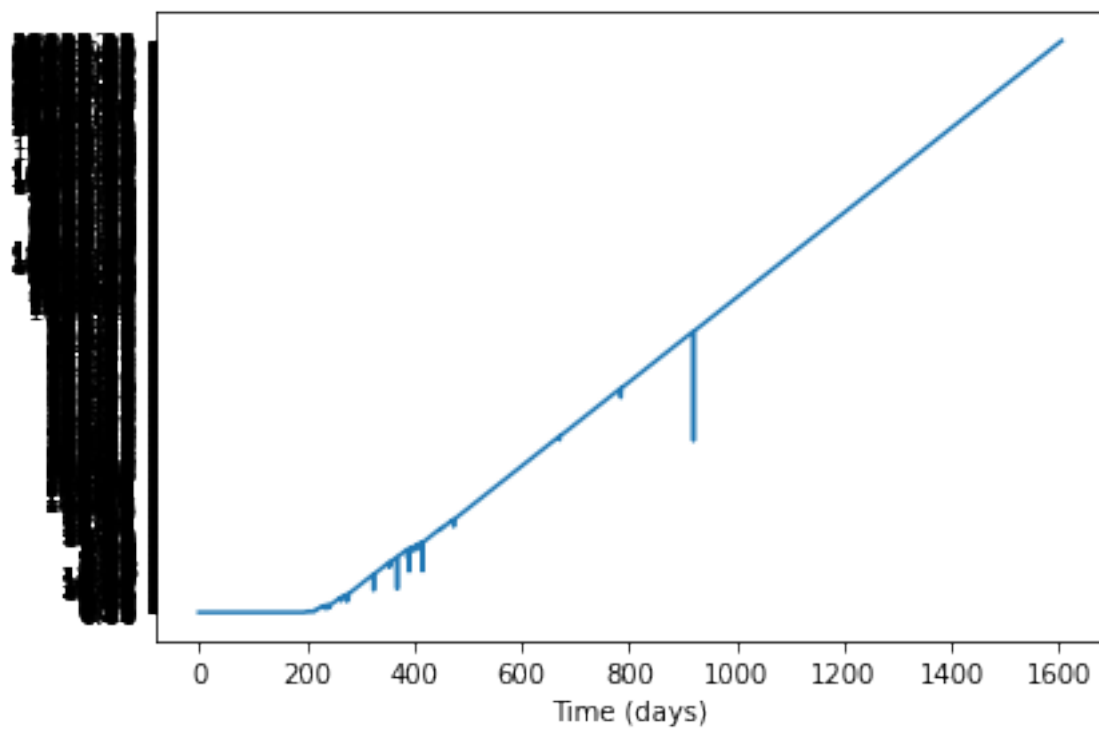


Рисунок 4.6. График цен BitCoin

График получился не очень красивым из-за того, что слишком много данных, можно приблизить и посмотреть какой-то один сегмент. Давайте перейдём к спектограмме.

```
1 spectrum = wave.make_spectrum()  
2 spectrum.plot_power()  
3 decorate(xlabel='Frequency (1/days)',  
4         ylabel='Power',  
5         **loglog)
```

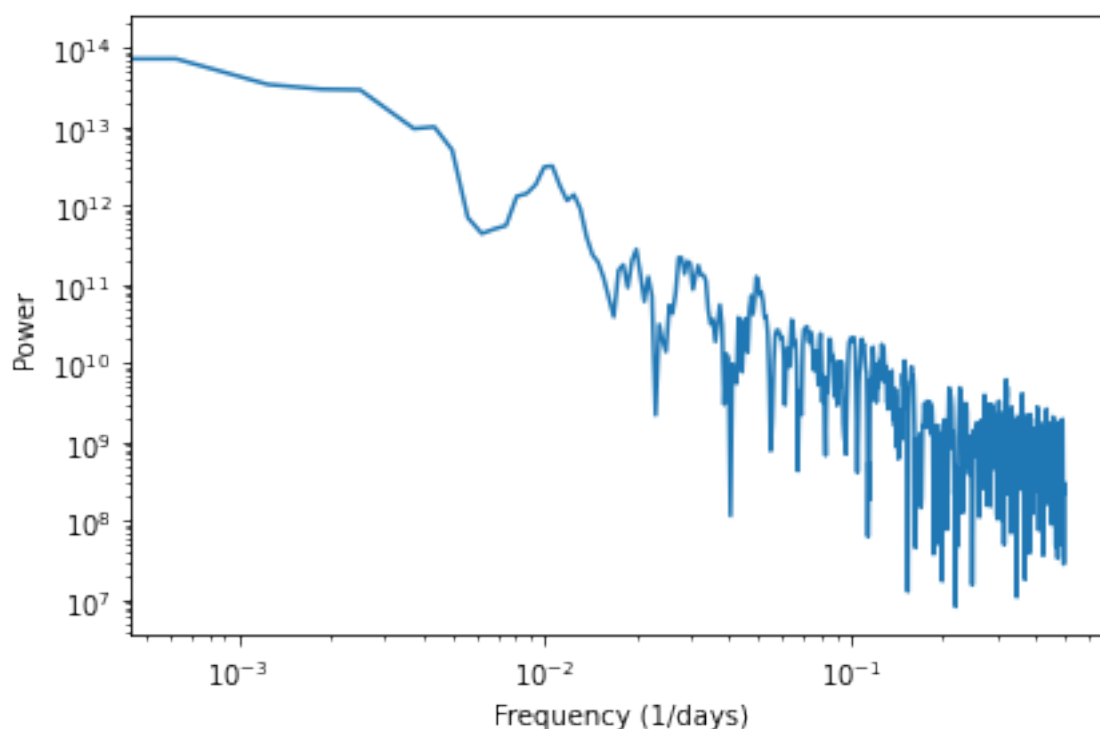


Рисунок 4.7. Спектрограмма цен BitCoin в логорифмическом формате

Больше всего это походе на красный шум, так как связь похожа на отбраную квадратную.

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.

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

1 noise1 = UncorrelatedPoissonNoise(0.001)
2 wave1 = noise1.make_wave(duration=1, framerate = 11025)
3 wave1.plot()
```

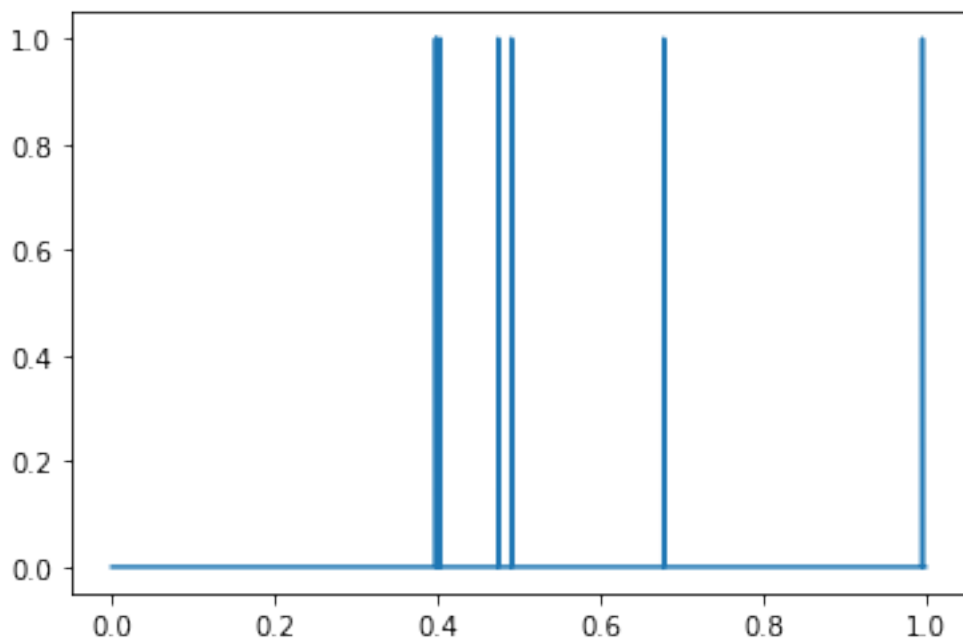


Рисунок 4.8. Получившийся сигнал

```

1 noise2 = UncorrelatedPoissonNoise(1)
2 wave2 = noise2.make_wave(duration=1, framerate = 11025)
3 wave2.plot()

```

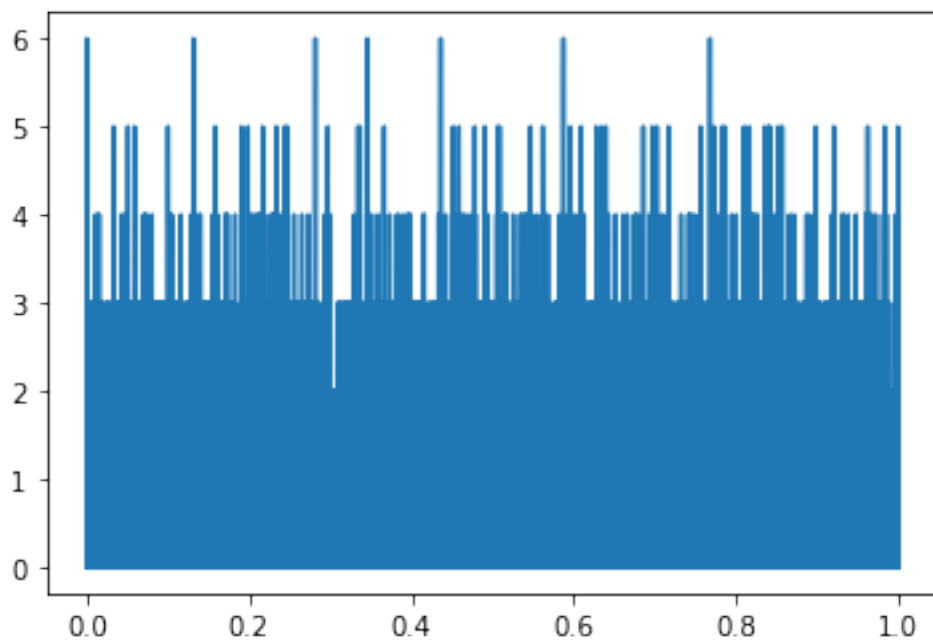


Рисунок 4.9. Получившийся сигнал

```

1 spectrum1 = wave1.make_spectrum()
2 spectrum2 = wave2.make_spectrum()
3 spectrum1.plot_power()
4 spectrum2.plot_power()

```

```

5
6 decorate(xlabel='Frequency (Hz)',
7          ylabel='Power',
8          **loglog)

```

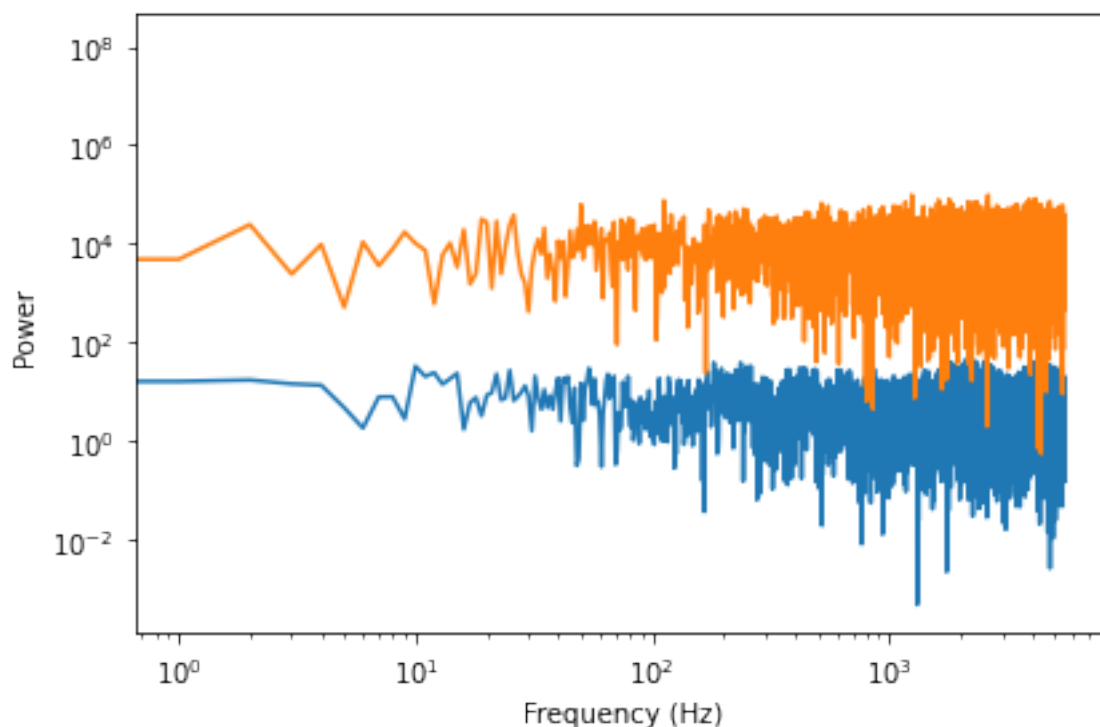


Рисунок 4.10. Сравнение спектров

При увеличении значения амплитуды сигнал всё больше походит 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 Восс-Маккартни.

Исследуйте этот метод, реализуйте его, вычислите спектр и подтвердите, что он имеет желаемое отношение между мощностью и частотой.

```

1 def iterpink(depth):
2     values = np.random.randn(depth)
3     smooth = np.random.randn(depth)
4     source = np.random.randn(depth)
5     sumvals = values.sum()
6     i = 0
7     while True:
8         yield sumvals + smooth[i]
9         i += 1
10        if i == depth:
11            i = 0
12            smooth = np.random.randn(depth)
13            source = np.random.randn(depth)
14            continue
15        c = 0

```



```

16     while not (i >> c) & 1:
17         c += 1
18         sumvals += source[i] - values[c]
19         values[c] = source[i]

1 def pink_noise(points, depth=16):
2     a = []
3     s = iterpink(depth)
4     for n in range(points):
5         a.append(next(s))
6     return np.array(a)

1 ys = pink_noise(11025,16)
2 wave = Wave(ys)
3 wave.plot()

```

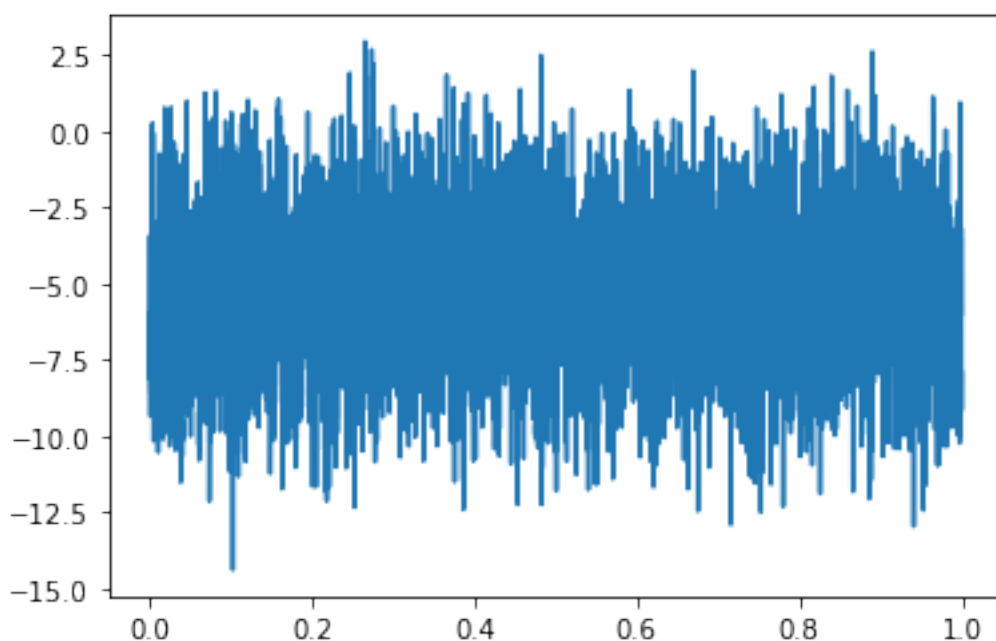


Рисунок 4.11. Сгенерированный сигнал

В итоге получили сигнал розового шума:

```

1 spectrum = wave.make_spectrum()
2 spectrum.hs[0] = 0
3 spectrum.plot_power()
4 decorate(xlabel='Frequency (Hz)',
5         ylabel='Power',
6         **loglog)

```

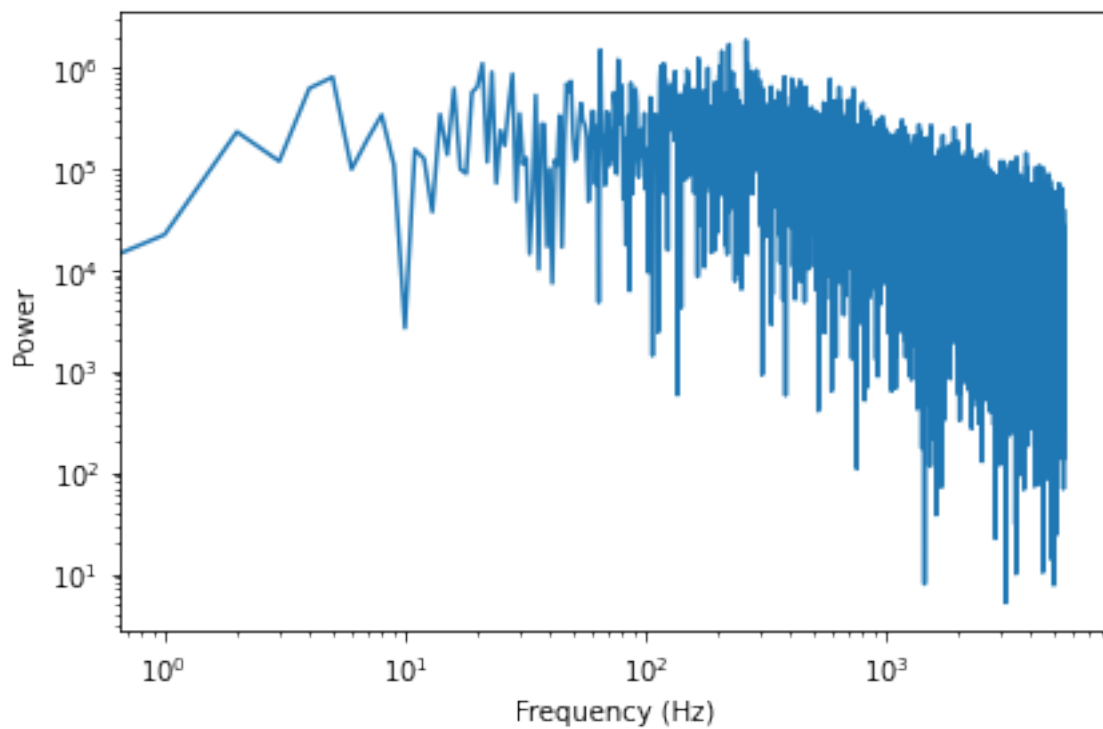


Рисунок 4.12. Розовый шум

4.6. Вывод

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

5. Автокорреляция

5.1. Упражнение 1

Оцените высоты тона вокального чирпа для нескольких времён начала сегмента.

```
1 if not os.path.exists('28042__bcjordan__voicedownbew.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/28042
3     __bcjordan__voicedownbew.wav
4 wave = read_wave('code_28042__bcjordan__voicedownbew.wav')
5 wave.normalize()
6 duration = 0.01
7 segment1 = wave.segment(start=0.1, duration=duration)
8 segment1.plot()
9 segment2 = wave.segment(start=0.2, duration=duration)
10 segment2.plot()
11 segment3 = wave.segment(start=0.3, duration=duration)
12 segment3.plot()
```

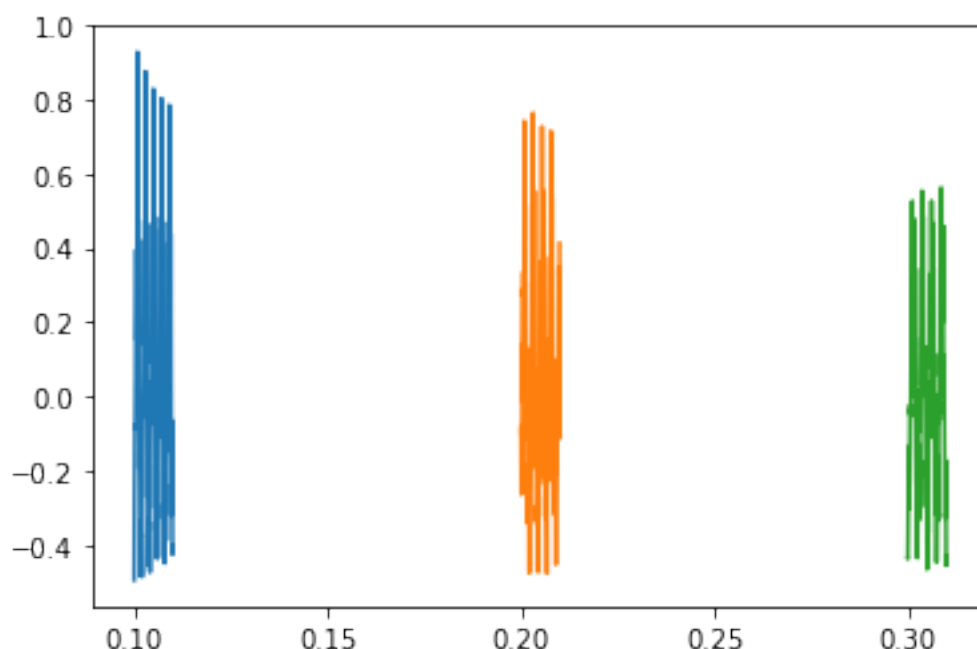


Рисунок 5.1. График выбранных сегментов

Для определения высоты тона используем автокорреляцию.

