|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 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](#_Toc55237975)

[Исходные данные 4](#_Toc55237976)

[Тестирование 8](#_Toc55237977)

[Краткие выводы 12](#_Toc55237978)

[Текст программы 13](#_Toc55237979)

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: 

**Непрерывный (интегральный) вариант**

Матрица: 

Свободные члены: 

**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:



Графики исходной и приближенной функции

Вариант 21:

f(x)

P2(x)

График

Вариант 25:

Тестирование

Вариант 21

Вариант 25

Краткие выводы

Текст программы

#include <iostream>

#include <algorithm>

#include <math.h>

#include <fstream>

#include <functional>

#include <iomanip>

using namespace std;

#define M\_PI 3.14159265358979323846

const double minValue = 0.00001;

const double eps = 0.0001;

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

//распечатка вектора

void printVector(double\* vector, int N, bool expFormat = false) {

printf(" ");

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

if (expFormat) {

printf("%.10e ", vector[i]);

}

else {

printf("%.7f ", vector[i]);

}

}

printf("\n");

}

//распечатка матрицы

void printMatrix(double\*\* matrix, int N, bool expFormat = false) {

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

printVector(matrix[i], N, expFormat);

}

printf("\n");

}

//создаем в памяти матрицу

void createMatrix(double\*\*\* pMatrix, int N) {

auto& matrix = \*pMatrix;

matrix = new double\* [N];

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

matrix[i] = new double[N];

}

}

//заполняем матрицу нулями

void fillMatrixAsEmpty(double\*\* matrix, int N) {

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

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

matrix[i][j] = 0.0;

}

}

}

//заполняем матрицу как единичную

void fillMatrixAsE(double\*\* matrix, int N) {

fillMatrixAsEmpty(matrix, N);

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

matrix[i][i] = 1.0;

}

}

//копируем значения одной матрицы в другую

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];

}

}

}

//копируем значения одной матрицы в другую

void copyVectorToVector(double\* srcVector, double\* dstVector, int N) {

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

dstVector[i] = srcVector[i];

}

}

//очистка вектора

void vectorClear(double\* A, int N)

{

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

A[i] = 0.0;

}

//умножение матрицы на вектор

void matrixMulVec(double\*\* A, double\* B, double\* C, int N)

{

vectorClear(C, N);

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

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

C[i] += A[i][j] \* B[j];

}

//умножение матрицы на матрицу

void matrixMul(double\*\* A, double\*\* B, double\*\* C, int N)

{

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

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

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

C[i][j] += A[i][k] \* B[k][j];

}

//умножение матрицы на скаляр

void matrixMulScalar(double\*\* A, double scalar, double\*\* C, int N)

{

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

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

C[i][j] = A[i][j] \* scalar;

}

//вычитание матрицы

void matrixSub(double\*\* A, double\*\* B, double\*\* C, int N, double scalar = 1.0)

{

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

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

C[i][j] = A[i][j] - B[i][j] \* scalar;

}

//получение матрицы PA

double\*\* matrixPA(double\*\* A, int\* P, int N)

{

double\*\* PA = new double\* [N];

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

PA[i] = A[P[i]];

}

return PA;

}

//сложение векторов

void vectorAdd(double\* A, double\* B, double\* C, int N, double alpha = 1.0)

{

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

C[i] = A[i] + B[i] \* alpha;

}

//вычитание векторов

void vectorSub(double\* A, double\* B, double\* C, int N)

{

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

C[i] = A[i] - B[i];

}

//умножение вектора на скаляр

void vectorMulScalar(double\* A, double scalar, double\* C, int N)

{

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

C[i] = A[i] \* scalar;

}

//скалярное произведенеие векторов

double vectorMul(double\* v1, double\* v2, int N)

{

double sum = 0;

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

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

return sum;

}

//норма вектора

double vectorEuNorm(double\* vector, int N)

{

double sum = 0;

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

sum += pow(vector[i], 2);

return sqrt(sum);

}

double vectorNorm(double\* v, int N)

{

double max = abs(v[0]);

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

if (abs(v[i]) > max)

max = abs(v[i]);

return max;

}

//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;

}

//LU разложение

void LUdecomposition(double\*\* L, double\*\* U, int\* P, int& rank, double& sign, int N) {

//иницилизируем подстановку P

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

P[i] = i;

