# 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.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib widget
```

Рис. 6.1. Импортирование и конфигурирование зависимостей

Из предыдущей работы возьмём функцию построения простых графиков.

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

Рис. 6.2. Функция построения графиков

```
def dft(array: np.ndarray) -> np.ndarray:
    """ Discrete Fourier Transform"""
    def discrete_fourier_function(
        array: np.ndarray,
        k: int
    ) -> np.imag:

        def F(item, n):
            return item * np.e ** (
                (-1) * ((2 * np.pi * k * n * 1j) /
                        array.size - 1)
                    )

        result = np.zeros(array.size, dtype=np.complex_)
        for idx in range(array.size):
            result[idx] = F(array[idx], idx)
        return np.sum(result)

    result = np.zeros(array.shape, dtype=np.complex_)
    for idx in range(array.size):
        result[idx] = discrete_fourier_function(array, idx)
    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).

```python
def spectrum(array: np.ndarray) -> np.ndarray:
    tmp_freqs = np.abs(array)
    tmp_freqs = tmp_freqs[tmp_freqs.size // 2:]
    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.

```python
def fragmentize(
        array: np.ndarray,
        size: int) -> np.ndarray:
    fragments = np.array_split(array, range(size, array.size, size))
    fragments = np.array(fragments)
    return fragments
```

*Рис. 6.5.* Функция создания фрагментов

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

```python
def stft(array: np.ndarray):
    """Short Timed Fourier Transform"""
    tmp = []
    for part in array:
        tmp.append(spectrum(part))
    return np.array(tmp)
```

*Рис. 6.6.* Функция вычисления КВПФ

Зададим сигнал y.

```python
def y(x):
    if x < np.pi:
        return np.sin(0.5 * 2 * np.pi * x)
    else:
        return np.sin(0.8 * 2 * np.pi * x)
```

*Рис. 6.7.* Сигнал

И на основе этого сигнала y создадим 50 значений

```python
SAMPLES_COUNT = 50
signal_array = np.zeros(SAMPLES_COUNT)
for i in range(1, SAMPLES_COUNT):
    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 непопаданию в поддиапзон для соответственных гармоник спектра.

```python
def make_bins(values: np.ndarray):
    """
    Создаём диапазоны для частей,
    где за горизонтальное значение берётся
    индекс части, а по вертикали не/попадание
    в диапазон данных
    """
    bins = np.linspace(np.min(values), np.max(values), num=100)
    binned = np.zeros(shape=(bins.shape[0], values.shape[0]), dtype=np.int8)

    for line_idx in range(len(values)):
        for el_idx in range(values[line_idx].size):
            """Итерируемся по значениям частот"""
            current_value = values[line_idx, el_idx]

            for bin_idx in range(0, bins.size, 2):
                high_value = bins[bin_idx + 1]
                """
                Если частота попала в диапазон относительно
                большего значения поддиапозона, то выставляем попадание
                """
                if high_value >= current_value:
                    binned[bin_idx - 1, line_idx] = 1
                    break

    """Добавляем ещё одно значение для pcolormesh (иначе график не будет отрисовываться)"""
    bins = np.append(bins, np.max(values) + bins[0] - bins[1])
    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

```
SIZES = [1, 5, 10, 25, 50,]
fig, axs = plt.subplots(nrows=1, ncols=len(SIZES), figsize=(15, 10))
fig.tight_layout(pad=2.5)
for idx, size in enumerate(SIZES):
    fragment_size = size
    fragments = fragmentize(signal_array, fragment_size)
    stft_array = stft(fragments)

    bins, values = make_bins(stft_array)
    x_graph = np.arange(1, fragments.shape[0] + 2)
    y_graph = bins
    z_graph = values