```
1 lags1, corrs1 = autocorr(segment1)
2 plt.plot(lags1, corrs1, color='green')
3 decorate(xlabel='Lag (index)', ylabel='Correlation', ylim=[-1, 1])
4
5 lags2, corrs2 = autocorr(segment2)
6 plt.plot(lags2, corrs2)
7 decorate(xlabel='Lag (index)', ylabel='Correlation', ylim=[-1, 1])
8
9 lags3, corrs3 = autocorr(segment3)
10 plt.plot(lags3, corrs3, color='red')
11 decorate(xlabel='Lag (index)', ylabel='Correlation', ylim=[-1, 1])
```

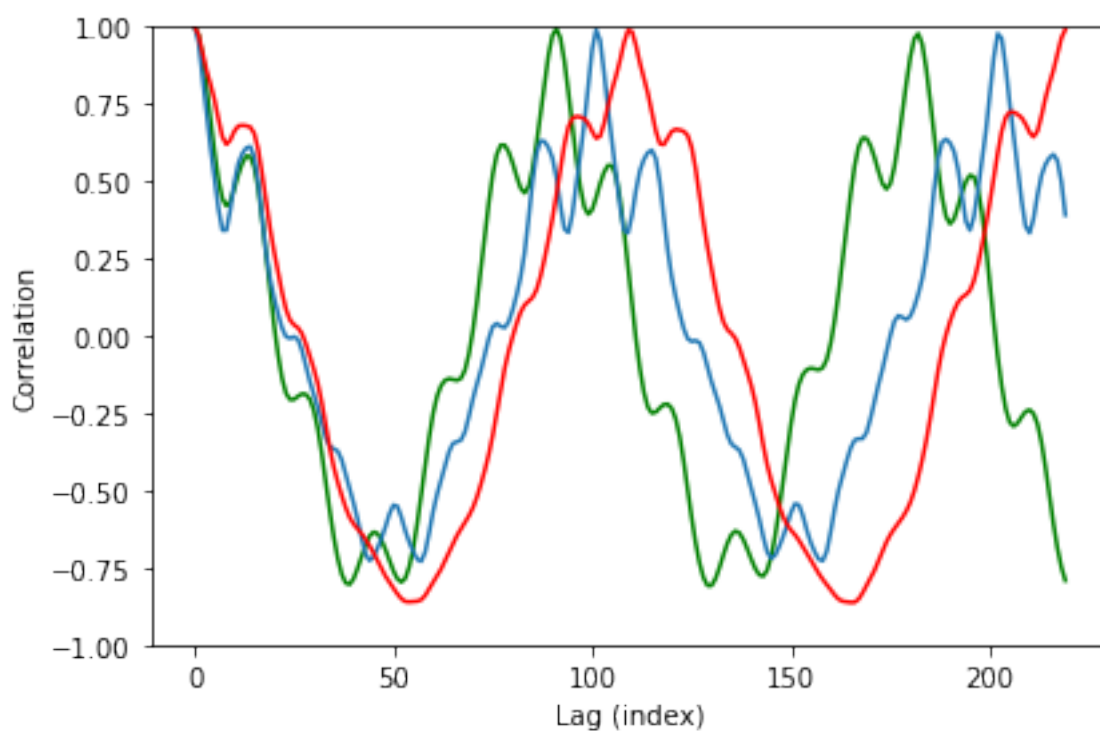


Рисунок 5.2. Автокорреляция сигналов

Узнаем значения lag:

```

1 low, high = 50, 200
2 lag1 = np.array(corr1[low:high]).argmax() + low
3
4 low, high = 50, 200
5 lag2 = np.array(corr2[low:high]).argmax() + low
6
7 low, high = 50, 200
8 lag3 = np.array(corr3[low:high]).argmax() + low
9
10 lag1, lag2, lag3

1 (91, 101, 109)

```

Периоды:

```

1 period1 = lag1 / segment1.framerate
2 period2 = lag2 / segment2.framerate
3 period3 = lag3 / segment3.framerate
4 period1, period2, period3

1 (0.0020634920634920637, 0.002290249433106576, 0.002471655328798186)

```

F max:

```

1 frequency1 = 1 / period1
2 frequency2 = 1 / period2
3 frequency3 = 1 / period3
4 frequency1, frequency2, frequency3

1 (484.6153846153846, 436.63366336633663, 404.5871559633028)

```

5.2. Упражнение 2

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

Возьмём тот же звук.

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

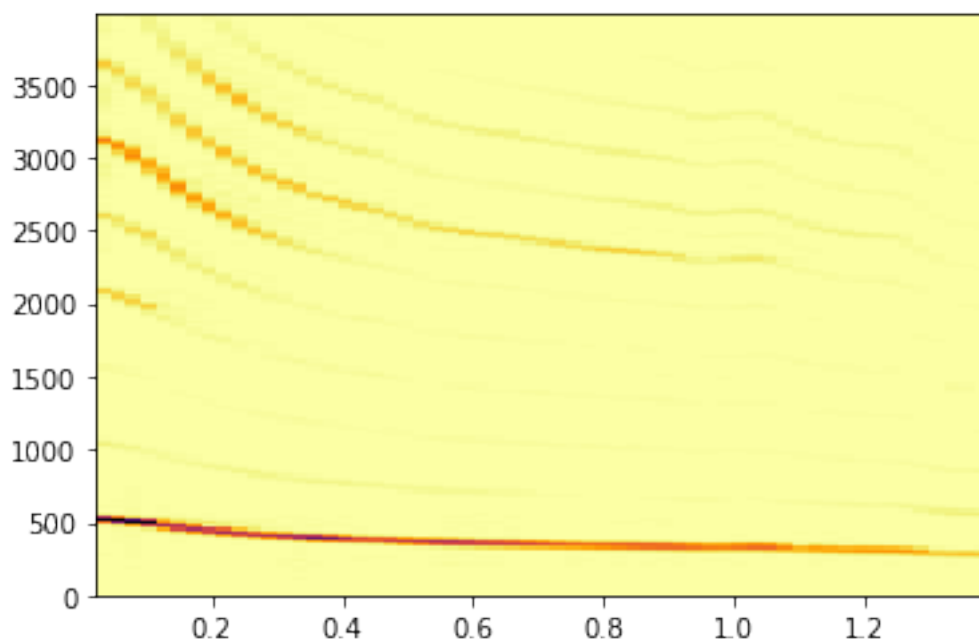


Рисунок 5.3. Спектрограмма записи

Берём код из предыдущих пунктов и соединяем в одну функцию:

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

```
1 estimate_fundamental(segment1)
```

```
1 484.6153846153846
```

Чтобы сделать оценку высоты тона на всей спектрограмме надо разделить всё на маленькие сегменты и работать с ними.

```
1 duration = wave.duration
2 step = 0.02
3 start = 0
4 time = []
5 freq = []
6 while start + step < duration:
7     time.append(start + step/2)
8     freq.append(estimate_fundamental(wave.segment(start=start, duration=step)))
9     start += step
10 wave.make_spectrogram(2048).plot(high=900)
```

```

11 plt.plot(time, freq, color='blue')
12 decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')

```

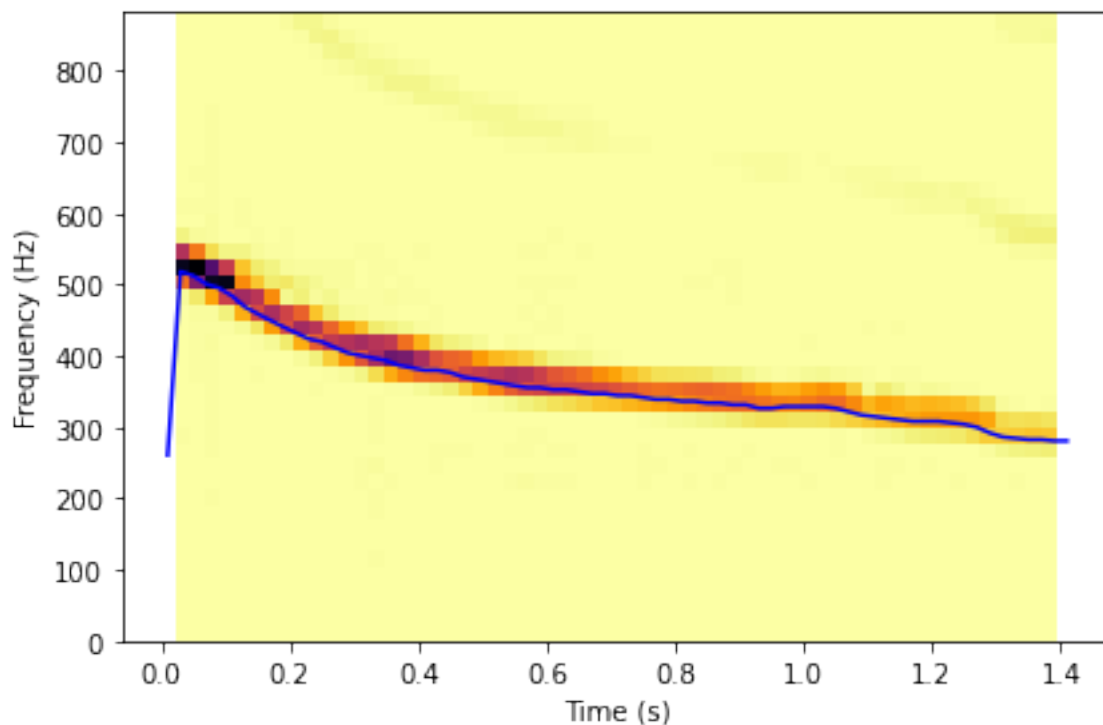


Рисунок 5.4. Результат

5.3. Упражнение 3

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

```

1 if not os.path.exists('market-price.csv'):
2     !wget https://github.com/wooftown/spbstu-telecom/raw/main/Content/market-
      price.csv
3 import pandas as pd
4
5 df = pd.read_csv('market-price.csv', parse_dates=[0])
6 ys = df['market-price']
7 ts = df.index
8
9 w = Wave(ys, framerate=1)
10 w.plot()

```

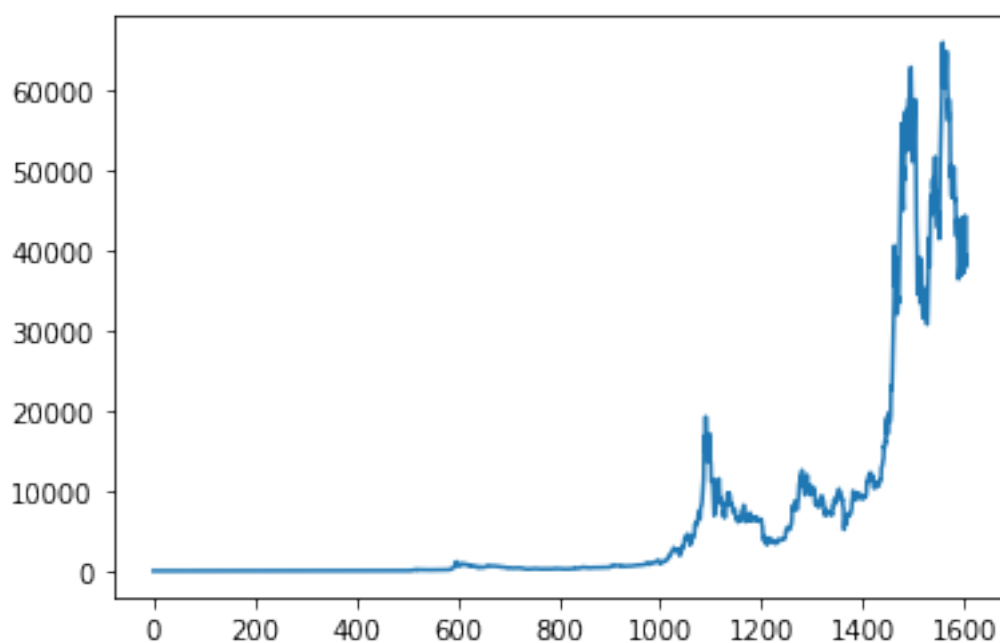


Рисунок 5.5. График цены на BitCoin

Вычислим автокорреляцию:

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

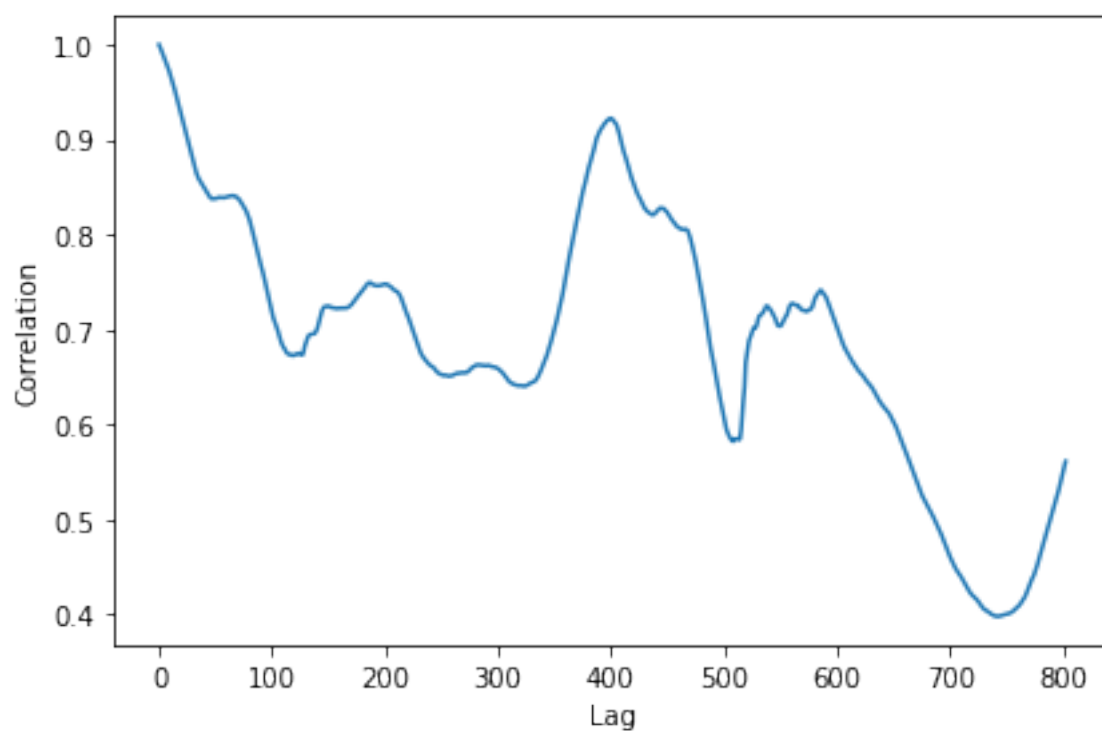


Рисунок 5.6. Автокорреляция функции цены на BitCoin

Функция ведёт себя нестабильно, есть резкие повышения и спады. Имеется намёк на периодичность.

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 подавленной основной. Прочтите этот блокнот и «погоняйте» примеры. Выберите другой сегмент записи и вновь поработайте с примерами.

```
1 if not os.path.exists('100475__iluppai__saxophone-weep.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/100475
      __iluppai__saxophone-weep.wav
3 wave = read_wave('100475__iluppai__saxophone-weep.wav')
4 wave.normalize()
5 spectrogram = wave.make_spectrogram(seg_length=1024)
6 spectrogram.plot(high=3000)
7 decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

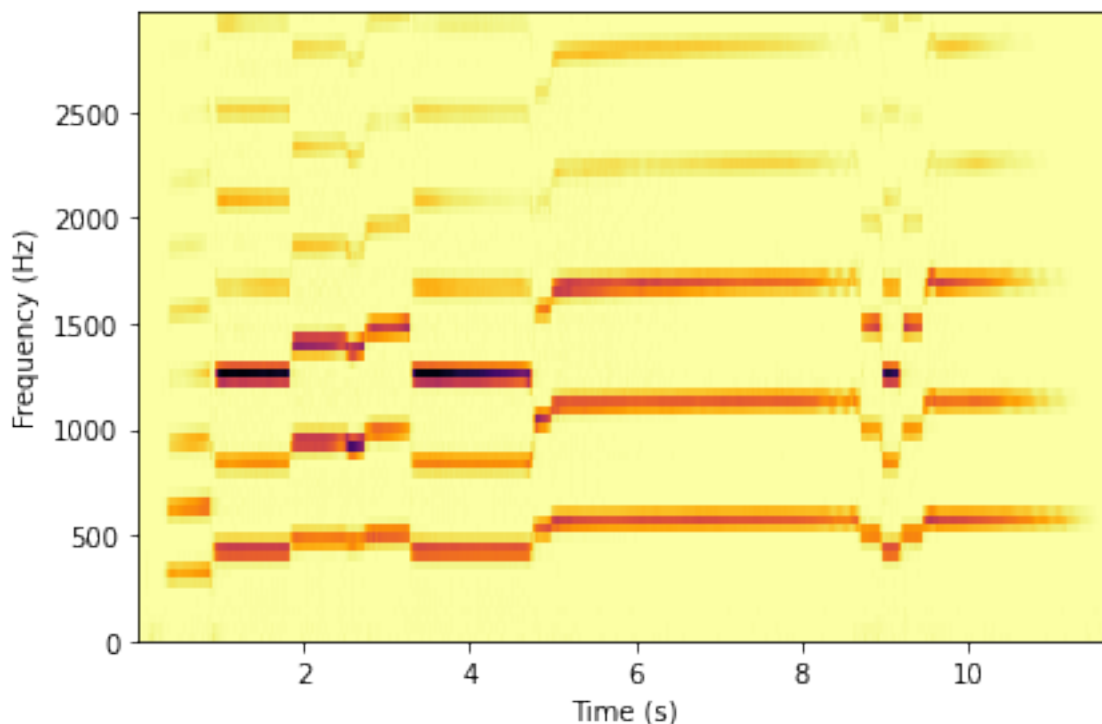


Рисунок 5.7. Спектрограмма звука

Видна гармоническая структура во времени. Возьмём отрезок и прогоним его через все функции из блокнота.

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

1 spectrum = segment.make_spectrum()
2 spectrum.plot(high=5000)
3 decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

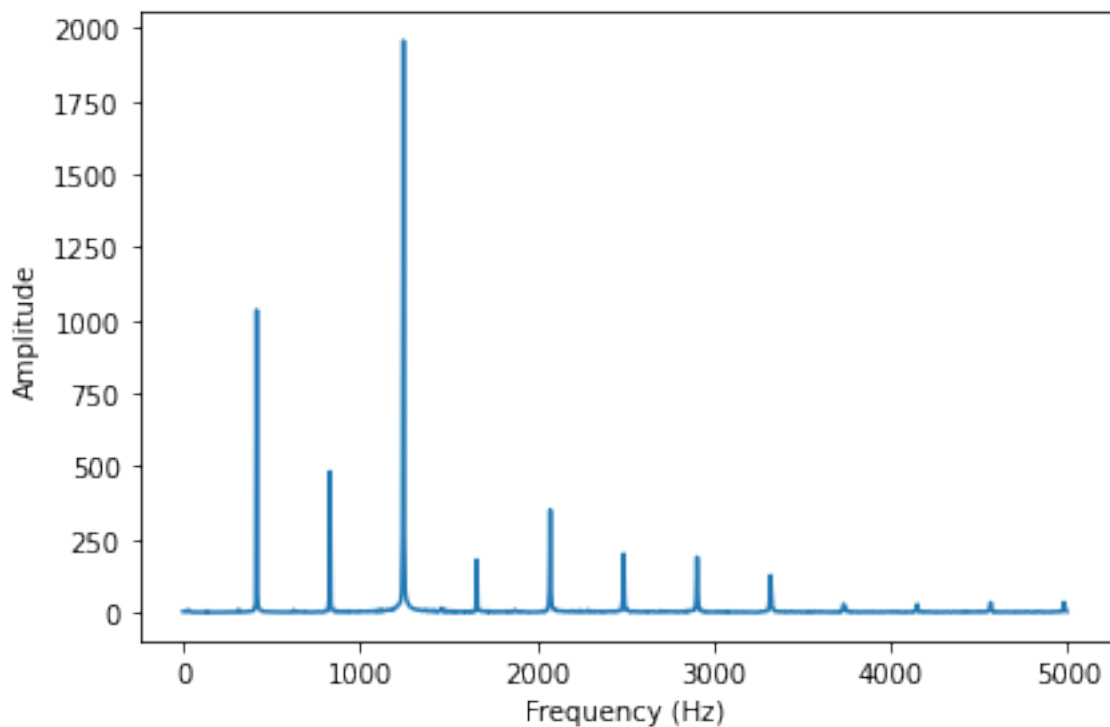



Рисунок 5.8. Спектр звука

Спектр напомнил квадратный сигнал. Пики припались на 1245, 415, 830 Гц.

Далее сравним наш сегмент с треугольным сигналом с такой же низкой частотой пика.

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

У данных сигналов одинаковая воспринимаемая частота звука.

Для понимания процесса восприятия основной частоты используем АКФ.

```
1 def autocorr2(segment):
2     corrs = np.correlate(segment.ys, segment.ys, mode='same')
3     N = len(corrs)
4     lengths = range(N, N//2, -1)
5
6     half = corrs[N//2:].copy()
7     half /= lengths
8     half /= half[0]
9     return half
```

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

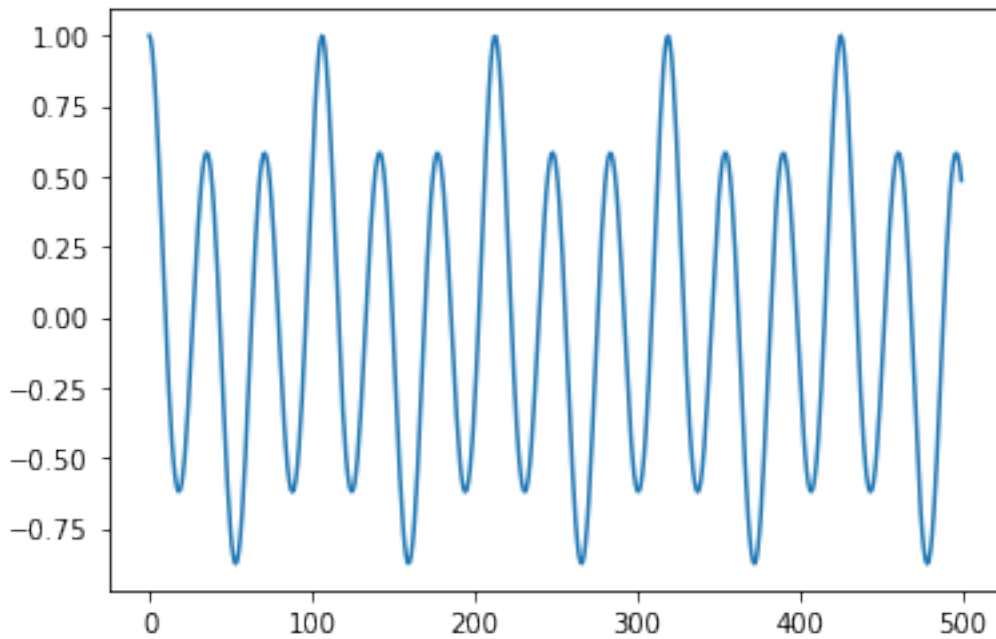


Рисунок 5.9. Автокорреляция

Видим пик рядом с lag 100. Найдём основную частоту при помощи написанной ранее функции.

```
1 estimate_fundamental(segment)
```

Даже если мы уберём основной тон (416 Гц), звук будет восприниматься также...

```
1 spectrum2 = segment.make_spectrum()
```

```
2 spectrum2.high_pass(600)
```

```
3 spectrum2.plot(high=5000)
```

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

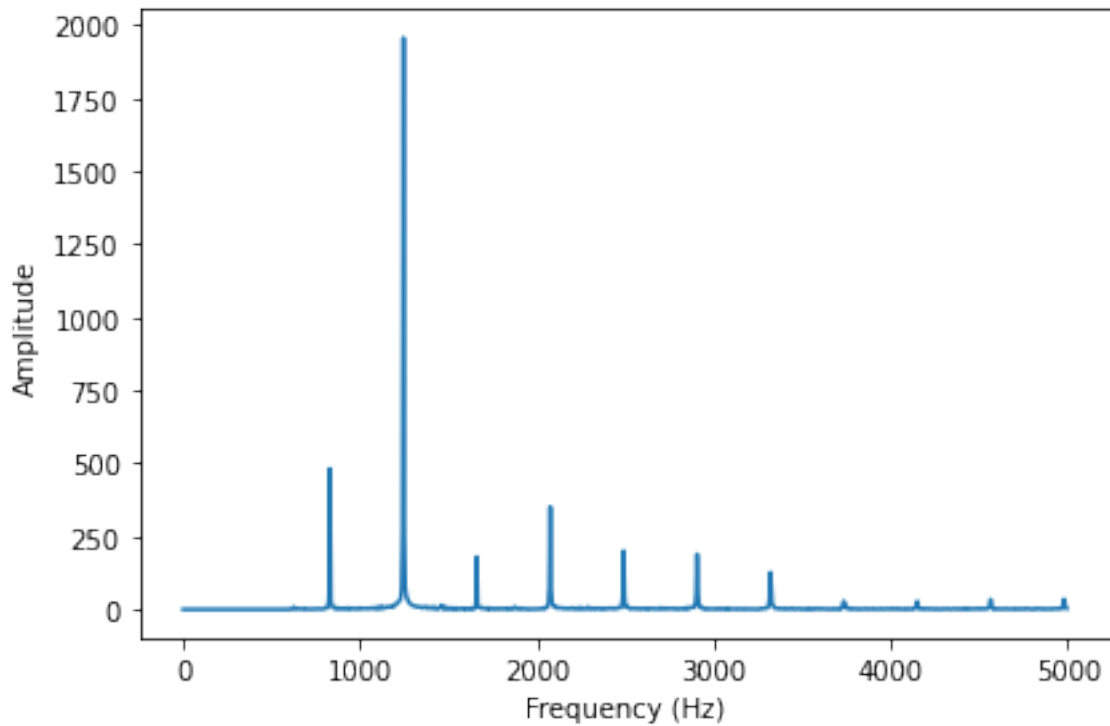


Рисунок 5.10. Спектр сигнала

Это явление называется "missing fundamental". Для понимания, что мы слышим частоту которой нет можно снова обратиться к АКФ.

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

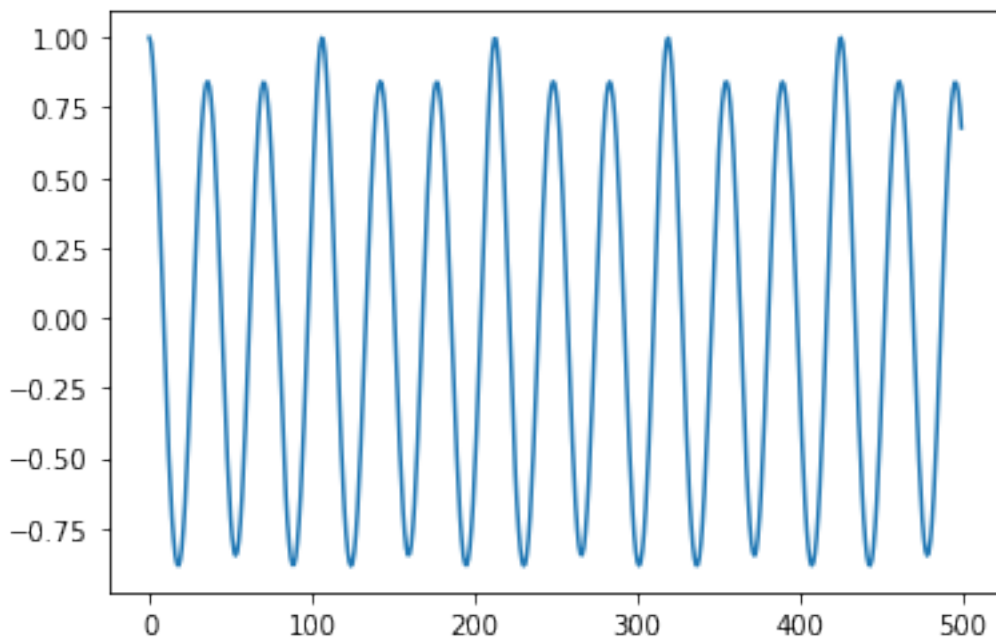


Рисунок 5.11. Автокорреляция

Это так работает, потому что более высокие компоненты сигнала являются гармониками 416 Гц. Весь этот пример указывает нам на то, что восприятие высоты 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 практике был обработан сигнал с "missing fundamental". Когда мы убрали основной тон, всё равно звук звучал также.

