Цель:

Изучение методов Фурье-анализа дискретных и цифровых сигналов.

Ход работы:

1. **Написать программу вычисления прямого и обратного дискретного преобразования Фурье в матричной форме.**

Дискретное преобразование Фурье является наиболее популярным математическим аппаратом в Фурье-анализе, поскольку оно применимо к цифровым сигналам и для данного преобразования существует множество вариантов быстрых реализаций, которые реализуются в сигнальных процессорах.

Формула прямого дискретного преобразования Фурье задается как:

$$\dot{U}_k = \frac{1}{N} \sum_{n=0}^{N-1} u_n e^{-j\frac{2\pi n}{N}k} \tag{1}$$

Нормирующий коэффициент 1/N, как правило, применяется при вычислении обратного ДПФ вместо прямого. Обратное дискретное преобразование Фурье задается формулой:

$$u_n = \sum_{k=0}^{N-1} \left(\sum_{m=-\infty}^{\infty} \dot{C}_{k+mN} \right) e^{j\frac{2\pi n}{N}k} = \sum_{k=0}^{N-1} \dot{U}_k e^{j\frac{2\pi n}{N}k} \tag{2}$$

где n принимает значения от 0 до N − 1.

Поскольку прямое и обратное ДПФ можно интерпретировать в терминах операций над векторами, это преобразование удобно представлять в матричной форме соответственно

$$\vec{U} = \vec{u}\, F^H \tag{3}$$

$$\vec{u} = \vec{U}\, F \tag{4}$$

Матрица F имеет вид:

$$\mathbf{F} = \begin{pmatrix} e^{j\frac{2\pi 0}{N}0} & e^{j\frac{2\pi 1}{N}0} & \dots & e^{j\frac{2\pi (N-1)}{N}0} \\ e^{j\frac{2\pi 0}{N}1} & e^{j\frac{2\pi 1}{N}1} & \dots & e^{j\frac{2\pi (N-1)}{N}1} \\ \vdots & \vdots & \ddots & \vdots \\ e^{j\frac{2\pi 0}{N}(N-1)} & e^{j\frac{2\pi 1}{N}(N-1)} & \dots & e^{j\frac{2\pi (N-1)}{N}(N-1)} \end{pmatrix},$$

а F^H является эрмитово сопряженной с F матрицей.

Будем проверять работу программы на примере функции вида $u(t) = \sin(2\pi ft)$ с частотой $f = 20$ Гц. Построим сигнал:

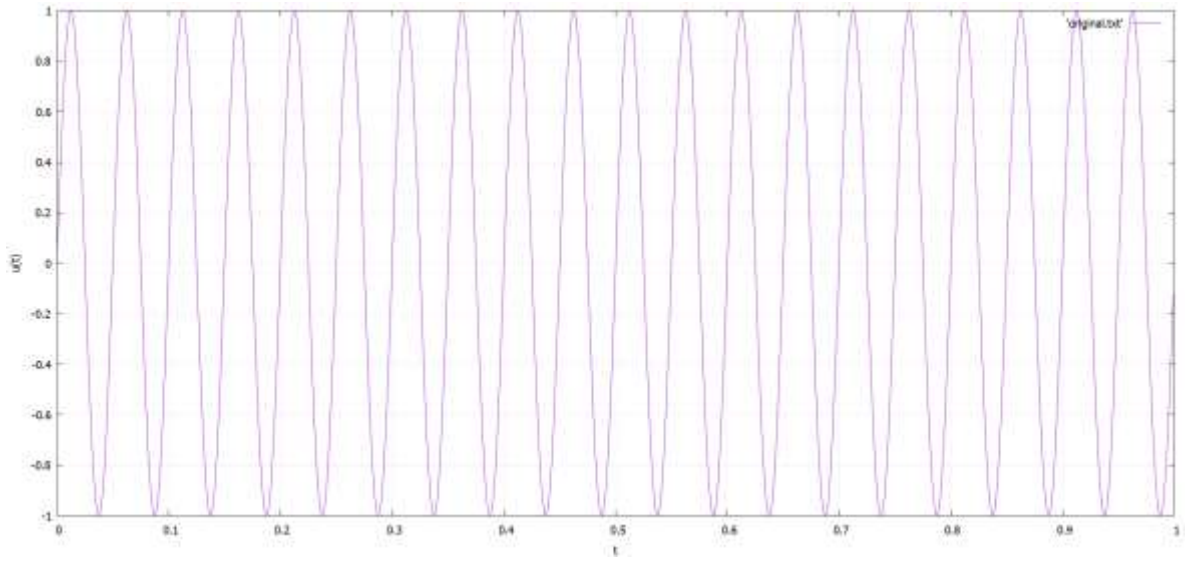

Рисунок 1 – График исходного сигнала

В результате работы прямого ДПФ получили спектр сигнала:

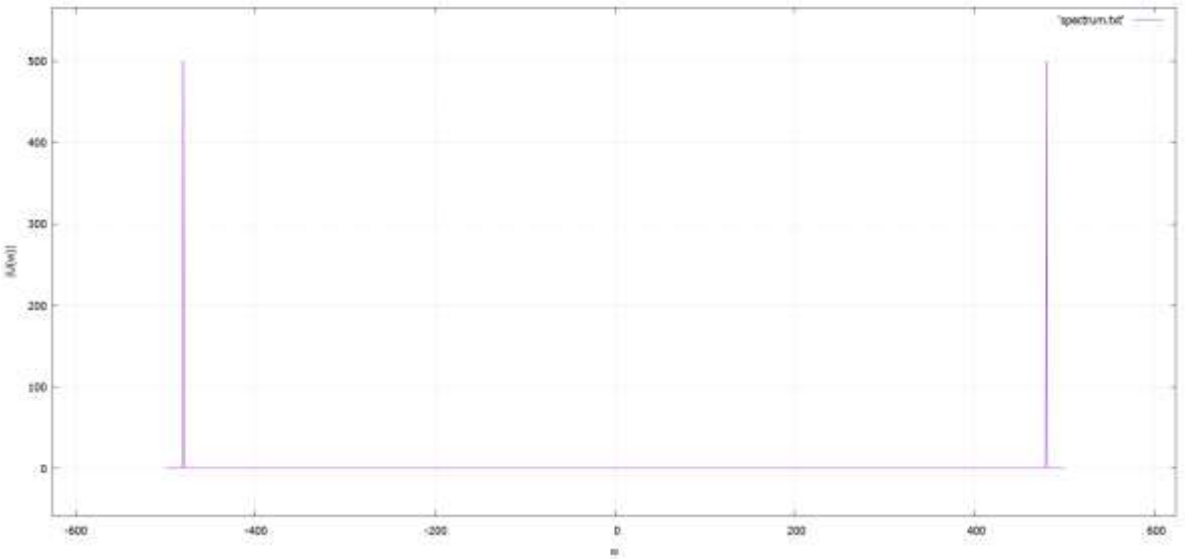

Рисунок 2 – Спектр исходного сигнала

В результате отработки обратного ДПФ, получили сигнал, совпадающий с исходным, что видно на рисунке 3.

