Санкт-Петербургский Государственный Политехнический Университет
Институт Компьютерных Наук и Технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Отчёт по лабораторной работе №6
на тему
Дискретное косинусное преобразование

Работу выполнил
Студент группы 3530901/80203
Тарасенко Н.С.
Преподаватель
Богач Н.В.

Санкт-Петербург, 2021 год

1 Настройка проекта

Перед тем как выполнять задания необходимо настроить проект и сделать все необходимые импорты:

```
from __future__ import print_function, division

import thinkdsp
import thinkplot
import thinkstats2
import matplotlib.pyplot as plt

import numpy as np
import scipy.fftpack

import warnings
warnings.filterwarnings('ignore')

import dct

%matplotlib inline
```

Рис. 1: 2

2 Упражнение номер №1

Убедиться, что analyze1 требует времени пропорционально n^3, $analyze2$ n^2.

Начнём с шумового сигнала и массива величин степени двойки:

```
from thinkdsp import UncorrelatedGaussianNoise

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

```
(16384,)
```

Рис. 2: 2

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

```python
from scipy.stats import linregress

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

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

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

    return slope
```

```python
PI2 = np.pi * 2

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

```python
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.ys[:N]
        result = %timeit -r1 -o func(ys, freqs, ts)
        results.append(result)

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

```python
ns = 2 ** np.arange(6, 13)
ns
```

Рис. 3: 2

Выведем результаты для analyze1:

```
[ ]: bests = run_speed_test(ns, analyze1)
     plot_bests(ns, bests)
```

```
64
124 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops eac
h)
128
312 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops eac
h)
256
1.46 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops eac
h)
512
5.46 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops eac
h)
1024
28.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2048
208 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
4096
1.01 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
[ ]: 2.215896159852855
```

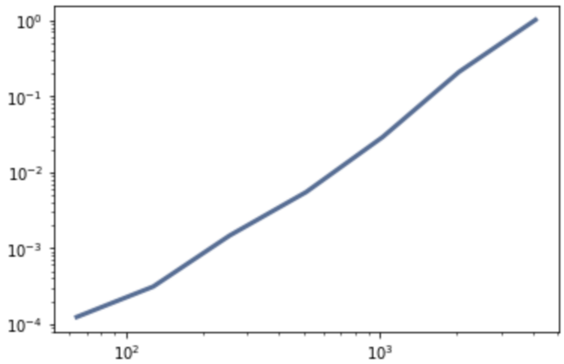


Рис. 4: 2

Расчетный наклон близок к 2, а не к 3, как ожидалось. Одна из возможностей состоит в том, что производительность np.linalg.solve почти квадратична в этом диапазоне размеров массива.

Выведем результаты для analyze2:

```
[ ]: bests2 = run_speed_test(ns, analyze2)
     plot_bests(ns, bests2)
```

```
64
59.2 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops ea
ch)
128
228 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops eac
h)
256
836 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops eac
h)
512
3.84 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops eac
h)
1024
16.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops eac
h)
2048
66.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
4096
258 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
[ ]: 2.0327951145865506
```

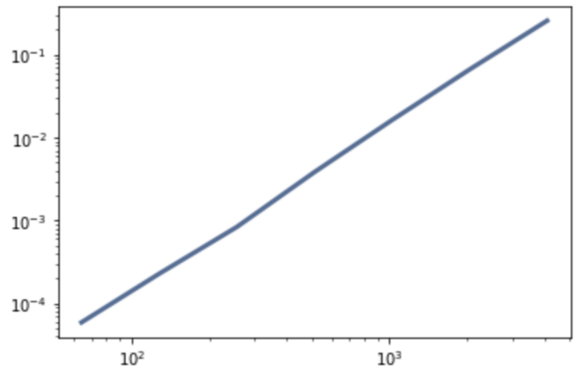


Рис. 5: 2

Как и ожидалось, результаты для analysis2 попадают в прямую линию с предполагаемым наклоном, близким к 2.

Вот результаты для scipy.fftpack.dct

