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

Отчёт по лабораторной работе №6
Дисциплина: Телекоммуникационные технологии
Тема: Дискретное косинусное преобразование

Работу выполнил:
Ляшенко В.В.
Группа: 3530901/80201
Преподаватель:
Богач Н.В.

Санкт-Петербург
2021

Оглавление

Список иллюстраций

Листинги

Глава 1

Упражнение 6.1

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.

```
from thinkdsp import UncorrelatedGaussianNoise

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

Листинг 1.1: Создание шумового сигнала

Затем построим массив из степеней двойки.

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

Листинг 1.2: Создание массива

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

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

Листинг 1.3: Функция run_speed_test

```
from scipy.stats import linregress

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

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

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

    return slope
```

Листинг 1.4: Функция plot_bests

Используя функцию `run_speed_test`, получим результаты работы `analyze1` (Рис.1.1), а затем с помощью `plot_bests` построим график на основании полученных данных.

```
res1 = run_speed_test(ns, analyze1)
plot_bests(ns, res1)
```

Листинг 1.5: Получение результатов для analyze1

```
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. 1.1: Результаты analyze1

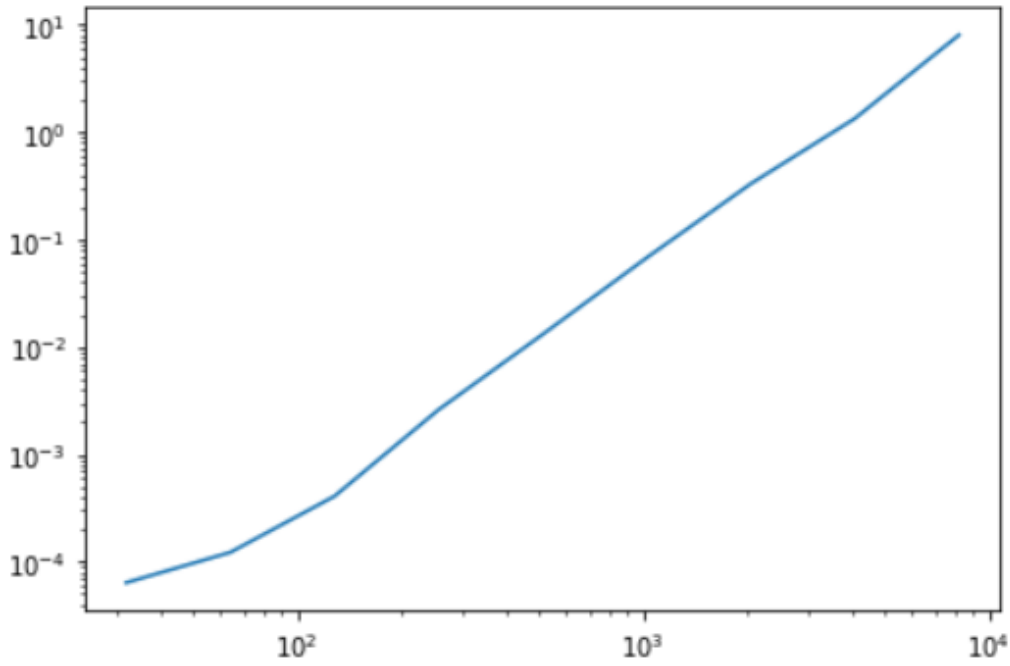

Рис. 1.2: График analyze1

Наклон полученного графика равен 2.2 (Рис.1.2), что не соотвествует 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.1.3).

```
res2 = run_speed_test(ns, analyze2)
plot_bests(ns, res2)
```