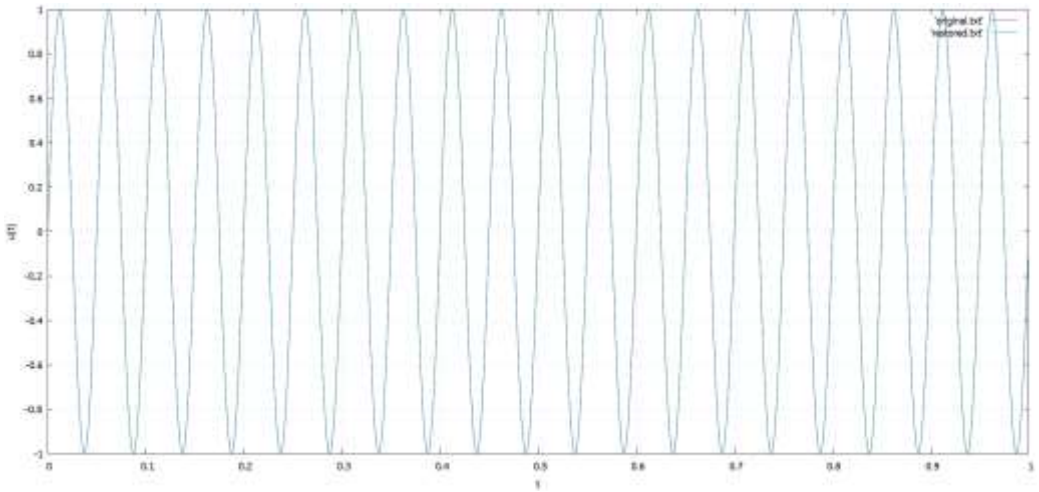

Рисунок 3 – Восстановление исходного сигнала с помощью обратного ДПФ

Вывод

В ходе выполнения задания, мы познакомились с понятием дискретного преобразования Фурье, его формулами, а также матричным представлением. Была написана программа, реализующая прямое и обратное ДПФ. Прямое ДПФ переводит сигнал в частотную область, т.е. представляет нам его спектр. Обратное ДПФ из частотной области возвращает сигнал во временную.

2. Продемонстрировать с помощью написанной программы свойства линейности, сдвига сигнала во времени и равенство Парсеваля.

2.1. Линейность:

$$\alpha_1 u_k^{(1)} + \alpha_2 u_k^{(2)} \leftrightarrow \alpha_1 \dot{U}_n^{(1)} + \alpha_2 \dot{U}_n^{(2)} \qquad (5)$$

Суть свойства линейности заключается в том, что ДПФ суммы двух сигналов равно сумме ДПФ каждого из этих сигналов по отдельности.

Для демонстрации свойства линейности возьмем две функции вида $u_1(t) = \sin(2\pi f_1 t)$ и $u_2(t) = \sin(2\pi f_2 t)$ с частотам $f1$ = 20 Гц и $f2$ = 10 Гц соответственно. Коэффициенты α_1 и α_2 зададим равными $\alpha_1 = 10$, $\alpha_2 = 15$.

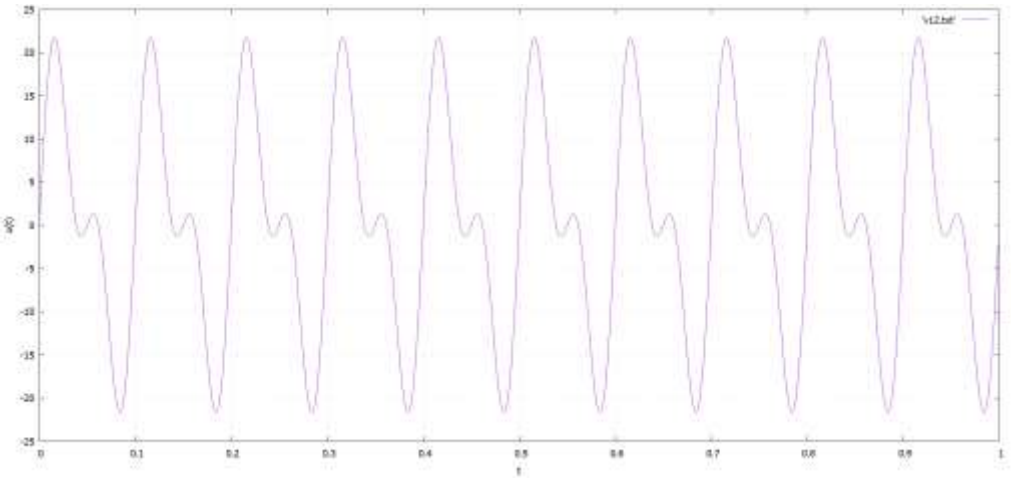

Рисунок 4 – Сумма сигналов $\alpha_1 u_1$ и $\alpha_2 u_2$

В результате работы прямого ДПФ получили спектр суммы сигналов, который соответствует спектру, который мы получили бы, сложив построенные отдельно спектры первого и второго сигналов.

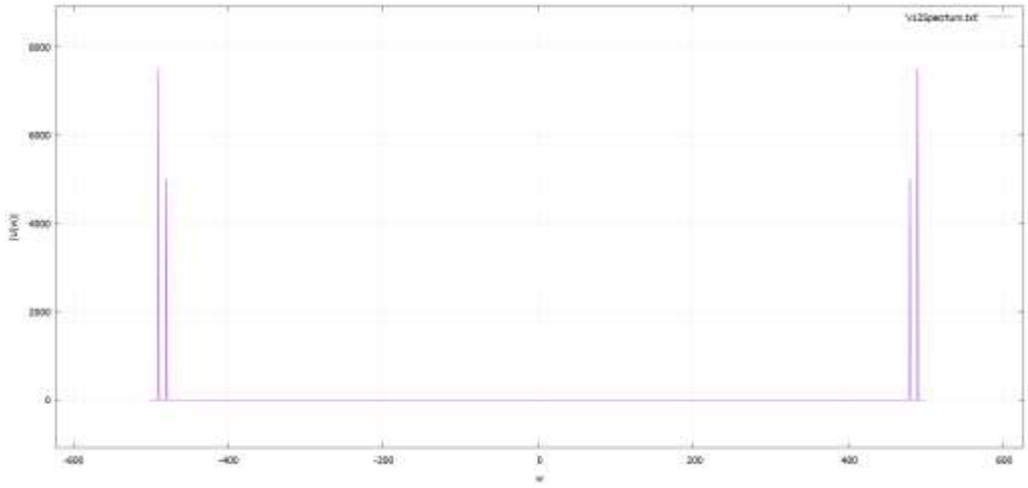

Рисунок 5 – Спектр суммы сигналов $\alpha_1 u_1$ и $\alpha_2 u_2$

В результате отработки обратного ДПФ, получили сигнал, совпадающий с исходным.

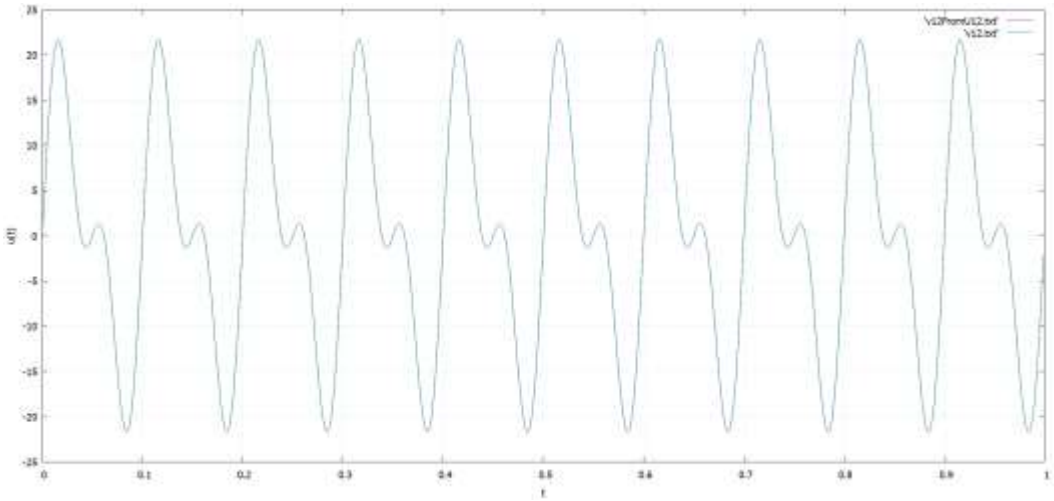

Рисунок 6 – Обратное ДПФ от спектра суммы сигналов

2.2. Задержка (сдвиг сигнала во времени):

$$u_{k-\tau} \leftrightarrow \dot{U}_n e^{-j\frac{2\pi\tau n}{N}} \tag{6}$$

Суть свойства сдвига во времени заключается в том, что сдвиг сигнала на τ отсчетов приводит к повороту фазового спектра, в то время как амплитудный спектр не меняется.

