**Липецкий государственный технический университет**

Факультет автоматизации и информатики

Кафедра автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА № *6*

по дисциплине «Численные методы»

«Численное дифференцирование,

интегрирование функций и решение дифференциальных

уравнений

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. Данное задание использует результаты лабораторной работы №5:

Задана функция (см. таблицу 1). Разбить отрезок [c, d] на m подотрезков, для каждого из которых построить интерполяционные многочлены Лагранжа

Ln порядков n 2,3,4 с равномерной сеткой в соответствии со значением заданной функции в узлах сетки. Построить на одном графике графики многочленов с отмеченными на них исходными точками и график заданной функции.

Найти точные (для исходной функции) и приближенные (для многочленов) значения производных в узлах сетки, где n 2,3,4. Вывести сравнительную таблицу.

2. Данное задание использует результаты лабораторной работы №5:

Задана функция (см. таблицу 1). Разбить отрезок [c, d] на m подотрезков, для каждого из которых построить интерполяционные

многочлены Ньютона N n порядков n = 2,3,4:

для четных вариантов – с использованием интерполирования

назад

с равномерной сеткой в соответствии со значением заданной функции в узлах

сетки. Построить на одном графике графики многочленов с отмеченными на

них исходными точками и график заданной функции.

Найти точные (для исходной функции) и приближенные (для многочленов) значения производных в узлах сетки, где n = 2,3,4. Вывести сравнительную таблицу.

3. Задана функция (см. таблицу 1). Найти интеграл, используя формулу Ньютона-Котоса: для четных вариантов – при n = 4,5.

Найти погрешности интегрирования. Построить на одном графике

графики многочленов Лагранжа заданной степени с отмеченными на них

исходными точками и график исходной функции.

Для всех методов сделать сводную таблицу и вывод.

4. Задана функция (см. таблицу 1). Найти интеграл, используя формулы левых, правых, средних прямоугольников, трапеций и Симпсона с точностью

10e-6.

Построить на одном графике графики многочленов Лагранжа заданной

степени с отмеченными на них исходными точками и график исходной

функции.

Для всех методов сделать сводную таблицу и вывод.

5. Задано дифференциальное уравнение первого порядка (см. таблицу 1) с начальным условием

y(0) = 1.

5.1. Решить задачу Коши для дифференциального уравнения

аналитически. Построить график интегральной кривой на отрезке [c, d].

5.2. Решить дифференциальное уравнение на отрезке [c, d] методами:

Эйлера, уточненным методом Эйлера, Рунге-Кутта 4 порядка, Рунге-Кутта 5

порядка, Адамса-Башфорта 4 порядка, Адамса-Моултона 4 порядка с

точностью до 10e-6

Построить на одном графике график точной интегральной кривой и

приближенных интегральных кривых, полученных заданными методами, на

отрезке [c, d].

Для всех методов сделать сводную таблицу и вывод.

6. Задано дифференциальное уравнение первого порядка (см. таблицу 2) с

начальным условием.

6.1. Решить задачу Коши для дифференциального уравнения

аналитически. Построить график интегральной кривой на отрезке [a, b].

6.2. Решить дифференциальное уравнение на отрезке [a, b] методами:

Эйлера, уточненным методом Эйлера, Рунге-Кутта 4 порядка, Рунге-Кутта 5

порядка, Адамса-Башфорта 4 порядка, Адамса-Моултона 4 порядка с

точностью до 10e-6 Построить на одном графике график точной интегральной кривой и приближенных интегральных кривых, полученных заданными методами, на отрезке [a, b].

Для всех методов сделать сводную таблицу и вывод.

3. Ход работы

1. Многочлен Лагранжа - это интерполяционный многочлен, используемый для аппроксимации функции на заданном интервале с помощью многочлена низкой степени. Многочлен Лагранжа является линейной комбинацией n+1 узловых точек и соответствующих коэффициентов Лагранжа.

Пусть дана некоторая функция f(x) и n+1 узловых точек (x0,y0), (x1,y1), ..., (xn,yn). Многочлен Лагранжа L(x) для этой функции на интервале [x0,xn] определяется следующим образом:

L(x) = y0L0(x) + y1L1(x) + ... + yn\*Ln(x),

где Li(x) - многочлены Лагранжа первой степени, определяемые формулой:

Li(x) = (x-x0)(x-x1)...(x-x(i-1))(x-x(i+1))...(x-xn) / ((xi-x0)(xi-x1)...(xi-x(i-1))(xi-x(i+1))...(xi-xn)).

Многочлен Лагранжа интерполирует функцию f(x) в узловых точках и при этом имеет степень не выше n, т.е. он является многочленом степени n или ниже.

Функция многочлена Лагранжа:

double lagrange\_poly(double x, const vector<double>& xi, const vector<double>& yi, int n) {

double L = 0;

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

double l = 1;

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

if (i != j) {

l \*= (x - xi[j]) / (xi[i] - xi[j]);

}

}

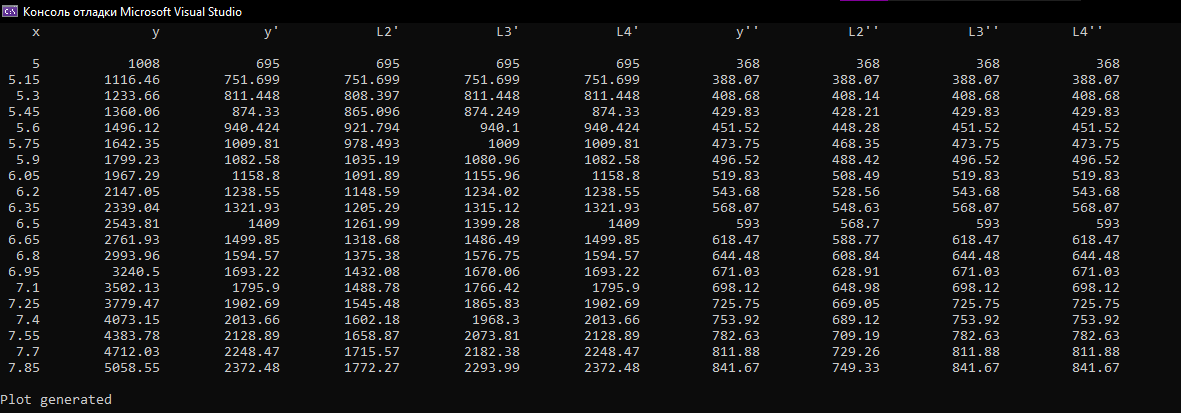
L += yi[i] \* l;

}

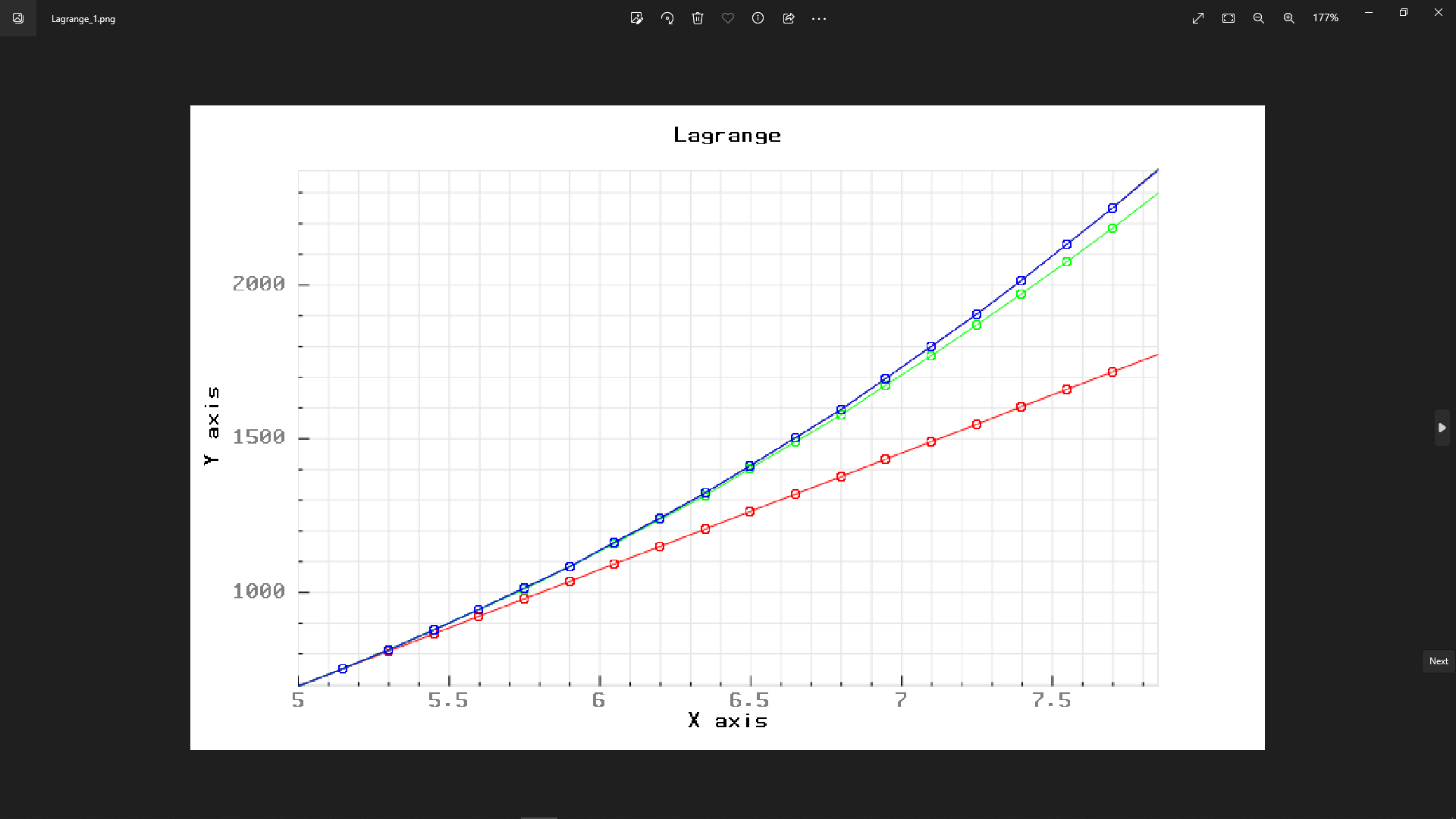
return L;

