**Липецкий государственный технический университет**

Факультет автоматизации и информатики

Кафедра автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА № *5*

по дисциплине «Численные методы»

«Аппроксимация функций:

интерполяция и сглаживание»

Студент

Группа АС 21-1 \_\_\_\_\_\_\_\_\_\_\_\_ Станиславчук С.М.

подпись, дата

Руководитель

Д.т.н, профессор кафедры АСУ \_\_\_\_\_\_\_\_\_\_\_\_ Седых И.А.

подпись, дата

Липецк 2023 г.

Содержание:

2. Задание кафедры.

3. Ход работы.

4. Выводы и сравнения результатов.

5. Полный код программ.

2. Задание кафедры

1) Интерполяционный многочлен **Лагранжа** по функции + график.

2) Интерполяционный многочлен **Ньютона** по функции + график

3) Разбить отрезок на m подотрезков, для каждого построить интерполяционный многочлен **Лагранжа** или **Ньютона**. + график с отмеченными на нем исходными точками + таблица разделенных разностей.

4) Разбить отрезок функции на m\*n подотрезков + найти значения в узлах сетки + построить **линейный** интерполяционный сплайн и вывести его коэффициенты.

5) Разбить отрезок функции на m\*n подотрезков. Найти значения функции в узлах сетки + построить **кубический** интерполяционный сплайн и вывести его график.

6) Построить линейную аппроксимацию функции y = a + bx;  
аппроксимирующую функцию по варианту. Для полученных аппроксимирующих функций вывести результаты в виде

таблицы (x, y, Y), вывести найденные параметры функций, вывести средние

квадратические ошибки аппроксимации.

3. Ход работы

1) Многочлен Лагранжа

Многочлен Лагранжа - это многочлен степени (n-1), который проходит через n точек на плоскости. Он используется для интерполяции функций, т.е. построения функции, которая проходит через заданные точки.

Пусть у нас есть n точек на плоскости, заданных координатами (x1, y1), (x2, y2), ..., (xn, yn). Тогда многочлен Лагранжа выглядит следующим образом:

L(x) = y1 \* l1(x) + y2 \* l2(x) + ... + yn \* ln(x),

где L(x) - многочлен Лагранжа, y1, y2, ..., yn - значения функции в точках x1, x2, ..., xn, а l1(x), l2(x), ..., ln(x) - n-1 степенные многочлены, называемые многочленами Лагранжа

C++ реализация

double LagrangeBasis(double x, const vector<double>& nodes, int i)

{

double result = 1;

for (int j = 0; j < nodes.size(); j++)

{

if (j != i)

{

result \*= (x - nodes[j]) / (nodes[i] - nodes[j]);

}

}

return result;

}

double LagrangeInterpolation(double x, const vector<double>& nodes, const vector<double>& values)

{

double result = 0;

for (int i = 0; i < nodes.size(); i++)

{

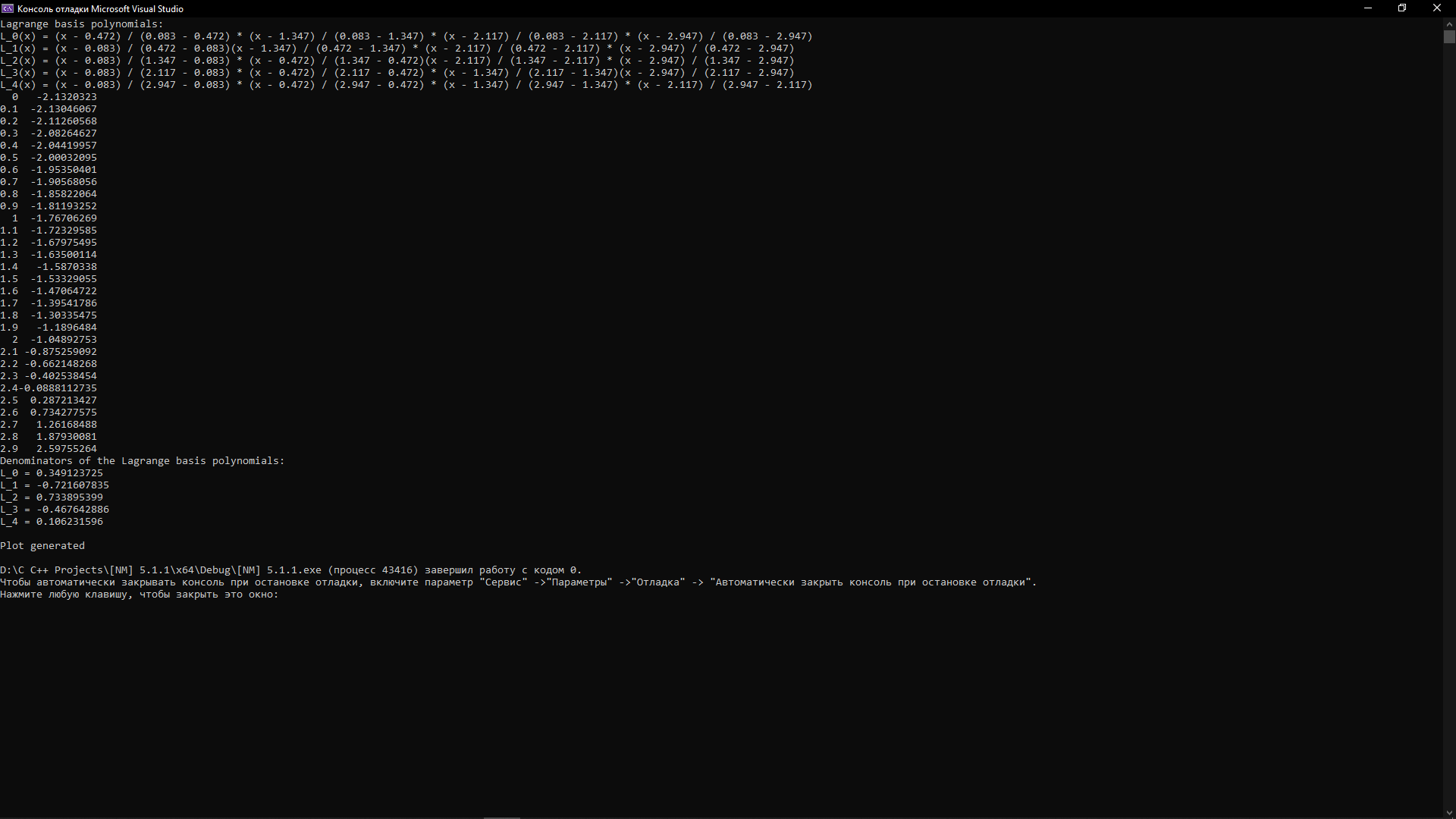
result += values[i] \* LagrangeBasis(x, nodes, i);

}

return result;

}

Результат программы:



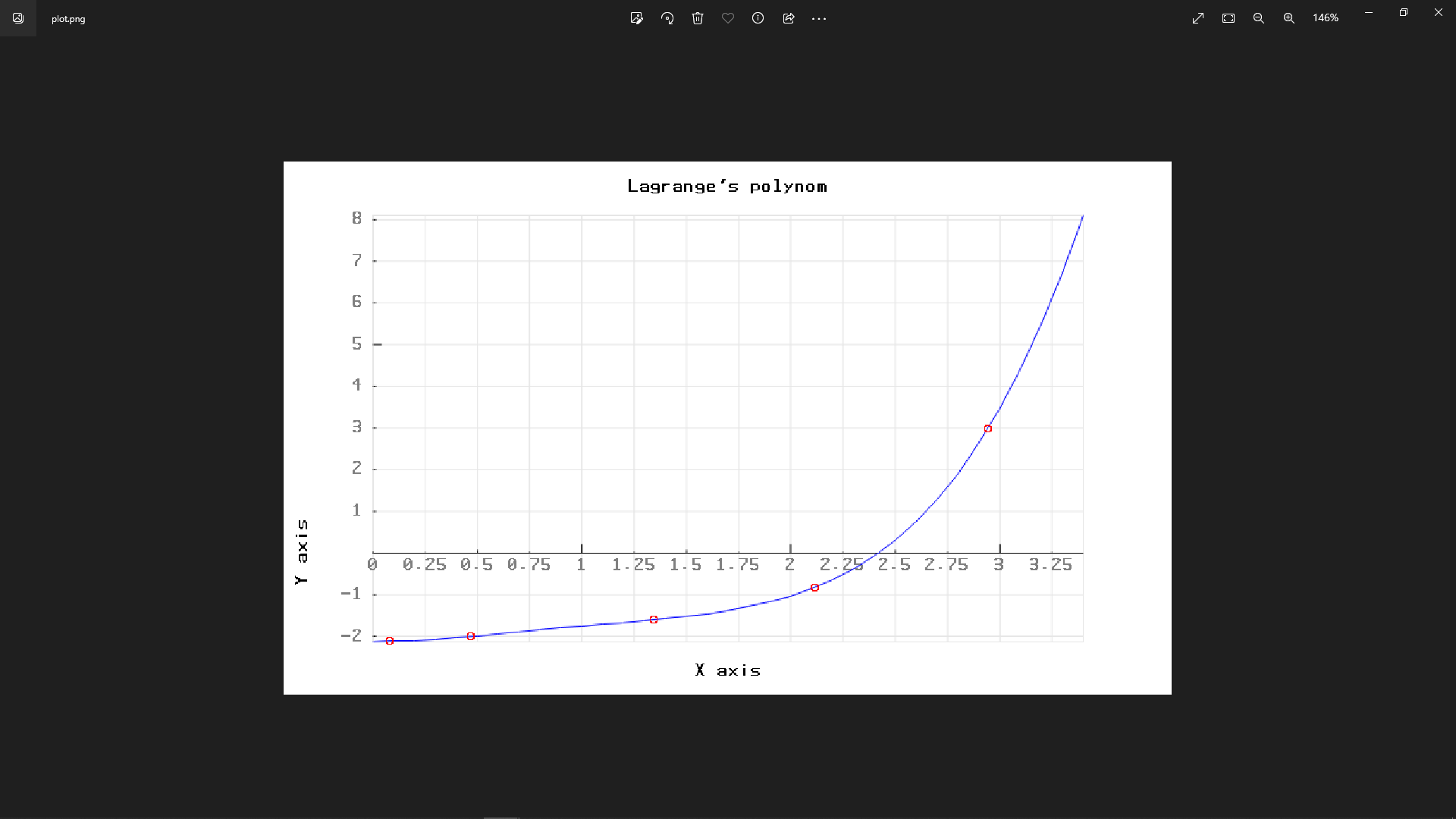


График функции точно совпал с заданными точками

2) Многочлен Ньютона

Многочлен Ньютона - это еще один метод интерполяции функций, который использует многочлены для приближения неизвестной функции на заданном отрезке. Этот метод был предложен английским математиком Исааком Ньютоном.

Многочлен Ньютона строится на основе разделенных разностей, которые являются рекурсивными коэффициентами для построения многочлена.

Предположим, что у нас есть набор узлов (x0, y0), (x1, y1), ..., (xn, yn), и мы хотим найти многочлен, который проходит через эти точки. Многочлен Ньютона может быть выражен следующим образом:

f(x) = f[x0] + f[x0,x1](x - x0) + f[x0,x1,x2](x - x0)(x - x1) + ... + f[x0,x1,...,xn](x - x0)(x - x1)...(x - xn-1),

где f[xi] - это значение функции в точке xi, а f[xi, xj, ..., xk] - это разделенная разность, которая может быть вычислена следующим образом:

f[xi, xj] = (f(xi) - f(xj)) / (xi - xj),

f[xi, xj, ..., xk] = (f[xi, xj, ..., xk-1] - f[xj, xk, ..., xk-1]) / (xi - xk).

Алгоритм программы:  
vector<vector<double>> divided\_differences\_table(const vector<double>& nodes, const vector<double>& values) {

int n = nodes.size();

vector<vector<double>> f(n, vector<double>(n));

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

f[i][0] = values[i];

}

for (int j = 1; j < n; ++j) {

for (int i = j; i < n; ++i) {

f[i][j] = (f[i][j - 1] - f[i - 1][j - 1]) / (nodes[i] - nodes[i - j]);

}

}

return f;

}

double newton\_interpolation\_polynomial(const vector<double>& nodes, const vector<double>& values, double x) {

vector<vector<double>> f = divided\_differences\_table(nodes, values);

double p = f[0][0];

double prod = 1.0;

int n = nodes.size();

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

prod \*= (x - nodes[i - 1]);

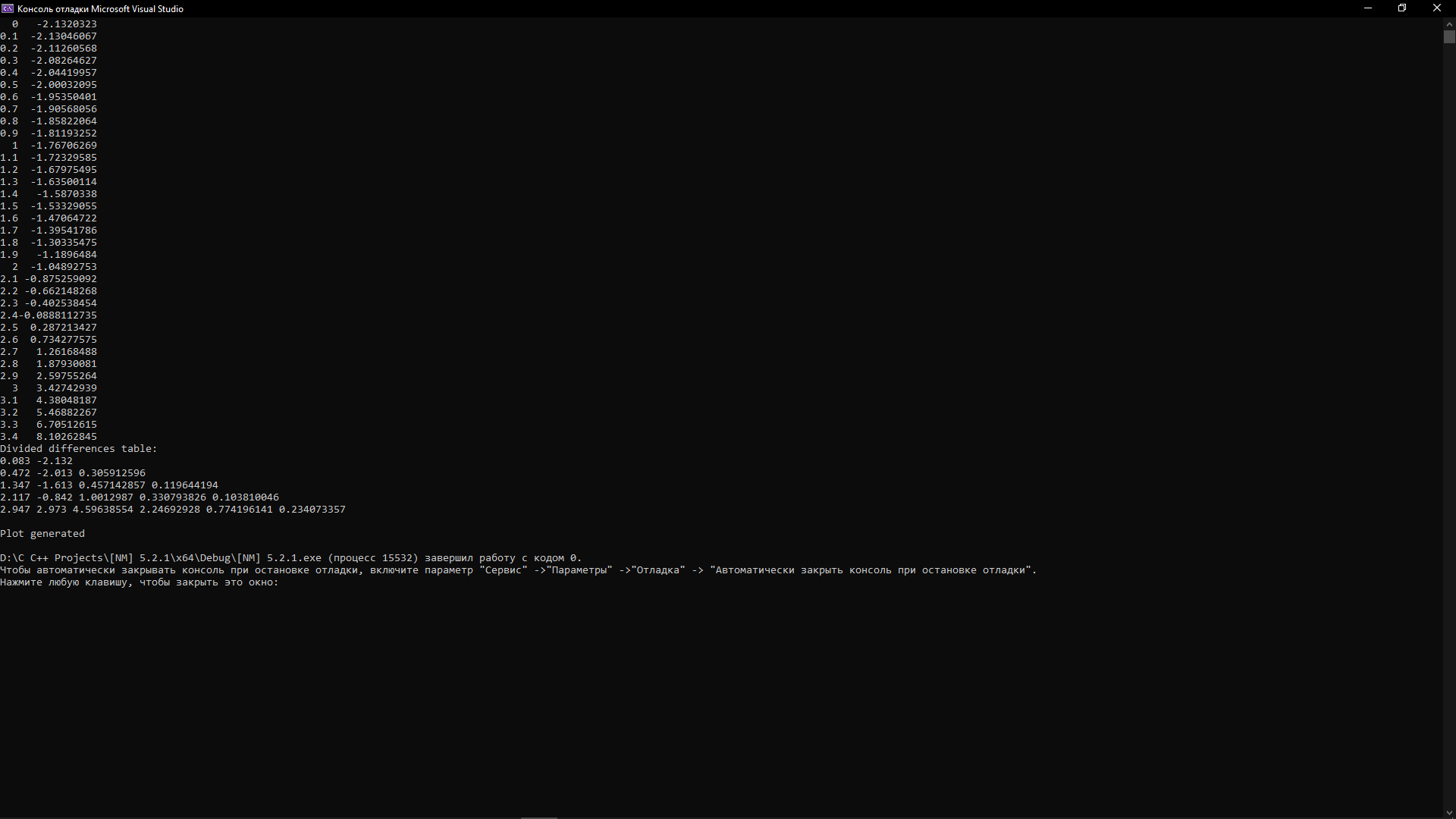
p += prod \* f[i][i];

