Программа представляет из себя генератор периодического несинусоидального сигнала с различным количеством гармоник и их амплитуд, частот дискретизации. Параметры программа получает из файла "params.txt", где данные хранятся в таком порядке: количество гармоник, их характеристики (амплитуды, частоты, фаза), частота дискретизации, дельта t и количество точек. Полученные гармоники складываются в результирующую и их значения сохраняются в массив и в файл "result.txt". Полученный файл со сгенерированными гармониками и результирующей гармоникой можно визуализировать с помощью скрипта "graph.py".

После генерации непосредственно результирующей гармоники необходимо её разложить с помощью преобразований Фурье с использованием библиотеки "fftw3". Разложив полученные гармоники получаем файл "dft\_results.txt", в котором находятся распознанные гармоники и их характеристики в виде амплитуды, частоты и фазы. Для визуализации полученных гармоник используется скрипт "graph\_res.py" и файл с рассчитанными по параметрам гармоники "restored.txt". Дополнительно в консоль выводятся ошибки RMSE для найденных гармоник при разложении результирующей гармоники.

Разберем код основного файла для генерации гармоник " HarmonicGenerator.cpp"

Прежде всего подключаются все необходимые **заголовочные файлы** и **определения:**

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include <complex>

#include <algorithm>

#include <windows.h>

#include <fftw3.h>

#pragma comment(lib, "libfftw3-3.lib")

Далее для удобства работы с введенными данными гармоник определим следующие **структуры данных**:

typedef complex<double> Complex;

struct Harmonic {

double amplitude;

double phase;

double frequency;

};

struct DFTPeak {

double frequency;

double amplitude;

double phaseDegrees;

};

Где:

* Complex — тип для работы с комплексными числами.
* Harmonic — структура для хранения параметров гармоники (амплитуда, фаза, частота).
* DFTPeak — структура для хранения результатов DFT (частота, амплитуда, фаза).

**Определим все необходимые функции**

Функция преобразования Фурье:

vector<Complex> computeDFT(const vector<double>& signal) {

int N = signal.size();

vector<Complex> result(N);

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

double real = 0.0;

double imag = 0.0;

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

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

real += signal[n] \* cos(angle);

imag -= signal[n] \* sin(angle);

}

result[k] = Complex(real, imag);

}

return result;

}

Функция вычисляет дискретное преобразование Фурье (DFT) для входного сигнала. Использует формулу DFT для вычисления комплексных коэффициентов.

Функция анализа результатов преобразования Фурье:

vector<DFTPeak> analyzeDFT(const vector<complex<double>>& dftResult, double samplingFreq, int numPoints) {

vector<DFTPeak> peaks;

int N = numPoints;

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

double amplitude = 2.0 \* abs(dftResult[k]) / N;

if (amplitude < 1e-6) continue;

double phase = arg(dftResult[k]) \* 180.0 / M\_PI + 90.0;

phase = fmod(phase + 180.0, 360.0) - 180.0;

double frequency = k \* samplingFreq / N;

peaks.push\_back({ frequency, amplitude, phase });

}

return peaks;

}

Функция анализирует результаты DFT и извлекает пики (частоты, амплитуды, фазы. Использует амплитуду и фазу для каждого коэффициента DFT.

Функция загрузки параметров из файла

bool loadFromFile(const string& filename, vector<Harmonic>& harmonics, int& numHarmonics, double& samplingFreq, double& deltaT, int& numPoints) {

ifstream file(filename);

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

cerr << "Error opening file '" << filename << "'" << endl;

return false;

}

file >> numHarmonics;

if (numHarmonics <= 0 || numHarmonics > maxHarmonics) {

cerr << "Invalid number of harmonics in file." << endl;

return false;

}

harmonics.resize(numHarmonics);

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

file >> harmonics[i].amplitude

>> harmonics[i].frequency

>> harmonics[i].phase;

}

file >> samplingFreq >> deltaT >> numPoints;

return true;

}

Функция загружает параметры из файла: количество гармоник, их параметры, частоту дискретизации, шаг по времени и количество точек из файла "params.txt".

После определения необходимых функций опишем основную функцию "main", выполняющую описанные действия.

В начале основной функции устанавливается кодировка консоли а также определяются переменные, используемые в дальнейшей работе кода. Создается вектор *harmonics* для хранения параметров гармоник. Инициализируются переменные для хранения количества гармоник, метода ввода, частоты дискретизации, шага по времени и количества точек.

