|  |  |  |
| --- | --- | --- |
| в | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра прикладной математики | | |
| Курсовой проект | | |
| по дисциплине «Численные методы» | | |
| **Метод конечных элементов решения краевых задач  эллиптического уравнения** | | |
|  | | |
|  |  |  |
|  | Группа ПМ-01 | самсонов семён |
|  |  |  |
|  | Вариант 65 |  |
|  |  |  |
|  | Преподаватель | Персова Марина Геннадьевна |
|  |  |  |
|  |  |  |
| Новосибирск, 2022 | | |

# Задание

МКЭ для двумерной краевой задачи для эллиптического уравнения в полярной системе координат. Базисные функции билинейные на прямоугольниках. Краевые условия всех типов. Коэффициент разложить по билинейным базисным функциям. Матрицу СЛАУ генерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

# Постановка задачи

Решаемое уравнение в общем виде:

Область интегрирования:

Граница интегрирования:

Краевые условия:

- Первые:

- Вторые:

- Третьи:

- коэффициент диффузии,

- коэффициент теплообмена.

Согласно условиям варианта, в задаче используется полярная система координат . Для неё формулы операторов уравнения определяются следующим видом:

А само уравнение в таком случае будет выглядеть следующим образом:

Его невязка будет иметь следующий вид:

Потребуем, чтобы эта невязка была ортогональна (в смысле скалярного произведения пространства ) некоторому пространству функций , которое мы будем называть **пространством пробных функций**, т.е.:

Воспользуемся формулой Грина:

Для данного варианта:

Выражая из неё



и подставляя в исходное, получаем:

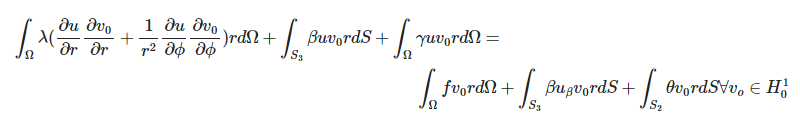
Интеграл по границе разложим по границам , и с учётом краевых условий:

Поскольку на границе краевыми условиями не определяется значение , слагаемое следует исключить из уравнения, потребовав, чтобы пространство пробных функций содержало только функции, которые принимают нулевые значения на границе . Поэтому в качестве выберем - пространство пробных функций , которые на границе удовлетворяют нулевым первым краевым условиям. С учётом этого, уравнение принимает вид:

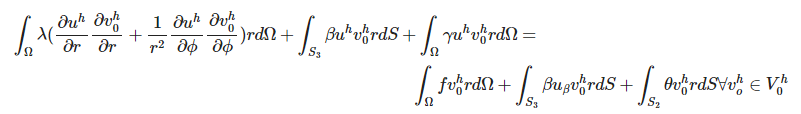
# Конечномерная аппроксимация

При построении конечноэлеменных аппроксимаций по методу Галеркина пространства и заменяются конечномерными пространстами и . При этом чаще всего в МКЭ функции из этих пространств являются элементами одного и того же конечномерного пространства , которое мы всегда будем определять как линейное пространство, натянутое на базисные функции . При этом функции являются набором финитных кусочно-полиномиальных функций, а приближённое решение , полученное как линейная комбинация .

Получим аппроксимацию уравнения Галеркина на конечномерных пространствах и . Исходное уравнение:

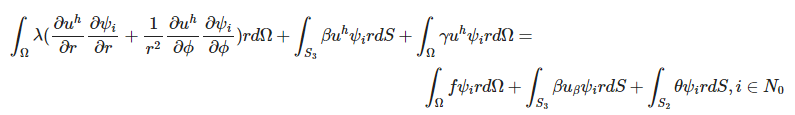


Для аппроксимации заменим функцию аппроксимирующей её функцией , а функцию - функцией :



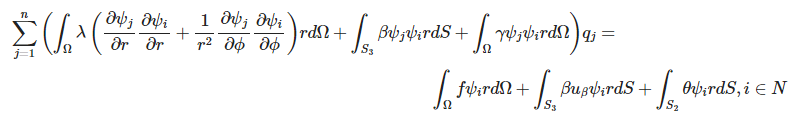
Поскольку любая функция может быть представлена в виде линейной комбинации:

вариационное уравнение эквивалентно следующей системе:



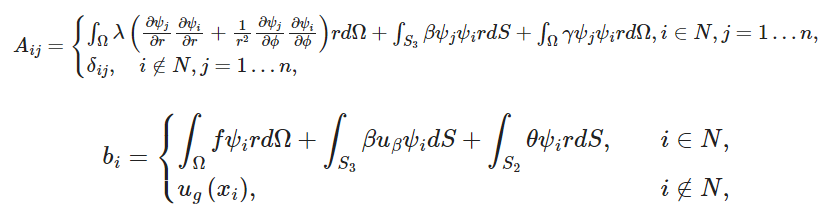
Таким образом, МКЭ-решение удовлетворяет полученной системе уравнений. Поскольку , оно может быть представлено в виде линейной комбинации базисных функций пространства :

Подставляя это выражение в предыдущее уравнение, получаем СЛАУ для компонент вектора весов с индексами :



При решении краевой задачи с использованием базисных функций, принимающих нулевые значения во всех узлах сетки, кроме одного, конечноэлементная СЛАУ для вектора весов может быть записана в матричном виде:

Где компоненты матрицы и вектора определяются соотношениями:



в которых - символ Кронекера ().

Без учёта краевых условий (их мы будем накладывать после того, как матрица была посчитана), матрица является суммой матриц жёсткости и массы :

# Базисные функции

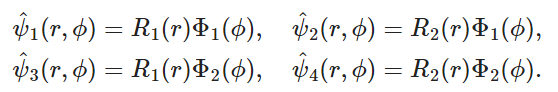
В данной задаче мы будем работать с билинейными базисными функциями на прямоугольниках. Для начала разобьём области интегрирования на прямоугольники :

Ячейки этой сетки строятся в виде прямого (декартова) произведения независимых друг от друга одномерных сеток .

Билинейные базисные функции определяются следующим образом. На отрезке задаются две одномерные линейные функции:

Аналогично, на отрезке задаются две одномерные линейные функции:

Локальные базисные функции на конечном элементе представляются в виде произведения линейный функций:



Очевидно, что функция равна единице в узле и нулю во всех остальных узлах ; остальные функции аналогично равны 1 только в соответствующем им узле и нулю во всех остальных.

# Вычисление локальных матриц

Локальные матрицы - матрицы, являющиеся частью глобальных матриц, и вычисляющиеся на одном конечном элементе.

Компоненты локальных матриц и будут вычисляться следующим образом:

Учитывая разбиение области интегрирования на прямоугольники , компоненты локальных матриц принимают следующий вид:

При чём, - усреднённое на прямоугольнике значение , а .

Перед тем, как записать аналитические выражения для вычисления элементов локальной матрицы , вычислим вспомогательные интегралы:

Теперь вычислим компоненты матрицы жёсткости:

Перед тем, как записать аналитические выражения для вычисления элементов локальной матрицы , вычислим вспомогательные интегралы:

Вычислим компоненты локальной матрицы массы:

Локальные компоненты вектора будут вычисляться следующим образом:

Вместо того, чтобы считать интегралы, будем вычислять с учётом того, что функция на конечном элементе представлена в виде разложения по базисным функциям , где - число локальных базисных функций конечного элемента . Тогда локальный вектор может быть выичслен через матрицу , являющейся матрицей массы с параметром :

# Учёт краевых условий

Для учёта *третьего* *краевого условия*, будем считать локальные матрицу и вектор по грани

Если краевое задано на грани, лежащей вдоль оси :

Если краевое задано на грани, лежащей вдоль оси :

Для учёта *второго краевого условия*, будем считать локальный вектор правой части по грани:

Если краевое условие задано вдоль оси :

Если краевое условие задано вдоль оси :

*Первое краевое условие* будет учитываться по следующему принципу:

Если краевое условие задано для узла, то элемент матрицы заменяется значением , все остальные элементы строки матрицы зануляются (), а в позицию вектора ставится значение .

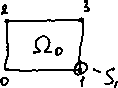
# Тестирование

## Простейший тест на константной функции и одним краевым первого рода

Точная функция: ,

Значение функций: , .