```
64
8.51 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops e
ach)
128
9.25 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops e
ach)
256
9.77 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops e
ach)
512
10.3 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops e
ach)
1024
14.1 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100000 loops e
ach)
2048
24.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops ea
ch)
4096
46 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops eac
h)
```

```
]: 0.3809384112664403
```

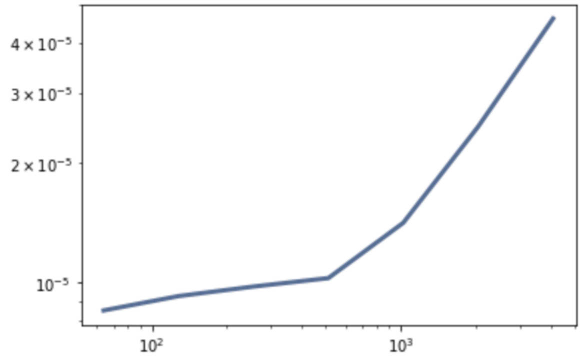


Рис. 6: 2

Эта реализация dct еще быстрее. Линия изогнута, что означает, что либо мы еще не видели асимптотическое поведение, либо асимптотическое поведение не является простым показателем. Фактически, как мы скоро увидим, время выполнения пропорционально logn.

На следующем рисунке показаны все три кривые на одних и тех же осях.

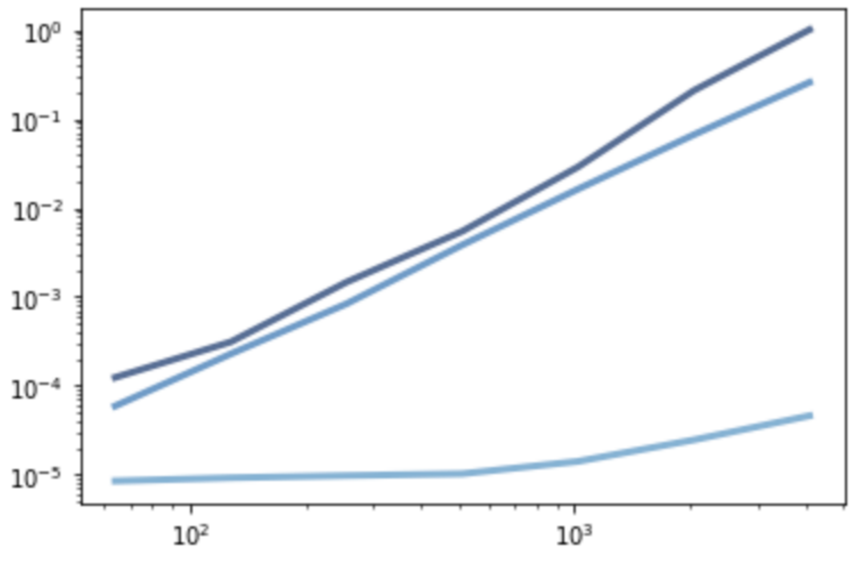


Рис. 7: 2

3 Упражнение номер №2

Реализовать алгоритм описанный в учебнике в данной главе. Проверить на записи сколько можно компонентов удалить до того как разница станет заметной.

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

Возьмем запись саксофона из репозитория, выделим небольшой сегмент, построим DCT этого сегмента:

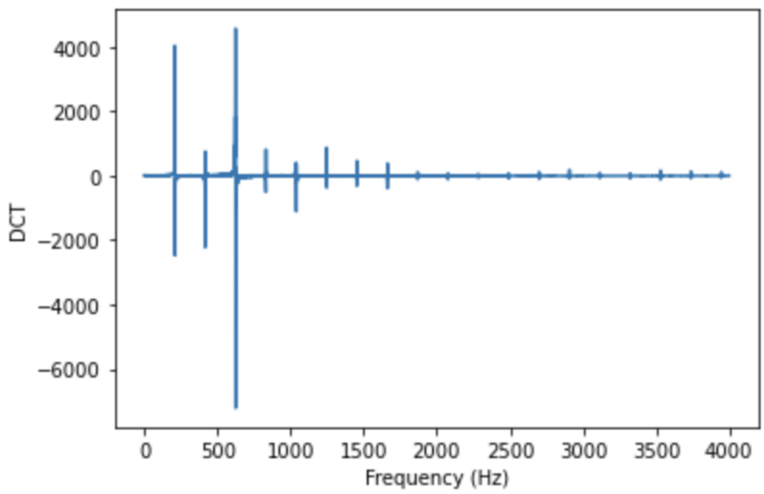


Рис. 8: 2

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