Листинг 1.6: Получение результатов для analyze2

```
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. 1.3: Результаты analyze2

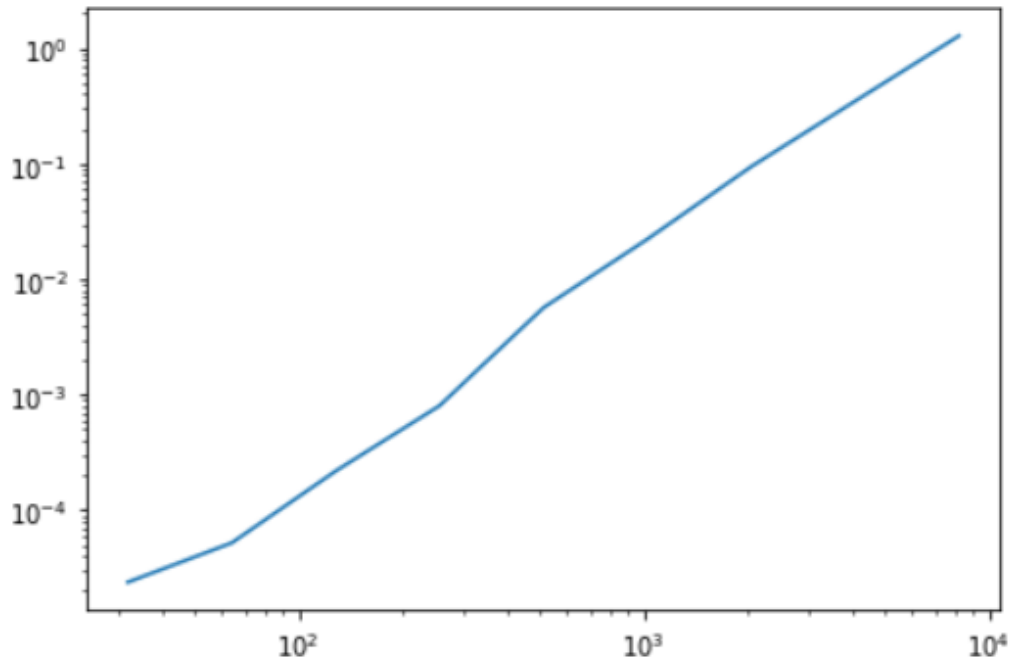

Рис. 1.4: График analyze2

Наклон данного графика равен 2.1 (Рис.1.4), что соотвествует нашим ожиданиям. Также посмотрим на функцию `scipy.fftpack.dct` (Рис.1.5).

```
import scipy.fftpack

def scipy_dct(ys, freqs, ts):
    return scipy.fftpack.dct(ys, type=3)

res3 = run_speed_test(ns, scipy_dct)
plot_bests(ns, res3)
```

Листинг 1.7: Получение результатов для scipy.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. 1.5: Результаты scipy.fftpack.dct