    axs[idx].set_title(f'Размер фрагмента: {fragment_size}')
    axs[idx].set_ylabel(f'Частота')
    axs[idx].set_xlabel(f'Номер фрагмента')
    axs[idx].pcolormesh(
        x_graph,
        y_graph,
        z_graph,
        shading='flat',
        vmin=stft_array.min(),
        vmax=stft_array.max())
```

*Рис. 6.10.* Алгоритм построения графиков спектров

Итого получаем следующие графики

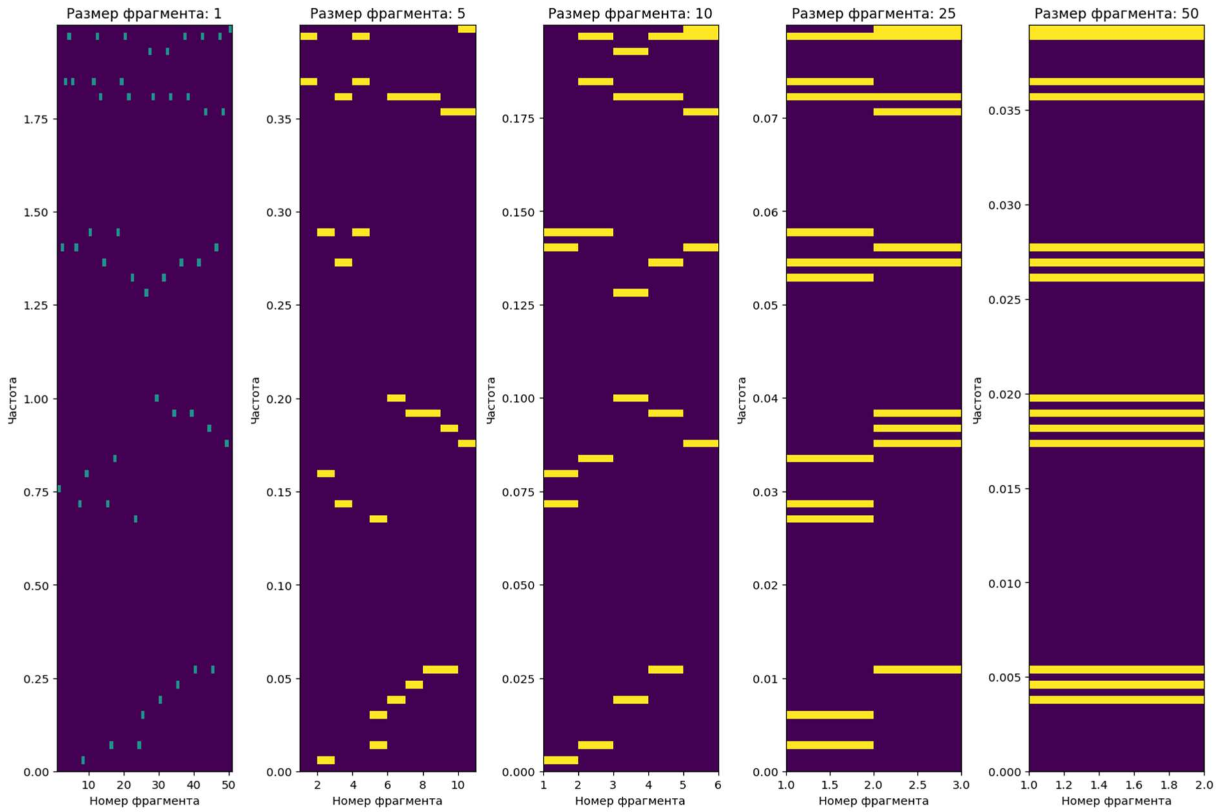


*Рис. 6.11.* Графики КВПФ для фрагментов разного размера

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():

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

или для дискретного сигнала S:

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

В таком виде выражение F() для непрерывного и дискретного случаев задает, очевидно, ВКФ исследуемого сигнала с функицей, 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.

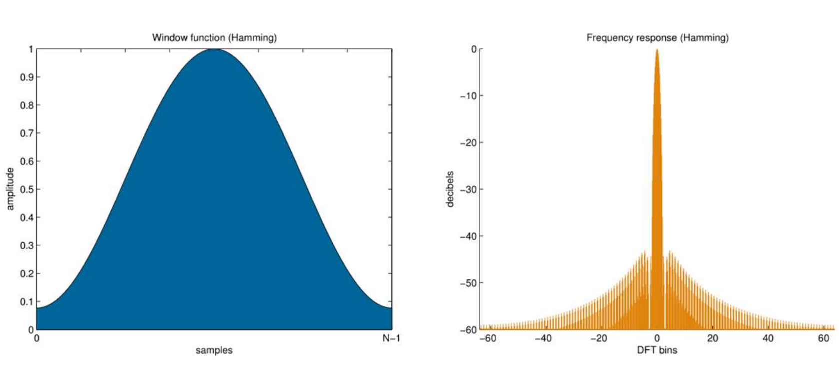


*Рис. 6.12.* Окно Хэмминга и его спектр

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.

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

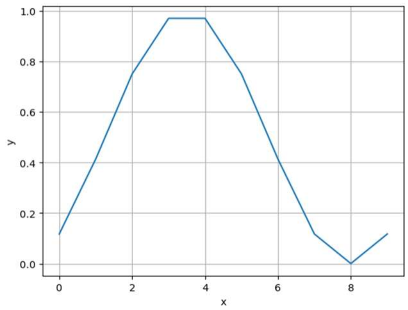


*Рис. 6.13.* Окно Хэмминга

Из предыдущей работы возьмём функцию dft и немного её видоизменим. Добавим внутреннею 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.

