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

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:

$$-div\big(\lambda grad(u)\big) + \sigma\frac{\partial u}{\partial t} + \chi\frac{\partial^2 u}{\partial t^2} = f. \quad (1)$$

Область интегрирования: Ω,
Граница интегрирования: $S = S_1 \vee S_2 \vee S_3$,
Краевые условия:
- Первые: $u|_{S_1} = u_g$,
- Вторые: $\lambda\frac{\partial u}{\partial n}\Big|_{S_2} = \theta$,
- Третьи: $\lambda\frac{\partial u}{\partial n} + \beta\big(u - u_\beta\big) = 0$,

λ – коэффициент диффузии,
β – коэффициент теплообмена,
Начальные условия:
- Первые: $u_{t=t_0} = u^0$,
- Вторые: $\frac{\partial u}{\partial t}\Big|_{t=t_0} = u'^0$

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

$$-\frac{1}{r}\frac{\partial}{\partial r}\left(r\lambda\frac{\partial u}{\partial r}\right) - \frac{1}{r^2}\frac{\partial}{\partial \varphi}\left(\lambda\frac{\partial u}{\partial \varphi}\right) + \sigma\frac{\partial u}{\partial t} + \chi\frac{\partial^2 u}{\partial t^2} = f, \quad (2)$$

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

$$G_{ij} = \int_\Omega \lambda\left(\frac{\partial \psi_j}{\partial r}\frac{\partial \psi_i}{\partial r} + \frac{1}{r^2}\frac{\partial \psi_j}{\partial \varphi}\frac{\partial \psi_i}{\partial \varphi}\right) r dr d\varphi\,, \quad (2.1)$$

$$M_{ij} = \int_\Omega \gamma\psi_j\psi_i r dr d\varphi, \quad (2.2)$$

$$b_i = \int_\Omega f\psi_i r\, dr d\varphi, \quad (2.3)$$

здесь ψ_i – базисные функции конечномерного пространства, определяемые следующим образом:

$$\psi_1(r,\varphi) = R_1(r)\Phi_1(\varphi), \qquad \psi_2(r,\varphi) = R_2(r)\Phi_1(\varphi),$$
$$\psi_3(r,\varphi) = R_1(r)\Phi_2(\varphi), \qquad \psi_4(r,\varphi) = R_2(r)\Phi_2(\varphi),$$

а функции $R_i(r)$ и $\Phi_j(\varphi)$ определяются в свою очередь следующим способом:

$$\mathrm{R}_1(r) = \frac{r_p + h_r - r}{h_r}, \qquad \mathrm{R}_2(r) = \frac{r - r_p}{h_r},$$

$$\Phi_{1(r)} = \frac{\varphi_s + h_\varphi - \varphi}{h_\varphi}, \Phi_{2(r)} = \frac{\varphi - \varphi_s}{h_\varphi}.$$

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 η_i^j:

$$u(x,y,t) \approx u^{j-2}(x,y)\,\eta_2^j(t) + u^{j-1}(x,y)\,\eta_1^j(t) + u^j(x,y)\,\eta_0^j(t), \tag{3}$$

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

$$\eta_2^j(t) = \frac{(t - t_{j-1})(t - t_j)}{\Delta t_1 \Delta t}, \tag{4.1}$$

$$\eta_1^j(t) = -\frac{(t - t_{j-2})(t - t_j)}{\Delta t_1 \Delta t_0}, \tag{4.2}$$

$$\eta_0^j(t) = \frac{(t - t_{j-2})(t - t_{j-1})}{\Delta t \Delta t_0}, \tag{4.3}$$

где:

$$\Delta t = t_j - t_{j-2}, \qquad \Delta t_1 = t_{j-1} - t_{j-2}, \qquad \Delta t_0 = t_j - t_{j-1}$$

