|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования  «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра прикладной математики | | | |
|  | | | |
|  | | | |
| Курсовая работа по курсу | | | |
| «Уравнения математической физики» | | | |
|  | | | |
|  | Группа | ПМ-01 |
|  |  |
| Студент | Самсонов семён |
|  | евгеньевич |
|
|  |
| Новосибирск | | | |

2023

Оглавление

[Теоретическая часть 3](#_Toc135344977)

[Постановка задачи 3](#_Toc135344978)

[Дискретизация по времени 4](#_Toc135344979)

[Описание разработанной программы 7](#_Toc135344980)

[Структуры данных, используемые для задания расчётной области и конечноэлементной сетки 7](#_Toc135344981)

[Структура основных модулей программы 7](#_Toc135344982)

[Тестирование программы 9](#_Toc135344983)

[Базовые тесты для проверки работоспособности программы 9](#_Toc135344984)

[Тестирование на порядок аппроксимации 10](#_Toc135344985)

[Тестирование на порядок сходимости 11](#_Toc135344986)

[Выводы 12](#_Toc135344987)

[Код программы 13](#_Toc135344988)

# Теоретическая часть

## Постановка задачи

Целью курсового проекта является решение гиперболической задачи с эллиптическим оператором из курсового проекта по численным методам за 5 семестр с использованием явной трёхслойной схемы по времени. Решаемое уравнение имеет следующий общий вид:

Область интегрирования: ,  
Граница интегрирования: ,  
Краевые условия:  
 - Первые: ,  
 - Вторые: ,  
 - Третьи: ,  
 – коэффициент диффузии,  
 – коэффициент теплообмена,  
Начальные условия:  
 - Первые: ,  
 - Вторые: либо .

В условиях текущего варианта задания, решение должно происходить в полярных координатах, поэтому уравнение примет следующий вид:

а матрицы массы и жёсткости и вектор правой части будут считаться следующим образом:

здесь – базисные функции конечномерного пространства, определяемые следующим образом:

а функции и определяются в свою очередь следующим способом:

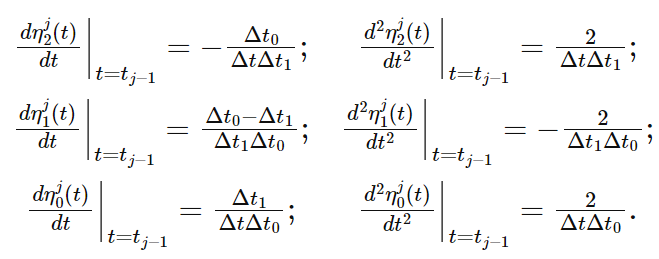
## Дискретизация по времени

Для дискретизации исходного решения по времени на сетке с неравномерным шагом удобно разложить решение на несколько слоёв (применимо к данному варианту, на 3 слоя) с помощью базисных квадратичных функций по времени :

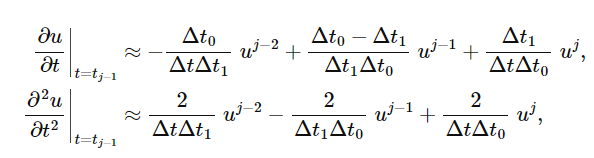
где функции – базисные квадратичные полиномы Лагранжа, которые могут быть записаны в виде:

где:

Чтобы получить явную схему аппроксимации исходного уравнения (1) с неравномерным шагом по времени, вычислим значения первой и второй производных функций (4.1) – (4.3):



С учётом уравнения (3) первая и вторая производные функции по времени аппроксимируются так:



и тогда аппроксимация дифференциального уравнения (1) по времени может быть записана в виде:

Конечномерная аппроксимация краевой задачи для уравнения (5) приведёт к матричному уравнению вида

которое фактически является СЛАУ для вектора с матрицей и вектором правой части следующего вида:

В данных уравнениях, матрицы и являются матрицами массы из прошлой курсовой работы, за исключением, что вместо коэффициента используются коэффициенты и соответственно. Соответственно, чтобы найти их, можно воспользоваться следующими формулами:

Использование трёхслойной явной схемы подразумевает, что значения и изначально известны. Первое значение () задаётся начальным условием, второе же значение можно находить двумя вариантами:

1. Используя начальное условие, аналогичное первому:
2. Используя приближённое значение на временном слое:

В данной работе используется первый вариант.

# Описание разработанной программы

## Структуры данных, используемые для задания расчётной области и конечноэлементной сетки

Для задания расчётной области используются следующие структуры:

1. Структура, хранящая координаты узлов, node:

struct Node {

   double r = 0.0;

   double phi = 0.0;

};

1. Структура S1\_node, хранящая узлы, на которых заданы первые краевые условия:

struct S1\_node {

   int node = 0;

   int funcNum = 0;

};

Для задания конечноэлементной сетки используется структура Rectangle:

struct Rectangle {

   int a = 0;

   int b = 0;

   int c = 0;

   int d = 0;

   int regionNum = 0;

};

## Структура основных модулей программы

Программа состоит из нескольких модулей:

1. Модуль SparseMatrix – определяет класс разреженных строчно-столбцовых матриц, а также основные операции с ними: умножение на число, на вектор, сложение, копирование, вывод в удобном для человеческого восприятия виде.
2. Модуль LU – определяет класс LU-разложенных разреженных строчно-столбцовых матриц и некоторые операции над ними: умножение вектора на L- или U-компоненту, транспонирование компонент, решение систем линейных уравнений относительно компонент и заданного вектора правой части.
3. Модуль IterSolvers – определяет итерационные методы решения систем линейных уравнений на основе разреженных матриц. Содержит в себе 6 методов решения СЛАУ:
   1. Метод смежных градиентов для несимметричных матриц;
   2. Метод смежных градиентов с диагональным предобуславливанием;
   3. Метод смежных градиентов с LU-предобуславливанием;
   4. Метод ЛОС;
   5. Метод ЛОС с диагональным предобуславливанием;
   6. Метод ЛОС с LU-предобуславливанием;
4. Модуль gaussian\_quadrature – определяет методы численного интегрирования методом Гаусса для одно- и двухмерных функций. Включает в себя метод Гаусса по двум точкам и по четырём точкам.
5. Основной модуль программы, содержащий в себе:
   1. метод генерации портрета разреженной матрицы на основе входных данных;
   2. Методы построения локальных матриц, применяя методы численного интегрирования;
   3. Метод добавления локальных матриц к глобальной;
   4. Метод учёта первых краевых условий.

# Тестирование программы

Для составления тестов для проверки работоспособности программы использовалось следующее уравнение:

Пространственная область исследования представляет из себя квадрат размером 6х6 с 9 узлами:

## Базовые тесты для проверки работоспособности программы

1. Тестирование на функции

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | | | |

Результат работы программы:

| Время | норма погрешности |
| --- | --- |
| 0 | 0.00000000000000e+00 |
| 1 | 0.00000000000000e+00 |
| 2 | 1.25607396694702e-15 |
| 3 | 1.33226762955019e-15 |
| 4 | 2.66453525910038e-15 |
| 5 | 3.55271367880050e-15 |
| 6 | 6.21724893790088e-15 |
| 7 | 6.21724893790088e-15 |
| 8 | 4.44089209850063e-15 |
| 9 | 7.10542735760100e-15 |
| 10 | 1.24344978758018e-14 |

1. Тестирование на функции

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | | | |

Результат работы программы:

| Время | норма погрешности |
| --- | --- |
| 0 | 0.00000000000000e+00 |
| 1 | 0.00000000000000e+00 |
| 2 | 3.41111394591414e-15 |
| 3 | 4.44089209850063e-15 |
| 4 | 2.66453525910038e-15 |
| 5 | 0.00000000000000e+00 |
| 6 | 1.77635683940025e-15 |
| 7 | 1.77635683940025e-15 |
| 8 | 3.55271367880050e-15 |
| 9 | 3.55271367880050e-15 |
| 10 | 1.77635683940025e-15 |

## Тестирование на порядок аппроксимации

1. Тестирование на функции

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | | | |

Результат работы программы:

| Время | норма погрешности |
| --- | --- |
| 0 | 0.00000000000000e+00 |
| 1 | 0.00000000000000e+00 |
| 2 | 2.51214793389404e-15 |
| 3 | 0.00000000000000e+00 |
| 4 | 3.55271367880050e-15 |
| 5 | 3.55271367880050e-15 |
| 6 | 0.00000000000000e+00 |
| 7 | 1.42108547152020e-14 |
| 8 | 1.42108547152020e-14 |
| 9 | 1.42108547152020e-14 |
| 10 | 1.42108547152020e-14 |

1. Тестирование на функции

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | | | |

Результат работы программы:

| Время | норма погрешности |
| --- | --- |
| 0 | 0.00000000000000e+00 |
| 1 | 0.00000000000000e+00 |
| 2 | 1.50000000000000e+00 |
| 3 | 3.14441127380945e+00 |
| 4 | 4.44713689876385e+00 |
| 5 | 5.32714427706435e+00 |
| 6 | 5.85763177135487e+00 |
| 7 | 6.14585572273876e+00 |
| 8 | 6.28499903466718e+00 |
| 9 | 6.34146360474597e+00 |
| 10 | 6.35698315117543e+00 |

## Тестирование на порядок сходимости

Формула для подсчёта порядка сходимости:

1. Тестирование на функции

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | | | |

Результаты при :

| Шаг | Норма погрешности |  |
| --- | --- | --- |
| 1 | 1.19756615034468e+01 |  |
| 0.5 | 3.15578759035698e+00 | 3.79482 |
| 0.25 | 8.00329512433713e-01 | 3.94311 |
| 0.125 | 2.00909784092126e-01 | 3.98352 |

# Выводы

1. Использование явной трёхслойной схемы по времени позволяет достичь *второго порядка аппроксимации* по переменной времени. При этом при решении уравнений более высокого порядка в решении появляется значительная погрешность, уменьшающаяся дроблением временной сетки.
2. При тестировании на неполиномиальной функции и дроблении сетки пополам, норма погрешности полученного решения уменьшалась в 4 раза, следовательно, *порядок сходимости явной трёхслойной схемы по времени равен двум*.

# Код программы

Полные исходники программы, в том числе с cmake-файлом, находятся в GitHub-репозитории: <https://github.com/SemafonKA/2d_bilinear_polar_hyperbolic>.