}

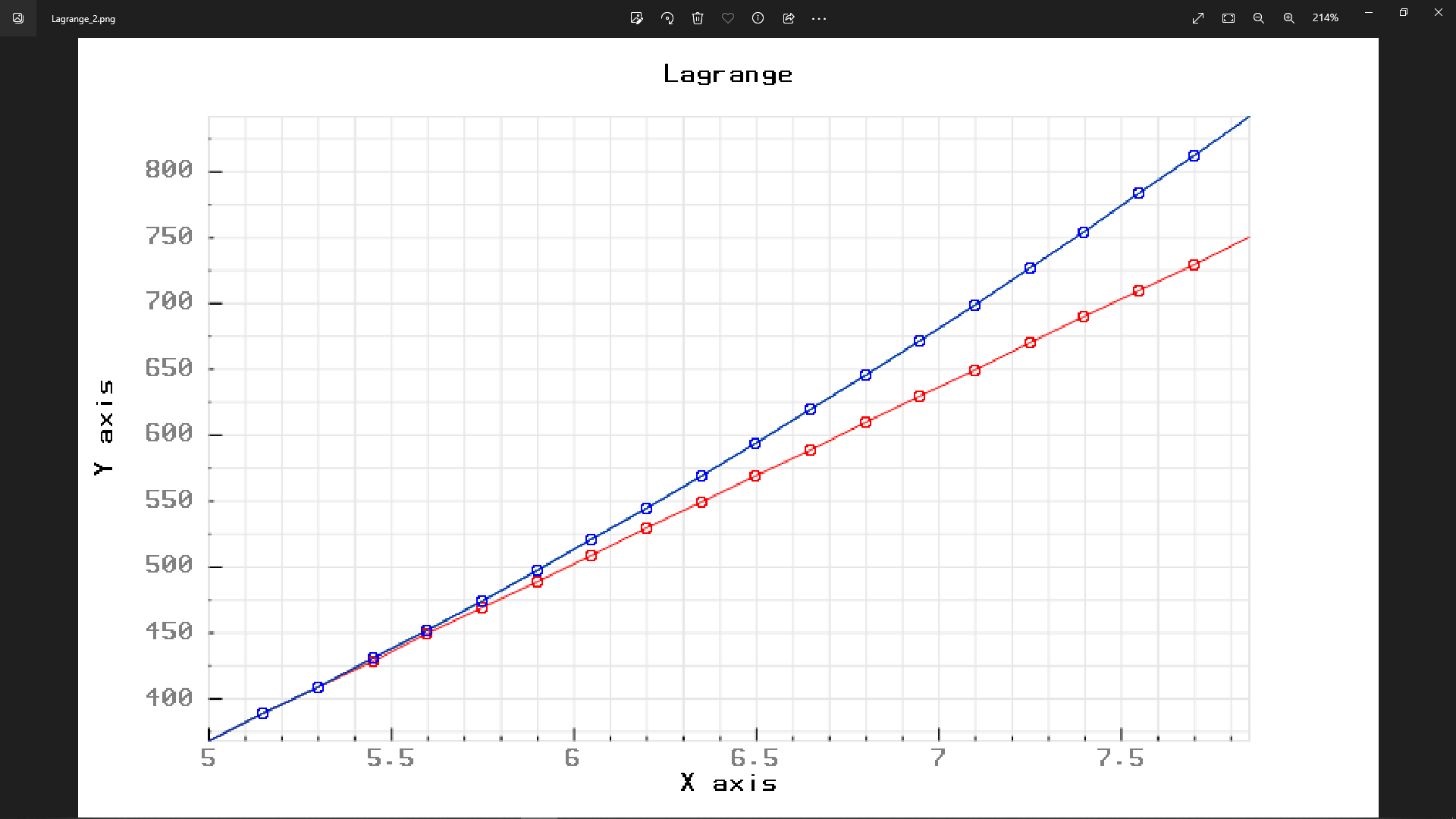
Результат программы:



Первая степень производной:



Вторая степень производной:



2. Интерполирование назад (backward interpolation) - это метод интерполяции, при котором на основе заданных значений функции y\_i в узлах сетки x\_i, где x\_i монотонно возрастают, находится значение функции y(x) в произвольной точке x на отрезке [x\_n, x\_0] где x\_n - последний узел сетки.

Основной принцип интерполяции назад заключается в построении интерполяционного многочлена Лагранжа, который проходит через последние узлы сетки и использует их значения для нахождения значения функции в произвольной точке на отрезке [x\_n, x\_0]. При этом для нахождения коэффициентов многочлена Лагранжа используются формулы, основанные на конечных разностях.

Интерполяция назад широко применяется в численных методах для решения дифференциальных уравнений с обратным временем, когда необходимо вычислить значения функции на предыдущих временных шагах по уже вычисленным значениям на более поздних временных шагах.

Функция интерполирования назад:

double newton\_poly(double x, const vector<double>& xi, const vector<double>& yi, int n) {

double res = yi[n];

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

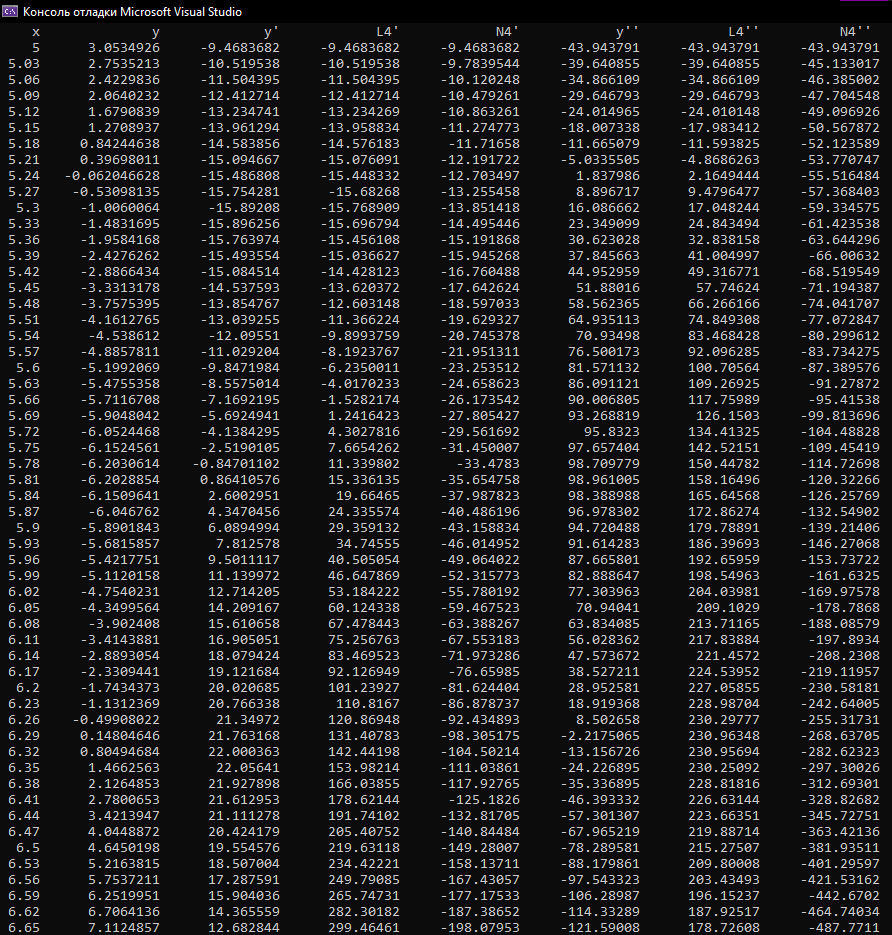
res = yi[i] + (x - xi[i]) \* res;

}

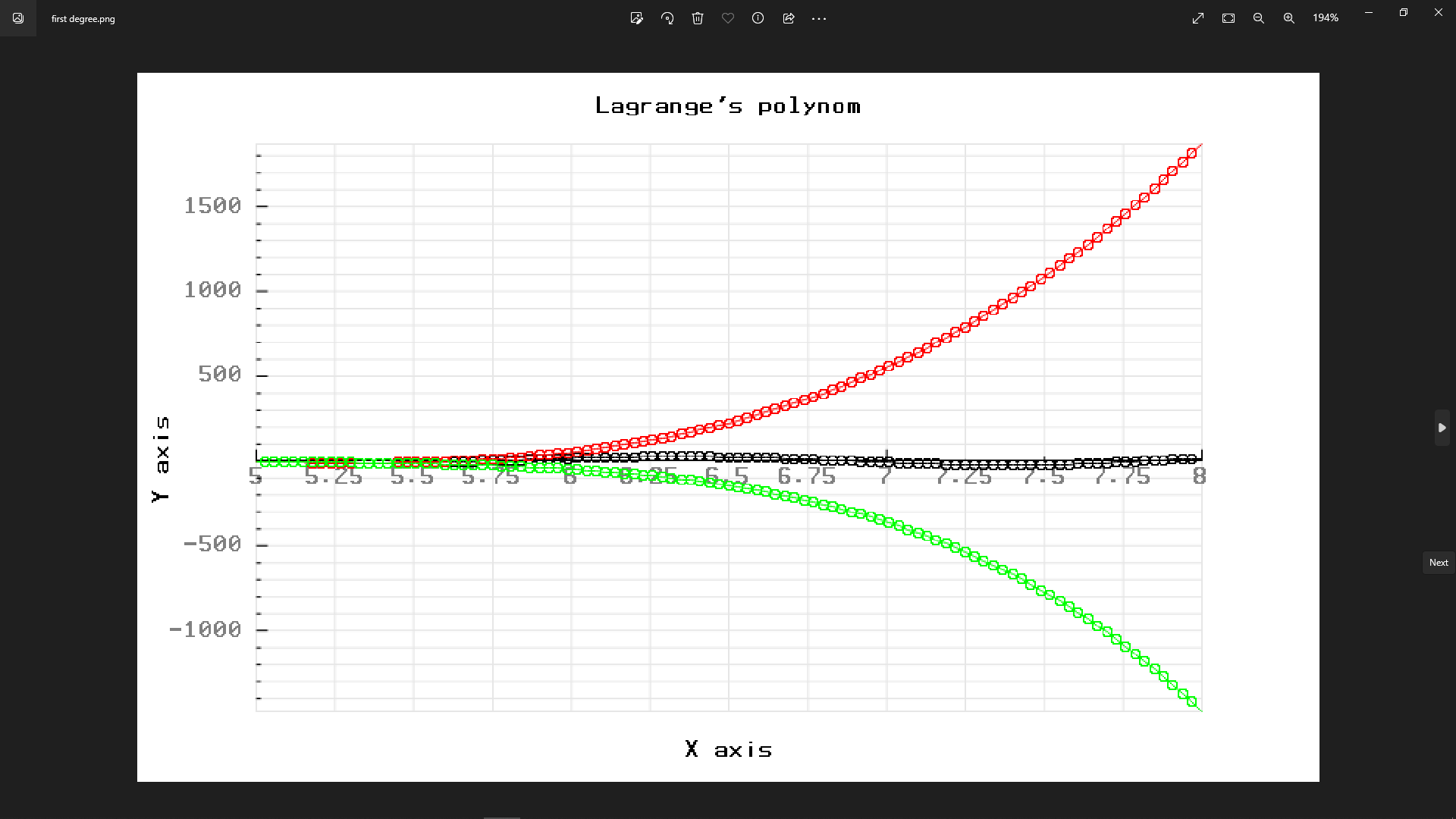
return res;

}

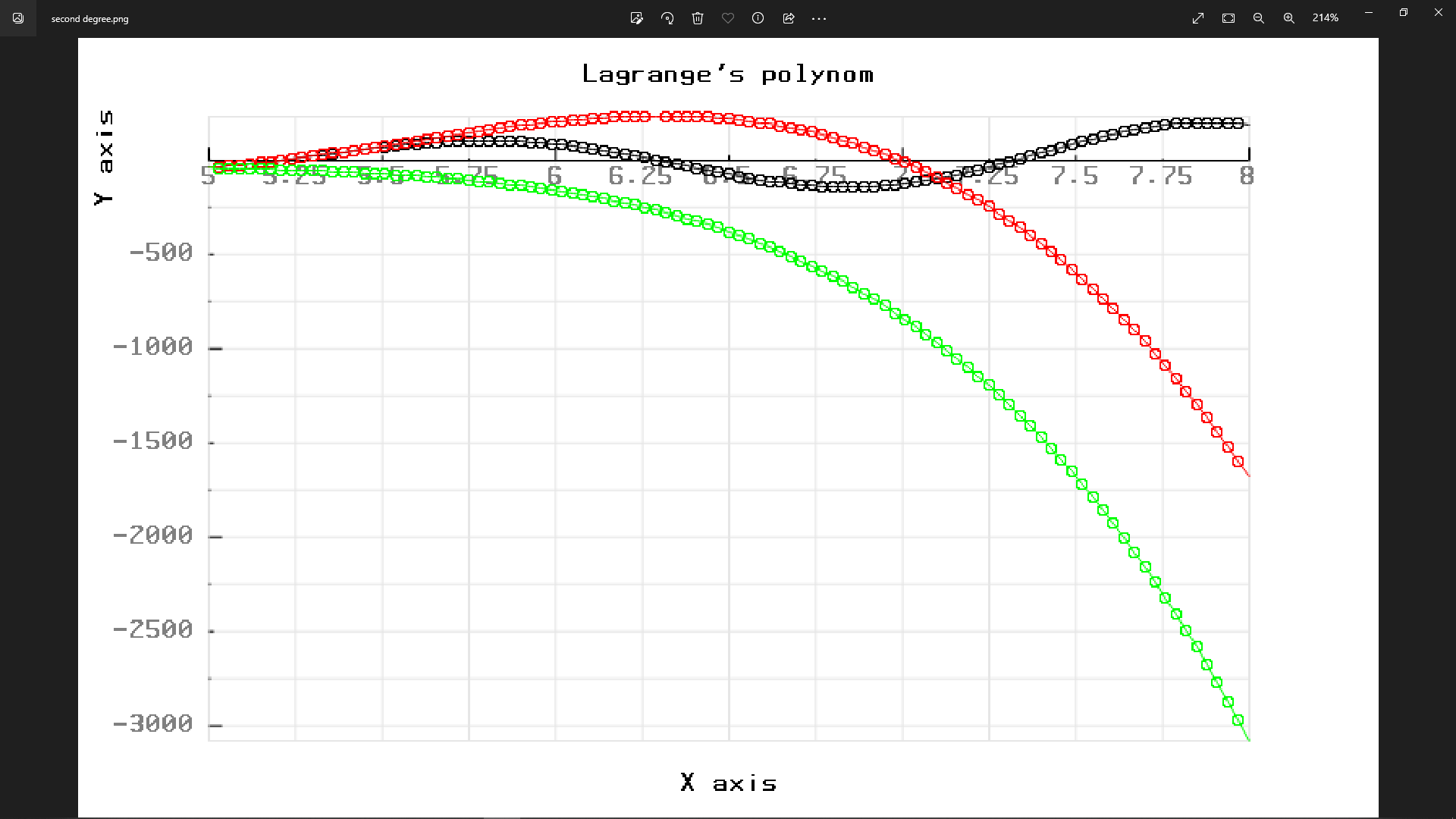
Результат программы:



Первая производная:



Вторая производная:



3.

