**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по практической работе №5**

**по дисциплине «Вычислительная математика»**

**Тема: Интерполяционные и аппроксимационные формулы для равноотстоящих узлов.**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 1304 |  | Чернякова В.А. |
| Преподаватель |  | Попова Е.В. |

Санкт-Петербург

2022

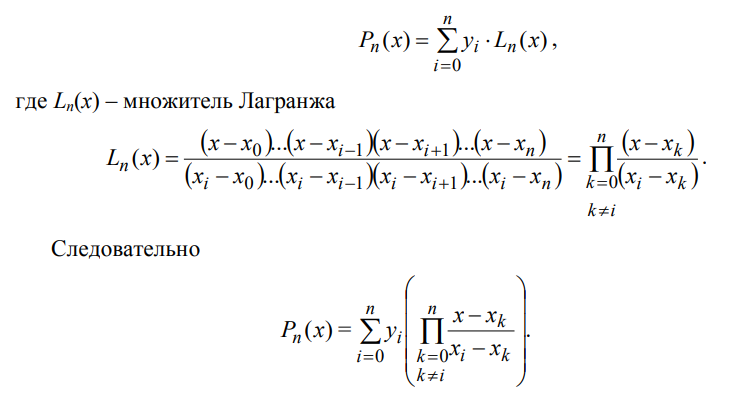
## Цель работы.

## Исследование методов интерполяции и аппроксимации для равноотстоящих узлов с последующей реализацией на одном из языков программирования.

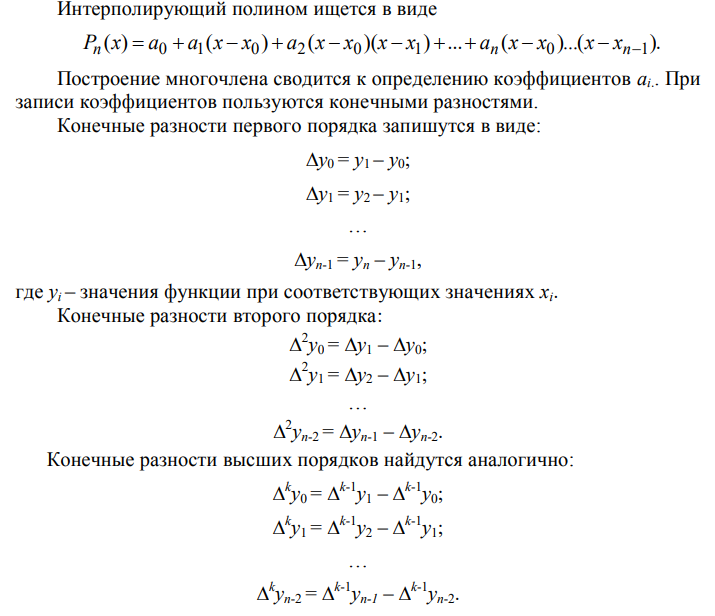
## Основные теоретические положения.

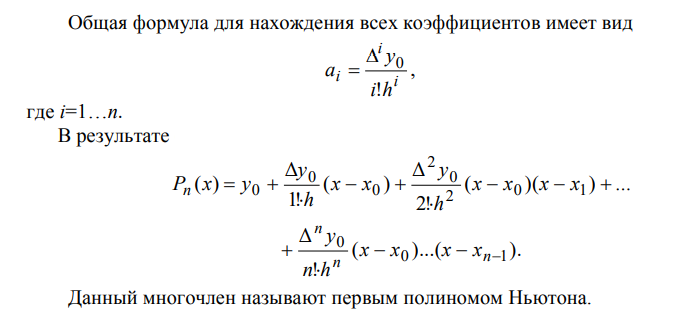
Значения функции заданы в точках необходимо найти промежуточные значения.

Интерполяционный многочлен Лагранжа.

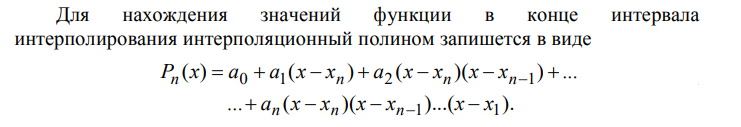


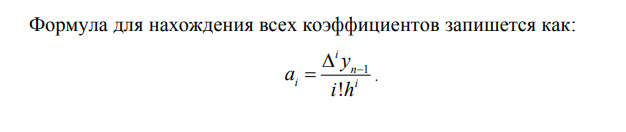
Первый интерполяционный многочлен Ньютона. Точка интерполирования находится в начале таблицы.

