Лабораторная работа №6
Дискретное косинусное преобразование

Кобыжев Александр

11 апреля 2021 г.

Оглавление

Список иллюстраций

Листинги

Глава 1

Упражнение 6.1

Начнём с шумового сигнала и массива величин степени двойки:

```
signal = thinkdsp.UncorrelatedGaussianNoise()
noise = signal.make_wave(duration=1.0, framerate=16384)
noise.ys.shape
```

Листинг 1.1: Создание сигнала

`Shape` сигнала составляет 16384.

```
ns = 2 ** np.arange(6, 15)
ns
```

Листинг 1.2: Массив степеней двойки

Следующая функция берёт массив результатов временного эксперимента, отображает результаты и выстраивает прямую линию.

```
def plot_bests(bests):
    thinkplot.plot(ns, bests)
    thinkplot.config(xscale='log', yscale='log', legend=False)

    x = np.log(ns)
    y = np.log(bests)
    t = scipy.stats.linregress(x,y)
    slope = t[0]

    return slope
```

Листинг 1.3: Функция `plot_bests`

Рассмотрим результаты для `analyze1`.

```
results = []
```

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

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

Листинг 1.4: Работа с `analyze1`

```
64
657 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
128
1.42 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
256
3.64 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
512
9.54 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
1024
33.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2048
152 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
4096
676 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
8192
4.19 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
16384
25.7 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

1.9111873370989283
```

Листинг 1.5: Результат работы

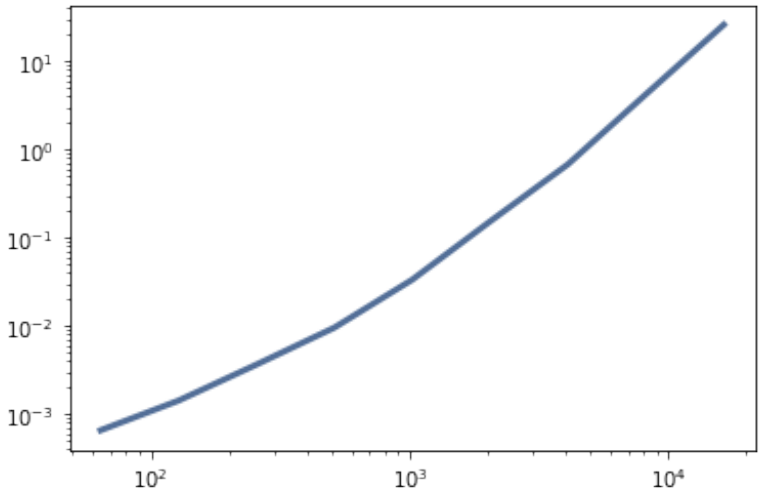

Рис. 1.1: Визуализация данных с `analyze1`

Наклон близок к 2, а не к 3. Одна из причин состоит в том, что производительность `np.linalg.solve` почти квадратична в этом диапазоне размеров массива.

Линия изогнута, что говорит нам о том, что мы не достигли размера массива, при котором среда выполнения показывает кубический рост. Я думаю, что при больших размерах массива наклон в конечном итоге сходится к 3.

Теперь рассмотрим результат для `analyze2`:

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

bests2 = [result.best for result in results]
plot_bests(bests2)
```

Листинг 1.6: Работа с `analyze2`

```
42.5 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
212 µs ±0 ns per loop (mean ±std. dev. of 1 run, 1000 loops each)
1.04 ms ±0 ns per loop (mean ±std. dev. of 1 run, 1000 loops each)
3.22 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100 loops each)
```

```
12.2 ms ± 0 ns per loop (mean ±std. dev. of 1 run, 100 loops each)
47.9 ms ± 0 ns per loop (mean ±std. dev. of 1 run, 10 loops each)
189 ms ± 0 ns per loop (mean ±std. dev. of 1 run, 1 loop each)
722 ms ± 0 ns per loop (mean ±std. dev. of 1 run, 1 loop each)
3.05 s ± 0 ns per loop (mean ±std. dev. of 1 run, 1 loop each)

1.9772047966153776
```

Листинг 1.7: Результат работы

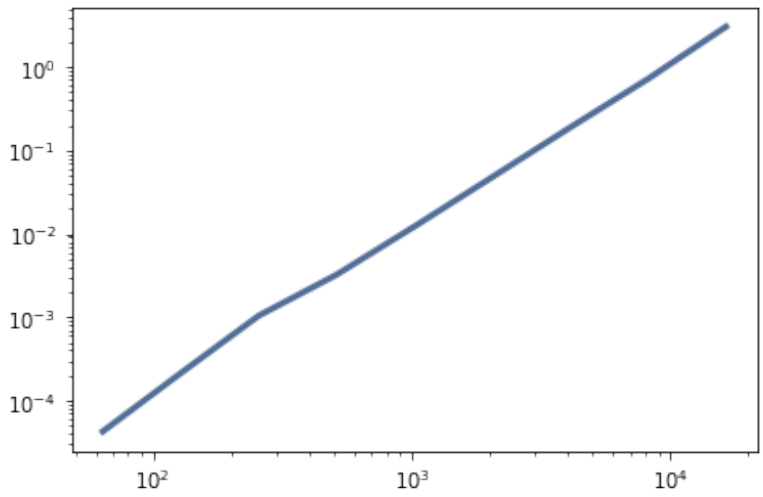

Рис. 1.2: Визуализация данных с `analyze2`

Результаты для `analyze2`, как и ожидалось, складываются в прямую линию с предполагаемым наклоном, близким к 2.

Рассмотрим результаты для `scipy.fftpack.dct`:

```
results = []
for N in ns:
    ys = noise.ys[:N]
    result = %timeit -o scipy.fftpack.dct(ys, type=3)
    results.append(result)