```
def wft(array: np.ndarray,
        m: int,
        hem_window_array: np.ndarray) -> np.ndarray:
    """ Windowed Fourier Transform.

    Positional arguments
    array -- массив всех частей
    m -- индекс текущей части
    hem_window_array -- массив окна Хэмминга
    """
    def _wft(
        k: int,
    ) -> np.imag:

        def window(n: int, m: int):
            """ Обнуление окна, если индекс выходит за границы массива"""
            if (
                n - m >= 0
                and
                n - m < hem_window_array.size
            ):
                return hem_window_array[n - m]
            return 0

        def F(item, n):
            return item * window(n, m) * np.e ** (
                (-1) * ((2 * np.pi * k * n * 1j) /
                        array.size - 1)
                    )

        result = np.zeros(array.size, dtype=np.complex_)
        for idx in range(array.size):
            result[idx] = F(array[idx], idx)
        return np.sum(result)

    result = np.zeros(array.shape, dtype=np.complex_)
    for idx in range(array.size):
        result[idx] = _wft(idx)

    return result
```

*Рис. 6.14.* Алгоритм ОКПФ

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

```python
def wsp(array: np.ndarray,
        window_size: int):
    """ Windowed Spectrum """

    def _wsp(part_idx):
        tmp = wft(array, part_idx, hem_window(window_size))
        tmp = np.abs(tmp)
        tmp = tmp[tmp.size // 2:]
        return 2 * tmp / array.size

    result = []
    for idx in range(array.size):
        result.append(_wsp(idx))
    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).

```python
SAMPLES_COUNT = 30
WINDOW_STEP = 5
fig, axs = plt.subplots(ncols=2, nrows=(SAMPLES_COUNT // WINDOW_STEP) // 2, figsize=(12, 10))
fig.tight_layout(pad=1)
for idx, ax in enumerate(axs.ravel()):
    window_size = (idx + 1) * WINDOW_STEP
    signal_array = np.zeros(SAMPLES_COUNT)
    for i in range(1, SAMPLES_COUNT):
        signal_array[i - 1] = y((2 * np.pi * i) / SAMPLES_COUNT)

    spectrums = wsp(signal_array, window_size)
    spectrums = spectrums.T
    x_range = np.arange(start=1, stop=SAMPLES_COUNT - 1)
    y_bins = np.linspace(start=spectrums.min(), stop=spectrums.max(), num=spectrums.shape[0])
    y_bins = np.flip(y_bins)
    y_bins = np.round(y_bins, decimals=2)

    im = ax.imshow(spectrums)
    ax.set_xticks(np.arange(len(x_range)), labels=x_range)
    ax.set_yticks(np.arange(len(y_bins)), labels=y_bins)
    ax.set_title(f"Размер окна Хэминга: {window_size}")
plt.show()
```

