Ниже представлен код программы, которая генерирует исходный сигнал, потом квантованный, а затем экспортирует данные в файлы

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#include <algorithm> // Для std::round

const double PI = 3.14159265358979323846;

// Функция для генерации сигнала

std::vector<double> generateSignal(int numHarmonics, const std::vector<double>& amplitudes, double samplingRate, double initialPhase, int numPoints, double deltaT) {

std::vector<double> signal(numPoints, 0.0);

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

double t = i \* deltaT;

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

int harmonicFrequency = 50 \* (h + 1); // Гармоники кратные 50 Гц

signal[i] += amplitudes[h] \* sin(2 \* PI \* harmonicFrequency \* t + initialPhase);

}

}

return signal;

}

// Функция для преобразования дискретного сигнала в цифровой (квантование)

std::vector<double> convertToDigital(const std::vector<double>& signal, int levels) {

double maxAmplitude = \*std::max\_element(signal.begin(), signal.end());

double minAmplitude = \*std::min\_element(signal.begin(), signal.end());

double range = maxAmplitude - minAmplitude;

std::vector<double> digitalSignal(signal.size());

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

// Квантование и масштабирование

digitalSignal[i] = minAmplitude + (std::round((signal[i] - minAmplitude) / range \* (levels - 1))) \* (range / (levels - 1));

}

return digitalSignal;

}

// Функция для экспорта данных в файл

void exportToFile(const std::vector<double>& signal, const std::string& filename) {

std::ofstream outFile(filename);

for (double value : signal) {

outFile << value << std::endl;

}

outFile.close();

}

int main() {

// Начальные параметры для проверки

int numHarmonics = 3; // Количество гармоник

std::vector<double> amplitudes = {1.0, 0.5, 0.3}; // Амплитуды гармоник

double samplingRate = 1000.0; // Частота дискретизации (Гц)

double initialPhase = 0.0; // Начальный сдвиг (радианы)

int numPoints = 100; // Количество точек (уменьшено для наглядности)

double deltaT = 1.0 / samplingRate; // Шаг времени (секунды)

int quantizationLevels = 16; // Уровни квантования для цифрового сигнала

// Генерация сигнала

std::vector<double> analogSignal = generateSignal(numHarmonics, amplitudes, samplingRate, initialPhase, numPoints, deltaT);

// Преобразование в цифровой сигнал

std::vector<double> digitalSignal = convertToDigital(analogSignal, quantizationLevels);

// Экспорт данных в файлы

exportToFile(analogSignal, "analog\_signal.txt");

exportToFile(digitalSignal, "digital\_signal.txt");

std::cout << "Дискретный и цифровой сигналы успешно сгенерированы и экспортированы в файлы 'analog\_signal.txt' и 'digital\_signal.txt'" << std::endl;

return 0;

}

Далее представлен код на python который визуализирует график сигнала.

import matplotlib.pyplot as plt

import numpy as np

# Чтение данных из файлов

with open('analog\_signal.txt', 'r') as file:

    analog\_signal = [float(line) for line in file]

with open('digital\_signal.txt', 'r') as file:

    digital\_signal = [float(line) for line in file]

# Построение графика

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

# Аналоговый сигнал (линия)

plt.plot(analog\_signal, label='Дискретный сигнал', color='blue', linewidth=2)

# Цифровой сигнал (столбчатая диаграмма)

plt.stem(digital\_signal, label='Цифровой сигнал', linefmt='red', markerfmt='ro', basefmt=' ')

# Настройки графика

plt.title('Дискретный и цифровой сигналы')

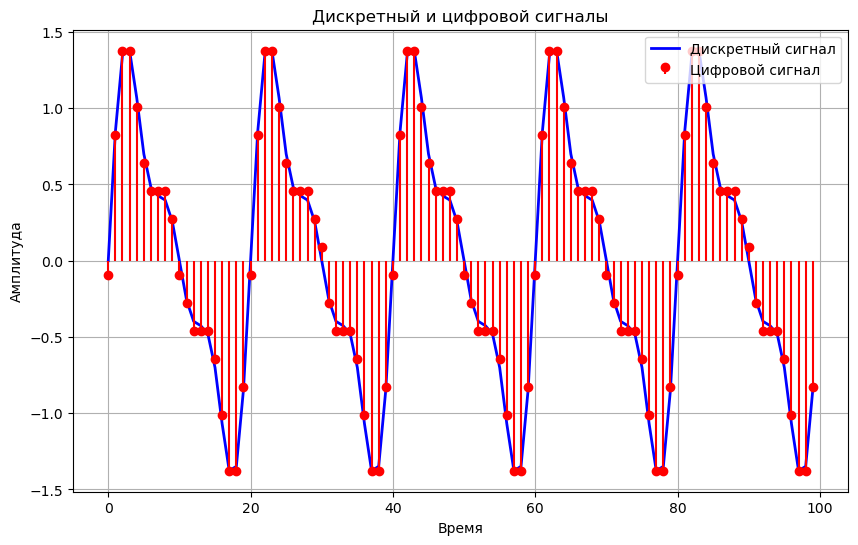
plt.xlabel('Время')

plt.ylabel('Амплитуда')

plt.legend()

plt.grid(True)

plt.show()



