

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа компьютерных технологий и информационных систем

Отчёт по лабораторным работам

Дисциплина: Телекоммуникационные технологии.

Выполнил студент гр. 5130901/10101 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ Н.В. Богач
(подпись)

“07” февраля 2024 г.

Санкт-Петербург

2024

Оглавление

1. Лабораторная работа 1. Сигналы и звуки.....	2
1.1. Упражнение 1.2.....	2
1.2. Упражнение 1.3.....	5
1.3. Упражнение 1.4.....	8
2. Лабораторная работа 2. Гармоники.	9
2.1. Упражнение 2.2.....	9
2.2. Упражнение 2.3.....	13
2.3. Упражнение 2.4.....	14
2.4. Упражнение 2.5.....	16
2.5. Упражнение 2.6.....	17
3. Лабораторная работа 3. Аперiodические сигналы.	19
3.1. Упражнение 3.1.....	19
3.2. Упражнение 3.2.....	20
3.3. Упражнение 3.3.....	22
3.4. Упражнение 3.4.....	22
3.5. Упражнение 3.5.....	23
3.6. Упражнение 3.6.....	24
4. Приложение:	27

1. Лабораторная работа 1. Сигналы и звуки.

1.1. Упражнение 1.2.

Скачаем с сайта <https://freesound.org/> образец звука и различными способами исследуем его. Для удобной работы с сигналами здесь, и в дальнейших работах будем использовать библиотеку `thinkdsp`.

Откроем скачанный файл, нормализуем и выведем на экран. Код будет выглядеть следующим образом:

```
1 from thinkdsp import read_wave
2
3 wave = read_wave('680840__seth_makes_sounds__homemade.wav')
4 wave.normalize()
5 wave.make_audio()
6 wave.plot()
```

Результат выполнения кода выглядит следующим образом:

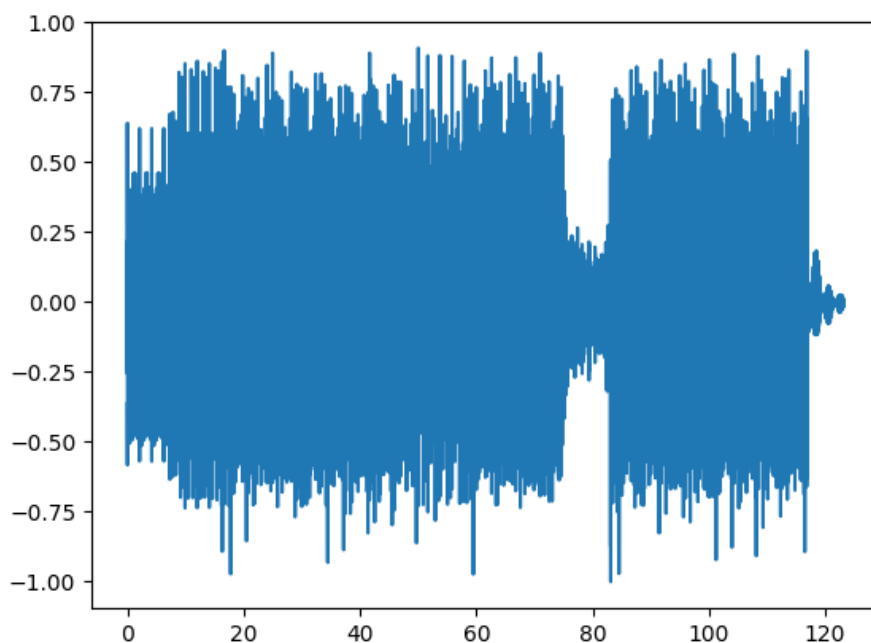


Рис. 1.1. Спектрограмма аудио файла.

Данный отрезок слишком длинный, выделим из него отрезок длиной пол секунды, начиная с 40 секунды аудио файла. Выведем полученный сегмент на экран, используя следующий код:

```
1 segmet = wave.segment(start=40.0, duration=0.5)
2 segmet.make_audio()
3 segmet.plot()
```

Спектрограмма заданного сегмента выглядит следующим образом:

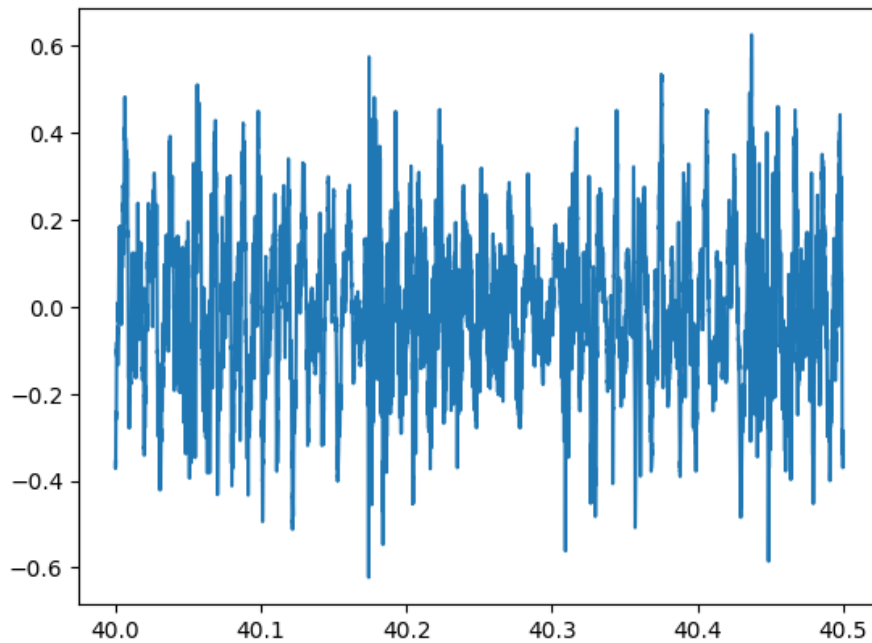


Рис. 1.2. Спектрограмма аудио файла с 40.0 по 40.5 секунды.

Разложим полученный отрезок в спектр и выведем на экран. Код будет выглядеть следующим образом:

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

Этот код выведет спектр до 5000 частоты т.к. далее частоты равны примерно нулю:

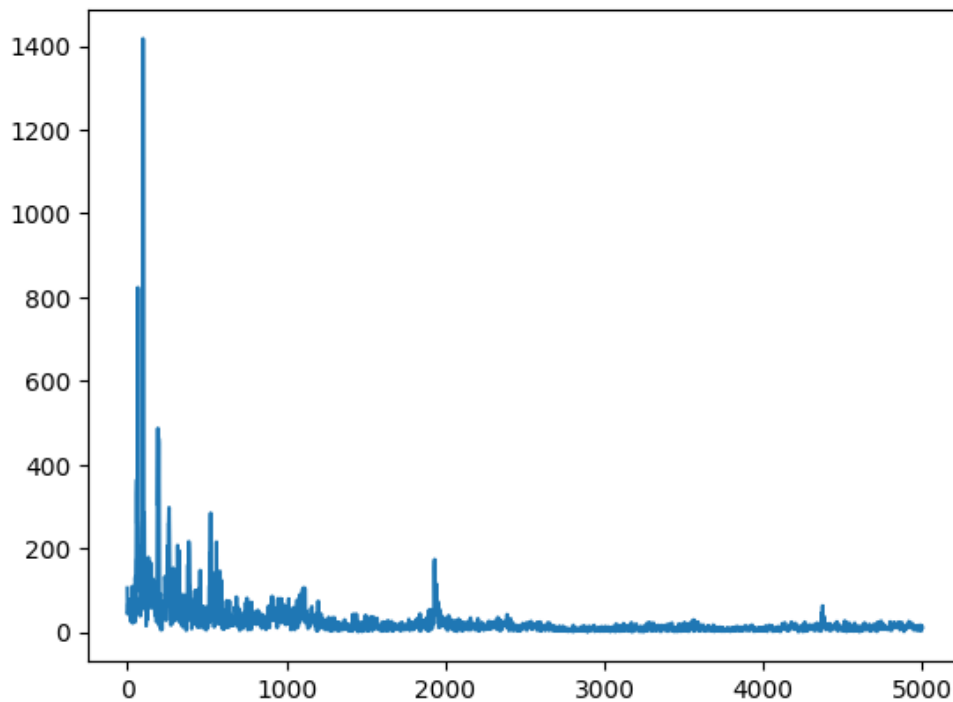


Рис. 1.3. Результат разложения сегмента в спектр.

Доминантной частотой в этом отрывке является 98 Гц.

Теперь поэкспериментируем с функциями `high_pass`, `low_pass` и `band_stop`, которые фильтруют гармоники.

Начнем с `low_pass`:

```
1 spectrum.make_wave().make_audio()  
2 spectrum.low_pass(2000)  
3 spectrum.plot(high=5000)  
4 spectrum.make_wave().make_audio()
```

Данный код сохраняет музыкальный фрагмент (для дальнейшего сравнения), после чего применяет функцию `low_pass` и выводит его спектр на экран, а также опять сохраняет фрагмент. Полученный спектр выглядит следующим образом:

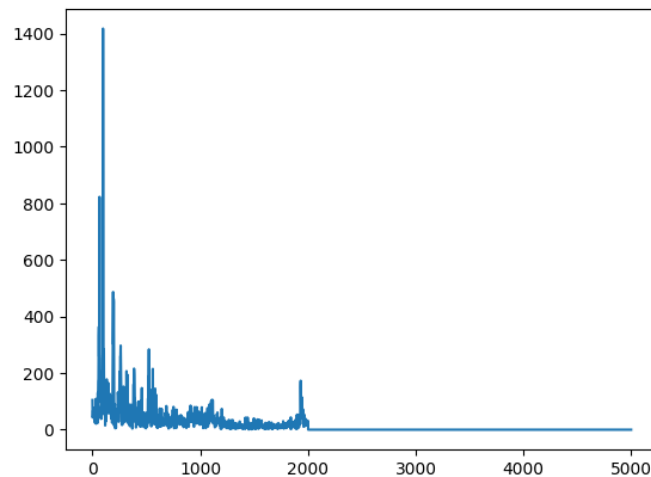


Рис. 1.4. Спектр фрагмента после применения `low_pass`.

Как видно из рисунка выше, данная функция полностью убрала частоты, выше 2000. Таким образом звук стал более «глухим» и «отдаленным».