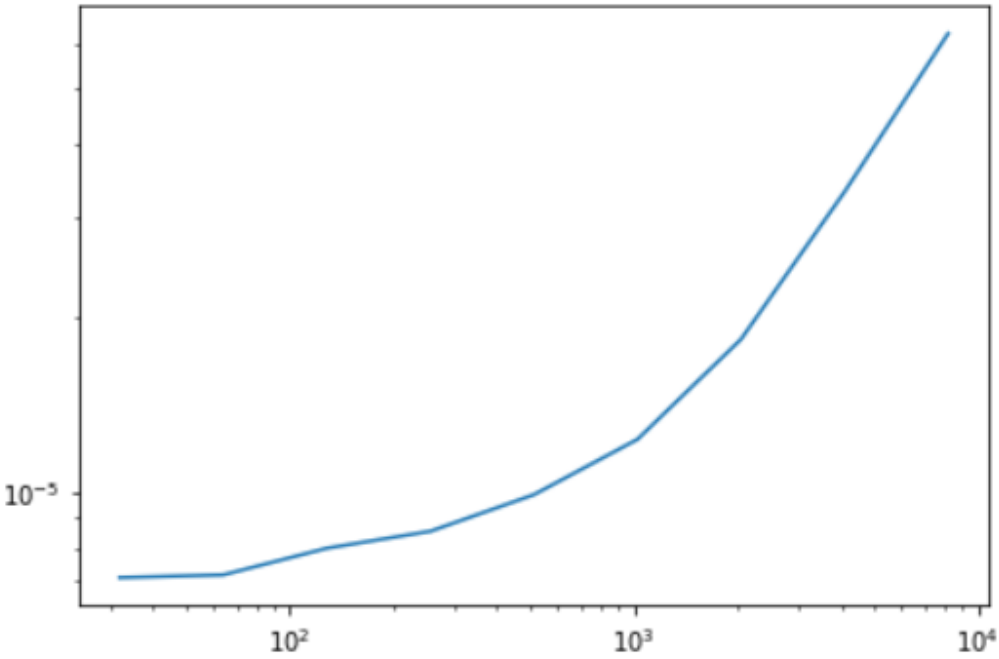

Рис. 1.6: График scipy.fftpack.dct

Данная функция работает еще быстрее. Как мы можем видеть на рис.1.6, её график изогнут, что означает, либо что мы еще не видели асимптотическое поведение, либо асимптотическое поведение не является простым показателем n. Ее выполнения пропорционально $n \log n$.

Сравним полученные результаты (Рис.1.7).

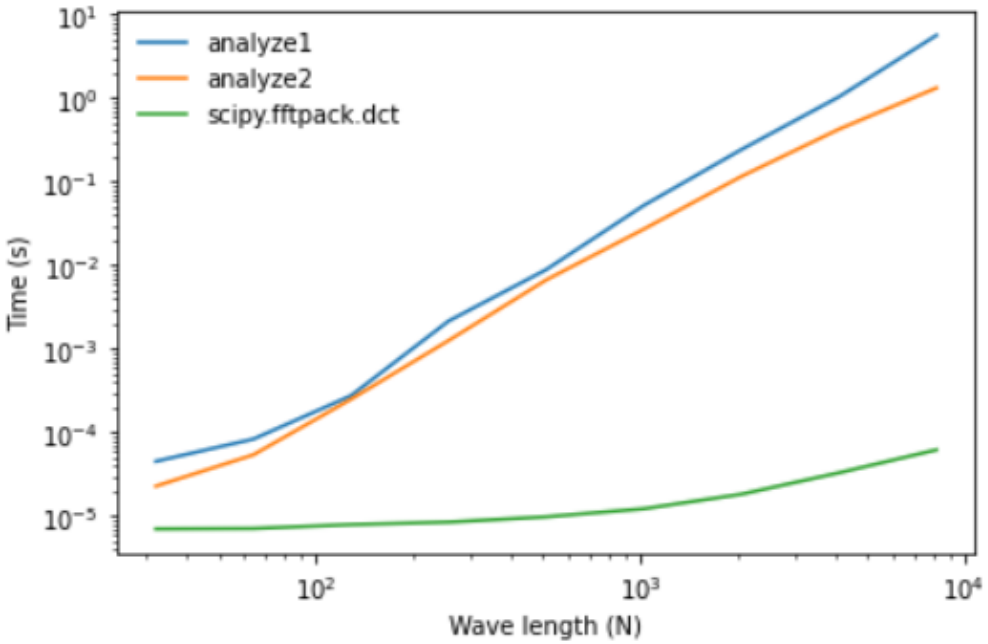

Рис. 1.7: Сравнение трех функций

Глава 2

Упражнение 6.2

2.1 Алгоритм сжатия

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.

2.2 Работа с коротким сегментом

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

