|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Министерство науки и высшего образования РФ | | | | | | | | |  | |
|  | | | | |  | | |  | | | | |
|  | | | | ФГБОУ ВО «Пермский государственный национальный исследовательский университет» | | | | |  | | | |
|  | | | | |  | | |  | | | | |
|  | | | **Численные методы**  Лабораторная работа №5  «Интерполирование. Среднеквадратичное приближение»  Варианты: №21b, №25b | | | | | | |  | | |
|  | | | | |  | | |  | | | | |
|  | Работу выполнили  Студенты гр. ПМИ-4:  Колесников А.С  Пухов Н.А. | | | | |  | Проверил  профессор, доктор физико-математических наук  Русаков С.В  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2020 г. | | | | |  | |
|  | | | | |  | | |  | | | | |
|  | | | | | Пермь 2020 | | |  | | | | |

СОДЕРЖАНИЕ

[Задание 3](#_Toc56458667)

[Исходные данные 4](#_Toc56458668)

[Решение 5](#_Toc56458669)

[Тестирование 12](#_Toc56458670)

[Краткие выводы 18](#_Toc56458671)

[Текст программы 19](#_Toc56458672)

Задание

*Задача* - приблизить заданную функцию  на отрезке .

1. Построить таблицу ,  .

По полученной таблице произвести интерполяцию с помощью

* формулы Ньютона;
* кубических сплайнов дефекта 1.

Провести оценку погрешности в узлах .

Сравнить оценку погрешности с реальной погрешностью.

1. Выполнить среднеквадратичное приближение заданной функции на заданном отрезке c помощью полинома второго порядка

* дискретный вариант (по таблице из п.1);
* непрерывный (интегральный) вариант.

Провести оценку погрешности.

Построить график приближаемой и приближающих функций.

1. Методом обратного интерполирования, используя интерполяционную формулу Ньютона, найти корень уравнения

.

Константа *с* выбирается таким образом, чтобы существовал корень на отрезке .

*Указание.*

1. При построении параметров кубического сплайна воспользоваться алгоритмом прогонки. Выполнить оценку погрешности аппроксимации значений первой производной в узлах интерполяции.

Исходные данные

Вариант №21:



Вариант №25:



Решение

**Построение таблицы разделенных разностей**

Строим таблицу



где 



где 

**Интерполяция с помощью формулы Ньютона**

Интерполяционная формула Ньютона:



Погрешность формулы Ньютона



**Интерполяция с помощью кубических сплайнов дефекта 1**

Метод прогонки:









Затраты 

Для начала положим:



Оценка погрешности аппроксимации значений первой производной в узлах интерполяции:





Оценка погрешности аппроксимации функции:





где  и  находится по формулам:



Сравнение оценки погрешности с реальной погрешностью:



**Среднеквадратичное приближение заданной функции на заданном отрезке с помощью полинома второго порядка**

Погрешность приближения 

**Дискретный вариант**

Матрица: 

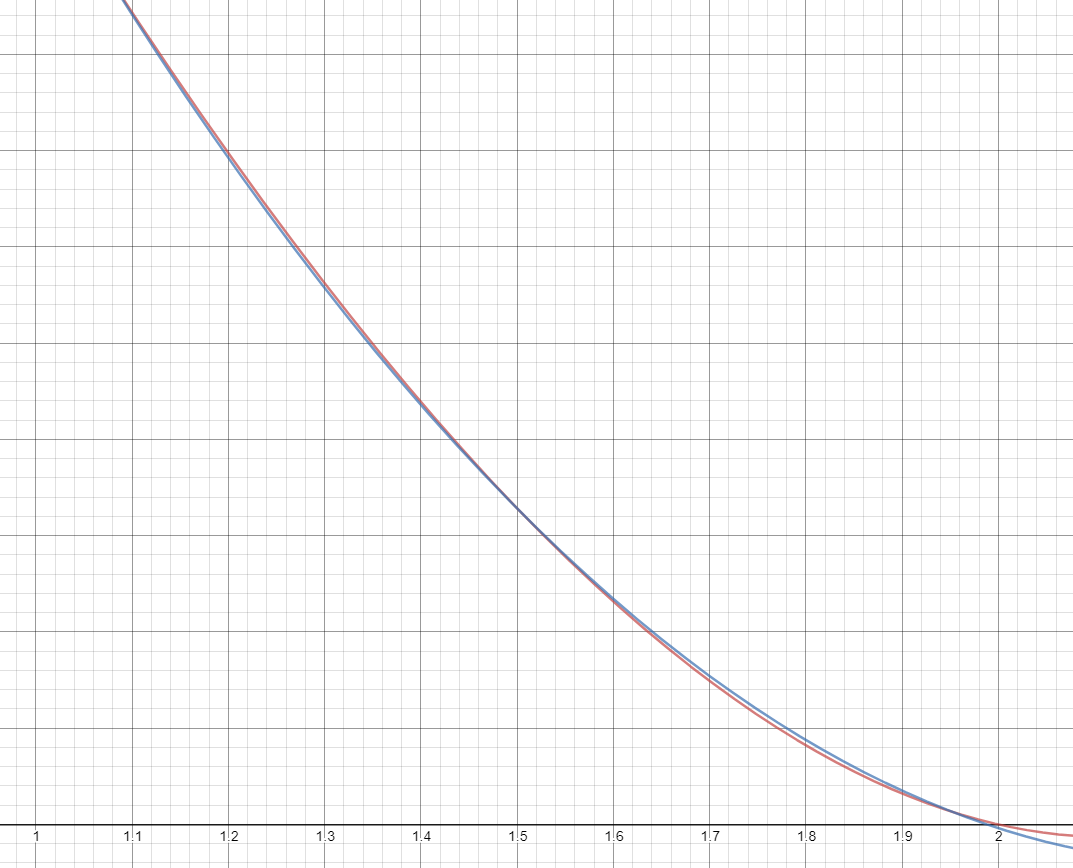
Свободные члены: 