*Рис. 6.16.* Создание спектрограмм для разных размеров окна Хэмминга

Получим следующие графики:

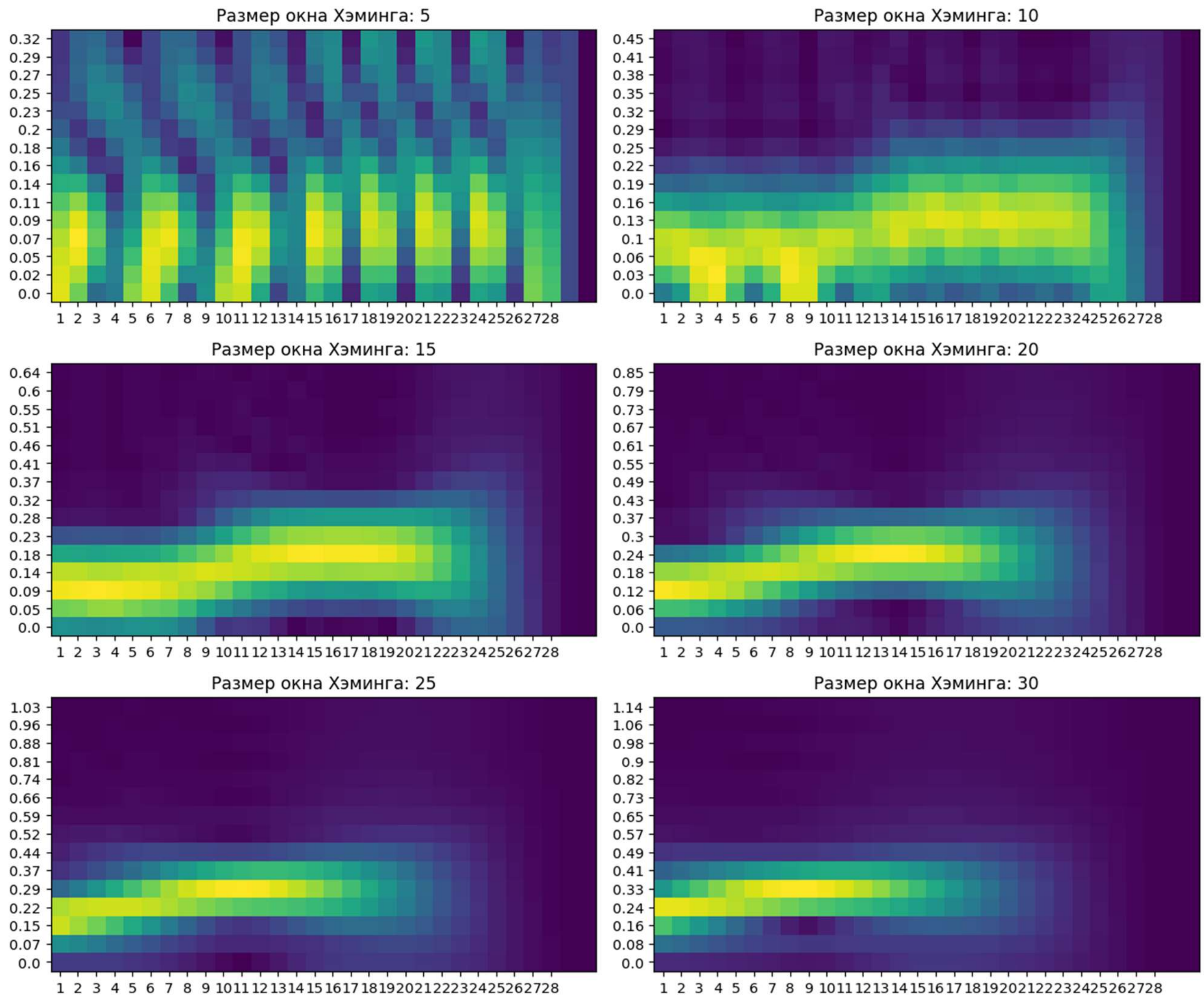


*Рис. 6.17.* Создание спектрограмм для разных размеров окна Хэмминга

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).