

Лабораторная работа №6
Дискретное косинусное преобразование

Крынский Павел

26 мая 2021 г.

Оглавление

1	Упражнение 6.1	4
2	Упражнение 6.2	8
3	Упражнение 6.3	12
4	Выводы	13

Список иллюстраций

1.1	Визуализация данных	7
2.1	Визуализация сжатого звука	8
2.2	Визуализация сжатого звука	9

Листинги

1.1	Тестирующая функция и функция для оценки наклона графиков	4
1.2	Массив степеней двойки	4
1.3	Результат работы	5
2.1	Загрузка звука и сегмент	8
2.2	Функция <code>compress</code>	9
2.3	Сжатие звука	9
2.4	Воспроизведение сжатого звука	9
2.5	Функция <code>make_dct_spectrogram</code>	10
2.6	Сжатие звука	10
2.7	Воспроизведение сжатого звука	10
2.8	Воспроизведение оригинального звука	10

Глава 1

Упражнение 6.1

Я добавил тестирующую функцию и функцию для оценки наклона графиков:

```
1 def speed_test(ns, func):
2     res = []
3     for N in ns:
4         print(N)
5         ts = (0.5 + np.arange(N)) / N
6         freqs = (0.5 + np.arange(N)) / 2
7         ys = noise.ys[:N]
8         result = %timeit -r1 -o func(ys, freqs, ts)
9         res.append(result)
10
11     bests = [result.best for result in res]
12     return bests
13
14 def fit_slope(ns, bests):
15     x = np.log(ns)
16     y = np.log(bests)
17     t = linregress(x,y)
18     slope = t[0]
19
20     return slope
```

Листинг 1.1: Тестирующая функция и функция для оценки наклона графиков

Далее я создал сигнал, на котором будет происходить тестирование и массив со степенями двойки с 4-ой по 11-ю.

```
1 signal = thinkdsp.UncorrelatedGaussianNoise()
```

```

2 noise = signal.make_wave(duration=1.0, framerate=16384)
3 ns = 2** np.arange(4,12)

```

Листинг 1.2: Массив степеней двойки

Ниже я привел результаты запуска тестирующей функции для трех функций(analyze1, analyze2, fftpack.dct):

```

1 16
2 24.9 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
3 32
4 44.8 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
5 64
6 106 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
7 128
8 352 µs ±0 ns per loop (mean ±std. dev. of 1 run, 1000 loops each)
9 256
10 1.68 ms ±0 ns per loop (mean ±std. dev. of 1 run, 1000 loops each)
11 512
12 9.8 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100 loops each)
13 1024
14 40.5 ms ±0 ns per loop (mean ±std. dev. of 1 run, 10 loops each)
15 2048
16 191 ms ±0 ns per loop (mean ±std. dev. of 1 run, 10 loops each)
17
18 16
19 14.9 µs ±0 ns per loop (mean ±std. dev. of 1 run, 100000 loops
    each)
20 32
21 24.6 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
22 64
23 57 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
24 128
25 177 µs ±0 ns per loop (mean ±std. dev. of 1 run, 10000 loops each)
26 256
27 660 µs ±0 ns per loop (mean ±std. dev. of 1 run, 1000 loops each)
28 512
29 4.16 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100 loops each)
30 1024
31 17 ms ±0 ns per loop (mean ±std. dev. of 1 run, 100 loops each)
32 2048
33 63.8 ms ±0 ns per loop (mean ±std. dev. of 1 run, 10 loops each)
34
35 16

```

```

36 8.18  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
37 32
38 7.91  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
39 64
40 8.11  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
41 128
42 9.56  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
43 256
44 8.77  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
45 512
46 10.3  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
47 1024
48 12.5  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)
49 2048
50 17.7  $\mu$ s  $\pm$ 0 ns per loop (mean  $\pm$ std. dev. of 1 run, 100000 loops
    each)

```

Листинг 1.3: Результат работы

Также приведу показатели уклона трех функций: 1.9205374674429947
1.8105335861011087 0.14312061938412077

И для наглядности я вывел три графика для сравнения:

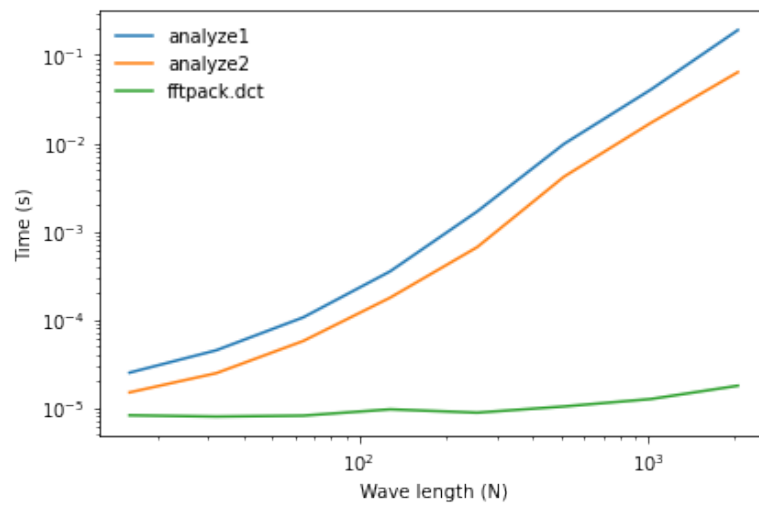


Рис. 1.1: Визуализация данных

У функции `fftpack.dct` лучшие временные траты.