Построим сигнал, сдвинутый относительно исходного (рисунок 1) на τ = 10. И построим спектр этого сигнала. Как видно по рисунку 8, спектры исходного и сдвинутого сигналов совпадают.

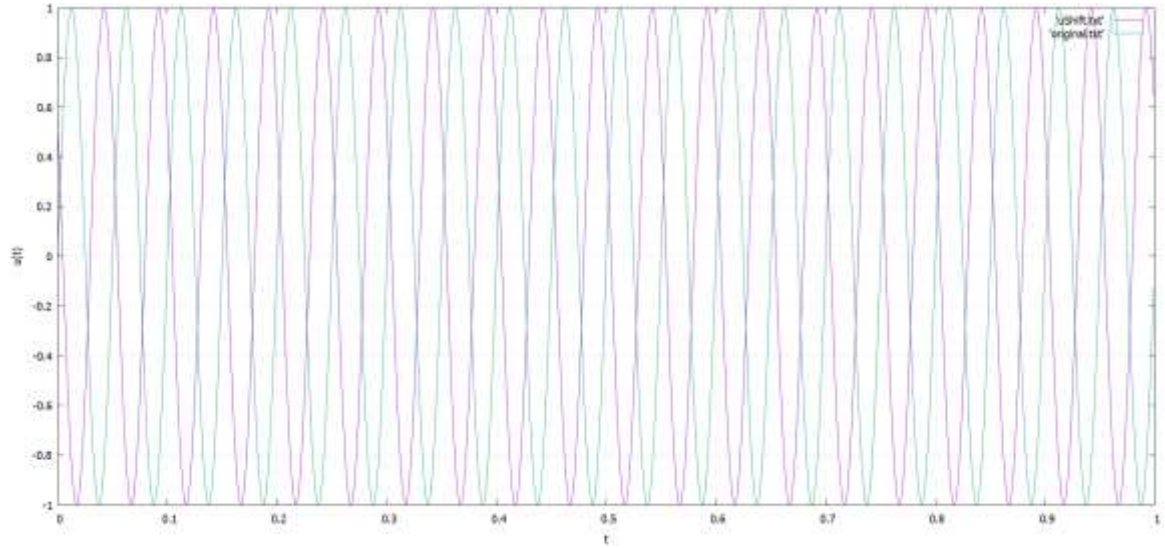

Рисунок 7 – Исходный сигнал и сигнал, сдвинутый на τ отсчетов

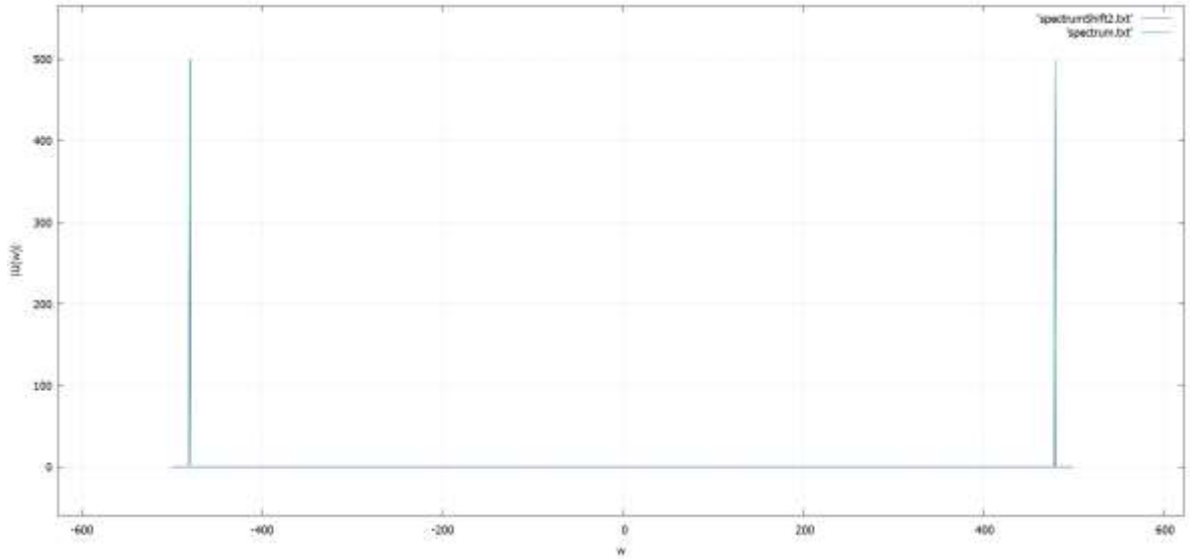

Рисунок 8 – Спектры исходного и сдвинутого сигналов

2.3. Равенство Парсеваля:

$$\sum_{k=0}^{N-1} u_k^2 = \frac{1}{N} \sum_{n=0}^{N-1} |\dot{U}_n|^2 \qquad (7)$$

Теорема Парсеваля устанавливает равенство между энергией сигнала и энергией его спектра. По результатам работы программы получили, что энергии раны, а значит равенство Парсеваля выполняется.

$$\sum_{k=0}^{N-1} u_k^2 = 0,5$$

$$\frac{1}{N} \sum_{k=0}^{N-1} |\dot{U}_n|^2 = 0,5$$

Вывод

В ходе выполнения задания, мы познакомились со свойствами дискретного преобразования Фурье, а также программно их реализовали и убедились в корректности работы.

3. Произвести декодирование аудио-файла с записью тонального сигнала (Dual-Tone Multi-Frequency (DTMF)) сигнала в формате WAV PCM 16 bit, mono.

Данный способ кодирования предполагает, что кодируемое значение представляется в виде пары различных частот (f1/f2) в соответствии с таблицей, приведенной на рисунке 9. Затем, сигнал представляется в виде отсчетов суммы двух синусоид соответствующих частот. Для декодирования сигнала необходимо произвести прямое дискретное преобразование для имеющегося набора отсчетов и определить частоты используемых для кодирования синусоид по полученному амплитудному спектру. Поиск происходит по нахождению максимума в верхней и нижней группах частот.