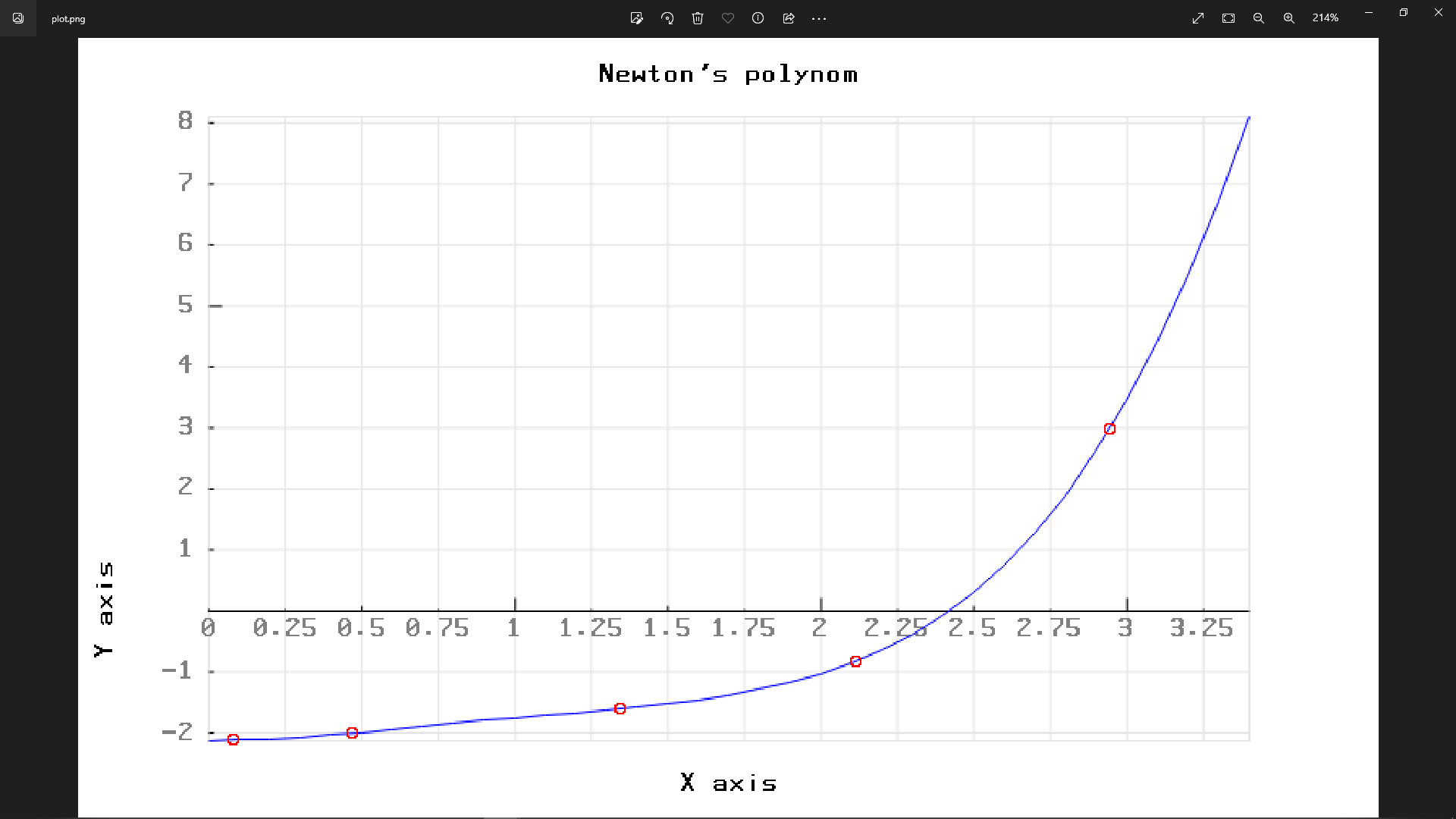
}

return p;

}

Результат программы:





Аналогично правильно построился график и с многочленом Ньютона.

3) Метод Лагранжа 3ей степени для функции sin(Pi \* x)

// функция для вычисления значения многочлена Лагранжа порядка 3 в заданной точке по равномерной сетке

double lagrange\_poly(double x, const vector<double>& x\_values, const vector<double>& y\_values) {

double result = 0.0;

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

double term = y\_values[i];

for (size\_t j = 0; j < x\_values.size(); ++j) {

if (j != i) {

term \*= (x - x\_values[j]) / (x\_values[i] - x\_values[j]);

}

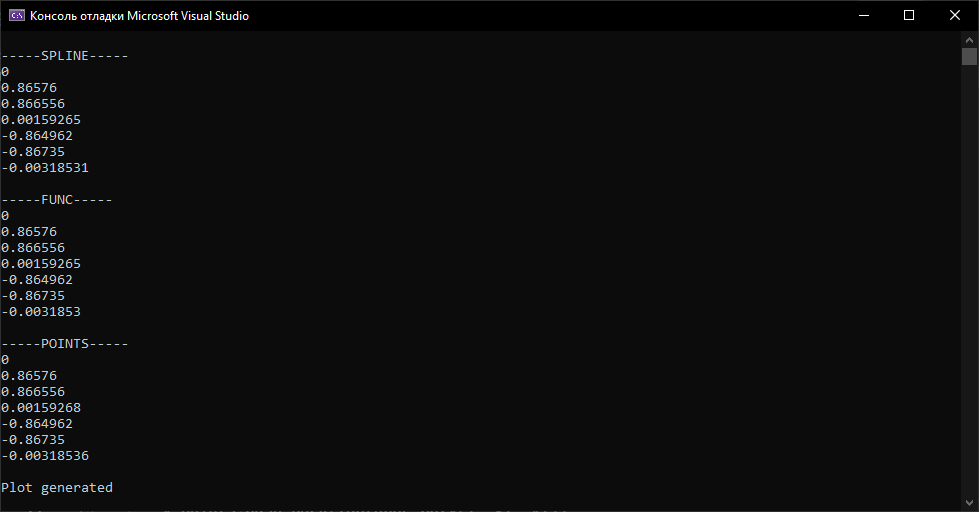
}

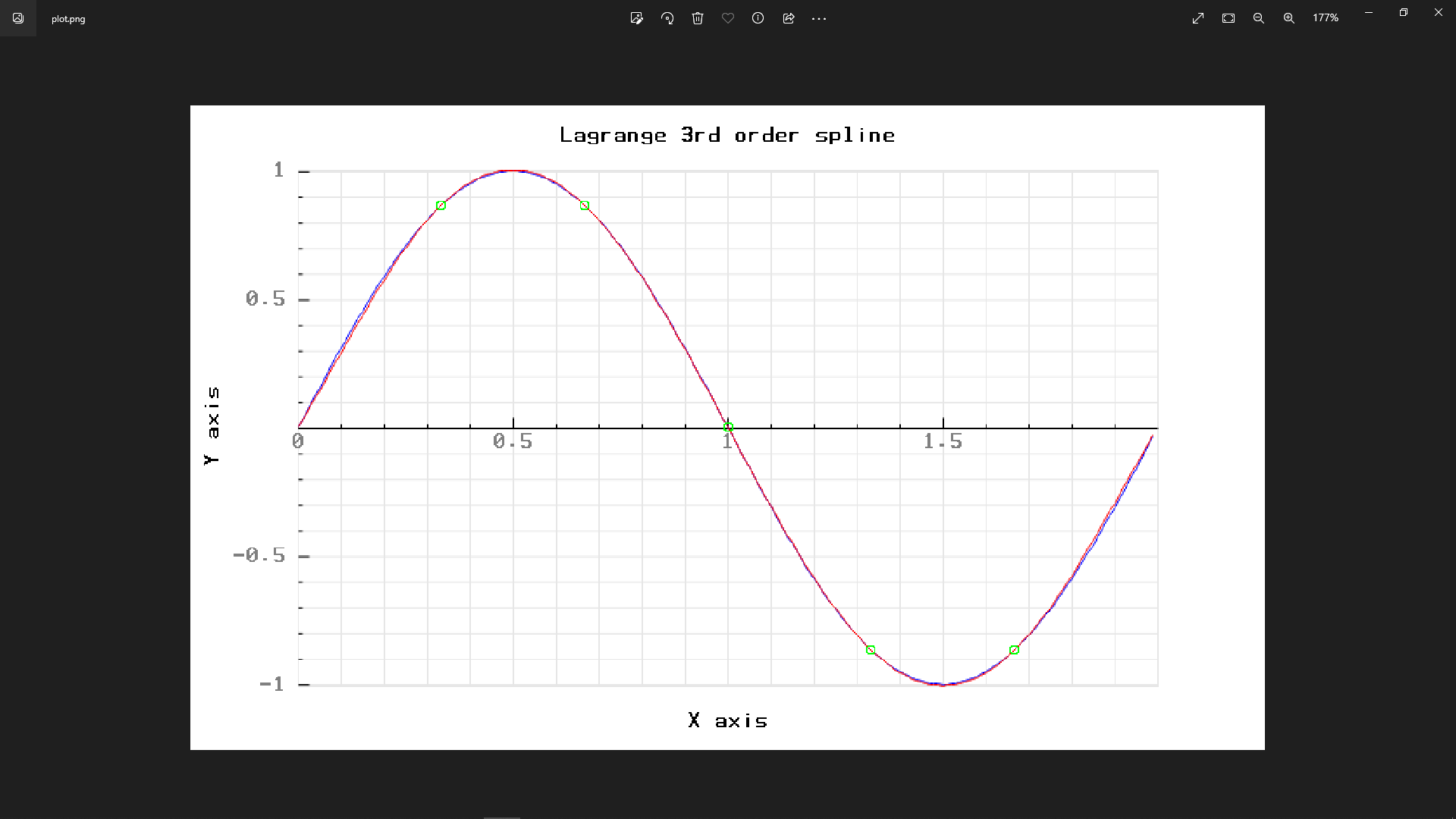
result += term;