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

$$\left.\frac{d\eta_2^j(t)}{dt}\right|_{t=t_{j-1}} = -\frac{\Delta t_0}{\Delta t \Delta t_1}; \qquad \left.\frac{d^2\eta_2^j(t)}{dt^2}\right|_{t=t_{j-1}} = \frac{2}{\Delta t \Delta t_1};$$

$$\left.\frac{d\eta_1^j(t)}{dt}\right|_{t=t_{j-1}} = \frac{\Delta t_0 - \Delta t_1}{\Delta t_1 \Delta t_0}; \qquad \left.\frac{d^2\eta_1^j(t)}{dt^2}\right|_{t=t_{j-1}} = -\frac{2}{\Delta t_1 \Delta t_0};$$

$$\left.\frac{d\eta_0^j(t)}{dt}\right|_{t=t_{j-1}} = \frac{\Delta t_1}{\Delta t \Delta t_0}; \qquad \left.\frac{d^2\eta_0^j(t)}{dt^2}\right|_{t=t_{j-1}} = \frac{2}{\Delta t \Delta t_0}.$$

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

$$\left.\frac{\partial u}{\partial t}\right|_{t=t_{j-1}} \approx -\frac{\Delta t_0}{\Delta t \Delta t_1}\, u^{j-2} + \frac{\Delta t_0 - \Delta t_1}{\Delta t_1 \Delta t_0}\, u^{j-1} + \frac{\Delta t_1}{\Delta t \Delta t_0}\, u^{j},$$
$$\left.\frac{\partial^2 u}{\partial t^2}\right|_{t=t_{j-1}} \approx \frac{2}{\Delta t \Delta t_1}\, u^{j-2} - \frac{2}{\Delta t_1 \Delta t_0}\, u^{j-1} + \frac{2}{\Delta t \Delta t_0}\, u^{j},$$

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

$$\begin{gathered}\chi\left(\frac{2}{\Delta t \Delta t_1} u^{j-2} - \frac{2}{\Delta t_1 \Delta t_0} u^{j-1} + \frac{2}{\Delta t \Delta t_0} u^{j}\right) + \\ +\sigma\left(-\frac{\Delta t_0}{\Delta t \Delta t_1} u^{j-2} + \frac{\Delta t_0 - \Delta t_1}{\Delta t_1 \Delta t_0} u^{j-1} + \frac{\Delta t_1}{\Delta t \Delta t_0} u^{j}\right) - \\ -div\left(\lambda\, grad\left(u^{j-1}\right)\right) = f^{j-1}\end{gathered} \tag{5}$$

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

$$\begin{gathered}\frac{2}{\Delta t \Delta t_1} M^{\chi} q^{j-2} - \frac{2}{\Delta t_1 \Delta t_0} M^{\chi} q^{j-1} + \frac{2}{\Delta t \Delta t_0} M^{\chi} q^{j} - \\ -\frac{\Delta t_0}{\Delta t \Delta t_1} M^{\sigma} q^{j-2} + \frac{\Delta t_0 - \Delta t_1}{\Delta t_1 \Delta t_0} M^{\sigma} q^{j-1} + \frac{\Delta t_1}{\Delta t \Delta t_0} M^{\sigma} q^{j} + G q^{j-1} = b^{j-1}\end{gathered}$$

которое фактически является СЛАУ для вектора q^j с матрицей A и вектором правой части d следующего вида:

$$A = \frac{2}{\Delta t \Delta t_0} M^{\chi} + \frac{\Delta t_1}{\Delta t \Delta t_0} M^{\sigma},$$

$$d = b^{j-1} - \frac{2}{\Delta t \Delta t_1} M^{\chi} q^{j-2} + \frac{2}{\Delta t_1 \Delta t_0} M^{\chi} q^{j-1} + \frac{\Delta t_0}{\Delta t \Delta t_1} M^{\sigma} q^{j-2} - \frac{\Delta t_0 - \Delta t_1}{\Delta t_1 \Delta t_0} M^{\sigma} q^{j-1} - G q^{j-1}.$$

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

$$M^{\sigma}_{ij} = \int_{\Omega} \sigma\, \psi_j \psi_i r\, dr d\varphi,$$

$$M^{\chi}_{ij} = \int_{\Omega} \chi\, \psi_j \psi_i r\, dr d\varphi.$$

Использование трёхслойной явной схемы подразумевает, что значения q^0 и q^1 изначально известны. Первое значение (q^0) задаётся начальным условием, второе же значение можно находить двумя вариантами:

