

Санкт-Петербургский государственный политехнический
университет Петра Великого

**Высшая школа интеллектуальных систем и
суперкомпьютерных технологий**

Лабораторная работа

Дискретное косинусное преобразование

Работу выполнил студент
3-го курса, группа 3530901/80201
Солянкин Илья Андреевич

Преподаватель:
Богач Наталья Владимировна

Санкт-Петербург 2021

Содержание

1	Часть №1: <code>timeint</code>	5
2	Часть №2: Реализация алгоритма сжатия звука	9
3	Часть №3: <code>phase.ipynb</code>	13
4	Выводы	18

Список иллюстраций

1	Создание сигнала	5
2	Создание массива	5
3	Тестирование <code>analyze1</code>	6
4	Тестирование <code>analyze2</code>	7
5	Сравнение результатов	8
6	Получение звука	9
7	Получение сегмента	9
8	Полученный график амплитуды сегмента	10
9	Результат применения функции <code>compress</code> к сегменту	11
10	Получение сегмента	11
11	Получение спектограммы для всех сегментов сигнала	12
12	Получение спектограммы для всех сегментов сигнала	12
13	Результат вызова <code>plot-three</code> с <code>spectrum</code>	14
14	Результат вызова <code>plot-three</code> с <code>spectrum2</code>	15
15	Результат вызова <code>rotate-angle</code> с <code>spectrum3</code>	16
16	Результат вызова <code>random-angle</code> с <code>spectrum4</code>	17

Листинги

1	Функция plot-bests	5
2	Функция analyze1	5
3	Функция analyze2	6
4	Сравнение результатов	7
5	Получение графика амплитуды сегмента	9
6	Функция compress	10
7	Применение функции compress к сегменту	10
8	Класс make-dct-spectrogram	11
9	Функция plot-angle	13
10	Функция plot-three	13
11	Вызов plot-three с spectrum	13
12	Функция zero-angle	14
13	Вызов plot-three с spectrum2	14
14	Функция rotate-angle	15
15	Вызов plot-three с spectrum3	15
16	Функция random-angle	16
17	Вызов plot-three с spectrum4	16

1 Часть №1: timeint

В первой части шестой лабораторной работы нам необходимо проверить тот факт, что `analyze1` требует времени пропорционально n^2 , а `analyze2` пропорционально n^3 путем запуска их с несколькими разными массивами. Для этого необходимо воспользоваться `timeint`

Для этого сначала создадим сигнал на основе некоррелируемом гауссовском шуме:

```
In [2]: signal = UncorrelatedGaussianNoise()
noise = signal.make_wave(duration=1.0, framerate=16384)
noise.ys.shape

Out[2]: (16384,)
```

Рис. 1: Создание сигнала

Далее создаем массив с тестовыми данными:

```
In [3]: ns = 2 ** np.arange(6, 13)
ns

Out[3]: array([ 64, 128, 256, 512, 1024, 2048, 4096], dtype=int32)
```

Рис. 2: Создание массива

Теперь создадим функцию, которая будет строить результаты и рисовать прямую линию из массива результатов из временного эксперимента:

```
1 def plot_bests(bests):
2     thinkplot.plot(ns, bests)
3     thinkplot.config(xscale='log', yscale='log', legend=False)
4
5     x = np.log(ns)
6     y = np.log(bests)
7     t = linregress(x,y)
8     slope = t[0]
9
10    return slope
```

Листинг 1: Функция `plot-bests`

Сразу после этого создаем функцию `analyze1`

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

Листинг 2: Функция `analyze1`

Протестируем данную функцию и посмотрим получившиеся результаты:

```
B [6]: 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 analyze1(ys, freqs, ts)
    results.append(result)

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

```
125 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
426 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
1.78 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
9.63 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
53.7 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
224 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
1.03 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Out[6]: 2.2148335375849397