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

vector<Harmonic> harmonics(maxHarmonics);

int numHarmonics, inputMethod;

double samplingFreq, deltaT;

int numPoints;

Далее вызывается функция loadFromFile, которая загружает параметры из файла params.txt. Если загрузка прошла успешно, переменная success будет равна true.

bool success = loadFromFile("params.txt", harmonics, numHarmonics, samplingFreq, deltaT, numPoints);

Соответственно, если условие соблюдается, и загрузка выполнена успешно, то в дальнейшем выполняется код по генерации гармоник по заданным параметрам – создается двумерный вектор signalValues для хранения значений каждой гармоники, где для каждой вычисляются значения сигнала в зависимости от времени и параметров гармоники.

vector<vector<double>> signalValues(numHarmonics, vector<double>(numPoints));

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

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

double time = timeIndex \* deltaT;

double phaseRad = harmonics[harmonicIndex].phase \* M\_PI / 180.0;

signalValues[harmonicIndex][timeIndex] = harmonics[harmonicIndex].amplitude \*

sin(2 \* M\_PI \* harmonics[harmonicIndex].frequency \* time + phaseRad);

}

}

После генерации открывается файл result.txt для записи значений времени каждой гармоники и результирующего сигнала.

ofstream outputFile("result.txt");

if (outputFile.is\_open()) {

outputFile << "Время\t";

for (int i = 1; i <= numHarmonics; ++i) {

outputFile << "Гармоника\_" << i << "\t";

}

outputFile << "Результирующая\_гармоника\n";

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

double time = timeIndex \* deltaT;

outputFile << time << "\t";

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

outputFile << signalValues[harmonicIndex][timeIndex] << "\t";

}

outputFile << resultantSignal[timeIndex] << "\n";

}

outputFile.close();

cout << "Данные успешно записаны в файл result.txt!" << endl;

}

else {

cerr << "Не удалось открыть файл для записи!" << endl;

}

После генерации гармоник и формирования массива результирующей гармоники вызывается функция computeDFT для вычисления дискретного преобразования Фурье (DFT) результирующего сигнала. После чего функция analyzeDFT для анализа результатов DFT и извлечения пиков (частот, амплитуд, фаз).

vector<complex<double>> dftResult = computeDFT(resultantSignal);

vector<DFTPeak> peaks = analyzeDFT(dftResult, samplingFreq, numPoints);

После преобразований необходимо проверить насколько точно были проведены операции разложения. Для этого создается двумерный вектор *restoredHarmonics* для хранения восстановленных гармоник, где каждая восстановленная гармоника вычисляют значения сигнала в зависимости от времени и параметров гармоники.

vector<vector<double>> restoredHarmonics(peaks.size(), vector<double>(numPoints));

for (size\_t harmonicIdx = 0; harmonicIdx < peaks.size(); ++harmonicIdx) {

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

double t = timeIdx \* deltaT;

double phaseRad = peaks[harmonicIdx].phaseDegrees \* M\_PI / 180.0;

restoredHarmonics[harmonicIdx][timeIdx] = peaks[harmonicIdx].amplitude \*

sin(2 \* M\_PI \* peaks[harmonicIdx].frequency \* t + phaseRad);

}

}

После генерации полученные значения необходимо снова записать в файл, но уже под припиской восстановленных данных – "restored.txt".

ofstream restoredFile("restored.txt");

if (restoredFile.is\_open()) {

restoredFile << "Время\t";

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

restoredFile << "Гармоника\_" << i + 1 << "\t";

}

restoredFile << "\n";

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

double time = timeIdx \* deltaT;

restoredFile << time << "\t";

for (size\_t harmonicIdx = 0; harmonicIdx < peaks.size(); ++harmonicIdx) {

restoredFile << restoredHarmonics[harmonicIdx][timeIdx] << "\t";

}

restoredFile << "\n";

}

restoredFile.close();

cout << "Восстановленные гармоники записаны в restored.txt!" << endl;

}

else {

cerr << "Ошибка при создании файла restored.txt!" << endl;

}

Для проверки точности проведенных преобразований выполняется функция для расчета ошибок RMSE. Создается вектор rmseErrors для хранения значения ошибки, где каждая восстановленная гармоника ищет ближайшую исходная гармонику по частоте. Если найдена соответствующая гармоника, вычисляется ошибка RMSE и выводится в консоль.

vector<double> rmseErrors(peaks.size(), 0.0);

for (size\_t restoredIdx = 0; restoredIdx < peaks.size(); ++restoredIdx) {

double restoredFreq = peaks[restoredIdx].frequency;

int closestOrigIdx = -1;

double minDiff = INFINITY;