```python
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')
```

Рис. 9: 2

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

```
20292	22050	92.02721088435374
```

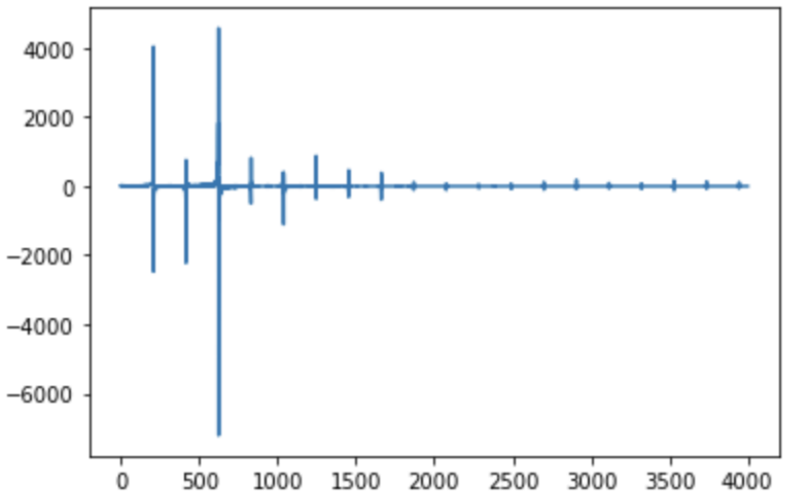


Рис. 10: 2

Результат схож с изначальным

Чтобы сжать более длинный сегмент, мы можем сделать спектрограмму ДКП. Следующая функция похожа на wave.make$_{s}pectrogram$, .

```python
from thinkdsp import Spectrogram

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)
```

Рис. 11: 2

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