```
from thinkdsp import read_wave

wave = read_wave('100475__iluppai__saxophone-weep.wav')
segment = wave.segment(start=1.2, duration=0.5)
segment.normalize()
segment.make_audio()
```

Листинг 2.1: Выделение сегмента

Теперь вычислим ДКП этого сегмента.

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

Листинг 2.2: Приминение ДКП к сегменту

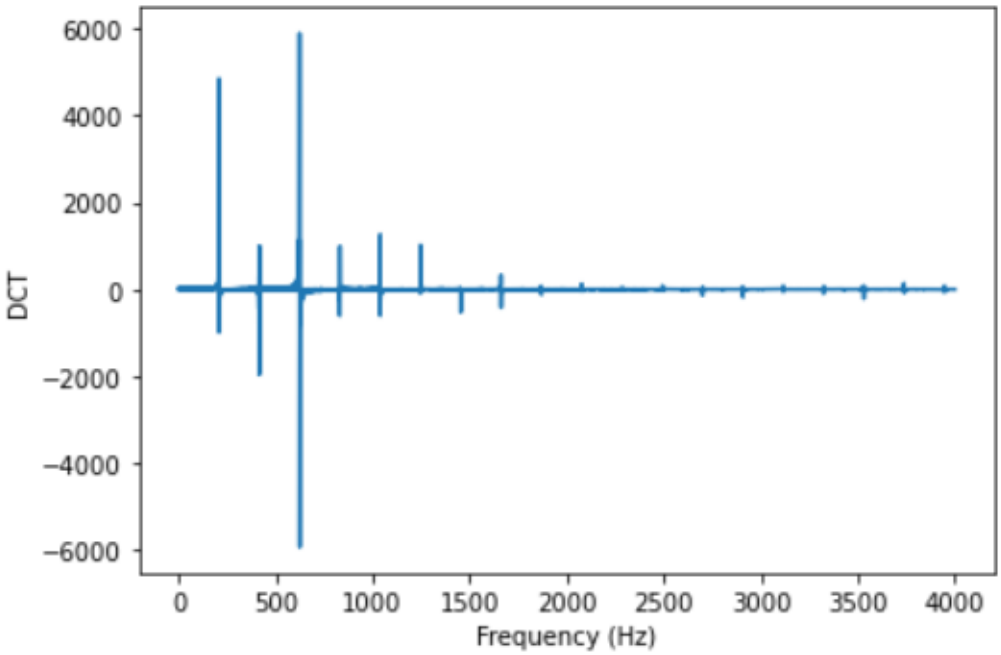

Рис. 2.1: ДКП сегмента

Как мы можем видеть на рис.2.1, 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(dct.amps):
        if np.abs(amp) < thresh:
            dct.hs[i] = 0
            count += 1

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

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

Листинг 2.3: Удаление неслышимых компонентов

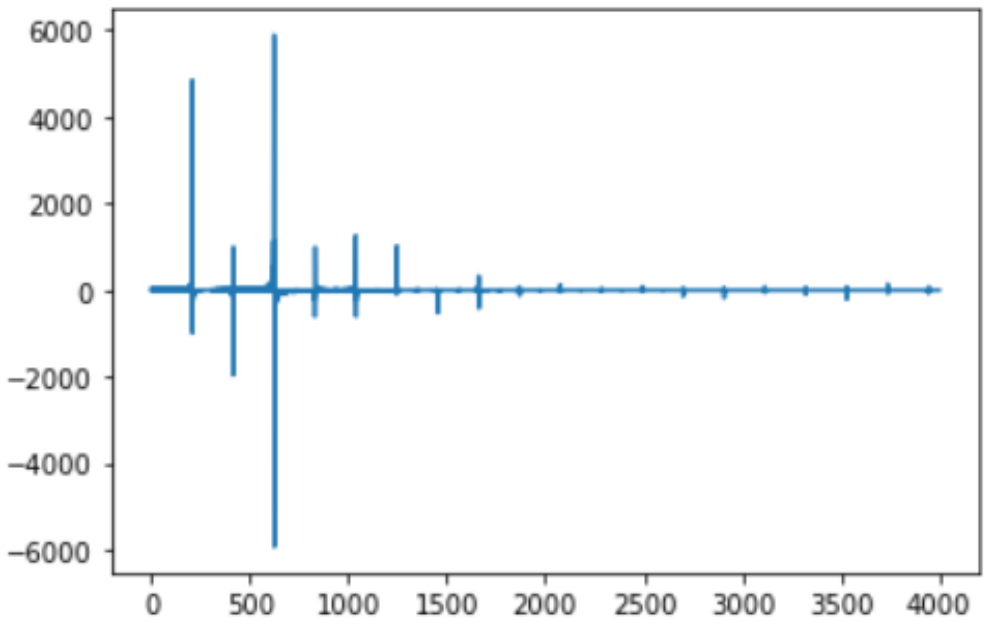

Рис. 2.2: ДКП фильтрованного сегмента

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

2.3 Работа с длинным сегментом

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.

