Міністерство освіти і науки України

Національний університет “Львівська політехніка”

Інститут комп’ютерних наук та інформаційних технологій

Кафедра програмного забезпечення



**Звіт**

Про виконання лабораторної роботи № 8

**На тему:**

«Наближення дискретних (таблично заданих) функцій»

**Лектор:**

доц. каф. ПЗ

Мельник Н. Б.

**Виконав:**

ст. гр. ПЗ-18

Юшкевич А.І.

**Прийняв:**

проф. каф. ПЗ

Гавриш В.І.

« … » … 2023 р.

∑ = \_\_\_\_\_\_\_\_\_\_\_\_

Львів – 2023

**Тема роботи:** Наближення дискретних (таблично заданих) функцій**.**

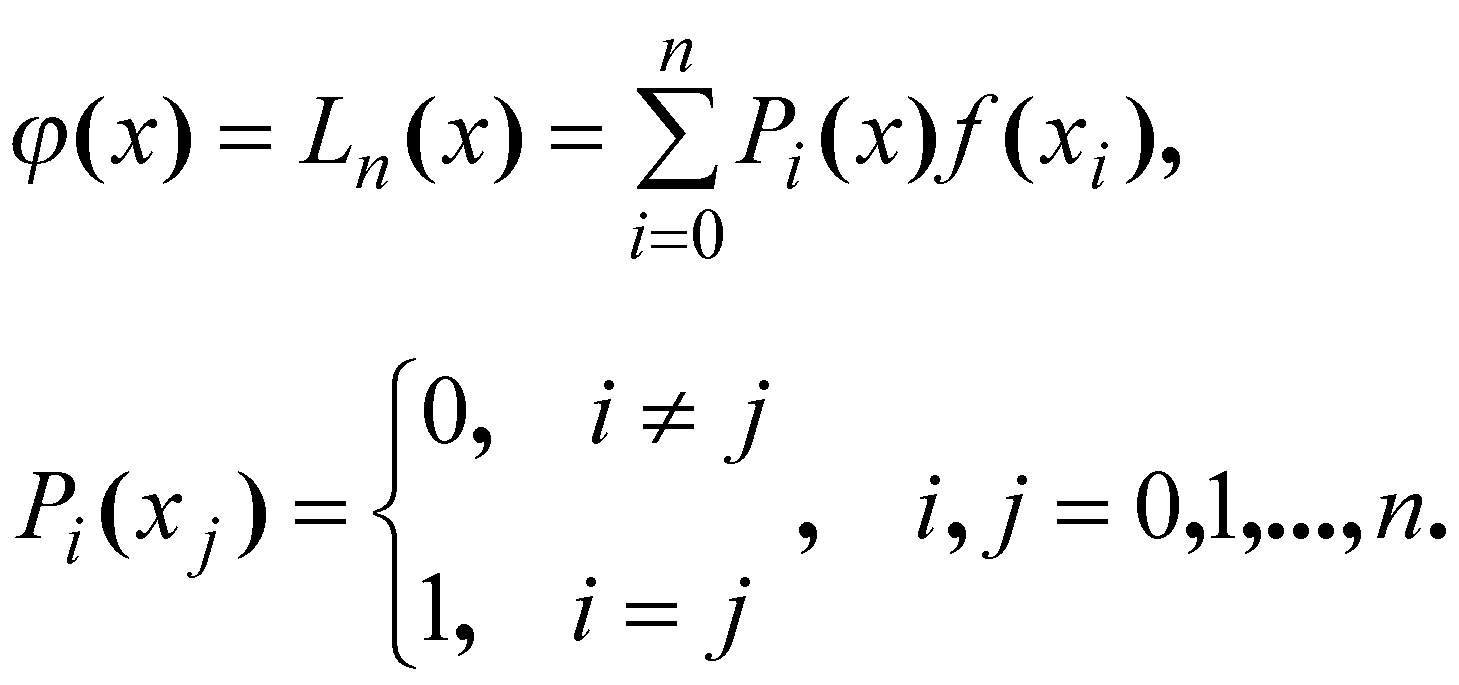
**Мета роботи:** Ознайомлення на практиці з методами інтерполяції функцій.