Проект имеет следующую иерархию:

* CMakeLists.txt
* iofiles
  + nodes.txt
  + rectangles.txt
  + s1\_nodes.txt
* lib
  + main.cpp
  + Constants.h
  + gaussian\_quadrature
    - gaussian\_quadrature.h
  + three\_steppers
    - Headers
      * IterSolvers.h
      * LU.h
      * SparseMatrix.h
    - Resourses
      * IterSolvers.cpp
      * LU.cpp
      * SparseMatrix.cpp

Файл CMakeLists.txt имеет следующее содержание:

cmake\_minimum\_required(VERSION 3.0.0)

project(

   2d\_bilinear\_polar\_hyperbolic

   VERSION 0.1.0

   LANGUAGES CXX

)

add\_library(

   IterSolvers

   STATIC

   ./lib/three\_steppers/Resourses/IterSolvers.cpp

)

add\_library(

   LU

   STATIC

   ./lib/three\_steppers/Resourses/LU.cpp

)

add\_library(

   SparseMatrix

   STATIC

   ./lib/three\_steppers/Resourses/SparseMatrix.cpp

)

add\_executable(${PROJECT\_NAME} ./lib/main.cpp)

target\_link\_libraries(

   ${PROJECT\_NAME}

   IterSolvers

   LU

   SparseMatrix

)

set(CPACK\_PROJECT\_NAME ${PROJECT\_NAME})

set(CPACK\_PROJECT\_VERSION ${PROJECT\_VERSION})

if(MSVC)

   set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} /std:c++20")

else()

   set(CMAKE\_CXX\_STANDARD 20)

endif(MSVC)

include(CPack)

#### Файл main.cpp:

#include <fstream>

#include <iostream>

#include <vector>

#include "gaussian\_quadrature/gaussian\_quadrature.h"

#include "three\_steppers/Headers/IterSolvers.h"

// Файл, содержащий в себе пути до файлов, функции f, lambda и gamma

#include "Constants.h"

using namespace std;

#pragma region GLOBAL\_OBJECTS

// Глобальная разреженная матрица системы

SparseMatrix global\_mat;

// Глобальный вектор системы

vector<double> global\_b;

vector<double> global\_d;

// Массив прямоугольников

vector<Rectangle> rectangles;

// Массив узлов

vector<Node> nodes;

// Массив сопоставления узлов и первых краевых

vector<S1\_node> s1\_nodes;

#pragma endregion GLOBAL\_OBJECTS

std::vector<double> operator+(const std::vector<double>& left, const std::vector<double>& right) {

   std::vector<double> result(left);

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

      result.at(i) += right.at(i);

   }

   return result;

}

// Функция R билинейного базиса

double func\_R(int ind, double rp, double hr, double r) {

   ind = ind % 2;

   if (ind == 0) {

      return (rp + hr - r) / hr;

   }

   return (r - rp) / hr;

}

// Производная функции R билинейного базиса

double func\_R\_dif(int ind, double rp, double hr, double r) {

   ind = ind % 2;

   if (ind == 0) {

      return -1.0 / hr;

   }

   return 1.0 / hr;

}

// Функция phi билинейного базиса

double func\_Phi(int ind, double phi\_s, double h\_phi, double phi) {

   ind = (ind / 2) % 2;

   if (ind == 0) {

      return (phi\_s + h\_phi - phi) / h\_phi;

   }

   return (phi - phi\_s) / h\_phi;

}

// Производная функции phi билинейного базиса

double func\_Phi\_dif(int ind, double phi\_s, double h\_phi, double phi) {

   ind = (ind / 2) % 2;

   if (ind == 0) {

      return -1.0 / h\_phi;

   }

   return 1.0 / h\_phi;

}

void readDataFromFiles() {

   // Считывание данных для структуры узлов nodes

   auto nodesFile = ifstream(GlobalPaths::nodesPath);

   if (!nodesFile.is\_open()) throw runtime\_error("Не удалось открыть файл " + GlobalPaths::nodesPath);

   int size;

   nodesFile >> size;

   nodes.resize(size);

   for (auto& node : nodes) {

      nodesFile >> node.r >> node.phi;

   }

   nodesFile.close();

   // Считывание данных для структуры прямоугольников rectangles

   auto rectanglesFile = ifstream(GlobalPaths::rectanglesPath);

   if (!rectanglesFile.is\_open()) throw runtime\_error("Не удалось открыть файл " + GlobalPaths::rectanglesPath);

   rectanglesFile >> size;

   rectangles.resize(size);

   for (auto& rect : rectangles) {

      rectanglesFile >> rect.a >> rect.b >> rect.c >> rect.d >> rect.regionNum;

   }

   rectanglesFile.close();

   // Считывание данных для первых краевых условий s1\_nodes

   auto s1\_nodesFile = ifstream(GlobalPaths::s1\_nodesPath);

   if (!s1\_nodesFile.is\_open()) throw runtime\_error("Не удалось открыть файл " + GlobalPaths::s1\_nodesPath);

   s1\_nodesFile >> size;

   s1\_nodes.resize(size);

   for (auto& s1 : s1\_nodes) {

      s1\_nodesFile >> s1.node >> s1.funcNum;

   }

   s1\_nodesFile.close();

}

void generatePortrait() {

   global\_mat.di.resize(nodes.size());

   global\_mat.ig.resize(nodes.size() + 1);

   for (auto& rect : rectangles) {

      const int elems[4] = {rect.a, rect.b, rect.c, rect.d};

      for (int i = 0; i < 4; i++) {

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

            // Если элемент в верхнем прямоугольнике, то скипаем

            if (elems[k] > elems[i]) continue;

            bool isExist = false;

            // Пробегаем по всей строке для проверки, существует ли такой элемент

            for (auto it = global\_mat.ig[elems[i]]; it < global\_mat.ig[elems[i] + 1ll]; it++) {

               if (global\_mat.jg[it] == elems[k]) {

                  isExist = true;

                  break;

               }

            }

            if (!isExist) {

               // Ищем, куда вставить элемент портрета

               auto it = global\_mat.ig[elems[i]];

               while (it < global\_mat.ig[elems[i] + 1ll] && global\_mat.jg[it] < elems[k]) it++;

               // Для вставки нужно взять итератор массива от начала, так что...

               global\_mat.jg.insert(global\_mat.jg.begin() + it, elems[k]);

               // Добавляем всем элементам ig с позиции elems[i]+1 один элемент

               for (auto j = elems[i] + 1; j < global\_mat.ig.size(); j++) global\_mat.ig[j]++;

            }

         }

      }

   }

   global\_mat.ggl.resize(global\_mat.jg.size());

   global\_mat.ggu.resize(global\_mat.jg.size());

}

Matrix getLocalG(const Rectangle& rect) {

   Matrix g = {};

   double rp = nodes[rect.a].r;

   double hr = abs(nodes[rect.b].r - nodes[rect.a].r);

   double phi\_s = nodes[rect.a].phi;

   double h\_phi = abs(nodes[rect.c].phi - nodes[rect.a].phi);

   int i, j;

   // [i] & [j] variables are linked to [solverFunc] function

   auto solverFunc = [&](double r, double phi) {

      double ans = 0.0;

      ans += func\_Phi(i, phi\_s, h\_phi, phi) \* func\_R\_dif(i, rp, hr, r) \* func\_Phi(j, phi\_s, h\_phi, phi) \*

             func\_R\_dif(j, rp, hr, r);

      ans += (1.0 / (r \* r)) \* func\_R(i, rp, hr, r) \* func\_Phi\_dif(i, phi\_s, h\_phi, phi) \* func\_R(j, rp, hr, r) \*

             func\_Phi\_dif(j, phi\_s, h\_phi, phi);

      ans \*= lambda\_value(rect.regionNum, r, phi) \* r;

      return ans;

   };

   auto solver = Gaussian\_4p::TwoDimentionalSolver::withStep(rp, hr, phi\_s, h\_phi, solverFunc);

   for (i = 0; i < 4; i++) {

      for (j = 0; j < 4; j++) {

         g[i][j] = solver.compute();  // here i and j are linked to solver by lambda

      }

   }

   // debug output

#ifndef NDEBUG

   cout << "Local\_G:" << endl;

   for (i = 0; i < 4; i++) {

      for (j = 0; j < 4; j++) {

         cout << std::format(" {: .5f}", g[i][j]);

      }

      cout << endl;

   }

   cout << endl;

   // All tmp must be almost equal zero

   cout << "All this values must be zero:\n";

   for (i = 0; i < 4; i++) {

      double tmp = 0;

      for (j = 0; j < 4; j++) {

         tmp += g[i][j];

      }

      cout << " " << (tmp > 5e-14 ? tmp : 0);

   }

   cout << endl << endl;

#endif

   return g;

}

Matrix getLocalM\_chi(const Rectangle& rect, bool getWithoutChi = false) {

   Matrix m = {};

   std::function<double(int, double, double)> maybeChi;

   if (getWithoutChi == true) {

      maybeChi = [](int reg, double r, double phi) { return 1.0; };

   } else {

      maybeChi = [](int reg, double r, double phi) { return chi\_value(reg, r, phi); };

   }

   double rp = nodes[rect.a].r;

   double hr = abs(nodes[rect.b].r - nodes[rect.a].r);

   double phi\_s = nodes[rect.a].phi;

   double h\_phi = abs(nodes[rect.c].phi - nodes[rect.a].phi);

   int i, j;

   // [i] & [j] variables are linked to [solverFunc] function

   auto solverFunc = [&](double r, double phi) {

      double res = 1.0;

      res \*= func\_R(i, rp, hr, r) \* func\_Phi(i, phi\_s, h\_phi, phi);

      res \*= func\_R(j, rp, hr, r) \* func\_Phi(j, phi\_s, h\_phi, phi);

      res \*= r \* maybeChi(rect.regionNum, r, phi);

      return res;

   };

   auto solver = Gaussian\_4p::TwoDimentionalSolver::withStep(rp, hr, phi\_s, h\_phi, solverFunc);

   for (i = 0; i < 4; i++) {

      for (j = 0; j < 4; j++) {

         m[i][j] = solver.compute();  // [i] & [j] variables are linked to [solverFunc] function

      }

   }

#ifndef NDEBUG

   cout << "Local\_M" << endl;

   for (i = 0; i < 4; i++) {

      for (j = 0; j < 4; j++) {

         cout << std::format(" {: .5f}", m[i][j]);

      }

      cout << endl;

   }

   cout << endl;

#endif

   return m;

}

Matrix getLocalM\_sigma(const Rectangle& rect, bool getWithoutSigma = false) {

   Matrix m = {};

   std::function<double(int, double, double)> maybeSigma;

   if (getWithoutSigma == true) {

      maybeSigma = [](int reg, double r, double phi) { return 1.0; };

   } else {

      maybeSigma = [](int reg, double r, double phi) { return sigma\_value(reg, r, phi); };

   }

   double rp = nodes[rect.a].r;

   double hr = abs(nodes[rect.b].r - nodes[rect.a].r);

   double phi\_s = nodes[rect.a].phi;

   double h\_phi = abs(nodes[rect.c].phi - nodes[rect.a].phi);

   int i, j;

   // [i] & [j] variables are linked to [solverFunc] function

   auto solverFunc = [&](double r, double phi) {

      double res = 1.0;

      res \*= func\_R(i, rp, hr, r) \* func\_Phi(i, phi\_s, h\_phi, phi);

      res \*= func\_R(j, rp, hr, r) \* func\_Phi(j, phi\_s, h\_phi, phi);

      res \*= r \* maybeSigma(rect.regionNum, r, phi);

      return res;

   };

   auto solver = Gaussian\_4p::TwoDimentionalSolver::withStep(rp, hr, phi\_s, h\_phi, solverFunc);

   for (i = 0; i < 4; i++) {

      for (j = 0; j < 4; j++) {

         m[i][j] = solver.compute();  // [i] & [j] variables are linked to [solverFunc] function

      }

   }

#ifndef NDEBUG

   cout << "Local\_M" << endl;

   for (i = 0; i < 4; i++) {

      for (j = 0; j < 4; j++) {

         cout << std::format(" {: .5f}", m[i][j]);

      }

      cout << endl;

   }

   cout << endl;

#endif

   return m;

