**МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ**

**НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС**

**«ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ»**

**НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»**

**КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ**

**Лабораторна робота №5**

**з курсу «Чисельні методи»**

**тема: «Інтерполяційні поліноми»**

**Виконав: студент 2 курсу**

**групи КА-23**

**Деундяк О.В.**

**Прийняв: Коновалюк М. М.**

**Київ – 2014р.**

**Задача 7**

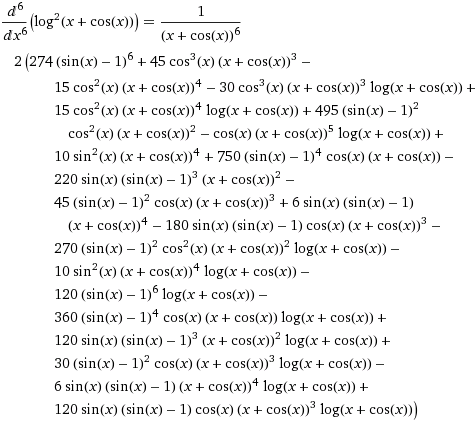
Функція:

**Допрограмовий етап**

Таблиця значень:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *X* | *0* | *1* | *2* | *3* | *4* |
| *F(x)* | *0* | *0.1866056* | *0.2114718* | *0.4873973* | *1.458955024* |

6 похідна:



Супремум 6 похідної функції на відрізку [0,4]

Отже, похибка інтерполяції поліномом:

**Текст програми:**

**Matrix.h**

#pragma once

#include <vector>

#include <string>

#include <fstream>

using std::vector;

using std::string;

using std::ifstream;

typedef double Item;

class Matrix

{

vector <vector <Item>> A;

size\_t n;

vector<Item> SolveL(const vector<Item> &b) const;

vector<Item> SolveU(const vector<Item> &b) const;

public:

Matrix(size\_t dimension, Item fill);

Matrix(size\_t dimension);

Matrix(size\_t dimension, ifstream &file);

Matrix(const vector<vector<Item>> &A);

Item Determinant() const;

bool DiagDom() const;

Matrix Inverse();

Matrix operator\*(const Matrix &other) const;

vector<Item> operator\*(const vector<Item> &vec) const;

vector<Item> Solve(vector<Item> b) const;

vector<Item> Solve2(const vector<Item> &b) const;

friend std::ostream &operator<<(std::ostream &os, const Matrix &m);

void Show() const;

void LU(Matrix &L, Matrix &U, vector<Item> &b) const;

};

**Matrix.cpp**

#include "Matrix.h"

#include <iomanip>

#include <iostream>

#include <iomanip>

#include <fstream>

#include <exception>

#include <algorithm>

#include "Vector.h"

extern const double eps;

const long MAXIT = 1000;

using std::cin;

//using std::cout;

using std::endl;

using std::ifstream;

Matrix::Matrix(size\_t n\_, Item fill) :

n(n\_)

{

for (size\_t i = 1; i <= n; i++)

{

vector<Item> T;

for (size\_t j = 1; j <= n; j++)

T.push\_back(fill);

A.push\_back(T);

}

}

Matrix::Matrix(const vector<vector<Item>> &A\_) : A(A\_), n(A.size())

{

size\_t n = A.size();

for (size\_t i = 0; i < n; i++)

if (A[i].size() != n)

throw std::invalid\_argument("Not square matrix");

}

Matrix::Matrix(size\_t n\_, ifstream &file) :

n(n\_)

{

for (size\_t i = 1; i <= n; i++)

{

vector<Item> T;

Item t;

for (size\_t j = 1; j <= n; j++)

{

file >> t;

T.push\_back(t);

}

A.push\_back(T);

}

}

void Matrix::LU(Matrix &L, Matrix &U, vector<Item> &v) const

{

U = \*this;

for (size\_t i = 0; i < n; i++) // for all diagonal elements

{

{ // find max element in raw and place it in diagonal position

Item max = abs(U.A[i][i]);

size\_t maxi = i;

for (size\_t j = i; j < n; j++)

if (abs(U.A[j][i]) > max)

{

max = abs(U.A[j][i]);

maxi = j;