Метод Ньютона-Котеса - это численный метод интегрирования функции, который основан на замене исходной функции на интерполяционный полином на заданном интервале интегрирования и на последующем вычислении интеграла от этого полинома.

Существует несколько формул Ньютона-Котеса различных порядков точности. Они различаются количеством точек, взятых для построения интерполяционного полинома. Наиболее часто используемые формулы - это формулы Ньютона-Котеса второго и пятого порядков точности.

Формула Ньютона-Котеса второго порядка точности использует три равноотстоящие точки на интервале интегрирования, чтобы построить интерполяционный полином второй степени. Формула Ньютона-Котеса пятого порядка точности использует шесть равноотстоящих точек на интервале интегрирования, чтобы построить интерполяционный полином пятой степени.

Метод:

// Вычисление интегралов методом Ньютона-Котеса

double I1 = 0, I2 = 0;

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

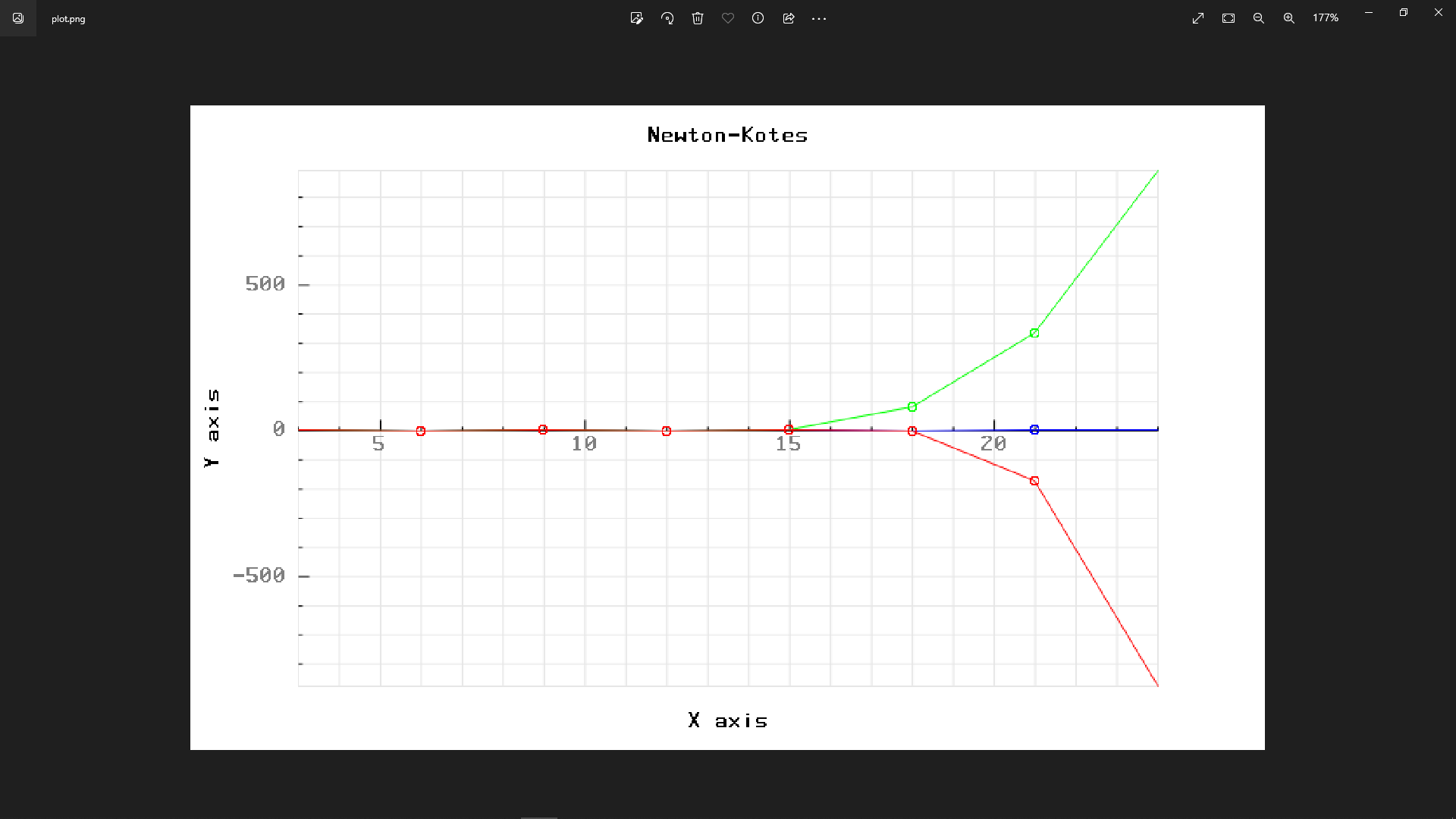
double x = xs[i];

double y = f(x, a, b, k);

I1 += L1[i] \* h;

I2 += L2[i] \* h;

}



4. Используемые функции:

double leftRectangleMethod(double h) {

double sum = 0;

double x = c;

while (x + h <= d) {

sum += function(x) \* function(x);

x += h;

}

return h \* sum;

}

double rightRectangleMethod(double h) {

double sum = 0;

double x = c + h;

while (x <= d) {

sum += function(x) \* function(x);

x += h;

}

return h \* sum;

}

double middleRectangleMethod(double h) {

double sum = 0;

double x = c + h / 2;

while (x < d) {

sum += function(x) \* function(x);

x += h;

}

return h \* sum;

}

double trapezoidMethod(double h) {

double sum = function(c) \* function(c) + function(d) \* function(d);

double x = c + h;

while (x < d) {

sum += 2 \* function(x) \* function(x);

x += h;

}

return h / 2 \* sum;

}

double simpsonMethod(double h) {

double sum = function(c) \* function(c) + function(d) \* function(d);

double x = c + h;

while (x < d) {

sum += 2 \* function(x) \* function(x - h / 2) + 4 \* function(x - h / 2) \* function(x - h / 2);

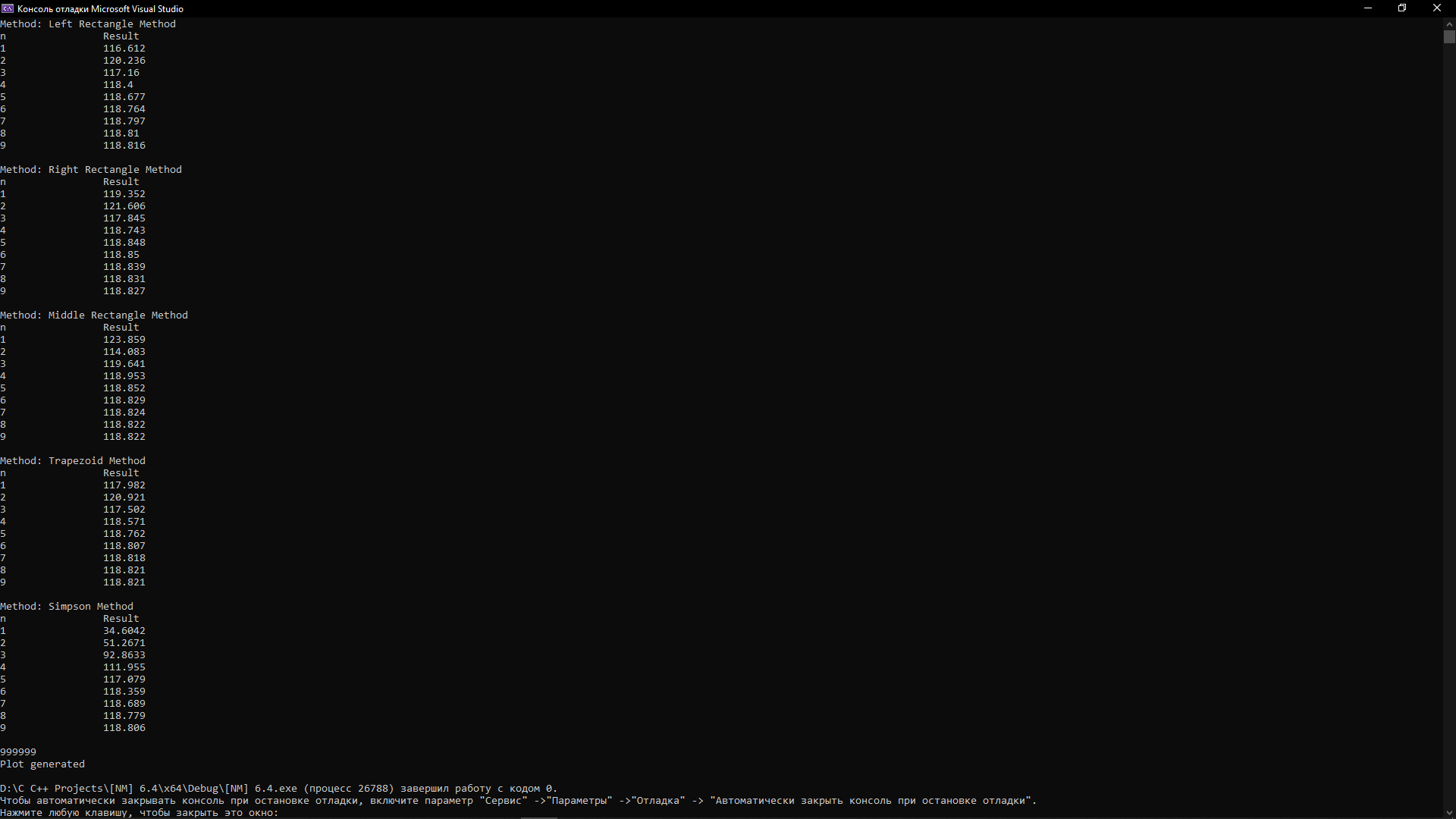
x += h;