```
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)
```

Листинг 2.4: Функция make_dct_spectrogram

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

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

Листинг 2.5: Сжатие сегментов

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.

Глава 3

Упражнение 6.3

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.

3.1 Пилообразный сигнал

В начале мы возьмем пилообразный сигнал с частотой 500 и построим его спектр, а затем построим его угловую часть (Рис.3.1).

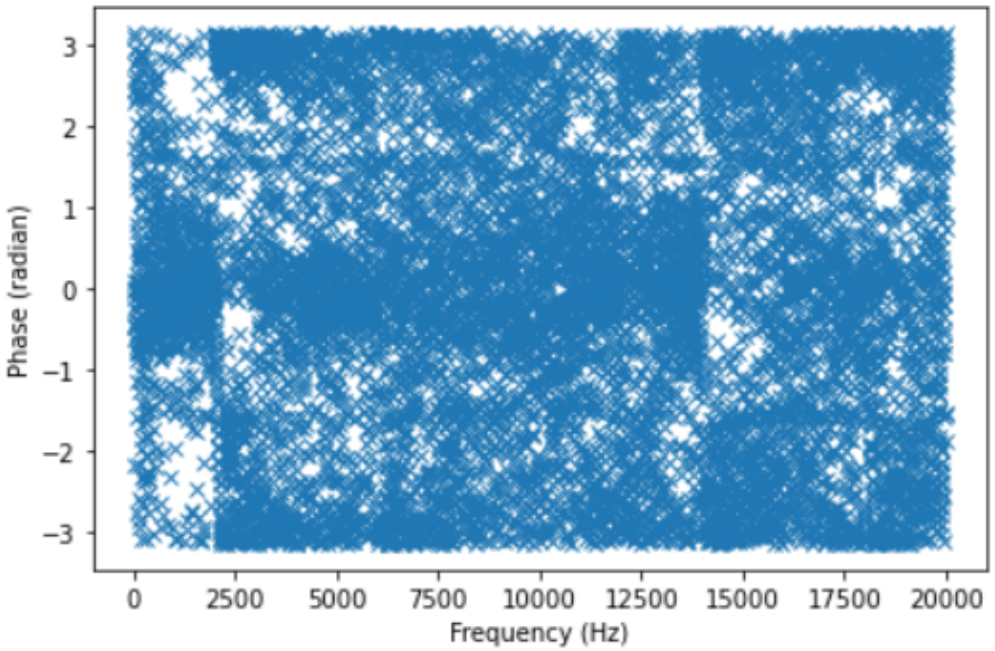

Рис. 3.1: Пилообразный сигнал. Угловая часть спектра

При построении всех углов, мы получаем довольно запутанную картинку (Рис.3.1).

Но если мы выберем только те частоты, где величина превышает порог, мы увидим, что в углах есть структура. Каждая гармоника смещена от предыдущей на доли радиана (Рис.3.2).