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

Листинг 1.8: Работа с `scipy.fftpack.dct`

```
5.33 µs ±538 ns per loop (mean ±std. dev. of 7 runs, 100000 loops
    each)
```

```
5.25 µs ±171 ns per loop (mean ±std. dev. of 7 runs, 100000 loops each)
6.55 µs ±408 ns per loop (mean ±std. dev. of 7 runs, 100000 loops each)
7.7 µs ±128 ns per loop (mean ±std. dev. of 7 runs, 100000 loops each)
9.5 µs ±339 ns per loop (mean ±std. dev. of 7 runs, 100000 loops each)
14.9 µs ±407 ns per loop (mean ±std. dev. of 7 runs, 100000 loops each)
27.3 µs ±141 ns per loop (mean ±std. dev. of 7 runs, 10000 loops each)
54.5 µs ±470 ns per loop (mean ±std. dev. of 7 runs, 10000 loops each)
107 µs ±797 ns per loop (mean ±std. dev. of 7 runs, 10000 loops each)

0.5564395911984347
```

Листинг 1.9: Результат работы

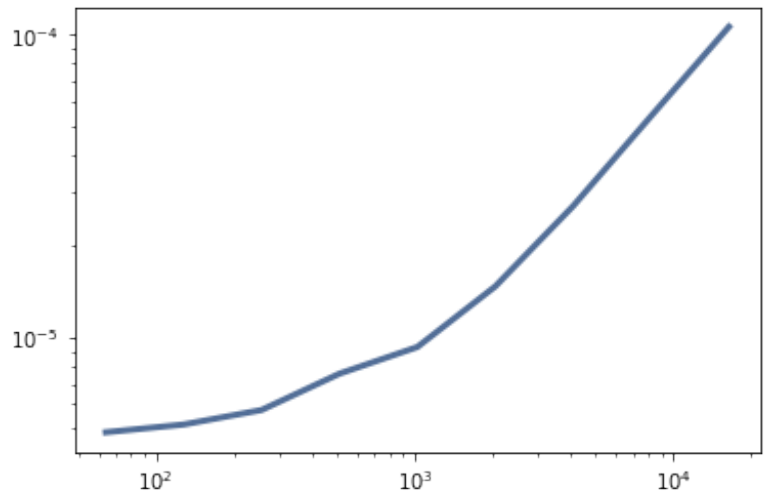

Рис. 1.3: Визуализация данных с `scipy.fftpack.dct`

Эта реализация `dct` ещё быстрее. Линия изогнута, что означает, что либо мы ещё не видели асимптотическое поведение, либо асимптотическое поведение не является простым показателем n. Фактически, как мы скоро увидим, время выполнения пропорционально $nlogn$.

Теперь рассмотрим все кривые на одних осях:

```
thinkplot.preplot(3)
thinkplot.plot(ns, bests, label='analyze1')
thinkplot.plot(ns, bests2, label='analyze2')
thinkplot.plot(ns, bests3, label='fftpack.dct')
thinkplot.config(xscale='log', yscale='log', legend=True,
    loc='upper left')
```

Листинг 1.10: Визуализация всех кривых

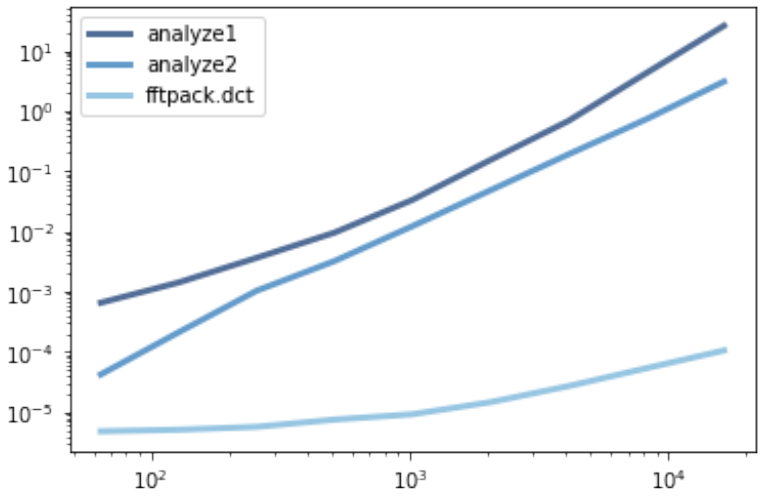

Рис. 1.4: Визуализация всех кривых

Глава 2

Упражнение 6.2

`thinkdsp` предоставляет класс `Dct`, похожий на `Spectrum`, но использующий DCT вместо FFT.

Воспользуемся записью саксофона из предыдущей лабораторной работы:

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

Листинг 2.1: Загрузка звука

Теперь сделаем короткий сегмент, а затем ДКП этого сегмента:

```
segment = wave.segment(start=2.0, duration=0.5)
segment.normalize()
segment.make_audio()
```

Листинг 2.2: Сегмент звука

```
seg_dct = segment.make_dct()
seg_dct.plot(high=4000)
thinkplot.config(xlabel='Frequency (Hz)', ylabel='DCT')
```

Листинг 2.3: Сжатие звука

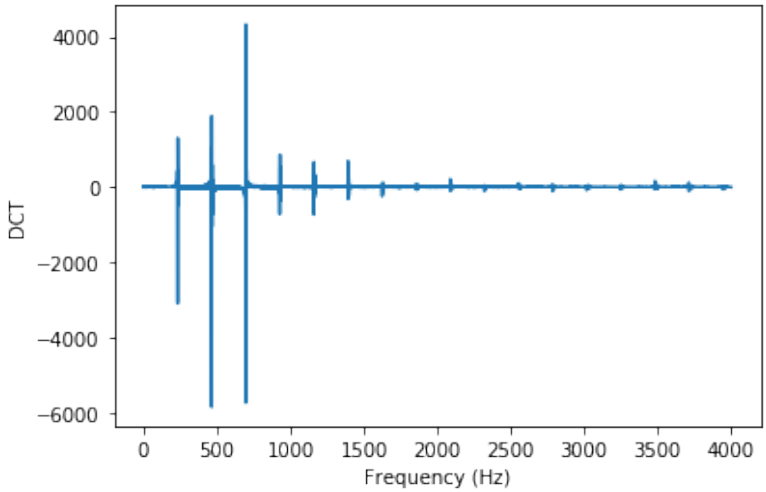

Рис. 2.1: Визуализация сжатого звука

Есть только несколько гармоник со значительной амплитудой, и многие записи близки к нулю. Следующая функция принимает ДКП и устанавливает для элементов ниже порога значение 0.