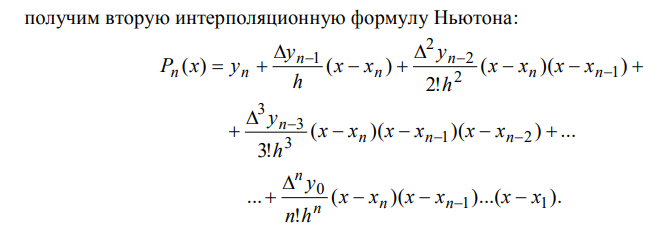




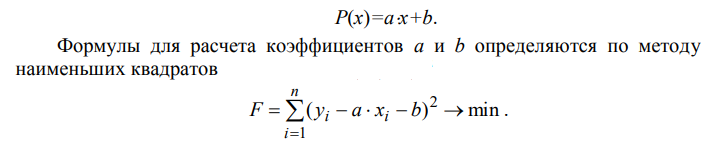
Второй интерполяционный многочлен Ньютона. Точка интерполирования находится в конце таблицы.

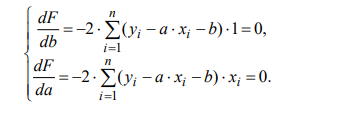


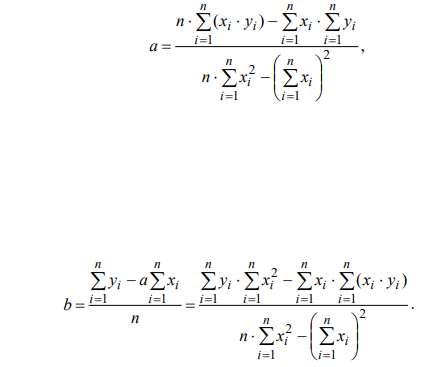




Аппроксимация функции. Необходимо найти эмпирическую формулу, значения которой при мало бы отличались от входных данных. Будет использоваться линейная аппроксимация, при которой данные описываются линейной зависимостью







## Этапы выполнения практической работы.

1. Выбрать три точки, не входящие таблицу данных в начале, конце, по середине таблицы.

2. Составить необходимые подпрограммы-функции.

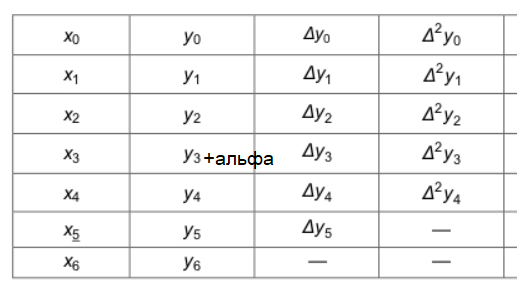
3. Составить головную программу, содержащую обращение к

соответствующим подпрограммам и осуществляющую печать результатов

4. Используя многочлен Лагранжа, найти приближенные значения функции. Подсчитать точные значения, абсолютную погрешность, относительную погрешность, верные и значащие цифры.

5. Используя многочлены Ньютона найти все, что перечислено в предыдущем пункте, дополнительно – в процентном соотношении точку приемлемого использования первого многочлена по сравнению со вторым многочленом при работе с точкой в начале таблицы и наоборот.

6. Составить таблицу конечных разностей до второго порядка. К одному из значений прибавить погрешность (), превосходящую допустимую. Показать, как при этом изменятся конечные разности большего порядка.



7. При аппроксимировании функции использовать для нахождения новых значений те же три точки. Рассчитать все, что указано в пункте 4.

8. Составить сводную таблицу по результатам исследования.

## Выполнение практической работы.

**Вариант 2**

**y=cos((pi\*x)/2)**

**Область определения (-∞;+∞)**

1. Выберем три точки, не входящие таблицу данных в начале, конце, по середине таблицы.

Дано:

Выбранные точки: x = 0.03, 0.55, 1.21

1. Составим необходимые подпрограммы-функции.

**F.** Функция вычисляющая значение cos((pi\*x)/2). Возвращает значение заданной функции в выбранной точке.

*double F(double x){*

*return cos((M\_PI\*x)/2);*

*}*

**LagrangePolynomial.** Функция выполняющая вычисление полинома Лагранжа. Она обрабатывает массивы, в которых хранятся соответствующие координаты x и y, а также число x, для которого необходимо найти значение функции. Переменная n отвечает за количество слагаемых. На основе теоретических положений реализован алгоритм.

