

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

Телекоммуникационные системы

Отчёт по лабораторным работам

Работу выполнил:
Назаров Д. Ю.
Группа:
3530901/80201
Преподаватель:
Богач Н. В.

Санкт-Петербург
2021

Содержание

1. Лабораторная работа 1	9
1.1. Упражнение 1	9
1.2. Упражнение 2	10
1.3. Упражнение 3	13
1.4. Упражнение 4	15
1.5. Выводы	17
2. Лабораторная работа 2	18
2.1. Упражнение 1	18
2.2. Упражнение 2	19
2.3. Упражнение 3	24
2.4. Упражнение 4	26
2.5. Упражнение 5	28
2.6. Упражнение 6	30
2.7. Выводы	32
3. Лабораторная работа 3	33
3.1. Упражнение 1	33
3.2. Упражнение 2	35
3.3. Упражнение 3	37
3.4. Упражнение 4	38
3.5. Упражнение 5	39
3.6. Упражнение 6	41
3.7. Выводы	46
4. Лабораторная работа 4	47
4.1. Упражнение 1	47
4.2. Упражнение 2	52
4.3. Упражнение 3	54
4.4. Упражнение 4	56
4.5. Упражнение 5	60
4.6. Выводы	63
5. Лабораторная работа 5	64
5.1. Упражнение 1	64
5.2. Упражнение 2	67
5.3. Упражнение 3	70
5.4. Упражнение 4	72
5.5. Выводы	79

6. Лабораторная работа 6	80
6.1. Упражнение 1	80
6.2. Упражнение 2	85
6.3. Упражнение 3	88
6.4. Выводы	94
7. Лабораторная работа 7	95
7.1. Упражнение 1	95
7.2. Упражнение 2	96
7.3. Выводы	98
8. Лабораторная работа 8	99
8.1. Упражнение 1	99
8.2. Упражнение 2	102
8.3. Упражнение 3	105
8.4. Выводы	108
9. Лабораторная работа 9	109
9.1. Упражнение 1	109
9.2. Упражнение 2	114
9.3. Упражнение 3	117
9.4. Упражнение 4	121
9.5. Упражнение 5	123
9.6. Выводы	126
10. Лабораторная работа 10	127
10.1. Упражнение 1	127
10.2. Упражнение 2	130
10.3. Выводы	135
11. Лабораторная работа 11	136
11.1. Упражнение 1	136
11.2. Упражнение 2	137
11.3. Упражнение 3	138
11.4. Выводы	144
12. Лабораторная работа 12	145
12.1. Передача сигнала	145
12.2. Добавление нарушений канала	147
12.3. Восстановление времени	149
12.3.1. Проблема 1st	149
12.3.2. Разные часы	150
12.3.3. Блок синхронизации многофазных часов	151
12.3.4. Использование блока многофазной синхронизации	154

12.4. Многолучевость	155
12.5. Эквалайзер	156
12.5.1. Эквалайзер CMA	156
12.5.2. Эквалайзер LMS DD	157
12.6. Фазовая и точная частотная коррекция	159
12.7. Расшифровка	160
12.8. Выводы	162

Список иллюстраций

1.1.	Результат взаимодействия с виджетами Ipython из конца файла	9
1.2.	Полный график выбранного звука	10
1.3.	Спектр сегмента звука	11
1.4.	Отфильтрованный спектр звука	12
1.5.	Сложный сигнал	13
1.6.	Спектр сложного сигнала	14
1.7.	Изначальный график звука	15
1.8.	Ускоренный график звука	16
2.1.	График для последнего примера	18
2.2.	График созданного пилообразного сигнала	20
2.3.	График спектра созданного пилообразного сигнала	21
2.4.	Сравнение спектров треугольного и пилообразного сигналов	22
2.5.	Сравнение спектров прямоугольного и пилообразного сигналов	23
2.6.	График спектра прямоугольного сигнала с завернутыми гармониками	24
2.7.	График треугольного сигнала частотой 440 Гц	26
2.8.	Сравнительный график для сигнала до изменений и после	27
2.9.	Сравнительный график спектра до изменений и после	29
2.10.	Сравнительный график спектра до изменений и после	31
2.11.	График для измененного сигнала	31
3.1.	Спектр утечки	33
3.2.	Спектр созданных окон	34
3.3.	Спектр сегмента звука	36
3.4.	Спектр созданного звука	37
3.5.	Спектр записи глиссандо	38
3.6.	Спектр созданного глиссандо C3-F3	40
3.7.	Спектр записи гласных звуков	41
3.8.	Спектр звука а	42
3.9.	Спектр звука е	43
3.10.	Спектр звука і	43
3.11.	Спектр звука о	44
3.12.	Спектр звука и	45
4.1.	Спектр звука ветра	48
4.2.	Логарифмический спектр звука ветра	48
4.3.	Спектrogramма звука ветра	49
4.4.	Спектр звука костра	50
4.5.	Логарифмический спектр звука костра	50
4.6.	Спектrogramма звука костра	51
4.7.	Оценка мощности сегментов звука ветра	53
4.8.	График цены биткоина	54
4.9.	Логарифмический спектр цены биткоина	55

4.10. График при низкой амплитуде	57
4.11. График при большой амплитуде	58
4.12. Спектры мощности для двух сигналов	58
4.13. График розового шума	61
4.14. Логарифмический спектр мощности розового шума	61
4.15. Соотношение мощности и частоты у розового шума	62
5.1. График автокорреляции первого сегмента	64
5.2. График автокорреляции второго сегмента	65
5.3. График спектрограммы сегмента	68
5.4. График спектрограммы с наложенной оценкой высоты	69
5.5. График изменения цены Bitcoin	70
5.6. График автокорреляции цены Bitcoin	71
5.7. График гармонической структуры записи	72
5.8. График спектра выбранного сегмента	73
5.9. График автокорреляции сегмента	74
5.10. График сегмента без основной частоты	75
5.11. График автокорреляции сегмента без основной частоты	76
5.12. График сегмента без основной частоты и гармоник выше 1200 Гц	77
5.13. График автокорреляции сегмента без гармоник выше 1200 Гц .	78
6.1. Результаты analyze1	81
6.2. График analyze1	82
6.3. Результаты analyze2	82
6.4. График analyze2	83
6.5. Результаты scipy.fftpack.dct	83
6.6. График scipy.fftpack.dct	84
6.7. Сравнение трёх функций	84
6.8. ДКП сегмента	85
6.9. ДКП фильтрованного сегмента	86
6.10. Пилообразный сигнал. Угловая часть спектра	88
6.11. Пилообразный сигнал. Угловая часть спектра с порогом	89
6.12. Пилообразный сигнал. Нулевые углы	89
6.13. Пилообразный сигнал. Поворот угла	90
6.14. Пилообразный сигнал. Случайные углы	90
6.15. Гобой. Угловая часть спектра	90
6.16. Гобой. Нулевые углы	91
6.17. Гобой. Поворот угла	91
6.18. Гобой. Случайные углы	91
6.19. Саксофон. Нулевые углы	92
6.20. Саксофон. Поворот угла	92
6.21. Саксофон. Случайные углы	93
7.1. График "полного" БПФ для треугольного сигнала из chap07.ipynb	95
8.1. График гауссова окна при std = 2	100
8.2. График гауссова окна при std = 5	100

8.3.	График гауссова окна при std = 20	101
8.4.	График для гауссова окна и его БПФ при std = 2	103
8.5.	График для гауссова окна и его БПФ при std = 0.1	103
8.6.	График для гауссова окна и его БПФ при std = 20	104
8.7.	Сравнительный график для различных окон	106
8.8.	Сравнительный график ДПФ для различных окон	107
8.9.	Сравнительный график ДФП в логарифмическом масштабе	107
9.1.	Сигнал Facebook	109
9.2.	Спектр Facebook	110
9.3.	Выходной сигнал	110
9.4.	Спектр выходного сигнала	111
9.5.	Отношение входных и выходных данных	111
9.6.	Фильтры нарастающей суммы и интегрирования	112
9.7.	Сравнение отношения и фильтра	112
9.8.	Сравнение суммирования и фильтрации	113
9.9.	Треугольный сигнал	114
9.10.	Результат diff	115
9.11.	Результат differentiate	115
9.12.	Спектр в сигнал	116
9.13.	Прямоугольный сигнал	117
9.14.	Результат cumsum	118
9.15.	Результат integrate	118
9.16.	Спектр в сигнал	119
9.17.	Сравнение функций	120
9.18.	Результирующий сигнал	121
9.19.	Спектр результирующего сигнала	122
9.20.	Кубический сигнал	123
9.21.	Вторая разность кубического сигнала	124
9.22.	Вторая производная кубического сигнала	125
9.23.	Сравнение фильтра	125
10.1.	График изменённого выстрела	128
10.2.	Спектр изменённого выстрела	128
10.3.	График изменённой скрипки	129
10.4.	График сигнала из женского клуба	131
10.5.	Спектр сигнала из женского клуба	131
10.6.	Логарифмический спектр сигнала из женского клуба	132
10.7.	Исходный сигнал пианино	133
10.8.	Трансформированный сигнал пианино	133
10.9.	Сигнал пианино, трансформированный с помощью свёртки	134
11.1.	График из последнего примера	136
11.2.	График для звука барабана	138
11.3.	График для спектра звука барабана	139
11.4.	График для отфильтрованного спектра сигнала	140

11.5. График спектра для сигнала после выборки	141
11.6. График спектра для сигнала после выборки без спектральных копий	141
11.7. Сравнительный график для отфильтрованного спектра и после выборки	142
11.8. График для сигнала после выборки	143
12.1. Flowgraph избыточной пропускной способности	145
12.2. Избыточная пропускная способность	146
12.3. Flowgraph созвездия QPSK	146
12.4. Созвездие QPSK	147
12.5. Flowgraph с нарушениями канала	148
12.6. Созвездия с нарушениями	148
12.7. Flowgraph проблемы ISI	149
12.8. Сравнение символов, отфильтрованных RC и RRC	149
12.9. Flowgraph разные часы	150
12.10 Работа разных часов	150
12.11 Flowgraph с использованием блока	151
12.12 Нет смещения часов	152
12.13 Есть смещение часов	152
12.14 Flowgraph с несколькими фазами	153
12.15 Несколько фаз	153
12.16 Flowgraph с блоком многофазной синхронизации	154
12.17 Использование блока многофазной синхронизации	154
12.18 Добавление частотного сдвига	155
12.19 Flowgraph многолучевость	155
12.20 Использование блока многофазной синхронизации	156
12.21 Flowgraph с эквалайзером CMA	157
12.22 Использование эквалайзера CMA	157
12.23 Flowgraph с эквалайзером LMS DD	158
12.24 Использование эквалайзера LMS DD	158
12.25 Flowgraph для исправления сдвига фазы и частоты	159
12.26 Исправление сдвига фазы и частоты	159
12.27 Flowgraph декодирования	160
12.28 Сравнение данных	161

1. Лабораторная работа 1

1.1. Упражнение 1

If you have Jupyter, load chap01.ipynb, read through it, and run the examples. You can also view this notebook at
↪ <https://tinyurl.com/thinkdsp01>.

Запустим chap01.ipynb в Pycharm, для этого склонируем репозиторий ThinkDSP, скачаем пакетный менеджер Anaconda для языка Python, подключим необходимые библиотеки.

Ячейки из Jupiter Notebook успешно запускаются.

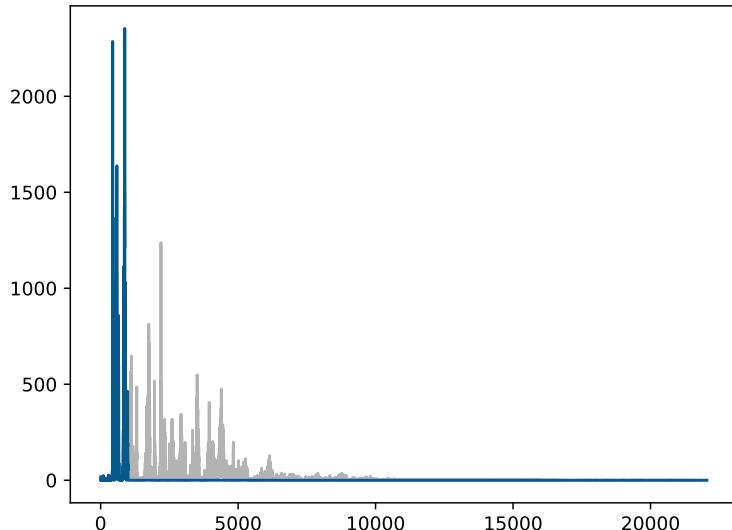


Рис. 1.1. Результат взаимодействия с виджетами Ipython из конца файла

1.2. Упражнение 2

Go to <https://freesound.org> and download a sound sample that includes music, speech, or other sounds that have a well-defined pitch. Select a roughly half-second segment where the pitch is constant. Compute and plot the spectrum of the segment you selected. What connection can you make between the timbre of the sound and the harmonic structure you see in the spectrum?

Use `high_pass`, `low_pass`, and `band_stop` to filter out some of the harmonics. Then convert the spectrum back to a wave and listen to it. How does the sound relate to the changes you made in the spectrum?

Скачаем с сайта <https://freesound.org> звук, имеющий чётко выраженную высоту. Я выбрал звук игры простой мелодии на пианино.

```
wave_1_2 = read_wave('..../32158_zin_piano-2-140bpm.wav')
wave_1_2.normalize()
wave_1_2.make_audio()
wave_1_2.plot()
```

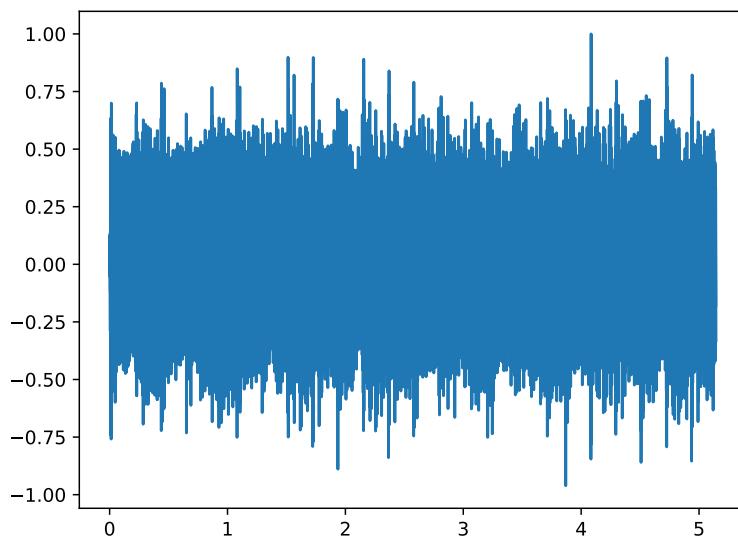


Рис. 1.2. Полный график выбранного звука

Выделим из этого звука сегмент длиной примерно в 0.2 секунды и создадим спектр этого сегмента.

```
segment_1_2 = wave_1_2.segment(start=0, duration=0.22)
spectrum_1_2 = segment_1_2.make_spectrum()
```

```
spectrum_1_2.plot(high=4500)
```

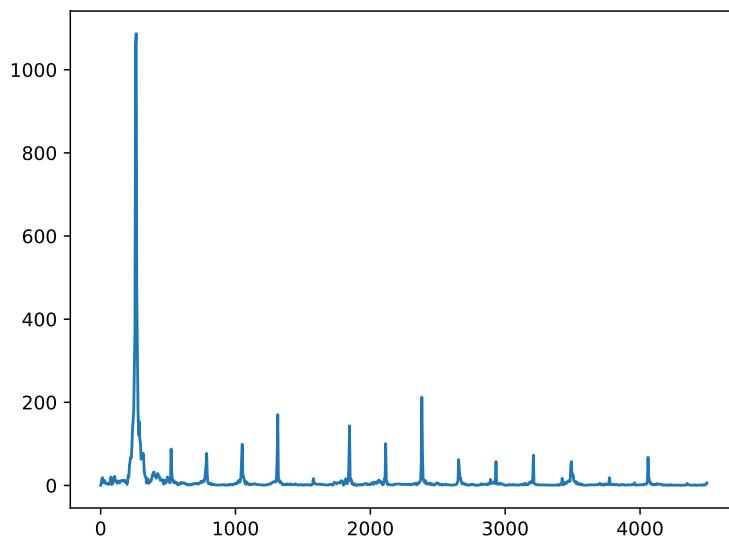


Рис. 1.3. Спектр сегмента звука

Тембр звука зависит от частот в спектре и их интенсивностей. Чем больше различных частот в спектре, тем богаче и насыщеннее будет тембр.

Выведем точки с наибольшими частотами и используем `low_pass` и `high_pass`, чтобы отфильтровать основную частоту. После этого сконвертируем сегмент обратно в волну и прослушаем его.

```
print(spectrum_1_2.peaks()[:10])

spectrum_1_2.plot(high=4500, color='0.7')
spectrum_1_2.low_pass(300)
spectrum_1_2.high_pass(220)
spectrum_1_2.plot(high=400, color="#045a8d")
decorate(xlabel='Frequency (Hz)')

spectrum_1_2.make_wave().make_audio()
```

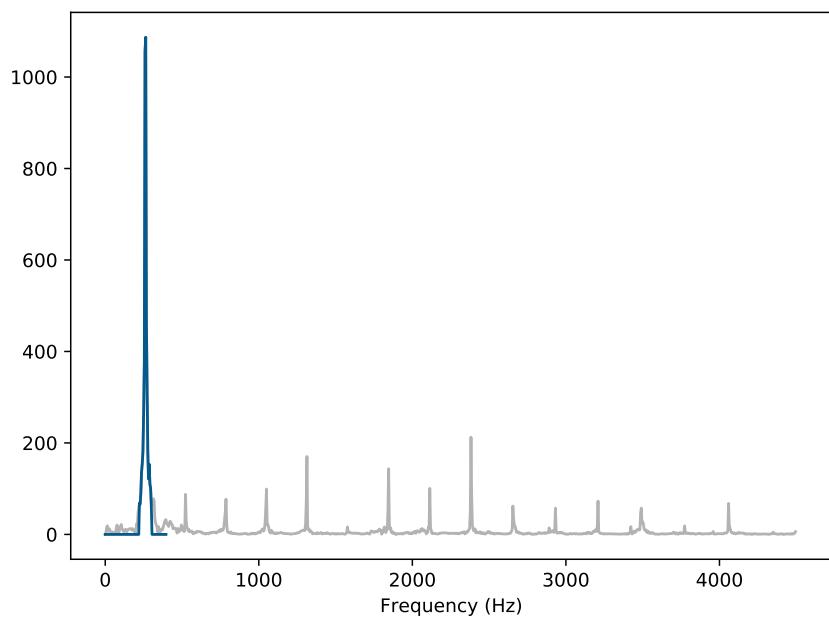


Рис. 1.4. Отфильтрованный спектр звука

Звук стал похож звук синуса или косинуса из упражнения 1.1.

1.3. Упражнение 3

Synthesize a compound signal by creating SinSignal and CosSignal objects and adding them up. Evaluate the signal to get a Wave, and listen to it. Compute its Spectrum and plot it. What happens if you add frequency components that are not multiples of the fundamental?

Создадим сигнал, складывая сигналы синусов и косинусов.
Используем `make_wave`, чтобы создать волну и прослушать её.

```
signal_1_3 = (CosSignal(freq=1000, amp=1.0) +
               SinSignal(freq=600, amp=2.0) +
               CosSignal(freq=400, amp=0.33) +
               CosSignal(freq=2000, amp=1.5))
signal_1_3.plot()
plt.savefig(filePath + "3.compound.signal" + fileExtension)
plt.close()

wave_1_3 = signal_1_3.make_wave(duration=3)
wave_1_3.apodize()
wave_1_3.make_audio()

spectrum_1_3 = wave_1_3.make_spectrum()
```

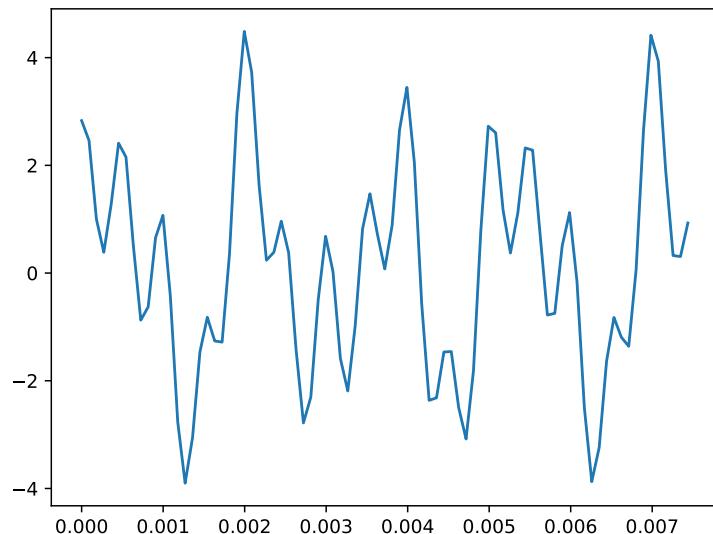


Рис. 1.5. Сложный сигнал

Звук данного сигнала схож со звуком гудка поезда.

Создадим спектр данного сигнала.

```
plt.savefig(filePath + "3.compound.spectrum" + fileExtension)
plt.close()
```

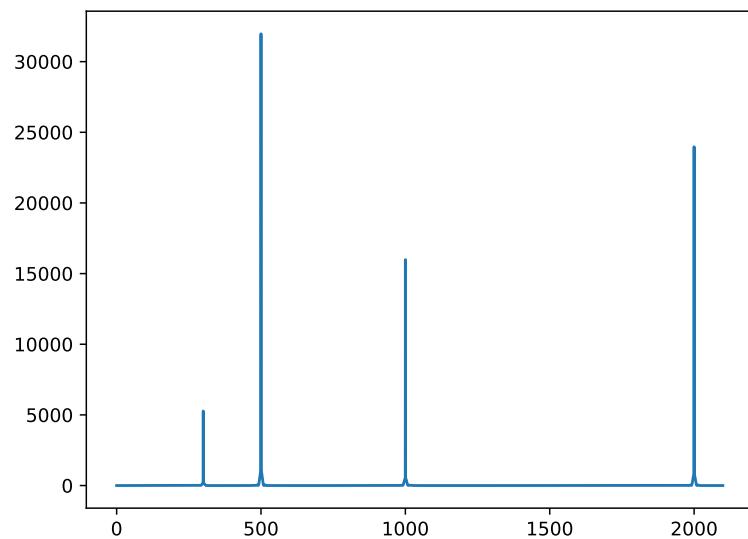


Рис. 1.6. Спектр сложного сигнала

Если добавить к сигналу частотные компоненты, которые нельзя получить умножением основной частоты на целое число, то звук станет менее гармоничным, и при сильном отклонении может получиться очень неприятный звук.

1.4. Упражнение 4

```
Write a function called stretch that takes a Wave and a stretch
factor and speeds up or slows down the wave by modifying ts and framerate.
Hint: it should only take two lines of code.
```

Напишем функцию stretch.

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

Используем звук из упражнения 1.2, чтобы проверить работу функции. Ускорим звук в два раза и сравним графики.

```
wave_1_4 = read_wave('..../32158_zin_piano-2-140bpm.wav')
wave_1_4.normalize()
wave_1_4.make_audio()
wave_1_4.plot()
```

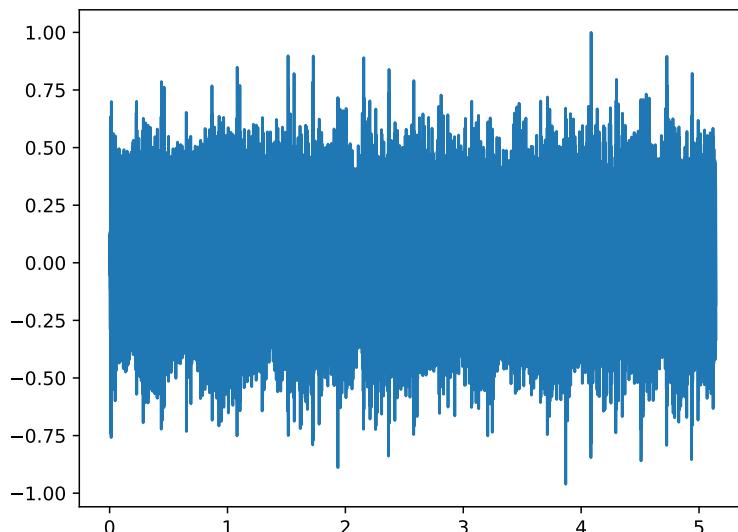


Рис. 1.7. Изначальный график звука

```
stretch(wave_1_4, 2)
wave_1_4.make_audio()

wave_1_4.plot()
```

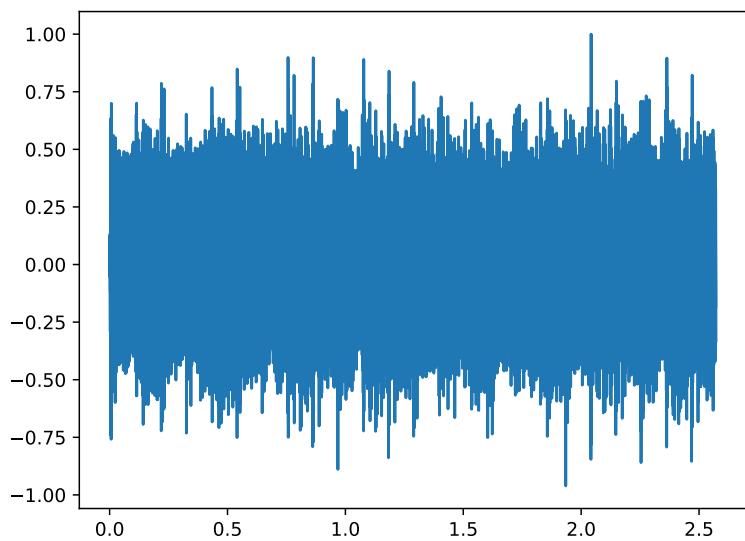


Рис. 1.8. Ускоренный график звука

Звук пианино стал более быстрым и как будто на несколько октав выше.

1.5. Выводы

В результате выполнения данной работы мы познакомились с понятиями сигнала извука, а также понятиями описывающими их. Кроме того, мы научились работать с сигналами, спектрами и звуковыми волнами, применяя различные функции, написанные на языке Python.

2. Лабораторная работа 2

2.1. Упражнение 1

If you use Jupyter, load chap02.ipynb and try out the examples.
You can also view the notebook at <https://tinyurl.com/thinkdsp02>.

В этом пункте лабораторной работы необходимо открыть `chap02.ipynb` и пройтись по всем программам из этого файла.

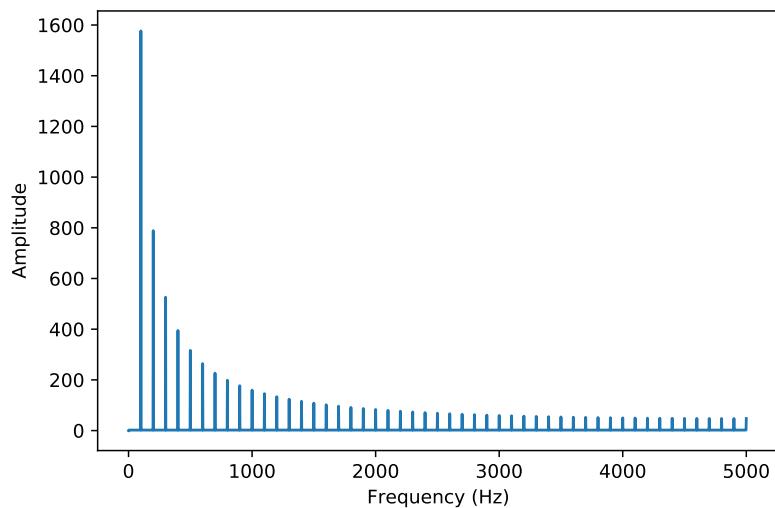


Рис. 2.1. График для последнего примера

Все программы были успешно запущены, так, например, в последнем примере, с помощью параметра `framerate` можно было делать звук более приглушенным или наоборот, более разборчивым.

2.2. Упражнение 2

A sawtooth signal has a waveform that ramps up linearly from -1 to 1, then drops to -1 and repeats. See https://en.wikipedia.org/wiki/Sawtooth_wave

Write a class called `SawtoothSignal` that extends `Signal` and provides `evaluate` to evaluate a sawtooth signal.

Compute the spectrum of a sawtooth wave. How does the harmonic structure compare to triangle and square waves?

Сначала напишем класс `SawtoothSignal`, расширяющий класс `Signal` и предоставляющий метод `evaluate` для оценки пилообразного сигнала и метод `period` для быстрого вычисления периода сигнала.

```
class SawtoothSignal(Signal):

    def __init__(self, freq=440, amp=1.0, offset=0, func=np.sin):
        self.freq = freq
        self.amp = amp
        self.offset = offset
        self.func = func

    def period(self):
        return 1.0 / self.freq

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

После этого создадим объект этого класса, преобразуем его в волну и построим график волны полученного сигнала.

```
sawtooth_signal_2 = SawtoothSignal()
sawtooth_wave_2 = sawtooth_signal_2.make_wave(sawtooth_signal_2.period() * 5,
→ framerate=40000)
sawtooth_wave_2.plot()
```

Теперь посмотрим на построенный график.

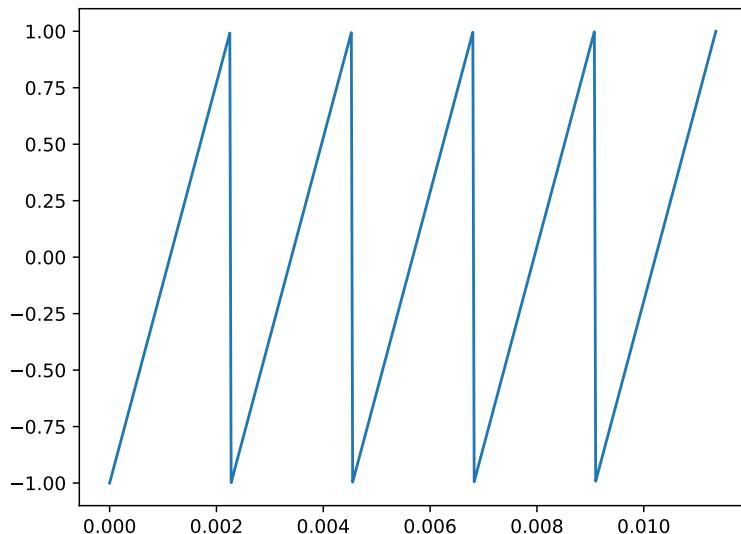


Рис. 2.2. График созданного пилообразного сигнала

Как мы можем видеть, созданный сигнал соответствует всем требованиям. Теперь вычислим спектр для данного сигнала, для этого снова вызовем метод `make_wave` и воспользуемся методом `make_spectrum`. После создания спектра создадим график для него.

```
sawtooth_wave_2 = sawtooth_signal_2.make_wave(0.5, framerate=40000)
sawtooth_spectrum_2 = sawtooth_wave_2.make_spectrum()
sawtooth_spectrum_2.plot()
decorate(xlabel='Frequency (Hz)')
```

И взглянем на получившийся график.

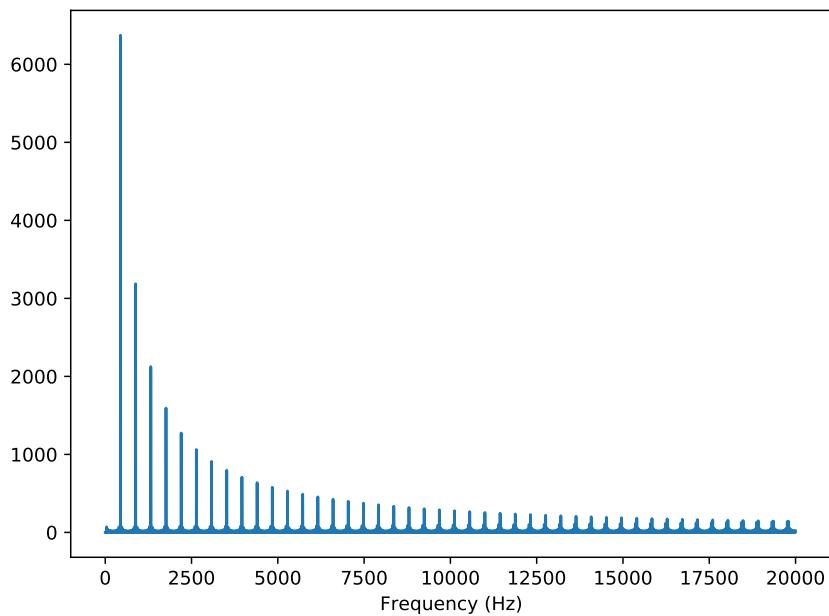


Рис. 2.3. График спектра созданного пилообразного сигнала

Теперь сравним гармоническую структуру пилообразного сигнала и аналогичную структуру треугольного сигнала. Для этого сперва создадим треугольный сигнал с параметрами, аналогичными созданному пилообразному сигналу.

После этого построим график, на котором будут отображаться оба спектра, наложенных друг на друга. Спектр пилообразного сигнала будет нарисован серым цветом, а спектр треугольного сигнала - голубым.

```
sawtooth_spectrum_2.plot(color='gray')
triangle_signal_2 = TriangleSignal(amp=0.79).make_wave(duration=0.5,
→ framerate=40000)
triangle_signal_2.make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

Посмотрим на получившийся график.

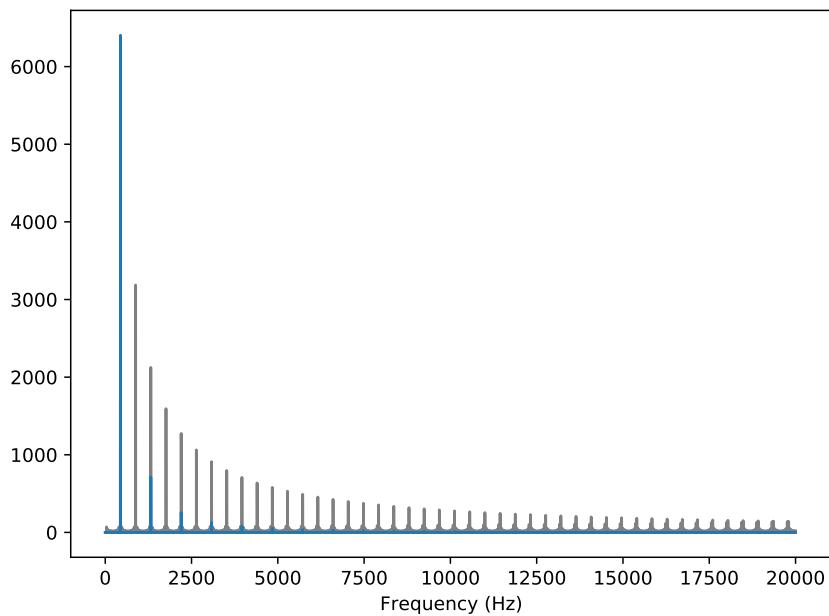


Рис. 2.4. Сравнение спектров треугольного и пилообразного сигналов

Как можно видеть из графика, пилообразный сигнал снижается медленнее, чем треугольный, а также в отличие от треугольного сигнала, у пилообразного присутствуют и четные и нечетные гармоники.

Теперь аналогичным образом сравним гармоническую структуру пилообразного сигнала и аналогичную структуру прямоугольного сигнала.

Создадим прямоугольный сигнал с теми же параметрами, построим для него спектр и отобразим этот спектр и спектр пилообразного сигнала на одном графике. Цветовое отображение осталось прежним - серый цвет для пилообразного сигнала и голубой для прямоугольного.

```
sawtooth_spectrum_2.plot(color='gray')
square_signal_2 = SquareSignal(amp=0.5).make_wave(duration=0.5,
↪ framerate=40000)
square_signal_2.make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

Взглянем на получившийся график.

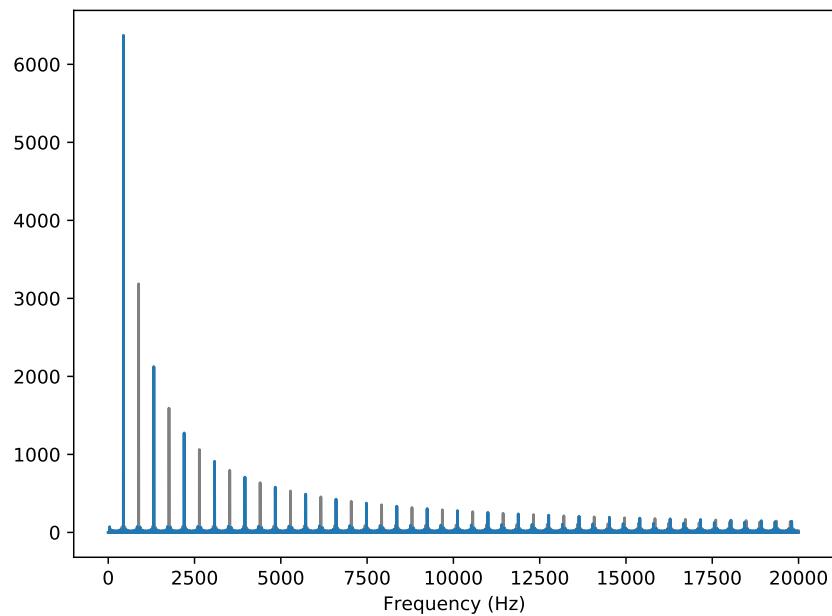


Рис. 2.5. Сравнение спектров прямоугольного и пилообразного сигналов

Как и в предыдущем сравнении, у пилообразного сигнала, в отличие от прямоугольного, есть и четные и нечетные гармоники, однако теперь амплитуды частот у обоих сигналов снижаются с одинаковой скоростью.

2.3. Упражнение 3

Make a square signal at 1100 Hz and make a wave that samples it at 10000 frames per second. If you plot the spectrum, you can see that most of the harmonics are aliased. When you listen to the wave, can you hear the aliased harmonics?

Первым делом построим необходимый сигнал, его спектр и построим график для полученного спектра.

```
square_signal_3 = SquareSignal(freq=1100).make_wave(framerate=10000)
square_signal_3.make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

Посмотрим полученный график.

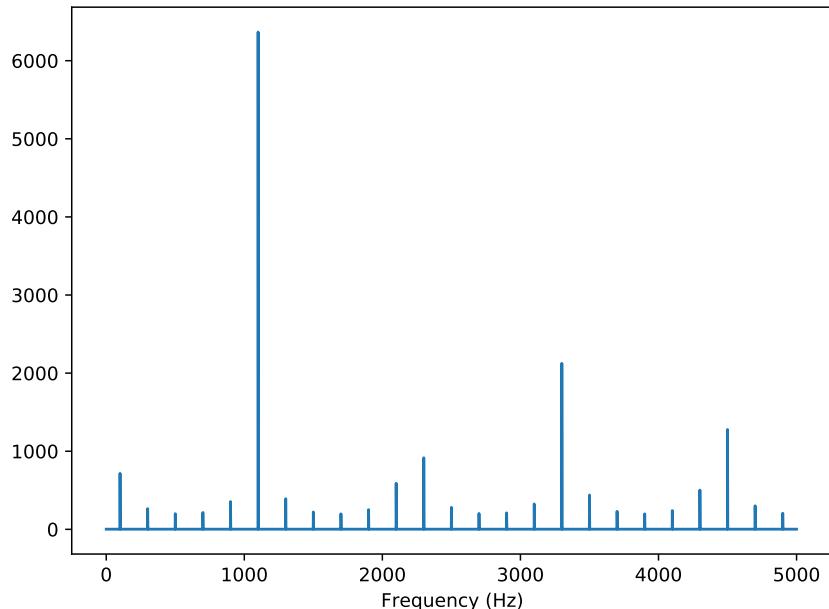


Рис. 2.6. График спектра прямоугольного сигнала с завернутыми гармониками

Видно, что основная гармоника находится на 1100 Гц, первая гармоника на 3300 Гц, а следующие гармоники находятся не на своих местах. Проблема заключается в том, что при взятии выборок в дискретные моменты времени теряется информация о том, что было между ними. Выборки с достаточно высокой частотой при относительно небольшой скорости выборки будут неотличимы от выборок с низкой частотой при той же скорости. Этот эффект называется биениями (алиасингом).

Попробуем услышать завернутые гармоники. Для этого создадим запись нашего сигнала.

```
square_signal_3.make_audio()
```

Основной тон будет гармоникой биения на частоте 200Гц, для проверки этого утверждения создадим синусоидальную волну данной частоты и послушаем ее.

```
SinSignal(200).make_wave(duration=0.5, framerate=10000).make_audio()
```

2.4. Упражнение 4

If you have a spectrum object, spectrum, and print the first few values of spectrum.fs, you'll see that they start at zero. So spectrum.hs[0] is the magnitude of the component with frequency 0. But what does that mean?

Try this experiment:

1. Make a triangle signal with frequency 440 and make a Wave with duration 0.01 seconds. Plot the waveform.
2. Make a Spectrum object and print spectrum.hs[0]. What is the amplitude and phase of this component?
3. Set spectrum.hs[0] = 100. Make a Wave from the modified Spectrum and plot it. What effect does this operation have on the waveform?

Сначала создадим треугольный сигнал частотой 440Гц и график для него.

```
triangle_signal_4 = TriangleSignal(freq=440).make_wave(duration=0.01)
triangle_signal_4.plot()
decorate(xlabel='Time (s)')
```

Затем взглянем на получившийся график.

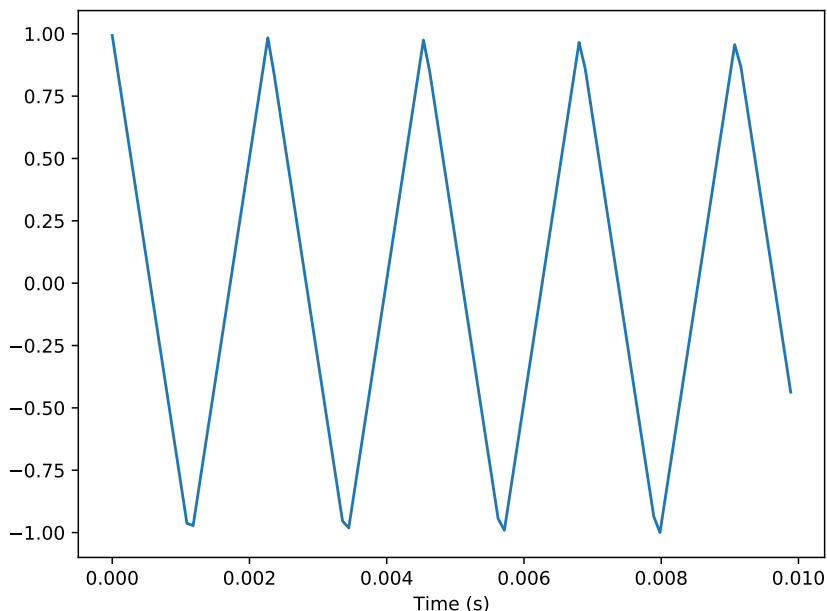


Рис. 2.7. График треугольного сигнала частотой 440 Гц

После этого создадим Spectrum и выведем нулевое значение из массива hs. Данный элемент массива равен $1.0436096431476471e - 14 + 0j$.

Затем изменим значение этого элемента, создадим из измененного спектра новый сигнал и нарисуем сравнительный график, на котором изначальный сигнал будет отображен серым цветом, а новый - голубым.

```
spectrum_4.hs[0] = 100
triangle_signal_4.plot(color='gray')
spectrum_4.make_wave().plot()
decorate(xlabel='Time (s)')
```

Взглянем на получившийся график.

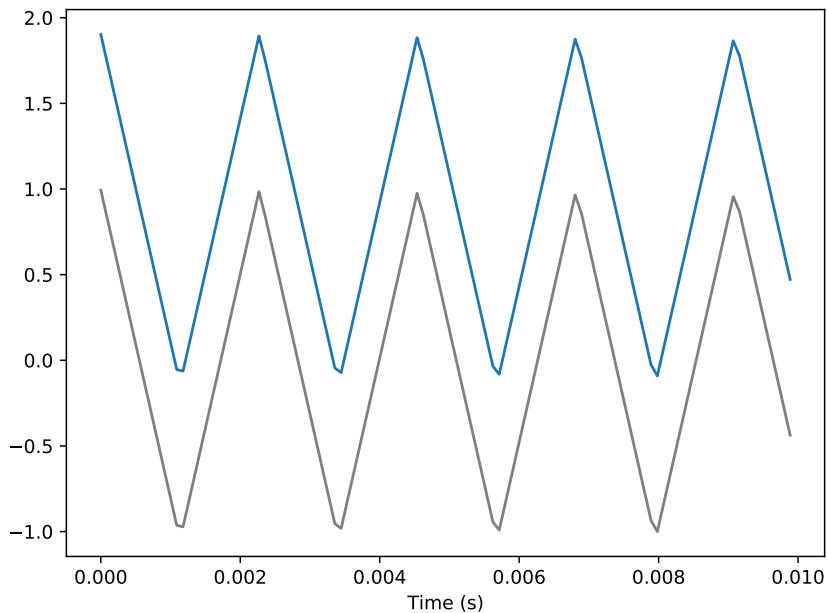


Рис. 2.8. Сравнительный график для сигнала до изменений и после

Как можно видеть, сигнал был смещен по оси у наверх, по сравнению с изначальным положением.

2.5. Упражнение 5

Write a function that takes a Spectrum as a parameter and modifies it by dividing each element of hs by the corresponding frequency from fs. Hint: since division by zero is undefined, you might want to set spectrum.hs[0] = 0.

Test your function using a square, triangle, or sawtooth wave.

1. Compute the Spectrum and plot it.
2. Modify the Spectrum using your function and plot it again.
3. Make a Wave from the modified Spectrum and listen to it. What effect does this operation have on the signal?