(f_1/f_2)	1209 Гц	1336 Гц	1477 Гц	1633 Гц
697 Гц	1	2	3	A
770 Гц	4	5	6	B
852 Гц	7	8	9	C
941 Гц	*	0	#	D

Таблица 1 – Таблица частот для кодирования DTMF сигналов

Считаем WAV-файл и выведем его содержимое. Тогда увидим, что исходный сигнал имеет вид:

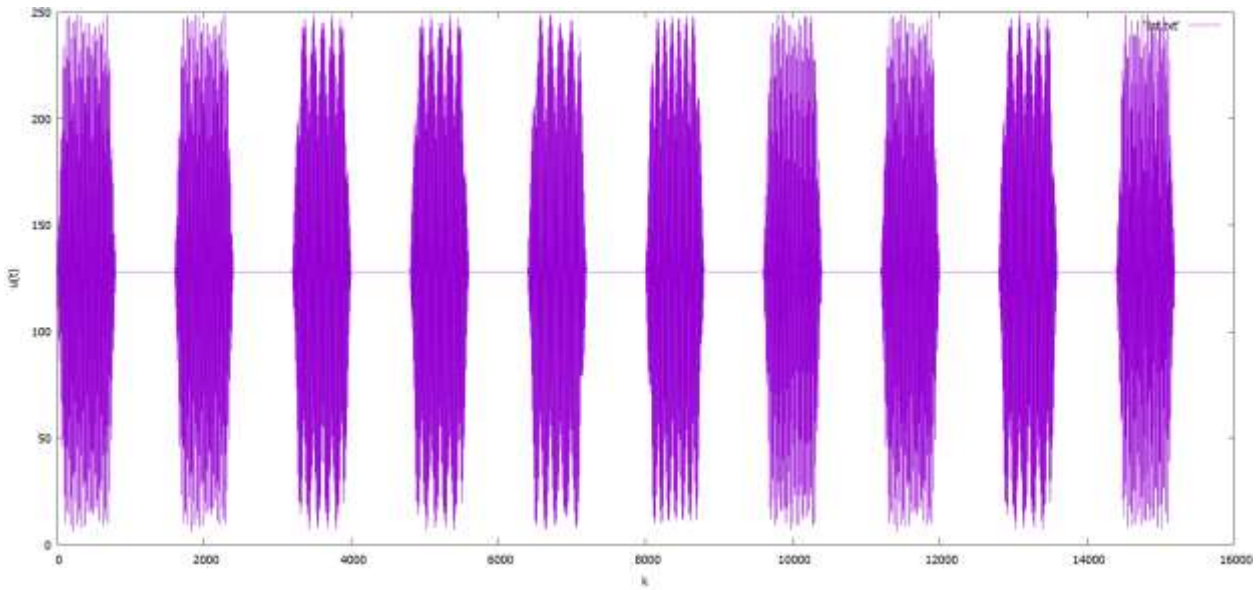

Рисунок 9 – Содержимое WAV-файла

Заметим, что звук и тишина в звуковом файле чередуются и длятся каждый по 800 семплов. Будем считывать по эти 800 семплов и переводить в частотную область с помощью

прямого ДПФ. Из амплитудного спектра будем получать по две частоты, задающие звук, и декодировать с помощью таблицы 1.

В результате работы программы получили список из 10 пар частот:

f1 = 770, f2 = 1340
f1 = 770, f2 = 1340
f1 = 790, f2 = 1480
f1 = 790, f2 = 1480
f1 = 700, f2 = 1340
f1 = 700, f2 = 1480
f1 = 770, f2 = 1210
f1 = 770, f2 = 1340
f1 = 790, f2 = 1480
f1 = 870, f2 = 1210

И расшифровали закодированную последовательность нажатий:

Frequency decoding: 5 5 6 6 2 3 4 5 6 7

Вывод

В ходе выполнения задания, мы научились применять свои знания о дискретном преобразовании Фурье: с помощью прямого ДПФ мы по частям получали амплитудный спектр WAV-файла и декодировали его.

4. Выполнить оценку смещения между двумя изображениями путем анализа фазового спектра

Нужно выполнить оценку смещения между двумя изображениями в формате BMP24 путем анализа фазового спектра. Для оценки смещения будет применяется метод фазовой корреляции. Метод фазовой корреляции как раз и применяют для поиска соответствия между смещенными, повернутыми и разномасштабными изображениями.

Изображение представляет собой заголовок и массив пикселей, каждый из которых задается тремя цветами (R, G, B). Все преобразования мы будем проводить над одним из трех цветов, например, над красным. Из обоих изображений, извлекаем наборы красного цвета и формируем из них две матрицы, соответствующие размерам изображений. Далее применяем для них прямое двумерное ДПФ по формуле (2).

Теперь будем вычислять взаимный фазовый спектр по формуле:

$$R = \frac{U_1 \circ U_2^*}{|U_1 \circ U_2^*|} \quad (8)$$

где ◦ – поэлементное умножение. Над результатом выполним обратное ДПФ и получим POC - функцию – функцию фазовой корреляции. Если $g(x, y) = f(x-a, y-b)$, т.е. одна функция сдвинута относительно другой, то POC-функция является дельта-функцией с пиком в точке с координатами (a, b).

Будем находить смещение между изображениями:

Рисунок 10 – truck_second

Рисунок 11 – truck_first

В результате работы программы получим координаты пикселя-смещения одного изображения относительно второго: x = 66, y = 20

Сдвинем второе изображение относительно первого на эти координаты и наложим изображения для оценки корректности работы программы.

Рисунок 12 – Совмещенные изображения

Вывод

В ходе выполнения задания, мы научились анализировать фазовые спектры, узнали о методе фазовой корреляции и смогли его применить. С помощью POC-функции мы смогли найти смещение между двумя изображении и убедились в корректности работы программы. Метод фазовой корреляции активно применяется в задачах совмещения, при поиске смещения, для оценки движения объектов на изображениях, для поиска соответствий, построения составного изображения, а также для решения биометрических задач.

Заключение:

В ходе выполнения данной лабораторной работы мы изучили методы Фурье-анализа дискретных и цифровых сигналов. Мы научились программно реализовывать выполнение прямого и обратного ДПФ, а также изучили его свойства. Помимо этого, сразу применили полученные знания на практике при декодировании аудиофайла и поиске смещения между изображениями. Изучая смещение, мы также познакомились с понятием фазовой корреляции, которое активно используется при работе с изображениями.

Таким образом, при выполнении данной лабораторной работы мы научились переводить сигналы из временной области в частотную, а также анализировать их амплитудные и фазовые спектры.

Приложение 1.

Листинг программы:

Задания 2.1 и 2.2

dsp2.h

```
#pragma once
#include <iostream>
#include <vector>
#include <complex>
#include<cmath>
using namespace std;


vector<vector<complex<double>>> Exponential_Matrix_Creation(int N);

vector<vector<complex<double>>> Transpose(vector<vector<complex<double>>>& matrix);

void Save_To_File(vector<double> time, vector<complex<double>> values, const string
filename, int num);

vector<vector<complex<double>>>
Hermitian_Transpose_Matrix_Creation(vector<vector<complex<double>>>& matrix);

vector<complex<double>> Calculate_Function_Values(double t1, double t2, double
freaquancy, double dt);

vector<complex<double>> Calculate_Function_Values_with_shift(double t1, double t2, double
freaquancy, double dt, double shift);

double Function(double time, double frequency);

vector<complex<double>> Multiplication(vector<complex<double>> v,
vector<vector<complex<double>>> matrix);

void Parseval_Equality(vector<complex<double>> u1, vector<complex<double>> u2);

vector<complex<double>> Linearity(vector<double> time_values, vector<complex<double>> v,
vector<complex<double>> v2, vector<complex<double>> u1, vector<complex<double>> u2);
```

dsp2.cpp

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <vector>
#include <complex>
#include<cmath>
#include <limits.h>
#include <fstream>
#include "dsp2lab.h"
using namespace std;

vector<vector<complex<double>>> Exponential_Matrix_Creation(int N) {
	vector<vector<complex<double>>> res(N, vector<complex<double>>(N));
	complex<double> I(0, 1);

	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++) {
			complex<double> element(exp(I * complex<double>(2 * M_PI * i * j / N,
0)));
			res[i][j] = element;
		}
```