}

if (max < eps)

throw std::runtime\_error("LU decomposition doesn't exists");

std::swap(U.A[i], U.A[maxi]);

std::swap(L.A[i], L.A[maxi]);

std::swap(v[i], v[maxi]);

}

for (size\_t j = i; j < n; j++)

{

L.A[j][i] = U.A[j][i] / U.A[i][i];

if (i !=j)

U.A[j] = U.A[j] + (-L.A[j][i]) \* U.A[i];

}

}

}

Matrix Matrix::operator\*(const Matrix &other) const

{

Matrix P(n, 0);

for (size\_t row = 0; row < n; row++)

for (size\_t col = 0; col < n; col++)

for (size\_t inner = 0; inner < n; inner++)

P.A[row][col] += A[row][inner] \* other.A[inner][col];

return P;

}

vector<Item> Matrix::SolveL(const vector<Item> &b) const

{

vector<Item> r;

r.reserve(n);

Item t;

for (size\_t i = 0; i < n; i++)

{

t = 0;

for (size\_t j = 0; j < i; j++)

t += A[i][j] \* r[j];

r.push\_back((b[i] - t) / A[i][i]);

}

return r;

}

vector<Item> Matrix::SolveU(const vector<Item> &b) const

{

vector<Item> r;

r.resize(n);

Item t;

for (size\_t i = n; i > 0; i--)

{

t = 0;

for (size\_t j = n-1; j > i - 1; j--)

t += A[i - 1][j] \* r[j];

r[i - 1] = (b[i - 1] - t) / A[i - 1][i - 1];

}

return r;

}

vector<Item> Matrix::Solve(vector<Item> b) const

{

Matrix L(n, 0), U(n, 0);

LU(L, U, b);

return U.SolveU(L.SolveL(b));

}

Item Matrix::Determinant() const

{

Matrix L(n, 0), U(n, 0);

Item det = 1;

LU(L, U, vector<Item>(n, 0));

for (size\_t i = 0; i < n; i++)

det \*= U.A[i][i];

return det;

}

Matrix Matrix::Inverse()

{

Matrix m\_inv(n, 0);

vector<Item> b, r;

b.resize(n);

for (size\_t i = 0; i < n; i++)

{

for (size\_t j = 0; j < n; j++)

b[j] = (i == j) ? 1 : 0;

r = Solve(b);

for (size\_t j = 0; j < n; j++)

(m\_inv.A)[j][i] = r[j];

}

return m\_inv;

}

vector<Item> Matrix::operator\*(const vector<Item> &vec) const

{

vector<Item> res;

for (size\_t i = 0; i < n; i++)

{

Item r = 0;

for (size\_t j = 0; j < n; j++)

r += A[i][j] \* vec[j];

res.push\_back(r);

}

return res;

}

bool Matrix::DiagDom() const

{

Item s;

for (size\_t i = 0; i < n; i++)

{

s = 0;

for (size\_t j = 0; j < n; j++)

s += (i != j) ? abs(A[i][j]) : 0.0;

if (abs(A[i][i]) < s)

return false;

}

return true;

}

Item operator\*(const vector<Item> &v1, const vector<Item> &v2)

{

size\_t l = std::min(v1.size(), v2.size());

Item r = 0;

for (size\_t i = 0; i < l; i++)

r += v1[i] \* v2[i];

return r;

}

vector<Item> operator-(const vector<Item> &v1, const vector<Item> &v2)

{

size\_t l = std::min(v1.size(), v2.size());

vector<Item> r;

r.resize(l);

for (size\_t i = 0; i < l; i++)

r[i] = v1[i] - v2[i];

return r;

}

vector<Item> Matrix::Solve2(const vector<double> &b) const

{

Matrix L = \*this, U = \*this;

for (size\_t i = 0; i < n; i++)

{

for (size\_t j = i + 1; j < n; j++)

L.A[i][j] = 0;

}

for (size\_t i = 0; i < n; i++)

for (size\_t j = 0; j < n; j++)

U.A[i][j] = (j <= i) ? 0 : -A[i][j];

vector<Item> x = b, e = b - (\*this)\*x;;

Item ep = sqrt(e\*e);

long it = 1;

while (ep > eps)

{

x = L.SolveL(U\*x + b);

e = b - (\*this)\*x;

ep = sqrt(e \* e);