}

std::vector<double> getLocalB(const Rectangle& rect, double t) {

   std::vector<double> b(4);

   double rp = nodes[rect.a].r;

   double hr = abs(nodes[rect.b].r - nodes[rect.a].r);

   double phi\_s = nodes[rect.a].phi;

   double h\_phi = abs(nodes[rect.c].phi - nodes[rect.a].phi);

   // int i;

   // Данный код по хорошему должен был работать лучше, поскольку использует

   // более точный интеграл, но по факту - нифига подобного

   //

   // auto solverFunc = [&](double r, double phi) {

   //   double res = 1.0;

   //   res \*= func\_R(i, rp, hr, r) \* func\_Phi(i, phi\_s, h\_phi, phi);

   //   res \*= r \* f\_value(rect.regionNum, r, phi);

   //   return res;

   //};

   // auto solver = Gaussian\_4p::TwoDimentionalSolver::withStep(rp, hr, phi\_s,

   //    h\_phi, solverFunc);

   // for (i = 0; i < 4; i++) {

   //    b[i] = solver.compute();

   // }

   auto M = getLocalM\_chi(rect, true);

   for (int i = 0; i < 4; i++) {

      b[i] = 0;

      b[i] += M[i][0] \* f\_value(rect.regionNum, nodes[rect.a], t);

      b[i] += M[i][1] \* f\_value(rect.regionNum, nodes[rect.b], t);

      b[i] += M[i][2] \* f\_value(rect.regionNum, nodes[rect.c], t);

      b[i] += M[i][3] \* f\_value(rect.regionNum, nodes[rect.d], t);

   }

#ifndef NDEBUG

   // debug output

   cout << "Local b:\n";

   for (int i = 0; i < 4; i++) {

      cout << std::format(" {: .5f}", b[i]);

   }

   cout << endl << endl;

#endif

   return b;

}

void addLocalMatrixToGlobal(const Rectangle& rect, SparseMatrix& globalMat, const Matrix& localMat) {

   const int elems[4] = {rect.a, rect.b, rect.c, rect.d};

   for (int i = 0; i < 4; i++) {

      // добавляем все внедиагональные элементы на строке elems[i]

      for (int k = 0; k < i; k++) {

         // Если элемент в верхнем прямоугольнике, то скипаем

         if (elems[k] > elems[i]) {

            continue;

         }

         auto id = globalMat.ig[elems[i]];

         for (id; id < globalMat.ig[elems[i] + 1ll] && globalMat.jg[id] != elems[k]; id++)

            ;

         globalMat.ggl[id] += localMat[i][k];

         globalMat.ggu[id] += localMat[i][k];

      }

      // добавляем диагональные элементы

      globalMat.di[elems[i]] += localMat[i][i];

   }

}

void addLocalbToGlobal(const Rectangle& rect, std::vector<double>& globalVec, const std::vector<double>& localVec) {

   const int elems[4] = {rect.a, rect.b, rect.c, rect.d};

   for (int i = 0; i < 4; i++) {

      globalVec[elems[i]] += localVec[i];

   }

}

void include\_s1(double t) {

   for (const auto& node : s1\_nodes) {

      double u = s1\_u\_value(node.funcNum, nodes[node.node], t);

      // ставим на диагональ значение 1

      global\_mat.di[node.node] = 1;

      // ставим в соответствующую ячейку вектора b значение u

      global\_d[node.node] = u;

      // зануляем строку в нижнем треугольнике

      for (auto j = global\_mat.ig[node.node]; j < global\_mat.ig[node.node + 1ll]; j++) {

         global\_mat.ggl[j] = 0;

      }

      // зануляем строку в верхнем треугольнике

      for (int i = node.node + 1; i < global\_mat.Size(); i++) {

         for (auto j = global\_mat.ig[i]; j < global\_mat.ig[i + 1ll]; j++) {

            if (global\_mat.jg[j] == node.node) {

               global\_mat.ggu[j] = 0;

               break;

            }

         }

      }

   }

}

int main() {

   setlocale(LC\_ALL, "ru-RU.utf8");

   readDataFromFiles();

   generatePortrait();

   global\_b.resize(global\_mat.Size());

   std::array<std::vector<double>, 3> slices;  // Последние 3 решения для трёхслойки

   auto global\_M\_sigma = SparseMatrix::copyShape(global\_mat);

   auto global\_M\_chi = SparseMatrix::copyShape(global\_mat);

   auto global\_G = SparseMatrix::copyShape(global\_mat);

   // Считаем глобальные матрицы

   for (const auto& rect : rectangles) {

      addLocalMatrixToGlobal(rect, global\_G, getLocalG(rect));

      addLocalMatrixToGlobal(rect, global\_M\_chi, getLocalM\_chi(rect));

      addLocalMatrixToGlobal(rect, global\_M\_sigma, getLocalM\_sigma(rect));

   }

   // Получаем временную сетку

   constexpr double tBegin = 0.0;

   constexpr double tEnd = 10.0;

   constexpr double tStep = 1.0;

   const double tCount = std::floor((tEnd - tBegin) / tStep + 1);

   std::vector<double> t;

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

      t.push\_back(tBegin + i \* tStep);

   }

   // Получаем первые два решения из начальных условий

   slices[0].resize(nodes.size());

   slices[1].resize(nodes.size());

   slices[2].resize(nodes.size());

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

      slices[0].at(i) = s1\_u\_value(0, nodes[i], t[0]);

      slices[1].at(i) = s1\_u\_value(0, nodes[i], t[1]);

   }

   // Инициализируем решатель

   IterSolvers::LOS::Init\_LuPrecond(global\_mat.Size(), global\_mat);

   std::vector<double> errors = {0.0, 0.0};

   // Вычисляем остальные слои:

   for (size\_t j = 2; j < tCount; j++) {

      double dt = t[j] - t[j - 2];

      double dt1 = t[j - 1] - t[j - 2];

      double dt0 = t[j] - t[j - 1];

      // Зануляем и вычисляем вектор правой части

      for (auto& el : global\_b) el = 0.0;

      for (const auto& rect : rectangles) {

         addLocalbToGlobal(rect, global\_b, getLocalB(rect, t[j - 1]));

      }

      // Вычисляем глобальную матрицу A

      global\_mat = (2.0 / dt / dt0) \* global\_M\_chi + (dt1 / dt / dt0) \* global\_M\_sigma;

      // Вычисляем глобавльный вектор правой части d

      global\_d = global\_b;

      global\_d = global\_d + (-2.0 / dt / dt1) \* global\_M\_chi \* slices[0] + (2 / dt1 / dt0) \* global\_M\_chi \* slices[1];

      global\_d = global\_d + (dt0 / dt / dt1) \* global\_M\_sigma \* slices[0] +

                 (-(dt0 - dt1) / dt1 / dt0) \* global\_M\_sigma \* slices[1];

      global\_d = global\_d + (-1) \* global\_G \* slices[1];

      // Накладываем первые краевые

      include\_s1(t[j]);

      // Решаем СЛАУ, получаем решение

      double eps;

      IterSolvers::LOS::LuPrecond(global\_mat, global\_d, slices[2], eps);

      cout << "Текущее время: " << t[j] << endl << endl;

      cout << "Полученное решение: \n";

      for (auto el : slices[2]) {

         cout << std::format("{: .14e}\n", el);

      }

      cout << endl << "Погрешность решения: \t";

      double err = 0.0;

      for (int i = 0; i < slices[2].size(); i++) {

         double local\_err = slices[2][i] - s1\_u\_value(0, nodes[i], t[j]);

         err += local\_err \* local\_err;

         cout << std::format("{: .14e}\n", local\_err);

      }

      err = std::sqrt(err);

      cout << endl << "Норма погрешности: " << err << "\n\n\n";

      errors.push\_back(err);

      // Сдвигаем слои

      slices[0] = slices[1];

      slices[1] = slices[2];

   }

   // Деструктуризируем решатель

   IterSolvers::Destruct();

   // Выводим отдельно табличку вида | время | норма погрешности |

   cout << "| время |  норма погрешности   |\n";

   cout << "|  ---  |  ---                 |\n";

   for (int i = 0; i < tCount; i++) {

      cout << std::format("| {:5.2} | {:.14e} |\n", t[i], errors[i]);

   }

}

#### Файл Constants.h:

/\*

   Файл, содержащий в себе только вынесенные константы и константные функции для main.cpp

   Ни в коем случае не добавлять его никуда, кроме main.cpp!

\*/

#pragma once

#include <array>

#include <stdexcept>

#include <string>

namespace GlobalPaths {

// Пути файлов:

const std::string filesPath = "../../iofiles/";

const std::string nodesPath = filesPath + "nodes.txt";

const std::string rectanglesPath = filesPath + "rectangles.txt";

const std::string s1\_nodesPath = filesPath + "s1\_nodes.txt";

}  // namespace GlobalPaths

#pragma region TYPEDEFINES

using Matrix = std::array<std::array<double, 4>, 4>;

/// <summary>

/// Структура прямоугольника, имеет 4 номера вершины: [a], [b], [c], [d], а также

/// номер области, в которой находится сам прямоугольник, [region]

/// </summary>

struct Rectangle {

   int a = 0;

   int b = 0;

   int c = 0;

   int d = 0;

   int regionNum = 0;

   std::string toString() const {

      std::string out = "( ";

      out += "a: " + std::to\_string(a);

      out += ", b: " + std::to\_string(b);

      out += ", c: " + std::to\_string(c);

      out += ", d: " + std::to\_string(d);

      out += ", region: " + std::to\_string(regionNum);

      out += " )";

      return out;

   }

};

/// <summary>

/// Структура описания узла сетки. Содержит координаты этого узла [r] и [phi]

/// </summary>

struct Node {

   double r = 0.0;

   double phi = 0.0;

};

struct S1\_node {

   int node = 0;

   int funcNum = 0;

};

#pragma endregion TYPEDEFINES

double f\_value(int regionNum, double r, double phi, double t) {

   switch (regionNum) {

      case 0: {

         return -1;

      }

      default:

         throw std::runtime\_error("Значения функции f для региона с номером " + std::to\_string(regionNum) +

                                  " не найдено.");

   }

}

double f\_value(int regionNum, Node node, double t) { return f\_value(regionNum, node.r, node.phi, t); }

double lambda\_value(int regionNum, double r, double phi) {

   switch (regionNum) {

      case 0: {

         return r;

      }

      default:

         throw std::runtime\_error("Значения функции lambda для региона с номером " + std::to\_string(regionNum) +

                                  " не найдено.");

   }

}

double lambda\_value(int regionNum, Node node) { return lambda\_value(regionNum, node.r, node.phi); }

double s1\_u\_value(int s1\_funcNum, double r, double phi, double t) {

   switch (s1\_funcNum) {

      case 0: {

         return t + r;

      }

      default:

         throw std::runtime\_error("Значения функции u для s1-краевого с номером " + std::to\_string(s1\_funcNum) +

                                  " не найдено.");

   }

}

double s1\_u\_value(int s1\_funcNum, Node node, double t) { return s1\_u\_value(s1\_funcNum, node.r, node.phi, t); }

double chi\_value(int regionNum, double r, double phi) {

   switch (regionNum) {

         case 0: return 1;

      default:

         throw std::runtime\_error("Значения функции chi для области с номером " + std::to\_string(regionNum) +