}

return result;

}





Данный сплайн также оказался достаточно точным

4) Линейный интерполяционный сплайн

double linear\_spline(double x, const vector<double>& xi, const vector<double>& yi) {

int n = xi.size();

double sum = 0;

for (int i = 0; i < n - 1; i++) {

if (x >= xi[i] && x <= xi[i + 1]) {

double h = xi[i + 1] - xi[i];

double a = yi[i];

double b = (yi[i + 1] - yi[i]) / h;

double c = (yi[i + 1] - yi[i] - b \* h) / (h \* h);

sum = a + b \* (x - xi[i]) + c \* (x - xi[i]) \* (x - xi[i]);

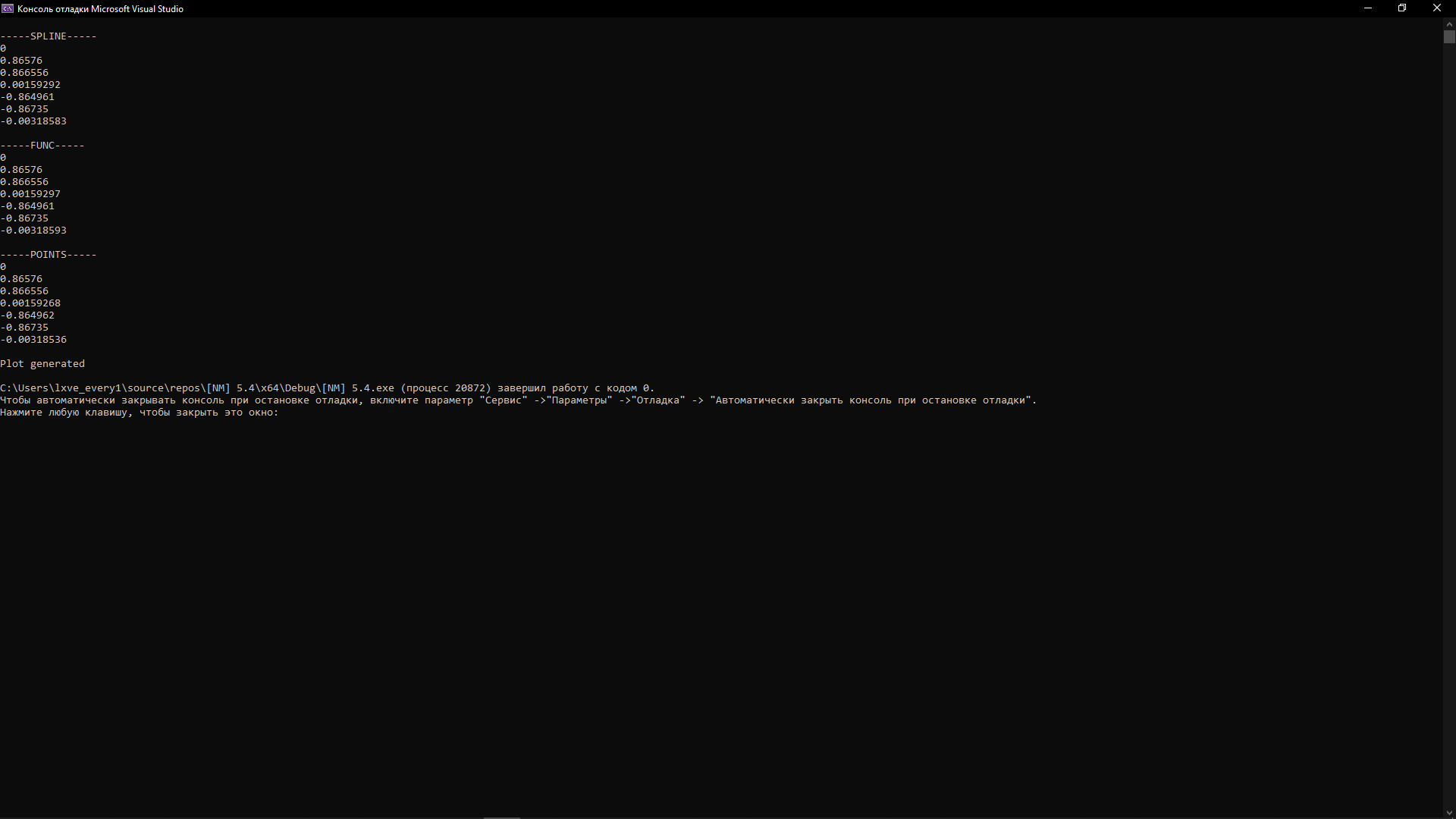
break;

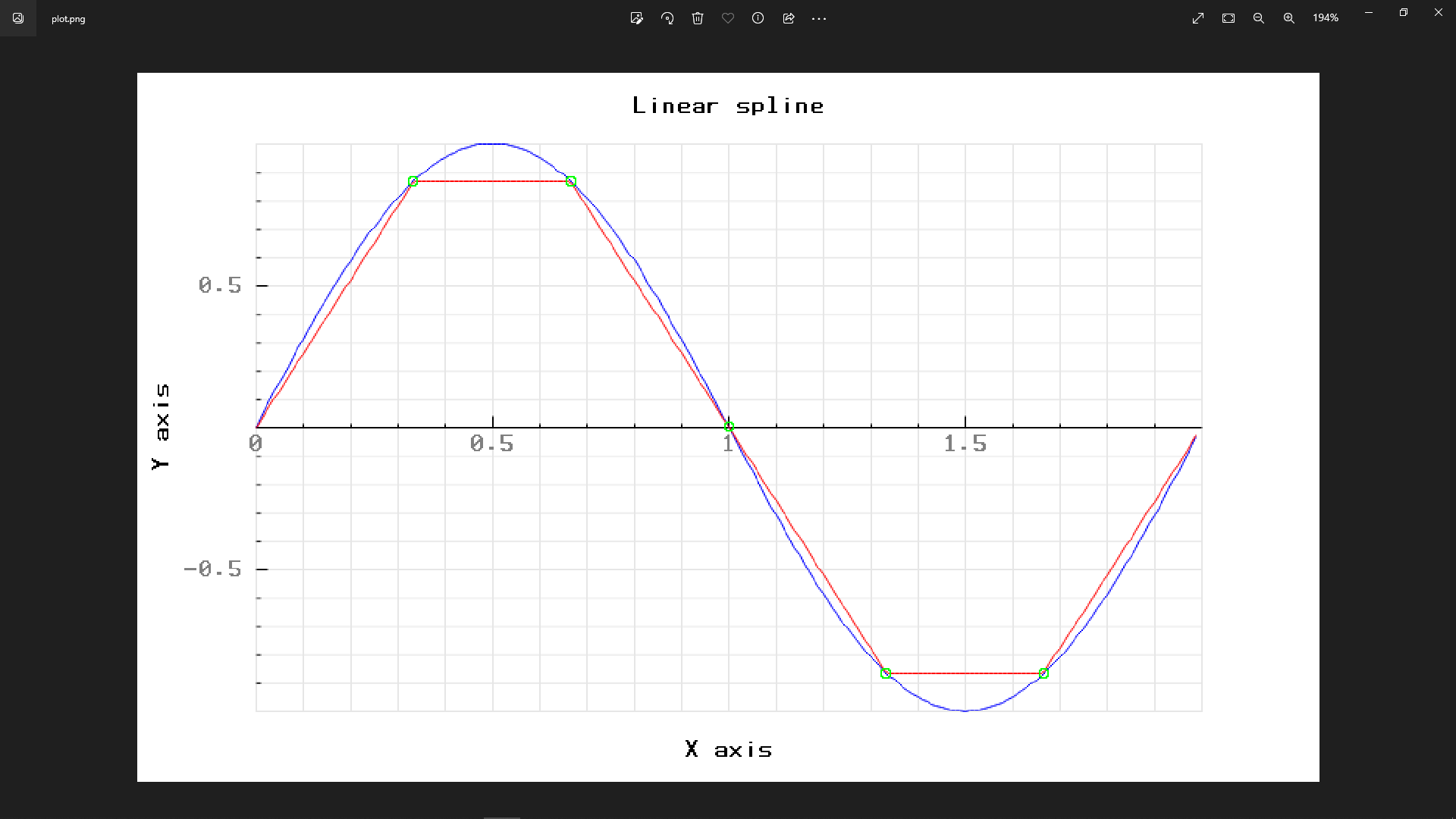
}

}

return sum;

}





Линейный сплайн оказался угловатым, но без проблем попал в указанные точки.

