ThinkDSP. Лабораторная 6. Дискретное косинусное преобразование.

Шерепа Никита

8 мая 2021 г.

Содержание

Список иллюстраций

Листинги

1 Упражение 6.1

1. **Задание**

   Убедитесь, что `analyze1` требует времени пропорционально n^3, а `analyze2` - пропорционально n^2. Для этого запустите их с несколькими разными массивами и засекайте время работы.

2. **Ход работы**

   Для начала создадим сигнал и вычислим его `Shape`

```
from thinkdsp import UncorrelatedGaussianNoise

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

Output
(16384,)
```

   Листинг 1: Создание сигнала и его `Shape`

   Следующая функция - `plot_bests()` - отображает массив значений выстраивает из них прямую линию

```
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 2: Функция `plot_bests()`

   Теперь поработаем с `analyze1()` с помощью функции `run_speed_test()`

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

```
    return amps
```

Листинг 3: Функция analyze1

```
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 4: Функция run_speed_test()

```
ns = 2 ** np.arange(6, 13)
bests = run_speed_test(ns, analyze1)
plot_bests(ns, bests)
```

Листинг 5: Работа с analyze1

```
Output

64
93.5 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    10000 loops each)
128
321 µs ±0 ns per loop (mean ±std. dev. of 1 run, 1000
    loops each)
256
2 ms ± 0 ns per loop (mean ±std. dev. of 1 run, 100
    loops each)
512
7.45 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100
    loops each)
1024
35.4 ms ±0 ns per loop (mean ±std. dev. of 1 run, 10
    loops each)
2048
187 ms ±0 ns per loop (mean ±std. dev. of 1 run, 10
```

```
    loops each)
4096
880 ms ± 0 ns per loop (mean ±std. dev. of 1 run, 1
    loop each)

2.2185829742670475
```

Листинг 6: Результат

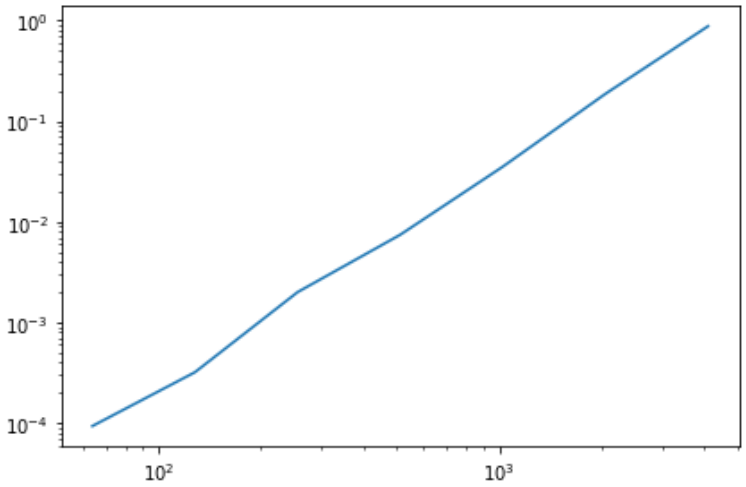

Рис. 1: Результат `analyze1()`

Наклон близок к 2, а не к 3, как ожидалось. Производительность np.linalg.solve почти квадратична в этом диапазоне размеров массива.

Теперь поработаем с `analyze2()`

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

Листинг 7: Функция `analyze2()`

```
bests2 = run_speed_test(ns, analyze2)
plot_bests(ns, bests2)
```

Листинг 8: Работа с `analyze2()`

```
Output

64
54.4 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    10000 loops each)
128
220 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    10000 loops each)
256
1.18 ms ±0 ns per loop (mean ±std. dev. of 1 run,
    1000 loops each)
512
4.38 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100
    loops each)
1024
16.7 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100
    loops each)
2048
74.5 ms ±0 ns per loop (mean ±std. dev. of 1 run, 10
    loops each)
4096
307 ms ±0 ns per loop (mean ±std. dev. of 1 run, 1
    loop each)

2.0723003647065377
```