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

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

Листинг 2.4: Функция `compress`

Если мы применим его к сегменту, мы можем удалить более 90% элементов:

```
seg_dct = segment.make_dct()
compress(seg_dct, thresh=10)
seg_dct.plot(high=4000)
```

Листинг 2.5: Сжатие звука

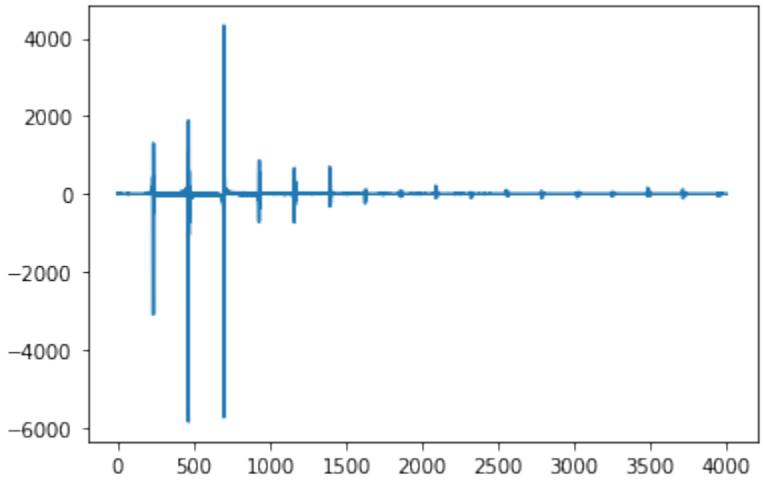

Рис. 2.2: Визуализация сжатого звука

```
seg2 = seg_dct.make_wave()
seg2.make_audio()
```

Листинг 2.6: Воспроизведение сжатого звука

Результат звучит точно так же, по ощущениям ничего не изменилось.

Чтобы сжать более длинный сегмент, мы можем сделать спектрограмму ДКП. Следующая функция похожа на `wave.make_spectrogram` за исключением того, что использует ДКП.

```
def make_dct_spectrogram(wave, seg_length):
    """Computes the DCT spectrogram of the wave.

    seg_length: number of samples in each segment

    returns: Spectrogram
    """
    window = np.hamming(seg_length)
    i, j = 0, seg_length
    step = seg_length // 2

    # map from time to Spectrum
    spec_map = {}

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

```
18
19          # the nominal time for this segment is the midpoint
20          t = (segment.start + segment.end) / 2
21          spec_map[t] = segment.make_dct()
22
23          i += step
24          j += step
25
26      return thinkdsp.Spectrogram(spec_map, seg_length)
```

Листинг 2.7: Функция `make_dct_spectrogram`

Теперь мы можем составить DCT-спектрограмму и применить сжатие к каждому сегменту:

```
1 spectro = make_dct_spectrogram(wave, seg_length=1024)
2 for t, dct in sorted(spectro.spec_map.items()):
3     compress(dct, thresh=0.2)
```

Листинг 2.8: Сжатие звука

В большинстве сегментов сжатие составляет 75-80%. Чтобы услышать, как это звучит, мы можем преобразовать спектрограмму обратно в волну и воспроизвести её.

```
1 wave2 = spectro.make_wave()
2 wave2.make_audio()
```

Листинг 2.9: Воспроизведение сжатого звука

Так же прослушаем оригинал для сравнения.

```
1 wave.make_audio()
```

Листинг 2.10: Воспроизведение оригинального звука

При сжатии слышно характерный треск во время воспроизведения аудио, так что можно смело сказать, что нам удалось сжать аудиозапись.

Глава 3

Упражнение 6.3

3.1 Пилообразная волна

Для начала воспользуемся пилообразной формой волны, а затем перейдём к более естественным звукам.

```
signal = thinkdsp.SawtoothSignal(freq=300, offset=0)
wave = signal.make_wave(duration=0.5, framerate=40000)
wave.make_audio()
```

Листинг 3.1: Создание пилообразного сигнала

```
wave.segment(duration=0.01).plot()
```

Листинг 3.2: Визуализация сигнала

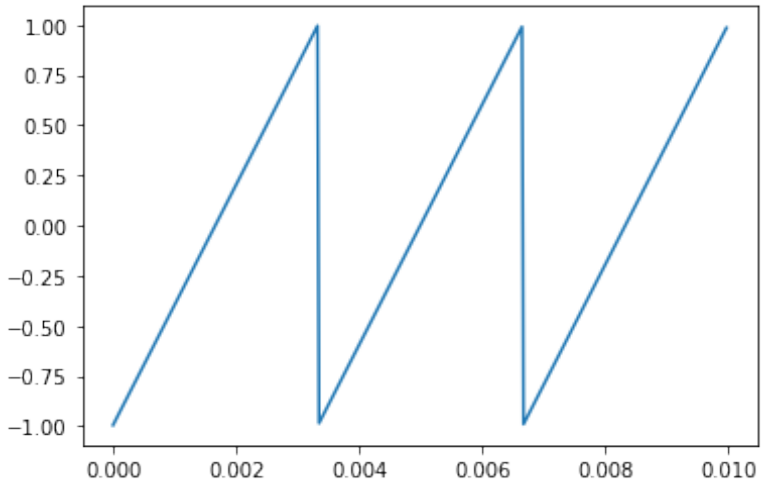

Рис. 3.1: Визуализация сигнала

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

Листинг 3.3: Спектр сигнала

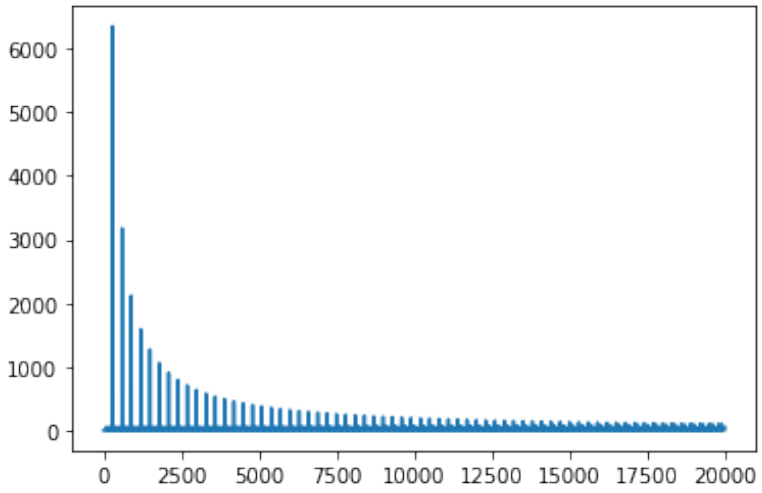

Рис. 3.2: Спектр сигнала

Рассмотрим угловую часть спектра при помощи функции `plot_angle`.