Теперь, ниже будет код который реализует преобразование Фурье и быстрое преобразование Фурье на квантованном сигнале, а затем производит обратное преобразование Фурье для восстановления сигнала с 5% погрешностью

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#include <algorithm>

#include <complex>

#include <numeric>

const double PI = 3.14159265358979323846;

const double ERROR\_THRESHOLD = 0.05; // 5% погрешность

// Функция для генерации сигнала

std::vector<double> generateSignal(int numHarmonics, const std::vector<double>& amplitudes,

double samplingRate, double initialPhase, int numPoints, double deltaT) {

std::vector<double> signal(numPoints, 0.0);

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

double t = i \* deltaT;

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

int harmonicFrequency = 50 \* (h + 1); // Гармоники кратные 50 Гц

signal[i] += amplitudes[h] \* sin(2 \* PI \* harmonicFrequency \* t + initialPhase);

}

}

return signal;

}

// Функция для преобразования дискретного сигнала в цифровой (квантование)

std::vector<double> convertToDigital(const std::vector<double>& signal, int levels) {

double maxAmplitude = \*std::max\_element(signal.begin(), signal.end());

double minAmplitude = \*std::min\_element(signal.begin(), signal.end());

double range = maxAmplitude - minAmplitude;

std::vector<double> digitalSignal(signal.size());

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

digitalSignal[i] = minAmplitude + (std::round((signal[i] - minAmplitude) / range \* (levels - 1))) \* (range / (levels - 1));

}

return digitalSignal;

}

// Прямое преобразование Фурье (DFT)

std::vector<std::complex<double>> computeDFT(const std::vector<double>& signal) {

int N = signal.size();

std::vector<std::complex<double>> dft(N);

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

dft[k] = 0;

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

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

dft[k] += signal[n] \* std::complex<double>(cos(angle), sin(angle));

}

}

return dft;

}

// Алгоритм БПФ (рекурсивная реализация Cooley-Tukey)

std::vector<std::complex<double>> computeFFT(const std::vector<double>& signal) {

int N = signal.size();

// Базовый случай

if (N == 1) {

return {std::complex<double>(signal[0], 0)};

}

// Разделение на четные и нечетные

std::vector<double> even(N/2), odd(N/2);

for (int i = 0; i < N/2; ++i) {

even[i] = signal[2\*i];

odd[i] = signal[2\*i + 1];

}

// Рекурсивные вызовы

auto evenFFT = computeFFT(even);

auto oddFFT = computeFFT(odd);

// Объединение результатов

std::vector<std::complex<double>> fft(N);

for (int k = 0; k < N/2; ++k) {

double angle = -2 \* PI \* k / N;

std::complex<double> twiddle = std::complex<double>(cos(angle), sin(angle));

fft[k] = evenFFT[k] + twiddle \* oddFFT[k];

fft[k + N/2] = evenFFT[k] - twiddle \* oddFFT[k];

}

return fft;

}

// Обратное преобразование Фурье

std::vector<double> computeInverseFFT(const std::vector<std::complex<double>>& fft) {

int N = fft.size();

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

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

std::complex<double> sum = 0;

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

double angle = 2 \* PI \* k \* n / N;

sum += fft[k] \* std::complex<double>(cos(angle), sin(angle));

}

signal[n] = sum.real() / N; // Масштабирование

}

return signal;

}

// Функция для вычисления погрешности между двумя сигналами

double calculateError(const std::vector<double>& original, const std::vector<double>& reconstructed) {

double sumSqOriginal = 0.0;

double sumSqDiff = 0.0;

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

sumSqOriginal += original[i] \* original[i];

double diff = original[i] - reconstructed[i];

sumSqDiff += diff \* diff;

}

return sqrt(sumSqDiff / sumSqOriginal);

}

// Функция для экспорта данных в файл

void exportToFile(const std::vector<double>& signal, const std::string& filename) {

std::ofstream outFile(filename);

for (double value : signal) {

outFile << value << std::endl;

}

outFile.close();

}

void exportComplexToFile(const std::vector<std::complex<double>>& signal, const std::string& filename) {

std::ofstream outFile(filename);

for (auto value : signal) {

outFile << value.real() << " " << value.imag() << std::endl;

}

outFile.close();

}

int main() {

// Начальные параметры для проверки

int numHarmonics = 3;

std::vector<double> amplitudes = {1.0, 0.5, 0.3};

double samplingRate = 1000.0;

double initialPhase = 0.0;

int numPoints = 128; // Для БПФ лучше использовать степень двойки

double deltaT = 1.0 / samplingRate;

int quantizationLevels = 16;

// Генерация сигнала

std::vector<double> analogSignal = generateSignal(numHarmonics, amplitudes, samplingRate, initialPhase, numPoints, deltaT);

// Преобразование в цифровой сигнал

std::vector<double> digitalSignal = convertToDigital(analogSignal, quantizationLevels);