++it;

if (it > MAXIT)

throw std::exception("Too many iterations");

}

return x;

}

std::ostream &operator<<(std::ostream &os, const Matrix &m)

{

os << std::left;

for (size\_t i = 0; i < m.A.size(); i++)

{

for (size\_t j = 0; j < m.A[i].size(); j++)

os << std::setw(9) << m.A[i][j] << " ";

os << std::endl;

}

return os;

}

**Vector.h**

#pragma once

#include "Polynomial.h"

#include <vector>

vector<Item> operator+(const vector<Item> &v1, const vector<Item> &v2);

vector<Item> operator\*(const vector<Item> v, double n);

vector<Item> operator\*(double n, const vector<Item> v);

std::ostream &operator<<(std::ostream &os, const vector<Item> &v);

**Vector.cpp**

#include "Polynomial.h"

#include "Vector.h"

#include <algorithm>

vector<Item> operator+(const vector<Item> &v1, const vector<Item> &v2)

{

size\_t lmin = v1.size(), lmax = v2.size();

if (lmin > lmax)

std::swap(lmin, lmax);

vector<Item> res(lmax);

for (size\_t i = 0; i < lmin; i++)

res[i] = v1[i] + v2[i];

for (size\_t i = lmin; i < lmax; i++)

res[i] = (lmax == v1.size()) ? v1[i] : v2[i];

return res;

}

vector<Item> operator\*(const vector<Item> v, double n)

{

vector<Item> r(v.size());

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

r[i] = v[i] \* n;

return r;

}

vector<Item> operator\*(double n, const vector<Item> v)

{

return v \* n;

}

std::ostream &operator<<(std::ostream &os, const vector<Item> &v)

{

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

os << v[i] << " ";

return os;

}

**Polynomial.h**

#pragma once

#include <vector>

#include <iostream>

using std::vector;

extern const double eps;

typedef double Item;

class Polynomial

{

vector<Item> vec; // from x^0 to x^n

public:

// Constructors

explicit Polynomial(const vector<Item> &A);

Item operator()(Item x) const; // Gives value in point x

Polynomial Derivative() const;

// Arithmetic operators

Polynomial operator\*(Item n) const;

friend Polynomial operator\*(Item n, const Polynomial p) { return p\*n; }

Polynomial operator\*(const Polynomial &other) const;

Polynomial Polynomial::operator+(const Polynomial &other) const;

vector<Item> C() const { return vec; };

friend std::ostream &operator<<(std::ostream &os, const Polynomial &p);

};

**Polynomial.cpp**

#include "Polynomial.h"

#include "Vector.h"

#include <iostream>

Polynomial::Polynomial(const vector<Item> &A) : vec(A)

{

}

Item Polynomial::operator()(Item x) const

{

Item r = 0, p = 1;

for (auto i = vec.begin(); i != vec.end(); i++)

{

r += \*i \* p;

p \*= x;

}

return r;

}

Polynomial Polynomial::operator+(const Polynomial &other) const

{

return Polynomial(vec + other.vec);

}

Polynomial Polynomial::operator\*(double n) const

{

return Polynomial(vec \* n);

}

std::ostream &operator<<(std::ostream &os, const Polynomial &p)

{

os.setf(std::ios::fixed, std::ios::floatfield);

std::streamsize pr = os.precision(10);

for (int i = p.vec.size() - 1; i >= 0; i--)

if (abs(p.vec[i]) > eps)

{

os << std::showpos << p.vec[i] << std::noshowpos;

if (i > 0)

os << "x";

if (i > 1)

os << "^" << i;

os << ' ';

}

os.precision(pr);

return os;

}

Polynomial Polynomial::operator\*(const Polynomial &other) const

{

vector<Item> res(vec.size() + other.vec.size(), 0);

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

for (size\_t j = 0; j < other.vec.size(); j++)

res[i + j] += vec[i] \* other.vec[j];

if (!res.empty())

{

auto i = res.end() - 1;

while (!res.empty() && abs(\*i) < eps)

i = res.erase(i) - 1;

}

return Polynomial(res);

}

Polynomial Polynomial::Derivative() const

{

vector<Item> der(1, 0);

for (size\_t i = 1; i < vec.size(); i++)

{

der[i - 1] = i \* vec[i];

}

return Polynomial(der);