Теперь к исходному сегменту применим метод `high_pass`:

```
1 spectrum.make_wave().make_audio()  
2 spectrum.high_pass(1000)  
3 spectrum.plot(high=5000)  
4 spectrum.make_wave().make_audio()
```

Полученный спектр имеет следующий вид:

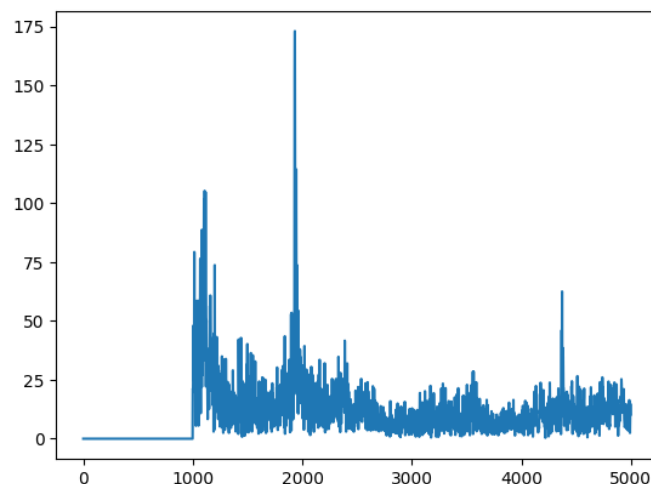


Рис. 1.5. Спектр фрагмента после применения `high_pass`.

Как видно по спектру, эта функция убирает все частоты ниже заданной. Таким образом звук сильно поменял свое звучание, став более шипящим и менее глубоким.

И последняя функция `band_stop`:

```
1 spectrum.make_wave().make_audio()  
2 spectrum.band_stop(low_cutoff=100, high_cutoff=1000)  
3 spectrum.plot(high=5000)  
4 spectrum.make_wave().make_audio()
```

Полученный спектр выглядит следующим образом:

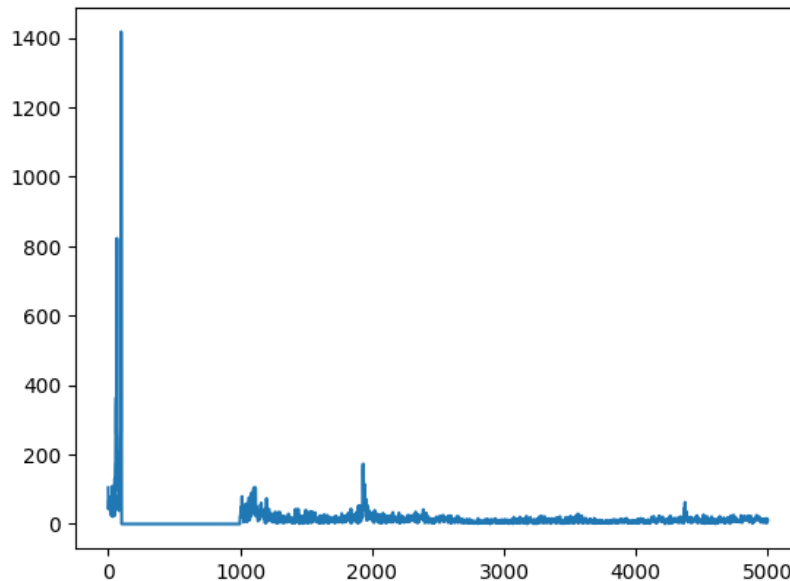


Рис. 1.6. Спектр фрагмента после применения `band_stop`.

Как мы видим, данная функция убирает частоты из заданного диапазона. Звук фрагмента при удалении частот со 100 Гц до 1000 Гц сильно изменился, в нем практически не слышны ударные.

1.2. Упражнение 1.3.

Создадим сигнал, состоящий из синусов, разной частоты, однако кратных одному числу, например 200:

```
1 from thinkdsp import SinSignal  
2  
3 signal = (SinSignal(freq=400, amp=1.0) +  
4           SinSignal(freq=600, amp=1.0) +  
5           SinSignal(freq=800, amp=1.0))  
6 signal.plot()
```

Полученный сигнал имеет следующий вид:

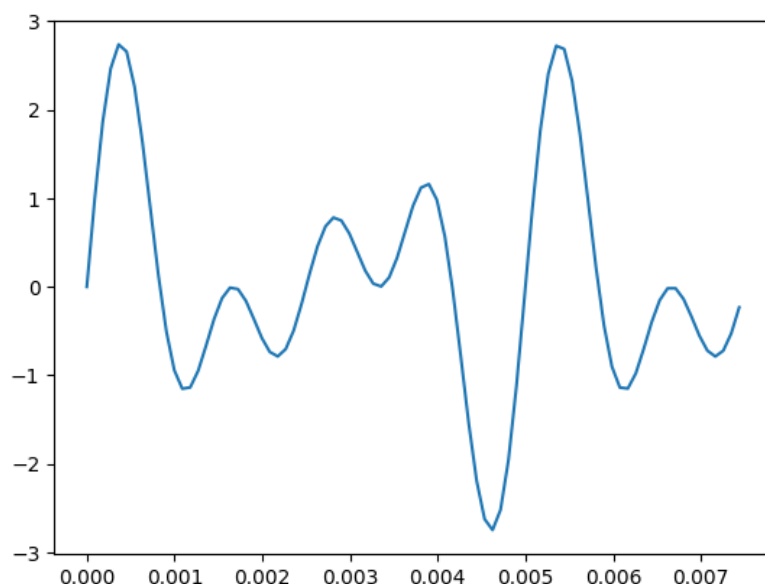


Рис. 1.7. Сигнал, полученный суммой синусов разной частоты.

Создадим файл для прослушивания этого звука, длиной 1 секунда:

```
1 wave = signal.make_wave(duration=1)
2 wave.apodize()
3 wave.make_audio()
```

Полученный звуковой файл является однотонным писком, похожим на звук гудка, но монотонного.

Выведем спектр полученного сигнала:

```
1 spectrum = wave.make_spectrum()
2 spectrum.plot(high=2000)
```

Результат выглядит следующим образом:

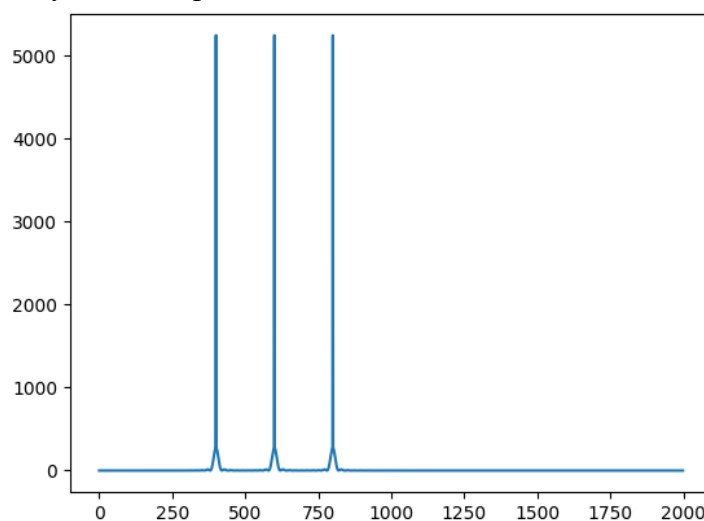


Рис. 1.8. Спектр сигнала, полученного суммой синусов разной частоты.

Как видим, спектр полностью соответствует ожидания, на нем пики находятся именно в тех частотах, которые мы указывали при создании.

Теперь изменим наш сигнал, добавив частоту, не кратную 200:



```
1 signal += SinSignal(freq=450, amp=1.0)
2 signal.plot()
3 signal.make_wave().make_audio()
```

Полученный сигнал имеет следующий вид:

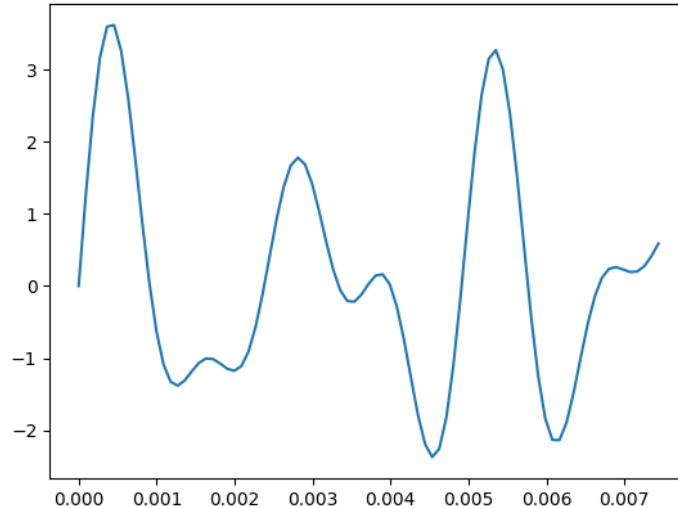


Рис. 1.9. Сигнал, после добавления синуса не кратной частоты.

Полученный сигнал сильно отличается от того, который был ранее. Так же аудио файл тоже чуть-чуть отличается. В монотонном звуке гудка различим какой-то посторонний периодический сигнал.

Выведем спектр полученного сигнала:



```
1 wave = signal.make_wave(duration=1)
2 wave.apodize()
3 spectrum = wave.make_spectrum()
4 spectrum.plot(high=2000)
```

Полученный спектр имеет следующий вид:

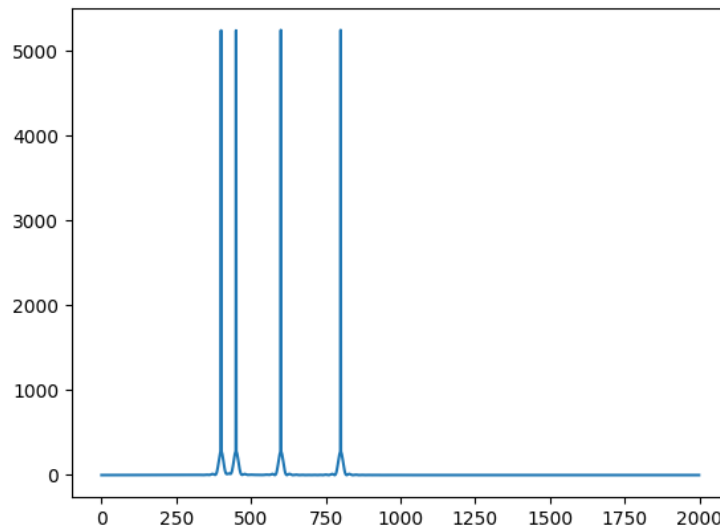


Рис. 1.10. Спектр сигнала, после добавления синуса не кратной частоты.