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

double origFreq = harmonics[origIdx].frequency;

double diff = abs(restoredFreq - origFreq);

if (diff < minDiff) {

minDiff = diff;

closestOrigIdx = origIdx;

}

}

if (closestOrigIdx != -1 && minDiff < 1e-6) {

double sumSq = 0.0;

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

double origVal = signalValues[closestOrigIdx][timeIdx];

double restoredVal = restoredHarmonics[restoredIdx][timeIdx];

sumSq += (origVal - restoredVal) \* (origVal - restoredVal);

}

rmseErrors[restoredIdx] = sqrt(sumSq / numPoints);

cout << "Ошибка RMSE для гармоники " << restoredIdx + 1

<< " (частота " << restoredFreq << " Гц): "

<< rmseErrors[restoredIdx] << endl;

}

}

Код файла для визуализации сгенерированных гармоник "graph.py":

import matplotlib.pyplot as plt

import numpy as np

def read\_data\_from\_file(filename):

with open(filename, 'r') as file:

lines = file.readlines()

headers = lines[0].strip().split('\t')

data = []

for line in lines[1:]:

values = list(map(float, line.strip().split('\t')))

data.append(values)

return headers, np.array(data)

def plot\_harmonics(headers, data):

time\_column = headers.index('Время')

harmonics\_columns = [header for header in headers if 'Гармоника\_' in header]

resultant\_column = headers[-1]

fig, ax = plt.subplots(figsize=(10, 6))

cmap = plt.get\_cmap('tab10')

for i, column\_name in enumerate(harmonics\_columns):

index = headers.index(column\_name)

ax.plot(data[:, time\_column], data[:, index], label=column\_name, color=cmap(i))

index = headers.index(resultant\_column)

ax.plot(data[:, time\_column], data[:, index], label=resultant\_column, linewidth=3, color='k')

ax.set\_xlabel('Время')

ax.set\_ylabel('Значение')

ax.legend(loc='best')

ax.grid(True)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

filename = 'result.txt'

headers, data = read\_data\_from\_file(filename)

plot\_harmonics(headers, data)

Код файла для визуализации сгенерированных гармоник "graph\_res.py":

import matplotlib.pyplot as plt

import numpy as np

def read\_data\_from\_file(filename):

with open(filename, 'r') as file:

lines = file.readlines()

headers = lines[0].strip().split('\t')

data = []

for line in lines[1:]:

values = list(map(float, line.strip().split('\t')))

data.append(values)

return headers, np.array(data)

def plot\_harmonics(headers, data):

time\_column = headers.index('Время')

harmonics\_columns = [header for header in headers if 'Гармоника\_' in header]

resultant\_column = headers[-1]

fig, ax = plt.subplots(figsize=(10, 6))

cmap = plt.get\_cmap('tab10')

for i, column\_name in enumerate(harmonics\_columns):

index = headers.index(column\_name)

ax.plot(data[:, time\_column], data[:, index], label=column\_name, color=cmap(i)) # Используем cmap для назначения цвета

ax.set\_xlabel('Время')

ax.set\_ylabel('Значение')

ax.legend(loc='best')

ax.grid(True)

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

filename = 'restored.txt'

headers, data = read\_data\_from\_file(filename)

plot\_harmonics(headers, data)

Для демонстрации работы программы будет использовано 4 набора данных параметров гармоник:

1)

2

1.0 5 0

0.7 10 45

1000 0.001 1000

2)

3

0.8 50 15

0.6 100 -30

0.5 250 180

5000 0.0002 500

3)

3

1.0 10 30

0.5 20 45

0.3 30 60

1000 0.001 400

4)

3

1.0 40 30

0.7 80 -60

0.4 120 90

2400 0.00041667 240

Выполним всю последовательность работы программы – получение входных параметров гармоник, генерацию гармоник, сохранение в файл, разложение по методу Фурье, сохранение в файл, проверка на ошибки. И соответственно выведем результаты работы программы.