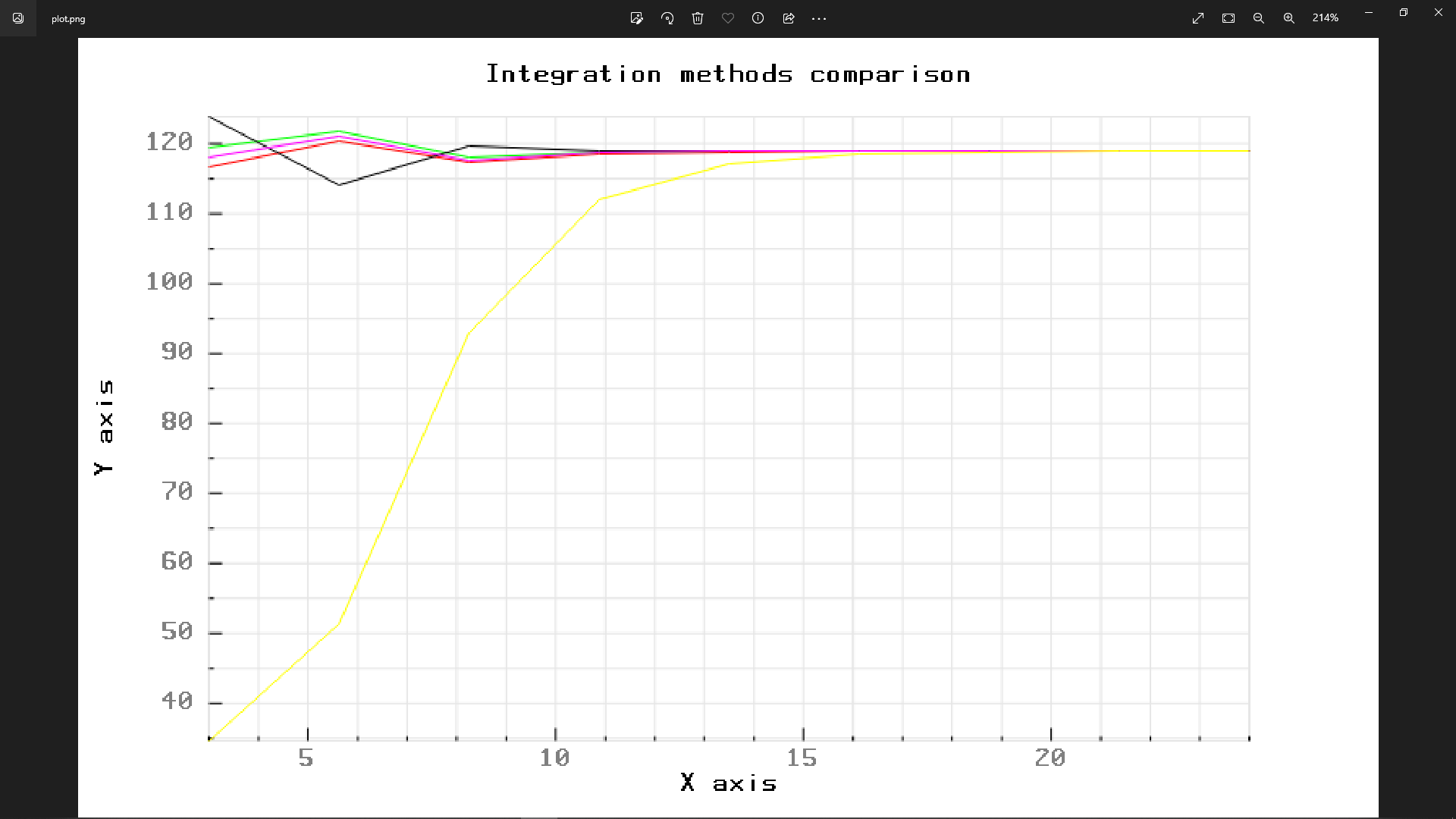
}

return h / 6 \* sum;

}

Результат программы:





5. Функции:

// Define the Euler method

void euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double yn1 = yn + h \* f(xn);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the improved Euler method

void improved\_euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn);

double k2 = f(xn + h);

double yn1 = yn + h \* (k1 + k2) / 2;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta 4th order method

void runge\_kutta\_4(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1;

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn);

double k2 = f(xn + h / 2);

double k3 = f(xn + h / 2);

double k4 = f(xn + h);

double yn1 = yn + h \* (k1 + 2 \* k2 + 2 \* k3 + k4) / 6;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta5th order method

void runge\_kutta\_5(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1;

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn);

double k2 = f(xn + h / 4);

double k3 = f(xn + h / 4);

double k4 = f(xn + h / 2);

double k5 = f(xn + 3 \* h / 4);

double k6 = f(xn + h);

double yn1 = yn + h \* (k1 \* 7 / 90 + k3 \* 32 / 90 + k4 \* 12 / 90 + k5 \* 32 / 90 + k6 \* 7 / 90);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Bashforth 4th order method

void adams\_bashforth\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Update x vector with initial values

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

y.push\_back(y\_prev[i]);

}

// Iterate using the Adams-Bashforth 4th order method

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

double yn = y[i - 1] + h \* (55 \* f(x[i]) - 59 \* f(x[i - 2]) + 37 \* f(x[i - 3]) - 9 \* f(x[i - 4])) / 24;

double xn = x[i - 1] + h;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Moulton 4th order method

void adams\_moulton\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Iterate using the Adams-Moulton 4th order method

for (int i = 3; i < m; i++) {

double yn = y[i - 1] + h \* (9 \* f(x[i]) + 19 \* f(x[i - 1]) - 5 \* f(x[i - 2]) + f(x[i - 3])) / 24;

double xn = x[i - 1] + h;

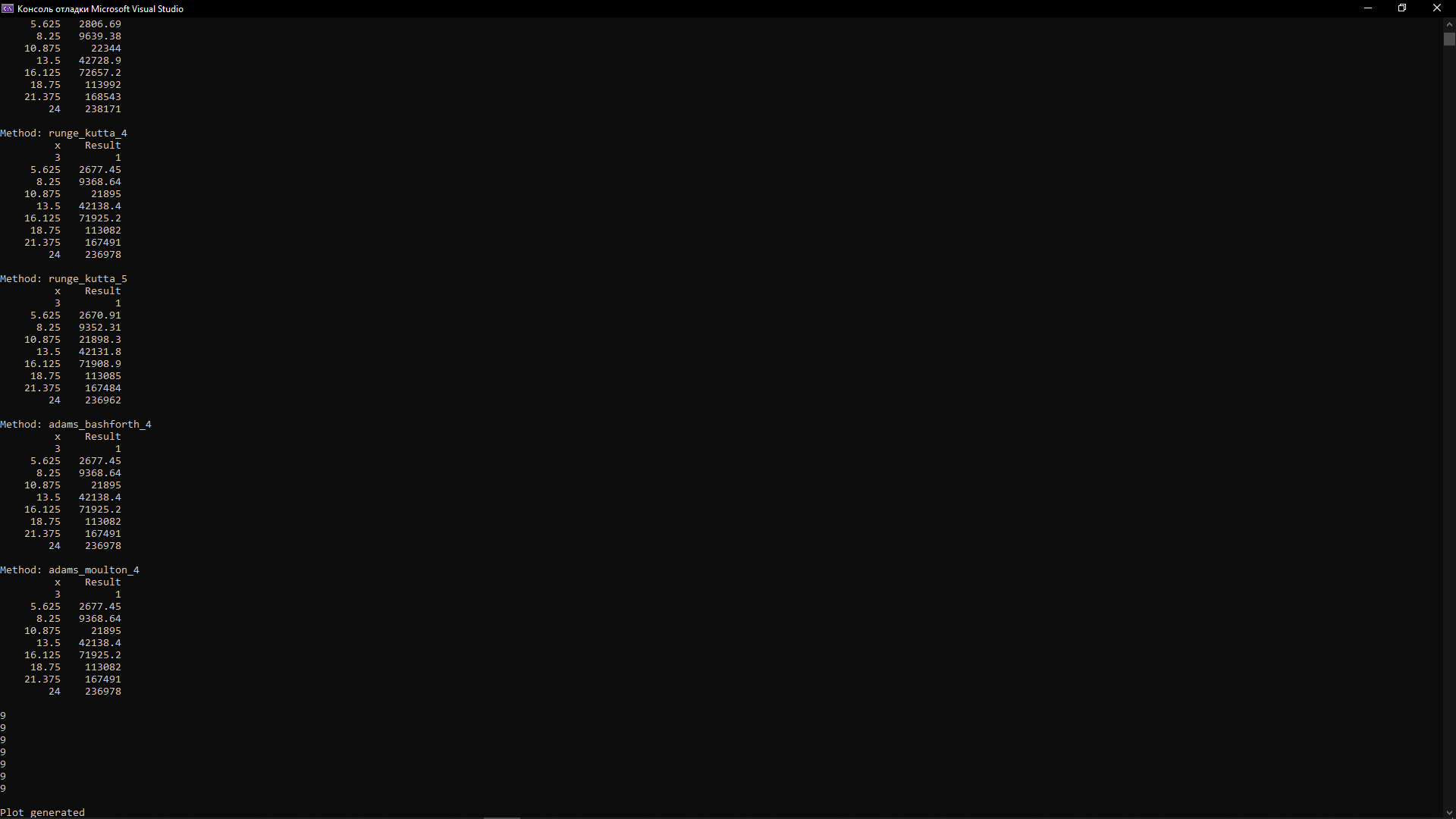
x.push\_back(xn);

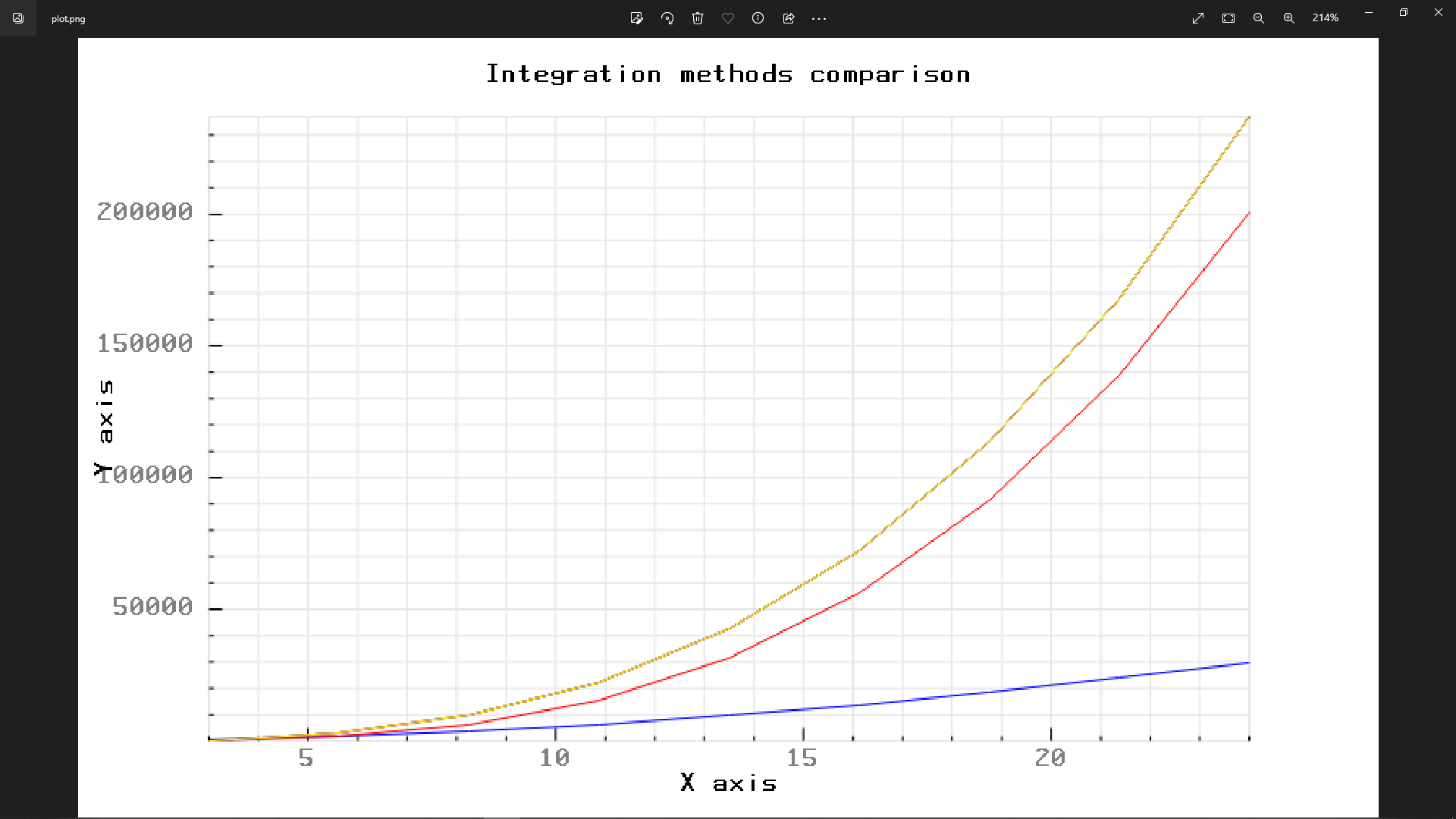
y.push\_back(yn);

}

}

Результат программы:





6. Функции:

// Define the Euler method

void euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double yn1 = yn + h \* f(xn, yn);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the improved Euler method

void improved\_euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn, yn);

double k2 = f(xn + h, yn + h \* k1);

double yn1 = yn + h \* (k1 + k2) / 2;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta 4th order method

void runge\_kutta\_4(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0;

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn, yn);

double k2 = f(xn + h / 2, yn + h \* k1 / 2);

double k3 = f(xn + h / 2, yn + h \* k2 / 2);

double k4 = f(xn + h, yn + h \* k3);

double yn1 = yn + h \* (k1 + 2 \* k2 + 2 \* k3 + k4) / 6;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta5th order method

void runge\_kutta\_5(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0;

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn, yn);

double k2 = f(xn + h / 4, yn + h \* k1 / 4);

double k3 = f(xn + h / 4, yn + h \* (k1 / 8 + k2 / 8));

double k4 = f(xn + h / 2, yn - h \* (k2 / 2) + h \* k3);

double k5 = f(xn + 3 \* h / 4, yn + h \* (k1 \* 3 / 16 + k4 \* 9 / 16));

double k6 = f(xn + h, yn + h \* (-k1 \* 3 / 7 + k2 \* 2 / 7 + k3 \* 12 / 7 - k4 \* 12 / 7 + k5 \* 8 / 7));

double yn1 = yn + h \* (k1 \* 7 / 90 + k3 \* 32 / 90 + k4 \* 12 / 90 + k5 \* 32 / 90 + k6 \* 7 / 90);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Bashforth 4th order method

void adams\_bashforth\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Update x vector with initial values

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

y.push\_back(y\_prev[i]);

}

// Iterate using the Adams-Bashforth 4th order method

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

double yn = y[i - 1] + h \* (55 \* f(x[i], y[i - 1]) - 59 \* f(x[i - 2], y[i - 2]) + 37 \* f(x[i - 3], y[i - 3]) - 9 \* f(x[i - 4], y[i - 4])) / 24;

double xn = x[i - 1] + h;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Moulton 4th order method

void adams\_moulton\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Iterate using the Adams-Moulton 4th order method

for (int i = 3; i < m; i++) {

double yn = y[i - 1] + h \* (9 \* f(x[i], y[i]) + 19 \* f(x[i - 1], y[i - 1]) - 5 \* f(x[i - 2], y[i - 2]) + f(x[i - 3], y[i - 3])) / 24;

double xn = x[i - 1] + h;

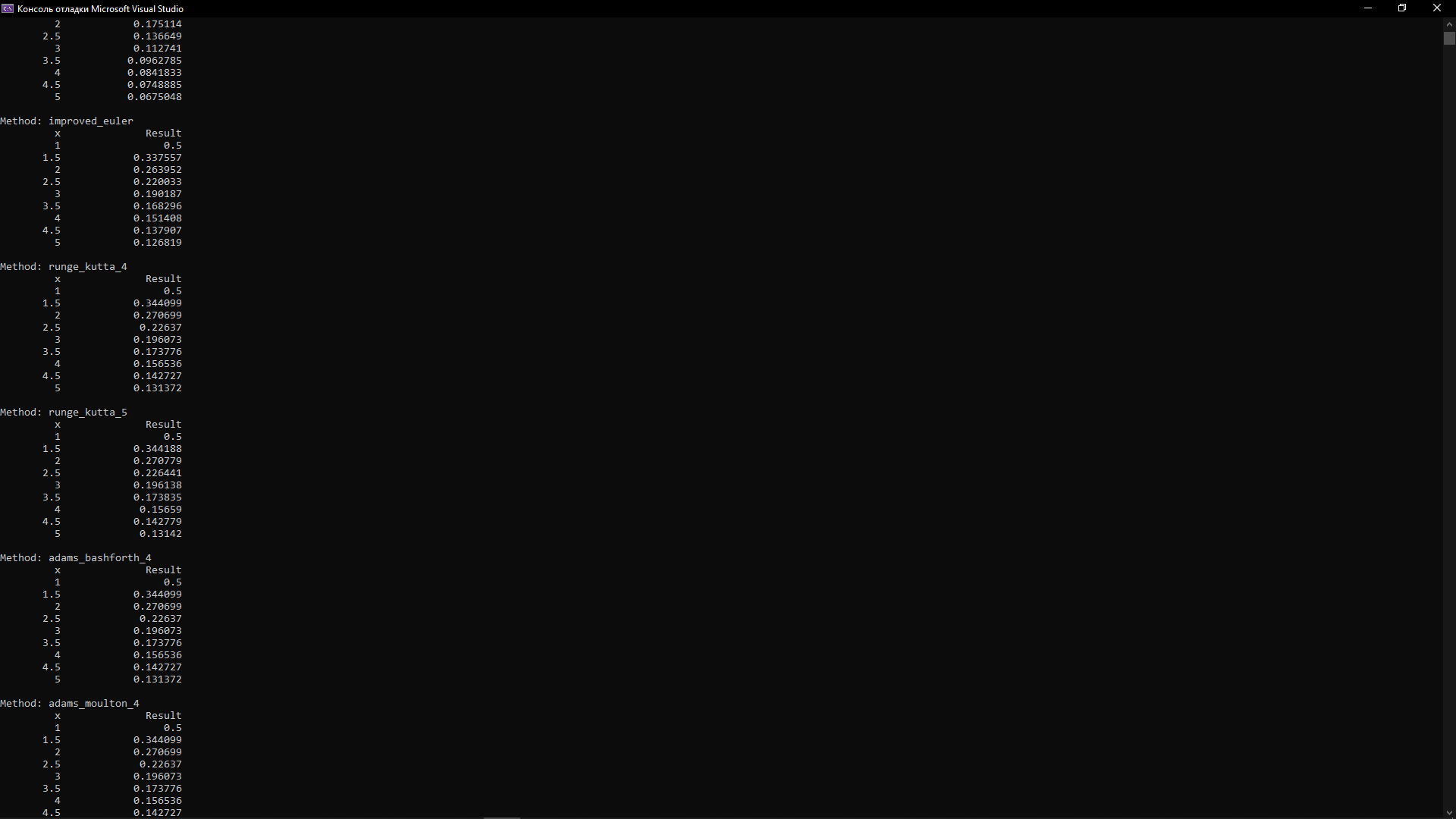
x.push\_back(xn);

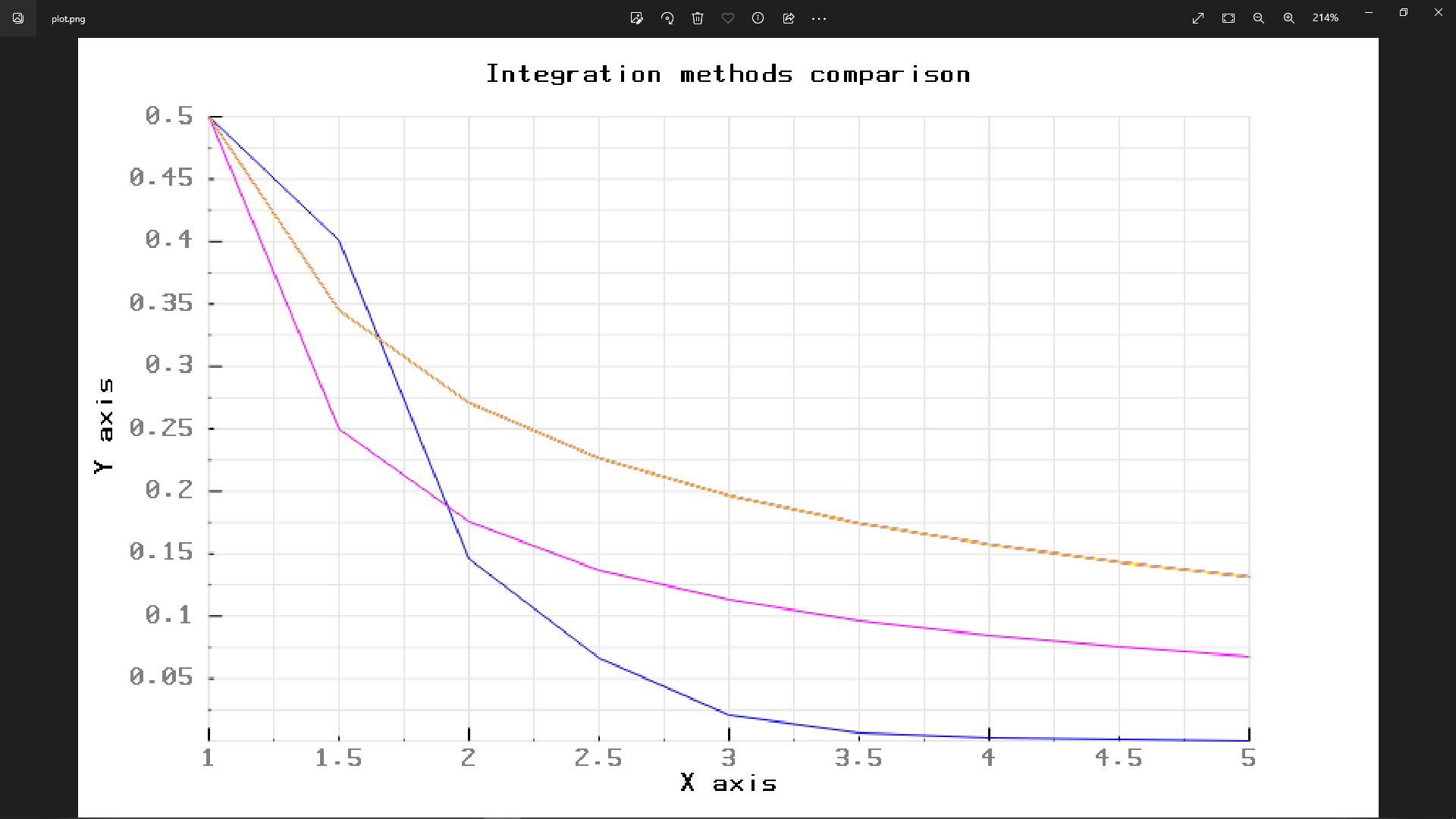
y.push\_back(yn);

}

}

Результат:





Выводы:

1. Приближение функции первой производной (L1) при помощи многочлена Лагранжа 4 и 5 степени дает очень большую погрешность, что говорит о том, что эти методы не подходят для приближения первой производной на данном интервале.

Приближение функции второй производной (L2) при помощи многочлена Лагранжа 4 и 5 степени дает приемлемую погрешность, но на отдельных точках наблюдается большая ошибка. Наиболее точное приближение достигается при использовании многочлена Лагранжа 2 степени.

С увеличением степени многочлена Лагранжа увеличивается точность приближения, но также увеличивается и погрешность на отдельных точках, что может привести к сильным отклонениям при интерполяции функции на большом интервале.

2. Многочлены Лагранжа хорошо приближают исходную функцию вблизи узлов интерполяции, но могут значительно отличаться от неё на других участках.

Интерполяция назад может дать достаточно точные результаты, если шаг интерполяции достаточно мал и функция достаточно гладкая.

При использовании интерполяции назад для нахождения производных может потребоваться использование многочлена высокого порядка, что может привести к сильной осцилляции между узлами интерполяции.

Возможно, необходимо использовать более точные методы вычисления производных, такие как численное дифференцирование или методы, основанные на интерполяции вперёд.

3. Из результатов метода Ньютона-Котеса можно сделать вывод, что точность численного интегрирования может значительно изменяться в зависимости от выбранной формулы интегрирования (n=4 или n=5). Ошибка для метода Ньютона-Котеса с n=4 составляет 1235.15, тогда как ошибка для метода Ньютона-Котеса с n=5 составляет 532.922.

Из результатов многочленов Лагранжа 4 и 5 порядка можно сделать вывод, что при интерполяции функции с помощью многочлена Лагранжа 4 порядка точность интерполяции оставляет желать лучшего. Это подтверждается тем, что ошибка интерполяции при использовании многочлена Лагранжа 4 порядка составляет около 1000 для всех значений x. Однако, при использовании многочлена Лагранжа 5 порядка, ошибка интерполяции снижается до значения порядка 10-100 для всех значений x, что говорит о том, что многочлен Лагранжа 5 порядка дает более точную интерполяцию функции.