}

**Spline.h**

#pragma once

#include"Polynomial.h"

#include <vector>

using std::vector;

class Spline

{

vector<Polynomial> P;

vector<Item> X;

public:

Spline(Item x);

void Add(const Polynomial &p, Item x);

Item operator()(Item x) const; // Gives value in point x

friend std::ostream &operator<<(std::ostream &os, const Spline &p);

};

**Spline.cpp**

#include "Spline.h"

#include <algorithm>

#include <iostream>

Spline::Spline(Item x)

{

X.push\_back(x);

}

void Spline::Add(const Polynomial &p, Item x)

{

P.push\_back(p);

X.push\_back(x);

}

std::ostream &operator<<(std::ostream &os, const Spline &s)

{

for (size\_t i = 0; i < s.P.size(); i++)

{

os << s.P[i] << " on [" << s.X[i] << "," << s.X[i + 1] << "]\n";

}

return os;

}

Item Spline::operator()(Item x) const

{

size\_t i;

for (i = 0; i < P.size() - 1 && x > X[i + 1]; i++);

return P[i](x);

}

**Interpolation.h**

#pragma once

#include "Polynomial.h"

#include "Spline.h"

#include <vector>

using std::vector;

class Interpolation

{

Item OriginalF(Item x) const;

vector<Item> X\_;

vector<Item> Y\_;

Polynomial Newton1\_(const vector<Item> &X, const vector<Item> &Y) const;

Polynomial Lagrange\_(const vector<Item> &X, const vector<Item> &Y) const;

public:

Interpolation();

Interpolation(Item a, Item b, int n);

Polynomial Lagrange() const;

Polynomial Newton1() const;

Polynomial Newton2() const;

Item Left() const { return X\_[0]; };

Item Right() const { return X\_[X\_.size() - 1]; };

size\_t n() const { return X\_.size(); };

Spline Spline2() const;

};

**Interpolation.cpp**

#include "Interpolation.h"

#include "Vector.h"

#include "Polynomial.h"

#include "Spline.h"

#include "Matrix.h"

#include <fstream>

#include <iostream>

#include <map>

//#include <cmath>

const double eps = 1E-6;

const std::string file = "input.txt";

Interpolation::Interpolation()

{

std::ifstream cin(file);

int n;

if (!(cin >> n))

throw std::runtime\_error("Error reading from the file");

for (int i = 0; i < n && cin.good(); i++)

{

Item x, y;

cin >> x >> y;

X\_.push\_back(x);

Y\_.push\_back(y);

}

}

Interpolation::Interpolation(Item a, Item b, int n)

{

Item x = a, dx = (b - a) / n;

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

{

X\_.push\_back(x);

Y\_.push\_back(OriginalF(x));

x += dx;

}

}

Item Interpolation::OriginalF(double x) const

{

return pow(log(x + cos(x)), 2);

}

Polynomial Interpolation::Lagrange() const

{

return Lagrange\_(X\_, Y\_);

}

Polynomial Interpolation::Lagrange\_(const vector<Item> &X, const vector<Item> &Y) const

{

size\_t n = X.size();

Polynomial L({ 0 });

for (size\_t i = 0; i < n; i++)

{

Polynomial l({ 1 });

for (size\_t j = 0; j < n; j++)

if (i != j)

l = l \* Polynomial({ -X[j] / (X[i] - X[j]), 1 / (X[i] - X[j])});

L = L + Y[i] \* l;

}

return L;

}

Polynomial Interpolation::Newton1() const

{

return Newton1\_(X\_, Y\_);

}

Polynomial Interpolation::Newton2() const

{

vector<Item> X1(X\_.size()), Y1(X\_.size());

for (size\_t i = 0; i < X1.size(); i++)

{

X1[i] = X\_[X\_.size() - i - 1];

Y1[i] = Y\_[X\_.size() - i - 1];

}

return Newton1\_(X1, Y1);

}

Polynomial Interpolation::Newton1\_(const vector<Item> &X, const vector<Item> &Y) const

{

size\_t n = X.size(), i = 0;

vector<vector<Item>> Z(n, vector<Item>(n));

for (size\_t i = 0; i < n; i++)

Z[i][0] = Y[i];

for (size\_t i = 1; i < n; i++)