Графики исходной и приближенной функций

Вариант 21:

 <-красная

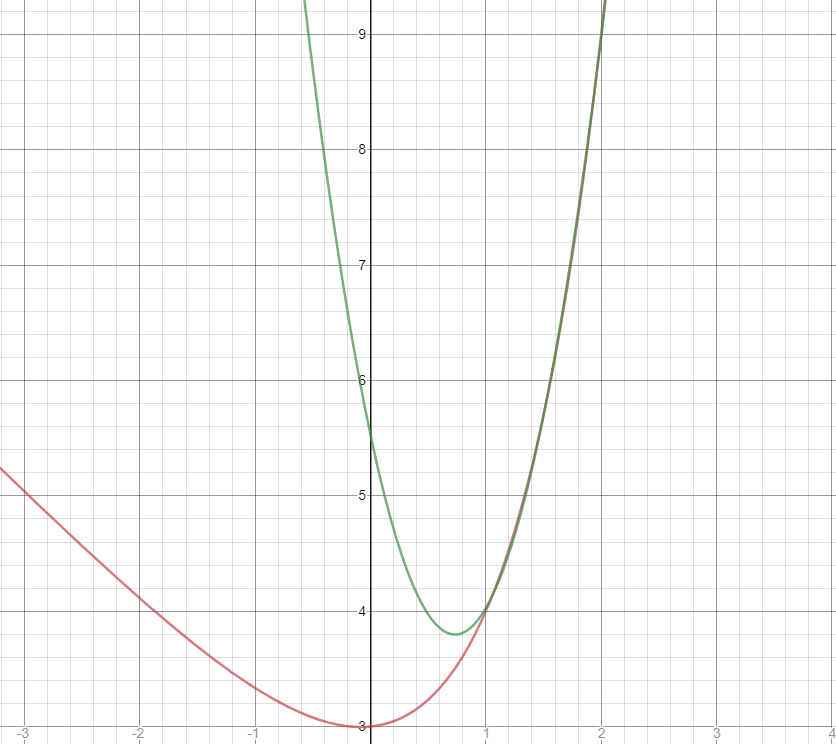
 <- синяя



Вариант 25:

 <-красная

 <-зеленая



**Непрерывный (интегральный) вариант**

Матрица: 

Свободные члены: 

Графики исходной и приближенной функций

Вариант 21:

 <-красная

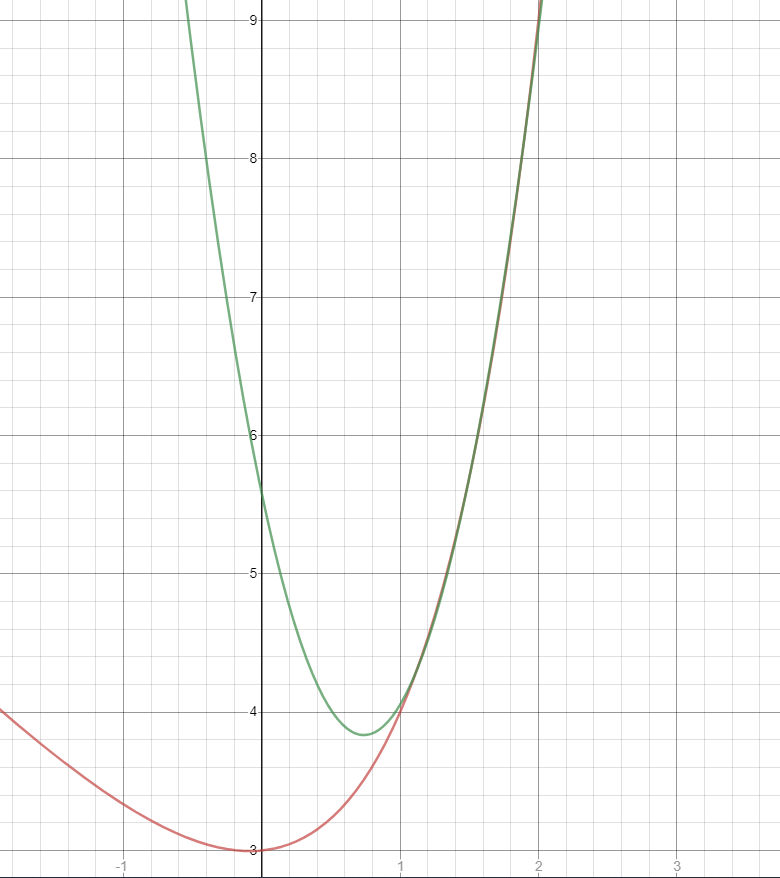
 <-синяя



Вариант 25:

 <-красная

 <-зеленая



**Метод обратного интерполирования**

Требуется определить корень уравнения  и удалось выделить отрезок , на котором функция  монотонно возрастает (или убывает) и содержит корень. Тогда

1. Выберем на этом отрезке точки 
2. Вычислим значения 
3. В силу монотонности функции  на отрезке  имеем  (или )
4. Строим интерполяционный многочлен , для которого 

Очевидно , где  - корень уравнения. При этом погрешность решения можно определить с помощью формулы:



Тестирование

Вариант 21

Интерполяционная формула Ньютона

Таблица разделенных разностей

1.00 1.000000 -1.513016 0.552780 0.136996 0.025464 0.003786

1.20 0.697397 -1.291904 0.634977 0.157367 0.029250

1.40 0.439016 -1.037913 0.729398 0.180767

1.60 0.231433 -0.746154 0.837858

1.80 0.082202 -0.411011

2.00 0.000000

M6 = 0.443620

| x| f(x)| Pn(x)| Delta| Оценка|

|1.10| 0.84355| 0.84355| 0.000000397157235| 0.000000582251250|

|1.30| 0.56229| 0.56229| 0.000000134973380| 0.000000194083750|

|1.50| 0.32843| 0.32843| 0.000000098322261| 0.000000138631250|

|1.70| 0.14901| 0.14901| 0.000000140423343| 0.000000194083750|

|1.90| 0.03213| 0.03213| 0.000000429883182| 0.000000582251250|

Интерполяция кубическим сплайном

M5 = 0.640010

|x[i]| df/dx(x[i])| m[i]| Delta| Оценка|

|1.00| -1.6137056| -1.6137056| 0.000000000000000| 0.000017066933333|

|1.20| -1.4075659| -1.4075699| 0.000003993257645| 0.000017066933333|

|1.40| -1.1707736| -1.1707773| 0.000003649700362| 0.000017066933333|

|1.60| -0.8987707| -0.8987746| 0.000003948539674| 0.000017066933333|

|1.80| -0.5863213| -0.5863278| 0.000006448489675| 0.000017066933333|

|2.00| -0.2274113| -0.2274113| 0.000000000000000| 0.000017066933333|

M4 = 0.923340

| x| f(x)| S31(f;x)| Abs(f(x)-S31(f;x))| Оценка|

|1.10| 0.84355| 0.84354| 0.000001962520115| 0.000004700596667|

|1.30| 0.56229| 0.56229| 0.000002377608772| 0.000004700596667|

|1.50| 0.32843| 0.32842| 0.000002713818210| 0.000004700596667|

|1.70| 0.14901| 0.14901| 0.000003063441670| 0.000004700596667|

|1.90| 0.03213| 0.03213| 0.000003751974861| 0.000004700596667|

Среднеквадратичное приближение

Дискретный вариант

Матрица

6.00000 9.00000 14.20000

9.00000 14.20000 23.40000

14.20000 23.40000 39.96640

Вектор правых частей

2.4500479177452 2.9697552707224 3.7235263934722

P2(x) = (3.38507) + (-3.06823)\*x + (0.68688)\*x^2

Норма погрешности: 0.0101961545898968

Непрерывный вариант

Матрица

1.00000 1.50000 2.33333

1.50000 2.33333 3.75000

2.33333 3.75000 6.20000

Вектор правых частей

0.3853900817779 0.4934322833226 0.6490922956725

P2(x) = (3.39398) + (-3.07177)\*x + (0.68531)\*x^2

Норма погрешности: 0.0029895133181514

Решение уравнения методом обратной интерполяции

Таблица разделенных разностей

1.00 1.000000 -0.660931 0.201645 -0.266559 0.859561 -9.601252

0.70 1.200000 -0.774051 0.406513 -1.055462 10.460813

0.44 1.400000 -0.963471 1.055828 -8.350799

0.23 1.600000 -1.340205 4.721961

0.08 1.800000 -2.433023

0.00 2.000000

c = 0.3284

Корень=1.50513

Невязка=Abs(f(x)-c)=0.005317

Вариант 25

Интерполяционная формула Ньютона

Таблица разделенных разностей

1.00 4.000000 2.685964 2.264389 0.927384 0.284859 0.069999

1.20 4.537193 3.591720 2.820819 1.155271 0.354857

1.40 5.255537 4.720047 3.513981 1.439157

1.60 6.199546 6.125640 4.377475

1.80 7.424674 7.876630

2.00 9.000000

M6 = 15.824000

| x| f(x)| Pn(x)| Delta| Оценка|

|1.10| 4.24837| 4.24838| 0.000011363997116| 0.000020769000000|

|1.30| 4.87117| 4.87116| 0.000003904375158| 0.000006923000000|

|1.50| 5.69615| 5.69616| 0.000002876566765| 0.000004945000000|

|1.70| 6.77301| 6.77300| 0.000004156939480| 0.000006923000000|

|1.90| 8.16363| 8.16364| 0.000012882429838| 0.000020769000000|

Интерполяция кубическим сплайном

M5 = 14.403000

|x[i]| df/dx(x[i])| m[i]| Delta| Оценка|

|1.00| 2.2958369| 2.2958369| 0.000000000000000| 0.000384080000000|

|1.20| 3.1057260| 3.1056623| 0.000063619315166| 0.000384080000000|

|1.40| 4.1146299| 4.1145646| 0.000065239036501| 0.000384080000000|

|1.60| 5.3714527| 5.3713789| 0.000073705022104| 0.000384080000000|

|1.80| 6.9371157| 6.9369796| 0.000136091195766| 0.000384080000000|

