

Практическое занятие №6

Армбристер Никита Владиславович

Группа ПМИ-31

Вариант 1

**Цель:** сформировать практические навыки применения алгоритмов прямого и обратного дискретного преобразования Фурье (DFT и IDFT) и их быстрых реализаций (FFT и IFFT) для анализа одномерных сигналов

**Задание:**

1. На языке программирования C++ реализовать алгоритмы прямого и обратного преобразований Фурье (DFT и IDFT), быстрые алгоритмы прямого и обратного преобразований Фурье для вектора чётной длины (FFT и IFFT). Арифметические операции выполнять при использовании шаблонного класса `std::complex`.
2. Задать  $N=2^n, n \in \mathbb{N}$  отсчётов зашумлённого сигнала вида:

$$z(j) = A \cos(2\pi \omega_1 j/N + \varphi) + B \cos(2\pi \omega_2 j/N)$$

3. Выполнить для полученного набора данных дискретное преобразование Фурье. Замерить время выполнения кода при использовании DFT и FFT. Заполнить таблицу (отразить только те частоты  $m$ , на которых отличны от нуля амплитудный или фазовый спектры)

$m$	$\Re z$	$\Re \hat{z}$	$\Im \hat{z}$	Амплитуда, $ \hat{z} $	Фаза, $\varphi$

4. Найти высокочастотные шумовые компоненты сигнала и обнулите их коэффициенты  $\hat{z}$  в DFT
5. Вычислить IDFT от модифицированного результата DFT (с обнулёнными шумовыми компонентами). Изобразить на одном графике исходный набор значений и результат фильтрации после IDFT. Графики должны быть непрерывными, используйте сплайны.
6. Задать  $N=2^n, n \in \mathbb{N}$  отсчетов зашумленного сигнала вида:

$$z(j) = \begin{cases} 0, & 0 \leq j < \frac{N}{4} \\ A + B \cos\left(\frac{2\pi \omega_2 j}{N}\right), & \frac{N}{4} \leq j \leq \frac{N}{2} \\ 0, & \frac{N}{2} < j \leq \frac{3N}{4} \\ A + B \cos\left(\frac{2\pi \omega_2 j}{N}\right), & \frac{3N}{4} \leq j \leq N \end{cases}$$

Получится ли эффективно решить задачу о фильтрации данного сигнала с помощью DFT? Ответ обоснуйте

**Вариант 1:**

n	A	B	$\omega_1$	$\omega_2$	$\varphi$
9	2.44	0.1	1	184	$\frac{\pi}{4}$

**Ссылка на репозиторий с проектом:** <https://github.com/Gribnik24/NumericalMethods-PracticalTasks/tree/main/Task%20%232.6%20Fourier%20transform>

**Решение:**

- 1) Реализуем на языке C++ алгоритмы прямого преобразования Фурье DFT:

```
vector<Complex> dft(const vector<Complex>& input) {
    int N = input.size();
    vector<Complex> output(N, 0);

    for (int m = 0; m < N; m++) {
        for (int n = 0; n < N; n++) {
            double angle = -2 * PI * m * n / N;
            output[m] += input[n] * exp(Complex(0, angle));
        }
    }
    return output;
}
```

Обратного преобразования Фурье IDFT:

```
vector<Complex> idft(const vector<Complex>& input) {
    int N = input.size();
    vector<Complex> output(N, 0);

    for (int n = 0; n < N; n++) {
        for (int m = 0; m < N; m++) {
            double angle = 2 * PI * m * n / N;
            output[n] += input[m] * exp(Complex(0, angle));
        }
        output[n] /= N;
    }
    return output;
}
```

Быстрый алгоритм прямого преобразования Фурье FFT:

```

vector<Complex> fft(const vector<Complex>& input) {
    int N = input.size();

    if (N == 1) {
        return input;
    }

    int M = N / 2;

    // 1. Разделяем на четные и нечетные
    vector<Complex> u(M), v(M);
    for (int k = 0; k < M; k++) {
        u[k] = input[2 * k];      // z(2k)
        v[k] = input[2 * k + 1];  // z(2k+1)
    }

    // 2. Рекурсивно вычисляем FFT для половин
    vector<Complex> u_hat = fft(u);
    vector<Complex> v_hat = fft(v);

    // 3. Объединяем результаты
    vector<Complex> output(N);

    for (int m = 0; m < M; m++) {
        // 3. Для m = 0...M-1
        double angle = -2 * PI * m / N;
        Complex twiddle = exp(Complex(0, angle));
        output[m] = u_hat[m] + twiddle * v_hat[m];

        // 4. Для m = M...N-1
        int l = m;
        output[l + M] = u_hat[l] - twiddle * v_hat[l];
    }

    return output;
}

