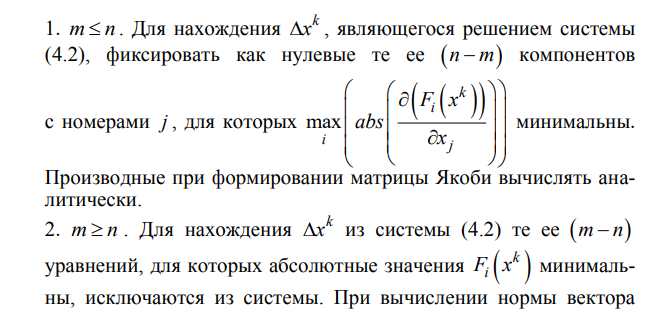
|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра прикладной математики | | |
| Практическое работа № 4 | | |
| по дисциплине «Численные методы» | | |
| **Решение систем нелинейных уравнений методом ньютона** | | |
|  | | |
|  |  |  |
| Группа ПМ-01 | будник светлана |
|  | самсонов семён |
| Вариант 11 |  |
|  |  |
| Преподаватель | Задорожный александр геннадьевич |
|  | Патрушев Илья игоревич |
|  |  |
| Новосибирск, 2023 | | |

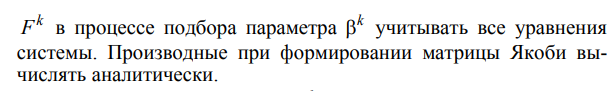
# Цели работы

Разработать программу решения системы нелинейных уравнений (СНУ) методом Ньютона. Провести исследования метода для нескольких систем размерности от 2 до 10.

# Входные данные

**Задания (1,2,6):**







# Теоретическая часть

Пусть дана СНУ в виде:

(4.1)

Обозначим через решение, полученное на -й итерации процесса Ньютона (для первой итерации – начальное приближение). Запишем исходную систему в виде , , где ,  – искомое решение. Выполним линеаризацию i-го уравнения системы (4.1) с использованием его разложения в ряд Тейлора в окрестности точки :

, ,

или в матричном виде:

(4.2)

где – значение вектор-функции при; – матрица Якоби.

Это система уравнений, линейных относительно приращений. Решив эту систему, найдем направление поиска решения. Для поиска следующего приближения вдоль направления организуем итерационный процесс:

,

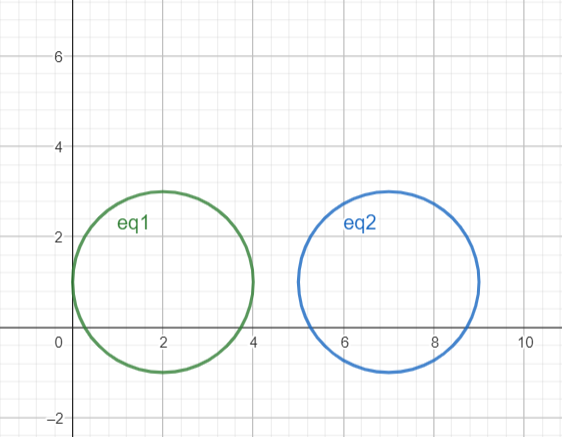
где – параметр итерационного процесса поиска , (), – номер итерации поиска оптимального значения . Параметр будем искать следующим образом: сначала (т. е. после нахождения направления) принимается равным 1 и вычисляется значение ; далее, пока норма больше, чем норма, уменьшается вдвое.

Для получения численных значений элементов матрицы Якоби воспользуемся следующей формулой:

# Исследование

**Тесты:**

1. **Две не пересекающиеся окружности:**



**Начальное приближение не лежит на оси симметрии (4, 4)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 0.01562 | 4.500000000000000e+00 | 1.011997767857142e+00 | 3.18e+00 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 0.01562 | 4.500000000000000e+00 | 1.011997767857142e+00 | 3.18e+00 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 0.01562 | 4.500000000000000e+00 | 1.011997767857142e+00 | 3.18e+00 |

**Начальное приближение лежит на оси, соединяющей центры окружностей (4.5, 1)**

Решение найти не удаётся, поскольку матрица Якоби в этой точке не имеет диагонального преобладания.

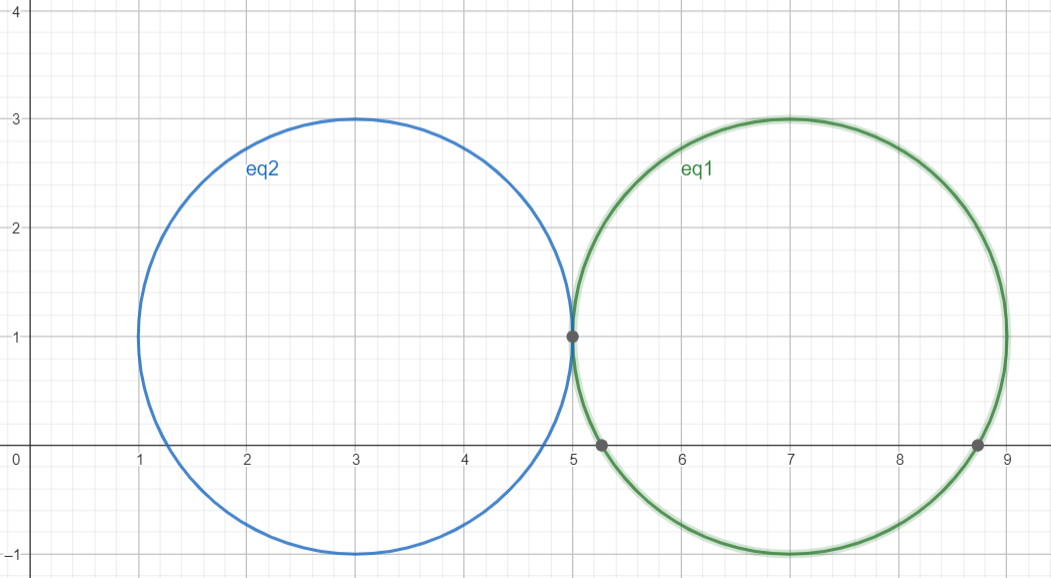
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 1 | 1.00000 | -nan(ind) | -inf | -nan(ind) |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 1 | 1.00000 | -nan(ind) | -inf | -nan(ind) |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 1 | 1.00000 | -nan(ind) | -inf | -nan(ind) |

**Начальное приближение лежит на оси, перпендикулярной оси, соединяющей центры окружностей и пересекающей ее на равных расстояниях от центров окружностей (4.5, 2)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 5 | 0.01562 | 4.500000000000000e+00 | 9.692959082188743e-01 | 3.18e+00 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 5 | 0.01562 | 4.500000000000000e+00 | 9.692959082188743e-01 | 3.18e+00 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 5 | 0.01562 | 4.500000000000000e+00 | 9.692959167950940e-01 | 3.18e+00 |

1. **Две окружности пересекаются в одной точке**

***Точка пересечения в (5,1)***



**Начальное приближение не лежит на оси симметрии (4, 4)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 11 | 1.00000 | 5.000000000000000e+00 | 1.001627604166699e+00 | 3.75e-06 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 11 | 1.00000 | 5.000000000000000e+00 | 1.001627604166699e+00 | 3.75e-06 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 11 | 1.00000 | 5.000000000000000e+00 | 1.001627604162562e+00 | 3.75e-06 |