Листинг 9: Результат

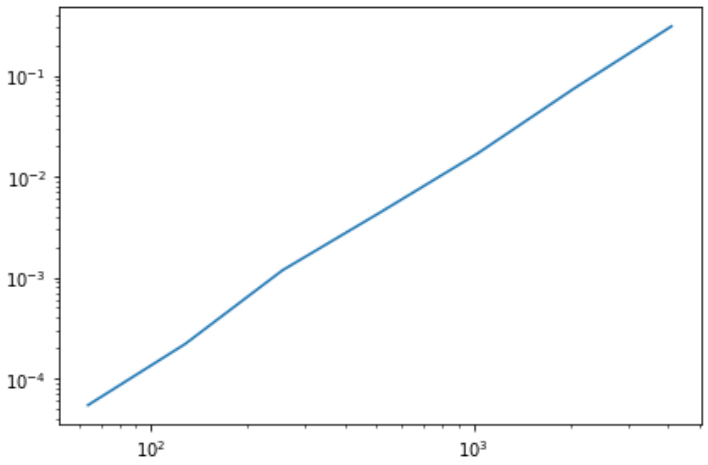

Рис. 2: Результат `analyze2()`

С analyze2() ситуация уже получше: наклон ближе к 2, как и ожидалось.

Теперь поэкспериментируем с `scipy.fftpack.dct`

```
import scipy.fftpack

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

Листинг 10: Функция `scipy_dct()`

```
bests3 = run_speed_test(ns, scipy_dct)
plot_bests(ns, bests3)
```

Листинг 11: Работа с `scipy_dct()`

```
Output

64
7.48 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    100000 loops each)
128
7.73 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    100000 loops each)
256
8.44 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    100000 loops each)
512
9.88 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    100000 loops each)
1024
13 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    100000 loops each)
2048
21.2 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    10000 loops each)
4096
35.9 µs ±0 ns per loop (mean ±std. dev. of 1 run,
    10000 loops each)


0.3686126799836147
```

Листинг 12: Результат

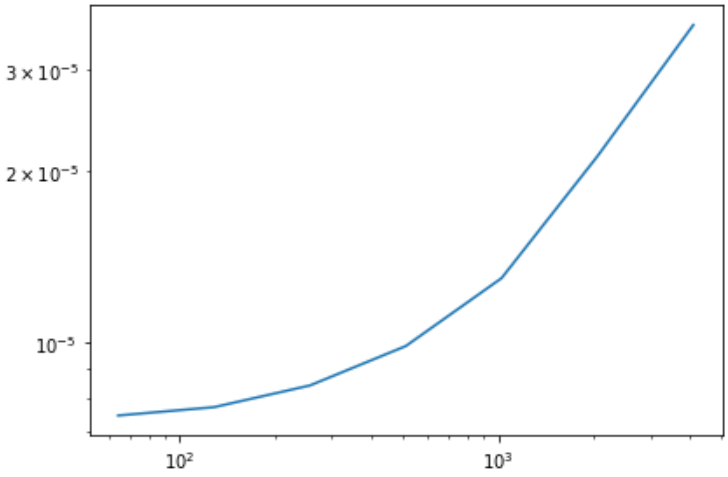

Рис. 3: Результат `scipy_dct()`

Результат получили намного быстрее по времени, так как её асимптотическая сложность функции = `nlogn`

Подведем итоги и отобразим все кривые на одном графике

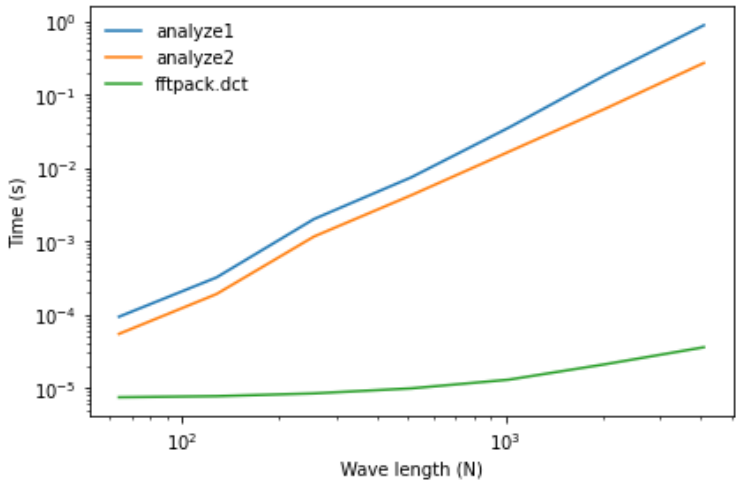

Рис. 4: Итог

2 Упражение 6.2

1. **Задание**

   Реализуйте версию алгоритма ДКП - сжатие звука и изображений - и примените его для записи музыки и речи. Сколько компонент можно удалить до того, как разница станет заметной?

2. **Ход работы**

   Возьмем звук саксофона из 5ой лабораторной работы и выделим из него сегмент