6. Дискретное косинусное преобразование

6.1. Упражнение 1

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

```
1 def analyze1(ys, fs, ts):
2     args = np.outer(ts, fs)
3     M = np.cos(PI2 * args)
4     amps = np.linalg.solve(M, ys)
5     return amps
6 def analyze2(ys, fs, ts):
7     args = np.outer(ts, fs)
8     M = np.cos(PI2 * args)
9     amps = M.dot(ys) / 2
10    return amps
```

Возьмём размеры массива как степени 2.

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

1 best_analyze1 = []
2 for n in ns:
3     ts = (0.5 + np.arange(n)) / n
4     freqs = (0.5 + np.arange(n)) / 2
5     ys = wave.ys[:n]
6     best = %timeit -r1 -o analyze1(ys,freqs,ts)
7     best_analyze1.append(best.best)
8 best_analyze2 = []
9 for n in ns:
10    ts = (0.5 + np.arange(n)) / n
11    freqs = (0.5 + np.arange(n)) / 2
12    ys = wave.ys[:n]
13    best = %timeit -r1 -o analyze2(ys,freqs,ts)
14    best_analyze2.append(best.best)
15 best_dct = []
16 for n in ns:
17    ys = wave.ys[:n]
18    best = %timeit -r1 -o scipy.fftpack.dct(ys, type=3)
19    best_dct.append(best.best)
20 plt.plot(ns, best_analyze1, label='analyze1')
21 plt.plot(ns, best_analyze2, label='analyze2')
22 plt.plot(ns, best_dct, label='fftpack.dct')
23 loglog = dict(xscale='log', yscale='log')
24 decorate(xlabel='Wave length (N)', ylabel='Time (s)', **loglog)
```

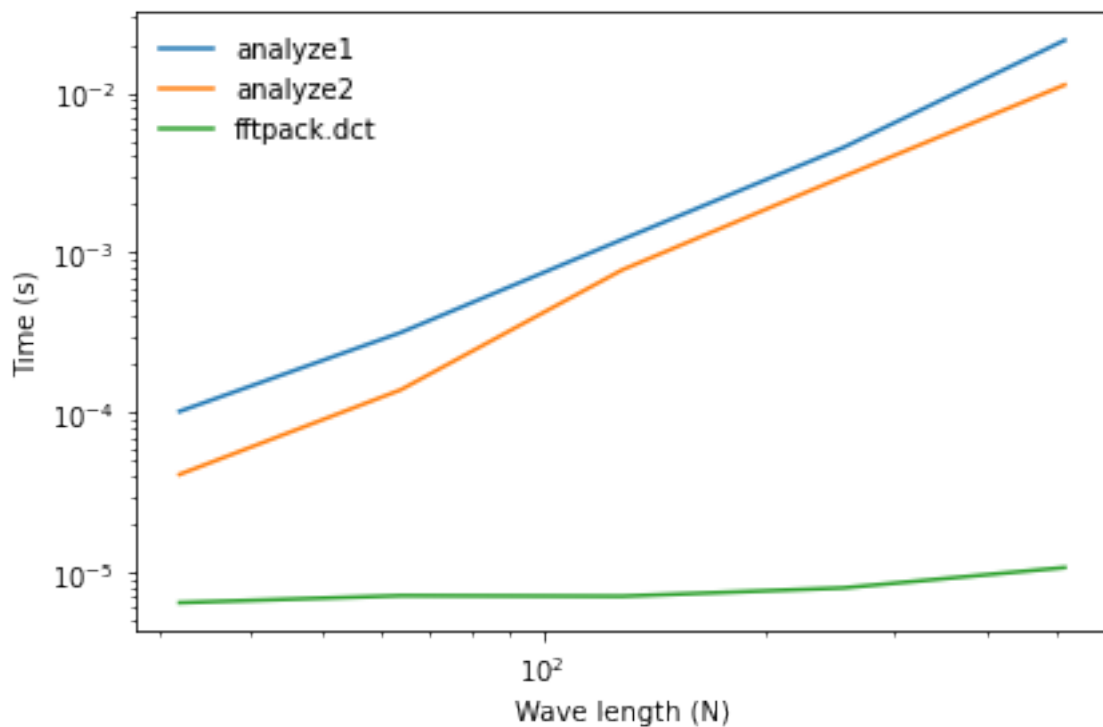


Рисунок 6.1. Время работы различных методов ДКП

Не смотря на теоритическое время исполнения, время `analyze1` получилось пропорциональным n^2 .

6.2. Упражнение 2

Реализовать алгоритм сжатия для музыки или речи.

Выберем звук для сжатия:

```

1 if not os.path.exists('164718__bradovic__piano.wav'):
2     !wget https://github.com/wooftown/spbstu-telecom/raw/main/Content/164718
      __bradovic__piano.wav
3     wave = read_wave('164718__bradovic__piano.wav')
```

Для начала возьмём небольшой сегмент:

```

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

Вместо DFT используем DCT.

```

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

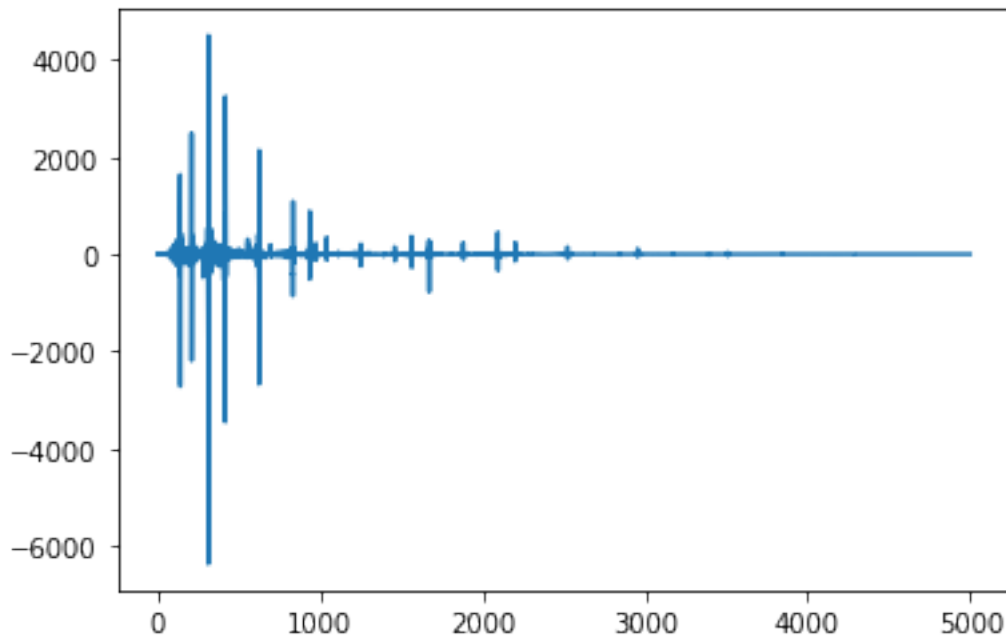


Рисунок 6.2. Спектр сигнала полученный при помощи ДКП

```

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

```

```

1 filtering(dct, 1000)
2 dct.plot(high = 5000)

```

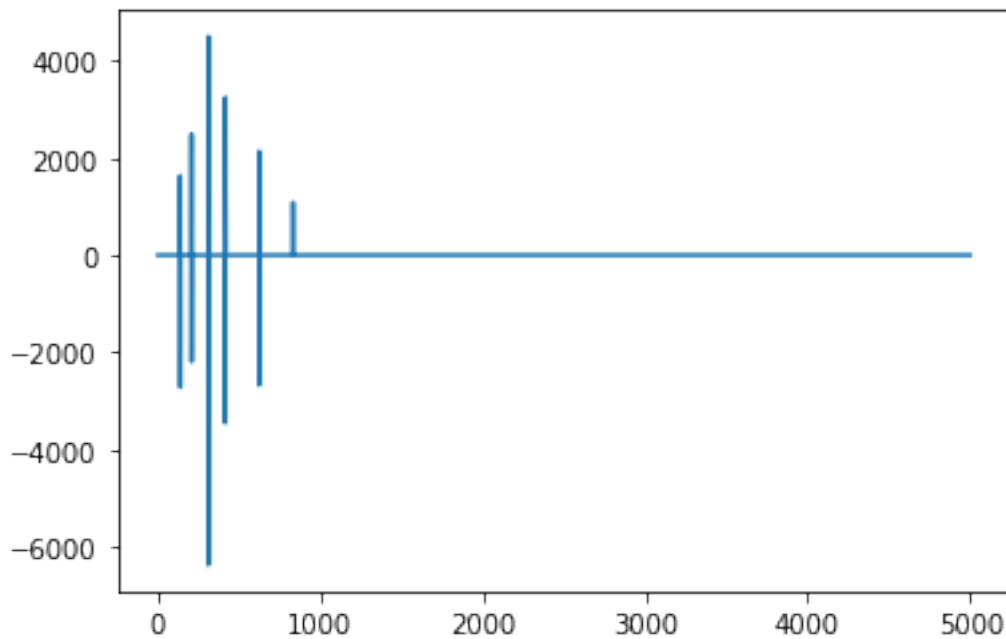


Рисунок 6.3. ДКП после фильтрации

Теперь осталось подобрать значение, чтобы выходной файл звучал как входной. Для сохранения памяти необходимо использовать разреженные массивы.

6.3. Упражнение 3

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

```
1 signal = SawtoothSignal(freq=500, offset=0)
2 wave = signal.make_wave(duration=0.5, framerate=40000)
3 wave.segment(start=0.005,duration=0.01).plot()
4 decorate(xlabel='Time (s)')
```

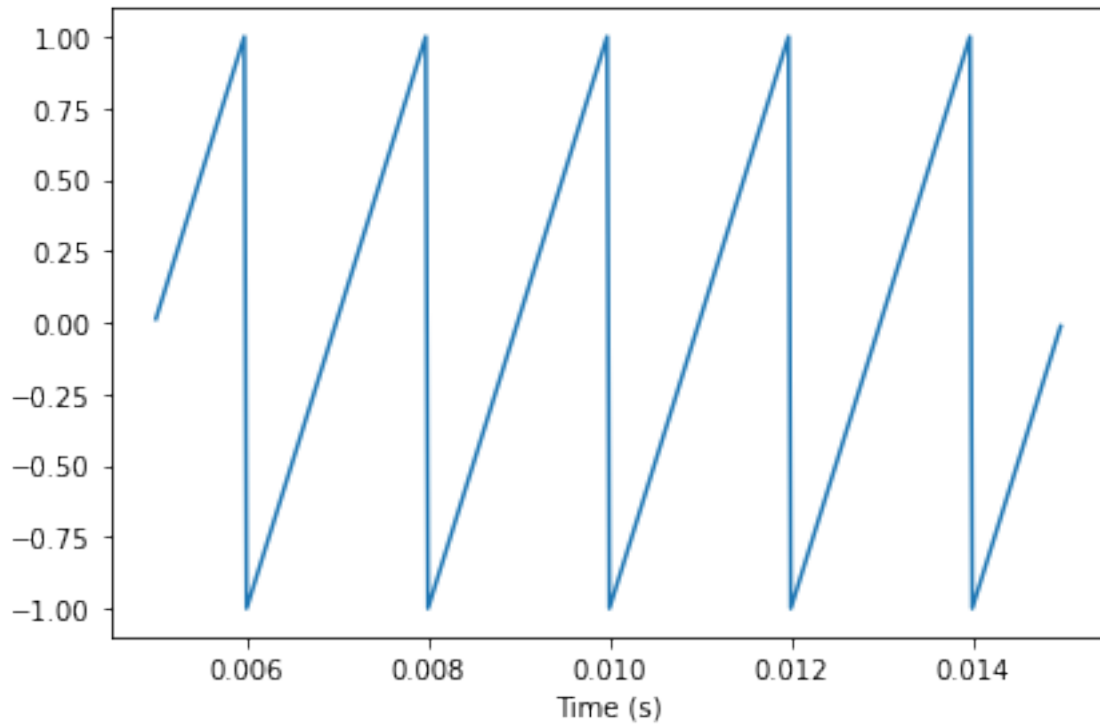


Рисунок 6.4. Выбранный сегмент

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

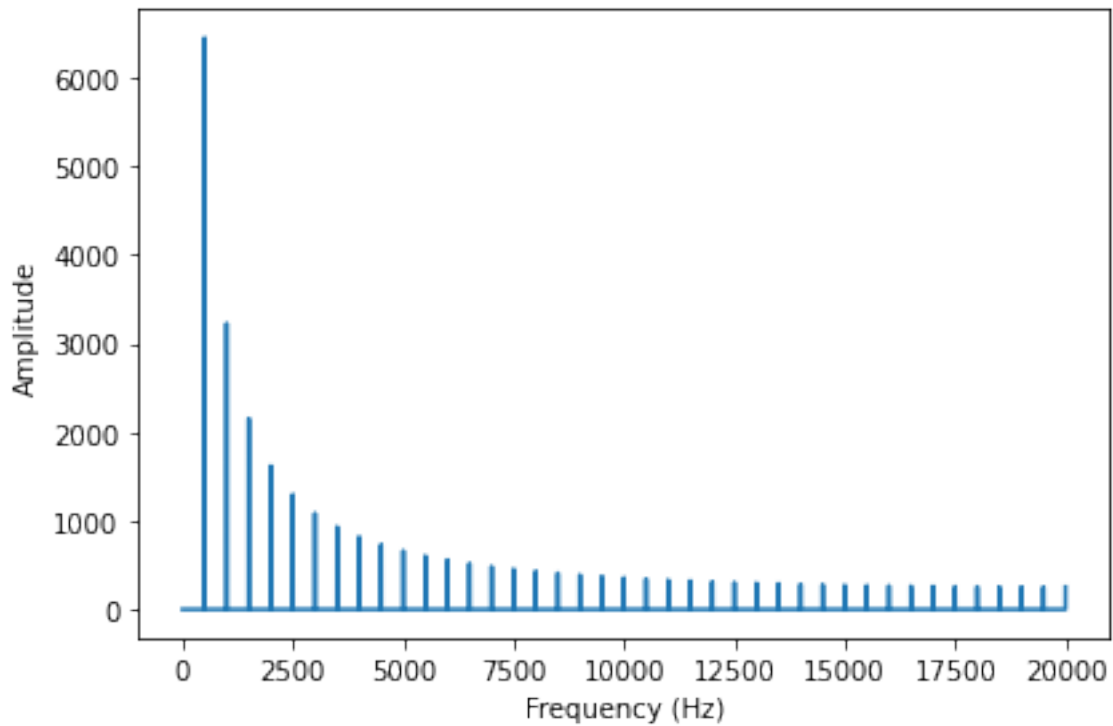



Рисунок 6.5. Спектр сегмента

```

1 def plot_angle(spectrum, thresh=1):
2     angles = spectrum.angles
3     angles[spectrum.amps < thresh] = np.nan
4     plt.plot(spectrum.fs, angles, 'x')
5     decorate(xlabel='Frequency (Hz)',
6             ylabel='Phase (radian)')
7
8 def plot_three(spectrum, thresh=1):
9     """Plot amplitude, phase, and waveform.
10
11     spectrum: Spectrum object
12     thresh: threshold passed to plot_angle
13     """
14     plt.figure(figsize=(10, 4))
15     plt.subplot(1,3,1)
16     spectrum.plot()
17     plt.subplot(1,3,2)
18     plot_angle(spectrum, thresh=thresh)
19     plt.subplot(1,3,3)
20     wave = spectrum.make_wave()
21     wave.unbias()
22     wave.normalize()
23     wave.segment(duration=0.01).plot()
24     display(wave.make_audio())
25
26 def zero_angle(spectrum):
27     res = spectrum.copy()
28     res.hs = res.amps
29     return res
30
31 def rotate_angle(spectrum, offset):

```

```

32 res = spectrum.copy()
33 res.hs *= np.exp(1j * offset)
34 return res
35
36 def random_angle(spectrum):
37     res = spectrum.copy()
38     angles = np.random.uniform(0, PI2, len(spectrum))
39     res.hs *= np.exp(1j * angles)
40     return res

```

1 plot_three(spectrum)

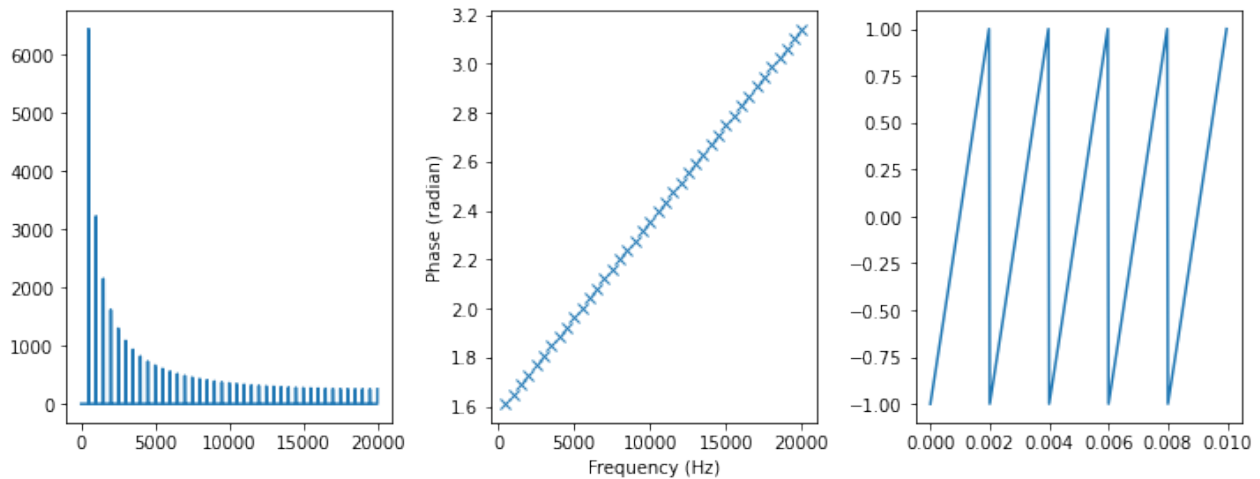


Рисунок 6.6. Получившиеся графики

```

1 spectrum2 = zero_angle(spectrum)
2 plot_three(spectrum2)

```

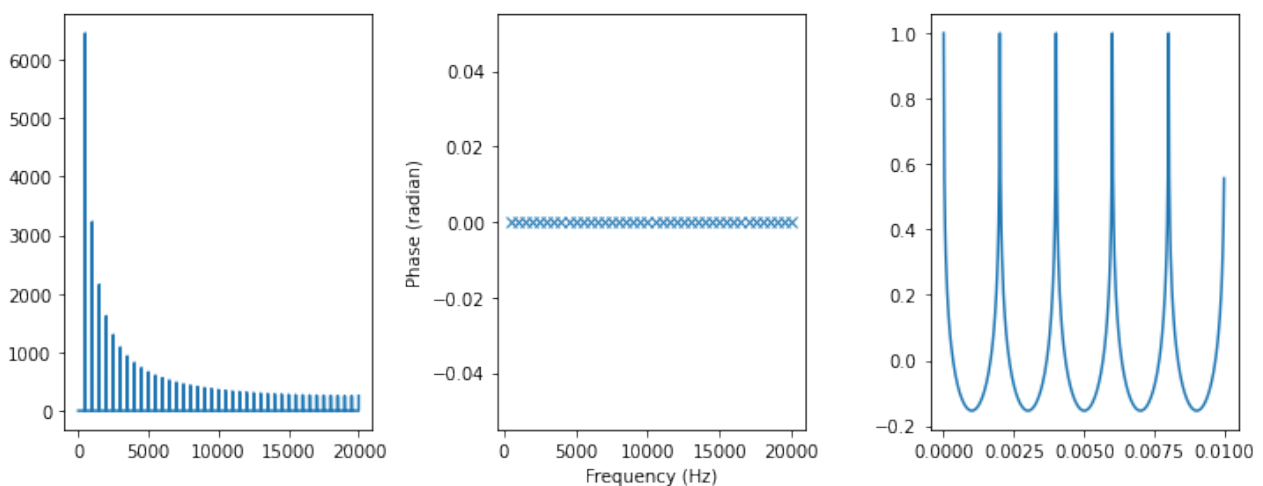


Рисунок 6.7. Получившиеся графики

```

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

```

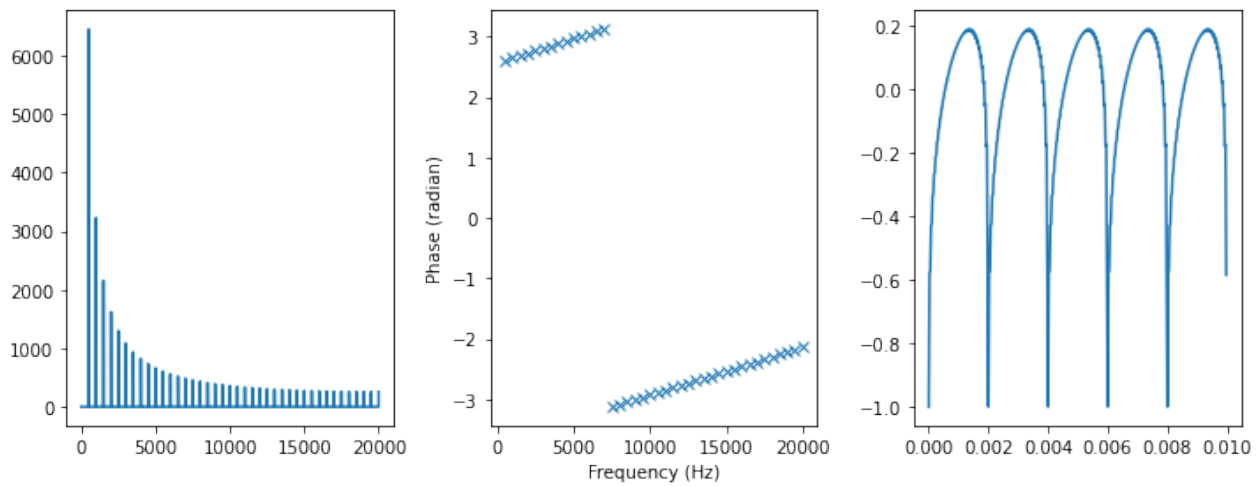


Рисунок 6.8. Получившиеся графики

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

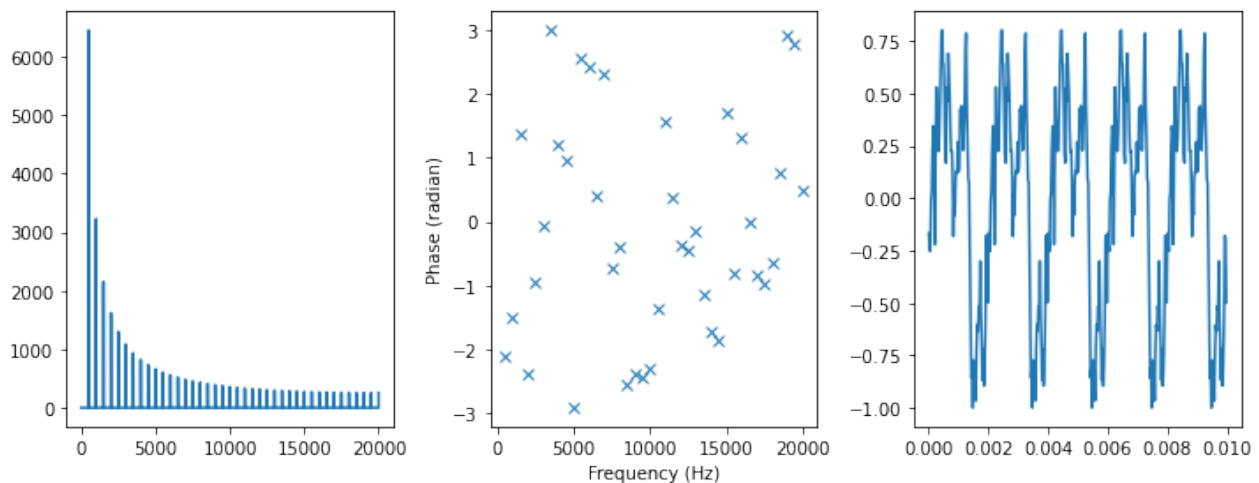


Рисунок 6.9. Получившиеся графики

Теперь возьмём другой звук и сделаем всё тоже самое:

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

1 plot_three(spectrum)
```