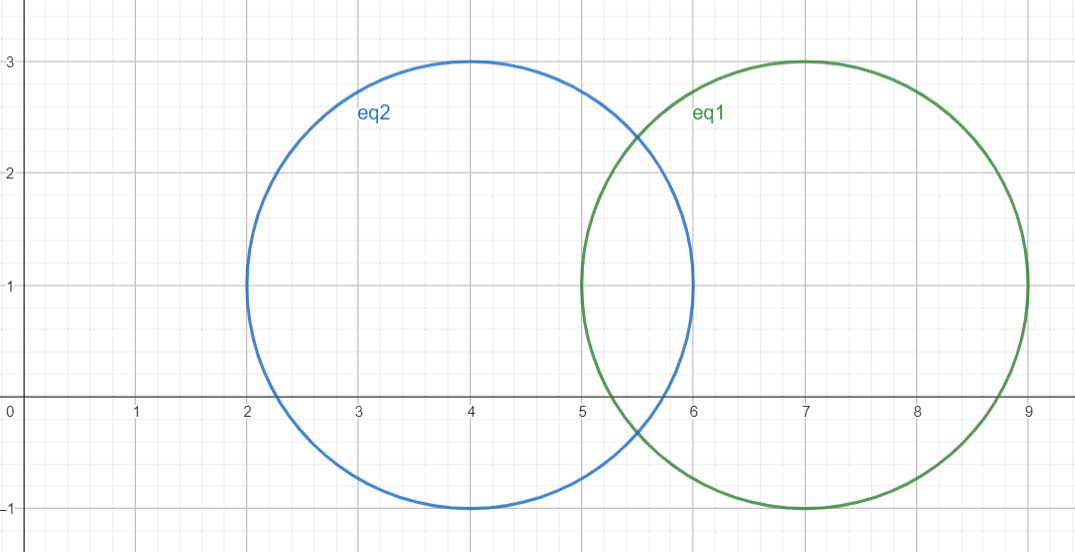
**Начальное приближение лежит на оси, перпендикулярной оси, соединяющей центры окружностей и пересекающей ее на равных расстояниях от центров окружностей (5, 2)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 9 | 1.00000 | 5.000000000000000e+00 | 1.001953125000000e+00 | 5.39e-06 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 9 | 1.00000 | 5.000000000000000e+00 | 1.001953125000000e+00 | 5.39e-06 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 9 | 1.00000 | 5.000000000000000e+00 | 1.001953124995069e+00 | 5.39e-06 |

**Начальное приближение лежит в центре или внутри одной из окружностей (5.5, 2)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 9 | 1.00000 | 5.000000000000000e+00 | 1.002441406250000e+00 | 8.43e-06 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 9 | 1.00000 | 5.000000000000000e+00 | 1.002441406250000e+00 | 8.43e-06 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 9 | 1.00000 | 5.000000000000000e+00 | 1.002441406299384e+00 | 8.43e-06 |

1. **Две окружности пересекаются в двух точках**



**Начальное приближение не лежит на оси симметрии (5, 3)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875656167979e+00 | 2.38e-09 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875656167979e+00 | 2.38e-09 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875656167986e+00 | 2.38e-09 |

**Начальное приближение лежит на оси, перпендикулярной оси, соединяющей центры окружностей и пересекающей ее на равных расстояниях от центров окружностей (5.5, 2)**

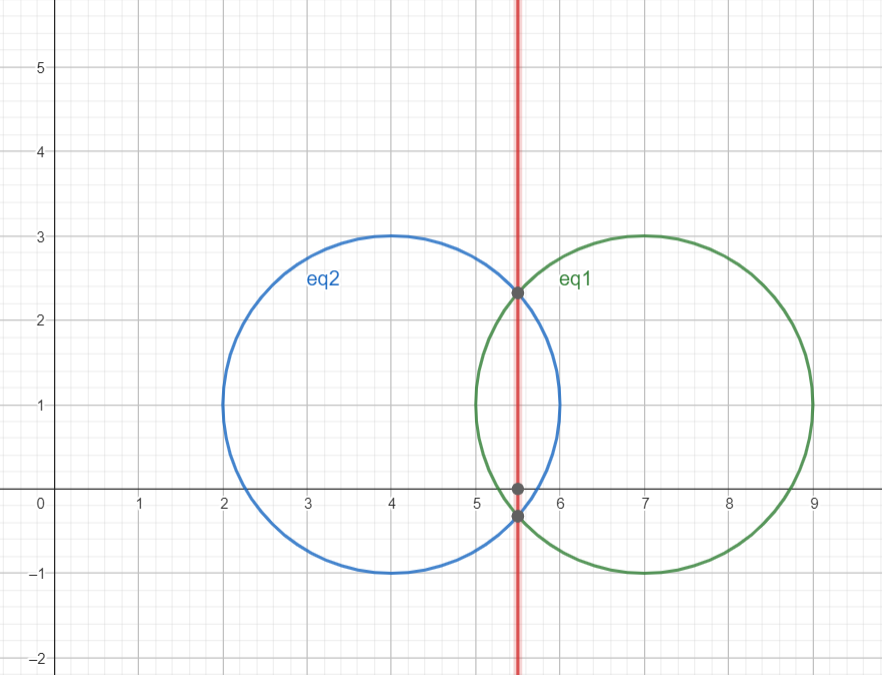
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 1.00000 | 5.500000000000000e+00 | 2.322876024190402e+00 | 1.38e-06 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 1.00000 | 5.500000000000000e+00 | 2.322876024190402e+00 | 1.38e-06 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 1.00000 | 5.500000000000000e+00 | 2.322876024190498e+00 | 1.38e-06 |

**Начальное приближение лежит в центре или внутри одной из окружностей (6.5, 2.5)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875736454466e+00 | 3.03e-07 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875736454466e+00 | 3.03e-07 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875736454588e+00 | 3.03e-07 |

1. **Две окружности и прямая**

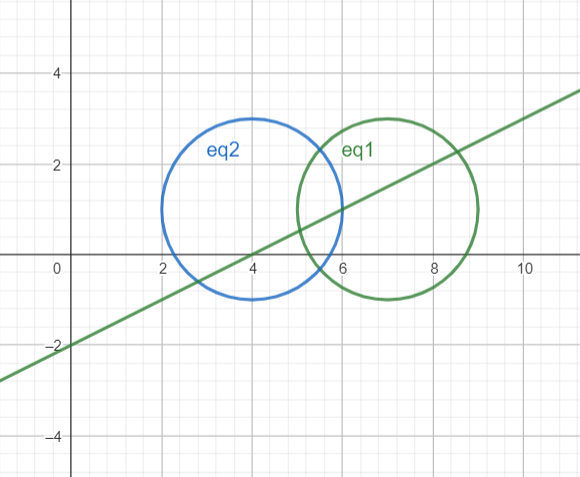
***Есть решение:***



Начальное приближение (5.5, 3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875655555685e+00 | 8.75e-11 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 1.00000 | 5.500000000000000e+00 | 2.322875655555686e+00 | 8.75e-11 |