4. Из результатов видно, что при увеличении количества разбиений от 1 до 9, результаты методов левых, правых и средних прямоугольников сначала колеблются, а затем стабилизируются вокруг значения интеграла. При этом метод трапеций и метод Симпсона дают более точный результат с меньшим колебанием.

Также заметно, что метод Симпсона сходится к значению интеграла быстрее, чем другие методы. Это связано с тем, что он использует квадратичные полиномы для аппроксимации функции, в то время как методы левых, правых и средних прямоугольников и метод трапеций используют линейные аппроксимации.

Таким образом, для данной функции метод Симпсона дает наиболее точный результат с наименьшим количеством разбиений. Однако для других функций может быть эффективнее использование других методов.

5. На основе предоставленных результатов можно сделать вывод, что различные методы численного интегрирования могут давать значительно разные результаты, особенно если шаг интегрирования выбирается недостаточно малым. Например, результаты методов default func и euler значительно отличаются от результатов других методов. Также можно заметить, что методы с более высоким порядком точности (improved\_euler, runge\_kutta\_4, runge\_kutta\_5, adams\_bashforth\_4, adams\_moulton\_4) дают результаты, близкие друг к другу.

6. Из результатов можно сделать вывод, что все методы численного решения дифференциального уравнения первого порядка дают схожие значения функции в точках, где она была вычислена. Однако, точность методов различается и может зависеть от конкретного решаемого уравнения. Методы Рунге-Кутты 4 и 5 порядков, а также Адамса-Башфорта 4 порядка и Адамса-Мультона 4 порядка показывают сходные результаты и более точные, чем методы Эйлера и улучшенного Эйлера.

Полный код программ:

1.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

#include <iomanip>

using namespace std;

double func(double x, double a, double b, double k) {

return pow(log(x), a / b) \* sin(k \* x);

}

double diff\_func(double x, double a, double b, double k) {

double ln\_x = log(x);

return pow(ln\_x, a / b - 1) \* (a / b \* cos(k \* x) \* sin(ln\_x) + k \* pow(ln\_x, a / b) \* cos(k \* x));

}

double diff2\_func(double x, double a, double b, double k) {

double ln\_x = log(x);

double sin\_kx = sin(k \* x);

double cos\_kx = cos(k \* x);

double pow\_ln\_x = pow(ln\_x, a / b);

double a\_b = a / b;

return pow\_ln\_x \* (pow\_ln\_x \* cos\_kx - 2 \* a\_b \* ln\_x \* sin\_kx) + 2 \* a\_b \* cos\_kx \* sin(ln\_x);

}

double lagrange\_poly(double x, const vector<double>& xi, const vector<double>& yi, int n) {

double L = 0;

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

double l = 1;

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

if (i != j) {

l \*= (x - xi[j]) / (xi[i] - xi[j]);

}

}

L += yi[i] \* l;

}

return L;

}

void plot(vector<double> x, vector<double> y, vector<double> L2, vector<double> L3, vector<double> L4) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* seriesXY = GetDefaultScatterPlotSeriesSettings();

seriesXY->xs = &x;

seriesXY->ys = &y;

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

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

seriesXY->linearInterpolation = true;

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

ScatterPlotSeries\* seriesXYCircles = GetDefaultScatterPlotSeriesSettings();

seriesXYCircles->xs = &x;

seriesXYCircles->ys = &y;

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

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

seriesXYCircles->linearInterpolation = false;

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

ScatterPlotSeries\* seriesL2 = GetDefaultScatterPlotSeriesSettings();

seriesL2->xs = &x;

seriesL2->ys = &L2;

seriesL2->lineType = toVector(L"solid");

seriesL2->pointType = toVector(L"circles");

seriesL2->linearInterpolation = true;

seriesL2->color = CreateRGBColor(0, 1, 0);

ScatterPlotSeries\* seriesL2Circles = GetDefaultScatterPlotSeriesSettings();

seriesL2Circles->xs = &x;

seriesL2Circles->ys = &L2;

seriesL2Circles->lineType = toVector(L"solid");

seriesL2Circles->pointType = toVector(L"circles");

seriesL2Circles->linearInterpolation = false;

seriesL2Circles->color = CreateRGBColor(0, 1, 0);

ScatterPlotSeries\* seriesL3 = GetDefaultScatterPlotSeriesSettings();

seriesL3->xs = &x;

seriesL3->ys = &L3;

seriesL3->lineType = toVector(L"solid");

seriesL3->pointType = toVector(L"circles");

seriesL3->linearInterpolation = true;

seriesL3->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* seriesL3Circles = GetDefaultScatterPlotSeriesSettings();

seriesL3Circles->xs = &x;

seriesL3Circles->ys = &L3;

seriesL3Circles->lineType = toVector(L"solid");

seriesL3Circles->pointType = toVector(L"circles");

seriesL3Circles->linearInterpolation = false;

seriesL3Circles->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* seriesL4 = GetDefaultScatterPlotSeriesSettings();

seriesL4->xs = &x;

seriesL4->ys = &L4;

seriesL4->lineType = toVector(L"solid");

seriesL4->pointType = toVector(L"circles");

seriesL4->linearInterpolation = true;

seriesL4->color = CreateRGBColor(1, 1, 0);

ScatterPlotSeries\* seriesL4Circles = GetDefaultScatterPlotSeriesSettings();

seriesL4Circles->xs = &x;

seriesL4Circles->ys = &L4;

seriesL4Circles->lineType = toVector(L"solid");

seriesL4Circles->pointType = toVector(L"circles");

seriesL4Circles->linearInterpolation = false;

seriesL4Circles->color = CreateRGBColor(1, 1, 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(seriesXY);

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

settings->scatterPlotSeries->push\_back(seriesL2);

settings->scatterPlotSeries->push\_back(seriesL2Circles);

settings->scatterPlotSeries->push\_back(seriesL3);

settings->scatterPlotSeries->push\_back(seriesL3Circles);

settings->scatterPlotSeries->push\_back(seriesL4);

settings->scatterPlotSeries->push\_back(seriesL4Circles);

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 a = 9, b = 7, k = 3, c = 3, d = 24;

const int m = 8;

double h = (d - c) / (m - 1);

vector<double> xi;

vector<double> yi;

double step = (d - c) / (m - 1);

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

xi.push\_back(i);

yi.push\_back(func(i, a, b, k));

}

vector<vector<double>> L(3); // вектор для многочленов порядков 2, 3, 4

vector<double> x, y, y\_diff\_exact, y\_diff2\_exact;

vector<vector<double>> y\_diff\_approx(3), y\_diff2\_approx(3);

double x\_point = c;

while (x\_point <= d) {

x.push\_back(x\_point);

y.push\_back(func(x\_point, a, b, k));

y\_diff\_exact.push\_back(diff\_func(x\_point, a, b, k));

y\_diff2\_exact.push\_back(diff2\_func(x\_point, a, b, k));

for (int i = 0; i < 3; i++) { // вычисление приближенных значений производных

y\_diff\_approx[i].push\_back((lagrange\_poly(x\_point, xi, yi, i + 2) - lagrange\_poly(x\_point - step, xi, yi, i + 2)) / step);

y\_diff2\_approx[i].push\_back((lagrange\_poly(x\_point + step, xi, yi, i + 2) - 2 \* lagrange\_poly(x\_point, xi, yi, i + 2) + lagrange\_poly(x\_point - step, xi, yi, i + 2)) / pow(step, 2));

}

x\_point += h;

}

for (int i = 0; i < 3; i++) { // вычисление многочленов порядков 2, 3, 4

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

L[i].push\_back(lagrange\_poly(j, xi, yi, i + 2));

}

}

// вывод таблицы сравнения точных и приближенных значений производных

cout << setw(5) << "x" << setw(15) << "y" << setw(15) << "Exact y'" << setw(15) << "L2'" << setw(15) << "L3'" << setw(15) << "L4'" << setw(15) << "Exact y''" << setw(15) << "L2''" << setw(15) << "L3''" << setw(15) << "L4''\n\n";

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

cout << setw(5) << x[i] << setw(15) << y[i] << setw(15) << y\_diff\_exact[i] << setw(15) << y\_diff\_approx[0][i] << setw(15) << y\_diff\_approx[1][i] << setw(15) << y\_diff\_approx[2][i] << setw(15) << y\_diff2\_exact[i] << setw(15) << y\_diff2\_approx[0][i] << setw(15) << y\_diff2\_approx[1][i] << setw(15) << y\_diff2\_approx[2][i] << "\n";

}

plot(x, y, L[0], L[1], L[2]);

return 0;

}

2. #include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

#include <iomanip>

using namespace std;

double func(double x, double a, double b, double k) {

return pow(log(x), a / b) \* sin(k \* x);

}

double diff\_func(double x, double a, double b, double k) {

double ln\_x = log(x);

return pow(ln\_x, a / b - 1) \* (a / b \* cos(k \* x) \* sin(ln\_x) + k \* pow(ln\_x, a / b) \* cos(k \* x));

}

double diff2\_func(double x, double a, double b, double k) {

double ln\_x = log(x);

double sin\_kx = sin(k \* x);

double cos\_kx = cos(k \* x);

double pow\_ln\_x = pow(ln\_x, a / b);

double a\_b = a / b;

return pow\_ln\_x \* (pow\_ln\_x \* cos\_kx - 2 \* a\_b \* ln\_x \* sin\_kx) + 2 \* a\_b \* cos\_kx \* sin(ln\_x);

}

double backward\_difference(int n, const vector<double>& y) {

if (n >= y.size()) {

return 0;

}

double res = y[n];

int i = 1;

while (n - i >= 0) {

res -= y[n - i];

i++;

}

return res;

}

double lagrange\_poly(double x, const vector<double>& xi, const vector<double>& yi, int n) {

double L = 0;

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

double l = 1;

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

if (i != j) {

l \*= (x - xi[j]) / (xi[i] - xi[j]);

}

}

L += yi[i] \* l;

}

return L;

}

void plot(vector<double> x, vector<double> y, vector<double> L2, vector<double> L3, vector<double> L4) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* seriesXY = GetDefaultScatterPlotSeriesSettings();

seriesXY->xs = &x;

seriesXY->ys = &y;

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

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

seriesXY->linearInterpolation = true;

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

ScatterPlotSeries\* seriesXYCircles = GetDefaultScatterPlotSeriesSettings();

seriesXYCircles->xs = &x;

seriesXYCircles->ys = &y;

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

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

seriesXYCircles->linearInterpolation = false;

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

ScatterPlotSeries\* seriesL2 = GetDefaultScatterPlotSeriesSettings();

seriesL2->xs = &x;

seriesL2->ys = &L2;

seriesL2->lineType = toVector(L"solid");

seriesL2->pointType = toVector(L"circles");

seriesL2->linearInterpolation = true;

seriesL2->color = CreateRGBColor(0, 1, 0);

ScatterPlotSeries\* seriesL2Circles = GetDefaultScatterPlotSeriesSettings();

seriesL2Circles->xs = &x;

seriesL2Circles->ys = &L2;

seriesL2Circles->lineType = toVector(L"solid");

seriesL2Circles->pointType = toVector(L"circles");

seriesL2Circles->linearInterpolation = false;

seriesL2Circles->color = CreateRGBColor(0, 1, 0);

ScatterPlotSeries\* seriesL3 = GetDefaultScatterPlotSeriesSettings();

seriesL3->xs = &x;

seriesL3->ys = &L3;

seriesL3->lineType = toVector(L"solid");

seriesL3->pointType = toVector(L"circles");

seriesL3->linearInterpolation = true;

seriesL3->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* seriesL3Circles = GetDefaultScatterPlotSeriesSettings();

seriesL3Circles->xs = &x;

seriesL3Circles->ys = &L3;

seriesL3Circles->lineType = toVector(L"solid");

seriesL3Circles->pointType = toVector(L"circles");

seriesL3Circles->linearInterpolation = false;

seriesL3Circles->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* seriesL4 = GetDefaultScatterPlotSeriesSettings();

seriesL4->xs = &x;

seriesL4->ys = &L4;

seriesL4->lineType = toVector(L"solid");

seriesL4->pointType = toVector(L"circles");

seriesL4->linearInterpolation = true;

seriesL4->color = CreateRGBColor(1, 1, 0);