Область интегрирования:



Краевые условия: первого рода в вершине (1).

Координаты узлов:

0: (1, 1),  
1: (3, 1),  
2: (1, 3),  
3: (3, 3)

Прямоугольники:

0: 0 1 2 3 | 0

Результаты тестирования:

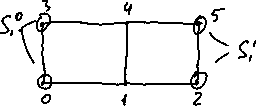
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № узла | Точное | Полученное | Вектор погрешности | Погрешность |
| 0 | 1 | 1.00000000000000 | 0 | 0 |
| 1 | 1 | 1.00000000000000 | 0 |
| 2 | 1 | 1.00000000000000 | 0 |
| 3 | 1 | 1.00000000000000 | 0 |

## Тест на линейной функции

Точная функция:

Значения функций:

Область интегрирования:



Координаты узлов:

0: (1, 1),  
1: (2, 1),  
2: (3, 1),  
3: (1, 4),  
4: (2, 4),  
5: (3, 4).

Прямоугольники:

0: 0 1 3 4 | 0,  
1: 1 2 4 5 | 0.

Краевые условия:

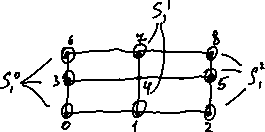
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № узла | Точное | Полученное | Вектор погрешности | Погрешность |
| 0 | 1 | 1.00000000000000 | 0 | 0.0104 |
| 1 | 2 | 1.98958333333333 | 0.0104166667 |
| 2 | 3 | 3.00000000000000 | 0 |
| 3 | 1 | 1.00000000000000 | 0 |
| 4 | 2 | 1.98958333333333 | 0.0104166667 |
| 5 | 3 | 3.00000000000000 | 0 |

## Тест на сходимость при дроблении сетки

Точная функция:

Значения функций:

Первая область интегрирования:



Координаты узлов:

0: (1, 1),  
1: (4, 1),  
2: (7, 1),  
3: (1, 3),   
4: (4, 3),  
5: (7, 3),   
6: (1, 5),   
7: (4, 5),  
8: (7, 5).

Прямоугольники:

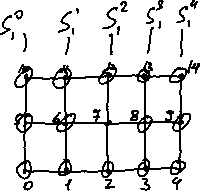
0: 0 1 3 4 | 0,  
1: 1 2 4 5 | 0,  
2: 3 4 6 7 | 0,  
3: 4 5 7 8 | 0.

Краевые условия:

Полученные значения:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № узла | Точное | Полученное | Вектор погрешности | Погрешность |
| 0 | 1 | 1 | 0 | 0.087129352 |
| 1 | 4 | 4 | 0 |
| 2 | 7 | 7 | 0 |
| 3 | 1 | 1 | 0 |
| 4 | 4 | 3.91287064840325 | 0.087129352 |
| 5 | 7 | 7 | 0 |
| 6 | 1 | 1 | 0 |
| 7 | 4 | 4 | 0 |
| 8 | 7 | 7 | 0 |

Вторая область интегрирования:



Координаты узлов:

0: (1, 1),  
1: (2.5, 1),  
2: (4, 1),  
3: (5.5, 1),  
4: (7, 1),  
5: (1, 3),  
6: (2.5, 3),  
7: (4, 3),  
8: (5.5, 3),  
9: (7, 3),  
10: (1, 5),  
11: (2.5, 5),  
12: (4, 5),  
13: (5.5, 5),  
14: (7, 5).

Прямоугольники:

0: 0 1 5 6 | 0,  
1: 1 2 6 7 | 0,  
2: 2 3 7 8 | 0,  
3: 3 4 8 9 | 0,  
4: 5 6 10 11 | 0,  
5: 6 7 11 12 | 0,  
6: 7 8 12 13 | 0,  
7: 8 9 13 14 | 0.