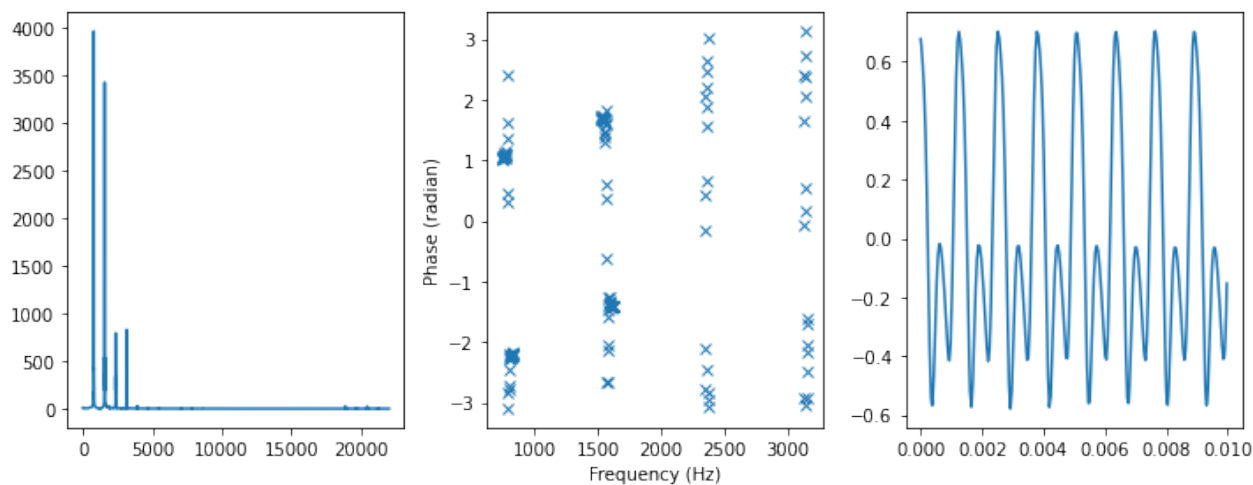


Рисунок 6.10. Получившиеся графики

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

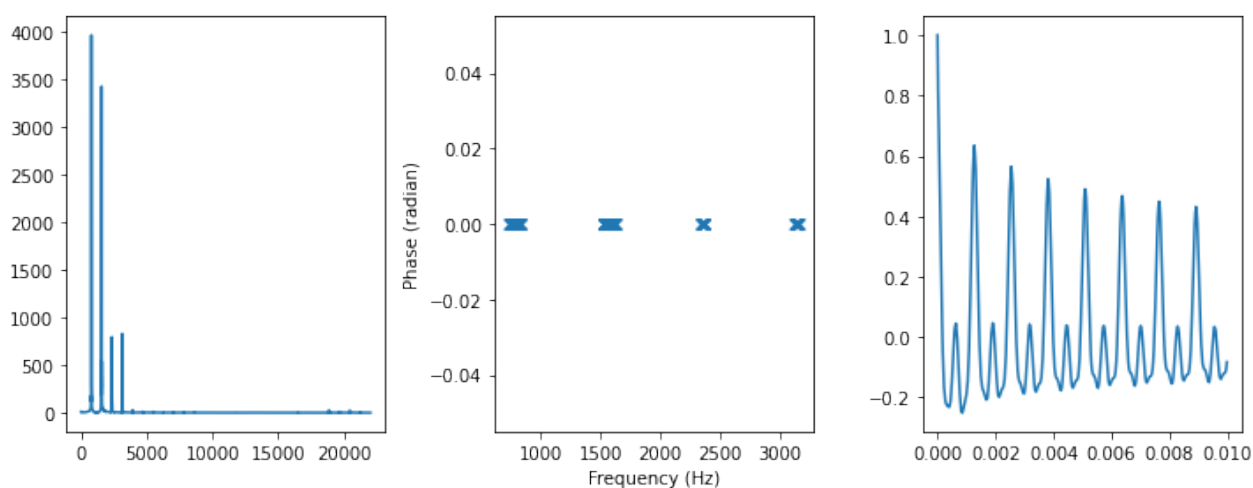


Рисунок 6.11. Получившиеся графики

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

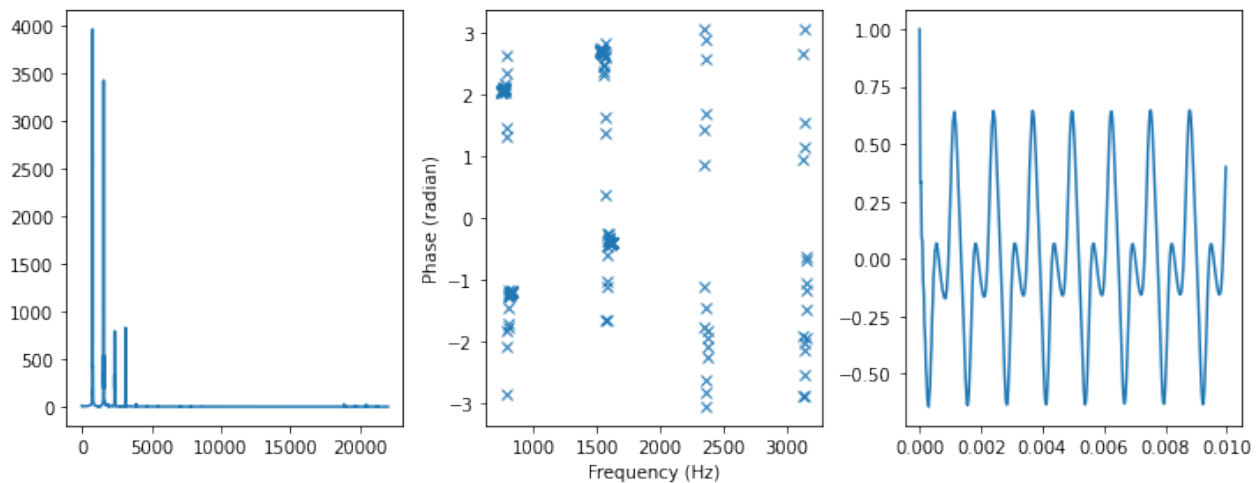


Рисунок 6.12. Получившиеся графики

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

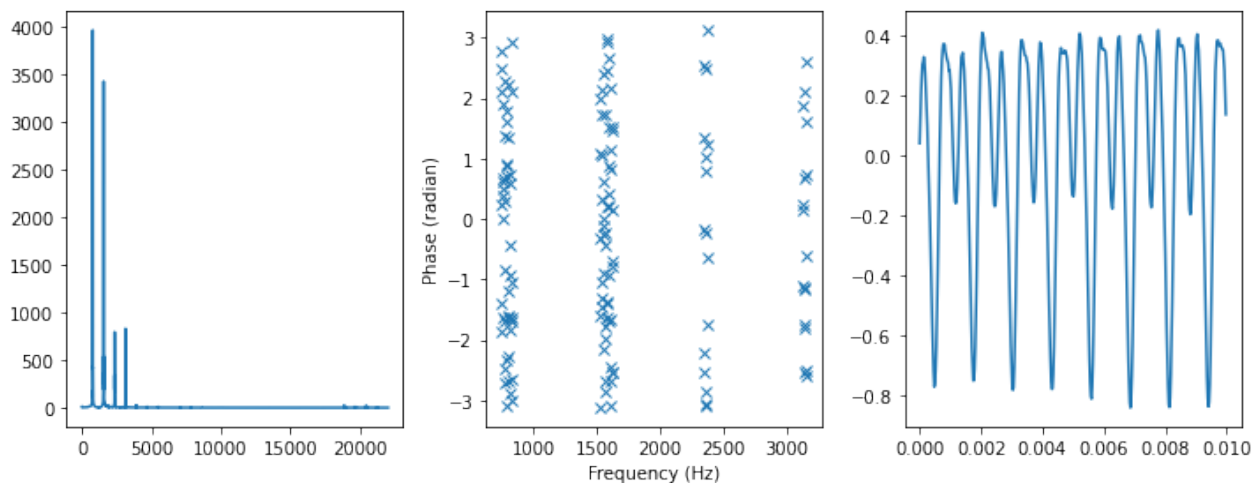


Рисунок 6.13. Получившиеся графики

Мы опять очень сильно изменили сигнал, но воспринимаемый звук весьма похож на начальный. Для звуков с простой гармонической структурой мы не слышим изменения в фазовой структуре, при условии что гармоническая структура неизменна.

6.4. Вывод

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

7. Дискретное преобразование Фурье

7.1. Упражнение 1

Реализовать алгоритм БПФ.

Возьмём простой массив для примера:

```
1 ys = [0.9,0.7,-0.6,-0.6]
2 hs = np.fft.fft(ys)

1 array([0.4+0.j , 1.5-1.3j, 0.2+0.j , 1.5+1.3j])
```

```
1 def dft(ys):
2     N = len(ys)
3     ts = np.arange(N) / N
4     freqs = np.arange(N)
5     args = np.outer(ts, freqs)
6     M = np.exp(1j * PI2 * args)
7     amps = M.conj().transpose().dot(ys)
8     return amps
```

Далее разделим массив на элементы:

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

И добавим рекурсивный вызов:

```
1 def my_fft(ys):
2     if len(ys) == 1:
3         return ys
4     He = my_fft(ys[::2])
5     Ho = my_fft(ys[1::2])
6     ns = np.arange(len(ys))
7     W = np.exp(-1j * PI2 * ns / len(ys))
8     return np.tile(He, 2) + W * np.tile(Ho, 2)
```

Таким образом, мы написали собственную функцию БПФ. Попробуем её на нашем массиве:

```
1 my_fft(ys)

1 array([0.4+0.00000000e+00j, 1.5-1.30000000e+00j, 0.2-1.2246468e-17j,
2        1.5+1.30000000e+00j])
```

Результат идентичен с библиотечной функцией.

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 дискретном сигнале. 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

Что случится, если при увеличении ширины гауссова окна `std` не увеличивать число элементов в окне `M`?

Возьмём функции для исследования данного вопроса:

```
1 def zero_pad(array, n):
2     """Extends an array with zeros.
3
4     array: NumPy array
5     n: length of result
6
7     returns: new NumPy array
8     """
9     res = np.zeros(n)
10    res[:len(array)] = array
11    return res
12
13
14 def plot_filter(M=11, std=2):
15     signal = SquareSignal(freq=440)
16     wave = signal.make_wave(duration=1, framerate=44100)
17     spectrum = wave.make_spectrum()
18
19     gaussian = scipy.signal.gaussian(M=M, std=std)
20     gaussian /= sum(gaussian)
21
22     ys = np.convolve(wave.ys, gaussian, mode='same')
23     smooth = Wave(ys, framerate=wave.framerate)
24     spectrum2 = smooth.make_spectrum()
25
26     # plot the ratio of the original and smoothed spectrum
27     amps = spectrum.amps
28     amps2 = spectrum2.amps
29     ratio = amps2 / amps
30     ratio[amps<560] = 0
31
32     # plot the same ratio along with the FFT of the window
33     padded = zero_pad(gaussian, len(wave))
34     dft_gaussian = np.fft.rfft(padded)
35
36     plt.plot(np.abs(dft_gaussian), color='gray', label='Gaussian filter')
37     plt.plot(ratio, label='amplitude ratio')
38
39     decorate(xlabel='Frequency (Hz)', ylabel='Amplitude ratio')
40     plt.show()
```

После сделаем такой виджет:

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

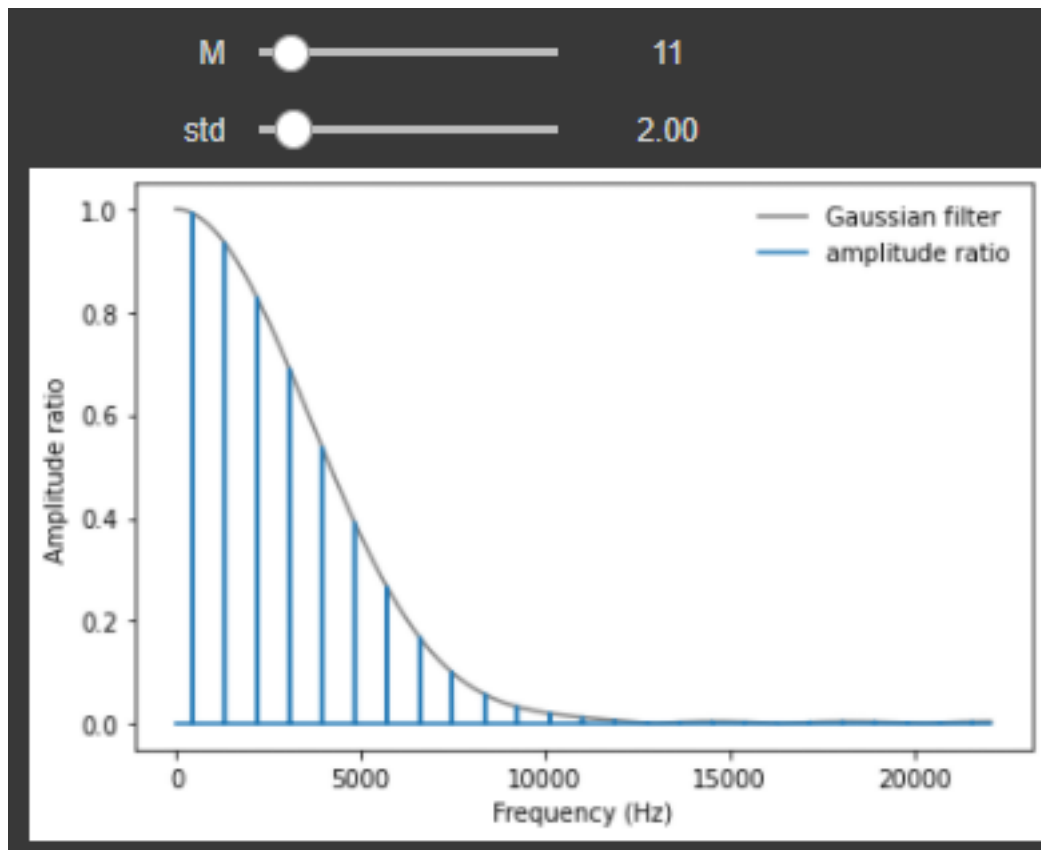


Рисунок 8.1. Гауссово окно для фильтрации

```

1 gaussian = scipy.signal.gaussian(M=11, std=11)
2 gaussian /= sum(gaussian)
3
4 plt.plot(gaussian, label='Gaussian')
5 decorate(xlabel='Index')

```

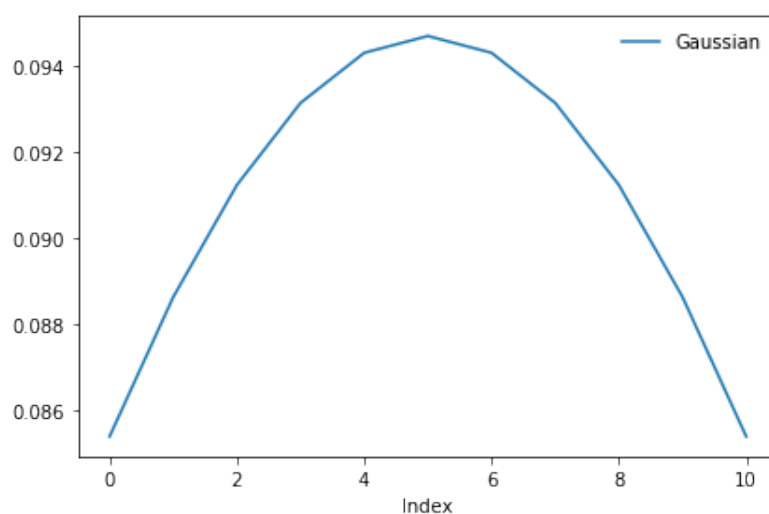


Рисунок 8.2. Гауссово окно

```

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

```



```

2 gaussian /= sum(gaussian)
3
4 plt.plot(gaussian, label='Gaussian')
5 decorate(xlabel='Index')

```

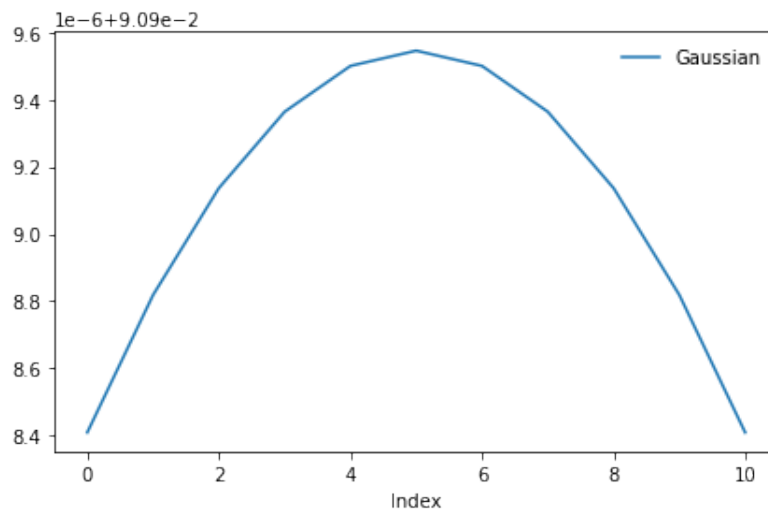


Рисунок 8.3. Гауссово окно

При увеличении std, кривая становится шире, а сам БПФ меньше.

8.2. Упражнение 2

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

```

1 gaussian = scipy.signal.gaussian(M=16, std=2)
2 gaussian /= sum(gaussian)
3
4 plt.plot(gaussian, label='Gaussian')
5 decorate(xlabel='Index')

```

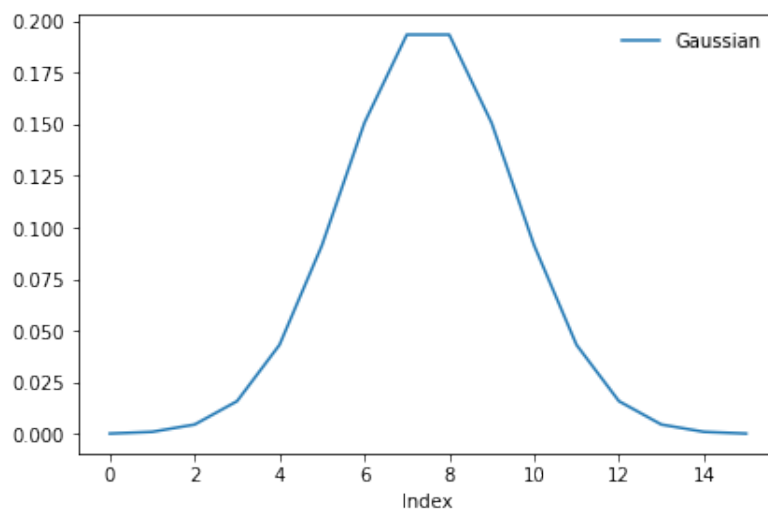


Рисунок 8.4. Гауссово окно

```

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

```

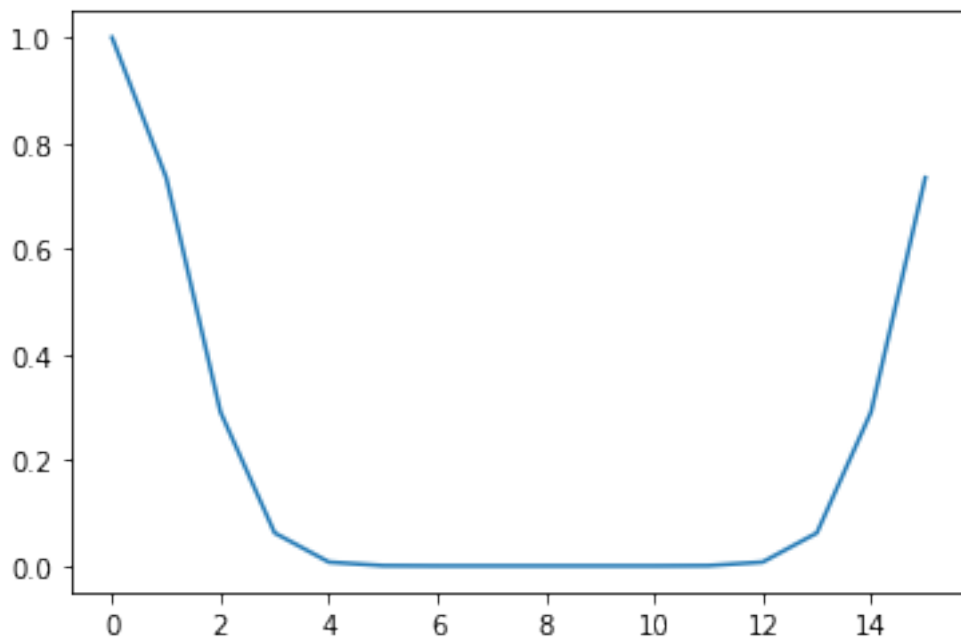


Рисунок 8.5. FFT применённое на окно

Далее выполним свёртку:

```

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

```

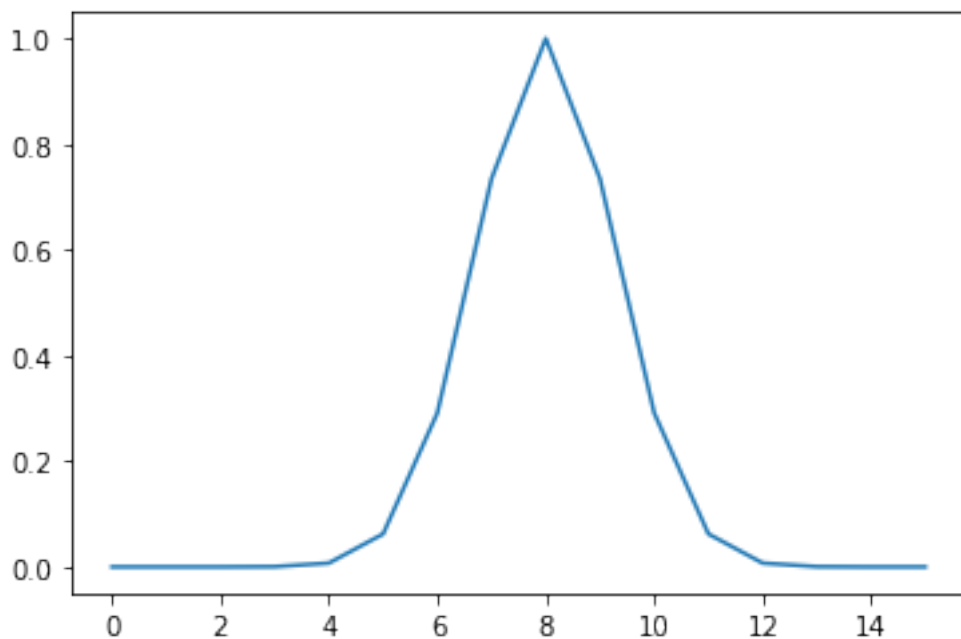


Рисунок 8.6. Результат

Если std гауссовой кривой увеличивается, то преобразование Фурье становится уже.