Сначала напишем нужную функцию.

```
def spectrum_divide(spectrum):  
    i = 0  
    while i < len(spectrum):  
        if i == 0:  
            spectrum.hs[i] = 0  
        else:  
            spectrum.hs[i] = spectrum.hs[i] / spectrum.fs[i]  
        i += 1
```

Затем создадим пилювидный сигнал длительностью в 1 секунду.

```
wave_5 = SawtoothSignal().make_wave(duration=1)
```

После этого вычислим его спектр, сохраним это значение и получим измененный спектр вызовом функции, затем построим график для сравнения значений спектров до изменений и после: значение до будет отрисовываться серым цветом, а значение после будет отрисовываться голубым цветом.

```
spectrum_5 = wave_5.make_spectrum()  
spectrum_5.plot(color='gray')  
spectrum_divide(spectrum_5)  
spectrum_5.high_pass(100)  
spectrum_5.scale(440)  
spectrum_5.plot()  
decorate(xlabel='Frequency (Hz)')
```

Взглянем на получившийся график.

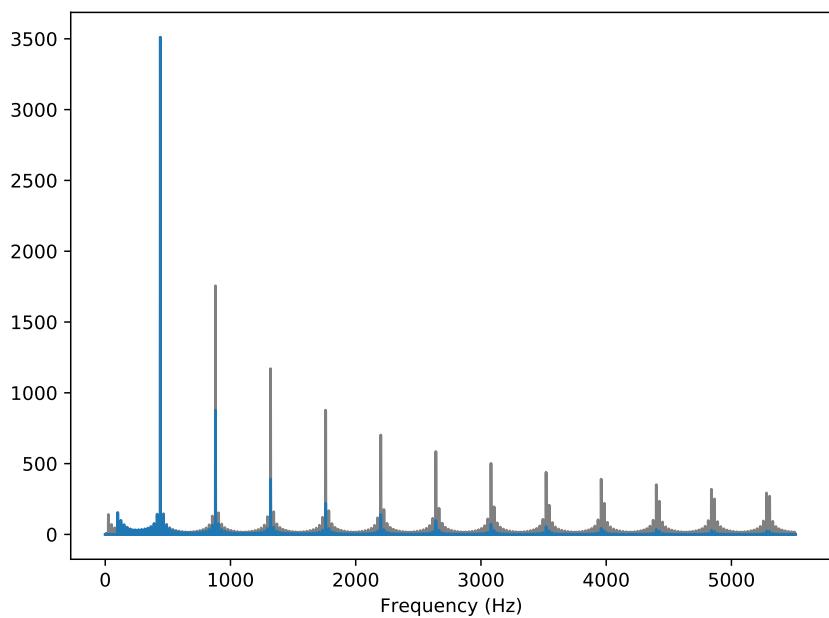


Рис. 2.9. Сравнительный график спектра до изменений и после

Для прослушивания воспользуемся методом `make_audio`.

```
wave_5.make_audio()  
changed_signal_5 = spectrum_5.make_wave()  
changed_signal_5.make_audio()
```

Сигнал после изменений стал более приглушенным по сравнению с изначальным вариантом.

2.6. Упражнение 6

Triangle and square waves have odd harmonics only; the sawtooth wave has both even and odd harmonics. The harmonics of the square and sawtooth waves drop off in proportion to $1/f$; the harmonics of the triangle wave drop off like $1/f^2$. Can you find a waveform that has even and odd harmonics that drop off like $1/f^2$?

Hint: There are two ways you could approach this: you could construct the signal you want by adding up sinusoids, or you could start with a signal that is similar to what you want and modify it.

Воспользуемся пиловидным сигналом, у него есть и четные и нечетные гармоники, но при этом они уменьшаются пропорционально $1/f$. Для того, чтобы это исправить, воспользуемся методом, написанным в прошлом упражнении. В результате изменения спектра делением значений hs на соответствующие значения fs получится необходимая по заданию пропорция $1/f^2$.

Сначала создадим сам сигнал.

```
sawtooth_signal_6 = SawtoothSignal().make_wave(duration=0.5, framerate=20000)
```

Затем воспользуемся методом, написанным в предыдущем упражнении и аналогично тому упражнению построим сравнительный график для спектра до изменений и после.

```
spectrum_6 = sawtooth_signal_6.make_spectrum()
spectrum_6.plot(color='gray')
spectrum_divide(spectrum_6)
spectrum_6.scale(440)
spectrum_6.high_pass(100)
spectrum_6.plot()
decorate(xlabel='Frequency (Hz)')
```

Взглянем на получившийся график.

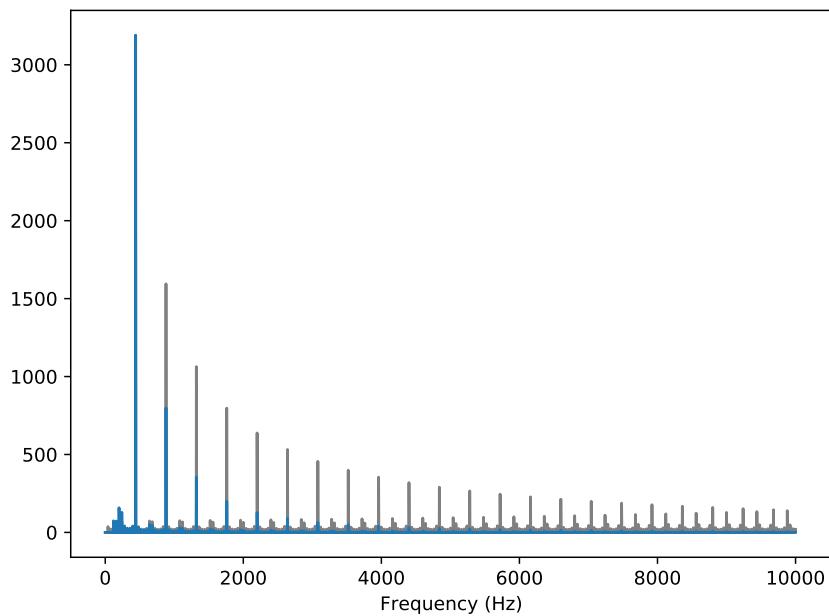


Рис. 2.10. Сравнительный график спектра до изменений и после

Теперь построим волну по получившемуся спектру и построим график для нового сигнала.

```
spectrum_6.make_wave().segment(duration=0.01).plot()
decorate(xlabel='Time (s)')
```

И взглянем на полученный график.

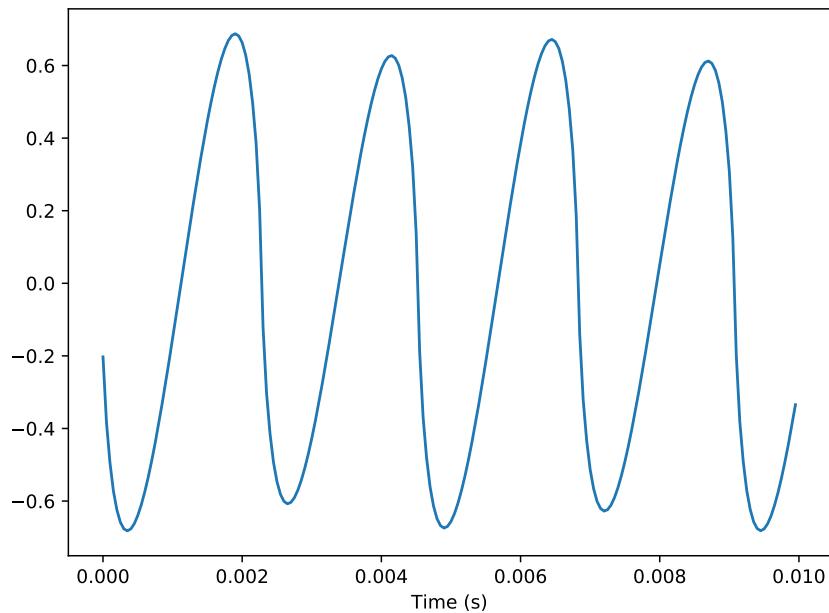


Рис. 2.11. График для измененного сигнала

Получившийся сигнал немного напоминает синусоиду.

2.7. Выводы

В данной работе были получены навыки работы с пиловидными, прямоугольными и треугольными сигналами, были рассмотрены некоторые из их характеристик и на практике изучено как их можно изменять, был изучен эффект биения - эффект, приводящий к наложению непрерывных сигналов при их дискретизации.

3. Лабораторная работа 3

3.1. Упражнение 1

```
the other windows provided by NumPy, and see what effect they have on leakage. See http://docs.scipy.org/doc/numpy/reference/routines.window.html.
```

```
"""
```

```
signal_3_1_1 = SinSignal(freq=440)
duration_3_1_1 = signal_3_1_1.period * 30.25
```

В этом пункте лабораторной необходимо открыть `chap03.ipynb` и пройтись по всем программам из этого файла, а также заменить окно Хемминга одним из других окон, предоставляемых NumPy, и посмотреть, как они влияют на утечки.

Для начала рассмотрим пример утечки.

```
signal_3_1_1 = SinSignal(freq=440)
duration_3_1_1 = signal_3_1_1.period * 30.25
wave_3_1_1 = signal_3_1_1.make_wave(duration_3_1_1)
spectrum_3_1_1 = wave_3_1_1.make_spectrum()

spectrum_3_1_1.plot(high=880)
decorate(xlabel='Frequency (Hz)')
```

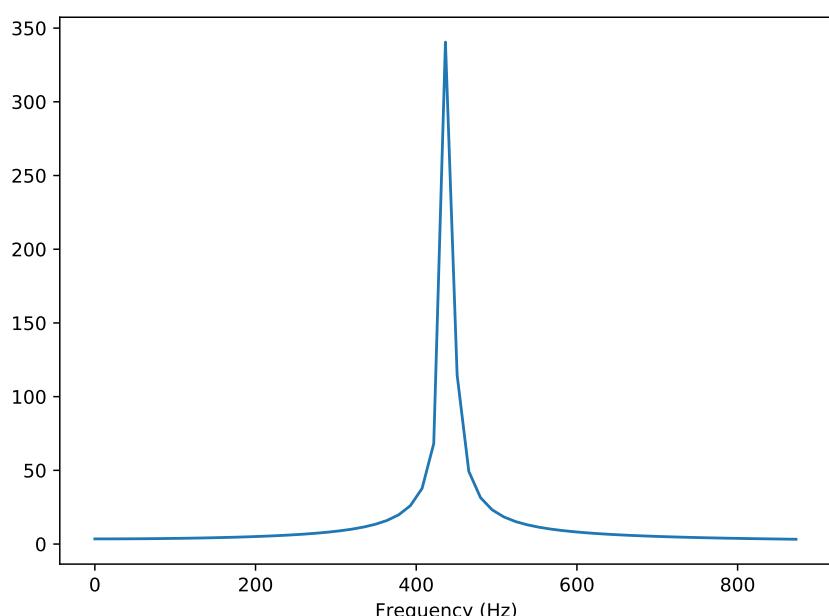


Рис. 3.1. Спектр утечки

Далее попробуем заменить окно Хэмминга на другое из 4 выбранных окон, предоставляемых NumPy:

```
window_type_names = [np.bartlett, np.blackman, np.hamming, np.hanning]

for window_type in window_type_names:
    wave_3_1_2 = signal_3_1_1.make_wave(duration_3_1_1)
    wave_3_1_2.ys *= window_type(len(wave_3_1_2.ys))

    spectrum_3_1_2 = wave_3_1_2.make_spectrum()
    spectrum_3_1_2.plot(high=880, label=window_type.__name__)

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

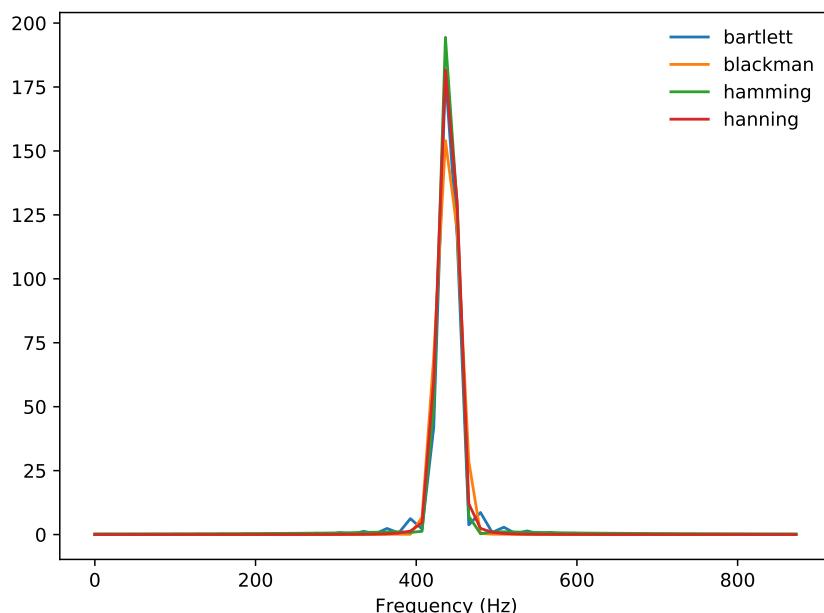


Рис. 3.2. Спектр созданных окон

Все четыре окна хорошо справляются с утечкой.

3.2. Упражнение 2

Exercise 3.2 Write a class called `SawtoothChirp` that extends `Chirp` and overrides `evaluate` to generate a sawtooth waveform with frequency that increases (or decreases) linearly.

В этом пункте лабораторной работы необходимо написать класс `SawtoothChirp`, который переопределяет `evaluate` для генерации пилообразного сигнала.

```
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
```

После написания класса удостоверимся, что переопределённый метод действительно генерирует пилообразный сигнал с линейно увеличивающейся или уменьшающейся частотой путём прослушивания получившегося звука и рассмотрения спектрограммы.

```
signal_3_2 = SawtoothChirp(start=220, end=880)
wave_3_2 = signal_3_2.make_wave(duration=1, framerate=4000)
wave_3_2.apodize()
wave_3_2.make_audio()
```

На полученной спектрограмме эффект биений очевиден, а прослушав полученный до этого звук внимательно, можно даже их и услышать - отражающиеся гармоники создают фоновое шипение.

```
spectrum_3_2 = wave_3_2.make_spectrogram(256)
spectrum_3_2.plot()
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

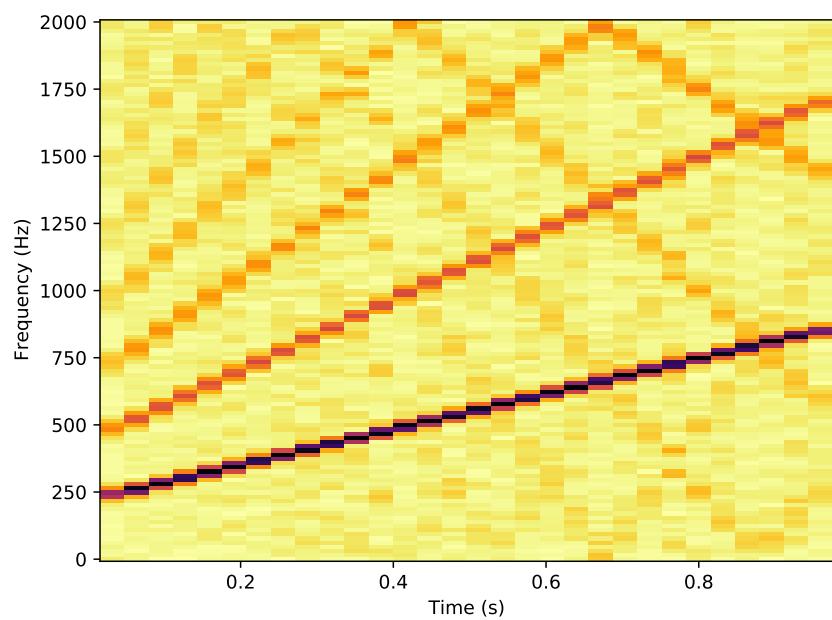


Рис. 3.3. Спектр сегмента звука

3.3. Упражнение 3

Exercise 3.3 Make a sawtooth chirp that sweeps from 2500 to 3000 Hz, then use it to make a wave with duration 1 s and framerate 20 kHz. Draw a sketch of what you think the spectrum will look like. Then plot the spectrum and see if you got it right.

В этом пункте лабораторной работы необходимо создать пилообразный chirp, меняющийся от 2500 до 3000 Гц, после чего использовать его для создания волны длительностью 1 с и частотой кадров 20 кГц.

```
signal_3_3 = SawtoothChirp(start=2500, end=3000)
wave_3_3 = signal_3_3.make_wave(duration=1, framerate=20000)
```

Поскольку основная частота колеблется от 2500 до 3000 Гц, я ожидаю увидеть что-то вроде высокой башни в этом диапазоне. Первая гармоника колеблется от 5000 до 6000 Гц, поэтому я ожидаю более короткую башню в данном диапазоне. Вторая гармоника колеблется от 7500 до 9000 Гц, поэтому я ожидаю что-то еще более короткое в этом диапазоне. Другие гармоники повсюду накладываются друг на друга, поэтому я ожидаю увидеть некоторую энергию на всех других частотах. Эта распределённая энергия создает интересные звуки.

Посмотрим на получившийся спектр.

```
wave_3_3.make_spectrum().plot()
decorate(xlabel='Frequency (Hz)')
```

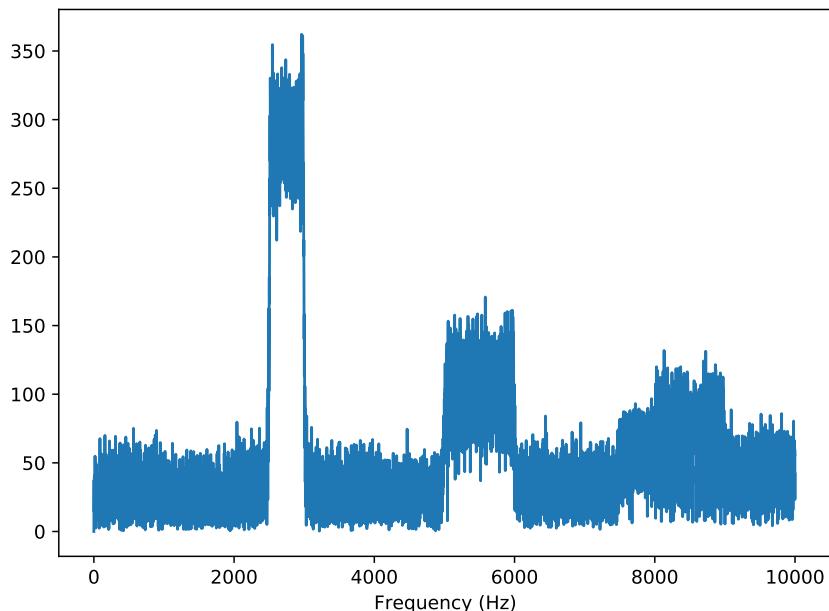


Рис. 3.4. Спектр созданного звука

Полученный спектр совпал с ожидаемым.

3.4. Упражнение 4

Exercise 3.4 In musical terminology, a “glissando” is a note that slides from one pitch to another, so it is similar to a chirp.

Find or make a recording of a glissando and plot a spectrogram of the first few seconds.

В этом пункте лабораторной работы необходимо найти запись глиссандо и сделать спектрограмму первых нескольких секунд.

```
wave_3_4 =  
    read_wave('..../153623_carlos-vaquero_violin-tenuto-non-vibrato-glissando-1.wav')  
wave_3_4.make_audio()
```

Теперь рассмотрим спектрограмму звука.

```
wave_3_4.make_spectrogram(1024).plot(high=10000)  
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

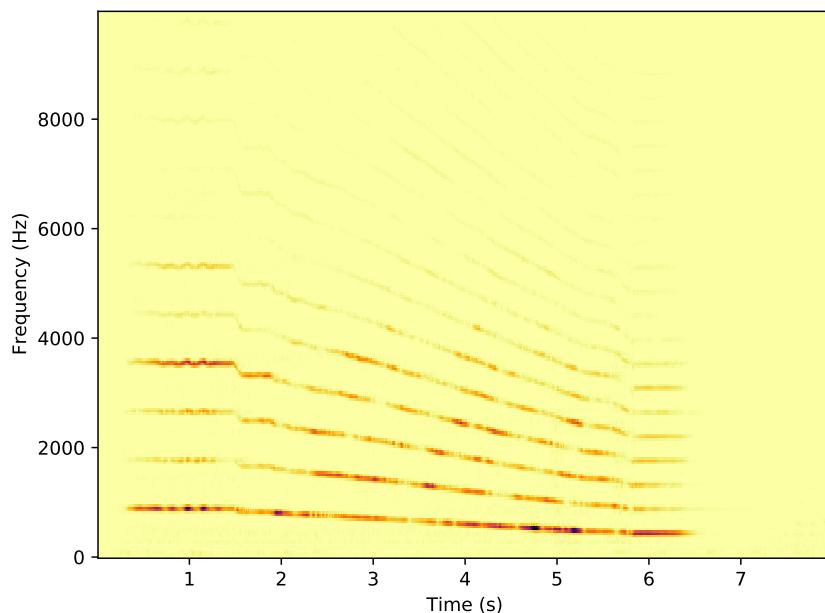


Рис. 3.5. Спектр записи глиссандо

3.5. Упражнение 5

Exercise 3.5 A trombone player can play a glissando by extending the trombone slide while blowing continuously. As the slide extends, the total length of the tube gets longer, and the resulting pitch is inversely proportional to length.

Assuming that the player moves the slide at a constant speed, how does frequency vary with time?

Write a class called `TromboneGliss` that extends `Chirp` and provides `evaluate`. Make a wave that simulates a trombone glissando from C3 up to F3 and back down to C3. C3 is 262 Hz; F3 is 349 Hz.

Plot a spectrogram of the resulting wave. Is a trombone glissando more like a linear or exponential chirp?

В этом пункте лабораторной работы необходимо написать класс `TromboneGliss`, который расширяет `Chirp` и переопределяет `evaluate`. Также необходимо создать сигнал, имитирующий глиссандо на тромбоне от C3 до F3 и обратно к C3 (нота C3 составляет 262 Гц; F3 составляет 349 Гц), после чего построить спектрограмму полученного сигнала.

Для начала напишем класс `TromboneGliss`:

```
class TromboneGliss(Chirp):
    def evaluate(self, ts):
        l1, l2 = 1.0 / self.start, 1.0 / self.end
        lengths = np.linspace(l1, l2, len(ts))
        freqs = 1 / lengths

        dts = np.diff(ts, prepend=0)
        dphis = 2 * np.pi * freqs * dts
        phases = np.cumsum(dphis)
        ys = self.amp * np.cos(phases)
        return ys
```

Теперь создадим сигнал, похожий на глиссандо на тромбоне:

```
note_c3 = 262
note_f3 = 349
signal_3_5_1 = TromboneGliss(note_c3, note_f3)
wave_3_5_1 = signal_3_5_1.make_wave(duration=1)
wave_3_5_1.apodize()
wave_3_5_1.make_audio()
```

```

signal_3_5_2 = TromboneGliss(note_f3, note_c3)
wave_3_5_2 = signal_3_5_2.make_wave(duration=1)
wave_3_5_2.apodize()
wave_3_5_2.make_audio()

wave_3_5 = wave_3_5_1 | wave_3_5_2
wave_3_5.make_audio()

```

Сделаем спектр полученного сигнала:

```

spectrum_3_5 = wave_3_5.make_spectrogram(1024)
spectrum_3_5.plot(high=1000)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')

```

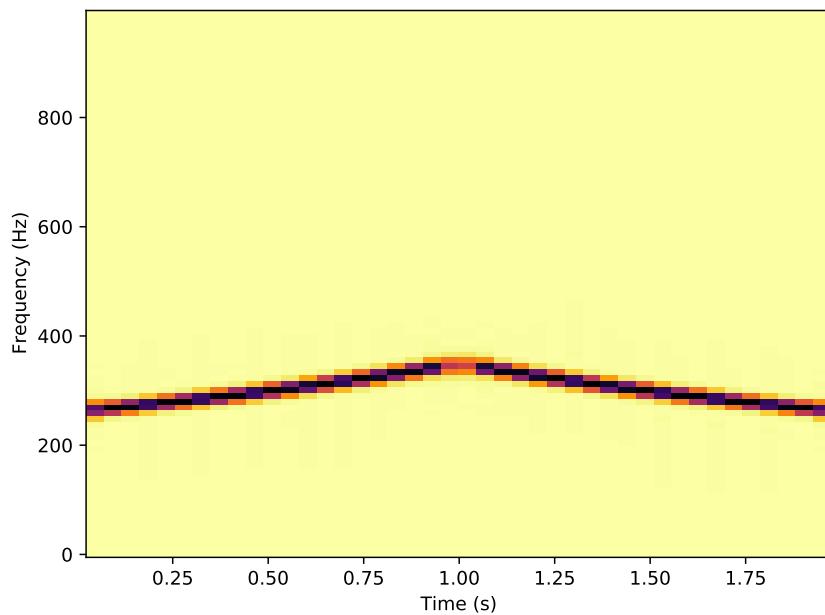


Рис. 3.6. Спектр созданного глиссандо C3-F3

Как мы можем видеть на рис. 3.6, глиссандо больше похоже на линейный chirp.

3.6. Упражнение 6

Exercise 3.6 Make or find a recording of a series of vowel sounds and look at the spectrogram. Can you identify different vowels?

Скачаем с <https://freesound.org> запись произнесенных гласных звуков и создадим спектрограмму.

```
wave_3_6 = read_wave('..../523141__bryanr17__mjoven.wav')
wave_3_6.make_audio()

high = 1000

wave_3_6.make_spectrogram(512).plot(high=high)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
plt.savefig(filePath + "6.vowel.spectrum" + fileExtension)
plt.close()
```

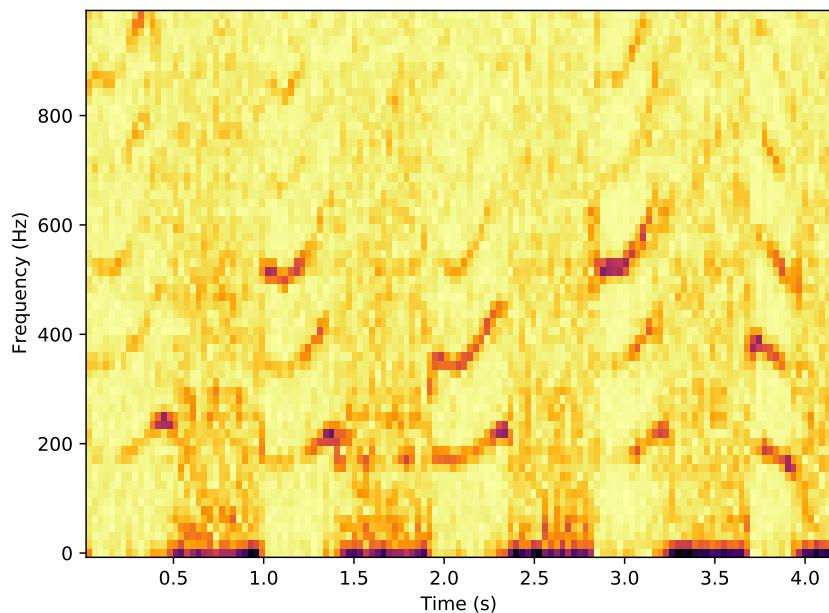


Рис. 3.7. Спектр записи гласных звуков

Как мы можем видеть на рис.3.7, разные гласные звуки имеют разные частоты, так что различить гласные по спектру возможно, но очень трудно.

Пики на спектрограмме называются формантами. Гласные звуки различаются соотношением амплитуд первых двух формант относительно основного тона. Посмотрим на спектры каждого звука.

```
a_segment = wave_3_6.segment(start=0, duration=0.5)
a_segment.make_spectrum().plot(high=high)
decorate(xlabel='Frequency (Hz)')
```

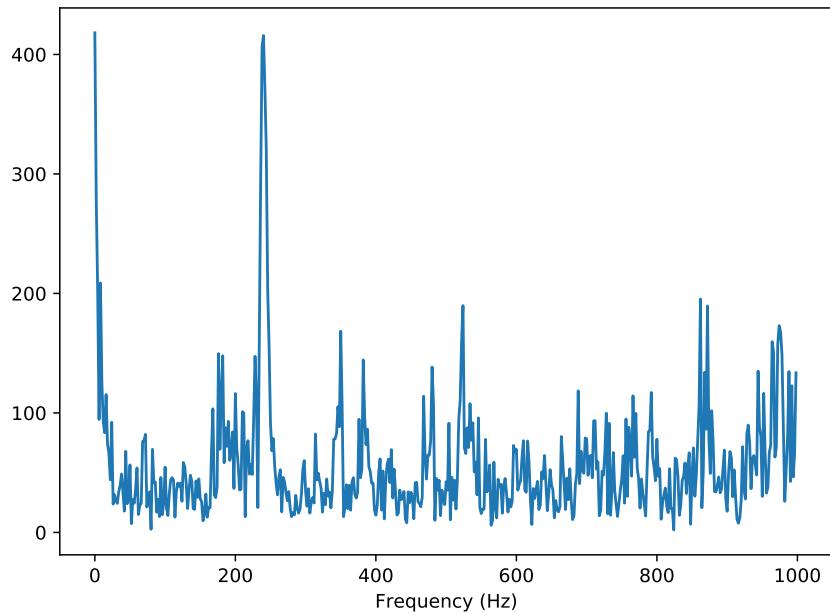


Рис. 3.8. Спектр звука а

На рис.3.8 видно, что основная частота находится между 200 и 300 Гц.

```
e_segment = wave_3_6.segment(start=0.9, duration=0.4)
e_segment.make_spectrum().plot(high=high)
decorate(xlabel='Frequency (Hz)')
```

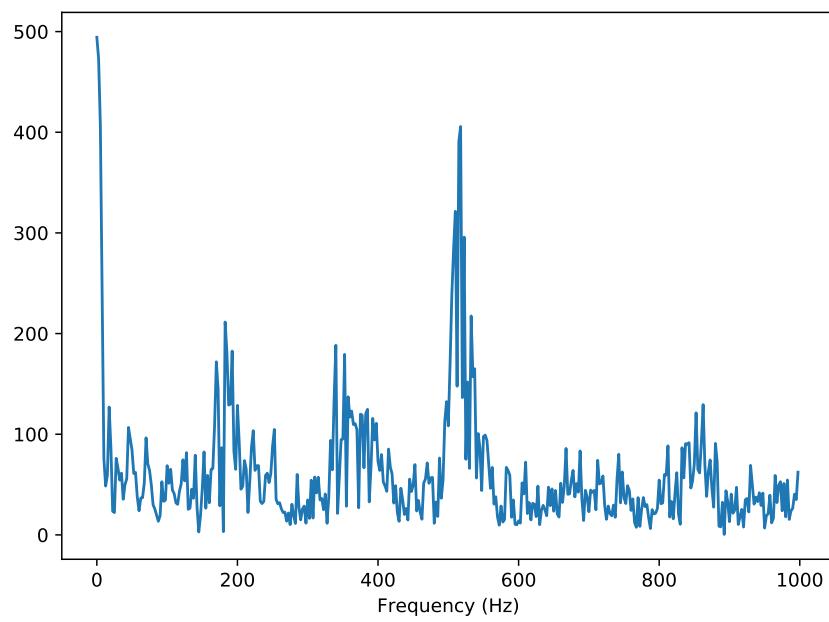


Рис. 3.9. Спектр звука е

Звук е имеет высокоамплитудную форманту около 500 Гц.

```
i_segment = wave_3_6.segment(start=1.8, duration=0.4)
i_segment.make_spectrum().plot(high=high)
decorate(xlabel='Frequency (Hz)')
```

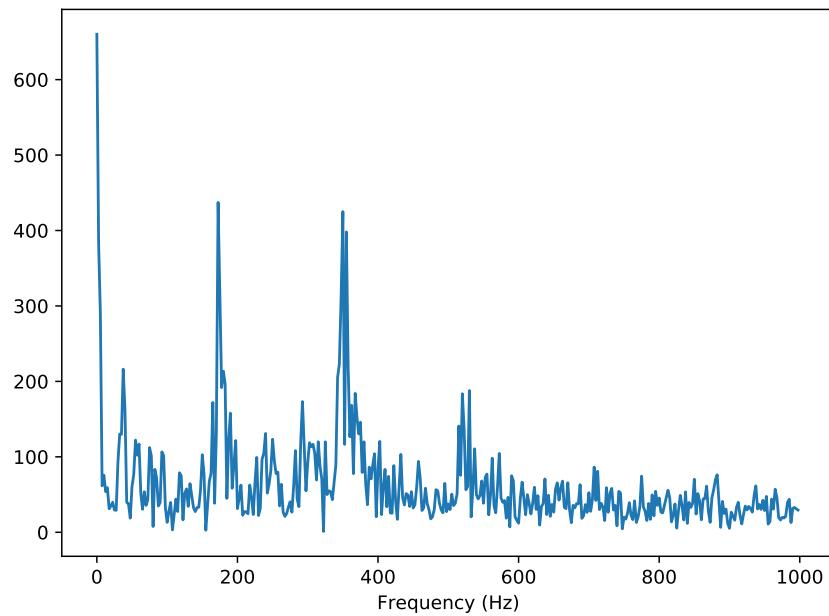


Рис. 3.10. Спектр звука i

На рис.3.10 мы видим два похожих пика на 200 и 400 Гц соответственно.

```
o_segment = wave_3_6.segment(start=2.8, duration=0.4)
o_segment.make_spectrum().plot(high=high)
decorate(xlabel='Frequency (Hz)')
```

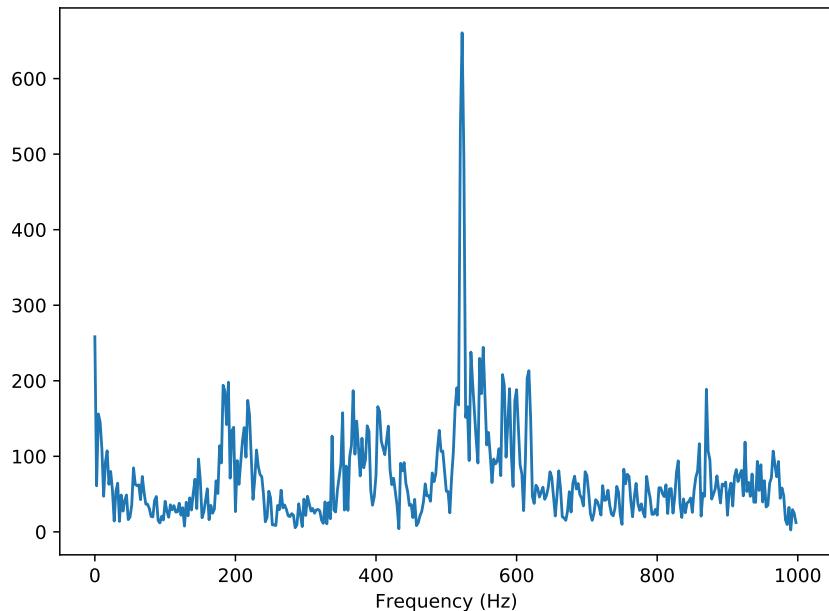


Рис. 3.11. Спектр звука о

На рис.3.11 видно что основная частота находится на 500 Гц.

```
u_segment = wave_3_6.segment(start=3.6, duration=0.25)
u_segment.make_spectrum().plot(high=high)
decorate(xlabel='Frequency (Hz)')
```

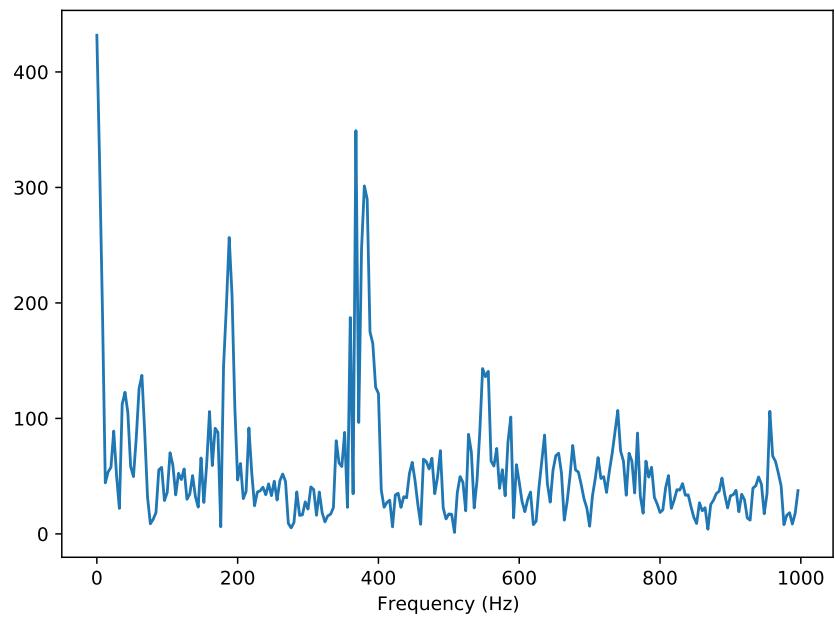


Рис. 3.12. Спектр звука и

На рис.3.12 мы видим два похожих пика на 200 и 400 Гц соответственно.

3.7. Выводы

В результате выполнения данной работы мы познакомились с понятиями апериодических сигналов, частотные компоненты которых изменяются во времени. К ним относятся практически все сигналы. А также с чирпами - сигналами с переменной частотой.

Кроме того, мы научились работать со спектрограммами и окнами.

Спектрограмма - это способ визуализации кратковременных ПФ. У неё на оси x время, а на оси y - частоты.

Окно - это функция, преобразующая апериодический сегмент во нечто похожее на периодическое.

4. Лабораторная работа 4

4.1. Упражнение 1

```
``A Soft Murmur'' is a web site that plays a mixture of natural noise sources, including rain, waves, wind, etc. At https://asoftmurmur.com/about/ you can find their list of recordings, most of which are at https://freesound.org.
```

Download a few of these files and compute the spectrum of each signal. Does the power spectrum look like white noise, pink noise, or Brownian noise? How does the spectrum vary over time?

Скачаем с сайта <https://freesound.org> два природных шума: звук ветра и тлеющего костра.

```
wave_1 = read_wave('..../139337_felix-blume_wind.wav')
wave_1.make_audio()
```

Выделим сегмент звука ветка длиной в одну секунду, вычислим его спектр и построим его график.

```
segment_1 = wave_1.segment(start=140, duration=1.0)
segment_1.make_audio()

spectrum_1 = segment_1.make_spectrum()
spectrum_1.plot_power()
decorate(xlabel='Frequency (Hz)')
```

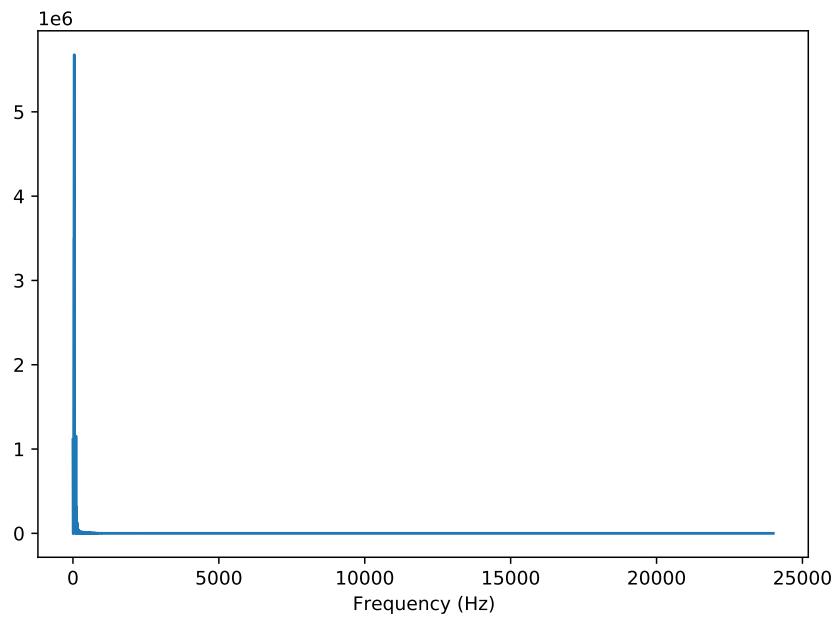


Рис. 4.1. Спектр звука ветра

Посмотрим на спектр звука ветра в логарифмической шкале.

```
spectrum_1.plot_power()  
loglog = dict(xscale='log',yscale='log')  
decorate(xlabel='Frequency (Hz)', **loglog)
```

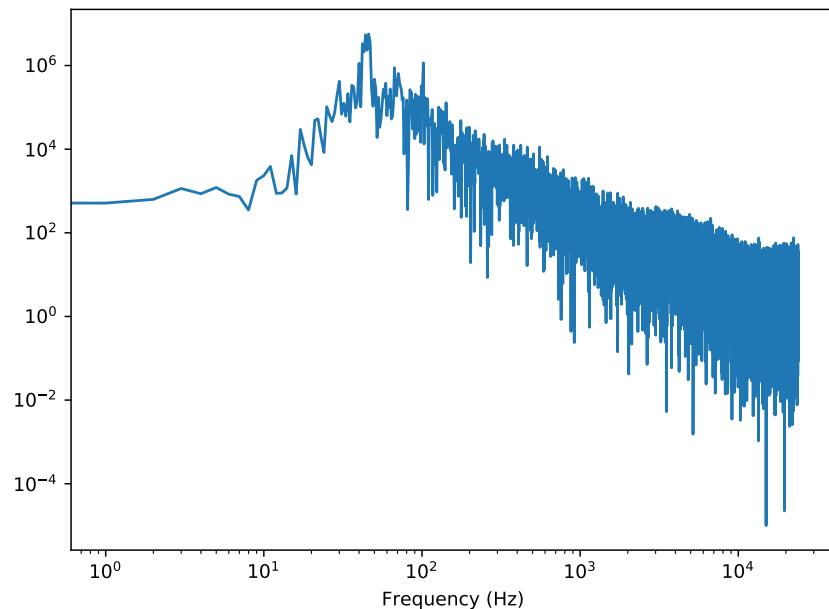


Рис. 4.2. Логарифмический спектр звука ветра

Наглон графика похож на график розового шума, но сложно сказать наверняка.

Построим спектrogramму звука ветра.

```
segment_1.make_spectrogram(512).plot(high=5000)
decorate(xlabel='Time(s)', ylabel='Frequency (Hz)')
```

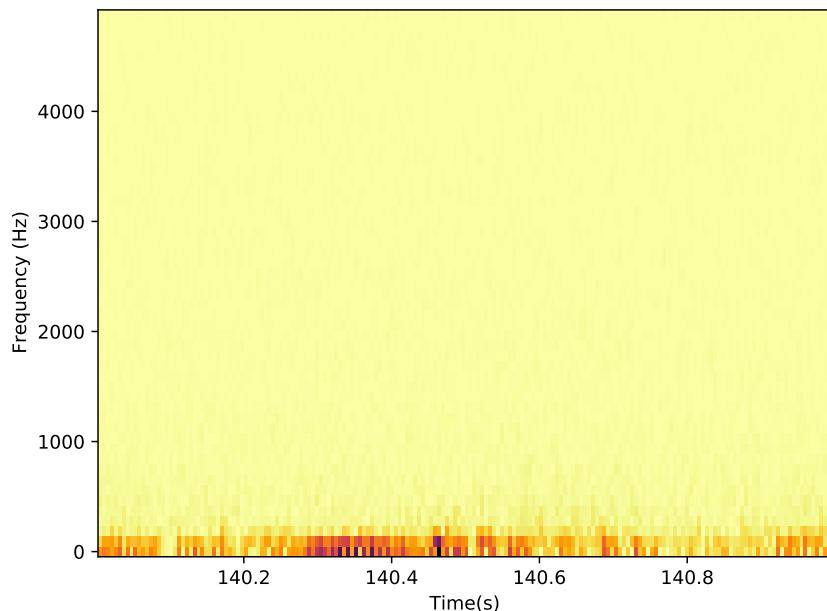


Рис. 4.3. Спектrogramма звука ветра

Воспользуемся функцией `estimate_slope`, чтобы узнать точный наклон спектра.

```
print(spectrum_1.estimate_slope().slope)
# -1.4066186066815058
```

Наклон примерно равен -1.4 , это довольно далеко от -2 , следовательно звук ветра - розовый шум.

Теперь проделаем тоже самое для звука костра. Выделим сегмент, вычислим спектр, построим по нему график и логарифмический график.