|2.00| 8.8875106| 8.8875106| 0.000000000000000| 0.000384080000000|

M4 = 13.111000

| x| f(x)| S31(f;x)| Abs(f(x)-S31(f;x))| Оценка|

|1.10| 4.24837| 4.24835| 0.000018749443126| 0.000073833166667|

|1.30| 4.87117| 4.87114| 0.000025297582101| 0.000073833166667|

|1.50| 5.69615| 5.69612| 0.000031352774504| 0.000073833166667|

|1.70| 6.77301| 6.77297| 0.000037761125407| 0.000073833166667|

|1.90| 8.16363| 8.16357| 0.000052385391797| 0.000073833166667|

Среднеквадратичное приближение

Дискретный вариант

Матрица

6.00000 9.00000 14.20000

9.00000 14.20000 23.40000

14.20000 23.40000 39.96640

Вектор правых частей

36.4169497312300 58.0860699092486 96.7611916797656

P2(x) = (5.52792) + (-4.72292)\*x + (3.22223)\*x^2

Норма погрешности: 0.0755427116983096

Непрерывный вариант

Матрица

1.00000 1.50000 2.33333

1.50000 2.33333 3.75000

2.33333 3.75000 6.20000

Вектор правых частей

5.9614353597610 9.3490423679279 15.1484832581530

P2(x) = (5.57921) + (-4.72902)\*x + (3.20390)\*x^2

Норма погрешности: 0.0213019607750351

Решение уравнения методом обратной интерполяции

Таблица разделенных разностей

4.00 1.000000 0.372306 -0.074779 0.015795 -0.002830 0.000406

4.54 1.200000 0.278418 -0.040037 0.006104 -0.000802

5.26 1.400000 0.211862 -0.022412 0.002525

6.20 1.600000 0.163248 -0.012959

7.42 1.800000 0.126958

9.00 2.000000

c = 5.6962

Корень=1.49971

Невязка=Abs(f(x)-c) =0.001368

Краткие выводы

Мы познакомились с такой непростой, но одновременно важной темой как приближение функций (аппроксимация или интерполирование). Приближение функций позволяет заменить исходную функцию некоторым полиномом на некотором отрезке и дает возможность проводить дальнейшие исследования уже над этим полиномом (например, можно найти корень). Мы узнали несколько методов аппроксимации: интерполяция с помощью формулы Ньютона, с помощью кубических сплайнов дефекта 1, а также среднеквадратичное приближение. Одним из популярных методов на сегодняшний день является метод интерполирования с помощью кубических сплайнов дефекта 1. Основным его преимуществом является отсутствие требования к гладкости функции, что позволяет интерполяционному процессу сходится при увеличении числа узлов сетки.

Текст программы

#include <iostream>

#include <algorithm>

#include <math.h>

#include <fstream>

#include <functional>

#include <iomanip>

#include <cstdarg>

using namespace std;

//Таблица

class Table

{

list<string> m\_colnames;

list<int> m\_precisions;

public:

Table(const list<string>& colnames, const list<int>& precisions)

: m\_colnames(colnames), m\_precisions(precisions)

{}

//распечатка заголовка таблицы

void printTableHeader()

{

cout << "|";

auto it = m\_precisions.begin();

for (const auto& colname : m\_colnames)

{

cout << setw(\*it \* 2) << colname << "|";

it++;

}

cout << "\n";

}

//распечатка строки таблицы

void printTableRow(const vector<double>& values)

{

cout << "|";

auto it = m\_precisions.begin();

for (const auto& value : values)

{

cout << setw(\*it \* 2) << setprecision(\*it) << fixed << value << "|";

it++;

}

cout << "\n";

}

};

//Вектор

class Vector

{

double\* m\_vector;

public:

int m\_N;

Vector() = default;

Vector(int N)

: m\_N(N)

{

m\_vector = new double[N];

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

m\_vector[i] = 0;

}

}

//обращение к компоненте вектора

double& operator[](int index) {

return m\_vector[index];

}

//распечатка вектора

void printVector(bool expFormat = false) {

printf(" ");

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

if (expFormat) {

printf("%.10e ", m\_vector[i]);

}

else {

printf("%.7f ", m\_vector[i]);

}

}

printf("\n");

}

};

//Матрица

class Matrix

{

Vector\* m\_vectors;

public:

int m\_N;

Matrix() = default;

Matrix(int N)

: m\_N(N)

{

m\_vectors = new Vector[N];

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

m\_vectors[i] = Vector(N);

}

}

Vector& operator[](int index) {

return m\_vectors[index];

}

};

//Функция и её производная

struct Function

{

using FuncType = std::function<double(double)>;