```

Обратного преобразования Фурье IFFT:

```
vector<Complex> ifft(const vector<Complex>& input) {
    int N = input.size();

    //  $z(-j) = z(N-j)$  - используем свойство периодичности
    vector<Complex> conjugated_input(N);
    for (int j = 0; j < N; j++) {
        conjugated_input[j] = conj(input[j]);
    }

    // Вычисляем FFT от сопряженного
    vector<Complex> temp = fft(conjugated_input);

    // Сопрягаем результат и делим на N
    vector<Complex> output(N);
    for (int j = 0; j < N; j++) {
        output[j] = conj(temp[j]) / double(N);
    }

    return output;
}
```

- 2) Зададим  $N=2^9=512$  отсчетов зашумленного сигнала вида

$$z(j) = A \cos(2\pi \omega_1 j / N + \varphi) + B \cos(2\pi \omega_2 j / N)$$

```
struct SignalParams {
    int N;
    double A, B;
    double omega1, omega2;
    double phi;
};

vector<Complex> generateSignal1(const SignalParams& params) {
    vector<Complex> signal(params.N);
    for (int j = 0; j < params.N; j++) {
        double value = params.A * cos(2 * PI * params.omega1 * j / params.N + params.phi) +
            params.B * cos(2 * PI * params.omega2 * j / params.N);
        signal[j] = value;
    }
    return signal;
}
```

- 3) Выполним для полученного набора данных дискретное преобразование Фурье. Замерим время выполнения кода при использовании DFT и FFT:

```
struct TimingResults {
    long long dft_time;
    long long fft_time;
};
```

```

AnalysisResults analyzeSignal(const vector<Complex>& signal) {
    AnalysisResults results;

    auto start_dft = high_resolution_clock::now();
    results.dft_result = dft(signal);
    auto end_dft = high_resolution_clock::now();

    auto start_fft = high_resolution_clock::now();
    results.fft_result = fft(signal);
    auto end_fft = high_resolution_clock::now();

    results.timing.dft_time = duration_cast<microseconds>(end_dft - start_dft).count();
    results.timing.fft_time = duration_cast<microseconds>(end_fft - start_fft).count();
    results.timing.speedup = double(results.timing.dft_time) / results.timing.fft_time;

    return results;
}

```

По заданию таблицу в пункте 3 необходимо заполнить только теми частотами  $m$ , где фазовый ИЛИ амплитудный спектр отличны от 0. Если придерживаться условия, то в таблицу у меня попадали фактически все значения так как там где амплитуда была равна 0, фаза от 0 отличалась:

```

void printResultsTable(const vector<Complex>& signal, const vector<Complex>& dft_result) {
    cout << fixed << setprecision(6);
    cout << setw(4) << "m" << setw(12) << "Re z" << setw(15) << "Re z_hat"
        << setw(15) << "Im z_hat" << setw(15) << "Amplitude" << setw(12) << "Phase" << endl;
    cout << string(75, '-') << endl;

    int N = signal.size();
    double amplitude_limit = 1e-6;
    double phase_limit = 1e-6;
    int count = 0;

    for (int m = 0; m < N; m++) {
        double amplitude = abs(dft_result[m]);
        double phase = arg(dft_result[m]);

        bool significant_amplitude = (amplitude > amplitude_limit);
        bool significant_phase = (abs(phase) > phase_limit);

        if (significant_amplitude || significant_phase) { // вариант с '|| significant_phase'
            cout << setw(4) << m << setw(12) << signal[m].real()
                << setw(15) << dft_result[m].real()
                << setw(15) << dft_result[m].imag()
                << setw(15) << amplitude
                << setw(12) << phase << endl;
            count++;
        }
    }

    cout << defaultfloat;
    cout << "Всего выведено компонент: " << count << " из " << N << endl;
}