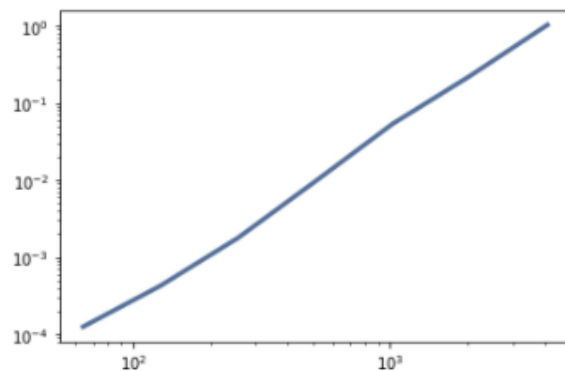


Рис. 3: Тестирование analyze1

Теперь создадим функцию analyze2

```
1 def analyze2(ys, fs, ts):
2     args = np.outer(ts, fs)
3     M = np.cos(PI2 * args)
4     amps = np.dot(M, ys) / 2
5     return amps
```

Листинг 3: Функция analyze2

И также протестируем ее:

```

B [8]: 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 analyze2(ys, freqs, ts)
    results.append(result)

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

```

```

76.2 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
260 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
1.52 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
7.68 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
22.5 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
84.6 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
316 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

```

Out[8]: 2.0226454501916873

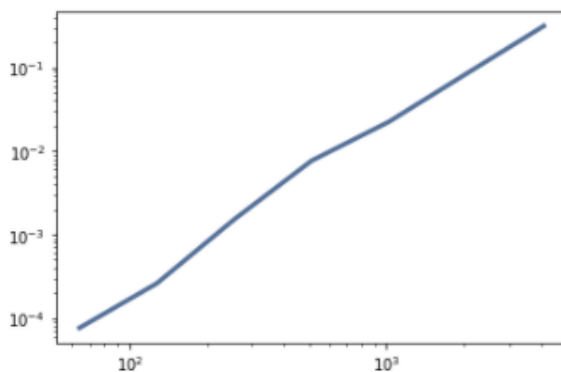


Рис. 4: Тестирование analyze2

Для более удобного сравнения отобразим оба графика на одном поле:

```

1 thinkplot.plot(ns, bests, label='analyze1')
2 thinkplot.plot(ns, bests2, label='analyze2')
3 decorate(xlabel='Wave length (N)', ylabel='Time (s)', **dict(xscale='log',
yscale='log'))

```

Листинг 4: Сравнение результатов

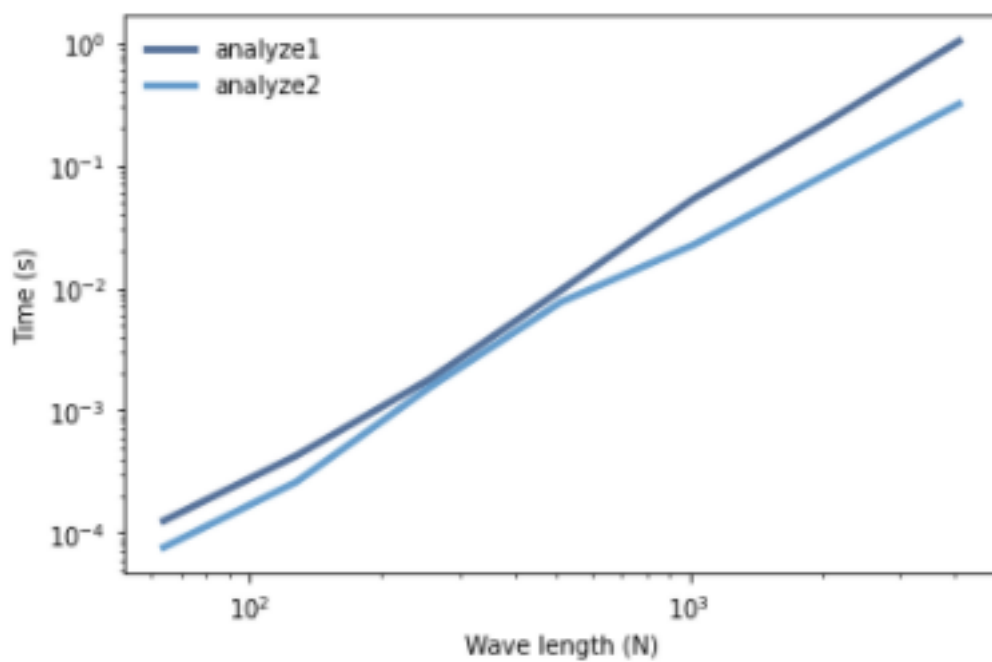


Рис. 5: Сравнение результатов

В результате можно сделать вывод, что `analyze2` работает быстрее, чем `analyze1`

2 Часть №2: Реализация алгоритма сжатия звука

Во втором пункте шестой лабораторной работы нам необходимо реализовать версию ДКП алгоритма для сжатия звука.

