

Санкт-Петербургский государственный политехнический  
университет Петра Великого.

**Высшая школа интеллектуальных систем и  
суперкомпьютерных технологий**

ЛАБОРАТОРНАЯ РАБОТА

Дискретное косинусное преобразование

Работу выполнила студентка:

\_\_\_\_\_ А. И. Луцкевич  
« \_\_\_\_ » \_\_\_\_\_ 2021 г.

Преподаватель лабораторных  
работ:

\_\_\_\_\_ Н. В. Богач  
« \_\_\_\_ » \_\_\_\_\_ 2021 г.

Санкт-Петербург, 2021 г.

## Суть работы 6.1:

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`.

Сначала необходимо создать сигнал, который будет основан на некоррелируемом гауссовском шуме. А также создадим массив с некоторыми тестовыми данными (степени двойки с 5 по 12).

```
In [3]: signal = UncorrelatedGaussianNoise()
noise = signal.make_wave(duration=1.0, framerate=16384)
noise.ys.shape

Out[3]: (16384,)

In [4]: ns = 2 ** np.arange(5, 12)
ns

Out[4]: array([ 32,  64, 128, 256, 512, 1024, 2048], dtype=int32)
```

Создадим сначала две функции: первая будет строить результаты и рисовать линию из массива результатов временного эксперимента, а вторая будет `analyze1`, которую и необходимо проверить.

```
In [4]: def plot_bests(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

In [5]: def analyze1(ys, fs, ts):
        args = np.outer(ts, fs)
        M = np.cos(PI2 * args)
        amps = np.linalg.solve(M, ys)
        return amps
```

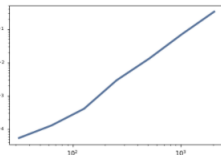
Далее протестируем эту функцию и зафиксируем получившиеся результаты.

```
In [7]: 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)
```

```
54.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
130 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)
421 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)
2.93 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
13.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
69.6 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
333 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
Out[7]: 2.172947418143087
```



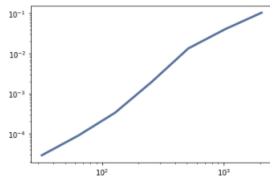
Теперь создадим функцию analyze2.

```
In [8]: def analyze2(ys, fs, ts):  
        args = np.outer(ts, fs)  
        M = np.cos(PI2 * args)  
        amps = np.dot(M, ys) / 2  
        return amps
```

И таким же образом протестируем ее.

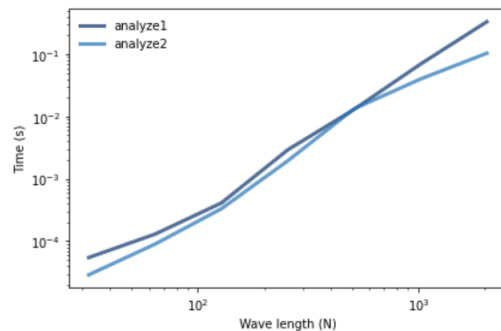
```
In [9]: results = []  
        for N in ns:  
            ts = (0.5 + np.arange(N)) / N  
            freqs = (0.5 + np.arange(N)) / 2  
            ys = noise.ys[N]  
            result = %timeit -r1 -o analyze2(ys, freqs, ts)  
            results.append(result)  
  
        bests2 = [result.best for result in results]  
        plot_bests(bests2)  
  
28.8 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)  
89.8 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10000 loops each)  
332 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1000 loops each)  
1.95 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)  
13.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)  
40.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)  
104 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
```

Out[9]: 2.0846045672137273



Для того, чтобы сравнить результаты необходимо два этих графика отобразить на одном графике.

```
In [12]: thinkplot.plot(ns, bests, label='analyze1')  
         thinkplot.plot(ns, bests2, label='analyze2')  
         decorate(xlabel='Wave length (N)', ylabel='Time (s)', **dict(xscale='log', yscale='log'))
```



Посмотрев на два графика сразу, можно сделать вывод, что analyze2 работает быстрее, чем analyze1.

## Суть работы 6.2:

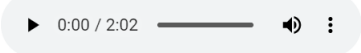
Необходимо реализовать версию ДКП алгоритма для сжатия звука

Возьмем для этого звук, который был использован в прошлой лабораторной работе и будем работать только с кусочком длиной 0.5 секунд начиная

с 10 секунды.