// 1. Прямое преобразование Фурье (DFT)

auto dftResult = computeDFT(digitalSignal);

// 2. Быстрое преобразование Фурье (FFT)

auto fftResult = computeFFT(digitalSignal);

// 3. Обратное преобразование Фурье для восстановления сигнала

auto reconstructedSignal = computeInverseFFT(fftResult);

// 4. Сравнение с оригинальным сигналом

double error = calculateError(analogSignal, reconstructedSignal);

std::cout << "Погрешность восстановления: " << error \* 100 << "%" << std::endl;

if (error > ERROR\_THRESHOLD) {

std::cout << "Предупреждение: Погрешность превышает 5%" << std::endl;

} else {

std::cout << "Погрешность в допустимых пределах (<= 5%)" << std::endl;

}

// Экспорт данных в файлы

exportToFile(analogSignal, "analog\_signal.txt");

exportToFile(digitalSignal, "digital\_signal.txt");

exportComplexToFile(dftResult, "dft\_result.txt");

exportComplexToFile(fftResult, "fft\_result.txt");

exportToFile(reconstructedSignal, "reconstructed\_signal.txt");

return 0;

}

Ниже приведен код на python для визуализации.

import matplotlib.pyplot as plt

import numpy as np

from matplotlib.gridspec import GridSpec

# Чтение данных из файлов

def read\_complex\_signal(filename):

    with open(filename, 'r') as file:

        data = [line.split() for line in file]

    return np.array([complex(float(re), float(im)) for re, im in data])

# Чтение данных

analog\_signal = np.loadtxt('analog\_signal.txt')

digital\_signal = np.loadtxt('digital\_signal.txt')

dft\_result = read\_complex\_signal('dft\_result.txt')

fft\_result = read\_complex\_signal('fft\_result.txt')

reconstructed\_signal = np.loadtxt('reconstructed\_signal.txt')

# Создаем фигуру с 6 графиками

plt.figure(figsize=(18, 12))

gs = GridSpec(3, 2, figure=plt.gcf())

# 1. Исходный аналоговый сигнал

ax1 = plt.subplot(gs[0, 0])

ax1.plot(analog\_signal, label='Исходный аналоговый сигнал', color='blue', linewidth=2)

ax1.set\_title('Исходный аналоговый сигнал')

ax1.set\_xlabel('Отсчеты')

ax1.set\_ylabel('Амплитуда')

ax1.grid(True)

ax1.legend()

# 2. Квантованный цифровой сигнал

ax2 = plt.subplot(gs[0, 1])

ax2.stem(digital\_signal, linefmt='red', markerfmt='ro', basefmt=' ', label='Квантованный цифровой сигнал')

ax2.set\_title('Цифровой сигнал после квантования')

ax2.set\_xlabel('Отсчеты')

ax2.set\_ylabel('Амплитуда')

ax2.grid(True)

ax2.legend()

# 3. Спектр после DFT (модуль)

ax3 = plt.subplot(gs[1, 0])

ax3.plot(np.abs(dft\_result), label='Спектр (DFT)', color='green')

ax3.set\_title('Спектр сигнала после DFT (модуль)')

ax3.set\_xlabel('Частота')

ax3.set\_ylabel('Амплитуда')

ax3.grid(True)

ax3.legend()

# 4. Спектр после FFT (модуль)

ax4 = plt.subplot(gs[1, 1])

ax4.plot(np.abs(fft\_result), label='Спектр (FFT)', color='purple')

ax4.set\_title('Спектр сигнала после FFT (модуль)')

ax4.set\_xlabel('Частота')

ax4.set\_ylabel('Амплитуда')

ax4.grid(True)

ax4.legend()

# 5. Восстановленный сигнал

ax5 = plt.subplot(gs[2, 0])

ax5.plot(reconstructed\_signal, label='Восстановленный сигнал', color='orange')

ax5.set\_title('Сигнал после обратного FFT')

ax5.set\_xlabel('Отсчеты')

ax5.set\_ylabel('Амплитуда')

ax5.grid(True)

ax5.legend()

# 6. Сравнение исходного и восстановленного сигналов

ax6 = plt.subplot(gs[2, 1])

ax6.plot(analog\_signal, label='Исходный сигнал', color='blue', alpha=0.7)

ax6.plot(reconstructed\_signal, label='Восстановленный сигнал', color='red', linestyle='--', alpha=0.7)

ax6.set\_title('Сравнение исходного и восстановленного сигналов')

ax6.set\_xlabel('Отсчеты')

ax6.set\_ylabel('Амплитуда')

ax6.grid(True)

ax6.legend()

plt.tight\_layout()

plt.show()

# Дополнительный график - только сравнение

plt.figure(figsize=(12, 6))

plt.plot(analog\_signal, label='Исходный сигнал', color='blue', linewidth=2)

plt.plot(reconstructed\_signal, label='Восстановленный сигнал', color='red', linestyle='--', linewidth=2)

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

plt.xlabel('Отсчеты')

plt.ylabel('Амплитуда')

plt.grid(True)

plt.legend()

plt.show()