```
from thinkdsp import read_wave

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

segment = wave.segment(start=1.2, duration=0.5)
segment.normalize()
segment.make_audio()
```

Листинг 13: Сегмент звука

   Теперь применим к сегменту ДКП

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

Листинг 14: Применяем ДКП

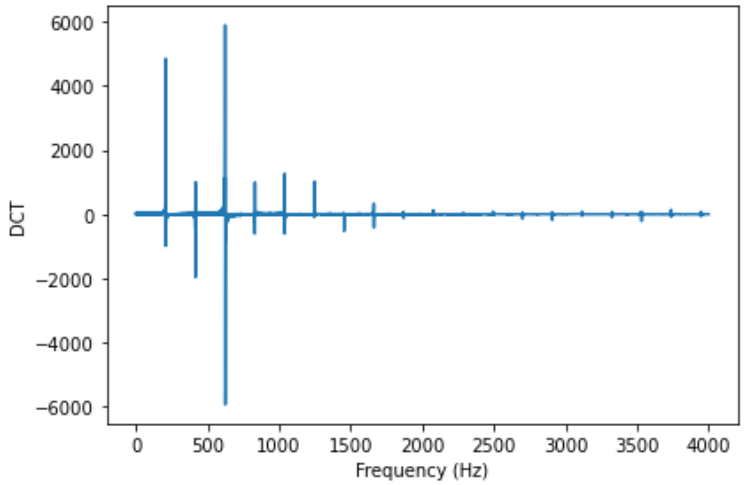

Рис. 5: Результат

По амплитуде выделяются мало гармоник, в основном большинство близки к нулю.

Рассмотрим функцию `compress()`. Она принимает ДКП и значение порога и выставляет значение = 0 элементам, значение которых ниже порога (обрубает)

```
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')
```

Листинг 15: Функция `compress()`

Теперь попробуем сжать звук - применим функцию `compress()`

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

Листинг 16: Сжимаем звук

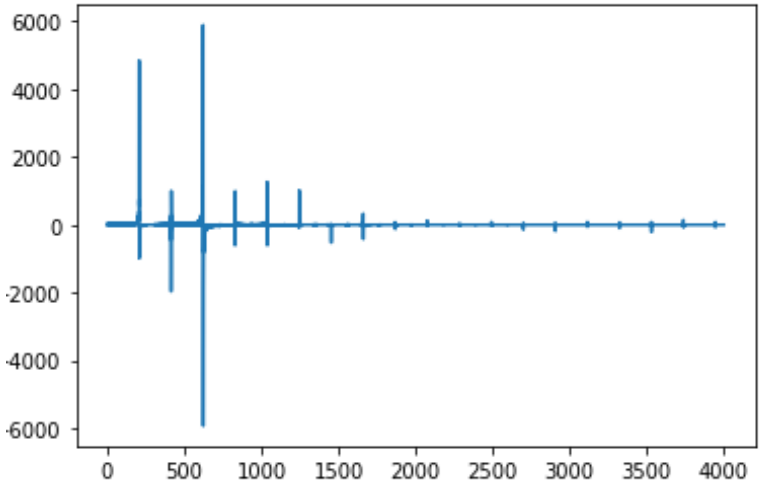

Рис. 6: Результат сжатия

Звук звучит точно также, как и до сжатия

Чтобы применить ДКП для более длинного сегмента, рассмотрим функцию `make_dct_spectrogram()`. Она похожа на `wave.make_spectorgram`, но в работе она использует 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

    # map from time to Spectrum
    spec_map = {}

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

        # the nominal time for this segment is the
            midpoint
        t = (segment.start + segment.end) / 2
        spec_map[t] = segment.make_dct()

        i += step
        j += step
```

```
21
22              return Spectrogram(spec_map, seg_length)
```

Листинг 17: Функция `make_dct_spectrogram()`

Теперь применим её ко всему звуку

```
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 18: Применяем `make_dct_spectrogram()`

В большиснтве сегментов сжатие = 75% - 80%

Теперь прослушаем результат

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

Листинг 19: Воспроизводим сжатый звук

В целом, звучание сохранилось, но появился сильно заметный разражающий треск на заднем фоне, который был еле слышен у несжатого звука.

3 Упражение 6.3

1. **Задание**

   Исследовать влияние фазы на восприятие звука. Поэкспериментировать с примерами.

2. **Ход работы**

   В качестве звука для эксперимента возьмем звук саксофона из прошлого пункта и выделим из него сегмент.