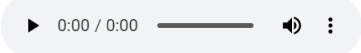
```
In [17]: wave = read_wave('6_2.wav')
         wave.make_audio()
```

Out[17]:



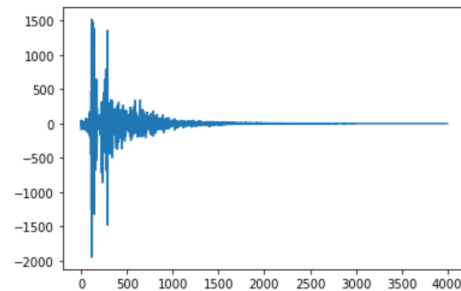
```
In [18]: segment = wave.segment(start=10, duration=0.5)
         segment.normalize()
         segment.make_audio()
```

Out[18]:



Выведем график амплитуды сегмента.

```
In [13]: seg_dct = segment.make_dct()
         seg_dct.plot(high=4000)
```



В сегменте присутствует много точек с нулевой амплитудой (ближе к концу).

Напишем функцию `compress`, которая будет занулять элементы. Причем она будет принимать `thresh` и занулять только те элементы, которые меньше порога `thresh`.

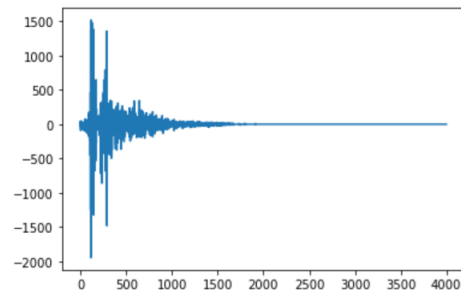
```
In [19]: def compress(dct, thresh=1):
         count = 0
         for i, amp in enumerate(dct.amps):
             if np.abs(amp) < thresh:
                 dct.hs[i] = 0
                 count += 1

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

И сразу применим написанную функцию к нашему сегменту.

```
In [15]: seg_dct = segment.make_dct()
compress(seg_dct, thresh=10)
seg_dct.plot(high=4000)

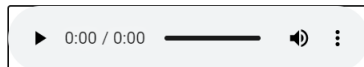
19744 22050 89.54195011337869
```



Было занулено 19744 элемента из 22050.

```
In [22]: seg2 = seg_dct.make_wave()
seg2.make_audio()
```

Out[22]:



При прослушивании данного сегмента появились некоторые шумы, хотя графики визуально не отличаются.

Дальше необходимо написать метод, который будет делать спектрограмму ДКП, чтобы сжать более длинный фрагмент.

```
In [23]: 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)
```

И осталось создать эту спектрограмму и применить compress к каждому сегменту (на рисунке ниже представлены только первые элементы).

```
In [24]: spectro = make_dct_spectrogram(wave, seg_length=1024)
         for t, dct in sorted(spectro.spec_map.items()):
             compress(dct, thresh=0.2)
```

```
1016 1024 99.21875
1010 1024 98.6328125
1007 1024 98.33984375
1002 1024 97.8515625
1001 1024 97.75390625
991 1024 96.77734375
983 1024 95.99609375
984 1024 96.09375
982 1024 95.8984375
988 1024 96.484375
970 1024 94.7265625
972 1024 94.921875
969 1024 94.62890625
968 1024 94.53125
971 1024 94.82421875
971 1024 94.82421875
968 1024 94.53125
```

Для сравнения с исходным сигналом переведем спектрограмму в сигнал.