*double LagrangePolynomial(double Xcoord[], double Ycoord[], double x, int n) {*

*double P = 0;*

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

*double L = 1;*

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

*if (j != i)*

*L = L \* (x - arrX[j]) / (arrX[i] - arrX[j]);*

*}*

*P = P + arrY[i] \* L;*

*}*

*return P;*

*}*

**NewtonPolynomial1.** Функция выполняющая вычисление первого полинома Ньютона. Она обрабатывает массивы, в которых хранятся соответствующие координаты x и y, а также число x, для которого необходимо найти значение функции. Переменная n отвечает за количество итераций. Переменная h – шаг узлов. На основе теоретических положений реализован алгоритм.

*double NewtonPolynomial1(double Xcoord[], double Ycoord[], double x, int n, double h) {*

*double P = 0;*

*double matrixY[n][n];*

*double A[n];*

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

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

*if (i == 0) {*

*matrixY[i][j] = Ycoord[j + 1] - Ycoord[j];*

*matrixY[i][j + 1] = 0;*

*} else {*

*if (matrixY[i - 1][j + 1] != 0) {*

*matrixY[i][j] = matrixY[i - 1][j + 1] - matrixY[i - 1][j];*

*matrixY[i][j + 1] = 0;*

*} else {*

*matrixY[i][j] = 0;*

*matrixY[i][j + 1] = 0;*

*}*

*}*

*}*

*A[i] = matrixY[i][0];*

*}*

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

*if (i == 0)*

*P += arrY[0];*

*else {*

*double el = 1;*

*double elh = 1;*

*for (int k = 0; k < i; k++) {*

*el \*= (x - Xcoord[k]);*

*elh \*= h;*

*}*

*P += ((el \* A[i - 1]) / (elh \* factorial(i)));*

*}*

*}*

*return P;*

*}*

**NewtonPolynomial2.** Функция выполняющая вычисление второго полинома Ньютона. Она обрабатывает массивы, в которых хранятся соответствующие координаты x и y, а также число x, для которого необходимо найти значение функции. Переменная n отвечает за количество итераций. Переменная h – шаг узлов. На основе теоретических положений реализован алгоритм.

*double NewtonPolynomial2(double Xcoord[], double Ycoord[], double x, int n, double h) {*

*double P = 0;*

*double matrixY[n][n],*

*double A[n];*

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

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

*if (i == 0) {*

*matrixY[i][j] = Ycoord[j + 1] - Ycoord[j];*

*matrixY[i][j + 1] = 0;*

*} else {*

*if (matrixY[i - 1][j + 1] != 0) {*

*matrixY[i][j] = matrixY[i - 1][j + 1] - matrixY[i - 1][j];*

*matrixY[i][j + 1] = 0;*

*} else {*

*matrixY[i][j] = 0;*

*matrixY[i][j+1] = 0;*

*}*

*}*

*}*

*A[i] = matrixY[i][n - 2 - i];*

*}*

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

*if (i == 0) P += Ycoord[n - 1];*

*else {*

*double el = 1;*

*double elh = 1;*

*for (int k = n; k > (n-i); k--) {*

*el \*= (x - Xcoord[k-1]);*

*}*

*for (int k = 0; k < i; k++) elh \*= h;*

*P += ((el \* A[i - 1]) / (elh \* factorial(i)));*

*}*

*}*

*return P;*

*}*

**Factorial.** Функция вычисляющая факториал числа. Так как для вычисления обоих полиномов Ньютона необходимо вычисление факториала была создана данная функция. Сокращает число кода.

*int Factorial(int i) {*

*double res = 1;*

*for (int iter = i; iter > 1; iter--)*

*res \*= iter;*

*return res;*

*}*

**Approximation.** Функция линейной аппроксимации. Она обрабатывает массивы, в которых хранятся соответствующие координаты x и y, а также число x, для которого необходимо найти значение функции. Переменная n отвечает за количество итераций. На основе теоретических положений реализован алгоритм.

*double Approximation(double Xcoord[], double Ycoord[], double x, int n) {*

*double a,b;*

*double sumX = 0, sumY = 0, sumXY = 0, sumDX = 0;*

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

*sumX += Xcoord[i];*

*sumY += Ycoord[i];*

*sumXY += (Xcoord[i] \* Ycoord[i]);*

*sumDX += Xcoord[i] \* Xcoord[i];*

*}*

*a = (n\*sumXY - sumX \* sumY) / (n \* sumDX - sumX \* sumX);*

*b = (sumY - a \* sumX) / n;*

*return a\*x + b;*

*}*

1. Составлена головная программа, содержащая обращение к соответствующим подпрограммам и осуществляющая печать результатов.