8.3. Упражнение 3

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

Возьмём примеры из моего решения третьей главы:

```
1 class SawtoothSignal(Sinusoid):
2
3     def evaluate(self,ts):
4         cycles = self.freq * ts + self.offset / (pi / 2)
5         frac, _ = np.modf(cycles)
6         u = unbias(frac)
7         high, low = abs(max(u)), abs(min(u))
8         ys = self.amp * u / max(high,low)
9         return ys
10
11 saw_signal = SawtoothSignal(200)
12 saw_signal.plot()
```

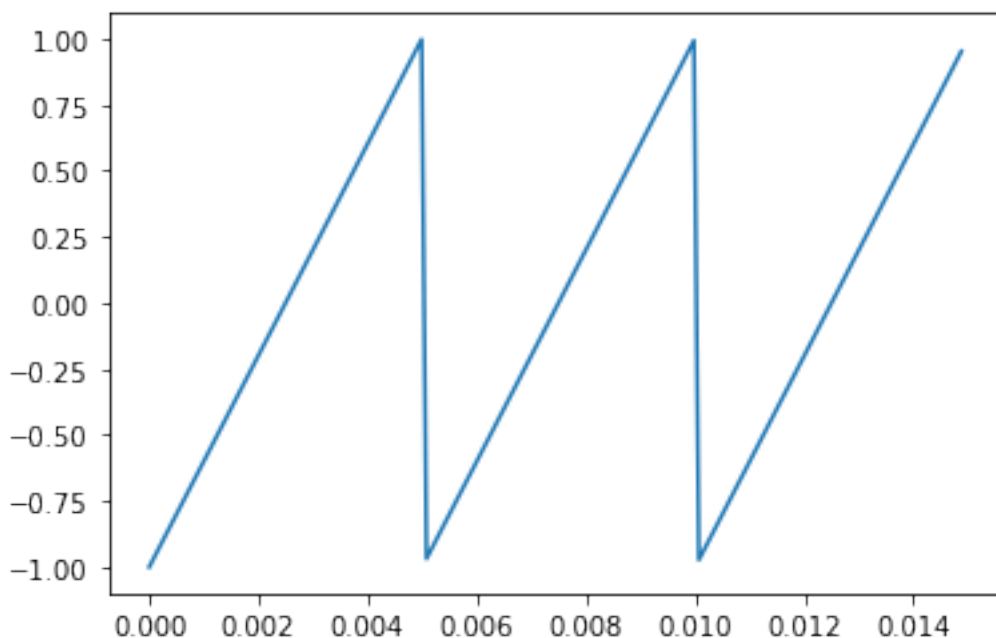


Рисунок 8.7. Пилообразный сигнал

```
1 M = 16
2 std = 2
3
4 g = scipy.signal.gaussian(M,std)
5 br = np.bartlett(M)
6 bl = np.blackman(M)
7 hm = np.hamming(M)
8 hn = np.hanning(M)
9
10 array = [g,br,bl,hm,hn]
11 labels = ['gauss','bartlett','blackman','hamming','hanning']
12
13 for elem, label in zip(array,labels):
14     elem /= sum(elem)
15     plt.plot(elem,label=label)
16 plt.legend()
```

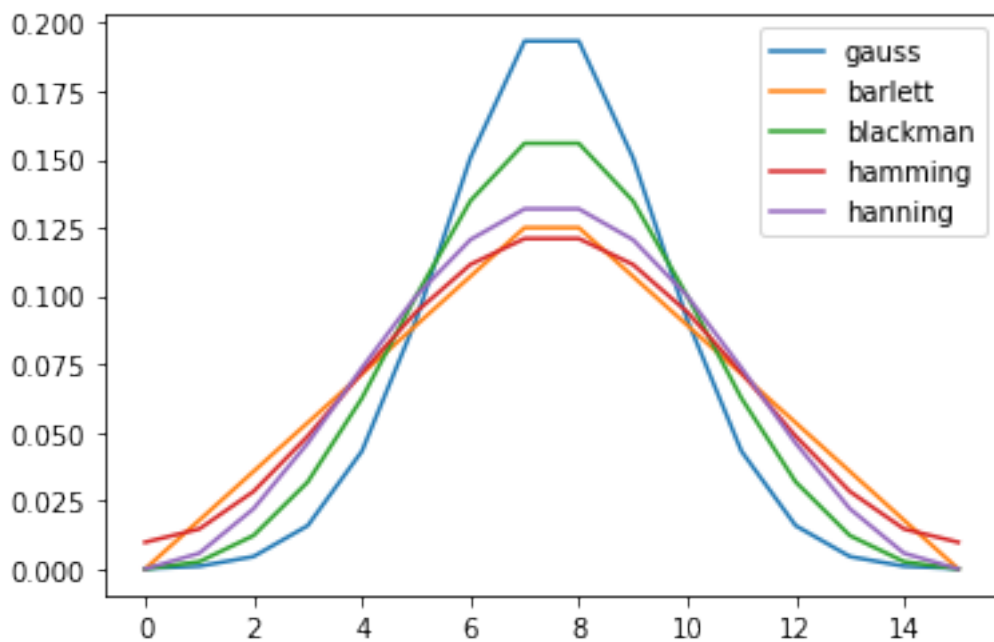


Рисунок 8.8. Применение различных окон на выбранный сигнал

Дополним окна нулями и выведем ДПФ:

```

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

```

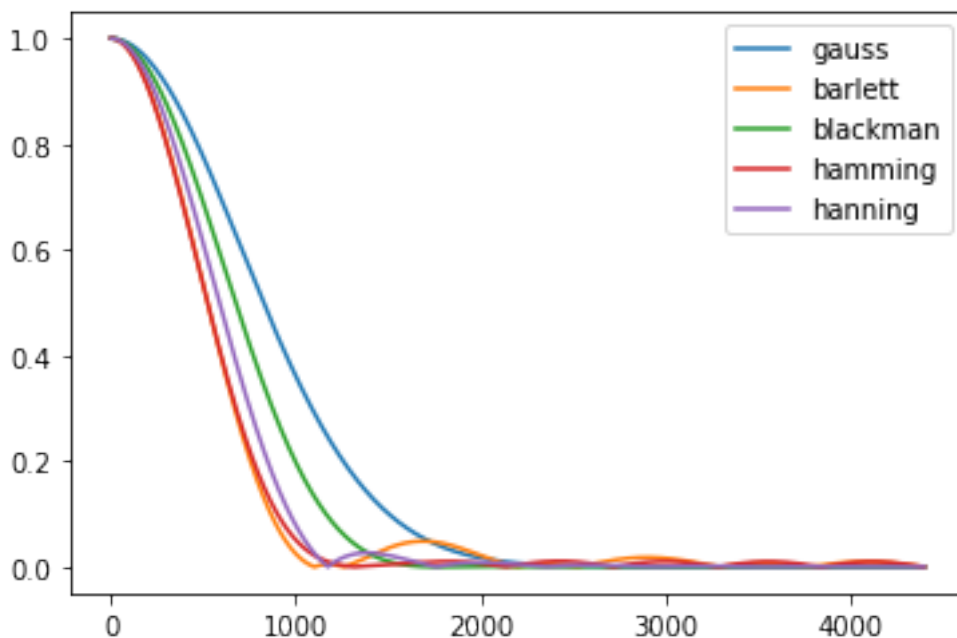


Рисунок 8.9. Применение различных окон на выбранный сигнал

Хэнинг лучше всего подоёдет для фильтрации низких частот, я так решил из-за колокольчиков.

```

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

```

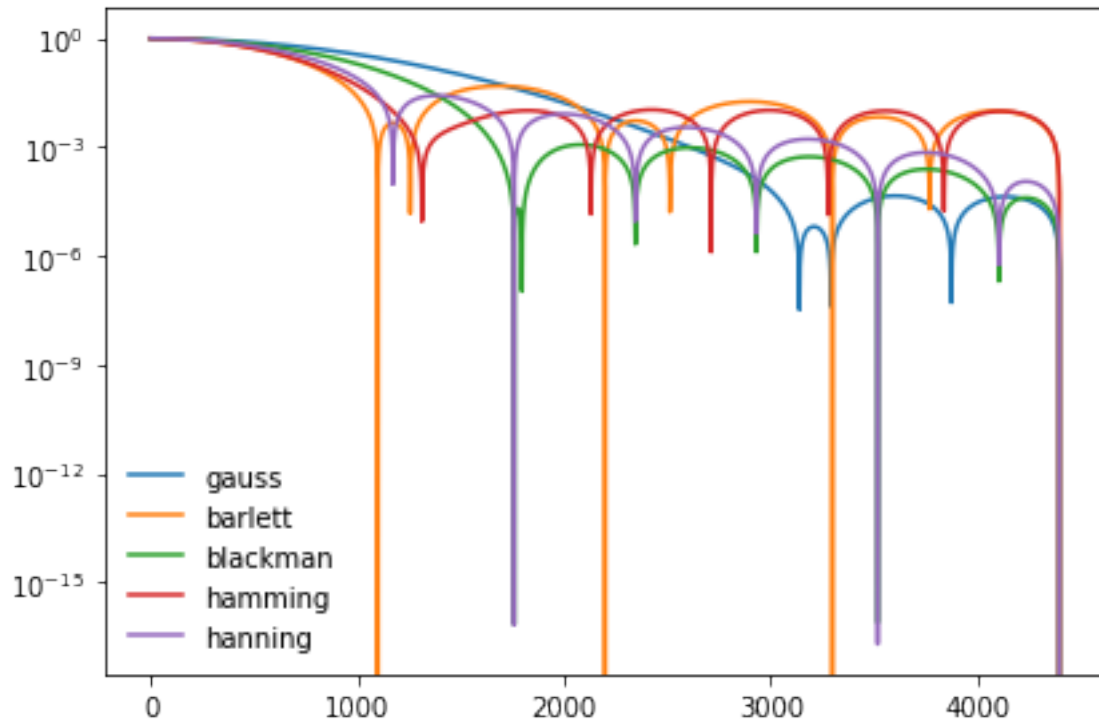


Рисунок 8.10. Логорифмический масштаб

Смотря на логорифмический масштаб можно сделать такой же вывод.

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 сегментам сигнала. В упражнениях были исследованы различные свойства данных явлений.

9. Дифференциация и интеграция

9.1. Упражнение 1

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

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

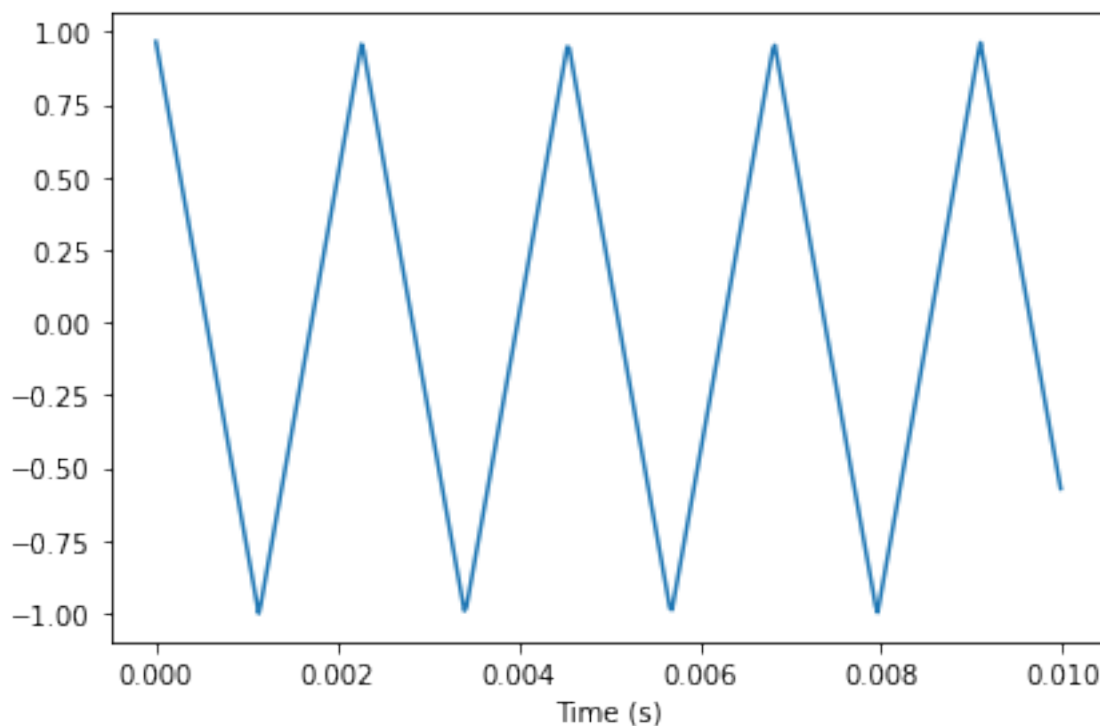


Рисунок 9.1. График сигнала

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

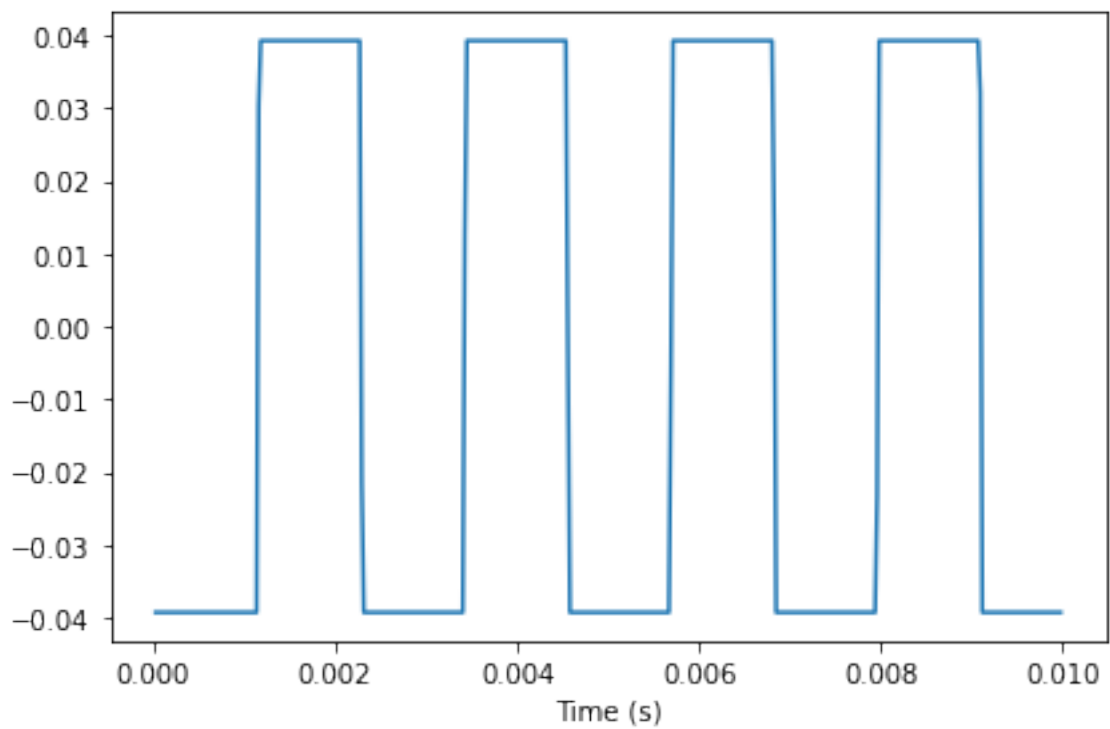


Рисунок 9.2. Сигнал после применения diff

В итоге получили прямоугольный сигнал с такой же частотой.

```

1 differentiate_wave = wave.make_spectrum().differentiate().make_wave()
2 differentiate_wave.plot()
3 decorate(xlabel='Time (s)')

```

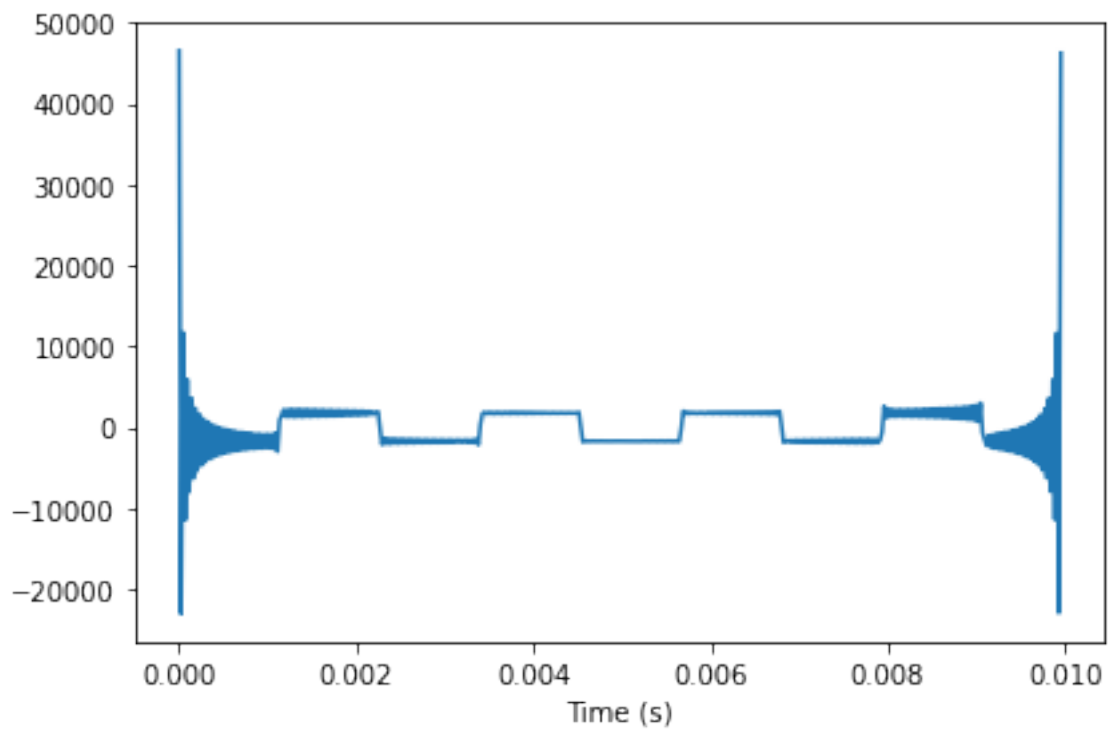


Рисунок 9.3. Сигнал после применения differentiate

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 прямоугольного сигнала, примените `integrate` и напечатайте результат. Преобразуйте спектр обратно в сигнал и напечатайте его. Есть различия в воздействии `cumsum` и `integrate` на этот сигнал?

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

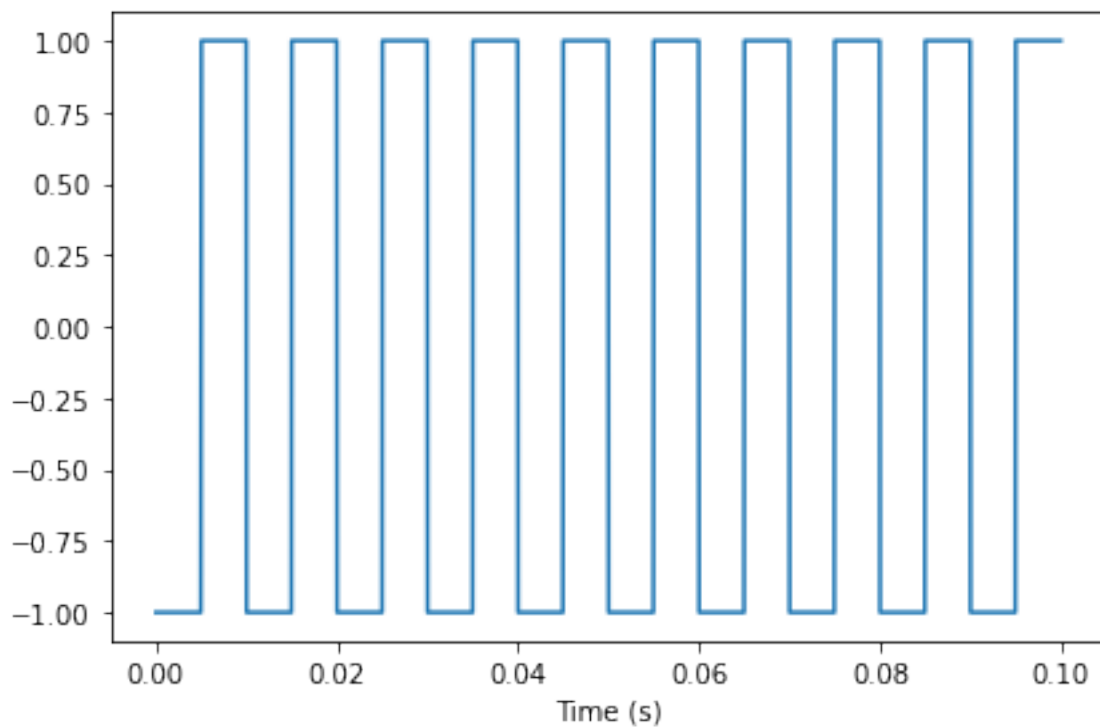


Рисунок 9.4. Рассматриваемый сигнал

Кумулятивная сумма:

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

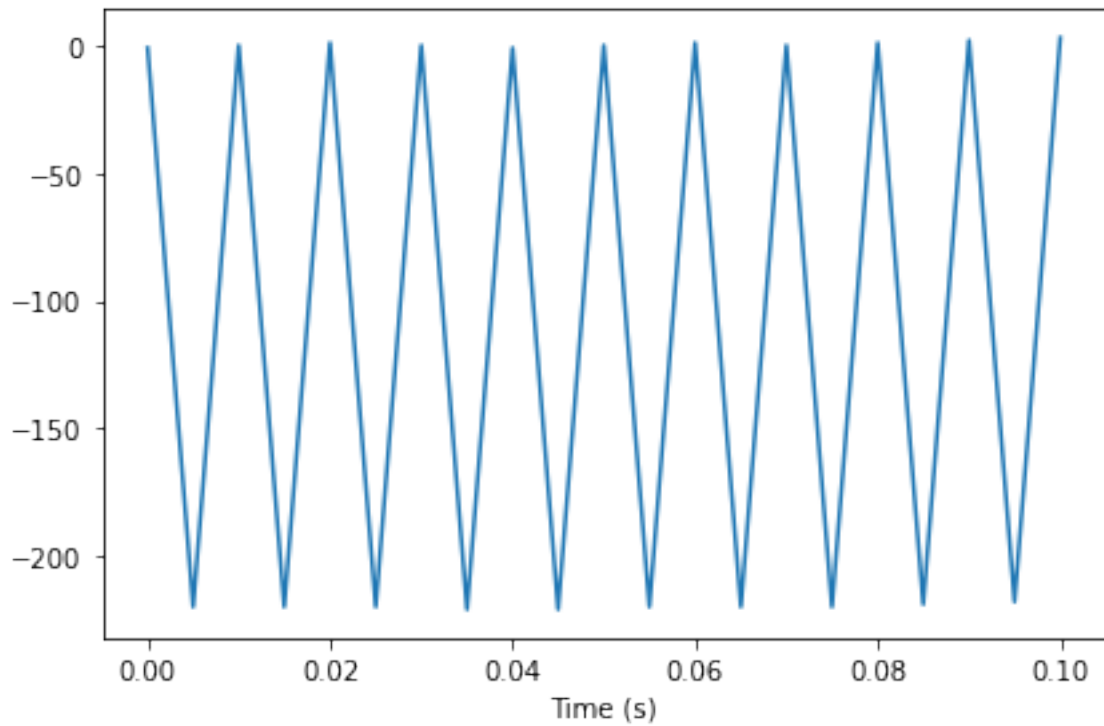



Рисунок 9.5. Рассматриваемый сигнал после применения cumsum

Получили треугольный сигнал (скорей всего это он, но если чуть-чуть изменить сигнал то фаза смещается).

Теперь интеграл спектра:

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

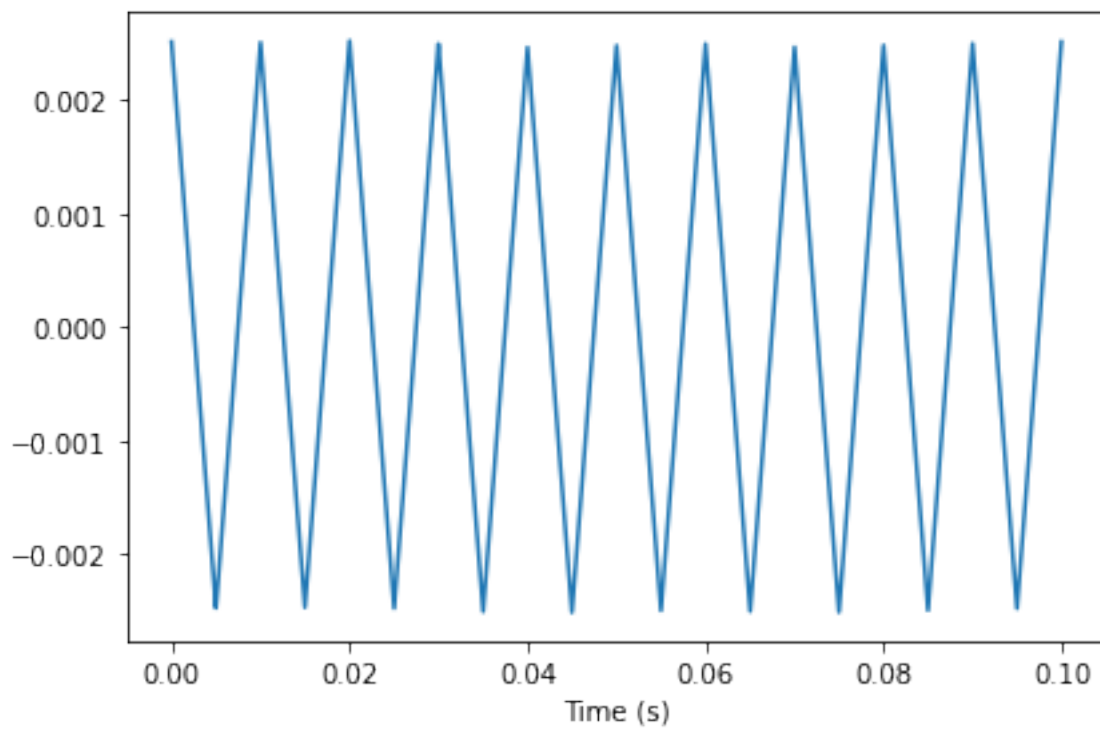


Рисунок 9.6. Рассматриваемый сигнал после применения integrate

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

```

1 cumsum_wave.unbias()
2 cumsum_wave.normalize()
3 int_wave.normalize()
4 cumsum_wave.plot()
5 int_wave.plot()

```

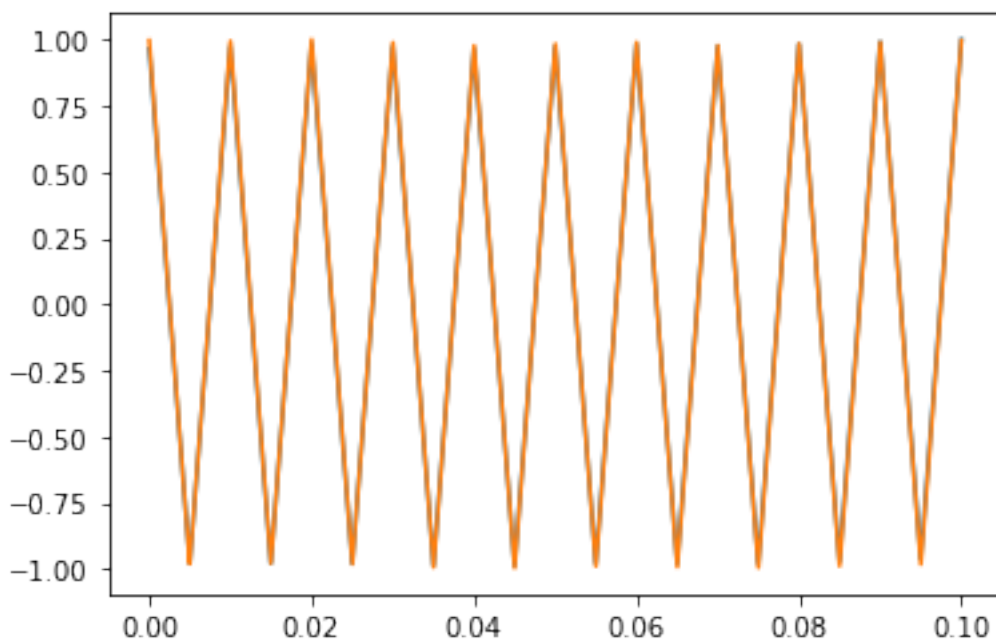


Рисунок 9.7. Сравнение полученных графиков

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`. Напечатайте результирующий сигнал и его спектр. Какова математическая форма сигнала? Почему он напоминает синусоиду?