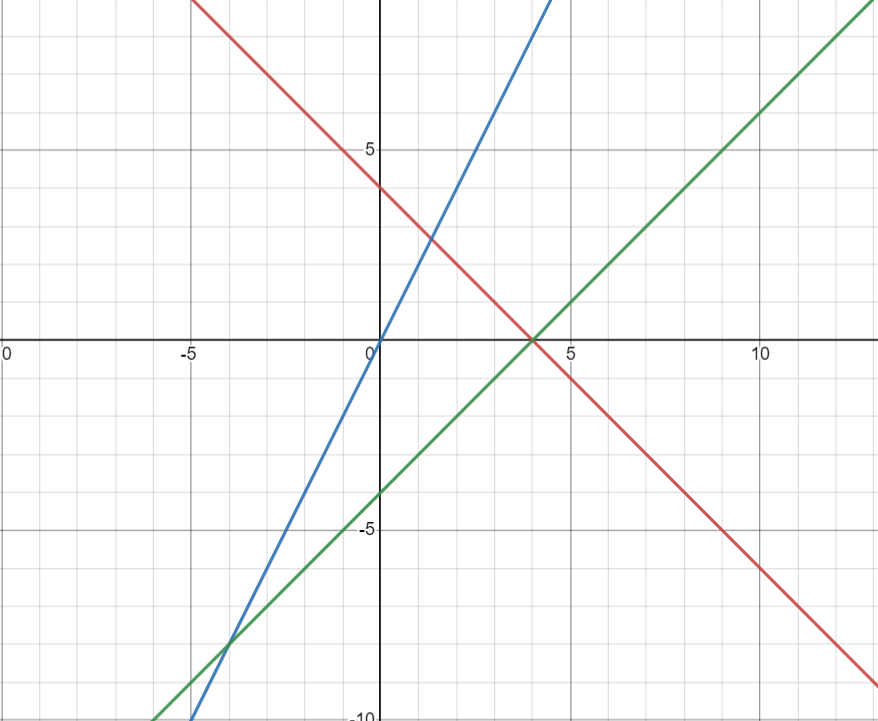
1. **Нет решения уравнения:**



Начальное приближение (5.5; 3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 0.01562 | 5.727342106545140e+00 | 2.368462559302731e+00 | 1.80e+00 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 0.01562 | 5.556929485038580e+00 | 2.020033081664295e+00 | 1.61e+00 |

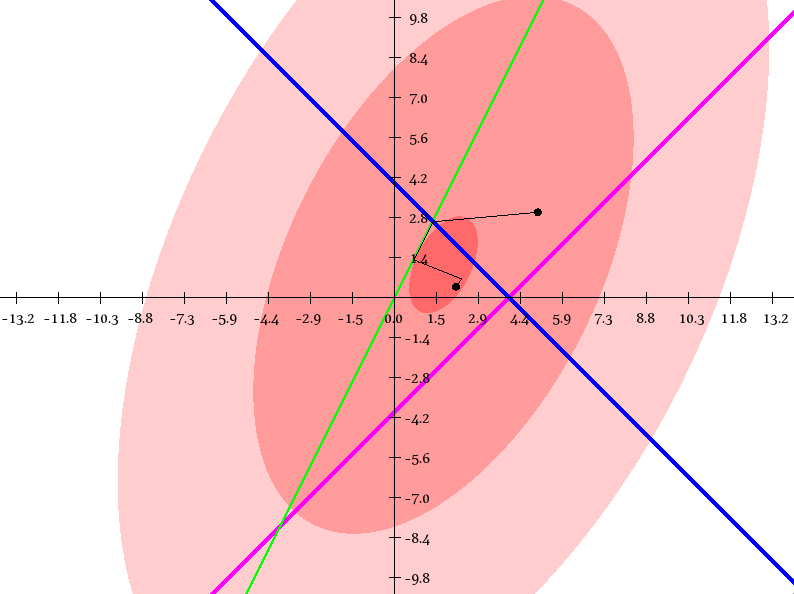
1. **Три попарно пересекающиеся прямые**



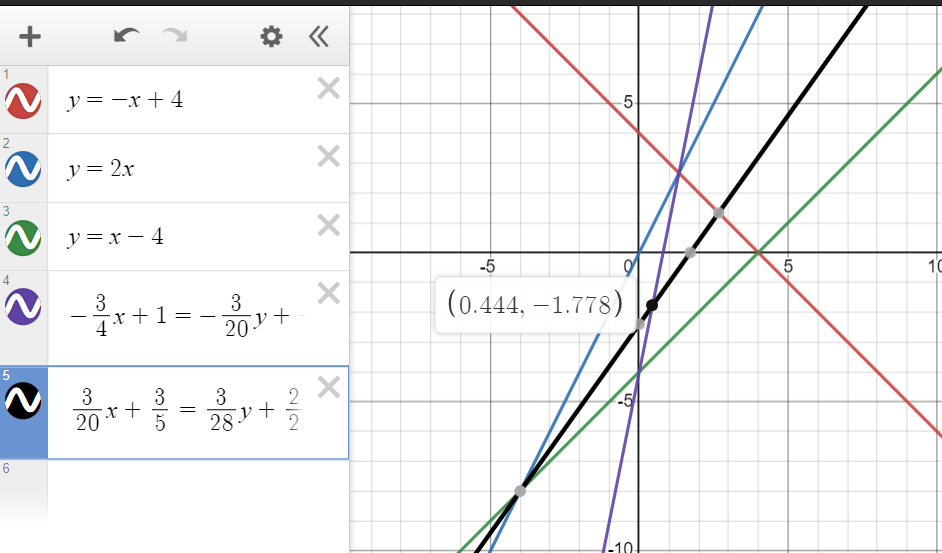
(5, 3) за границами треугольника:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 0.01562 | 2.135416666666667e+00 | 3.958333333333334e-01 | 4.72e+00 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 0.01562 | 1.833333333770020e+00 | 1.666666675313035e-01 | 4.66e+00 |

График движения точки метода:

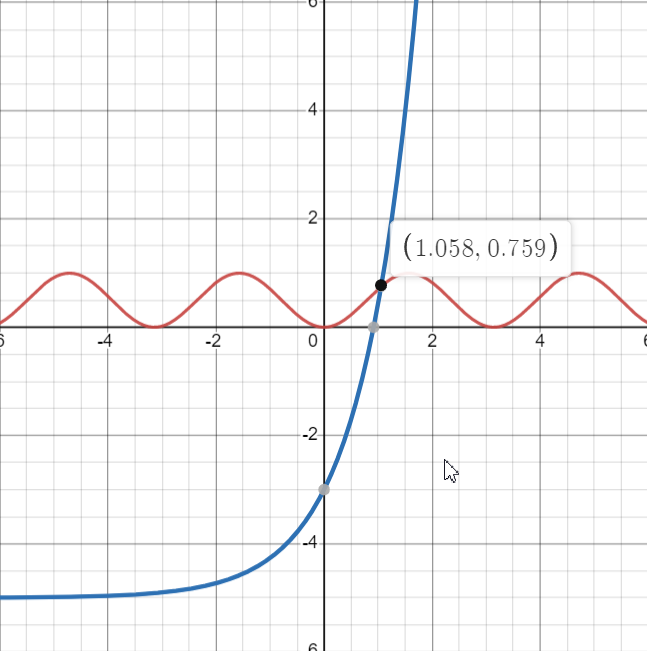


() точка пересечения медиан:



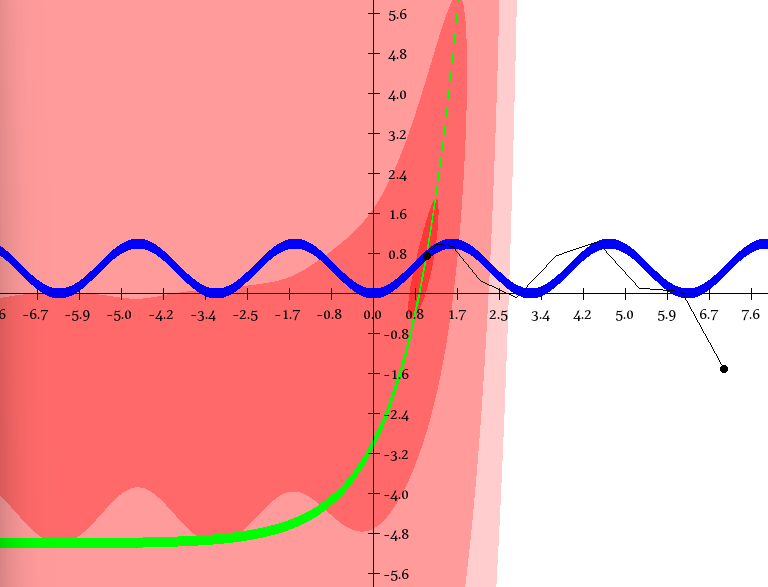
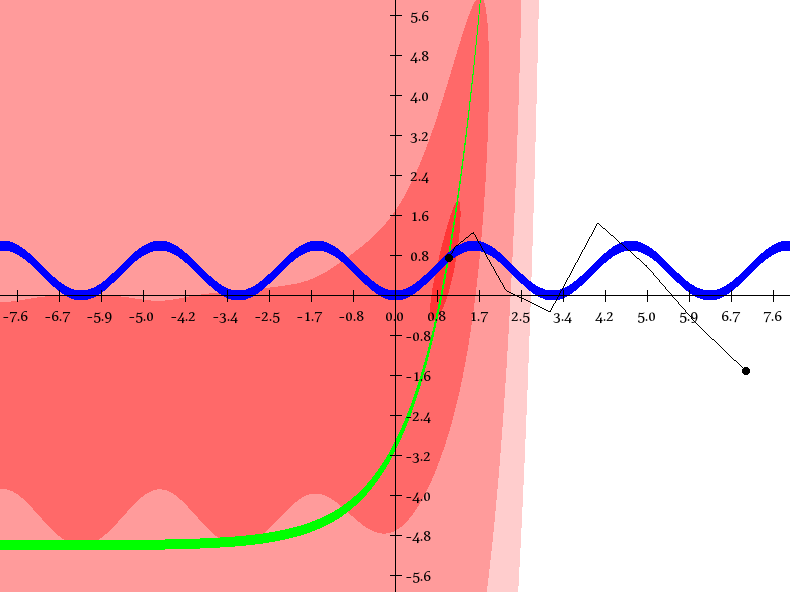
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 4 | 0.01562 | 2.135416666666667e+00 | 3.958333333333332e-01 | 4.72e+00 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 3 | 0.01562 | 1.833333334065087e+00 | 1.666666670356387e-01 | 4.66e+00 |

**Тестирование на тригонометрических функциях:**



Начальное приближение: (7, -1.5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 вариант | | | | |
| итерация |  |  |  |  |
| 10 | 1.00000 | 1.057610204424534e+00 | 7.589627631248483e-01 | 1.37e-08 |
| 2 вариант | | | | |
| итерация |  |  |  |  |
| 10 | 1.00000 | 1.057610204424534e+00 | 7.589627631248483e-01 | 1.37e-08 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 16 | 1.00 | 1.057612469384893e+00 | 7.589682146616747e-01 | 8.38e-06 |
| 6 вариант | | | | |
| итерация |  |  |  |  |
| 10 | 0.10 | 1.057610557810077e+00 | 7.589637043514670e-01 | 1.28e-06 |

Движение метода при h=1.0 Движение метода при h=0.1

# Выводы

1. С помощью метода Ньютона не всегда можно получить решение системы нелинейных уравнений. Это происходит в том случае, если в процессе решения мы получаем вырожденную СЛАУ, например, если начальное приближение задано на оси, соединяющей центры окружностей.
2. Если у системы нелинейных уравнений нет решения, то результатом является некоторая точка, лежащая между графиками функций.
3. Если у системы нелинейных уравнений имеется несколько решений, то результатом является точка, расположенная ближе всего к точке начального приближения.
4. У трех попарно пересекающихся прямых решение всегда сходится в одну точку, независимо от начального приближения.
5. Численный метод, при котором уравнения, содержащие минимальные абсолютные значения Fk, исключаются из системы, практически не отличается от аналитического, что говорит о правильности работы программы.
6. От выбора шага зависит скорость сходимости решения – чем больше шаг, тем быстрее находится решение системы нелинейных уравнений.

# Код программы

Файлmain.cpp**:**

#include <iostream>

#include <fstream>

#include <vector>

#include "NewtonsSolver.h"

#include "GraphicDrawer.h"

using namespace std;

using NewtonsSolver = Newtons::NewtonsSolver;

/// <summary>

/// Получение функций F\_i

/// </summary>

/// <param name="funcNum"> - номер функции для вызова;</param>

/// <param name="x"> - параметры функции</param>

/// <returns>Значение функции в точке [x]</returns>

double F(size\_t funcNum, const std::vector<double>& x) {

switch (funcNum)

{

//case 0: return pow((x[0] - 2), 2) + pow((x[1]), 2) - 4;

//case 1: return pow((x[0] + 2), 2) + pow((x[1]), 2) - 4;

//

//case 0: return sin(x[0])\*sin(x[0]) - x[1];

//case 1: return 2\*exp(x[0]) - x[1] - 5;

//case 2: return x[0] \* x[0] + pow((x[1] - 2), 2) - 4;

case 0: return x[1] + x[0] - 4;

case 1: return x[1] - 2 \* x[0];

case 2: return x[1] - x[0] + 4;

default: throw std::runtime\_error("Неправильный номер функции F");

}

}

/// <summary>

/// Получение производных функций F\_i

/// </summary>

/// <param name="funcNum"> - номер функции для выбора производной;</param>

/// <param name="varNum"> - номер параметра, по которому берётся производная;</param>

/// <param name="x"> - параметры функции</param>

/// <returns>Значение функции в точке [x]</returns>

double div\_F(size\_t funcNum, size\_t varNum, const std::vector<double>& x) {

switch (funcNum)

{

case 0:

{

switch (varNum)

{

case 0:

return 2\*sin(x[0])\*cos(x[0]);

case 1:

return -1;

default: throw std::runtime\_error("Неправильный номер параметра функции");