FuncType m\_func;

FuncType m\_derivative;

Function(FuncType func, FuncType derivative)

: m\_func(func), m\_derivative(derivative)

{}

};

//Таблица узлов интерполяции

class NodeTable

{

public:

int m\_N;

function<double(int)> m\_x\_nodes;

function<double(int)> m\_y\_nodes;

NodeTable(int N, function<double(int)> x\_nodes, function<double(int)> y\_nodes)

: m\_N(N), m\_x\_nodes(x\_nodes), m\_y\_nodes(y\_nodes)

{}

//вычислить h(i)

double getH(int i) {

return m\_x\_nodes(i + 1) - m\_x\_nodes(i);

}

//вычислить индекс отрезка, которому принадлежит x

int calcIndex(double x) {

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

if (x >= m\_x\_nodes(i) && x <= m\_x\_nodes(i + 1))

return i;

}

throw std::logic\_error("x out of the range");

}

};

//Интерполяция (абстрактный класс)

class Interpolation

{

public:

Function m\_f;

NodeTable m\_nodeTable;

Interpolation(Function f, NodeTable splittingTable)

: m\_f(f), m\_nodeTable(splittingTable)

{}

//вычислить значение полученного полинома в произвольной точке

virtual double interpolate(double x) = 0;

};

//Ньютоновская интерполяция

class NewtonianInterpolation : public Interpolation

{

//таблица разделенных разностей (без столбца значений X)

Matrix m\_divDifferTable;

public:

NewtonianInterpolation(Function f, NodeTable nodeTable)

: Interpolation(f, nodeTable)

{

m\_divDifferTable = Matrix(m\_nodeTable.m\_N + 1);

fillDivDifferTable();

}

//вычислить значение полученного полинома в произвольной точке

double interpolate(double x) override {

double w = 1;

double result = m\_divDifferTable[0][0];

for (int i = 1; i < m\_divDifferTable.m\_N; i++) {

w \*= x - m\_nodeTable.m\_x\_nodes(i - 1);

result += m\_divDifferTable[0][i] \* w;

}

return result;

}

//распечатка таблицы разделенных разностей

void printDivDifferTable()

{

printf("Таблица разделенных разностей\n");

for (int i = 0; i < m\_divDifferTable.m\_N; i++) {

printf("%.2f ", m\_nodeTable.m\_x\_nodes(i));

for (int j = 0; j < m\_divDifferTable.m\_N - i; j++) {

printf("%.6f ", m\_divDifferTable[i][j]);

}

printf("\n");

}

}

private:

void fillDivDifferTable() {

//заполняем первый столбец

for (int i = 0; i < m\_divDifferTable.m\_N; i++) {

m\_divDifferTable[i][0] = m\_nodeTable.m\_y\_nodes(i);

}

//заполняем все последующие столбцы

for (int j = 1; j < m\_divDifferTable.m\_N; j++) {

for (int i = 0; i < m\_divDifferTable.m\_N - j; i++) {

m\_divDifferTable[i][j] =

(m\_divDifferTable[i + 1][j - 1] - m\_divDifferTable[i][j - 1]) / (m\_nodeTable.m\_x\_nodes(i + j) - m\_nodeTable.m\_x\_nodes(i));

}

}

}

};

//Кубическая сплайн интерполяция дефекта 1

class SplineInterpolation : public Interpolation

{

public:

Vector m\_vectorM;

SplineInterpolation(Function f, NodeTable nodeTable)

: Interpolation(f, nodeTable)

{

m\_vectorM = Vector(m\_nodeTable.m\_N + 1);

m\_vectorM[0] = f.m\_derivative(m\_nodeTable.m\_x\_nodes(0));

m\_vectorM[m\_vectorM.m\_N - 1] = f.m\_derivative(m\_nodeTable.m\_x\_nodes(m\_vectorM.m\_N - 1));

calcVectorM();

}

//вычислить значение полученного полинома в произвольной точке

double interpolate(double x) override {

return interpolate(x, nullptr, 0, 0);

}

//интерполяция + вычисление погрешности

double interpolate(double x, double\* errorEst, double M4, double M5) {

auto idx = m\_nodeTable.calcIndex(x);

auto h = m\_nodeTable.getH(idx);

auto tau = (x - m\_nodeTable.m\_x\_nodes(idx)) / h;

if (errorEst)

\*errorEst = tau \* Fi1(tau) \* pow(h, 4) \* M4 / 24 + (Fi1(tau) + Fi1(1 - tau)) \* pow(h, 5) \* M5 / 60;

return Fi0(tau) \* m\_nodeTable.m\_y\_nodes(idx) + Fi0(1 - tau) \* m\_nodeTable.m\_y\_nodes(idx + 1) + h \* (Fi1(tau) \* m\_vectorM[idx] - Fi1(1 - tau) \* m\_vectorM[idx + 1]);

}

private:

//найти значения m(i)

void calcVectorM() {

Vector a(m\_vectorM.m\_N - 2);

Vector c(a.m\_N);

Vector b(a.m\_N);

Vector f(a.m\_N);