**Теоретичні відомості**

**Інтерполяційний поліном Лагранжа**

Один з підходів до задачі інтерполяції – метод Лагранжа. Основна ідея цього методу полягає в пошуку поліному, який приймає значення 1 в одному довільному вузлі інтерполяції і значення 0 у всіх інших вузлах.

Наближену функцію  представимо у вигляді



Оскільки точки  є коренями многочлена , то його можна записати наступним чином

,

а наближена функція , яку називають *інтерполяційним многочленом Лагранжа*, матиме вигляд



**Інтерполяційний поліном Ньютона**

Іншим підходом до задачі інтерполяції є метод Ньютона (метод розділених різниць). Нехай для функції  задано її значення в точках  Треба побудувати такий поліном  степеня не вище від  значення якого у вузлах інтерполювання збігаються із значенням функції , тобто



Поліном  будемо шукати у вигляді



розділена різниця го порядку.

Нехай вузли інтерполяції утворюють арифметичну прогресію  , крок інтерполяції.

Таким чином, скінченну різницю го порядку можна записати у вигляді



Розділену різницю го порядку можна виразити через скінченну різницю го порядку



Тоді наведений вище поліном можна записати у вигляді



отримане представлення називають *інтерполяційним поліномом Ньютона для інтерполяціїї вперед* для рівновіддалених вузлів інтерполяції*.*

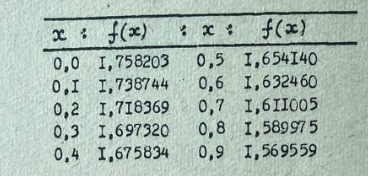
Формули Ньютона та Лагранжа характеризують один і той самий поліном, вони відрізняються лише алгоритмом його побудови.

**Індивідуальне завдання**

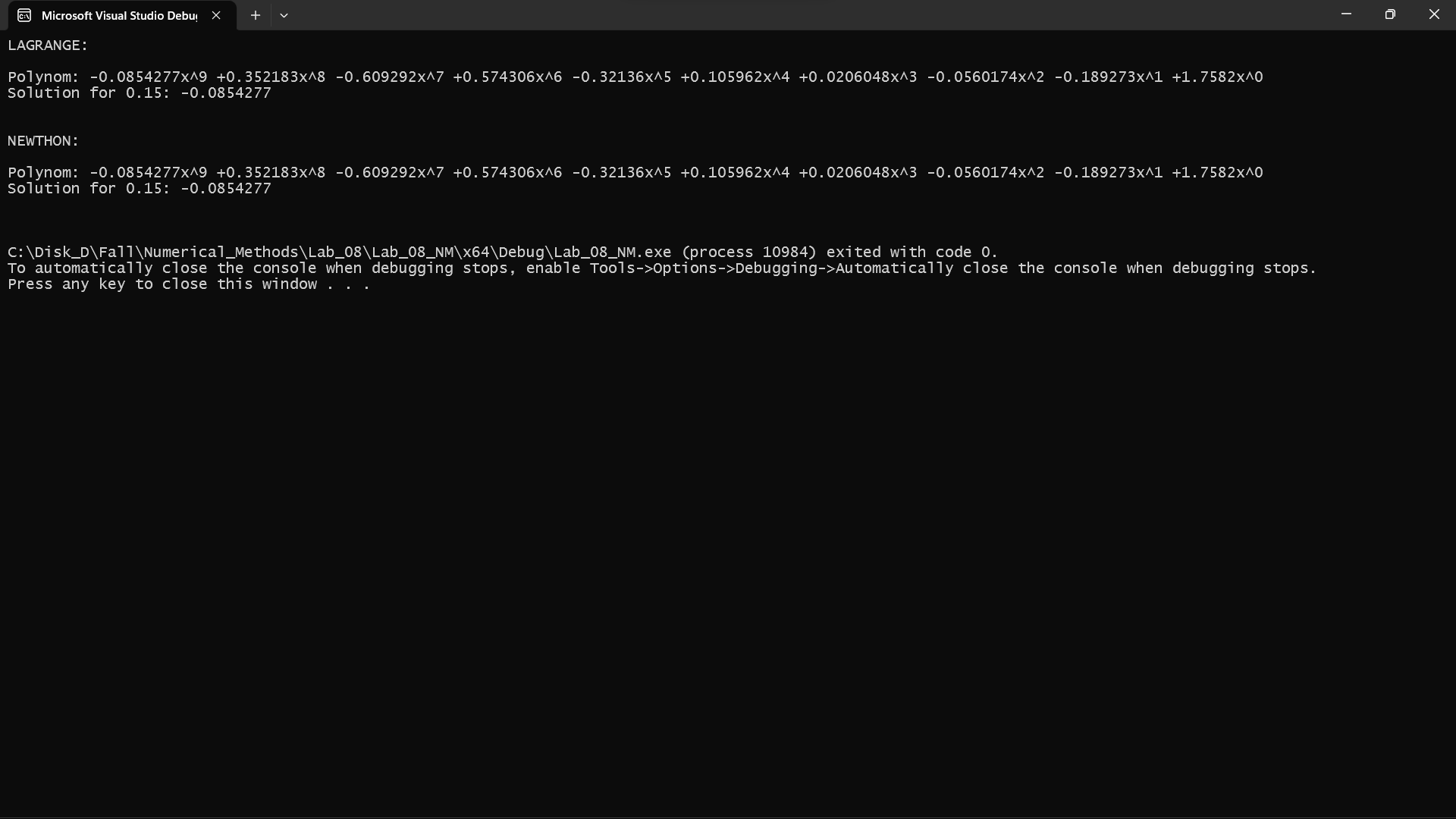
Варіант 22:

Використовуючи інтерполяційні поліноми Лагранжа та Ньютона, обчислити значення таблично заданої функції у точці x0.

*Таблично задана функція:*



**Результат виконання програми**



**Висновки**

У результаті виконання лабораторної роботи обчислено значення таблично заданої функції у точці x0 (x0 = 1.16887), використовуючи інтерполяційні поліноми Лагранжа та Ньютона. Методи Лагранжа та Ньютона націлені на знаходження інтерполяційного поліному. У випадку, якщо вони використовуються для знаходження інтерполяційного поліному однієї функції, двома методами виходить однаковий поліном.

**Додаток**

CInterpolation.h:

#pragma once

#include <iostream>

#include <vector>

using namespace std;

class CInterpolation

{

public:

CInterpolation() = delete;

CInterpolation(double\* m\_x, double\* m\_y, size\_t m\_size);

vector<double> Lagrange() const;

double FindByLagrange(double x) const;

vector<double> Newthon(double x) const;

double FindByNewthon(double x) const;

private:

vector<double> m\_x;

vector<double> m\_y;

size\_t m\_size;

double FindDifferences(int difference\_index, int x\_index) const;

double FindQ(double x, double movable\_x, double interpolation\_step) const;

vector<double> Forward(double interpolation\_step) const;

double Factorial(int x) const;

};

CInterpolation.cpp:

#include "CInterpolation.h"

CInterpolation::CInterpolation(double\* x, double\* y, size\_t size) {

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

this->m\_x.push\_back(x[i]);

this->m\_y.push\_back(y[i]);

}

this->m\_size = size;

}

vector<double> CInterpolation::Lagrange() const {

vector<double> solution;

vector<double> result;

vector<double> temp;

double free{ 0 };

result.push\_back(1);

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

if (i != 0)

result.push\_back(0);

temp.push\_back(1);

solution.push\_back(0);

}

for (int uni = 0; uni < m\_size; uni++) {

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

if (uni == i) {

free++;

continue;

}

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

temp[j] \*= -m\_x[i];

result[j + 1] += temp[j];

}

int j{ 0 };

for (; j < i + 2 - free; j++) {

temp[j] = result[j];

}

}

double division{ 1 };

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

if (i == uni)

continue;

division \*= m\_x[uni] - m\_x[i];

}

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

result[i] \*= m\_y[uni];

result[i] /= division;

solution[i] += result[i];

}

division = 1;

free = 0;

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

temp[i] = 1;

if (i == 0)

result[i] = 1;

else

result[i] = 0;

}

}

return solution;

}

double CInterpolation::FindByLagrange(double x) const{

vector<double> polynom = Lagrange();

double result{ 0 };

for (int i = polynom.size() - 1; i >= 0; i--) {

result = polynom[i] \* pow(x, i);