```
def plot_angle(spectrum, thresh=1):
    angles = spectrum.angles
```

```
    angles[spectrum.amps < thresh] = np.nan
    thinkplot.plot(spectrum.fs, angles, style='x')
    thinkplot.config(xlim=[0, spectrum.max_freq],
                     ylim =[-np.pi, np.pi])
```

Листинг 3.4: Функция `plot_angle`

На большинстве частот амплитуда мала, а угол - в значительной степени случайное число. Так что если мы построим все углы, получится небольшой беспорядок.

```
plot_angle(spectrum, thresh=0)
thinkplot.config(xlim=[0, spectrum.max_freq], ylim = [-np.pi,
    np.pi])
```

Листинг 3.5: Визуализация угловой части спектра с tresh

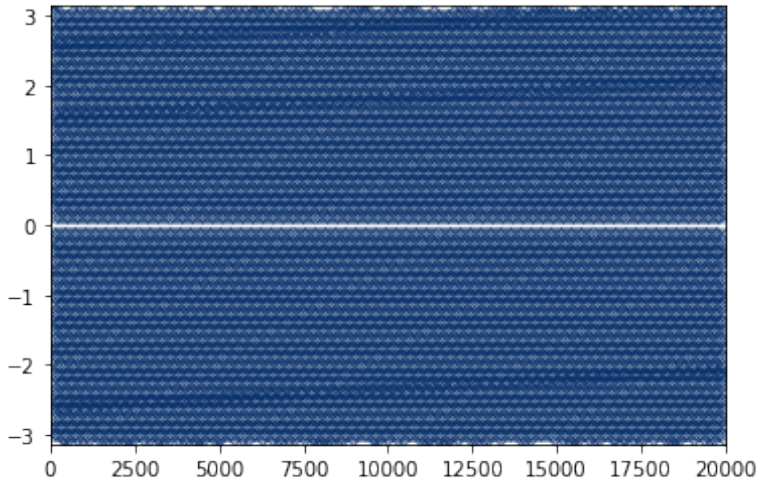

Рис. 3.3: Визуализация угловой части спектра с tresh = 0

```
plot_angle(spectrum, thresh=1)
thinkplot.config(xlim=[0, spectrum.max_freq], ylim = [-np.pi,
    np.pi])
```

Листинг 3.6: Визуализация угловой части спектра с tresh

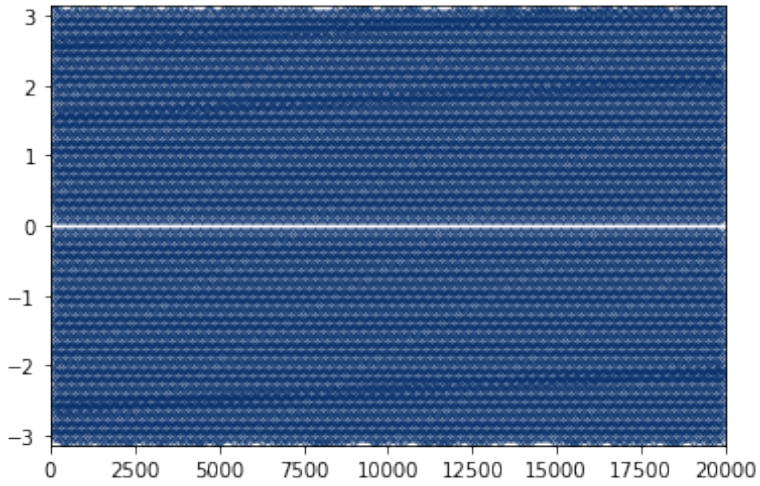

Рис. 3.4: Визуализация угловой части спектра с tresh = 1

Когда мы выбираем только те частоты, где величина превышает пороговое значение, мы видим, что в углах есть структура. Каждая гармоника смещена от предыдущей на доли радиана.

Следующая функция отображает амплитуды, углы и форму волны для заданного спектра.

```
def plot_three(spectrum, thresh=1):
    thinkplot.preplot(cols=3)
    spectrum.plot()
    thinkplot.subplot(2)
    plot_angle(spectrum, thresh=thresh)
    thinkplot.subplot(3)
    wave = spectrum.make_wave()
    wave.segment(duration=0.01).plot()
    wave.apodize()
    display(wave.make_audio())
```

Листинг 3.7: Функция `plot_three`

Теперь мы можем визуализировать неизменённый спектр:

```
plot_three(spectrum)
```

Листинг 3.8: Визуализация неизменённого спектра

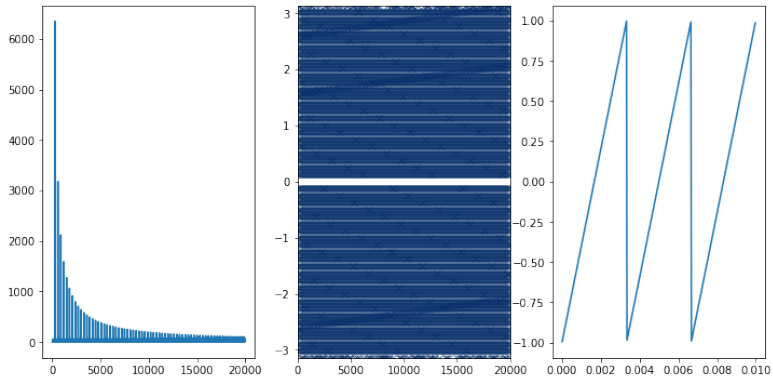

Рис. 3.5: Визуализация неизменённого спектра

Теперь рассмотрим, что произойдёт, если мы установим все углы в ноль.

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

Листинг 3.9: Функция `zero_angle`

Амплитуды не изменились, все углы равны нулю, и форма волны выглядит совсем иначе. Но волна звучит примерно так же, только появился какой-то шум, да и громкость стала немного ниже.

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

Листинг 3.10: Визуализация при нулевых углах

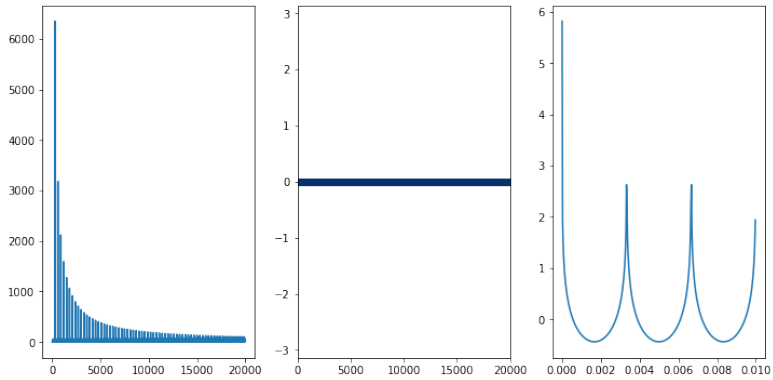

Рис. 3.6: Визуализация при нулевых углах

Если мы умножим комплексные компоненты на $exp(i\phi)$ это приведет к добавлению ϕ к углам:

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

Листинг 3.11: Функция `rotate_angle`

Эффект можно увидеть на рисунке ниже. Опять же, форма волны другая, но звучит примерно так же.

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

Листинг 3.12: Визуализация при повороте углов

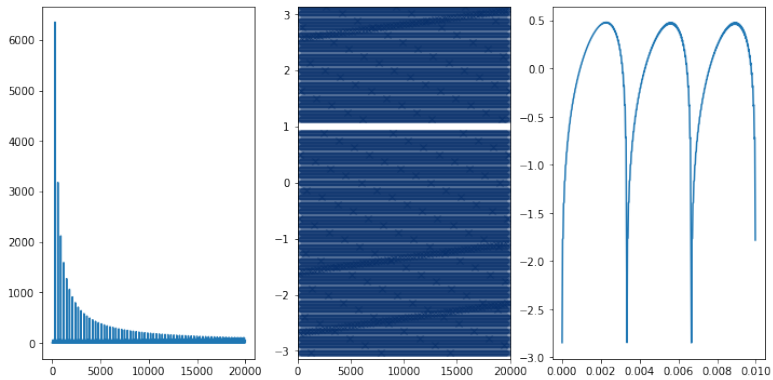

Рис. 3.7: Визуализация при повороте углов

Теперь рассмотрим случай, если мы установим углы на случайные значения.

```
PI2 = np.pi * 2

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

Листинг 3.13: Функция `random_angle`

Влияние на форму волны сильное, но воспринимаемый звук остаётся плюс минус прежним, правда он приобрёл "металлический"эффект.

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

Листинг 3.14: Визуализация при случайных углах

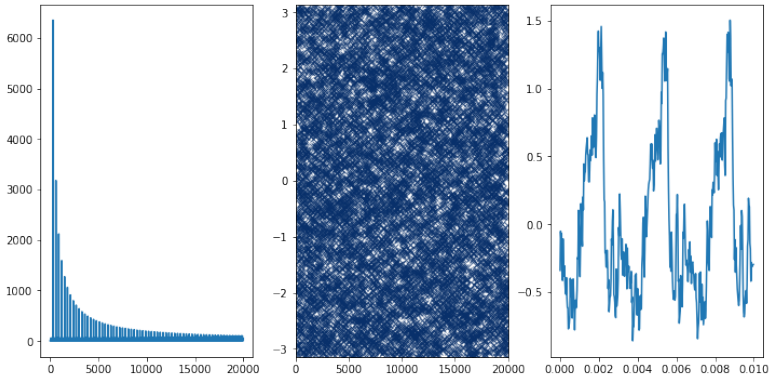

Рис. 3.8: Визуализация при случайных углах

3.2 Запись габоя

С более естественными звуками результаты несколько отличаются. Рассмотрим запись гобоя.

```
wave = thinkdsp.read_wave('120994__thirsk__120-oboe.wav')
wave.make_audio()
```

Листинг 3.15: Загрузка звука

```
segment = wave.segment(start=0.2, duration=0.6)
spectrum = segment.make_spectrum()
plot_three(spectrum, thresh=50)
```

Листинг 3.16: Визуализация неизменённого звука

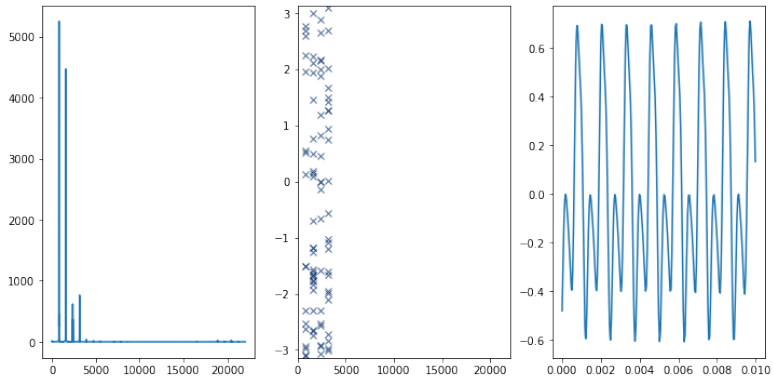

Рис. 3.9: Визуализация неизменённого звука

Здесь все углы установлены в ноль.

```
spectrum2 = zero_angle(spectrum)
plot_three(spectrum2, thresh=50)
```