Краевые условия:

Полученные значения:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № узла | Точное | Полученное | Вектор погрешности | Погрешность |
| 0 | 1 | 1 | 0 | 0.0064433352 |
| 1 | 2.5 | 2.5 | 0 |
| 2 | 4 | 4 | 0 |
| 3 | 5.5 | 5.5 | 0 |
| 4 | 7 | 7 | 0 |
| 5 | 1 | 1 | 0 |
| 6 | 2.5 | 1 | 0 |
| 7 | 4 | 3.99355666471265 | 0.0064433352 |
| 8 | 5.5 | 4 | 0 |
| 9 | 7 | 5.5 | 0 |
| 10 | 1 | 7 | 0 |
| 11 | 2.5 | 1 | 0 |
| 12 | 4 | 1 | 0 |
| 13 | 5.5 | 2.5 | 0 |
| 14 | 7 | 4 | 0 |

Отношение погрешностей:

# Код программы

Файл main.cpp:

#include <iostream>

#include <fstream>

#include <vector>

#include "ThreeSteppers/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;

// Локальная матрица, будь то M, G, и прочие матрицы одного размера

Matrix local\_mat;

// Локальный вектор, в пару локальным матрицам

vector<double> local\_b;

// Массив прямоугольников

vector<Rectangle> rectangles;

// Массив узлов

vector<Node> nodes;

// Массив сопоставления узлов и первых краевых

vector<S1\_node> s1\_nodes;

// Массив сопоставления рёбер и вторых краевых

vector<S23\_edge> s2\_edges;

// Массив сопоставления рёбер и третьих краевых

vector<S23\_edge> s3\_edges;

#pragma endregion GLOBAL\_OBJECTS

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();

// Считывание данных для вторых краевых условий s2\_edges

auto s2\_edgesFile = ifstream(GlobalPaths::s2\_edgesPath);

if (!s2\_edgesFile.is\_open())

throw runtime\_error("Не удалось открыть файл " + GlobalPaths::s2\_edgesPath);

s2\_edgesFile >> size;

s2\_edges.resize(size);

for (auto& s2 : s2\_edges)

{

s2\_edgesFile >> s2.node1 >> s2.node2 >> s2.funcNum;

}

s2\_edgesFile.close();

// Считывание данных для третьих краевых условий s3\_edges

auto s3\_edgesFile = ifstream(GlobalPaths::s3\_edgesPath);

if (!s3\_edgesFile.is\_open())

throw runtime\_error("Не удалось открыть файл " + GlobalPaths::s3\_edgesPath);

s3\_edgesFile >> size;

s3\_edges.resize(size);

for (auto& s3 : s3\_edges)

{

s3\_edgesFile >> s3.node1 >> s3.node2 >> s3.funcNum;

}

s3\_edgesFile.close();

}

void generatePortrait() {

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

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

for (auto& rect : rectangles)

{

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 треугольнике, то скипаем

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

{

continue;

}

bool isExist = false;

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

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

{

if (global\_mat.jg[it] == elems[k]) isExist = true;

}

if (!isExist)

{

// Ищем, куда вставить элемент портрета

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

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

}

void addLocalG(const Rectangle& rect) {

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);

double lambda = (lambda\_value(rect.regionNum, nodes[rect.a]) +

lambda\_value(rect.regionNum, nodes[rect.b]) +

lambda\_value(rect.regionNum, nodes[rect.c]) +

lambda\_value(rect.regionNum, nodes[rect.d])) / 4;

double integrals[9];

integrals[0] = rp / hr + 0.5;

integrals[1] = -integrals[0];

integrals[2] = (((rp \* rp) / (hr \* hr)) + (2 \* rp / hr) + 1) \* std::log(abs((rp + hr) / rp)) - (rp / hr) - 1.5;

integrals[3] = (rp\*rp)/(hr\*hr) \* std::log(abs((rp+hr) / rp)) - rp/hr + 0.5;

integrals[4] = - (rp/hr + rp\*rp/(hr\*hr)) \* std::log(abs((rp+hr)/rp)) + rp/hr + 0.5;

integrals[5] = h\_phi / 3;

integrals[6] = h\_phi / 6;

integrals[7] = 1 / h\_phi;

integrals[8] = -integrals[7];

// Численное интегрирование по Гауссу 3 порядка

//double rValue[2] = {

// (2\*rp + hr) / 2 - (hr) / (2\*sqrt(3)),

// (2 \* rp + hr) / 2 + (hr) / (2 \* sqrt(3))

//};

