Московский авиационный институт

(Национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра «Вычислительная математика и программирование»

Лабораторные работы

по курсу «Численные методы»

Вариант 15

Выполнил: Наседкин Г.К.

Группа: М8О-305Б-20

Проверила: доц. Демидова О. Л.

Дата:

Оценка:

Москва, 2023

Оглавление

Лабораторная работа 1

Задание 1

Реализовать алгоритм LU -разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

$$\begin{cases} -9 \cdot x_1 + 8 \cdot x_2 + 8 \cdot x_3 + 6 \cdot x_4 = -81 \\ -7 \cdot x_1 - 9 \cdot x_2 + 5 \cdot x_3 + 4 \cdot x_4 = -50 \\ -3 \cdot x_1 - x_2 + 8 \cdot x_3 = -69 \\ 3 \cdot x_1 - x_2 - 4 \cdot x_3 - 5 \cdot x_4 = 48 \end{cases}$$

Теоретические сведения

LU-разложение матрицы представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т. е. A=LU, где L — нижняя треугольная матрица, U — верхняя треугольная матрица.

LU-разложение может быть построено с использованием метода Гаусса. Рассмотрим k-й шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k-го столбца матрицы $A^{(k-1)}$. С этой целью используется операция:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \mu_i^{(k)} a_{kj}^{(k-1)}, \text{ где } \mu_i^{(k)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}; i = \overline{k+1, n}; j = \overline{k, n}$$

В терминах матричных операций такая операция эквивалентна умножению $A^{(k)} = M_k A^{(k-1)}$, где элементы матрицы M определяются следующим образом

$$m_{ij}^k = \begin{cases} 1, i = j \\ 0, i \neq j, j \neq k \\ -\mu_{k+1}^{(k)}, i \neq j, j = k \end{cases}$$

В результате прямого хода метода Гаусса получим $A^{(n-1)}$=U,

$$A = A^{(0)} = M_1^{-1} A^{(1)} = M_1^{-1} M_2^{-1} A^{(2)} = M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1} A^{(n-1)},$$

де $A^{(n-1)}=U$ — верхняя треугольная матрица, а $L=M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}$ — нижняя треугольная матрица, имеющая вид

$$L=\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \mu_2^{(1)} & 1 & 0 & 0 & 0 & 0 \\ \mu_3^{(1)} & \mu_3^{(2)} & 1 & 0 & 0 & 0 \\ \dots & \dots & \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mu_n^{(1)} & \mu_n^{(2)} & \mu_n^{(k+1)} & \dots & \mu_n^{(n-1)} & 1 \end{pmatrix}$$

В дальнейшем LU-разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида Ax=b. Действительно, подставляя LU-разложение в СЛАУ, получим LUx=b, или Ux = $L^{-1}b$ $Ux=L^{-1}b$. Т.е. процесс решения СЛАУ сводится к двум простым этапам:

1) На первом этапе решается СЛАУ $Ly=b$. Поскольку матрица системы – нижняя треугольная, решение можно записать в явном виде:

$$y_1=b_1,\ y_i=b_i-\sum_{j=1}^{i-1} l_{ij}y_j,\ i=\overline{2,n}$$

2) На втором этапе решается СЛАУ $Ux=y$ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

$$x_1=\frac{y_n}{u_{nn}},\ x_i=\frac{1}{u_{ii}}$$

Зная LU-разложение легко также получить определитель матрицы A и обратную ей матрицу. Определитель находится как произведение элементов на главных диагоналях матриц L и U:

$$detA=\prod_{i=1}^{n} l_{ii}\cdot u_{ii}$$

Обратную матрицу можно найти из отношения $AA^{-1}=LUA^{-1}=E$. Это уравнение также можно решить методом LU-разложения.

Код

Matrix.h

```
#pragma once
#include <vector>
#include <fstream>
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

//////////////////////////////////////////////////////////
// Class
//////////////////////////////////////////////////////////

template <typename _T>
class Matrix {
public:
    vector<vector<_T>> data;
    pair<size_t, size_t> dim;

    Matrix(size_t, size_t=0); // create matrix nxm
    Matrix(pair<size_t, size_t> p) : Matrix(p.first, p.second) {}
    Matrix() : Matrix((int)0) {} // defolt constructor
    Matrix(Matrix<_T>&); // copy matryx
    Matrix(char*); // load matrix from file
    Matrix(vector<_T>, size_t=0, size_t=0); // load matrix from vector
    vector<_T>& operator[](size_t); // get line
    void printM(); // print matrix
    void recordM(char*); // record matrix to file
    void sumL(size_t, size_t, _T, size_t=0, size_t=0); // adds the first row to the second row
with a coefficient
    void subL(size_t i, size_t j, _T coef, size_t from=0, size_t to=0) { this->sumL(i, j, -coef,
from, to); }
    void sumC(size_t, size_t, _T, size_t=0, size_t=0); // adds the first row to the second row
with a coefficient
    void subC(size_t i, size_t j, _T coef, size_t from=0, size_t to=0) { this->sumC(i, j, -coef,
from, to); }
    void swapL(size_t, size_t); // swap lines
    void swapC(size_t, size_t); // swap colomns
    pair<size_t, size_t> max(size_t, size_t, size_t, size_t); // max in rect
    size_t maxC(size_t, size_t=0, size_t=0); // max in one colomn
    size_t maxL(size_t, size_t=0, size_t=0); // max in one line
    Matrix<_T> T();
    double norm2();
    double normC();
    void resize(size_t, size_t);
};

//////////////////////////////////////////////////////////
// Operators
//////////////////////////////////////////////////////////

template <typename _T>
Matrix<_T> operator*(Matrix<_T> left, Matrix<_T> right) {
    if (left.dim.second != right.dim.first) {
        cerr << "Dimension error: can't multiply matrix " << left.dim.first << "x" <<
                left.dim.second << " with matrix " << right.dim.first << "x" << right.dim.second
<< "\n";
        exit(1);
    }
    Matrix<_T> res(left.dim.first, right.dim.second);
    for (size_t i = 0; i < left.dim.first; i++)
        for (size_t j = 0; j < right.dim.second; j++)
            for (size_t k = 0; k < right.dim.first; k++)
                res[i][j] += left[i][k] * right[k][j];
    return res;
}

template <typename _T>
Matrix<_T> operator-(Matrix<_T> m) {
    Matrix<_T> res(m.dim);
    for (size_t i = 0; i < res.dim.first; i++)
        for (size_t j = 0; j < res.dim.second; j++)
            res[i][j] = -m[i][j];
    return res;
}
```

```
template <typename _T>
Matrix<_T> operator+(Matrix<_T> left, Matrix<_T> right) {
    if (left.dim.first != right.dim.first && left.dim.second != right.dim.second) {
        cerr << "Dimension error: can't sum matrix " << left.dim.first << "x" <<
                left.dim.second << " with matrix " << right.dim.first << "x" << right.dim.second
<< "\n";
        exit(1);
    }
    Matrix<_T> res(left);
    for (size_t i = 0; i < left.dim.first; i++)
        for (size_t j = 0; j < right.dim.second; j++)
            res[i][j] = res[i][j] + right[i][j];
    return res;
}
template <typename _T>
Matrix<_T> operator-(Matrix<_T> left, Matrix<_T> right) { return left + (-right); }

template <typename _T>
Matrix<_T> operator*(Matrix<_T> left, _T right) {
    Matrix<_T> res(left);
    for (size_t i = 0; i < left.dim.first; i++)
        for (size_t j = 0; j < left.dim.second; j++)
            res[i][j] = left[i][j] * right;
    return res;
}
template <typename _T>
Matrix<_T> operator*(_T left, Matrix<_T> right) { return right * left; }

////////////////////////////////////////////////////////////
// Realization
////////////////////////////////////////////////////////////

template <typename _T>
Matrix<_T>::Matrix(size_t n, size_t m) {
    m = (m == 0) ? n : m;
    this->dim = {n, m};
    this->data.resize(n, vector<_T>(m));
}

template <typename _T>
Matrix<_T>::Matrix(Matrix<_T>& m) {
    this->dim = m.dim;
    this->data = m.data;
}

template <typename _T>
Matrix<_T>::Matrix(char* s) {
    ifstream input(s);
    input >> this->dim.first >> this->dim.second;
    this->data.resize(this->dim.first, vector<_T>(this->dim.second));
    for (size_t i = 0; i < this->dim.first; i++)
        for (size_t j = 0; j < this->dim.second; j++)
            input >> this->data[i][j];
}

template <typename _T>
Matrix<_T>::Matrix(vector<_T> v, size_t n, size_t m) {
    n = (n == 0) ? 1        : n;
    m = (m == 0) ? v.size() : m;
    this->dim = {n, m};
    this->data.resize(n, vector<_T>(m));
    for (size_t i = 0; i < n; i++)
        for (size_t j = 0; j < m; j++)
            this->data[i][j] = v[i * m + j];
}

template <typename _T>
vector<_T>& Matrix<_T>::operator[](size_t i) { return this->data[i]; }

template <typename _T>
void Matrix<_T>::printM() {
    cout.precision(4);
    for (size_t i = 0; i < this->dim.first; i++) {
        for (size_t j = 0; j < this->dim.second; j++)
            cout << this->data[i][j] << '\t';
        cout << '\n';
    }
}
```

```
template <typename _T>
void Matrix<_T>::recordM(char* s) {
    ofstream output(s);
    output.precision(4);
    output << this->dim.first << ' ' << this->dim.second << '\n';
    for (size_t i = 0; i < this->dim.first; i++) {
        for (size_t j = 0; j < this->dim.second; j++)
            output << this->data[i][j] << '\t';
        output << '\n';
    }
}

template <typename _T>
void Matrix<_T>::sumL(size_t i, size_t j, _T coef, size_t from, size_t to) {
    to = (to == 0) ? this->dim.second : to;
    for (; from < to; from++)
        this->data[j][from] += coef * this->data[i][from];
}

template <typename _T>
void Matrix<_T>::sumC(size_t i, size_t j, _T coef, size_t from, size_t to) {
    to = (to == 0) ? this->dim.first : to;
    for (; from < to; from++)
        this->data[from][j] += coef * this->data[from][i];
}

template <typename _T>
void Matrix<_T>::swapL(size_t i, size_t j) {
    for (size_t k = 0; k < this->dim.second; k++) swap(this->data[j][k], this->data[i][k]);
}
template <typename _T>
void Matrix<_T>::swapC(size_t i, size_t j) {
    for (size_t k = 0; k < this->dim.first;  k++) swap(this->data[k][j], this->data[k][i]);
}

template <typename _T>
pair<size_t, size_t> Matrix<_T>::max(size_t fromC, size_t toC, size_t fromL, size_t toL) {
    pair<size_t, size_t> res {fromL, fromC}; // {line, colomn}
    for (size_t i; fromC < toC; fromC++)
        for (i = fromL; i < toL; i++)
            if (abs(this->data[res.first][res.second]) < abs(this->data[i][fromC]))
                res = {i, fromC};
    return res; // {line, colomn}
}

template <typename _T>
size_t Matrix<_T>::maxC(size_t Colomn, size_t from, size_t to) {
    to = (to == 0) ? this->dim.second : to;
    return this->max(Colomn, Colomn + 1, from, to).first;
}

template <typename _T>
size_t Matrix<_T>::maxL(size_t Line, size_t from, size_t to) {
    to = (to == 0) ? this->dim.first : to;
    return this->max(from, to, Line, Line + 1).second;
}

template <typename _T>
Matrix<_T> Matrix<_T>::T() {
    Matrix<_T> res(this->dim.second, this->dim.first);
    for (size_t i = 0; i < res.dim.first; i++)
        for (size_t j = 0; j < res.dim.second; j++)
            res[i][j] = this->data[j][i];
    return res;
}

template <typename _T>
double Matrix<_T>::norm2() {
    double res = 0;
    for (size_t i = 0; i < this->dim.first; i++)
        for (size_t j = 0; j < this->dim.second; j++)
            res += pow(this->data[i][j], 2);
    res = pow(res, 0.5d);
    return res;
}

template <typename _T>
double Matrix<_T>::normC() {
    double res = 0, cur = 0;
    for (size_t i = 0; i < this->dim.first; i++) {
```

```
        cur = 0;
        for (size_t j = 0; j < this->dim.second; j++)
            cur += abs(this->data[i][j]);
        res = (cur > res) ? cur : res;
    }
    return res;
}

template <typename _T>
Matrix<_T> E(size_t n) {
    Matrix<_T> res(n);
    for (size_t i = 0; i < n; i++) res[i][i] = 1;
    return res;
}

template <typename _T>
void Matrix<_T>::resize(size_t n, size_t m) {
    this->dim = {n, m};
    this->data.resize(n);
    for (size_t i = 0; i < n; i++) this->data[i].resize(m);
}
```

LU.h

```
#include "../../Matrix.h"

template <typename T>
pair<Matrix<T>, vector<pair<size_t, size_t>>> LUdecompos(Matrix<T>& m) {
    if (m.dim.first != m.dim.second) {
        cerr << "LU decomposition working only for square matrix\n"; exit(1);
    }
    Matrix<T> LUm(m);
    vector<pair<size_t, size_t>> P;

    for (size_t j = 0, i, k; j < m.dim.first - 1; j++) {
        k = LUm.maxC(j, j);
        if (k != j) {
            LUm.swapL(j, k);
            P.emplace_back(j, k);
        }
        for (i = j + 1; i < m.dim.first; i++) {
            LUm[i][j] /= LUm[j][j];
            LUm.subL(j, i, LUm[i][j], j + 1);
        }
    }
    return {LUm, P};
}

template <typename T>
Matrix<T> LUsolve(Matrix<T>& m, Matrix<T>& b) {
    if (m.dim.first != m.dim.second || m.dim.first != b.dim.second) {
        cerr << "LU decomposition working only for square matrix\n"; exit(1);
    }
    Matrix<T> res(b);

    pair<Matrix<T>, vector<pair<size_t, size_t>>> resLU = LUdecompos(m);

    for (size_t n = 0, i, j; n < b.dim.first; n++) {
        for (pair<size_t, size_t>& x: resLU.second)
            swap(res[n][x.first], res[n][x.second]);
        for (i = 0; i < b.dim.second; i++) {
            for (j = 0; j < i; j++)
                res[n][i] -= res[n][j] * resLU.first[i][j];
        }
        cout.flush();
        for (i = b.dim.second - 1; i != INT64_MAX; i--) {
            for (j = i + 1; j < b.dim.second; j++)
                res[n][i] -= res[n][j] * resLU.first[i][j];
            res[n][i] /= resLU.first[i][i];
        }
    }
    return res;
}

template <typename T>
Matrix<T> inverseM(Matrix<T> m) {
    Matrix<T> e = E<T>(m.dim.second);
    return LUsolve(m, e);
```

```
}
```

Main.cpp

```
#include "LU.h"

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    Matrix<double> matrix((char*)"inputM.txt");
    pair<Matrix<double>, vector<pair<size_t, size_t>>> LUres = LUdecompos(matrix);
    Matrix<double> LUmatrix = LUres.first;
    LUmatrix.recordM((char*)"output.txt");

    Matrix<double> b((char*)"inputB.txt");
    Matrix<double> answer = LUsolve(matrix, b);
    answer.recordM((char*)"answer.txt");

    (inverseM(matrix).T() * matrix).printM();

    return 0;
}
```

Ввод

Input.txt

```
4 4
-9 8  8  6
-7 -9 5  4
-3 -1 8  0
3  -1 -4 -5
```

inputB.txt

```
1 4
-81 -50 -69 48
```

Вывод

outpu.txt

```
4 4
-9 8  8  6
0.7778   -15.22   -1.222   -0.6667
0.3333   0.2409   5.628 -1.839
-0.3333  -0.1095  -0.2607  -3.553
```

answer.txt

```
1 4
-1 2.917e-17   -9 -3
```

Консоль

```
1       1.11e-16        2.22e-16        0
-4.857e-17      1       5.551e-17       0
-5.551e-17      9.714e-17       1       5.551e-17
0       0       0       1
```

Задание 2

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

$$\begin{cases} 16 \cdot x_1 - 8 \cdot x_2 = 0 \\ -7 \cdot x_1 - 16 \cdot x_2 + 5 \cdot x_3 = -123 \\ 4 \cdot x_2 + 12 \cdot x_3 + 3 \cdot x_4 = -68 \\ -4 \cdot x_3 + 12 \cdot x_4 - 7 \cdot x_5 = 104 \\ -x_4 + 7 \cdot x_5 = 20 \end{cases}$$

Теоретические сведения

Метод прогонки является частным случаем метода Гаусса. Он применяется для решения СЛАУ с трехдиагональными матрицами. Рассмотрим СЛАУ:

$$\begin{cases} b_1 x_1 + c_1 x_2 = d_1 \\ a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2 \\ a_3 x_2 + b_3 x_3 + c_3 x_4 = d_3 \\ \dots \\ a_{n-1} x_{n-2} + b_{n-1} x_{n-1} + c_{n-1} x_n = d_{n-1} \\ a_n x_{n-1} + b_n x_n = d_n \end{cases}$$

При этом будем полагать, что $a_1 = 0$ и $c_n = 0$. Решение системы можно искать в виде:

$$x_i = P_i x_{i+1} + Q_i, i = \overline{1,n}$$

Здесь P_i и Q_i – прогоночные коэффициенты, определяемые по формулам:

$$P_1 = \frac{-c_1}{b_1}; Q_1 = \frac{d_1}{b_1}$$

$$P_i = \frac{-c_i}{b_i + a_i P_{i-1}}; Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}$$