Как и ожидалось, в спектре появился добавленный ранее сигнал.

1.3. Упражнение 1.4.

Напишем функцию для ускорения и замедления аудио. Для начала прочитаем аудио фрагмент и выведем его на экран:

```
1 from thinkdsp import read_wave
2
3 wave = read_wave('680840__seth_makes_sounds__homemade.wav')
4 wave.normalize()
5 wave.plot()
6 wave.make_audio()
```

Спектрограмма будет выглядеть следующим образом:

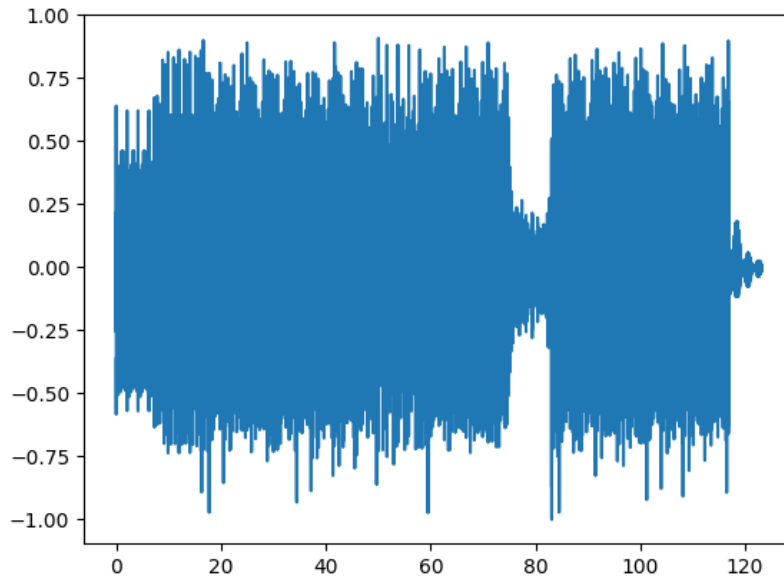


Рис. 1.11. Спектрограмма аудио файла.

Функция для ускорения будет выглядеть следующим образом:

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

Она изменяет ts (которое используется для корректного отображения временной шкалы в plot) и framerate, что, собственно, и ускоряет произведение.

Передадим функции значение 0.5, что эквивалентно ускорению в 2 раза:

```
1 stretch(wave, 0.5)
2 wave.plot()
3 wave.make_audio()
```

После выполнения мы получили аудио файл, который ускорен в 2 раза, как и ожидалось. Посмотрим на полученную спектрограмму:

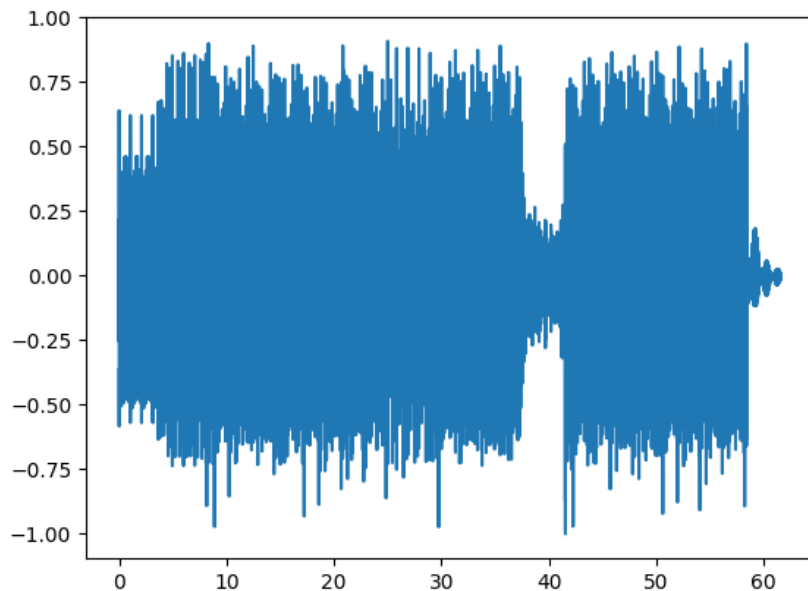


Рис. 1.12. Спектрограмма аудио файла после ускорения.

Как мы видим, полученная спектрограмма не отличается от исходной ничем, кроме длительности аудио фрагмента, он меньше в 2 раза.

2. Лабораторная работа 2. Гармоники.

2.1. Упражнение 2.2.

Разработаем класс, который бы наследовался от Sinusoid из thinkdsp, который позволял бы строить пилообразный сигнал (нарастает от -1 до 1, а затем резко падает до -1). Переопределить необходимо только функцию evaluate:

```
1 from thinkdsp import Sinusoid
2 from thinkdsp import normalize, unbias
3 import numpy as np
4
5 class SawtoothSignal(Sinusoid):
6
7     def evaluate(self, ts):
8         cycles = self.freq * ts + self.offset / np.pi / 2
9         frac, _ = np.modf(cycles)
10        ys = normalize(unbias(frac), self.amp)
11        return ys
```

Здесь:

cycles – число циклов со времени старта.

frac – дробная часть, растущая от 0 до 1 за период.

unbias – сдвигает frac так, что он растёт от -0.5 до 0.5.

normalize – нормализует функцию, чтоб она росла от $-self.amp$ до $self.amp$.

Создадим экземпляр этого класса и сразу же получим из него Wave, после чего выведем его часть на экран, дабы проверить, что функция реализованная корректно:

```
1 sawtooth = SawtoothSignal(200).make_wave(duration=0.5, framerate=40000)
2 sawtooth.segment(start=0, duration=0.005 * 4).plot()
```

Результат запуска выглядит следующим образом:

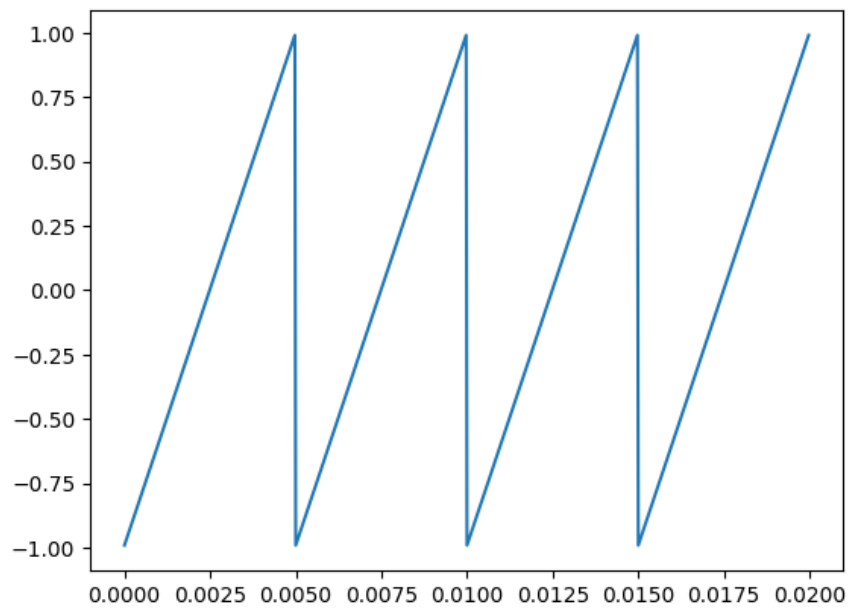


Рис. 2.1. Пилообразный сигнал.

Создадим спектр этого сигнала, выведем на экран, а также посмотрим наибольшие 10 пиков, дабы изучить каким образом частота зависит от амплитуды:

```

1 spectrum = sawtooth.make_spectrum()
2 spectrum.plot(high=10000)
3 spectrum.peaks()[:10]

```

График будет выглядеть таким образом:

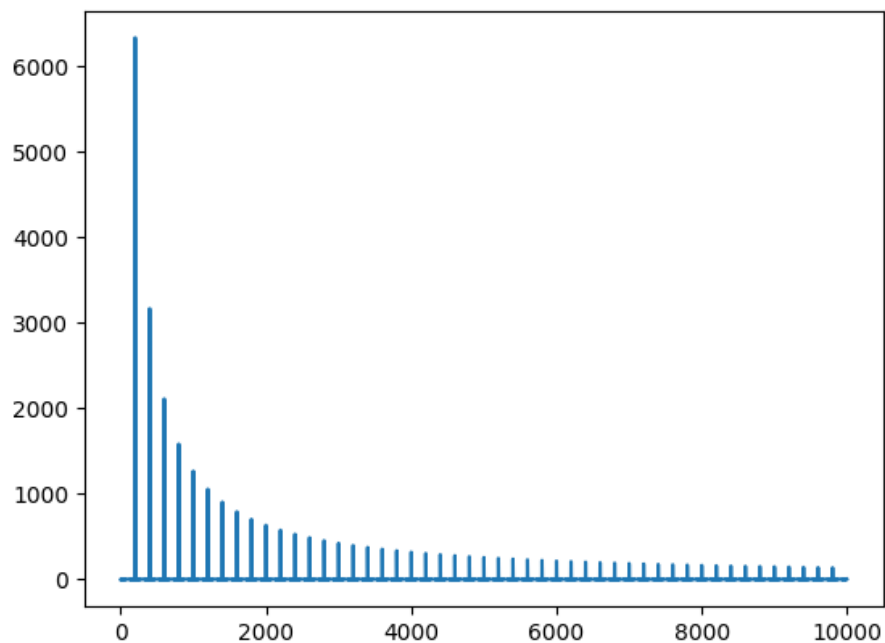


Рис. 2.2. Спектр пилообразного сигнала.

А также мы получаем следующий массив пиков:

```
[(6336.586158412468, 200.0),
 (3168.547531644226, 400.0),
 (2112.647887262727, 600.0),
 (1584.783147437211, 800.0),
 (1268.132547579567, 1000.0),
 (1057.089208734752, 1200.0),
 (906.3930765164864, 1400.0),
 (793.4141558611690, 1600.0),
 (705.5802559636873, 1800.0),
 (635.3480882678591, 2000.0)]
```

Как можно заметить, сигнал содержит как четные, так и нечетные гармоники, а также они уменьшаются пропорционально $1/f$.