}

}

case 1:

{

switch (varNum)

{

case 0:

return 2\*exp(x[0]);

case 1:

return -1;

default: throw std::runtime\_error("Неправильный номер параметра функции");

}

}

//case 2:

//{

// switch (varNum)

// {

// case 0:

// return 2 \* x[0];

// case 1:

// return 2 \* x[1] - 4;

// default: throw std::runtime\_error("Неправильный номер параметра функции");

// }

//}

default: throw std::runtime\_error("Неправильный номер функции F");

}

}

double div\_F\_numeric(size\_t funcNum, size\_t varNum, const std::vector<double>& vars) {

static double step = 1e-5;

static vector<double> r;

static vector<double> l;

r = vars;

l = vars;

r[varNum] += step;

l[varNum] -= step;

return (F(funcNum, r) - F(funcNum, l)) / (2.0 \* step);

}

int main() {

setlocale(LC\_ALL, "ru-RU");

constexpr size\_t varCount = 2;

constexpr size\_t funcCount = 3;

auto solver = NewtonsSolver(varCount, funcCount, F, div\_F\_numeric);

NewtonsSolver::TraceVector traceVector;

solver.EnableTracing(traceVector);

double eps;

vector<double> x = { 5.0, 3.0 };

int a = solver.Solve(x, eps, true);

cout << "Полученное решение:";

for (auto& el : x)

{

cout << format("{:23.15e}", el);

}

cout << "\n\n";

// if (true) - рисовать графики 😎

// if (false) - не рисовать графики 🙃

if (true)

{

constexpr size\_t width = 800, height = 600;

constexpr float scale = 0.035f;

constexpr float nearToFunc = 0.1f;

sf::Vector2f midPoint{ width / 2.0, height / 2.0 };

Newtons::GraphicDrawer drawer(width, height, L"Графики функций и движения метода", scale);

// Рисует функции, тепловое поле невязки и оси координат

drawer.DrawAll(18, 15, nearToFunc, funcCount, F);

// Рисует движение точки х в процессе решения метода

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

{

auto beginPoint = Newtons::GetCoordinatesIJ(sf::Vector2f(traceVector[i].prevX[0], traceVector[i].prevX[1]), midPoint, scale);

auto endPoint = Newtons::GetCoordinatesIJ(sf::Vector2f(traceVector[i].X[0], traceVector[i].X[1]), midPoint, scale);

sf::Vertex line[] = {

sf::Vertex(beginPoint, sf::Color::Blue),

sf::Vertex(endPoint, sf::Color::Blue)

};

drawer.window.draw(line, 2, sf::Lines);

}

// Рисует точку начала движения метода

sf::CircleShape startPoint(4);

auto startCirclePosition = Newtons::GetCoordinatesIJ(sf::Vector2f(traceVector[0].prevX[0], traceVector[0].prevX[1]), midPoint, scale);

startCirclePosition.x -= 4;

startCirclePosition.y -= 4;

startPoint.setPosition(startCirclePosition);

startPoint.setFillColor(sf::Color::Blue);

drawer.window.draw(startPoint);

// Рисует точку конца движения метода

sf::CircleShape endPoint(4);

auto endCirclePosition = Newtons::GetCoordinatesIJ(sf::Vector2f(traceVector[traceVector.Size()-1].X[0], traceVector[traceVector.Size() - 1].X[1]), midPoint, scale);

endCirclePosition.x -= 4;

endCirclePosition.y -= 4;

endPoint.setPosition(endCirclePosition);

endPoint.setFillColor(sf::Color::Blue);

drawer.window.draw(endPoint);

drawer.window.display();

drawer.AwaitCloseSync();

}

}

Файл GraphicDrawer.h:

#pragma once

#include <SFML/Graphics.hpp> // Сторонняя графическая библиотека SFML

namespace Newtons {

using namespace sf;

std::vector<sf::Color> functionColors = {

sf::Color::Blue,

sf::Color::Green,

sf::Color::Magenta,

sf::Color::Cyan,

sf::Color::Yellow,

};

inline sf::Vector2f GetCoordinatesXY(size\_t i, size\_t j, sf::Vector2f midPoint, float scale) {

sf::Vector2f point;

point.x = (i - midPoint.x) \* scale;

point.y = -(j - midPoint.y) \* scale;

return point;

}

inline sf::Vector2f GetCoordinatesIJ(sf::Vector2f xy, sf::Vector2f midPoint, float scale) {

sf::Vector2f point;

point.x = (xy.x) / scale + midPoint.x;

point.y = -(xy.y) / scale + midPoint.y;

return point;

}

inline double GetNormF(double x, double y, size\_t funcCount, std::function<double(size\_t, const std::vector<double>&)> functions) {

double res = 0;

double t;

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

{

t = functions(i, { x,y });

res += t \* t;

}

return std::sqrt(res);

}

class GraphicDrawer {

private:

std::size\_t \_width;

std::size\_t \_height;

Vector2f \_midPoint;

std::wstring \_title;

float \_scale = 1;

public:

GraphicDrawer(std::size\_t width, std::size\_t height, std::wstring title, float scale) {

\_scale = scale;

\_width = width;

\_height = height;

\_title = title;

\_midPoint = Vector2f(width / 2.0f, height / 2.0f);

window.create(VideoMode(\_width, \_height), \_title);

//window.setVerticalSyncEnabled(true);

window.clear(Color::White);

}

public:

RenderWindow window;

void DrawCoodrLines(size\_t xDivisorsCount, size\_t yDivisorsCount) {

// Отрисовка главных осей

Vertex xLine[] = {

Vertex(Vector2f(\_midPoint.x, 0.f), Color::Black),

Vertex(Vector2f(\_midPoint.x, window.getSize().y \* 1.f), Color::Black)

};

Vertex yLine[] = {

Vertex(Vector2f(0.f, \_midPoint.y), Color::Black),

Vertex(Vector2f(window.getSize().x \* 1.f, \_midPoint.y), Color::Black)

};

window.draw(xLine, 2, sf::Lines);

window.draw(yLine, 2, sf::Lines);

// крупные разделители, каждые 2 значения X и Y

if (xDivisorsCount % 2 == 0) xDivisorsCount++;

if (yDivisorsCount % 2 == 0) yDivisorsCount++;

Font font;

if (!font.loadFromFile("Kelvinch-Roman.otf"))

{

throw std::runtime\_error("Error when loading font");

}

Text text;

text.setFont(font);

text.setFillColor(Color::Black);

text.setCharacterSize(15);

int xStep = (\_width / xDivisorsCount);

for (int i = -(static\_cast<int>(xDivisorsCount) / 2); i <= static\_cast<int>(xDivisorsCount) / 2; i++)

{

Text text;

text.setFont(font);

text.setFillColor(Color::Black);

text.setCharacterSize(15);

Vertex line[] = {

Vertex(Vector2f(\_midPoint.x + i \* xStep, \_midPoint.y - 6), Color::Black),

Vertex(Vector2f(\_midPoint.x + i \* xStep, \_midPoint.y + 6), Color::Black)