Vector x(a.m\_N);

for (int i = 1, j = 0; i <= m\_vectorM.m\_N - 1; i++, j++) {

auto curH = m\_nodeTable.getH(i);

auto prevH = m\_nodeTable.getH(i - 1);

b[j] = prevH / (prevH + curH);

c[j] = 2;

a[j] = 1 - b[j];

f[j] = 3 \* a[j] \* (m\_nodeTable.m\_y\_nodes(i) - m\_nodeTable.m\_y\_nodes(i - 1)) / prevH +

3 \* b[j] \* (m\_nodeTable.m\_y\_nodes(i + 1) - m\_nodeTable.m\_y\_nodes(i)) / curH;

}

f[0] -= m\_nodeTable.getH(1) / (m\_nodeTable.getH(0) + m\_nodeTable.getH(1)) \* m\_vectorM[0];

f[f.m\_N - 1] -= m\_nodeTable.getH(m\_vectorM.m\_N - 2) / (m\_nodeTable.getH(m\_vectorM.m\_N - 2) + m\_nodeTable.getH(m\_vectorM.m\_N - 1)) \* m\_vectorM[m\_vectorM.m\_N - 1];

SweepMethod(a, c, b, x, f);

for (int i = 0; i < x.m\_N; i++) {

m\_vectorM[i + 1] = x[i];

}

}

double Fi0(double tau) {

return (1 + 2 \* tau) \* pow(1 - tau, 2);

}

double Fi1(double tau) {

return tau \* pow(1 - tau, 2);

}

//Метод прогонки (матрица состоит из 3-х диагоналей: a, c, b)

static void SweepMethod(Vector& a, Vector& c, Vector& b, Vector& x, Vector& f) {

Vector alpha(x.m\_N);

Vector beta(x.m\_N);

alpha[0] = -b[0] / c[0]; //b1 / c1

beta[0] = f[0] / c[0]; //f1 / c1

for (int i = 0, j = 1; i < alpha.m\_N - 1; i++, j++) {

double denominator = c[j] + a[j] \* alpha[i]; //c(j) - a(j) \* alpha(j)

alpha[i + 1] = -b[j] / denominator; //b(j) / den

beta[i + 1] = (f[j] - a[j] \* beta[i]) / denominator; //[f(j) + a(j) \* beta(j)] / den

}

x[x.m\_N - 1] = beta[beta.m\_N - 1];

for (int i = x.m\_N - 2; i >= 0; i--) {

x[i] = alpha[i] \* x[i + 1] + beta[i];

}

}

};

class LUDecomposition

{

double\*\* L, \*\* U;

int\* P;

double sign;

public:

int size;

LUDecomposition(double\*\* const matrix, const int size)

{

sign = 1.0;

this->size = size;

P = new int[size];

L = new double\* [size];

U = new double\* [size];

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

{

L[i] = new double[size];

U[i] = new double[size];

}

copyMatrixToMatrix(matrix, U, size);

//иницилизируем подстановку P

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

P[i] = i;

}

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

auto rowIdx = defineRowIdxWithMainValue(U, k, size);

if (k != rowIdx) {

//Смена строк

swap(U[k], U[rowIdx]);

swap(L[k], L[rowIdx]);

swap(P[k], P[rowIdx]);

sign \*= -1.0;

}

//главный элемент

double mainValue = U[k][k];

//if (abs(mainValue) < minValue) {

// //Определяем ранг матрицы

// rank = k;

// return;

//}

//заполняем матрицу L

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

L[i][k] = U[i][k];

}

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

U[k][j] /= mainValue;

}

//заполняем матрицу U

for (int i = k + 1; i < size; i++) {

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

U[i][j] = U[i][j] - L[i][k] \* U[k][j];

}

}

}

}

~LUDecomposition()

{

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

{

delete[] L[i], U[i];

}

delete[] P, L, U;

}

//Решение уравнения Ax = b, то есть LUx = Pb

void SolveSOLE(double\* X, double\* B) {

//Ax = b (A = PLU)

//LUx = Pb (Ux = y)

//Ly = Pb

double\* vectorY = new double[size];

//"умножаем" вектор b на матрицу перестановок

double\* vectorPB = new double[size];

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

vectorPB[i] = B[P[i]];

}

SolveLy(L, vectorY, vectorPB, size);

//printVector(Y, N);

SolveUx(U, X, vectorY, size);

}

//Получение обратной матрицы

void SolveBackwardMatrix(double\*\* X) {

//LUX = PE

double\* vectorX = new double[size];

double\* vectorE = new double[size];

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

vectorE[t] = 0.0;

}

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

//формируем вектор Ei

if (i != 0) {

vectorE[i - 1] = 0.0;

}

vectorE[i] = 1.0;

//получаем вектор-столбец X

SolveSOLE(vectorX, vectorE);

//записываем его в матрицу X

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

X[t][i] = vectorX[t];

}

}

}

private:

//определить строку с главным элементом (который максимален в текущем столбце)

int defineRowIdxWithMainValue(double\*\* matrix, int k, int N) {

int m = k;

double maxValue = 0.0;

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

if (abs(matrix[i][k]) > maxValue) {

m = i;

maxValue = abs(matrix[m][k]);

}

}

return m;

}

void copyMatrixToMatrix(double\*\* srcMatrix, double\*\* dstMatrix, int N) {

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

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

dstMatrix[i][j] = srcMatrix[i][j];