Сравним полученный спектр пилообразного сигнала с прямоугольным:

```
1 from thinkdsp import SquareSignal
2
3 sawtooth.make_spectrum().plot(high=10000, color='gray')
4 square = SquareSignal(freq=200, amp=0.5).make_wave(duration=0.5, framerate=40000)
5 sqere_spectrum = square.make_spectrum()
6 sqere_spectrum.plot(high=10000)
7 sqere_spectrum.peaks()[:10]
```

Стоит отметить, что прямоугольный сигнал создается с $amp=0.5$, чтоб выровнять спектрограмму для сравнения её с пилообразным сигналом.

Полученный график выглядит так:

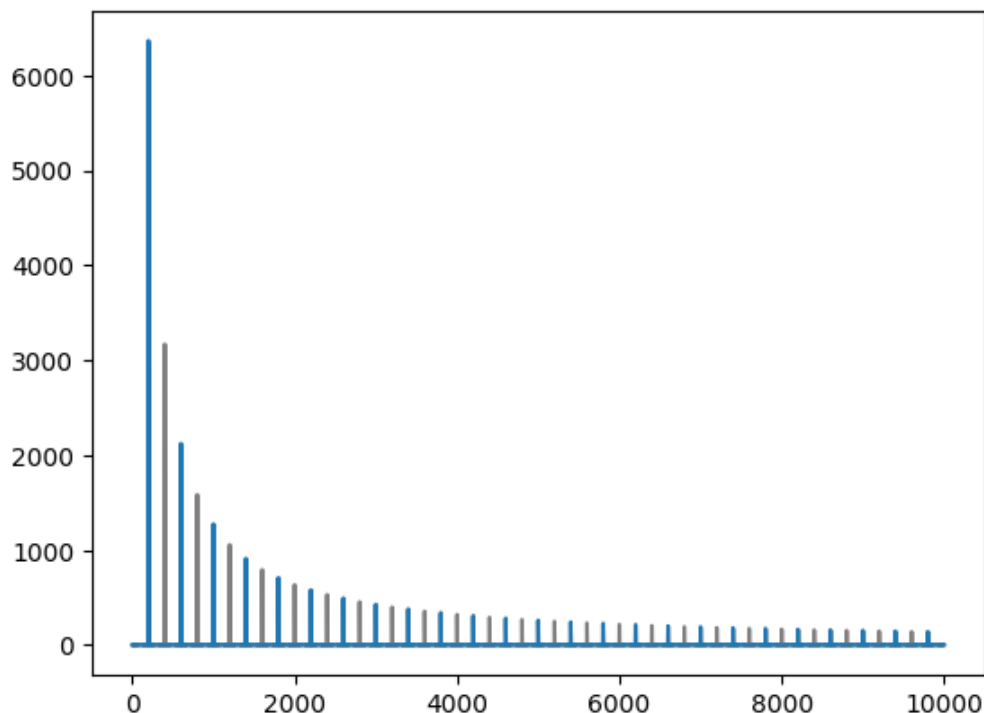


Рис. 2.3. Спектрограммы пилообразного и прямоугольного сигналов.

Для удобства сравнения так же проанализируем первые 10 пиков прямоугольного сигнала:

```
[(6366.41311530820, 200.0),
 (2122.71230545574, 600.0),
 (1274.31761659952, 1000.0),
 (910.967679013610, 1400.0),
 (709.300517302879, 1800.0),
 (581.126830719483, 2200.0),
 (492.527938505399, 2600.0),
 (427.675369582061, 3000.0),
 (378.189565961787, 3400.0),
 (339.219405574489, 3800.0)]
```

Стоит обратить внимание, что в отличие от пилообразного сигнала, прямоугольный сигнал имеет только нечетные гармоники, а вот зависимость падения от частоты сохраняется и пропорционально $1/f$.

Так же выполним аналогичное сравнение с треугольным сигналом:

```
1 from thinkdsp import TriangleSignal
2
3 sawtooth.make_spectrum().plot(high=10000, color='gray')
4 triangle = TriangleSignal(freq=200, amp=0.78).make_wave(duration=0.5, framerate=40000)
5 triangle_spectrum = triangle.make_spectrum()
6 triangle_spectrum.plot(high=10000)
7 triangle_spectrum.peaks()[10]
```

Аналогично прямоугольному сигналу необходимо изменить *amp*, однако для треугольного сигнала это значение равно 0.78.

График выглядит следующим образом:

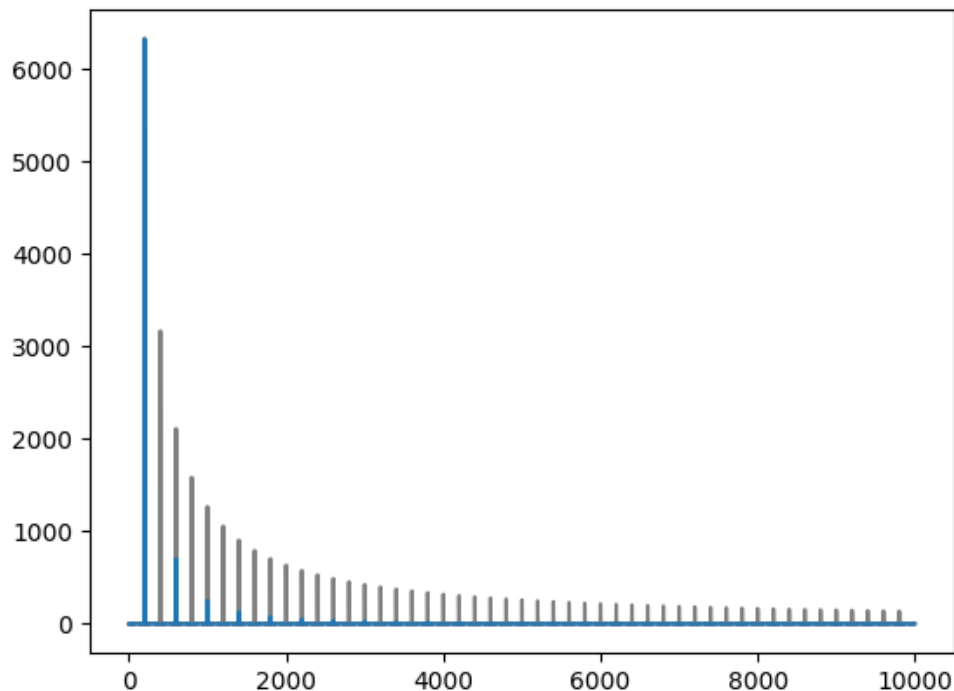


Рис. 2.4. Спектрограммы пилообразного и треугольного сигналов.

Для удобства сравнения так же проанализируем первые 10 пиков треугольного сигнала:

```
[(6322.961884943854, 200.0),
 (703.0137709503831, 600.0),
 (253.4183165242379, 1000.0),
 (129.5506855043607, 1400.0),
 (78.57692291999746, 1800.0),
 (52.77470536760347, 2200.0),
 (37.93526415250758, 2600.0),
 (28.62556659255658, 3000.0),
 (22.40446225716876, 3400.0),
 (18.04308559845682, 3800.0)]
```

Как мы видим, этот сигнал ведет себя совершенно отлично, от пилообразного. В первую очередь мы видим, что в нем присутствуют только нечетные гармоники, а также зависимость падения от частоты пропорциональна $1/f^2$.

2.2. Упражнение 2.3.

Создадим прямоугольный сигнал с частотой 1100 Гц и выборкой 10000 кадров в секунду. Отобразим получившийся спектр, а также первые 10 пиков:

```
1 from thinkdsp import SquareSignal
2
3 square = SquareSignal(1100).make_wave(duration=0.5, framerate=10000)
4 square_spectrum = square.make_spectrum()
5 square_spectrum.plot()
6 square_spectrum.peaks()[:10]
```

Получившийся спектр выглядит таким образом:

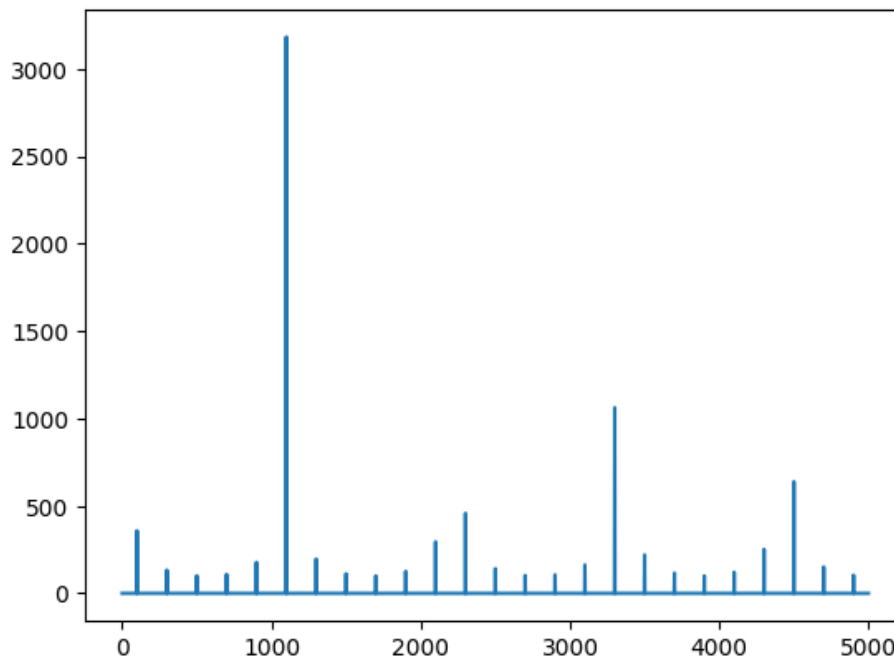


Рис. 2.5. Спектрограмма прямоугольного сигнала с частотой 1100 Гц и выборкой 10000 кадров в секунду.

А также для удобства анализа приведем старшие 10 пиков:

```
[(3183.622520909762, 1100.0),  
(1062.605379628311, 3300.0),  
(639.2453221499661, 4500.0),  
(458.4143857027373, 2300.0),  
(358.4343652372162, 100.0),  
(295.2134792809340, 2100.0),  
(251.7953698310349, 4300.0),  
(220.2689264585266, 3500.0),  
(196.4476698867248, 1300.0),  
(177.9095485479867, 900.0)]
```

Как мы помним, прямоугольный сигнал имеет только нечетные гармоники, а зависимость падения амплитуды от частоты пропорциональна $1/f$.