{

size\_t it1 = i;

for (size\_t j = 0; j < n - i; j++, it1++)

Z[j][i] = (Z[j][i - 1] - Z[j + 1][i - 1]) / (X[j] - X[it1]);

}

Polynomial L({ Z[0][0] }), l({ 1 });

for (size\_t i = 1; i < n; i++)

{

l = l \* Polynomial({-X[i - 1], 1});

L = L + Z[0][i] \* l;

}

return L;

}

Spline Interpolation::Spline2() const

{

const int c = 3;

size\_t n = X\_.size(), k = c \* (n - 1);

vector<vector<Item>> A(k, vector<Item>(k, 0));

vector<Item> b(k, 0);

Spline S(X\_[0]);

for (size\_t i = 0; i < n - 1; i++) // values

{

for (size\_t j = 0; j < c; j++)

{

A[i \* 2][c \* i + j] = pow(X\_[i], j);

A[i \* 2 + 1][c \* i + j] = pow(X\_[i + 1], j);

}

b[i \* 2] = Y\_[i];

b[i \* 2 + 1] = Y\_[i + 1];

}

for (size\_t i = 1; i < n - 1; i++) // derivatives

{

for (size\_t j = 1; j < c; j++)

{

double a = pow(X\_[i], j - 1), b = j \* a;

A[i + 2 \* n - 3][j + c\*(i - 1)] = j \* pow(X\_[i], j - 1);

}

for (size\_t j = 1; j < c; j++)

A[i + 2\*n - 3][j + c \* i] = -int(j) \* pow(X\_[i], j - 1);

b[i + 2\*n - 3] = 0;

}

for (size\_t j = 1; j < c; j++) // edge condition

A[3 \* n - 4][j] = j \* pow(X\_[0], j - 1);

b[3 \* n - 4] = 0;

Matrix m(A);

vector<Item> r = m.Solve(b), r1;

r1.resize(c);

for (size\_t i = 0; i < n - 1; i++)

{

for (size\_t j = 0; j < c; j++)

r1[j] = r[i \* c + j];

S.Add(Polynomial(r1), X\_[i + 1]);

}

return S;

}

**main.cpp**

#include "Interpolation.h"

#include <iostream>

#include <fstream>

using std::endl;

double F(double x)

{

return pow(log(x + cos(x)), 2);

}

int main()

{

Interpolation inter;

Polynomial p({ 0 });

Spline s(0);

try

{

p = inter.Lagrange();

s = inter.Spline2();

std::cout << "Lagrange polynomial:\n" << inter.Lagrange() << endl << endl

<< "Newton forawrd polynomial\n" << inter.Newton1() << endl << endl

<< "Newton backward polynomial\n" << inter.Newton2() << endl << endl << endl

<< "Quadratic spline:\n" << inter.Spline2() << endl;

}

catch (std::exception e)

{

std::cout << e.what();

}

std::ofstream debug("debug.txt");

{

debug << "Preciosion analysis:\nNewton polynomial\n";

Item x = inter.Left(), dx = (inter.Right() - inter.Left()) / (inter.n() – 1) / 5;

while (x - inter.Right() < eps)

{

debug.precision(5);

debug.setf(std::ios::fixed, std::ios::floatfield);

debug << "x=" << x << " y=" << F(x) << " Interpol=" << p(x) << " delta=" << abs(F(x) - p(x)) << endl;

x += dx;

}

}

{

debug << "\nQuadratic spline polynomial\n";

Item x = inter.Left(), dx = (inter.Right() - inter.Left()) / (inter.n() – 1) / 5;

while (x - inter.Right() < eps)

{

debug.precision(5);

debug.setf(std::ios::fixed, std::ios::floatfield);

debug << "x=" << x << " y=" << F(x) << " Interpol=" << s(x) << " delta=" << abs(F(x) - s(x)) << endl;

x += dx;

}

}

std::cin.get();