*int main () {*

*double Xcoord[6] = {0.0, 0.1, 0.4, 0.7, 1.0, 1.30};*

*double Ycoord[6] = {1.0, 0.99, 0.81, 0.45, 0.0, -0.45};*

*double x = 0.03; //0.03, 0.55, 1.21*

*for (int i = 0; i < 6; i++) {*

*Ycoord[i] = F(Xcoord[i]);*

*}*

*std::cout << "LagrangePolynomial = " << LagrangePolynomial(Xcoord, Ycoord, x, 6) << "\n";*

*std::cout << "NewtonPolynomial1 = " << NewtonPolynomial1(Xcoord, Ycoord, x, 6, 0.1) << "\n";*

*std::cout << "NewtonPolynomial2 = " << NewtonPolynomial2(Xcoord, Ycoord, x, 6, 0.1) << "\n";*

*std::cout << "Approximation = " << Approximation(Xcoord, Ycoord, x, 6);*

*return 0;*

*}*

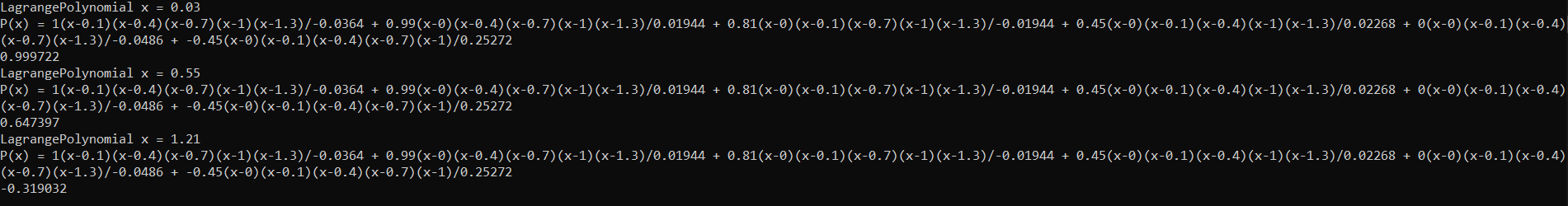
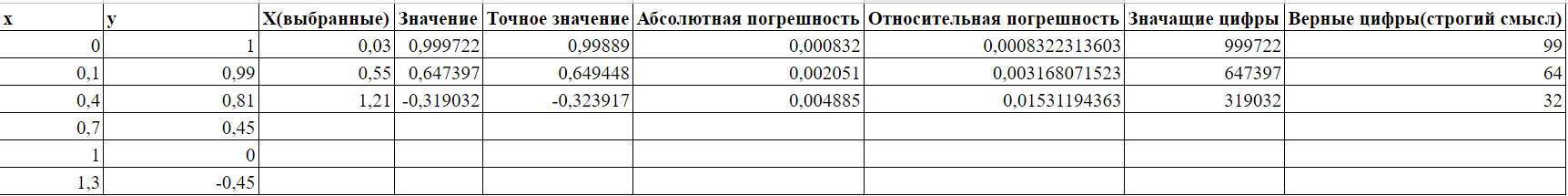
1. ****Используя многочлен Лагранжа, найдем приближенные значения функции. Подсчитаем точные значения, абсолютную погрешность, относительную погрешность, верные и значащие цифры.

Рисунок 1 – интерполяционный многочлен Лагранжа.

Таблица 1 – результат работы метода Лагранжа.

1. Используя многочлены Ньютона найдем все, что перечислено в предыдущем пункте.

Рисунок 2 – первый интерполяционный многочлен Ньютона.