```

Таблица значимых компонент DFT:					
m	Re z	Re z_hat	Im z_hat	Amplitude	Phase
1	1.640599	441.687180	441.687180	624.640000	0.785398
2	1.662970	0.000000	-0.000000	0.000000	-1.143959
3	1.748859	-0.000000	0.000000	0.000000	2.983157
4	1.546216	-0.000000	-0.000000	0.000000	-2.738580
5	1.645323	-0.000000	-0.000000	0.000000	-2.332014
6	1.649299	-0.000000	0.000000	0.000000	2.949058
7	1.471430	0.000000	0.000000	0.000000	1.407664
8	1.618630	0.000000	0.000000	0.000000	1.294596
506	1.903147	-0.000000	0.000000	0.000000	1.809061
507	1.856921	0.000000	-0.000000	0.000000	-0.111611
508	1.715533	0.000000	-0.000000	0.000000	-0.052696
509	1.875869	0.000000	-0.000000	0.000000	-0.404094
510	1.747654	0.000000	-0.000000	0.000000	-0.514675
511	1.682944	441.687180	-441.687180	624.640000	-0.785398
Всего выведено компонент: 511 из 512					

Но если убрать из условия отбора фазу, то можно выделить как раз лишь несколько значений:

```
void printResultsTable(const vector<Complex>& signal, const vector<Complex>& dft_result) {
    cout << fixed << setprecision(6);
    cout << setw(4) << "m" << setw(12) << "Re z" << setw(15) << "Re z_hat"
        << setw(15) << "Im z_hat" << setw(15) << "Amplitude" << setw(12) << "Phase" << endl;
    cout << string(75, '-') << endl;

    int N = signal.size();
    double amplitude_limit = 1e-6;
    double phase_limit = 1e-6;
    int count = 0;

    for (int m = 0; m < N; m++) {
        double amplitude = abs(dft_result[m]);
        double phase = arg(dft_result[m]);

        bool significant_amplitude = (amplitude > amplitude_limit);
        bool significant_phase = (abs(phase) > phase_limit);

        if (significant_amplitude) { // вариант без '|| significant_phase'
            cout << setw(4) << m << setw(12) << signal[m].real()
                << setw(15) << dft_result[m].real()
                << setw(15) << dft_result[m].imag()
                << setw(15) << amplitude
                << setw(12) << phase << endl;
            count++;
        }
    }

    cout << defaultfloat;
    cout << "Всего выведено компонент: " << count << " из " << N << endl;
}
```

Таблица значимых компонент DFT:					
m	Re z	Re z_hat	Im z_hat	Amplitude	Phase
1	1.640599	441.687180	441.687180	624.640000	0.785398
184	-2.357540	25.600000	0.000000	25.600000	0.000000
328	0.309873	25.600000	0.000000	25.600000	0.000000
511	1.682944	441.687180	-441.687180	624.640000	-0.785398
Всего выведено компонент: 4 из 512					

- 4) Найдем высокочастотные шумовые компоненты сигнала и обнулим их коэффициенты  $\hat{z}$  в DFT. Для этого возьмем 10% измерений, которые содержат наименьшие по модулю частоты.

```
vector<Complex> filterHighFrequencies(const vector<Complex>& dft_result) {
    int N = dft_result.size();
    vector<Complex> filtered = dft_result;

    // Оставляем только низкие частоты (первые и последние 10%)
    int keep_count = N / 10; // 10% от N

    cout << "Простая фильтрация:" << endl;
    cout << "Сохраняем частоты: m = 0..." << keep_count << " и " << N - keep_count << "...N-1" << endl;
    cout << "Обнуляем все остальные частоты" << endl;

    int removed_count = 0;
    for (int k = keep_count + 1; k < N - keep_count; k++) {
        if (abs(filtered[k]) > 1e-6) {
            removed_count++;
        }
        filtered[k] = 0;
    }

    cout << "Удалено компонент с ненулевой амплитудой: " << removed_count << endl;
    return filtered;
}
```

```
=====
ПУНКТ 4: ФИЛЬТРАЦИЯ ШУМОВЫХ КОМПОНЕНТ

Простая фильтрация:
Сохраняем частоты: m = 0...51 и 461...N-1
Обнуляем все остальные частоты
Удалено компонент с ненулевой амплитудой: 2
```