```
1018    1024    99.4140625
1016    1024    99.21875
1014    1024    99.0234375
1017    1024    99.31640625
1016    1024    99.21875
1017    1024    99.31640625
1016    1024    99.21875
1020    1024    99.609375
1014    1024    99.0234375
1005    1024    98.14453125
1009    1024    98.53515625
1015    1024    99.12109375
1015    1024    99.12109375
1016    1024    99.21875
1016    1024    99.21875
1015    1024    99.12109375
1017    1024    99.31640625
1020    1024    99.609375
1013    1024    98.92578125
```

Рис. 12: 2

В большинстве сегментов сжатияие составляет 75-80 процентов.

Чтобы услышать, как это звучать, мы можем преобразовать спектрограмму обратно в волну и воспроизвести ее. При сжатии слышно характерный треск во время воспроизведения аудио, так что можно смело сказать, что нам удалось сжать аудиозапись.

4 Упражнение номер №3

Изучить блокнот phase.ipynb

Используем сигнал с пилообразной формой волны. Выведем спектр:

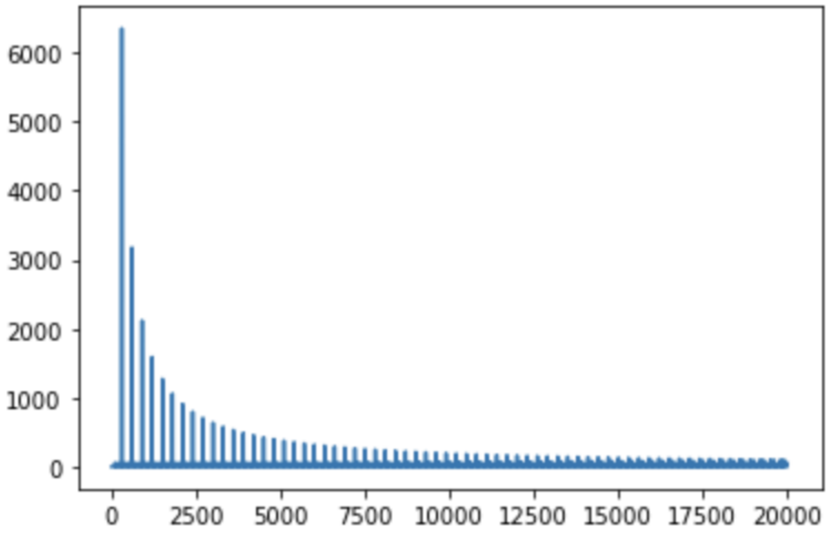
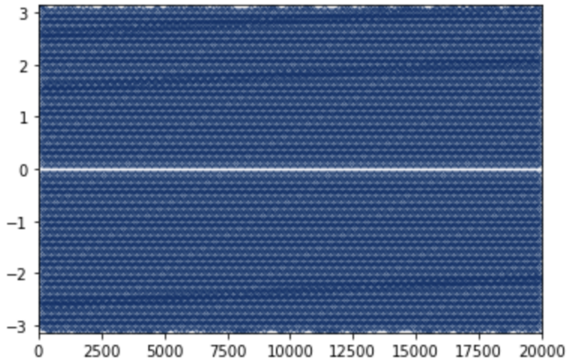


Рис. 13: 2

Рассмотрим угловую часть спектра.

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




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

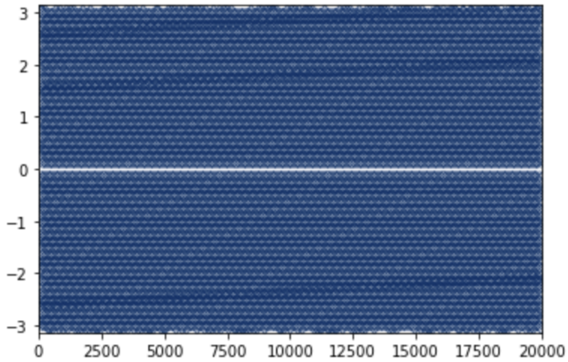


Рис. 14: 2

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.

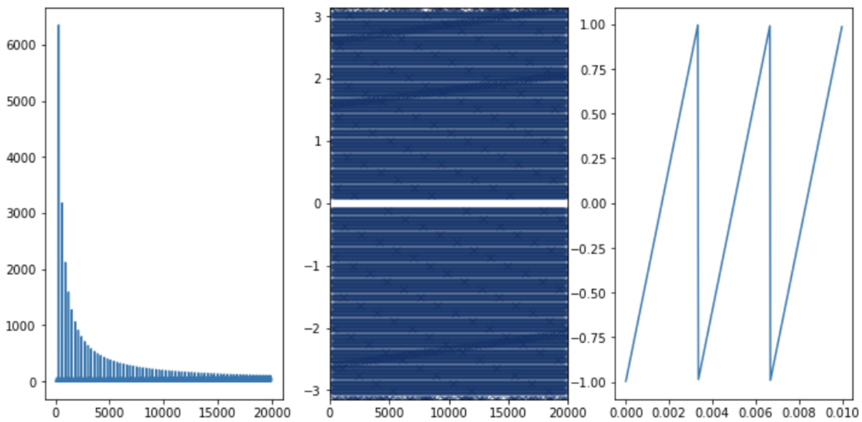


Рис. 15: 2