//double phiValue[2] = {

// (2 \* phi\_s + h\_phi) / 2 - (h\_phi) / (2 \* sqrt(3)),

// (2 \* phi\_s + h\_phi) / 2 + (h\_phi) / (2 \* sqrt(3))

//};

//integrals[0] = 1.0/2 \* (rValue[0] + rValue[1]) / (hr);

//integrals[1] = -integrals[0];

//integrals[2] = 1.0/2 \* (

// (rp + hr - rValue[0]) \* (rp + hr - rValue[0]) / rValue[0]

// + (rp + hr - rValue[1]) \* (rp + hr - rValue[1]) / rValue[1]) / hr;

//integrals[3] = 1.0 / 2 \* (

// (rValue[0] - rp) \* (rValue[0] - rp) / rValue[0]

// + (rValue[1] - rp) \* (rValue[1] - rp) / rValue[1]) / hr;

//integrals[4] = 1.0 / 2 \* (

// (rp + hr - rValue[0]) \* (rValue[0] - rp) / rValue[0]

// + (rp + hr - rValue[1]) \* (rValue[1] - rp) / rValue[1]) / hr;

//integrals[5] = 1.0 / 2 \* (

// (phi\_s + h\_phi - phiValue[0]) \* (phi\_s + h\_phi - phiValue[0])

// + (phi\_s + h\_phi - phiValue[1]) \* (phi\_s + h\_phi - phiValue[1])) / h\_phi;

//integrals[6] = 1.0 / 2 \* (

// (phi\_s + h\_phi - phiValue[0]) \* (phiValue[0] - phi\_s)

// + (phi\_s + h\_phi - phiValue[1]) \* (phiValue[1] - phi\_s)) / h\_phi;

//integrals[7] = 1.0 / h\_phi;

//integrals[8] = -integrals[7];

local\_mat[0][0] = lambda \* (integrals[0] \* integrals[5] + integrals[2] \* integrals[7]);

local\_mat[0][1] = lambda \* (integrals[1] \* integrals[5] + integrals[4] \* integrals[7]);

local\_mat[0][2] = lambda \* (integrals[0] \* integrals[6] + integrals[2] \* integrals[8]);

local\_mat[0][3] = lambda \* (integrals[1] \* integrals[6] + integrals[4] \* integrals[8]);

local\_mat[1][0] = local\_mat[0][1];

local\_mat[1][1] = lambda \* (integrals[0] \* integrals[5] + integrals[3] \* integrals[7]);

local\_mat[1][2] = local\_mat[0][3];

local\_mat[1][3] = lambda \* (integrals[0] \* integrals[6] + integrals[3] \* integrals[8]);

local\_mat[2][0] = local\_mat[0][2];

local\_mat[2][1] = local\_mat[1][2];

local\_mat[2][2] = lambda \* (integrals[0] \* integrals[5] + integrals[2] \* integrals[7]);

local\_mat[2][3] = lambda \* (integrals[1] \* integrals[5] + integrals[4] \* integrals[7]);

local\_mat[3][0] = local\_mat[0][3];

local\_mat[3][1] = local\_mat[1][3];

local\_mat[3][2] = local\_mat[2][3];

local\_mat[3][3] = lambda \* (integrals[0] \* integrals[5] + integrals[3] \* integrals[7]);

cout << "Local\_G:" << endl;

for (int i = 0; i < 4; i++)

{

double tmp = 0;

for (int j = 0; j < 4; j++)

{

tmp += local\_mat[i][j];

}

cout << " " << tmp;

}

cout << endl;

}

void addLocalM(const Rectangle& rect) {

double rp = nodes[rect.a].r;

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

double phi\_s = nodes[rect.a].phi;

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

double gamma[4] = {

gamma\_value(rect.regionNum, nodes[rect.a]),

gamma\_value(rect.regionNum, nodes[rect.b]),

gamma\_value(rect.regionNum, nodes[rect.c]),

gamma\_value(rect.regionNum, nodes[rect.d])