- 5) Вычислим IDFT от модифицированного результата DFT и выполним экспорт данных в .csv формат:

```

void exportToCSV(const string& filename,
    const vector<Complex>& original_signal,
    const vector<Complex>& filtered_signal,
    const vector<Complex>& dft_original,
    const vector<Complex>& dft_filtered) {
    ofstream file(filename);

    file << "index,original_signal_real,filtered_signal_real,"
        << "dft_original_real,dft_original_imag,dft_original_amplitude,"
        << "dft_filtered_real,dft_filtered_imag,dft_filtered_amplitude" << endl;

    int N = original_signal.size();
    for (int i = 0; i < N; i++) {
        file << i << ","
            << original_signal[i].real() << ","
            << filtered_signal[i].real() << ","
            << dft_original[i].real() << ","
            << dft_original[i].imag() << ","
            << abs(dft_original[i]) << ","
            << dft_filtered[i].real() << ","
            << dft_filtered[i].imag() << ","
            << abs(dft_filtered[i]) << endl;
    }

    file.close();
    cout << "Данные экспортированы в файл: " << filename << endl;
}

```

```

// ===== ПУНКТ 5 =====
printSectionHeader("ПУНКТ 5: ЭКСПОРТ ДАННЫХ ДЛЯ ВИЗУАЛИЗАЦИИ");
vector<Complex> reconstructed = idft(filtered_dft);

exportToCSV("signal_analysis.csv", signal, reconstructed,
    analysis.dft_result, filtered_dft);

```

По заданию необходимо на графике изобразить исходный набор значений и результат фильтрации после IDFT. Также указано, что графики должны быть непрерывными, для этого необходимо использовать сплайны. Пользуясь наработками из 3 практического задания, которое было посвящено изучению сглаживающих сплайнов, воспользуемся отсюда классом SmoothingSpline. Напишем функцию plot\_task5\_results, с помощью которой изобразим график с применением сплайна. Параметр сглаживания  $r$  я взял за 0.01 – подобная величина позволит сохранить и отразить визуально исходный шум:

```
def plot_task5_results():
    """График для пункта 5: исходный и отфильтрованный сигнал"""
    # Чтение данных
    df = pd.read_csv('signal_analysis.csv')

    # Создание сплайнов
    indices = df['index'].values
    original_signal = df['original_signal_real'].values
    filtered_signal = df['filtered_signal_real'].values

    # Сплайн для исходного сигнала
    spline_original = SmoothingSpline()
    spline_original.initialize(indices, original_signal, 0.01)
    original_smooth = [spline_original.evaluate(x) for x in indices]

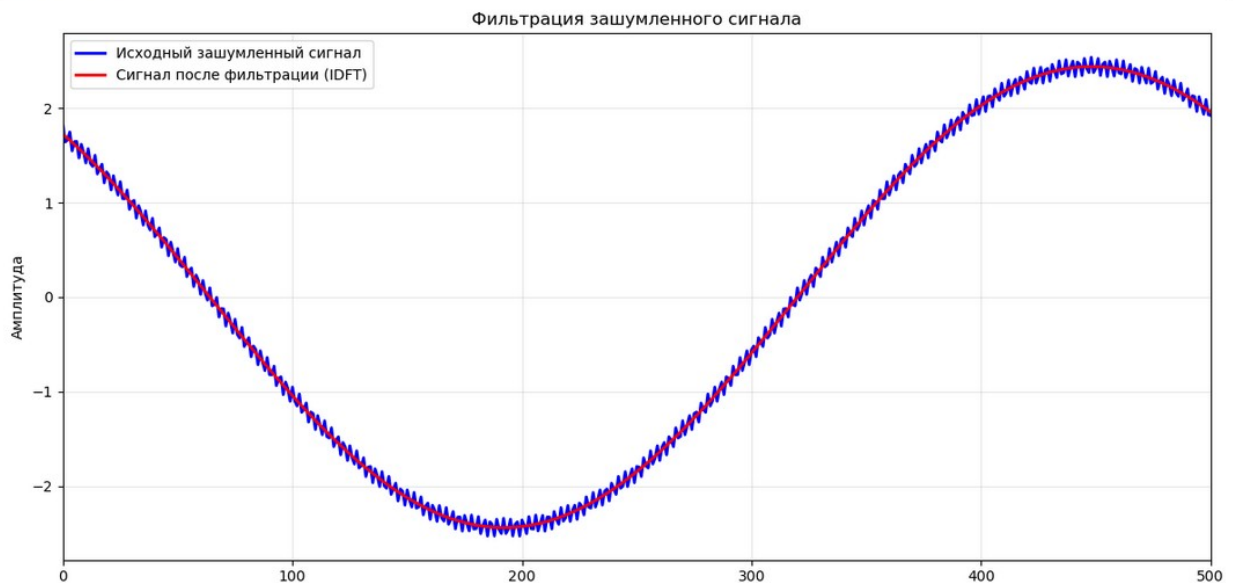
    # Сплайн для фильтрованного сигнала
    spline_filtered = SmoothingSpline()
    spline_filtered.initialize(indices, filtered_signal, 0.01)
    filtered_smooth = [spline_filtered.evaluate(x) for x in indices]

    # Построение графика
    plt.figure(figsize=(12, 6))
    plt.plot(indices, original_smooth, 'b-', linewidth=2, label='Исходный зашумленный сигнал')
    plt.plot(indices, filtered_smooth, 'r-', linewidth=2, label='Сигнал после фильтрации (IDFT)')

    plt.title('Фильтрация зашумленного сигнала')
    plt.xlabel('Время')
    plt.ylabel('Амплитуда')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.xlim(0, 500)

    plt.tight_layout()
    plt.show()
```

```
plot_task5_results()
```