Начнем с того, что скачаем звук серены при воздушной атаке:

```
B [10]: wave = read_wave('412171_screamstudio_air-raid-siren-alarm.wav')
        wave.make_audio()
```

Out[10]:

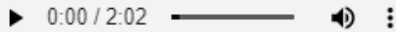


Рис. 6: Получение звука

Далее нам необходимо выделить некоторый сегмент. Я выбрал сегмент с 5 секунды длительностью 0,5 секунды:

```
B [11]: segment = wave.segment(start=5, duration=0.5)
        segment.normalize()
        segment.make_audio()
```

Out[11]:

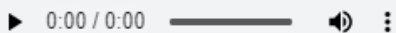


Рис. 7: Получение сегмента

После этого выведем график амплитуды нашего сегмента:

```
1 seg_dct = segment.make_dct()
2 seg_dct.plot(high=4000)
3 decorate(xlabel='Frequency (Hz)', ylabel='DCT')
```

Листинг 5: Получение графика амплитуды сегмента

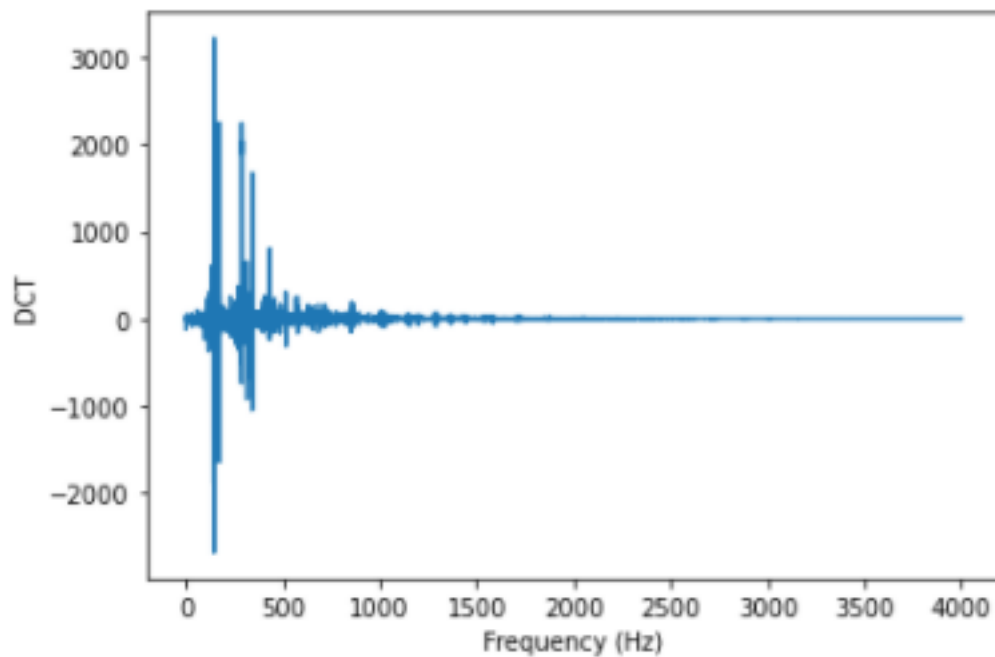


Рис. 8: Полученный график амплитуды сегмента

По графику видно, что в сегменте очень много точек с нулевой амплитудой. Напишем функцию `compress` для зануления элементов, которые ниже порога `thresh`:

```

1  def compress(dct, thresh=1):
2      count = 0
3      for i, amp in enumerate(dct.amps):
4          if np.abs(amp) < thresh:
5              dct.hs[i] = 0
6              count += 1
7
8      n = len(dct.amps)
9      print(count, n, 100 * count / n, sep='\t')
```