После того как будут найдены прогоночные коэффициенты (прямой ход), можно вычислить значения неизвестных путем обратной подстановки (обратный ход):

$$x_n = P_n x_{n+1} + Q_n = 0 \cdot x_{n+1} + Q_n = Q_n$$

$$x_{n-1} = P_{n-1} x_n + Q_{n-1}$$

$$x_{n-2}=P_{n-2}x_{n-1}+Q_{n-2}$$

.....

$$x_1=P_1x_2+Q_1$$

Достаточным условием корректности метода прогонки и устойчивости его к погрешностям вычислений является условие преобладания диагональных коэффициентов:

$$|b_i|\geq|a_i|+c_i\vee$$

Код

TMA.h

```
#include "../../Matrix.h"

template <typename T>
bool CheckCondition(Matrix<T>& m) {
    bool res = true;
    int counter = 0;
    for (size_t i = 0; i < m.dim.second; i++) {
        counter += (abs(m[1][i]) > abs(m[0][i]) + abs(m[2][i])) ? 1 : -1;
        res = (res && // предыдущее значение
              abs(m[1][i]) >= abs(m[0][i]) + abs(m[2][i]) && // проверка качель
              (i <= 0 || m.dim.second - 1 <= i || (m[0][i] != 0 && m[2][i] != 0))); // не ноль
при i=2,n-1
    }
    return res && (counter >= 0);
}

template <typename T>
Matrix<T> TMAsolve(Matrix<T>& m, Matrix<T>& d) {
    if (!CheckCondition(m)) cout << "Warning! for this matrix, the method is unstable!\n";
    size_t N = m.dim.second;
    Matrix<T> res(d);

    vector<T> P(N), Q(N);
    P[0] = -m[2][0] / m[1][0]; P[N - 1] = 0;
    for (size_t i = 1; i < N - 1; i++)
        P[i] = -m[2][i] / (m[1][i] + m[0][i] * P[i - 1]);

    for (size_t n = 0; n < d.dim.first; n++) {
        Q[0] = d[n][0] / m[1][0];
        for (size_t i = 1; i < N - 1; i++)
            Q[i] = (d[n][i] - m[0][i] * Q[i - 1]) / (m[1][i] + m[0][i] * P[i - 1]);
        Q[N - 1] = (d[n][N - 1] - m[0][N - 1] * Q[N - 2]) / (m[1][N - 1] + m[0][N - 1] * P[N -
2]);

        res[n][N - 1] = Q[N - 1];
        for (size_t i = N - 2; i != INT64_MAX; i--)
            res[n][i] = res[n][i + 1] * P[i] + Q[i];
    }
    return res;
}
```

main.cpp

```
#include "TMA.hpp"

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    Matrix<double> matrix((char*)"inputABC.txt");
    Matrix<double> b((char*)"inputD.txt");
    Matrix<double> answer = TMAsolve(matrix, b);
    answer.recordM((char*)"answer.txt");
    return 0;
}
```

Вывод

1 5
2 4 -9 8 4

Задание 3

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

$$\begin{cases} -14 \cdot x_1 + 6 \cdot x_2 + x_3 - 5 \cdot x_4 = 95 \\ -6 \cdot x_1 + 27 \cdot x_2 + 7 \cdot x_3 - 6 \cdot x_4 = -41 \\ 7 \cdot x_1 - 5 \cdot x_2 - 23 \cdot x_3 - 8 \cdot x_4 = 69 \\ 3 \cdot x_1 - 8 \cdot x_2 - 7 \cdot x_3 + 26 \cdot x_4 = 27 \end{cases}$$

Теоретические сведения

Рассмотрим СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

с невырожденной матрицей.

Приведем СЛАУ к эквивалентному виду

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \ldots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \ldots + \alpha_{2n}x_n \\ \ldots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \ldots + \alpha_{nn}x_n \end{cases}$$

или в векторно-матричной форме $x = \beta + \alpha x$.

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий. Разрешим систему относительно неизвестных при ненулевых диагональных элементах $a_{ii} \neq 0, i = \overline{1,n}$ (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением). Получим следующие выражения для компонентов вектора β и матрицы α эквивалентной системы:

$$\beta_i = \frac{b_i}{a_{ii}}$$

$$\alpha_{ij} = \frac{-a_{ij}}{a_{ii}}, i \neq j; \alpha_{ij} = 0, i = j$$

В качестве нулевого приближения $x^{(0)}$ вектора неизвестных примем вектор правых частей $x^{(0)}=\beta$. Тогда **метод простых итераций** примет вид:

$$\begin{cases} x^{(0)}=\beta \\ x^{(1)}=\beta+\alpha x^{(0)} \\ x^{(2)}=\beta+\alpha x^{(1)} \\ \dots \\ x^{(k)}=\beta+\alpha x^{(k-1)} \end{cases}$$

Достаточным условием сходимости является диагональное преобладание матрицы A по строкам или по столбцам:

$$|a_{ii}|>\sum_{j=1,i\neq j}^{n} a_{ij} \vee$$

Критерием окончания итерационного процесса может служить неравенство $\left\|x^{(k)}-x^{(k-1)}\right\|\leq\varepsilon$.

Метод простых итераций довольно медленно сходится. Для его ускорения существует **метод Зейделя**, заключающийся в том, что при вычислении компонента x_i^{k+1} вектора неизвестных на (k+1)-й итерации используются $x_1^{k+1}, x_2^{k+1}, \dots, x_{i-1}^{k+1}$, уже вычисленные на (k+1)-й итерации. Тогда метод Зейделя для известного вектора на k-ой итерации имеет вид:

$$\begin{cases} x_1^{k+1}=\beta_1+\alpha_{11}x_1^k+\alpha_{12}x_2^k+\dots+\alpha_{1n}x_n^k \\ x_2^{k+1}=\beta_2+\alpha_{21}x_1^k+\alpha_{22}x_2^k+\dots+\alpha_{2n}x_n^k \\ x_3^{k+1}=\beta_3+\alpha_{31}x_1^k+\alpha_{32}x_2^k+\dots+\alpha_{3n}x_n^k \\ \dots \\ x_n^{k+1}=\beta_n+\alpha_{n1}x_1^{k+1}+\alpha_{n2}x_2^{k+1}+\dots+\alpha_{nn-1}x_n^{k+1}+\alpha_{nn}x_n^k \end{cases}$$

Код

Iter.h

```
#include "../../Matrix.h"

template <typename T>
Matrix<T> Iter(Matrix<T>& m, Matrix<T>& b, double epsilon, bool flag) {
    size_t N = m.dim.second;
    Matrix<T> alpha(m), betha(b);
    for (size_t i = 0; i < N; i++) {
        for (size_t j = 0; j < N; j++)
            if (i != j)
                alpha[i][j] /= -alpha[i][i];
        for (size_t n = 0; n < b.dim.first; n++) betha[n][i] /= alpha[i][i];
        alpha[i][i] = 0;
    }

    double normAlpha = alpha.normC();
    Matrix<T> res(betha.dim);
    for (size_t n = 0; n < b.dim.first; n++) {
        Matrix<T> cur(betha[n], N, 1), was;
        int counter = 0;
        do {
            counter++;
            was = cur;
```

```
            if (flag) cur = betha.T() + alpha * was;
            else
                for (size_t i = 0; i < N; i++) {
                    cur[i][0] = betha[n][i];
                    for (size_t j = 0; j < N; j++)
                        if (i != j)
                            cur[i][0] += alpha[i][j] * cur[j][0];
                        else cur[i][0] += alpha[i][j] * was[j][0];
                }
        } while ((cur - was).normC() * (normAlpha / (1 - normAlpha)) > epsilon);
        cout << "counter = " << counter << '\n';
        for (size_t i = 0; i < N; i++)
            res[n][i] = cur[i][0];
    }
    return res;
}

template <typename T>
Matrix<T> SimpleIter(Matrix<T>& m, Matrix<T>& b, double epsilon) {
    return Iter(m, b, epsilon, true);
}
template <typename T>
Matrix<T> Zaydel(Matrix<T>& m, Matrix<T>& b, double epsilon) {
    return Iter(m, b, epsilon, false);
}
```

main.cpp

```
#include "Iter.h"

double epsilon = 0.01;

int main() {
    Matrix<double> matrix((char*)"inputM.txt");
    Matrix<double> b((char*)"inputB.txt");
    char TheMethod;
    cout << "What TheMethod do you want to use?\n"
    "1 - SimpleIter\n"
    "2 - Zaydel\n";
    do { cout << ">>> "; cin >> TheMethod; } while (TheMethod != '1' && '2' != TheMethod);
    Matrix<double> answer;
    if (TheMethod == '1') answer = SimpleIter(matrix, b, epsilon);
    else                  answer = Zaydel(matrix, b, epsilon);
    answer.recordM((char*)"answer.txt");
    (matrix * answer.T()).printM(); // проверка
    return 0;
}
```

Вывод

answer.txt

```
1 4
-8 -2 -5 -5.331e-05
```

Консоль

```
What TheMethod do you want to use?
1 - SimpleIter
2 - Zaydel
>>> 2
counter = 5
95
-41
69.01
27
```

Задание 4

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

$$\begin{pmatrix} -3 & -1 & 3 \\ -1 & 8 & 1 \\ 3 & 1 & 5 \end{pmatrix}$$

Теоретические сведения

Метод вращений Якоби применим только для симметрических матриц) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^T AU$, где Λ – диагональная матрица с собственными значениями на главной диагонали.

Пусть дана симметрическая матрица A. Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения следующий:

Пусть известна матрица $A^{(k)}$ на k–й итерации, при этом для k=0: $A^{(0)} = A$.

1. Выбирается максимальный по модулю недиагональный элемент $a_{ij}^{(k)}$ матрицы $A^{(k)}\left(a_{ij}^{(k)} \vee \max\limits_{l<m}\left|a_{lm}^{(k)}\right|\right)$.
2. Ставится задача найти такую ортогональную матрицу $U^{(k)}$, чтобы в результате преобразования подобия $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$ произошло обнуление элемента $a_{ij}^{(k+1)}$ матрицы $A^{(k+1)}$. В качестве ортогональной матрицы выбирается матрица вращения, имеющая следующий вид:

$$U^k = \begin{pmatrix} & & i & & j & & \\ 1 & & \vdots & & \vdots & & \\ & \ddots & \vdots & & \vdots & 0 & \\ & & 1 & \vdots & & & \\ \cdots & \cdots & \cos\varphi^{(k)} & \cdots & -\sin\varphi^{(k)} & \cdots & i \\ & & \vdots & 1 & \vdots & & \\ & & \vdots & \ddots & \vdots & & \\ & & \vdots & & 1 & \vdots & \\ \cdots & \cdots & \sin\varphi^{(k)} & \cdots & \cos\varphi^{(k)} & \cdots & j \\ & & \vdots & & \vdots & 1 & \\ & 0 & \vdots & & \vdots & \ddots & \\ & & \vdots & & \vdots & & 1 \end{pmatrix},$$

Угол вращения $\varphi^{(k)}$ определяется из условия $a_{ij}^{(k+1)}=0$:

$$\varphi^{(k)}=\frac{1}{2}arctg\frac{2a_{ij}^{(k)}}{a_{ii}^{(k)}-a_{jj}^{(k)}}$$

причем если $a_{ii}^{(k)}=a_{jj}^{(k)}$, то $\varphi^{(k)}=\frac{\pi}{4}$.

3. Строится матрица $A^{(k+1)}$

$$A^{(k+1)}=U^{(k)T}A^{(k)}U^{(k)}$$

В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

$$t\left(A^{(k+1)}\right)=\left(\sum_{l,m;l<m}\left(a_{lm}^{(k+1)}\right)^2\right)^{\frac{1}{2}}$$

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U=U^{(0)}U^{(1)}\ldots U^{(k)}$.

Код

Jacobi.h

```
#include "../../Matrix.h"

template <typename T>
T t(Matrix<T>& m) {
    T res = 0;
    for (size_t i = 0; i < m.dim.first; i++)
        for (size_t j = i + 1; j < m.dim.first; j++)
            res += m[i][j] * m[i][j];
    return pow(res, 0.5);
}

template <typename T>
pair<vector<T>, Matrix<T>> Jacobi(Matrix<T>& m, double epsilon) {
    size_t N = m.dim.second;
    for (size_t i = 0; i < N; i++)
        for (size_t j = i + 1; j < N; j++)
            if (m[i][j] != m[j][i]) {
                cerr << "Jacobi working only for simmetric matrix\n"; exit(1);
            }
    Matrix<T> alpha(m), u(N);
    Matrix<T> resM = E<T>(N);
    do {
```

```
        for (size_t i = 0; i < N; i++)
            for (size_t j = i + 1; j < N; j++) {
                u = E<T>(N);
                double phi = (alpha[i][i] == alpha[j][j]) ? M_PI / 4 : 0.5 * atan(2 * alpha[i][j]
/ (alpha[i][i] - alpha[j][j]));
                u[i][i] = cos(phi); u[i][j] = -sin(phi);
                u[j][i] = sin(phi); u[j][j] =  cos(phi);
                alpha = u.T() * alpha * u;
                resM = resM * u;
            }
    } while (t(alpha) > epsilon);
    vector<T> resV(N);
    for (size_t i = 0; i < N; i++) resV[i] = alpha[i][i];
    return {resV, resM};
}
```

main.cpp

```
#include "Jacobi.h"

double epsilon = 0.01;

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    Matrix<double> matrix((char*)"input.txt");
    pair<vector<double>, Matrix<double>> answer = Jacobi(matrix, epsilon);
    answer.second.recordM((char*)"answer.txt");
    for (size_t i = 0; i < matrix.dim.first; i++)
        cout << answer.first[i] << '\n';
    return 0;
}
```

Вывод

answer.txt

```
3 3
0.9415   -0.01028 0.337
0.104 0.9596   -0.2614
-0.3207  0.2811   0.9045
```

консоль

```
-4.13231
8.30368
5.82863
```

Задание 5

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

$$\begin{pmatrix} 1 & 7 & -1 \\ -2 & 2 & -2 \\ 9 & -7 & 3 \end{pmatrix}$$

Теоретические сведения

В основе QR-алгоритма лежит представление матрицы в виде $A=QR$, где Q – ортогональная матрица ($Q^{-1}=Q^T$), а R – верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR-разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.

Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

$$H=E-\frac{2}{v^T v}\,v\,v^T,\ где\ v-произвольный\ ненулевой\ столбец.$$

Рассмотрим случай, когда необходимо обратить в нуль все элементы какого-либо вектора кроме первого, т.е. построить матрицу Хаусхолдера такую, что

$$\tilde{b}=Hb,\ b=\left(b_1,b_2,\ldots,b_n\right)^T,\ \tilde{b}=\left(\widetilde{b_1},0,\ldots,0\right)^T$$

Тогда вектор v определится следующим образом:

$$v=b+sign\left(b_1\right)\left\|b\right\|_2 e_1$$

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов получить ее QR – разложение.

Процедура QR-разложения многократно используется в QR-алгоритме вычисления собственных значений. Строится следующий итерационный процесс:

$$A^{(0)} = A,$$

$A^{(0)} = Q^{(0)} R^{(0)}$ – производится QR-разложение,

$A^{(1)} = R^{(0)} Q^{(0)}$ – производится перемножение матриц,

……..

$A^{(k)} = Q^{(k)} R^{(k)}$ – разложение

$A^{(k+1)} = R^{(k)} Q^{(k)}$ – перемножение

Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы $A^{(k)}$ в произведение ортогональной $Q^{(k)}$ и верхней треугольной $R^{(k)}$ матриц, а на втором – полученные матрицы перемножаются в обратном порядке.

При отсутствии у матрицы кратных собственных значений последовательность $A^{(k)}$ сходится к верхней треугольной матрице (в случае, когда все собственные значения вещественны) или к верхней квазитреугольной матрице (если имеются комплексносопряженные пары собственных значений).

Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать следующее неравенство: $\left(\sum_{l=m+1}^{n} \left(a_{lm}^{(k)} \right)^2 \right)^{\frac{1}{2}} \le \varepsilon$. При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

Код

QR.h

```
#include "../../Matrix.h"

template <typename T>
T dist(pair<T, T>& left, pair<T, T>& right) {
    return pow(pow(left.first - right.first, 2) + pow(left.second - right.second, 2), 0.5);
}

template <typename T>
pair<Matrix<T>, Matrix<T>> QRdecompose(Matrix<T>& m) {
    size_t N = m.dim.second;
    Matrix<T> Q = E<T>(N), R(m), v(N, 1), H(N);

    for (size_t i = 0; i < N - 1; i++) {
        T temp = 0;
        for (size_t j = i; j < N; j++) {
            v[j][0] = R[j][i];
            temp = temp + R[j][i] * R[j][i];
        }
        v[i][0] = v[i][0] + ((R[i][i] < 0) ? -pow(temp, 0.5) : pow(temp, 0.5));
        H = E<T>(N) - (v * v.T()) * (T)(2.d / (v.T() * v)[0][0]);
```

```
        R = H * R;
        Q = Q * H;
        v[i][0] = 0;
    }
    return {Q, R};
}

template <typename T>
pair<vector<T>, vector<pair<T, T>>> QRsolve(Matrix<T>& m, double epsilon) {
    size_t N = m.dim.second;
    Matrix<T> A(m);
    vector<pair<T, T>> resC(N);
    vector<T> resR(N);
    size_t counter = 0;
    bool flag;
    vector<bool> complex(N, false);
    do {
        flag = true;
        pair<Matrix<T>, Matrix<T>> qr = QRdecompose(A);
        A = qr.second * qr.first;
        for (size_t i = 0; i < N; i++) {
            T sum = (T)0;
            for (size_t j = i + 1; j < N; j++) sum = sum + A[j][i] * A[j][i];
            if (pow(sum, 0.5) > epsilon) {
                complex[i] = complex[i + 1] = true;
                T b = A[i][i] + A[i + 1][i + 1], c = A[i + 1][i] * A[i][i + 1];
                T d = pow(b, 2) - 4 * c;
                pair<T, T> sqr1 = {-b / 2.d,   pow(d, 0.5) / 2.d},
                           sqr2 = {-b / 2.d, - pow(d, 0.5) / 2.d};
                flag = flag && (max(dist(sqr1, resC[i]), dist(sqr2, resC[i + 1])) < epsilon);
                resC[i++] = sqr1;
                resC[i] = sqr2;
            } else {
                resR[i] = A[i][i];
                complex[i] = false;
            }
        }
        ++counter;
    } while (!flag);
    cout << "counter = " << counter << '\n';
    size_t l = 0, k = 0;
    for (size_t i = 0; i < N; i++)
        if (complex[i]) resC[l++] = resC[i];
        else resR[k++] = resR[i];
    resR.resize(k);
    resC.resize(l);
    A.printM();
    return {resR, resC};
}
```

main.cpp

```
#include "QR.h"

double epsilon = 0.01;

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    Matrix<double> matrix((char*)"input.txt");
    auto [R, C] = QRsolve(matrix, epsilon);
    cout << "R = \n";
    for (size_t i = 0; i < R.size(); i++) cout << R[i] << '\n';
    cout << "C = \n";
    for (size_t i = 0; i < C.size(); i++)
        cout << C[i].first << " + " << C[i].second << "i\n";
    return 0;
}
```

Вывод

```
counter = 57
5.673   7.245   -7.87
-3.318  2.596   5.7
4.307e-25       -2.794e-25      -2.27
R =
-2.27
C =
-4.135 + 6.414i
-4.135 + -6.414i
```

Лабораторная работа 2

Задание 1

Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

$$\sin x - x^2 + 1 = 0$$

Теоретические сведения

Метод простых итераций:

Пусть требуется решить трансцендентное уравнение вида $f(x)=0$. Предположим, что корень уравнения отделен и находится на отрезке $[a,b]$. Кроме того, функция удовлетворяет некоторым дополнительным условиям:

- на концах отрезка функция имеет разные знаки;
- $\forall\ x \in [a,b]: f'(x) \neq 0$;
- первая и вторая производные имеют постоянные знаки.

Для того чтобы построить итерационный процесс, согласно методу простой итерации уравнение $f(x)=0$ заменяется эквивалентным уравнением: $x=\varphi(x)$, причем $\varphi(x)$ – непрерывная функция. Выберем некоторое нулевое приближение x^0, а затем организуем итерационный процесс по схеме:

$$x^{(k+1)} = \varphi\left(x^{(k)}\right)$$

Условие $\left|\varphi'(x)\right| \leq q < 1, \forall\ x \in [a,b]$ является достаточным условием сходимости итераций, если начальное приближение выбрано в некоторой окрестности корня.

Метод Ньютона:

Предположим, что на интервале $[a,b]$ требуется определить корень уравнения $f(x)=0$. Для того чтобы построить итерационный процесс согласно методу Ньютона, непрерывная функция $f(x)$ на интервалах $x \in [a,b]$ должна удовлетворять условиям, аналогичным условиям в методе итераций:

Правило построения итерационной последовательности получается путем замены нелинейной функции $f(x)$ ее линейной моделью на основе

формулы Тейлора. Выберем в окрестности решения уравнения две соседние точки, так что $x^{(k+1)}=x^{(k)}+\varepsilon$, и запишем разложение функции в ряд Тейлора:

$$f\left(x^{(k+1)}\right)=f\left(x^{(k)}\right)+f'\left(x^{(k)}\right)\left(x^{(k+1)}-x^{(k)}\right)+\frac{1}{2}f''\left(x^{(k)}\right)\left(x^{(k+1)}-x^{(k)}\right)^2+\ldots$$

Учитывая, что $f\left(x_{k+1}\right)\approx 0$ и оставляя только линейную часть разложения ряда, запишем соотношение метода Ньютона:

$$x^{(k+1)}=x^{(k)}-\frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$$

Используя условие сходимости метода простых итераций, получим достаточное условие сходимости метода Ньютона в форме

$$\left|f(x)\cdot f''(x)\right|<\left(f'(x)\right)^2, x\in[a,b]$$

Для выбора начального приближения $x^{(0)}$ используется теорема, которая гласит, что в качестве начального приближения нужно выбрать тот конец интервала, где знак функции совпадает со знаком второй производной:

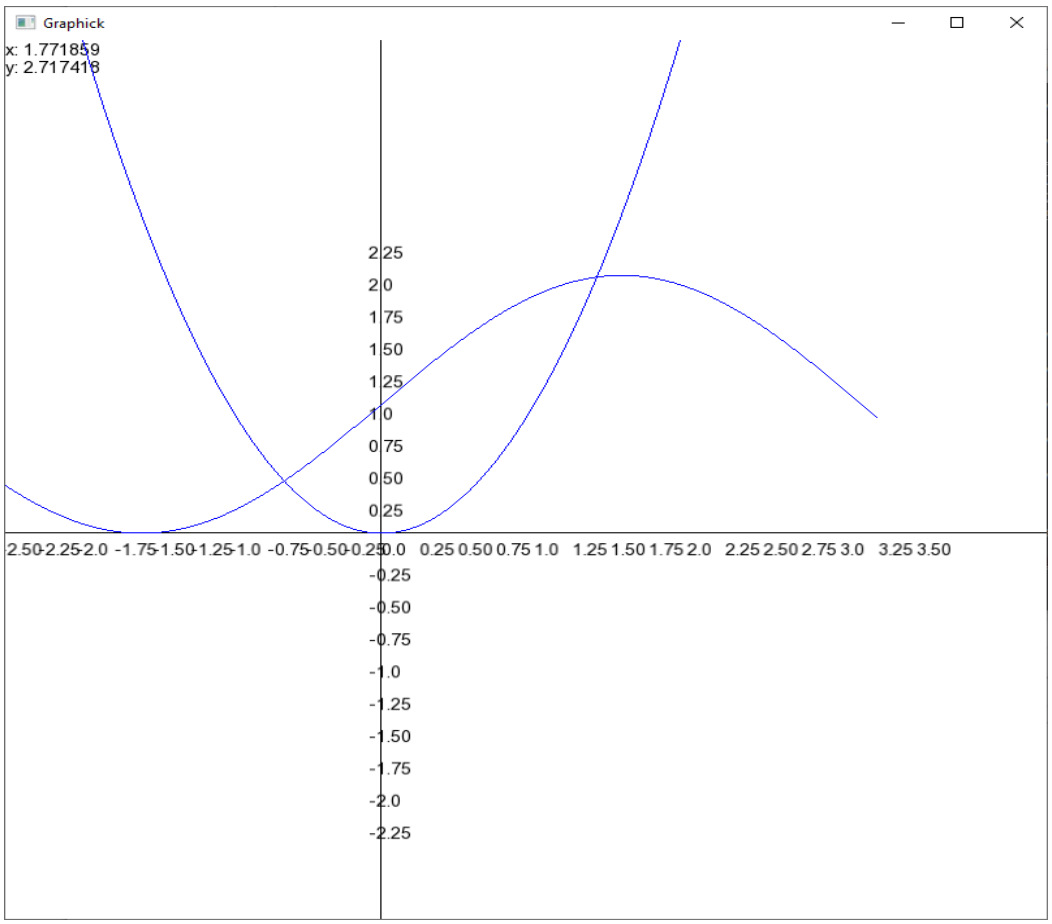
$$x^{(0)}=\begin{cases} a, если f(a)\cdot f''(a)>0 \\ b, если f(b)\cdot f''(b)>0 \end{cases}$$

Для завершения итерационного процесса используется правило:

$$\left|x^{(k+1)}-x^{(k)}\right|<\varepsilon$$

Подготовка уравнения к решению методами Ньютона и простой итерации:

1. Графически определим интервал положительного корня уравнения $f(x)=0$:



$$x_1 \in [1.25, 1.5]$$

2. Найдем 1 и 2 производные функции $f(x)$:

$$f'(x) = \cos(x) - 2x$$

$$f''(x) = -\sin(x) - 2$$

3. Подберем подходящие функции $\varphi(x)$ и найдем их производные:

$$x = \ln\left(3\sqrt{x+1} - 0.5\right) = \varphi_1(x)$$

$$\varphi_1'(x) = \frac{\cos(x)}{2\sqrt{\sin(x)+1}}$$

Код:

Graphica.h

```
#include <cmath>
#include <SFML/Graphics.hpp>
// -lsfml-graphics -lsfml-window -lsfml-system
using namespace std;
using str = std::string;
using Point = sf::Vector2f;
using vp = std::vector<Point>;
using vvp = std::vector<vp>;
int scw = 800, sch = 800;
sf::Event event;
sf::Vector2f CameraPos(scw / 2, sch / 2);
sf::Vector2i MouseBuffer;
sf::Mouse Mouse;
sf::Font font; sf::Text text;
sf::CircleShape PointShape;
float pointRad = 7.f;
```

```
int countOfStep = 5000;
float Start = -3.25, End = 3.25;
float step = (End - Start) / countOfStep;
float MaxX = End + 0.25, MinX = Start - 0.25, MaxY = 2.25, MinY = -2.25;
vvp graph(0);
vp points(0);
vector<sf::Color> colors(0);
float zoom = std::pow(1.1, 50);
bool showPoints = true;

void makeGraph(double (*foo)(double), sf::Color color=sf::Color::Blue) {
    colors.push_back(color);
    graph.push_back({});
    for (double x = Start; x <= End; x += step)
        graph[graph.size() - 1].push_back({float(x), float(-foo(x))});
}
void makeGraphByPoints(vector<pair<double, double>> foo, sf::Color color=sf::Color::Blue) {
    colors.push_back(color);
    graph.push_back({});
    for (pair<double, double>& x: foo)
        graph[graph.size() - 1].push_back({float(x.first), float(-x.second)});
}

void addPoint(double x, double y) {
    points.emplace_back(x, -y);
}

void ShowGraphics() {
    sf::RenderWindow window(sf::VideoMode(scw, sch), "Graphick");
    font.loadFromFile("arial.ttf");
    text.setFont(font);
    text.setCharacterSize(14);
    text.setFillColor(sf::Color::Black);

    PointShape.setFillColor(sf::Color::Green);
    PointShape.setRadius(pointRad);
    while (window.isOpen()) {
        if (window.hasFocus()) {
            if (Mouse.isButtonPressed(Mouse.Left))
                CameraPos += sf::Vector2f(Mouse.getPosition(window) - MouseBuffer);
        }
        MouseBuffer = Mouse.getPosition(window);
        window.clear(sf::Color::White);
        text.setPosition(0, 0);
        str word = "x: "   + std::to_string((MouseBuffer.x - CameraPos.x) / zoom) +
                   "\ny: " + std::to_string((-MouseBuffer.y + CameraPos.y) / zoom);
        text.setString(word);
        window.draw(text);
        // axis
        sf::Vertex line[2] = {
            sf::Vertex(sf::Vector2f(CameraPos.x, 0), sf::Color::Black),
            sf::Vertex(sf::Vector2f(CameraPos.x, sch), sf::Color::Black)
        }; // |
        window.draw(line, 2, sf::Lines);
        // -
        line[0] = sf::Vertex(sf::Vector2f(0, CameraPos.y), sf::Color::Black);
        line[1] = sf::Vertex(sf::Vector2f(scw, CameraPos.y), sf::Color::Black);
        window.draw(line, 2, sf::Lines);
        // numbers on axis
        for (float x = MinX; x <= MaxX; x += 0.25) {
            text.setPosition(sf::Vector2f(x, 6.f / zoom) * zoom + CameraPos);
            text.setString(((x < 0) ? "-" : "\0") + to_string(abs(int(x))) + "." +
to_string(abs(int(x * 100)) % 100));
            window.draw(text);
        }
        for (float y = MinY; y <= MaxY; y += 0.25) {
            if (int(y) == 0 && int(y * 10) % 10 == 0) continue;
            text.setPosition(sf::Vector2f(-10.f / zoom, -y) * zoom + CameraPos);
            text.setString(((y < 0) ? "-" : "\0") + to_string(abs(int(y))) + "." +
to_string(abs(int(y * 100)) % 100));
            window.draw(text);
        }
        // graph
        for (int j = 0; j < graph.size(); j++)
            for (int i = 0; i < graph[j].size() - 1; i++) {
                line[0] = sf::Vertex(sf::Vector2f(graph[j][i].x, graph[j][i].y) * zoom +
CameraPos, colors[j]);
                line[1] = sf::Vertex(sf::Vector2f(graph[j][i + 1].x, graph[j][i + 1].y) * zoom +
CameraPos, colors[j]);
                window.draw(line, 2, sf::Lines);
```

```
            }
        // points
        if (showPoints)
            for (int i = 0; i < points.size(); i++) {
                PointShape.setPosition(sf::Vector2f{points[i].x - pointRad / zoom,
                    points[i].y - pointRad / zoom} * zoom + CameraPos);
                window.draw(PointShape);
            }

        while (window.pollEvent(event))
            if (event.type == sf::Event::Closed) window.close();
            else if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Escape) window.close();
                if (event.key.code == sf::Keyboard::H) showPoints = !showPoints;
            } else if (event.type == sf::Event::MouseWheelScrolled) {
                float coef = std::pow(1.1, event.mouseWheelScroll.delta);
                zoom *= coef;
                CameraPos = (sf::Vector2f)MouseBuffer - ((sf::Vector2f)MouseBuffer - CameraPos) *
coef;
            }
        window.display();
    }
}

void SetDiapazon(float s, float e) {
    Start = s, End = e;
    step = (End - Start) / countOfStep;
    MaxX = ceil(End) + 0.25, MinX = floor(Start) - 0.25;
}
```

tools.h

```
#include <iostream>
#include "../../Graphica.h"
double epsilon = 0.0001;

float Newton(float (*foo)(float), float (*foo1)(float), float (*foo2)(float), float epsilon,
float a, float b) {
    float x = (foo(a) * foo2(a) <= 0 || abs(foo(a) * foo2(a)) >= powf(foo1(a), 2)) ? b : a;

    if (foo(x) * foo2(x) <= 0 || abs(foo(x) * foo2(x)) >= powf(foo1(x), 2)) {
        cerr << "convergence conditions of the method are not met\n"; exit(1);
    }

    float was;
    size_t counter = 0;
    do {
        counter++;
        was = x;
        x = x - foo(x) / foo1(x);
    } while (abs(x - was) > epsilon);
    cout << "counter = " << counter << '\n';
    return x;
}

float Iter(float (*foo)(float), float (*foo1)(float), float epsilon, float a, float b) {
    float q = 0;
    for (float x = a; x <= b; x += (b - a) / 1000) {
        if (foo(x) < a || foo(x) > b) {
            cerr << "convergence conditions of the method are not met\n"; exit(1);
        }
        q = max(q, abs(foo1(x)));
    }
    cout << "q = " << q << '\n';
    float was, x = (b - a) / 2.f;
    size_t counter = 0;
    do {
        counter++;
        was = x;
        x = foo(x);
    } while (abs(x - was) > epsilon);
    cout << "counter = " << counter << '\n';
    return x;
}
```

main.cpp

```
#include "tools.h"

float foo(float x) { return sin(x) - x * x + 1.f; }
float foo1(float x) { return cos(x) - 2 * x; }
```

```
float foo2(float x) { return - sin(x) - 2.f; }

int main() {
    makeGraph([](double x){ return sin(x) + 1; });
    makeGraph([](double x){ return x * x; });
    sf::Thread thread(ShowGraphics); thread.launch();
    float a, b; cout << "a = "; cin >> a; cout << "b = "; cin >> b;
    cout << "Choose method: 1 - Newton, 2 - Iter\n> ";
    int choose; cin >> choose;
    float res;
    if (choose == 1)
        cout << Newton(foo, foo1, foo2, epsilon, a, b) << '\n';
    else
        cout << Iter([](float x) { return powf(sin(x) + 1, 0.5); },
                     [](float x) { return 0.5f * cos(x) / powf(sin(x) + 1, 0.5); },
                     epsilon, a, b) << '\n';
    thread.wait();
    return 0;
}
```

Вывод:

Первый запуск

```
a = 1.25
b = 1.5
Choose method: 1 - Newton, 2 - Iter
> 1
counter = 3
1.40962
```

Второй запуск

```
a = 1.25
b = 1.5
Choose method: 1 - Newton, 2 - Iter
> 2
q = 0.112933
counter = 6
1.40962
```

Задание 2

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

$$x_1^2/a^2+x_2^2/(a/2)^2-1=0$$
$$ax_2-e^{x_1}-x_1=0$$
$$a=3$$

Теоретические сведения

Метод простых итераций:

Решение системы нелинейных уравнений вида

$$\begin{cases} f_1(x_1,x_2,x_3,\ldots,x_n)=0 \\ f_2(x_1,x_2,x_3,\ldots,x_n)=0 \\ \ldots \\ f_n(x_1,x_2,x_3,\ldots,x_n)=0 \end{cases}$$

возможно, если все функции в системе непрерывны и дифференцируемы в окрестности решения.