5) Кубический интерполяционный сплайн

Алгоритм программы:

// функция для вычисления значения кубического сплайна в заданной точке по равномерной сетке

double cubic\_poly(double x, const vector<double>& x\_values, const vector<double>& y\_values) {

int n = x\_values.size() - 1;

int i = 0;

while (i < n - 1 && x\_values[i] < x) {

i++;

}

if (i == 0) {

return y\_values[0] + (y\_values[1] - y\_values[0]) / (x\_values[1] - x\_values[0]) \* (x - x\_values[0]);

}

else if (i == n) {

return y\_values[n] + (y\_values[n] - y\_values[n - 1]) / (x\_values[n] - x\_values[n - 1]) \* (x - x\_values[n]);

}

else {

double h = x\_values[i] - x\_values[i - 1];

double a = (y\_values[i] - y\_values[i - 1]) / h;

double b = (y\_values[i + 1] - y\_values[i]) / h;

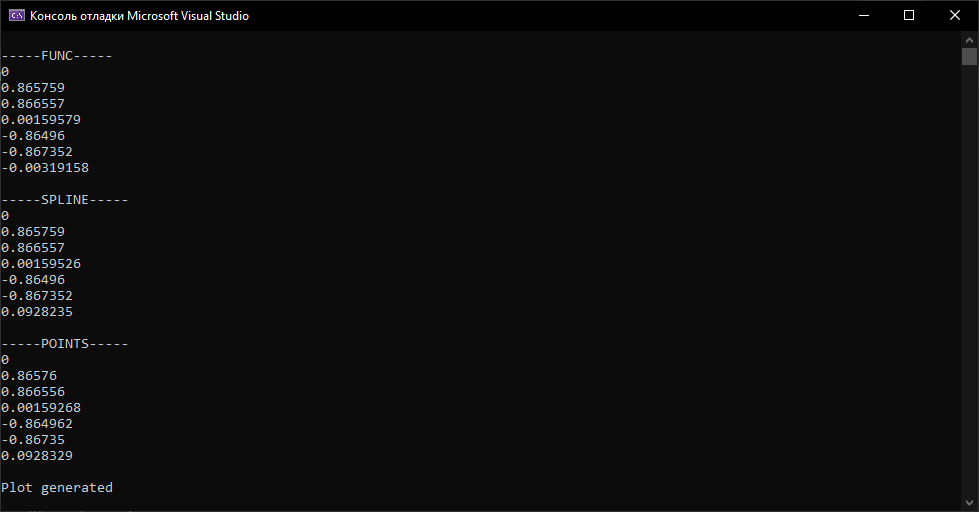
double c = (b - a) / (2 \* h);

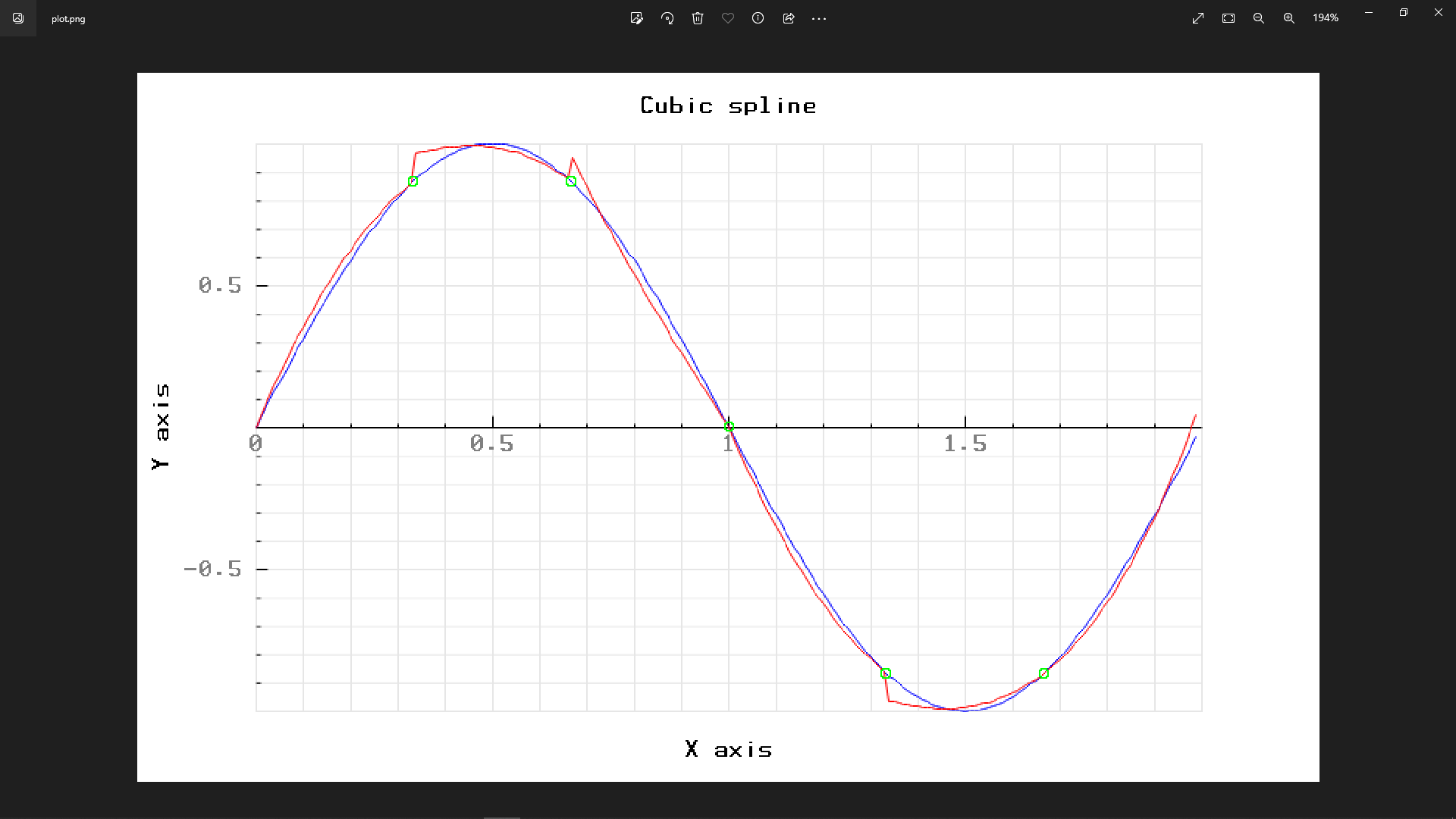
double d = a + h \* c;

return y\_values[i] + (x - x\_values[i]) \* (d + (x - x\_values[i]) \* (c + (x - x\_values[i]) \* b));

}

}





В моем случае точки абсолютно правильно совпали. Но в тоже время в трех узлах имеются дальнейшие отклонения от функции.

6) Аппроксимация

// Функция для построения линейной аппроксимирующей функции

void linear\_approximation(vector<double> x, vector<double> y, double& a, double& b, double& mse) {

int n = x.size();

double sx = 0, sy = 0, sxy = 0, sx2 = 0;

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

sx += x[i];

sy += y[i];

sxy += x[i] \* y[i];

sx2 += x[i] \* x[i];

}

double det = n \* sx2 - sx \* sx;

a = (sy \* sx2 - sx \* sxy) / det;

b = (n \* sxy - sx \* sy) / det;

// Вычисление среднеквадратической ошибки

mse = 0;

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

double y\_pred = a + b \* x[i];

mse += (y[i] - y\_pred) \* (y[i] - y\_pred);

}

mse = sqrt(mse / n);

}

// Функция для построения аппроксимирующей функции по точкам

void point\_approximation(vector<double> x, vector<double> y, double& a, double& b, double& mse) {

int n = x.size();

double sum\_x = 0, sum\_y = 0, sum\_xy = 0, sum\_x2 = 0;

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

sum\_x += log(x[i]);

sum\_y += log(y[i]);

sum\_xy += log(x[i]) \* log(y[i]);

sum\_x2 += log(x[i]) \* log(x[i]);

}

b = (n \* sum\_xy - sum\_x \* sum\_y) / (n \* sum\_x2 - sum\_x \* sum\_x);

a = exp((sum\_y - b \* sum\_x) / n);

// Вычисление среднеквадратической ошибки

mse = 0;

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

double y\_pred = a \* pow(x[i], b);

mse += (y[i] - y\_pred) \* (y[i] - y\_pred);