}

}

}

//Решение уравнения Ly = Pb

void SolveLy(double\*\* triangleMatrix, double\* X, double\* B, int N) {

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

X[i] = B[i] / triangleMatrix[i][i];

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

X[i] -= X[j] \* triangleMatrix[i][j] / triangleMatrix[i][i];

}

}

}

//Решение уравнения Ux = y

void SolveUx(double\*\* triangleMatrix, double\* X, double\* B, int N) {

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

X[i] = B[i];

for (int j = N - 1; j > i; j--) {

X[i] -= X[j] \* triangleMatrix[i][j];

}

}

}

};

class RmsApproximation

{

using FuncType = std::function<double(double)>;

double\* m\_fg; //столбец свободных членов

double\* m\_c;

double\* m\_f;

double\*\* m\_gg; //матрица

int m\_vector\_size;

double m\_step = 0.2;

double normError = 0;

double m\_a, m\_b; //концы отрезка

FuncType m\_gVector[3] = {

[](double x) {

return 1;

},

[](double x) {

return x;

},

[](double x) {

return x \* x;

}

};

public:

//Дискретный вариант

RmsApproximation(FuncType function, double a, double b)

{

m\_a = a;

m\_b = b;

m\_vector\_size = 3;

m\_c = new double[m\_vector\_size];

m\_fg = new double[m\_vector\_size];

//Свободные члены (f,g)

for (size\_t i = 0; i < m\_vector\_size; i++)

{

double sum = 0;

for (double x = m\_a; x <= m\_b; x += m\_step)

{

sum += function(x) \* m\_gVector[i](x);

}

m\_fg[i] = sum;

}

//Создание матрицы

m\_gg = new double\* [m\_vector\_size];

for (size\_t i = 0; i < m\_vector\_size; i++)

{

m\_gg[i] = new double[m\_vector\_size];

}

//Ее заполнение (g,g)

for (size\_t i = 0; i < m\_vector\_size; i++)

{

for (size\_t j = 0; j < m\_vector\_size; j++)

{

double sum = 0;

for (double x = m\_a; x <= m\_b; x += m\_step)

{

sum += m\_gVector[i](x) \* m\_gVector[j](x);

}

m\_gg[i][j] = sum;

}

}

m\_f = new double[1.0 / m\_step + 1];

double x = m\_a;

for (size\_t i = 0; i < 1.0 / m\_step + 1; i++)

{

m\_f[i] = function(x);

x += m\_step;

}

}

//Непрерывный вариант

RmsApproximation(double gg[3][3], double fg[3])

{

m\_a = 2;

m\_b = 1;

m\_vector\_size = 3;

m\_c = new double[m\_vector\_size];

m\_fg = new double[m\_vector\_size];

for (size\_t i = 0; i < m\_vector\_size; i++)

{

m\_fg[i] = fg[i];

}

m\_gg = new double\* [m\_vector\_size];

for (size\_t i = 0; i < m\_vector\_size; i++)

{

m\_gg[i] = new double[m\_vector\_size];

}

for (size\_t i = 0; i < m\_vector\_size; i++)

{

for (size\_t j = 0; j < m\_vector\_size; j++)

{

m\_gg[i][j] = gg[i][j];

}

}

}

~RmsApproximation()

{

for (size\_t i = 0; i < m\_vector\_size; i++)

{

delete[] m\_gg[i];

}

delete[] m\_gg, m\_fg, m\_c, m\_f;

}

void GetSolve()

{

LUDecomposition LUdec(m\_gg, m\_vector\_size);

LUdec.SolveSOLE(m\_c, m\_fg);

}

//Дискретный вариант

void GetNormError()

{

//Погрешность

FuncType fun\_G = [&](double x) {

return m\_c[0] + (m\_c[1] \* x) + (m\_c[2] \* x \* x);

};

double G = 0;

for (double x = m\_a; x < m\_b; x += m\_step)

{

G += pow(fun\_G(x), 2);

}

double F = 0;

for (size\_t i = 0; i < 1.0 / m\_step + 1; i++)

{

F += m\_f[i] \* m\_f[i];

}

normError = sqrt(F - G);

}

//Непрерывный вариант

void GetNormError(double ff, double gg)

{

normError = sqrt(abs(ff - gg));

}

void PrintSolve()

{

printf("Матрица\n");

for (size\_t i = 0; i < m\_vector\_size; i++)

{

for (size\_t j = 0; j < m\_vector\_size; j++)

{

cout << fixed << setw(10) << setprecision(5) << m\_gg[i][j] << "\t";

}

cout << endl;

}

printf("\nВектор правых частей\n");

for (size\_t i = 0; i < m\_vector\_size; i++)

{

printf("%.13f\t", m\_fg[i]);

}

printf("\n\nP2(x) = (%.5f) + (%.5f)\*x + (%.5f)\*x^2\n", m\_c[0], m\_c[1], m\_c[2]);

printf("Норма погрешности: %.16f\n", normError);

}

double getIntegralOfG2(double x) {

return pow(m\_c[2], 2) \* pow(x, 5) / 5 +

m\_c[1] \* m\_c[2] \* pow(x, 4) / 2 +

(pow(m\_c[1], 2) + 2 \* m\_c[0] \* m\_c[2]) \* pow(x, 3) / 3 +

m\_c[0] \* m\_c[1] \* pow(x, 2) +

pow(m\_c[0], 2) \* x;