}

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

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

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 < N; i++) {

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

//printMatrix(L, N);

}

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

U[k][j] /= mainValue;

}

//заполняем матрицу U

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

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

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

//printMatrix(U, N);

}

}

}

}

//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];

}

}

}

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

void SolveSOLE(double\*\* L, double\*\* U, double\* X, int\* P, double\* B, int N) {

//Ax = b (A = PLU)

//LUx = Pb (Ux = y)

//Ly = Pb

double\* vectorY = new double[N];

//"умножаем" вектор b на матрицу перестановок

double\* vectorPB = new double[N];

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

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

}

SolveLy(L, vectorY, vectorPB, N);

//printVector(Y, N);

SolveUx(U, X, vectorY, N);

}

void SolveBackwardMatrix(double\*\* L, double\*\* U, double\*\* X, int\* P, int N) {

//LUX = PE

double\* vectorX = new double[N];

double\* vectorE = new double[N];

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

vectorE[t] = 0.0;

}

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

//формируем вектор Ei

if (i != 0) {

vectorE[i - 1] = 0.0;

}

vectorE[i] = 1.0;

//получаем вектор-столбец X

SolveSOLE(L, U, vectorX, P, vectorE, N);

//записываем его в матрицу X

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

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

}

}

}

//транспонировать матрицу

void transpose(double\*\* matrix, int N) {

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

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

swap(matrix[j][i], matrix[i][j]);

}

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////

//Нахождение максимального элемента в матрице, находящегося не на диагонали

void searchMaxElemMatrix(double\*\* matrix, const int N, int& imax, int& jmax) {

double max = 0.0;

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

{

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

{

if (i != j && abs(matrix[i][j]) > max)

{

max = abs(matrix[i][j]);

imax = i;

jmax = j;

}

}

}

}

//tan(2\*alpha) = 2\*a[i][j]/(a[i][i]-a[j][j])

double getAlpha(double\*\* matrix, int imax, int jmax) {

double alpha;

if (matrix[imax][imax] - matrix[jmax][jmax] == 0)

{

alpha = M\_PI / 4;

}

else

{

alpha = atan(2 \* matrix[imax][jmax] / (matrix[imax][imax] - matrix[jmax][jmax])) / 2;

}

return alpha;

}

//Является ли матрица диагональной

bool isMatrixDiagonal(double\*\* matrix, const int N) {

double kvSum = 0.0;

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

{

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

{

if (i != j)

{

kvSum += matrix[i][j] \* matrix[i][j];

}

}

}

return kvSum < minValue\* minValue;

}

//Получение диагональной матрицы

double\*\* getNewDiagonalMatrixByRotation(double\*\* matrix, const int N) {

double\* vectorI = new double[N];

double\* vectorJ = new double[N];

double\*\* rotatedMatrix = nullptr;

createMatrix(&rotatedMatrix, N);

copyMatrixToMatrix(matrix, rotatedMatrix, N);

//вспомогательная матрица

double\*\* B = nullptr;

createMatrix(&B, N);

while (!isMatrixDiagonal(rotatedMatrix, N)) {

//printMatrix(rotatedMatrix, N);

int imax, jmax;

searchMaxElemMatrix(rotatedMatrix, N, imax, jmax);

double alpha = getAlpha(rotatedMatrix, imax, jmax);

double c = cos(alpha);

double s = sin(alpha);

//результат умножения матрицы A в k-ом состоянии на матрицу вращения справа

copyMatrixToMatrix(rotatedMatrix, B, N);

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

B[m][imax] = c \* rotatedMatrix[m][imax] + s \* rotatedMatrix[m][jmax];

B[m][jmax] = -s \* rotatedMatrix[m][imax] + c \* rotatedMatrix[m][jmax];

}

//результат умножения матрицы B на матрицу вращения слева

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

vectorI[m] = c \* B[imax][m] + s \* B[jmax][m];

vectorJ[m] = -s \* B[imax][m] + c \* B[jmax][m];

}

swap(B[imax], vectorI);

swap(B[jmax], vectorJ);

copyMatrixToMatrix(B, rotatedMatrix, N);

}

return rotatedMatrix;

}

//Получение максимального собственного значения

double getMaxEigenvalue(double\*\* matrix, const int N) {

double max = 0;

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

{

if (abs(matrix[i][i]) > max)

max = abs(matrix[i][i]);