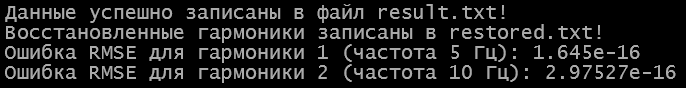


Рисунок 1 – Выполнение программы и вывод в консоли первого набора данных

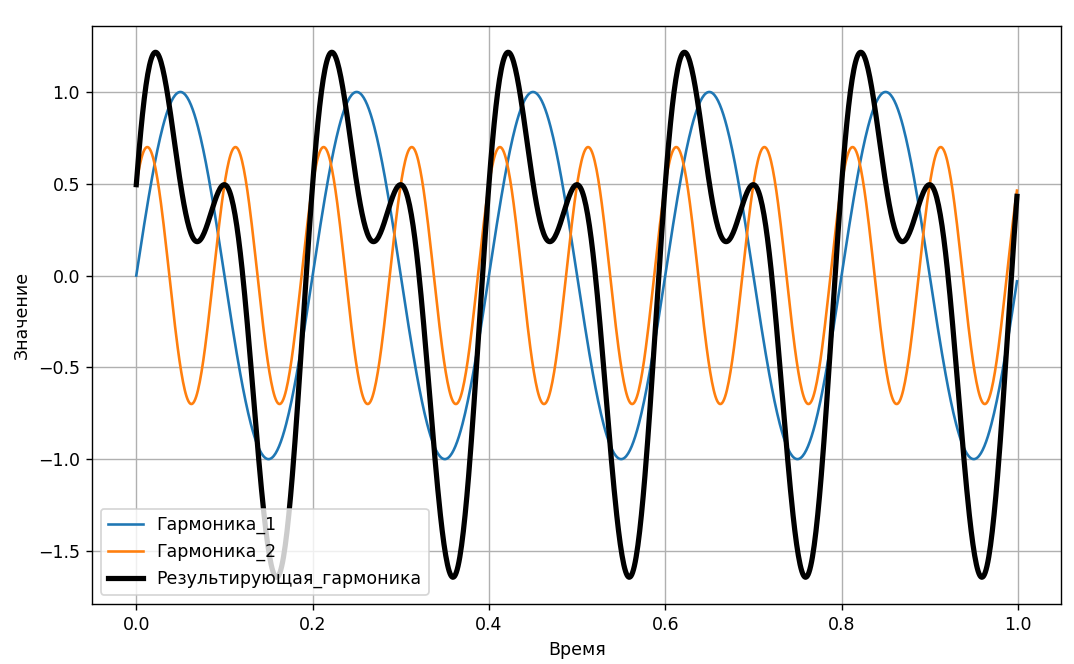


Рисунок 2 – График сгенерированных и результирующей гармоник первого набора данных

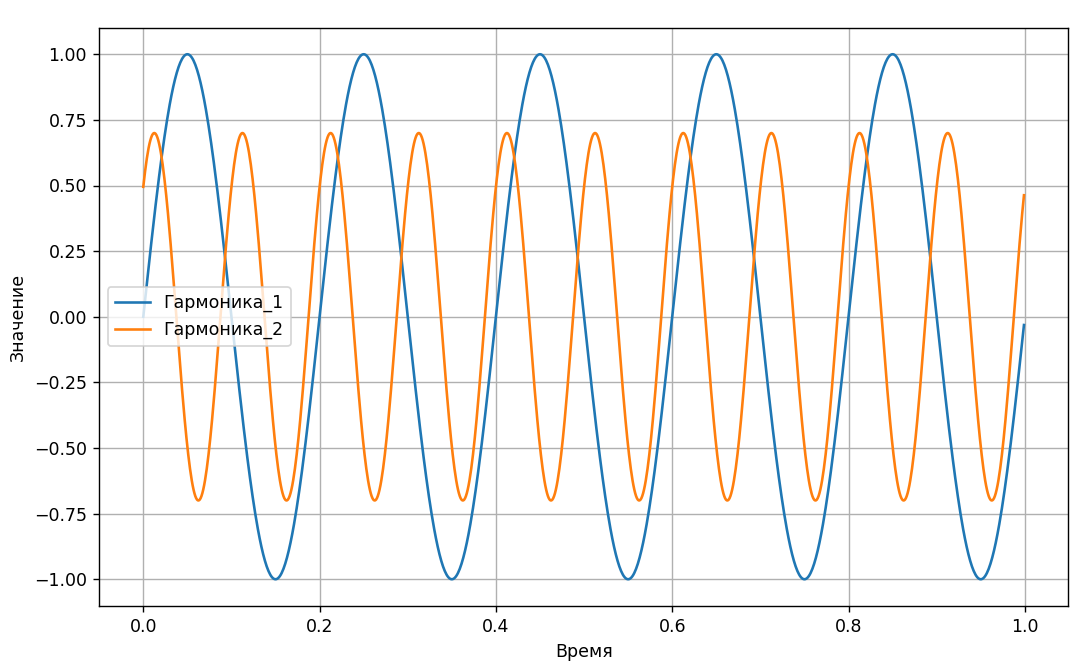


Рисунок 3 – График восстановленных гармоник первого набора данных

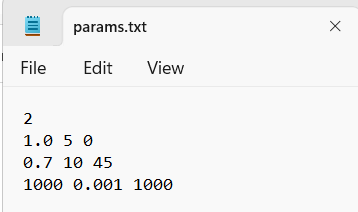


Рисунок 4 – Пример наполнения файла "params.txt"