}

};

int main()

{

system("chcp 1251");

int N = 5;

double A = 1;

double B = 2;

//функция и её производная

Function func(

[](double x) { return pow(3, x) + 2 - x; },

[](double x) { return pow(3, x) \* log(3) - 1; }

);

//максимальная производная n-ого порядка функции f на отрезке [1, 2] (нужна для формулы оценки интерполяции)

double M6 = 15.824; //6 порядка

double M5 = 14.403; //5 порядка

double M4 = 13.111; //4 порядка

printf("Интерполяционная формула Ньютона\n");

auto nodesX = [&](int i) { return A + i \* (B - A) / N; };

auto nodesY = [&](int i) { return func.m\_func(nodesX(i)); };

NodeTable nodeTable(N, nodesX, nodesY);

NewtonianInterpolation newtonianInterpolation(func, nodeTable);

newtonianInterpolation.printDivDifferTable();

printf("\nM6 = %f\n", M6);

Table table({ "x", "f(x)", "Pn(x)", "Delta", "Оценка" }, { 2, 5, 5, 15, 15 });

table.printTableHeader();

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

double x = 1.0 + (i + 0.5) \* (2.0 - 1.0) / N;

double Pn = newtonianInterpolation.interpolate(x);

//расчет оценки

double w = 1;

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

w \*= x - nodeTable.m\_x\_nodes(i);

double deltaEst = w \* M6 / 720;

table.printTableRow({ x, func.m\_func(x), Pn, abs(func.m\_func(x) - Pn), abs(deltaEst) });

}

printf("\nИнтерполяция кубическим сплайном\n");

SplineInterpolation splineInterpolation(func, nodeTable);

//Вычисление производных в узлах интерполяции

printf("\nM5 = %f\n", M5);

Table table2({ "x[i]", "df/dx(x[i])", "m[i]", "Delta", "Оценка" }, { 2, 7, 7, 15, 15 });

table2.printTableHeader();

for (int i = 0; i < N + 1; i++) {

auto x = nodeTable.m\_x\_nodes(i);

auto dF = func.m\_derivative(x);

auto m = splineInterpolation.m\_vectorM[i];

double h = nodeTable.getH(0);

double deltaEst = M5 \* pow(h, 4) / 60;

table2.printTableRow({ x, dF, m, abs(dF - m), deltaEst });

}

//Вычисление промежуточных значений

printf("\nM4 = %f\n", M4);

Table table3({ "x", "f(x)", "S31(f;x)", "Abs(f(x)-S31(f;x))", "Оценка" }, { 2, 5, 5, 15, 15 });

table3.printTableHeader();

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

double x = 1.0 + (i + 0.5) \* (2.0 - 1.0) / N;

double deltaEst;

double S31 = splineInterpolation.interpolate(x, &deltaEst, M4, M5);

double h = nodeTable.getH(0);

double deltaEst2 = (M4 / 384 + M5 \* h / 240) \* pow(h, 4); //deltaEst < deltaEst2

table3.printTableRow({ x, func.m\_func(x), S31, abs(func.m\_func(x) - S31), deltaEst });

}

printf("\nСреднеквадратичное приближение\n\n");

RmsApproximation disRms(func.m\_func, 1, 2);

disRms.GetSolve();

disRms.GetNormError();

printf("Дискретный вариант\n");

disRms.PrintSolve();

double fg[3] = {

5.96143535976102,

9.349042367927889,

15.148483258152980

};

double gg[3][3] = {

{1., 3. / 2., 7. / 3.},

{3. / 2., 7. / 3., 15. / 4.},

{7. / 3., 15. / 4., 31. / 5.}

};

//double f[6] = { func.m\_func(1.0), func.m\_func(1.2), func.m\_func(1.4), func.m\_func(1.6), func.m\_func(1.8), func.m\_func(2.0), };

RmsApproximation contRms(gg, fg);

contRms.GetSolve();

contRms.GetNormError(37.5829, contRms.getIntegralOfG2(B) - contRms.getIntegralOfG2(A));

printf("\nНепрерывный вариант\n");

contRms.PrintSolve();

printf("\nРешение уравнения методом обратной интерполяции\n");

//выбираем отрезок [a, b], где находится предполагаемый корень, а также выбираем n отрезков разбиения

int n = 5;

double a = 1.0;

double b = 2.0;

double c = func.m\_func(1.5); //правая часть уравнения f(x) = c

auto nodesX2 = [&](int i) { return a + i \* (b - a) / n; };

auto nodesY2 = [&](int i) { return func.m\_func(nodesX2(i)); };

NodeTable nodeTable2(n, nodesY2, nodesX2); //наоборот

NewtonianInterpolation newtonianInterpolation2(func, nodeTable2);

newtonianInterpolation2.printDivDifferTable();

auto x = newtonianInterpolation2.interpolate(c);

printf("c = %.4f\n", c);

printf("Корень=%.5f\n", x);

printf("Невязка=Abs(f(x)-c)=%f\n", abs(func.m\_func(x) - c));

}