}

return max;

}

//Вычисление евклидовой нормы матрицы

double computEuclidNorm(double\*\* A, double\*\* trA, const int N) {

double\*\* newMatrix = nullptr;

createMatrix(&newMatrix, N);

fillMatrixAsEmpty(newMatrix, N);

matrixMul(trA, A, newMatrix, N);

//printMatrix(newMatrix, N);

newMatrix = getNewDiagonalMatrixByRotation(newMatrix, N);

//printMatrix(newMatrix, N);

double eigenvalue = getMaxEigenvalue(newMatrix, N);

return sqrt(eigenvalue);

}

//чтение входных данных

void readInputData(const char\* filename, double\*\*\* pA, double\*\* pB, int\* pN) {

std::ifstream file(filename);

if (!file.is\_open())

throw std::exception();

int idx = 0;

double value;

file >> \*pN;

int N = \*pN;

createMatrix(pA, N);

\*pB = new double[N];

while (file >> value)

{

if (idx < N \* N) {

(\*pA)[idx / N][idx % N] = value;

}

else {

(\*pB)[idx % N] = value;

}

idx++;

}

}

void printIterHeader(bool isJnorm = true)

{

cout << setw(7) << "Itr" << "|" << setw(12) << "x" << "|" << setw(12) << "y" << "|" << setw(20) << "Норма невязки" << "|" << setw(20) << "F1" << "|";

cout << setw(20) << "F2" << "|" << setw(20);

if (isJnorm)

cout << "Норма якобиана";

cout << "|\n";

}

void printIterStep(int itr, double x, double y, double Rnorm, double F1, double F2, double Jnorm, bool isJnorm = true)

{

cout << setw(7) << itr << "|";

cout << setw(12) << setprecision(7) << fixed << x << "|";

cout << setw(12) << setprecision(7) << fixed << y << "|";

cout << setw(20) << setprecision(15) << Rnorm << "|";

cout << setw(20) << setprecision(15) << F1 << "|";

cout << setw(20) << setprecision(15) << F2 << "|";

if (isJnorm)

cout << setw(20) << setprecision(15) << Jnorm << "|";

cout << endl;

}

void printIterHeader2()

{

cout << setw(7) << "Itr" << "|" << setw(12) << "x" << "|" << setw(12) << "y" << "|" << setw(12) << "alpha" << "|" << setw(20) << "Норма невязки" << "|" << setw(20) << "F1" << "|";

cout << setw(20) << "F2" << "|" << setw(20);

cout << "FF" << "|" << setw(12) << "k\n";

}

void printIterStep2(int itr, double x, double y, double alpha, double Rnorm, double F1, double F2, double FF, int k)

{

cout << setw(7) << itr << "|";

cout << setw(12) << setprecision(7) << fixed << x << "|";

cout << setw(12) << setprecision(7) << fixed << y << "|";

cout << setw(12) << setprecision(7) << fixed << alpha << "|";

cout << setw(20) << setprecision(15) << Rnorm << "|";

cout << setw(20) << setprecision(15) << F1 << "|";

cout << setw(20) << setprecision(15) << F2 << "|";

cout << setw(20) << setprecision(15) << FF << "|";

cout << setw(12) << setprecision(7) << fixed << k;

cout << endl;

}

//вычислить якобиан в точке

void calculateJacobian(double\*\* outMatrix, double\* vectorX, FuncType Jacobian[2][2]) {

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

outMatrix[i][j] = Jacobian[i][j](vectorX);

}

}

}

//метод простой итерации

void SimpleIterationMethod(double\*\* calcJ, double\*\* trCalcJ, double\* vectorX, FuncType F[2], FuncType Fi[2], FuncType Jacobian[2][2])

{

auto prevVectorX = new double[2]{ 0, 0 };

auto vectorF = new double[2];

auto tempVector = new double[2];

double error = 0;

int itr = 1;

printIterHeader(true);

do {

for (int i = 0; i < 2; i++) {

tempVector[i] = Fi[i](vectorX);

}

for (int i = 0; i < 2; i++) {

vectorX[i] = tempVector[i];

}

for (int i = 0; i < 2; i++) {

vectorF[i] = F[i](vectorX);

}

//норма невязки

auto Rnorm = vectorEuNorm(vectorF, 2);