С помощью функции отобразим неизмененный спектр

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

Рис. 16: 2

Рассмотрим ситуация когда все углы устанавливаются в 0, для этого напишем метод позволяющий это сделать.

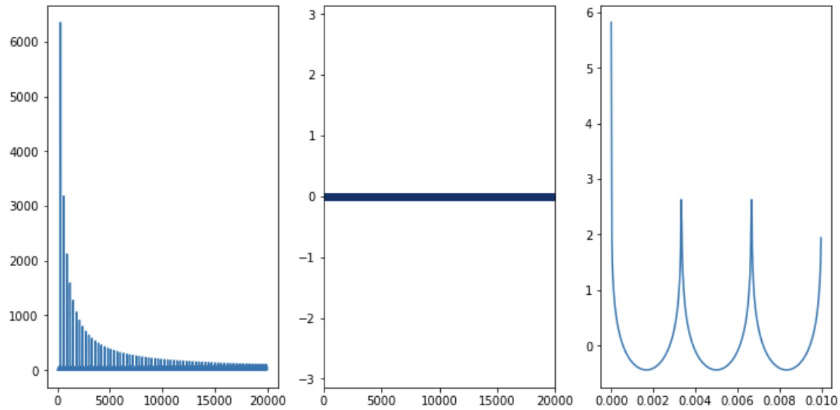


Рис. 17: 2

Если мы умножим комплексные компоненты на exp(iφ) , это приведет к добавлению φ к углам:

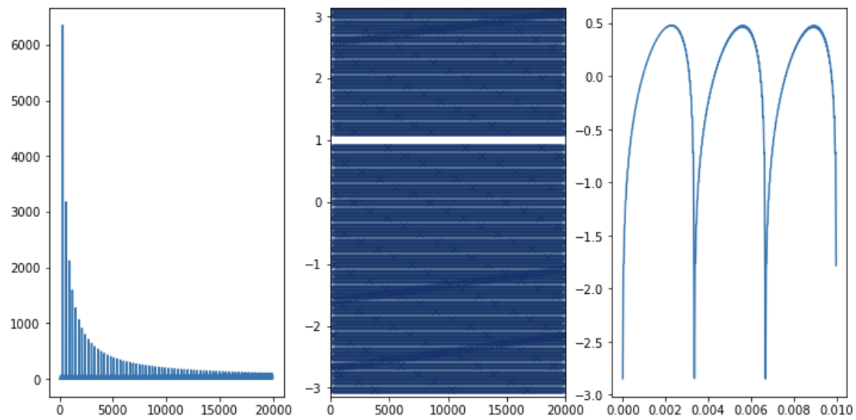


Рис. 18: 2

Рассмотрим ситуация когда все углы устанавливаются в случайные значения, для этого напишем метод позволяющий это сделать.

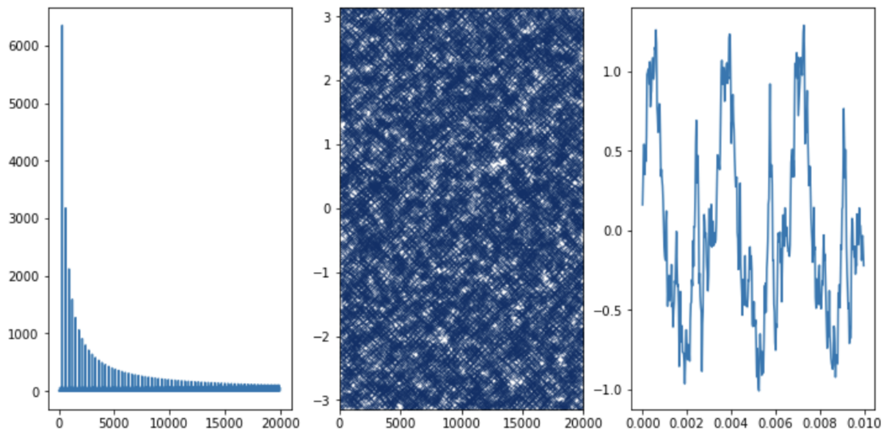


Рис. 19: 2

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