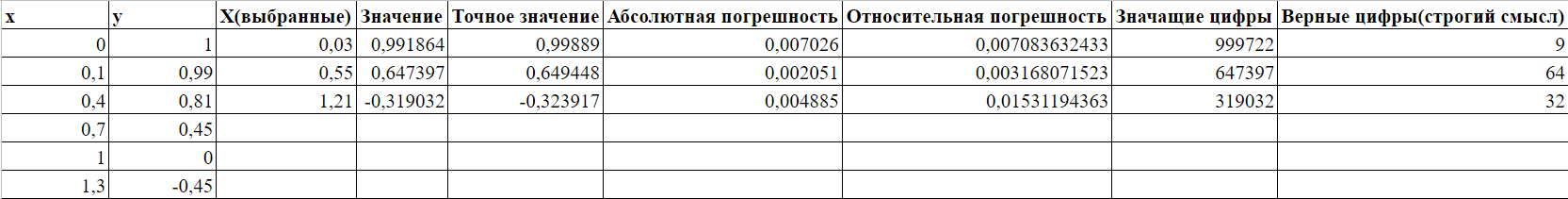
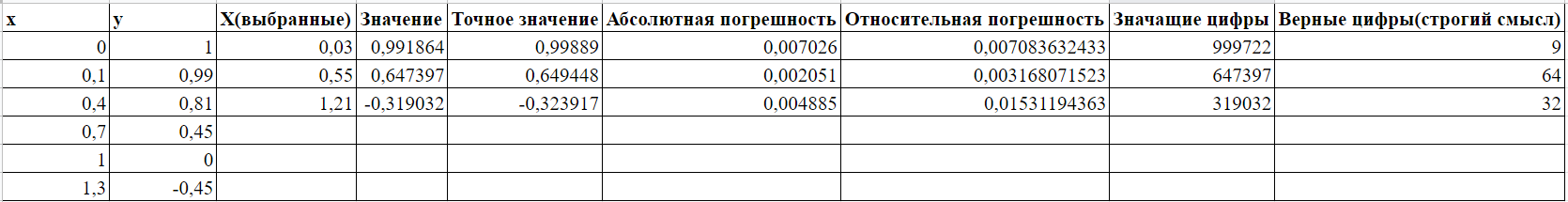
Таблица 2 – результат работы метода Ньютона 1.

Рисунок 3 – второй интерполяционный многочлен Ньютона.

Таблица 3 – результат работы метода Ньютона 2.

1. Составим таблицу конечных разностей до второго порядка. К одному из значений , а именно к y3 прибавим погрешность (), превосходящую допустимую. Покажем, как при этом изменятся конечные разности большего порядка.

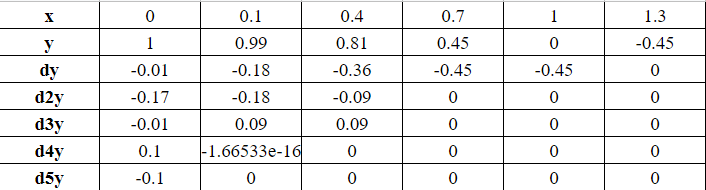
Таблица 4 – конечные разности.

Таблица 5 – конечные разности с a = 1.

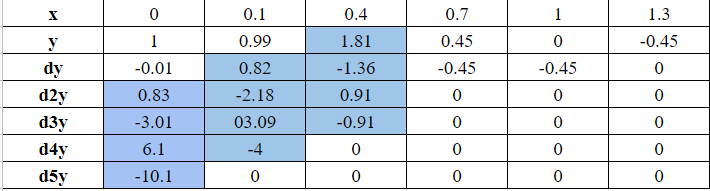
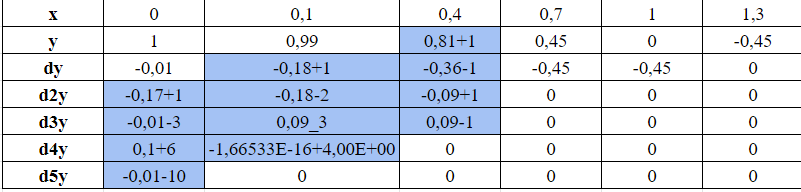
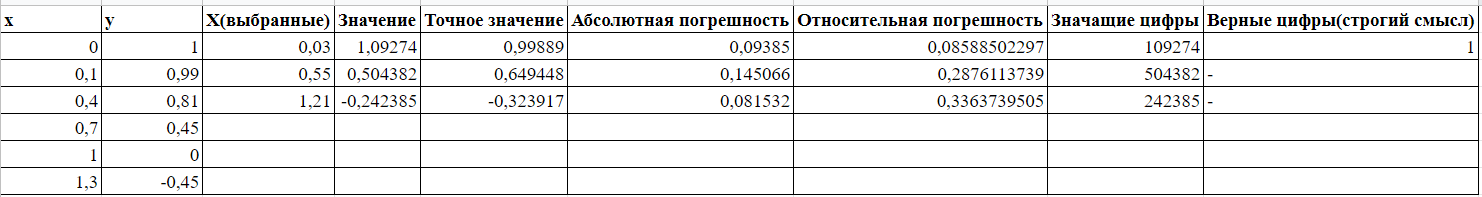


Таблица 6 – конечные разности. Демонстрация изменений значений.