//норма якобиана

calculateJacobian(calcJ, vectorX, Jacobian);

copyMatrixToMatrix(calcJ, trCalcJ, 2);

transpose(trCalcJ, 2);

double euclidNorm = computEuclidNorm(calcJ, trCalcJ, 2);

//вычисление погрешности

vectorSub(vectorX, prevVectorX, tempVector, 2);

error = vectorNorm(tempVector, 2);

printIterStep(itr, vectorX[0], vectorX[1], Rnorm, vectorF[0], vectorF[1], euclidNorm, true);

copyVectorToVector(vectorX, prevVectorX, 2);

itr++;

} while (error > eps);

}

//заполнение матрицы производных

void calculateDFMatrix(double\*\* matrix, double\* vectorX, FuncType Derivative[2][2])

{

for (int i = 0; i < 2; i++)

{

for (int j = 0; j < 2; j++)

{

matrix[i][j] = Derivative[i][j](vectorX);

}

}

}

//Метод Ньютона

void NewtonsMethod(double\* vectorX, FuncType F[2], FuncType Derivative[2][2])

{

int itr = 1;

double\* tempVector = new double[2];

double\* vectorF = new double[2];

double\*\* DFMatrix = nullptr;

createMatrix(&DFMatrix, 2);

double\* vectorDelta = new double[2];

double error = 0;

printIterHeader(false);

for (int i = 0; i < 2; i++) {

vectorF[i] = F[i](vectorX);

}

//Матрица U

double\*\* U = nullptr;

createMatrix(&U, 2);

//Матрица L

double\*\* L = nullptr;

createMatrix(&L, 2);

do

{

//вычисляем матрицу производных в точке

calculateDFMatrix(DFMatrix, vectorX, Derivative);

//решаем СЛАУ методом LU-разложения

fillMatrixAsEmpty(L, 2);

copyMatrixToMatrix(DFMatrix, U, 2);

int rank = 2; //ранг матрицы

int P[2]; //"матрица" перестановок (на самом деле подстановка)

double sign = 1.0;

LUdecomposition(L, U, P, rank, sign, 2);

if (rank != 2)

throw std::exception();

for (int i = 0; i < 2; i++) {

tempVector[i] = -vectorF[i];

}

double vectorDeltaX[2];

SolveSOLE(L, U, vectorDeltaX, P, tempVector, 2);

//получили дельта вектор, откуда надо вычленить след. приближение

vectorAdd(vectorDeltaX, vectorX, tempVector, 2);

copyVectorToVector(tempVector, vectorX, 2);

for (int i = 0; i < 2; i++) {

vectorF[i] = F[i](vectorX);

}

//Норма невязки

double Rnorm = vectorEuNorm(vectorF, 2);

//дельта

error = vectorNorm(vectorDeltaX, 2);

printIterStep(itr, vectorX[0], vectorX[1], Rnorm, vectorF[0], vectorF[1], 0, false);

itr++;

} while (error > eps);

}

void GradientDescentMethod(double\* vectorX, FuncType F[2], FuncType Der[2], FuncType OptF)

{

int itr = 1;

auto prevVectorX = new double[2]{ 0, 0 };

double\* tempVector = new double[2];

double\* vectorDer = new double[2];

auto vectorF = new double[2];

double alpha = 1.0;

double lamda = 0.5;

double error = 0.0;

printIterHeader2();

for (int i = 0; i < 2; i++) {

vectorDer[i] = Der[i](vectorX);

}

do {

int k = 0;

double alphaK = alpha;

double Fvalue = OptF(vectorX);

double newFvalue = 0;

do {

//пытаемся обнаружить место, где значение функции меньше, чем в текущем положении (если мы перепрыгнули например минимум)

vectorAdd(vectorX, vectorDer, tempVector, 2, -alphaK);

alphaK \*= lamda;

k++;

newFvalue = OptF(tempVector);

} while (newFvalue >= Fvalue);

copyVectorToVector(tempVector, vectorX, 2);

for (int i = 0; i < 2; i++) {

vectorDer[i] = Der[i](vectorX);

}

for (int i = 0; i < 2; i++) {

vectorF[i] = F[i](vectorX);

}

//норма невязки

auto Rnorm = vectorEuNorm(vectorF, 2);