```
wave_1_2 = read_wave('..../83986_inchadney_fireplace.wav')
wave_1_2.make_audio()

segment_1_2 = wave_1_2.segment(start=8, duration=1.0)
segment_1_2.make_audio()

spectrum_1_2 = segment_1_2.make_spectrum()
spectrum_1_2.plot_power()
decorate(xlabel='Frequency (Hz)')
```

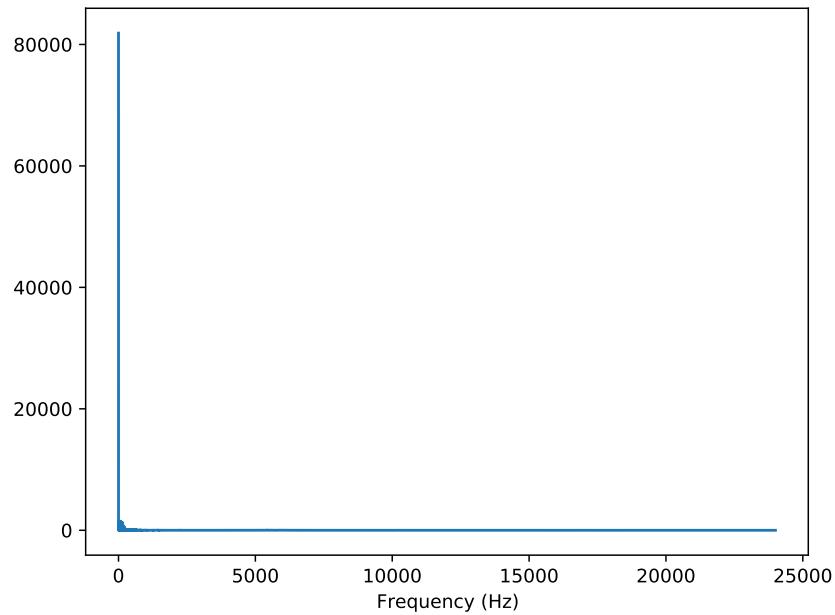


Рис. 4.4. Спектр звука костра

```
spectrum_1_2.plot_power()  
decorate(xlabel='Frequency (Hz)', **loglog)
```

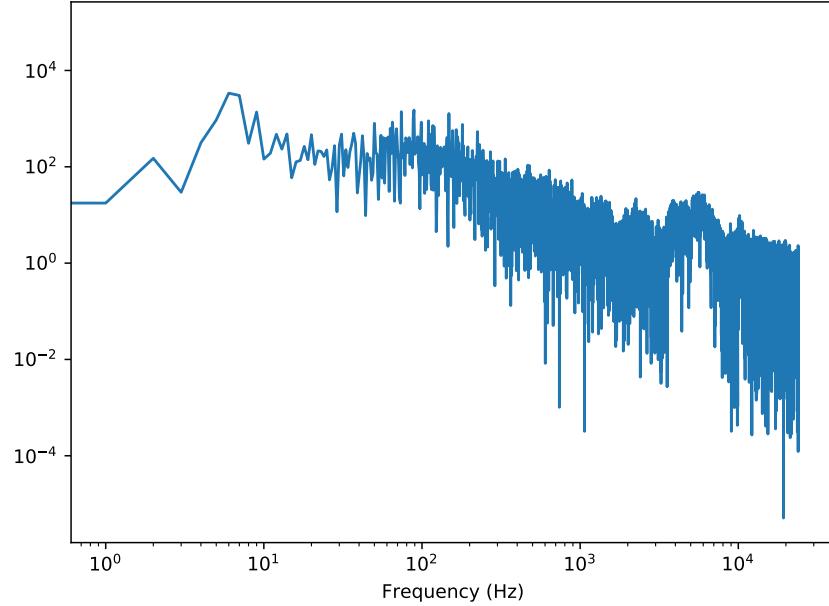


Рис. 4.5. Логарифмический спектр звука костра

Наклон у этого спектра чуть выше, чем у предыдущего, но, скорее всего, это тоже розовый шум.

Построим спектrogramму звука костра.

```
segment_1_2.make_spectrogram(512).plot(high=5000)
decorate(xlabel='Time(s)', ylabel='Frequency (Hz)')
```

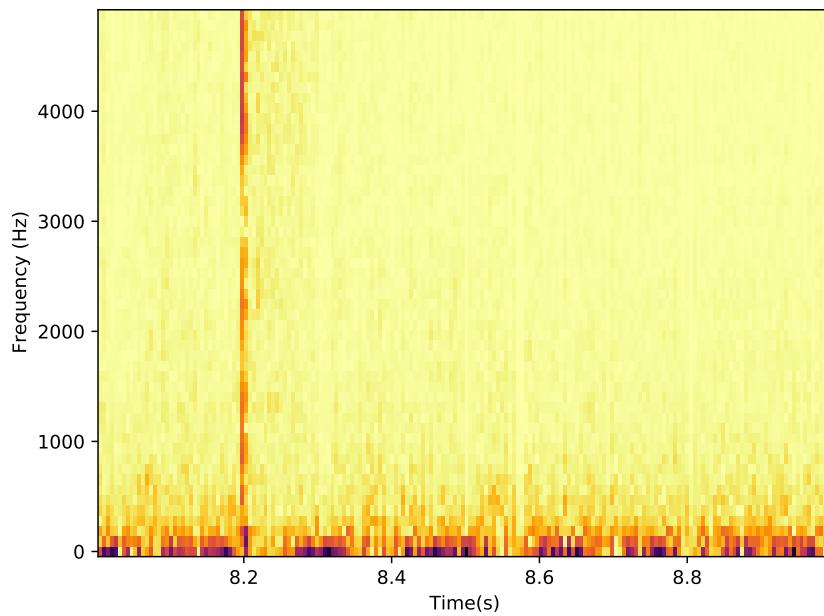


Рис. 4.6. Спектрограмма звука костра

Вычислим наклон спектра звука костра.

```
print(spectrum_1_2.estimate_slope().slope)
# -1.0766220332743197
```

Теперь наклон ближе к -1 , можно уверенно сказать, что это розовый шум.

4.2. Упражнение 2

In a noise signal, the mixture of frequencies changes over time. In the long run, we expect the power at all frequencies to be equal, but in any sample, the power at each frequency is random.

To estimate the long-term average power at each frequency, we can break a long signal into segments, compute the power spectrum for each segment, and then compute the average across the segments. You can read more about this algorithm at https://en.wikipedia.org/wiki/Bartlett's_method.

Implement Bartlett's method and use it to estimate the power spectrum for a noise wave. Hint: look at the implementation of `make_spectrogram`.

Реализуем метод Барретта.

```
def bartlett_spectrum(wave, seg_length=512, win_flag=True):
    spectrum_list = wave.make_spectrogram(seg_length,
                                           → win_flag).spec_map.values()
    psd_array = [spectrum.power for spectrum in spectrum_list]

    return Spectrum(np.sqrt(sum(psd_array) / len(psd_array)),
                    next(iter(spectrum_list)).fs,
                    wave framerate)
```

Затем используем его для оценки мощности разных сегментов из звука ветра.

```
psd_2_1 = bartlett_spectrum(wave_1.segment(start=120, duration=1.0))
psd_2_2 = bartlett_spectrum(wave_1.segment(start=150, duration=1.0))
psd_2_3 = bartlett_spectrum(wave_1.segment(start=160, duration=1.0))

psd_2_1.plot_power()
psd_2_2.plot_power()
psd_2_3.plot_power()

decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         **loglog)
```

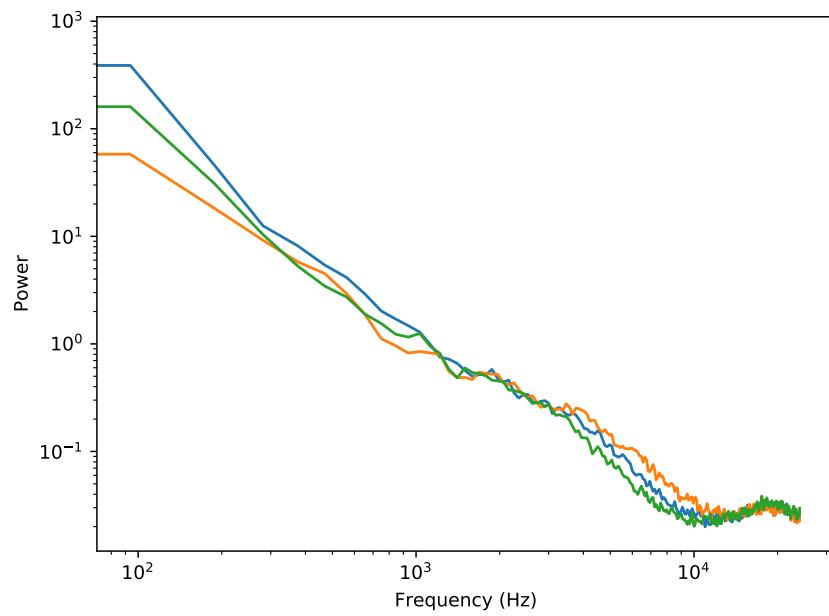


Рис. 4.7. Оценка мощности сегментов звука ветра

Можем видеть, что для разных сегментов связь между мощностью и частотой остаётся постоянной.

4.3. Упражнение 3

At <https://coindesk.com/price/bitcoin> you can download the daily price of a BitCoin as a CSV file. Read this file and compute the spectrum of BitCoin prices as a function of time. Does it resemble white, pink, or Brownian noise?

Скачаем с сайта <https://coindesk.com> ежедневную цену биткоина за всё время его существования. Воспользуемся библиотекой Pandas для работы с CSV таблицей.

```
df_3 = pd.read_csv('..../BTC_USD_2013-10-01_2021-05-08-CoinDesk.csv',
                   parse_dates=[0])
```

Создадим сигнал и его график.

```
ys_3 = df_3['Closing Price (USD)']
ts_3 = df_3.index

wave_3 = Wave(ys_3, ts_3, framerate=1)
wave_3.plot()
decorate(xlabel='Time (days)')
```

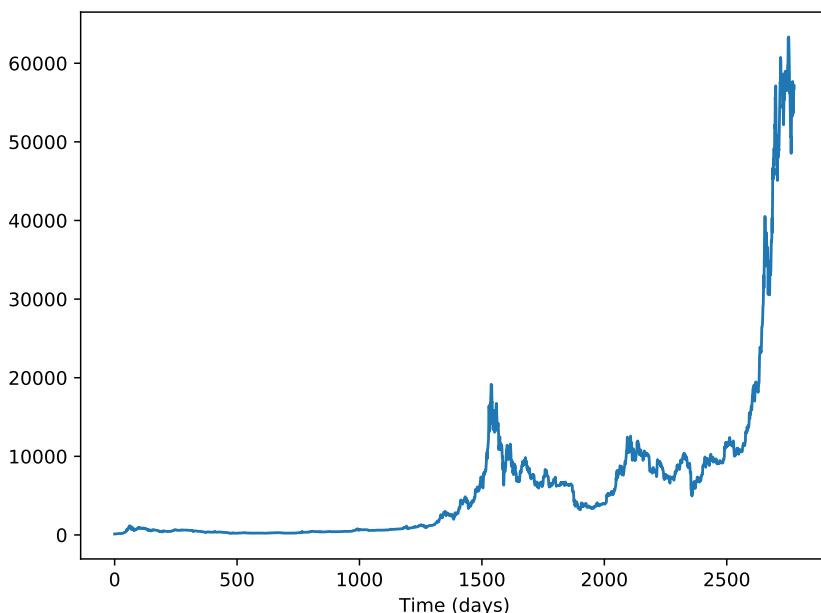


Рис. 4.8. График цены биткоина

Построим логарифмический спектр цены биткоина.

```
spectrum_3 = wave_3.make_spectrum()
spectrum_3.plot_power()
decorate(xlabel='Frequency (1/days)', **loglog)
```

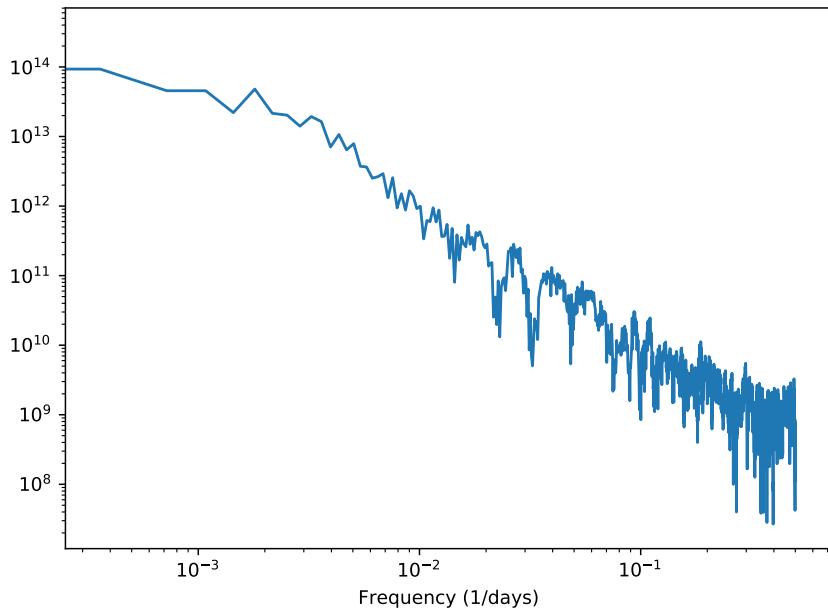


Рис. 4.9. Логарифмический спектр цены биткоина

По спектру можем предположить, что это либо розовый, либо красный шум. Вычислим точный наклон спектра.

```
print(spectrum_3.estimate_slope()[0])
# -1.7851119050620752
```

Наклон графика примерно равен -1.8 , то есть наклон находится в границах розового шума, но наклон очень близок к -2 , поэтому нельзя точно сказать, что это не красный шум.

4.4. Упражнение 4

```
Write a class called `UncorrelatedPoissonNoise` that inherits from `_Noise` and provides `evaluate`. It should use `np.random.poisson` to generate random values from a Poisson distribution. The parameter of this function, `lam`, is the average number of particles during each interval. You can use the attribute `amp` to specify `lam`. For example, if the framerate is 10 kHz and `amp` is 0.001, we expect about 10 "clicks" per second.
```

```
Generate about a second of UP noise and listen to it. For low values of `amp`, like 0.001, it should sound like a Geiger counter. For higher values it should sound like white noise. Compute and plot the power spectrum to see whether it looks like white noise.
```

Напишем класс UncorrelatedPoissonNoise, наследующий класс _Noise и переопределяющий метод evaluate. Используем np.random.poisson для генерации случайных значений по распределению Пуассона.

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

Сгенерируем секунду шума, используя этот класс с амплитудой 0.001 и частотой 10кГц.

```
amp_4 = 0.001
framerate_4 = 10000
duration_4 = 1

signal_4 = UncorrelatedPoissonNoise(amp=amp_4)
wave_4 = signal_4.make_wave(duration=duration_4, framerate=framerate_4)
wave_4.make_audio()
```

Получившийся звук действительно похож на счётчик Гейгера.

Сравним получившиеся количество "частиц" с ожидаемым.

```
expected_4 = amp_4 * framerate_4 * duration_4
actual_4 = sum(wave_4.ys)
print(expected_4, actual_4)
# 10.0 10
```

Построим график сигнала.

```
wave_4.plot()
```

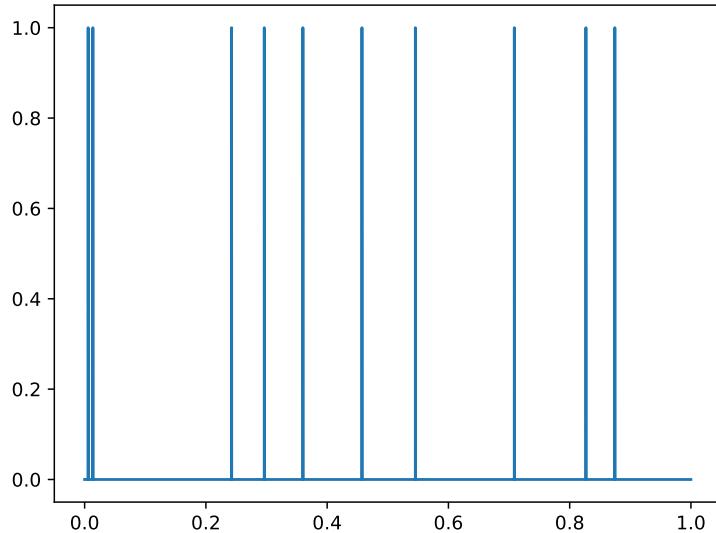


Рис. 4.10. График при низкой амплитуде

Теперь увеличим амплитуду.

```
amp_4 = 2

signal_4_2 = UncorrelatedPoissonNoise(amp=amp_4)
wave_4_2 = signal_4_2.make_wave(duration=duration_4, framerate=framerate_4)
wave_4_2.make_audio()
```

Теперь звук действительно больше похож на белый шум.

Сравним ожидаемое и действительное количество "частиц".

```
expected_4_2 = amp_4 * framerate_4 * duration_4
actual_4_2 = sum(wave_4_2.ys)
print(expected_4_2, actual_4_2)
# 20000 20051
```

Построим график нового сигнала.

```
wave_4_2.plot()
```

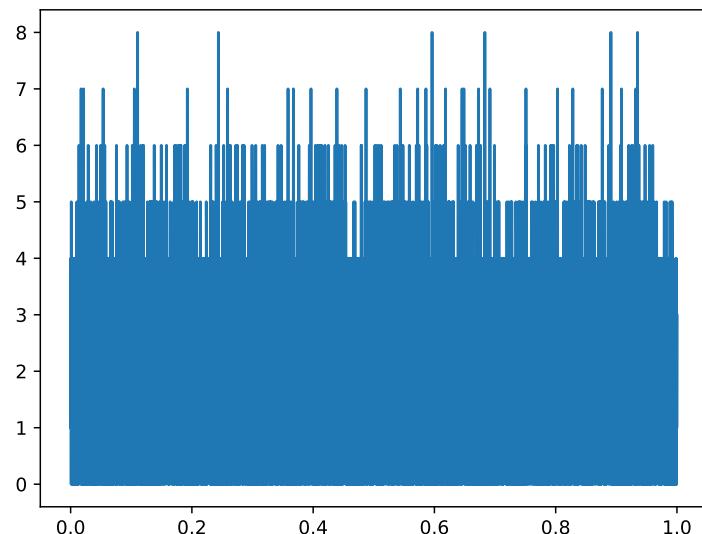


Рис. 4.11. График при большой амплитуде

Построим спектры мощности для обоих сигналов.

```
spectrum_4 = wave_4.make_spectrum()
spectrum_4_2 = wave_4_2.make_spectrum()
spectrum_4.plot_power(alpha=0.5)
spectrum_4_2.plot_power(alpha=0.5)
decorate(xlabel='Frequency (Hz)',
         ylabel='Power',
         **loglog)
```

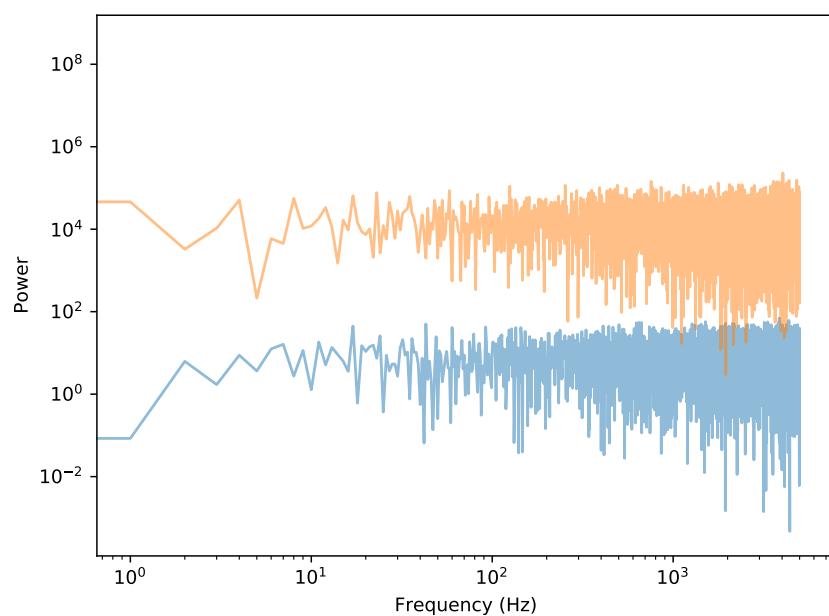


Рис. 4.12. Спектры мощности для двух сигналов

Наклоны графиков практически полностью совпадают и по ним можно понять, что получился действительно белый шум.

4.5. Упражнение 5

The algorithm in this chapter for generating pink noise is conceptually simple but computationally expensive. There are more efficient alternatives, like the Voss-McCartney algorithm. Research this method, implement it, compute the spectrum of the result, and confirm that it has the desired relationship between power and frequency.

Напишем функцию, создающую розовый шум по методу Восса-Маккарти. Данный алгоритм имеет сложность $O(n * \log(n))$.

```
def voss_mccartney_pink_noise(n_rows, n_cols=16):
    array = np.empty((n_rows, n_cols))
    array.fill(np.nan)
    array[0, :] = np.random.random(n_cols)
    array[:, 0] = np.random.random(n_rows)

    cols = np.random.geometric(0.5, n_rows)
    cols[cols >= n_cols] = 0
    rows = np.random.randint(n_rows, size=n_rows)
    array[rows, cols] = np.random.random(n_rows)

    df = pd.DataFrame(array)
    df.fillna(method='ffill', axis=0, inplace=True)

    # noinspection PyArgumentList
    return df.sum(axis=1).values
```

Создадим сигнал сначала с маленьким количеством значений, затем с большим.

```
wave = Wave(voss_mccartney_pink_noise(2 ** 10))
wave.unbias()
wave.normalize()

wave.plot()
```

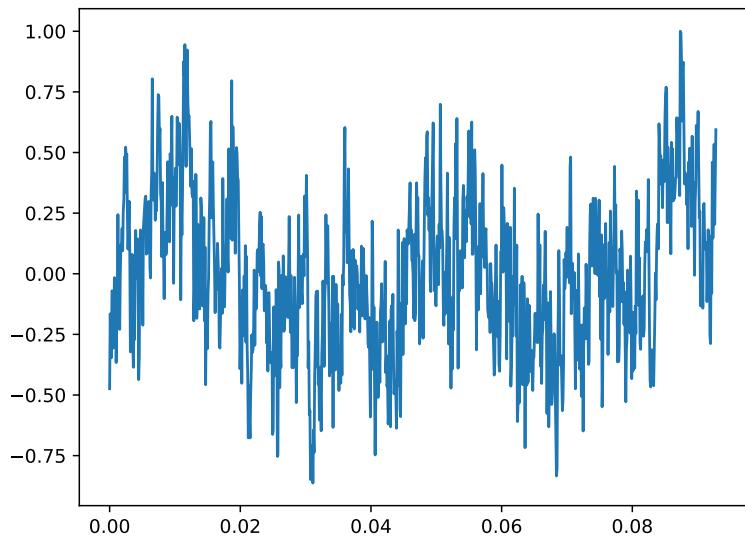


Рис. 4.13. График розового шума

```
wave.make_audio()

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

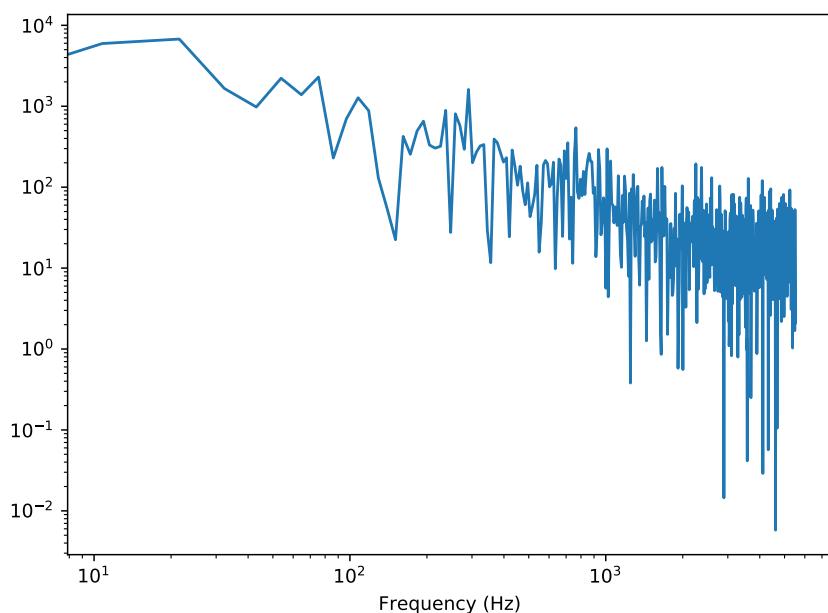


Рис. 4.14. Логарифмический спектр мощности розового шума

Наклон получившегося спектра близок к -1, значит это точно розовый шум.

```
print(spectrum.estimate_slope().slope)
# -0.9961332129929098
```

Воспользуемся методом Бартлетта из 4.2, для того, чтобы проверить соотношение между мощностью и частотой на большей выборке.

```
seg_length = 2 ** 16
iterations_amount = 100
wave_5 = Wave(voss_mccartney_pink_noise(seg_length * iterations_amount))

spectrum = bartlett_spectrum(wave_5, seg_length=seg_length, win_flag=False)
spectrum.hs[0] = 0

spectrum.plot_power()
decorate(xlabel='Frequency (Hz)',
         **loglog)
```

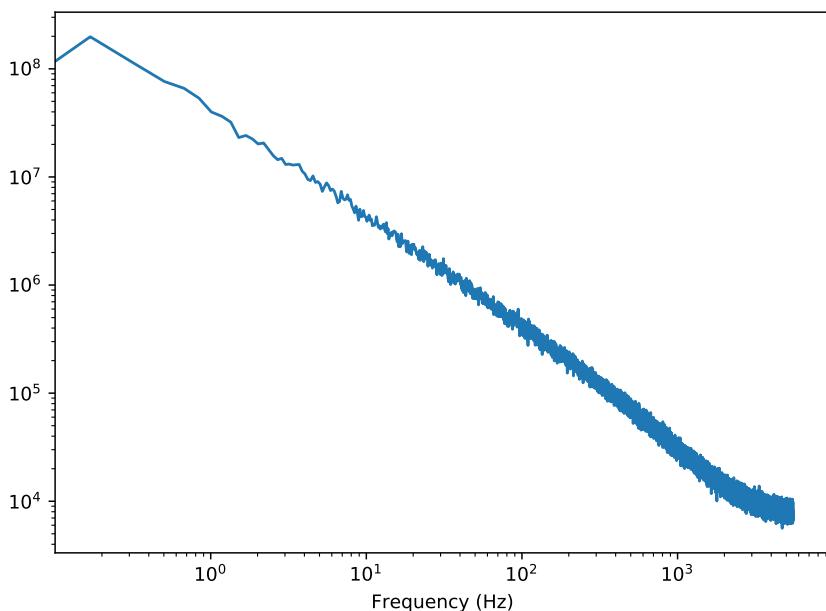


Рис. 4.15. Соотношение мощности и частоты у розового шума

График соотношения очень близок к прямой, за исключением небольших отклонений на высоких частотах. Наклон этого графика также примерно равен -1.

```
print(spectrum.estimate_slope().slope)
# -1.0009495578549672
```

4.6. Выводы

В результате выполнения данной работы мы познакомились с различными видами шумов и для работы с ними изучили метод Бартлетта и алгоритм Восса-Маккарти. Метод Бартлетта используется для оценки спектров мощности, а алгоритм Восса-Маккарти позволяет более эффективно генерировать розовый шум.

5. Лабораторная работа 5

5.1. Упражнение 1

The Jupyter notebook for this chapter, chap05.ipynb, includes an interaction that lets you compute autocorrelations for different lags. Use this interaction to estimate the pitch of the vocal chirp for a few different start times.

Скопируем запись вокального чирпа и выделим из нее один сегмент небольшой длительности.

```
wave_1 = read_wave('..../28042_bcjordan_voicedownbew.wav')
wave_1.normalize()
segment_1 = wave_1.segment(duration=0.01)
```

Далее воспользуемся автокорреляцией и построим получившийся график зависимости.

```
lags_1, correlation_1 = autocorr(segment_1)
plt.plot(lags_1, correlation_1)
decorate(xlabel='Lags', ylabel='Correlation')
```

Взглянем на график, чтобы определить границы, в которых находится искомый пик.

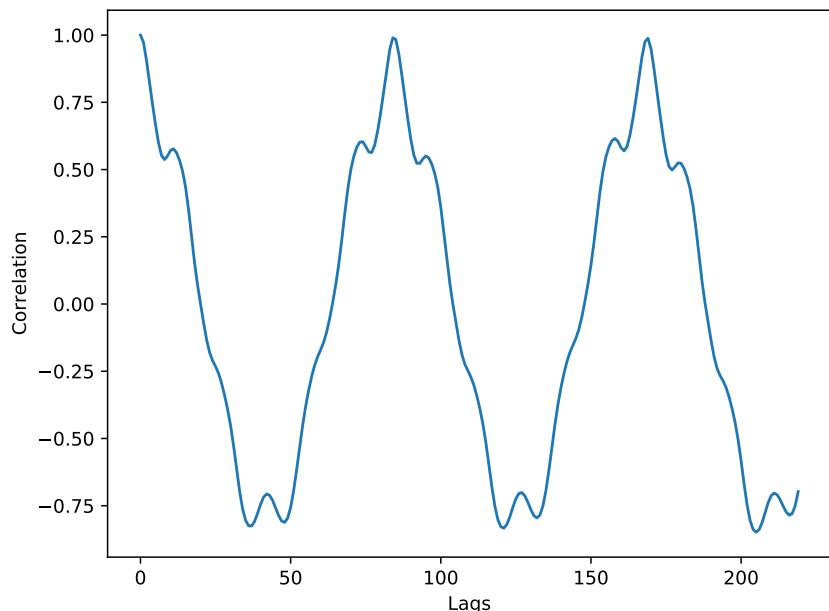


Рис. 5.1. График автокорреляции первого сегмента

Первый пик находится между значениями 50 и 100. Для уточнения значения пика воспользуемся функцией `argmax`.

```
peak_1 = np.array(correlation_1[50:100]).argmax() + 50
print(peak_1)
```

Уточненное значение `lab` равно 84. Вычислим частоту полученного пика.

```
frequency_1 = segment_1.framerate / peak_1
print(frequency_1)
```

Вычисленное значение составило 525 Гц. Повторяем предыдущие действия с автокорреляцией для другого фрагмента записи.

```
segment_1 = wave_1.segment(start=1, duration=0.01)
lags_1, correlation_1 = autocorr(segment_1)
plt.plot(lags_1, correlation_1)
decorate(xlabel='Lags', ylabel='Correlation')
```

Взглянем на график для определения границ пика.

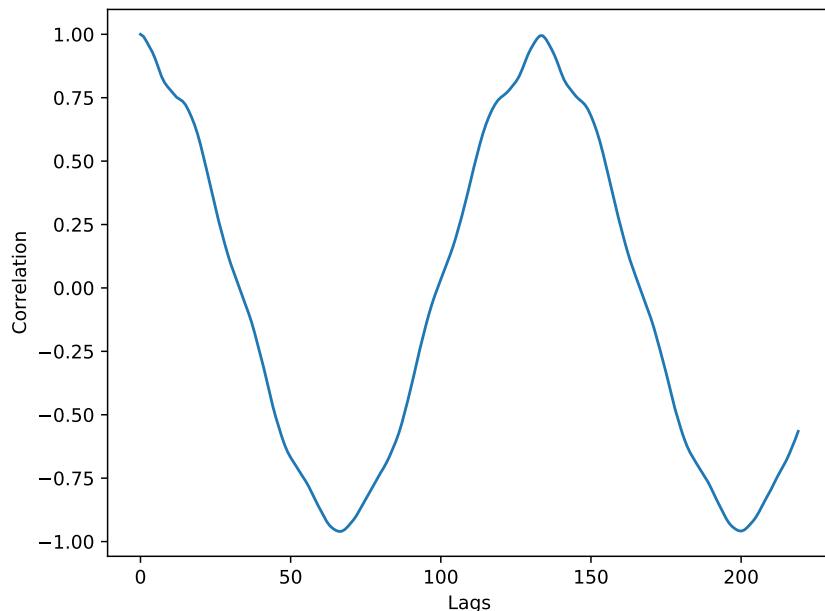


Рис. 5.2. График автокорреляции второго сегмента

Второй пик находится между значениями 100 и 150. Уточним его значение с помощью функции `argmax`.

```
peak_1 = np.array(correlation_1[100:150]).argmax() + 100
print(peak_1)
```

Уточненное значение `lab` равно 134. Затем вычислим его частоту.

```
frequency_1 = segment_1.framerate / peak_1
print(frequency_1)
```

Вычисленное значение составило 329.1 Гц. В результате данного упражнения приходим к выводу что при увеличении значения для начала сегмента записи уменьшается частота для пиков.

5.2. Упражнение 2

The example code in chap05.ipynb shows how to use autocorrelation to estimate the fundamental frequency of a periodic signal. Encapsulate this code in a function called estimate_fundamental, and use it to track the pitch of a recorded sound.

Поскольку задача определения границ для уточнения и определения пика достаточно трудоемкая, то эти границы будут указываться вручную при вызове функции. Создадим необходимую функцию с данным условием.

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

Проверим полученную функцию на ранее расчитанных данных.

```
wave_2 = read_wave('.. /28042_bcjordan_voicedownbew.wav')
wave_2.normalize()
segment_2 = wave_2.segment(duration=0.01)
frequency_2 = estimate_fundamental(segment_2, 50, 100)
print(frequency_2)
```

Полученное значение равно 525 Гц, что соответствует посчитаному ранее значению. Построим спектrogramму записи, ограничим высоту графика для лучшего изображения.

```
spectrogram_2 = wave_2.make_spectrogram(2048)
spectrogram_2.plot(high=1000)
decorate(xlabel='Time', ylabel='Frequency')
plt.savefig(filePath + "2.wave.spectrogram" + fileExtension)
```

Взглянем на получившуюся спектrogramму.

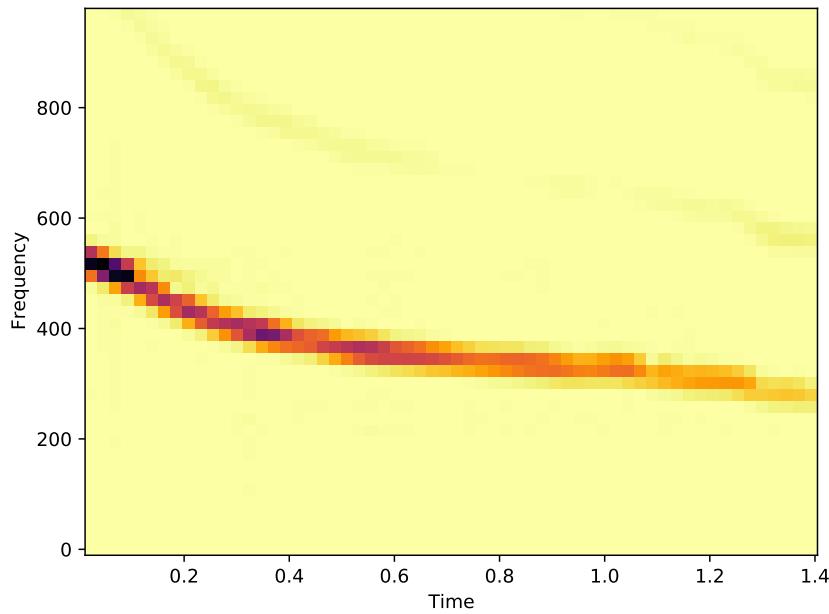


Рис. 5.3. График спектограммы сегмента

Сформируем списки для исходных и конечных данных, нужных для последовательного наложения оценки высоты на спектрограмму.

```
start_2 = 0.0
end_2 = 1.4
step_2 = (end_2 - start_2) / 100
values_2 = np.arange(start_2, end_2, step_2)
timestamps_2 = []
frequency_list_2 = []
```

Заполним списки для выходных данных, используя созданную функцию.

```
for value in values_2:
    timestamps_2.append(value + step_2 / 2)
    segment = wave_2.segment(start=value, duration=0.01)
    frequency = estimate_fundamental(segment, 70, 150)
    frequency_list_2.append(frequency)
```

Построим график для получившихся значений, белой линией будет указана наложенная оценка высоты.

```
spectrogram_2 = wave_2.make_spectrogram(2048)
spectrogram_2.plot(high=1000)
plt.plot(timestamps_2, frequency_list_2, color='white')
decorate(xlabel='Time', ylabel='Frequency')
```

Посмотрим на получившийся график.

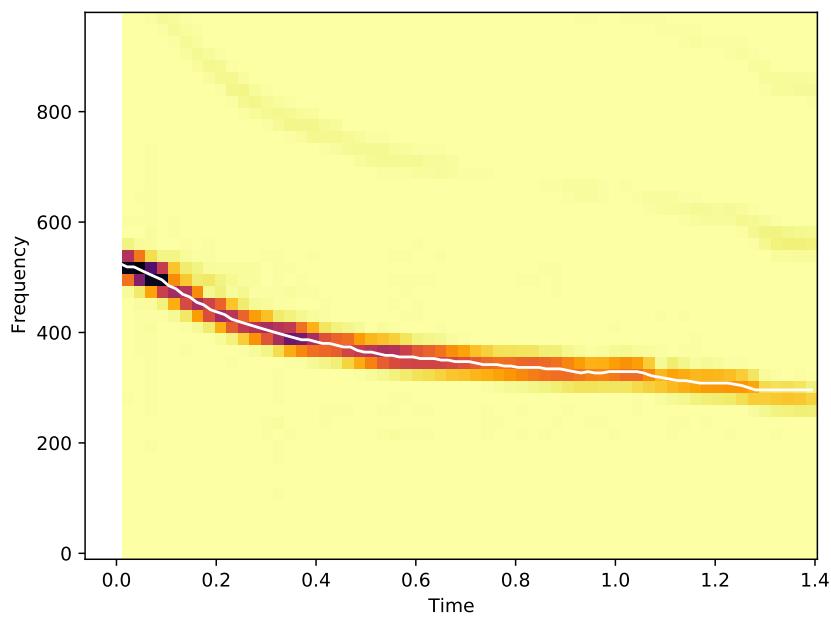


Рис. 5.4. График спектрограммы с наложенной оценкой высоты

Белая линия совпадает с кривой на спектрограмме, что говорит о правильной работе созданной функции.

5.3. Упражнение 3

Exercise 5.3

If you did the exercises in the previous chapter, you downloaded the historical price of BitCoins and estimated the power spectrum of the price changes. Using the same data, compute the autocorrelation of BitCoin prices. Does the autocorrelation function drop off quickly? Is there evidence of periodic behavior?

Сначала прочитаем полученные данные о Bitcoin и построим по ним график.

```
data_frame_3 = pd.read_csv('..../BTC_USD_2013-10-01_2021-05-08-CoinDesk.csv',
                           parse_dates=[0])
ys_3 = data_frame_3['Closing Price (USD)']
timestamps_3 = data_frame_3.index
wave_3 = Wave(ys_3, timestamps_3, framerate=1)
wave_3.plot()
decorate(xlabel='Days', ylabel='Price')
```

Посмотрим получившийся график.

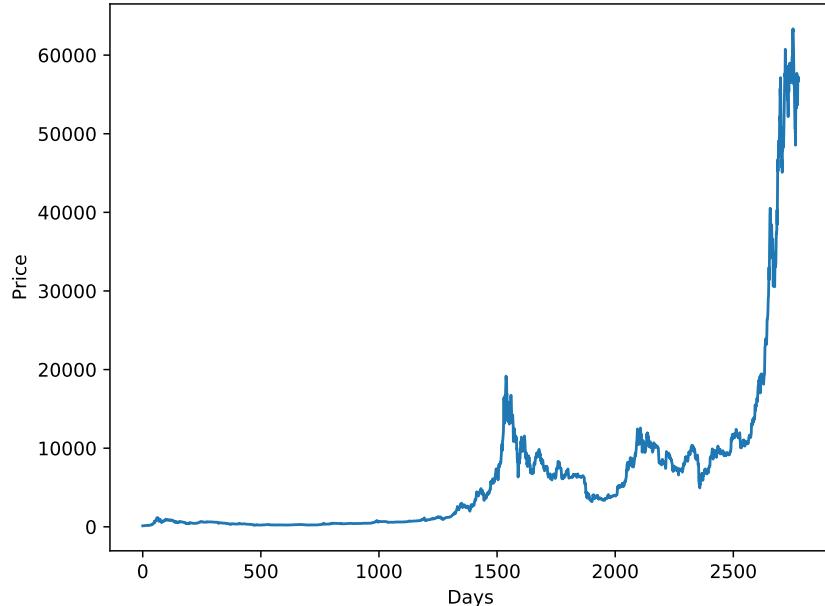


Рис. 5.5. График изменения цены Bitcoin

Затем вычислим автокоррекцию.

```
lags_3, correlation_3 = autocorr(wave_3)
plt.plot(lags_3, correlation_3)
decorate(xlabel='Lags', ylabel='Correlation')
```

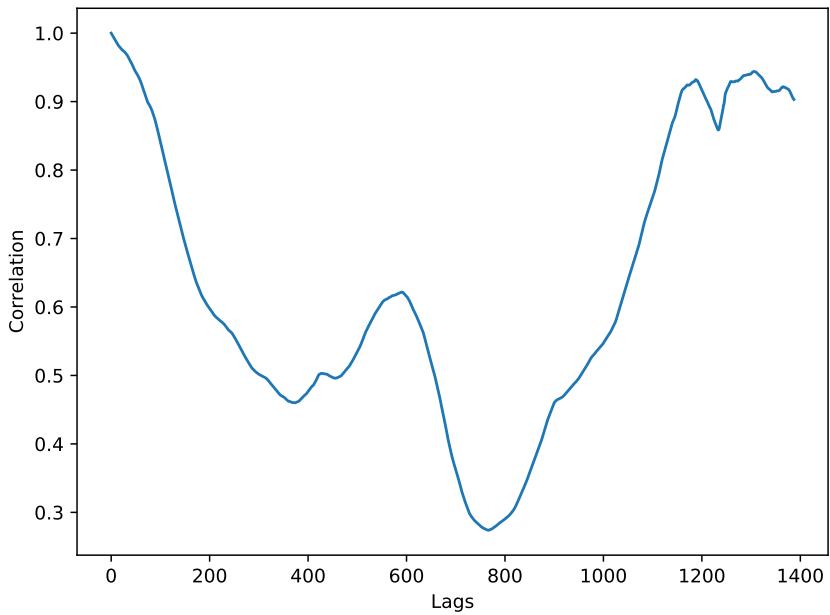


Рис. 5.6. График автокорреляции цены Bitcoin

В предыдущей лабораторной работе было вычислено, что перед нами розовый шум, который очень близок к красному, и по данному графику видно, что курс биткоина действительно является розовым шумом, близким к красному.

5.4. Упражнение 4

In the repository for this book you will find a Jupyter notebook called `saxophone.ipynb` that explores autocorrelation, pitch perception, and a phenomenon called the missing fundamental. Read through this notebook and run the examples. Try selecting a different segment of the recording and running the examples again.

Vi Hart has an excellent video called “What is up with Noises? (The Science and Mathematics of Sound, Frequency, and Pitch)”; it demonstrates the missing fundamental phenomenon and explains how pitch perception works (at least, to the degree that we know). Watch it at
↪ https://www.youtube.com/watch?v=i_0DXxNeaQ0.

По примеру `saxophone.ipynb` сначала скопируем запись.

```
wave_4 = read_wave('..../100475_iluppai_saxophone-weep.wav')
wave_4.normalize()
wave_4.make_audio()
```

Затем посмотрим гармоническую структуру записи.

```
gram_4 = wave_4.make_spectrogram(seg_length=1024)
gram_4.plot(high=3000)
decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

Взглянем на построенный график.

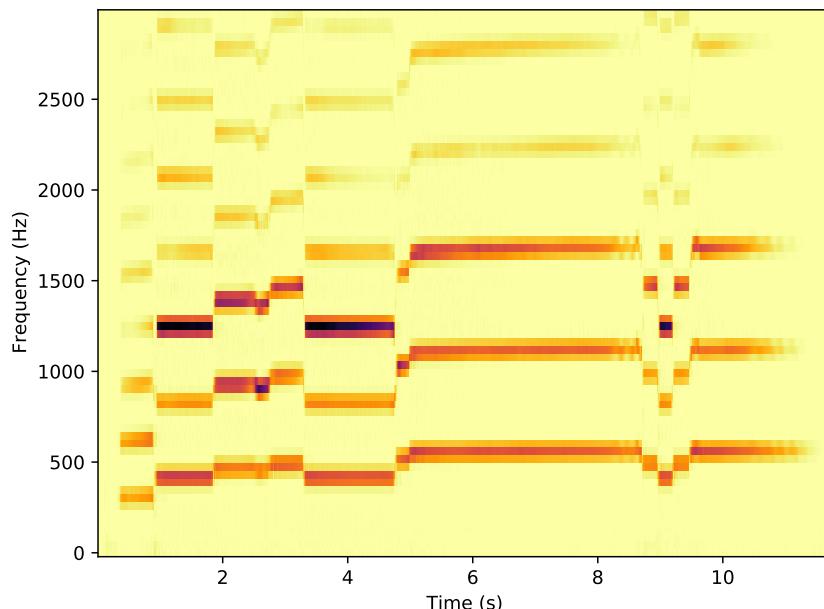


Рис. 5.7. График гармонической структуры записи

После этого выберем другой промежуток записи для рассмотрения.

```
start_4 = 1.0
duration_4 = 0.5
segment_4 = wave_4.segment(start=start_4, duration=duration_4)
segment_4.make_audio()
```

Построим спектр этого промежутка.

```
spectrum_4 = segment_4.make_spectrum()
spectrum_4.plot(high=3000)
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

И взглянем на график этого спектра.