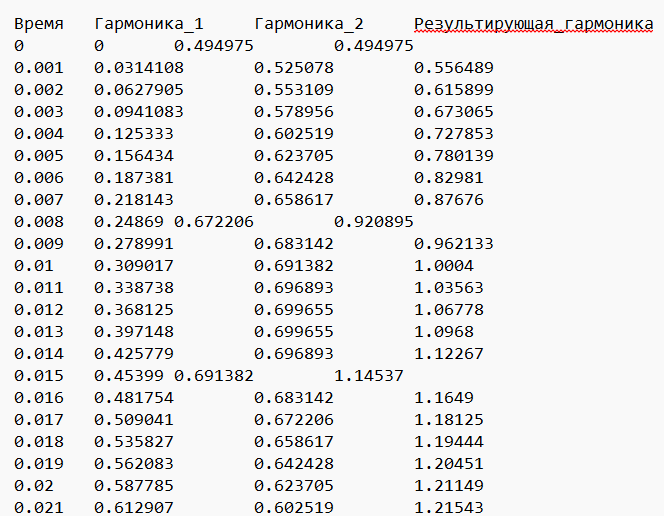


Рисунок 5 – Пример наполнения файла "result.txt"

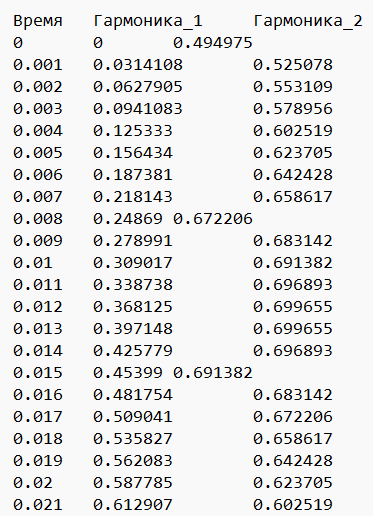


Рисунок 6 – Пример наполнения файла "restored.txt"