Ожидается, что гармоники будут на 3300, 5500, 7700 и 9900 Гц. Как мы видим, пики есть на 1100 и 3300 Гц, однако дальше наблюдается эффект биения, поэтому вместо 5500 мы получаем лишь 4500 ($10000 - 5500$), а следующая гармоника вместо 7700 получается 2300 ($10000 - 7700$). Из-за этого сигнал звучит совершенно по-другому, а именно появляются лишние низкие частоты (например, 100), а также посторонние не кратные частоты (2300), которые сильно выбиваются.

Продemonстрируем это, создав аудио файл, исходного прямоугольного сигнала, а также две синусоиды на 2300 и 100 Гц и убедимся, что они достаточно заметны:

```
1 square.make_audio()  
2  
3 from thinkdsp import SinSignal  
4  
5 SinSignal(2300).make_wave(duration=0.5, framerate=10000).make_audio()  
6 SinSignal(100).make_wave(duration=0.5, framerate=10000).make_audio()
```

При прослушивании этих записей действительно заметно, что они сильно выбиваются из исходного сигнала.

2.3. Упражнение 2.4.

Создадим треугольный сигнал и выведем его график на экран:

```
1 from thinkdsp import TriangleSignal  
2  
3 triangle = TriangleSignal().make_wave(duration=0.01)  
4 triangle.plot()
```

График выглядит следующим образом:

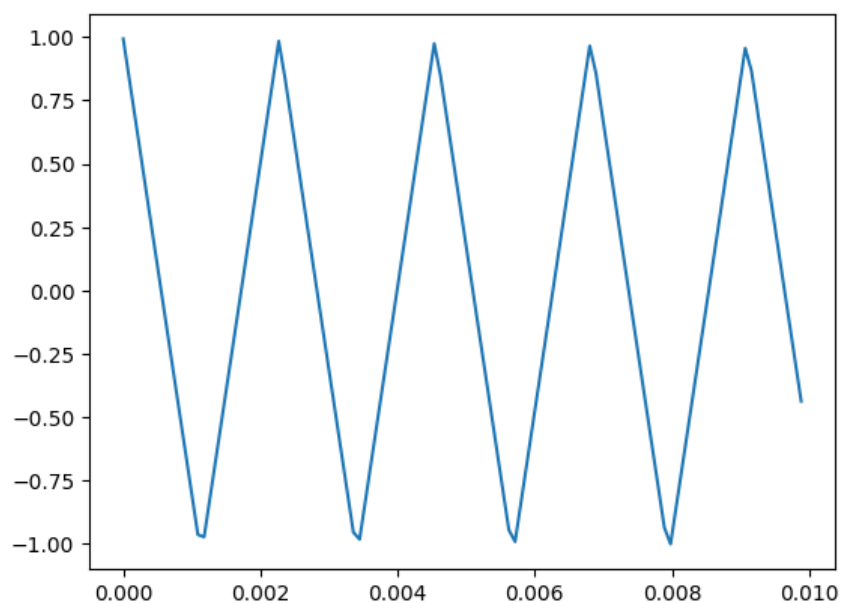


Рис. 2.6. График треугольного сигнала.

Теперь получим спектр этого сигнала и выведем первый элемент массива `hs`, который является результатом БПФ:

```
1 spectrum = triangle.make_spectrum()
2 spectrum.hs[0]
```

Результат примерно равен нулю:

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

Изменим его значение на 100 и посмотрим на результат:

```
1 spectrum.hs[0] = 100
2 triangle.plot(color='gray')
3 spectrum.make_wave().plot()
```

В результате получаем следующий график:

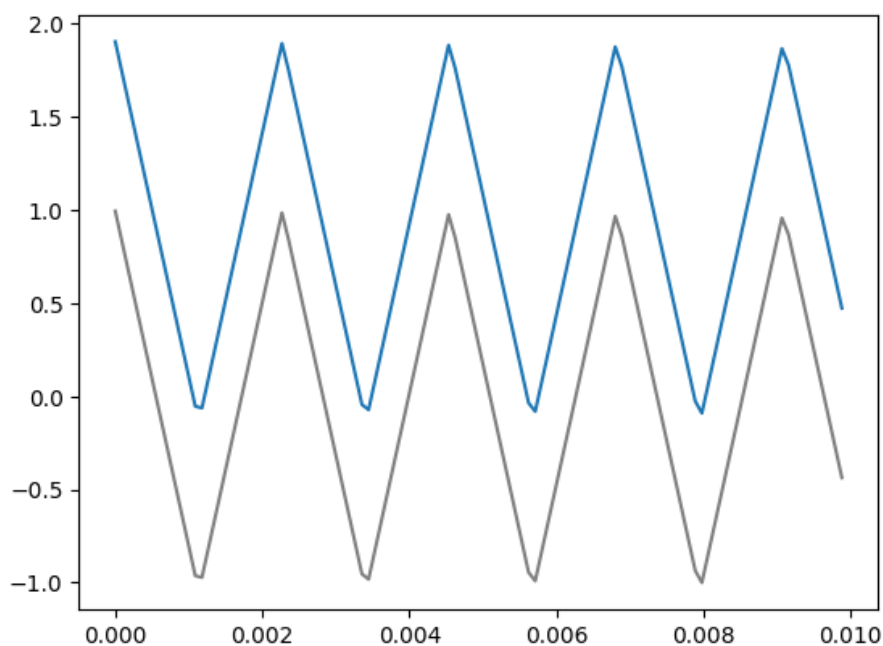


Рис. 2.7. График треугольного сигнала, после изменения hs .

Как можно заметить, получившийся сигнал отличается от исходного только вертикального смещения.

2.4. Упражнение 2.5.

Напишем функцию `filter_spectrum`, которая принимает спектр и изменяет его, выполняя деление каждый элемент hs , на соответствующую частоты из fs :

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

Создадим треугольный сигнал, выведем его в виде аудио для дальнейшего сравнения, после чего вызовем нашу функцию фильтрации. Выведем спектрограмму до и после, а также аудио файл после использования функции:

```
1 wave = TriangleSignal(freq=440).make_wave(duration=0.5)
2 wave.make_audio()
3 spectrum = wave.make_spectrum()
4 spectrum.plot(high=10000, color='gray')
5 filter_spectrum(spectrum)
6 spectrum.scale(440)
7 spectrum.plot(high=10000)
8 filtered = spectrum.make_wave()
9 filtered.make_audio()
```

Получившийся график выглядит следующим образом:

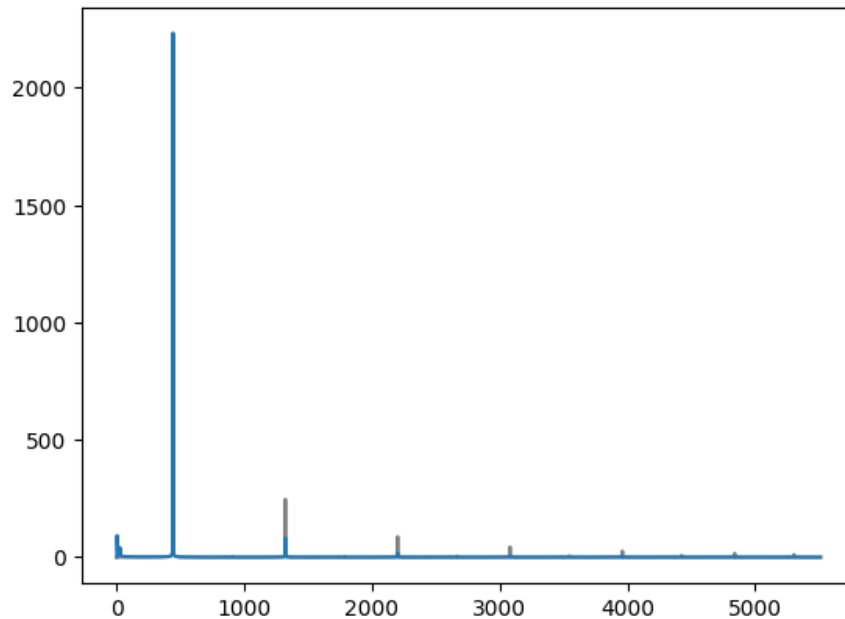


Рис. 2.8. Спектрограмма до и после использования функции.

Как можно заметить, чем больше частота, тем меньше становится пик после использования функции. Это логично, ведь деление происходит именно на частоту. Результат похож на low_pass фильтр.

Так же при прослушивании полученных аудио, мы получаем схожий результат.

2.5. Упражнение 2.6.

Создадим сигнал, в котором есть как четные, так и нечетные гармоники, которые спадают пропорционально $1/f^2$. Сигнал будем собирать, используя несколько синусоид:

```

1 from thinkdsp import SinSignal
2 import numpy as np
3
4 freqs = np.arange(500, 9500, 500)
5 amps = (1 / freqs**2) * 10 ** 5
6 signal = sum(SinSignal(freq, amp) for freq, amp in zip(freqs, amps))
7 wave = signal.make_wave(duration=0.5, framerate=20000)
8 wave.segment(duration=0.01).plot()
9 wave.make_audio()
```

Как мы видим. Частоты будут изменяться от 500 до 9000 с шагом 500. Амплитуда вычисляется, путем деления $1/f^2$, как это требует задание.