```
	}
	return res;
}

vector<vector<complex<double>>> Transpose(vector<vector<complex<double>>>& matrix) {
	complex<double> tmp;
    for (int i = 0; i < matrix.size(); i++) {
	    for (int j = i; j < matrix[i].size(); j++) {
			tmp = matrix[i][j];
		matrix[i][j] = matrix[j][i];
		matrix[j][i] = tmp;
	    }
    }
	return matrix;
}

void Save_To_File(vector<double> time, vector<complex<double>> values, const string
filename, int num) {
	ofstream fout(filename);

	switch (num) {
	case 1:
		if (fout.is_open()) {
			for (int i = 0; i < values.size(); i++)
				fout << time[i] << " " << values[i].real() << endl;
		}
		else
			cout << "Can`t open the file" << endl;
		break;
	case 2:
		if (fout.is_open()) {
			int N = values.size();
			int z = 0;
			for (int i = -N / 2; i < N / 2; i++) {
				fout << i << "  " << abs(values[z].imag()) << endl;
				z++;
			}
		}
		else
			cout << "Can`t open the file" << endl;
		break;
	}

	fout.close();
}

vector<vector<complex<double>>>
Hermitian_Transpose_Matrix_Creation(vector<vector<complex<double>>> &matrix) {
	vector<vector<complex<double>>> res = Transpose(matrix);
	for (int i = 0; i < matrix.size(); i++)
		for (int j = 0; j < matrix[i].size(); j++)
			res[i][j] = conj(res[i][j]);
	return res;
}

vector<complex<double>> Calculate_Function_Values(double t1, double t2, double
freaquancy, double dt) {
	vector<complex<double>> result;

	for (double i = t1; i < t2; i += dt) {
		complex<double> value(Function(i, freaquancy), 0);
		result.push_back(value);
	}
	return result;
}
```

```
vector<complex<double>> Calculate_Function_Values_with_shift(double t1, double t2, double
freaquancy, double dt, double shift) {
	vector<complex<double>> result;

	for (double i = t1; i < t2; i += dt) {
		double time = i - shift;
		complex<double> value(Function(time, freaquancy), 0);
		result.push_back(value);
	}
	return result;
}

double Function(double time, double frequency) {
	return sin(2 * M_PI * frequency * time);
}

vector<complex<double>> Multiplication(vector<complex<double>> v,
vector<vector<complex<double>>> matrix) {
	int i = 0;
	vector<complex<double>> res(matrix[i].size());
	for (i = 0; i < matrix.size(); i++) {
		complex<double> tmp(0, 0);
		for (int j = 0; j < v.size(); j++) {
			tmp += v[j] * matrix[j][i];
		}
		res[i] = tmp;
	}
	return res;
}

void Parseval_Equality(vector<complex<double>> u1, vector<complex<double>> u2) {
	//complex<double> u1Sum(0, 0); complex<double> u2Sum(0, 0);
	double u1Sum = 0, u2Sum = 0;

	for (int i = 0; i < u1.size(); i++) {
		u1Sum += u1[i].real() * u1[i].real();
	}

	for (int j = 0; j < u2.size(); j++) {
		u2Sum += abs(u2[j].imag()) * abs(u2[j].imag());
	}

	double N = u1.size();
	u2Sum *= 1 / N;

	cout << 1 / N * u1Sum << " ? " << 1 / N * u2Sum << endl;
}

vector<complex<double>> Linearity(vector<double> time_values, vector<complex<double>> v,
vector<complex<double>> v2, vector<complex<double>> u1, vector<complex<double>> u2) {
	int a1 = 10, a2 = 15;
	for (int i = 0; i < v.size(); i++) {
		v[i] *= a1;
		v2[i] *= a2;
	}
	for (int j = 0; j < v2.size(); j++) {
		u1[j] *= a1;
		u2[j] *= a2;
	}

	vector<complex<double>> v12(v.size());
	for (int l = 0; l < v.size(); l++)
		v12[l] = v[l] + v2[l];
```

```
	vector<complex<double>> U(u1.size());
	for (int t = 0; t < u1.size(); t++)
		U[t] = u1[t] + u2[t];

	Save_To_File(time_values, v12, "v12.txt", 1);
	return U;
}

void timeShist(vector<complex<double>> v, vector<complex<double>> u, double shift) {




}
```

main.cpp

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <vector>
#include <complex>
#include<cmath>
#include <fstream>

#include "dsp2lab.h"
using namespace std;

/*Написать программу вычисления прямого и обратного дискретного преобразования Фурье в
матричной форме*/

int main() {
	/*double f1 = 10;
	double f2 = 20;
	double T = 10;
	double Fd = 1000;
	double dt = T / Fd;*/

	double f1 = 20;
	double f2 = 10;
	double T = 1;
	double dt = T / 1000;
	complex<double> I(0, 1);


	vector<complex<double>> v = Calculate_Function_Values(0, T, f1, dt);

	vector<vector<complex<double>>> expMatr = Exponential_Matrix_Creation(v.size());
	vector<vector<complex<double>>> hermMatr =
Hermitian_Transpose_Matrix_Creation(expMatr);

	vector<complex<double>> u1 = Multiplication(v, expMatr); //ппф
	vector<complex<double>> u2 = Multiplication(u1, hermMatr); //опф

	for (int i = 0; i < u2.size(); i++)
		u2[i] = u2[i] / complex<double>(u2.size(), 0);

	vector<double> time_values;
	for (double i = 0; i < T; i += dt)
		time_values.push_back(i);

	Save_To_File(time_values, u2, "restored.txt", 1);
	Save_To_File(time_values, v, "original.txt", 1);
	Save_To_File(time_values, u1, "spectrum.txt", 2);

	/*Продемонстрировать с помощью написанной программы свойства линейности,
	сдвига сигнала во времени и равенство Парсеваля.*/
```

```
	cout << "Parseval's equality is fulfilled?" << endl;
	Parseval_Equality(u2, u1); /*Мощность исходного сигнала и мощность спектра сигнала одинаковы*/

	//линейность
	vector<complex<double>> v2 = Calculate_Function_Values(0, T, f2, dt);
	vector<complex<double>> u3 = Multiplication(v2, expMatr);

	vector<complex<double>> u12 = Linearity(time_values, v, v2, u1, u3);
	Save_To_File(time_values, u12, "v12Specrtum.txt", 2);

	vector<complex<double>> v3 = Multiplication(u12, hermMatr);
	for (int i = 0; i < v3.size(); i++)
		v3[i] = v3[i] / complex<double>(v3.size(), 0);

	Save_To_File(time_values, v3, "v12FromU12.txt", 1);


	//сдвиг во времени
	double tau = 10;
	//vector<complex<double>> uShift = Calculate_Function_Values_with_shift(0, T, f1, dt, tau);
	vector<complex<double>> uShift;
	for (double i = 0; i < T; i += dt) {
		uShift.push_back(sin(2 * M_PI * f1 * i - tau));
	}
	Save_To_File(time_values, uShift, "uShift.txt", 1);

	vector<complex<double>> uShiftDirect = Multiplication(v, expMatr);
	//Save_To_File(time_values, uShiftDirect, "spectrumShift.txt", 2);

	double N = uShift.size();

	for (int i = 0; i < N; i++) {
		//complex<double> tmp(0, -2 * M_PI * tau * i / N);
		complex<double> num(exp(-I * complex<double>(2 * M_PI * f1 * tau, 0)));
		//uShiftDirect[i] *= exp(tmp);
		uShiftDirect[i] *= num;
	}

	Save_To_File(time_values, uShiftDirect, "spectrumShift2.txt", 2);

	/*vector<complex<double>> uShiftInverse = Multiplication(uShiftDirect, hermMatr);
	for (int t = 0; t < uShiftInverse.size(); t++)
		uShiftInverse[t] = uShiftInverse[t] / N;

	Save_To_File(time_values, uShiftInverse, "uShiftInverse.txt", 1);*/

}
```

