

РАБОТА №6. ЧАСТОТНО-ВРЕМЕННЫЕ СПОСОБЫ АНАЛИЗА СИГНАЛОВ С ПРИМЕНЕНИЕМ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

Цель работы: изучение возможностей частотно-временного анализа сигналов на базе ДПФ.

Планируемая продолжительность: от 2 до 4 академических часов.

Тип работы:  с использованием компьютерных средств.

Коротко-временное преобразование Фурье в Smath Studio

Проблема ДПФ и НПФ в том, что результаты этих преобразований не зависят от времени, то есть не могут показать изменение частотного/фазового спектра на протяжении длительности сигнала. Для решения этой проблемы на базе ДПФ реализуют коротко-временное преобразование Фурье (КВПФ), являющееся **частотно-временным** способом анализа сигналов. Результатом этого преобразования является двумерный массив, каждый столбец которого представляет собой частотный спектр на основе ДПФ для какого-то короткого участка исследуемого сигнала, причем каждый последующий столбец также является частотным спектром, но для последующих участков сигнала той же длительности. КВПФ используется, к примеру, в кодировщике звуковых файлов в формат MP3 (файл MP3 не содержит значения амплитуд звука, в нем хранятся спектры коротких фрагментов исходного звукового файла).

Импортируем и сконфигурируем следующие зависимости.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 %matplotlib widget
```

Рис. 6.1. Импортирование и конфигурирование зависимостей

Из предыдущей работы возьмём функцию построения простых графиков.

```
1 def plot(*args):
2     ax = plt.figure()
3     for idx in range(0, len(args), 2):
4         x, y = args[idx], args[idx + 1]
5         plt.plot(x, y)
6     plt.grid(True)
7     plt.ylabel('y')
8     plt.xlabel('x')
9     plt.show()
```

Рис. 6.2. Функция построения графиков

```
1 def dft(array: np.ndarray) -> np.ndarray:
2     """ Discrete Fourier Transform """
3     def discrete_fourier_function(
4         array: np.ndarray,
5         k: int
6     ) -> np.imag:
7
8         def F(item, n):
9             return item * np.e ** (
10                 (-1) * ((2 * np.pi * k * n * 1j) /
11                     array.size - 1)
12                 )
13
14         result = np.zeros(array.size, dtype=np.complex_)
15         for idx in range(array.size):
16             result[idx] = F(array[idx], idx)
17         return np.sum(result)
18
19     result = np.zeros(array.shape, dtype=np.complex_)
20     for idx in range(array.size):
21         result[idx] = discrete_fourier_function(array, idx)
22     return result
```

Рис. 6.3. Реализация ДПФ

Спектр сигнала – это модуль его преобразования Фурье. Зададим функцию получения спектра, правой половины (половина массива модуля ДПФ, каждый элемент умножен надвое).

```

1 def spectrum(array: np.ndarray) -> np.ndarray:
2     tmp_freqs = np.abs(array)
3     tmp_freqs = tmp_freqs[tmp_freqs.size // 2:]
4     return 2 * tmp_freqs / array.size

```

Рис. 6.4. Формирование спектра

Так как КВПФ вычисляется для отдельных фрагментов сигнала, зададим функцию фрагментирования на части размера size массива сигнала array.

```

1 def fragmentize(
2     array: np.ndarray,
3     size: int) -> np.ndarray:
4     fragments = np.array_split(array, range(size, array.size, size))
5     fragments = np.array(fragments)
6     return fragments

```

Рис. 6.5. Функция создания фрагментов

На основе этих функций можно вычислить КВПФ, как двухмерный массив, содержащий «слеplенные» в виде вертикальных полос спектры для каждого из последовательно идущих фрагментов.

```

1 def stft(array: np.ndarray):
2     """Short Timed Fourier Transform"""
3     tmp = []
4     for part in array:
5         tmp.append(spectrum(part))
6     return np.array(tmp)

```

Рис. 6.6. Функция вычисления КВПФ

Зададим сигнал y.