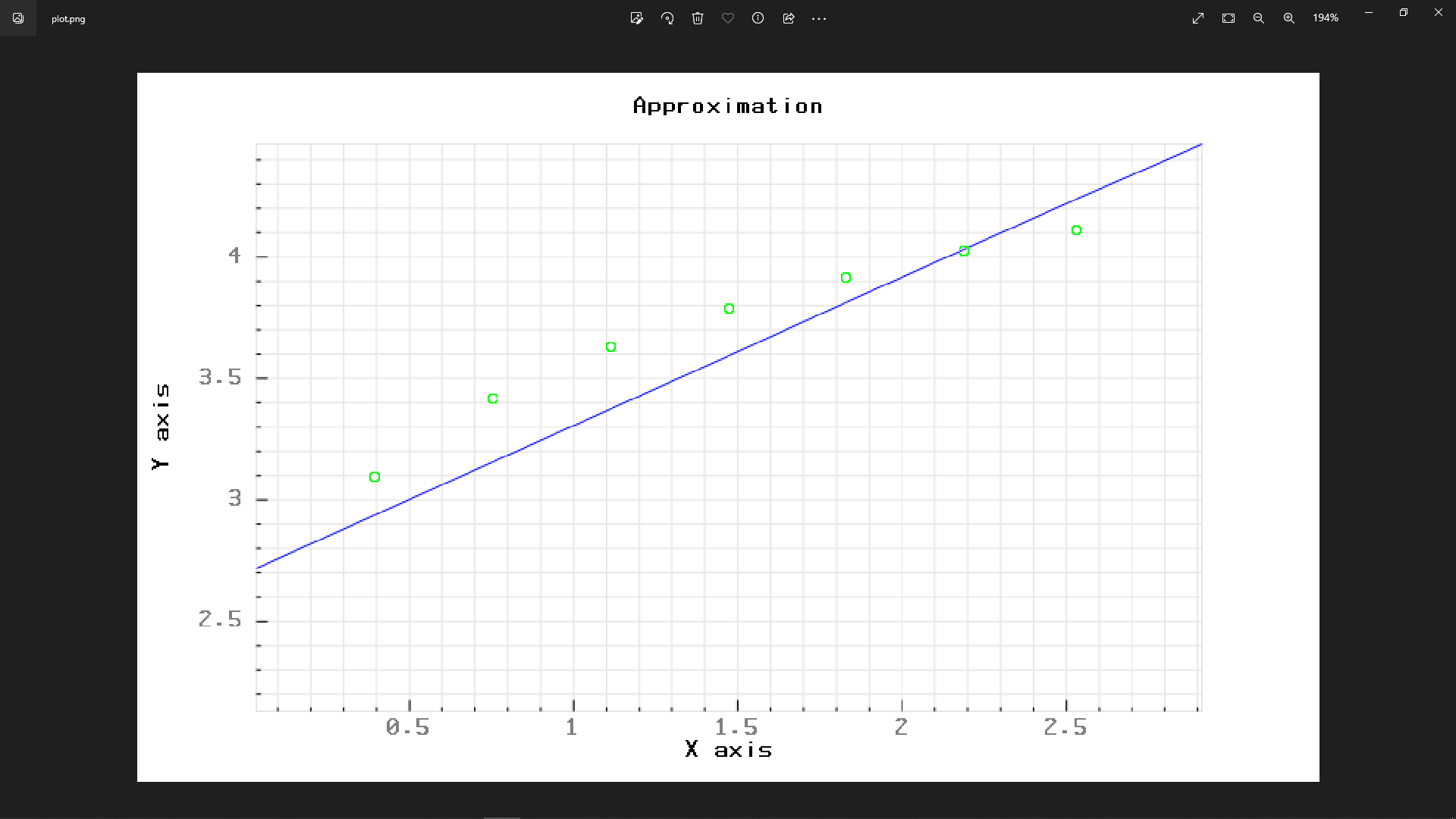
}

mse = sqrt(mse / n);

}

Результат:





5.

Таблица и выводы для заданий 1, 2:

|  |  |  |
| --- | --- | --- |
| x | Лагранж | Ньютон |
| 0 | 1.08534734 | 1.08534734 |
| 1 | -2.24251862 | -2.24251862 |
| 2 | -2.9076354 | -2.9076354 |
| 3 | -3.14539743 | -3.14539743 |
| 4 | -4.08101295 | -4.08101295 |
| 5 | -5.72950403 | -5.72950403 |
| 6 | -6.99570652 | -6.99570652 |

оба метода показали одинаковый результат

Таблица и выводы для заданий 3-4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | Лагранж 3 | Линейный | Кубический | Функция |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0.80597 | 0.779184 | 0.818107 | 0.808736 |
| 2 | 0.953552 | 0.866397 | 0.936426 | 0.951351 |
| 3 | 0.308285 | 0.261082 | 0.263848 | 0.31038 |
| 4 | -0.58359 | -0.51834 | -0.622023 | -0.586238 |
| 5 | -1.00471 | -0.866156 | -0.98647 | -0.999997 |
| 6 | -0.577005 | -0.521684 | -0.619525 | -0.590102 |

Полином Лагранжа третьего порядка и кубический полином показали на порядок более точные результаты, по сравнению с линейным полиномом. И все же полином Лагранжа является самым точным среди этих методов. На втором месте по точности – кубический полином, на третьем – линейный.

Программы:  
Задание 1.

#include <iostream>

#include <cmath>

#include <pbPlots.cpp>

#include <supportLib.cpp>

#include <iomanip>

using namespace std;

double LagrangeBasis(double x, const vector<double>& nodes, int i)

{

double result = 1;

for (int j = 0; j < nodes.size(); j++)

{

if (j != i)

{

result \*= (x - nodes[j]) / (nodes[i] - nodes[j]);

}

}

return result;

}

double LagrangeInterpolation(double x, const vector<double>& nodes, const vector<double>& values)

{

double result = 0;

for (int i = 0; i < nodes.size(); i++)

{

result += values[i] \* LagrangeBasis(x, nodes, i);

}

return result;

}

void plot(vector<double> x, vector<double> y, vector<double> nodes, vector<double> values) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* series1 = GetDefaultScatterPlotSeriesSettings();

series1->xs = &x;

series1->ys = &y;

series1->lineType = toVector(L"solid");

series1->pointType = toVector(L"circles");

series1->linearInterpolation = true;

series1->color = CreateRGBColor(0, 0, 1);

ScatterPlotSeries\* series2 = GetDefaultScatterPlotSeriesSettings();

series2->xs = &nodes;

series2->ys = &values;

series2->lineType = toVector(L"solid");

series2->pointType = toVector(L"circles");

series2->linearInterpolation = false;

series2->color = CreateRGBColor(1, 0, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Lagrange's polynom");

settings->xLabel = toVector(L"X axis");

settings->yLabel = toVector(L"Y axis");

settings->scatterPlotSeries->push\_back(series1);

settings->scatterPlotSeries->push\_back(series2);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>\* pngData = ConvertToPNG(imageReference->image);

WriteToFile(pngData, "plot.png");

cout << "\nPlot generated\n";

DeleteImage(imageReference->image);

}

int main()

{

cout << setprecision(9);

//Mine

//vector<double> nodes = { 0.083, 0.472, 1.347, 2.117, 2.947 };

//vector<double> values = { -2.132, -2.013, -1.613, -0.842, 2.973 };

//From example

vector<double> nodes = { 0.351, 0.867, 3.315, 5.013, 6.432 };

vector<double> values = { -0.572, -2.015, -3.342, -5.752, -6.911 };

// output the Lagrange basis polynomials

cout << "Lagrange basis polynomials:\n";

for (int i = 0; i < nodes.size(); i++)

{

cout << "L\_" << i << "(x) = ";

for (int j = 0; j < nodes.size(); j++)

{

if (j != i)

{

cout << "(x - " << nodes[j] << ") / (" << nodes[i] << " - " << nodes[j] << ")";

if (j < nodes.size() - 1 && j != i - 1)

{

cout << " \* ";

}

}

}

cout << endl;

}

// interpolate the values at some points

double x;

vector<double> x\_vector;

vector<double> y\_vector;

for (x = 0; x < 6.9; x += 1)

{

double y = LagrangeInterpolation(x, nodes, values);

cout << setw(3) << x << setw(13) << y << endl;

x\_vector.push\_back(x);

y\_vector.push\_back(y);

}

cout << "Denominators of the Lagrange basis polynomials:\n";

for (int i = 0; i < nodes.size(); i++)

{

double denominator = 1;

for (int j = 0; j < nodes.size(); j++)

{

if (j != i)

{

denominator \*= (nodes[i] - nodes[j]);

}

}

cout << "L\_" << i << " = " << 1.0 / denominator << endl;

}

plot(x\_vector, y\_vector, nodes, values);

return 0;

}

Задание 2:

#include <iostream>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

#include <iomanip>

using namespace std;

vector<vector<double>> divided\_differences\_table(const vector<double>& nodes, const vector<double>& values) {

int n = nodes.size();

vector<vector<double>> f(n, vector<double>(n));

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

f[i][0] = values[i];

}

for (int j = 1; j < n; ++j) {

for (int i = j; i < n; ++i) {

f[i][j] = (f[i][j - 1] - f[i - 1][j - 1]) / (nodes[i] - nodes[i - j]);