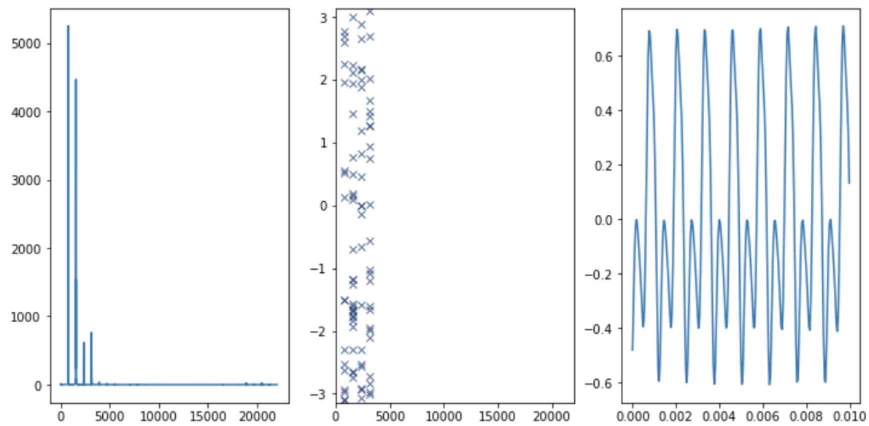


Рис. 20: 2

Здесь все углы установлены в ноль:

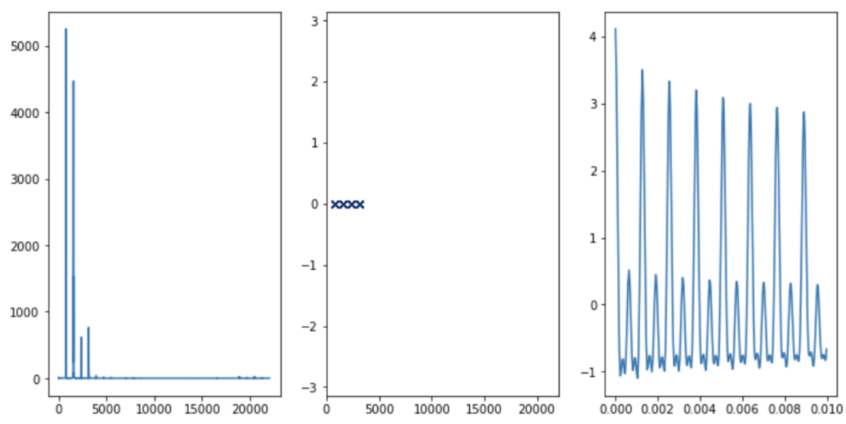


Рис. 21: 2

Здесь все углы повернуты на 1 радиан:

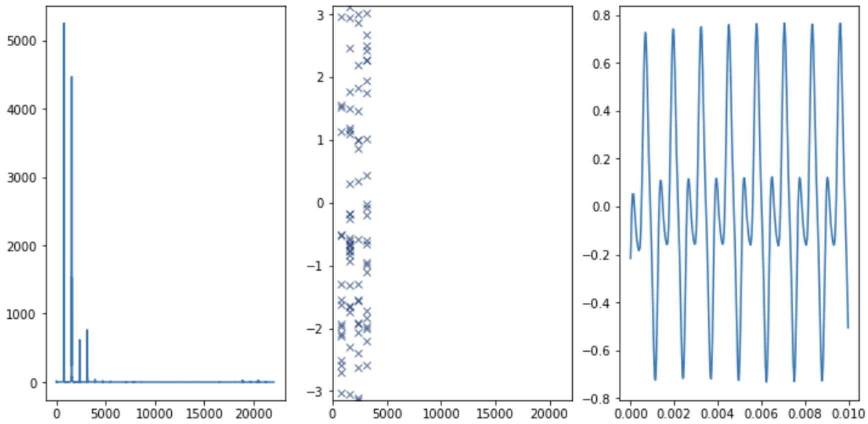


Рис. 22: 2

Здесь все углы принимают случайные значения.

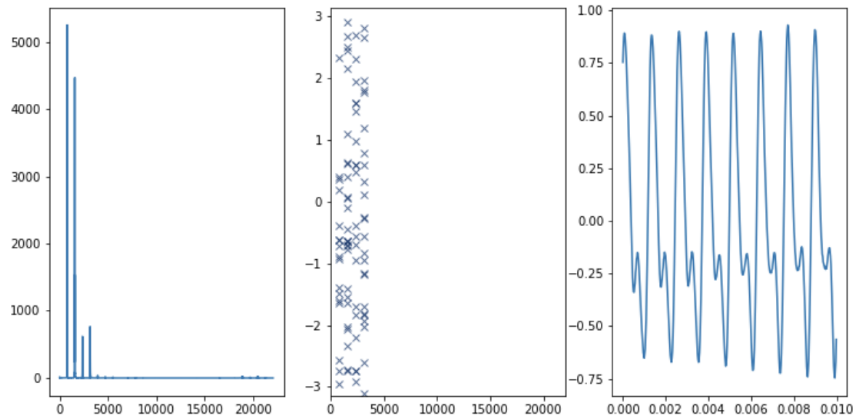


Рис. 23: 2

Установка углов в ноль понижает уровень громкости, поворот углов на радиан не вызвал особых изменений. В случае со случайными значениями появились эффекты ревебрации.

Проделаем те же операции с небольшим сегментом звука саксофона:

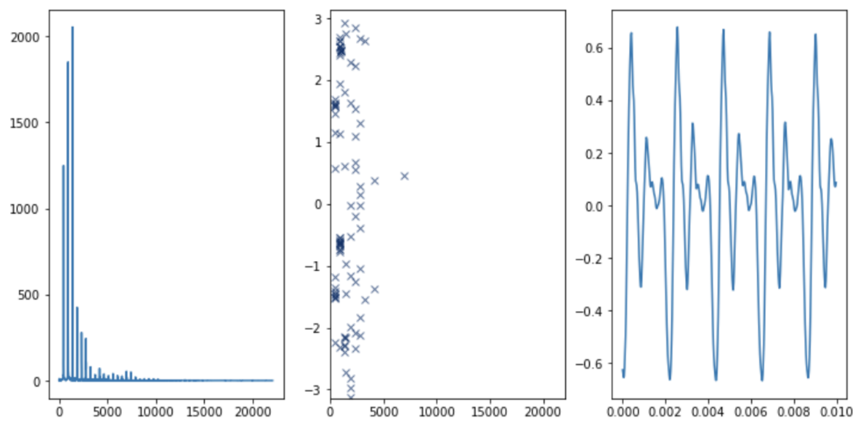


Рис. 24: 2

Здесь все углы установлены в ноль:

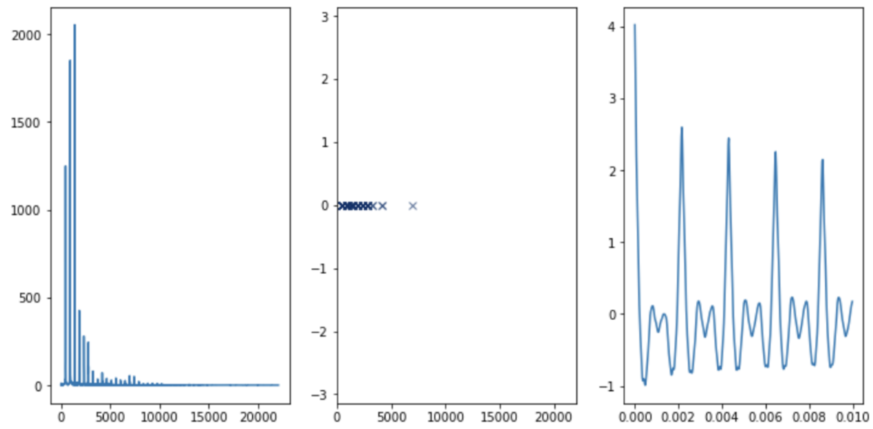