Задание 2.3

Dsp2naum3.h

```
#pragma once
#include <iostream>
#include <fstream>
#include <vector>
#include <complex>
#include <cmath>
using namespace std;
#define N 800

typedef struct WAVHEADER {
    char chunkId[4];
```

```
    unsigned long chunkSize;
    char format[4];
    char subchunk1Id[4];
    unsigned long subchunk1Size;
    unsigned short audioFormat;
    unsigned short numChannels;
    unsigned long sampleRate;
    unsigned long byteRate;
    unsigned short blockAlign;
    unsigned short bitsPerSample;
};

struct chunk_t
{
    char ID[4];
    unsigned long size;
};

void Wav_Reader(const char* fileName, const char* fileToSave);

int Find_Max(vector<complex<double>> u, int start, int finish);

vector<vector<complex<double>>> Exponential_Matrix_Creation(int num);

vector<vector<complex<double>>> Transpose(vector<vector<complex<double>>>& matrix);

vector<vector<complex<double>>>
Hermitian_Matrix_Creation(vector<vector<complex<double>>>& matrix);

vector<complex<double>> Multiplication(vector<complex<double>> v,
vector<vector<complex<double>>> matrix);

void Frequency_Decoding(vector<double> f1, vector<double> f2);
```

dsp2num3.cpp

```
#define _USE_MATH_DEFINES
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <vector>
#include <complex>
#include <limits.h>
#include <cmath>
#include "dsp2.h"

using namespace std;

void Wav_Reader(const char* fileName, const char* fileToSave) {
    WAVHEADER header;
    chunk_t chunk;
    FILE* fin = fopen(fileName, "rb");

    fread(&header, sizeof(header), 1, fin);

    cout << "File Size:  " << header.chunkSize << endl;
    cout << "Number of Channels: " << header.numChannels << endl;
    cout << "Sample Rate: " << header.sampleRate << endl;
    cout << "Bits per Sample: " << header.bitsPerSample << endl;


    while (true) {
        fread(&chunk, sizeof(chunk), 1, fin);
        if (*(unsigned int*)&chunk.ID == 0x61746164)
            break;
        fseek(fin, sizeof(chunk), SEEK_CUR);
```

```
}

int sample_size = header.bitsPerSample / 8;
int samples_count = chunk.size * 8 / header.bitsPerSample;
cout << "Samples count =  " << samples_count << endl << endl;

short int* value = new short int[samples_count];
memset(value, 0, sizeof(short int) * samples_count);

for (int i = 0; i < samples_count; i++) {
    fread(&value[i], sample_size, 1, fin);
}

ofstream fout(fileToSave);

for (int i = 0; i < samples_count; i++)
    fout << i << "  " << value[i] << "\n";

fclose(fin);
fout.close();


complex<double> I(0, 1);
vector<complex<double>> v;
for (int i = 0; i < samples_count; i++) {
    complex<double> num((double)value[i], 0);
    v.push_back(num);
}

vector<double> f1;
vector<double> f2;

for (int t = 0; t < samples_count / N; t+=2) {
    vector<complex<double>> U;

    for (int i = 0; i < N; ++i) {
        U.push_back(v[i + t * N]);
    }

    vector<vector<complex<double>>> expMatr = Exponential_Matrix_Creation(N);
    vector<vector<complex<double>>> hermMatr = Hermitian_Matrix_Creation(expMatr);

    vector<complex<double>> U1 = Multiplication(U, hermMatr);

    ofstream spec;
    spec.open("spec.txt");
    int z = 0;
    for (int i = -N/2; i < N/2; i++) {
        spec << i << "  " << abs(U1[z].imag()) << endl;
        z++;
    }
    spec.close();

    double F1 = Find_Max(U1, 0, N / 8);
    double F2 = Find_Max(U1, N / 8, N / 4);

    if ((F1 != 0) || (F2 != 0)) {

        cout << "f1 = " << F1 * 10 << ", " << "f2 = " << F2 * 10 << endl;
        f1.push_back(F1 * 10);
        f2.push_back(F2 * 10);
    }
}

/*ofstream spec;
```

```
    spec.open("spec.txt");
    int z = 0;
    //for (int i = -(samples_count / 2); i < samples_count / 2; i++) {
    for (int i = 0; i < sumSpec.size(); i++) {
        spec << i << "  " << abs(sumSpec[i].imag()) << endl;

    }
    spec.close(); */

    cout << "\nFrequency decoding: ";
    Frequency_Decoding(f1, f2);
    fclose(fin);
}

void Frequency_Decoding(vector<double> f1, vector<double> f2) {
    double first[4] = { 697, 770, 852, 941 };
    double second[4] = { 1209, 1336, 1477, 1633 };

    char symbols[4][4] = { {'1', '2', '3', 'A'}, {'4', '5', '6', 'B'}, {'7', '8', '9',
'C'}, {'*', '0', '#', 'D'} };

    for (int t = 0; t < f1.size(); t++) {
        double F1 = f1[t];

        for (int i = 0; i < 4; i++) {

            if (abs(F1 - first[i]) <= 40) {

                double F2 = f2[t];

                for (int j = 0; j < 4; j++) {
                    if (abs(F2 - second[j]) <= 40) {
                        cout << symbols[i][j] << " ";
                    }
                }
            }
        }
    }
}

int Find_Max(vector<complex<double>> u, int start, int finish) {
    double A = 0;
    int result = 0;

    for (int i = start + 1; i < finish; ++i) {
        if (u[i].imag() >= A) {
            A = u[i].imag();
            result = i + 1;
        }
    }

    return result;
}

vector<vector<complex<double>>> Exponential_Matrix_Creation(int num) {
    vector<vector<complex<double>>> res(num, vector<complex<double>>(num));
    complex<double> I(0, 1);

    for (int i = 0; i < num; i++) {
        for (int j = 0; j < num; j++) {
            complex<double> element(exp(I * complex<double>(2 * M_PI * i * j / num, 0)));
            res[i][j] = element;
        }
    }
    return res;
```

```
}

vector<vector<complex<double>>> Transpose(vector<vector<complex<double>>>& matrix) {
    complex<double> tmp;
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = i; j < matrix[i].size(); j++) {
            tmp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = tmp;
        }
    }
    return matrix;
}

vector<vector<complex<double>>>
Hermitian_Matrix_Creation(vector<vector<complex<double>>>& matrix) {
   // vector<vector<complex<double>>> res = matrix;
    vector<vector<complex<double>>> res = Transpose(matrix);
    for (int i = 0; i < matrix.size(); i++)
        for (int j = 0; j < matrix[i].size(); j++)
            res[i][j] = conj(res[i][j]);
    return res;
}

vector<complex<double>> Multiplication(vector<complex<double>> v,
vector<vector<complex<double>>> matrix) {
    int i = 0;
    vector<complex<double>> res(matrix[i].size());
    for (i = 0; i < matrix.size(); i++) {
        complex<double> tmp(0, 0);
        for (int j = 0; j < v.size(); j++) {
            tmp += v[j] * matrix[j][i];
        }
        res[i] = tmp;
    }
    return res;
}
```

