**Лабораторная работа №2**

В лабораторной работе 2 необходимо из квантованного сигнала, получить исходный, который был создан в лабораторной работе 1

Код был модернизирован, были добавлены 2 функции: прямое дискретное преобразование Фурье и Обратное дискретное преобразование Фурье Погрешность должна быть не более 5%.

## **Код программы**

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include <complex>

struct Harmonic {

double amplitude;

double frequency;

};

struct Sample {

double time;

double value;

};

void dft(const std::vector<std::complex<double>>& in, std::vector<std::complex<double>>& out) {

int N = in.size();

out.resize(N);

for (int k = 0; k < N; ++k) {

std::complex<double> sum(0.0, 0.0);

for (int n = 0; n < N; ++n) {

double angle = -2 \* M\_PI \* k \* n / N;

sum += in[n] \* std::exp(std::complex<double>(0, angle));

}

out[k] = sum;

}

}

void idft(const std::vector<std::complex<double>>& in, std::vector<std::complex<double>>& out) {

int N = in.size();

out.resize(N);

for (int n = 0; n < N; ++n) {

std::complex<double> sum(0.0, 0.0);

for (int k = 0; k < N; ++k) {

double angle = 2 \* M\_PI \* k \* n / N;

sum += in[k] \* std::exp(std::complex<double>(0, angle));

}

out[n] = sum / static\_cast<double>(N);

}

}

void applyHannWindow(std::vector<std::complex<double>>& signal) {

int N = signal.size();

for (int n = 0; n < N; ++n) {

double window = 0.5 \* (1 - cos(2 \* M\_PI \* n / (N - 1)));

signal[n] \*= window;

}

}

int main() {

std::string filename = "parameters.txt";

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "File opening error " << filename << std::endl;

return 1;

}

int numHarmonics;

double baseFrequency;

double samplingFrequency;

double initialPhase;

file >> numHarmonics;

file >> baseFrequency;

file >> samplingFrequency;

file >> initialPhase;

std::vector<Harmonic> harmonics(numHarmonics);

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

file >> harmonics[i].amplitude;

harmonics[i].frequency = baseFrequency \* (i + 1);

}

file.close();

int numSamples = static\_cast<int>(samplingFrequency / baseFrequency) \* 10;

double period = 1.0 / baseFrequency;

double dt = period / numSamples;

std::vector<double> signal(numSamples);

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

double t = i \* dt;

double signalValue = 0.0;

for (const auto& harmonic : harmonics) {

signalValue += harmonic.amplitude \* std::sin(2 \* M\_PI \* harmonic.frequency \* t + initialPhase);

}

signal[i] = signalValue;

}

std::ofstream signalOutFile("signal.txt");

if (!signalOutFile.is\_open()) {

std::cerr << "Error opening the file for recording the signal." << std::endl;

return 1;

}

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

double t = i \* dt;

signalOutFile << t << " " << signal[i] << std::endl;

}

signalOutFile.close();

std::cout << "The signal is recorded in a file signal.txt" << std::endl;

std::string inputFilename = "signal.txt";

std::string outputFilename = "quantized\_signal.txt";

std::ifstream inputFile(inputFilename);

if (!inputFile.is\_open()) {

std::cerr << "File opening error " << inputFilename << std::endl;

return 1;

}

std::vector<Sample> signalSamples;

double time, value;

while (inputFile >> time >> value) {

signalSamples.push\_back({ time, value });

}

inputFile.close();

if (signalSamples.empty()) {

std::cerr << "file " << inputFilename << " empty." << std::endl;

return 1;

}

int numLevels = 16;

double minSignal = signalSamples[0].value;

double maxSignal = signalSamples[0].value;

for (const auto& sample : signalSamples) {

if (sample.value < minSignal) {

minSignal = sample.value;

}

if (sample.value > maxSignal) {

maxSignal = sample.value;

}

}

std::ofstream outputFile(outputFilename);

if (!outputFile.is\_open()) {

std::cerr << "File opening error " << outputFilename << std::endl;

return 1;

}

std::vector<Sample> quantizedSignal;

for (const auto& sample : signalSamples) {

double normalizedValue = (sample.value - (maxSignal + minSignal) / 2.0) / ((maxSignal - minSignal) / 2.0);

double quantizedNormalized = std::round(normalizedValue \* (numLevels - 1)) / (numLevels - 1);

double quantizedValue = quantizedNormalized \* ((maxSignal - minSignal) / 2.0) + (maxSignal + minSignal) / 2.0;

outputFile << sample.time << " " << quantizedValue << std::endl;

quantizedSignal.push\_back({ sample.time, quantizedValue });