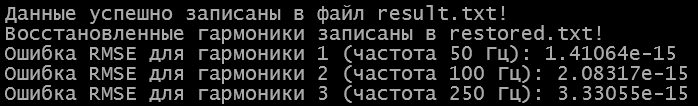


Рисунок 7 – Выполнение программы и вывод в консоли второго набора данных

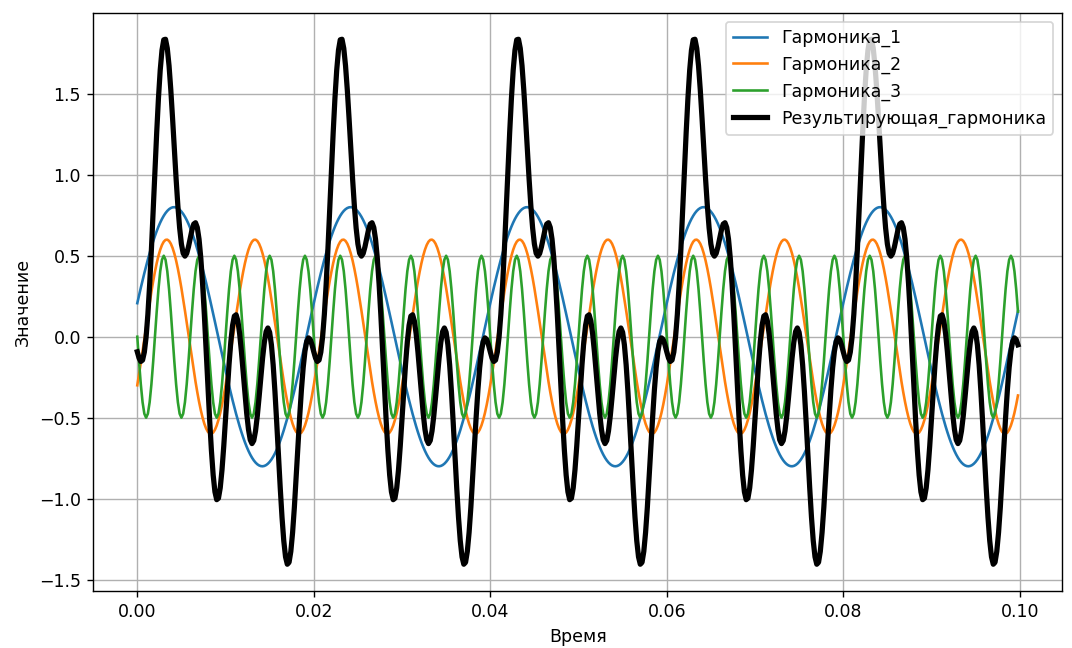


Рисунок 8 – График сгенерированных и результирующей гармоник второго набора данных