Для использования метода итераций система уравнений записывается в эквивалентной форме:

$$\begin{cases} x_1=\phi_1(x_1,x_2,x_3,\ldots,x_n) \\ x_2=\phi_2(x_1,x_2,x_3,\ldots,x_n) \\ \ldots \\ x_n=\phi_n(x_1,x_2,x_3,\ldots,x_n) \end{cases},$$

где ϕ_i – итерирующие непрерывно дифференцируемые функции.

Тогда, если известно начальное приближение $X^{(0)}$, то можно построить алгоритм метода простых итераций:

$$\begin{cases} x_1^{(k+1)} = \phi_1\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots, x_n^{(k)}\right) \\ x_2^{(k+1)} = \phi_2\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots, x_n^{(k)}\right) \\ \ldots \\ x_n^{(k+1)} = \phi_n\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots, x_n^{(k)}\right) \end{cases}$$

Предложение формулировки достаточного условия сходимости для многомерного случая выглядит следующим образом. Метод простых итераций сходится к решению системы нелинейных уравнений, если какая-либо норма матрицы Якоби $J\left(X^{(k)}\right)$, построенная по правым частям ϕ_i эквивалентной системы в замкнутой области G, меньше единицы на каждой итерации:

$$J\left(X^{(k)}\right) = \left[\frac{\partial \phi_i\left(X^{(k)}\right)}{\partial x_j}\right]; \left\|J\left(X^{(k)}\right)\right\| \le q < 1$$

Для практических расчетов чаще всего используют матричную норму, определенную в области решения G, которую обязательно проверяют в начальном приближении:

$$\max_{x \epsilon G} \left\|\phi'(x)\right\| = \max_{x \epsilon G} \left\{ \max_i \sum_{j=1}^{n} \left| \frac{\partial \phi_i(X)}{\partial x_j} \right| \right\}$$

В практических вычислениях в качестве условия окончания итераций обычно используется критерий

$$\left| X^{(k+1)} - X^{(k)} \right| < \varepsilon$$

Метод Ньютона:

Пусть дана система нелинейных уравнений. Все функции системы непрерывны на некотором интервале $[a, b]$ и дифференцируемы вплоть до вторых производных.

Итерационный процесс нахождения решения, который носит название метода Ньютона для систем, записывается в виде решения матричного уравнения:

$$X^{(k+1)} = X^{(k)} - J^{-1}\left(X^{(k)}\right) \cdot f\left(X^{(k)}\right),$$

где $J\left(X^{(k)}\right)$ – матрица Якоби.

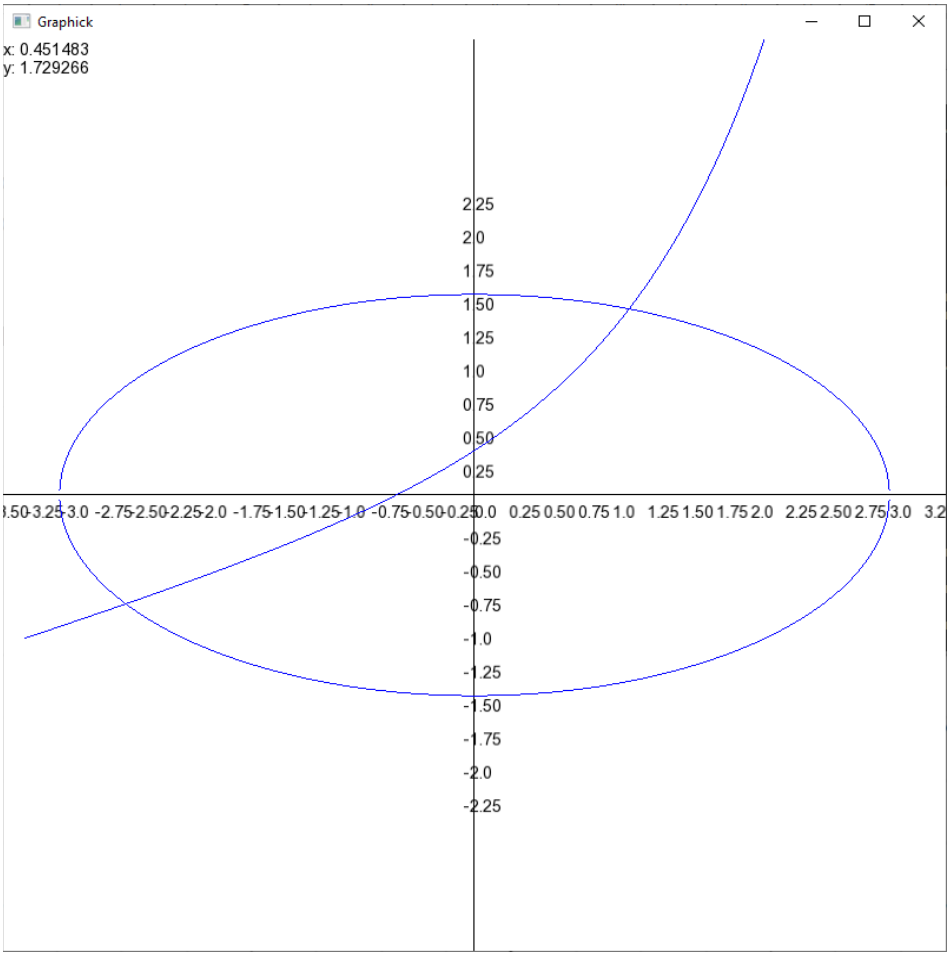
$$J\left(X^{(k)}\right)=\begin{vmatrix} \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_1} & \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_2} & \cdots & \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_n} \\ \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_1} & \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_2} & \cdots & \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n\left(X^{(k)}\right)}{\partial x_1} & \frac{\partial f_n\left(X^{(k)}\right)}{\partial x_2} & \cdots & \frac{\partial f_n\left(X^{(k)}\right)}{\partial x_n} \end{vmatrix}$$

В практических вычислениях в качестве условия окончания итераций обычно используют критерий, выполняющийся для всех переменных системы:

$$\left|X^{(k+1)}-X^{(k)}\right|<\varepsilon$$

Подготовка системы к решению методами Ньютона и простой итерации

1. Графически определим примерное расположение корня системы:



В качестве начального приближения возьмем область

$x_1\in[1;1.25], x_2\in[1.25,1.5]$.

2. Вычислим матрицу Якоби для данной системы уравнений:

$$J(X)=\begin{vmatrix} \frac{2x_1}{a^2} & \frac{8x_2}{a^2} \\ -e^{x_1}-1 & a \end{vmatrix}$$

3. Подберем подходящие итерирующие функции ϕ_i и найдем матрицу Якоби для системы, записанной в эквивалентной форме:

$$\begin{cases} x_1=\frac{a\,x_2-\exp^{x_1}+4\,x_2}{5} \\ x_2=\frac{\sqrt{a^2-x_1^2}}{2} \end{cases}$$

$$J(\phi)=\begin{vmatrix} \frac{4-e^{x_1}}{5} & \frac{a}{5} \\ -\frac{x_1}{2\sqrt{\left(a^2-x_1^2\right)}} & 0 \end{vmatrix}$$

Код:

tools.h

```
#include "../../Graphica.h"
#include "../../Matrix.h"
#include <iostream>

using MF = Matrix<double (*)(Matrix<double>)>;

template <typename T>
double trashDetM(Matrix<T> m) {
    if (m.dim.first == 1) return m[0][0];
    if (m.dim.first == 2) return m[0][0] * m[1][1] - m[0][1] * m[1][0];
    double res = 1;
    return res;
}

Matrix<double> subsM(MF m, Matrix<double> x) {
    Matrix<double> res(m.dim);
    for (size_t i = 0; i < m.dim.first; i++)
        for (size_t j = 0; j < m.dim.second; j++)
            res[i][j] = m[i][j](x);
    return res;
}

Matrix<double> Newton(MF foo, MF J, Matrix<double> G, double epsilon) {
    Matrix<double> x(G.dim.first, 1);
    for (size_t i = 0; i < x.dim.first; i++)
        x[i][0] = (G[i][1] + G[i][0]) / 2;

    Matrix<double> was;
    size_t counter = 0;
    do {
        counter++;
        was = x;
        double detJ = trashDetM(subsM(J, x));
        for (size_t i = 0; i < x.dim.first; i++) {
            MF Ax(J);
            for (size_t j = 0; j < x.dim.first; j++) Ax[j][i] = foo[j][0];
            x[i][0] = x[i][0] - trashDetM(subsM(Ax, was)) / detJ;
        }
    } while ((x - was).normC() > epsilon);
    cout << "counter = " << counter << '\n';
    return x;
}

Matrix<double> Iter(MF foo, MF foo1, Matrix<double> G, double epsilon) {
    size_t N = foo.dim.first;
    double q = 0;
    Matrix<double> was, x(N, 1);
    for (size_t i = 0; i < N; i++) x[i][0] = G[i][0];
```

```
    while (x[N - 1][0] < G[N - 1][1] > 0) {
        for (size_t i = 0; i < N; i++) {
            double temp = 0;
            for (size_t j = 0; j < N; j++) temp += abs(foo1[i][j](x));
            q = max(q, temp);
        }
        x[0][0] += (G[0][1] - G[0][0]) / 100;
        for (size_t i = 0; i < N - 1; i++)
            if (x[i][0] > G[i][1]) {
                x[i][0] = G[i][0];
                x[i + 1][0] += (G[i + 1][1] - G[i + 1][0]) / 100;
            }
    }
    cout << "q = " << q << '\n';
    for (size_t i = 0; i < N; i++) x[i][0] = (G[i][1] + G[i][0]) / 2;
    size_t counter = 0;
    do {
        counter++;
        was = x;
        x = subsM(foo, x);
    } while ((x - was).normC() > epsilon);
    cout << "counter = " << counter << '\n';
    return x;
}
```

main.cpp

```
#include "tools.h"

double a        = 3;
double epsilon  = 0.0001;

int main() {
    makeGraph([](double x){ return 0.5 * sqrt(a * a - x * x); });
    makeGraph([](double x){ return -0.5 * sqrt(a * a - x * x); });
    makeGraph([](double x){ return (exp(x) + x) / a; });
    sf::Thread thread(ShowGraphics); thread.launch();

    Matrix<double> res, G(2, 2);
    cout << "l1 = "; cin >> G[0][0]; cout << "r1 = "; cin >> G[0][1];
    cout << "l2 = "; cin >> G[1][0]; cout << "r2 = "; cin >> G[1][1];

    cout << "Choose method: 1 - Newton, 2 - Iter\n> ";
    int choose; cin >> choose;
    MF foo(2, 1), J(2, 2);
    if (choose == 1) {
        foo[0][0] = [](Matrix<double> x) { return pow(x[0][0] / a, 2) + 4 * pow(x[1][0] / a, 2) -
1; };
        foo[1][0] = [](Matrix<double> x) { return a * x[1][0] - exp(x[0][0]) - x[0][0]; };

        J[0][0] = [](Matrix<double> x) { return 2 * x[0][0] / (a * a); };
        J[0][1] = [](Matrix<double> x) { return 8 * x[1][0] / (a * a); };
        J[1][0] = [](Matrix<double> x) { return - exp(x[0][0]) - 1; };
        J[1][1] = [](Matrix<double> x) { return a; };

        res = Newton(foo, J, G, epsilon);
        cout << res[0][0] << ' ' << res[1][0] << '\n';
    } else {
        foo[0][0] = [](Matrix<double> x) { return (a * x[1][0] - exp(x[0][0]) + 4 * x[0][0]) / 5;
};
        foo[1][0] = [](Matrix<double> x) { return sqrt(a * a - x[0][0] * x[0][0]) / 2; };

        J[0][0] = [](Matrix<double> x) { return (4 - exp(x[0][0])) / 5; };
        J[0][1] = [](Matrix<double> x) { return a / 5; };
        J[1][0] = [](Matrix<double> x) { return -0.5 * x[0][0] / sqrt(a * a - x[0][0] * x[0][0]);
};
        J[1][1] = [](Matrix<double> x) { return (double)0; };

        res = Iter(foo, J, G, epsilon);
        cout << res[0][0] << ' ' << res[1][0] << '\n';
    }

    thread.wait(); thread.terminate();
    return 0;
}
```

Вывод:

Первый запуск

```
l1 = 1
r1 = 1.25
```

```
l2 = 1.25
r2 = 1.5
Choose method: 1 - Newton, 2 - Iter
> 1
counter = 2
1.11781 1.39199
```

Второй запуск

```
l1 = 1
r1 = 1.25
l2 = 1.25
r2 = 1.5
Choose method: 1 - Newton, 2 - Iter
> 2
q = 0.856344
counter = 6
1.11776 1.39198
```

Лабораторная работа 3

Задание 1

Используя таблицу значений Y_i функции $y=f(x)$, вычисленных в точках $X_i, i=0,\ldots,3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значение погрешности интерполяции в точке X^*.

15. $y = ctg(x) + x$,а) $X_i = \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8}, \frac{4\pi}{8}$; б) $X_i = \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}, \frac{\pi}{2}$; $X^* = \frac{3\pi}{16}$.

Теоретические сведения

Пусть на отрезке $[a,b]$ задано множество несовпадающих точек x_i (интерполяционных узлов), в которых известны значения функции $f_i=f(x_i)$. Приближающая функция $\varphi(x,a)$ такая, что выполняются равенства

$$\varphi(x_i, a_0, \ldots, a_n)=f(x_i)$$

называется интерполяционной.

Наиболее часто в качестве приближающей функции используют многочлены степени n:

$$P_n(x)=\sum_{i=0}^{n} a_i x^i$$

Произвольный многочлен может быть записан в виде

$$L_n(x)=\sum_{i=0}^{n} f_i l_i(x)$$

Здесь $l_i(x)$ – многочлены степени n, так называемые лагранжевы многочлены влияния, которые удовлетворяют условию $l_i(x_j)=\begin{cases}1, npui=j\\0, npui\neq j\end{cases}$ и, соответственно,

$$l_i(x)=\prod_{j=0, j\neq i}^{n} \frac{(x-x_i)}{(x_i-x_j)},$$

а интерполяционный многочлен запишется в виде

$$L_n(x)=\sum_{i=0}^{n} f_i \prod_{j=0, j\neq i}^{n} \frac{(x-x_i)}{(x_i-x_j)}$$

Интерполяционный многочлен, записанный в этой форме, называется **интерполяционным многочленом Лагранжа**.

Недостатком интерполяционного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют интерполяционный многочлен в форме Ньютона.

Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются $f(x_i, x_j)$ и определяются через разделенные разности нулевого порядка:

$$f(x_i, x_j) = \frac{f_i - f_j}{x_i - x_j},$$

разделенные разности второго порядка определяются через разделенные разности первого порядка:

$$f(x_i, x_j, x_k) = \frac{f(x_i, x_j) - f(x_j, x_j)}{x_i - x_k}$$

Разделенная разность $n - k + 2$ определяется соостношениями

$$f(x_i, x_j, x_k, \ldots, x_{n-1}, x_n) = \frac{f(x_i, x_j, x_k, \ldots, x_{n-1}) - f(x_j, x_k, \ldots, x_n)}{x_i - x_n}$$

Интерполяционный многочлен Ньютона может быть представлен в виде:

$$P_n(x) = f(x_0) + (x - x_0) f(x_1, x_0) + (x - x_0)(x - x_1) f(x_0, x_1, x_2) + \ldots + (x - x_0)(x - x_1)\ldots(x - x_n) f(x_0, x_1, \ldots, x_n)$$

Отметим, что при добавлении новых узлов первые члены многочлена Ньютона остаются неизменными.

Для повышения точности интерполяции в сумму могут быть добавлены новые члены, что требует подключения дополнительных интерполяционных узлов. При этом безразлично, в каком порядке подключаются новые узлы. Этим формула Ньютона выгодно отличается от формулы Лагранжа.