ScatterPlotSeries\* seriesL4Circles = GetDefaultScatterPlotSeriesSettings();

seriesL4Circles->xs = &x;

seriesL4Circles->ys = &L4;

seriesL4Circles->lineType = toVector(L"solid");

seriesL4Circles->pointType = toVector(L"circles");

seriesL4Circles->linearInterpolation = false;

seriesL4Circles->color = CreateRGBColor(1, 1, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Backward interpolation");

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

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

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

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

settings->scatterPlotSeries->push\_back(seriesL2);

settings->scatterPlotSeries->push\_back(seriesL2Circles);

settings->scatterPlotSeries->push\_back(seriesL3);

settings->scatterPlotSeries->push\_back(seriesL3Circles);

settings->scatterPlotSeries->push\_back(seriesL4);

settings->scatterPlotSeries->push\_back(seriesL4Circles);

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 a = 9, b = 7, k = 3, c = 3, d = 24; // d = 5

const int m = 8;

double h = (d - c) / (m);

vector<double> xi;

vector<double> yi;

double step = (d - c) / (m);

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

xi.push\_back(i);

yi.push\_back(func(i, a, b, k));

}

vector<vector<double>> L(3);

vector<double> x, y, y\_diff\_exact, y\_diff2\_exact;

vector<vector<double>> y\_diff\_approx(3), y\_diff2\_approx(3);

double x\_point = c;

while (x\_point <= d) {

x.push\_back(x\_point);

y.push\_back(func(x\_point, a, b, k));

y\_diff\_exact.push\_back(diff\_func(x\_point, a, b, k));

y\_diff2\_exact.push\_back(diff2\_func(x\_point, a, b, k));

for (int i = 0; i < 3; i++) {

y\_diff\_approx[i].push\_back(backward\_difference(i, yi));

y\_diff2\_approx[i].push\_back(backward\_difference(i, y\_diff\_approx[i]));

}

x\_point += step;

}

for (int i = 0; i < 3; i++) { // вычисление многочленов порядков 2, 3, 4

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

L[i].push\_back(lagrange\_poly(j, xi, yi, i + 2));

}

}

//plot(x, y, L[0], L[1], L[2]);

cout << "Exact values of derivatives:" << endl;

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

cout << "x = " << x[i] << ", y' = " << y\_diff\_exact[i] << ", y'' = " << y\_diff2\_exact[i]

<< endl;

}

cout << "Approximate values of derivatives:" << endl;

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

cout << "x = " << x[i] << ", N'2 = " << y\_diff\_approx[0][i] << ", N'3 = " << y\_diff\_approx[1][i]

<< ", N'4 = " << y\_diff\_approx[2][i] << ", N''2 = " << y\_diff2\_approx[0][i] << ", N''3 = "

<< y\_diff2\_approx[1][i] << ", N''4 = " << y\_diff2\_approx[2][i] << endl;

}

plot(x, y, L[0], L[1], L[2]);

return 0;

}

3.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// Функция, которую необходимо проинтегрировать

double f(double x, double a, double b, double k) {

return pow(log(x), a / b) \* sin(k \* x);

}

// Вычисление значения полинома Лагранжа

double lagrange(double x, const vector<double>& xs, const vector<double>& ys, int n) {

double result = 0;

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

double term = ys[i];

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

if (j != i) {

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

}

}

result += term;

}

return result;

}

void plot(vector<double> x, vector<double> y, vector<double> L4, vector<double> L5) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* seriesXY = GetDefaultScatterPlotSeriesSettings();

seriesXY->xs = &x;

seriesXY->ys = &y;

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

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

seriesXY->linearInterpolation = true;

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

ScatterPlotSeries\* seriesXYCircles = GetDefaultScatterPlotSeriesSettings();

seriesXYCircles->xs = &x;

seriesXYCircles->ys = &y;

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

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

seriesXYCircles->linearInterpolation = false;

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

ScatterPlotSeries\* seriesL2 = GetDefaultScatterPlotSeriesSettings();

seriesL2->xs = &x;

seriesL2->ys = &L4;

seriesL2->lineType = toVector(L"solid");

seriesL2->pointType = toVector(L"circles");

seriesL2->linearInterpolation = true;

seriesL2->color = CreateRGBColor(0, 1, 0);

ScatterPlotSeries\* seriesL2Circles = GetDefaultScatterPlotSeriesSettings();

seriesL2Circles->xs = &x;

seriesL2Circles->ys = &L4;

seriesL2Circles->lineType = toVector(L"solid");

seriesL2Circles->pointType = toVector(L"circles");

seriesL2Circles->linearInterpolation = false;

seriesL2Circles->color = CreateRGBColor(0, 1, 0);

ScatterPlotSeries\* seriesL3 = GetDefaultScatterPlotSeriesSettings();

seriesL3->xs = &x;

seriesL3->ys = &L5;

seriesL3->lineType = toVector(L"solid");

seriesL3->pointType = toVector(L"circles");

seriesL3->linearInterpolation = true;

seriesL3->color = CreateRGBColor(1, 0, 0);

ScatterPlotSeries\* seriesL3Circles = GetDefaultScatterPlotSeriesSettings();

seriesL3Circles->xs = &x;

seriesL3Circles->ys = &L5;

seriesL3Circles->lineType = toVector(L"solid");

seriesL3Circles->pointType = toVector(L"circles");

seriesL3Circles->linearInterpolation = false;

seriesL3Circles->color = CreateRGBColor(1, 0, 0);

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Newton-Kotes");

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

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

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

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

settings->scatterPlotSeries->push\_back(seriesL2);

settings->scatterPlotSeries->push\_back(seriesL2Circles);

settings->scatterPlotSeries->push\_back(seriesL3);

settings->scatterPlotSeries->push\_back(seriesL3Circles);

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 a = 9, b = 7, k = 3, c = 3, d = 24;

const int n1 = 4, n2 = 5, m = 8;

double h = (d - c) / (m - 1);

vector<double> xs(m);

vector<double> ys(m);

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

xs[i] = c + i \* h;

ys[i] = f(xs[i], a, b, k);

}

// Вычисление интерполяционных значений

vector<double> L1(m), L2(m);

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

L1[i] = lagrange(xs[i], xs, ys, n1);

L2[i] = lagrange(xs[i], xs, ys, n2);

}

// Вычисление интегралов методом Ньютона-Котоса

double I1 = 0, I2 = 0;

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

double x = xs[i];

double y = f(x, a, b, k);

I1 += L1[i] \* h;

I2 += L2[i] \* h;

}

// Вычисление погрешностей

double E1 = abs(I1 - 11.67028773);

double E2 = abs(I2 - 11.67028773);

// Вывод результатов

cout << "Integral (Newton-Cotes, n=4): " << I1 << endl;

cout << "Integral (Newton-Cotes, n=5): " << I2 << endl;

cout << "Error (n=4): " << E1 << endl;

cout << "Error (n=5): " << E2 << endl;

cout << "Lagrange 4 order: ";

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

cout << L1[i] << " ";

}

cout << endl;

cout << "Lagrange 5 order: ";

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

cout << L2[i] << " ";

}

cout << endl;

plot(xs, ys, L1, L2);

return 0;

}

4.

#include <iostream>

#include <cmath>

#include <vector>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

double a = 9, b = 7, k = 3, c = 3, d = 24, m = 8;

double function(double x) {

return pow(log(x), a / b) \* sin(k \* x);

}

double leftRectangleMethod(double h) {

double sum = 0;

double x = c;

while (x + h <= d) {

sum += function(x) \* function(x);

x += h;

}

return h \* sum;

}

double rightRectangleMethod(double h) {

double sum = 0;

double x = c + h;

while (x <= d) {

sum += function(x) \* function(x);

x += h;

}

return h \* sum;

}

double middleRectangleMethod(double h) {

double sum = 0;

double x = c + h / 2;

while (x < d) {

sum += function(x) \* function(x);

x += h;

}

return h \* sum;

}

double trapezoidMethod(double h) {

double sum = function(c) \* function(c) + function(d) \* function(d);

double x = c + h;

while (x < d) {

sum += 2 \* function(x) \* function(x);

x += h;

}

return h / 2 \* sum;

}

double simpsonMethod(double h) {

double sum = function(c) \* function(c) + function(d) \* function(d);

double x = c + h;

while (x < d) {

sum += 2 \* function(x) \* function(x - h / 2) + 4 \* function(x - h / 2) \* function(x - h / 2);

x += h;

}

return h / 6 \* sum;

}

void table(vector<double> res, string method) {

cout << "Method: " << method << endl;

cout << "n \t\t Result" << endl;

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

cout << i + 1 << " \t\t " << res[i] << endl;

}

cout << endl;

}

void plot(vector<double> x, vector<double> leftRes, vector<double> rightRes, vector<double> middleRes, vector<double> trapRes, vector<double> simpRes) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* seriesLeftRes = GetDefaultScatterPlotSeriesSettings();

seriesLeftRes->xs = &x;

seriesLeftRes->ys = &leftRes;

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

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

seriesLeftRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesRightRes = GetDefaultScatterPlotSeriesSettings();

seriesRightRes->xs = &x;

seriesRightRes->ys = &rightRes;

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

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

seriesRightRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesMiddleRes = GetDefaultScatterPlotSeriesSettings();

seriesMiddleRes->xs = &x;

seriesMiddleRes->ys = &middleRes;

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

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

seriesMiddleRes->linearInterpolation = true;

seriesMiddleRes->color = CreateRGBColor(0, 0, 0);

ScatterPlotSeries\* seriesTrapRes = GetDefaultScatterPlotSeriesSettings();

seriesTrapRes->xs = &x;

seriesTrapRes->ys = &trapRes;

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

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

seriesTrapRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesSimpRes = GetDefaultScatterPlotSeriesSettings();

seriesSimpRes->xs = &x;

seriesSimpRes->ys = &simpRes;

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

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

seriesSimpRes->linearInterpolation = true;

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

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Integration methods comparison");

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

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

vector<ScatterPlotSeries\*> series({ seriesLeftRes, seriesRightRes, seriesMiddleRes, seriesTrapRes, seriesSimpRes });

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

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

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

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

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

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, y;

vector<double> leftRes, rightRes, middleRes, trapRes, simpRes;

double h = (d - c) / m;

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

x.push\_back(i);

}

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

leftRes.push\_back(leftRectangleMethod(h));

rightRes.push\_back(rightRectangleMethod(h));

middleRes.push\_back(middleRectangleMethod(h));

trapRes.push\_back(trapezoidMethod(h));

simpRes.push\_back(simpsonMethod(h));

h /= 2;

}

table(leftRes, "Left Rectangle Method");

table(rightRes, "Right Rectangle Method");

table(middleRes, "Middle Rectangle Method");

table(trapRes, "Trapezoid Method");

table(simpRes, "Simpson Method");

cout << x.size() << leftRes.size() << rightRes.size() << middleRes.size() << trapRes.size() << simpRes.size();

plot(x, leftRes, rightRes, middleRes, trapRes, simpRes);

return 0;

}

5.