}

return result;

}

double CInterpolation::FindDifferences(int difference\_index, int x\_index) const{

double result{ 0 };

if (x\_index > m\_size - difference\_index - 2) {

cout << "error" << endl << endl;

result = 0;

}

else if (difference\_index == 0) {

result = m\_y[x\_index + 1] - m\_y[x\_index];

}

else {

result = FindDifferences(difference\_index - 1, x\_index + 1) - FindDifferences(difference\_index - 1, x\_index);

}

return result;

}

double CInterpolation::FindQ(double x, double movable\_x, double interpolation\_step) const {

return (x - movable\_x) / interpolation\_step;

}

vector<double> CInterpolation::Forward(double interpolation\_step) const {

vector<double> difference;

vector<double> r;

vector<double> solution;

vector<double> result;

vector<double> temp;

difference.push\_back(m\_y[0]);

result.push\_back(1);

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

if (i != 0)

result.push\_back(0);

temp.push\_back(1);

solution.push\_back(0);

r.push\_back(m\_x[0] + i \* interpolation\_step);

if (i != m\_size - 1)

difference.push\_back(FindDifferences(i, 0));

}

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

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

for (int j = 0; j < l + 1; j++) {

temp[j] \*= -r[l];

result[j + 1] += temp[j];

}

for (int k = 0; k < m\_size; k++) {

temp[k] = result[k];

}

}

int j{ 0 };

for (; j < m\_size - i - 1; j++) {

for (int k = 0; k < m\_size - 1; k++)

{

if (k == 0) {

temp[k] = 0;

}

temp[k + 1] = result[k];

}

for (int k = 0; k < m\_size; k++)

result[k] = temp[k];

}

double division = pow(interpolation\_step, i);

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

if (result[j] != 0) {

result[j] /= division;

result[j] \*= difference[i];

result[j] /= Factorial(i);

}

solution[j] += result[j];

}

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

temp[i] = 1;

if (i == 0)

result[i] = 1;

else

result[i] = 0;

}

}

return solution;

}

double CInterpolation::Factorial(int x) const{

int result{ 0 };

if (x == 0) {

return 1;

}

else {

result = x \* Factorial(x - 1);

}

return result;

}

vector<double> CInterpolation::Newthon(double x) const{

vector<double> result;

bool isEquidistant{ true };

double difference{ 0 };

difference = m\_x[1] - m\_x[0];

double interpolation\_step = fabs(difference);

if ((x - m\_x[0]) < (m\_x[m\_x.size() - 1] - x)) {

result = Forward(interpolation\_step);

}

return result;

}

double CInterpolation::FindByNewthon(double x) const {

vector<double> polynom = Newthon(x);

double result{ 0 };

for (int i = polynom.size() - 1; i >= 0; i--) {

result = polynom[i] \* pow(x, i);

}

return result;

}

Lab\_08\_NM.cpp:

#include <iostream>

#include "CInterpolation.h"

int main()

{

const size\_t m\_size{ 10 };

double m\_x[m\_size]{ 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 };

double m\_y[m\_size]{ 1.758203, 1.738744, 1.718369, 1.697320, 1.675834, 1.654140, 1.632460, 1.611005, 1.589975, 1.569559 };

const double x{ 0.15 };

CInterpolation in(m\_x, m\_y, m\_size);

vector<double> lagrange\_polynom = in.Lagrange();

double lagrange\_result = in.FindByLagrange(x);

vector<double> newthon\_polynom = in.Newthon(x);

double newthon\_result = in.FindByNewthon(x);

cout << "LAGRANGE:\n\nPolynom: ";

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

if (lagrange\_polynom[i] >= 0)

cout << "+";

cout << lagrange\_polynom[i] << "x^" << m\_size - i - 1 << " ";

}

cout << "\nSolution for " << x << ": " << lagrange\_result << "\n\n\n";

cout << "NEWTHON:\n\nPolynom: ";

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

if (newthon\_polynom[i] >= 0)

cout << "+";

cout << newthon\_polynom[i] << "x^" << m\_size - i - 1 << " ";

}

cout << "\nSolution for " << x << ": " << newthon\_result << "\n\n\n";

}