```

1 def y(x):
2     if x < np.pi:
3         return np.sin(0.5 * 2 * np.pi * x)
4     else:
5         return np.sin(0.8 * 2 * np.pi * x)

```

Рис. 6.7. Сигнал

И на основе этого сигнала у создадим 50 значений

```
1 SAMPLES_COUNT = 50
2 signal_array = np.zeros(SAMPLES_COUNT)
3 for i in range(1, SAMPLES_COUNT):
4     signal_array[i - 1] = y((2 * np.pi * i) / SAMPLES_COUNT)
```

Рис. 6.8. Сигнал

Теперь нужно реализовать функцию попадания (или не попадания) в поддиапазон диапазона частот, отрисовывать график мы будем при помощи `pcolormesh`, поэтому диапазоны будем создавать с учётом его особенностей. Возвращать данная функция будет две переменные: равномерно разделённый диапазон, и двумерный массив `binned[y][x]`, где элементы по `y` – это поддиапазон частот, а элементы по `x` – это значения спектра для частей, обращение к элементу по `y x` вернёт либо 0, либо 1, что соответствует попаданию или не попаданию в поддиапазон для соответственных гармоник спектра.

```
1 def make_bins(values: np.ndarray):
2     """
3     Создаём диапазоны для частей,
4     где за горизонтальное значение берётся
5     индекс части, а по вертикали не/попадание
6     в диапазон данных
7     """
8     bins = np.linspace(np.min(values), np.max(values), num=100)
9     binned = np.zeros(shape=(bins.shape[0], values.shape[0]), dtype=np.int8)
10
11     for line_idx in range(len(values)):
12         for el_idx in range(values[line_idx].size):
13             """Итерируемся по значениям частот"""
14             current_value = values[line_idx, el_idx]
15
16             for bin_idx in range(0, bins.size, 2):
17                 high_value = bins[bin_idx + 1]
18                 """
19                 Если частота попала в диапазон относительно
20                 большего значения поддиапозона, то выставляем попадание
21                 """
22                 if high_value >= current_value:
23                     binned[bin_idx - 1, line_idx] = 1
24                     break
25
26             """Добавляем ещё одно значение для pcolormesh (иначе график не будет отрисовываться)"""
27             bins = np.append(bins, np.max(values) + bins[0] - bins[1])
28     return bins, binned
```

Рис. 6.9. Функция попадания в диапазон

На основе всех реализованных функций теперь можно создать несколько графиков с разными размерами фрагментов (или же размерами окон). В данном случае будем использовать размеры, при которых массив из 50 значений поделится на равные части. Затем итерируемся по этим размерам и создаём на их основе фрагменты массива сигнала, для этих фрагментов высчитываем КВПФ. Создаём диапазоны и выводим всё полученное на графики