График сигнала будет выглядеть следующим образом:

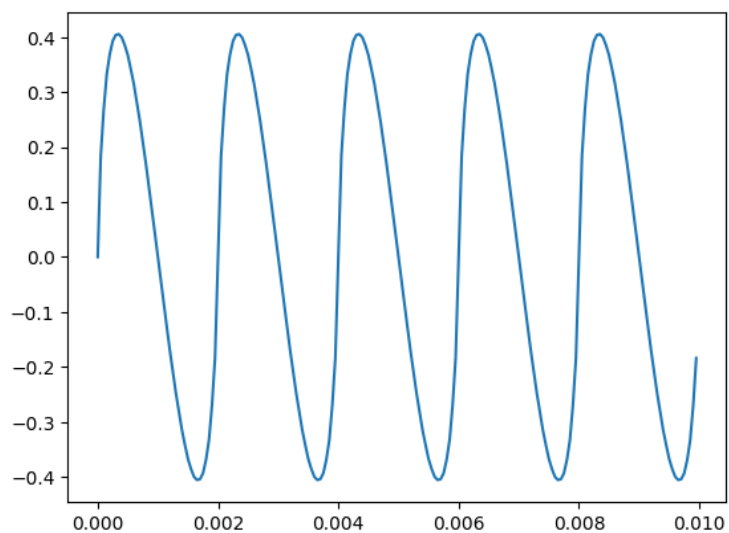


Рис. 2.9. Сигнал с четными и нечетными гармониками и амплитудой пропорциональной $1/f^2$
 Убедимся, что полученный сигнал соответствует требованиям, выведем его спектрограмму:

```

1 spectrum = wave.make_spectrum()
2 spectrum.plot()
3 spectrum.peaks()[ :10]

```

График выглядит следующим образом:

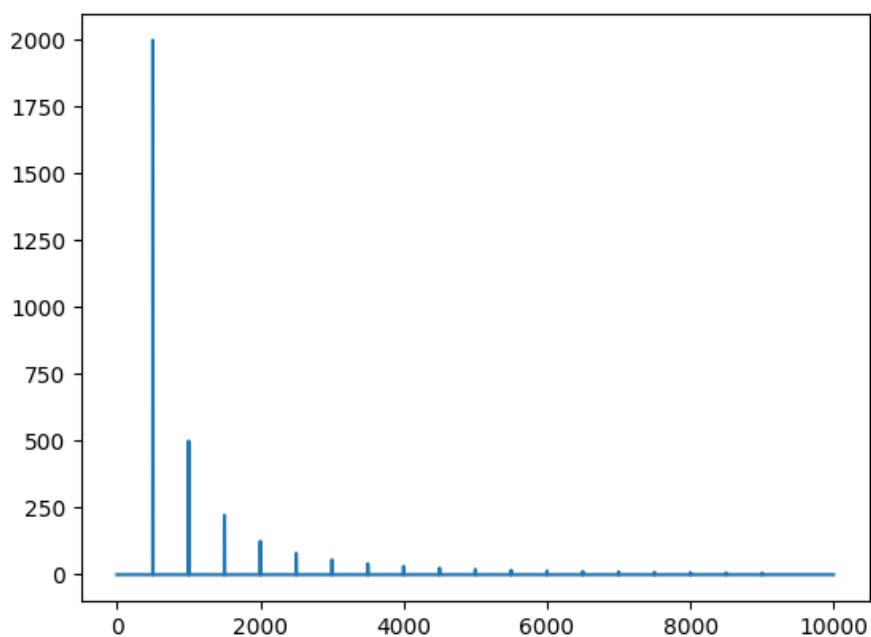


Рис. 2.10. Спектрограмма сигнала.

Для удобства анализа посмотрим на первые 10 пиков сигнала:

```
[(2000.0, 500.0),  
(500.00, 1000.0),  
(222.22, 1500.0),  
(125.00, 2000.0),  
(80.000, 2500.0),  
(55.555, 3000.0),  
(40.816, 3500.0),  
(31.250, 4000.0),  
(24.691, 4500.0),  
(20.000, 5000.0)]
```

Как мы видим, спектрограмма соответствует требованиям задания.

3. Лабораторная работа 3. Аперiodические сигналы.

3.1. Упражнение 3.1.

Выполним сравнение различных оконных функций, а именно стандартной hamming и bartlett, blackman, hanning.

Создадим сначала синусоидальный сигнал, с частотой 440 и сделаем, чтоб сигнал начинался с 0, а заканчивался в 1. Выведем его спектрограмму:

```
1 from thinkdsp import SinSignal  
2 import matplotlib.pyplot as plt  
3 import numpy as np  
4  
5 signal = SinSignal(freq=440)  
6 duration = signal.period * 30.25  
7 wave = signal.make_wave(duration)  
8 spectrum = wave.make_spectrum()  
9 spectrum.plot(high=880)
```

Результат выглядит следующим образом:

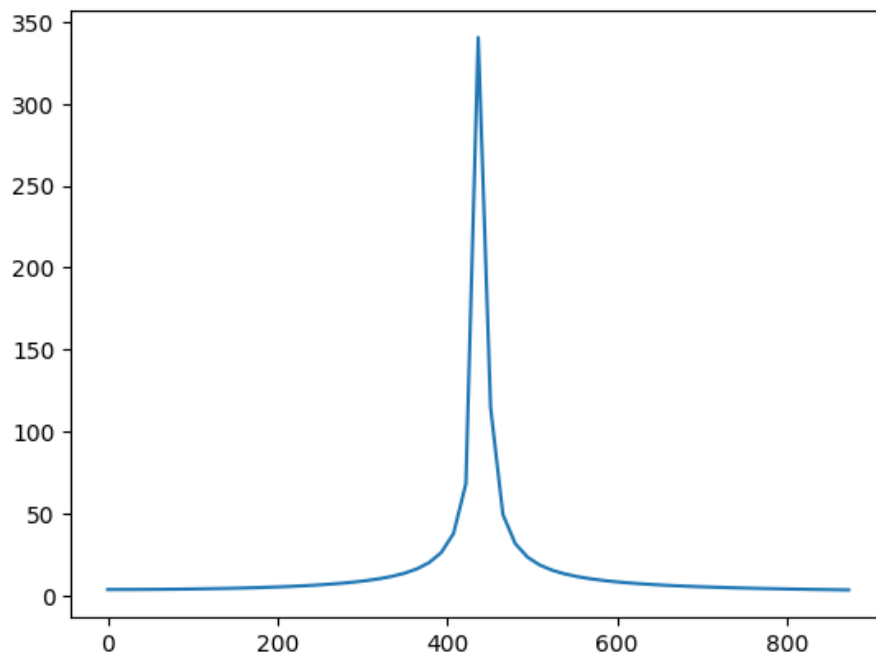


Рис. 3.1. Спектрограмма сигнала.

А теперь применим к этой спектрограмме различные оконные функции:



```
1 for window_func in [np.bartlett, np.blackman, np.hamming, np.hanning]:
2     wave = signal.make_wave(duration)
3     wave.ys *= window_func(len(wave.ys))
4
5     spectrum = wave.make_spectrum()
6     spectrum.plot(high=880, label=window_func.__name__)
7 plt.legend(loc='best')
8 plt.xlabel('Frequency (Hz)')
```

Результат выглядит следующим образом:

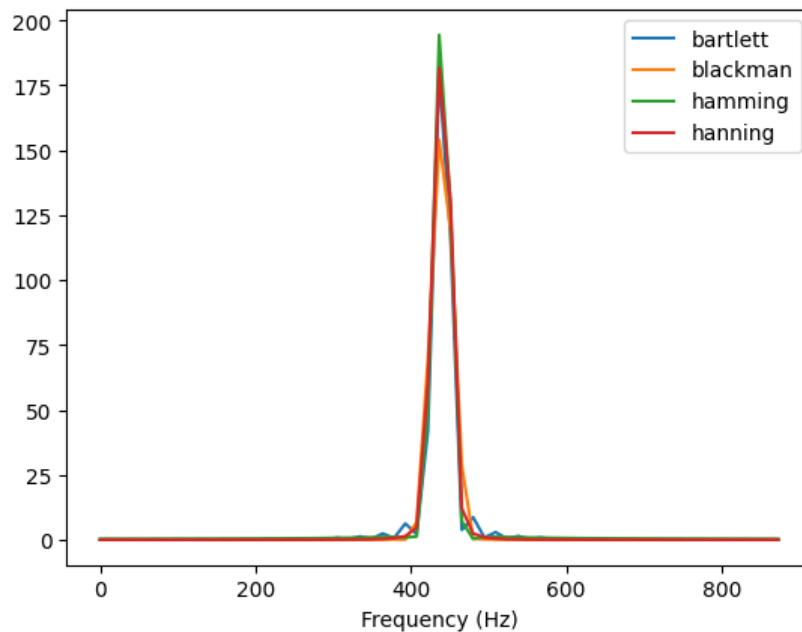


Рис. 3.2. Спектрограммы после применения оконных функций.

Как мы видим, функция Хемминга показывает себя лучше всего, именно поэтому она выбрана как универсальный вариант.

3.2. Упражнение 3.2.

Создадим класс, расширяющий Chirp для создания увеличивающегося пилообразного сигнала:



```
1 from thinkdsp import Chirp
2 from thinkdsp import normalize, unbias, PI2, decorate
3 import numpy as np
4
5 class SawtoothChirp(Chirp):
6
7     def evaluate(self, ts):
8         freqs = np.linspace(self.start, self.end, len(ts))
9         dts = np.diff(ts, prepend=0)
10        dphis = PI2 * freqs * dts
11        phases = np.cumsum(dphis)
12        cycles = phases / PI2
13        frac, _ = np.modf(cycles)
14        ys = normalize(unbias(frac), self.amp)
15        return ys
```

Эта функция является совмещением функции, созданной в 2.1 и функции для создания Chirp. Создадим экземпляр этого класса с начальной частотой 220 и конечной 880:



```
1 signal = SawtoothChirp(start=220, end=880)
2 wave = signal.make_wave(duration=1, framerate=4000)
3 wave.apodize()
4 wave.segment(duration=0.05).plot()
5 wave.make_audio()
```

Убедимся в том, что полученная запись соответствует ожиданиям:

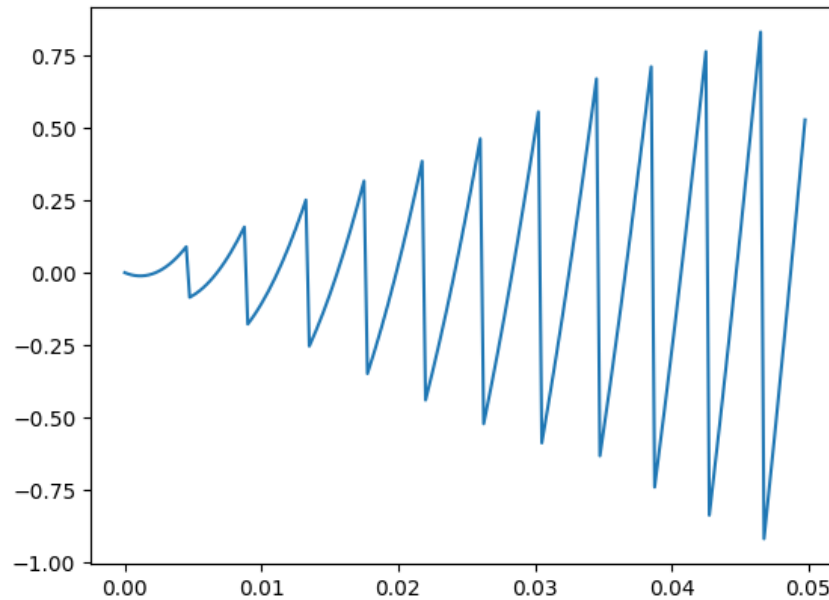


Рис. 3.3. пилообразный чирп.