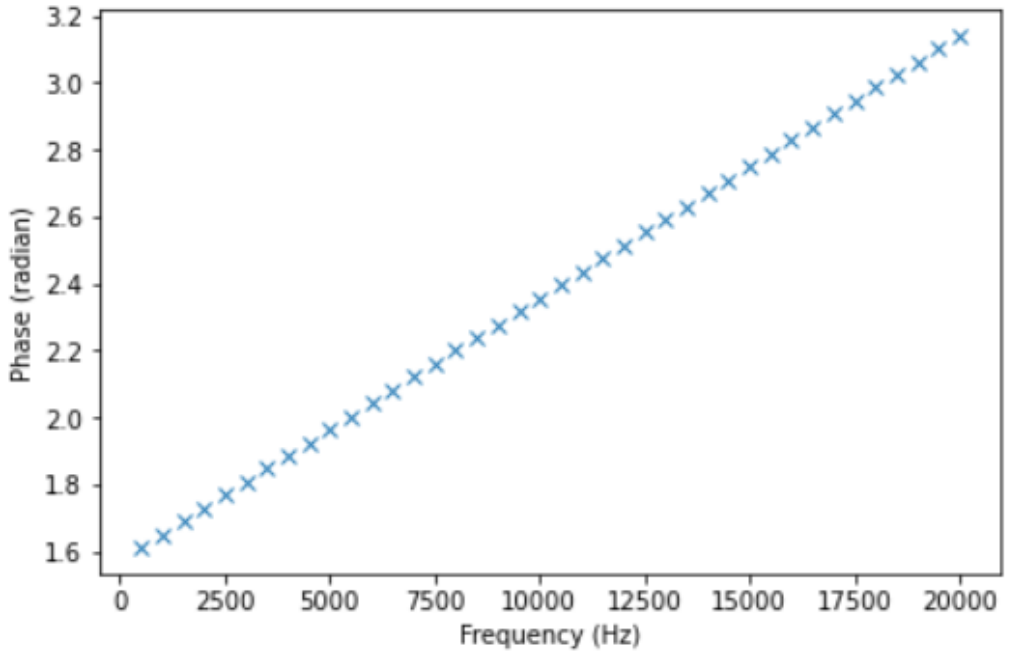

Рис. 3.2: Пилообразный сигнал. Угловая часть спектра с порогом

Теперь посмотрим, что произойдет, если мы установим все углы в ноль (Рис.3.3).

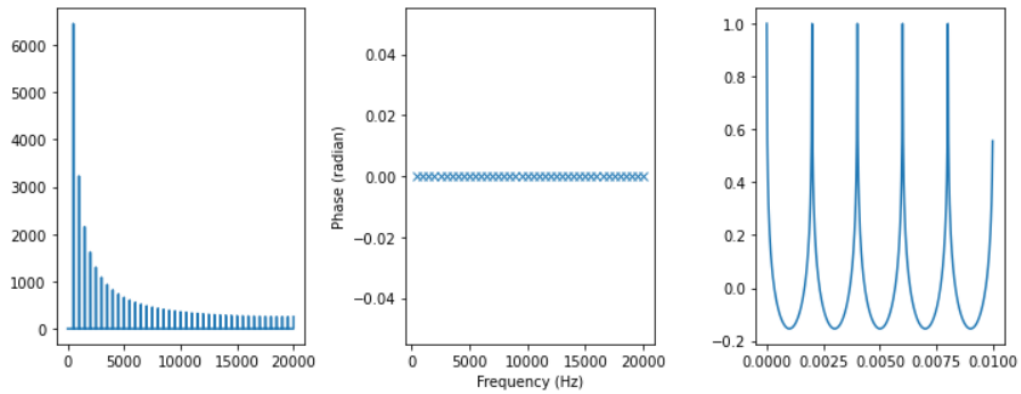

Рис. 3.3: Пилообразный сигнал. Нулевые углы

Амплитуда волны стала ниже, но это из-за способа нормализации волны, а не из-за изменений в фазовой структуре.

Теперь выполним поворот угла (Рис.3.4).

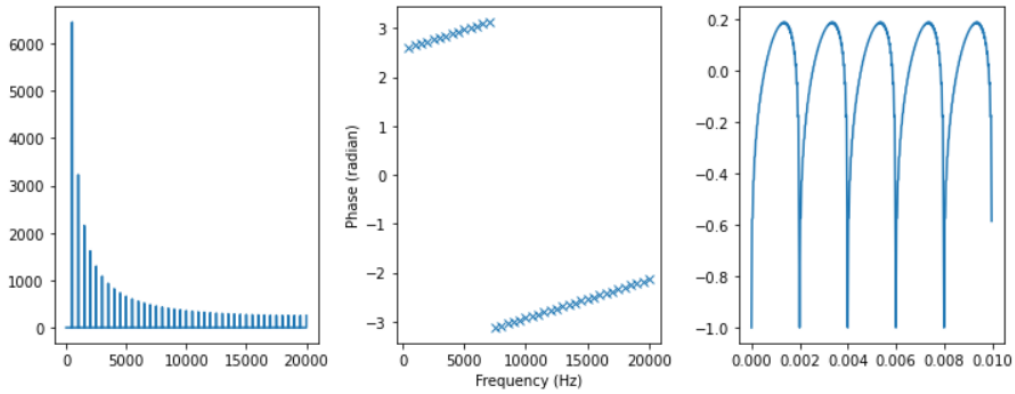

Рис. 3.4: Пилообразный сигнал. Поворот угла

Также посмотрим, что произойдет, если мы установим для углов случайные значения (Рис.3.5).

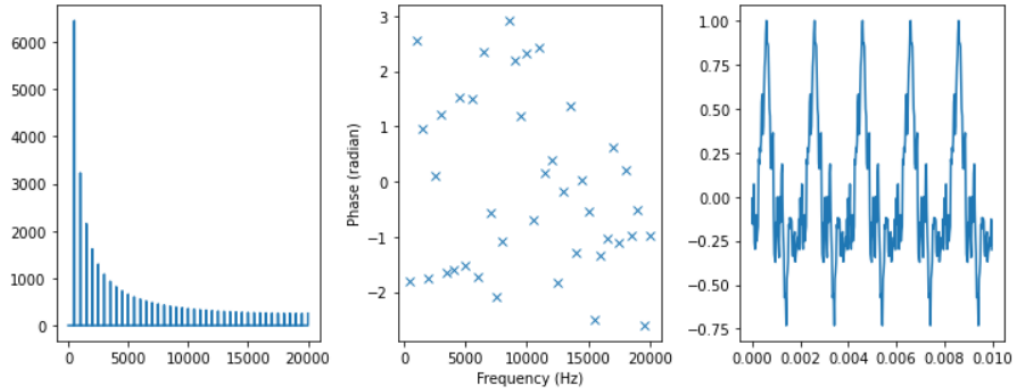

Рис. 3.5: Пилообразный сигнал. Случайные углы

3.2 Гобой

Теперь поработаем с другими звуками. Воспользуемся записью гобоя. Выделим из неё новый сегмент. Снова построим спектр и его угловую часть (Рис.3.6).