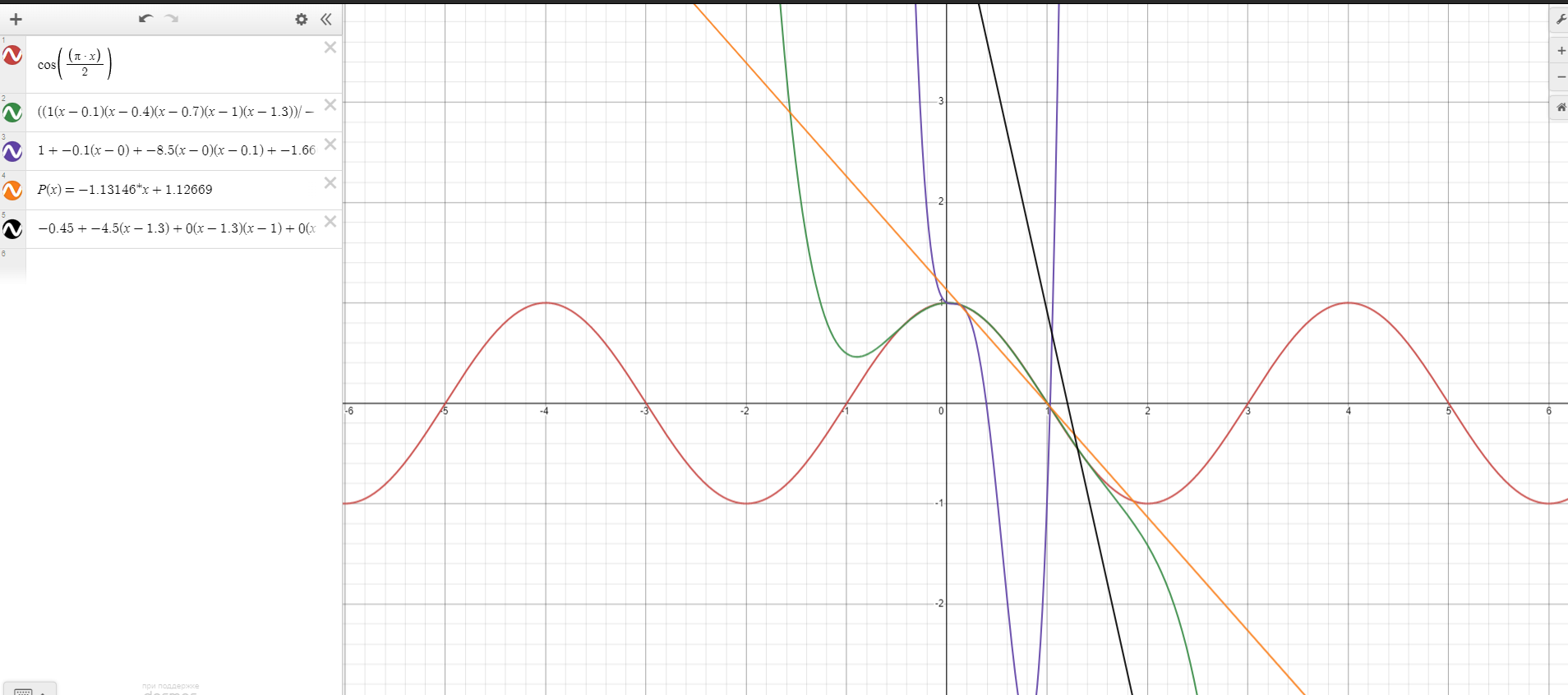


1. Осуществим аппроксимирование функции.

P(x) = -1.13146\*x+1.12669

Таблица 6 – результат работы метода аппроксимация.

1. Для того, чтобы сделать вывод построим график, на котором будет показана сама функция и каждый из методов.

Рисунок 4 – график оригинальной функции (красный) и приближённых функций: метод Лагранжа (зеленый), методы Ньютона (фиолетовый и чёрный) и линия аппроксимации (оранжевый).

Вывод: Интерполяционный многочлен Лагранжа (зеленый цвет) на небольшом отрезке совпадает с графиком функции. Первый интерполяционный многочлен Ньютона при приближении значения функции к 1 на небольшом отрезке, включающем 0, совпадает с функцией. Второй же многочлен Ньютона (черная линия) пресекает функцию в единственной точке. Линия апроксимации (оранжевый цвет) пересекает исходную функцию в трех точках.