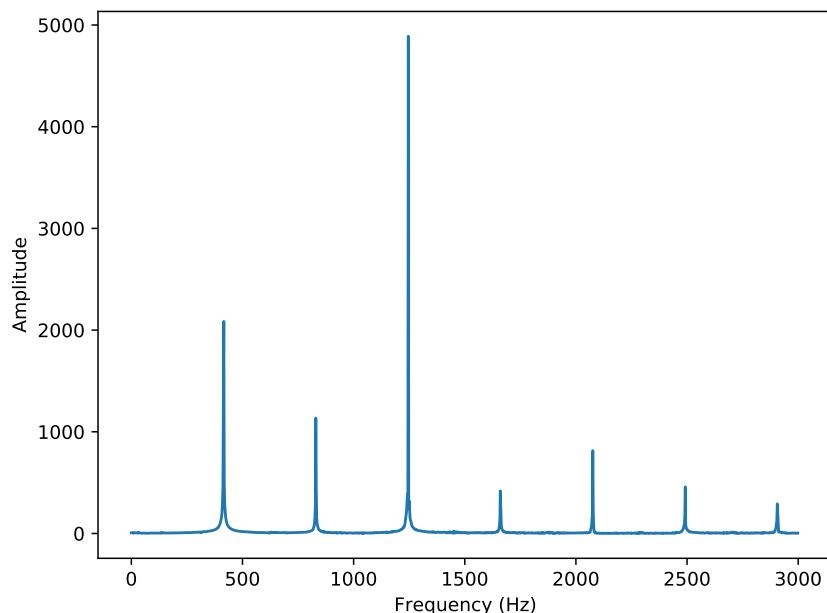


Рис. 5.8. График спектра выбранного сегмента

Затем рассмотрим значения пиков.

```
print(spectrum_4.peaks()[:10])
```

Основные частоты тут 1246 Гц, 416 Гц, 830 Гц. Высота 416 Гц будет восприниматься как основная, хоть она и не доминирующая, для примера возьмем треугольную волну с такой же частотой и послушаем ее, чтобы убедиться, что она очень похожа на нашу.

```
TriangleSignal(freq=416).make_wave(duration=0.5).make_audio()
segment_4.make_audio()
```

Чтобы понять, почему мы воспринимаем основную высоту, не являющуюся доминирующей, используем автокорреляцию.

```
lags_4, corrs_4 = autocorr(segment_4)
plt.plot(corrs_4[:200])
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1.05, 1.05])
```

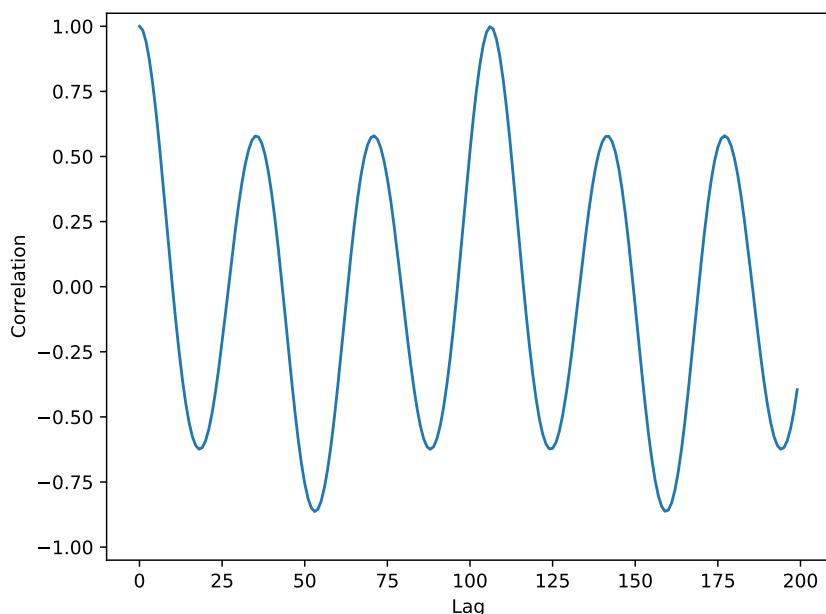


Рис. 5.9. График автокорреляции сегмента

Первый пик находится между значениями 100 и 125, рассчитаем его частоту.

```
estimate_fundamental(segment_4, 100, 125)
```

Это значение соответствует 416Гц, что значит что мы воспринимаем наивысший пик корреляционной функции, а не самый высокий компонент спектра. Теперь удалим основную частоту.

```
spectrum_4_2 = segment_4.make_spectrum()
spectrum_4_2.high_pass(450)
spectrum_4_2.plot(high=3000)
```

Посмотрим как это выглядит на графике

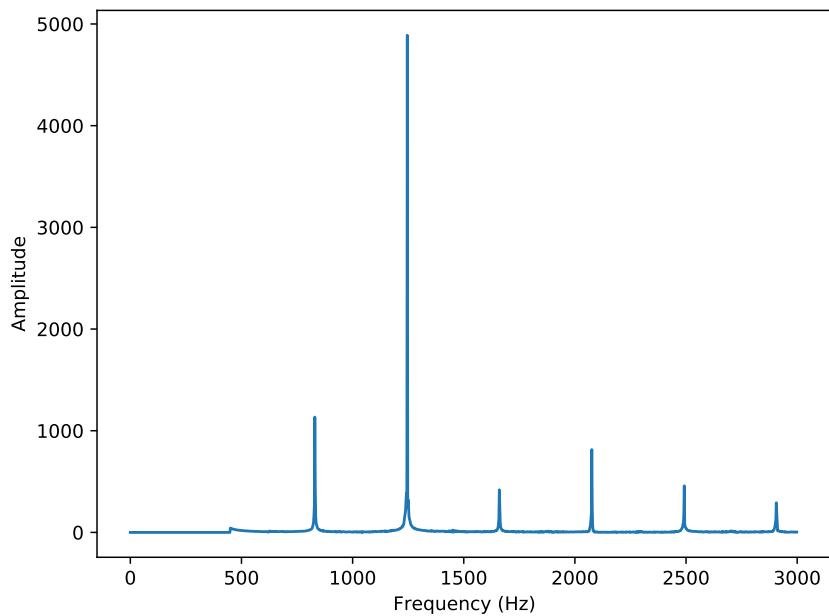


Рис. 5.10. График сегмента без основной частоты

Прослушаем полученную запись.

```
segment_4_2 = spectrum_4_2.make_wave()
segment_4_2.make_audio()
```

Воспринимаемая высота звука до сих пор на 416 Гц, хотя мы убрали пик на этой частоте, это явление называется "подавленная основная". Обратимся к автокорреляции для того, чтобы понять, что происходит.

```
lags_4_2, corrs_4_2 = autocorr(segment_4_2)
plt.plot(corrs_4_2[:200])
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1.05, 1.05])
```

Посмотрим на получившийся график автокорреляции

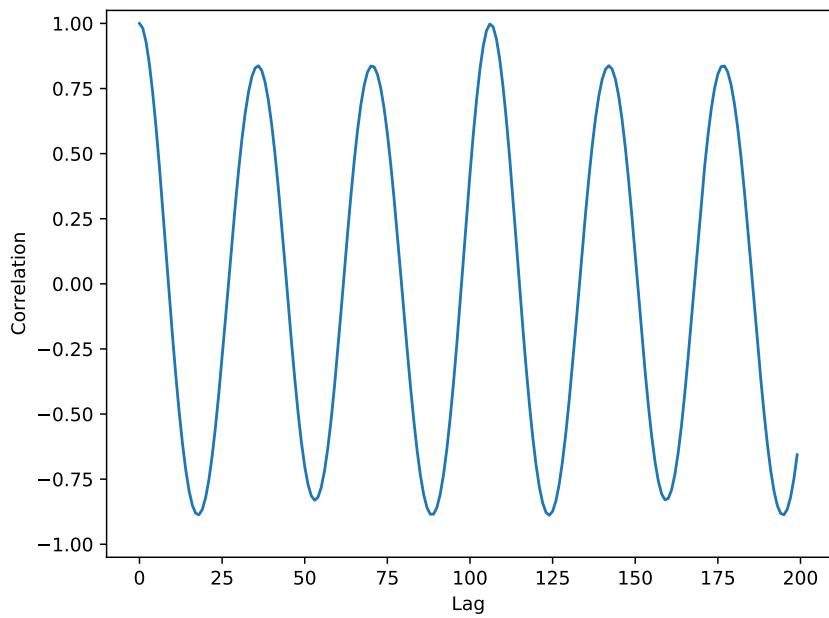


Рис. 5.11. График автокорреляции сегмента без основной частоты

Наивысший пик все еще соответствует 416Гц, помимо этого у нас есть и другие гармоники, которые надо рассчитать.

```
estimate_fundamental(segment_4_2, 25, 50)
estimate_fundamental(segment_4_2, 50, 90)
```

Получены значения 1260 Гц и 621 Гц. Если избавимся от гармоник выше 1200 Гц, то эффект пропадет.

```
spectrum_4_3 = segment_4_2.make_spectrum()
spectrum_4_3.high_pass(450)
spectrum_4_3.low_pass(1200)
spectrum_4_3.plot(high=3000)
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

Посмотрим на новый график

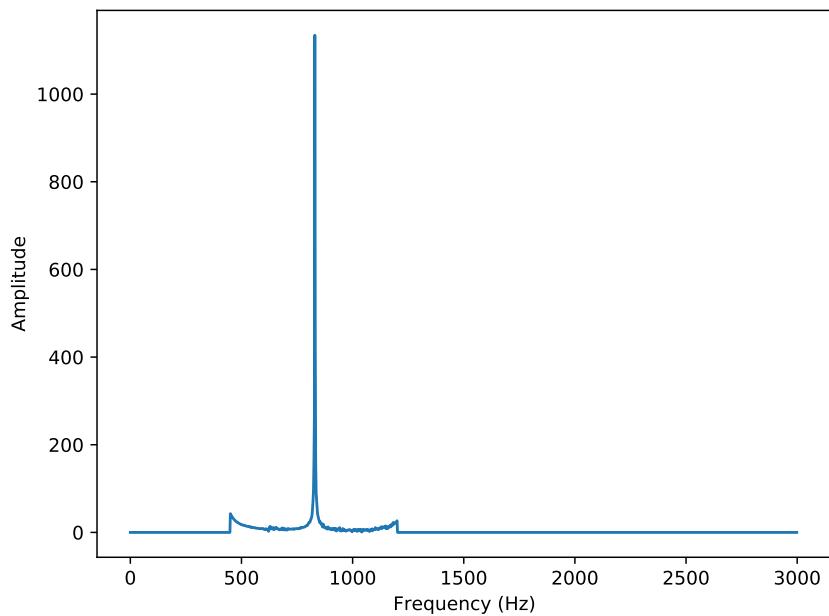


Рис. 5.12. График сегмента без основной частоты и гармоник выше 1200 Гц

Прослушаем новую запись.

```
segment_4_3 = spectrum_4_3.make_wave()
segment_4_3.make_audio()
```

Звук теперь стал более механическим и действительно имеет более высокую частоту. Обратимся снова к автокорреляции.

```
lags_4_3, corrs_4_3 = autocorr(segment_4_3)
plt.plot(corrs_4_3[:200])
decorate(xlabel='Lag', ylabel='Correlation', ylim=[-1.05, 1.05])
```

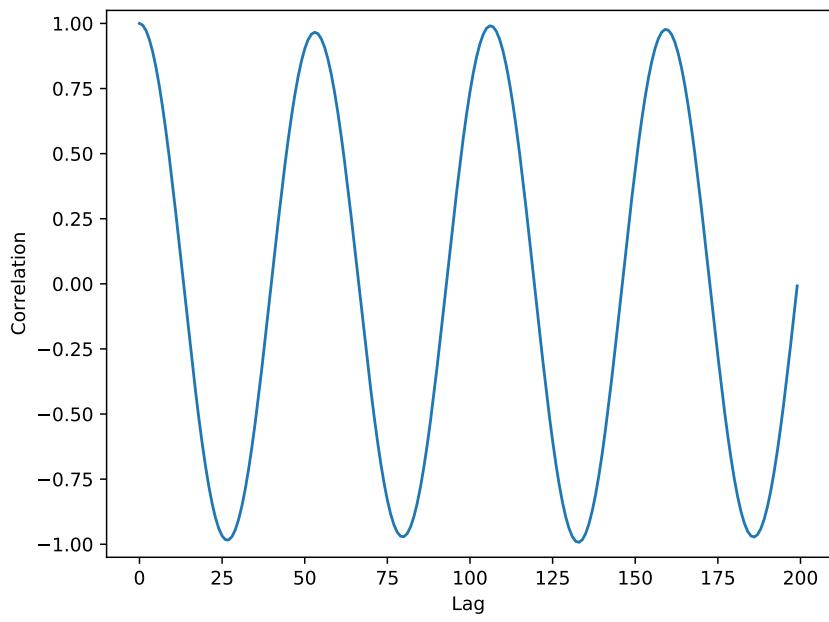


Рис. 5.13. График автокорреляции сегмента без гармоник выше 1200 Гц

Теперь основная частота находится на 830 Гц. Для сравнения создадим треугольную волну с теми же параметрами и прослушаем ее.

```
TriangleSignal(freq=830).make_wave(duration=0.5).make_audio()
```

Звучание у треугольной волны с частотой 830 Гц и у получившегося сигнала похожи, что подтверждает слова о том, что основная частота находится на 830 Гц.

5.5. Выводы

В данной работе мы познакомились с автокорреляцией, научились пользоваться ей для оценки высоты тона, а также познакомились с понятием "подавленная основная", экспериментально выяснили, что восприятие высоты звука не полностью основано на спектральном анализе, но также определяется автокорреляцией.

6. Лабораторная работа 6

6.1. Упражнение 1

```
Exercise 6.1 In this chapter I claim that analyze1 takes time proportional
to n^3 and analyze2 takes time proportional to n^2. To see if that's true,
↪ run
them on a range of input sizes and time them. In Jupyter, you can use the
“magic command” %timeit.
If you plot run time versus input size on a log-log scale, you should get a
straight line with slope 3 for analyze1 and slope 2 for analyze2.
You also might want to test dct_iv and scipy.fftpack.dct.
```

Убедимся в том, что analyze1 требует времени пропорционально n^3 , а analyze2 - пропорционально n^2 . Для этого будем запускать их и засекать время работы. Сначала создадим шумовой сигнал.

```
signal_6_1 = UncorrelatedGaussianNoise()
noise_6_1 = signal.make_wave(duration=1.0, framerate=16384)
```

Затем построим массив из степеней двойки.

```
ns = 2 ** np.arange(5, 14)
```

Далее нам понадобятся две функции run_speed_test и plot_bests.

```
def run_speed_test(ns, func):
    results = []
    for N in ns:
        print(N)
        ts = (0.5 + np.arange(N)) / N
        freqs = (0.5 + np.arange(N)) / 2
        ys = noise_6_1.ys[:N]
        result = %timeit -r1 -o func(ys, freqs, ts)
        results.append(result)

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

loglog = dict(xscale='log',yscale='log')

def plot_bests(ns, bests):
    plt.plot(ns, bests)
```

```
decorate(**loglog)
x = np.log(ns)
y = np.log(best)
t = linregress(x, y)
slope = t[0]

return slope
```

Используя функцию `run_speed_test`, получим результаты работы `analyze1` (Рис.6.1), а затем с помощью `plot_bests` построим график на основании полученных данных.

```
bests_6_1 = run_speed_test(ns, analyze1)
print(plot_bests(ns, bests_6_1))
```

```
32
63.8 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
64
122 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
128
407 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
256
2.6 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
512
13.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
1024
67.8 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2048
331 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
4096
1.34 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
8192
8.04 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Рис. 6.1. Результаты `analyze1`

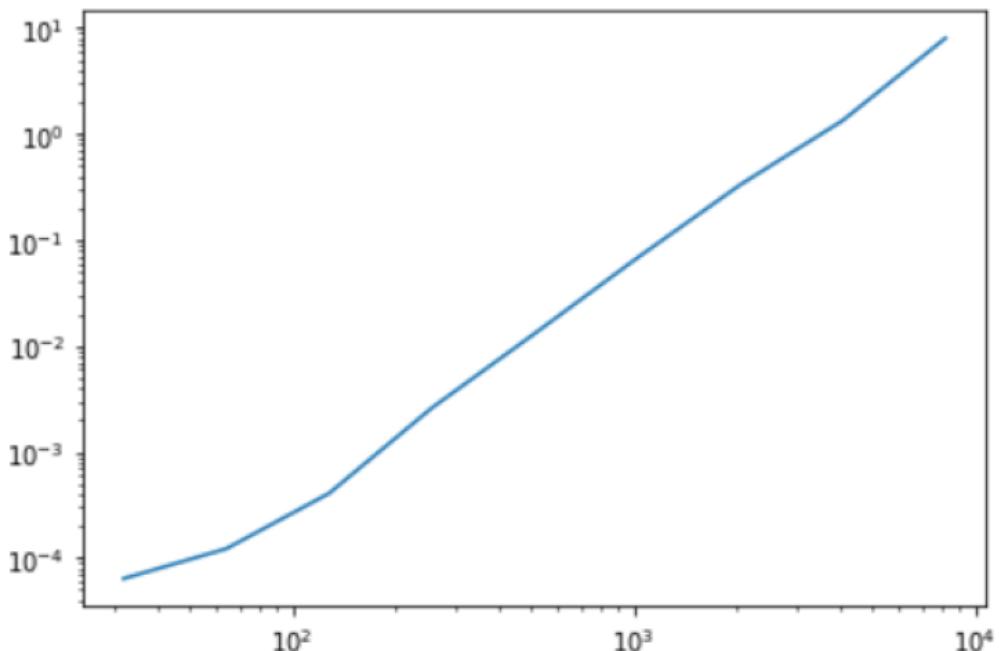


Рис. 6.2. График analyze1

Наклон полученного графика равен 2.2 (Рис.6.2), что не соответствует нашим ожиданиям. Возможно так произошло из-за недостаточной выборки в массиве.

Теперь посмотрим на функцию analyze2. Построим её график при тех же данных в массиве (Рис.6.3).

```
bests_6_1_2 = run_speed_test(ns, analyze2)
plot_bests(ns, bests_6_1_2)
```

```
32
23.7 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
64
52 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
128
218 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
256
801 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
512
5.66 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
1024
22.2 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2048
94.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
4096
349 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
8192
1.28 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Рис. 6.3. Результаты analyze2

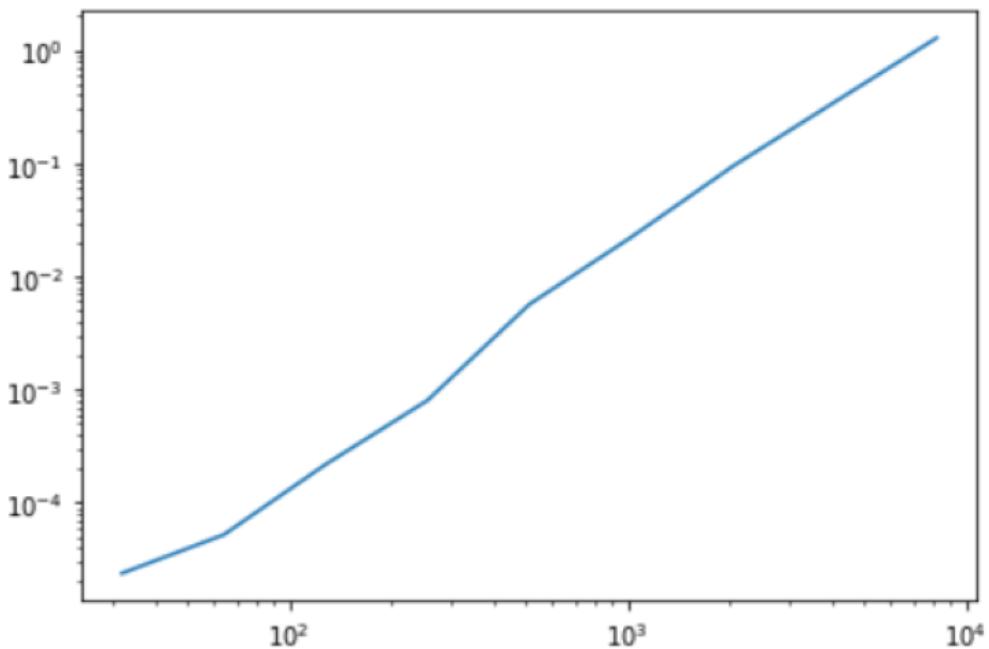


Рис. 6.4. График analyze2

Наклон данного графика равен 2.1 (Рис.6.4), что соответствует нашим ожиданиям.

Также посмотрим на функцию `scipy.fftpack.dct` (Рис.6.5).

```
bests_6_1_3 = run_speed_test(ns, scipy_dct)
plot_bests(ns, bests_6_1_3, "fftpack.dct")
```

```
32
7.1 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
64
7.18 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
128
7.98 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
256
8.54 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
512
9.89 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
1024
12.3 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
2048
18.4 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops each)
4096
33.2 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
8192
62.5 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
```

Рис. 6.5. Результаты `scipy.fftpack.dct`

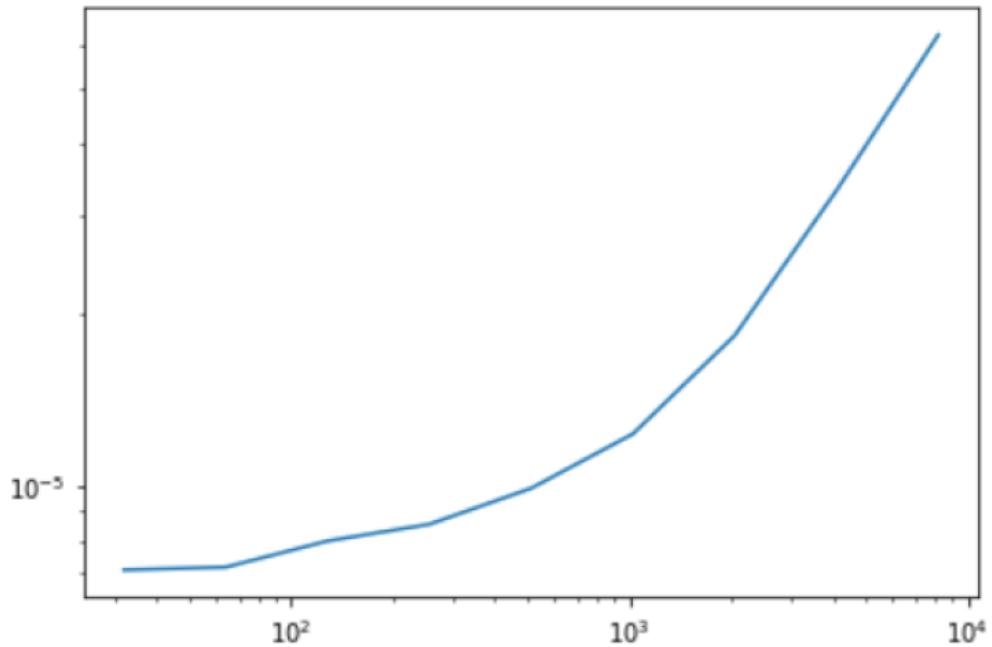


Рис. 6.6. График `scipy.fftpack.dct`

Данная функция работает еще быстрее. Как мы можем видеть на рис.6.6, её график изогнут, что означает, либо что мы еще не видели асимптотическое поведение, либо асимптотическое поведение не является простым показателем. Её выполнения пропорционально $n \log n$.

Сравним полученные результаты (Рис.6.7).

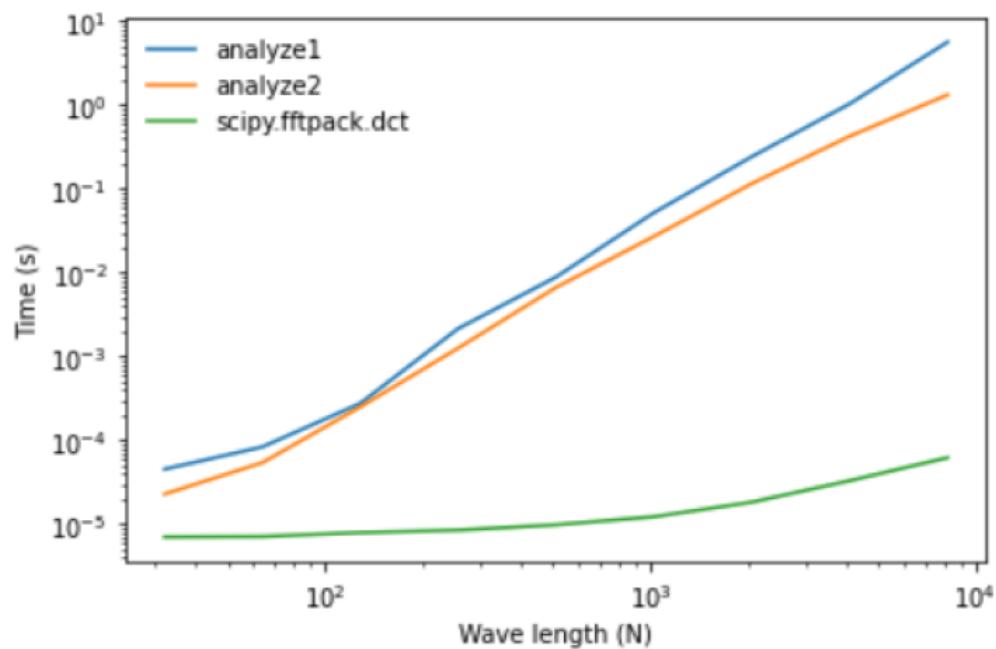


Рис. 6.7. Сравнение трёх функций

6.2. Упражнение 2

Реализуем алгоритм сжатия звука с помощью ДКП. Применим его для записи музыки.

Алгоритм должен иметь следующие шаги:

1. Разбиваем длинный сигнал на сегменты.
2. Вычисляем ДКП каждого сегмента.
3. Определяем частотные компоненты с такой амплитудой, что их не слышно, и удаляем их, сохраняя только оставшиеся частоты и амплитуды
4. При воспроизведении сигнала загружает частоты и амплитуды каждого сегмента и применяем обратное ДКП.

Используем звук саксофона из предыдущей лабораторной и выделим из него небольшой сегмент.

```
wave_6_2 = read_wave('100475_iluppa1_saxophone-weep.wav')
segment_6_2 = wave_6_2.segment(start=1.2, duration=0.5)
segment_6_2.normalize()
segment_6_2.make_audio()
```

Теперь вычислим ДКП этого сегмента.

```
seg_dct_6_2 = segment_6_2.make_dct()
seg_dct_6_2.plot(high=4000)
decorate(xlabel='Frequency (Hz)', ylabel='DCT')
```

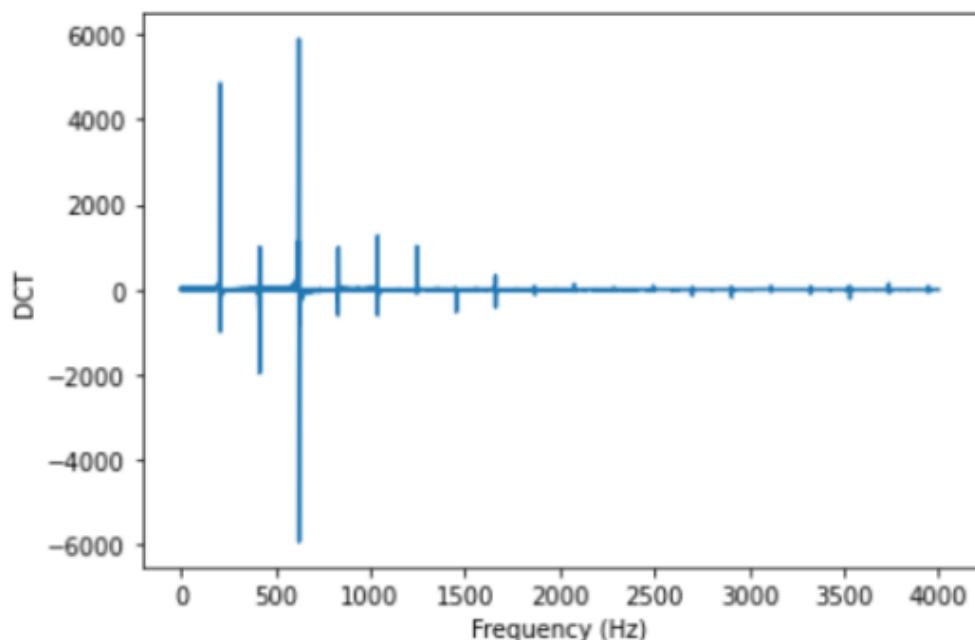


Рис. 6.8. ДКП сегмента

Как мы можем видеть на рис.6.8, только несколько частот имеют довольно большую амплитуду, остальные же близки к нулю.

Зануляем слишком маленькие частоты, используя функцию `compress`.

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

    n = len(dctamps)
    print(count, n, 100 * count / n, sep='\t')

seg_dct_6_2 = segment_6_2.make_dct()
compress(seg_dct_6_2, thresh=10)
seg_dct_6_2.plot(high=4000)
```

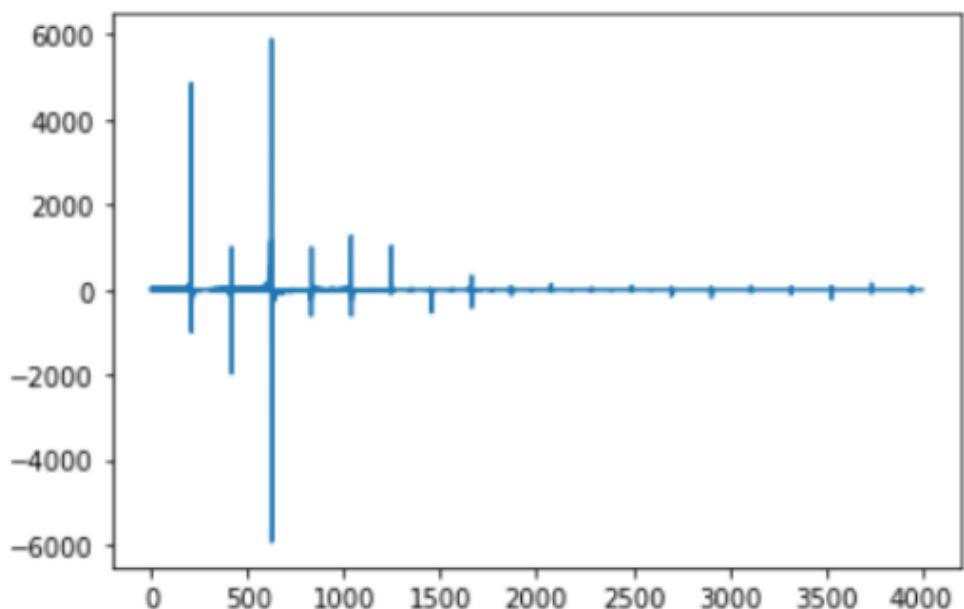


Рис. 6.9. ДКП фильтрованного сегмента

В результате применения `compress`, мы удалили около 93% гармоник (Рис.2.2). На слух звучание двух сегментов одинаковое.

Чтобы сжать более длинный сегмент, мы можем сделать спектрограмму ДКП. Используем функцию `make_dct_spectrogram`, которая похожа на `make_spectrogram`, но в ней используется ДКП.

```
def make_dct_spectrogram(wave, seg_length):
    window = np.hamming(seg_length)
    i, j = 0, seg_length
    step = seg_length // 2
```

```

spec_map = {}

while j < len(wave.ys):
    segment = wave.slice(i, j)
    segment.window(window)

    t = (segment.start + segment.end) / 2
    spec_map[t] = segment.make_dct()

    i += step
    j += step

return Spectrogram(spec_map, seg_length)

```

Теперь мы можем составить ДКП-спектрограмму и использовать сжатие для каждого сегмента.

```

spectrum_6_2 = make_dct_spectrogram(wave_6_2, seg_length=1024)
for t, dct in sorted(spectrum_6_2.spec_map.items()):
    compress(dct, thresh=0.2)

```

В результате применения `compress`, мы удалили от 70 до 99% гармоник у каждого сегмента.

Теперь послушаем и сравним звучание до и после сжатия. В этом случае разница более заметная. Полученный звук напоминает немного искаженный оригинал.

6.3. Упражнение 3

Воспользуемся блокнотом `phase.ipynb`, в котором исследуется влияние фазы на восприятие звука. Запустим примеры из него. Затем выберем другой сегмент звука и вновь поработаем с примерами.

В начале мы возьмем пилообразный сигнал с частотой 500 и построим его спектр, а затем построим его угловую часть (Рис.6.10).

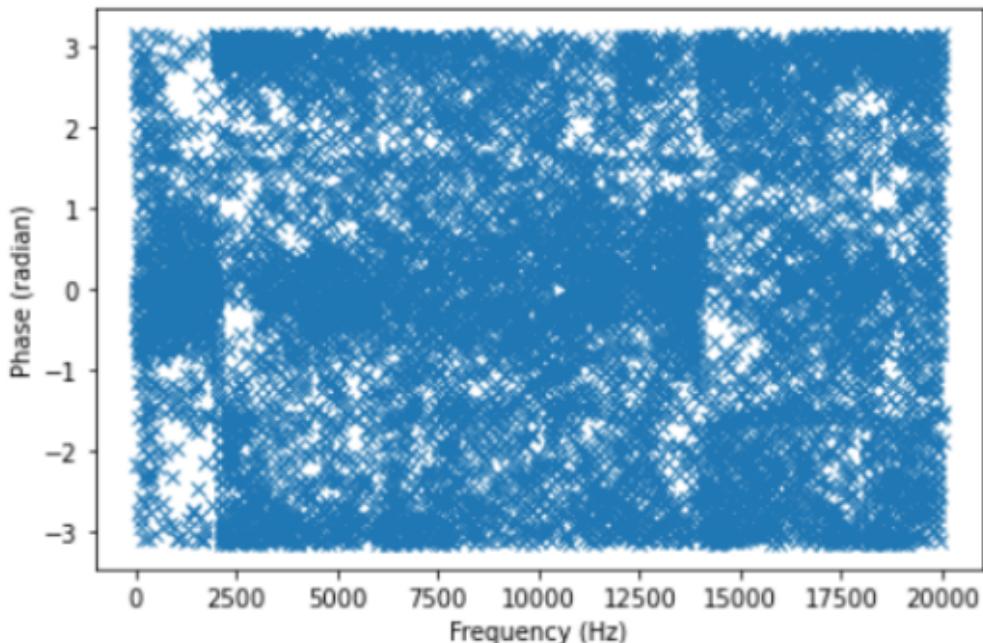


Рис. 6.10. Пилообразный сигнал. Угловая часть спектра

При построении всех углов, мы получаем довольно запутанную картинку (Рис.6.10). Но если мы выберем только те частоты, где величина превышает порог, мы увидим, что в углах есть структура. Каждая гармоника смешена от предыдущей на доли радиана (Рис.6.11).

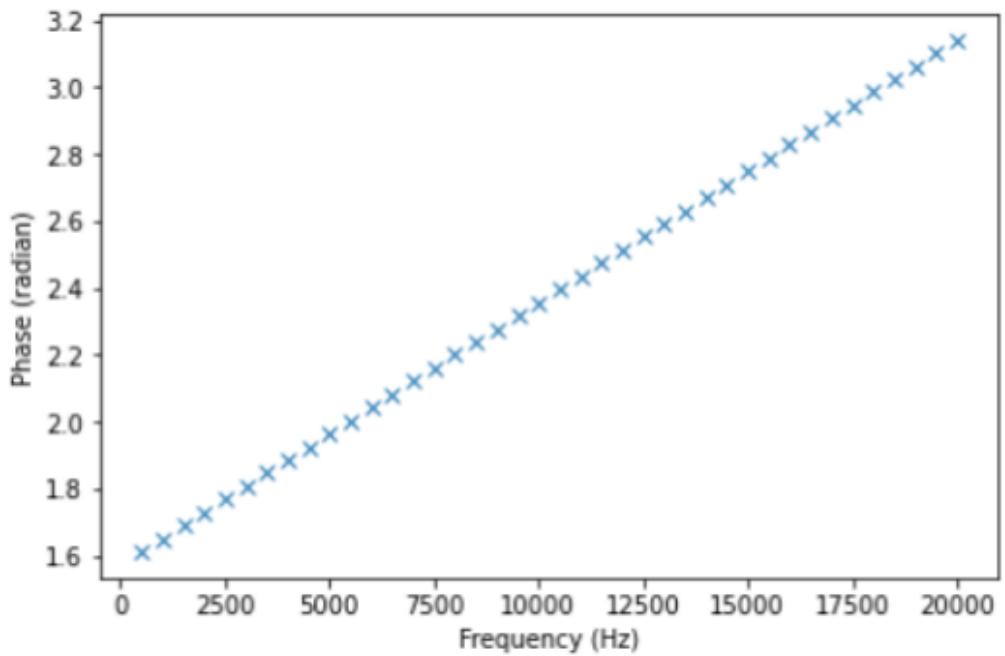


Рис. 6.11. Пилообразный сигнал. Угловая часть спектра с порогом

Теперь посмотрим, что произойдет, если мы установим все углы в ноль (Рис.6.12).

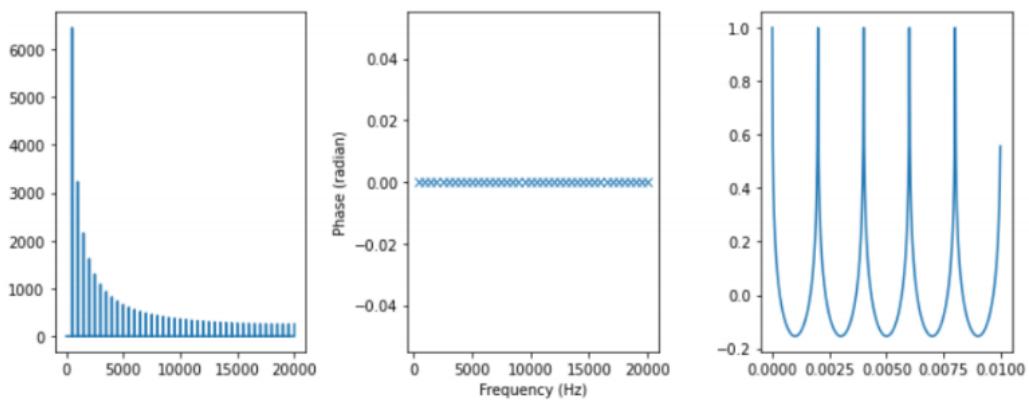


Рис. 6.12. Пилообразный сигнал. Нулевые углы

Амплитуда волны стала ниже, но это из-за способа нормализации волны, а не из-за изменений в фазовой структуре. Теперь выполним поворот угла (Рис.6.13).

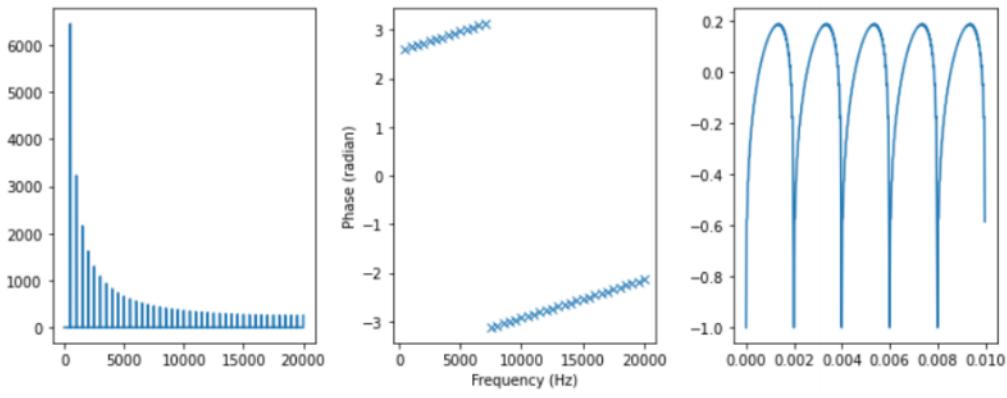


Рис. 6.13. Пилообразный сигнал. Поворот угла

Также посмотрим, что произойдет, если мы установим для углов случайные значения (Рис.6.14).

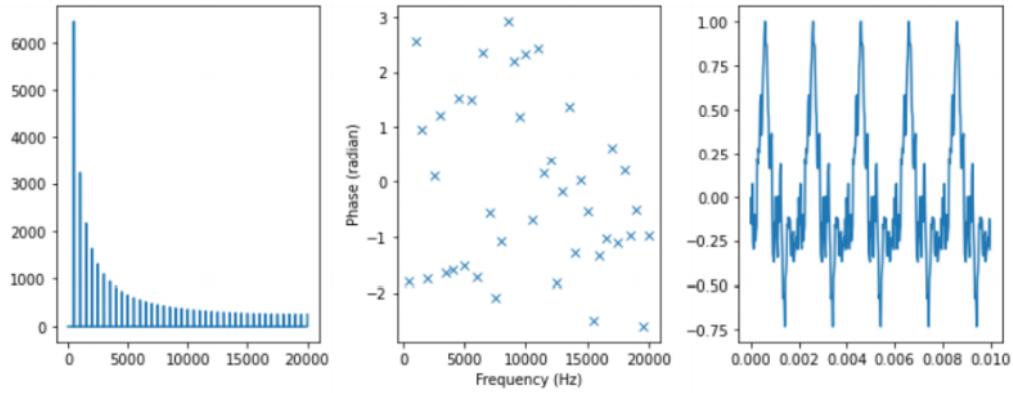


Рис. 6.14. Пилообразный сигнал. Случайные углы

Теперь поработаем с другими звуками. Воспользуемся записью гобоя. Выделим из неё новый сегмент. Снова построим спектр и его угловую часть (Рис.6.15).

```
return Spectrogram(spec_map, seg_length)
```

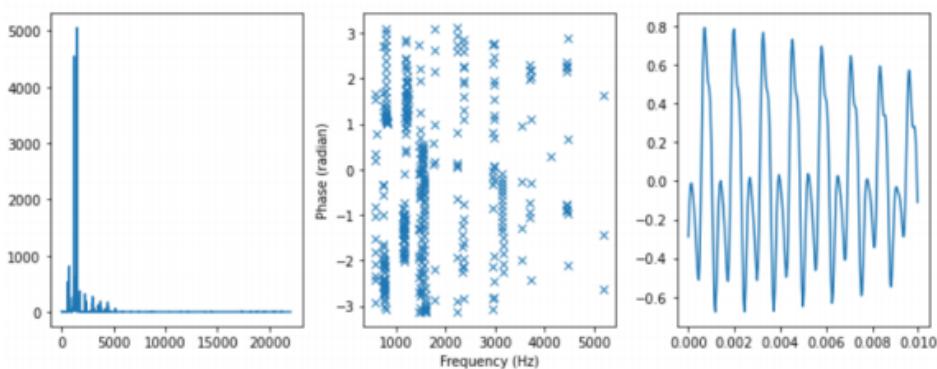


Рис. 6.15. Гобой. Угловая часть спектра

Теперь установим все углы в ноль (Рис.6.16).

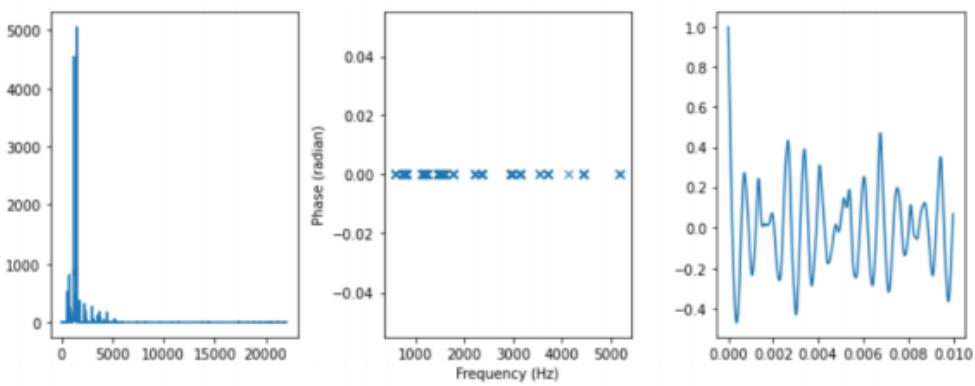


Рис. 6.16. Гобой. Нулевые углы

Изменение фазовой структуры, создает эффект "звона", когда громкость меняется со временем.

Затем повернём угол на один радиан (Рис.6.17).

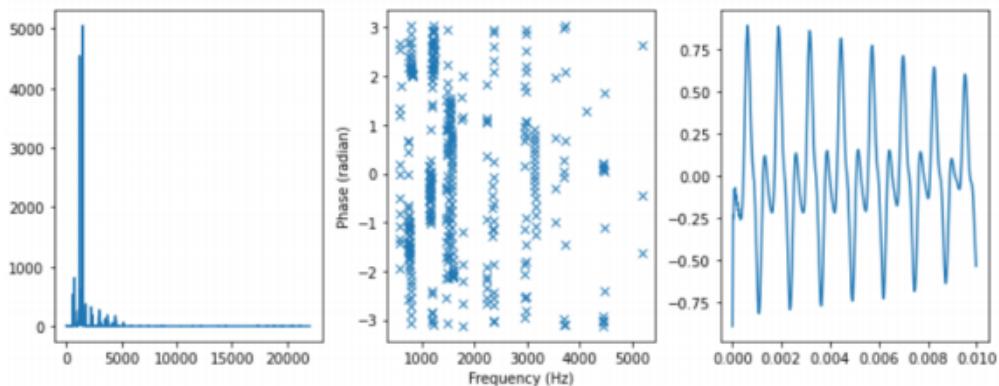


Рис. 6.17. Гобой. Поворот угла

Вращение углов не вызывает "звона".

Далее установим случайные значения (Рис.6.18).

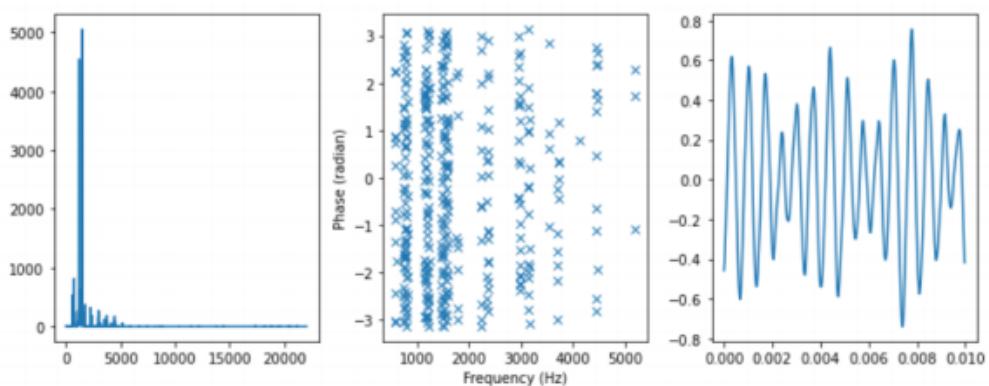


Рис. 6.18. Гобой. Случайные углы

Случайный выбор углов вызывает некоторый звон и добавляет хриплости звуку.

Проведя аналогичные исследования на саксофоне, мы получаем такие же результаты.

Обнуление вызывает звон, вращение мало влияет, а использование случайных значений добавляет хрипы.

Отличительной чертой саксофона от других звуков является то, что основная частота не является доминирующей. Попробуем её отфильтровать и посмотреть на результат. Обнуление (Рис.6.19).