Листинг 3.17: Визуализация при нулевых углах

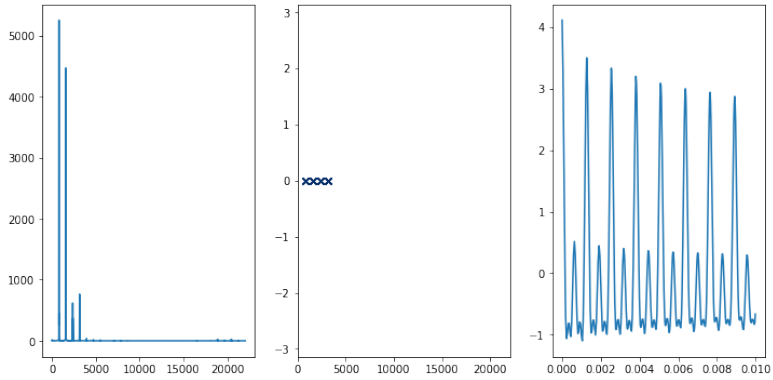

Рис. 3.10: Визуализация при нулевых углах

Теперь рассмотрим случай, когда углы повёрнуты на 1 радиан.

```
spectrum3 = rotate_angle(spectrum, 1)
plot_three(spectrum3, thresh=50)
```

Листинг 3.18: Визуализация при повороте углов

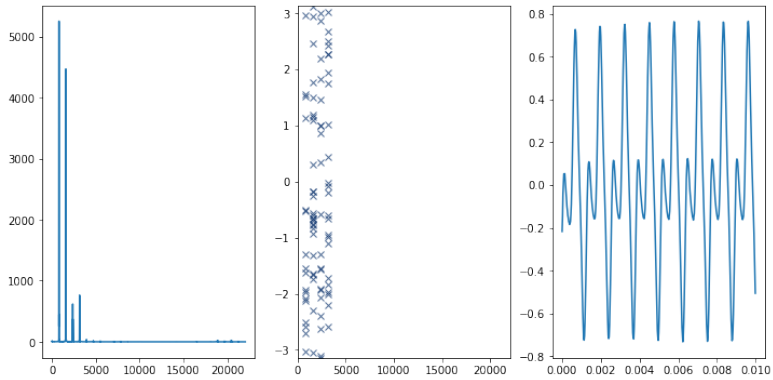

Рис. 3.11: Визуализация при повороте углов

Теперь рассмотрим случайные углы.

```
spectrum4 = random_angle(spectrum)
plot_three(spectrum4, thresh=50)
```

Листинг 3.19: Визуализация при случайных углах

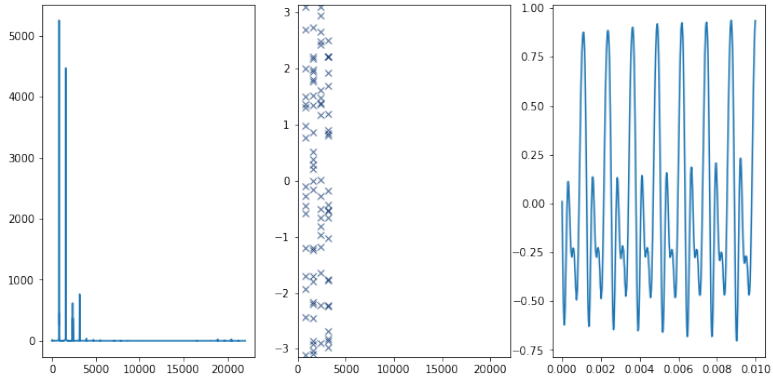

Рис. 3.12: Визуализация при случайных углах

Как мне кажется, установка углов в ноль снижает громкость, поворот углов не имеет вообще никакого эффекта, а случайные углы придают звуку какой-то эффект реверберации.

Попробуем то же самое с отрывком из записи саксофона.

3.3 Запись саксофона

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

Листинг 3.20: Загрузка звука

```
segment = wave.segment(start=2.5, duration=0.6)
spectrum = segment.make_spectrum()
plot_three(spectrum, thresh=50)
```

Листинг 3.21: Визуализация неизменённого звука

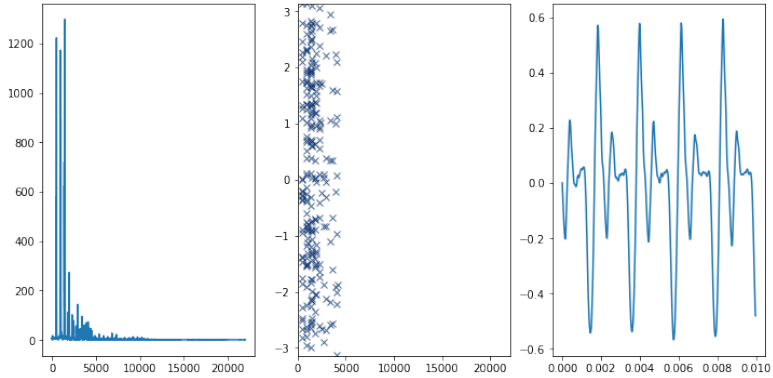

Рис. 3.13: Визуализация неизменённого звука

Здесь все углы установлены в ноль.

```
spectrum2 = zero_angle(spectrum)
plot_three(spectrum2, thresh=50)
```

Листинг 3.22: Визуализация при нулевых углах

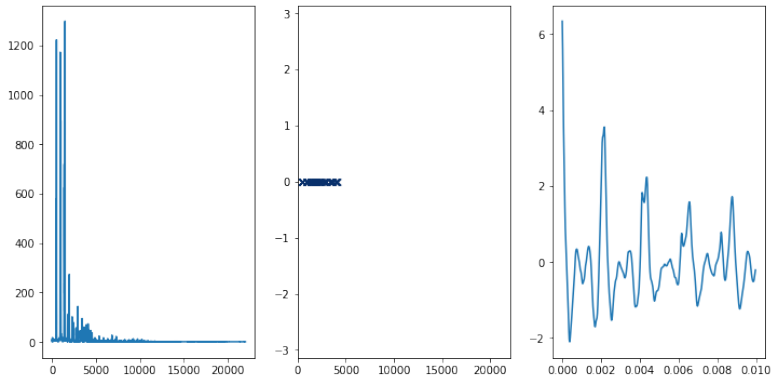

Рис. 3.14: Визуализация при нулевых углах

Теперь рассмотрим случай, когда углы повёрнуты на 1 радиан.

```
spectrum3 = rotate_angle(spectrum, 1)
plot_three(spectrum3, thresh=50)
```