Main.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <vector>
#include <complex>
#include <cmath>
#include "dsp2.h"
using namespace std;

int main() {

    Wav_Reader("7.wav", "list.txt");
    return 0;
}
```

Задание 2.4

Dsp2num4.h

```
#pragma once
#include <iostream>
#include <vector>
#include <complex>
#include<cmath>
using namespace std;

typedef struct BFH {
```

```
	unsigned short bfType;
	unsigned int bfSize;
	unsigned short bfReserved1;
	unsigned  short bfReserved2;
	unsigned int bfOffBits;
} BITMAPFILEHEADER;

typedef struct BIH {
	unsigned int biSize;
	unsigned int biWidth;
	unsigned int biHeight;
	unsigned short biPlanes;
	unsigned short biBitCount;
	unsigned int biCompression;
	unsigned int biSizeImage;
	unsigned int biXPelsPerMeter;
	unsigned int biYPelsPerMeter;
	unsigned int biClrUsed;
	unsigned int biClrImportant;
} BITMAPINFOHEADER;

typedef struct RGB {
	unsigned char rgbBlue;
	unsigned char rgbGreen;
	unsigned char rgbRed;
}RGBQUAD;

RGBQUAD** Read_bmp(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih);

void Color_selection(RGBQUAD** rgb, double* buffer, int a, int b);

vector<vector<complex<double>>> Color_matrix(double* buffer, int a, int b);

vector<vector<complex<double>>> Exponential_Matrix_Creation(int N);

void ExpMatrix_HermConjMatrix_Creation(vector<vector<complex<double>>>& F,
vector<vector<complex<double>>>& hermCongF, int N);

vector<vector<complex<double>>>
Hermitian_Transpose_Matrix_Creation(vector<vector<complex<double>>>& matrix);

vector<vector<complex<double>>> Multiply_complex_complex(vector<vector<complex<double>>>
firstMatrix, vector<vector<complex<double>>> secondMatrix);

vector<vector<complex<double>>> Multiply_elem(vector<vector<complex<double>>>
firstMatrix, vector<vector<complex<double>>> secondMatrix);

vector<vector<complex<double>>> Hermitian_Matrix(vector<vector<complex<double>>>&
matrix);

vector<vector<complex<double>>> Div_elem(vector<vector<complex<double>>> matrix);

vector<vector<complex<double>>>
Multiply_number_to_complex_matrix(vector<vector<complex<double>>> matrix, double value);

vector<int> Find_max_index(vector<vector<complex<double>>> matrix);

void Print_tank(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih, RGBQUAD** rgb1,
int a1, int b1, RGBQUAD** rgb2, int a2, int b2, int x, int y);

void Print(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih, RGBQUAD** rgb1, int a1,
int b1, RGBQUAD** rgb2, int a2, int b2, int x, int y);
```

dsp2num4.cpp

```
#define _USE_MATH_DEFINES
```

```
#include <iostream>
#include <vector>
#include <complex>
#include<cmath>
#include "dsp2image.h"
using namespace std;

RGBQUAD** Read_bmp(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih) {
	int k = 0;
	k = fread(bfh, sizeof(*bfh) - 2, 1, f);
	if (k == 0) {
		cout << "Reading file error";
		return 0;
	}
	k = fread(bih, sizeof(*bih), 1, f);
	if (k == 0) {
		cout << "Reading file error";
		return 0;
	}
	int a = bih->biHeight;
	int b = bih->biWidth;
	RGBQUAD** rgb = new RGBQUAD * [a];
	for (int i = 0; i < a; i++) {
		rgb[i] = new RGBQUAD[b];
	}
	int pad = 4 - (b * 3) % 4;
	for (int i = 0; i < a; i++) {
		fread(rgb[i], sizeof(RGBQUAD), b, f);
		if (pad != 4) {
			fseek(f, pad, SEEK_CUR);
		}
	}
	return rgb;
}

void Color_selection(RGBQUAD** rgb, double* buffer, int a, int b) {
	const int buff_size = a * b * sizeof(unsigned char);

	size_t position = 0;
	buffer[position] = 0;

	for (int i = 0; i < b; i++) {
		for (int j = 0; j < a; j++) {

			buffer[position] = rgb[i][j].rgbRed;
			position ++;
		}
	}
}

vector<vector<complex<double>>> Color_matrix(double* buffer, int a, int b) {
	vector<vector<complex<double>>> res(b, vector<complex<double>>(a));
	size_t position = 0;

	for (int i = 0; i < b; i++) {
		for (int j = 0; j < a; j++) {

			res[i][j] = buffer[position];
			position ++;
		}
	}
	return res;
}

vector<vector<complex<double>>> Exponential_Matrix_Creation(int N) {
```

```
	vector<vector<complex<double>>> res(N);
	complex<double> I(0, 1);

	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++) {
			complex<double> element(exp(I * complex<double>(2 * M_PI * i * j / N,
0)));
			res[i].push_back(element);
		}
	}
	return res;
}

void ExpMatrix_HermConjMatrix_Creation(vector<vector<complex<double>>> &F,
vector<vector<complex<double>>> &hermCongF, int N) {
	complex<double> I(0, 1);

	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++) {
			complex<double> element(exp(I * complex<double>(2 * M_PI * i * j / N,
0)));
			F[i].push_back(element);
			hermCongF[j].push_back(conj(element));
		}
	}
}

vector<vector<complex<double>>>
Hermitian_Transpose_Matrix_Creation(vector<vector<complex<double>>>& matrix) {
	vector<vector<complex<double>>> res(matrix[0].size());
	for (int i = 0; i < matrix.size(); i++) {
		for (int j = 0; j < matrix[0].size(); j++) {
			res[j].push_back(conj(matrix[i][j]));
		}
	}
	return res;
}

vector<vector<complex<double>>> Hermitian_Matrix(vector<vector<complex<double>>>& matrix)
{
	vector<vector<complex<double>>> res(matrix[0].size());

	for (int i = 0; i < matrix.size(); ++i) {
		for (int j = 0; j < matrix[0].size(); ++j) {
			res[j].push_back(matrix[i][j]);
		}
	}
	return res;
}

vector<vector<complex<double>>> Multiply_complex_complex(vector<vector<complex<double>>>
firstMatrix, vector<vector<complex<double>>> secondMatrix) {
	int i = 0;
	vector<vector<complex<double>>> res(firstMatrix.size());

	for (i = 0; i < firstMatrix.size(); i++) {
		for (int j = 0; j < secondMatrix[0].size(); j++) {
			complex<double> tmp(0, 0);
			for (int k = 0; k < firstMatrix[0].size(); k++) {
				complex<double> firstElem = secondMatrix[k][j];
				complex<double> secondElem = firstMatrix[i][k];
				tmp += firstElem * secondElem;
			}
			res[i].push_back(tmp);
		}
```

```
	}
	return res;
}

vector<vector<complex<double>>> Multiply_elem(vector<vector<complex<double>>>
firstMatrix, vector<vector<complex<double>>> secondMatrix) {
	vector<vector<complex<double>>> result(firstMatrix.size());

	for (int i = 0; i < firstMatrix.size(); ++i) {
		for (int j = 0; j < firstMatrix[0].size(); ++j) {
			result[i].push_back(firstMatrix[i][j] * secondMatrix[i][j]);
		}
	}
	return result;
}

vector<vector<complex<double>>> Div_elem(vector<vector<complex<double>>> matrix) {
	vector<vector<complex<double>>> res(matrix.size());

	for (int i = 0; i < matrix.size(); ++i) {
		for (int j = 0; j < matrix[0].size(); ++j) {
			double tmp = abs(matrix[i][j]);
			res[i].push_back(matrix[i][j] / tmp);
		}
	}
	return res;
}

vector<vector<complex<double>>>
Multiply_number_to_complex_matrix(vector<vector<complex<double>>> matrix, double value) {
	vector<vector<complex<double>>> res(matrix.size());

	for (int i = 0; i < matrix.size(); ++i) {
		for (int j = 0; j < matrix[0].size(); ++j) {
			complex<double> tmp = matrix[i][j];
			res[i].push_back(tmp * value);
		}
	}
	return res;
}

vector<int> Find_max_index(vector<vector<complex<double>>> matrix) {
	vector<int> res;
	int x = 0, y = 0;
	double max = 0;
	for (int i = 0; i < matrix.size(); ++i) {
		for (int j = 0; j < matrix[0].size(); ++j) {
			if (abs(matrix[i][j]) > max) {
				max = abs(matrix[i][j]);
				x = i;
				y = j;
			}
		}
	}
	res.push_back(x);
	res.push_back(y);
	return res;
}

void Print_tank(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih, RGBQUAD** rgb1,
int a1, int b1, RGBQUAD** rgb2, int a2, int b2, int x, int y) {

	bih->biHeight = b1 + y;
	bih->biWidth = a1 + x;
	fwrite(bfh, sizeof(*bfh) - 2, 1, f);
```

