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 $y=\varphi(x)$ представимо у вигляді

$$\varphi(x) = L_n(x) = \sum_{i=0}^{n} P_i(x) f(x_i),$$

$$P_i(x_j) = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}, \quad i, j = 0, 1, ..., n.$$

Оскільки точки $x_0, x_1, ..., x_{i-1}, x_{i+1}, .., x_n$ є коренями многочлена $P_i(x)$, то його можна записати наступним чином

$$P_i(x) = \frac{(x-x_0)(x-x_1)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_0)(x_i-x_1)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)},$$

а наближена функція $\varphi(x)$, яку називають *інтерполяційним многочленом Лагранжа*, матиме вигляд

$$\varphi(x) = L_n(x) = \sum_{i=0}^{n} \frac{(x-x_0)(x-x_1)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_0)(x_i-x_1)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)} f(x_i).$$

Інтерполяційний поліном Ньютона

Іншим підходом до задачі інтерполяції є метод Ньютона (метод розділених різниць). Нехай для функції $y = f(x)$ задано її значення в точках $x_0, x_1, ..., x_n$. Треба побудувати такий поліном $P_n(x)$ степеня не вище від n, значення якого у вузлах інтерполювання збігаються із значенням функції $y = f(x)$, тобто

$$P_n(x_i) = y_i \ (i = 0, 1, ..., n).$$

Поліном $P_n(x)$ будемо шукати у вигляді

$$P_n(x) = f(x_0) + P_n(x_0, x_1)(x-x_0) + P_n(x_0, x_1, x_2)(x-x_0)(x-x_1) +$$
$$+ ... + P_n(x_0, x_1, ..., x_n)(x-x_0)(x-x_1)...(x-x_{n-1}),$$

$$P_n(x_0, x_1, ..., x_n) = \frac{P_n(x_1, ..., x_n) - P_n(x_0, ..., x_{n-1})}{x_n - x_0} -$$ розділена різниця $n-$го порядку.

Нехай вузли інтерполяції утворюють арифметичну прогресію $x_i = x_0 + ih$ $(i = 0,1,2,...,n)$, $h-$крок інтерполяції.

Таким чином, скінченну різницю $n-$го порядку можна записати у вигляді

$$\Delta^n f(x_i) = \Delta(\Delta^{n-1} f(x_i)) = \Delta^{n-1} f(x_{i+1}) - \Delta^{n-1} f(x_i).$$

Розділену різницю $n-$го порядку можна виразити через скінченну різницю $n-$го порядку

$$P_n(x_0, x_1, ..., x_n) = \frac{\Delta^n f(x_0)}{n! h^n}.$$

Тоді наведений вище поліном можна записати у вигляді

$$P_n(x) = f(x_0) + \frac{\Delta f(x_0)(x-x_0)}{1! h} + \frac{\Delta^2 f(x_0)(x-x_0)(x-x_1)}{2! h^2} + \\ + ... + \frac{\Delta^n f(x_0)(x-x_0)(x-x_1)...(x-x_{n-1})}{n! h^n},$$

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 x_0.

Таблично задана функція:

x	f(x)	x	f(x)
0,0	I,758203	0,5	I,654I40
0,I	I,738744	0,6	I,632460
0,2	I,7I8369	0,7	I,6II005
0,3	I,697320	0,8	I,589975
0,4	I,675834	0,9	I,569559

Результат виконання програми

```
Microsoft Visual Studio Debu
LAGRANGE:

Polynom: -0.0854277x^9 +0.352183x^8 -0.609292x^7 +0.574306x^6 -0.32136x^5 +0.105962x^4 +0.0206048x^3 -0.0560174x^2 -0.189273x^1 +1.7582x^0
Solution for 0.15: -0.0854277


NEWTHON:

Polynom: -0.0854277x^9 +0.352183x^8 -0.609292x^7 +0.574306x^6 -0.32136x^5 +0.105962x^4 +0.0206048x^3 -0.0560174x^2 -0.189273x^1 +1.7582x^0
Solution for 0.15: -0.0854277
```

Висновки

У результаті виконання лабораторної роботи обчислено значення таблично заданої функції у точці x_0 (x_0 = 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";
}
```