```
segment = wave.segment(start=1.0, duration=0.9)
```

Листинг 3.1: Выделение сегмента

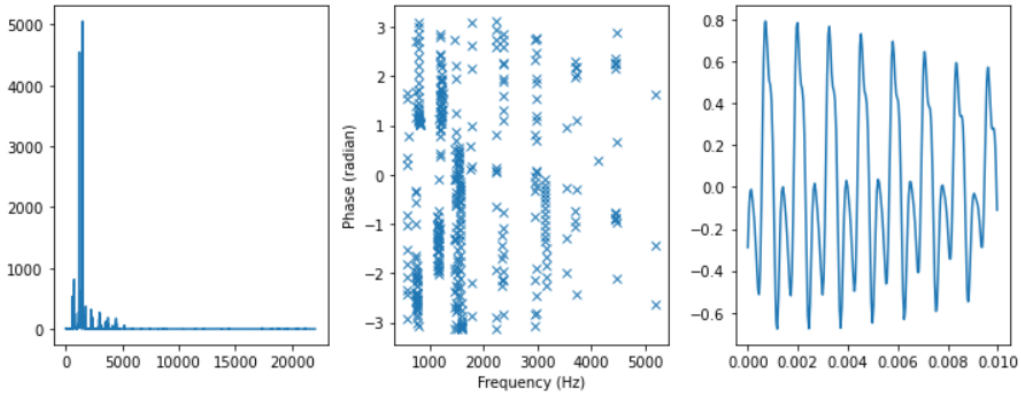

Рис. 3.6: Гобой. Угловая часть спектра

Теперь установим все углы в ноль (Рис.3.7).

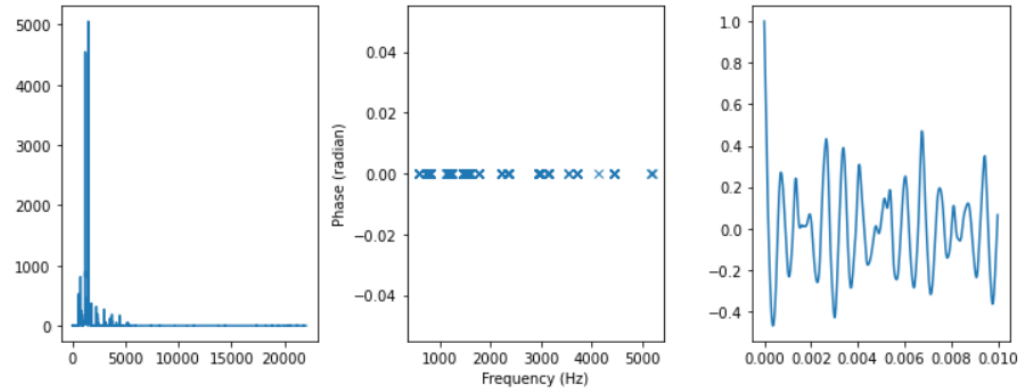

Рис. 3.7: Гобой. Нулевые углы

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

Затем повернём угол на один радиан (Рис.3.8).

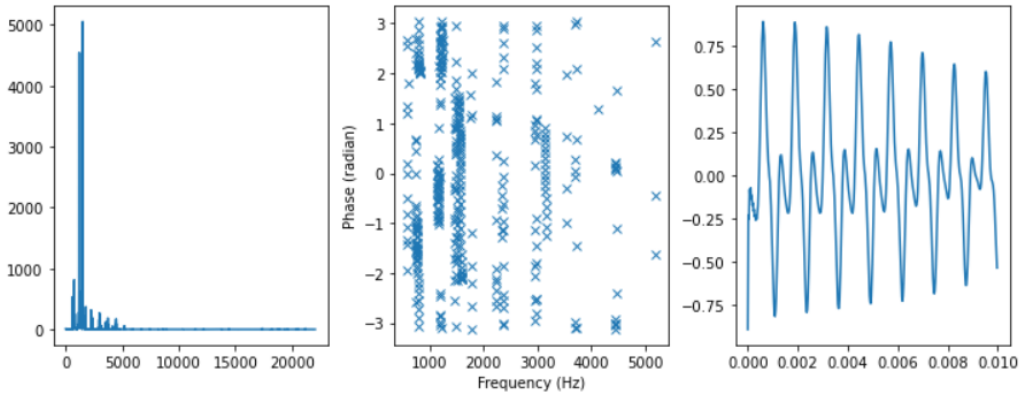

Рис. 3.8: Гобой. Поворот угла

Вращение углов не вызывает «звона».

Далее установим случайные значения (Рис.3.9).

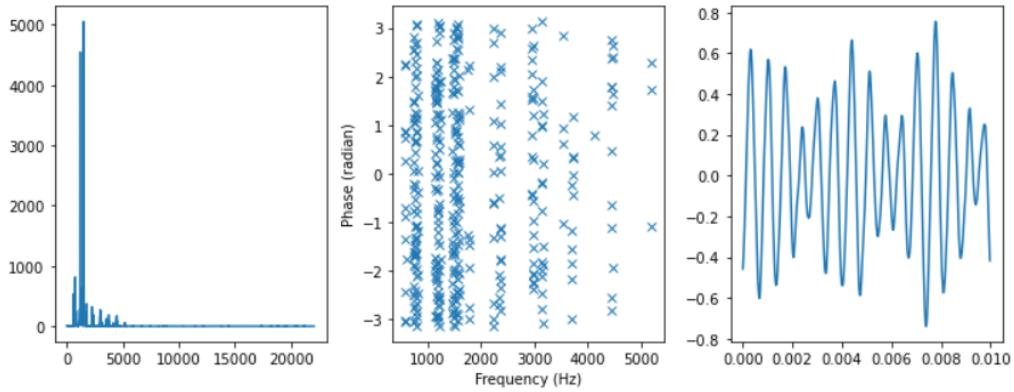

Рис. 3.9: Гобой. Случайные углы

Случайный выбор углов вызывает некоторый звон и добавляет хриплости звуку.

3.3 Саксофон