                                  " не найдено.");

   }

}

double chi\_value(int regionNum, Node node) { return chi\_value(regionNum, node.r, node.phi); }

double sigma\_value(int regionNum, double r, double phi) {

   switch (regionNum) {

         case 0: return 1;

      default:

         throw std::runtime\_error("Значения функции sigma для области с номером " + std::to\_string(regionNum) +

                                  " не найдено.");

   }

}

double sigma\_value(int regionNum, Node node) { return sigma\_value(regionNum, node.r, node.phi); }

#### Файл gaussian\_quadrature.h:

#pragma once

#include <functional>

#include <cmath>

namespace Gaussian\_2p {

   constexpr int numPoint = 2;

   constexpr double weights[] = {1.0, 1.0};

   constexpr double points[] = {

      -0.577350269189625764509148780,    // -1 / sqrt(3)

      0.577350269189625764509148780,     //  1 / sqrt(3)

   };

   /// <summary>

   /// Class for compute integral of one-dimentional function, like \int\_0^2 {2x + x dx}

   /// </summary>

   class OneDimentionSolver {

   public:

      double from = 0.0;

      double to = 0.0;

      std::function<double(double)> computeFunc;

   public:

      OneDimentionSolver(double from, double to, std::function<double(double)> func) :

         from(from), to(to), computeFunc(func) {}

      static OneDimentionSolver withStep(double from, double step, std::function<double(double)> func) {

         return OneDimentionSolver(from, from + step, func);

      }

   public:

      /// <summary>

      /// Function to compute quadrature with other range. Doesn't override previous ranges

      /// </summary>

      double computeWithRange(double from, double to) {

         const double coefs[] = {

            (to - from) / 2.0,

            (to + from) / 2.0,

         };

         double result = 0.0;

         for (auto i = 0; i < numPoint; i++)

         {

            result += weights[i] \* computeFunc(coefs[0] \* points[i] + coefs[1]);

         }

         return coefs[0] \* result;

      }

      /// <summary>

      /// Function to compute quadrature by this equation:

      /// (to-from) / 2 \* (w1 \* f((to-from)/2 \* x1 + (to+from)/2 ) + w2 \* f((to-from)/2 \* x1 + (to+from)/2 )

      /// </summary>

      /// <returns>result of computation (integral from [from] to [to])</returns>

      double compute() {

         return computeWithRange(from, to);

      }

      /// <summary>

      /// Function to compute quadrature with other range with step. Doesn't override previous ranges

      /// </summary>

      double computeWithStep(double from, double step) {

         return computeWithRange(from, from + step);

      }

   };

   /// <summary>

   /// Class for compute two-dimentional functions like int\_xfrom^xTo {int\_yFrom^yTo {x^2 + y^2 dy} dx}

   /// </summary>

   class TwoDimentionalSolver {

   public:

      double xFrom = 0.0;

      double xTo = 0.0;

      double yFrom = 0.0;

      double yTo = 0.0;

      std::function<double(double, double)> computeFunc;

   public:

      TwoDimentionalSolver(double xFrom, double xTo, double yFrom, double yTo, std::function<double(double, double)> computeFunc) :

         xFrom(xFrom), xTo(xTo), yFrom(yFrom), yTo(yTo), computeFunc(computeFunc) {}

      static TwoDimentionalSolver withStep(double xFrom, double xStep, double yFrom, double yStep, std::function<double(double, double)> computeFunc) {

         return TwoDimentionalSolver(xFrom, xFrom + xStep, yFrom, yFrom + yStep, computeFunc);

      }

   public:

      /// <summary>

      /// Function that return computed integral with fixed range, doesn't change in-object range

      /// </summary>

      /// <returns>Result of computation</returns>

      double computeWithRange(double xFrom, double xTo, double yFrom, double yTo) {

         const double x\_coefs[] = {

            (xTo - xFrom) / 2.0,

            (xTo + xFrom) / 2.0,

         };

         const double y\_coefs[] = {

            (yTo - yFrom) / 2.0,

            (yTo + yFrom) / 2.0,

         };

         double result = 0.0;

         for (auto i = 0; i < numPoint; i++)

         {

            for (auto j = 0; j < numPoint; j++)

            {

               result += weights[i] \* weights[j]

                  \* computeFunc(x\_coefs[0] \* points[i] + x\_coefs[1], y\_coefs[0] \* points[j] + y\_coefs[1]);

            }

         }

         return x\_coefs[0] \* y\_coefs[0] \* result;

      }

      /// <summary>

      /// Function that return computed integral with fixed in-object range

      /// </summary>

      /// <returns>Result of computation</returns>

      double compute() {

         return computeWithRange(xFrom, xTo, yFrom, yTo);

      }

      /// <summary>

      /// Function that return computed integral with fixed range, doesn't change in-object range

      /// </summary>

      /// <returns>Result of computation</returns>

      double computeWithStep(double xFrom, double xStep, double yFrom, double yStep) {

         return computeWithRange(xFrom, xFrom + xStep, yFrom, yFrom + yStep);

      }

   };

}

namespace Gaussian\_4p {

   constexpr int numPoint = 4;

   constexpr double weights[] = {

      0.347854845137453857373063949222,

      0.652145154862546142626936050778,

      0.652145154862546142626936050778,

      0.347854845137453857373063949222

   };

   constexpr double points[] = {

      -0.86113631159405257522394648889281,

      -0.33998104358485626480266575910324,

      0.33998104358485626480266575910324,

      0.86113631159405257522394648889281,

   };

   /// <summary>

   /// Class for compute integral of one-dimentional function, like \int\_0^2 {2x + x dx}

   /// </summary>

   class OneDimentionSolver {

   public:

      double from = 0.0;

      double to = 0.0;

      std::function<double(double)> computeFunc;

   public:

      OneDimentionSolver(double from, double to, std::function<double(double)> func) :

         from(from), to(to), computeFunc(func) {}

      static OneDimentionSolver withStep(double from, double step, std::function<double(double)> func) {

         return OneDimentionSolver(from, from + step, func);

      }

   public:

      /// <summary>

      /// Function to compute quadrature with other range. Doesn't override previous ranges

      /// </summary>

      double computeWithRange(double from, double to) {

         const double coefs[] = {

            (to - from) / 2.0,

            (to + from) / 2.0,

         };

         double result = 0.0;

         for (auto i = 0; i < numPoint; i++)

         {

            result += weights[i] \* computeFunc(coefs[0] \* points[i] + coefs[1]);

         }

         return coefs[0] \* result;

      }

      /// <summary>

      /// Function to compute quadrature by this equation:

      /// (to-from) / 2 \* (w1 \* f((to-from)/2 \* x1 + (to+from)/2 ) + w2 \* f((to-from)/2 \* x1 + (to+from)/2 )

      /// </summary>

      /// <returns>result of computation (integral from [from] to [to])</returns>

      double compute() {

         return computeWithRange(from, to);

      }

      /// <summary>

      /// Function to compute quadrature with other range with step. Doesn't override previous ranges

      /// </summary>

      double computeWithStep(double from, double step) {

         return computeWithRange(from, from + step);

      }

   };

   /// <summary>

   /// Class for compute two-dimentional functions like int\_xfrom^xTo {int\_yFrom^yTo {x^2 + y^2 dy} dx}

   /// </summary>

   class TwoDimentionalSolver {

   public:

      double xFrom = 0.0;

      double xTo = 0.0;

      double yFrom = 0.0;

      double yTo = 0.0;

      std::function<double(double, double)> computeFunc;

   public:

      TwoDimentionalSolver(double xFrom, double xTo, double yFrom, double yTo, std::function<double(double, double)> computeFunc) :

         xFrom(xFrom), xTo(xTo), yFrom(yFrom), yTo(yTo), computeFunc(computeFunc) {}

      static TwoDimentionalSolver withStep(double xFrom, double xStep, double yFrom, double yStep, std::function<double(double, double)> computeFunc) {

         return TwoDimentionalSolver(xFrom, xFrom + xStep, yFrom, yFrom + yStep, computeFunc);

      }

   public:

      /// <summary>

      /// Function that return computed integral with fixed range, doesn't change in-object range

      /// </summary>

      /// <returns>Result of computation</returns>

      double computeWithRange(double xFrom, double xTo, double yFrom, double yTo) {

         const double x\_coefs[] = {

            (xTo - xFrom) / 2.0,

            (xTo + xFrom) / 2.0,

         };

         const double y\_coefs[] = {

            (yTo - yFrom) / 2.0,

            (yTo + yFrom) / 2.0,

         };

         double result = 0.0;

         for (auto i = 0; i < numPoint; i++)

         {

            for (auto j = 0; j < numPoint; j++)

            {

               result += weights[i] \* weights[j]

                  \* computeFunc(x\_coefs[0] \* points[i] + x\_coefs[1], y\_coefs[0] \* points[j] + y\_coefs[1]);

            }

         }

         return x\_coefs[0] \* y\_coefs[0] \* result;

      }

      /// <summary>

      /// Function that return computed integral with fixed in-object range

      /// </summary>

      /// <returns>Result of computation</returns>

      double compute() {

         return computeWithRange(xFrom, xTo, yFrom, yTo);

      }

      /// <summary>

      /// Function that return computed integral with fixed range, doesn't change in-object range

      /// </summary>

      /// <returns>Result of computation</returns>

      double computeWithStep(double xFrom, double xStep, double yFrom, double yStep) {

         return computeWithRange(xFrom, xFrom + xStep, yFrom, yFrom + yStep);

      }

   };

}

#### Файл SparseMatrix.h:

#pragma once

#include <cmath>

#include <format>

#include <fstream>

#include <stdexcept>

#include <string>

#include <vector>

std::vector<double> ReadVecFromFile(size\_t size, const std::string& path);

/// <summary>

/// Класс объектов матриц, хранящихся в разреженном строчно-столбцовом виде

/// <para> Точность хранения элементов - double </para>

/// </summary>

class SparseMatrix {

   // Переменные матрицы

  public:

   /// <summary>

   /// Массив индексов строк/столбцов, вида 0, 0, 0 + k2, ..., 0+k2+...+kn, где ki - число элементов в i cтроке/столбце

   /// <para> Помимо этого первый элемент i строки можно найти как ggl[ig[i]] </para>

   /// <para> Пример массива для матрицы 3х3: </para>

   ///

   /// <para> Матрица: </para>

   /// <para> | 1 2 0 | </para>

   /// <para> | 3 8 1 | </para>

   /// <para> | 0 2 4 | </para>

   /// <para> ig: { 0, 0, 1, 2 } </para>

   /// </summary>

   std::vector<uint32\_t> ig;

   /// <summary>

   /// Массив индексов столбцов/строк элементов (ставит индекс в соответствие элементу)

   /// <para> Пример массива для матрицы 3х3: </para>

   ///

   /// <para> Матрица: </para>

   /// <para> | 1 2 0 | </para>

   /// <para> | 3 8 1 | </para>

   /// <para> | 0 2 4 | </para>

   /// <para> jg: { 0, 1 } </para>

   /// </summary>

   std::vector<uint16\_t> jg;

   /// <summary>

   /// Массив элементов нижнего треугольника матрицы

   /// <para> Пример массива для матрицы 3х3: </para>

   ///

   /// <para> Матрица: </para>

   /// <para> | 1 2 0 | </para>

   /// <para> | 3 8 1 | </para>

   /// <para> | 0 2 4 | </para>

   /// <para> ggl: { 3, 2 } </para>

   /// </summary>

   std::vector<double> ggl;