Код:

tools.h

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

size_t LDim;
vector<double> Lcoef, LXcopy;
size_t Lleft, Lright;

double (*Lagrange(double (*foo)(double), vector<double> X, double x, int dim))(double) {
    dim++;
    LDim = dim;
    Lcoef.resize(LDim);
```

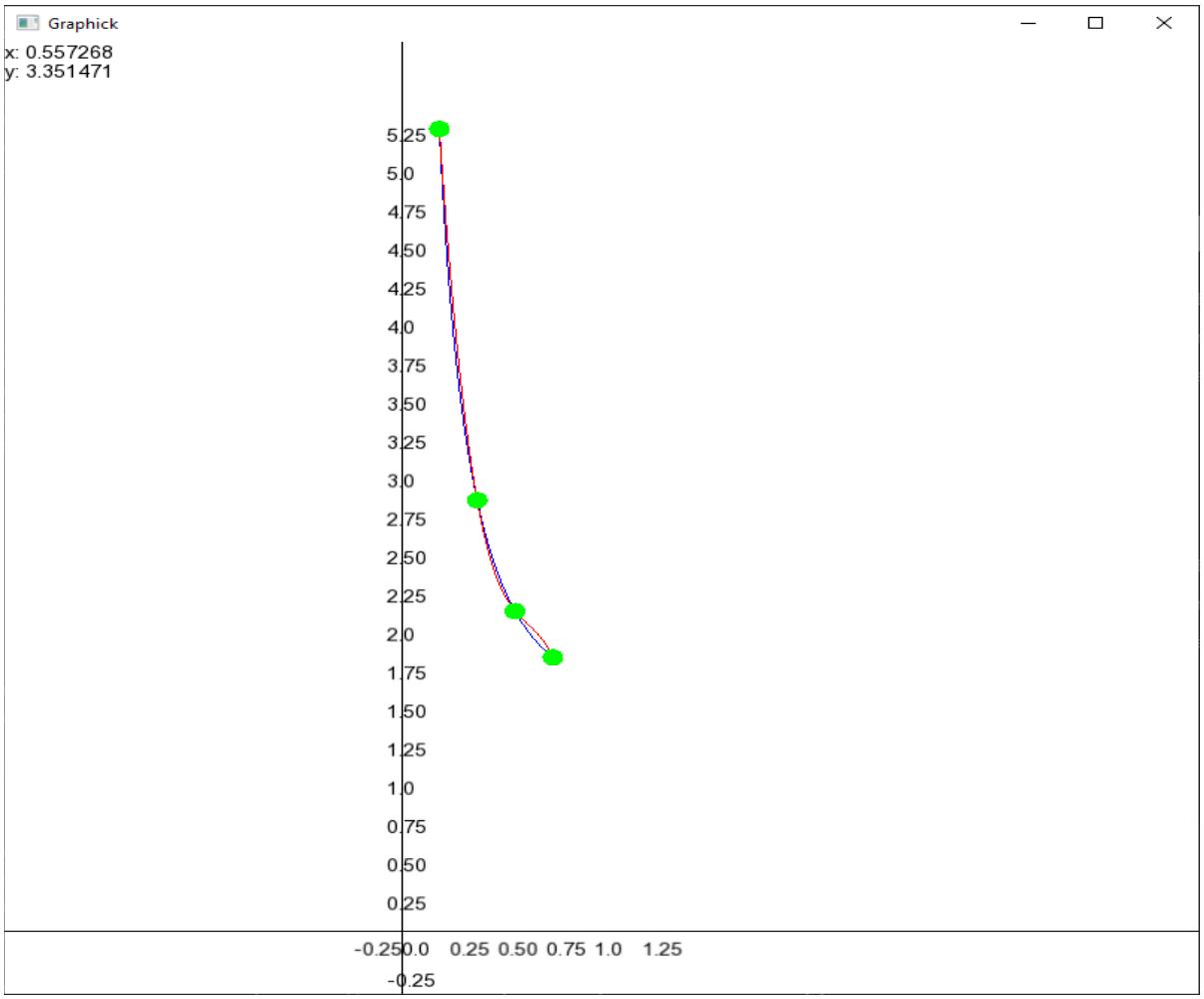
```
    LXcopy = X;
    Lleft = 0; Lright = X.size();
    for (int i = 0; i < X.size() - dim; i++)
        if ((foo(X[Lleft]) + foo(X[Lright - 1])) / 2 > x) Lright--;
        else                                              Lleft++;
    for (size_t i = Lleft; i < Lright; i++) {
        double w = 1;
        for (size_t j = Lleft; j < Lright; j++)
            if (i != j)
                w *= X[i] - X[j];
        Lcoef[i] = foo(X[i]) / w;
    }
    return [](double x) {
        double ans = 0;
        for (size_t i = Lleft; i < Lright; i++) {
            double temp = Lcoef[i];
            for (size_t j = Lleft; j < Lright; j++)
                if (i != j)
                    temp *= x - LXcopy[j];
            ans += temp;
        }
        return ans;
    };
}

size_t NDim;
vector<double> Ncoef, NXcopy;
size_t Nleft, Nright;

double f(double (*foo)(double), vector<double> x, size_t left, size_t right) {
    if (right - left == 0) return foo(x[left]);
    return (f(foo, x, left, right - 1) - f(foo, x, left + 1, right)) / (x[left] - x[right]);
}

double (*Newton(double (*foo)(double), vector<double> X, double x, int dim))(double) {
    dim++;
    NDim = dim;
    Ncoef.resize(NDim);
    NXcopy = X;
    for (size_t i = 0; i < NDim; i++) Ncoef[i] = f(foo, X, 0, i);
    Nleft = 0; Nright = X.size();
    for (int i = 0; i < X.size() - dim; i++)
        if ((foo(X[Nleft]) + foo(X[Nright - 1])) / 2 > x) Nright--;
        else                                              Nleft++;
    return [](double x) {
        double ans = 0;
        for (size_t i = Nleft; i < Nright; i++) {
            double temp = Ncoef[i];
            for (size_t j = Nleft; j < i; j++)
                temp *= x - NXcopy[j];
            ans += temp;
        }
        return ans;
    };
}
```

main.cpp

```
#include "tools.h"
#include "../../Graphica.h"

double foo(double x) { return tan(M_PI_2 - x) + x; }

int main() {
    vector<double> X1 = {M_PI_2 / 8, M_PI_2 / 4, M_PI_2 * 3 / 8, M_PI_2 / 2},
                   X2 = {M_PI_2 / 8, M_PI_2 / 4, M_PI_2 * 3 / 8, M_PI_2 / 2};
    SetDiapazon(X1[0], X1[3]);
    makeGraph(foo);
    for (size_t i = 0; i < X1.size(); i++) addPoint(X1[i], foo(X1[i]));
    double x = M_PI_2 * 3 / 16;

    int n; cout << "n for Lagrange = "; cin >> n;
    double (*ans1)(double) = Lagrange(foo, X1, x, n);
    double val1 = ans1(x), real1 = foo(x);
    cout << "val1 = " << val1 << "\nAbsolute interpolation error is: " << abs(val1 - real1) << '\
n';
    makeGraph(ans1, sf::Color::Green);

    cout << "n for Newton = "; cin >> n;
    double (*ans2)(double) = Newton(foo, X2, x, n);
```

```
    double val2 = ans2(x), real2 = foo(x);
    cout << "val2 = " << val2 << "\nAbsolute interpolation error is: " << abs(val2 - real2) << '\
n';
    makeGraph(ans2, sf::Color::Red);

    cout << "Green - Lagrange\nRed - Newton\n";
    ShowGraphics();
    return 0;
}
```

Вывод:




```
n for Lagrange = 3
val1 = 3.7237
Absolute interpolation error is: 0.132621
n for Newton = 3
val2 = 3.7237
Absolute interpolation error is: 0.132621
Green - Lagrange
Red - Newton
```

Задание 2

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x=x_0$ и $x=x_4$. Вычислить значение функции в точке $x=X^*$.

15. $X^* = 2.66666667$

i	0	1	2	3	4
x_i	1.0	1.9	2.8	3.7	4.6
f_i	2.8069	1.8279	1.6091	1.5713	1.5663

Теоретические сведения

Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении сплайн-интерполяции. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени:

$$S(x)=\sum_{k=0}^{n} a_{ik}x^k, x_{i-1}\le x\le x_i, i=\overline{1,n}$$

который в узлах интерполяции принимает значения аппроксимируемой функции и непрерывен вместе со своими $(n-1)$ производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства $(n-1)$ производных соответствующих многочленов. На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как

$$S(x)=a_i+b_i(x-x_{i-1})+c_i(x-x_{i-1})^2+d_i(x-x_{i-1})^3,$$

$$x_{i-1}\le x\le x_i, i=\overline{1,n}$$

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить 4n неизвестных a_i, b_i, c_i, d_i. Эти коэффициенты ищутся из условий в узлах сетки.

Код:

tools.h

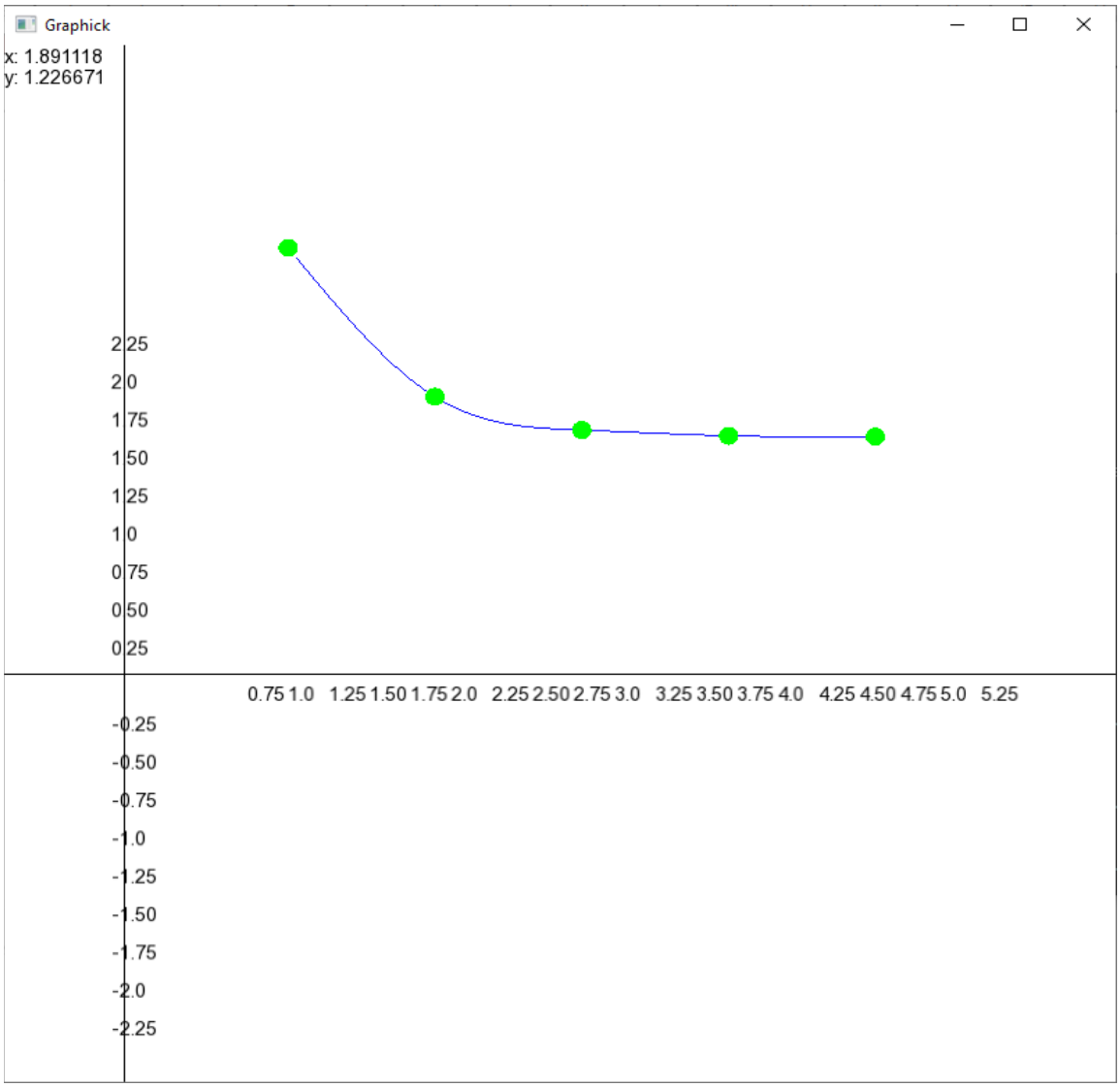
```
#include <iostream>
#include <vector>
#include <fstream>
#include "../../lab1/TMA/TMA.hpp"
size_t Dim;
vector<double> Xcopy;
Matrix<double> coef;

double (*Spline(vector<double> foo, vector<double> X))(double) {
    Dim = X.size() - 1;
    coef.resize(Dim, Dim); coef[2][0] = 0;
    Xcopy = X;
    vector<double> h(Dim + 1, 0);
    for (size_t i = 1; i <= Dim; i++) h[i] = X[i] - X[i - 1];
    Matrix<double> m(3, Dim - 1), d(1, Dim - 1);
    for (size_t i = 2; i <= Dim; i++) {
        m[0][i - 2] = (i == 2) ? 0 : h[i - 1];
        m[1][i - 2] = 2 * (h[i] + h[i - 1]);
        m[2][i - 2] = (i == Dim) ? 0 : h[i];

        d[0][i - 2] = 3 * ((foo[i] - foo[i - 1]) / h[i] - (foo[i - 1] - foo[i - 2]) / h[i - 1]);
    }
    Matrix<double> c = TMAsolve(m, d);
    for (size_t i = 0; i < Dim; i++) {
        coef[0][i] = foo[i];
        if (i < Dim - 1) {
            coef[2][i + 1] = c[0][i];
            coef[1][i] = (foo[i + 1] - foo[i]) / h[i + 1] - h[i + 1] * (coef[2][i + 1] + 2 *
coef[2][i]) / 3;
            coef[3][i] = (coef[2][i + 1] - coef[2][i]) / (3 * h[i + 1]);
        } else {
            coef[1][i] = (foo[i + 1] - foo[i]) / h[i + 1] - 2 * h[i + 1] * coef[2][i] / 3;
            coef[3][i] = -coef[2][i] / (3 * h[i + 1]);
        }
    }
    return [](double x) {
        if (x < Xcopy[0] || Xcopy[Dim] < x) {
            cerr << "bruh\n"; exit(1);
        }
        size_t i;
        for (i = 0; Xcopy[i] < x; i++) {}
        i--;
        return coef[0][i] + coef[1][i] * pow((x - Xcopy[i]), 1) +
               coef[2][i] * pow((x - Xcopy[i]), 2) + coef[3][i] * pow((x - Xcopy[i]), 3);
    };
}
```

main.cpp

```
#include "tools.h"
#include "../../Graphica.h"

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    ifstream input("input.txt");
    size_t n; input >> n;
    vector<double> X(n), foo(n);
    for (size_t i = 0; i < n; i ++) input >> X[i];
    for (size_t i = 0; i < n; i ++) input >> foo[i];
    for (size_t i = 0; i < n; i ++) addPoint(X[i], foo[i]);
    double x; input >> x;
    double (*res)(double) = Spline(foo, X);
    cout << res(x) << '\n';

    SetDiapazon(1.05, 4.55);
    makeGraph(res);
    ShowGraphics();
    return 0;
}
```

Вывод:

```
1.61495
```



Задание 3

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

15.

i	0	1	2	3	4	5
x_i	1.0	1.9	2.8	3.7	4.6	5.5
y_i	3.4142	2.9818	3.3095	3.8184	4.3599	4.8318

Теоретические сведения

Пусть задана таблично в узлах x_j функция $y_i=f(x_i)$. При этом значения функции y_i определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени n, у которого неизвестны коэффициенты a_i, $F_n(x)=\sum_{i=0}^{n} a_i x^i$. Неизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции.

$$\Phi=\sum_{j=0}^{n}\left[F_n(x_j)-y_i\right]^2$$

Минимума Φ можно добиться только за счет изменения коэффициентов многочлена $F_n(x)$. Необходимые условия экстремума имеют вид

$$\frac{\partial \Phi}{\partial a_k}=2\sum_{j=0}^{N}\left[\sum_{j=0}^{n} a_i x_j^i-y_j\right]x_j^k=0, k=\overline{0,n}$$

Эту систему для удобства преобразуют к следующему виду:

$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+1}=\sum_{j=0}^{N} y_j x_j^k, k=\overline{0,n}$$

Эта система называется нормальной системой метода наименьших квадратов (МНК) и представляет собой систему линейных алгебраических уравнений относительно коэффициентов a_i. Решив систему, построим многочлен $F_n(x)$, приближающий функцию $f(x)$ и минимизирующий квадратичное отклонение.

Код:

tools.h

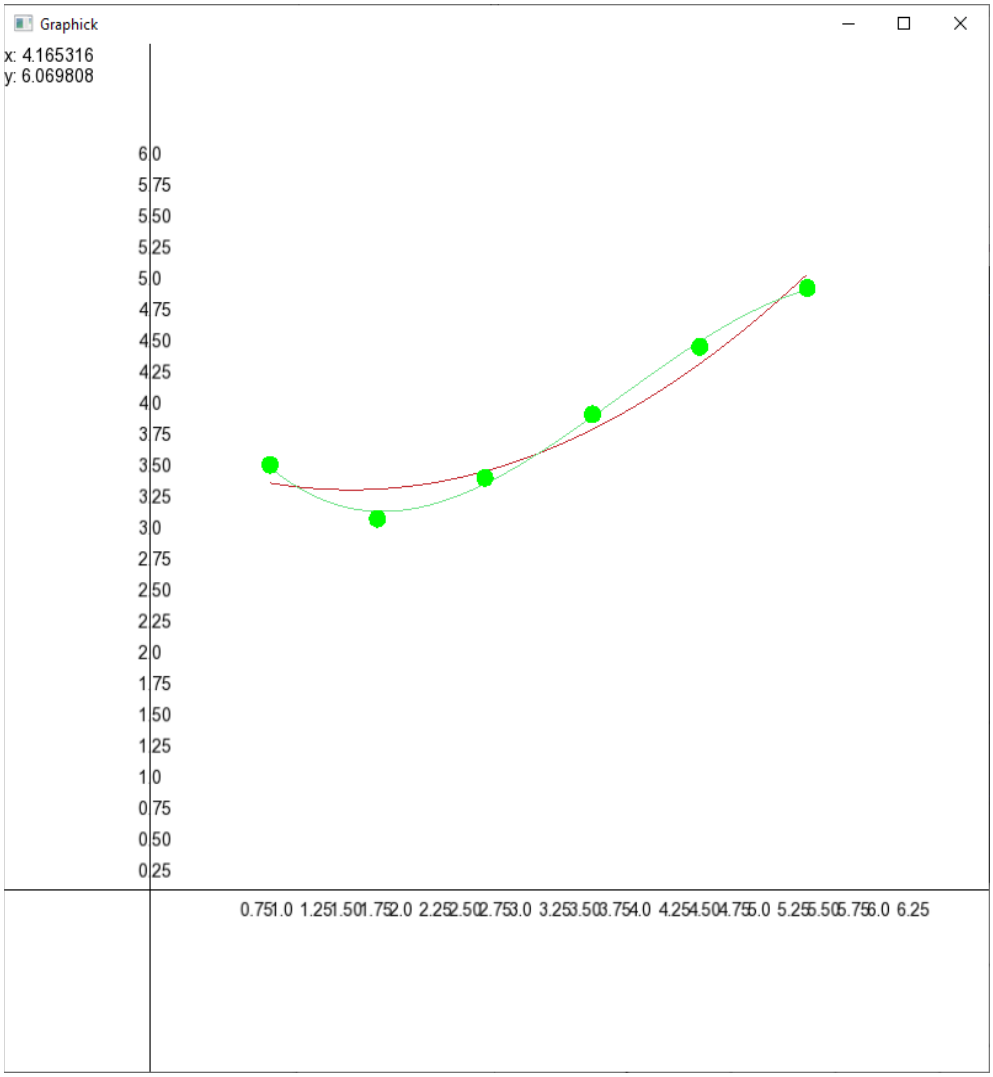
```
#include "../../lab1/LU/LU.h"
#include "../../lab1/Iter/Iter.h"
size_t Dim;
vector<double> coef;

double (*Approximation(vector<double> foo, vector<double> X, size_t dim))(double) {
    Dim = dim + 1;
    coef.resize(Dim);
    Matrix<double> m(Dim, Dim), d(1, Dim);
    for (size_t i = 0; i < Dim; i++) {
        for (size_t j = 0; j < Dim; j++)
            if (i > j) m[j][i] = m[i][j];
            else
                for (size_t k = 0; k < X.size(); k++)
                    m[j][i] += pow(X[k], i + j);
        for (size_t j = 0; j < X.size(); j++)
            d[0][i] += foo[j] * pow(X[j], i);
    }
    Matrix<double> resM = LUsolve(m, d);
    for (size_t i = 0; i < Dim; i++)
        coef[i] = resM[0][i];
    return [](double x) {
        double res = coef[0];
        for (size_t i = 1; i < Dim; i++)
            res += coef[i] * pow(x, i);
        return res;
    };
}
```

main.cpp