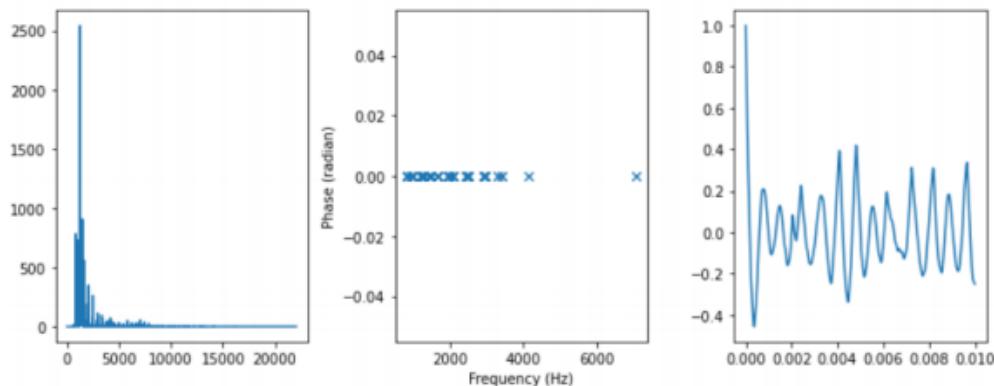


Рис. 6.19. Саксофон. Нулевые углы

Поворот (Рис.6.20).

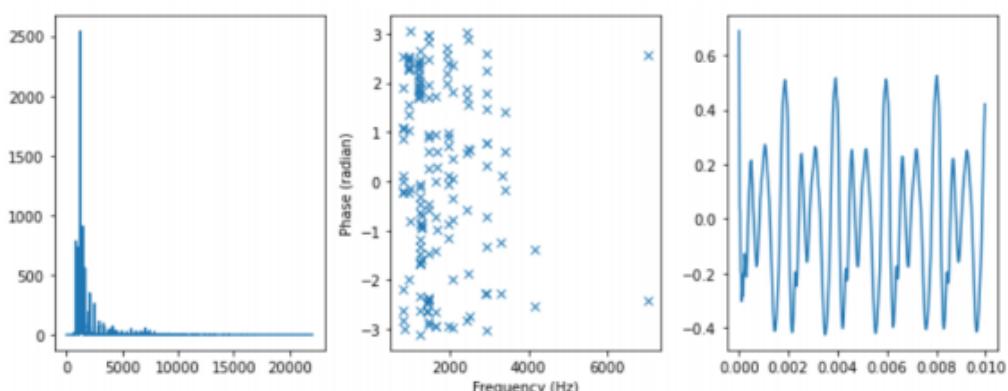


Рис. 6.20. Саксофон. Поворот угла

Случайные значения (Рис.6.21).

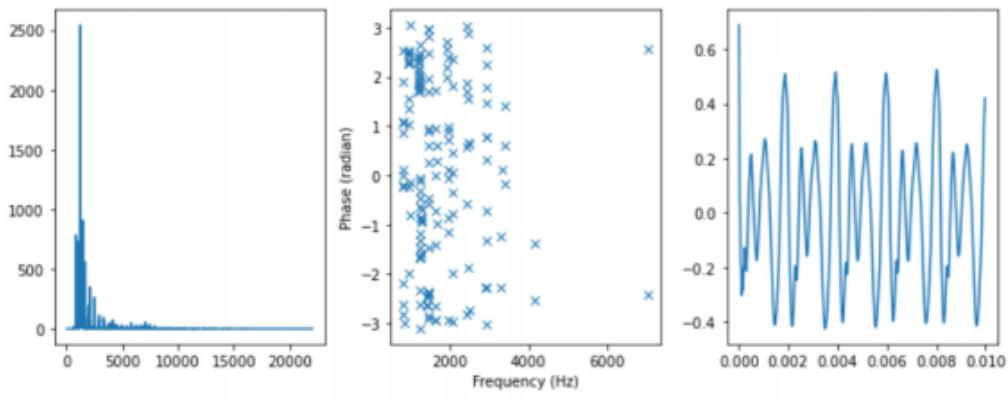


Рис. 6.21. Саксофон. Случайные углы

Таким образом, мы не слышим изменений в фазовой структуре звука, если он простой, имеет простую гармоническую структуру и если гармоническая структура не изменилась. Возможным исключением являются звуки с низкой амплитудой у основной частоты.

6.4. Выводы

В результате выполнения данной работы мы изучили дискретное косинусное преобразование и научились работать с ним. Также при помощи анализа разных звуков мы исследовали влияние фазы на восприятие звука.

7. Лабораторная работа 7

7.1. Упражнение 1

The notebook for this chapter, `chap07.ipynb`, contains additional examples and explanations.
Read through it and run the code.

Файл `chap07.ipynb` был прочтён, все примеры успешно запущены.

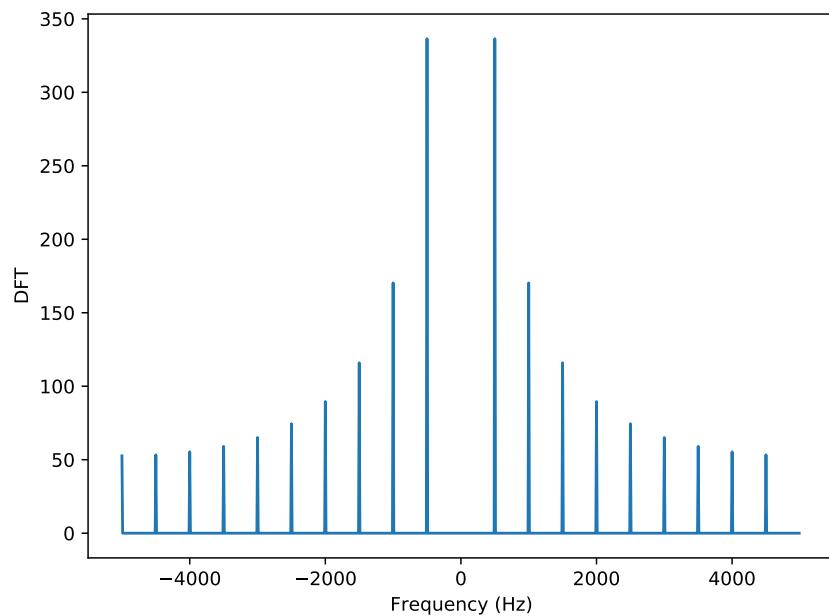


Рис. 7.1. График "полного" БПФ для треугольного сигнала из `chap07.ipynb`

7.2. Упражнение 2

In this chapter, I showed how we can express the DFT and inverse DFT as matrix multiplications. These operations take time proportional to N^2 , where N is the length of the wave array. That is fast enough for many applications, but there is a faster algorithm, the Fast Fourier Transform (FFT), which takes time proportional to $N * \log(N)$.

The key to the FFT is the Danielson-Lanczos lemma:

$$DFT(y)[n] = DFT(e)[n] + \exp(-2 * \pi * i * n / N) DFT(o)[n]$$

Where $DFT(y)[n]$ is the n th element of the DFT of y ; e is the even elements of y , and o is the odd elements of y .

This lemma suggests a recursive algorithm for the DFT:

1. Given a wave array, y , split it into its even elements, e , and its odd elements, o .
2. Compute the DFT of e and o by making recursive calls.
3. Compute $DFT(y)$ for each value of n using the Danielson-Lanczos lemma.

For the base case of this recursion, you could wait until the length of y is 1. In that case, $DFT(y) = y$. Or if the length of y is sufficiently small, you could compute its DFT by matrix multiplication, possibly using a precomputed matrix.

Реализуем алгоритм быстрого преобразования Фурье (БПФ), имеющий сложность $N \log N$. В этом нам поможет лемма Дэниелсона-Ланцоша:

$$DFT(y)[n] = DFT(e)[n] + \exp(-2\pi in/N) DFT(o)[n]$$

где $DFT(y)[n]$ - это n -й элемент ДПФ по y , e - чётные элементы y и o - нечётные.

Лемма предлагает рекурсивный алгоритм ДПФ:

1. Дан массив сигнала y . Разделим его на чётные элементы e и нечётные элементы o .
2. Вычислим $DFT e$ и o , делая рекурсивные вызовы.

3. Вычислим $DFT(y)$ для каждого значения n , используя лемму Дэниелсона-Ланцоша.

В простейшем случае эту рекурсию надо продолжать, пока длина не дойдет до единицы. Тогда $DFT(y) = y$. А если длина достаточно мала, можно вычислить его ДПФ перемножением матриц, используя заранее вычисленные матрицы.

Для начала вычислим БПФ для небольшого сигнала с помощью библиотечной функции `np.fft.fft`:

```
ys_2 = [-0.2, 0.9, 0.5, -0.3]
hs_2 = np.fft.fft(ys_2)

# [ 0.9+0.j -0.7-1.2j -0.3+0.j -0.7+1.2j]
print(hs_2)
```

Теперь реализуем алгоритм, следуя лемме.

```
def effective_fft(ys):
    # noinspection PyPep8Naming
    N = len(ys)
    if N == 1:
        return ys

    fft_even = np.tile(effective_fft(ys[::2]), 2)
    fft_odd = np.tile(effective_fft(ys[1::2]), 2)

    ns = np.arange(N)
    exp = np.exp(-1j * 2 * np.pi * ns / N)

    return fft_even + exp * fft_odd
```

Проверим работу функции.

```
hs2_2 = effective_fft(ys_2)
print(np.sum(np.abs(hs_2 - hs2_2)))
# 4.065457153363882e-16
```

Результат совпадает в пределах погрешности, следовательно, алгоритм был реализован верно.

7.3. Выводы

В результате выполнения данной работы мы изучили дискретное преобразование Фурье и быстрое преобразование Фурье. Во многих случаях достаточно ДПФ, которое работает за N^2 , но иногда требуется БПФ, которое работает за $N \log N$.

8. Лабораторная работа 8

8.1. Упражнение 1

The notebook for this chapter is `chap08.ipynb`. Read through it and run the code.
It contains an interactive widget that lets you experiment with the → parameters of the Gaussian window to see what effect they have on the cutoff frequency.
What goes wrong when you increase the width of the Gaussian, `std`, without increasing the number of elements in the window, `M`?

Используем интерактивный виджет, для того чтобы поэкспериментировать с параметрами Гауссова окна и проанализировать их влияние на частоту среза. В коде `lab8.py` будет использована функция `plot_filter` для построения графиков.

```
def plot_filter(M=20, std=2, name="normal"):  
    signal = SquareSignal(freq=440)  
    wave = signal.make_wave(duration=1, framerate=44100)  
    spectrum = wave.make_spectrum()  
  
    gaussian = scipy.signal.gaussian(M=M, std=std)  
    gaussian /= sum(gaussian)  
  
    ys = np.convolve(wave.ys, gaussian, mode='same')  
    smooth = Wave(ys, framerate=wave.framerate)  
    spectrum2 = smooth.make_spectrum()  
  
    amps = spectrumamps  
    amps2 = spectrum2amps  
    ratio = amps2 / amps  
    ratio[amps < 560] = 0  
  
    padded = zero_pad(gaussian, len(wave))  
    dft_gaussian = np.fft.rfft(padded)  
  
    plt.plot(np.abs(dft_gaussian), color='gray', label='Gaussian filter')  
    plt.plot(ratio, label='amplitude ratio')  
  
    decorate(xlabel='Frequency (Hz)', ylabel='Amplitude ratio')
```

Для начала возьмем значения $M = 20$ и $std = 2$.

```
plot_filter(M=20, std=2)
```

Взглянем на получившийся график.

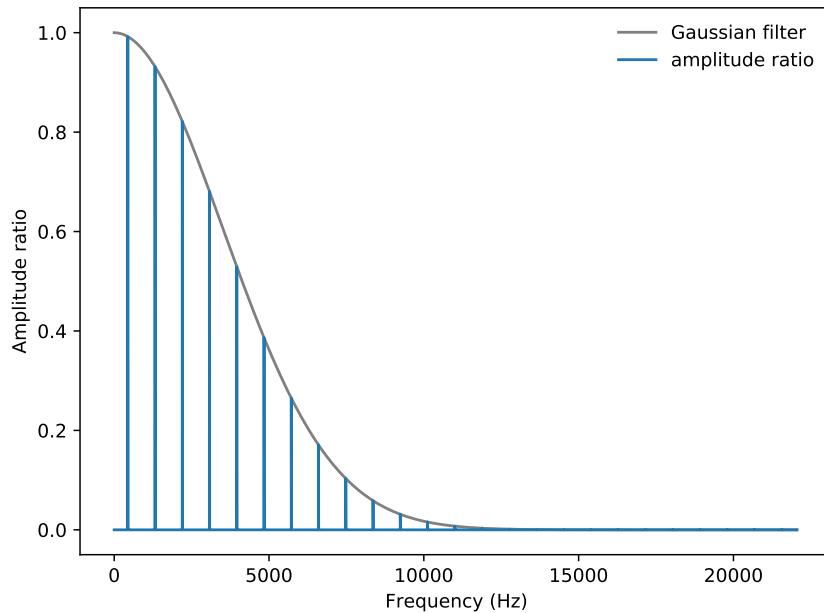


Рис. 8.1. График гауссова окна при $std = 2$

Теперь начнем увеличивать параметр std , сначала до значения 5.

```
plot_filter(M=20, std=5, name="partly.changed")
```

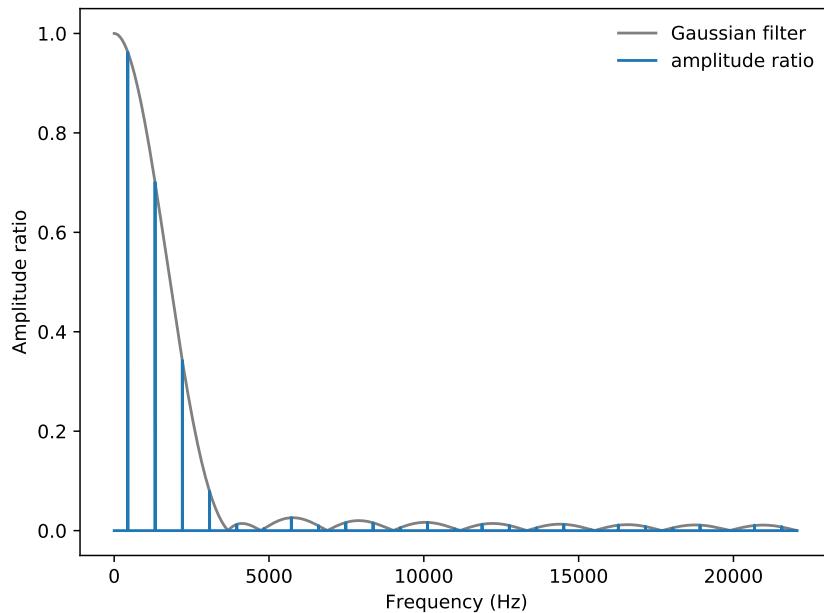


Рис. 8.2. График гауссова окна при $std = 5$

Затем до значения 20.

```
plot_filter(M=20, std=20, name="fully.changed")
```

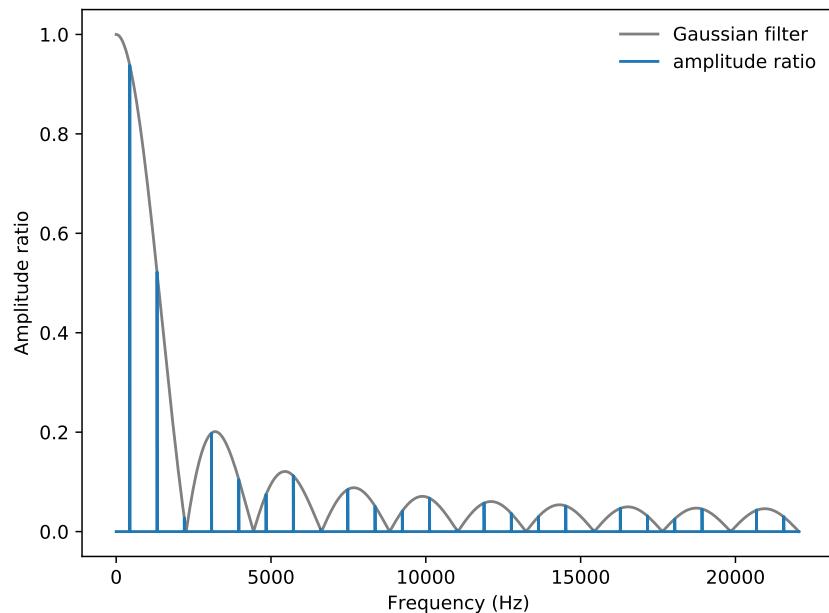


Рис. 8.3. График гауссова окна при $std = 20$

Как можно видеть, без увеличения M окно постепенно сжимается, высокочастотные гармоники спадают медленнее, из-за чего появляются боковые лепестки.

8.2. Упражнение 2

In this chapter I claimed that the Fourier transform of a Gaussian curve is also a Gaussian curve. For discrete Fourier transforms, this relationship is approximately true.

Try it out for a few examples. What happens to the Fourier transform as you vary std?

Напишем функцию, которая будет строить графики для гауссова окна и его БПФ.

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

Выберем изначальные параметры для окна $M = 32$ и $std = 2$.

```
gaussian(32, 2, name="2.normal.plot")
```

Взглянем на получившийся график.

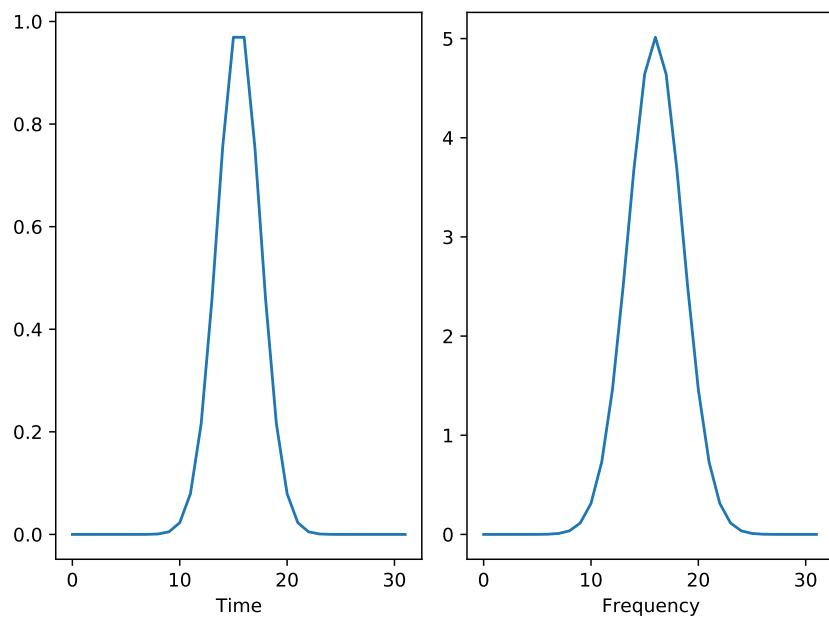


Рис. 8.4. График для гауссова окна и его БПФ при $std = 2$

Теперь уменьшим значение std до 0.1.

```
gaussian(32, 0.1, name="2.compressed.plot")
```

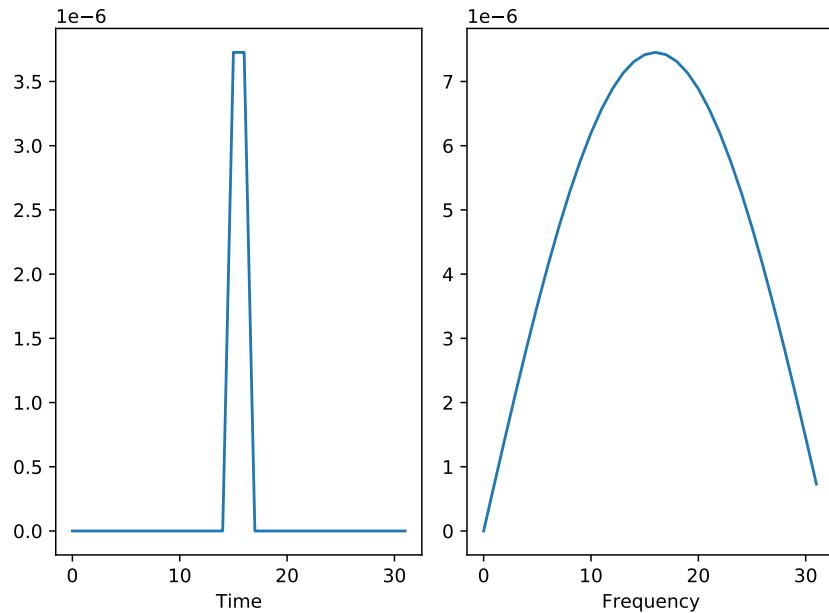


Рис. 8.5. График для гауссова окна и его БПФ при $std = 0.1$

При уменьшении std график для гауссова окна сжимается, а график для БПФ наоборот, расширяется.

Теперь увеличим значение std до 20.

```
gaussian(32, 10, name="2.bloated.plot")
```

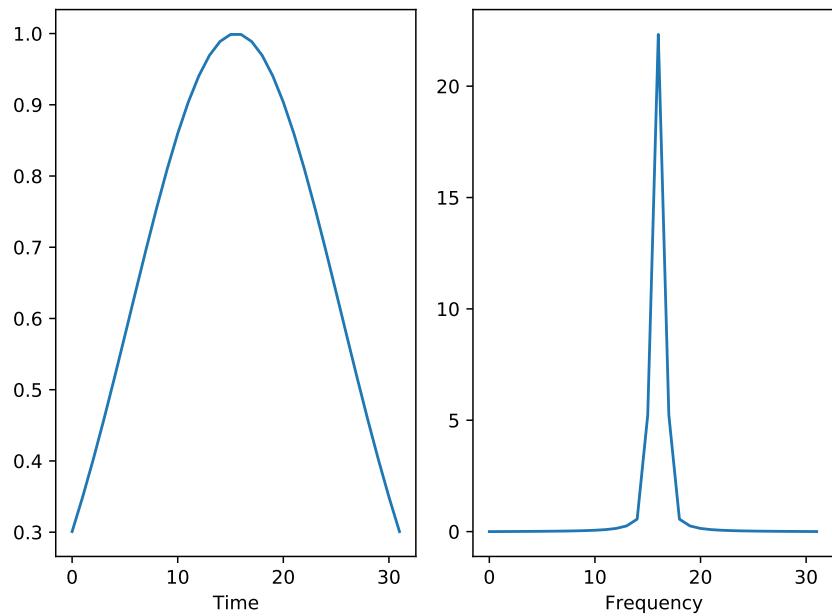


Рис. 8.6. График для гауссова окна и его БПФ при $std = 20$

При увеличении std график для гауссова окна расширяется, а график для БПФ наоборот, сжимается. Это позволяет сказать об обратной зависимости между ними.

8.3. Упражнение 3

If you did the exercises in Chapter 3, you saw the effect of the Hamming window, and some of the other windows provided by NumPy, on spectral leakage. We can get some insight into the effect of these windows by looking at their DFTs.

In addition to the Gaussian window we used in this window, create a Hamming window with the same size. Zero pad the windows and plot their DFTs. Which window acts as a better low-pass filter? You might find it useful to plot the DFTs on a log-y scale.

Experiment with a few different windows and a few different sizes.

В одной из предыдущих работ мы изучали влияние на утечки спектра окна Хэмминга и других окон, для того чтобы лучше их изучить, рассмотрим их ДПФ.

Первым делом построим все изучаемые окна, а также прямоугольный сигнал для дальнейшего использования.

```
M_3 = 30
std_3 = 2.5
square_signal_3 = SquareSignal(freq=440)
wave_3 = square_signal_3.make_wave(duration=1, framerate=40000)
blackman_3 = np.blackman(M_3)
bartlett_3 = np.bartlett(M_3)
hamming_3 = np.hamming(M_3)
hanning_3 = np.hanning(M_3)
gaussian_3 = scipy.signal.gaussian(M=M_3, std=std_3)

windows_3 = [blackman_3, bartlett_3, gaussian_3, hanning_3, hamming_3]
names_3 = ['blackman', 'bartlett', 'gaussian', 'hanning', 'hamming']
```

Затем построим сравнительный график для изучаемых окон.

```
for window_3 in windows_3:
    window_3 /= sum(window_3)

for window_3, name_3 in zip(windows_3, names_3):
    plt.plot(window_3, label=name_3)
decorate(xlabel='Index')
```

И взглянем на получившийся график.

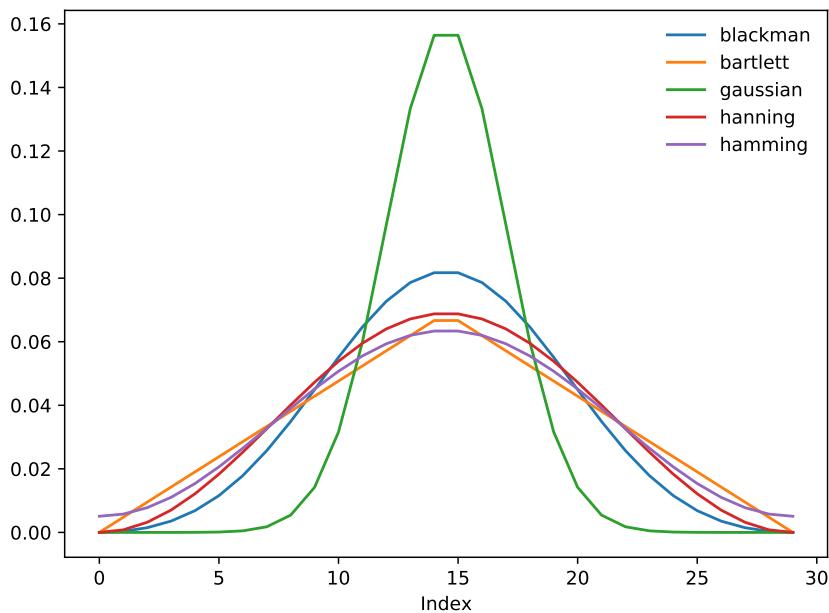


Рис. 8.7. Сравнительный график для различных окон

Их графики довольно похожи друг на друга. Теперь напишем функцию для построения сравнительного графика ДПФ изучаемых окон, в том числе и в логарифмическом масштабе.

```
def zero_pad(array, n):
    res = np.zeros(n)
    res[:len(array)] = array
    return res

def window_dfts(windows, names, file_name, yscale):
    for window, name in zip(windows, names):
        padded = zero_pad(window, len(wave_3))
        dft_window = np.fft.rfft(padded)
        plt.plot(abs(dft_window), label=name)
    if yscale != '':
        decorate(xlabel='Frequency (Hz)', yscale=yscale)
    else:
        decorate(xlabel='Frequency (Hz)')
```

Теперь вызовем ее для построения ДПФ.

```
window_dfts(windows_3, names_3, file_name="3.windows.frequency.comparison",
            yscale='')
```

Взглянем на получившийся график.

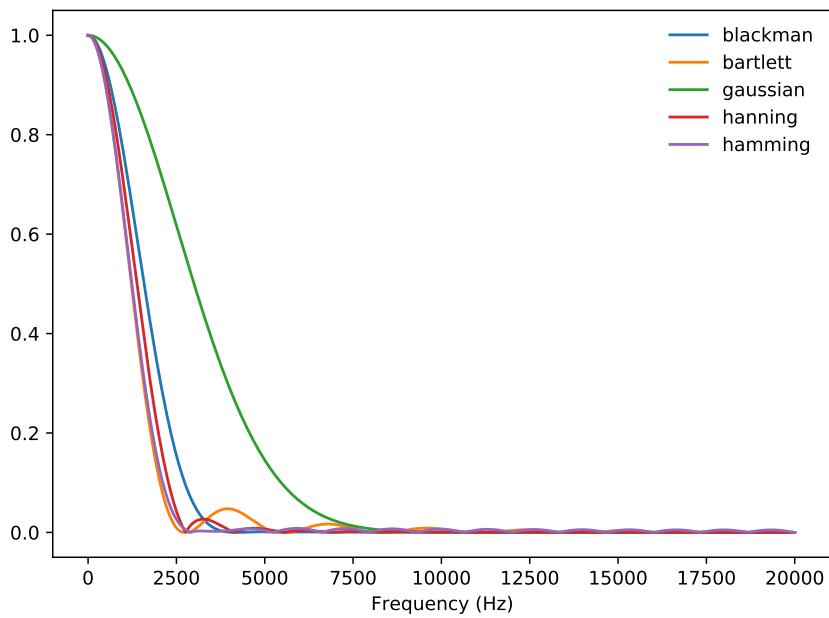


Рис. 8.8. Сравнительный график ДПФ для различных окон

И проделаем тоже самое для графика в логарифмическом масштабе.

```
window_dfts(windows_3, names_3,
↪ file_name="3.windows.frequency.log.comparison", yscale='log')
```

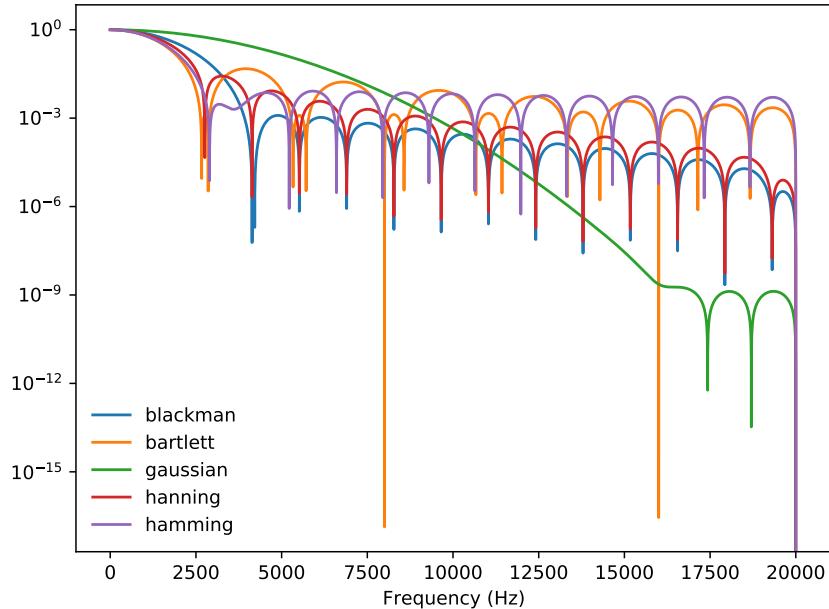


Рис. 8.9. Сравнительный график ДФП в логарифмическом масштабе

В результате анализа всех графиков можно прийти к выводу, что окно Хемминга дает меньше всего "выпуклостей" и, как видно из логарифмического графика, имеет самые стойкие боковые лепестки, окно Хеннинга в свою очередь имеет наилучшее сочетание быстрого падения и минимальных боковых лепестков.

8.4. Выводы

В результате выполнения данной работы были изучены понятия фильтрации и свертки, а также различные окна через их ДПФ, получены навыки работы с Гауссовым окном.

9. Лабораторная работа 9

9.1. Упражнение 1

В начале мы запустим примеры из `chap09.ipynb`.

В пособии сказано, что некоторые примеры не работают с апериодическими сигналами. Заменим периодический пилообразный сигнал на непериодические данные Facebook и посмотрим, что случится.

Сначала создадим сигнал (Рис.9.1).

```
import pandas as pd
from thinkdsp import Wave
df = pd.read_csv('FB_2.csv', header=0, parse_dates=[0])
ys = df['Close']
in_wave = Wave(ys, framerate=1)
in_wave.plot()
decorate(xlabel='Time (days)', ylabel='Price ($)')
```

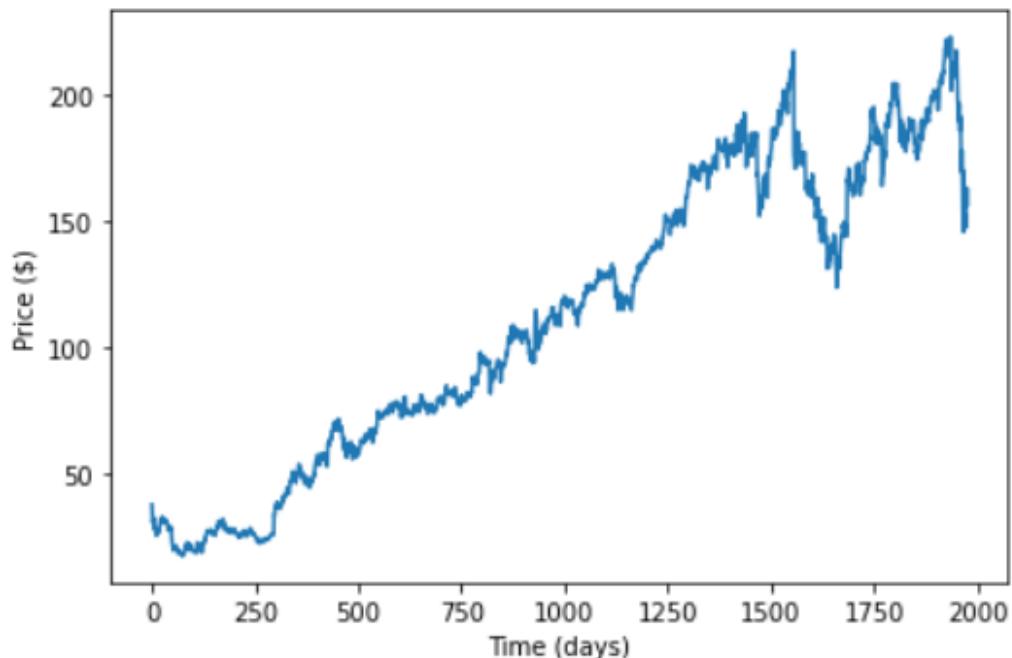


Рис. 9.1. Сигнал Facebook

Построим его спектр (Рис.9.2).

```
in_spectrum = in_wave.make_spectrum()
in_spectrum.plot()
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

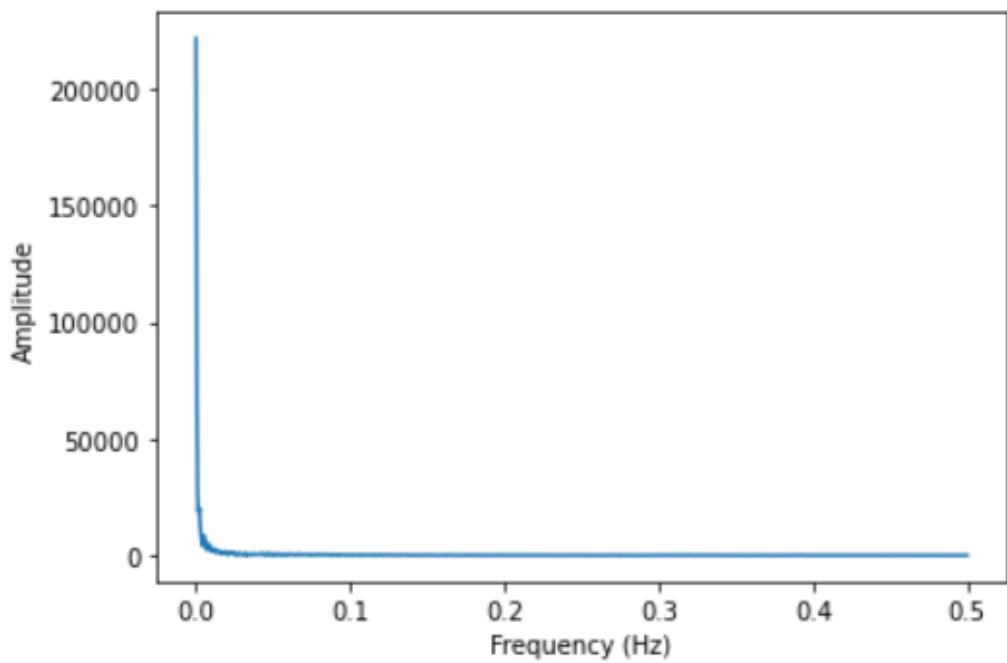


Рис. 9.2. Спектр Facebook

Теперь получим выходной сигнал, который является совокупной суммой входных сигналов (Рис.9.3), и его спектр (Рис.9.4).

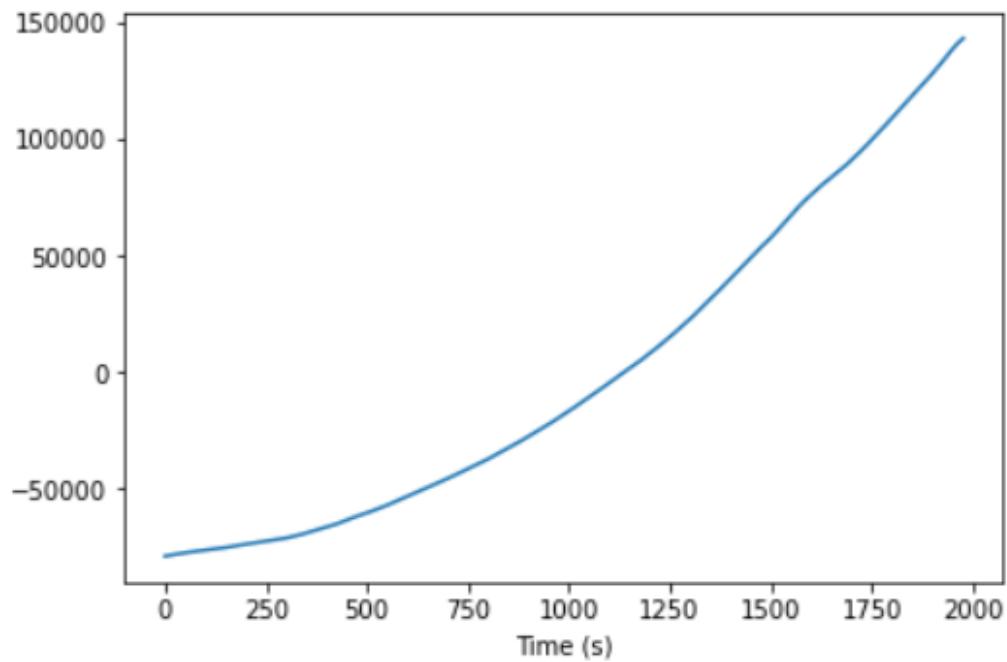


Рис. 9.3. Выходной сигнал

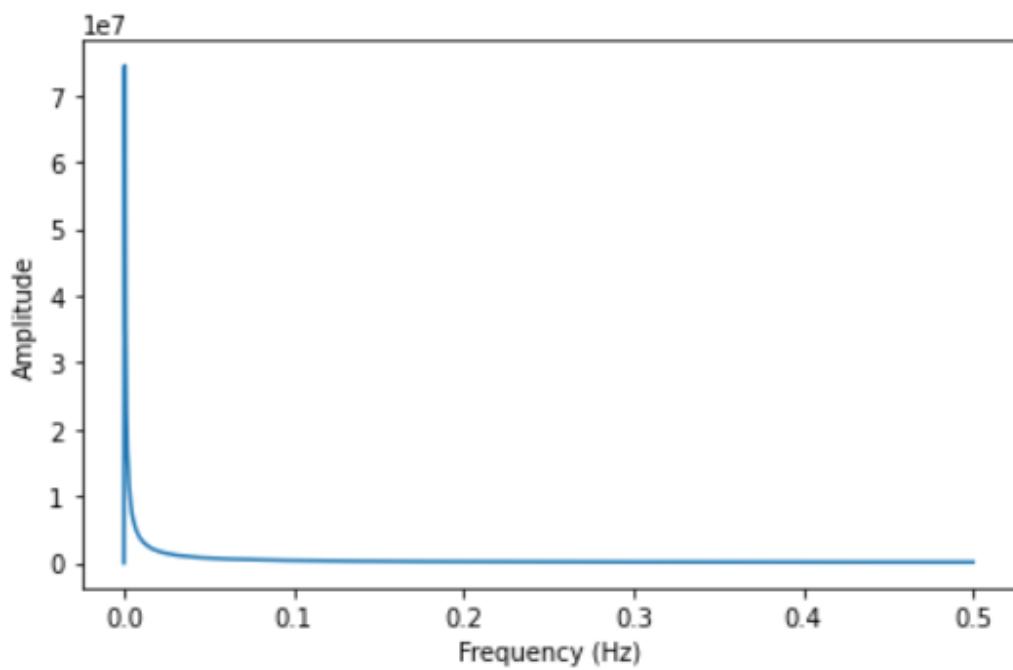


Рис. 9.4. Спектр выходного сигнала

Теперь посмотрим на отношение между входными и выходными данными (Рис.9.5).

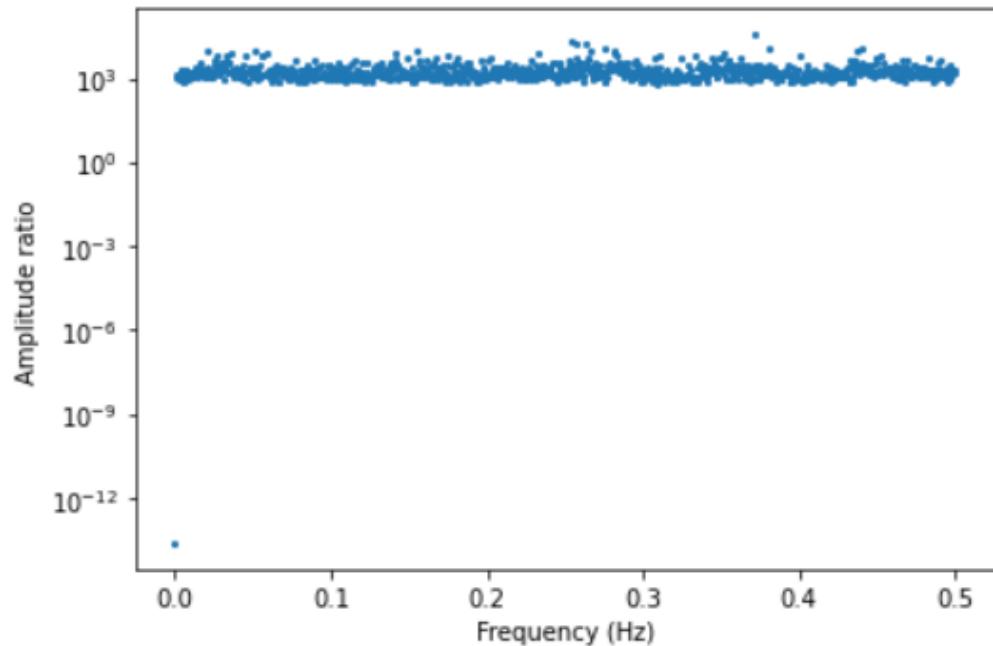


Рис. 9.5. Отношение входных и выходных данных

Построим фильтр для нарастающей суммы и сравним его с фильтром интегрирования.

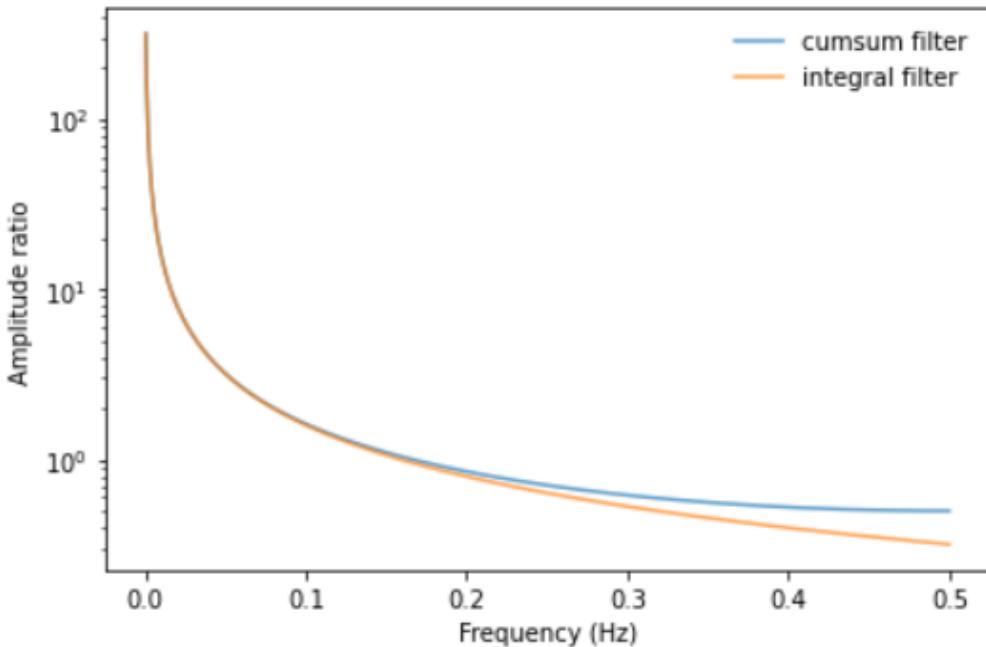


Рис. 9.6. Фильтры нарастающей суммы и интегрирования

Как мы видим на рис.9.6, графики сначала полностью совпадают, а под конец немного расходятся.

Затем мы можем сравнить вычисленное отношение с фильтром.

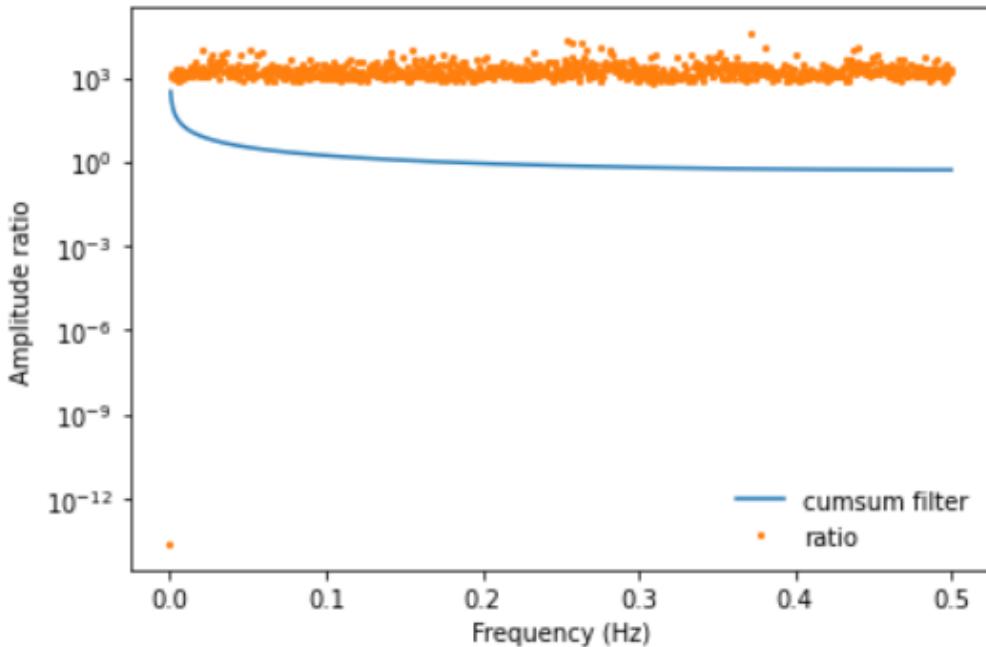


Рис. 9.7. Сравнение отношения и фильтра

Здесь случается первое расхождение (Рис.9.7). Данные графики должны совпадать. Это означало бы, что фильтр cumsum является обратным фильтру diff. Но этого не происходит.

Теперь применим фильтр cumsum в частотной области.

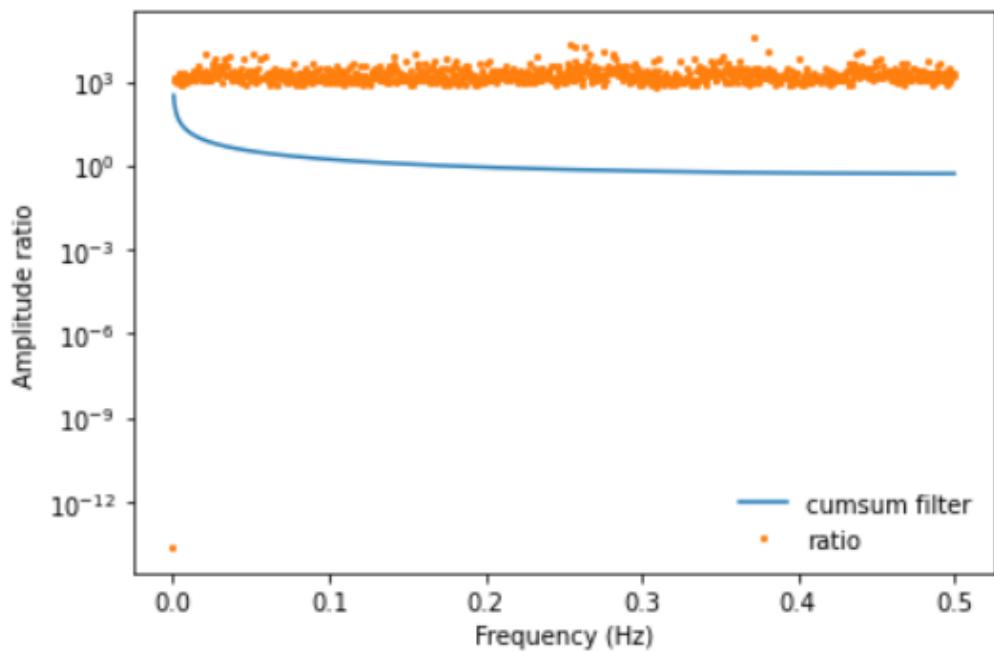


Рис. 9.8. Сравнение суммирования и фильтрации

На рис.9.8 вновь видим пример неправильной работы. Графики не совпадают.

9.2. Упражнение 2

В этом упражнении изучается влияние `diff` и `differentiate` на сигнал.

Создадим треугольный сигнал и напечатаем его (Рис.9.9).