   /// <summary>

   /// Массив элементов верхнего треугольника матрицы

   /// <para> Пример массива для матрицы 3х3: </para>

   ///

   /// <para> Матрица: </para>

   /// <para> | 1 2 0 | </para>

   /// <para> | 3 8 1 | </para>

   /// <para> | 0 2 4 | </para>

   /// <para> ggu: { 2, 1 } </para>

   /// </summary>

   std::vector<double> ggu;

   /// <summary>

   /// Массив элементов диагонали матрицы

   /// <para> Пример массива для матрицы 3х3: </para>

   ///

   /// <para> Матрица: </para>

   /// <para> | 1 2 0 | </para>

   /// <para> | 3 8 1 | </para>

   /// <para> | 0 2 4 | </para>

   /// <para> di: { 1, 8, 4 } </para>

   /// </summary>

   std::vector<double> di;

   // Методы матрицы

  public:

   size\_t Size() const;

   //Сложение разреженных матриц одной формы

   SparseMatrix operator+ (const SparseMatrix& other) const;

   /// <summary>

   /// Умножение матрицы на вектор

   /// </summary>

   std::vector<double> MultToVec(const std::vector<double>& right) const;

   std::vector<double>& MultToVec(const std::vector<double>& right, std::vector<double>& result) const;

   std::vector<double> operator\*(const std::vector<double>& right) const;

   // Умножение матрицы на число

   SparseMatrix multToScalar(double scalar) const;

   // Умножение матрицы на число

   SparseMatrix operator\*(double scalar) const { return multToScalar(scalar); }

   // Умножение матрицы на число

   friend SparseMatrix operator\*(double scalar, const SparseMatrix& right) { return right \* scalar; }

   /// <summary>

   /// Умножение транспонированной матрицы на вектор

   /// </summary>

   std::vector<double> TranspMultToVec(const std::vector<double>& right) const;

   std::vector<double>& TranspMultToVec(const std::vector<double>& right, std::vector<double>& result) const;

   SparseMatrix& operator=(SparseMatrix&& right) noexcept;

   double val(uint16\_t row, uint16\_t column);

   std::string toStringAsDense() {

      std::string out = "[ ";

      auto size = Size();

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

         if (i != 0) out += "  ";

         out += "[ ";

         for (auto j = 0; j < size; j++) {

            out += std::format("{: 15.5f}", val(i, j));  // std::to\_string(val(i, j));

            if (j + 1ll < size) out += ", ";

         }

         out += " ]";

         if (i + 1ll < size) out += "\n";

      }

      out += " ]";

      return out;

   }

   // Конструкторы матрицы

  public:

   SparseMatrix();

   // Конструктор копирования

   SparseMatrix(const SparseMatrix& right);

   // Конструктор перемещения (нужен для метода ReadFromFiles)

   SparseMatrix(SparseMatrix&& right) noexcept;

   // Статические методы матрицы

  public:

   // Конструктор копирования формы матрицы

   static SparseMatrix copyShape(const SparseMatrix& other);

   static SparseMatrix ReadFromFiles(uint16\_t matrixSize, const std::string& igP, const std::string& jgP,

                                     const std::string& gglP, const std::string& gguP, const std::string& diP);

};

#### Файл LU.h:

#pragma once

#include <vector>

#include "SparseMatrix.h"

// Неполное разложение LU(sq) матрицы разреженного строчно-столбцового формата SparseMatrix

// Не хранит портрет матрицы, но использует портрет исходной матрицы (а также ссылается на неё)

class LU {

// Блок внутренних переменных разложения LU

public:

   const SparseMatrix\* parent = nullptr;

   // Вектор диагональных элементов LU разложения. В данном случае диагонали L и U совпадают

   std::vector<double> di;

   // Вектор элементов нижнего треугольника L

   std::vector<double> ggl;

   // Вектор элементов верхнего треугольника U

   std::vector<double> ggu;

// Блок основных конструкторов класса

public:

   /// <summary>

   /// Конструктор с резервированием памяти под разложение

   /// </summary>

   /// <param name="diSize"> - размер диагонали,</param>

   /// <param name="luSize"> - размер массивов нижнего и верхнего треугольника</param>

   LU(size\_t diSize, size\_t luSize);

   /// <summary>

   /// Конструктор с построением неполного LU(sq)-разложения по матрице mat

   /// </summary>

   /// <param name="mat"> - матрица, по которой построится LU-разложение, с привязкой этой матрицы к объекту</param>

   LU(const SparseMatrix& mat);

// Блок основных нестатических методов класса

public:

   /// <summary>

   /// Разложить матрицу mat в неполное LU(sq) - разложение

   /// </summary>

   /// <param name="mat"> - матрица, которую требуется разложить. Она же будет использоваться для просмотра портрета матриц</param>

   void MakeLuFor(const SparseMatrix& mat);

   /// <summary>

   /// Метод изменения размера разложения

   /// </summary>

   /// <param name="diSize"> - размер диагонали,</param>

   /// <param name="luSize"> - размер массивов нижнего и верхнего треугольника</param>

   void Resize(size\_t diSize, size\_t luSize);

// Умножение матриц на вектор

   /// <summary>

   /// Умножение нижней матрицы L на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы L</param>

   /// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

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

   /// <summary>

   /// Умножение нижней матрицы L на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы L;</param>

   /// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

   /// <returns>Ссылка на вектор ans</returns>

   std::vector<double>& LMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

   /// <summary>

   /// Умножение нижней матрицы L^T на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы L</param>

   /// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

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

   /// <summary>

   /// Умножение нижней матрицы L^T на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы L;</param>

   /// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

   /// <returns>Ссылка на вектор ans</returns>

   std::vector<double>& LTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

   /// <summary>

   /// Умножение верхней матрицы U на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы U</param>

   /// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

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

   /// <summary>

   /// Умножение верхней матрицы U на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы U;</param>

   /// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

   /// <returns>Ссылка на вектор ans</returns>

   std::vector<double>& UMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

   /// <summary>

   /// Умножение верхней матрицы U^T на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы U</param>

   /// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

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

   /// <summary>

   /// Умножение верхней матрицы U^T на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

   /// </summary>

   /// <param name="vec"> - вектор, на который будет происходить умножение матрицы U;</param>

   /// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

   /// <returns>Ссылка на вектор ans</returns>

   std::vector<double>& UTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

// Решение слау с использованием матриц и вектора правой части

   /// <summary>

   /// Решение слау вида Lx = right. Не выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

   /// <returns>ссылка на вектор x</returns>

   std::vector<double>& LSlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

   /// <summary>

   /// Решение слау вида Lx = right. Выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <returns>Полученный вектор x</returns>

   std::vector<double> LSlauSolve(const std::vector<double>& right) const;

   /// <summary>

   /// Решение слау вида L^T \* x = right. Не выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

   /// <returns>ссылка на вектор x</returns>

   std::vector<double>& LTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

   /// <summary>

   /// Решение слау вида L^T \* x = right. Выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <returns>Полученный вектор x</returns>

   std::vector<double> LTranspSlauSolve(const std::vector<double>& right) const;

   /// <summary>

   /// Решение слау вида Ux = right. Не выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

   /// <returns>ссылка на вектор x</returns>

   std::vector<double>& USlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

   /// <summary>

   /// Решение слау вида Ux = right. Выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <returns>Полученный вектор x</returns>

   std::vector<double> USlauSolve(const std::vector<double>& right) const;

   /// <summary>

   /// Решение слау вида U^T \* x = right. Не выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

   /// <returns>ссылка на вектор x</returns>

   std::vector<double>& UTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

   /// <summary>

   /// Решение слау вида U^T \* x = right. Выделяет память под вектор x, не меняет матрицы LU

   /// </summary>

   /// <param name="right"> - вектор правой части уравнения;</param>

   /// <returns>Полученный вектор x</returns>

   std::vector<double> UTranspSlauSolve(const std::vector<double>& right) const;

};

#### Файл IterSolvers.h:

#pragma once

#include <vector>

#include <stdexcept>

#include <format>

#include <iostream>

#include "SparseMatrix.h"

#include "LU.h"

namespace Vec {

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

   // l or r may be similar vectors to ans

   inline void Mult(const std::vector<double>& l, const std::vector<double>& r, std::vector<double>& ans);

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

}

namespace IterSolvers {

   extern double minEps;

   extern size\_t maxIter;

   extern bool globalDebugOutput;

   extern std::vector<double>\* \_tmp1, \* \_tmp2,

      \* \_tmp3, \* \_tmp4, \* \_tmp5, \* \_tmp6;

   extern LU\* \_lu\_mat;

   inline void VecInit(std::vector<double>\*& vec, size\_t size);

   namespace MSG\_Assimetric {

      void Init\_Default(size\_t size);

      size\_t Default(const SparseMatrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

      void Init\_DiagPrecond(size\_t size);

      size\_t DiagPrecond(const SparseMatrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

      void Init\_LuPrecond(size\_t diSize, const SparseMatrix& A);

      size\_t LuPrecond(const SparseMatrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

   }

   namespace LOS {

      extern size\_t resetIter;

      void Init\_Default(size\_t size);

      size\_t Default(const SparseMatrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

      void Init\_DiagPrecond(size\_t size);

      size\_t DiagPrecond(const SparseMatrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

      void Init\_LuPrecond(size\_t diSize, const SparseMatrix& A);

      size\_t LuPrecond(const SparseMatrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

   }

   void Destruct() noexcept;

};

#### Файл SparseMatrix.cpp:

#include "../Headers/SparseMatrix.h"

using namespace std;

vector<double> ReadVecFromFile(size\_t size, const string& path) {

   vector<double> vec(size);

   auto file = ifstream(path);

   if (!file.is\_open()) {

      throw runtime\_error("Файл " + path + " отсутствует в директории");

   }

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

      file >> vec[i];

   }

   file.close();

   return vec;

}

// Методы матрицы

SparseMatrix SparseMatrix::operator+(const SparseMatrix& other) const {

   SparseMatrix result(\*this);

   for (size\_t i = 0; i < result.di.size(); i++) {

      result.di.at(i) += other.di.at(i);

   }

   for (size\_t i = 0; i < result.ggl.size(); i++) {

      result.ggl.at(i) += other.ggl.at(i);

      result.ggu.at(i) += other.ggu.at(i);

   }

   return result;

}

size\_t SparseMatrix::Size() const { return di.size(); }

/// <summary>

/// Умножение матрицы на вектор

/// </summary>

vector<double> SparseMatrix::MultToVec(const vector<double>& right) const {

   vector<double> result(right.size());

   return MultToVec(right, result);

}

/// <summary>

/// Умножение матрицы на вектор

/// </summary>

vector<double>& SparseMatrix::MultToVec(const vector<double>& right, vector<double>& result) const {

   if (right.size() != di.size()) throw runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (right.size() != result.size()) throw runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   for (uint16\_t i = 0; i < result.size(); i++) {

      // Умножаем диагональ

      result[i] = di[i] \* right[i];

      // Умножаем нижний и верхний треугольники

      for (uint32\_t j = ig[i]; j < ig[i + 1ll]; j++) {

         result[i] += ggl[j] \* right[jg[j]];

         result[jg[j]] += ggu[j] \* right[i];