6) Изменим вид сигнала из пункта 2 под новый формат:

$$z(j) = \begin{cases} 0, & 0 \leq j < \frac{N}{4} \\ A + B \cos\left(\frac{2\pi\omega_2 j}{N}\right), & \frac{N}{4} \leq j \leq \frac{N}{2} \\ 0, & \frac{N}{2} < j \leq \frac{3N}{4} \\ A + B \cos\left(\frac{2\pi\omega_2 j}{N}\right), & \frac{3N}{4} \leq j \leq N \end{cases}$$

Зададим его и также экспортируем в .csv формат

```
vector<Complex> generateSignal2(const SignalParams& params) {
    vector<Complex> signal(params.N, 0);
    for (int j = params.N / 4; j <= params.N / 2; j++) {
        signal[j] = params.A + params.B * cos(2 * PI * params.omega2 * j / params.N);
    }
    for (int j = 3 * params.N / 4; j < params.N; j++) {
        signal[j] = params.A + params.B * cos(2 * PI * params.omega2 * j / params.N);
    }
    return signal;
}
```

```
void exportDiscontinuousSignal(const string& filename, const vector<Complex>& signal) {
    ofstream file(filename);

    file << "index,signal_real" << endl;

    int N = signal.size();
    for (int i = 0; i < N; i++) {
        file << i << "," << signal[i].real() << endl;
    }

    file.close();
    cout << "Данные нового сигнала экспортированы в файл: " << filename << endl;
}

void analyzeDiscontinuousSignal(const SignalParams& params) {
    cout << "\n" << string(70, '=') << "\n";
    cout << "АНАЛИЗ НОВОГО СИГНАЛА (ПУНКТ 6)" << "\n";

    // Генерируем сигнал с разрывами
    vector<Complex> signal = generateSignal2(params);

    // Экспортируем для визуализации
    exportDiscontinuousSignal("discontinuous_signal.csv", signal);
}
```