Проведя аналогичные исследования на саксофоне, мы получаем такие же результаты. Обнуление вызывает звон, вращение мало влияет, а использование случайных значений добавляет хрипы.

3.4 Саксофон с удаленной основной частотой

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

Обнуление (Рис.3.10).

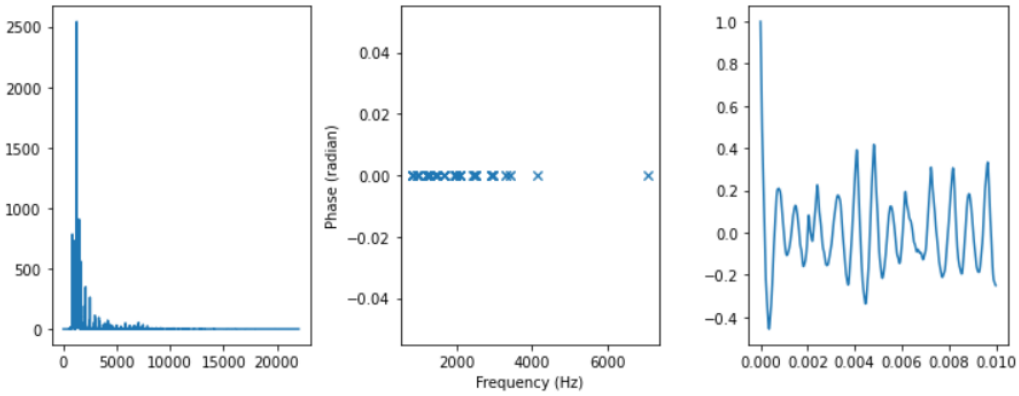

Рис. 3.10: Саксофон. Нулевые углы

Поворот (Рис.3.11).

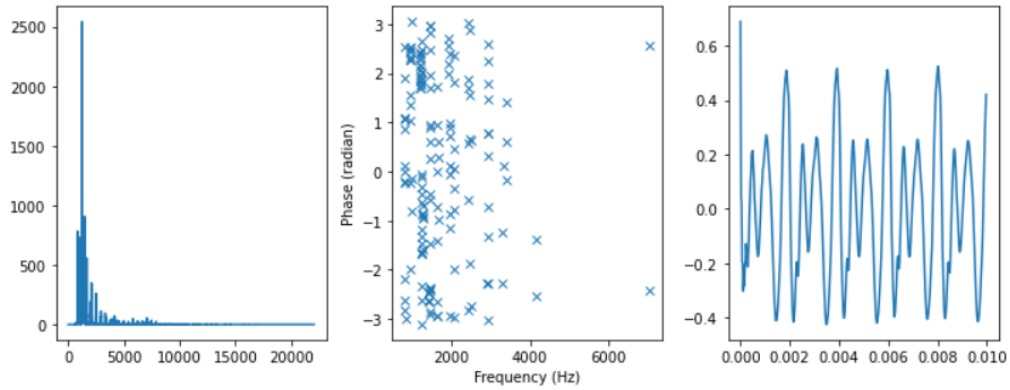

Рис. 3.11: Саксофон. Поворот угла

Случайные значения (Рис.3.12).

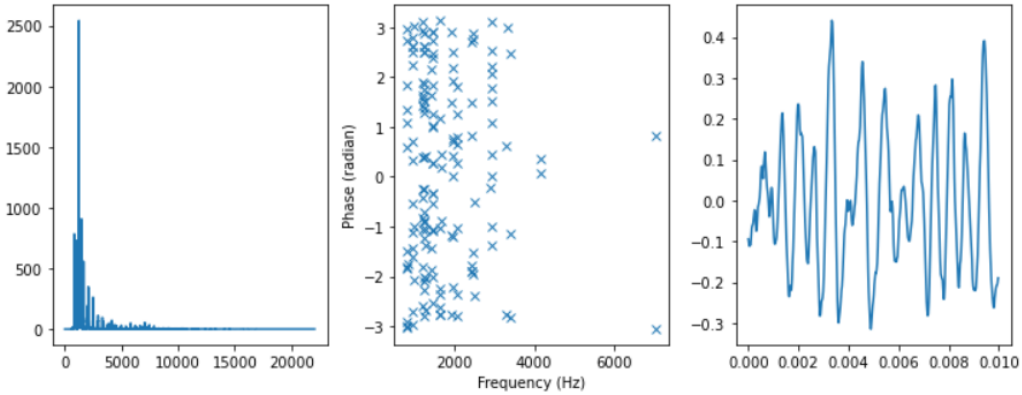

Рис. 3.12: Саксофон. Случайные углы

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.

Глава 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.