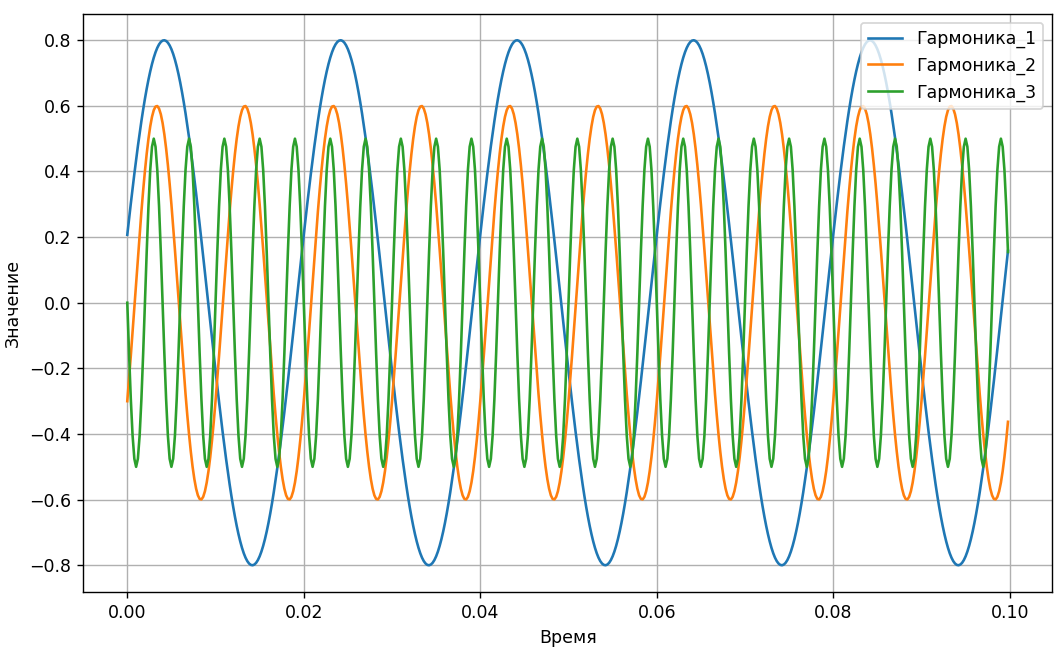


Рисунок 9 – График восстановленных гармоник второго набора данных

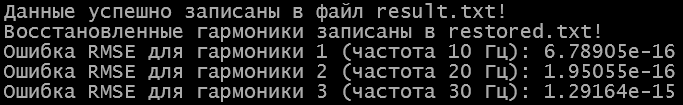


Рисунок 10 – Выполнение программы и вывод в консоли третьего набора данных

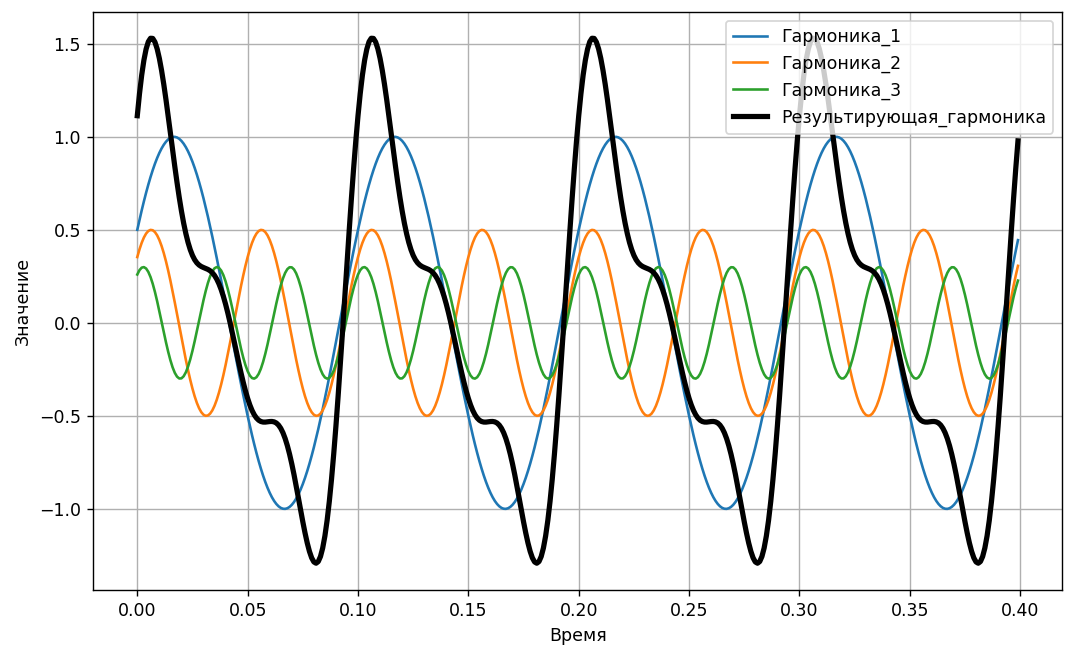


Рисунок 11 – График сгенерированных и результирующей гармоник третьего набора данных