Как мы видим, это действительно пилообразный чирп, так же это слышно по аудио записи.

Создадим спектрограмму заданного сигнала:



```
1 sp = wave.make_spectrogram(256)
2 sp.plot()
3 decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

Получившаяся спектрограмма выглядит следующим образом:

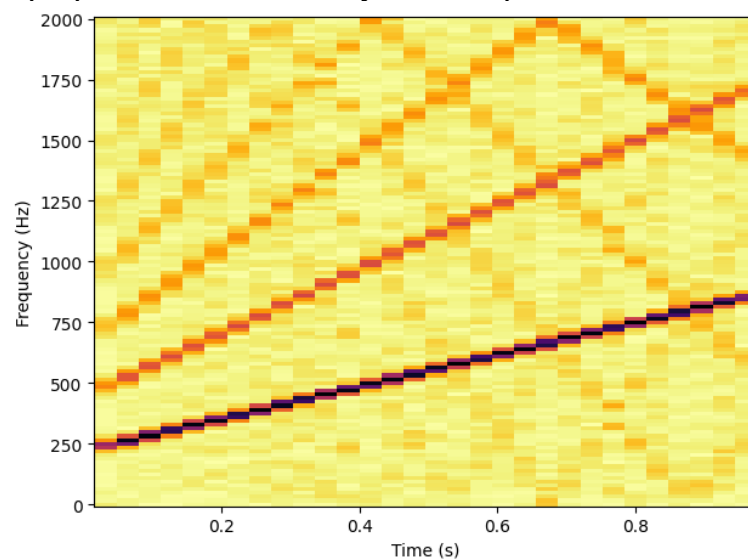


Рис. 3.4. Спектрограмма пилообразного сигнала.

Здесь стоит отметить, что в связи с маленьким фреймрейтом мы получаем достаточно шумный спектр из-за заворота (биения), этот эффект хорошо заметен, как «отскакивающие» от верхней границы темные участки.

3.3. Упражнение 3.3.

Попробуем предположить, какой будет иметь вид спектр пилообразного чирпа с начальной частотой 2500, конечной 3000 и фреймрейтом 20000.

Поскольку пилообразный сигнал содержит как четные, так и нечетные гармоники, а также они уменьшаются пропорционально $1/f$ можно предположить, что первый пик будет размазан между начальной и конечной частотой, второй будет в 2 раза меньше на частотах с 5000 до 6000, а третий будет на частотах с 7500 до 9000 Гц и будет в 3 раза меньше пика с 2500 до 3000. Убедимся в этом, создав необходимый чирп:

```
1 from lab_3_2 import SawtoothChirp
2
3 signal = SawtoothChirp(start=2500, end=3000)
4 wave = signal.make_wave(duration=1, framerate=20000)
5 wave.make_spectrum().plot(scalex=500)
6 wave.make_audio()
```

Получившаяся спектрограмма выглядит следующим образом:

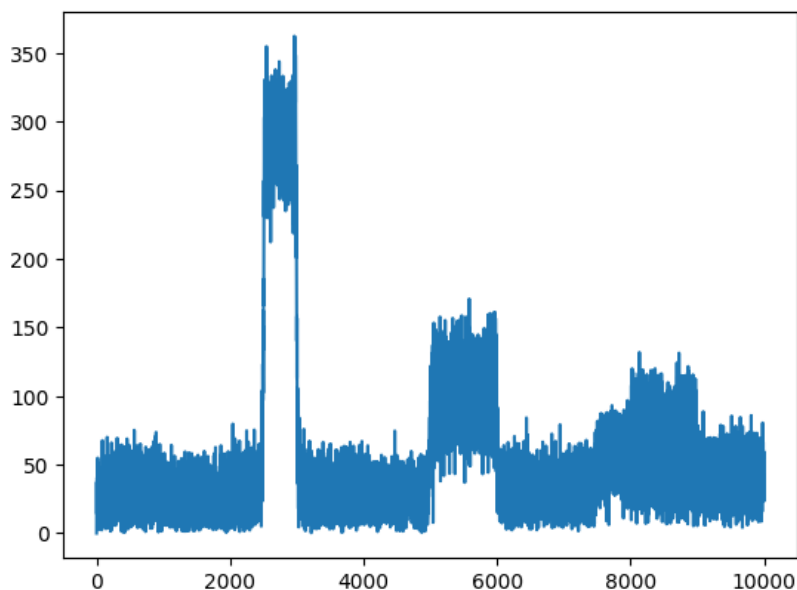


Рис. 3.5. Спектрограмма пилообразного чирпа.

Как мы и ожидали спектрограмма полностью соответствует ожиданиям.

3.4. Упражнение 3.4.

Скачаем глissандо (нота, меняющаяся от одной высоты к другой) и создадим его спектрограмму:

```
1 from thinkdsp import read_wave, decorate
2 wave = read_wave('code_72475__rockwehrmann__glissup02.wav')
3 wave.make_audio()
4 wave.make_spectrogram(512).plot(high=5000)
5 decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
```

Результирующая спектрограмма выглядит следующим образом:

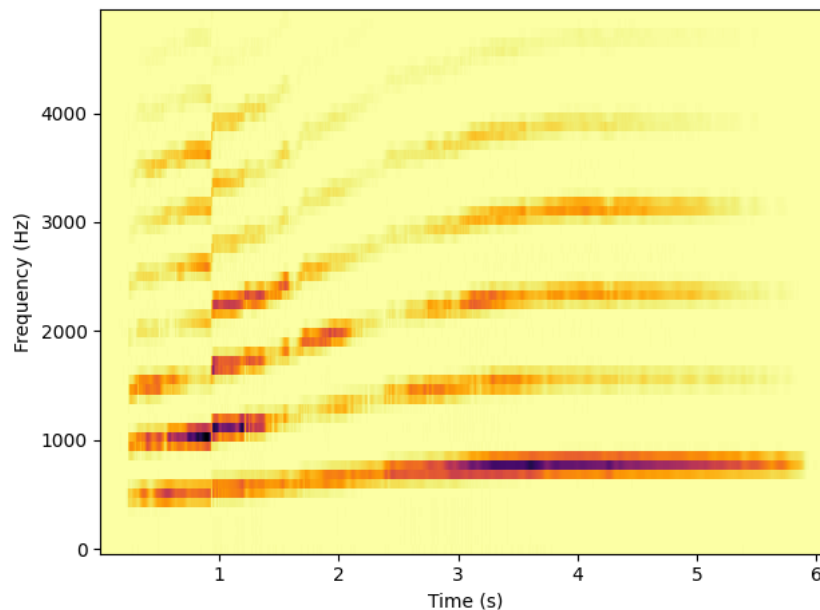


Рис. 3.6. Спектрограмма глissандо.

Как можно заметить, частота меняется вместе с изменением высоты ноты (под конец в произведении нота почти не менялась, поэтому и спектрограмма сохраняла свое состояние).

3.5. Упражнение 3.5.

Создадим класс, который будет имитировать глissандо на тромбоне, при постоянной скорости изменения трубы, если частота звука обратно пропорциональна длине:

```
1 from thinkdsp import Chirp, decorate, PI2
2 import numpy as np
3
4 class TromboneGliss(Chirp):
5
6     def evaluate(self, ts):
7         l1, l2 = 1.0 / self.start, 1.0 / self.end
8         lengths = np.linspace(l1, l2, len(ts))
9         freqs = 1 / lengths
10
11         dts = np.diff(ts, prepend=0)
12         dphis = PI2 * freqs * dts
13         phases = np.cumsum(dphis)
14         ys = self.amp * np.cos(phases)
15         return ys
```

В строках с 7 по 9 имитируется изменение длины кулисы тромбона, а далее стандартное объявление функции evaluate.