Напишем функцию для визуализации только этих исходных данных:

```
def plot_task6_results():
    """График для пункта 6: кусочно-постоянный сигнал"""
    # Чтение данных
    df = pd.read_csv('discontinuous_signal.csv')

    # Создание сплайнов
    indices = df['index'].values
    signal = df['signal_real'].values

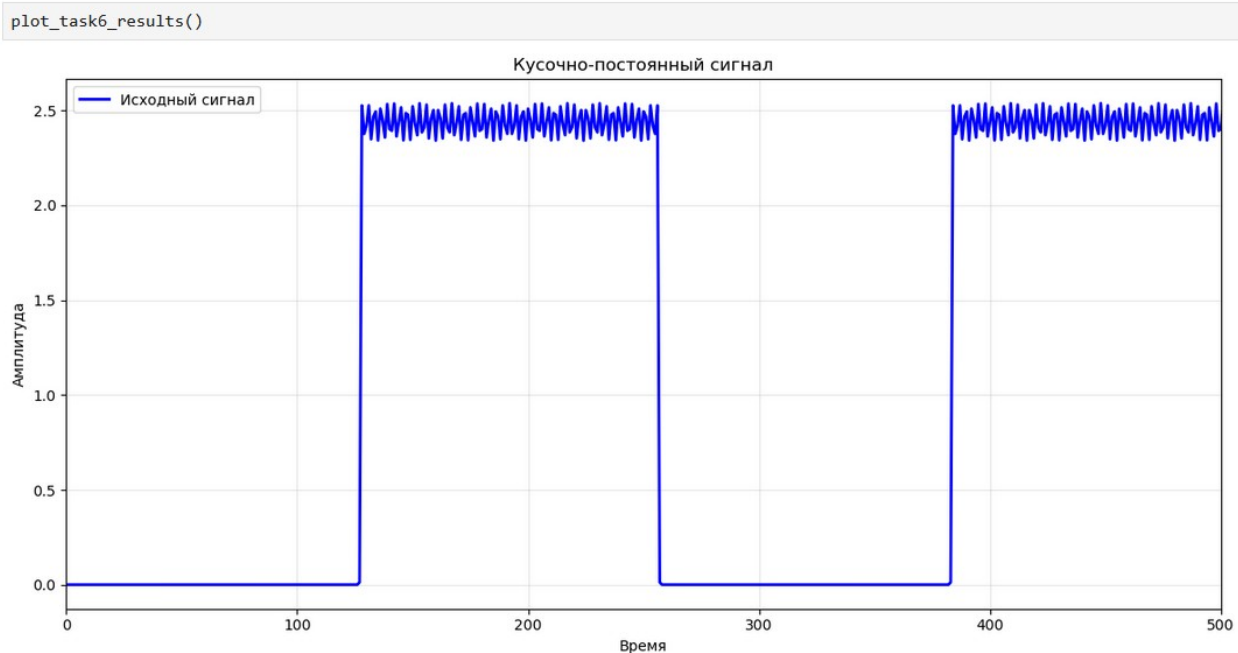
    # Сплайн для сигнала
    spline = SmoothingSpline()
    spline.initialize(indices, signal, 0.01)
    signal_smooth = [spline.evaluate(x) for x in indices]

    # Построение графика
    plt.figure(figsize=(12, 6))
    plt.plot(indices, signal_smooth, 'b-', linewidth=2, label='Исходный сигнал')

    plt.title('Кусочно-постоянный сигнал')
    plt.xlabel('Время')
    plt.ylabel('Амплитуда')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.xlim(0, 500)

    plt.tight_layout()
    plt.show()
```

Применим ее и получим такой график:



Замечаем, что данный сигнал будет кусочно-заданным и даже цикличным. Сначала идет промежуток с постоянным значением амплитуды 0, потом некие зашумленные данные.

Для подобного сигнала DFT не позволит решить задачу фильтрации эффективно. Каждый такой резкий скачок создает мгновенно бесконечно много высоких частот (явление Гиббса) и эти частоты не будут являться шумом, они отражают логичную информацию об изменении формы сигнала. В обычно плавном сигнале (как например в пункте 2) высокие частоты – это шум, в этом же сигнале – информация о том, где скачки. Если высокие частоты убрать, то эти скачки размажутся. Это будет уже не фильтрацией, а скорее искажением. DFT хорошо работает именно с плавными сигналами, но портит сигналы с резкими скачками по причине того, что там фактически происходит разложение на косинусы и синусы, которые являются гладкими функциями.

Для сигнала из пункта 6 лучше сработает вейвлет-преобразование, так как оно обладает «локальным» анализом, которое позволит отследить подобные скачки