};

window.draw(line, 2, sf::Lines);

text.setPosition(Vector2f(\_midPoint.x + i \* xStep - 15, \_midPoint.y + 10));

text.setString(std::format("{: 4.1f}", GetCoordinatesXY(\_midPoint.x + i \* xStep, 0, \_midPoint, \_scale).x));

window.draw(text);

}

int yStep = (\_height / yDivisorsCount);

for (int i = -(static\_cast<int>(yDivisorsCount) / 2); i <= static\_cast<int>(yDivisorsCount) / 2; i++)

{

if (i == 0) continue;

Vertex line[] = {

Vertex(Vector2f(\_midPoint.x - 6, \_midPoint.y + i \* yStep), Color::Black),

Vertex(Vector2f(\_midPoint.x + 6, \_midPoint.y + i \* yStep), Color::Black)

};

window.draw(line, 2, sf::Lines);

text.setPosition(Vector2f(\_midPoint.x + 10, \_midPoint.y + i \* yStep - 10));

text.setString(std::format("{: 4.1f}", GetCoordinatesXY(0, \_midPoint.y + i \* yStep, \_midPoint, \_scale).y));

window.draw(text);

}

}

void DrawHeatMap(VertexArray& points, std::size\_t funcCount, std::function<double(std::size\_t, const std::vector<double>&)> F) {

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

{

for (std::size\_t j = 0; j < \_height; j++)

{

auto point = GetCoordinatesXY(i, j, \_midPoint, \_scale);

double norm = GetNormF(point.x, point.y, funcCount, F);

if (norm >= 25)

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), sf::Color::White);

}

else if (norm > 15.0)

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), sf::Color(255, 0, 0, 50));

}

else if (norm > 5.0)

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), sf::Color(255, 0, 0, 100));

}

else if (norm > 1)

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), sf::Color(255, 0, 0, 150));

}

else if (norm > 0.1)

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), sf::Color(255, 0, 0, 200));

}

else

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), sf::Color(255, 0, 0, 230));

}

}

}

}

void DrawGraphics(VertexArray& points, float nearToFunc, std::size\_t funcCount, std::function<double(std::size\_t, const std::vector<double>&)> F) {

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

{

for (std::size\_t j = 0; j < \_height; j++)

{

auto point = Newtons::GetCoordinatesXY(i, j, \_midPoint, \_scale);

for (int fn = 0; fn < funcCount; fn++)

{

if (abs(F(fn, { point.x, point.y })) < nearToFunc)

{

points[i \* \_height + j] = sf::Vertex(sf::Vector2f(i, j), functionColors[fn]);

break;

}

}

}

}

}

void DrawAll(size\_t xDivCount, size\_t yDivCount, float nearToFunc, std::size\_t funcCount, std::function<double(std::size\_t, const std::vector<double>&)> F) {

VertexArray points(Points, \_width \* \_height);

DrawHeatMap(points, funcCount, F);

DrawGraphics(points, nearToFunc, funcCount, F);

window.draw(points);

DrawCoodrLines(xDivCount, yDivCount);

}

void AwaitCloseSync() {

while (window.isOpen())

{

Event event;

while (window.waitEvent(event))

{

if (event.type == Event::Closed)

{

window.close();

}

}

}

}

};

}

Файл NewtonSolvers.h:

#pragma once

#include "LU solver/headers/ProfileLU.h"

#include <cmath>

#include <functional>

#include <algorithm>

#include <iostream>

#include <format>

namespace Newtons {

namespace Vec {

inline double Scalar(const std::vector<double>& l, const std::vector<double>& r);

inline double Norm(const std::vector<double>& vec);

// ans = left + coef \* right

inline void AddVec(

const std::vector<double>& left,

double coef,

const std::vector<double>& right,

std::vector<double>& ans)

{

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

{

ans[i] = left[i] + coef \* right[i];

}

}

}

class NewtonsSolver {

public:

struct TraceElement {

int iterationNum{};

std::vector<double> prevX;

std::vector<double> X;

std::vector<double> dX;

double eps{};

double prevEps{};

TraceElement() noexcept {}

TraceElement(const TraceElement& elem) noexcept {

iterationNum = elem.iterationNum;

prevX = (elem.prevX);

X = (elem.X);

dX = (elem.dX);

eps = elem.eps;

prevEps = elem.prevEps;

}

TraceElement(TraceElement&& elem) noexcept {

iterationNum = elem.iterationNum;

prevX = std::move(elem.prevX);

X = std::move(elem.X);

dX = std::move(elem.dX);

eps = elem.eps;

prevEps = elem.prevEps;

}

};

class TraceVector {

private:

std::vector<TraceElement> \_traceVec;

public:

TraceElement& operator[](size\_t ind) {

return \_traceVec[ind];

}

void Push(TraceElement&& elem) {

\_traceVec.push\_back(elem);

}

void Push(TraceElement& elem) {

\_traceVec.push\_back(elem);

}

void Clear() {

\_traceVec.clear();

}

size\_t Size() const {

return \_traceVec.size();

}

void Resize(size\_t newSize) {

return \_traceVec.resize(newSize);

}

};

private:

// Переменные для матриц и векторов, используемых в солвере

ProfileMatrix \_profMat;

Matrix \_mat;

std::vector<double> \_F;

std::vector<double> \_x;

std::vector<double> \_dx;

std::vector<double> \_dx\_trim;

// Переменные для хранения фунцкий и их дифференциалов

std::function<double(std::size\_t, const std::vector<double>&)> \_functions;

std::function<double(std::size\_t, const std::size\_t, std::vector<double>&)> \_differentials;

// Переменные для хранения количества функций и переменных

size\_t \_funcCount;

size\_t \_varCount;

// Вектор для формирования масок

std::vector<std::pair<size\_t, double>> \_pairVec;

// Для обозначения статуса маски

enum class MaskType {

None,

MoreVars,

MoreFuncs

};

MaskType \_maskType = MaskType::None;

// Маска для исключения лишних переменных или функций

std::vector<bool> \_mask;

// Указатель на массив для трассировки метода (получение результата вычислений на каждом шагу)

TraceVector\* \_traceVector;

public:

/// <summary>

/// Инициализатор солвера методом Ньютона

/// </summary>

/// <param name="variableCount"> - количество переменных в системе</param>

/// <param name="funcCount"> - количество функций в системе</param>

/// <param name="functions"> - функция, в которой заданы все функции F\_j системы</param>

/// <param name="differentials"> - функция, в которой заданы все дифференциалы системы. </param>

NewtonsSolver(

size\_t variableCount,

size\_t funcCount,

std::function<double(size\_t, const std::vector<double>&)> functions,

std::function<double(size\_t, size\_t, const std::vector<double>&)> differentials)

{

\_varCount = variableCount;

\_funcCount = funcCount;

\_x.resize(variableCount);

\_dx.resize(variableCount);

\_functions = functions;

\_differentials = differentials;

size\_t minSize = std::min(variableCount, funcCount);

\_mat.resize(minSize, minSize);

\_F.resize(minSize);

if (variableCount != funcCount)

{

if (variableCount > funcCount)

{

\_dx\_trim.resize(funcCount);

}

\_mask.resize(std::max(variableCount, funcCount));

\_pairVec.resize(std::max(variableCount, funcCount));

}

}

private:

// Определяет тип маски и получает маску, меняет \_mask и \_maskType

inline void \_GetMask();

// Находит матрицу Якоби с учётом маски и записывает её в \_mat

void \_GetJacobi();

// Находит вектор функций F для решения системы

void \_GetF();

// Находит норму вектора из значений фунций F в точке x

double \_GetNormF(const std::vector<double>& x) {

double res = 0;

double t;

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

{

t = \_functions(i, x);

res += t \* t;

}

return std::sqrt(res);

}

public:

// Минимальное значение невязки вектора решения

double minEps = 1e-5;

// Максимальное число итераций

int maxIter = 100;

// Критический коэффициент, после которого метод завершается с ошибкой сходимости

// По умолчанию равен числу, соответствующему 6 дроблениям коэффициента на 2

double criticalCoef = 1.0 / (1 << 6);

// Метод для решения системы нелинейных уравнений

// - init\_x - начальное приближение, в том числе итоговое решение

// - eps - полученная невязка решения

// Возможный возврат:

// -1 - ошибка сходимости (при любом допустимо возможном шаге невязка возрастает)

// -2 - выход по превышению числа итераций

// Положительное число - число итераций сходимости метода

int Solve(std::vector<double>& init\_x, double& eps, const bool debugOutput = false);

class TraceVector;

void EnableTracing(TraceVector& traceVector) {

\_traceVector = &traceVector;

traceVector.Clear();

}

};

}

Файл NewtonsSolver.cpp:

#include "NewtonsSolver.h"

namespace Newtons {

void NewtonsSolver::\_GetMask() {

if (\_varCount == \_funcCount)

{

\_maskType = MaskType::None;

}

else if (\_varCount < \_funcCount)

{

size\_t delta = \_funcCount - \_varCount;

// Найдём для каждой функции их абсолютное значение в точке х

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

{

\_pairVec[i].first = i;

\_pairVec[i].second = std::abs(\_functions(i, \_x));

}

// Сортируем эту последовательность по возрастанию значений

std::sort(\_pairVec.begin(), \_pairVec.end(), [](const std::pair<size\_t, double>& l, const std::pair<size\_t, double>& r)

{

return l.second < r.second;

}

);

// Исключаем индексы первых delta элементов из расчётов (помечаем их нулями в маске)

size\_t i;

for (i = 0; i < delta; i++)

{

\_mask[\_pairVec[i].first] = false;

}

// Остальные помечаем единицами

for (i; i < \_funcCount; i++)

{

\_mask[\_pairVec[i].first] = true;

}

\_maskType = MaskType::MoreFuncs;

}

else // \_varCount > \_funcCount

{

size\_t delta = \_varCount - \_funcCount;

// Найдём для каждой переменной максимальное абсолютное значение среди прозводных функций

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

{

\_pairVec[i].first = i;

// Перебираем производные всех функций, находим максимум для i переменной

double a = 0;

for (size\_t k = 0; k < \_funcCount; k++)

{

a = std::max(a, std::abs(\_differentials(k, i, \_x)));

}

\_pairVec[i].second = a;

}

// Сортируем эту последовательность по возрастанию значений

std::sort(\_pairVec.begin(), \_pairVec.end(), [](const std::pair<size\_t, double>& l, const std::pair<size\_t, double>& r)

{

return l.second < r.second;

}

);

// Исключаем индексы первых delta элементов из расчётов (помечаем их нулями в маске)

size\_t i;

for (i = 0; i < delta; i++)

{

\_mask[\_pairVec[i].first] = false;

}

// Остальные помечаем единицами

for (i; i < \_varCount; i++)

{

\_mask[\_pairVec[i].first] = true;

}

\_maskType = MaskType::MoreVars;

}

}

void NewtonsSolver::\_GetJacobi() {

switch (\_maskType)

{

case MaskType::None:

{

for (size\_t func = 0; func < \_funcCount; func++)

{

for (size\_t var = 0; var < \_varCount; var++)

{

\_mat(func, var) = \_differentials(func, var, \_x);

}

}

}

break;

case MaskType::MoreVars:

{

for (size\_t func = 0; func < \_funcCount; func++)

{

for (size\_t var = 0, dvar = 0; var < \_varCount; var++)

{

// Считаем только для тех переменных, которые не исключены маской из расчётов

if (\_mask[var])

{

\_mat(func, dvar) = \_differentials(func, var, \_x);

dvar++;

}

}

}

}

break;

case MaskType::MoreFuncs:

{

for (size\_t func = 0, dfunc = 0; func < \_funcCount; func++)

{

// Считаем только те функции, которые не исключены маской из расчётов

if (\_mask[func])

{

for (size\_t var = 0; var < \_varCount; var++)

{

\_mat(dfunc, var) = \_differentials(func, var, \_x);

}

dfunc++;

}

}

}

break;

}

}

void NewtonsSolver::\_GetF() {

if (\_maskType == MaskType::MoreFuncs)

{

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

{

// Считаем только те функции, которые не исключены из расчётов маской

if (\_mask[i])

{

\_F[k] = -\_functions(i, \_x);

k++;

}

}

}

else

{

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

{

\_F[i] = -\_functions(i, \_x);

}

}

}

// Метод для решения системы нелинейных уравнений

// - init\_x - начальное приближение, в том числе итоговое решение

// - eps - полученная невязка решения

// Возможный возврат:

// - [-1] - ошибка сходимости (при любом допустимо возможном шаге невязка возрастает)

// - [-2] - выход по превышению числа итераций

// - [Положительное число] - число итераций сходимости метода

int NewtonsSolver::Solve(std::vector<double>& init\_x, double& eps, const bool debugOutput) {

std::swap(init\_x, \_x);

eps = \_GetNormF(\_x);

int it;

for (it = 1; it <= maxIter && eps > minEps; it++)

{

\_GetMask();

\_GetJacobi();

\_profMat.MakeFromMatrix(\_mat);

\_profMat.LUdecompose();

\_GetF();

if (\_dx\_trim.size() != 0)

{

LU::ProfileSolver::Solve(\_profMat, \_dx\_trim, \_F);

// Переписываем обрезанный вектор \_dx\_trim в полноценный \_dx

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

{

if (\_mask[i])

{

\_dx[i] = \_dx\_trim[k];

k++;

}

else

{

\_dx[i] = 0;

}

}

}

else

{

LU::ProfileSolver::Solve(\_profMat, \_dx, \_F);

}

double coef = 2;

double newEps = eps;

while (eps <= newEps && coef > criticalCoef)

{

coef /= 2;

Vec::AddVec(\_x, coef, \_dx, init\_x);

newEps = \_GetNormF(init\_x);

}

if (coef <= criticalCoef)

{

init\_x = \_x;

if (debugOutput)

{

std::cout << "Выход по ошибке сходимости\n";

std::cout << std::format("\tКоэф. \\beta:{:15.5f}\n\n", coef);

}

return -1; // Ошибка - вылет по сходимости метода

}

if (\_traceVector)

{

TraceElement elem;

elem.iterationNum = it;

elem.prevX = \_x;

elem.X = init\_x;

elem.dX = \_dx;

elem.eps = newEps;

elem.prevEps = eps;

\_traceVector->Push(std::move(elem));