Листинг 3.23: Визуализация при повороте углов

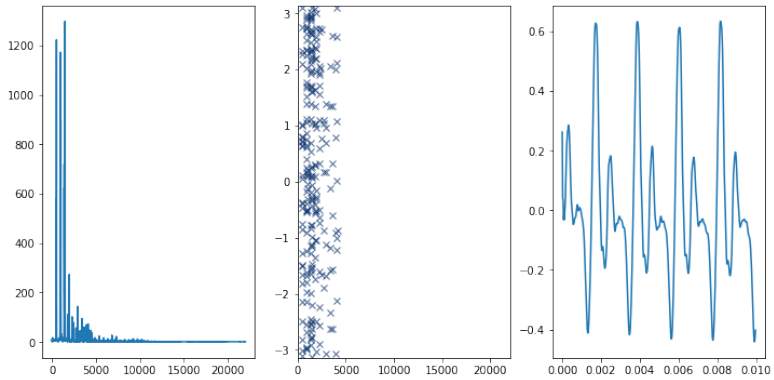

Рис. 3.15: Визуализация при повороте углов

Теперь рассмотрим случайные углы.

```
spectrum4 = random_angle(spectrum)
plot_three(spectrum4, thresh=50)
```

Листинг 3.24: Визуализация при случайных углах

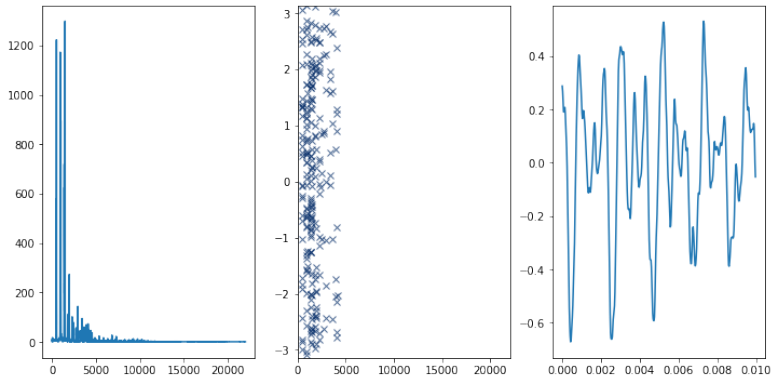

Рис. 3.16: Визуализация при случайных углах

Прослушав все звуки можно сделать вывод, что обнуление делает со звуком какие-то странные вещи, будто бы он стал каким-то "пьяным вра-

щение вновь не имеет никакого эффекта, а случайные углы добавляют вновь эффект реверберации.

Также я знаю, что саксофон отличается от других звуков тем, что основной компонент не является доминирующим. Я предполагаю, что для подобных звуков ухо использует что-то вроде автокорреляции в дополнение к спектральному анализу, и возможно, что этот вторичный режим анализа более чувствителен к фазовой структуре.

Если это так, эффект должен быть более сильным, если фундаментальный вообще отсутствует. Давайте в этом убедимся.

3.4 Урезание фундаментальных гармоник

```
spectrum.high_pass(600)
spectrum.plot(high=4000)
```

Листинг 3.25: Урезание частот звука

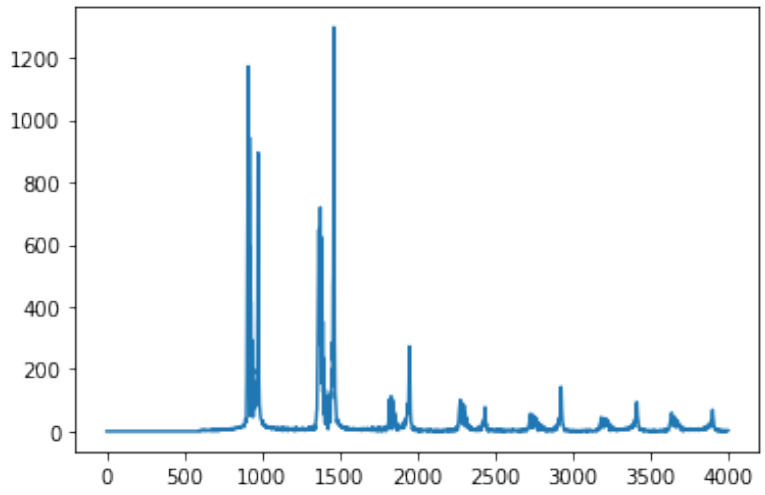

Рис. 3.17: Урезание частот звука

```
segment = wave.segment(start=2.5, duration=0.6)
spectrum = segment.make_spectrum()
plot_three(spectrum, thresh=50)
```

Листинг 3.26: Визуализация неизменённого звука

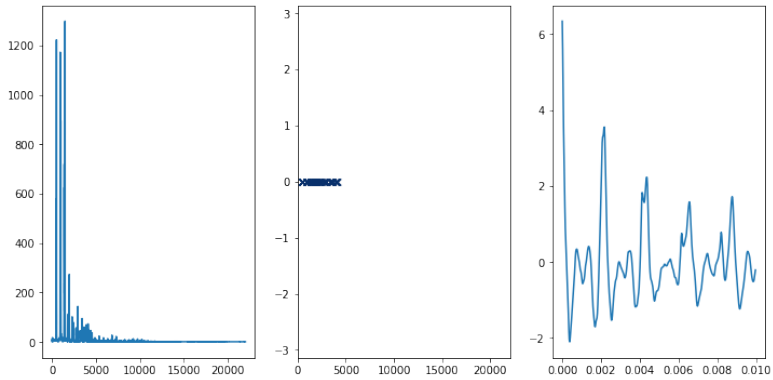

Рис. 3.18: Визуализация неизменённого звука

Здесь все углы установлены в ноль.

```
spectrum2 = zero_angle(spectrum)
plot_three(spectrum2, thresh=50)
```

Листинг 3.27: Визуализация при нулевых углах

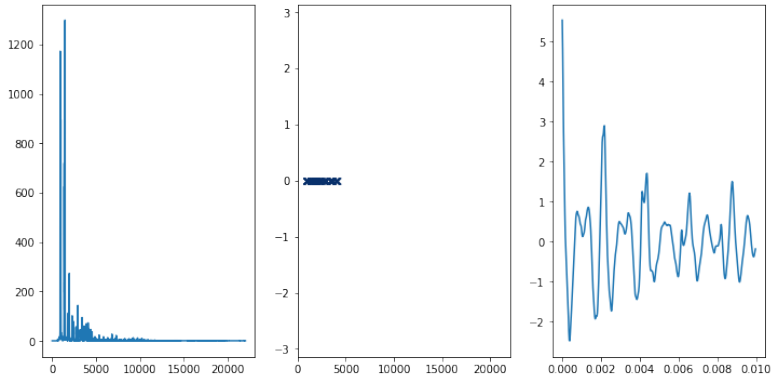

Рис. 3.19: Визуализация при нулевых углах

Теперь рассмотрим случай, когда углы повёрнуты на 1 радиан.

```
spectrum3 = rotate_angle(spectrum, 1)
plot_three(spectrum3, thresh=50)
```