};

double tmp[4][4];

tmp[0][0] = hr \* h\_phi \* (gamma[0] \* (rp / 4 + hr / 20) / 4

+ gamma[1] \* (rp / 12 + hr / 30) / 4

+ gamma[2] \* (rp / 4 + hr / 20) / 12

+ gamma[3] \* (rp / 12 + hr / 30) / 12

);

tmp[0][1] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 30) / 4

+ gamma[1] \* (rp / 12 + hr / 20) / 4

+ gamma[2] \* (rp / 12 + hr / 30) / 12

+ gamma[3] \* (rp / 12 + hr / 20) / 12

);

tmp[0][2] = hr \* h\_phi \* (gamma[0] \* (rp / 4 + hr / 20) / 12

+ gamma[1] \* (rp / 12 + hr / 30) / 12

+ gamma[2] \* (rp / 4 + hr / 20) / 12

+ gamma[3] \* (rp / 12 + hr / 30) / 12

);

tmp[0][3] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 30) / 12

+ gamma[1] \* (rp / 12 + hr / 20) / 12

+ gamma[2] \* (rp / 12 + hr / 30) / 12

+ gamma[3] \* (rp / 12 + hr / 20) / 12

);

tmp[1][0] = tmp[0][1];

tmp[1][1] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 20) / 4

+ gamma[1] \* (rp / 4 + hr / 5) / 4

+ gamma[2] \* (rp / 12 + hr / 20) / 12

+ gamma[3] \* (rp / 4 + hr / 5) / 12

);

tmp[1][2] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 30) / 12

+ gamma[1] \* (rp / 12 + hr / 20) / 12

+ gamma[2] \* (rp / 12 + hr / 30) / 12

+ gamma[3] \* (rp / 12 + hr / 20) / 12

);

tmp[1][3] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 20) / 12

+ gamma[1] \* (rp / 4 + hr / 5) / 12

+ gamma[2] \* (rp / 12 + hr / 20) / 12

+ gamma[3] \* (rp / 4 + hr / 5) / 12

);

tmp[2][0] = tmp[0][2];

tmp[2][1] = tmp[1][2];

tmp[2][2] = hr \* h\_phi \* (gamma[0] \* (rp / 4 + hr / 20) / 12

+ gamma[1] \* (rp / 12 + hr / 30) / 12

+ gamma[2] \* (rp / 4 + hr / 20) / 4

+ gamma[3] \* (rp / 12 + hr / 30) / 4

);

tmp[2][3] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 30) / 12

+ gamma[1] \* (rp / 12 + hr / 20) / 12

+ gamma[2] \* (rp / 12 + hr / 30) / 4

+ gamma[3] \* (rp / 12 + hr / 20) / 4

);

tmp[3][0] = tmp[0][3];

tmp[3][1] = tmp[1][3];

tmp[3][2] = tmp[2][3];

tmp[3][3] = hr \* h\_phi \* (gamma[0] \* (rp / 12 + hr / 20) / 12

+ gamma[1] \* (rp / 4 + hr / 5) / 12

+ gamma[2] \* (rp / 12 + hr / 20) / 4

+ gamma[3] \* (rp / 4 + hr / 5) / 4

);

for (int i = 0; i < 4; i++)

{

for (int k = 0; k < 4; k++)

{

local\_mat[i][k] += tmp[i][k];

}

}

}

void addLocalB(const Rectangle& rect) {

double rp = nodes[rect.a].r;

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

double phi\_s = nodes[rect.a].phi;

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

double f[4] = {

f\_value(rect.regionNum, nodes[rect.a]),

f\_value(rect.regionNum, nodes[rect.b]),

f\_value(rect.regionNum, nodes[rect.c]),

f\_value(rect.regionNum, nodes[rect.d])

};

double tmp[4][4];

// Матрица массы с gamma = const = 1

tmp[0][0] = hr \* h\_phi \* ((rp / 4 + hr / 20) / 4

+ (rp / 12 + hr / 30) / 4

+ (rp / 4 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 12

);

tmp[0][1] = hr \* h\_phi \* ((rp / 12 + hr / 30) / 4

+ (rp / 12 + hr / 20) / 4

+ (rp / 12 + hr / 30) / 12

+ (rp / 12 + hr / 20) / 12

);

tmp[0][2] = hr \* h\_phi \* ((rp / 4 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 12