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

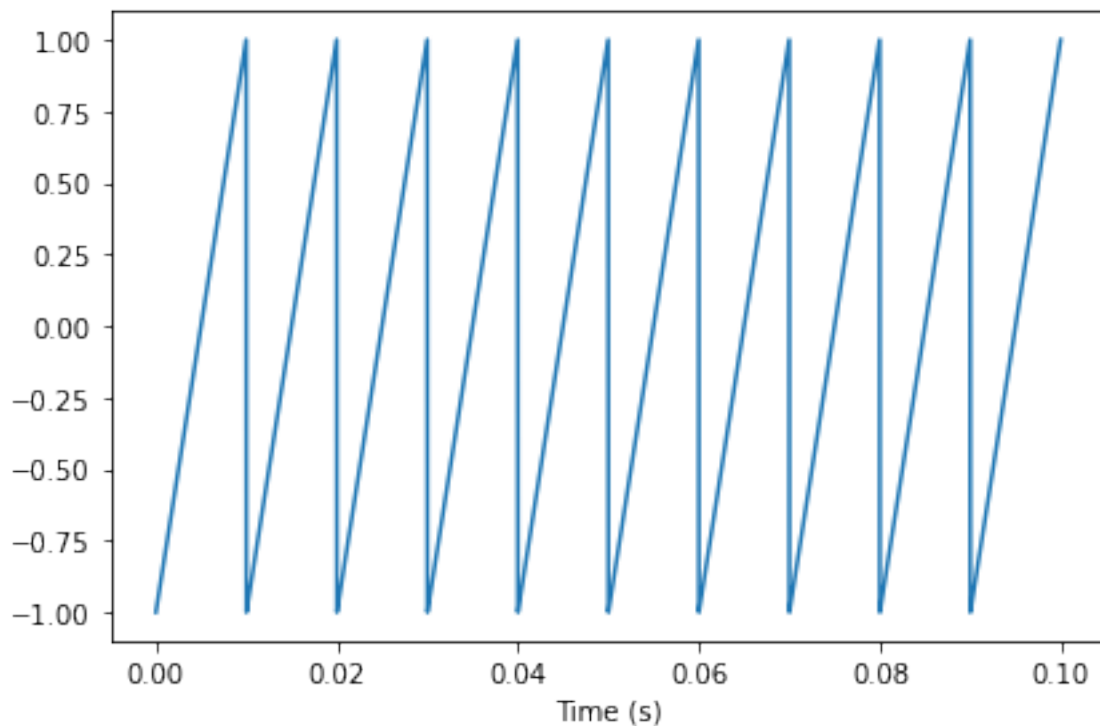


Рисунок 9.8. Пилообразный сигнал

```
1 spectrum = wave.make_spectrum().integrate().integrate()
2 spectrum.hs[0] = 0
3
4 wave1 = spectrum.make_wave()
5 wave1.plot()
6 decorate(xlabel='Time (s)')
```

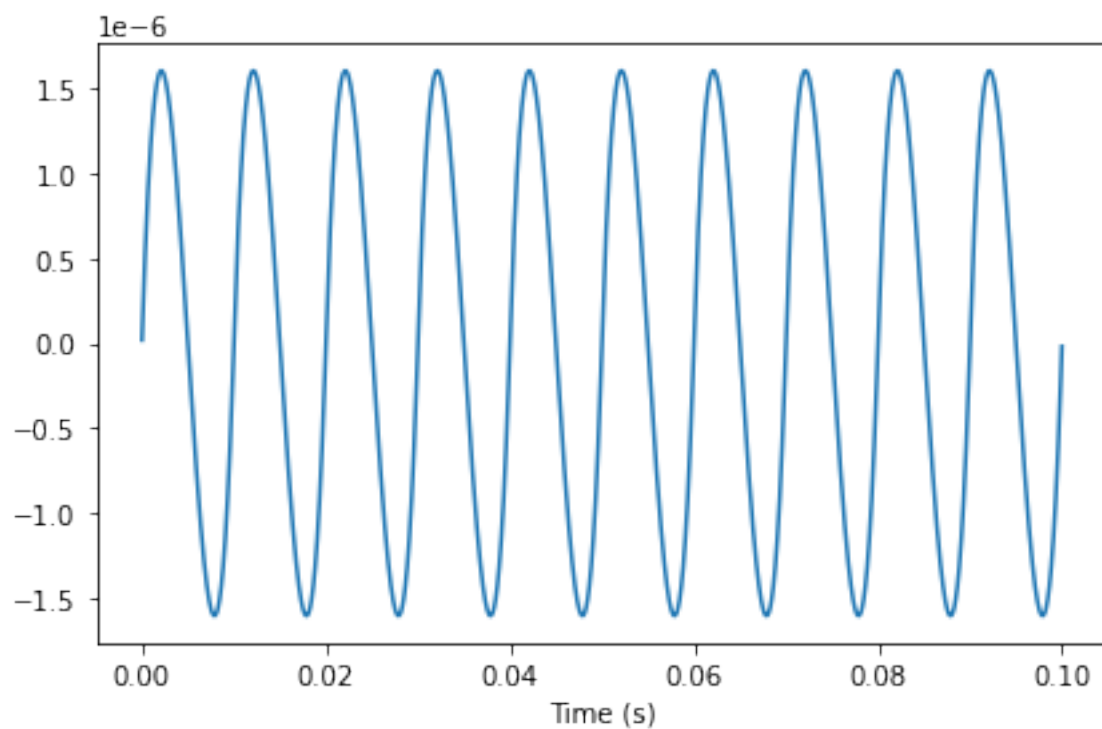


Рисунок 9.9. Изменённый сигнал

Сигнал действительно напоминает синусоиду. Причиной стала фильтрация низких частот, за исключением основной.

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

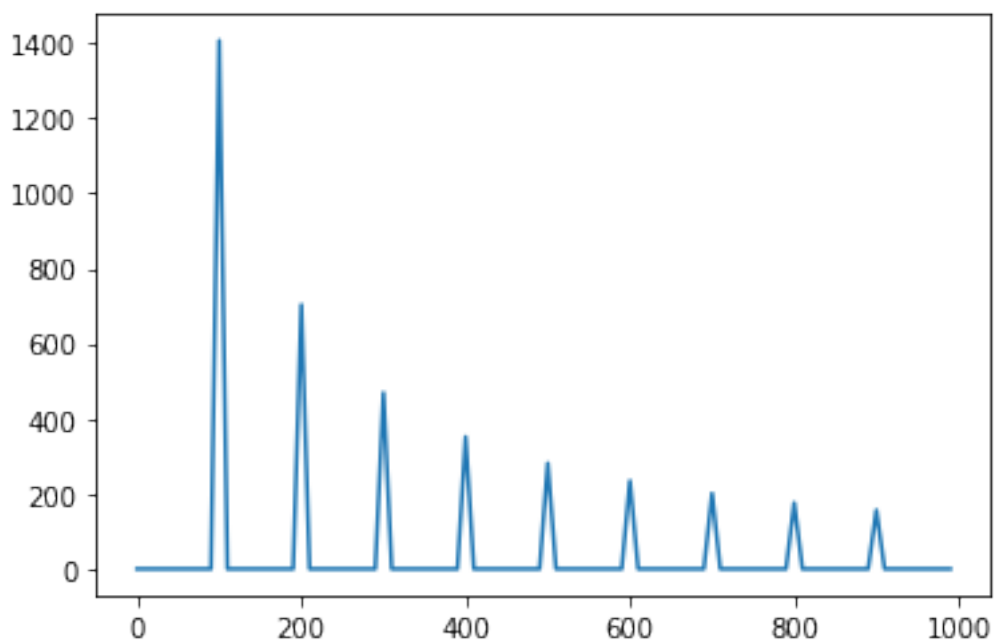


Рисунок 9.10. Спектр исходного сигнала

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

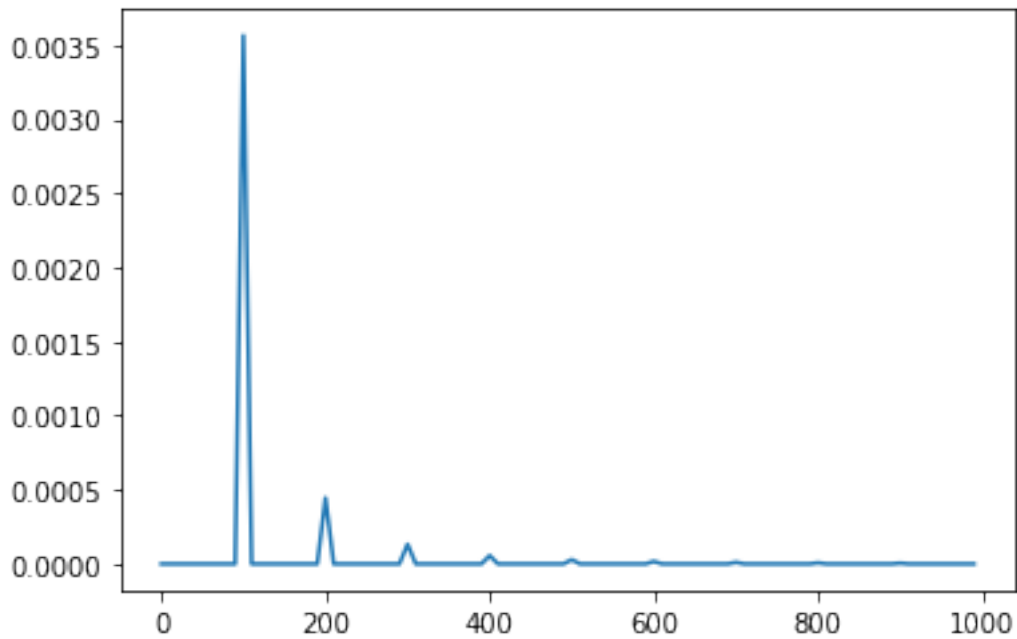


Рисунок 9.11. Спектр нового сигнала

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? Распечатайте фильтры, соответствующие второй разнице и второй производной. Сравните их.

Тут надо тонко подобрать параметры, чтобы сигнал красиво отображался при таком маленьком framerate.

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

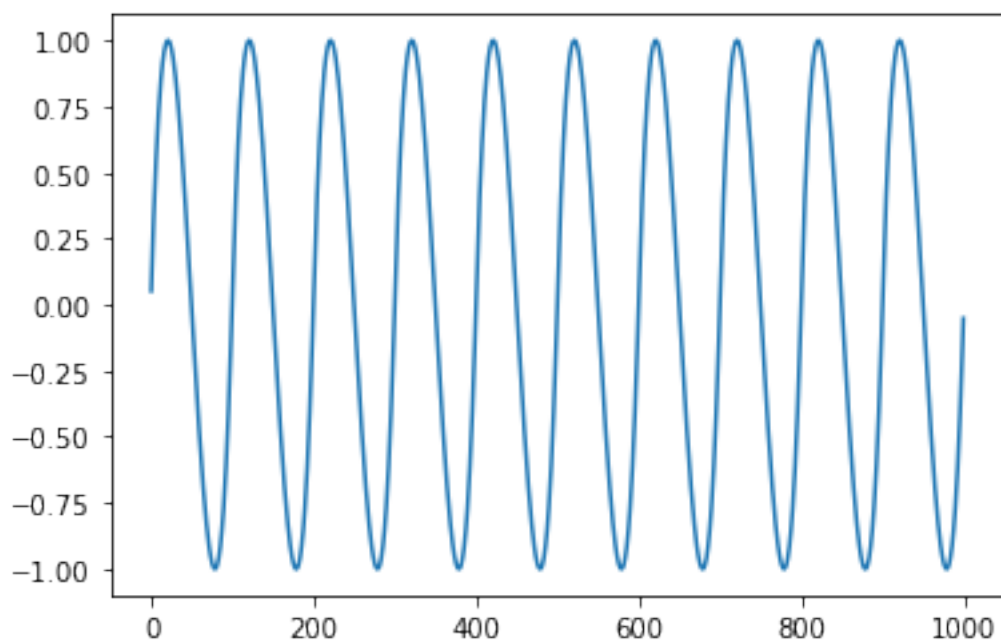


Рисунок 9.12. Кубический сигнал

Первая разность - параболы.

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

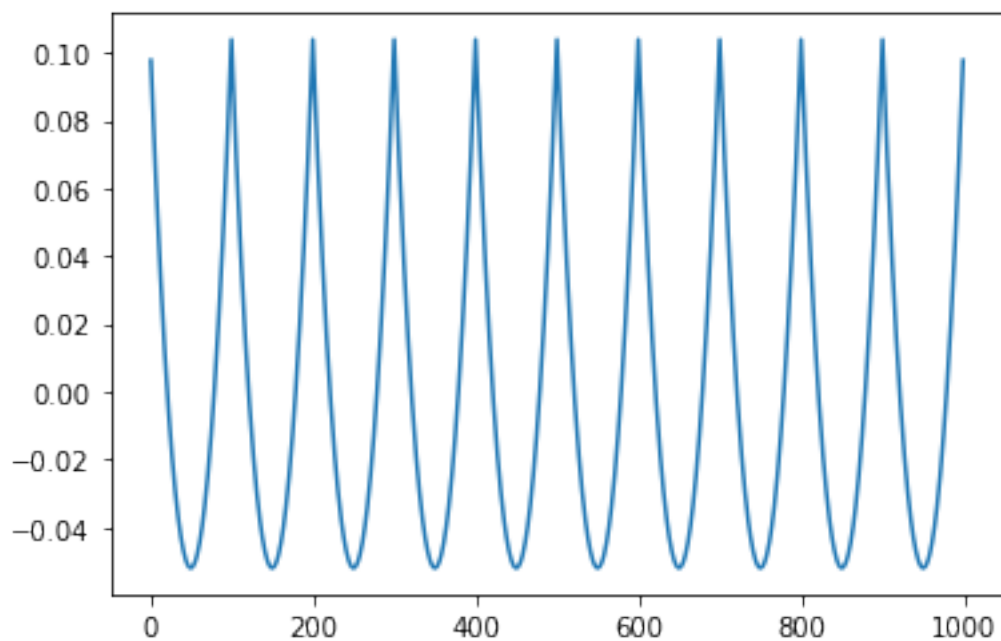


Рисунок 9.13. Первая разность

Вторая разность - пилообразный сигнал.

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

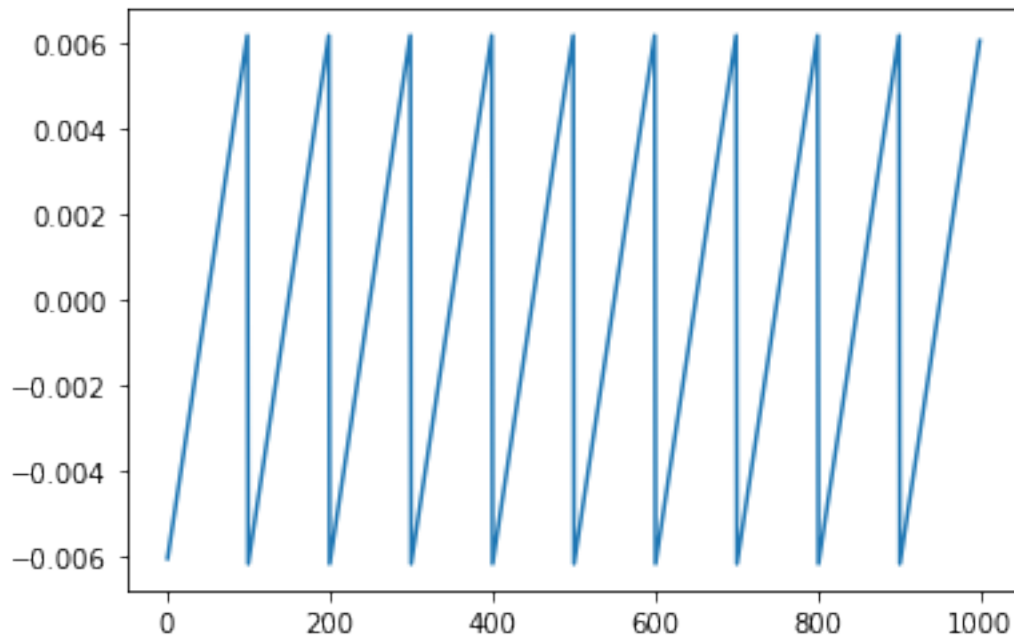


Рисунок 9.14. Вторая разность

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

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

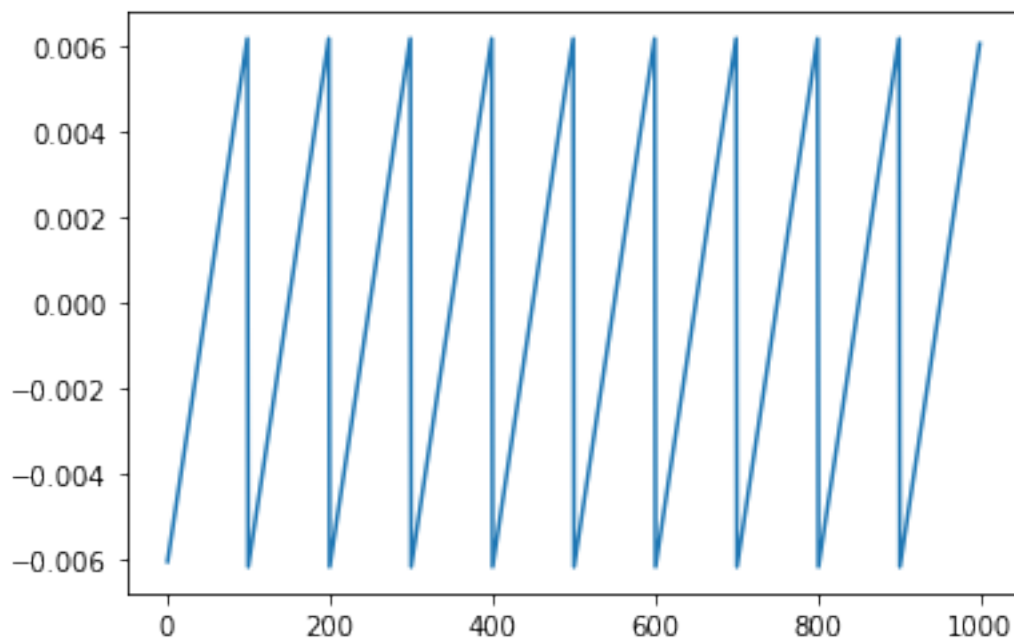


Рисунок 9.15. Полученный сигнал со звоном

Фильтры:

```

1 diff_filter.plot(label='2nd diff')
2 deriv_filter.plot(label='2nd deriv')
3
4 decorate(xlabel='Frequency (Hz)',
5          ylabel='Amplitude ratio')

```

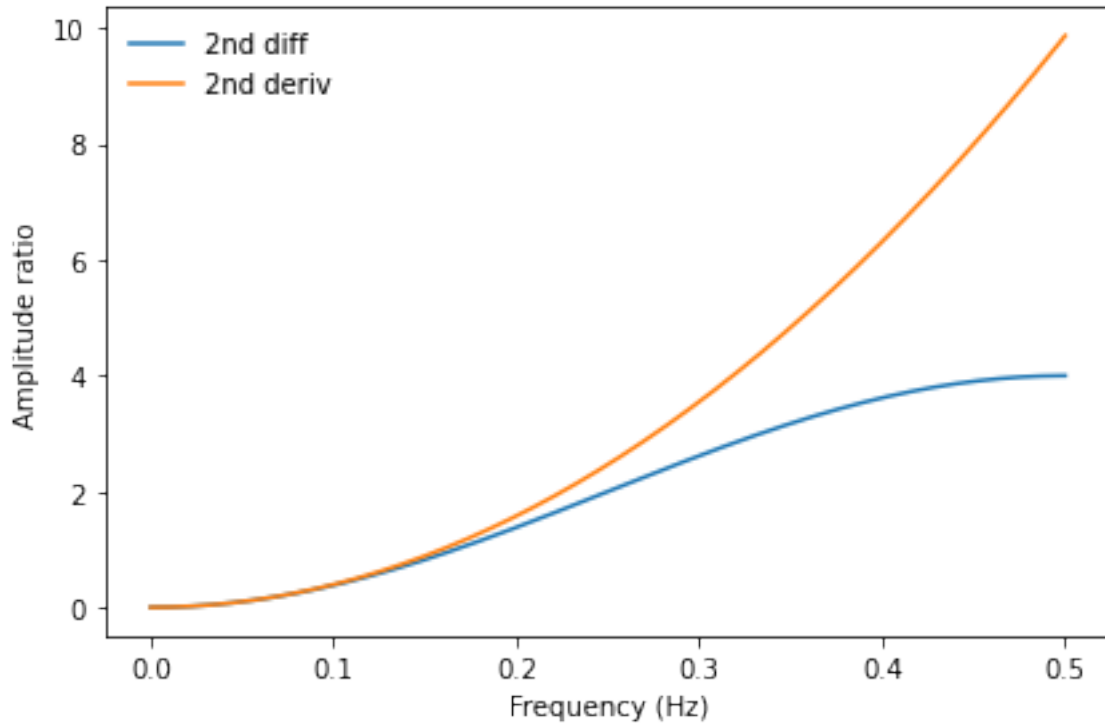


Рисунок 9.16. Полученные фильтры

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

9.5. Вывод

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

10. Сигналы и системы

10.1. Упражнение 1

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

```
1 if not os.path.exists('180960__kleeb__gunshot.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/180960
      __kleeb__gunshot.wav
3 response = read_wave('180960__kleeb__gunshot.wav')
4
5 start = 0.12
6 response = response.segment(start=start)
7 response.shift(-start)
8
9 response.normalize()
10 response.plot()
11 decorate(xlabel='Time (s)')
```