```
wave =
    read_wave('res/100475__iluppai__saxophone-weep.wav')
wave.make_audio()
segment = wave.segment(start=1.9, duration=0.6)

spectrum = segment.make_spectrum()
plot_three(spectrum, thresh=50)
```

Листинг 20: Сегмент звука саксофона

Теперь отобразим 1. Амплитуду сигнала 2. Угловую часть спектра 3. Форму волны

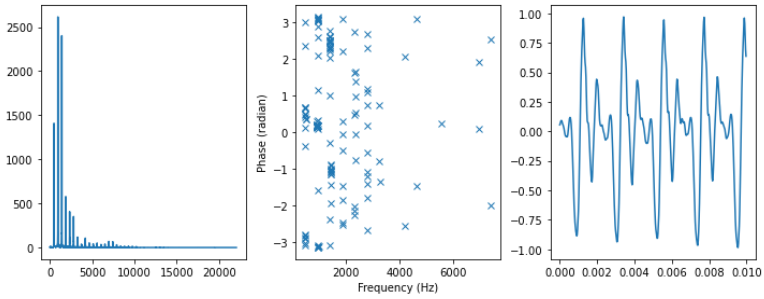

Рис. 7: Результат сжатия

Теперь установим все углы на ноль

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

spectrum2 = zero_angle(spectrum)
```

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

Листинг 21: Все углы на ноль

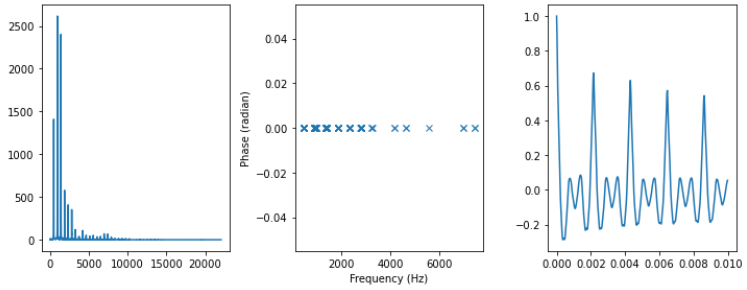

Рис. 8: Все углы на ноль

Амплитуда не изменилась, но изменилась форма волны. Теперь сегмент звучит волнообразно: сначала утихает, затем резко возрастает. Также звук стал тише.

Теперь "повернем"углы на 1 радиан.

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

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

Листинг 22: Повернули углы

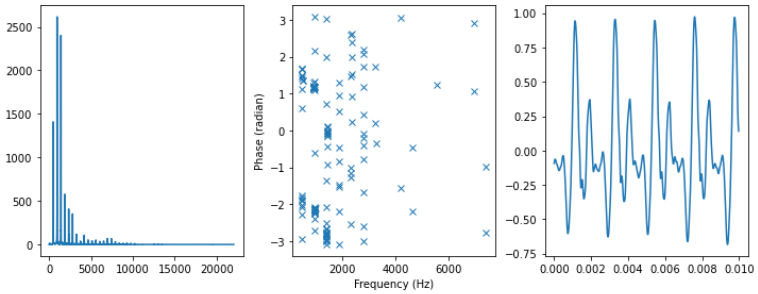

Рис. 9: Повернули углы

Звук нормализировался и звучит монотонно, по сравнению с прошлым результатом.

Теперь пусть каддый угол будет равен случайному значению

```
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

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

Листинг 23: Рандомные углы

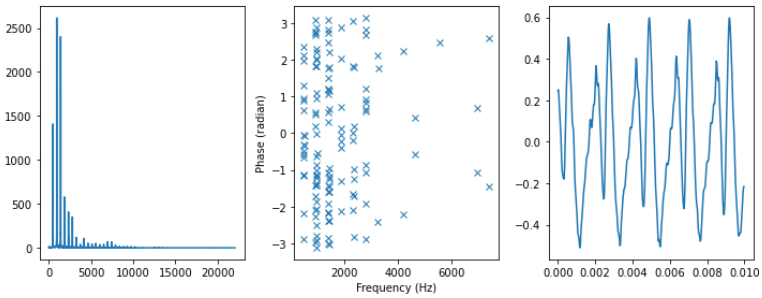

Рис. 10: Рандомные углы

Звук как-будто раздвоился, по сравнению с прошлым результатом.

У звука саксофона есть особенность - основной компонент не является доминирующим. Фазовую структуру звуков в "пропавшей"частотой человеческое ухо может воспринимать. Автор предполагает, что ухо использует что-то вроде автокорреляции.

4 Вывод

В результате выполнения лабораторной работы получены навыки работы с дискретным косинусным преобразованием (ДКП). Также изучено влияние фазы на восприятие звука.