```
from thinkdsp import TriangleSignal
triangle = TriangleSignal(freq=50).make_wave(duration=0.1, framerate=44100)
triangle.plot()
decorate(xlabel='Time (s)')
```

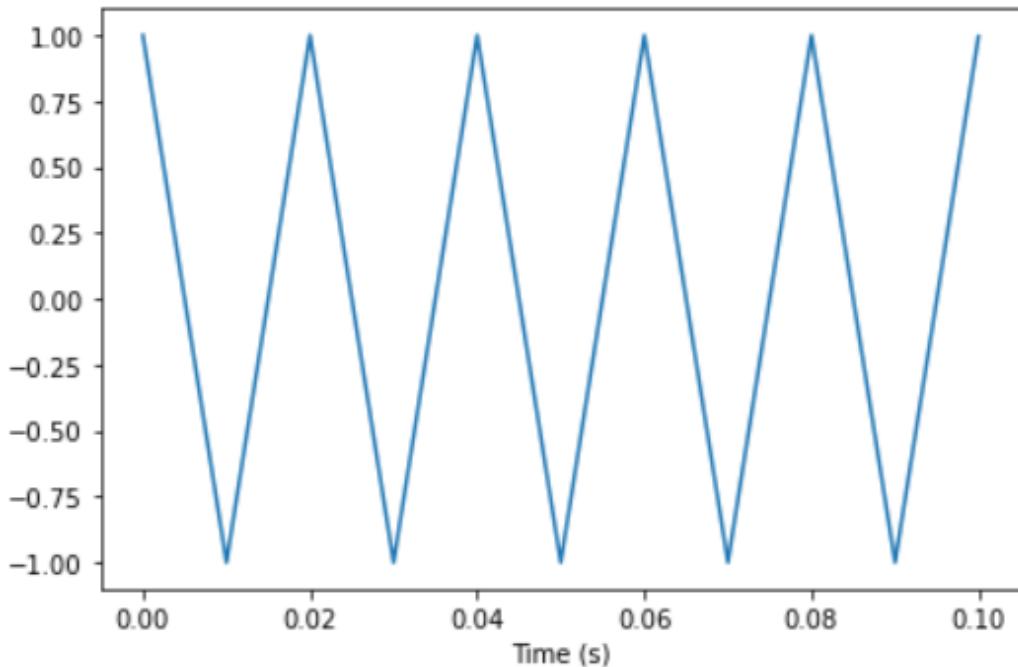


Рис. 9.9. Треугольный сигнал

Применим к нему `diff` и напечатаем результат (Рис.9.10).

```
out_wave = triangle.diff()
out_wave.plot()
decorate(xlabel='Time (s)')
```

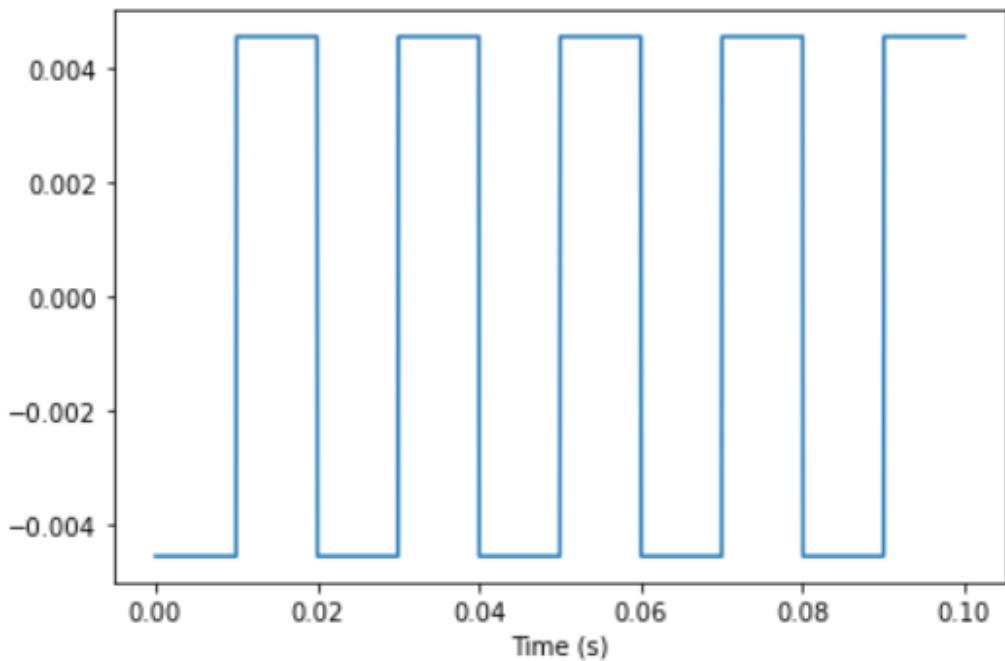


Рис. 9.10. Результат diff

Вычислим спектр треугольного сигнала, применим к нему differentiate и напечатаем (Рис.9.11).

```
out_wave2 = triangle.make_spectrum().differentiate().make_wave()
out_wave2.plot()
decorate(xlabel='Frequency (Hz)')
```

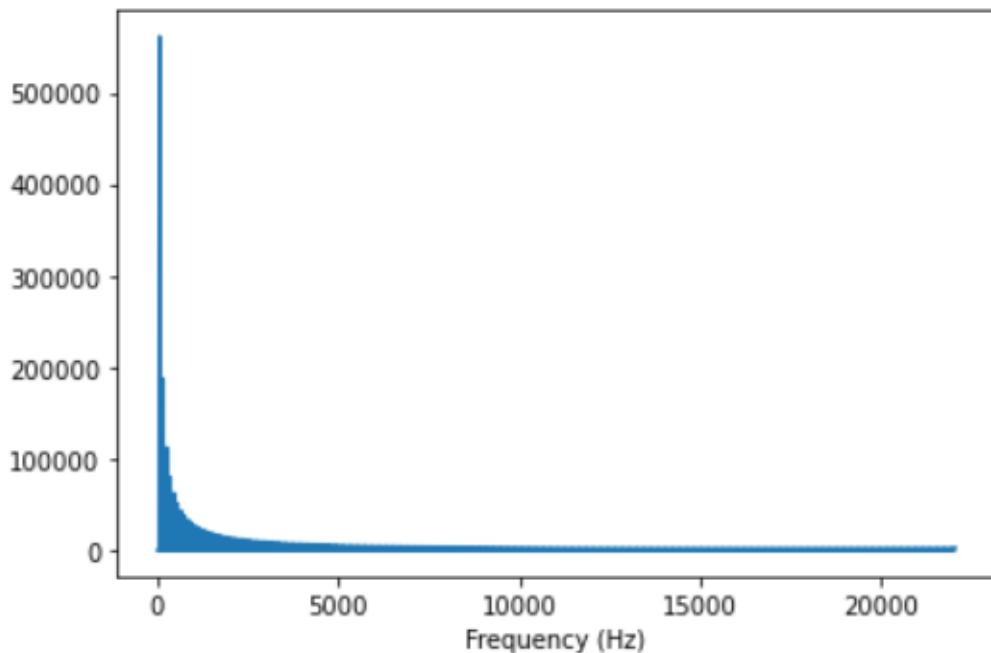


Рис. 9.11. Результат differentiate

Теперь преобразуем спектр обратно в сигнал и напечатаем его (Рис.9.12).

```
out_wave2.make_wave().plot()  
decorate(xlabel='Time (s)')
```

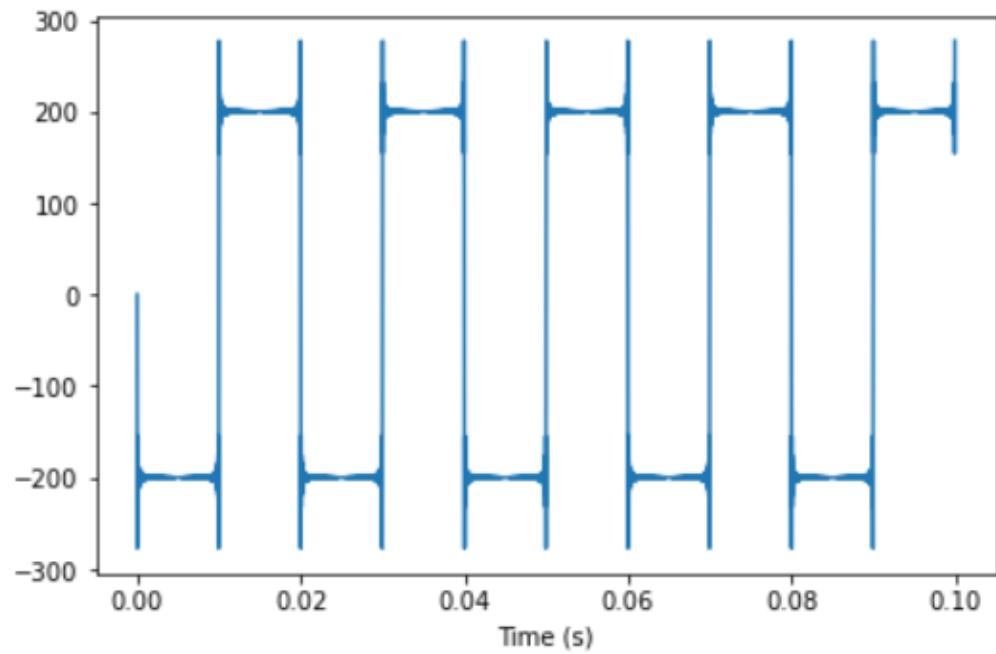


Рис. 9.12. Спектр в сигнал

Когда мы берём спектральную производную, мы получаем "звук" вокруг разрывов.

С математической точки зрения это происходит, потому что производная треугольного сигнала не определена в вершинах треугольников.

9.3. Упражнение 3

В данном упражнении изучается влияние `cumsum` и `integrate` на сигнал. Создадим прямоугольный сигнал и напечатаем его (Рис.9.13).

```
from thinkdsp import SquareSignal
square = SquareSignal(freq=50).make_wave(duration=0.1, framerate=44100)
square.plot()
decorate(xlabel='Time (s)')
```

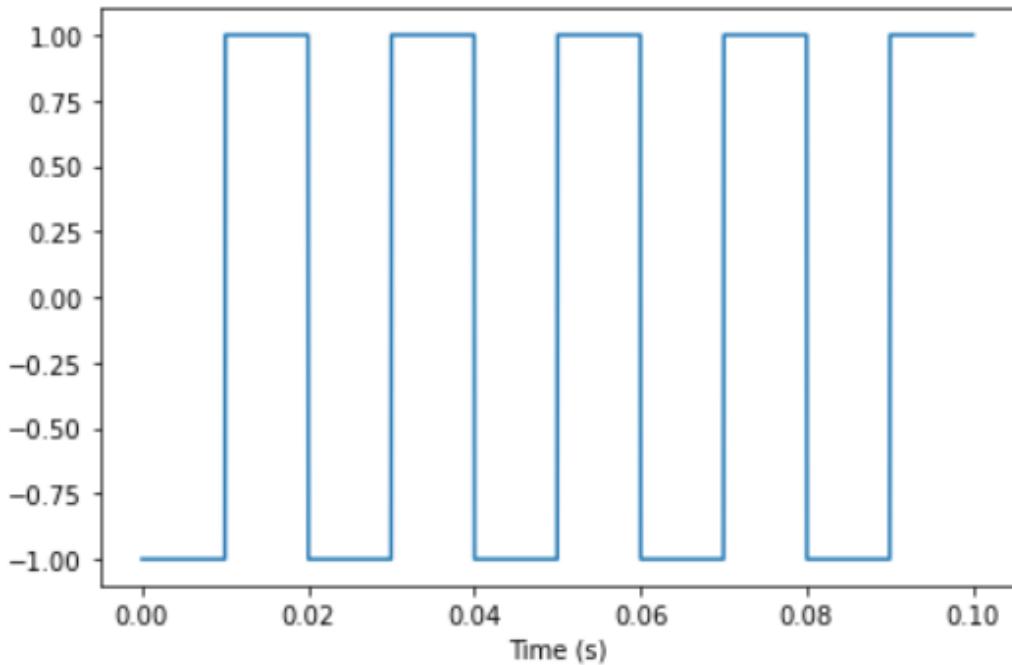


Рис. 9.13. Прямоугольный сигнал

Применим к нему `cumsum` и напечатаем результат (Рис.9.14).

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

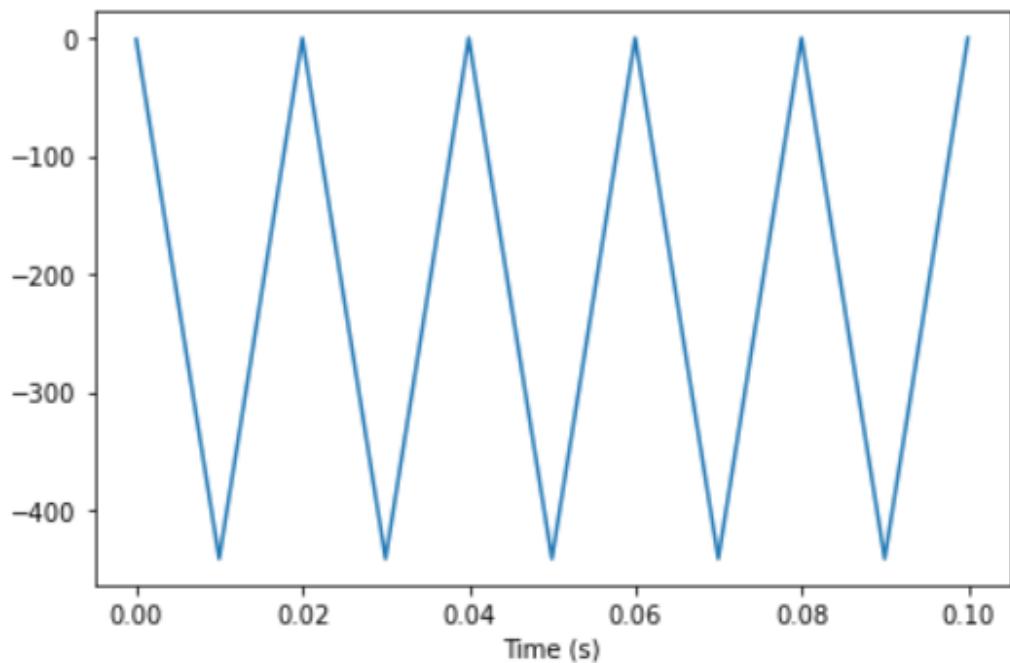


Рис. 9.14. Результат cumsum

Вычислим спектр прямоугольного сигнала, применим к нему integrate и напечатаем (Рис.9.15).

```
spectrum = square.make_spectrum().integrate()  
spectrum.plot()  
decorate(xlabel='Frequency (Hz)')
```

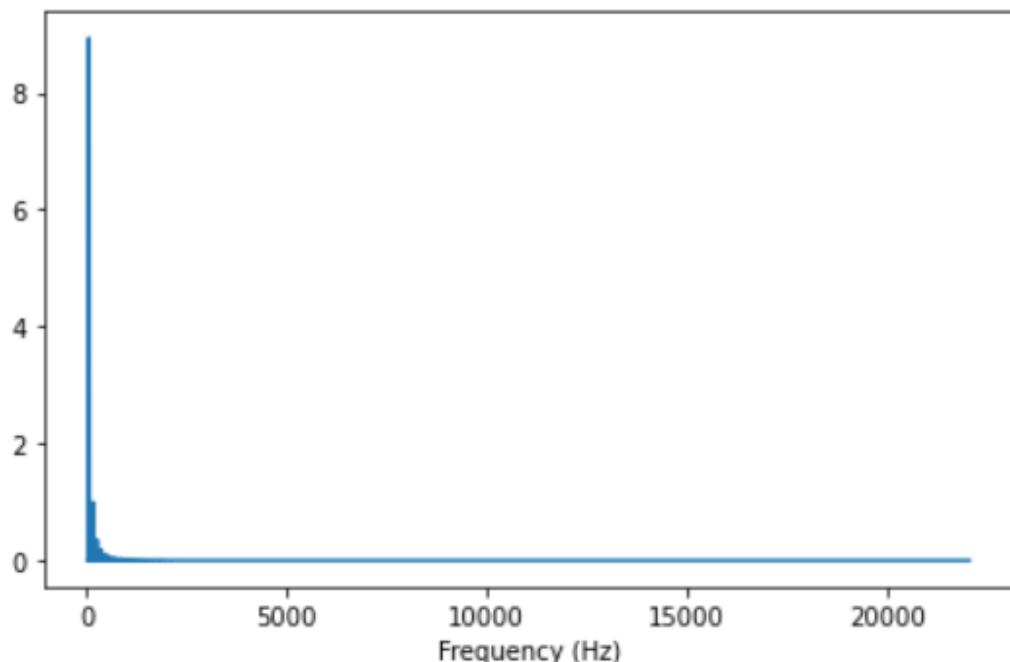


Рис. 9.15. Результат integrate

Теперь преобразуем спектр обратно в сигнал и напечатаем его (Рис.9.16).

```
spectrum.hs[0] = 0
out_wave2 = spectrum.make_wave()
out_wave2.plot()
decorate(xlabel='Time (s)')
```

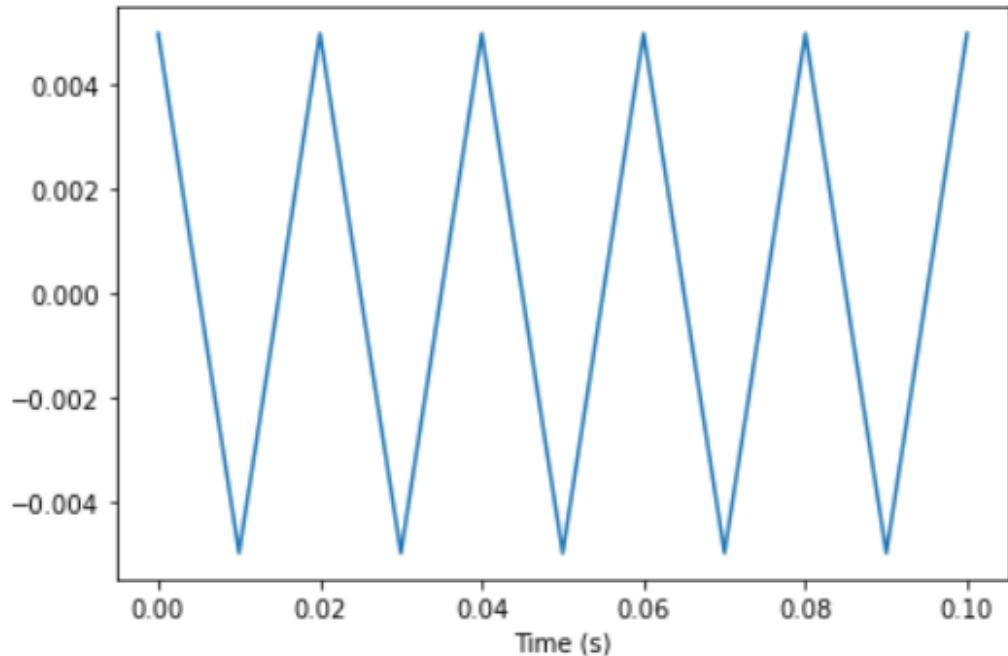


Рис. 9.16. Спектр в сигнал

Результаты `cumsum` и `integrate` с виду получились одинаковыми. Проверим, есть ли какие-то различия между этими функциями (Рис.9.17).

```
out_wave.unbias()
out_wave.normalize()
out_wave2.normalize()
out_wave.plot()
out_wave2.plot()
decorate(xlabel='Time (s)')
out_wave.max_diff(out_wave2)
```

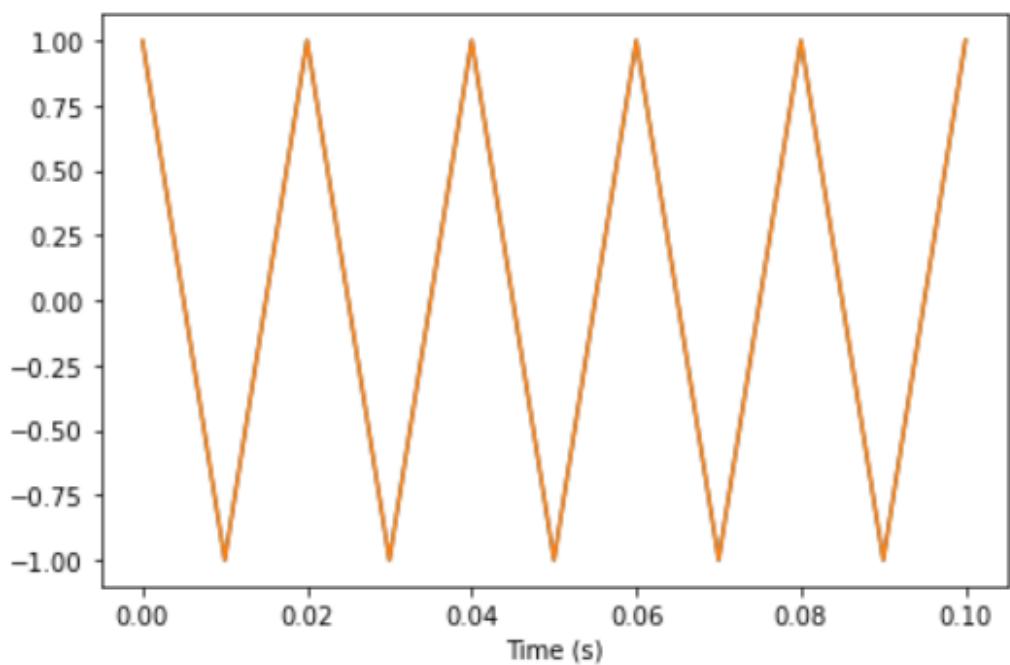


Рис. 9.17. Сравнение функций

Численно они тоже схожи, но с точностью около 3 цифр.

9.4. Упражнение 4

В этом упражнении изучается влияние двойного интегрирования.

Создадим пилообразный сигнал, вычислим его спектр и дважды применим `integrate`. Затем напечатаем результирующий сигнал и его спектр.

```
from thinkdsp import SawtoothSignal
sawtooth = SawtoothSignal(freq=50).make_wave(duration=0.1, framerate=44100)
spectrum = sawtooth.make_spectrum().integrate().integrate()
spectrum.hs[0] = 0
out_wave = spectrum.make_wave()
out_wave.plot()
decorate(xlabel='Time (s)')
```

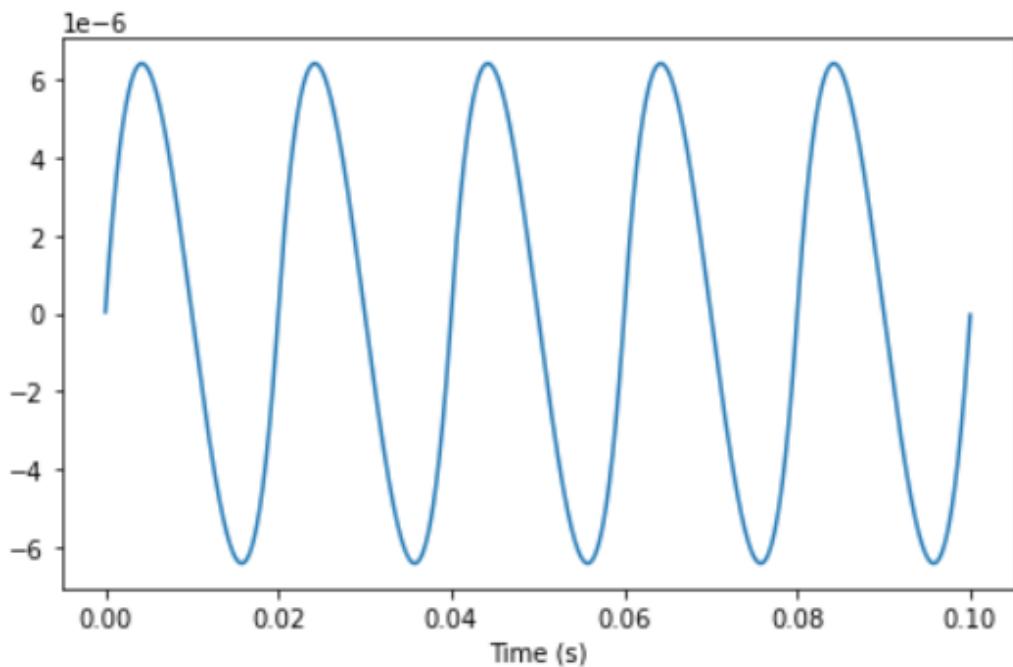


Рис. 9.18. Результирующий сигнал

```
out_wave.make_spectrum().plot(high=500)
decorate(xlabel='Frequency (Hz)')
```

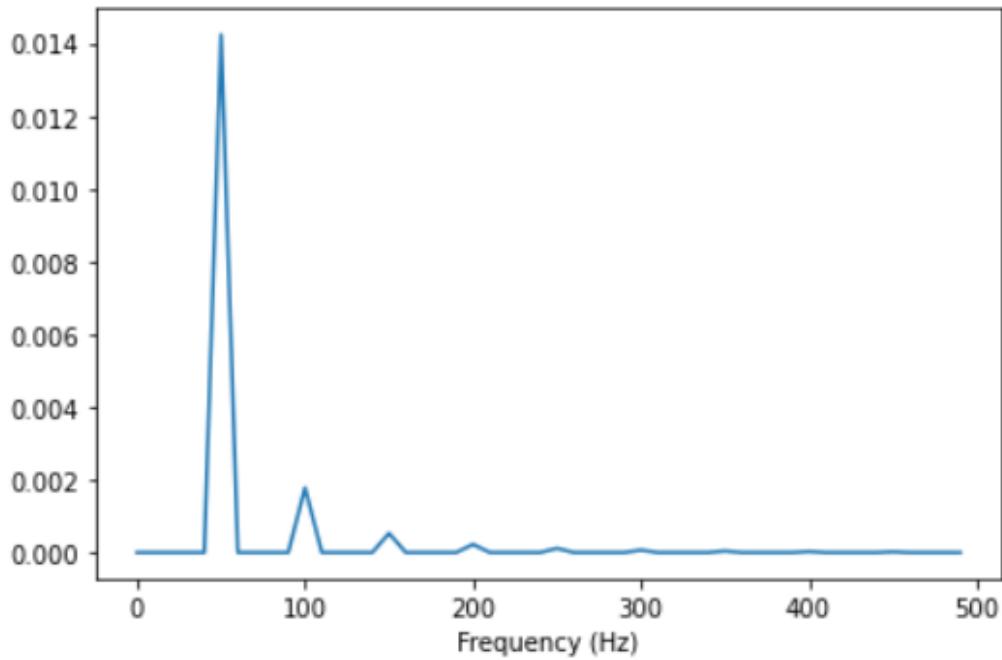


Рис. 9.19. Спектр результирующего сигнала

Из рис.9.18 мы можем видеть, что результат напоминает синусоиду. Причина в том, что интегрирование действует как фильтр низких частот. Из спектра видно, что после двойного применения функции `integrate` мы отфильтровали почти все, кроме основных частот (Рис.9.19).

9.5. Упражнение 5

В данном упражнении изучается влияние второй разности и второй производной.

Создадим CubicSignal, определённый в thinkdsp.

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

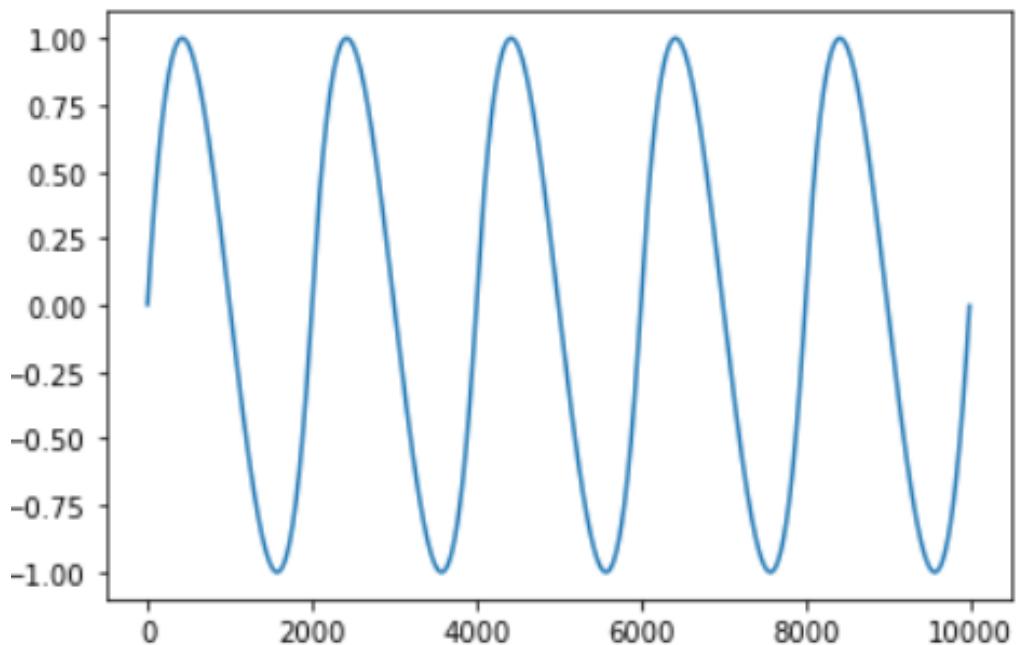


Рис. 9.20. Кубический сигнал

Вычислим его вторую разность, дважды применив diff. Напечатаем получившийся результат (Рис.9.21).

```
out_wave = cubic.diff()
out_wave = out_wave.diff()
out_wave.plot()
decorate(xlabel='Time (s)')
```

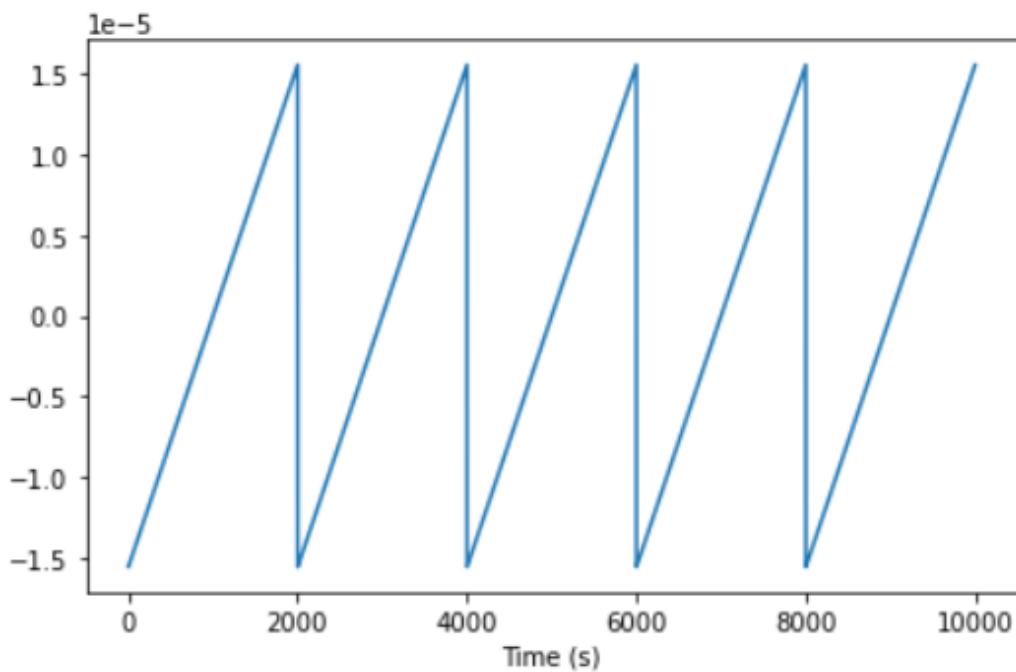


Рис. 9.21. Вторая разность кубического сигнала

Получившийся результат похож на пилообразный сигнал.

Вычислим вторую производную, дважды применив `differentiate` к спектру. Напечатаем получившийся результат (Рис.9.22).

```
spectrum = cubic.make_spectrum().differentiate().differentiate()
out_wave2 = spectrum.make_wave()
out_wave2.plot()
decorate(xlabel='Time (s)')
```

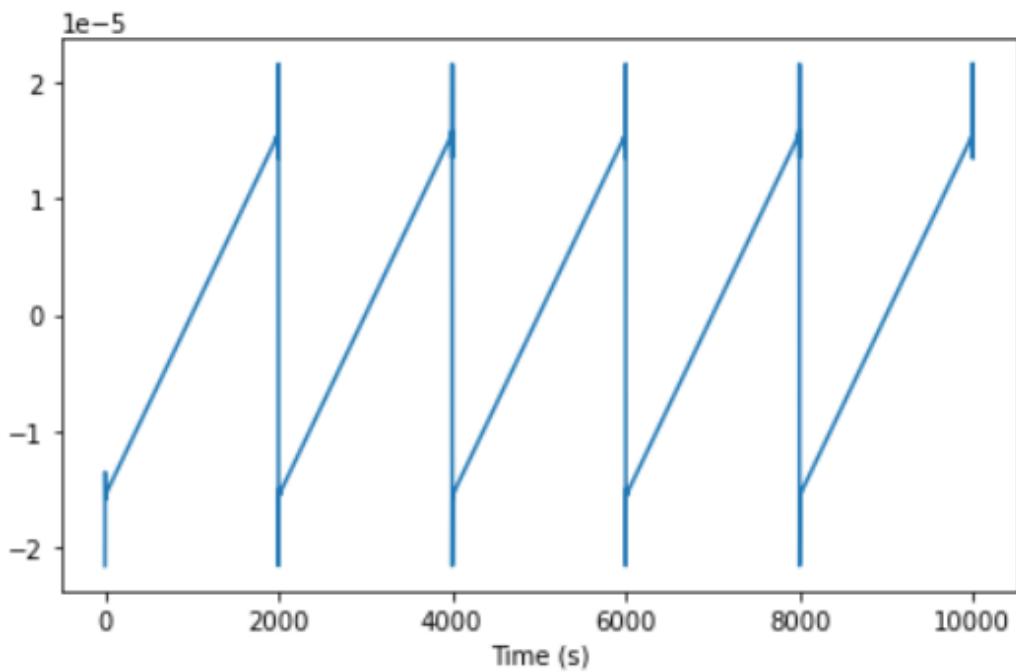


Рис. 9.22. Вторая производная кубического сигнала

Как видим, мы получили пилообразную форму с некоторым звоном. Распечатаем фильтры, соответствующие второй разности и второй производной, и сравним их (Рис.9.23).

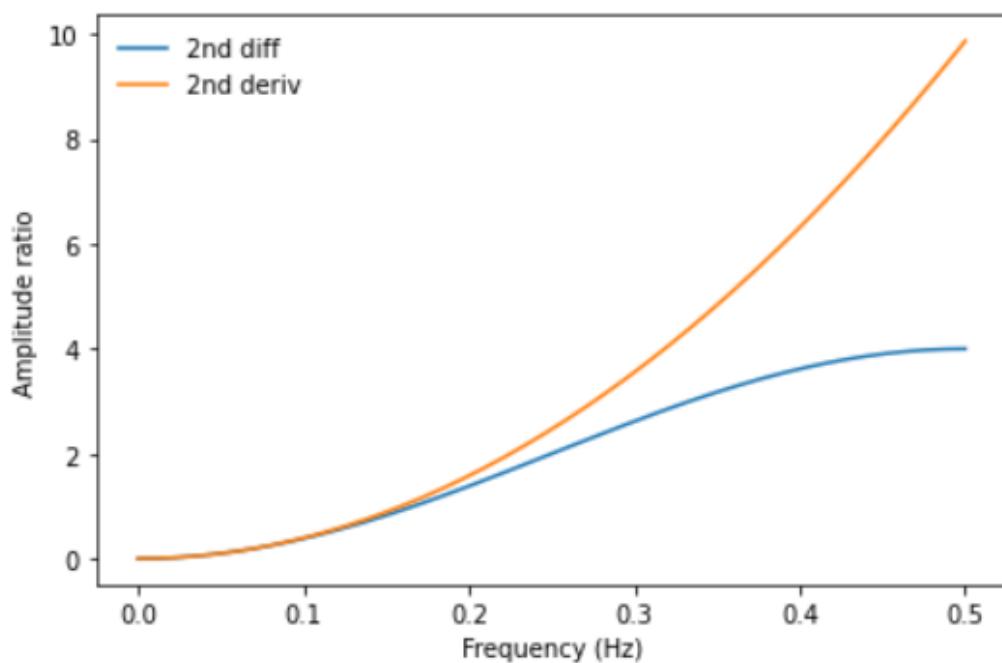


Рис. 9.23. Сравнение фильтра

Оба фильтра являются фильтрами высоких частот, которые усиливают компоненты самых высоких частот. Вторая производная параболическая, поэтому она сильнее всего усиливает самые высокие частоты. Вторая разность - хорошее приближение второй производной только на самых низких частотах, затем она существенно отклоняется.

9.6. Выводы

В результате выполнения данной работы мы изучили функции дифференцирования и интегрирования, а также научились применять их на различных сигналах.

10. Лабораторная работа 10

10.1. Упражнение 1

In this chapter I describe convolution as the sum of shifted, scaled copies of a signal. Strictly speaking, this operation is linear convolution, which does not assume that the signal is periodic.

But when we multiply the DFT of the signal by the transfer function, that operation corresponds to circular convolution, which assumes that the signal is periodic. As a result, you might notice that the output contains an extra note at the beginning, which wraps around from the end.

Fortunately, there is a standard solution to this problem. If you add enough zeros to the end of the signal before computing the DFT, you can avoid wrap-around and compute a linear convolution.

Modify the example in `chap10soln.ipynb` and confirm that zero-padding eliminates the extra note at the beginning of the output.

Модифицируем исходную программу, сначала сократив оба сигнала до 2^{16} , затем добавив в конец массивов такое количество нулей, чтобы размер массивов стал 2^{17} .

```
response_1 = read_wave('.. /180960_kleeb_gunshot.wav')

start_1 = 0.12
response_1 = response_1.segment(start=start_1)
response_1.shift(-start_1)

response_1.truncate(2 ** 16)
response_1.zero_pad(2 ** 17)

response_1.normalize()
response_1.plot()
decorate(xlabel='Time (s)')
```

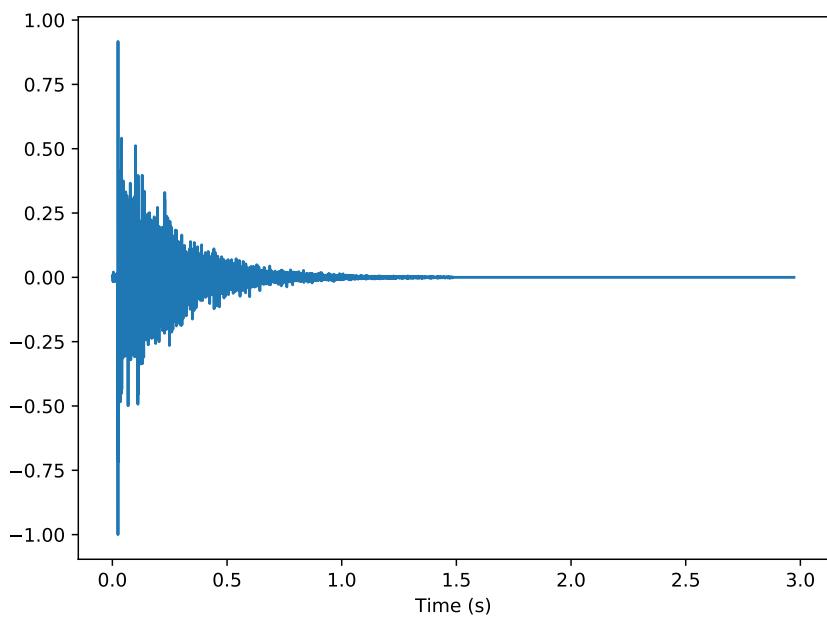


Рис. 10.1. График изменённого выстрела

Построим спектр изменённого выстрела.

```
transfer_1 = response_1.make_spectrum()  
transfer_1.plot()  
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

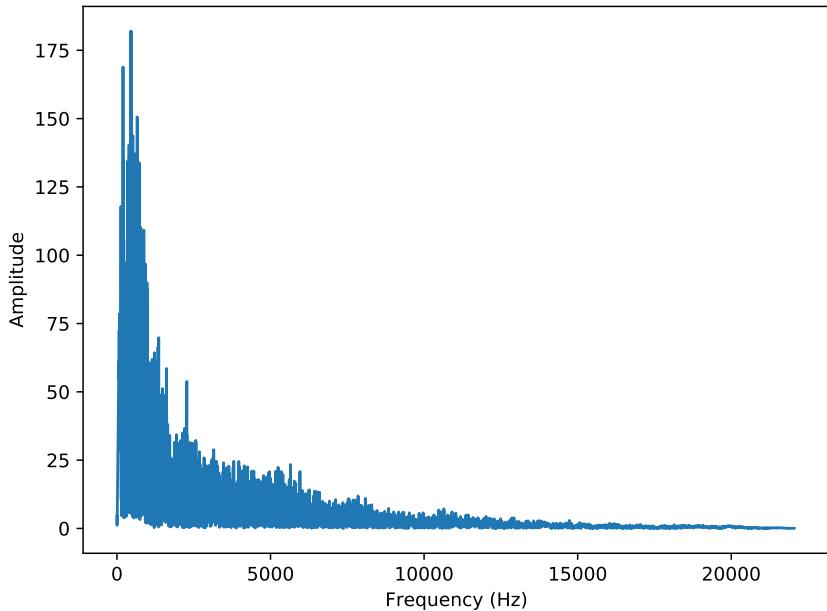


Рис. 10.2. Спектр изменённого выстрела

Теперь также добавим нулей для скрипки и посмотрим для неё график.

```

violin_1 = violin_1.segment(start=start_1)
violin_1.shift(-start_1)

violin_1.truncate(2 ** 16)
violin_1.zero_pad(2 ** 17)

violin_1.normalize()
violin_1.plot()

```

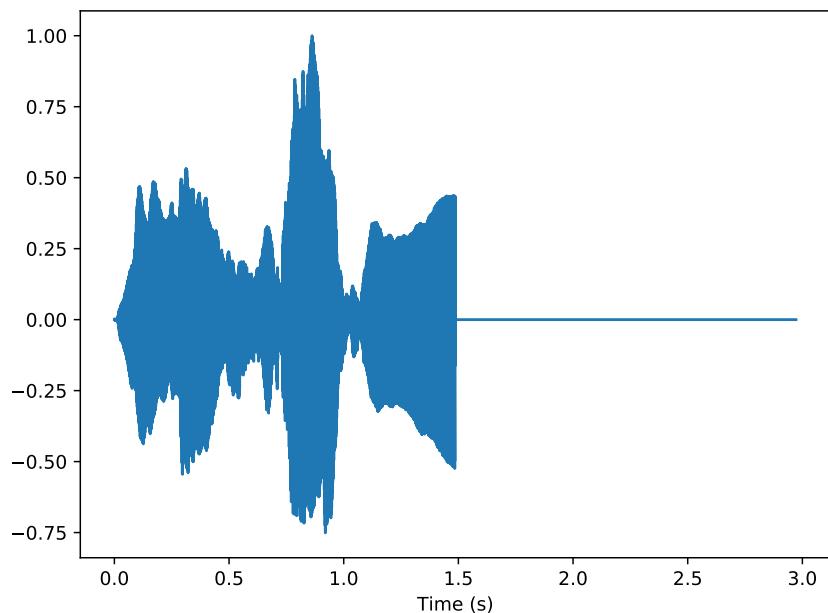


Рис. 10.3. График изменённой скрипки

Теперь наложим спектр выстрела на спектр скрипки.

```

spectrum_1 = violin_1.make_spectrum()

output_1 = (spectrum_1 * transfer_1).make_wave()
output_1.normalize()

output_1.make_audio()

output_1.plot()

```

После сравнения трансформированного звука скрипки из `chap10.ipynb` и нашего можно понять, что "затёкшей" ноты в начале больше не слышно.