# приложение

#define \_USE\_MATH\_DEFINES

#include <cmath>

#include <iostream>

double F(double x){

return cos((M\_PI\*x)/2);

}

double LagrangePolynomial(double Xcoord[], double Ycoord[], double x, int n) {

double P = 0;

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

double L = 1;

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

if (j != i)

L = L \* (x - Xcoord[j]) / (Xcoord[i] - Xcoord[j]);

}

P = P + Ycoord[i] \* L;

}

/\*std::cout<<"P(x) = ";

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

double r = 1;

std::cout<<"((";

std::cout<<Ycoord[i];

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

if (i != j){

std::cout<<"(x-";

std::cout<<Xcoord[j];

std::cout<<")";

r \*= Xcoord[i] - Xcoord[j];

}

}

//std::cout << r << ' ';

std::cout<<")/"<<r<<")";

if (i != 5)

std::cout<<" + ";

} std::cout << "\n";\*/

return P;

}

int Factorial(int i) {

double res = 1;

for (int iter = i; iter > 1; iter--)

res \*= iter;

return res;

}

/\*void NewtonPolynomial1(double Xcoord[], double Ycoord[], double x, int n, double h) {

double P = 0;

double matrixY[n][n];

double A[n];

//arrY[2] += 1;

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

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

if (i == 0) {

matrixY[i][j] = Ycoord[j + 1] - Ycoord[j];

matrixY[i][j + 1] = 0;

} else {

if (matrixY[i - 1][j + 1] != 0) {

matrixY[i][j] = matrixY[i - 1][j + 1] - matrixY[i - 1][j];

matrixY[i][j + 1] = 0;

} else {

matrixY[i][j] = 0;

matrixY[i][j + 1] = 0;

}

}

}

A[i] = matrixY[i][0];

}

std::cout << "\n";

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

std::cout<<Xcoord[i]<<" ";

std::cout<<std::endl;

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

std::cout<<Ycoord[i]<<" ";

std::cout<<std::endl;

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

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

std::cout << matrixY[i][j] << "\t";

std::cout << "\n";

}

/\*std::cout<<"P(x) = ";

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

if (i == 0){

std::cout<<Ycoord[0];

P += Ycoord[0];

}

else {

double el = 1;

double elh = 1;

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

el \*= (x - Xcoord[k]);

elh \*= h;

}

P += ((el \* A[i - 1]) / (elh \* Factorial(i)));

std::cout << A[i - 1] / (elh \* Factorial(i));

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

std::cout<<"(x-"<<Xcoord[k]<<")";

}

}

if (i != 5)

std::cout << " + ";

}

std::cout << "\n";

//return P;

}\*/

void NewtonPolynomial1(double array\_x[6], double array\_y[6], double x, double h){

double P = 0;

double matrix\_y[6][6] = {0}; // матрица конечных разностей

double A[6]; // массив коэффициентов ai

array\_y[2] += 1;

int k = 6; // Для цикла J

//заполним матрицу конечных разностей и массив коэффициентов

for(int i = 0; i < 6; i++){

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

if(i == 0){

matrix\_y[i][j] = array\_y[j];

}

else{

matrix\_y[i][j] = matrix\_y[i - 1][j + 1] - matrix\_y[i - 1][j];

}

}

k--;

A[i] = matrix\_y[i][0] / (Factorial(i) \* pow(h, i)); //вычисление коэффициента ai

}

std::cout << "\n";

for (int i = 0; i < 6; i++)

std::cout<<array\_x[i]<<" ";

std::cout<<std::endl;

for( int i = 0; i < 6; i ++){

for (int j = 0; j < 6; j++)

std::cout << matrix\_y[i][j] << "\t";

std::cout << "\n";

}

//вычисление многочлена Ньютона

P = A[0];

double X[6] = {1};

X[0] = x - array\_x[0];

for(int i = 1; i < 6; i++){

X[i] = X[i-1]\*(x - array\_x[i]);

P = P + A[i]\*X[i-1];

}

//return P;

}

void NewtonPolynomial2(double Xcoord[], double Ycoord[], double x, int n, double h) {

double P = 0;

double matrixY[n][n];

double A[n];

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

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

if (i == 0) {

matrixY[i][j] = Ycoord[j + 1] - Ycoord[j];

matrixY[i][j + 1] = 0;