//вычисление погрешности

vectorSub(vectorX, prevVectorX, tempVector, 2);

error = vectorNorm(tempVector, 2);

printIterStep2(itr, vectorX[0], vectorX[1], alphaK, Rnorm, vectorF[0], vectorF[1], OptF(vectorX), k);

copyVectorToVector(vectorX, prevVectorX, 2);

itr++;

} while (error > eps);

}

int main()

{

system("chcp 1251");

//система уравнений

FuncType F[2] = {

[](double\* vectorX) {

return vectorX[0] + cos(vectorX[1] + 0.5) - 0.8;

},

[](double\* vectorX) {

return sin(vectorX[0]) - 2 \* vectorX[1] - 1.6;

}

};

double\* vectorX0 = new double[2]{ -0.2, -0.8 };

double\* vectorX = new double[2];

printf("Вариант №25\n\n");

printf("x0 = %.3f y0 = %.3f\n\n", vectorX0[0], vectorX0[1]);

printf(" Метод простой итерации\n");

printf("Fi1(x,y)=0.8-cos(y+0.5)\n");

printf("Fi2(x,y)=sin(x)/2 - 0.8\n\n");

printf("Якобиан\n");

printf("0\tsin(y+0.5)\n");

printf("cos(x)/2\t0\n");

printf("\nзначение\n");

//векторная функция фи

FuncType Fi[2] = {

[](double\* vectorX) {

return 0.8 - cos(vectorX[1] + 0.5);

},

[](double\* vectorX) {

return sin(vectorX[0]) / 2 - 0.8;

},

};

//якобиан

FuncType Jacobian[2][2] = {

{

[](double\* vectorX) {

return 0;

},

[](double\* vectorX) {

return sin(vectorX[1] + 0.5);

}

},

{

[](double\* vectorX) {

return cos(vectorX[0]) / 2.0;

},

[](double\* vectorX) {

return 0;

}

},

};

double\*\* calcJacobian = nullptr;

createMatrix(&calcJacobian, 2);

double\*\* trCalcJacobian = nullptr;

createMatrix(&trCalcJacobian, 2);

calculateJacobian(calcJacobian, vectorX, Jacobian);

copyMatrixToMatrix(calcJacobian, trCalcJacobian, 2);

transpose(trCalcJacobian, 2);

printMatrix(calcJacobian, 2);

double euclidNorm = computEuclidNorm(calcJacobian, trCalcJacobian, 2);

printf("Евклидова норма = %.5f\n", euclidNorm);

copyVectorToVector(vectorX0, vectorX, 2);

//метод простой итерации

SimpleIterationMethod(calcJacobian, trCalcJacobian, vectorX, F, Fi, Jacobian);

printf("\n\nМетод Ньютона\n\n");

printf("Матрица производных\n");

printf("1\t-sin(y+0.5)\n");

printf("cos(x)\t-2\n\n");

//матрица производных

FuncType Derivative[2][2] = {

{

[](double\* vectorX) {

return 1.0;

},

[](double\* vectorX) {

return -sin(vectorX[1] + 0.5);

}

},

{

[](double\* vectorX) {

return cos(vectorX[0]);

},

[](double\* vectorX) {

return -2.0;

}

}

};

copyVectorToVector(vectorX0, vectorX, 2);

//Метод Ньютона

NewtonsMethod(vectorX, F, Derivative);

printf("\n\nМетод градиентного спуска\n");

printf("Вектор - градиент:\n");

printf("F'(x) = (x+cos(y+0.5)-0.8) + cos(x)(sin(x)-2y-1.6)\n");

printf("F'(y) = -sin(y+0.5)(x+cos(y+0.5)-0.8) - 2(sin(x)-2y-1.6)\n\n");

//целевая функция для поиска минимума

FuncType optF = [&](double\* vectorX) {

return pow(F[0](vectorX), 2) + pow(F[1](vectorX), 2);

};

//вычисление производной суммы квадратов компонент вектора-функции F

FuncType Der[2] = {

[&](double\* vectorX) {

return F[0](vectorX) \* Derivative[0][0](vectorX) + F[1](vectorX) \* Derivative[1][0](vectorX);

},

[&](double\* vectorX) {

return F[0](vectorX) \* Derivative[0][1](vectorX) + F[1](vectorX) \* Derivative[1][1](vectorX);

}

};

copyVectorToVector(vectorX0, vectorX, 2);

//метод градиентного спуска

GradientDescentMethod(vectorX, F, Der, optF);

return 0;

}