```
#include "tools.h"
#include "../../Graphica.h"

int main() {
    ifstream input("input.txt");
    size_t n; input >> n;
    vector<double> X(n), foo(n);
    for (size_t i = 0; i < n; i++) input >> X[i];
    for (size_t i = 0; i < n; i++) input >> foo[i];
    for (size_t i = 0; i < n; i++) addPoint(X[i], foo[i]);
    SetDiapazon(X[0], X[n - 1]);
    MaxY = 6; MinY = 0;

    sf::Thread thread(ShowGraphics); thread.launch();
    sf::Mutex mutex;
    int dim;
    while (true) {
        cout << "input dim: "; cin >> dim;
        if (dim == 0) break;
        double (*res)(double) = Approximation(foo, X, dim);
        double Sum_of_Squared_Errors = 0;
        for (size_t i = 0; i < n; i++) Sum_of_Squared_Errors += pow(foo[i] - res(X[i]), 2);
        cout << "Sum of Squared Errors = " << Sum_of_Squared_Errors << '\n';
        mutex.lock();
        makeGraph(res, sf::Color(rand() % 256, rand() % 256, rand() % 256));
        mutex.unlock();
    }
    thread.wait();
    return 0;
```

}

Вывод:




```
input dim: 2
Sum of Squared Errors = 0.125965
input dim: 3
Sum of Squared Errors = 0.00870066
input dim: 0
```

Задание 4

Вычислить первую и вторую производную от таблично заданной функции $y_i=f(x_i), i=0,1,2,3,4$ в точке $x=X^*$.

15. $X^* = 0.4$

i	0	1	2	3	4
x_i	0.0	0.2	0.4	0.6	0.8
y_i	1.0	1.4214	1.8918	2.4221	3.0255

Теоретические сведения

Формулы численного дифференцирования в основном используются при нахождении производных от функции $y=f(x)$, заданной таблично. Исходная функция $y_i=f(x_i)$ на отрезках$[x_j, x_{j+k}]$, заменяется некоторой приближающей, легко вычисляемой функцией $\varphi(x,\bar{a}), y=\varphi(x,\bar{a})+R(x)$, где $R(x)$ – остаточный член приближения, $\bar{a}$ – набор коэффициентов, вообще говоря, различный для каждого из рассматриваемых отрезков, и полагают, что $y'(x)\approx\varphi'(x,\bar{a})$. Наиболее часто в качестве приближающей функции $\varphi(x,\bar{a})$ берется интерполяционный многочлен $\varphi(x,\bar{a})=P_n(x)$, а производные соответствующих порядков определяются дифференцированием многочлена.

При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой. В этом случае:

$$y'\approx\frac{y_{i+1}-y_i}{x_{i+1}-x_i}=const$$

При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:

$$y'(x)\approx\frac{y_{i+1}-y_i}{x_{i+1}-x_i}+\frac{\frac{y_{i+2}-y_{i+1}}{x_{i+2}-x_{i+1}}-\frac{y_{i+1}-y_i}{x_{i+1}-x_i}}{x_{i+2}-x_i}\left(2x-x_i-x_{i+1}\right)$$

При равностоящих точках разбиения, данная формула обеспечивает второй порядок точности.

Для вычисления второй производной, необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем

$$y''(x) \approx 2\frac{\frac{y_{i+2}-y_{i+1}}{x_{i+2}-x_{i+1}} - \frac{y_{i+1}-y_i}{x_{i+1}-x_i}}{x_{i+2}-x_i}$$

Код:

tools.h

```
#include <iostream>
#include <vector>
#include <fstream>
using namespace std;
#define diff(n) (foo[n + 1] - foo[n]) / (X[n + 1] - X[n])

double derivative(vector<double> foo, vector<double> X, double x, size_t p) {
    size_t Dim = X.size();
    if (x < X[0] || X[Dim - 1] < x) {
        cerr << "bruh\n"; exit(1);
    }
    size_t i;
    for (i = 0; X[i] < x; i++) {}
    i--;
    if (p == 1)
        return diff(i) + (2 * x - X[i] - X[i + 1]) * (diff(i + 1) - diff(i)) / (X[i + 2] - X[i]);
    else
        return 2 * (diff(i + 1) - diff(i)) / (X[i + 2] - X[i]);
}
```

main.cpp

```
#include "tools.h"

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    ifstream input("input.txt");
    size_t n; input >> n;
    vector<double> X(n), foo(n);
    for (size_t i = 0; i < n; i ++) input >> X[i];
    for (size_t i = 0; i < n; i ++) input >> foo[i];
    double res = derivative(foo, X, 0.4, 1);
    cout << "res1 = " << res << '\n';
    res = derivative(foo, X, 0.4, 2);
    cout << "res2 = " << res << '\n';
    return 0;
}
```

Вывод:

```
res1 = 2.50175
res2 = 1.4975
```

Задание 5

Вычислить определенный интеграл $F=\int_{X_0}^{X_1} ydx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

15. $y=\dfrac{x}{x^4+81}$, $X_0=0, \quad X_k=2, \quad h_1=0.5, \quad h_2=0.25$;

Теоретические сведения

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл $F=\int_a^b f(x)dx$ не удается. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен.

При использовании интерполяционных многочленов различной степени, получают формулы численного интегрирования различного порядка точности.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку $\overline{x}_i=\dfrac{x_{i-1}+x_i}{2}$, получим **формулу прямоугольников**:

$$F=\int_a^b f(x)dx\approx\sum_{i=1}^{N} h_i f\left(\frac{x_{i-1}+x_i}{2}\right)$$

В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию $f(x)$ многочленом Лагранжа первой степени.

$$F=\int_a^b f(x)dx\approx\frac{1}{2}\sum_{i=1}^{N}(f_i+f_{i-1})h_i$$

Эта формула носит название **формулы трапеций**.

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой –

интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования: $x_{i-1}, x_{i-\frac{1}{2}} = \frac{x_{i-1}+x_i}{2}, x_i$.

Для случая $h_i = \frac{x_i - x_{i-1}}{2}$, получим **формулу Симпсона**:

$$F = \int_a^b f(x)\,dx \approx \frac{1}{3}\sum_{i=1}^{N}\left(f_{i-1} + 4f_{i-\frac{1}{2}} + f_i\right)h_i$$

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла порядка точности p на сетке с шагом $h: F_h$ и на сетке с шагом $kh: F_{kh}$, то

$$F = \int_a^b f(x)\,dx = F_h + \frac{F_h - F_{kh}}{k^p - 1} + O\left(h^{p+1}\right)$$

Код:

tools.h

```
#include <iostream>
#include <cmath>
using namespace std;

double rectangle(double (*foo)(double), double x0, double xk, double h) {
    size_t Dim = (xk - x0) / h;
    double res = 0;
    for (size_t i = 0; i <= Dim; i++)
        res += foo(x0 + h * double(i * 2 + 1) / 2);
    return res * h;
}

double trapezoid(double (*foo)(double), double x0, double xk, double h) {
    size_t Dim = (xk - x0) / h;
    double res = (foo(x0) + foo(xk)) / 2;
    for (size_t i = 1; i < Dim; i++)
        res += foo(x0 + h * i);
    return res * h;
}

double Simpson(double (*foo)(double), double x0, double xk, double h) {
    if ((xk - x0) / (2 * h) != int((xk - x0) / (2 * h))) {
        cerr << "wrong interval or legth of step\n"; exit(1);
    }
    size_t Dim = (xk - x0) / h;
    double res = foo(x0) - foo(xk);
    size_t i;
    for (i = 1; i <= Dim; i++)
        res += 4 * foo(x0 + h * (double(i) - 0.5)) + 2 * foo(x0 + h * i);
    return res * h / 6;
}
```

main.cpp

```
#include "tools.h"

double foo(double x) { return x / (pow(x, 4) + 81); }

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);

    double X0 = 0, Xk = 2, h1 = 0.5, h2 = 0.25, ans = 0.0232346849766;
    double rect1 = rectangle(foo, X0, Xk, h1),  rect2 = rectangle(foo, X0, Xk, h2),
           trap1 = trapezoid(foo, X0, Xk, h1),  trap2 = trapezoid(foo, X0, Xk, h2),
           simp1 = Simpson(foo, X0, Xk, h1),    simp2 = Simpson(foo, X0, Xk, h2);
    cout << "rect1    = " << rect1 << '\n'
```

```
        << "error1   = " << abs(ans - rect1) << '\n' << '\n'
        << "rect2    = " << rect2 << '\n'
        << "error2   = " << abs(ans - rect2) << '\n' << '\n'
        << "trap1    = " << trap1 << '\n'
        << "error1   = " << abs(ans - trap1) << '\n' << '\n'
        << "trap2    = " << trap2 << '\n'
        << "error2   = " << abs(ans - trap2) << '\n' << '\n'
        << "Simpson1 = " << simp1   << '\n'
        << "error1   = " << abs(ans - simp1) << '\n' << '\n'
        << "Simpson2 = " << simp2   << '\n'
        << "error2   = " << abs(ans - simp2) << '\n';

    return 0;
}
```

Вывод:

```
rect1    = 0.0338771
error1   = 0.0106424

rect2    = 0.0284973
error2   = 0.00526263

trap1    = 0.0230508
error1   = 0.000183848

trap2    = 0.0231887
error2   = 4.60166e-05

Simpson1 = 0.0232346
error1   = 7.28504e-08

Simpson2 = 0.0232347
error2   = 4.20767e-09
```

Лабораторная работа 4

Задание 1

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге-Ромберга и путем сравнения с точным решением.

15	$xy''+y'=0$, $y(1)=1$, $y'(1)=1$, $x\in[1,2], h=0.1$	$y=1+\ln\|x\|$

Теоретические сведения

Рассматривается задача Коши для одного дифференциального уравнения первого порядка разрешенного относительно производной

$$y'=f(x,y)$$

$$y(x_0)=y_0$$

Требуется найти решение на отрезке $[a,b]$, где $x_0=a$.

Введем разностную сетку на отрезке $[a,b]$: $\Omega^{(k)}=\{x_k=x_0+hk\}$.

Точки x_i называются узлами разностной сетки, расстояния между узлами — *шагом разностной сетки*, а совокупность значений какой-либо величины заданных в узлах сетки называется *сеточной функцией*.

Приближенное решение задачи Коши будем искать численно в виде сеточной функции.

Метод Эйлера (явный):

Метод Эйлера играет важную роль в теории численных методов решения ОДУ, хотя и не часто используется в практических расчетах из-за невысокой точности. Вывод расчетных соотношений для этого метода может быть произведен несколькими способами: с помощью геометрической интерпретации, с использованием разложения в ряд Тейлора, конечно разностным методом (с помощью разностной аппроксимации производной),

квадратурным способом (использованием эквивалентного интегрального уравнения).

Рассмотрим вывод соотношений метода Эйлера геометрическим способом. Решение в узле x_0 известно из начальных условий. Рассмотрим процедуру получения решения в узле x_1.

График функции $y^{(h)}$, которая является решением задачи Коши, представляет собой гладкую кривую, проходящую через точку (x_0, y_0), согласно условию $y(x_0)=y_0$, и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точке x_0 и равен значению правой части дифференциального уравнения в точке (x_0, y_0) согласно выражению $y'(x_0)=f(x_0, y_0)$. В случае небольшого шага разностной сетки h график функции и график касательной не успевают сильно разойтись друг от друга и можно в качестве значения решения в узле x_1 принять значение касательной y_1, вместо значения неизвестного точного решения $y_{1ист}$. При этом допускается погрешность $|y_1 - y_{1ист}|$ геометрически представленная отрезком CD. Из прямоугольного треугольника ABC находим CB=BA tg(CAB) или $\Delta y = hy'(x_0)$. Учитывая, что $\Delta y = y_1 - y_0$ и заменяя производную $y'(x_0)$ на правую часть дифференциального уравнения, получаем соотношение $y_1 = y_0 + hf(x_0, y_0)$. Считая теперь точку (x_1, y_1) начальной и повторяя все предыдущие рассуждения, получим значение y_2 в узле x_2.

Переход к произвольным индексам дает формулу метода Эйлера:

$$y_{k+1} = y_k + hf(x_k, y_k)$$

Метод Рунге-Кутты:

Семейство явных методов Рунге-Кутты p-го порядка записывается в виде совокупности формул:

$$y_{k+1} = y_k + \Delta y_k$$

$$\Delta y_k = \sum_{i=1}^{p} c_i K_i^k$$

$$K_i^k = hf\left(x_k + a_i h, y_k + h\sum_{j=1}^{i-1} b_{ij} K_j^k\right)$$

$$i = \overline{2, p}$$

Метод Рунге-Кутты четвертого порядка:

$$a=\left(0,\frac{1}{2},\frac{1}{2},1\right); b=\begin{pmatrix}0 & 0 & 0 & 0\\ 0.5 & 0 & 0 & 0\\ 0 & 0.5 & 0 & 0\\ 0 & 0 & 0.5 & 0\end{pmatrix}; c=\left(\frac{1}{6},\frac{1}{3},\frac{1}{3},\frac{1}{6}\right)$$

$$\Delta y_k=\frac{1}{6}\left(K_1^k+K_2^k+K_3^k+K_4^k\right)$$

$$K_1^k=hf\left(x_k,y_k\right); K_2^k=hf\left(x_k+0.5h,y_k+0.5K_1^k\right)$$

$$K_3^k=hf\left(x_k+0.5h,y_k+0.5K_2^k\right); K_4^k=hf\left(x_k+h,y_k+K_3^k\right)$$

Контроль шага интегрирования:

На каждом шаге *h* рассчитывается параметр

$$\theta^k=\left|\frac{K_2^k-K_3^k}{K_1^k-K_2^k}\right|$$

Если величина θ^k порядка нескольких сотых единицы, то расчет продолжается с тем же шагом, если θ^k больше одной десятой, то шаг следует уменьшить, если же θ^k меньше одной сотой, то шаг можно увеличить.

Таким образом с помощью определения величины θ^k можно организовать

алгоритм выбора шага h для явного метода Рунге-Кутты.

Метод Адамса:

При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

$$y_{k+1}=y_k+\frac{h}{24}\left(55f_k-59f_{k-1}+37f_{k-2}-9f_{k-3}\right)$$

Метод Адамса как и все многошаговые методы не является самостартующим, то есть для того, что бы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле x_0 решение y_0 известно из начальных условий, а в других трех узлах x_1, x_2, x_3 решения y_1, y_2, y_3 можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

Решение задачи Коши для системы обыкновенных дифференциальных уравнений:

Рассматривается задача Коши для системы дифференциальных уравнений первого порядка разрешенных относительно производной

$$\begin{cases} y_1' = f_1(x, y, y_1, y_2, \dots, y_n) \\ y_2' = f_2(x, y, y_1, y_2, \dots, y_n) \\ \dots \\ y_n' = f_n(x, y, y_1, y_2, \dots, y_n) \end{cases}$$

$$\begin{gathered} y_1(x_0) = y_{01} \\ y_2(x_0) = y_{02} \\ \dots \\ y_n(x_0) = y_{0n} \end{gathered}$$

К системе дифференциальных уравнений можно применить все методы рассмотренные выше. Уравнения решаются по порядку.

Для задачи Коши 2-го порядка $y'' = f(x, y, y')$ можно применить следующее разложение в систему (используя замену $z = y'(x)$:

$$\begin{cases} z' = f(x, y, z) \\ y' = z(x) \end{cases}$$

$$\begin{gathered} y(x_0) = y_0 \\ y'(x_0) = z_0 \end{gathered}$$

Код

tools.h

```
#include <iostream>
#include <cmath>
#include <vector>
#include <stdio.h>
#include <iomanip>
#include "../../Matrix.h"
using namespace std;