Рис. 25: 2

Здесь все углы повернуты на 1 радиан:

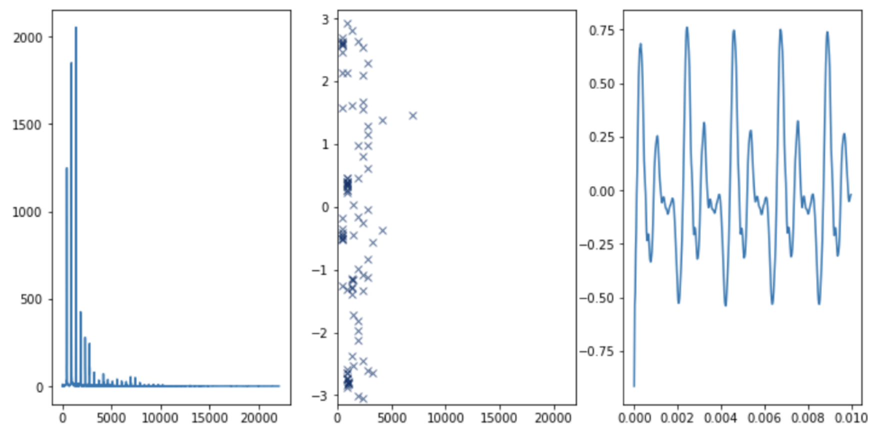


Рис. 26: 2

Здесь все углы принимают случайные значения.

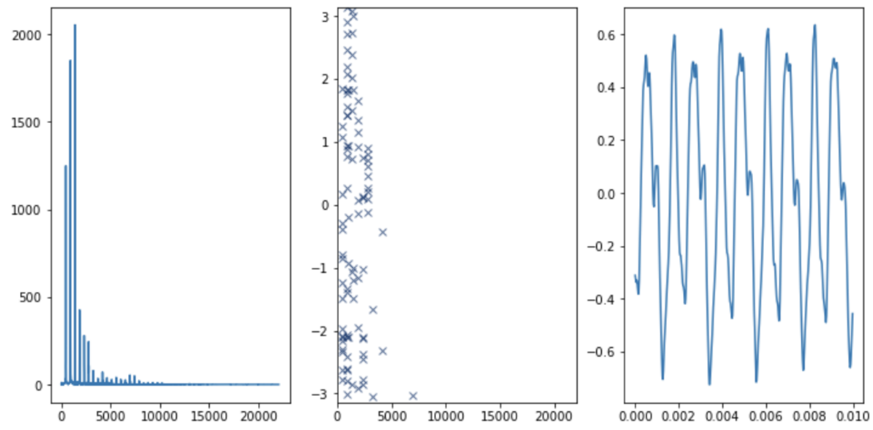


Рис. 27: 2

Таким образом после всех преобразований можно сделать вывод, что при установлении значений углов в 0 звук теряет небольшую часть громкости и становится тише, в случае с поворотом углом звук не изменяется, а при установлении случйных значений мы начинаем слышать шумы которые снижают качество звука (личная оценка)

Саксофон отличается от других звуков тем, что основной компонент не является доминирующим. 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.