```
1  SIZES = [1, 5, 10, 25, 50,]
2  fig, axs = plt.subplots(nrows=1, ncols=len(SIZES), figsize=(15, 10))
3  fig.tight_layout(pad=2.5)
4  for idx, size in enumerate(SIZES):
5      fragment_size = size
6      fragments = fragmentize(signal_array, fragment_size)
7      stft_array = stft(fragments)
8
9      bins, values = make_bins(stft_array)
10     x_graph = np.arange(1, fragments.shape[0] + 2)
11     y_graph = bins
12     z_graph = values
13
14     axs[idx].set_title(f'Размер фрагмента: {fragment_size}')
15     axs[idx].set_ylabel(f'Частота')
16     axs[idx].set_xlabel(f'Номер фрагмента')
17     axs[idx].pcolormesh(
18         x_graph,
19         y_graph,
20         z_graph,
21         shading='flat',
22         vmin=stft_array.min(),
23         vmax=stft_array.max())
```

Рис. 6.10. Алгоритм построения графиков спектров

Итого получаем следующие графики

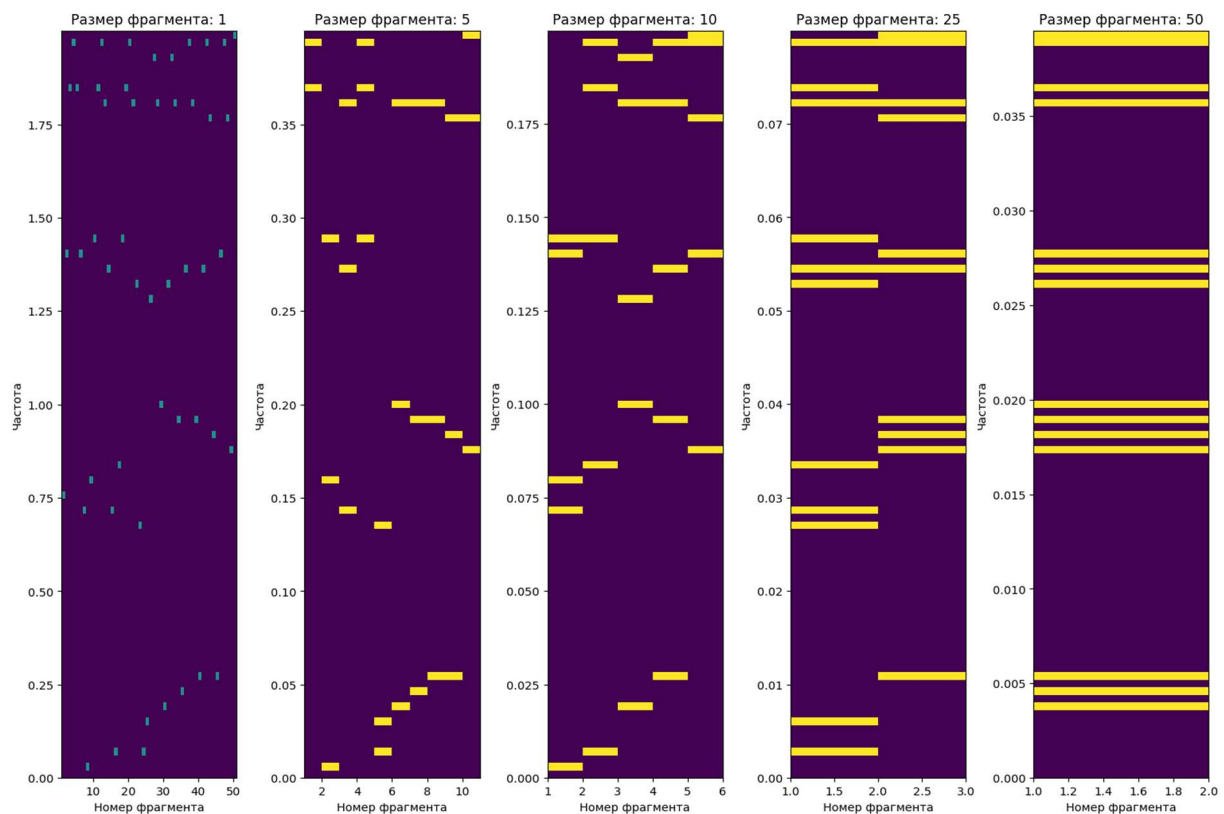


Рис. 6.11. Графики КВПФ для фрагментов разного размера

На основе этих графиков подтверждается основное свойство КВПФ: **при увеличении размера окна, увеличивается разрешение по частоте, но уменьшается разрешение по времени.** Иными словами, чем больше значений в фрагменте, тем больше частот мы можем увидеть в нём, но при этом точно сказать какая частота изменилась в определённый момент времени будет затруднительно, и наоборот для узкого окна, мы видим частоту в конкретный момент времени, но на большой дистанции мы не видим преобладающих частот.

Оконное преобразование Фурье

Еще одним способом анализа сигналов является оконное преобразование Фурье (ОКПФ). Его можно получить, если в выражении НПФ умножить под интегралом исследуемый сигнал на некоторое окно $W()$:

$$F(t, \omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(\tau) W(\tau - t) e^{-i\omega\tau} d\tau, \quad (6.1)$$

или для дискретного сигнала S :

$$F(m, \omega) = \frac{1}{\sqrt{2\pi}} \sum_{n=0}^N S_n \cdot W_{n-m} \cdot e^{-i\omega n}. \quad (6.2)$$

В таком виде выражение $F()$ для непрерывного и дискретного случаев задает, очевидно, ВКФ исследуемого сигнала с функцией, заданной произведением комплексной экспоненты на окно.

Назначение оконной функции – сгладить края анализируемого фрагмента сигнала, снизив выбросы. В качестве W выбирают функцию, которая на краях плавно приближается к нулю, и поскольку она умножается на сигнал, сам сигнал на краях также будет стремиться к нулю, из-за чего и сгладятся его края. Это необходимо, в первую очередь, когда речь идет о пофрагментном ДПФ – по сути, рассмотренное выше КВПФ является подобием ОКПФ (без плавного сдвига по времени), в котором используется прямоугольное окно (равно 0 вне фрагмента и 1 внутри фрагмента). Также очевидно, что оконная функция сдвигается вдоль сигнала, а значит **ОКПФ является частотно-временным способом анализа сигналов.**

Реализация ОКПФ

Обратите внимание, здесь используются функции из раздела КВПФ, если данный раздел вы делаете в отдельном документе – не забудьте скопировать в него зависимые функции.

Сперва выберем оконную функцию. Для простоты возьмем распространенное на практике окно Хэмминга.

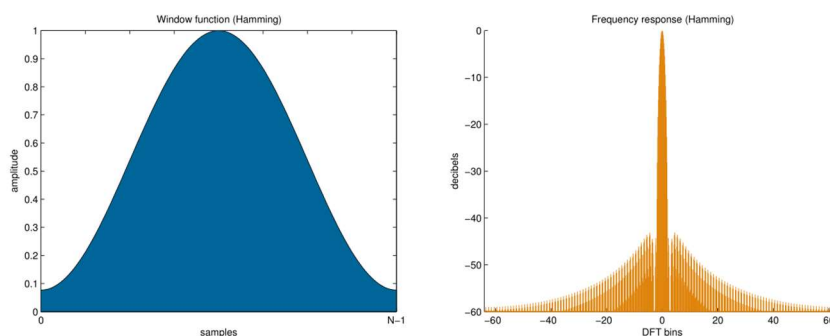


Рис. 6.12. Окно Хэмминга и его спектр

Окно задается выражением:

$$W_n = 0.5 \left(1 - \cos \left(\frac{2\pi n}{N-1} \right) \right),$$

где N - ширина окна. Чем шире окно, тем лучше разрешение по частоте, но хуже по времени и наоборот. В примере ниже N задает также число элементов в массиве окна.