Matrix<double> Butcher;

enum class Methods {
    Runge_Kutt_Method,
    ExplicitEulerMethod,
    MethodEuler_Cauchy,
    MethodAdams2,
    MethodAdams4
};

vector<vector<double>>  Runge_Kutt_Method3(vector<double  (*)(vector<double>)>,  vector<double>,
double, double, double, double (*)(double));
vector<vector<double>>  Runge_Kutt_Method4(vector<double  (*)(vector<double>)>,  vector<double>,
double, double, double, double (*)(double));
vector<vector<double>>  ExplicitEulerMethod(vector<double  (*)(vector<double>)>,  vector<double>,
double, double, double, double (*)(double));
vector<vector<double>>  MethodEuler_Cauchy(vector<double  (*)(vector<double>)>,  vector<double>,
double, double, double, double (*)(double));
vector<vector<double>>  MethodAdams2(vector<double  (*)(vector<double>)>,  vector<double>,  double,
double, double, double (*)(double));
```

```
vector<vector<double>> MethodAdams4(vector<double (*)(vector<double>)>, vector<double>, double,
double, double, double (*)(double));
vector<vector<double>> Method(vector<double (*)(vector<double>)>, vector<double>, double, double,
double, double (*)(double), Methods);

void print(size_t k, vector<double> x, vector<double> Delta, double y, double epsilon) {
    if (k == 0)
                                                                                                cout          <<
"-------------------------------------------------------------------------------------------------------
---------\n"
             << "|   k :      x      :      y      :      z      :      dy     :      dz     :
real    :   epsilon   |\n"
                 << "------+-------------+-------------+-------------+-------------+-------------
+-------------+-------------\n";
    cout << setprecision(4) << right << setfill(' ') << "| " << setw(3) << k << left;
    cout << setw(4) << " : " << setw(10) << x[0];
    cout << setw(4) << " : " << setw(10) << x[1];
    cout << setw(4) << " : " << setw(10) << x[2];
    cout << setw(4) << " : " << setw(10) << Delta[0];
    cout << setw(4) << " : " << setw(10) << Delta[1];
    cout << setw(4) << " : " << setw(10) << y;
    cout << setw(4) << " : " << setw(10) << epsilon << " |\n";
         cout  <<  "------+-------------+-------------+-------------+-------------+-------------
+-------------+-------------\n";
    // printf("| %3Ld | %.3lf | %.3lf | %.3lf | %.3lf | %.3lf | %.3lf | %.5lf |\n",
    //        k, x[0], x[1], x[2], Delta[0], Delta[1], y, epsilon);
}

vector<vector<double>> Method(vector<double (*)(vector<double>)> f, vector<double> x0, double a,
double b, double h, double (*foo)(double), Methods method) {
    size_t Dim = f.size();
    vector<double> xk = x0, Delta(Dim);
    size_t k = 0;
    vector<vector<double>> res = { xk };
    if (method == Methods::MethodAdams4) {
        res = Runge_Kutt_Method4(f, x0, a, a + h * 3, h, foo);
        xk = res[3];
        k = 3;
    } else if (method == Methods::MethodAdams2) {
        res = Runge_Kutt_Method4(f, x0, a, a + h, h, foo);
        xk = res[1];
        k = 1;
    }
    for (; xk[0] < b; k++) {
        switch (method) {
            case Methods::Runge_Kutt_Method: {
                size_t BDim = Butcher.dim.first;
                vector<vector<double>> K(Dim, vector<double>(BDim - 1, 0));
                for (size_t j = 0; j < BDim - 1; j++) {
                    vector<double> xj = xk;

                    xj[0] += Butcher[j][0] * h;
                    for (size_t i = 0; i < Dim; i++)
                        for (size_t k = 0; k < j; k++)
                            xj[i + 1] += Butcher[j][k + 1] * K[i][k];

                    for (size_t i = 0; i < Dim; i++) K[i][j] = h * f[i](xj);
                }
                for (size_t i = 0; i < Dim; i++) {
                    Delta[i] = 0;
                    for (size_t j = 0; j < BDim - 1; j++)
                        Delta[i] += Butcher[BDim - 1][j + 1] * K[i][j];
                }
                break;
            }

            case Methods::ExplicitEulerMethod:
                for (size_t i = 0; i < Dim; i++) Delta[i] = h * f[i](xk);
                break;
```

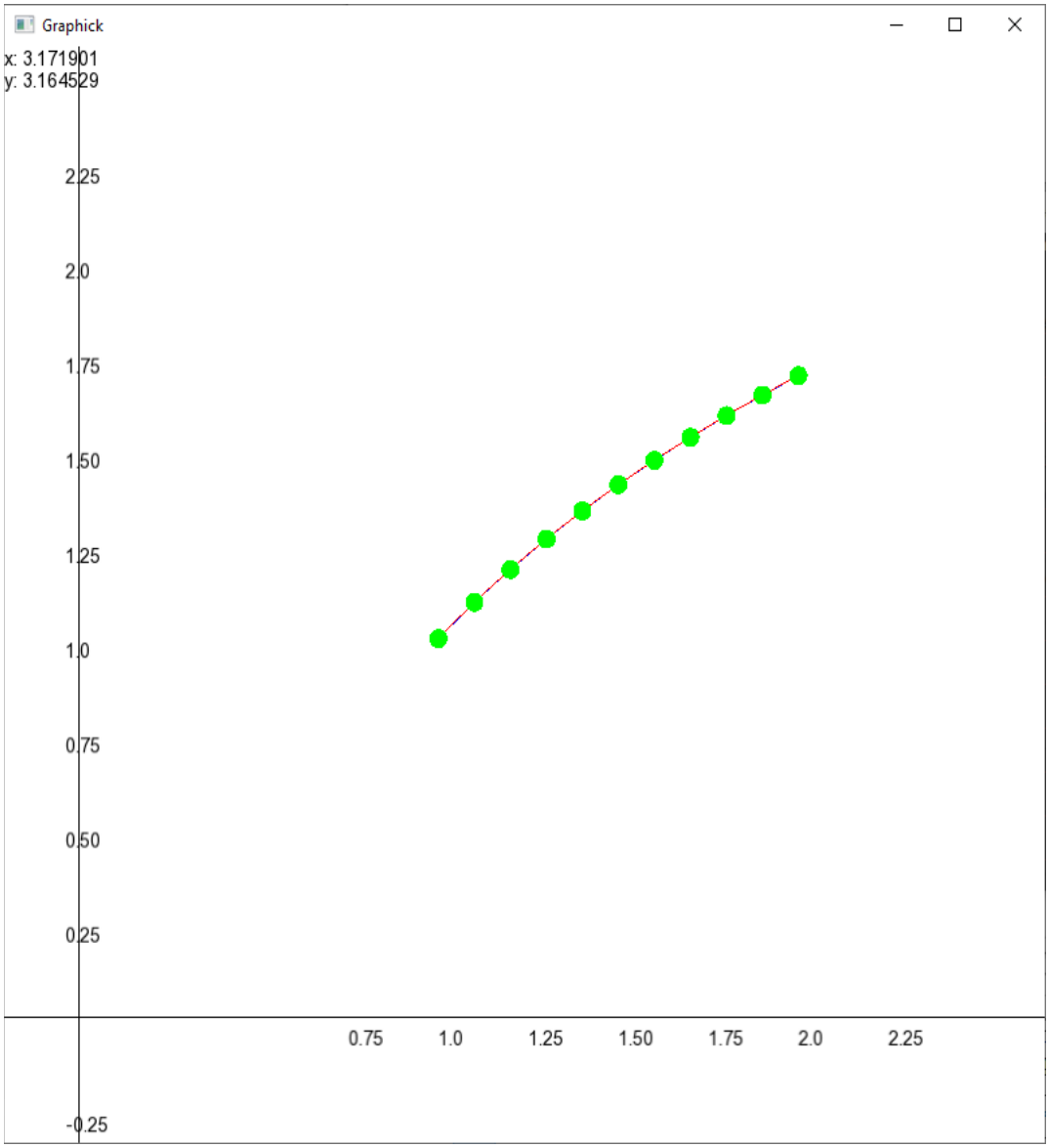
```
            case Methods::MethodEuler_Cauchy:
                for (size_t i = 0; i < Dim; i++) {
                    vector<double> xj = xk; xj[0] += h;
                    xj[i] = xk[i + 1] + h * f[i](xk);
                    Delta[i] = h * ( f[i](xk) + f[i](xj) ) / 2;
                }
                break;

            case Methods::MethodAdams2:
                for (size_t i = 0; i < Dim; i++)
                    Delta[i] = h * ( 3 * f[i](xk) - f[i](res[k - 1]) ) / 2;
                break;

            case Methods::MethodAdams4:
                for (size_t i = 0; i < Dim; i++)
                     Delta[i] = h * ( 55 * f[i](xk) - 59 * f[i](res[k - 1]) + 37 * f[i](res[k -
2]) - 9 * f[i](res[k - 3])) / 24;
                break;
        }
        print(k, xk, Delta, foo(xk[0]), abs(foo(xk[0]) - xk[1]));

        xk[0] += h;
        for (size_t i = 0; i < Dim; i++) xk[i + 1] += Delta[i];
        res.push_back(xk);
    }
    print(k, xk, Delta, foo(xk[0]), abs(foo(xk[0]) - xk[1]));
    return res;
}

vector<vector<double>>  Runge_Kutt_Method2(vector<double  (*)(vector<double>)>  f,  vector<double>
x0, double a, double b, double h, double (*foo)(double)) {
    Butcher.resize(3, 3);
    Butcher[0] = {    0,    0,    0 };
    Butcher[1] = {    1,    1,    0 };
    Butcher[2] = {    0,  0.5,  0.5 };
    return Method(f, x0, a, b, h, foo, Methods::Runge_Kutt_Method);
}
vector<vector<double>>  Runge_Kutt_Method3(vector<double  (*)(vector<double>)>  f,  vector<double>
x0, double a, double b, double h, double (*foo)(double)) {
    Butcher.resize(4, 4);
    Butcher[0] = {    0,    0,    0,    0 };
    Butcher[1] = { 1./3, 1./3,    0,    0 };
    Butcher[2] = { 2./3,    0, 2./3,    0 };
    Butcher[3] = {    0, 1./4,    0, 3./4 };
    return Method(f, x0, a, b, h, foo, Methods::Runge_Kutt_Method);
}
vector<vector<double>>  Runge_Kutt_Method4(vector<double  (*)(vector<double>)>  f,  vector<double>
x0, double a, double b, double h, double (*foo)(double)) {
    Butcher.resize(5, 5);
    Butcher[0] = {   0,    0,    0,    0,    0 };
    Butcher[1] = { 0.5,  0.5,    0,    0,    0 };
    Butcher[2] = { 0.5,    0,  0.5,    0,    0 };
    Butcher[3] = {   1,    0,    0,    1,    0 };
    Butcher[4] = {   0, 1./6, 1./3, 1./3, 1./6 };
    return Method(f, x0, a, b, h, foo, Methods::Runge_Kutt_Method);
}
vector<vector<double>>  ExplicitEulerMethod(vector<double  (*)(vector<double>)>  f,  vector<double>
x0, double a, double b, double h, double (*foo)(double)) {
    return Method(f, x0, a, b, h, foo, Methods::ExplicitEulerMethod);
}
vector<vector<double>>  MethodEuler_Cauchy(vector<double  (*)(vector<double>)>  f,  vector<double>
x0, double a, double b, double h, double (*foo)(double)) {
    return Method(f, x0, a, b, h, foo, Methods::MethodEuler_Cauchy);
}
vector<vector<double>>  MethodAdams2(vector<double  (*)(vector<double>)>  f,  vector<double>  x0,
double a, double b, double h, double (*foo)(double)) {
    return Method(f, x0, a, b, h, foo, Methods::MethodAdams2);
}
vector<vector<double>>  MethodAdams4(vector<double  (*)(vector<double>)>  f,  vector<double>  x0,
double a, double b, double h, double (*foo)(double)) {
```

```
    return Method(f, x0, a, b, h, foo, Methods::MethodAdams4);
}
```

main.cpp

```
#include "tools.h"
#include "../../Graphica.h"

double foo(double x) { return  1 + log(abs(x)); }

int main() {
    vector<vector<double>> res;
    vector<vector<vector<double>> (*)(vector<double (*)(vector<double>)>, vector<double>, double,
double, double, double (*)(double))> funcs = {
        Runge_Kutt_Method2,
        Runge_Kutt_Method3,
        Runge_Kutt_Method4,
        ExplicitEulerMethod,
        MethodEuler_Cauchy,
        MethodAdams2,
        MethodAdams4
    };
    double a = 1, b = 2, h = 0.1;
    vector<double> x0 = {a, 1, 1};
    vector<double (*)(vector<double>)> f(2);
    f[0] = [](vector<double> x) { return x[2]; };
    f[1] = [](vector<double> x) { return - x[2] / x[0]; };

    char TheMethod;
    cout << "What TheMethod do you want to use?\n"
    "1 - Runge_Kutt_Method2\n"
    "2 - Runge_Kutt_Method3\n"
    "3 - Runge_Kutt_Method4\n"
    "4 - ExplicitEulerMethod\n"
    "5 - MethodEuler_Cauchy\n"
    "6 - MethodAdams2\n"
    "7 - MethodAdams4\n";
    do { cout << ">>> "; cin >> TheMethod; } while (TheMethod < '1' || '7' < TheMethod);

    char reply;
    do { cout << "Want to use the standard value h = 0.1?\ny/n: "; cin >> reply;
    } while (reply != 'n' && reply != 'y');
    if (reply == 'n') { cout << "Enter a new h value: "; cin >> h; }

    res = funcs[TheMethod - '1'](f, x0, a, b, h, foo);
    vector<pair<double, double>> points;
    for (vector<double>& x: res) {
        addPoint(x[0], x[1]);
        points.emplace_back(x[0], x[1]);
    }
    SetDiapazon(a, b);
    makeGraphByPoints(points);
    makeGraph(foo, sf::Color::Red);
    ShowGraphics();
    return 0;
}
```

Вывод




```
What TheMethod do you want to use?
1 - Runge_Kutt_Method2
2 - Runge_Kutt_Method3
3 - Runge_Kutt_Method4
4 - ExplicitEulerMethod
5 - MethodEuler_Cauchy
6 - MethodAdams2
7 - MethodAdams4
>>> 3
Want to use the standard value h = 0.1?
y/n: y
```

k	x	y	z	dy	dz	real	epsilon
0	1	1	1	0.09531	-0.09091	1	0
1	1.1	1.095	0.9091	0.08701	-0.07576	1.095	2.781e-07
2	1.2	1.182	0.8333	0.08004	-0.0641	1.182	4.535e-07
3	1.3	1.262	0.7692	0.07411	-0.05495	1.262	5.686e-07
4	1.4	1.336	0.7143	0.06899	-0.04762	1.336	6.466e-07
5	1.5	1.405	0.6667	0.06454	-0.04167	1.405	7.01e-07
6	1.6	1.47	0.625	0.06062	-0.03676	1.47	7.398e-07
7	1.7	1.531	0.5882	0.05716	-0.03268	1.531	7.682e-07
8	1.8	1.588	0.5556	0.05407	-0.02924	1.588	7.892e-07
9	1.9	1.642	0.5263	0.05129	-0.02632	1.642	8.052e-07

10	2	1.693	0.5	0.05129	-0.02632	1.693	8.174e-07

Задание 2

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге-Ромберга и путем сравнения с точным решением.

15	x lnx y″ – y′–x ln³x y=0, y(1,2)-y`(1,2)=3,645, y(4,5)=9,899	$y(x)=\left(\frac{x}{e}\right)^x+\left(\frac{e}{x}\right)^x$

Теоретические сведения

Примером краевой задачи является двухточечная краевая задача для обыкновенного дифференциального уравнения второго порядка.

$$y''=f(x,y,y')$$

с граничными условиями, заданными на концах отрезка $[a,b]$.

$$\begin{matrix} y(a)=y_0 \\ y(b)=y_1 \end{matrix} - граничные условия\ 1\ рода$$

Следует найти такое решение $y(x)$ на этом отрезке, которое принимает на концах отрезка значения y_0, y_1.

Кроме граничных условий первого рода, используются еще условия на производные от решения на концах - граничные условия второго рода:

$$\begin{matrix} y'(a)=\hat{y}_0 \\ y'(b)=\hat{y}_1 \end{matrix}$$

или линейная комбинация решений и производных – граничные условия третьего рода:

$$\begin{matrix} \alpha y(a)+\beta y'(a)=\hat{y}_0 \\ \delta y(b)+\gamma y'(b)=\hat{y}_1 \end{matrix}$$

Возможно на разных концах отрезка использовать условия различных типов.

Метод стрельбы:

Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи.

Пусть надо решить краевую задачу краевыми условиями 1-го рода на отрезке $[a,b]$. Вместо исходной задачи формулируется задача Коши с уравнением $y''=f(x,y,y')$ и с граничными условиями первого рода в виде

$$y(a)=\eta$$
$$y'(a)=y_1$$

Положим сначала некоторое начальное значение параметру $\eta=\eta_0$, после чего решим каким-либо методом задачу Коши. Сравнивания значение решения этой задачи со значением y_1 в правом конце отрезка можно получить информацию для корректировки угла наклона касательной к решению на левом конце отрезка. Решим задачу Коши для нового значения $\eta=\eta_1$, получим другое решение. Так решение задачи на правом конце будет являться функцией одной переменной. Задачу можно сформулировать таким образом: требуется найти такое значение переменной η, чтобы решение задачи Коши в правом конце отрезка совпадало с граничным условием $\delta y(b)+\gamma y'(b)=\theta$. Другими словами решение исходной задачи эквивалентно нахождению корня уравнения

$$\Phi(\eta)=y(b,y_0,\eta)-y_1=0$$

Следующее значение искомого корня определяется по соотношению

$$\eta_{j+2}=\eta_{j+1}-\frac{\eta_{j+1}-\eta_j}{\Phi(\eta_{j+1})-\Phi(\eta_j)}\Phi(\eta_{j+1})$$

Конечно-разностный метод:

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке $[a,b]$

$$y''+p(x)y'+q(x)y=f(x)$$
$$y'(a)=c_1, a_2 y(b)+b_2 y'(b)=c_2$$

Введем разностную аппроксимацию производных следующим образом:

$$y_0'=\frac{-3y_0+4y_1-y_2}{2h}+O(h^2);$$

$$y_k'=\frac{y_{k+1}-y_{k-1}}{2h}+O(h^2), k\in[1,N-1];$$

$$y_k''=\frac{y_{k+1}-2y_k+y_{k-1}}{h^2}+O(h^2), k\in[1,N-1];$$

$$y'_N = \frac{y_{N-2} - 4y_{N-1} + 3y_N}{2h} + O(h^2);$$

Подставляя аппроксимации производных, получим систему уравнений для нахождения y_k:

$$\begin{cases} \dfrac{-3y_0 + 4y_1 - y_2}{2h} = c_1 \\ \dfrac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + p(x_k)\dfrac{y_{k+1} - y_{k-1}}{2h} + q(x_k)y_k = f(x_k) \\ a_2 y_N + b_2 \dfrac{y_{N-2} - 4y_{N-1} + 3y_N}{2h} = c_2 \end{cases}$$

Приводя подобные, получаем следующую систему

$$\begin{cases} \dfrac{-3}{2h}y_0 + \dfrac{2}{h}y_1 - \dfrac{1}{2h}y_2 = c_1 \\ \left(1 - \dfrac{p(x_k)h}{2}\right)y_{k-1} + \left(h^2 q(x_k) - 2\right)y_k + \left(1 + \dfrac{p(x_k)h}{2}\right)y_{k+1} = h^2 f(x_k), k = 2, \dots, N-2 \\ \dfrac{b_2}{2h}y_{N-2} - \dfrac{2b_2}{h}y_{N-1} + \left(a_2 + \dfrac{3b_2}{2h}\right)y_N = c_2 \end{cases}$$

Теперь система может быть решена методом прогонки. Получаем значения $y_1, \dots, y_{N-1}$. Теперь найдем y_0 и y_N из формул:

$$y_0 = \frac{-2hc_1}{3} + \frac{4}{3}y_1 - \frac{1}{3}y_2, \text{подставляянайденные } y_1, y_2$$

$$y_N = \frac{2hc_2 - b_2 y_{N-2} + 4b_2 y_{N-1}}{2ha_2 + 3b_2}, \text{подставляяянайденные } y_{N-2}, y_{N-1}$$

Оценить погрешность решения можно сравнивая с точным решением, если оно есть, или с помощью метода Рунге-Ромберга. Он позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла на сетке с шагом $h - y = y_h + O(h^p)$ и на сетке с шагом $kh - y = y_{kh} + O((kh)^p)$, то

$$y = y_h + \frac{y_h - y_{kh}}{k^p - 1} + O(h^{p+1})$$

Код:

tools.h