```

1 low = 262
2 high = 349
3 signal = TromboneGliss(high, low)
4 wave1 = signal.make_wave(duration=1)
5 wave1.apodize()
6 signal = TromboneGliss(low, high)
7 wave2 = signal.make_wave(duration=1)
8 wave2.apodize()
9 wave = wave1 | wave2
10 sp = wave.make_spectrogram(1024)
11 sp.plot(high=500)
12 decorate(xlabel='Time (s)', ylabel='Frequency (Hz)')
13 wave.make_audio()

```

Получившаяся спектрограмма выглядит следующим образом:

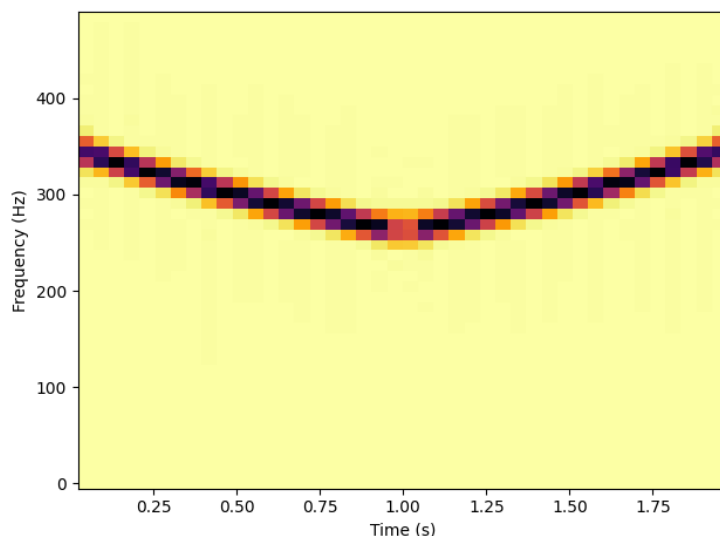


Рис. 3.7. Спектрограмма звука тромбона от C3 до F3 и обратно.

Кажется, что этот сигнал ближе к линейному, однако увеличим разброс до с 1000 Гц до 263:

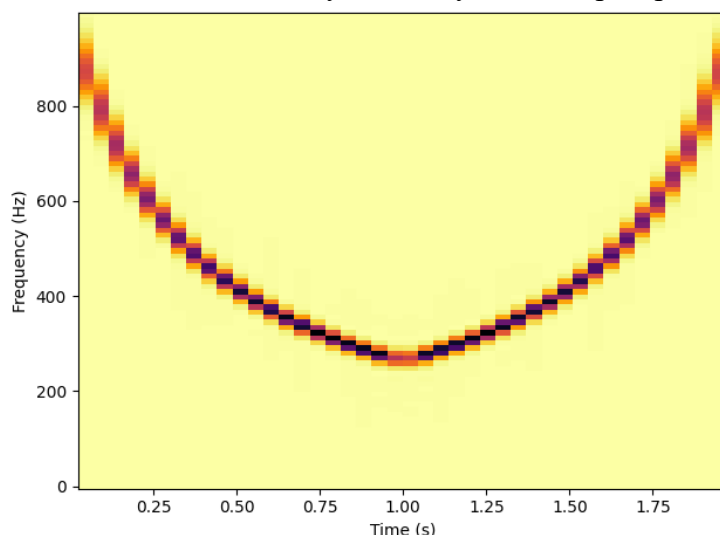


Рис. 3.8. Спектрограмма звука тромбона от 1000 до F3 и обратно.

Как мы видим, теперь сигнал ближе к экспоненциальному, нежели линейному т.к. функция обратно пропорциональна длине, а эта функция степенная, а не линейная.

3.6. Упражнение 3.6.

Далее возьмем аудио с гласными звуками и посмотрим на их спектрограммы, попробуем понять, как они отличаются.

Для начала загрузим аудио и посмотрим на полную спектрограмму:

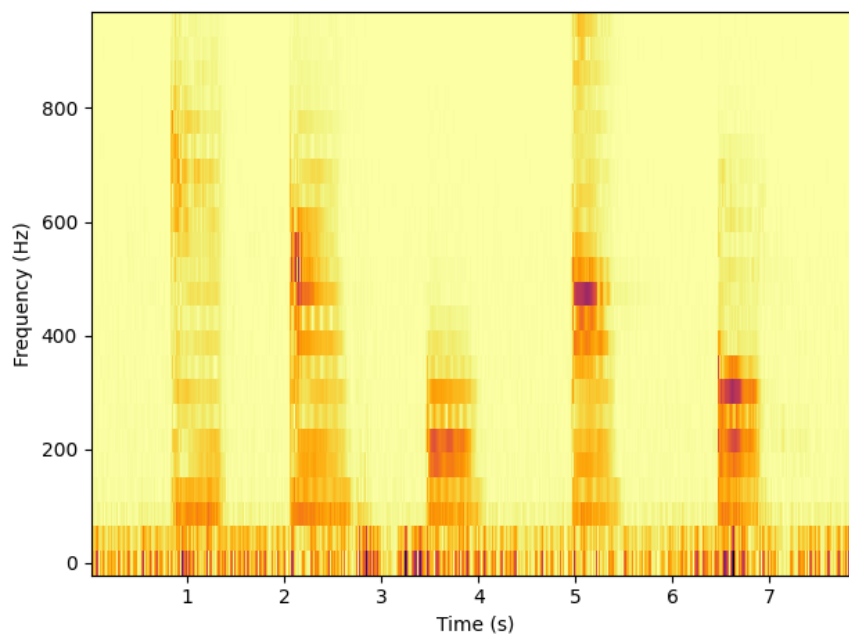


Рис. 3.9. Спектрограмма гласных звуков.

Теперь обрежем первый звук 'а' и посмотрим на его спектрограмму:

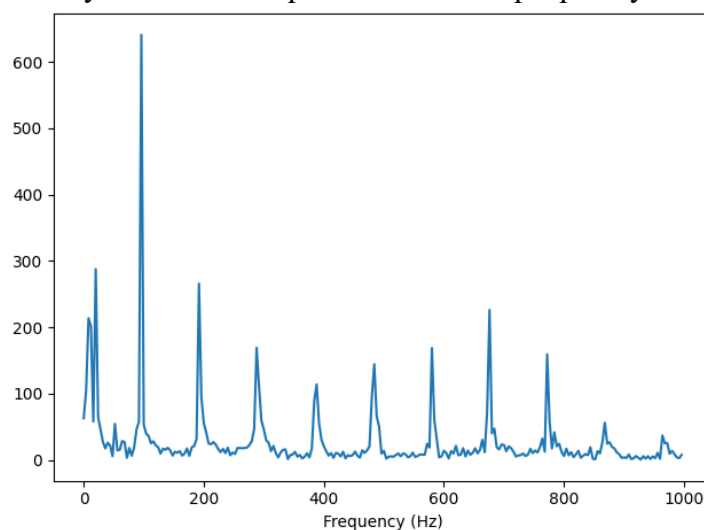


Рис. 3.10. Спектрограмма звука 'а'.

Теперь обрежем второй звук 'э' и посмотрим на его спектрограмму:

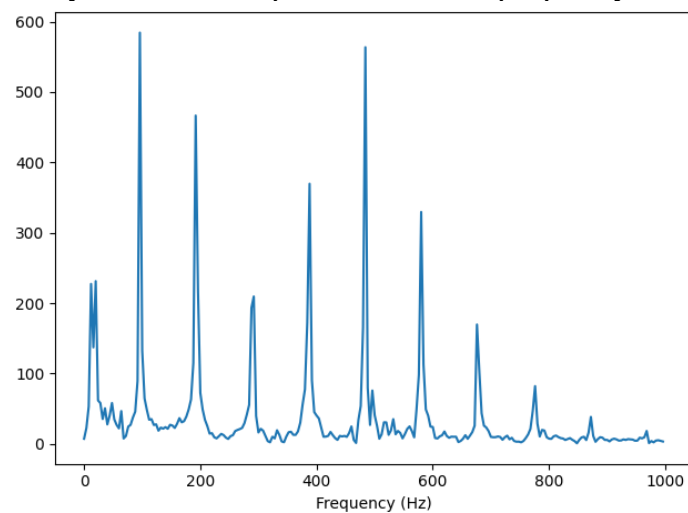


Рис. 3.11. Спектрограмма звука 'э'.

Теперь обрежем второй звук 'и' и посмотрим на его спектрограмму:

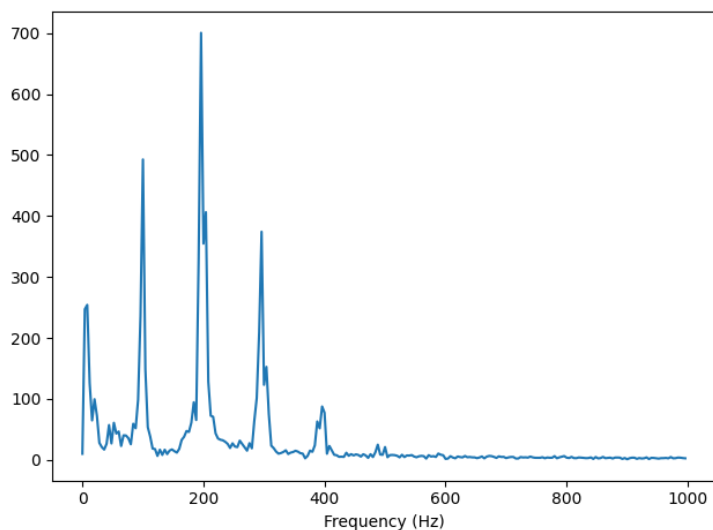


Рис. 3.12. Спектрограмма звука 'и'.

Теперь обрежем второй звук 'о' и посмотрим на его спектрограмму:

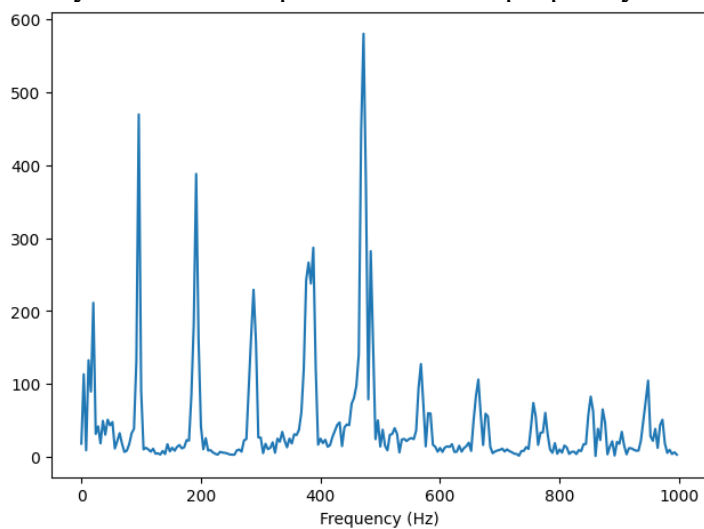


Рис. 3.13. Спектрограмма звука 'о'.

Теперь обрежем второй звук 'у' и посмотрим на его спектрограмму:

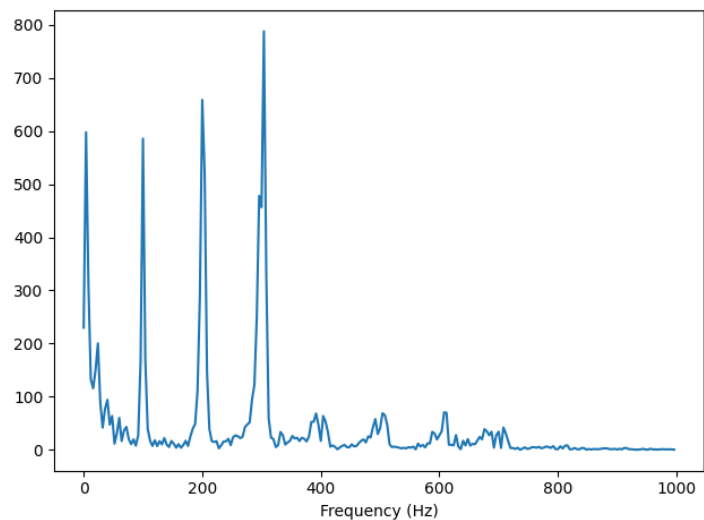


Рис. 3.14. Спектрограмма звука 'у'.

Как можно заметить несмотря на то, что звуки звучат достаточно похоже их спектрограммы сильно отличаются и дают возможность понять, какой действительно звук был произнесен.

Вероятно, именно этот метод позволяет распознавать речь и отдельные звуки в современных голосовых помощниках.

4. Приложение:

Ссылка на репозиторий с исходными кодами: https://github.com/DafterT/telecom_labs