```
1 def hem_window(N: int) -> np.ndarray:  
2     result = np.arange(start=1, stop=N + 1)  
3     result = 0.5 * (1 - np.cos(2 * ((np.pi * result) / (N - 1))))  
4     return result  
5 N = 10  
6 plot(np.arange(N), hem_window(N))
```

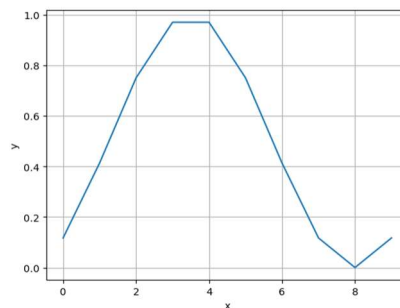


Рис. 6.13. Окно Хэмминга

Из предыдущей работы возьмём функцию `dft` и немного её видоизменим. Добавим внутреннюю функцию `window`, и в функции `F` значение элемента будем умножать на значения окна Хэмминга, если они попадают в диапазон этого окна, если они не попадают, то умножаем на 0.

```

1  def wft(array: np.ndarray,
2          m: int,
3          hem_window_array: np.ndarray) -> np.ndarray:
4      """ Windowed Fourier Transform.
5
6      Positional arguments
7      array -- массив всех частей
8      m -- индекс текущей части
9      hem_window_array -- массив окна Хэмминга
10     """
11     def _wft(
12         k: int,
13     ) -> np.imag:
14
15         def window(n: int, m: int):
16             """ Обнуление окна, если индекс выходит за границы массива """
17             if (
18                 n - m >= 0
19                 and
20                 n - m < hem_window_array.size
21             ):
22                 return hem_window_array[n - m]
23             return 0
24
25         def F(item, n):
26             return item * window(n, m) * np.e ** (
27                 (-1) * ((2 * np.pi * k * n * 1j) /
28                     array.size - 1)
29             )
30
31         result = np.zeros(array.size, dtype=np.complex_)
32         for idx in range(array.size):
33             result[idx] = F(array[idx], idx)
34         return np.sum(result)
35
36     result = np.zeros(array.shape, dtype=np.complex_)
37     for idx in range(array.size):
38         result[idx] = _wft(idx)
39
40     return result

```

Рис. 6.14. Алгоритм ОКПФ

На основе данных ОКПФ нужно получить спектр фрагментов, для этого реализуем функцию `wsp`.

```

1  def wsp(array: np.ndarray,
2         window_size: int):
3      """ Windowed Spectrum """
4
5      def _wsp(part_idx):
6          tmp = wft(array, part_idx, hem_window(window_size))
7          tmp = np.abs(tmp)
8          tmp = tmp[tmp.size // 2:]
9          return 2 * tmp / array.size
10
11     result = []
12     for idx in range(array.size):
13         result.append(_wsp(idx))
14     return np.array(result)

```

Рис. 6.15. Алгоритм создания массива спектрограммы

Из предыдущей работы возьмём функцию сигнала y и для него построим спектрограммы с разным размером окна Хэмминга, только на этот раз уменьшим количество элементов до 30 (чтобы повышение трудоемкости вычислений не вызывало существенное повышение времени исполнения).