Глава 2

Упражнение 6.2

Берем нашу запись сегмент, а затем ДКП этого сегмента:

```
1 wave = thinkdsp.read_wave('185347__lemoncreme__symphony-sounds.wav')
2 wave.make_audio()
3 sg = wave.segment(start=1.2, duration=0.5)
4 sg.normalize()
5 sg.make_audio()
6 sg_dct = sg.make_dct()
7 sg_dct.plot(high=1000)
8 decorate(xlabel='Frequency (Hz)', ylabel='DCT')
```

Листинг 2.1: Загрузка звука и сегмент

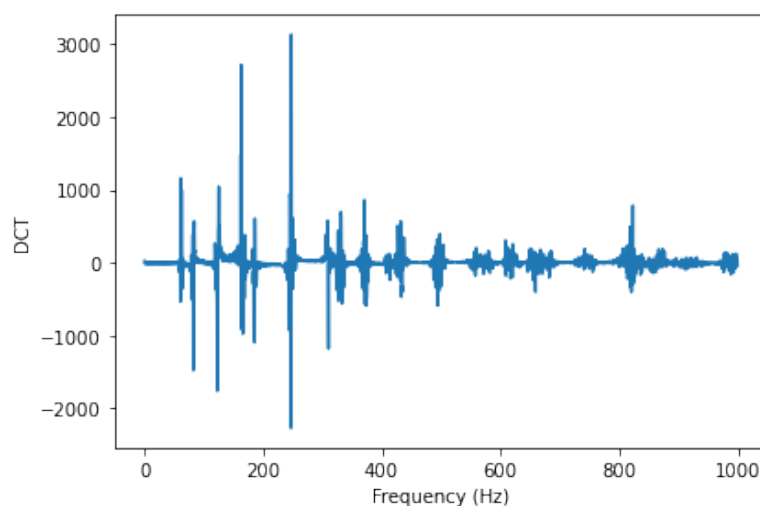


Рис. 2.1: Визуализация сжатого звука

Многие записи близки к нулю. Следующая функция принимает ДКП и устанавливает для элементов ниже порога значение 0.

```
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 2.2: Функция `compress`

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

```
1 sg_dct = sg.make_dct()
2 compress(sg_dct, thresh=10)
3 sg_dct.plot(high=1000)
```

Листинг 2.3: Сжатие звука

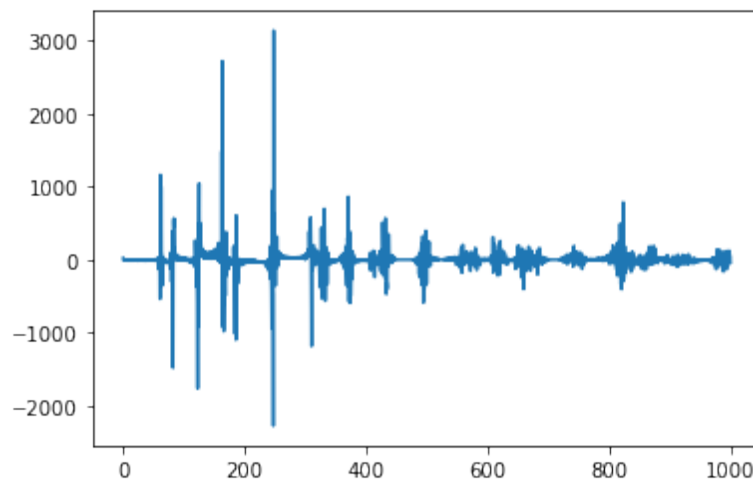


Рис. 2.2: Визуализация сжатого звука

```
1 sg2 = sg_dct.make_wave()
2 sg2.make_audio()
```

Листинг 2.4: Воспроизведение сжатого звука

Результат звучит точно так же.

Чтобы сжать более длинный сегмент, мы можем сделать спектрограмму 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
5
6     spec_map = {}
7
8     while j < len(wave.ys):
9         segment = wave.slice(i, j)
10        segment.window(window)
11
12        t = (segment.start + segment.end) / 2
13        spec_map[t] = segment.make_dct()
14
15        i += step
16        j += step
17
18    return thinkdsp.Spectrogram(spec_map, seg_length)
```

Листинг 2.5: 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 2.6: 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 2.7: Воспроизведение сжатого звука

Так же прослушаем оригинал для сравнения.

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

Листинг 2.8: 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

Теперь нам нужно запустить готовый `phase.ipynb` и посмотреть, что там происходит.

Если мы посмотрим на фазовые сдвиги каждой компоненты некоторого звука, то мы будем видеть только много случайных значений, однако если отфильтровать частоты с маленьким амплитудами, то начнет вырисовываться структура. Величина фазы от частоты компоненты может зависеть как линейно, так и случайно, однако в подавляющем большинстве случаев ухо не будет способно это воспринять.

Глава 4

Выводы

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