Листинг 3.28: Визуализация при повороте углов

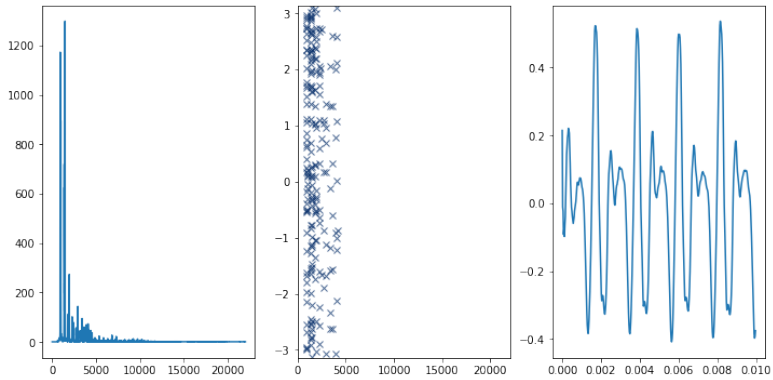

Рис. 3.20: Визуализация при повороте углов

Теперь рассмотрим случайные углы.

```
spectrum4 = random_angle(spectrum)
plot_three(spectrum4, thresh=50)
```

Листинг 3.29: Визуализация при случайных углах

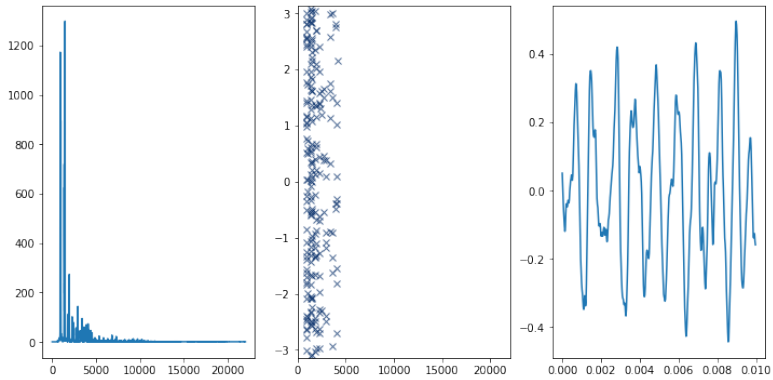

Рис. 3.21: Визуализация при случайных углах

Для этого сегмента изменение фазовой структуры имеет слышимый эффект, особенно случайные углы.

Если ухо использует что-то вроде автокорреляции для анализа подобных звуков, мы можем ожидать увидеть изменения в функции автокорреляции при изменении фазовой структуры.

3.5 Построение АКФ

Следующие функции строят АКФ для этих сегментов:

```
def autocorr(segment):
    corrs = np.correlate(segment.ys, segment.ys, mode='same')
    N = len(corrs)
    lengths = range(N, N//2, -1)

    half = corrs[N//2:].copy()
    half /= lengths
    half /= half[0]
    return half
```

Листинг 3.30: Функция `autocorr`

```
def plot_acf(spectrum):
    corrs = autocorr(spectrum.make_wave())
    thinkplot.plot(corrs[:200], linewidth=1)
```

Листинг 3.31: Функция `plot_acf`

```
plot_acf(spectrum)
plot_acf(spectrum2)
plot_acf(spectrum3)
plot_acf(spectrum4)
thinkplot.config(xlabel='Lag', ylabel='Correlation', ylim=[-1.05,
    1.05])
```

Листинг 3.32: Построение АКФ

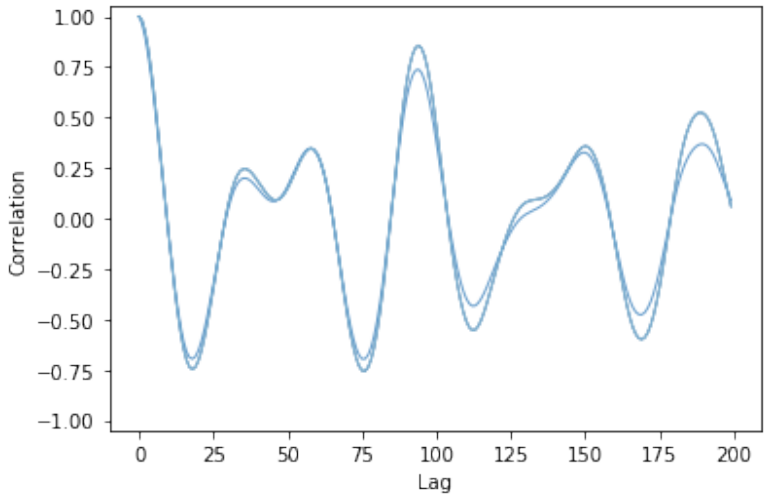

Рис. 3.22: Построение АКФ

Как мне кажется, изменение фазовой структуры оказывает некоторое влияние на АКФ, но не так сильно, и там нет ничего, что явно объясняет изменения в воспринимаемом звуке.

По крайней мере, для таких звуков, которые имеют простую гармоническую структуру, мы не слышим изменений в фазовой структуре, при условии, что гармоническая структура не изменилась.

Возможное исключение - звуки с низкой амплитудой на основной частоте. В этом случае мы используем что-то вроде автокорреляции для восприятия высоты звука, и это намек на то, что этот анализ может быть более чувствительным к фазовой структуре. Однако в АКФ нет ничего очевидного, объясняющего этот эффект.

Глава 4

Выводы

Во время выполнения лабораторной работы получены навыки работы с прямым и обратным дискретным косинусным преобразованием. Также получены навыки их практического применения.