}

outputFile.close();

std::cout << "The quantized signal is recorded in a file " << outputFilename << std::endl;

std::string inputFilename1 = "signal.txt";

std::string inputFilename2 = "quantized\_signal.txt";

std::string outputFilename3 = "overlapped\_signal.txt";

std::vector<Sample> originalSignal;

std::ifstream inputFile1(inputFilename1);

if (!inputFile1.is\_open()) {

std::cerr << "File opening error " << inputFilename1 << std::endl;

return 1;

}

std::ifstream inputFile2(inputFilename2);

if (!inputFile2.is\_open()) {

std::cerr << "File opening error " << inputFilename2 << std::endl;

return 1;

}

signalSamples.clear();

while (inputFile1 >> time >> value) {

originalSignal.push\_back({ time, value });

}

quantizedSignal.clear();

while (inputFile2 >> time >> value) {

quantizedSignal.push\_back({ time, value });

}

inputFile1.close();

inputFile2.close();

if (originalSignal.size() != quantizedSignal.size()) {

std::cerr << "The signals have different lengths." << std::endl;

return 1;

}

std::ofstream outputFile3(outputFilename3);

if (!outputFile3.is\_open()) {

std::cerr << "File opening error " << outputFilename3 << std::endl;

return 1;

}

for (size\_t i = 0; i < originalSignal.size(); ++i) {

outputFile3 << originalSignal[i].time << " " << originalSignal[i].value << " " << quantizedSignal[i].value << std::endl;

}

outputFile3.close();

std::cout << "Both signals are recorded in a file " << outputFilename3 << std::endl;

double mse = 0.0;

for (size\_t i = 0; i < originalSignal.size(); ++i) {

double error = originalSignal[i].value - quantizedSignal[i].value;

mse += error \* error;

}

mse /= originalSignal.size();

std::cout << "Average square error (SQR): " << mse << std::endl;

double signalPower = 0.0;

for (const auto& sample : originalSignal) {

signalPower += sample.value \* sample.value;

}

signalPower /= originalSignal.size();

double noisePower = mse;

double snr = 10 \* log10(signalPower / noisePower);

std::cout << "Signal-to-noise ratio (SNR) in decibels: " << snr << std::endl;

std::vector<std::complex<double>> timeDomain;

for (const auto& sample : quantizedSignal) {

timeDomain.push\_back(std::complex<double>(sample.value, 0.0));

}

std::vector<std::complex<double>> freqDomain;

dft(timeDomain, freqDomain);

int N = freqDomain.size();

std::vector<std::complex<double>> freqDomainFiltered = freqDomain;

for (int k = 0; k < N; ++k) {

bool isHarmonic = false;

for (const auto& harmonic : harmonics) {

if (std::abs(k \* baseFrequency - harmonic.frequency) < 0.5 \* baseFrequency) {

isHarmonic = true;

break;

}

}

if (!isHarmonic) freqDomainFiltered[k] = 0.0;

}

std::vector<std::complex<double>> restoredTimeDomain;

idft(freqDomainFiltered, restoredTimeDomain);

// Усиление восстановленного сигнала

double maxOriginal = 0.0;

for (const auto& sample : originalSignal) {

if (std::abs(sample.value) > maxOriginal) {

maxOriginal = std::abs(sample.value);

}

}

double maxRestored = 0.0;

for (const auto& sample : restoredTimeDomain) {

if (std::abs(sample.real()) > maxRestored) {

maxRestored = std::abs(sample.real());

}

}

// Вычисление коэффициента усиления

double gain = maxOriginal / maxRestored;

// Применение усиления

for (auto& sample : restoredTimeDomain) {

sample \*= gain;

}

std::ofstream restoredFile("restored\_signal.txt");

if (!restoredFile.is\_open()) {

std::cerr << "Ошибка открытия файла восстановленного сигнала" << std::endl;

return 1;

}

std::vector<double> restoredSignalValues;

for (size\_t i = 0; i < restoredTimeDomain.size(); ++i) {

restoredFile << originalSignal[i].time << " " << restoredTimeDomain[i].real() << std::endl;

}

restoredFile.close();

double mse\_restored = 0.0;

for (size\_t i = 0; i < originalSignal.size(); ++i) {

double error = originalSignal[i].value - restoredTimeDomain[i].real();

mse\_restored += error \* error;

}

mse\_restored /= originalSignal.size();

double relative\_error = mse\_restored / signalPower;

std::cout << "\nRecovery results:\n"

<< "1. Relative error: " << relative\_error \* 100 << "%\n"

<< "2. Files have been created::\n"

<< " - restored\_signal.txt - restored signal\n";

if (relative\_error <= 0.05) {

std::cout << "3. Status: Success! The error is within 5%\n";