```

1  SAMPLES_COUNT = 30
2  WINDOW_STEP = 5
3  fig, axs = plt.subplots(ncols=2, rows=(SAMPLES_COUNT // WINDOW_STEP) // 2, figsize=(12, 10))
4  fig.tight_layout(pad=1)
5  for idx, ax in enumerate(axs.ravel()):
6      window_size = (idx + 1) * WINDOW_STEP
7      signal_array = np.zeros(SAMPLES_COUNT)
8      for i in range(1, SAMPLES_COUNT):
9          signal_array[i - 1] = y((2 * np.pi * i) / SAMPLES_COUNT)
10
11     spectrums = wsp(signal_array, window_size)
12     spectrums = spectrums.T
13     x_range = np.arange(start=1, stop=SAMPLES_COUNT - 1)
14     y_bins = np.linspace(start=spectrums.min(), stop=spectrums.max(), num=spectrums.shape[0])
15     y_bins = np.flip(y_bins)
16     y_bins = np.round(y_bins, decimals=2)
17
18     im = ax.imshow(spectrums)
19     ax.set_xticks(np.arange(len(x_range)), labels=x_range)
20     ax.set_yticks(np.arange(len(y_bins)), labels=y_bins)
21     ax.set_title(f"Размер окна Хэмминга: {window_size}")
22 plt.show()

```

Рис. 6.16. Создание спектрограмм для разных размеров окна Хэмминга

Получим следующие графики:

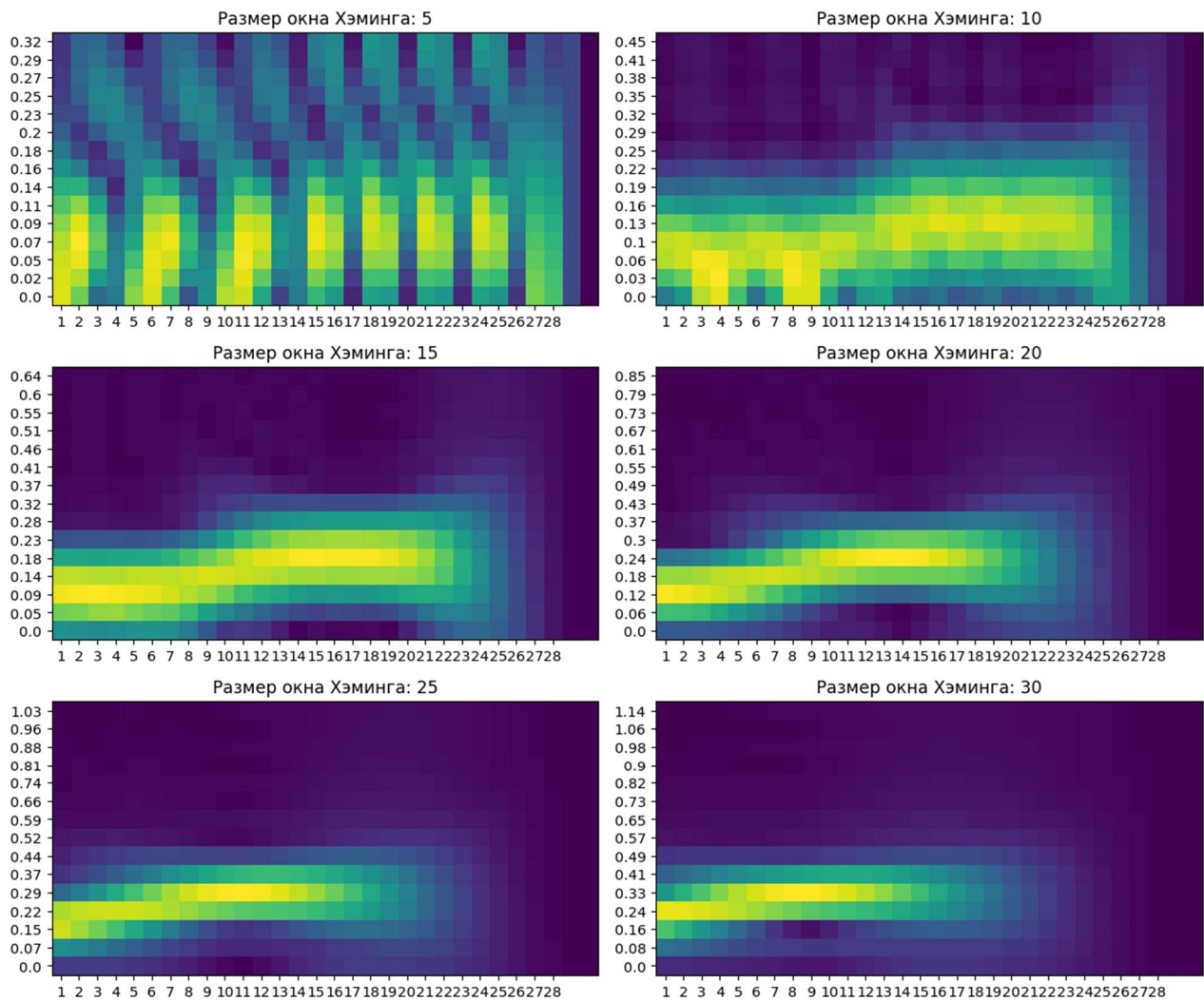


Рис. 6.17. Создание спектрограмм для разных размеров окна Хэмминга

Как раз эти графики и подтверждают, что **чем шире окно, тем лучше разрешение по частоте, но хуже по времени и наоборот.**

Задание

1. Для заданного вариантом сигнала (50 значений) реализуйте КВПФ при трех различных размерах фрагментов (size).
2. Выбрать size, при котором достигается наилучшее разрешение по времени и чистоте.
3. Для заданного сигнала реализуйте ОПФ (30 значений) для трех различных размеров окна (N)
4. Выбрать N, при котором достигается наилучшее разрешение по времени и частоте.
5. Сравнить полученные в пунктах КВПФ и ОКПФ спектрограммы. Привести в конце отчета эти два графика в одну строку. Сделать вывод о простоте визуального интерпретирования частотного состава сигнала (по какому из преобразований легче увидеть переход с одной частоты на другую).