}

\_x = init\_x;

eps = newEps;

if (debugOutput)

{

std::cout << std::format("Текущая итерация: {:>4}, текущая невязка: {:>10.2e}\n\tТекущая точка:", it, eps);

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

{

std::cout << std::format("{0:15.5f}", \_x[i]);

}

std::cout << "\n\tВектор сдвига:";

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

{

std::cout << std::format("{0:15.5f}", \_dx[i] \* coef);

}

std::cout << std::format("\n\tКоэф. \\beta:{:15.5f}", coef);

std::cout << "\n\n";

}

}

if (eps > minEps)

{

if (debugOutput)

std::cout << "Выход по превышению числа итераций\n\n";

return -2;

}

if (debugOutput)

std::cout << "Выход по достижению требуемой невязки решения\n\n";

return it - 1;

}

}

double Newtons::Vec::Scalar(const std::vector<double>& l, const std::vector<double>& r) {

if (l.size() != r.size()) throw std::runtime\_error("Size of vectors not same.");

double res = 0.0;

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

{

res += l[i] \* r[i];

}

return res;

}

double Newtons::Vec::Norm(const std::vector<double>& vec) {

return std::sqrt(Scalar(vec, vec));

}

Файл ProfileMatrix.h:

#pragma once

#include "Matrix.h"

#include <stdexcept>

class ProfileMatrix {

public:

enum class ProfileMatrixType

{

Empty,

ProfileOnly,

LUdecomposed

};

public:

std::vector<double> diag;

std::vector<std::size\_t> ia;

std::vector<double> al;

std::vector<double> au;

ProfileMatrixType type = ProfileMatrixType::Empty;

public:

ProfileMatrix() {}

ProfileMatrix(std::size\_t diagSize, std::size\_t aSize)

: diag(diagSize), ia(diagSize+1), al(aSize), au(aSize) { }

public:

// Return size of matrix (number of dimention of matrix, size of diag)

inline std::size\_t Size(void) const {

return diag.size();

}

// Return number of elements in lower or upper triangles of matrix (size of al)

inline std::size\_t Asize(void) const {

return al.size();

}

bool isEmpty() const { return type == ProfileMatrixType::Empty; }

bool isLU() const { return type == ProfileMatrixType::LUdecomposed; }

void MakeFromMatrix(const Matrix& mat);

void LUdecompose();

};

Файл ProfileMatrix.cpp:

#include "../headers/ProfileMatrix.h"

void ProfileMatrix::MakeFromMatrix(const Matrix& mat) {

if (mat.Cols() != mat.Rows())

throw std::runtime\_error("Bad matrix sizes (matrix should be squared)");

diag.resize(mat.Rows());

ia.resize(mat.Rows() + 1);

std::size\_t asize = 0;

for (std::size\_t i = 1; i < mat.Rows(); i++)

{

bool flag = false;

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

{

if (mat(i, j) != 0 || mat(j, i) != 0)

{

flag = true;

}

if (flag)

{

asize++;

}

}

}

al.resize(asize); au.resize(asize);

int s = 0;

for (std::size\_t i = 0; i < mat.Rows(); i++)

{

diag[i] = mat(i, i);

ia[i] = s;

bool flag = false;

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

{

if (!flag && (mat(i, j) != 0 || mat(j, i) != 0))

{

flag = true;

}

if (flag)

{

al[s] = mat(i, j);

au[s] = mat(j, i);

s++;

}

}

}

ia[mat.Rows()] = s;

type = ProfileMatrixType::ProfileOnly;

}

void ProfileMatrix::LUdecompose() {

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

{

size\_t i0 = ia[i];

size\_t i1 = ia[i + 1];

size\_t j = i - (ia[i + 1] - ia[i]);

double bdi = 0;

for (size\_t k = ia[i]; k < ia[i + 1]; k++, j++)

{

size\_t ki = ia[i];

size\_t kj = ia[j];

size\_t dif = k - ia[i] - ia[j + 1] + ia[j];

if (dif < 0)

kj -= dif;

else

ki += dif;

double bal = 0;

double bau = 0;

for (ki; ki < k; ki++, kj++)

{

bal += al[ki] \* au[kj];

bau += au[ki] \* al[kj];

}

al[k] = al[k] - bal;

au[k] = au[k] - bau;

au[k] = au[k] / diag[j];

bdi += al[k] \* au[k];

}

diag[i] -= bdi;

}

type = ProfileMatrixType::LUdecomposed;

}

Файл ProfileLU.h:

#pragma once

#include "ProfileMatrix.h"

namespace LU {

class ProfileSolver {

private:

ProfileSolver() {}

static void \_Direct(const ProfileMatrix& mat, std::vector<double>& x);

static void \_Reverse(const ProfileMatrix& mat, std::vector<double>& x);

public:

static void Solve(const ProfileMatrix& mat, std::vector<double>& x, const std::vector<double>& F) {

if (!mat.isLU()){

throw std::runtime\_error("Profile matrix is not LU decomposed, that ProfileSolver needs.");

}

x = F;

\_Direct(mat, x);

\_Reverse(mat, x);

}

};

}

Файл ProfileLU.cpp:

#include "../headers/ProfileLU.h"

void LU::ProfileSolver::\_Direct(const ProfileMatrix& mat, std::vector<double>& x) {

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

{

size\_t j = i - (mat.ia[i + 1] - mat.ia[i]);

double sum = 0;

for (size\_t k = mat.ia[i]; k < mat.ia[i + 1]; k++, j++)

{

sum += x[j] \* mat.al[k];

}

x[i] = (x[i] - sum) / mat.diag[i];

}

}

void LU::ProfileSolver::\_Reverse(const ProfileMatrix& mat, std::vector<double>& x) {

for (size\_t i = mat.Size(); i > 0; )

{

--i;

size\_t j = i - (mat.ia[i + 1] - mat.ia[i]);

for (size\_t k = mat.ia[i]; k < mat.ia[i + 1]; k++, j++)

{

x[j] -= x[i] \* mat.au[k];

}

}

}

Файл Matrix.h:

#pragma once

#include <vector>

class Matrix {

public:

std::vector<std::vector<double>> elems;

public:

Matrix() {}

Matrix(size\_t rows, size\_t cols) {

resize(rows, cols);

}

Matrix(std::vector<std::vector<double>>&& initMat) {

elems = std::move(initMat);

}

Matrix(const std::vector<std::vector<double>>& initMat) {

elems.resize(initMat.size());

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

{

elems[i].resize(initMat.size());

for (size\_t k = 0; k < elems[i].size(); k++)

{

elems[i][k] = initMat[i][k];

}

}

}

Matrix(const Matrix& initMat) : Matrix(initMat.elems) {}

public:

inline std::size\_t Cols(void) const {

if (Rows() == 0) return 0;

return elems[0].size();

}

inline std::size\_t Rows(void) const {

return elems.size();

}

double& operator() (std::size\_t x, std::size\_t y) {

return elems[x][y];

}

double operator() (std::size\_t x, std::size\_t y) const {

return elems[x][y];

}

void resize(std::size\_t rows, std::size\_t cols) {

elems.resize(cols);

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

{

elems[i].resize(rows);

}

}

};