}

**Результати роботи програми**

**Lagrange polynomial:**

**+0.0013239260x^4 +0.0608562273x^3 -0.2727058640x^2 +0.3971313107x**

**Newton forawrd polynomial**

**+0.0013239260x^4 +0.0608562273x^3 -0.2727058640x^2 +0.3971313107x**

**Newton backward polynomial**

**+0.0013239260x^4 +0.0608562273x^3 -0.2727058640x^2 +0.3971313107x**

**Quadratic spline:**

**+0.1866056000x^2 on [0.000000,1.000000]**

**-0.3483450000x^2 +1.0699012000x -0.5349506000 on [1.000000,2.000000]**

**+0.5994043000x^2 -2.7210960000x +3.2560466000 on [2.000000,3.000000]**

**+0.0962279240x^2 +0.2979622560x -1.2725407840 on [3.000000,4.000000]**

Precision analysis:

Newton polynomial

x=0.00000 y=0.00000 Interpol=0.00000 delta=0.00000

x=0.20000 y=0.02741 Interpol=0.06901 delta=0.04159

x=0.40000 y=0.07753 Interpol=0.11915 delta=0.04162

x=0.60000 y=0.12560 Interpol=0.15342 delta=0.02782

x=0.80000 y=0.16262 Interpol=0.17487 delta=0.01225

x=1.00000 y=0.18661 Interpol=0.18661 delta=0.00000

x=1.20000 y=0.19909 Interpol=0.19177 delta=0.00732

x=1.40000 y=0.20345 Interpol=0.19356 delta=0.00989

x=1.60000 y=0.20393 Interpol=0.19523 delta=0.00870

x=1.80000 y=0.20508 Interpol=0.20008 delta=0.00500

x=2.00000 y=0.21147 Interpol=0.21147 delta=0.00000

x=2.20000 y=0.22769 Interpol=0.23280 delta=0.00512

x=2.40000 y=0.25846 Interpol=0.26753 delta=0.00907

x=2.60000 y=0.30877 Interpol=0.31916 delta=0.01039

x=2.80000 y=0.38363 Interpol=0.39125 delta=0.00761

x=3.00000 y=0.48740 Interpol=0.48740 delta=0.00000

x=3.20000 y=0.62289 Interpol=0.61127 delta=0.01161

x=3.40000 y=0.79069 Interpol=0.76658 delta=0.02411

x=3.60000 y=0.98893 Interpol=0.95708 delta=0.03185

x=3.80000 y=1.21356 Interpol=1.18659 delta=0.02698

x=4.00000 y=1.45896 Interpol=1.45896 delta=0.00000

Quadratic spline polynomial

x=0.00000 y=0.00000 Interpol=0.00000 delta=0.00000

x=0.20000 y=0.02741 Interpol=0.00746 delta=0.01995

x=0.40000 y=0.07753 Interpol=0.02986 delta=0.04767

x=0.60000 y=0.12560 Interpol=0.06718 delta=0.05843

x=0.80000 y=0.16262 Interpol=0.11943 delta=0.04320

x=1.00000 y=0.18661 Interpol=0.18661 delta=0.00000

x=1.20000 y=0.19909 Interpol=0.24731 delta=0.04822

x=1.40000 y=0.20345 Interpol=0.28015 delta=0.07670

x=1.60000 y=0.20393 Interpol=0.28513 delta=0.08120

x=1.80000 y=0.20508 Interpol=0.26223 delta=0.05716

x=2.00000 y=0.21147 Interpol=0.21147 delta=0.00000

x=2.20000 y=0.22769 Interpol=0.17075 delta=0.05693

x=2.40000 y=0.25846 Interpol=0.17798 delta=0.08047

x=2.60000 y=0.30877 Interpol=0.23317 delta=0.07560

x=2.80000 y=0.38363 Interpol=0.33631 delta=0.04733

x=3.00000 y=0.48740 Interpol=0.48740 delta=0.00000

x=3.20000 y=0.62289 Interpol=0.66631 delta=0.04342

x=3.40000 y=0.79069 Interpol=0.85293 delta=0.06223

x=3.60000 y=0.98893 Interpol=1.04724 delta=0.05830

x=3.80000 y=1.21356 Interpol=1.24925 delta=0.03568

x=4.00000 y=1.45896 Interpol=1.45896 delta=0.00000

**Висновки:**

Поліноми Лагранжа, Н’ютона вперед та назад дали один і той же результат, отже і однакову похибки. Інтерполяція квадратним сплайном в порівнянні дала дещо більшу похибку, але того ж порядку. Тому всі методу дають непогані результати інтерполяції