10.2. Упражнение 2

The Open AIR library provides a "centralized... on-line resource for anyone interested in auralization and acoustical impulse response data" (<https://www.openairlib.net>). Browse their collection of impulse response data and download one that sounds interesting. Find a short recording that has the same sample rate as the impulse response you downloaded.

Simulate the sound of your recording in the space where the impulse response was measured, computed two way: by convolving the recording with the impulse response and by computing the filter that corresponds to the impulse response and multiplying by the DFT of the recording.

Скачаем с библиотеки OPEN AIR (<https://openairlib.net>) импульсную характеристику, имеющую подходящую частоту дискретизации. Была выбрана запись из Споканского женского клуба.

```
response_woman_club_2 = read_wave('..../spokane_womans_club_ir.wav')

duration_2 = 5
response_woman_club_2 = response_woman_club_2.segment(duration=duration_2)

response_woman_club_2.normalize()
response_woman_club_2.plot()
decorate(xlabel='Time (s)')
```

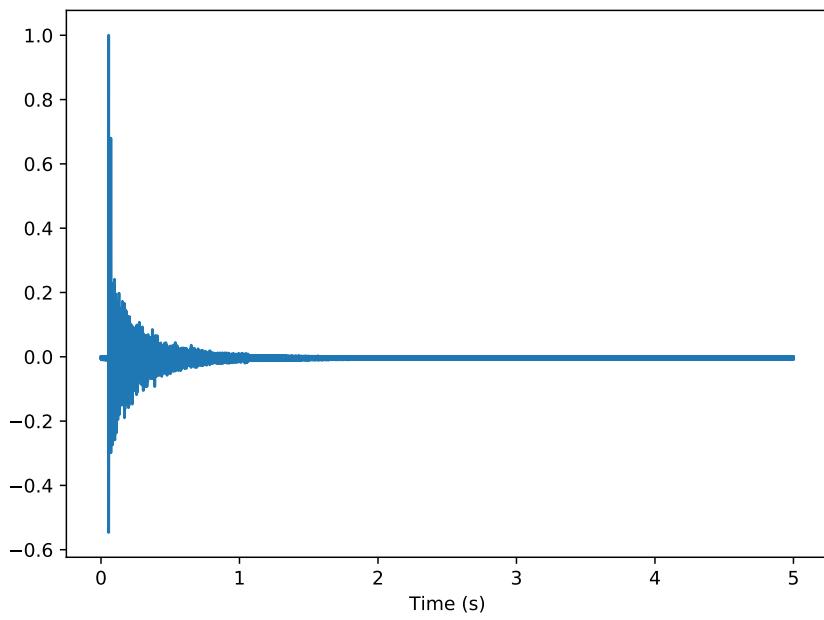


Рис. 10.4. График сигнала из женского клуба

Создадим спектр и логарифмический спектр сигнала из женского клуба.

```
transfer_2 = response_woman_club_2.make_spectrum()
transfer_2.plot()
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude')
```

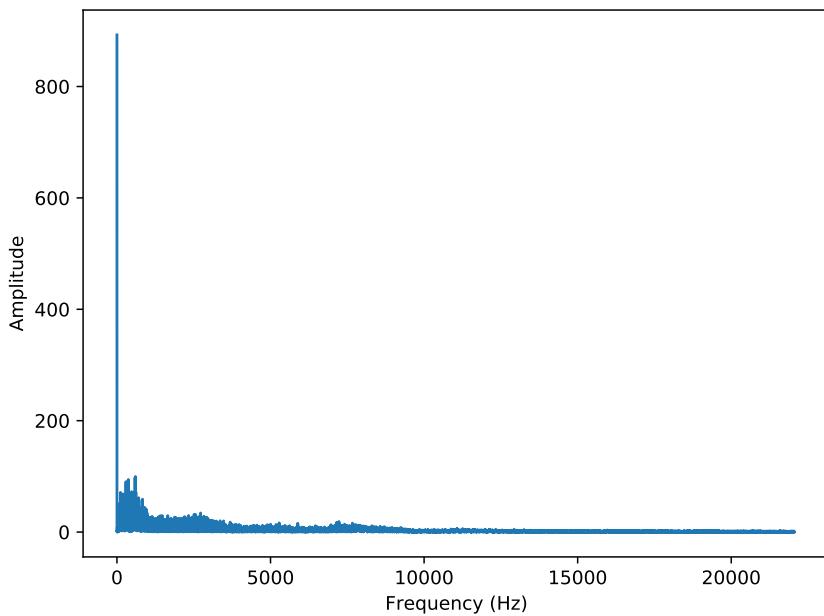


Рис. 10.5. Спектр сигнала из женского клуба

```
transfer_2.plot()
decorate(xlabel='Frequency (Hz)', ylabel='Amplitude',
        xscale='log', yscale='log')
```

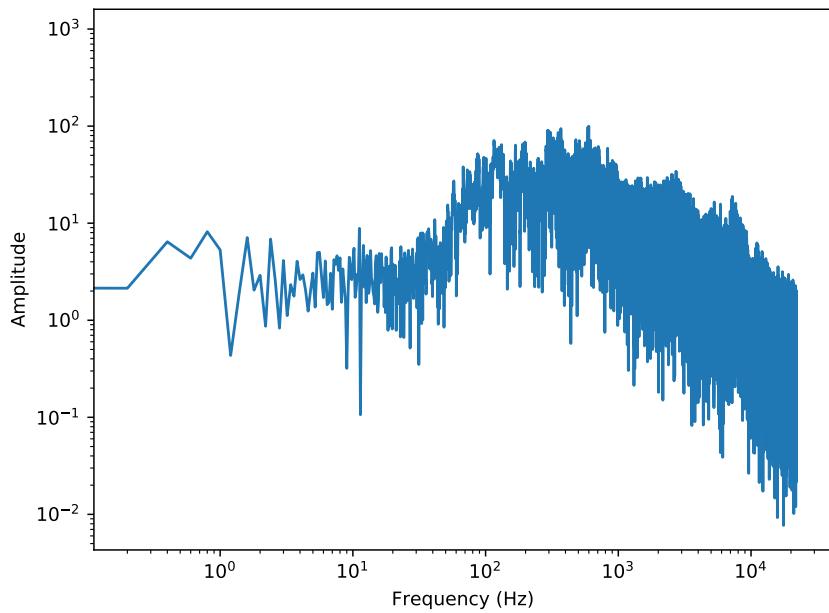


Рис. 10.6. Логарифмический спектр сигнала из женского клуба

Воспользуемся звуком пианино из упражнения 1.2 как трансформируемую запись.

```
wave_piano_2 = read_wave('..../32158_zin_piano-2-140bpm.wav')

wave_piano_2 = wave_piano_2.segment(duration=duration_2)

wave_piano_2.truncate(len(response_woman_club_2))
wave_piano_2.normalize()

wave_piano_2.make_audio()

spectrum_2 = wave_piano_2.make_spectrum()
```

Проверим, что запись имеет такую же частоту дискретизации, что и импульсная характеристика.

```
print(len(spectrum_2.hs), len(transfer_2.hs))
# 110251 110251
```

Частоты совпадают, следовательно можем использовать эту запись для трансформации.

```
output_2 = (spectrum_2 * transfer_2).make_wave()
output_2.normalize()

wave_piano_2.plot()
```

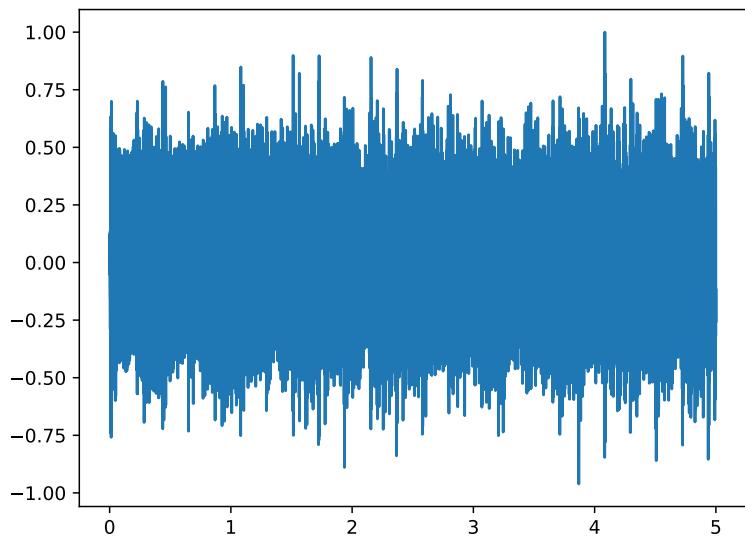


Рис. 10.7. Исходный сигнал пианино

```
output_2.plot()
```

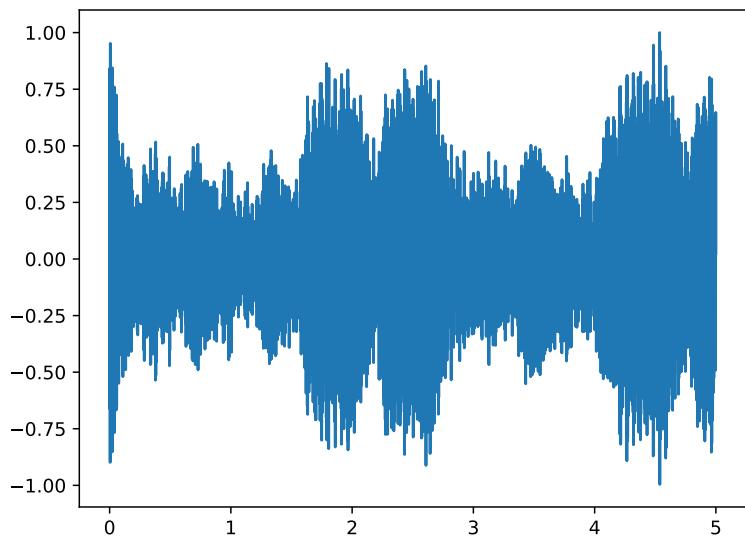


Рис. 10.8. Трансформированный сигнал пианино

Прослушаем трансформированный звук.

```
output_2.make_audio()
```

Пианино действительно как будто звучит в большом помещении.

Теперь получим трансформированный сигнал другим способом: используя свёртку с помощью функции `convolve`.

```
convolved_2 = wave_piano_2.convolve(response_woman_club_2)
convolved_2.normalize()
convolved_2.make_audio()

convolved_2.plot()
```

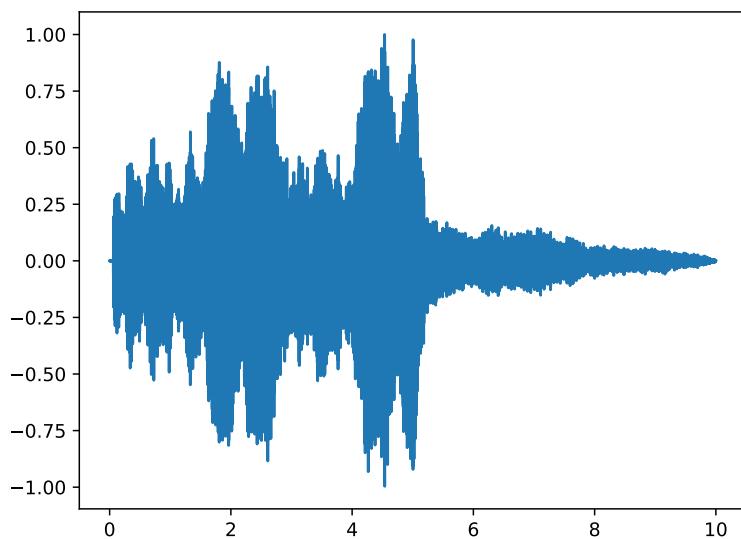


Рис. 10.9. Сигнал пианино, трансформированный с помощью свёртки

Пианино звучит также, как и в первом случае.

10.3. Выводы

В результате выполнения данной работы мы изучили линейные стационарные системы. Также мы научились моделировать звучание в пространстве двумя способами: свёрткой записи с импульсной характеристикой и умножением ДПФ записи на вычисленный фильтр, соответствующий импульсной характеристике.

11. Лабораторная работа 11

11.1. Упражнение 1

The code in this chapter is in `chap11.ipynb`. Read through it and listen to the examples.

Необходимо было открыть `chap11.ipynb`, прочитать пояснения и запустить все примеры.

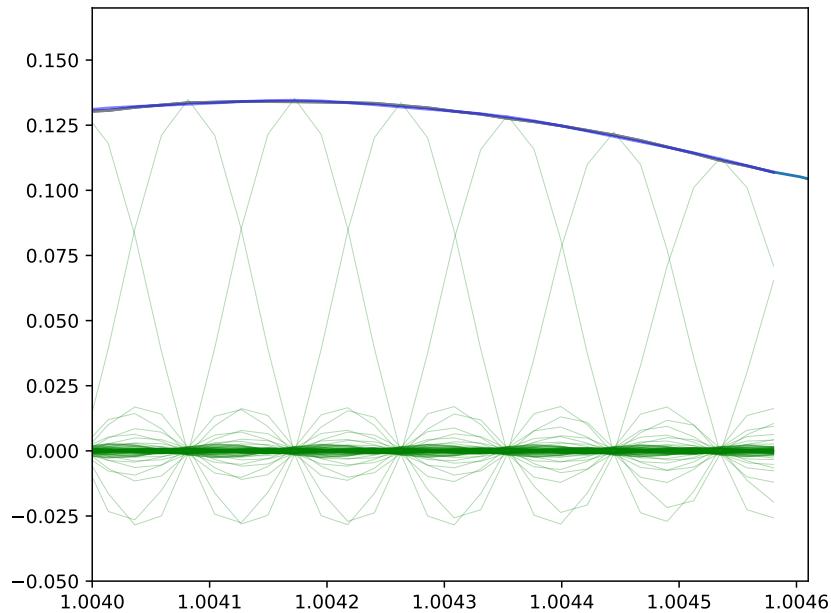


Рис. 11.1. График из последнего примера

Все примеры были успешно запущены.

11.2. Упражнение 2

Chris “Monty” Montgomery has an excellent video called “D/A and A/D | Digital Show and Tell”; it demonstrates the Sampling Theorem in action, and presents lots of other excellent information about sampling. Watch it at <https://www.youtube.com/watch?v=cIQ9IXSUzuM>.

В [видео](#) Криса Монтгомери демонстрируется теорема о выборках на практике и представляется множество другой информации о выборках. В частности было объяснено, почему аналоговый звук в пределах человеческого слуха может воспроизводиться с идеальной точностью с использованием 16-битного цифрового сигнала 44,1 кГц.

11.3. Упражнение 3

As we have seen, if you sample a signal at too low a framerate, frequencies above the folding frequency get aliased. Once that happens, it is no longer possible to filter out these components, because they are
↪ indistinguishable from lower frequencies.

It is a good idea to filter out these frequencies before sampling; a low-pass filter used for this purpose is called an anti-aliasing filter. Returning to the drum solo example, apply a low-pass filter before sampling, then apply the low-pass filter again to remove the spectral copies introduced by sampling. The result should be identical to the filtered
↪ signal.

Скопируем звук барабана и построим для него график.

```
wave_3 = read_wave('..../263868_kevcio_amen-break-a-160-bpm.wav')
wave_3.normalize()
wave_3.plot()
```

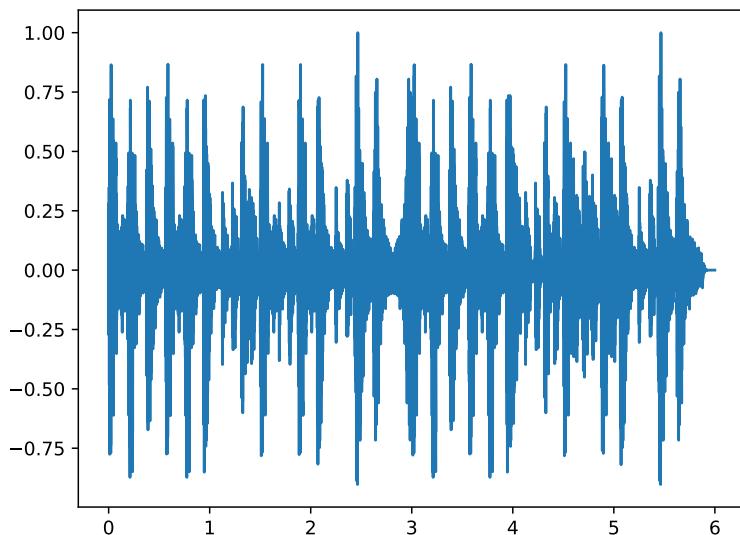


Рис. 11.2. График для звука барабана

Сделаем запись для прослушивания.

```
wave_3.make_audio()
```

Теперь получим спектр для сигнала и построим график для спектра.

```
spectrum_3 = wave_3.make_spectrum(full=True)
spectrum_3.plot()
```

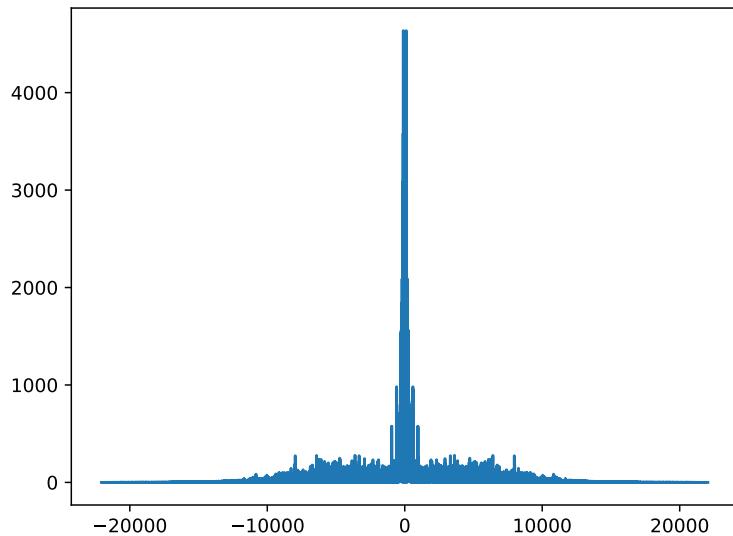


Рис. 11.3. График для спектра звука барабана

Уменьшаем частоту дискретизации в 3 раза, после чего высчитываем новое значение частоты свертки.

```
framerate_3 = wave_3 framerate / 3
cutoff_3 = framerate_3 / 2 - 1
```

Удаляем частоты выше новой частоты свертки и строим график для нового спектра.

```
spectrum_3.low_pass(cutoff_3)
spectrum_3.plot()
```

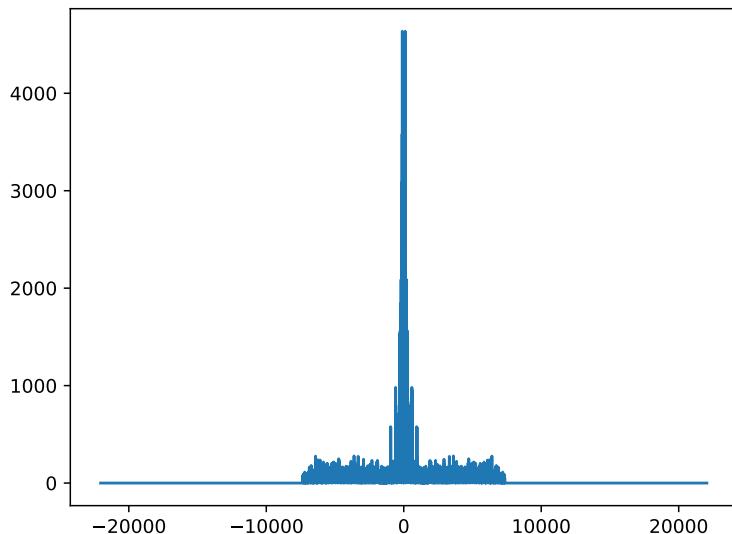


Рис. 11.4. График для отфильтрованного спектра сигнала

Создаем отфильтрованный звук для прослушивания.

```
filtered_signal_3 = spectrum_3.make_wave()
filtered_signal_3.make_audio()
```

Теперь напишем функцию для имитации выборки.

```
def sample_imitation(wave):
    ys = np.zeros(len(wave))
    ys[::3] = np.real(wave.ys[::3])
    return Wave(ys, framerate=wave.framerate)
```

И воспользуемся созданной функцией на отфильтрованном сигнале.

```
sampled_signal_3 = sample_imitation(filtered_signal_3)
sampled_signal_3.make_audio()
```

Теперь для полученного сигнала построим график для спектра.

```
sampled_signal_spectrum_3 = sampled_signal_3.make_spectrum(full=True)
sampled_signal_spectrum_3.plot()
```

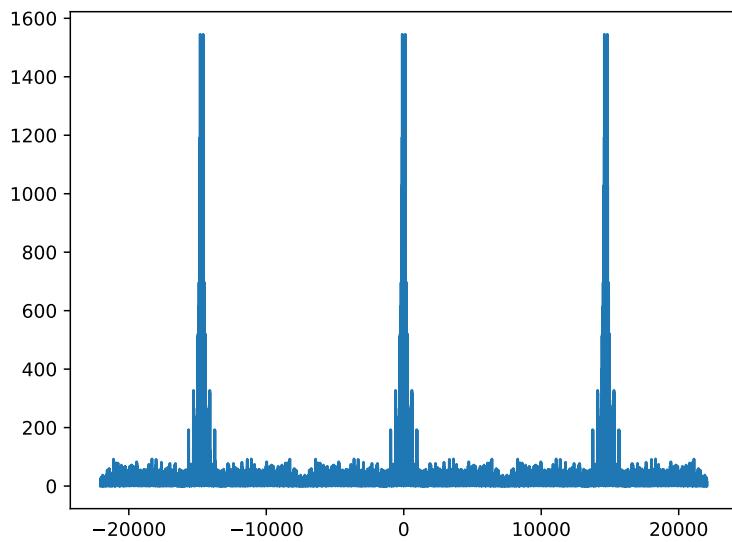


Рис. 11.5. График спектра для сигнала после выборки

Данный спектр содержит спектральные копии, которые слегка заметны при прослушивании звука. Уберем спектральные копии и посмотрим на измененный график.

```
sampled_signal_spectrum_3.low_pass(cutoff_3)  
sampled_signal_spectrum_3.plot()
```

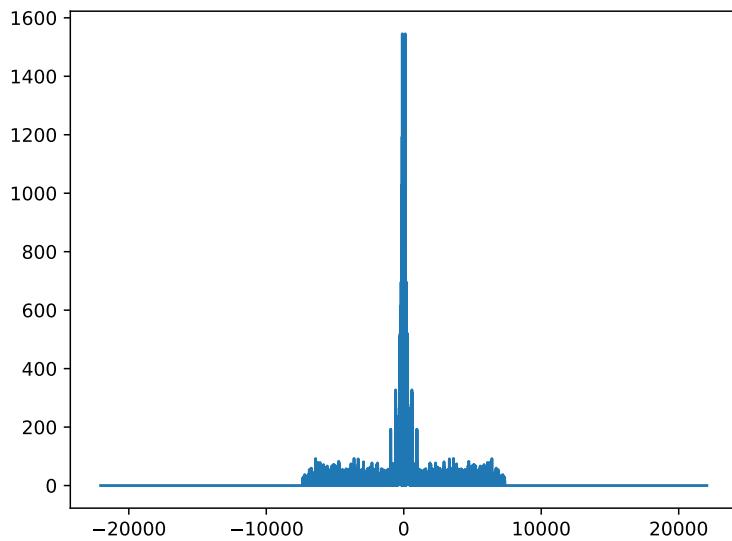


Рис. 11.6. График спектра для сигнала после выборки без спектральных копий

Масштабируем полученный спектр для восстановления энергии и сравним с отфильтрованным.

```
sampled_signal_spectrum_3.scale(3)
sampled_signal_spectrum_3.plot(color='gray')
spectrum_3.plot()
```

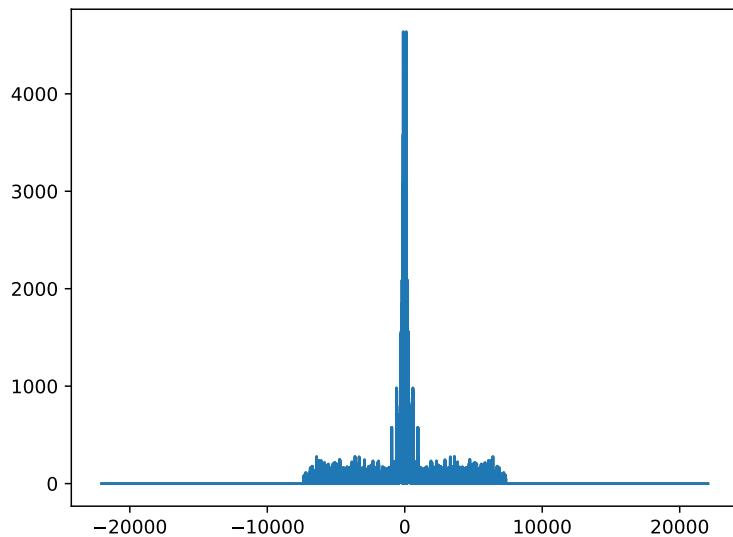


Рис. 11.7. Сравнительный график для отфильтрованного спектра и после выборки

На графике разница не видна, поэтому повсему пользуемся функцией `max_diff` и высчитаем максимальную разницу между спектрами.

```
spectrum_3.max_diff(sampled_signal_spectrum_3)
```

Значение составило $1.8189894035458565e - 12$, как и ожидалось, разницы практически нет. Теперь мы можем построить сигнал из спектра после выборки.

```
interpolated_signal_3 = sampled_signal_spectrum_3.make_wave()
interpolated_signal_3.make_audio()
```

И построим график для полученного сигнала.

```
interpolated_signal_3.plot()
```

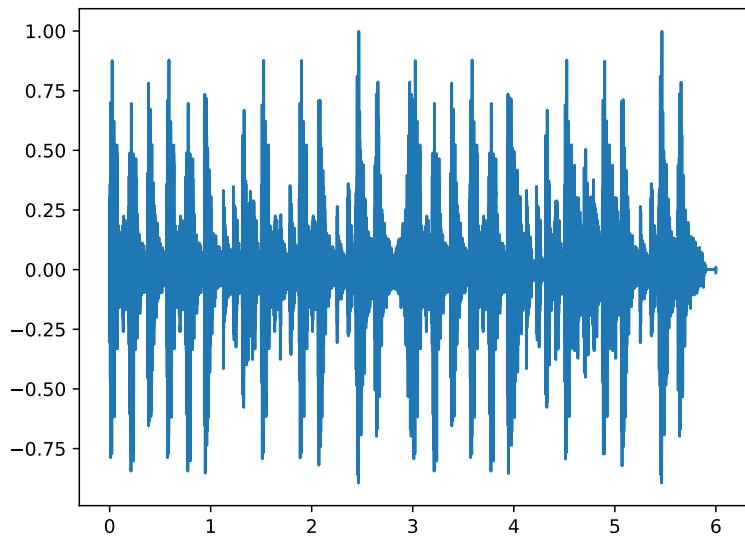


Рис. 11.8. График для сигнала после выборки

Как и ожидалось в начале работы, сигнал практически идентичен отфильтрованному.

11.4. Выводы

В данной лабораторной работе мы изучили теорему о выборках, которая является важнейшей в цифровой обработке сигналов, а также амплитудную модуляцию, которая играет важную роль в радиосвязи, получили навыки применения этих знаний на практике.

12. Лабораторная работа 12

12.1. Передача сигнала

В данной работе мы изучим, как строится PSK модулятор/демодулятор.

Первым этапом является передача сигнала QPSK. Сначала мы генерируем поток битов и модулируем его на сложное созвездие. Для этого мы используем блок Constellation Modulator. Объект созвездия позволяет нам определить, как кодируются символы.

Также нам требуется генератор случайного источника, который предоставляет упакованные байты со значениями 0-255 для модулятора созвездия.

Количество выборок на символ должно быть минимальным для обеспечения желаемой скорости передачи данных. В данной работе мы используем чуть большее число (4) для лучшей визуализации сигнала.

Наконец, мы устанавливаем значение избыточной пропускной способности. Таким образом, мы получаем flowgraph показанный на рис.12.1.

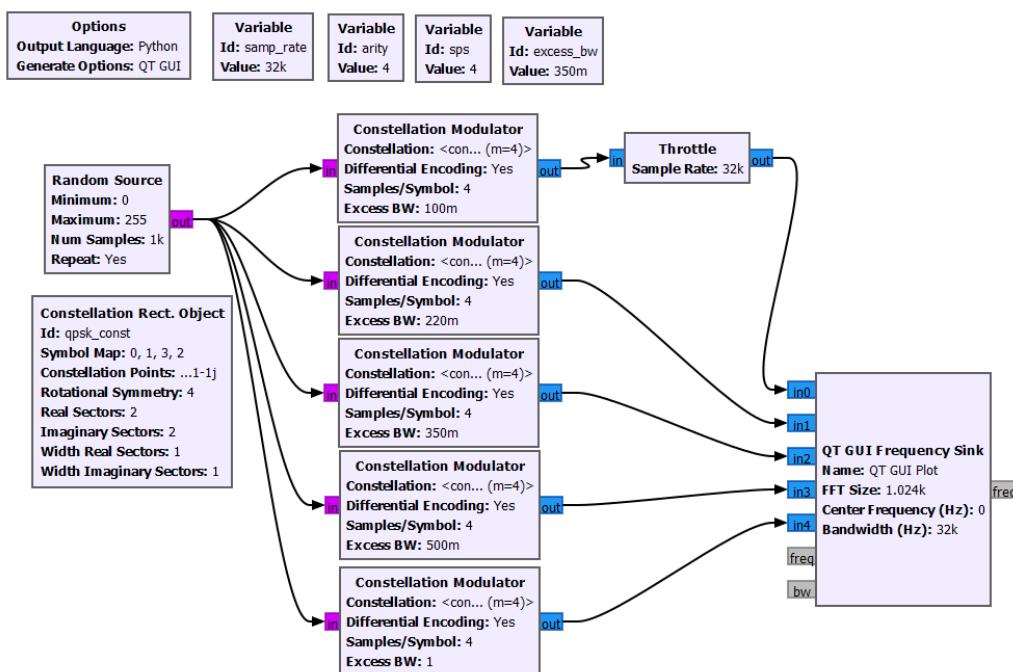


Рис. 12.1. Flowgraph избыточной пропускной способности

При запуске этого flowgraph, мы получаем график, показывающий различные значения избыточной пропускной способности (Рис.12.2). Типичные значения находятся в диапазоне от 0,2 (красная кривая) до 0,35 (зеленая кривая).

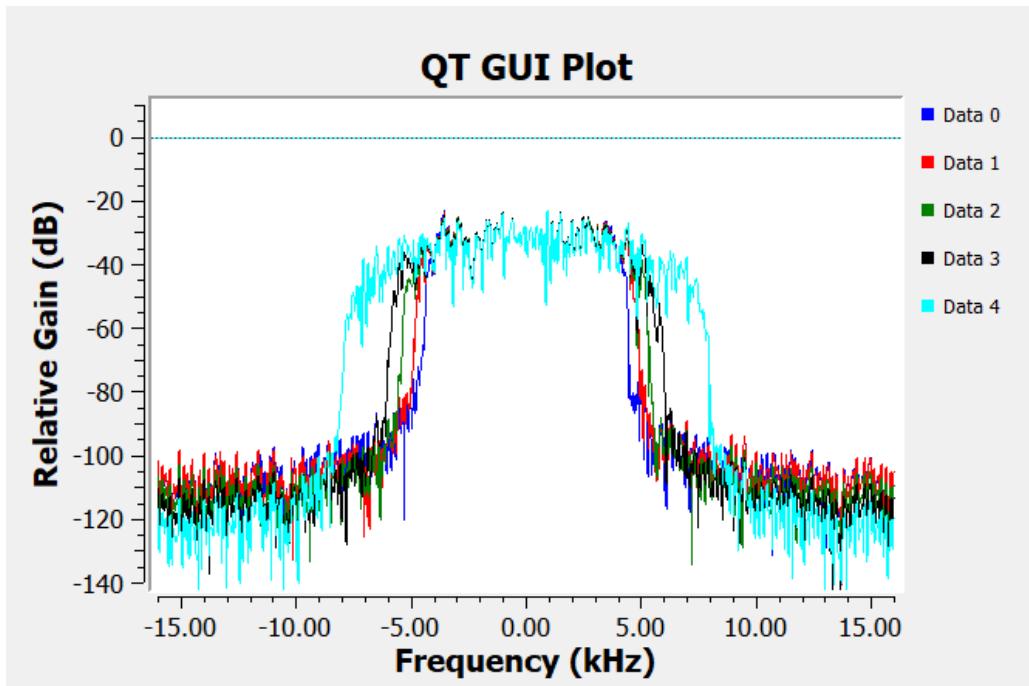


Рис. 12.2. Избыточная пропускная способность

Теперь запустим flowgraph `mpsk_stage1.grc`, передающий созвездие QPSK (Рис.12.3).

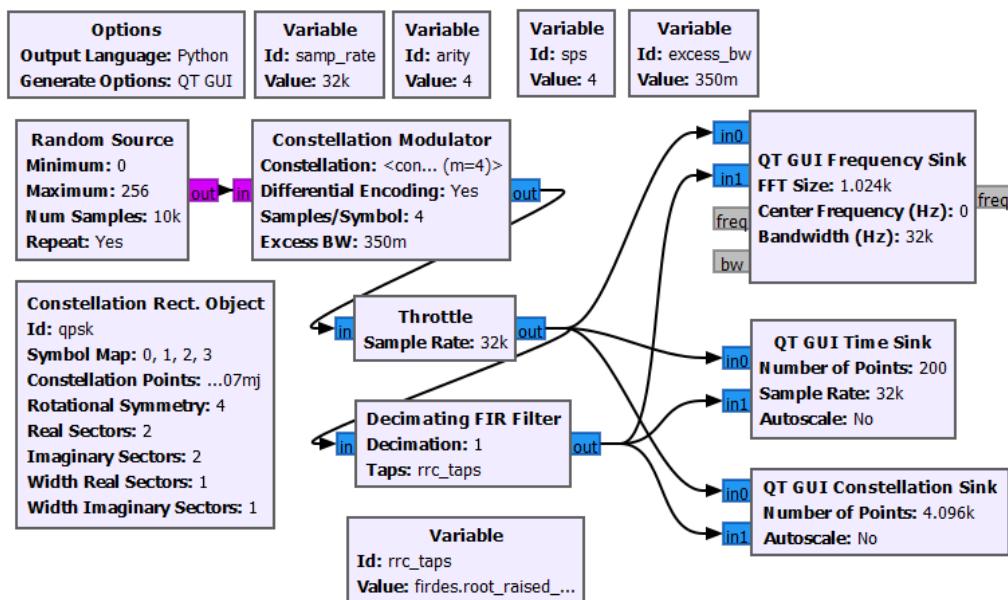


Рис. 12.3. Flowgraph созвездия QPSK

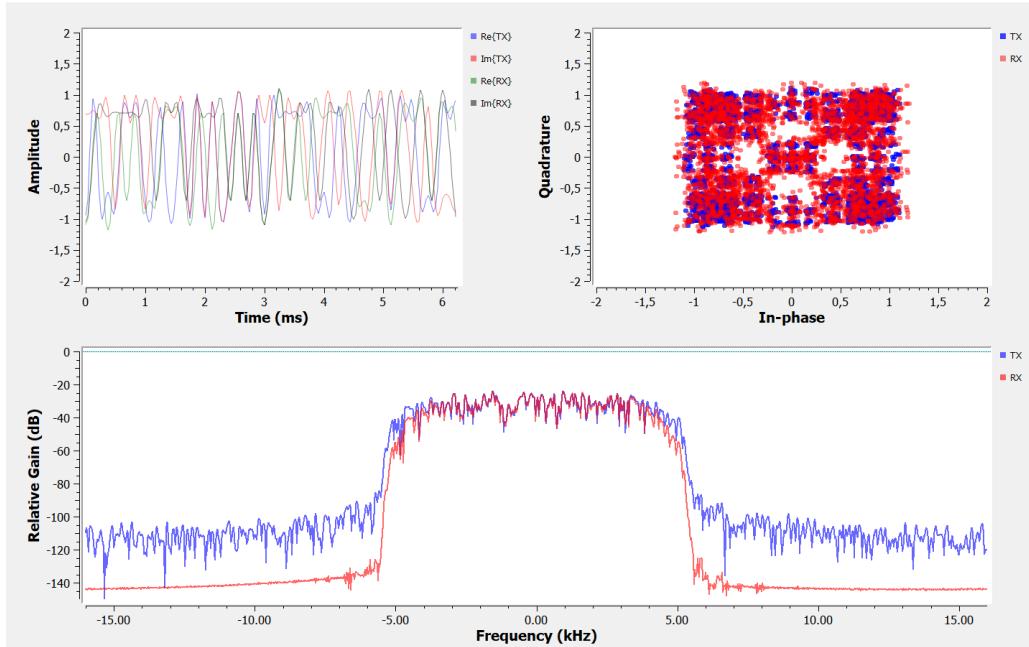


Рис. 12.4. Созвездие QPSK

На рис. 1.4 мы видим эффекты повышающей дискретизации и процесс фильтрации.

На передатчике мы используем фильтр RRC, который добавляет межсимвольные помехи (ISI). На приёмнике мы используем другой фильтр RRC для устранения ISI. Таким образом, мы сворачиваем два фильтра RRC вместе и получаем фильтр с приподнятым косинусом.

Фильтрация представляет собой свертку, поэтому выходной сигнал RRC-фильтра на приемной стороне представляет собой сигнал в форме приподнятого косинусоидального импульса с минимизированным ISI.

Получается, что на приёмнике мы используем согласованный фильтр.

Согласованный фильтр — линейный оптимальный фильтр, предназначенный для выделения сигналов известной формы на фоне шумов.

12.2. Добавление нарушений канала

Теперь мы рассмотрим влияние канала на то, как сигнал искажается между передачей и приёмом. Для начала мы будем использовать самый простой блок модели канала GNU Radio, который позволит нам смоделировать несколько основных проблем связанных с приёмником.

Проблемы:

1. Шум.

В нашем приемнике вызывается аддитивный белый гауссовский шум. Мы устанавливаем мощность этого шума, регулируя значение напряжения шума модели канала.

2. Часы, определяющие частоту радиомодулей.

Два радиомодуля имеют разные часы. Одно радио работает на частоте $f_c + f_{\text{delta_1}}$. Другое радио имеет другие часы и его реальная частота равна $f_c + f_{\text{delta_2}}$. Таким образом, полученный сигнал будет $f_{\text{delta_1}} + f_{\text{delta_2}}$.

3. Идеальная точка выборки.

Идеальная точка выборки связана с проблемой часов. Два радиомодуля работают с разной скоростью, поэтому идеальная точка выборки неизвестна.

В результате моделирования, полученный `flowgraph` позволяет нам поработать с аддитивным шумом, сдвигом частоты и временным сдвигом (Рис.12.5).

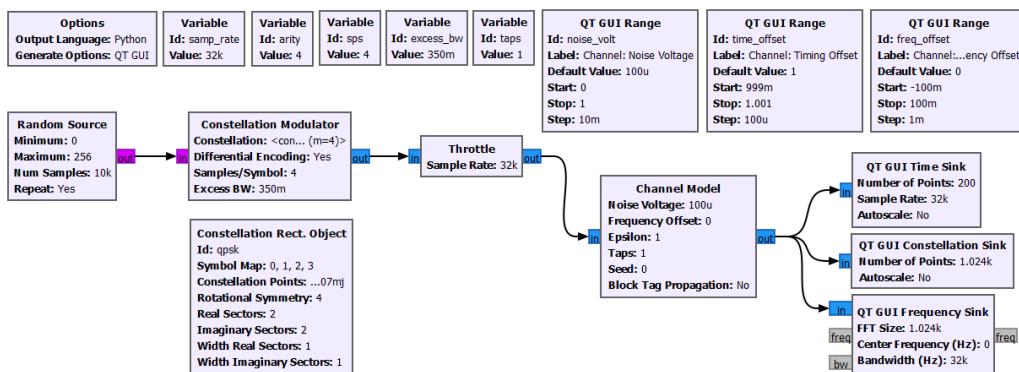


Рис. 12.5. `Flowgraph` с нарушениями канала

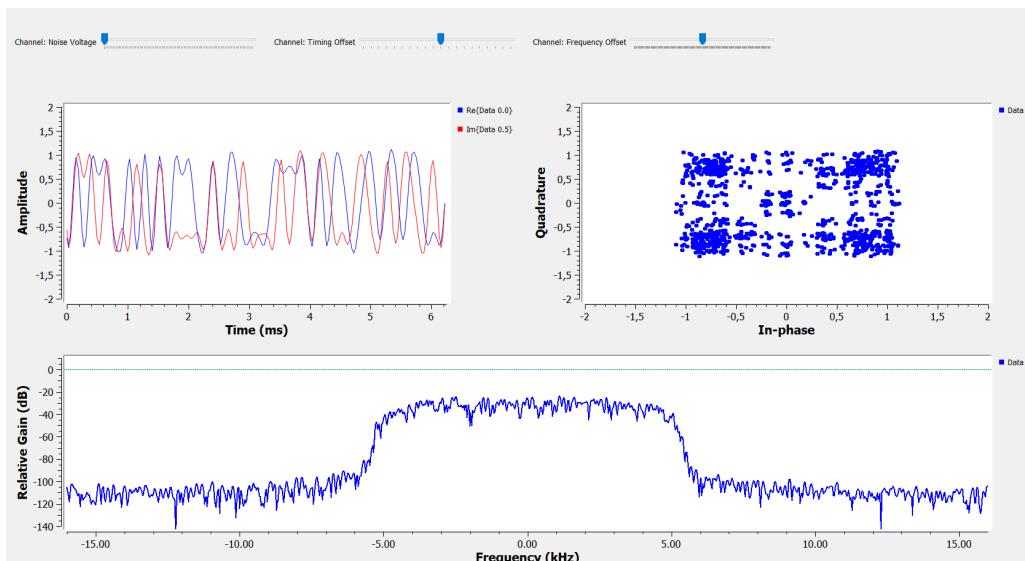


Рис. 12.6. Созвездия с нарушениями

Как мы можем видеть на рис.12.6, график созвездия показывает нам облако образцов хуже, чем на предыдущем этапе. Теперь мы должны отменить все эти эффекты.

12.3. Восстановление времени

Теперь перейдём к процессу восстановления. Здесь мы будем использовать алгоритм восстановления многофазных часов.

Мы начнем с восстановления времени. Мы пытаемся найти наилучшее время для дискретизации входящих сигналов, что позволит максимизировать отношение сигнал/шум (SNR) каждой выборки, а также уменьшить влияние межсимвольных помех (ISI).

12.3.1. Проблема ISI

Проблему ISI мы можем увидеть на примере flowgraph symbol_sampling.grc, где мы создаем четыре отдельных символа, а затем фильтруем их (Рис.12.7).

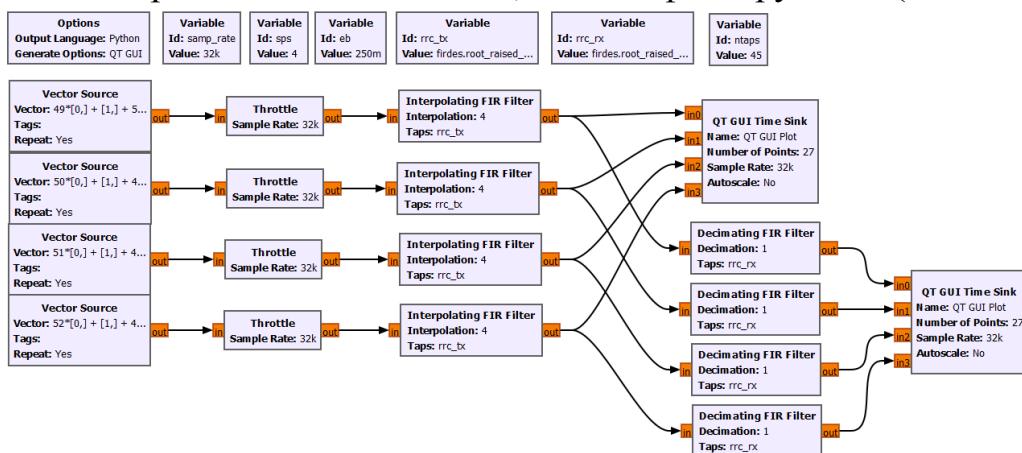


Рис. 12.7. Flowgraph проблемы ISI

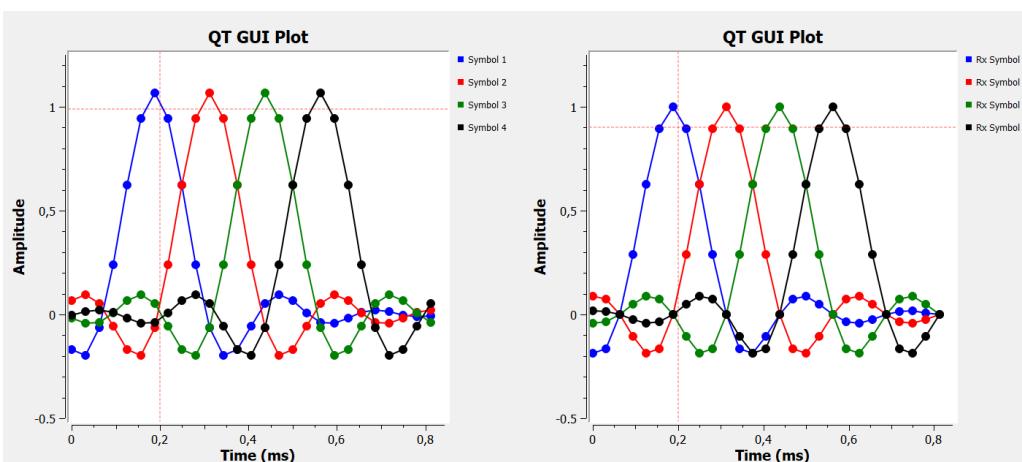


Рис. 12.8. Сравнение символов, отфильтрованных RC и RRC

На рис.12.8 мы видим выходные данные, которые показывают различия между символами, отфильтрованными RC и RRC.