} else {

if (matrixY[i - 1][j + 1] != 0) {

matrixY[i][j] = matrixY[i - 1][j + 1] - matrixY[i - 1][j];

matrixY[i][j + 1] = 0;

} else {

matrixY[i][j] = 0;

matrixY[i][j+1] = 0;

}

}

}

A[i] = matrixY[i][n - 2 - i];

}

std::cout << "\n";

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

std::cout<<Xcoord[i]<<" ";

std::cout<<std::endl;

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

std::cout<<Ycoord[i]<<" ";

std::cout<<std::endl;

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

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

std::cout << matrixY[i][j] << "\t";

std::cout << "\n";

}

/\*std::cout<<"P(x) = ";

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

if (i == 0) {

P += Ycoord[n - 1];

std::cout<<Ycoord[n-1];

}

else {

double el = 1;

double elh = 1;

for (int k = n; k > (n-i); k--) {

el \*= (x - Xcoord[k-1]);

}

for (int k = 0; k < i; k++) elh \*= h;

P += ((el \* A[i - 1]) / (elh \* Factorial(i)));

std::cout << A[i - 1] / (elh \* Factorial(i));

// std::cout << A[i - 1] / (elh \* factorial(i)) << " ";

for (int k = n; k > (n-i); k--) {

std::cout<<"(x-"<<Xcoord[k-1]<<")";

}

}

if (i != 5)

std::cout << " + ";

}\*/

std::cout << "\n";

//return P;

}

double Approximation(double Xcoord[], double Ycoord[], double x, int n) {

double a,b;

double sumX = 0, sumY = 0, sumXY = 0, sumDX = 0;

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

sumX += Xcoord[i];

sumY += Ycoord[i];

sumXY += (Xcoord[i] \* Ycoord[i]);

sumDX += Xcoord[i] \* Xcoord[i];

}

a = (n\*sumXY - sumX \* sumY) / (n \* sumDX - sumX \* sumX);

b = (sumY - a \* sumX) / n;

std::cout<<"P(x) = "<<a<<"\*x+"<<b<<std::endl;

return a\*x + b;

}

int main () {

double Xcoord[6] = {0.0, 0.1, 0.4, 0.7, 1.0, 1.30};

double Ycoord[6] = {1.0, 0.99, 0.81, 0.45, 0.0, -0.45};

double x1 = 0.03;

double x2 = 0.55;

double x3 = 1.21;

//for (int i = 0; i < 6; i++) {

// Ycoord[i] = F(Xcoord[i]);

//}

//std::cout<<"F(x1)\n"<<F(x1)<<std::endl;

//std::cout<<"F(x1)\n"<<F(x2)<<std::endl;

//std::cout<<"F(x1)\n"<<F(x3)<<std::endl;

//std::cout << "LagrangePolynomial x = 0.03\n" << LagrangePolynomial(Xcoord, Ycoord, x1, 6) << "\n";

//std::cout << "LagrangePolynomial x = 0.55\n" << LagrangePolynomial(Xcoord, Ycoord, x2, 6) << "\n";

//std::cout << "LagrangePolynomial x = 1.21\n" << LagrangePolynomial(Xcoord, Ycoord, x3, 6) << "\n";

//std::cout << "NewtonPolynomial1 x = 0.03\n";

//NewtonPolynomial1(Xcoord, Ycoord, x1, 6, 0.1);

//std::cout << "NewtonPolynomial1 x = 0.55\n";

//NewtonPolynomial1(Xcoord, Ycoord, x2, 0.1);

//std::cout << "NewtonPolynomial1 x = 1.21\n";

//NewtonPolynomial1(Xcoord, Ycoord, x3, 6, 0.1);

//std::cout << "NewtonPolynomial2 x = 0.03\n";

//NewtonPolynomial2(Xcoord, Ycoord, x1, 6, 0.1);

//std::cout << "NewtonPolynomial2 x = 0.55\n";

//NewtonPolynomial2(Xcoord, Ycoord, x2, 6, 0.1);

//std::cout << "NewtonPolynomial2 x = 1.21\n";

//NewtonPolynomial2(Xcoord, Ycoord, x3, 6, 0.1);

std::cout << "Approximation x = 0.03\n" << Approximation(Xcoord, Ycoord, x1, 6)<<std::endl;

std::cout << "Approximation x = 0.55\n" << Approximation(Xcoord, Ycoord, x2, 6)<<std::endl;

std::cout << "Approximation x = 1.21\n" << Approximation(Xcoord, Ycoord, x3, 6)<<std::endl;

return 0;

}