}

}

return f;

}

double newton\_interpolation\_polynomial(const vector<double>& nodes, const vector<double>& values, double x) {

vector<vector<double>> f = divided\_differences\_table(nodes, values);

double p = f[0][0];

double prod = 1.0;

int n = nodes.size();

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

prod \*= (x - nodes[i - 1]);

p += prod \* f[i][i];

}

return p;

}

void plot(vector<double> x, vector<double> y, vector<double> nodes, vector<double> values) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* series1 = GetDefaultScatterPlotSeriesSettings();

series1->xs = &x;

series1->ys = &y;

series1->lineType = toVector(L"solid");

series1->color = CreateRGBColor(0, 0, 1);

ScatterPlotSeries\* series2 = GetDefaultScatterPlotSeriesSettings();

series2->xs = &nodes;

series2->ys = &values;

series2->lineType = toVector(L"solid");

series2->pointType = toVector(L"circles");

series2->linearInterpolation = false;

series2->color = CreateRGBColor(1, 0, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Newton's polynom");

settings->xLabel = toVector(L"X axis");

settings->yLabel = toVector(L"Y axis");

settings->scatterPlotSeries->push\_back(series1);

settings->scatterPlotSeries->push\_back(series2);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>\* pngData = ConvertToPNG(imageReference->image);

WriteToFile(pngData, "plot.png");

cout << "\nPlot generated\n";

DeleteImage(imageReference->image);

}

int main() {

cout << setprecision(9);

//MINE

//vector<double> nodes = { 0.083, 0.472, 1.347, 2.117, 2.947 };

//vector<double> values = { -2.132, -2.013, -1.613, -0.842, 2.973 };

//From example

vector<double> nodes = { 0.351, 0.867, 3.315, 5.013, 6.432 };

vector<double> values = { -0.572, -2.015, -3.342, -5.752, -6.911 };

vector<double> x\_vector;

vector<double> y\_vector;

// Plot Newton interpolation polynomial

for (double x = 0; x < 6.9; x += 1) {

double y = newton\_interpolation\_polynomial(nodes, values, x);

cout << setw(3) << x << setw(13) << y << endl;

x\_vector.push\_back(x);

y\_vector.push\_back(y);

}

// Print divided differences table

vector<vector<double>> f = divided\_differences\_table(nodes, values);

int n = nodes.size();

cout << "Divided differences table:\n";

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

cout << nodes[i] << " ";

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

cout << f[i][j] << " ";

}

cout << "\n";

}

plot(x\_vector, y\_vector, nodes, values);

return 0;

}

Задание 3.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// функция для вычисления значения функции sin(3.14 \* x)

double f(double x) {

return sin(3.14 \* x);

}

// функция для вычисления значения многочлена Лагранжа порядка 3 в заданной точке по равномерной сетке

double lagrange\_poly(double x, const vector<double>& x\_values, const vector<double>& y\_values) {

double result = 0.0;

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

double term = y\_values[i];

for (size\_t j = 0; j < x\_values.size(); ++j) {

if (j != i) {

term \*= (x - x\_values[j]) / (x\_values[i] - x\_values[j]);

}

}

result += term;

}

return result;

}

// функция для построения графика

void plot(vector<double> x, vector<double> y, vector<double> x\_spline, vector<double> y\_spline, vector<double> x\_points, vector<double> y\_points) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* series1 = GetDefaultScatterPlotSeriesSettings();

series1->xs = &x;

series1->ys = &y;

series1->lineType = toVector(L"solid");

series1->color = CreateRGBColor(0, 0, 1);

ScatterPlotSeries\* series2 = GetDefaultScatterPlotSeriesSettings();

series2->xs = &x\_spline;

series2->ys = &y\_spline;

series2->lineType = toVector(L"solid");

series2->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* series3 = GetDefaultScatterPlotSeriesSettings();

series3->xs = &x\_points;

series3->ys = &y\_points;

series3->pointType = toVector(L"circles");

series3->linearInterpolation = false;

series3->color = CreateRGBColor(0, 1, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Lagrange 3rd order spline");

settings->xLabel = toVector(L"X axis");

settings->yLabel = toVector(L"Y axis");

settings->scatterPlotSeries->push\_back(series1);

settings->scatterPlotSeries->push\_back(series2);

settings->scatterPlotSeries->push\_back(series3);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>\* pngData = ConvertToPNG(imageReference->image);

WriteToFile(pngData, "plot.png");

cout << "\nPlot generated\n";

DeleteImage(imageReference->image);

}

int main() {

const double n = 5, m = 8, a = 13, h = 0.33333333, c = 0, d = 2, N = 7;

// задаем точки

vector<double> xi;

for (double i = c; i <= d; i += h) {

xi.push\_back(i);

}

vector<double> yi;

for (double x : xi) {

yi.push\_back(f(x));

}

// задаем значения для векторов x и y

vector<double> x, y, x\_spline, y\_spline;

double step = 0.3;

for (double i = c; i <= d; i += step) {

x.push\_back(i);

y.push\_back(f(i));

x\_spline.push\_back(i);

y\_spline.push\_back(lagrange\_poly(i, xi, yi));

}

vector<double> x\_points, y\_points;

for (double i = c; i <= d; i += h) {

x\_points.push\_back(i);

y\_points.push\_back(lagrange\_poly(i, xi, yi));

}

cout << "\n-----SPLINE-----\n";

for (int i = 0; i < x\_spline.size(); i++) {

cout << y\_spline[i] << "\n";

}

cout << "\n-----FUNC-----\n";

for (int i = 0; i < x.size(); i++) {

cout << y[i] << "\n";

}

cout << "\n-----POINTS-----\n";

for (int i = 0; i < x\_points.size(); i++) {

cout << y\_points[i] << "\n";

}

// строим графики

plot(x, y, x\_spline, y\_spline, x\_points, y\_points);

return 0;

}

Задание 4.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// функция для вычисления значения функции sin(3.14 \* x)

double f(double x) {

return sin(3.14 \* x);

}

double linear\_spline(double x, const vector<double>& xi, const vector<double>& yi) {

int n = xi.size();

double sum = 0;

for (int i = 0; i < n - 1; i++) {

if (x >= xi[i] && x <= xi[i + 1]) {

double h = xi[i + 1] - xi[i];

double a = yi[i];

double b = (yi[i + 1] - yi[i]) / h;

double c = (yi[i + 1] - yi[i] - b \* h) / (h \* h);

sum = a + b \* (x - xi[i]) + c \* (x - xi[i]) \* (x - xi[i]);

break;

}

}

return sum;

}

// функция для построения графика

void plot(vector<double> x, vector<double> y, vector<double> x\_spline, vector<double> y\_spline, vector<double> x\_points, vector<double> y\_points) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* series1 = GetDefaultScatterPlotSeriesSettings();

series1->xs = &x;

series1->ys = &y;

series1->lineType = toVector(L"solid");

series1->color = CreateRGBColor(0, 0, 1);

ScatterPlotSeries\* series2 = GetDefaultScatterPlotSeriesSettings();

series2->xs = &x\_spline;

series2->ys = &y\_spline;

series2->lineType = toVector(L"solid");

series2->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* series3 = GetDefaultScatterPlotSeriesSettings();

series3->xs = &x\_points;

series3->ys = &y\_points;

series3->pointType = toVector(L"circles");

series3->linearInterpolation = false;

series3->color = CreateRGBColor(0, 1, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Linear spline");

settings->xLabel = toVector(L"X axis");

settings->yLabel = toVector(L"Y axis");

settings->scatterPlotSeries->push\_back(series1);

settings->scatterPlotSeries->push\_back(series2);

settings->scatterPlotSeries->push\_back(series3);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>\* pngData = ConvertToPNG(imageReference->image);

WriteToFile(pngData, "plot.png");

cout << "\nPlot generated\n";

DeleteImage(imageReference->image);

}

int main() {

const double n = 5, m = 8, a = 13, h = 0.33333333, c = 0, d = 2, N = 7;

// задаем точки

vector<double> xi;

for (double i = c; i <= d; i += h) {

xi.push\_back(i);

}

vector<double> yi;

for (double x : xi) {

yi.push\_back(f(x));

}

// задаем значения для векторов x и y

vector<double> x, y, x\_spline, y\_spline;

double step = 0.3;

for (double i = c; i <= d; i += step) {

x.push\_back(i);

y.push\_back(f(i));

x\_spline.push\_back(i);

y\_spline.push\_back(linear\_spline(i, xi, yi));

}

vector<double> x\_points, y\_points;

for (double i = c; i <= d; i += h) {

x\_points.push\_back(i);

y\_points.push\_back(linear\_spline(i, xi, yi));

}

cout << "\n-----SPLINE-----\n";

for (int i = 0; i < x\_spline.size(); i++) {

cout << y\_spline[i] << "\n";