```
	fwrite(bih, sizeof(*bih), 1, f);
	int pad = 4 - ((a1 / 2) * 3) % 4;
	char buf = 0;

	RGBQUAD** resRGB = new RGBQUAD * [b1 + y];
	for (int i = 0; i < (b1 + y); i++) {
		resRGB[i] = new RGBQUAD[a1 + x];
	}

	for (int i = 0; i < b2; i++) {
		for (int j = 0; j < a2; j++) {
			resRGB[i][j].rgbBlue = rgb2[i][j].rgbBlue;
			resRGB[i][j].rgbGreen = rgb2[i][j].rgbGreen;
			resRGB[i][j].rgbRed = rgb2[i][j].rgbRed;
		}
	}

	for (int i = 0; i < b1; i++) {
		for (int j = 0; j < a1; j++) {
			resRGB[i + y][j + x].rgbBlue = rgb1[i][j].rgbBlue;
			resRGB[i + y][j + x].rgbGreen = rgb1[i][j].rgbGreen;
			resRGB[i + y][j + x].rgbRed = rgb1[i][j].rgbRed;
		}
	}


	for (int i = 0; i < (b1 + y); i++) {
		fwrite((resRGB[i]), sizeof(RGBQUAD), a1 + x, f);
		if (pad != 4) {
			fwrite(&buf, 1, pad, f);
		}
	}

}

void Print(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih, RGBQUAD** rgb1, int a1,
int b1, RGBQUAD** rgb2, int a2, int b2, int x, int y) {
	bih->biHeight = b1 + y;
	bih->biWidth = a1 + x;
	fwrite(bfh, sizeof(*bfh) - 2, 1, f);
	fwrite(bih, sizeof(*bih), 1, f);
	int pad = 4 - ((a1 / 2) * 3) % 4;
	char buf = 0;

	RGBQUAD** resRGB = new RGBQUAD * [b1 + y];
	for (int i = 0; i < (b1 + y); i++) {
		resRGB[i] = new RGBQUAD[a1 + x];
	}

	for (int i = 0; i < b2; i++) {
		for (int j = 0; j < a2; j++) {
			resRGB[i + y][j + x].rgbBlue = rgb2[i][j].rgbBlue;
			resRGB[i + y][j + x].rgbGreen = rgb2[i][j].rgbGreen;
			resRGB[i + y][j + x].rgbRed = rgb2[i][j].rgbRed;
		}
	}

	for (int i = 0; i < b1; i++) {
		for (int j = 0; j < a1; j++) {
			resRGB[i][j].rgbBlue = rgb1[i][j].rgbBlue;
			resRGB[i][j].rgbGreen = rgb1[i][j].rgbGreen;
			resRGB[i][j].rgbRed = rgb1[i][j].rgbRed;
		}
	}
```

```
	for (int i = 0; i < (b1 + y); i++) {
		fwrite((resRGB[i]), sizeof(RGBQUAD), a1 + x, f);
		if (pad != 4) {
			fwrite(&buf, 1, pad, f);
		}
	}

}
```

Main.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <vector>
#include <complex>
#include<cmath>
#include "dsp2image.h"
using namespace std;

int main() {

	BITMAPFILEHEADER bfh1;
	BITMAPINFOHEADER bih1;
	BITMAPFILEHEADER bfh2;
	BITMAPINFOHEADER bih2;
	FILE* f1, * f2, * f3;

	f1 = fopen("truck_first.bmp", "rb");
	if (f1 == NULL)
		cout << "Can`t open the image";

	RGBQUAD** rgb1 = Read_bmp(f1, &bfh1, &bih1);
	int a1 = bih1.biWidth;
	int b1 = bih1.biHeight;
	const int buff_size = a1 * b1 * sizeof(unsigned char);
	double* buffer = new double[buff_size];

	Color_selection(rgb1, buffer, a1, b1);
	vector<vector<complex<double>>> image1 = Color_matrix(buffer, a1, b1);

	f2 = fopen("truck_second.bmp", "rb");
	if (f2 == NULL)
		cout << "Can`t open the image";

	RGBQUAD** rgb2 = Read_bmp(f2, &bfh2, &bih2);
	int a2 = bih2.biWidth;
	int b2 = bih2.biHeight;
	const int buff_size2 = a2 * b2 * sizeof(unsigned char);
	double* buffer2 = new double[buff_size2];

	Color_selection(rgb2, buffer2, a2, b2);
	vector<vector<complex<double>>> image2 = Color_matrix(buffer2, a2, b2);

	vector<vector<complex<double>>> expMatr1(a1); vector<vector<complex<double>>>
hermMatr1(a1);
	ExpMatrix_HermConjMatrix_Creation(expMatr1, hermMatr1, a1);
	vector<vector<complex<double>>> expMatr2(b1); vector<vector<complex<double>>>
hermMatr2(b1);
	ExpMatrix_HermConjMatrix_Creation(expMatr2, hermMatr2, b1);


	//прямое двумерное дискретное преобразование фурье
	vector<vector<complex<double>>> U1 = Multiply_complex_complex(image1, hermMatr1);
	vector<vector<complex<double>>> U2 = Multiply_complex_complex(image2, hermMatr1);
	U1 = Multiply_complex_complex(Hermitian_Transpose_Matrix_Creation(U1), hermMatr2);
	U2 = Multiply_complex_complex(Hermitian_Transpose_Matrix_Creation(U2), hermMatr2);
```

```
	//взаимный фазовый спектр
	vector<vector<complex<double>>> R = Multiply_elem(Hermitian_Matrix(U1),
Hermitian_Transpose_Matrix_Creation(U2));
	R = Div_elem(R);

	vector<vector<complex<double>>> r =
Multiply_complex_complex(Multiply_number_to_complex_matrix(R, 1 / (double) a1),
expMatr1);
	r = Hermitian_Matrix(r);
	r = Multiply_complex_complex(Multiply_number_to_complex_matrix(r, 1/ (double) b1),
expMatr2);

	vector<int> result = Find_max_index(Hermitian_Matrix(r));
	int x = a1 - result[1];
	int y = b1 - result[0];

	cout << "x = " << x << "    y = " << (b1 - y) << endl;

	f3 = fopen("result.bmp", "wb");
	if (f3 == NULL)
		cout << "Can`t open the image";

	//Print(f3, &bfh1, &bih1, rgb2, a2, b2, rgb1, a1, b1, x, y); //танк
	Print(f3, &bfh1, &bih1, rgb1, a1, b1, rgb2, a2, b2, x, b1 - y); //фургон

	fclose(f1); fclose(f2); fclose(f3);
	/*back_to_file(rgb1, res);

	f2 = fopen("out.bmp", "wb");
	write_bmp(f2, rgb1, &bfh1, &bih1, a1, b1, b1);*/

}
```