}

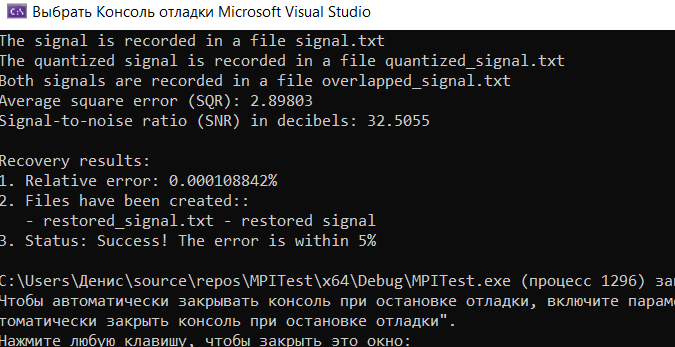
else {

std::cerr << "3. Error: Error exceeds 5%\n";

}

return 0;

}



Погрешность = 0.000108842%

## **Визуализация на Python**

import matplotlib.pyplot as plt

import numpy as np

signal\_data = np.loadtxt('signal.txt')

restored\_data = np.loadtxt('restored\_signal.txt')

*# Извлечение данных*

time\_signal = signal\_data[:, 0] *# Время для исходного сигнала*

value\_signal = signal\_data[:, 1] *# Значения исходного сигнала*

time\_restored = restored\_data[:, 0] *# Время для восстановленного сигнала*

value\_restored = restored\_data[:, 1] *# Значения восстановленного сигнала*

*# Построение графиков*

plt.figure(figsize=(10, 6))

plt.plot(time\_signal, value\_signal, label='Исходный сигнал', color='blue')

plt.plot(time\_restored, value\_restored, label='Восстановленный сигнал', color='red', linestyle='--')

plt.xlabel('Время (с)')

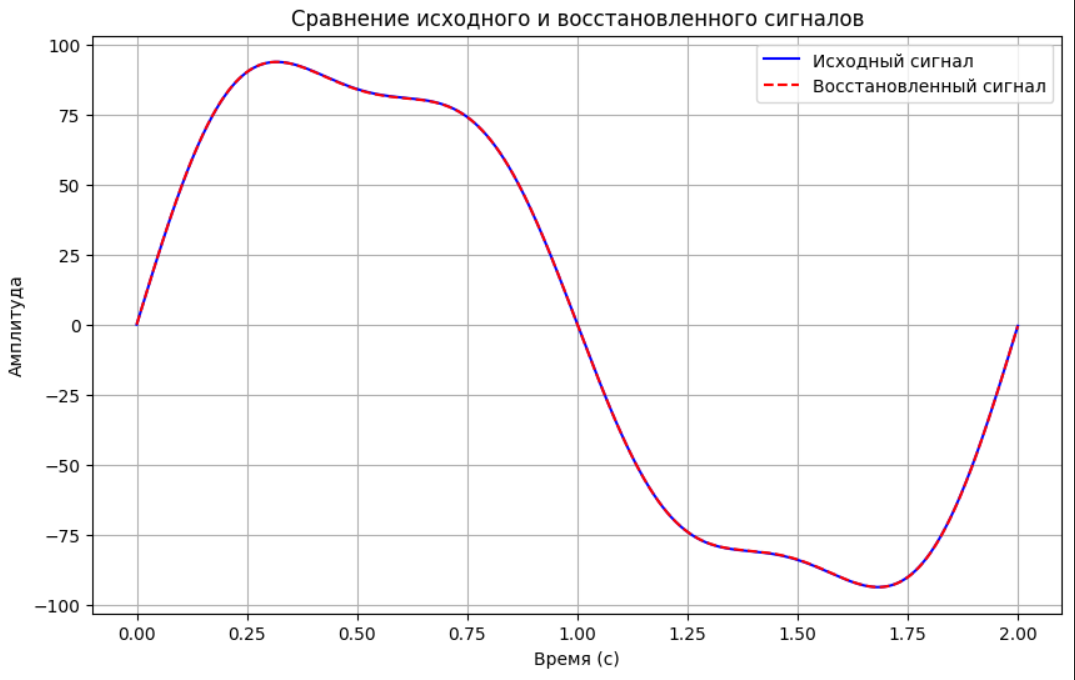
plt.ylabel('Амплитуда')

plt.title('Сравнение исходного и восстановленного сигналов')

plt.legend()

plt.grid(True)

plt.show()



## **Заключение**

В ходе лабораторной работы было выполнено восстановление исходного сигнала, сгенерированного в лабораторной работе №1, было использовано 2 важных функции прямое дискретное преобразование Фурье (DFT) и Обратное дискретное преобразование Фурье (IDFT). Результат получился точным с погрешностью равной 0.000108842%