Листинг 6: Функция `compress`

Применим написанную функцию к нашему сегменту:

```

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

Листинг 7: Применение функции `compress` к сегменту

20236 22050 91.77324263038548

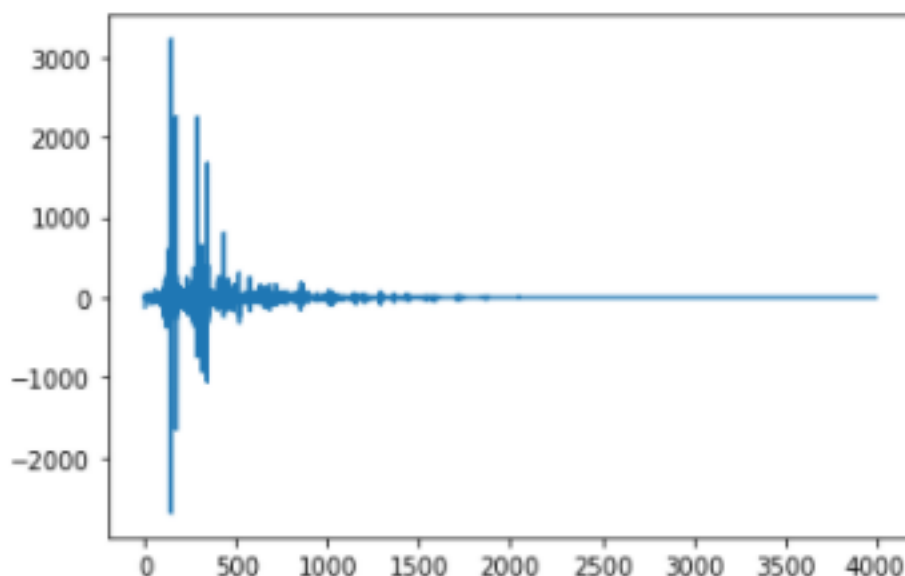


Рис. 9: Результат применения функции compress к сегменту

Визуально графики ничем не отличаются друг от друга.

Ради интереса создадим аудиодорожку из полученного сегмента после compress:

```
B [15]: seg2 = seg_dct.make_wave()  
        seg2.make_audio()
```

Out[15]:



Рис. 10: Получение сегмента

При прослушивании и сравнение полученной аудиодорожки с оригинальной можно сделать вывод, что, как мне показалось, появились некоторые шумы.

Чтобы сжать более длинный фрагмент нам необходимо написать класс, который будет делать спектограмму ДКП :

```
1 def make_dct_spectrogram(wave, seg_length):  
2     window = np.hamming(seg_length)  
3     i, j = 0, seg_length  
4     step = seg_length // 2  
5     spec_map = {}  
6  
7     while j < len(wave.ys):  
8         segment = wave.slice(i, j)  
9         segment.window(window)  
10
```