      }

   }

   return result;

}

vector<double> SparseMatrix::operator\*(const vector<double>& right) const { return MultToVec(right); }

SparseMatrix SparseMatrix::multToScalar(double scalar) const {

   SparseMatrix result(\*this);

   for (auto& el : result.di) {

      el \*= scalar;

   }

   for (size\_t i = 0; i < result.ggl.size(); i++) {

      result.ggl[i] \*= scalar;

      result.ggu[i] \*= scalar;

   }

   return result;

}

/// <summary>

/// Умножение транспонированной матрицы на вектор

/// </summary>

vector<double>& SparseMatrix::TranspMultToVec(const vector<double>& right, vector<double>& result) const {

   if (right.size() != di.size()) throw runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (right.size() != result.size()) throw runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   for (uint16\_t i = 0; i < result.size(); i++) {

      // Умножаем диагональ

      result[i] = di[i] \* right[i];

      // Умножаем нижний и верхний треугольники

      for (uint32\_t j = ig[i]; j < ig[i + 1ll]; j++) {

         result[i] += ggu[j] \* right[jg[j]];

         result[jg[j]] += ggl[j] \* right[i];

      }

   }

   return result;

}

/// <summary>

/// Умножение транспонированной матрицы на вектор

/// </summary>

vector<double> SparseMatrix::TranspMultToVec(const vector<double>& right) const {

   vector<double> result(right.size());

   return TranspMultToVec(right, result);

}

SparseMatrix& SparseMatrix::operator=(SparseMatrix&& right) noexcept {

   ig = std::move(right.ig);

   jg = std::move(right.jg);

   ggl = std::move(right.ggl);

   ggu = std::move(right.ggu);

   di = std::move(right.di);

   return \*this;

}

double SparseMatrix::val(uint16\_t row, uint16\_t column) {

   // if element lay on diagonal

   if (row == column) return di[row];

   auto& v = row > column ? ggl : ggu;

   if (row < column) std::swap(row, column);

   // find element by his pos

   auto i = ig[row];

   while (i < ig[row + 1ll] && jg[i] < column) i++;

   // if element exists, return him

   if (i < ig[row + 1ll] && jg[i] == column) return v[i];

   // else return zero

   return 0.0;

}

// Конструкторы матрицы

SparseMatrix::SparseMatrix() {}

SparseMatrix::SparseMatrix(const SparseMatrix& right)

    : ig{right.ig}, jg{right.jg}, ggl{right.ggl}, ggu{right.ggu}, di{right.di} {}

// Конструктор перемещения (нужен для метода ReadFromFiles)

SparseMatrix::SparseMatrix(SparseMatrix&& right) noexcept {

   ig = std::move(right.ig);

   jg = std::move(right.jg);

   ggl = std::move(right.ggl);

   ggu = std::move(right.ggu);

   di = std::move(right.di);

}

// Статические методы матрицы

SparseMatrix SparseMatrix::ReadFromFiles(uint16\_t matrixSize, const string& igP, const string& jgP, const string& gglP,

                                         const string& gguP, const string& diP) {

   SparseMatrix mat;

   bool isStartFromOne = false;

   {

      mat.ig.resize(matrixSize + 1ll);

      auto igS = ifstream(igP);

      if (!igS.is\_open()) throw runtime\_error("Файл " + igP + " отсутствует в директории.");

      for (uint16\_t i = 0; i <= matrixSize; i++) {

         igS >> mat.ig[i];

      }

      // Если массив ig в файле начинался с 1, то меняем его под наши параметры (под 0)

      if (isStartFromOne = mat.ig[0]) {

         for (uint16\_t i = 0; i <= matrixSize; i++) {

            mat.ig[i]--;

         }

      }

   }

   {

      auto jgS = ifstream(jgP);

      if (!jgS.is\_open()) throw runtime\_error("Файл " + jgP + " отсутствует в директории.");

      mat.jg.resize(mat.ig.back());

      for (uint32\_t i = 0; i < mat.jg.size(); i++) {

         jgS >> mat.jg[i];

         if (isStartFromOne) {

            mat.jg[i]--;

         }

      }

   }

   try {

      mat.di = ReadVecFromFile(matrixSize, diP);

      mat.ggl = ReadVecFromFile(mat.jg.size(), gglP);

      mat.ggu = ReadVecFromFile(mat.jg.size(), gguP);

   } catch (exception& e) {

      throw e;

   }

   return mat;

}

SparseMatrix SparseMatrix::copyShape(const SparseMatrix& other) {

   SparseMatrix mat;

   mat.ig = other.ig;

   mat.jg = other.jg;

   mat.di.resize(other.di.size());

   mat.ggl.resize(other.ggl.size());

   mat.ggu.resize(other.ggu.size());

   return mat;

}

#### Файл LU.cpp:

#include "../Headers/LU.h"

/// <summary>

/// Конструктор с резервированием памяти под разложение

/// </summary>

/// <param name="diSize"> - размер диагонали,</param>

/// <param name="luSize"> - размер массивов нижнего и верхнего треугольника</param>

LU::LU(size\_t diSize, size\_t luSize) {

   Resize(diSize, luSize);

}

/// <summary>

/// Конструктор с построением неполного LU(sq)-разложения по матрице mat

/// </summary>

/// <param name="mat"> - матрица, по которой построится LU-разложение, с привязкой этой матрицы к объекту</param>

LU::LU(const SparseMatrix& mat)

{

   MakeLuFor(mat);

}

/// <summary>

/// Разложить матрицу mat в неполное LU(sq) - разложение

/// </summary>

/// <param name="mat"> - матрица, которую требуется разложить. Она же будет использоваться для просмотра портрета матриц</param>

void LU::MakeLuFor(const SparseMatrix& mat) {

   parent = &mat;

   if (di.size() != mat.di.size())

      di.resize(mat.di.size());

   if (ggl.size() != mat.ggl.size())

      ggl.resize(mat.ggl.size());

   if (ggu.size() != mat.ggu.size())

      ggu.resize(mat.ggu.size());

   const auto& ig = mat.ig;

   const auto& jg = mat.jg;

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

   {

      double di\_accum = 0;

      for (size\_t j = ig[i]; j < ig[i + 1]; j++)

      {

         size\_t k = ig[i];

         size\_t v = ig[jg[j]];

         double ggl\_accum = 0, ggu\_accum = 0;

         while (k < j && v < ig[jg[j] + 1ll])

         {

            if (jg[k] > jg[v]) v++;

            else if (jg[k] < jg[v]) k++;

            else

            {

               ggl\_accum += ggl[k] \* ggu[v];

               ggu\_accum += ggl[v] \* ggu[k];

               k++;

               v++;

            }

         }

         ggl[j] = (mat.ggl[j] - ggl\_accum) / di[jg[j]];

         ggu[j] = (mat.ggu[j] - ggu\_accum) / di[jg[j]];

         di\_accum += ggl[j] \* ggu[j];

      }

      di[i] = sqrt(mat.di[i] - di\_accum);

   }

}

void LU::Resize(size\_t diSize, size\_t luSize) {

   di.resize(diSize);

   ggl.resize(luSize);

   ggu.resize(luSize);

}

/// <summary>

/// Умножение нижней матрицы L на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> LU::LMultToVec(const std::vector<double>& vec) const

{

   std::vector<double> ans(vec.size());

   return LMultToVec(vec, ans);

}

/// <summary>

/// Умножение нижней матрицы L на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& LU::LMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

   if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   for (uint16\_t i = 0; i < ans.size(); i++)

   {

      // Умножаем диагональ

      ans[i] = di[i] \* vec[i];

      // Умножаем нижний треугольник

      for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1ll]; j++)

      {

         ans[i] += ggl[j] \* vec[parent->jg[j]];

      }

   }

   return ans;

}

std::vector<double> LU::LTranspMultToVec(const std::vector<double>& vec) const

{

   std::vector<double> ans(vec.size());

   return LTranspMultToVec(vec, ans);

}

std::vector<double>& LU::LTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

   if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   for (uint16\_t i = 0; i < ans.size(); i++)

   {

      // Умножаем диагональ

      ans[i] = di[i] \* vec[i];

      // Умножаем на верхний треугольник с данными нижнего

      for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1ll]; j++)

      {

         ans[parent->jg[j]] += ggl[j] \* vec[i];

      }

   }

   return ans;

}

/// <summary>

/// Умножение верхней матрицы U на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> LU::UMultToVec(const std::vector<double>& vec) const

{

   std::vector<double> ans(vec.size());

   return UMultToVec(vec, ans);

}

/// <summary>

/// Умножение верхней матрицы U на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& LU::UMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

   if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   for (uint16\_t i = 0; i < ans.size(); i++)

   {

      // Умножаем диагональ

      ans[i] = di[i] \* vec[i];

      // Умножаем верхний треугольник

      for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1ll]; j++)

      {

         ans[parent->jg[j]] += ggu[j] \* vec[i];

      }

   }

   return ans;

}

std::vector<double> LU::UTranspMultToVec(const std::vector<double>& vec) const

{

   std::vector<double> ans(vec.size());

   return UTranspMultToVec(vec, ans);

}

std::vector<double>& LU::UTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

   if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   for (uint16\_t i = 0; i < ans.size(); i++)

   {

      // Умножаем диагональ

      ans[i] = di[i] \* vec[i];

      // Умножаем нижний треугольник с данными верхнего треугольника

      for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1ll]; j++)

      {

         ans[i] += ggu[j] \* vec[parent->jg[j]];

      }

   }

   return ans;

}

/// <summary>

/// Решение слау вида Lx = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& LU::LSlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

   if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   size\_t size = x.size();

   for (size\_t i = 0; i < size; i++)

   {

      x[i] = 0;

      for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

      {

         x[i] += x[parent->jg[j]] \* ggl[j];

      }

      x[i] = (right[i] - x[i]) / di[i];

   }

   return x;

}

/// <summary>

/// Решение слау вида Lx = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> LU::LSlauSolve(const std::vector<double>& right) const

{

   std::vector<double> x(right.size());

   return LSlauSolve(right, x);

}

std::vector<double>& LU::LTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

   if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   size\_t size = x.size();

   for (size\_t i = 0; i < size; i++)

      x[i] = 0;

   for (size\_t it = 0, i = size - 1; it < size; it++, i--)

   {

      x[i] = (right[i] - x[i]) / di[i];

      for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

      {

         x[parent->jg[j]] += ggl[j] \* x[i];

      }

   }

   return x;

}

std::vector<double> LU::LTranspSlauSolve(const std::vector<double>& right) const

{

   std::vector<double> x(right.size());

   return LTranspSlauSolve(right, x);

}

/// <summary>

/// Решение слау вида Ux = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& LU::USlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

   if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   size\_t size = x.size();

   for (size\_t i = 0; i < size; i++)

      x[i] = 0;

   for (size\_t it = 0, i = size - 1; it < size; it++, i--)

   {

      x[i] = (right[i] - x[i]) / di[i];

      for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

      {

         x[parent->jg[j]] += ggu[j] \* x[i];

      }

   }

   return x;

}

/// <summary>

/// Решение слау вида Ux = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> LU::USlauSolve(const std::vector<double>& right) const