}

cout << "\n-----FUNC-----\n";

for (int i = 0; i < x.size(); i++) {

cout << y[i] << "\n";

}

cout << "\n-----POINTS-----\n";

for (int i = 0; i < x\_points.size(); i++) {

cout << y\_points[i] << "\n";

}

// строим графики

plot(x, y, x\_spline, y\_spline, x\_points, y\_points);

return 0;

}

Задание 5.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// функция для вычисления значения функции sin(3.14 \* x)

double f(double x) {

return sin(3.14 \* x);

}

// функция для вычисления значения кубического сплайна в заданной точке по равномерной сетке

double cubic\_poly(double x, const vector<double>& x\_values, const vector<double>& y\_values) {

int n = x\_values.size() - 1;

int i = 0;

while (i < n - 1 && x\_values[i] < x) {

i++;

}

if (i == 0) {

return y\_values[0] + (y\_values[1] - y\_values[0]) / (x\_values[1] - x\_values[0]) \* (x - x\_values[0]);

}

else if (i == n) {

return y\_values[n] + (y\_values[n] - y\_values[n - 1]) / (x\_values[n] - x\_values[n - 1]) \* (x - x\_values[n]);

}

else {

double h = x\_values[i] - x\_values[i - 1];

double a = (y\_values[i] - y\_values[i - 1]) / h;

double b = (y\_values[i + 1] - y\_values[i]) / h;

double c = (b - a) / (2 \* h);

double d = a + h \* c;

return y\_values[i] + (x - x\_values[i]) \* (d + (x - x\_values[i]) \* (c + (x - x\_values[i]) \* b));

}

}

// функция для построения графика

void plot(vector<double> x, vector<double> y, vector<double> x\_spline, vector<double> y\_spline, vector<double> x\_points, vector<double> y\_points) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* series1 = GetDefaultScatterPlotSeriesSettings();

series1->xs = &x;

series1->ys = &y;

series1->lineType = toVector(L"solid");

series1->color = CreateRGBColor(0, 0, 1);

ScatterPlotSeries\* series2 = GetDefaultScatterPlotSeriesSettings();

series2->xs = &x\_spline;

series2->ys = &y\_spline;

series2->lineType = toVector(L"solid");

series2->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* series3 = GetDefaultScatterPlotSeriesSettings();

series3->xs = &x\_points;

series3->ys = &y\_points;

series3->pointType = toVector(L"circles");

series3->linearInterpolation = false;

series3->color = CreateRGBColor(0, 1, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Cubic spline");

settings->xLabel = toVector(L"X axis");

settings->yLabel = toVector(L"Y axis");

settings->scatterPlotSeries->push\_back(series1);

settings->scatterPlotSeries->push\_back(series2);

settings->scatterPlotSeries->push\_back(series3);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>\* pngData = ConvertToPNG(imageReference->image);

WriteToFile(pngData, "plot.png");

cout << "\nPlot generated\n";

DeleteImage(imageReference->image);

}

int main() {

const double n = 5, m = 8, a = 13, h = 0.33333333, c = 0, d = 2, N = 7;

// задаем точки

vector<double> xi;

for (double i = c; i <= d; i += h) {

xi.push\_back(i);

}

vector<double> yi;

for (double x : xi) {

yi.push\_back(f(x));

}

// задаем значения для векторов x и y

vector<double> x, y, x\_spline, y\_spline;

double step = 0.3;

for (double i = c; i <= d; i += step) {

x.push\_back(i);

y.push\_back(f(i));

x\_spline.push\_back(i);

y\_spline.push\_back(cubic\_poly(i, xi, yi));

}

cout << "\n-----FUNC-----\n";

for (int i = 0; i < x.size(); i++) {

cout << y[i] << "\n";

}

cout << "\n-----SPLINE-----\n";

for (int i = 0; i < x\_spline.size(); i++) {

cout << y\_spline[i] << "\n";

}

vector<double> x\_points, y\_points;

for (double i = c; i <= d; i += h) {

x\_points.push\_back(i);

y\_points.push\_back(cubic\_poly(i, xi, yi)); // заменяем на quadratic\_spline

}

cout << "\n-----POINTS-----\n";

for (int i = 0; i < x\_points.size(); i++) {

cout << y\_points[i] << "\n";

}

// строим графики

plot(x, y, x\_spline, y\_spline, x\_points, y\_points);

return 0;

}

Задание 6.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// Функция для построения линейной аппроксимирующей функции

void linear\_approximation(vector<double> x, vector<double> y, double& a, double& b, double& mse) {

int n = x.size();

double sx = 0, sy = 0, sxy = 0, sx2 = 0;

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

sx += x[i];

sy += y[i];

sxy += x[i] \* y[i];

sx2 += x[i] \* x[i];

}

double det = n \* sx2 - sx \* sx;

a = (sy \* sx2 - sx \* sxy) / det;

b = (n \* sxy - sx \* sy) / det;

// Вычисление среднеквадратической ошибки

mse = 0;

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

double y\_pred = a + b \* x[i];

mse += (y[i] - y\_pred) \* (y[i] - y\_pred);

}

mse = sqrt(mse / n);

}

// Функция для построения аппроксимирующей функции по точкам

void point\_approximation(vector<double> x, vector<double> y, double& a, double& b, double& mse) {

int n = x.size();

double sum\_x = 0, sum\_y = 0, sum\_xy = 0, sum\_x2 = 0;

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

sum\_x += log(x[i]);

sum\_y += log(y[i]);

sum\_xy += log(x[i]) \* log(y[i]);

sum\_x2 += log(x[i]) \* log(x[i]);

}

b = (n \* sum\_xy - sum\_x \* sum\_y) / (n \* sum\_x2 - sum\_x \* sum\_x);

a = exp((sum\_y - b \* sum\_x) / n);

// Вычисление среднеквадратической ошибки

mse = 0;

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

double y\_pred = a \* pow(x[i], b);

mse += (y[i] - y\_pred) \* (y[i] - y\_pred);

}

mse = sqrt(mse / n);

}

// функция для построения графика

void plot(vector<double> x\_approximation, vector<double> y\_approximation, vector<double> x\_points, vector<double> y\_points) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* series1 = GetDefaultScatterPlotSeriesSettings();

series1->xs = &x\_approximation;

series1->ys = &y\_approximation;

series1->lineType = toVector(L"solid");

series1->color = CreateRGBColor(0, 0, 1);

ScatterPlotSeries\* series2 = GetDefaultScatterPlotSeriesSettings();

series2->xs = &x\_points;

series2->ys = &y\_points;

series2->pointType = toVector(L"circles");

series2->linearInterpolation = false;

series2->color = CreateRGBColor(0, 1, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Approximation");

settings->xLabel = toVector(L"X axis");

settings->yLabel = toVector(L"Y axis");

settings->scatterPlotSeries->push\_back(series1);

settings->scatterPlotSeries->push\_back(series2);

DrawScatterPlotFromSettings(imageReference, settings, NULL);

vector<double>\* pngData = ConvertToPNG(imageReference->image);

WriteToFile(pngData, "plot.png");

cout << "\nPlot generated\n";

DeleteImage(imageReference->image);

}

int main() {

vector<double> x = { 0.034, 0.394, 0.754, 1.114, 1.474, 1.833, 2.193, 2.533, 2.913 };

vector<double> y = { 2.156, 2.988, 3.377, 3.708, 3.802, 3.9, 4.067, 4.129, 4.171 };

// Линейная аппроксимация

int i;

double a, b, mse\_linear, y\_pred;

linear\_approximation(x, y, a, b, mse\_linear);// Аппроксимация по точкам

double c, d, mse\_point;

point\_approximation(x, y, c, d, mse\_point);

vector<double> x\_approximation, y\_approximation, x\_points, y\_points;

// Вывод результатов

cout << "Linear approximation:\n";

cout << "a = " << a << "\n";

cout << "b = " << b << "\n";

cout << "E = " << mse\_linear << "\n";

cout << "x\t y\t Y\n";

for (int i = 0; i < x.size(); i++) {

double y\_pred = a + b \* x[i];

cout << x[i] << "\t " << y[i] << "\t " << y\_pred << "\n";

y\_approximation.push\_back(y\_pred);

}

x\_approximation = x;

cout << "\n";

cout << "Point approximation:\n";

cout << "c = " << c << "\n";

cout << "d = " << d << "\n";

cout << "E = " << mse\_point << "\n";

cout << "x\t y\t Y\n";

for (i = 0; i < x.size(); i++) {

y\_pred = c \* pow(x[i], d);

cout << x[i] << "\t " << y[i] << "\t " << y\_pred << "\n";

y\_points.push\_back(y\_pred);

}

x\_points = x;

plot(x\_approximation, y\_approximation, x\_points, y\_points);

return 0;

}