```

11         t = (segment.start + segment.end) / 2
12         spec_map[t] = segment.make_dct()
13
14         i += step
15         j += step
16
17     return Spectrogram(spec_map, seg_length)

```

Листинг 8: Класс make-dct-spectrogram

Теперь нам необходимо создать спектограмму ДКП и применить функцию `compress` к каждому сегменту:

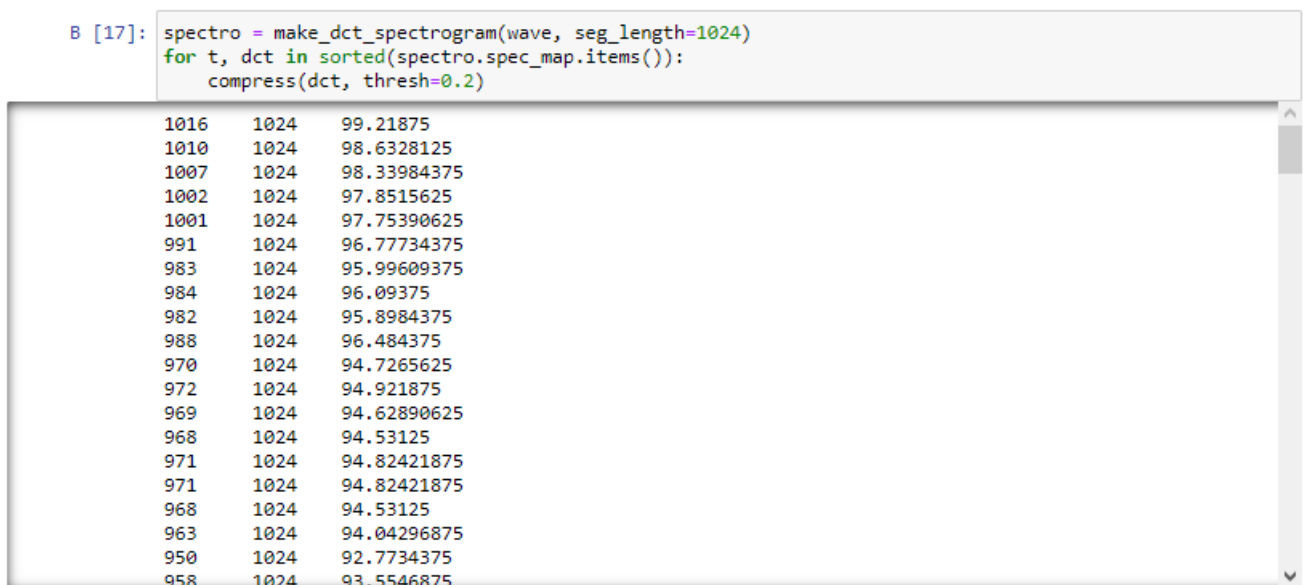


Рис. 11: Получение спектограммы для всех сегментов сигнала

Так как вывелось огромное количество сегментов, я привел в пример только часть из них. Все результаты можно увидеть в файле `Lab-6.ipynb`.

Наконец, переведем полученную спектограмму в сигнал, чтобы сравнить ее с исходным сигналом:



Рис. 12: Получение спектограммы для всех сегментов сигнала

При прослушивании исходного сигнала и полученного в результате, можно сделать вывод, что после всех манипуляций в сигнале появился шум, которым можно управлять с помощью взаимодействия с порогом `thres`.

3 Часть №3: phase.ipynb

В третьем пункте шестой лабораторной работы нам необходимо запустить блокнот phase.ipynb, пройтись по всем примерам, после чего выбрать любой другой сегмент и сделать с ним те же самые манипуляции.

В блокноте phase.ipynb содержится функция plot-angle, которая отображает амплитуды, форму волны и angle для спектра:

```
1 def plot_angle(spectrum, thresh=1):
2     angles = spectrum.angles
3     angles[spectrum.amps < thresh] = np.nan
4     thinkplot.plot(spectrum.fs, angles, 'x')
5     decorate(xlabel='Frequency (Hz)', ylabel='Phase (radian)')
```

Листинг 9: Функция plot-angle

Также возьмем уже написанную функцию plot-three, которая выводит на экран 3 графика и аудиодорожку из поданного сигнала:

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

Листинг 10: Функция plot-three

В качестве изначального сигнала возьмем запись гобоя из phase.ipynb и выделим сегмент с 2.3 секунды длительностью 0.7 секунды, после чего сразу вызовем plot-three с этим сегментом:

```
1 wave = read_wave('120994__thirsk__120-oboe.wav')
2 wave.make_audio()
3 segment = wave.segment(start=2.3, duration=0.7)
4
5 spectrum = segment.make_spectrum()
6 plot_three(spectrum, thresh=50)
```

Листинг 11: Вызов plot-three с spectrum

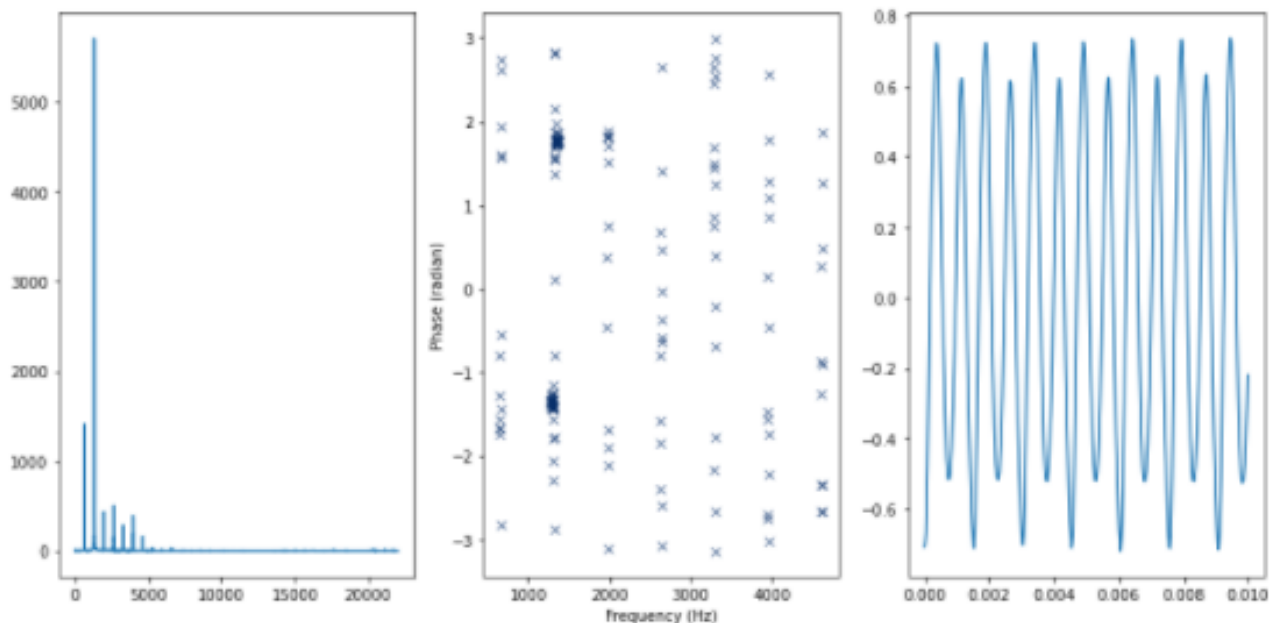


Рис. 13: Результат вызова plot-three с spectrum

Теперь возьмем функцию zero-angle, которая выдает результат, в котором $\text{angle} = 0$