{

   std::vector<double> x(right.size());

   return USlauSolve(right, x);

}

std::vector<double>& LU::UTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

   if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

   if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

   size\_t size = x.size();

   for (size\_t i = 0; i < size; i++)

   {

      x[i] = 0;

      for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

      {

         x[i] += x[parent->jg[j]] \* ggu[j];

      }

      x[i] = (right[i] - x[i]) / di[i];

   }

   return x;

}

std::vector<double> LU::UTranspSlauSolve(const std::vector<double>& right) const

{

   std::vector<double> x(right.size());

   return UTranspSlauSolve(right, x);

}

Файл IterSolvers.cpp:

#include "../Headers/IterSolvers.h"

using namespace std;

namespace Vec {

   inline double Scalar(const vector<double>& l, const vector<double>& r) {

      if (l.size() != r.size()) throw runtime\_error("Размеры векторов не совпадают");

      double res = 0.0;

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

      {

         res += l[i] \* r[i];

      }

      return res;

   }

   // l or r may be similar vectors to ans

   inline void Mult(const vector<double>& l, const vector<double>& r, vector<double>& ans) {

      if (ans.size() != l.size() || ans.size() != r.size()) throw runtime\_error("Ошибка: размеры векторов должны совпадать.");

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

      {

         ans[i] = l[i] \* r[i];

      }

   }

   inline vector<double> Mult(const vector<double>& l, const vector<double>& r) {

      if (r.size() != l.size()) throw runtime\_error("Ошибка: размеры векторов должны совпадать.");

      vector<double> ans(l.size());

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

      {

         ans[i] = l[i] \* r[i];

      }

      return ans;

   }

}

namespace IterSolvers {

   double minEps = 1e-8;

   size\_t maxIter = 2000;

   bool globalDebugOutput = false;

   std::vector<double>\* \_tmp1 = nullptr, \* \_tmp2 = nullptr,

      \* \_tmp3 = nullptr, \* \_tmp4 = nullptr, \* \_tmp5 = nullptr, \* \_tmp6 = nullptr;

   LU\* \_lu\_mat = nullptr;

   inline void VecInit(vector<double>\*& vec, size\_t size) {

      if (vec == nullptr)

      {

         vec = new vector<double>(size);

      } else if (vec->size() != size)

      {

         vec->resize(size);

      }

   }

   namespace MSG\_Assimetric {

      void Init\_Default(size\_t size) {

         VecInit(\_tmp1, size); // Массив для вектора r метода

         VecInit(\_tmp2, size); // Массив для вектора z

         VecInit(\_tmp3, size); // Массив для вектора t

         VecInit(\_tmp4, size); // Массив для временного вектора

      }

      size\_t Default(const SparseMatrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

         size\_t size = x.size();

         vector<double>& tmp = \*\_tmp4;

         vector<double>& r = \*\_tmp1;         // r0 = A^t \* (f - A \* x)

         A.MultToVec(x, tmp);

         for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

         A.TranspMultToVec(tmp, r);

         vector<double>& z = \*\_tmp2;

         z = r;                              // z0

         vector<double>& t = \*\_tmp3;

         double rPrevScalar = Vec::Scalar(r, r);

         double rScalar = 0;

         double a = 0;                       // alpha\_k

         double b = 0;                       // beta\_k

         double normF = Vec::Scalar(f, f);   // (f, f)

         eps = DBL\_MAX;

         size\_t iter = 0;

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

         {

            A.MultToVec(z, tmp);

            A.TranspMultToVec(tmp, t);                   // t = A^t \* A \* z\_k-1

            a = rPrevScalar / Vec::Scalar(t, z);         // a\_k = (r\_k-1, r\_k-1) / (t\_k-1, z\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               x[i] += a \* z[i];                         // x\_k = x\_k-1 + a \* z\_k-1

               r[i] -= a \* t[i];                         // r\_k = r\_k-1 - a \* t\_k-1

            }

            rScalar = Vec::Scalar(r, r);

            b = rScalar / rPrevScalar;                   // b = (r\_k, r\_k) / (r\_k-1, r\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               z[i] = r[i] + b \* z[i];                   // z\_k = r\_k + b \* z\_k-1

            }

            rPrevScalar = rScalar;

            eps = sqrt(rScalar / normF);

            // Выводим на то же место, что и раньше (со сдвигом каретки)

            if (debugOutput)

            {

               cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

            }

            if (isinf(eps))

            {

               break;

            }

         }

         if (debugOutput)

         {

            cout << endl;

            if (isinf(eps))

            {

               cout << "Выход по переполнению метода" << endl << endl;

            } else if (iter > maxIter)

            {

               cout << "Выход по числу итераций" << endl << endl;

            } else

            {

               cout << "Выход по относительной невязке" << endl << endl;

            }

         }

         return iter - 1;

      }

      void Init\_DiagPrecond(size\_t size) {

         Init\_Default(size);

         VecInit(\_tmp5, size); // Массив для вектора D

      }

      size\_t DiagPrecond(const SparseMatrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

         size\_t size = x.size();

         vector<double>& D = \*\_tmp5;         // D = обратный корень от диагонали матрицы

         for (uint16\_t i = 0; i < size; i++) D[i] = 1 / sqrt(A.di[i]);

         for (uint16\_t i = 0; i < size; i++) x[i] /= D[i];     // local\_x

         vector<double>& r = \*\_tmp1;             // r = U^-t \* A^t \* L^-t \* L^-1 (f - A \* x)

         vector<double>& tmp = \*\_tmp4;

         A.MultToVec(x, tmp);

         for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

         Vec::Mult(D, tmp, tmp);

         Vec::Mult(D, tmp, tmp);

         A.TranspMultToVec(tmp, r);

         Vec::Mult(D, r, r);

         vector<double>& z = \*\_tmp2;

         z = r;

         vector<double>& t = \*\_tmp3;             // t = U^-1 \* A^t \* L^-t \* L^-1 \* A \* U^-1 \* z

         double rPrevScalar = Vec::Scalar(r, r);         // (r\_k-1, r\_k-1)

         double rScalar = 0;

         double a = 0;                       // alpha\_k,

         double b = 0;                       // beta\_k

         double normF = Vec::Scalar(f, f);   // ||f||

         eps = sqrt(rPrevScalar / normF);

         size\_t iter;

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

         {

            Vec::Mult(D, z, t);

            A.MultToVec(t, tmp);

            Vec::Mult(D, tmp, tmp);

            Vec::Mult(D, tmp, tmp);

            A.TranspMultToVec(tmp, t);

            Vec::Mult(D, t, t);

            a = rPrevScalar / Vec::Scalar(t, z);         // a\_k = (r\_k-1, r\_k-1) / (t\_k-1, z\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               x[i] += a \* z[i];                         // local\_x\_k = local\_x\_k-1 + a \* z\_k-1

               r[i] -= a \* t[i];                         // r\_k = r\_k-1 - a \* t\_k-1

            }

            rScalar = Vec::Scalar(r, r);

            b = rScalar / rPrevScalar;                   // b = (r\_k, r\_k) / (r\_k-1, r\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               z[i] = r[i] + b \* z[i];                   // z\_k = r\_k + b \* z\_k-1

            }

            rPrevScalar = rScalar;

            eps = sqrt(rPrevScalar / normF);

            // Выводим на то же место, что и раньше (со сдвигом каретки)

            if (debugOutput)

            {

               cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

            }

            if (isinf(eps))

            {

               break;

            }

         }

         Vec::Mult(D, x, x);        // x = U^-1 \* local\_x

         if (debugOutput)

         {

            cout << endl;

            if (isinf(eps))

            {

               cout << "Выход по переполнению метода" << endl << endl;

            } else if (iter > maxIter)

            {

               cout << "Выход по числу итераций" << endl << endl;

            } else

            {

               cout << "Выход по относительной невязке" << endl << endl;

            }

         }

         return iter - 1;

      }

      void Init\_LuPrecond(size\_t diSize, const SparseMatrix& A) {

         VecInit(\_tmp1, diSize); // Массив для вектора r метода

         VecInit(\_tmp2, diSize); // Массив для вектора z

         VecInit(\_tmp3, diSize); // Массив для вектора t

         VecInit(\_tmp4, diSize); // Массив для временного вектора

         VecInit(\_tmp5, diSize); // Массив для вектора local\_x

         if (\_lu\_mat == nullptr)

         {

            \_lu\_mat = new LU(A);

         } else

         {

            \_lu\_mat->MakeLuFor(A);

         }

      }

      size\_t LuPrecond(const SparseMatrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

         size\_t size = x.size();

         LU& lu = \*\_lu\_mat;                        // неполное LU(sq) разложение для матрицы A

         lu.MakeLuFor(A);

         vector<double>& local\_x = \*\_tmp5;         // local\_x

         lu.UMultToVec(x, local\_x);

         vector<double>& r = \*\_tmp1;               // r = U^-t \* A^t \* L^-t \* L^-1 (f - A \* x)

         vector<double>& tmp = \*\_tmp4;

         A.MultToVec(x, r);

         for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

         lu.LSlauSolve(r, tmp);

         lu.LTranspSlauSolve(tmp, r);

         A.TranspMultToVec(r, tmp);

         lu.UTranspSlauSolve(tmp, r);

         vector<double>& z = \*\_tmp2;

         z = r;

         vector<double>& t = \*\_tmp3;               // t = U^-1 \* A^t \* L^-t \* L^-1 \* A \* U^-1 \* z

         double rPrevScalar = Vec::Scalar(r, r);   // (r\_k-1, r\_k-1)

         double rScalar = 0;

         double a = 0;                             // alpha\_k,

         double b = 0;                             // beta\_k

         double normF = Vec::Scalar(f, f);         // ||f||

         eps = sqrt(rPrevScalar / normF);

         size\_t iter;

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

         {

            lu.USlauSolve(z, tmp);

            A.MultToVec(tmp, t);

            lu.LSlauSolve(t, tmp);

            lu.LTranspSlauSolve(tmp, t);

            A.TranspMultToVec(t, tmp);

            lu.UTranspSlauSolve(tmp, t);

            a = rPrevScalar / Vec::Scalar(t, z);         // a\_k = (r\_k-1, r\_k-1) / (t\_k-1, z\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               local\_x[i] += a \* z[i];                   // local\_x\_k = local\_x\_k-1 + a \* z\_k-1

               r[i] -= a \* t[i];                         // r\_k = r\_k-1 - a \* t\_k-1

            }

            rScalar = Vec::Scalar(r, r);

            b = rScalar / rPrevScalar;                   // b = (r\_k, r\_k) / (r\_k-1, r\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               z[i] = r[i] + b \* z[i];                   // z\_k = r\_k + b \* z\_k-1

            }

            rPrevScalar = rScalar;

            eps = sqrt(rPrevScalar / normF);

            // Выводим на то же место, что и раньше (со сдвигом каретки)

            if (debugOutput)

            {

               cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

            }

            if (isinf(eps))

            {

               break;

            }

         }

         lu.USlauSolve(local\_x, x); // x = U^-1 \* local\_x

         if (debugOutput)

         {

            cout << endl;

            if (isinf(eps))

            {

               cout << "Выход по переполнению метода" << endl << endl;

            } else if (iter > maxIter)

            {

               cout << "Выход по числу итераций" << endl << endl;

            } else

            {

               cout << "Выход по относительной невязке" << endl << endl;