1) Используя начальное условие, аналогичное первому:

$$u|_{t=t_1} = u^1;$$

2) Используя приближённое значение на временном слое:

$$u^1 \approx u^0 + \frac{\partial u}{\partial t}\Big|_{t=t_0} (t - t_0)$$

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;
};
```

2. 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:
    a. Метод смежных градиентов для несимметричных матриц;
    b. Метод смежных градиентов с диагональным предобуславливанием;
    c. Метод смежных градиентов с LU-предобуславливанием;
    d. Метод ЛОС;
    e. Метод ЛОС с диагональным предобуславливанием;
    f. Метод ЛОС с LU-предобуславливанием;

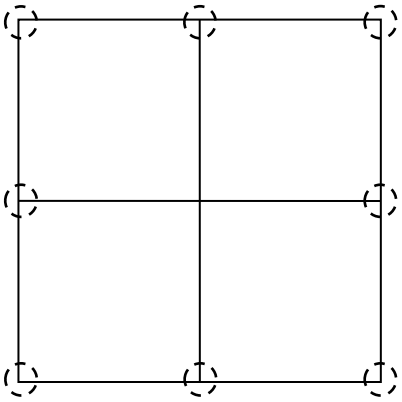
4. Модуль gaussian_quadrature – определяет методы численного интегрирования методом Гаусса для одно- и двухмерных функций. Включает в себя метод Гаусса по двум точкам и по четырём точкам.
5. Основной модуль программы, содержащий в себе:
    a. метод генерации портрета разреженной матрицы на основе входных данных;
    b. Методы построения локальных матриц, применяя методы численного интегрирования;
    c. Метод добавления локальных матриц к глобальной;
    d. Метод учёта первых краевых условий.

Тестирование программы

Для составления тестов для проверки работоспособности программы использовалось следующее уравнение:

$$-\frac{1}{r}\frac{\partial}{\partial r}\left(r\lambda\frac{\partial u}{\partial r}\right)-\frac{1}{r^2}\frac{\partial}{\partial \varphi}\left(\lambda\frac{\partial u}{\partial \varphi}\right)+\sigma\frac{\partial u}{\partial t}+\chi\frac{\partial^2 u}{\partial t^2}=f \quad (2)$$

Пространственная область исследования представляет из себя квадрат размером 6x6 с 9 узлами:



Базовые тесты для проверки работоспособности программы

1. Тестирование на функции $u = t$

$u = t$	$\lambda = 1$	$\sigma = 1$	$\chi = 1$
$f = 1$			

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

2. Тестирование на функции $u = t + r$

$u = t + r$	$\lambda = r$	$\sigma = 1$	$\chi = 1$
$f = -1$			

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

3. Тестирование на функции $u = t^2$

$u = t^2$	$\lambda = 1$	$\sigma = 1$	$\chi = 1$
$f = 2t + 2$			

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

4. Тестирование на функции $u = t^3$

$u = t^3$	$\lambda = 1$	$\sigma = 1$	$\chi = 1$
$f = 3t^2 + 6t$			

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:

$$\frac{\| u^* - u^h \|}{\| u^* - u^{\frac{h}{2}} \|} \approx 2^k$$

5. Тестирование на функции $u = e^t$

$u = e^t$	$\lambda = 1$	$\sigma = 1$	$\chi = 1$
$f = 2e^t$			

Результаты при $t = 4$:

Шаг	Норма погрешности	2^k
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[] = {
      -0.57735026918962576450914878780,    // -1 / sqrt(3)
      0.57735026918962576450914878780,     //  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[] = {
        0.34785484513745385737306394922 2,
        0.65214515486254614262693605077 8,
        0.65214515486254614262693605077 8,
        0.34785484513745385737306394922 2
    };
    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 строке/столбце
    /// <para> Помимо этого первый элемент i строки можно найти как ggl[ig[i]] </para>
    /// <para> Пример массива для матрицы 3x3: </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 3x3: </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 3x3: </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 3x3: </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 3x3: </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;
}
};
```