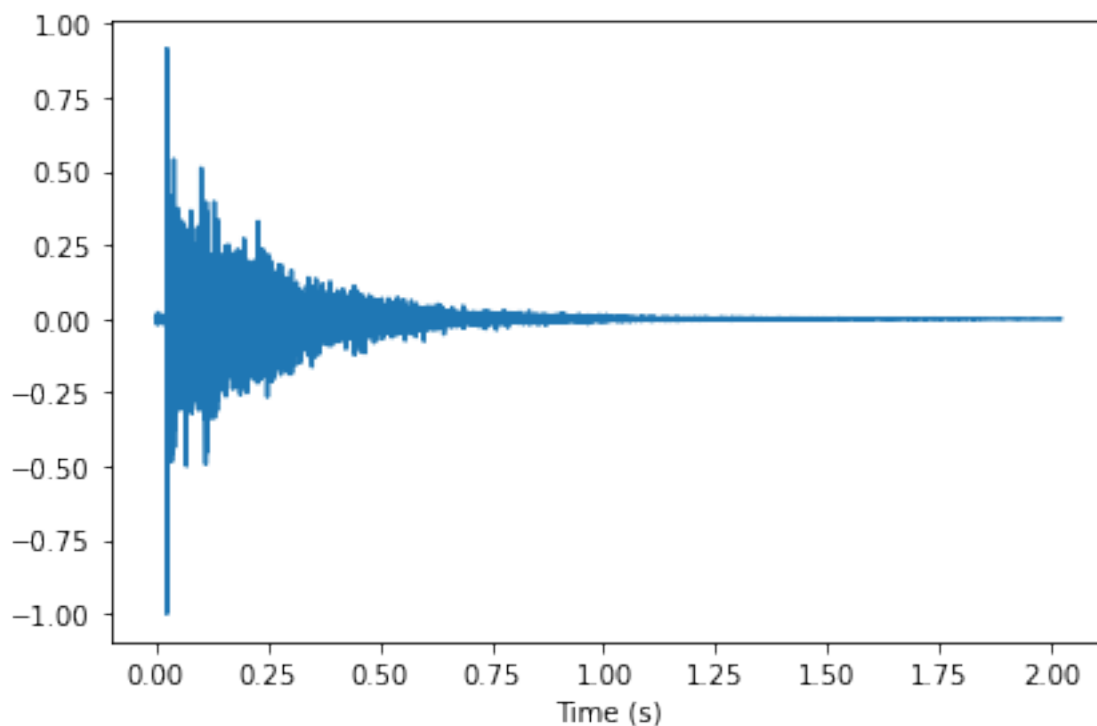


Рисунок 10.1. Сигнал

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

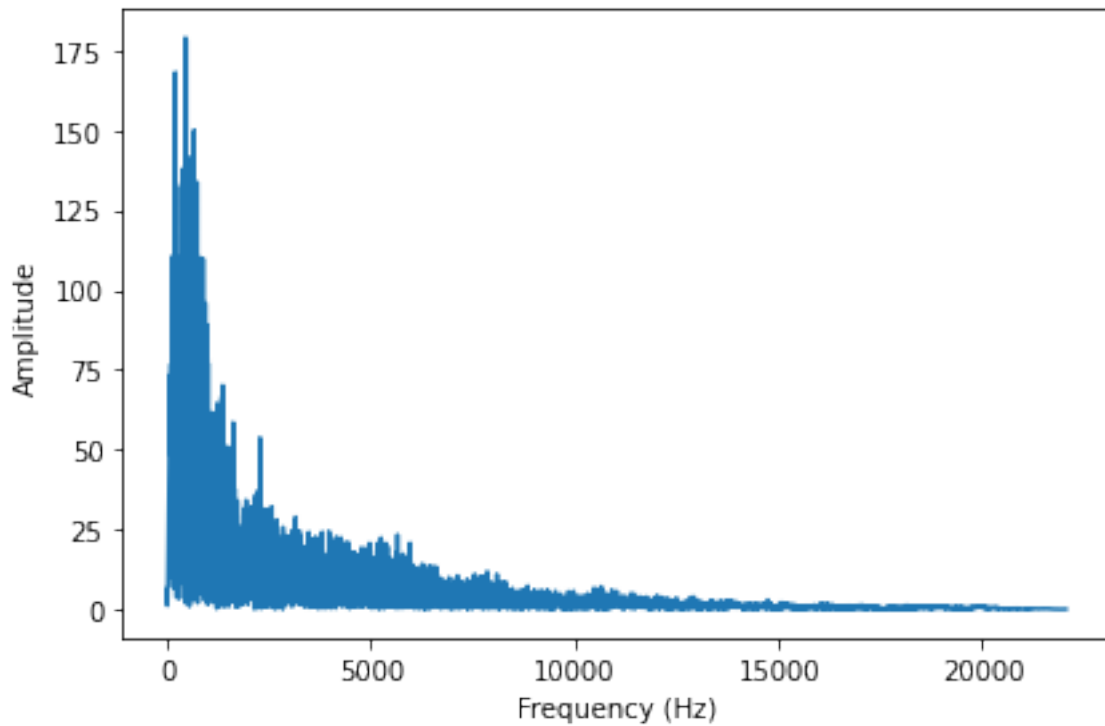


Рисунок 10.2. Спектр сигнала

Теперь перейдём к самой записе:

```
1 if not os.path.exists('92002__jcveliz__violin-original.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/92002
      __jcveliz__violin-original.wav
3
4 violin = read_wave('92002__jcveliz__violin-original.wav')
5
6 start = 0.11
7 violin = violin.segment(start=start)
8 violin.shift(-start)
9
10 violin.truncate(len(response))
11 violin.normalize()
12 violin.plot()
13 decorate(xlabel='Time (s)')
```

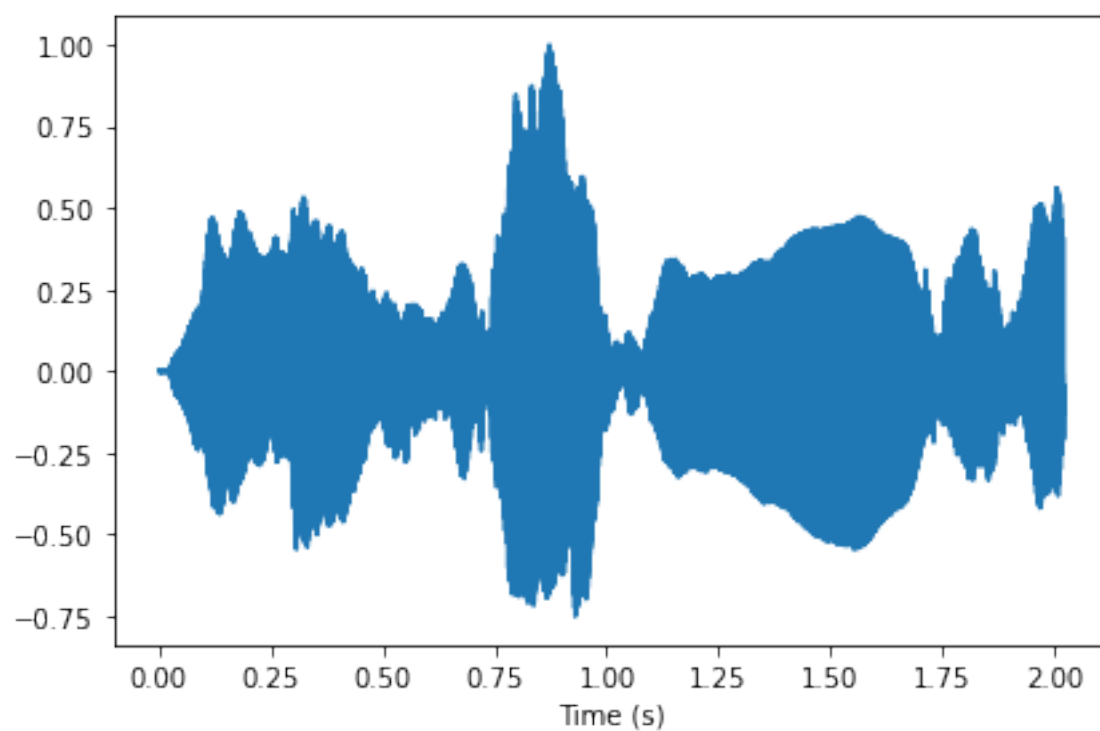


Рисунок 10.3. График сигнала

Произведём умножения спектров и посмотрим на результат:

```
1 spectrum = violin.make_spectrum()  
2 wave = (spectrum * transfer).make_wave()  
3 wave.normalize()  
4 wave.plot()
```

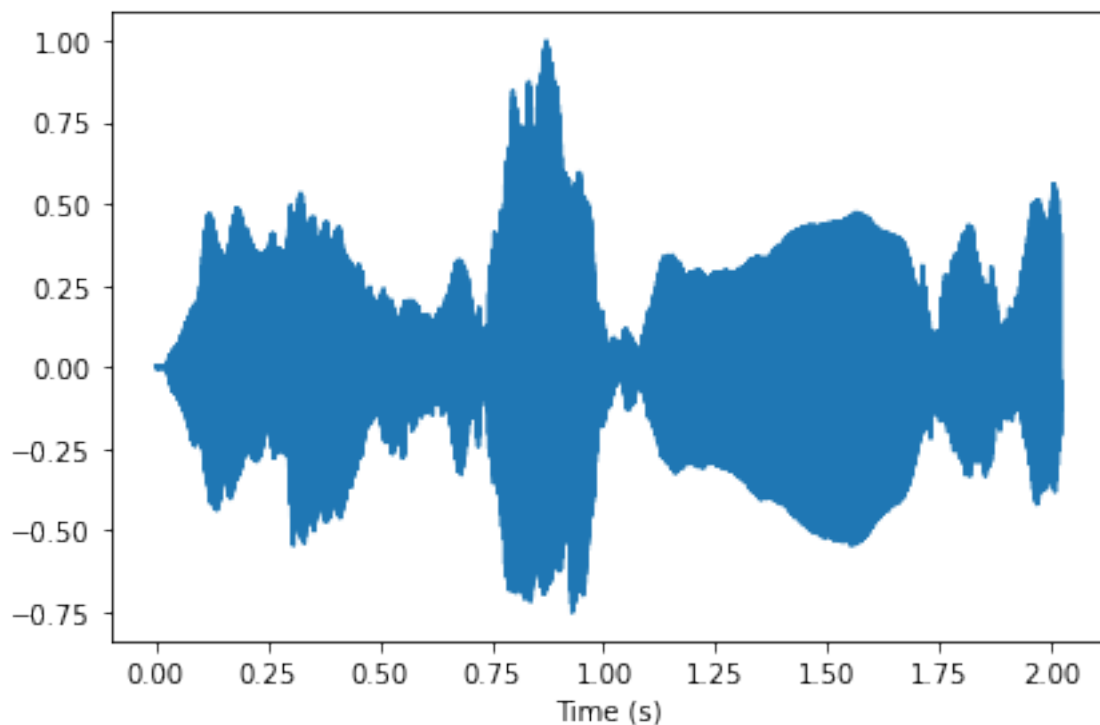


Рисунок 10.4. График получившегося сигнала

Проблема с нотой решена.

10.2. Упражнение 2

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

Я взял себе такую характеристику: https://www.openair.hosted.york.ac.uk/?page_id=745, а потом ничего не заработало и на выходе был дикий шум, так что я взял характеристику из учебника:

```

1 if not os.path.exists('stalbens_a_mono.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/
      stalbens_a_mono.wav
3
4 response = read_wave('stalbens_a_mono.wav')
5
6 start = 0
7 duration = 5
8 response = response.segment(duration=duration)
9 response.shift(-start)
10
11 response.normalize()
12 response.plot()
13 decorate(xlabel='Time (s)')
14 decorate(xlabel='Time (s)')

```

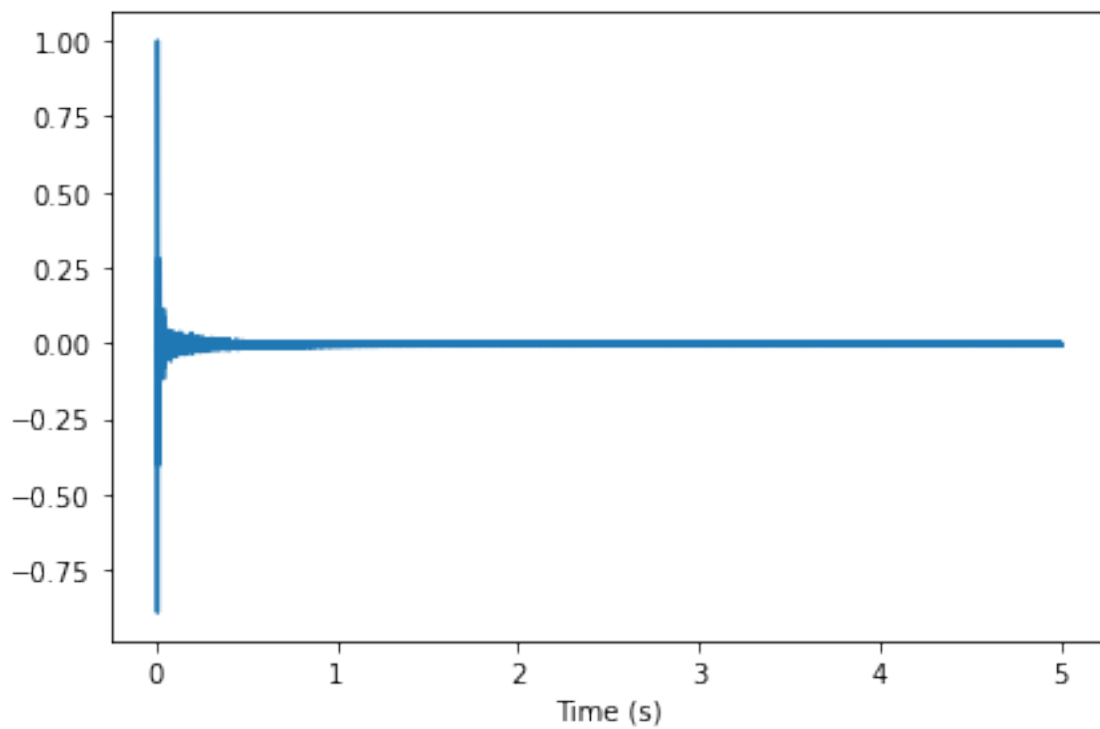


Рисунок 10.5. График загруженного сигнала

```

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

```

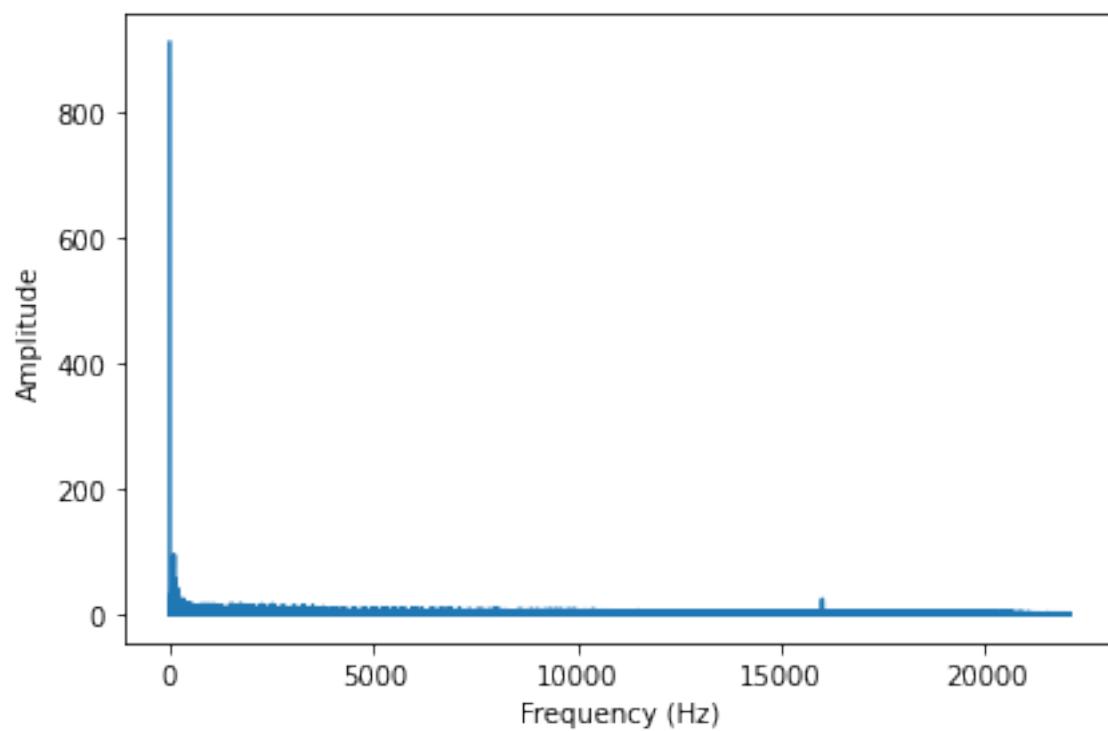


Рисунок 10.6. ДПФ импульсной характеристики

Теперь самое интересное, промоделируем запись в пространстве.

```
1 if not os.path.exists('164718__bradovic__piano.wav'):
2     !wget https://github.com/wooftown/spbstu-telecom/raw/main/Content/164718
      __bradovic__piano.wav
3
4 wave = read_wave('164718__bradovic__piano.wav')
5
6 start = 0.0
7 wave = wave.segment(start=start)
8 wave.shift(-start)
9
10 wave.truncate(len(response))
11 wave.normalize()
12 wave.plot()
13 decorate(xlabel='Time (s)')
```

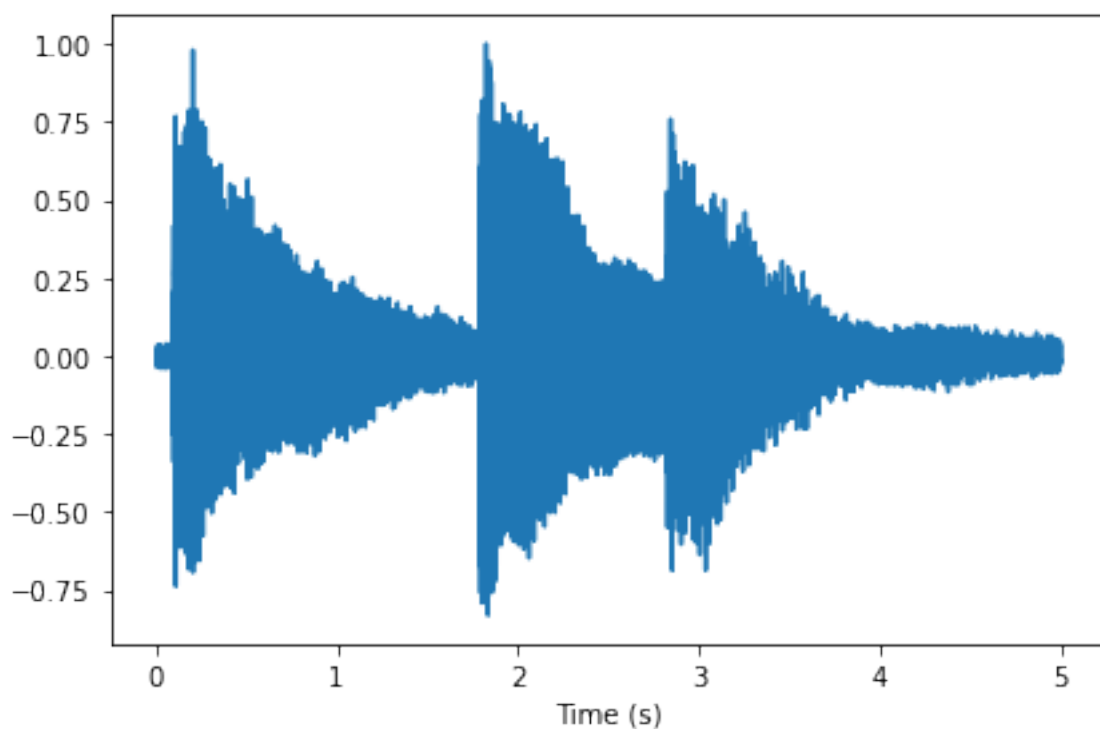


Рисунок 10.7. Сигнал звука пианино

С использованием свёртки:

```
1 convolved2 = wave.convolve(response)
2 convolved2.normalize()
3 convolved2.make_audio()
```

Через умножение:

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

В результате мы добились желаемой цели.

10.3. Вывод

В данной работе были рассмотрены 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

При взятии выборок из сигнала при слишком низкой чистоте кадров составляющие, большие частоты заворота дадут биения. В таком случае эти компоненты не отфильтруешь, поскольку 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 идентичен отфильтрованному сигналу.

```
1 if not os.path.exists('263868__kevcio__amen-break-a-160-bpm.wav'):
2     !wget https://github.com/AllenDowney/ThinkDSP/raw/master/code/263868
      __kevcio__amen-break-a-160-bpm.wav
3
4 wave = read_wave('263868__kevcio__amen-break-a-160-bpm.wav')
5 wave.plot()
```

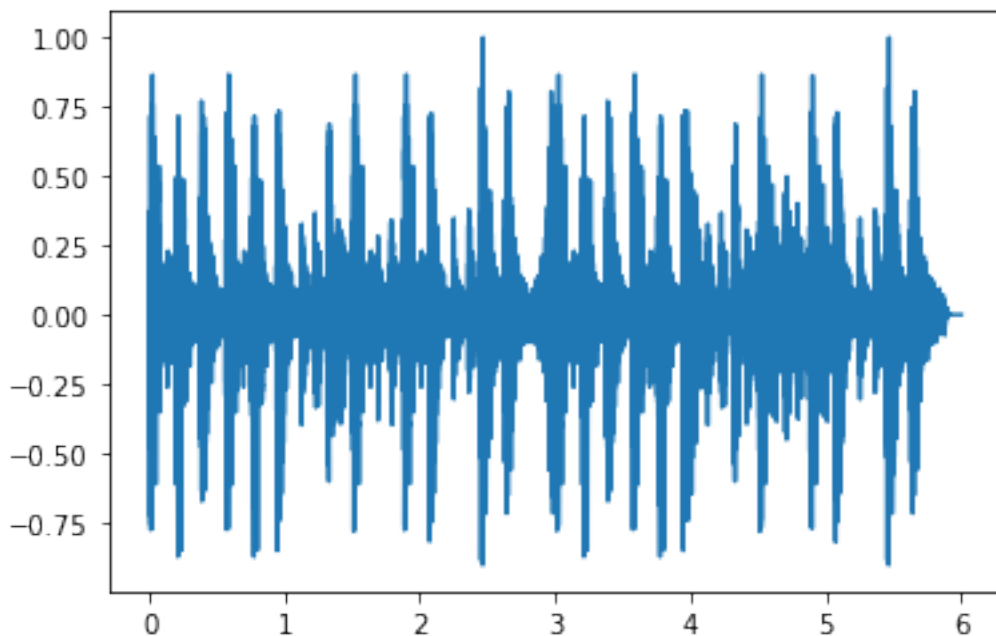


Рисунок 11.1. График "Соло на барабане"

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

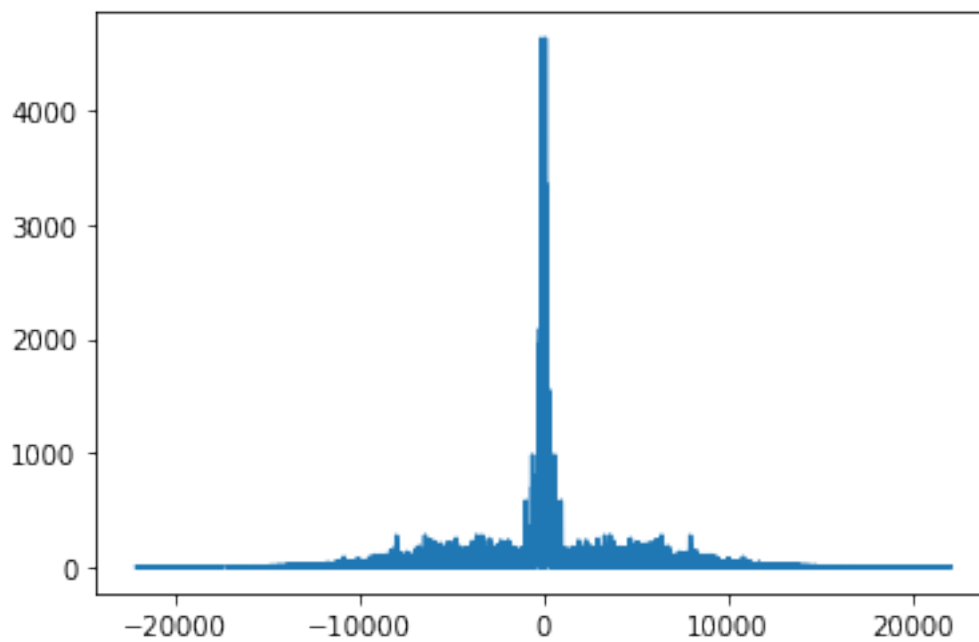



Рисунок 11.2. Спектр сигнала

Фильтр низких частот:

```

1 factor = 3
2 framerate = wave.framerate / factor
3 cutoff = framerate / 2 - 1
4
5 spectrum.low_pass(cutoff)
6 spectrum.plot()