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// Define constants

const double a = 9;

const double b = 7;

const double k = 3;

const double c = 3;

const double d = 24;

const double m = 8;

const double n = 5;

const double E = 1e-6;

const double h = (d - c) / m;

// Define the function to be differentiated

double f(double x) {

return pow(-1, int(x)) \* b + k \* a \* pow(x, 2) + d \* pow(x, 2) + m \* x + n;

}

// Define the Euler method

void euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double yn1 = yn + h \* f(xn);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the improved Euler method

void improved\_euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn);

double k2 = f(xn + h);

double yn1 = yn + h \* (k1 + k2) / 2;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta 4th order method

void runge\_kutta\_4(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1;

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn);

double k2 = f(xn + h / 2);

double k3 = f(xn + h / 2);

double k4 = f(xn + h);

double yn1 = yn + h \* (k1 + 2 \* k2 + 2 \* k3 + k4) / 6;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta5th order method

void runge\_kutta\_5(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = 1;

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn);

double k2 = f(xn + h / 4);

double k3 = f(xn + h / 4);

double k4 = f(xn + h / 2);

double k5 = f(xn + 3 \* h / 4);

double k6 = f(xn + h);

double yn1 = yn + h \* (k1 \* 7 / 90 + k3 \* 32 / 90 + k4 \* 12 / 90 + k5 \* 32 / 90 + k6 \* 7 / 90);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Bashforth 4th order method

void adams\_bashforth\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Update x vector with initial values

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

y.push\_back(y\_prev[i]);

}

// Iterate using the Adams-Bashforth 4th order method

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

double yn = y[i - 1] + h \* (55 \* f(x[i]) - 59 \* f(x[i - 2]) + 37 \* f(x[i - 3]) - 9 \* f(x[i - 4])) / 24;

double xn = x[i - 1] + h;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Moulton 4th order method

void adams\_moulton\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Iterate using the Adams-Moulton 4th order method

for (int i = 3; i < m; i++) {

double yn = y[i - 1] + h \* (9 \* f(x[i]) + 19 \* f(x[i - 1]) - 5 \* f(x[i - 2]) + f(x[i - 3])) / 24;

double xn = x[i - 1] + h;

x.push\_back(xn);

y.push\_back(yn);

}

}

void table(vector<double> x, vector<double> y, string method) {

cout << "Method: " << method << endl;

cout << setw(10) << "x" << setw(10) << "Result" << endl;

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

cout << setw(10) << x[i] << setw(10) << y[i] << endl;

}

cout << endl;

}

void def\_func(vector<double>& xs, vector<double>& ys) {

// Определяем начальное условие

double y0 = 1;

// Определяем шаг

double h = (d - c) / m;

xs.clear();

ys.clear();

// Заполняем векторы

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

xs.push\_back(x);

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

}

}

void plot(vector<double> x, vector<double> y, vector<double> leftRes, vector<double> rightRes, vector<double> middleRes, vector<double> trapRes, vector<double> simpRes) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* seriesXY = GetDefaultScatterPlotSeriesSettings();

seriesXY->xs = &x;

seriesXY->ys = &y;

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

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

seriesXY->linearInterpolation = true;

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

ScatterPlotSeries\* seriesLeftRes = GetDefaultScatterPlotSeriesSettings();

seriesLeftRes->xs = &x;

seriesLeftRes->ys = &leftRes;

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

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

seriesLeftRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesRightRes = GetDefaultScatterPlotSeriesSettings();

seriesRightRes->xs = &x;

seriesRightRes->ys = &rightRes;

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

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

seriesRightRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesMiddleRes = GetDefaultScatterPlotSeriesSettings();

seriesMiddleRes->xs = &x;

seriesMiddleRes->ys = &middleRes;

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

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

seriesMiddleRes->linearInterpolation = true;

seriesMiddleRes->color = CreateRGBColor(0, 0, 0);

ScatterPlotSeries\* seriesTrapRes = GetDefaultScatterPlotSeriesSettings();

seriesTrapRes->xs = &x;

seriesTrapRes->ys = &trapRes;

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

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

seriesTrapRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesSimpRes = GetDefaultScatterPlotSeriesSettings();

seriesSimpRes->xs = &x;

seriesSimpRes->ys = &simpRes;

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

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

seriesSimpRes->linearInterpolation = true;

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

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Integration methods comparison");

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

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

vector<ScatterPlotSeries\*> series({ seriesXY, seriesLeftRes, seriesRightRes, seriesMiddleRes, seriesTrapRes, seriesSimpRes });

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

settings->scatterPlotSeries->push\_back(series[i]);

}

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, y;

vector<double> eul, impEul, rk4, rk5, ab4, am4;

def\_func(x, y);

euler(x, eul);

improved\_euler(x, impEul);

runge\_kutta\_4(x, rk4);

runge\_kutta\_5(x, rk5);

adams\_bashforth\_4(x, ab4);

adams\_moulton\_4(x, am4);

while (x.size() > 9)

x.pop\_back();

while (eul.size() > 9)

eul.pop\_back();

while (rk4.size() > 9)

rk4.pop\_back();

while (rk5.size() > 9)

rk5.pop\_back();

while (ab4.size() > 9)

ab4.pop\_back();

while (am4.size() > 9)

am4.pop\_back();

table(x, y, "default func");

table(x, eul, "euler");

table(x, impEul, "improved\_euler");

table(x, rk4, "runge\_kutta\_4");

table(x, rk5, "runge\_kutta\_5");

table(x, ab4, "adams\_bashforth\_4");

table(x, am4, "adams\_moulton\_4");

/\*for (int i = 0; i < x.size(); i++)

cout << x[i] << "\n";\*/

cout << x.size() << "\n";

cout << y.size() << "\n";

cout << eul.size() << "\n";

cout << rk4.size() << "\n";

cout << rk5.size() << "\n";

cout << ab4.size() << "\n";

cout << am4.size() << "\n";

plot(x, y, eul, rk4, rk5, ab4, am4);

}

6.

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

#include <pbPlots.cpp>

#include <supportLib.cpp>

using namespace std;

// Define constants

const double a = 9;

const double b = 7;

const double k = 3;

const double c = 1;

const double d = 5;

const double m = 8;

const double n = 5;

const double \_\_\_y0 = 0.5;

const double h = (d - c) / m;

// Define the function to be differentiated

double f(double x, double y) {

return pow(y, 2) \* log(x) / x - y / x;

}

// Define the Euler method

void euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double yn1 = yn + h \* f(xn, yn);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the improved Euler method

void improved\_euler(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0; // Replace with your initial value

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn, yn);

double k2 = f(xn + h, yn + h \* k1);

double yn1 = yn + h \* (k1 + k2) / 2;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta 4th order method

void runge\_kutta\_4(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0;

x.clear();

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn, yn);

double k2 = f(xn + h / 2, yn + h \* k1 / 2);

double k3 = f(xn + h / 2, yn + h \* k2 / 2);

double k4 = f(xn + h, yn + h \* k3);

double yn1 = yn + h \* (k1 + 2 \* k2 + 2 \* k3 + k4) / 6;

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Runge-Kutta5th order method

void runge\_kutta\_5(vector<double>& x, vector<double>& y) {

double xn = c;

double yn = \_\_\_y0;

x.push\_back(xn);

y.push\_back(yn);

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

double k1 = f(xn, yn);

double k2 = f(xn + h / 4, yn + h \* k1 / 4);

double k3 = f(xn + h / 4, yn + h \* (k1 / 8 + k2 / 8));

double k4 = f(xn + h / 2, yn - h \* (k2 / 2) + h \* k3);

double k5 = f(xn + 3 \* h / 4, yn + h \* (k1 \* 3 / 16 + k4 \* 9 / 16));

double k6 = f(xn + h, yn + h \* (-k1 \* 3 / 7 + k2 \* 2 / 7 + k3 \* 12 / 7 - k4 \* 12 / 7 + k5 \* 8 / 7));

double yn1 = yn + h \* (k1 \* 7 / 90 + k3 \* 32 / 90 + k4 \* 12 / 90 + k5 \* 32 / 90 + k6 \* 7 / 90);

xn += h;

yn = yn1;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Bashforth 4th order method

void adams\_bashforth\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Update x vector with initial values

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

y.push\_back(y\_prev[i]);

}

// Iterate using the Adams-Bashforth 4th order method

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

double yn = y[i - 1] + h \* (55 \* f(x[i], y[i - 1]) - 59 \* f(x[i - 2], y[i - 2]) + 37 \* f(x[i - 3], y[i - 3]) - 9 \* f(x[i - 4], y[i - 4])) / 24;

double xn = x[i - 1] + h;

x.push\_back(xn);

y.push\_back(yn);

}

}

// Define the Adams-Moulton 4th order method

void adams\_moulton\_4(vector<double>& x, vector<double>& y) {

// Create vectors to store previous values

vector<double> x\_prev;

vector<double> y\_prev;

x.clear();

// Use Runge-Kutta 4th order method to calculate initial values

runge\_kutta\_4(x\_prev, y\_prev);

// Copy initial values to x and y vectors

x = x\_prev;

y = y\_prev;

// Iterate using the Adams-Moulton 4th order method

for (int i = 3; i < m; i++) {

double yn = y[i - 1] + h \* (9 \* f(x[i], y[i]) + 19 \* f(x[i - 1], y[i - 1]) - 5 \* f(x[i - 2], y[i - 2]) + f(x[i - 3], y[i - 3])) / 24;

double xn = x[i - 1] + h;

x.push\_back(xn);

y.push\_back(yn);

}

}

void table(vector<double> x, vector<double> y, string method) {

cout << "Method: " << method << endl;

cout << setw(10) << "x" << setw(20) << "Result" << endl;

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

cout << setw(10) << x[i] << setw(20) << y[i] << endl;

}

cout << endl;

}

void def\_func(vector<double>& xs, vector<double>& ys) {

// Определяем начальное условие

double y0 = \_\_\_y0;

// Определяем шаг

double h = (d - c) / m;

xs.clear();

ys.clear();

// Заполняем векторы

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

double y = f(x, y0);

xs.push\_back(x);

ys.push\_back(abs(y));

y0 = y; // Обновляем y0 для следующей итерации

}

}

void plot(vector<double> x, vector<double> y, vector<double> leftRes, vector<double> rightRes, vector<double> middleRes, vector<double> trapRes, vector<double> simpRes) {

RGBABitmapImageReference\* imageReference = CreateRGBABitmapImageReference();

ScatterPlotSeries\* seriesXY = GetDefaultScatterPlotSeriesSettings();

seriesXY->xs = &x;

seriesXY->ys = &y;

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

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

seriesXY->linearInterpolation = true;

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

ScatterPlotSeries\* seriesLeftRes = GetDefaultScatterPlotSeriesSettings();

seriesLeftRes->xs = &x;

seriesLeftRes->ys = &leftRes;

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

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

seriesLeftRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesRightRes = GetDefaultScatterPlotSeriesSettings();

seriesRightRes->xs = &x;

seriesRightRes->ys = &rightRes;

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

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

seriesRightRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesMiddleRes = GetDefaultScatterPlotSeriesSettings();

seriesMiddleRes->xs = &x;

seriesMiddleRes->ys = &middleRes;

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

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

seriesMiddleRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesTrapRes = GetDefaultScatterPlotSeriesSettings();

seriesTrapRes->xs = &x;

seriesTrapRes->ys = &trapRes;

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

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

seriesTrapRes->linearInterpolation = true;

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

ScatterPlotSeries\* seriesSimpRes = GetDefaultScatterPlotSeriesSettings();

seriesSimpRes->xs = &x;

seriesSimpRes->ys = &simpRes;

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

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

seriesSimpRes->linearInterpolation = true;

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

ScatterPlotSettings\* settings = GetDefaultScatterPlotSettings();

settings->width = 800;

settings->height = 480;

settings->title = toVector(L"Integration methods comparison");

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

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

vector<ScatterPlotSeries\*> series({ seriesXY, seriesLeftRes, seriesRightRes, seriesMiddleRes, seriesTrapRes, seriesSimpRes });

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

settings->scatterPlotSeries->push\_back(series[i]);

}

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, y;

vector<double> eul, impEul, rk4, rk5, ab4, am4;

def\_func(x, y);

euler(x, eul);

improved\_euler(x, impEul);

runge\_kutta\_4(x, rk4);

runge\_kutta\_5(x, rk5);

adams\_bashforth\_4(x, ab4);

adams\_moulton\_4(x, am4);

while (x.size() > 9)

x.pop\_back();

while (eul.size() > 9)

eul.pop\_back();

while (rk4.size() > 9)

rk4.pop\_back();

while (rk5.size() > 9)

rk5.pop\_back();

while (ab4.size() > 9)

ab4.pop\_back();

while (am4.size() > 9)

am4.pop\_back();

table(x, y, "default func");

table(x, eul, "euler");

table(x, impEul, "improved\_euler");

table(x, rk4, "runge\_kutta\_4");

table(x, rk5, "runge\_kutta\_5");

table(x, ab4, "adams\_bashforth\_4");

table(x, am4, "adams\_moulton\_4");

/\*for (int i = 0; i < x.size(); i++)

cout << x[i] << "\n";\*/

cout << x.size() << "\n";

cout << y.size() << "\n";

cout << eul.size() << "\n";

cout << rk4.size() << "\n";

cout << rk5.size() << "\n";

cout << ab4.size() << "\n";

cout << am4.size() << "\n";

plot(x, y, eul, rk4, rk5, ab4, am4);

}