+ (rp / 4 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 12

);

tmp[0][3] = hr \* h\_phi \* ((rp / 12 + hr / 30) / 12

+ (rp / 12 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 12

+ (rp / 12 + hr / 20) / 12

);

tmp[1][0] = tmp[0][1];

tmp[1][1] = hr \* h\_phi \* ((rp / 12 + hr / 20) / 4

+ (rp / 4 + hr / 5) / 4

+ (rp / 12 + hr / 20) / 12

+ (rp / 4 + hr / 5) / 12

);

tmp[1][2] = hr \* h\_phi \* ((rp / 12 + hr / 30) / 12

+ (rp / 12 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 12

+ (rp / 12 + hr / 20) / 12

);

tmp[1][3] = hr \* h\_phi \* ((rp / 12 + hr / 20) / 12

+ (rp / 4 + hr / 5) / 12

+ (rp / 12 + hr / 20) / 12

+ (rp / 4 + hr / 5) / 12

);

tmp[2][0] = tmp[0][2];

tmp[2][1] = tmp[1][2];

tmp[2][2] = hr \* h\_phi \* ((rp / 4 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 12

+ (rp / 4 + hr / 20) / 4

+ (rp / 12 + hr / 30) / 4

);

tmp[2][3] = hr \* h\_phi \* ((rp / 12 + hr / 30) / 12

+ (rp / 12 + hr / 20) / 12

+ (rp / 12 + hr / 30) / 4

+ (rp / 12 + hr / 20) / 4

);

tmp[3][0] = tmp[0][3];

tmp[3][1] = tmp[1][3];

tmp[3][2] = tmp[2][3];

tmp[3][3] = hr \* h\_phi \* ((rp / 12 + hr / 20) / 12

+ (rp / 4 + hr / 5) / 12

+ (rp / 12 + hr / 20) / 4

+ (rp / 4 + hr / 5) / 4

);

for (int i = 0; i < 4; i++)

{

double t = 0;

for (int k = 0; k < 4; k++)

{

t += tmp[i][k] \* f[k];

}

local\_b[i] = t;

}

}

void addLocalToGlobal(const Rectangle& rect) {

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++)

{

int id;

for (id = global\_mat.ig[elems[i]]; id < global\_mat.ig[elems[i] + 1] && global\_mat.jg[id] != elems[k]; id++);

global\_mat.ggl[id] += local\_mat[i][k];

global\_mat.ggu[id] += local\_mat[i][k];

}

// добавляем диагональные элементы и вектор b

global\_mat.di[elems[i]] += local\_mat[i][i];

global\_b[elems[i]] += local\_b[i];

}

}

void addLocalsToGlobal(const Rectangle& rect) {

addLocalG(rect);

addLocalM(rect);

addLocalB(rect);

addLocalToGlobal(rect);

}

void include\_s3() {

for (const auto& edge : s3\_edges)

{

double M[2][2];

double b[2];

double beta = (s3\_beta\_value(edge.funcNum, nodes[edge.node1])

+ s3\_beta\_value(edge.funcNum, nodes[edge.node2])) / 2;

double u[2] = {

s3\_u\_value(edge.funcNum, nodes[edge.node1]),

s3\_u\_value(edge.funcNum, nodes[edge.node2])

};

double rp = nodes[edge.node1].r;

double hr = nodes[edge.node2].r - nodes[edge.node1].r;

double phi\_s = nodes[edge.node1].phi;

double h\_phi = nodes[edge.node2].phi - nodes[edge.node1].phi;

// Если краевое задано вдоль оси r

if (hr > 1e-7)

{

M[0][0] = beta \* ((hr \* hr) / 12 + (rp \* hr) / 3);

M[0][1] = beta \* ((hr \* rp) / 6 + (hr \* hr) / 12);

M[1][0] = M[0][1];

M[1][1] = beta \* (rp\*hr/3 + hr\*hr/4);

b[0] = beta \* (u[0] \* (hr \* rp / 3 + hr \* hr / 12) + u[1] \* (hr \* rp / 6 + hr \* hr / 12));

b[1] = beta \* (u[0] \* (hr \* rp / 6 + hr \* hr / 12) + u[1] \* (hr \* rp / 3 + hr \* hr / 4));