```
1 def zero_angle(spectrum):
2     res = spectrum.copy()
3     res.hs = res.amps
4     return res
```

Листинг 12: Функция zero-angle

После этого создадим spectrum2, вызвав zero-angle с spectrum:

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

Листинг 13: Вызов plot-three с spectrum2

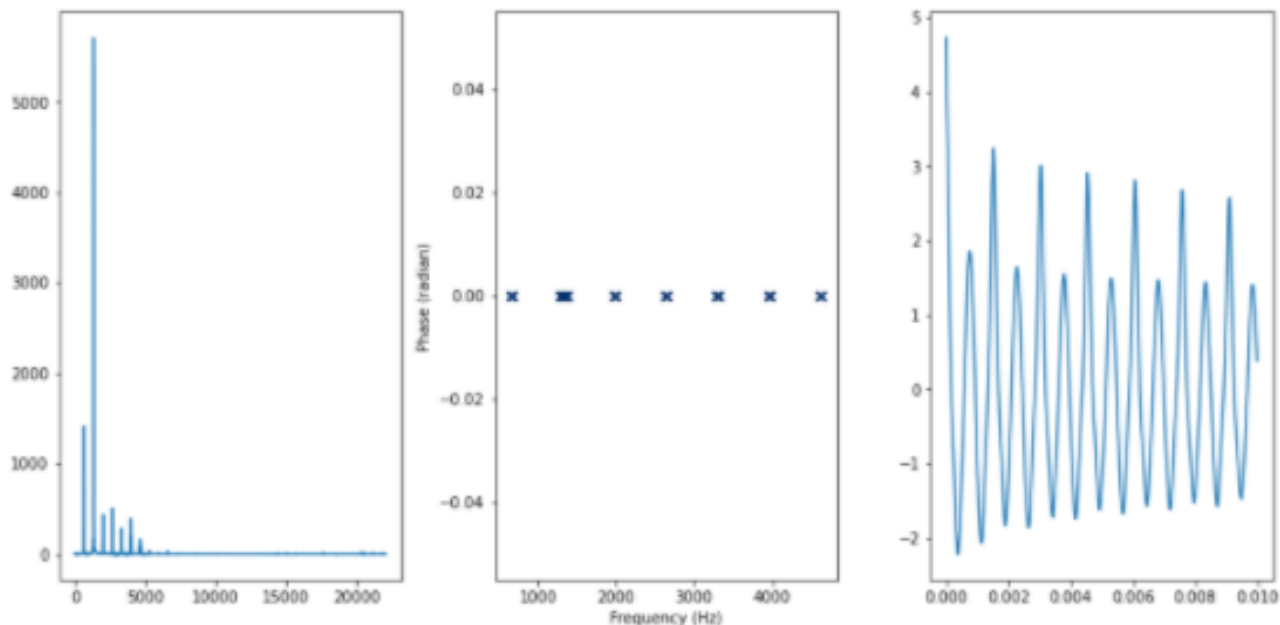
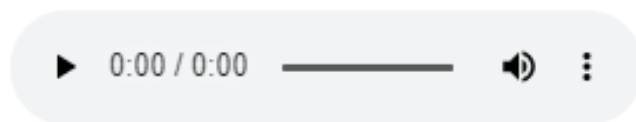


Рис. 14: Результат вызова plot-three с spectrum2

Теперь возьмем функцию rotate-angle, которая выдает результат, в котором angle изменен на 1 радиан:

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

Листинг 14: Функция rotate-angle

После этого создадим spectrum3, вызвав rotate-angle с spectrum:

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

Листинг 15: Вызов plot-three с spectrum3

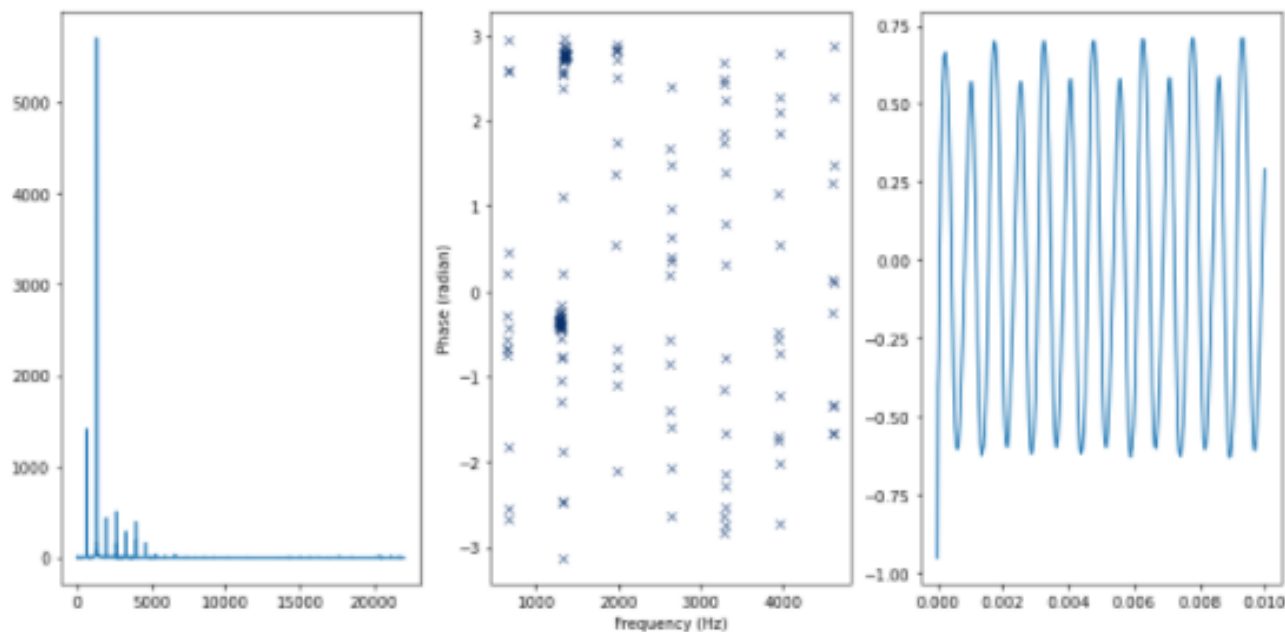


Рис. 15: Результат вызова rotate-angle с spectrum3

Теперь возьмем функцию random-angle, которая выдает результат, в котором angle имеет случайное значение:

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

Листинг 16: Функция random-angle

После этого создадим spectrum4, вызвав random-angle с spectrum:

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

Листинг 17: Вызов plot-three с spectrum4

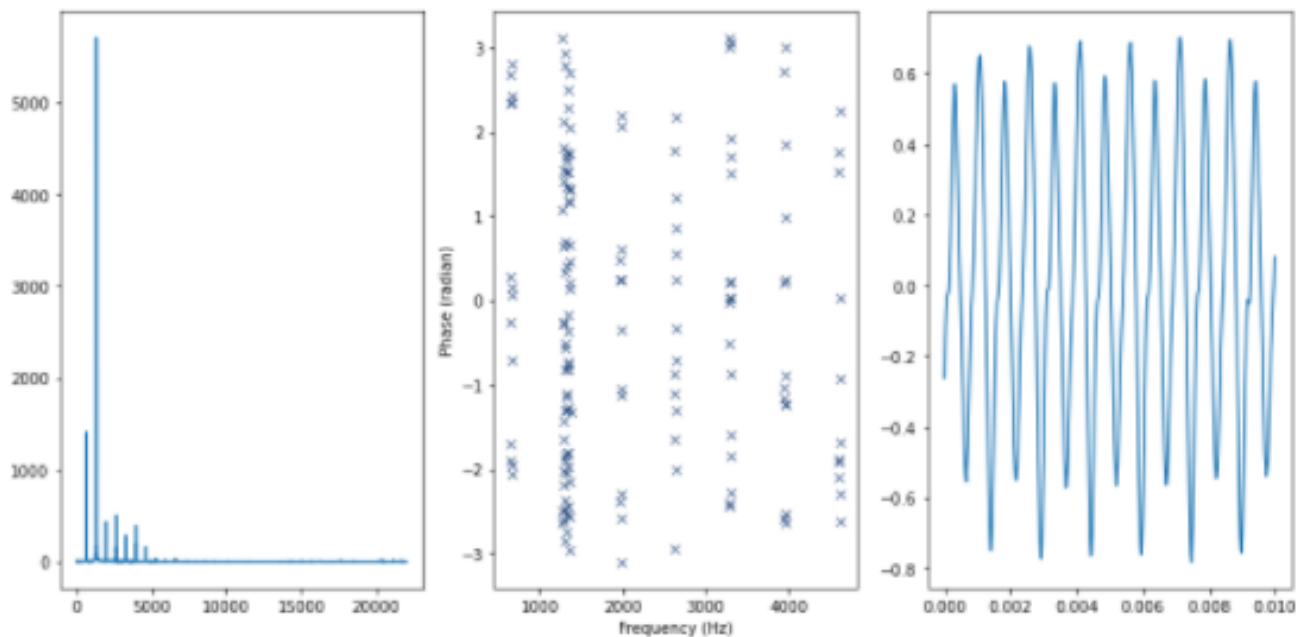
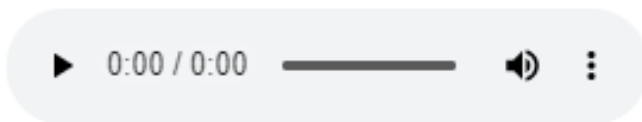


Рис. 16: Результат вызова random-angle с spectrum4

В результате можно сказать, что рандомизация добавила глухой эффект, а также, что изменение angle почти не влияет на конечный сигнал.

4 Выводы

В результате выполнения данной лабораторной работы мы изучили, что такое ДКС. Научились синтезировать ее и анализировать. Были проверены функции `analyze1` и `analyze2`, Мы вычислили какая из этих функций работает быстрее и на сколько. Также создали функцию `make-dct-spectrogram` для сжатия звуковой дорожки и сразу ее проверили. Наконец, мы поработали с блокнотом `phase.ipynb`, пройдя по всем примерам с другим сегментом.