```
In [25]: wave2 = spectro.make_wave()
         wave2.make_audio()
```

Out[25]:



В итоге всех проделанных действий появились шумы, которыми можно управлять с помощью значения переменной `thresh`.

## Суть работы 6.3:

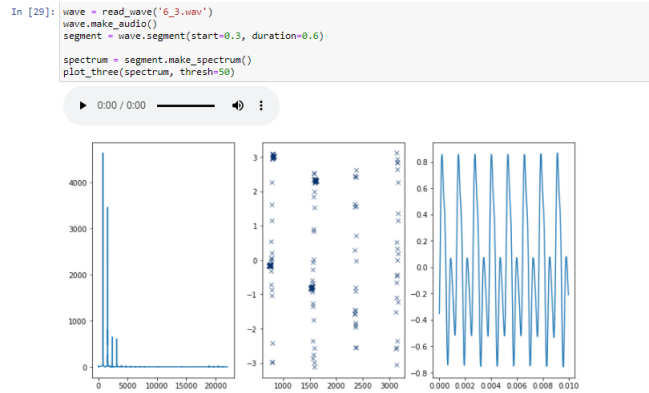
Необходимо запустить блокнот `phase.ipynb`, пройтись по всем примерам, после чего выбрать любой другой сегмент и сделать с ним те же самые манипуляции.

Функция `plot_angle` отображает амплитуды, форму волны и `angle` для спектра, а функция `plot_three`, выводит на экран 3 графика и аудиодорожку из поданного сигнала.

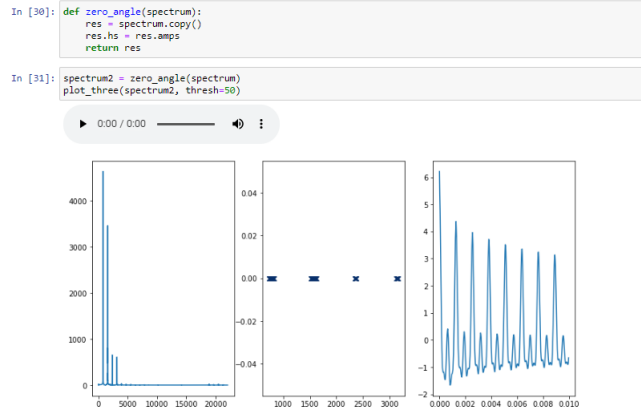
```
In [27]: def plot_angle(spectrum, thresh=1):
         angles = spectrum.angles
         angles[spectrum.amps < thresh] = np.nan
         thinkplot.plot(spectrum.fs, angles, 'x')
```

```
In [28]: def plot_three(spectrum, thresh=1):
         thinkplot.preplot(cols=3)
         spectrum.plot()
         thinkplot.subplot(2)
         plot_angle(spectrum, thresh=thresh)
         thinkplot.subplot(3)
         wave = spectrum.make_wave()
         wave.segment(duration=0.01).plot()
         wave.apodize()
         display(wave.make_audio())
```

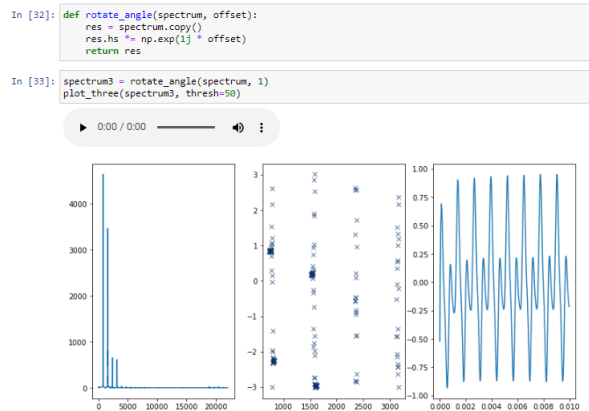
Попробуем на том же звуке гобоя выделить другой сегмент длительностью 0.6 секунд и вызовем `plot_three` с этим сегментом.



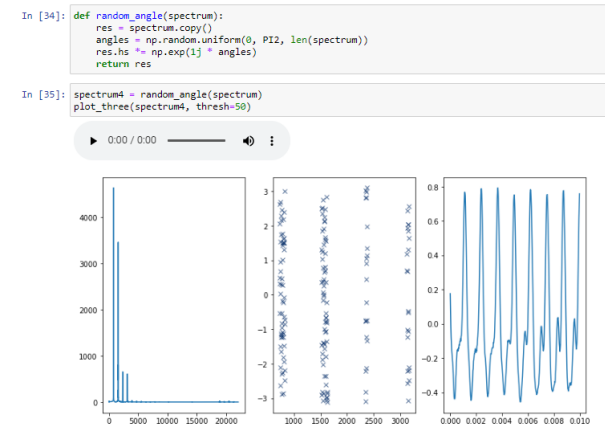
Используем функцию `zero_angle`, которая выдаст результаты, в котором `angle` равен 0, и вызовем эту функцию с `spectrum`.



Используем функцию `rotate_angle`, которая выдаст результаты, в котором `angle` изменен на 1 радиан, и вызовем эту функцию с `spectrum`.



Используем функцию `random_angle`, которая выдаст результаты, в котором случайный `angle`, и вызовем эту функцию с `spectrum`.



Изменения angle практически не влияют на конечный сигнал, а рандомизация добавила "глухой" эффект.

## Заключение:

По итогу выполнения данной лабораторной работы я изучила понятие ДКС - научилась синтезировать и анализировать ее. Также проводились вычисления, какая из функций (analyze1 и analyze2) работает быстрее и на сколько. Была также создана функция, которая сжимает дорожку.