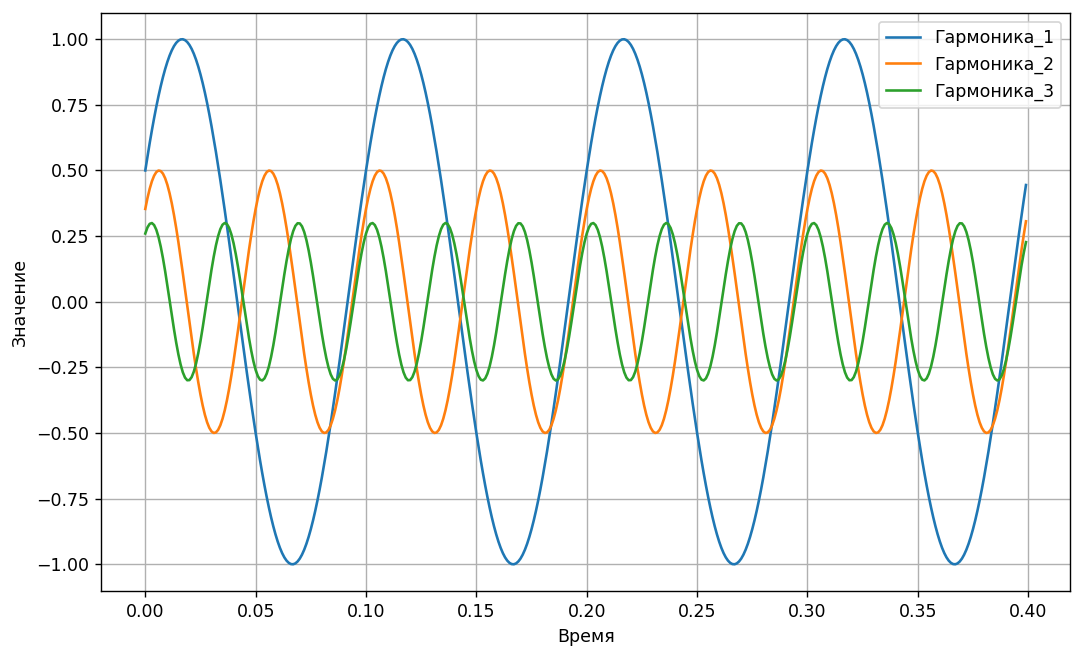


Рисунок 12 – График восстановленных гармоник третьего набора данных

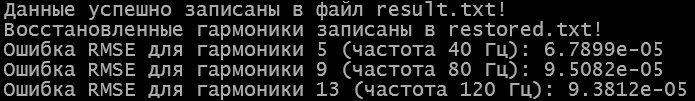


Рисунок 13 – Выполнение программы и вывод в консоли четвертого набора данных

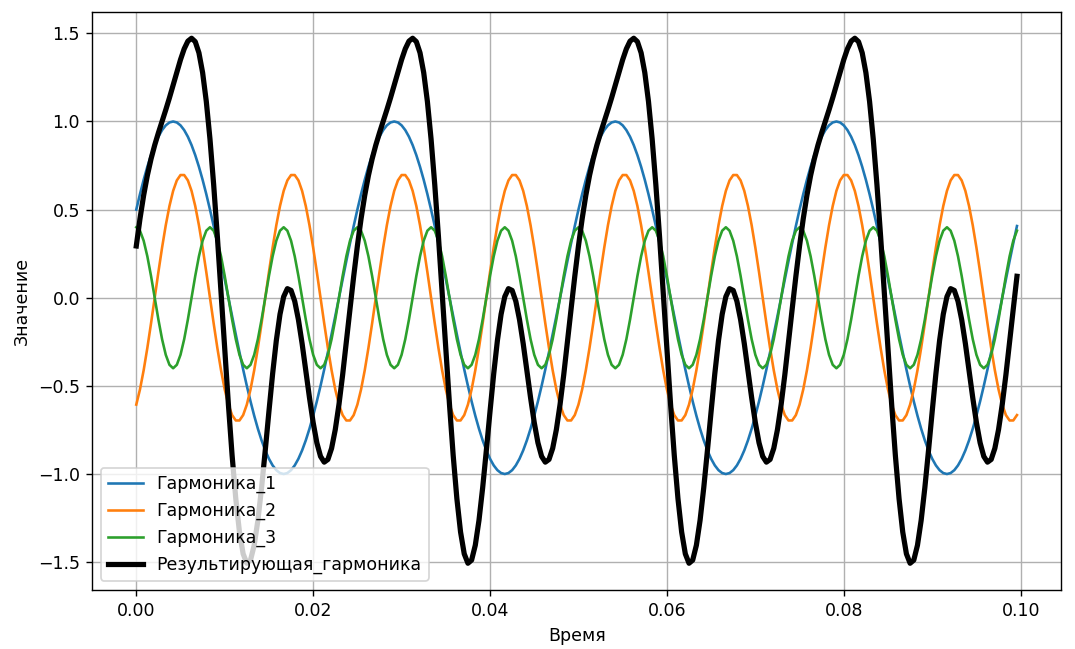


Рисунок 14 – График сгенерированных и результирующей гармоник четвертого набора данных

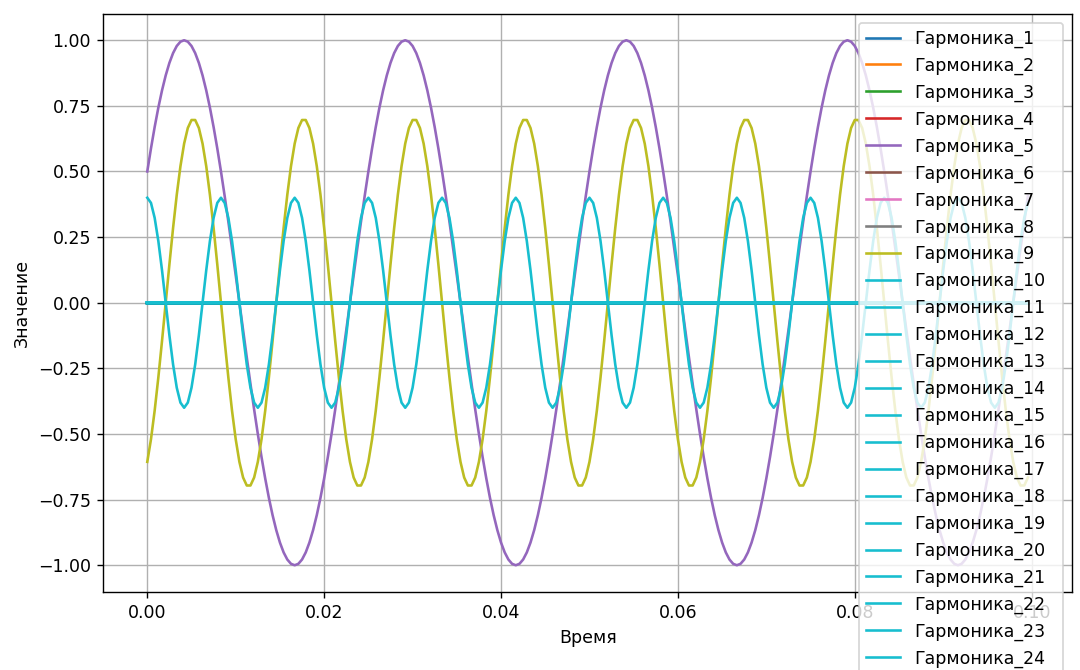


Рисунок 15 – График восстановленных гармоник четвертого набора данных