```

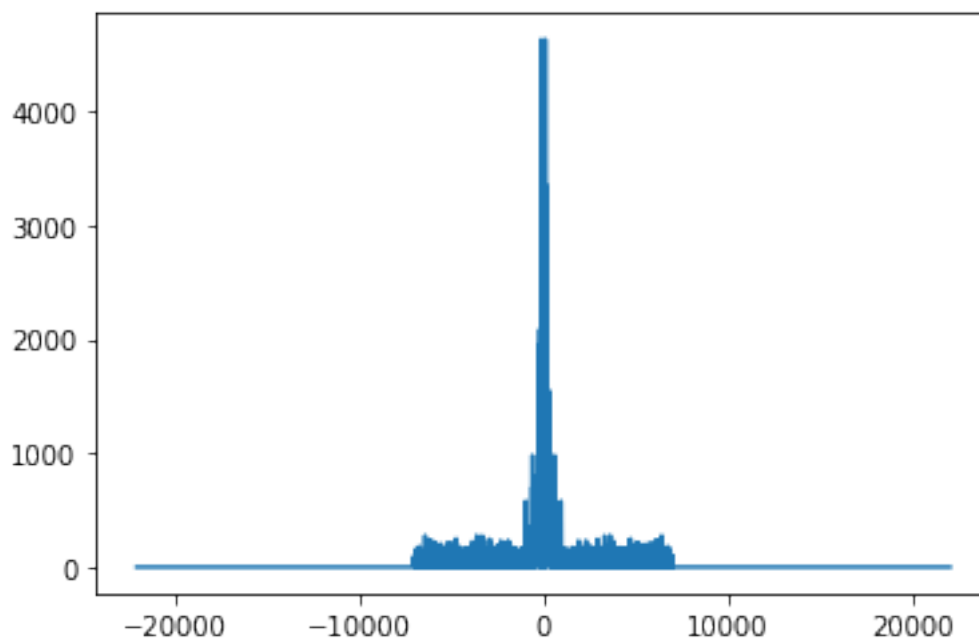


Рисунок 11.3. Отфильтрованный сигнал

Функция сэмплирования:

```

1 def sample(wave, factor):
2     """Simulates sampling of a wave.
3
4     wave: Wave object
5     factor: ratio of the new framerate to the original
6     """
7     ys = np.zeros(len(wave))
8     ys[::factor] = wave.ys[::factor]
9     return Wave(ys, framerate=wave.framerate)

```

```

1 sampled_spectrum = sampled.make_spectrum(full=True)
2 sampled_spectrum.plot()

```

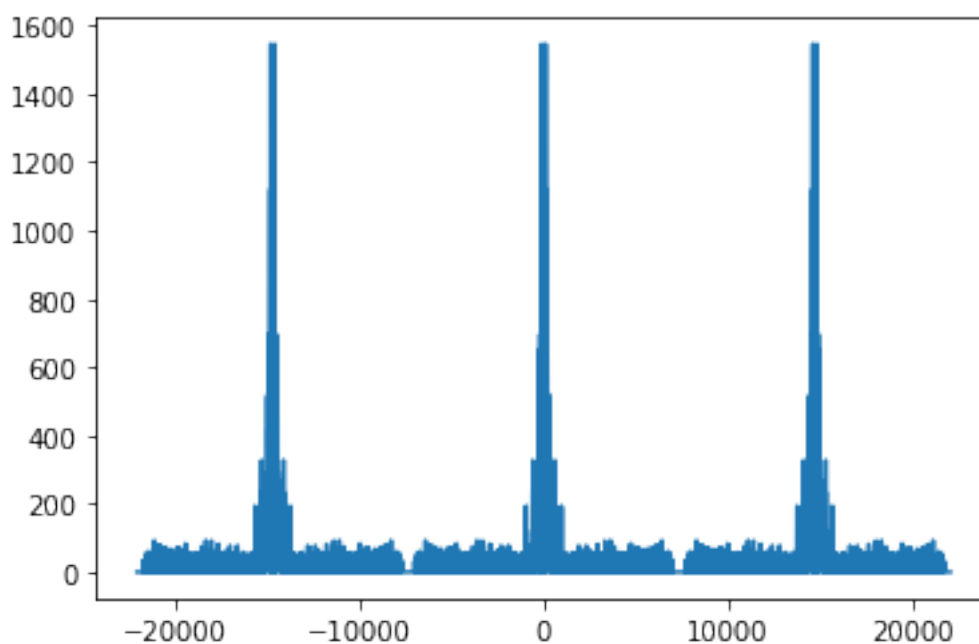


Рисунок 11.4. Получившийся спектр

Избавляемся от спектральных копий:

```

1 sampled_spectrum.low_pass(cutoff)
2 sampled_spectrum.plot()

```

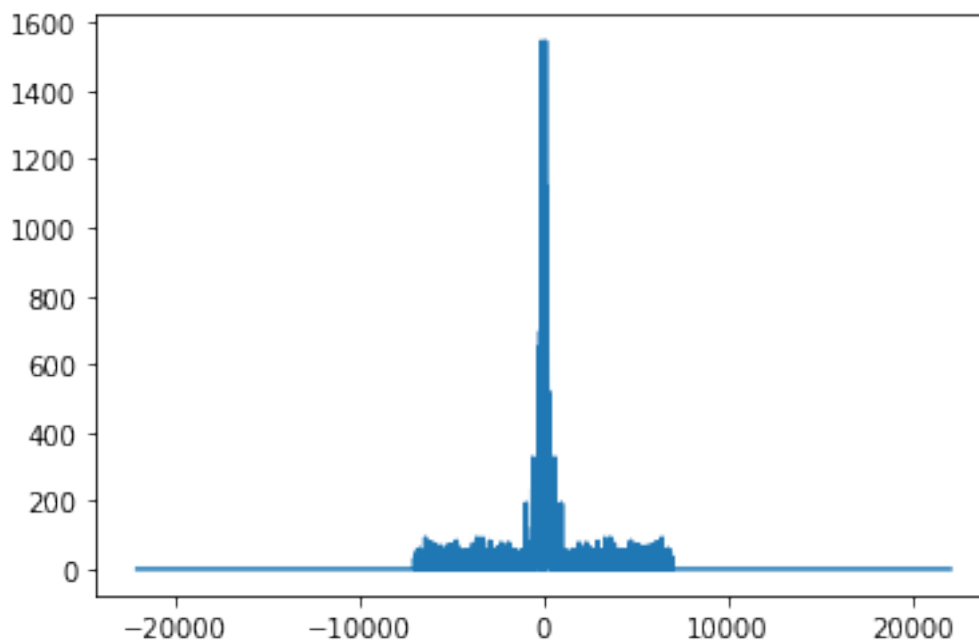


Рисунок 11.5. Результат избавления от копий

Получившийся звук отличается, чтобы понять в чём проблема сравним спектры:

```
1 spectrum.plot()
2 sampled_spectrum.plot()
```

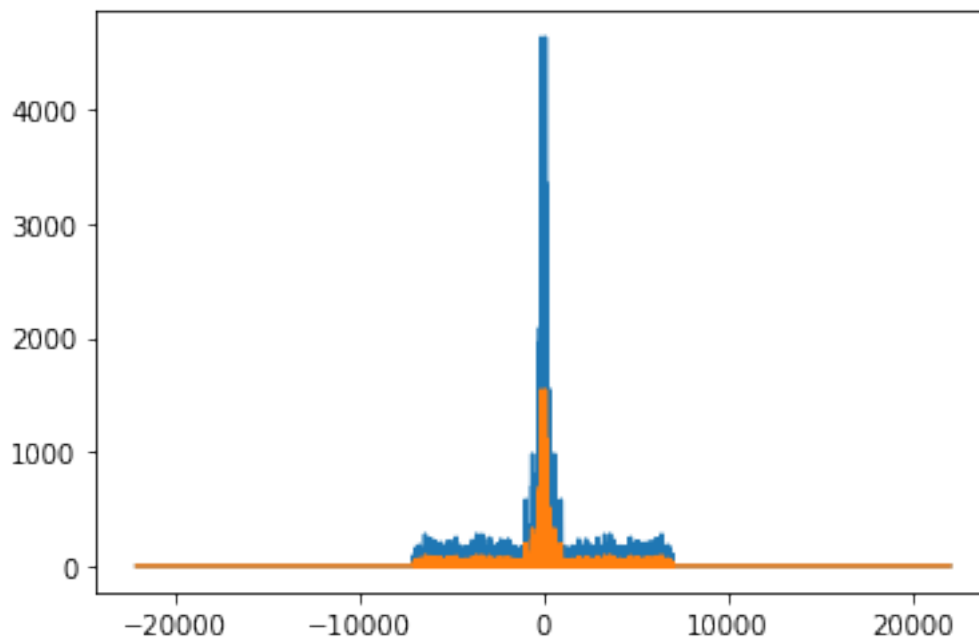


Рисунок 11.6. Сравнение спектров

Надо увеличить амплитуду в 3 раза:

```
1 sampled_spectrum.scale(factor)
2 sampled_spectrum.plot()
3 spectrum.plot()
```

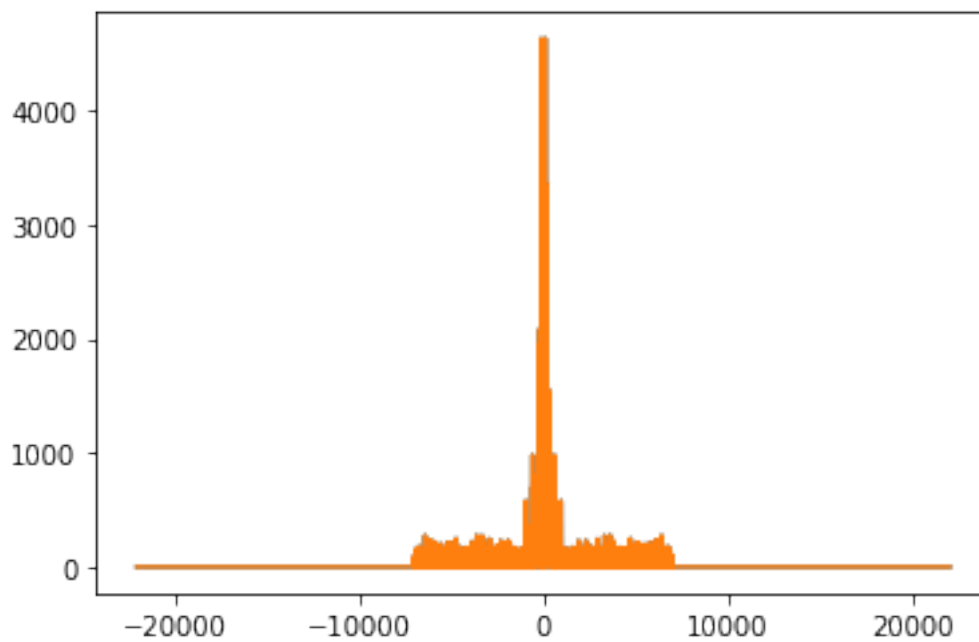


Рисунок 11.7. Сравнение спектров

Полученный звук, действительно, не сильно отличается от исходного.

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.

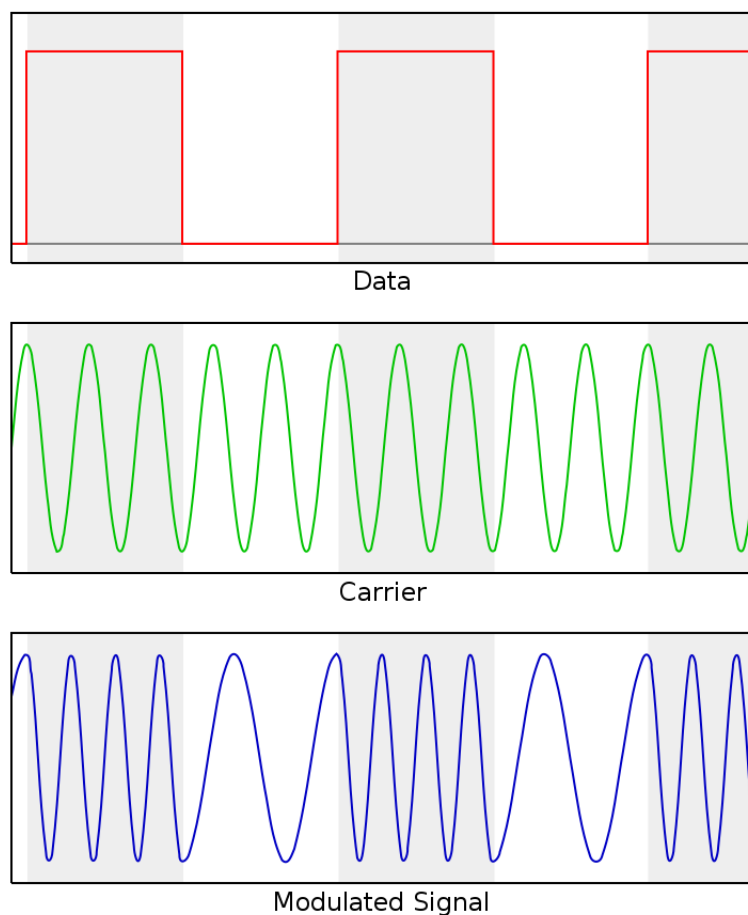


Рисунок 12.1. Пример FSK с двоичными данными

12.2. Схема в GNU Radio

Для изучения этого процесса в GNU Radio[1] необходимо построить следующую блок схему 12.2:

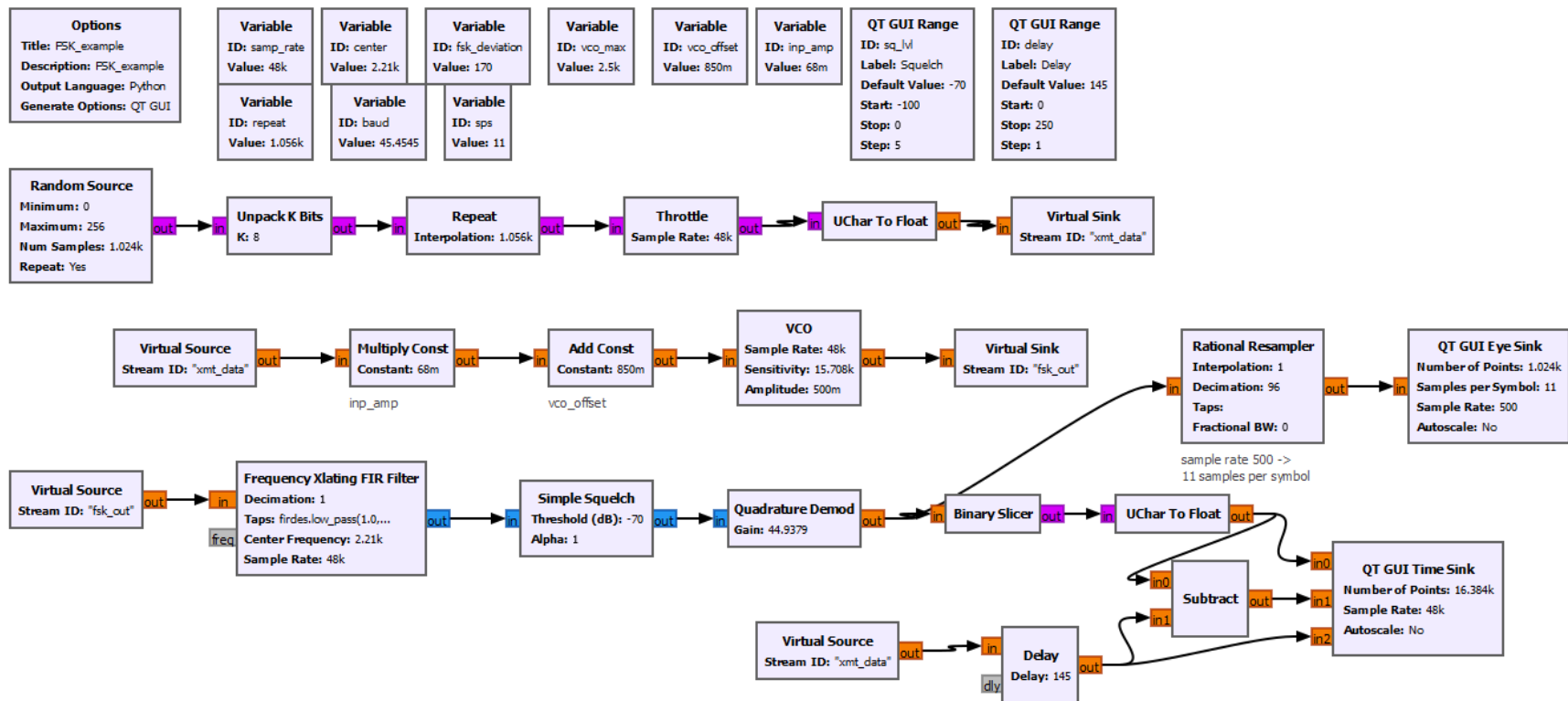


Рисунок 12.2. Схема FSK

Для лучшего понимания опишем используемые блоки:

- Variable - блок адресующий в уникальной переменной. При помощи ID можно передавать информацию через другие блоки.
- QT GUI Range - графический интерфейс для изменения заданной 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 повторений ввода, действующее как коэффициент 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 нужно 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 напряжением. Создает синусоиду на основе входной амплитуды.
- 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 информации в графическом интерфейсе.

В этом примере используется Baudot Radioteletype, следовательно битовое время = 22 миллисекунды. Получаем скорость передачи 45,4545. Коэффициент повторения равен $\text{samp_rate} * 0,022$.

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 = $(2 * \text{math.pi} * 2500 / 1) = 15708$. 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 $\text{inp_amp} = (1,0 * 0,068) + \text{vco_offset} = 0,918$, 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. Шумоподавител 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 и посмотрим что имеем.

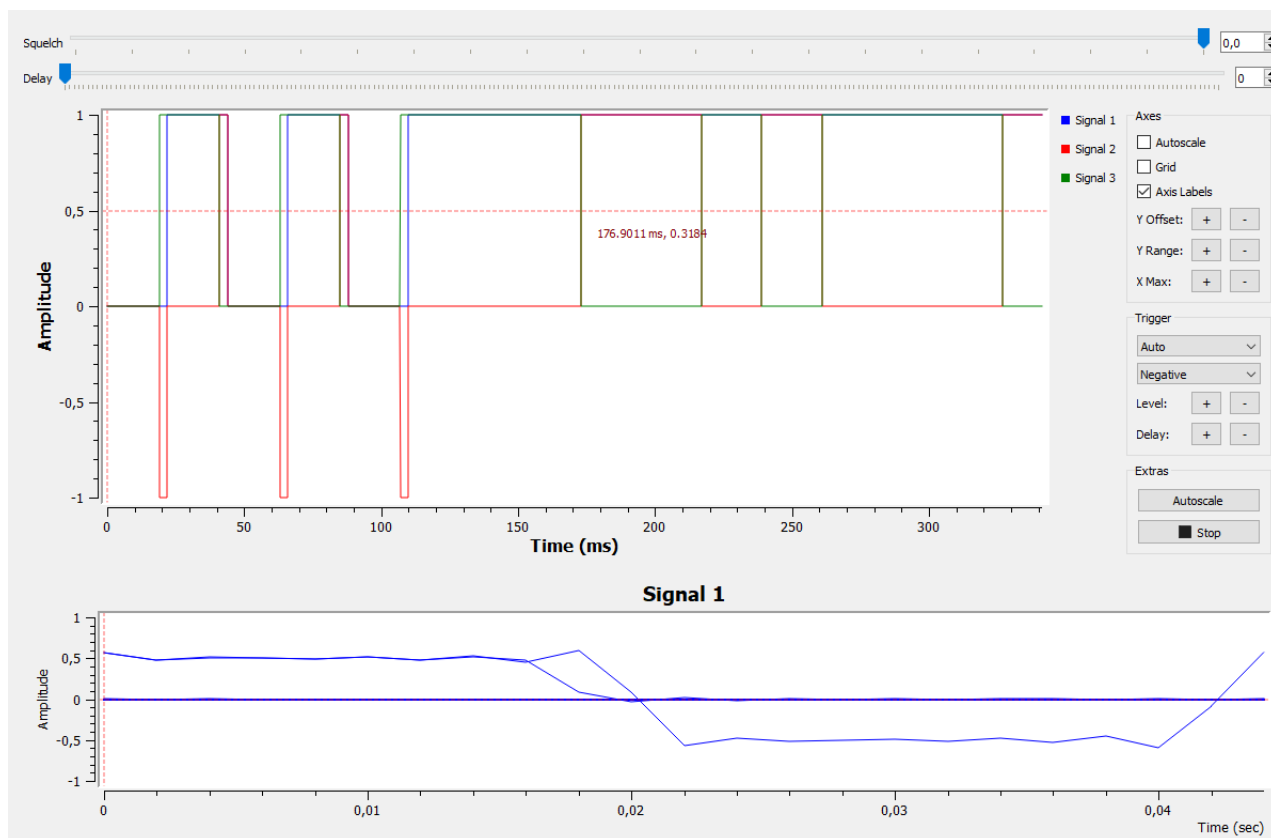


Рисунок 12.3. Тестирование без шума и задержки

На графике есть 3 сигнала. Синий сигнал - данные полученные приёмником. Зелёный сигнал - данные переданные передатчиком. Красный сигнал - разница между двумя предыдущими. Если всё передаётся верно, то он д. б. равен нулю. Видим, что передняя 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.

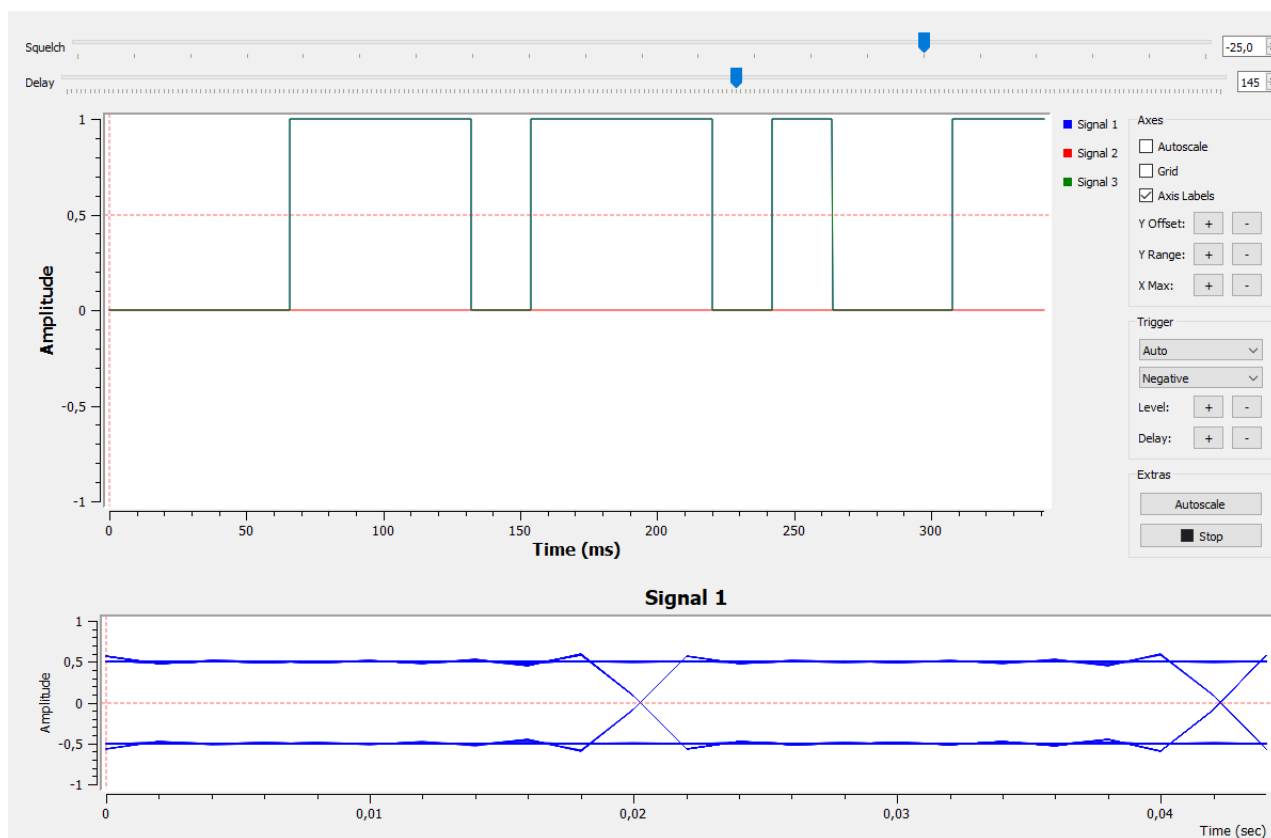


Рисунок 12.4. Тестирование с нужной задержкой

На рисунке видно, что мы подвергли 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.

Перечень использованных источников

1. GNU Radio official page. — URL: <https://www.gnuradio.org/>.
2. Метод Барлетта. — URL: https://en.wikipedia.org/wiki/Bartlett's_method.