Без фильтрации Найквиста (RRC - справа) мы можем увидеть, как в идеальной точке выборки каждого символа другие символы имеют некоторую энергию. Если мы суммируем эти символы вместе (как в непрерывном потоке), то энергия других выборок складывается вместе и искажает символ в этой точке.

И наоборот, на выходе с RC-фильтром энергия от других выборок равна 0 в идеальной точке дискретизации для данного символа во времени. Это означает, что если мы делаем выборку точно в правильной точке выборки, мы получаем энергию только от текущего символа без помех от других символов в потоке.

Это моделирование позволяет нам легко настроить количество выборок на символ, избыточную полосу пропускания фильтров RRC и количество ответвлений.

12.3.2. Разные часы

Затем мы посмотрим, как разные часы влияют на точки выборки между передатчиком и приемником. Используя пример потокового графа в `symbol_sampling_diff.grc`, мы моделируем влияние разных часов в передатчике и приемнике (Рис.12.9).

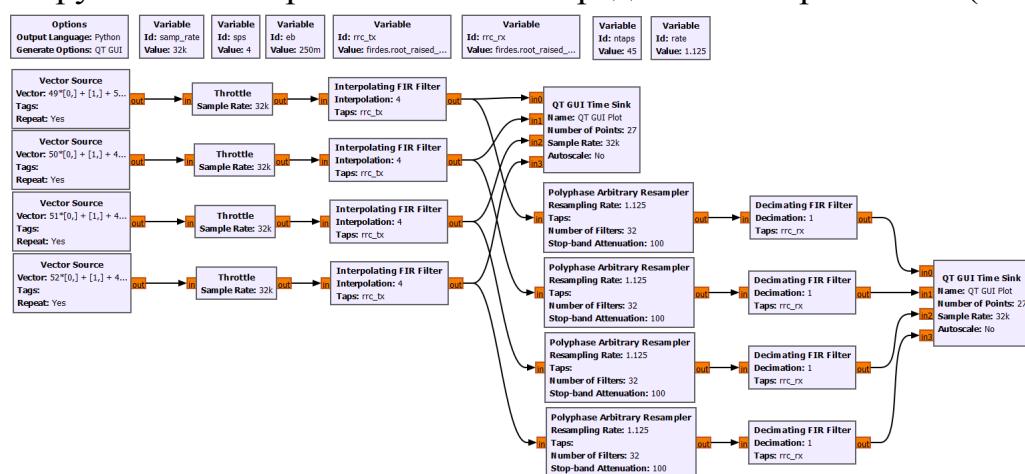


Рис. 12.9. Flowgraph разные часы

Все часы несовершенные, поэтому они (Рис.12.10):

1. начнут свою работу в разные моменты времени;
2. дрейфуют относительно других часов.

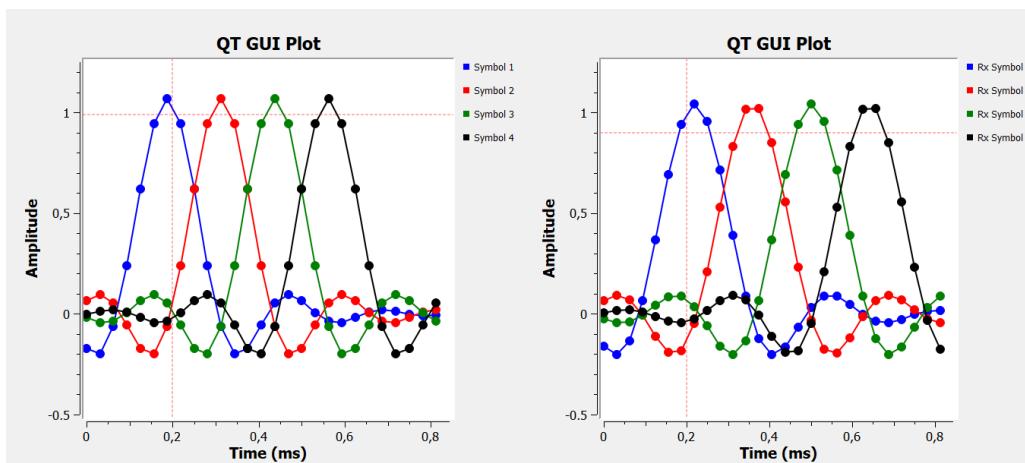


Рис. 12.10. Работа разных часов

Наша задача - синхронизировать часы передачи и приемника, используя только информацию в приемнике из входящих выборок. Это задача известна как восстановление часов или времени.

12.3.3. Блок синхронизации многофазных часов

Изучим как работает блок синхронизации многофазных часов перед его применением.

1. Один фильтр

Блок вычисляет первый дифференциал входящего сигнала, который будет связан с его смещением тактовой частоты. Используя пример `flowgraph symbol_differential_filter.grc` (Рис.12.11), мы можем увидеть графики с параметром скорости 1 (т.е. нет смещения часов).

Очевидно, что нам нужен образец 0,22 мс. Фильтр разности ($[-1, 0, 1]$) генерирует дифференциал символа, и, как показано на рис.12.12, выходной сигнал этого фильтра в правильной точке выборки равен 0. Затем мы можем инвертировать этот оператор и вместо этого сказать, когда выход дифференциального фильтра равен 0, мы нашли оптимальную точку выборки.

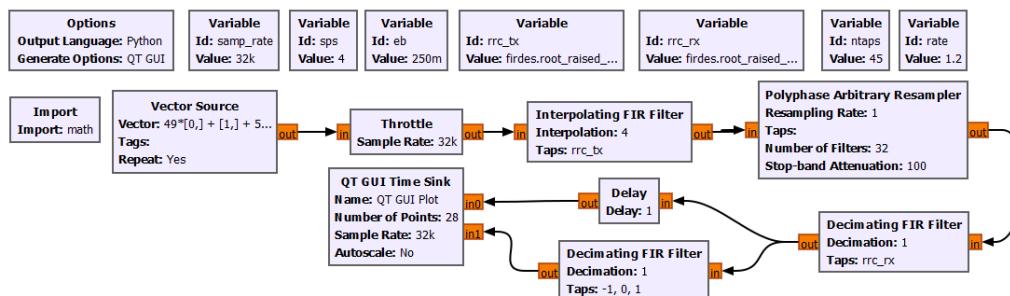


Рис. 12.11. Flowgraph с использованием блока

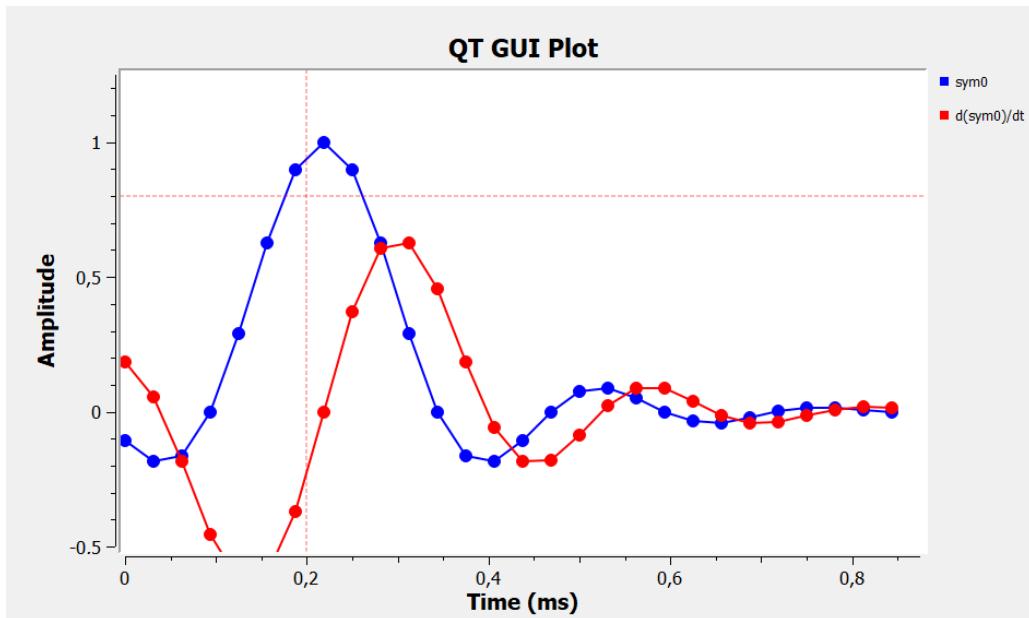


Рис. 12.12. Нет смещения часов

У нас есть временное смещение там, где пик символа выключен, а производный фильтр не показывает нам нулевую точку (Рис.12.13).

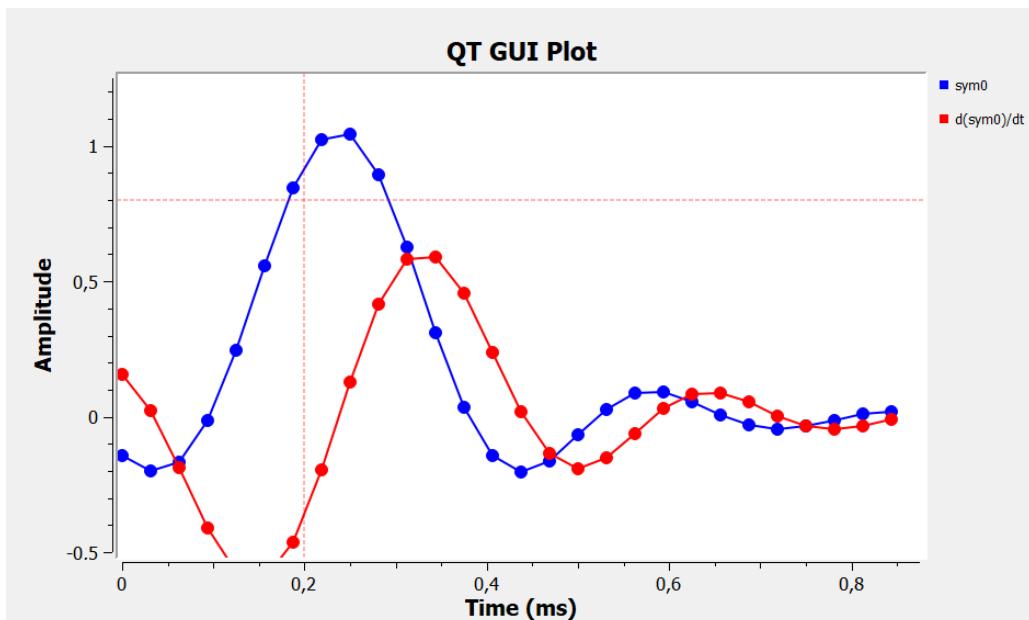


Рис. 12.13. Есть смещение часов

2. Несколько фильтров

Вместо использования одного фильтра мы можем создать серию фильтров, каждый с разной фазой. Если у нас достаточно фильтров на разных фазах, один из них имеет правильную фазу фильтра, которая даст нам желаемое значение синхронизации. Посмотрим на симуляцию, которая строит 5 фильтров, что означает 5 различных фаз (Рис.12.14).

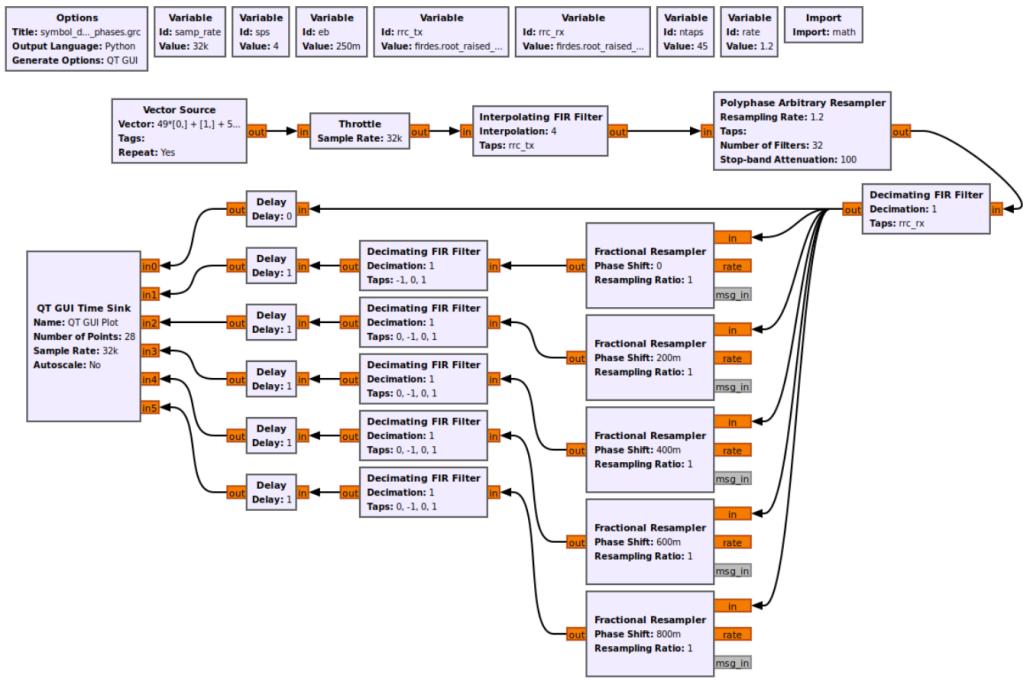


Рис. 12.14. Flowgraph с несколькими фазами

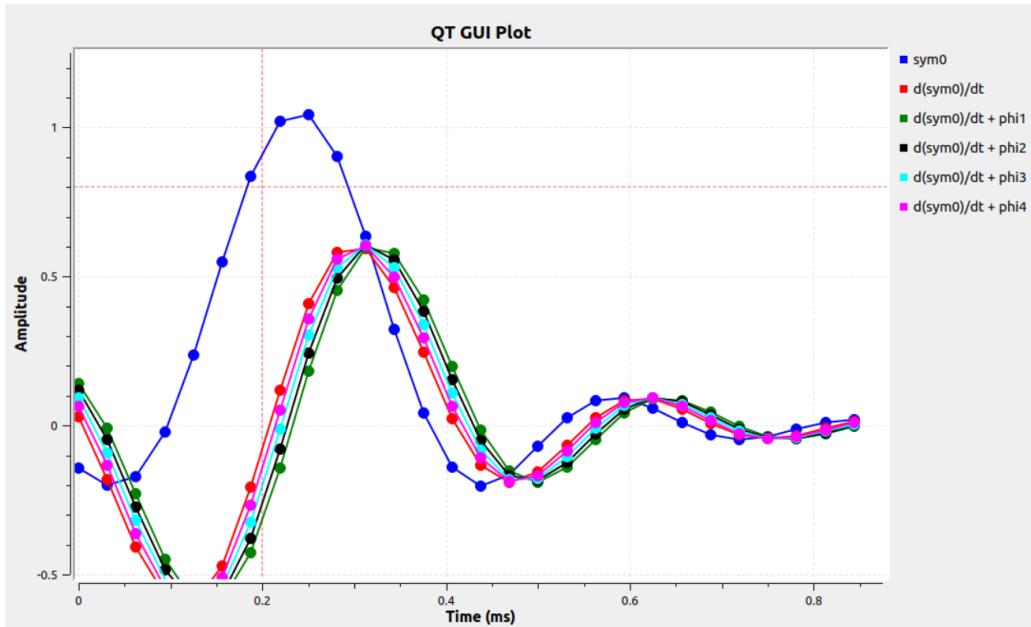


Рис. 12.15. Несколько фаз

На рис.12.15 мы можем видеть, что сигнал, помеченный как $d(sym0)/dt + \phi_3$, имеет точку отсчета в 0. Это говорит нам о том, что наша идеальная точка дискретизации находится при этом фазовом сдвиге. Следовательно, если мы возьмем RRC-фильтр нашего приемника и настроим его фазу на $\phi_3 = 3 * 2\pi/5$, то мы сможем исправить рассогласование по времени и выбрать идеальную точку дискретизации.

Итак, в данном примере мы использовали 5 фильтров в качестве идеальной точки выборки. Но этого недостаточно. Любое смещение выборки

между этими фазами по-прежнему приведет к несвоевременной выборке с добавленными ISI, как мы исследовали ранее.

Поэтому мы используем больше фильтров (32), чтобы получить максимальный коэффициент шума ISI, который меньше шума квантования 16-битного значения. То есть мы получим точность 16 бит. Для большей точности потребуется больше фильтров.

Затем мы используем контур управления 2-го порядка, который требуется, чтобы получить как правильную фазу фильтра, так и разницу в скорости между двумя тактовыми сигналами.

12.3.4. Использование блока многофазной синхронизации

Теперь мы используем этот блок в нашей симуляции. Пример flowgraph `mpsk_stage3.grc` принимает выходные данные модели канала и передает их через наш блок (Рис.12.16).

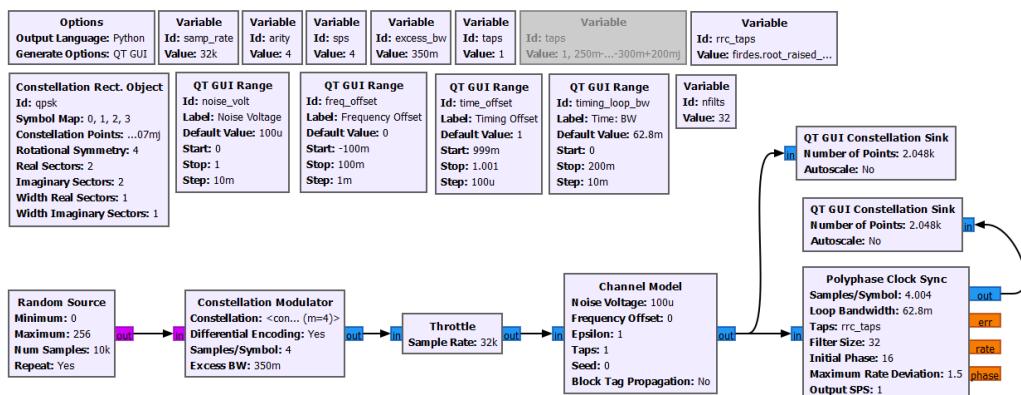


Рис. 12.16. Flowgraph с блоком многофазной синхронизации

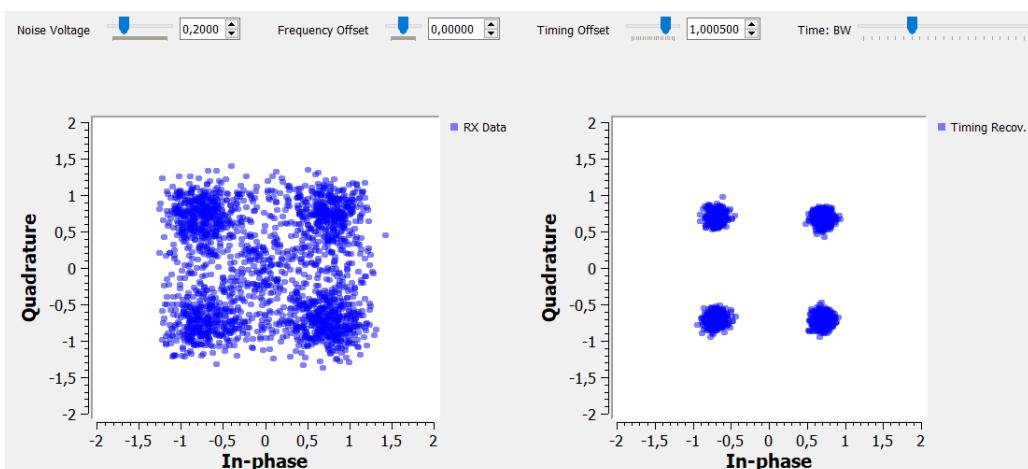


Рис. 12.17. Использование блока многофазной синхронизации

На рис.12.17 мы видим два созвездия: слева - полученный сигнал до восстановления синхронизации и справа - после восстановления синхронизации.

Мы можем изменять разные параметры наблюдать за изменением созвездия. Например, когда мы добавляем частотный сдвиг, мы видим, что созвездие становится кругом (Рис.12.18).

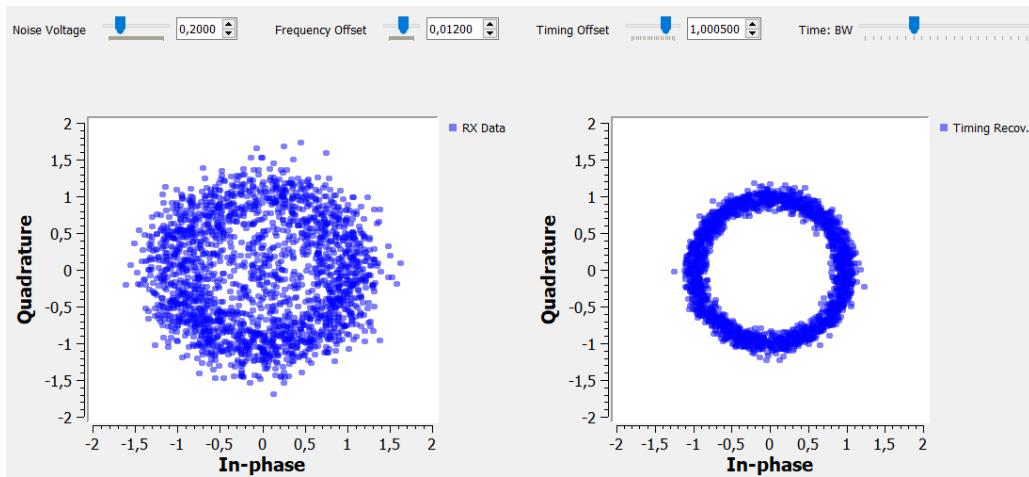


Рис. 12.18. Добавление частотного сдвига

12.4. Многолучевость

Сначала мы разберемся, что такое многолучевость. В большинстве коммуникационных сред у нас нет единственного пути для прохождения сигнала от передатчика к приемнику. Сигналы отражаются от различных поверхностей и приходят на приёмник в разное время в зависимости от длины пути. Их суммирование в приемнике вызывает искажения. Эти искажения вызывают внутрисимвольную и межсимвольную интерференции.

Нам нужно исправить это поведение, и мы можем сделать это, используя механизм, очень похожий на стереоэквалайзер. С помощью стереофонического эквалайзера мы можем изменить усиление определенных частот, чтобы либо подавить, либо усилить эти сигналы (Рис.12.19).

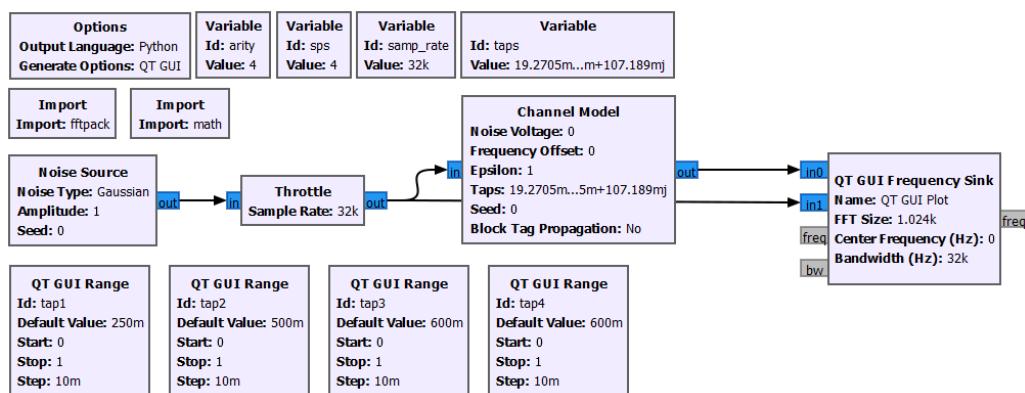


Рис. 12.19. Flowgraph многолучевость

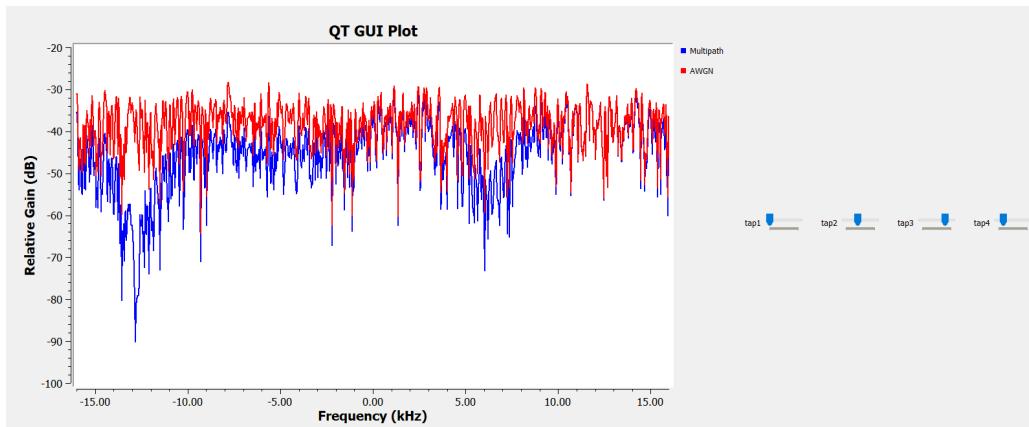


Рис. 12.20. Использование блока многофазной синхронизации

Как мы можем видеть на рис.12.20, многолучевой канал создает некоторые искажения в сигнале. Задача эквалайзера - отменить искажение, вызванное каналом, чтобы выходной сигнал эквалайзера был ровным. Но вместо того, чтобы настраивать каждый tap вручную, мы используем алгоритмы, которые обновляют эти taps за нас. Наша задача - использовать правильный алгоритм эквалайзера и настроить параметры.

Одним из важных параметров здесь является количество taps в эквалайзере. Как мы видим в нашем моделировании, пять taps дают довольно грубый контроль над частотной характеристикой. Чем больше taps, тем больше времени требуется как на вычисление ответвлений, так и на запуск эквалайзера против сигнала.

12.5. Эквалайзер

GNU Radio имеет два эквалайзера.

12.5.1. Эквалайзер СМА

СМА или алгоритм постоянного модуля - это слепой эквалайзер, который работает только с сигналами с постоянной амплитудой или модулем.

В примере `mpsk_stage4.grc` мы используем алгоритм СМА с 11 taps (Рис.12.20). Изменим параметры и посмотрим, как это влияет на производительность как с точки зрения вычислений, так и с точки зрения сигналов.

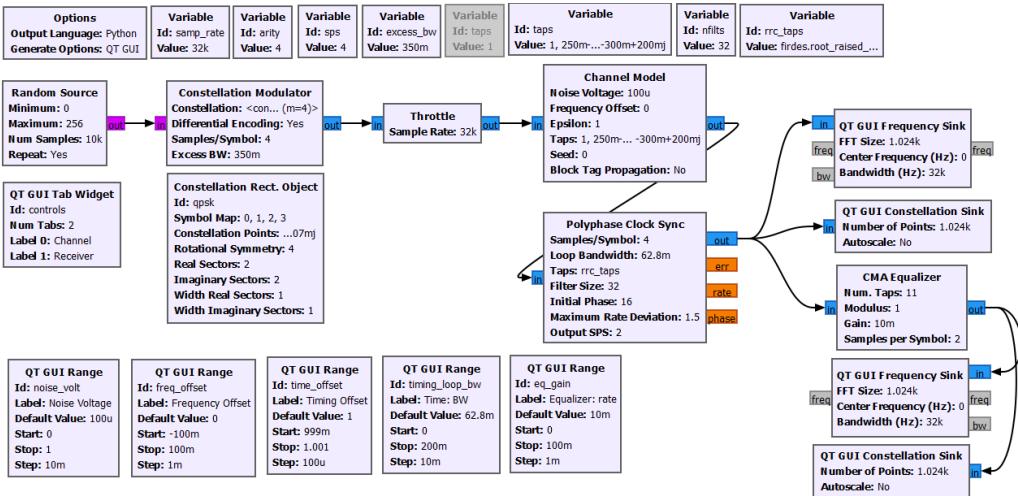


Рис. 12.21. Flowgraph с эквалайзером СМА

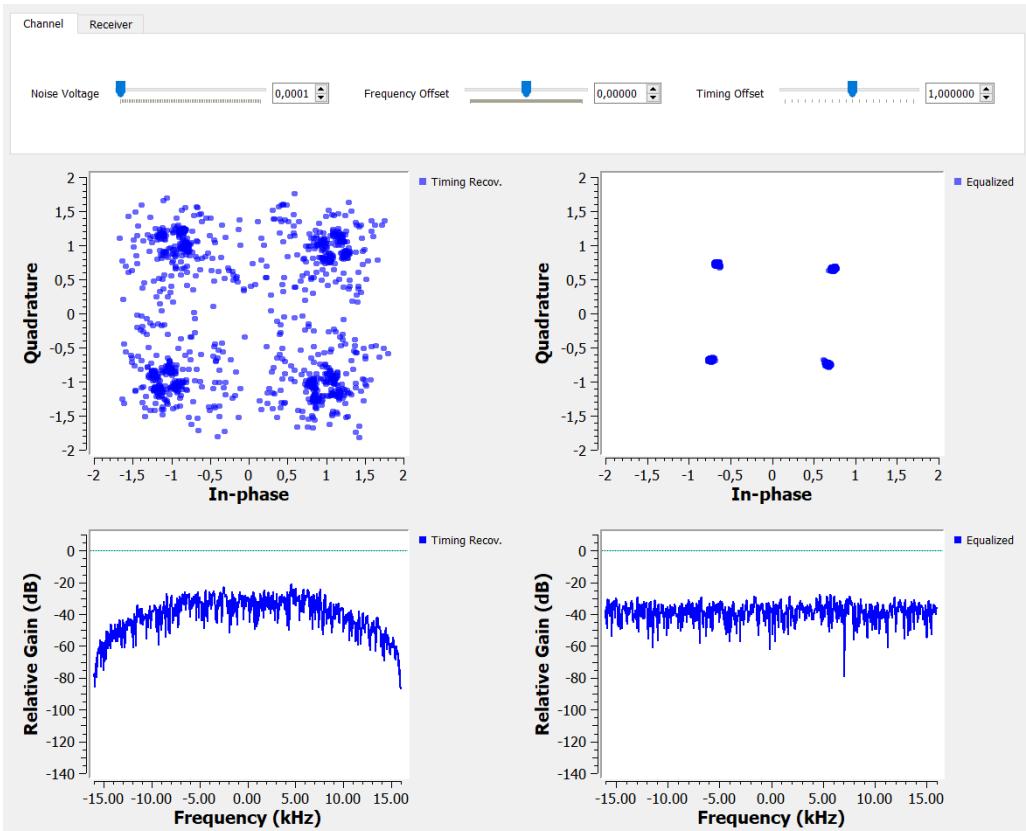


Рис. 12.22. Использование эквалайзера СМА

На рис.12.21 мы можем увидеть эффект синхронизированного по времени многолучевого сигнала до и после эквалайзера. До эквалайзера у нас очень некрасивый сигнал даже без шумов. Эквалайзер понимает, как инвертировать и сократить канал, чтобы у нас снова был хороший, чистый сигнал. Мы также можем видеть сам канал и то, как он красиво выравнивается после эквалайзера.

12.5.2. Эквалайзер LMS DD

Хорошей практикой является использование блока эквалайзера LMS DD.

LMS или алгоритм наименьших квадратов требует знания принимаемого сигнала. Эквалайзеру необходимо знать точки созвездия для корректировки, и он использует решения о выборках, чтобы сообщить, как обновлять ответвления для эквалайзера.

Заменим эквалайзер CMA на LMS DD (Рис.12.23).

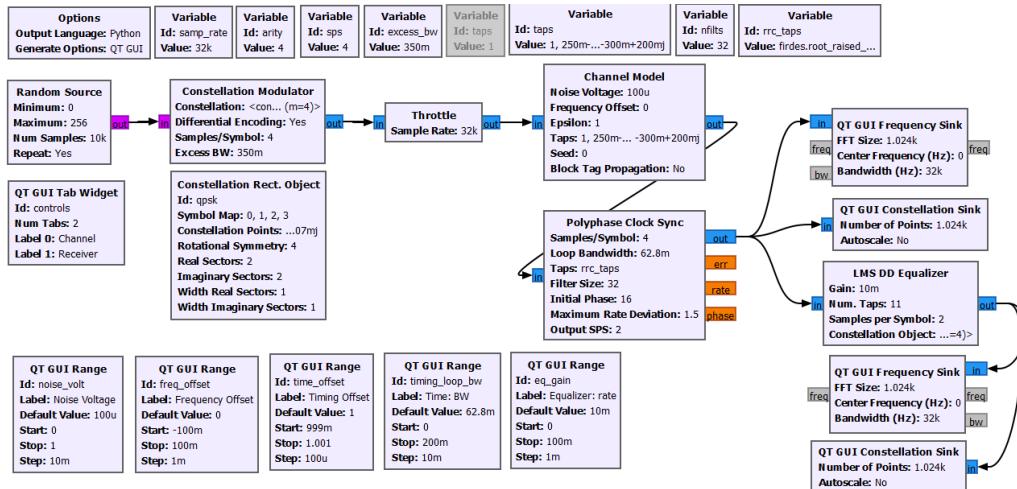


Рис. 12.23. Flowgraph с эквалайзером LMS DD

Результат работы этого эквалайзера мы можем видеть на рис.12.24.

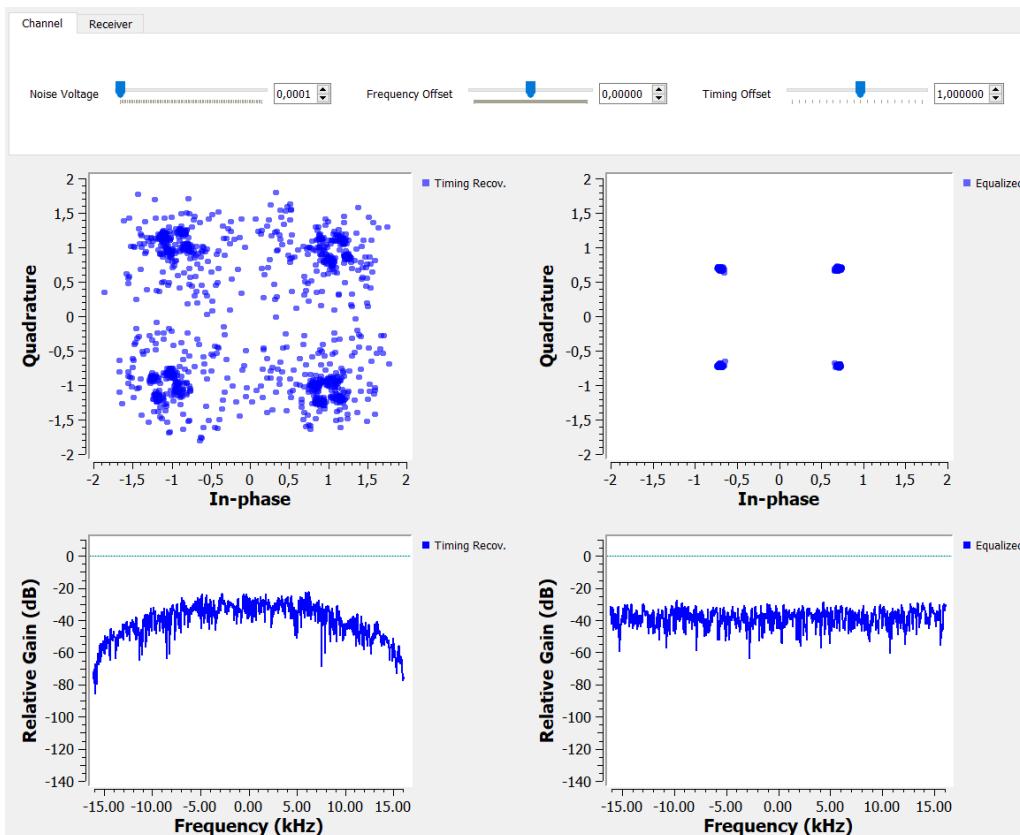


Рис. 12.24. Использование эквалайзера LMS DD

12.6. Фазовая и точная частотная коррекция

Мы выровняли канал, но у нас все еще есть проблема смещения фазы и частоты. Она выходит за пределами возможностей эквалайзера. Поэтому нам нужно исправить любой сдвиг фазы, а также любой сдвиг частоты.

На этом этапе мы будем использовать цикл второго порядка, чтобы мы могли отслеживать фазу и частоту. Тип восстановления, который мы здесь рассматриваем, предполагает, что мы выполняем точную частотную коррекцию. Поэтому мы должны находиться в приличном диапазоне идеальной частоты, чтобы цикл сошёлся.

Для этой задачи мы собираемся использовать цикл Костаса в примере `mpsk_stage5.grc` (Рис.12.24).

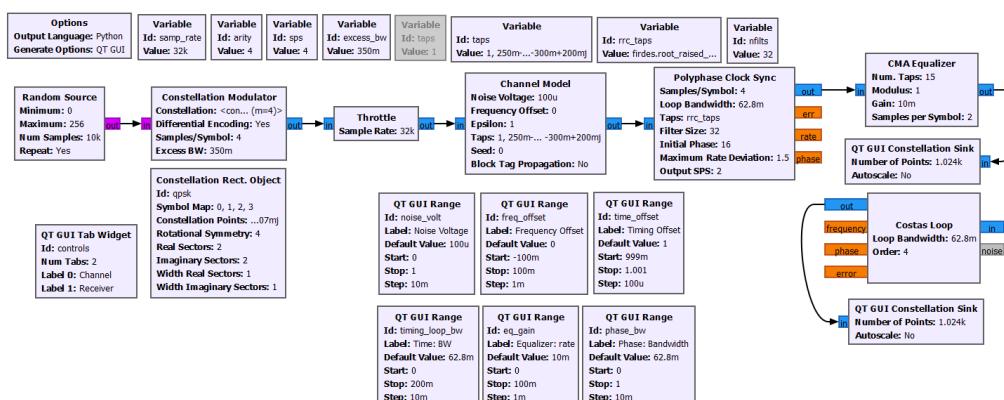


Рис. 12.25. Flowgraph для исправления сдвига фазы и частоты

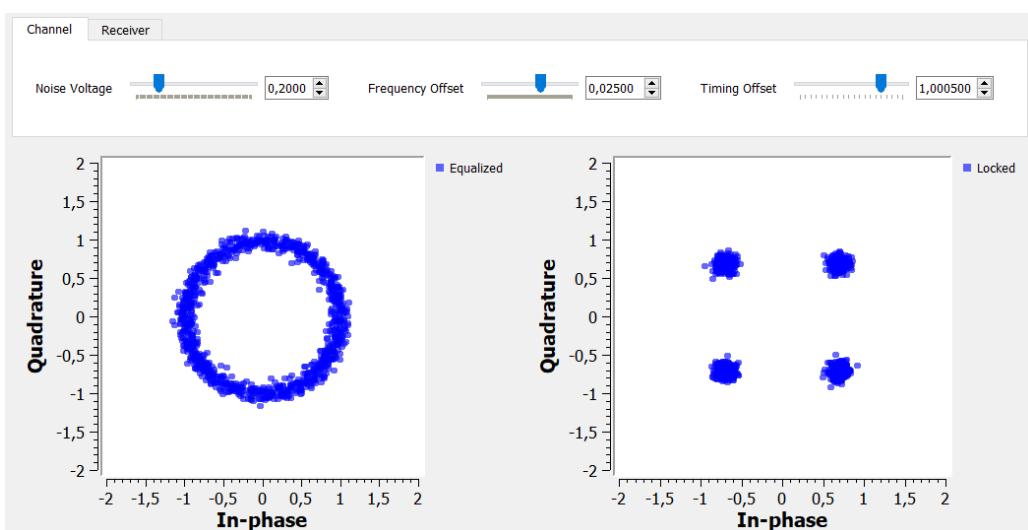


Рис. 12.26. Исправление сдвига фазы и частоты

На рис.12.25 мы установили шум, временной сдвиг, простой многолучевой канал и частотный сдвиг. После эквалайзера мы видим, что все символы находятся на единичном круге, но врачаются из-за смещения частоты, которое еще не исправлено. На выходе блока цикла Костаса мы можем видеть заблокированное созвездие и дополнительный шум.

12.7. Расшифровка

Теперь мы можем декодировать сигнал. Используя пример flowgraph `mpsk_stage6.grc` (Рис.12.26), мы вставляем Constellation Decoder после цикла Костаса, но наша работа еще не завершена. Мы передаём 4 дифференциальных символа, но мы не можем быть уверены, что у нас есть такое же отображение символов на точки созвездия, которое мы делали при передаче.

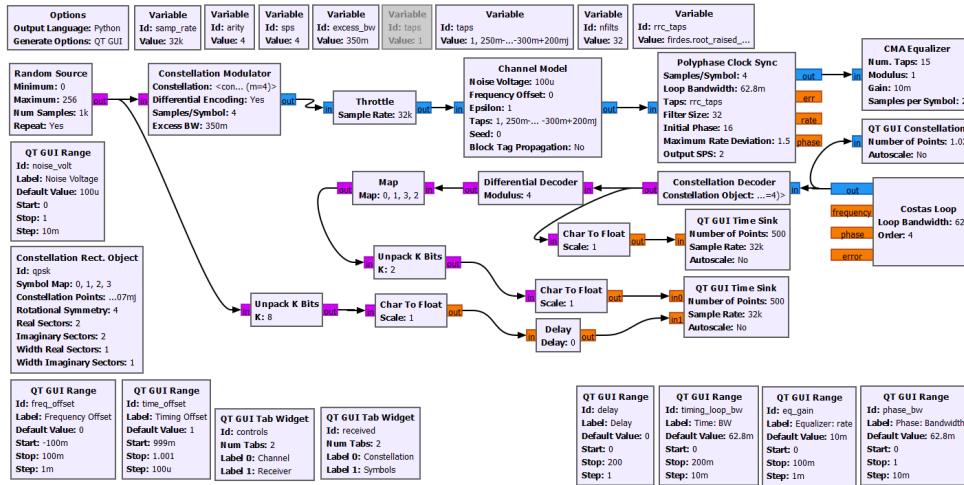


Рис. 12.27. Flowgraph декодирования

Flowgraph использует блок дифференциального декодера для преобразования кодированных дифференциальным кодом символов обратно в их исходные символы. Мы используем блок Map для преобразования символов из дифференциального декодера в исходные символы, которые мы передали. На данный момент у нас теперь есть исходные символы от 0 до 3, поэтому мы распакуем эти 2 бита на символ в биты, используя блок unpack bits. Теперь у нас есть исходный битовый поток данных.

Но как мы узнаем, что это исходный битовый поток? Для этого нам нужно сравнить переданные данные с входным битовым потоком. Однако прямое сравнение ничего не даст. Так как в цепочке приемника много блоков и фильтров, которые задерживают сигнал, принятый сигнал отстает на некоторое количество бит. Чтобы это исправить, необходимо добавить задержку Delay (Рис.12.27).

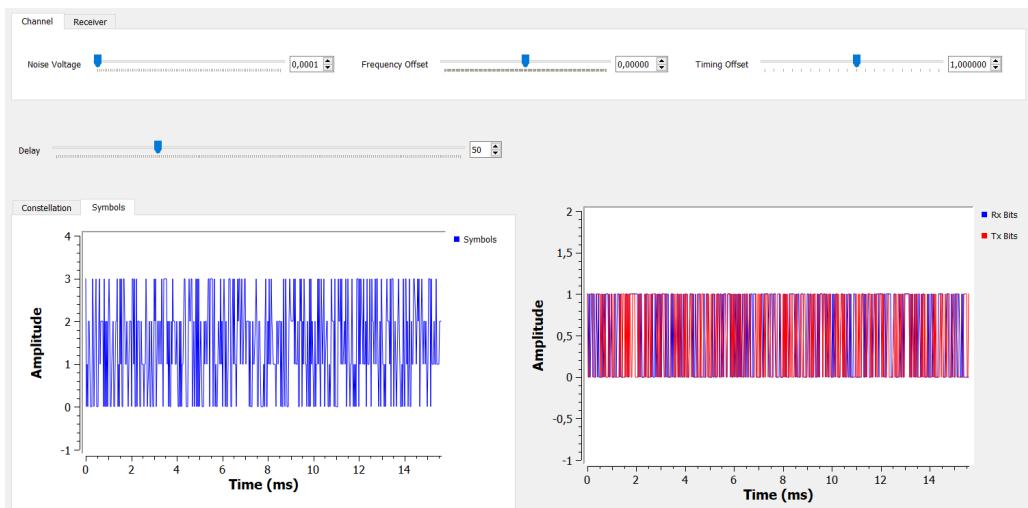


Рис. 12.28. Сравнение данных

12.8. Выводы

В результате выполнения данной работы мы изучили основные этапы, необходимые для восстановления сигналов, а также научились строить PSK модулятор/демодулятор для работы с сигналом.