```
#include "../first/tools.h"
#include "../../lab1/TMA/TMA.hpp"
#include "../../Graphica.h"
using namespace std;

void CreateGraphic(vector<vector<double>> v) {
    vector<pair<double, double>> points;
    for (vector<double>& x: v) {
        addPoint(x[0], x[1]);
        points.emplace_back(x[0], x[1]);
    }
    makeGraphByPoints(points);
}

void print(size_t k, double heta_j, double y, double epsilon) {
    if (k == 0)
        cout << "-------------------------------------------------\n"
             << "|   j :   heta_j    :      y      : |F(heta_j)| |\n"
             << "------+-------------+-------------+--------------\n";
    cout << setprecision(4) << right << setfill(' ') << "| " << setw(3) << k << left;
    cout << setw(4) << " : " << setw(10) << heta_j;
    cout << setw(4) << " : " << setw(10) << y;
    cout << setw(4) << " : " << setw(10) << epsilon << " |\n";
    cout << "------+-------------+-------------+--------------\n";
}

vector<vector<double>> ShootingMethod(vector<double (*)(vector<double>)> f, vector<double> x0,
double a, double b, double h, double epsilon, double (*foo)(double)) {
    double heta_was = 10, heta_cur = 12, F_cur, F_was, heta_res;
    vector<vector<double>> res;

    res = Runge_Kutt_Method4(f, {x0[0], x0[1], heta_was}, a, b, h, foo);
    CreateGraphic(res);
    double y_was = res[res.size() - 1][1];
    F_was = res[res.size() - 1][1] - res[res.size() - 1][2] - 3.645;
    print(0, heta_was, y_was, abs(F_was));

    res = Runge_Kutt_Method4(f, {x0[0], x0[1], heta_cur}, a, b, h, foo);
    CreateGraphic(res);
    double y_cur = res[res.size() - 1][1];
    F_cur = res[res.size() - 1][1] - res[res.size() - 1][2] - 3.645;
    print(1, heta_cur, y_cur, abs(F_cur));

    for (size_t k = 2; abs(F_cur) > epsilon; k++) {
        heta_res = heta_cur - (heta_cur - heta_was) * F_cur / (F_cur - F_was);

        y_was = y_cur;
        heta_was = heta_cur;

        heta_cur = heta_res;
        res = Runge_Kutt_Method4(f, {x0[0], x0[1], heta_cur}, a, b, h, foo);
        CreateGraphic(res);
        y_cur = res[res.size() - 1][1];

        F_was = F_cur;
        F_cur = res[res.size() - 1][1] - res[res.size() - 1][2] - 3.645;
        print(k, heta_cur, y_cur, abs(F_cur));
    }

    return res;
}
vector<vector<double>> FiniteDifferenceMethod(double (*p)(double), double (*q)(double), double
(*f)(double), double a, double b, double ya, double yb, double h) {
    vector<vector<double>> res;
    vector<double> qX, pX, fX;
    for (double x = a + h; x * sign(h) < b * sign(h); x += h) {
```

```
        pX.push_back(p(x));
        qX.push_back(q(x));
        fX.push_back(f(x));
    }
    size_t N = pX.size();

    Matrix<double> m(3, N), d(1, N);
    m[0][0] = 0; m[1][0] = 11; m[2][0] = 1;
    d[0][0] = 36.45;
    for (size_t i = 1; i < N - 1; i++) {
        m[0][i] = 1 - pX[i] * h / 2; m[1][i] = -2 + h * h * qX[i]; m[2][i] = 1 + pX[i] * h / 2;
        d[0][i] = h * h * fX[i];
    }
    m[0][N - 1] = 1 - pX[N - 1] * h / 2; m[1][N - 1] = -2 + h * h * qX[N - 1]; m[2][N - 1] = 0;
    d[0][N - 1] = h * h * fX[N - 1] - (1 + pX[N - 1] * h / 2) * yb;
    Matrix<double> y = TMAsolve(m, d);

    res.push_back({a, ya});
    for (size_t i = 0; i < N; i++) res.push_back({a + h * i, y[0][i]});
    res.push_back({b, yb});

    return res;
}
```

main.cpp

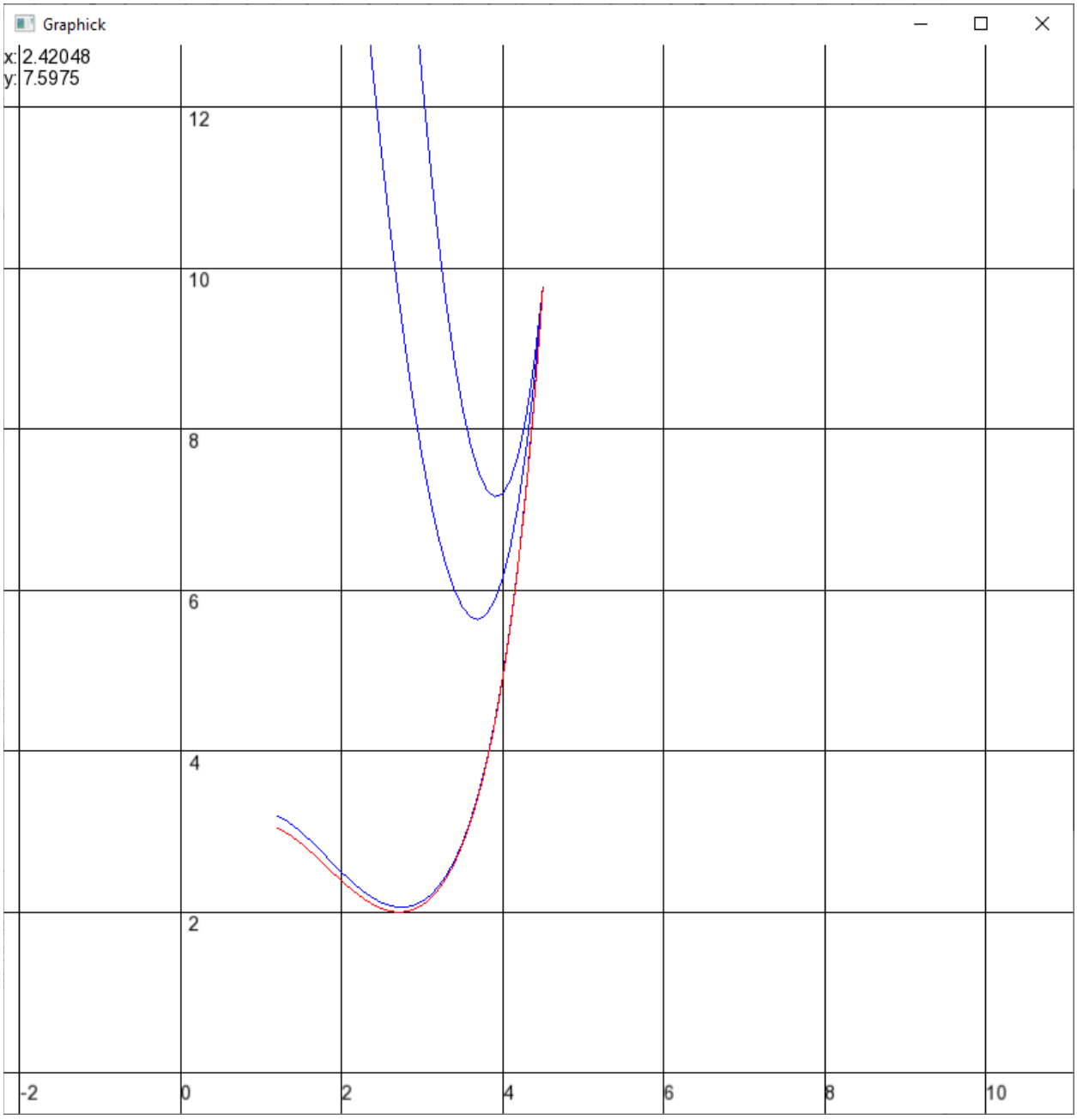
```
#include "tools.h"

double foo(double x) { return pow(x / M_E, x) + pow(M_E / x, x); }

int main() {
    SetDiapazon(1.2, 4.5);
    double a = 1.2, b = 4.5, h = -0.1, epsilon = 10e-3;
    vector<vector<double>> res;
    char TheMethod;
    cout << "What TheMethod do you want to use?\n"
    "1 - ShootingMethod\n"
    "2 - FiniteDifferenceMethod\n";
    do { cout << ">>> "; cin >> TheMethod; } while (TheMethod < '1' || '2' < TheMethod);

    printing = false;
    if (TheMethod == '1') {
        vector<double> x = {b, 9.76690936686, 10};
        vector<double (*)(vector<double>)> f = {
            [](vector<double> x) { return x[2]; },
              [](vector<double> x) { return (x[2] + x[1] * x[0] * pow(log(x[0]), 3)) / (x[0] *
log(x[0])); }
        };
        res = ShootingMethod(f, x, b, a, h, epsilon, foo);
    } else {
        res = FiniteDifferenceMethod([](double x) { return - 1. / (x * log(x)); },
        [](double x) { return - log(x) * log(x); }, [](double x) { return 0.; },
        1.2, 4.5, 3.04254890597, 9.76690936686, 0.1);
    }

    vector<pair<double, double>> points;
    for (vector<double>& x: res) {
        addPoint(x[0], x[1]);
        points.emplace_back(x[0], x[1]);
    }
    makeGraphByPoints(points);
    makeGraph(foo, sf::Color::Red);
    ShowGraphics();
    return 0;
}
```

Вывод:

Первый запуск

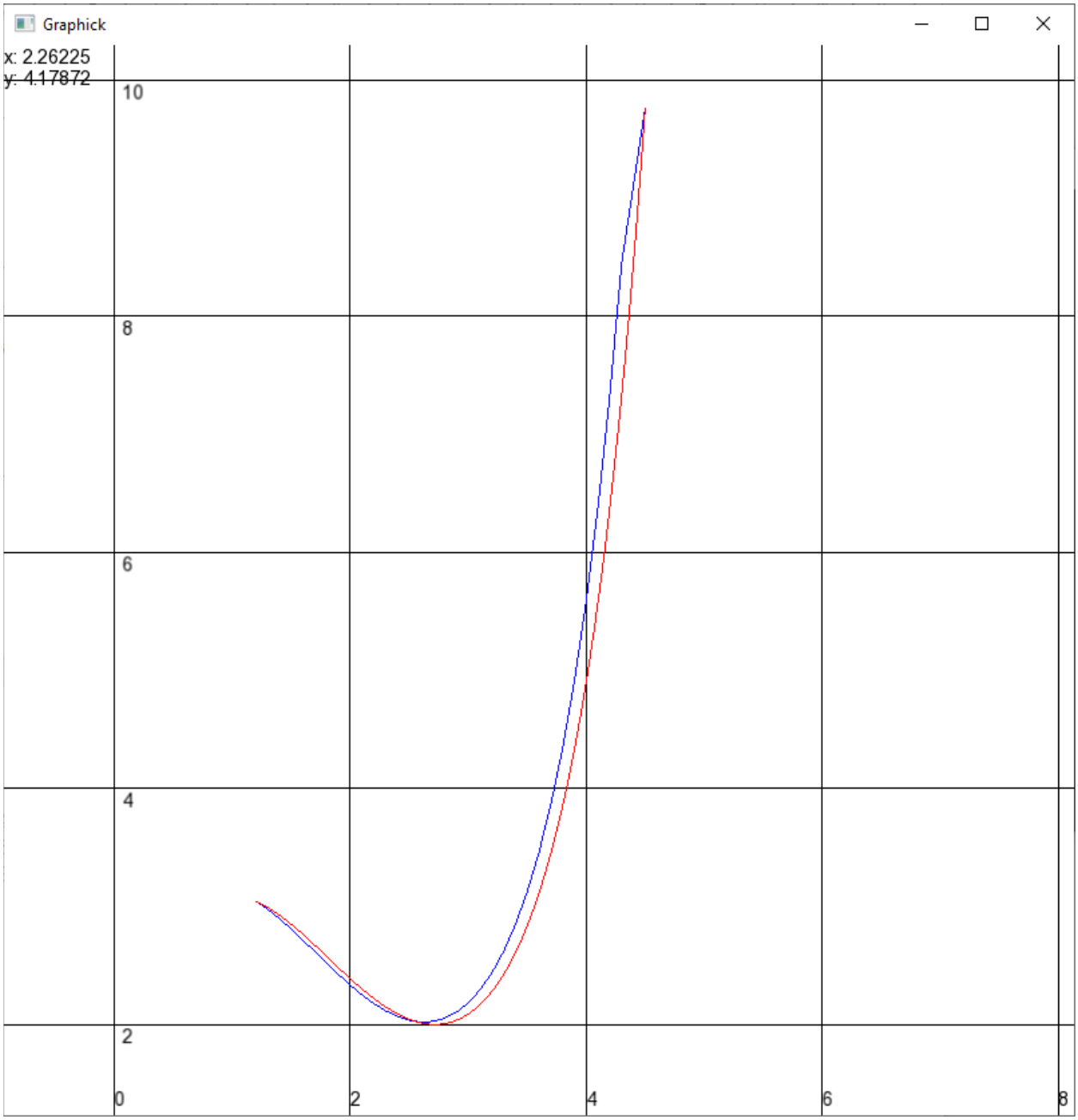
```
What TheMethod do you want to use?
1 - ShootingMethod
```

```
2 - FiniteDifferenceMethod
>>> 1
-------------------------------------------------
|   j :   heta_j     :       y       : |F(heta_j)| |
------+-------------+-------------+--------------
|   0 :  10          :  40.51       :  44.14       |
------+-------------+-------------+--------------
|   1 :  12          :  23.4        :  23.89       |
------+-------------+-------------+--------------
|   2 :  14.36       :  3.199       :  3.864e-14   |
------+-------------+-------------+--------------
```



Второй запуск

```
What TheMethod do you want to use?
1 - ShootingMethod
2 - FiniteDifferenceMethod
>>> 2
```



Где я ошибался:

Сначала не правильно понял что есть $\Phi(\eta)=0$.
Оказывается, что $\Phi(\eta)$, в моём случае, имеет вид $\Phi(\eta_i)=y_i-y'_i-3.645$, где y, y' - значение, полученное приближённо методом Рунге-Кутта 4 порядка.