            }

         }

         return iter - 1;

      }

   }

   namespace LOS {

      size\_t resetIter = 10;

      void Init\_Default(size\_t size) {

         VecInit(\_tmp1, size); // Массив для вектора r метода

         VecInit(\_tmp2, size); // Массив для вектора z

         VecInit(\_tmp3, size); // Массив для вектора p

         VecInit(\_tmp4, size); // Массив для вектора Ar

      }

      size\_t Default(const SparseMatrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

         auto size = x.size();

         vector<double>& r = \*\_tmp1;

         A.MultToVec(x, r);

         for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i]; // r0 = f - A \* x

         vector<double>& z = \*\_tmp2;         // z0

         z = r;

         vector<double>& p = \*\_tmp3;         // p0 = A \* z0

         A.MultToVec(z, p);

         vector<double>& Ar = \*\_tmp4;        // A \* r

         double ppScalar;

         double nev = Vec::Scalar(r, r);

         double ffScalar = Vec::Scalar(f, f);

         eps = nev / ffScalar;

         double a;                  // alpha

         double b;                  // beta

         size\_t iter;

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

         {

            ppScalar = Vec::Scalar(p, p);     // (p\_k-1, p\_k-1)

            a = Vec::Scalar(p, r) / ppScalar;    // (p\_k-1, r\_k-1) / (p\_k-1, p\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               x[i] += a \* z[i];                   // [x\_k] = [x\_k-1] + a\*z\_k-1

               r[i] -= a \* p[i];                   // [r\_k] = [r\_k-1] - a\*p\_k-1

            }

            A.MultToVec(r, Ar);      // A \* r\_k

            b = -Vec::Scalar(p, Ar) / ppScalar; // b = - (p\_k-1, A \* r\_k) / (p\_k-1, p\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               z[i] = r[i] + b \* z[i];             // [z\_k] = r\_k + b \* [z\_k-1]

               p[i] = Ar[i] + b \* p[i];            // [p\_k] = A \* r\_k + b \* [p\_k-1]

            }

            if (iter % resetIter == 0)

            {

               A.MultToVec(x, r);

               for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

               z = r;

               A.MultToVec(z, p);

            }

            nev = Vec::Scalar(r, r);

            eps = sqrt(nev / ffScalar);

            // Выводим на то же место, что и раньше (со сдвигом каретки)

            if (debugOutput)

            {

               //cout << format("Итерация: {0:<10} относительная невязка: {1:<15.3e}\n", iter, eps);

               cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

            }

            if (isinf(eps))

            {

               break;

            }

         }

         if (debugOutput)

         {

            cout << endl;

            if (isinf(eps))

            {

               cout << "Выход по переполнению метода" << endl << endl;

            } else if (iter > maxIter)

            {

               cout << "Выход по числу итераций" << endl << endl;

            } else

            {

               cout << "Выход по относительной невязке" << endl << endl;

            }

         }

         return iter - 1;

      }

      void Init\_DiagPrecond(size\_t size) {

         VecInit(\_tmp1, size); // Массив для вектора r метода

         VecInit(\_tmp2, size); // Массив для вектора z

         VecInit(\_tmp3, size); // Массив для вектора p

         VecInit(\_tmp4, size); // Массив для вектора Ar

         VecInit(\_tmp5, size); // Массив для вектора D

         VecInit(\_tmp6, size); // Массив для вектора tmp

      }

      size\_t DiagPrecond(const SparseMatrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

         auto size = x.size();

         vector<double>& D = \*\_tmp5;               // обратный корень от диагонали матрицы

         for (uint16\_t i = 0; i < size; i++) D[i] = 1 / sqrt(A.di[i]);

         vector<double>& r = \*\_tmp1;               // r0 = L^-1 \* (f - A \* x)

         A.MultToVec(x, r);

         for (auto i = 0; i < size; i++) r[i] = f[i] - r[i];

         Vec::Mult(D, r, r);

         vector<double>& z = \*\_tmp2;               // z0 = U^-1 \* r

         Vec::Mult(D, r, z);

         vector<double>& p = \*\_tmp3;               // p0 = L^-1 \* A \* z0

         A.MultToVec(z, p);

         Vec::Mult(D, p, p);

         vector<double>& Ar = \*\_tmp4;              // Ar = L^-1 \* A \* U^-1 \* r

         vector<double>& tmp = \*\_tmp6;

         double ppScalar;

         double nev = Vec::Scalar(r, r);

         double ffScalar = Vec::Scalar(f, f);

         eps = nev / ffScalar;

         double a;                  // alpha

         double b;                  // beta

         size\_t iter;

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

         {

            ppScalar = Vec::Scalar(p, p);          // (p\_k-1, p\_k-1)

            a = Vec::Scalar(p, r) / ppScalar;      // (p\_k-1, r\_k-1) / (p\_k-1, p\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               x[i] += a \* z[i];                   // [x\_k] = [x\_k-1] + a\*z\_k-1

               r[i] -= a \* p[i];                   // [r\_k] = [r\_k-1] - a\*p\_k-1

            }

            Vec::Mult(D, r, tmp);

            A.MultToVec(tmp, Ar);

            Vec::Mult(D, Ar, Ar);                  // Ar = L^-1 \* A \* U^-1 \* r

            b = -Vec::Scalar(p, Ar) / ppScalar;    // b = - (p\_k-1, L^-1 \* A \* U^-1 \* r\_k) / (p\_k-1, p\_k-1)

            Vec::Mult(D, r, tmp);                  // tmp = U^-1 \* r\_k

            for (uint16\_t i = 0; i < size; i++)

            {

               z[i] = tmp[i] + b \* z[i];            // [z\_k] = U^-1 \* r\_k + b \* [z\_k-1]

               p[i] = Ar[i] + b \* p[i];             // [p\_k] = A \* r\_k + b \* [p\_k-1]

            }

            if (iter % resetIter == 0)

            {

               A.MultToVec(x, r);

               for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

               Vec::Mult(D, r, r);

               Vec::Mult(D, r, z);

               A.MultToVec(z, p);

               Vec::Mult(D, p, p);

            }

            nev = Vec::Scalar(r, r);

            eps = sqrt(nev / ffScalar);

            // Выводим на то же место, что и раньше (со сдвигом каретки)

            if (debugOutput)

            {

               //cout << format("Итерация: {0:<10} относительная невязка: {1:<15.3e}\n", iter, eps);

               cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

            }

            if (isinf(eps))

            {

               break;

            }

         }

         if (debugOutput)

         {

            cout << endl;

            if (isinf(eps))

            {

               cout << "Выход по переполнению метода" << endl << endl;

            } else if (iter > maxIter)

            {

               cout << "Выход по числу итераций" << endl << endl;

            } else

            {

               cout << "Выход по относительной невязке" << endl << endl;

            }

         }

         return iter - 1;

      }

      void Init\_LuPrecond(size\_t diSize, const SparseMatrix& A) {

         VecInit(\_tmp1, diSize); // Массив для вектора r метода

         VecInit(\_tmp2, diSize); // Массив для вектора z

         VecInit(\_tmp3, diSize); // Массив для вектора p

         VecInit(\_tmp4, diSize); // Массив для вектора Ar

         VecInit(\_tmp5, diSize); // Массив для вектора tmp

         if (\_lu\_mat == nullptr)

         {

            \_lu\_mat = new LU(A);

         } else

         {

            \_lu\_mat->MakeLuFor(A);

         }

      }

      size\_t LuPrecond(const SparseMatrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

         auto size = x.size();

         LU& lu = \*\_lu\_mat;

         lu.MakeLuFor(A);

         vector<double>& tmp = \*\_tmp5;

         vector<double>& r = \*\_tmp1;               // r0 = L^-1 \* (f - A \* x)

         A.MultToVec(x, tmp);

         for (auto i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

         lu.LSlauSolve(tmp, r);

         vector<double>& z = \*\_tmp2;               // z0 = U^-1 \* r

         lu.USlauSolve(r, z);

         vector<double>& p = \*\_tmp3;               // p0 = L^-1 \* A \* z0

         A.MultToVec(z, tmp);

         lu.LSlauSolve(tmp, p);

         vector<double>& Ar = \*\_tmp4;              // Ar = L^-1 \* A \* U^-1 \* r

         double ppScalar;

         double nev = Vec::Scalar(r, r);

         double ffScalar = Vec::Scalar(f, f);

         eps = nev / ffScalar;

         double a;                  // alpha

         double b;                  // beta

         size\_t iter;

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

         {

            ppScalar = Vec::Scalar(p, p);          // (p\_k-1, p\_k-1)

            a = Vec::Scalar(p, r) / ppScalar;      // (p\_k-1, r\_k-1) / (p\_k-1, p\_k-1)

            for (uint16\_t i = 0; i < size; i++)

            {

               x[i] += a \* z[i];                   // [x\_k] = [x\_k-1] + a\*z\_k-1

               r[i] -= a \* p[i];                   // [r\_k] = [r\_k-1] - a\*p\_k-1

            }

            lu.USlauSolve(r, Ar);

            A.MultToVec(Ar, tmp);

            lu.LSlauSolve(tmp, Ar);                // Ar = L^-1 \* A \* U^-1 \* r

            //Vec::Mult(D, r, tmp);

            //A.MultToVec(tmp, Ar);

            //Vec::Mult(D, Ar, Ar);

            b = -Vec::Scalar(p, Ar) / ppScalar;    // b = - (p\_k-1, L^-1 \* A \* U^-1 \* r\_k) / (p\_k-1, p\_k-1)

            lu.USlauSolve(r, tmp);                 // tmp = U^-1 \* r\_k

            for (uint16\_t i = 0; i < size; i++)

            {

               z[i] = tmp[i] + b \* z[i];            // [z\_k] = U^-1 \* r\_k + b \* [z\_k-1]

               p[i] = Ar[i] + b \* p[i];             // [p\_k] = A \* r\_k + b \* [p\_k-1]

            }

            if (iter % resetIter == 0)

            {

               A.MultToVec(x, tmp);

               for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

               lu.LSlauSolve(tmp, r);

               lu.USlauSolve(r, z);

               A.MultToVec(z, tmp);

               lu.LSlauSolve(tmp, p);

            }

            nev = Vec::Scalar(r, r);

            eps = sqrt(nev / ffScalar);

            // Выводим на то же место, что и раньше (со сдвигом каретки)

            if (debugOutput)

            {

               //cout << format("Итерация: {0:<10} относительная невязка: {1:<15.3e}\n", iter, eps);

               cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

            }

            if (isinf(eps))

            {

               break;

            }

         }

         if (debugOutput)

         {

            cout << endl;

            if (isinf(eps))

            {

               cout << "Выход по переполнению метода" << endl << endl;

            } else if (iter > maxIter)

            {

               cout << "Выход по числу итераций" << endl << endl;

            } else

            {

               cout << "Выход по относительной невязке" << endl << endl;

            }

         }

         return iter - 1;

      }

   }

   void Destruct() noexcept {

      delete \_tmp1, \_tmp2, \_tmp3, \_tmp4, \_tmp5, \_tmp6;

      \_tmp1 = \_tmp2 = \_tmp3 = \_tmp4 = \_tmp5 = \_tmp6 = nullptr;

      delete \_lu\_mat;

      \_lu\_mat = nullptr;

   }

};