//b[0] = beta \* u \* (hr \* hr / 6 + rp \* hr / 2);

//b[1] = beta \* u \* (hr \* hr / 3 + rp \* hr / 2);

}

// Если краевое задано вдоль оси phi

else

{

M[0][0] = beta \* rp \* h\_phi / 3;

M[0][1] = beta \* rp \* h\_phi / 6;

M[1][0] = M[0][1];

M[1][1] = beta \* rp \* h\_phi / 3;

b[0] = beta \* rp \* (u[0] \* h\_phi / 3 + u[1] \* h\_phi / 6);

b[1] = beta \* rp \* (u[0] \* h\_phi / 6 + u[1] \* h\_phi / 3);

//b[0] = beta \* u \* rp \* h\_phi / 2;

//b[1] = b[0];

}

// добавляем полученный результат в глобальную матрицу

int elems[2] = { edge.node1, edge.node2 };

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

{

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

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

{

int id;

for (id = global\_mat.ig[elems[i]]; id < global\_mat.ig[elems[i] + 1] && global\_mat.jg[id] != elems[k]; id++);

global\_mat.ggl[id] += M[i][k];

global\_mat.ggu[id] += M[i][k];

}

// добавляем диагональные элементы и вектор b

global\_mat.di[elems[i]] += M[i][i];

global\_b[elems[i]] += b[i];

}

}

}

void include\_s2() {

for (const auto& edge : s2\_edges)

{

double b[2];

//double theta = (s2\_theta\_value(edge.funcNum, nodes[edge.node1])

// + s2\_theta\_value(edge.funcNum, nodes[edge.node2])) / 2;

double theta[2] = {

s2\_theta\_value(edge.funcNum, nodes[edge.node1]),

s2\_theta\_value(edge.funcNum, nodes[edge.node2])

};

double rp = nodes[edge.node1].r;

double hr = nodes[edge.node2].r - nodes[edge.node1].r;

double phi\_s = nodes[edge.node1].phi;

double h\_phi = nodes[edge.node2].phi - nodes[edge.node1].phi;

// Если краевое задано вдоль оси r

if (hr > 1e-7)

{

b[0] = (theta[0] \* (hr \* rp / 3 + hr \* hr / 12) + theta[1] \* (hr \* rp / 6 + hr \* hr / 12));

b[1] = (theta[0] \* (hr \* rp / 6 + hr \* hr / 12) + theta[1] \* (hr \* rp / 3 + hr \* hr / 4));

//b[0] = theta \* (hr \* hr / 6 + rp \* hr / 2);

//b[1] = theta \* (hr \* hr / 3 + rp \* hr / 2);

}

// Если краевое задано вдоль оси phi

else

{

b[0] = rp \* (theta[0] \* h\_phi / 3 + theta[1] \* h\_phi / 6);

b[1] = rp \* (theta[0] \* h\_phi / 6 + theta[1] \* h\_phi / 3);

//b[0] = theta \* rp \* h\_phi / 2;

//b[1] = b[0];

}

// добавляем полученный результат в глобальную матрицу

int elems[2] = { edge.node1, edge.node2 };

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

{

// добавляем вектор b

global\_b[elems[i]] += b[i];

}

}

}

void include\_s1() {

for (const auto& node : s1\_nodes)

{

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

// ставим на диагональ значение 1

global\_mat.di[node.node] = 1;

// ставим в соответствующую ячейку вектора b значение u

global\_b[node.node] = u;

// зануляем строку в нижнем треугольнике

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

{

global\_mat.ggl[j] = 0;

}

// зануляем строку в верхнем треугольнике

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

{

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

{

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

{

global\_mat.ggu[j] = 0;

break;

}

}

}

}

}

void main() {

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

readDataFromFiles();

generatePortrait();

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

local\_mat.resize(4);

for (auto& vec : local\_mat)

vec.resize(4);

local\_b.resize(4);

for (const auto& rect : rectangles)

{

addLocalsToGlobal(rect);

}

include\_s3();

include\_s2();

include\_s1();

vector<double> q;

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

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

double eps;

IterSolvers::LOS::LuPrecond(global\_mat, global\_b, q, eps);

cout << "Полученное решение: " << endl;

cout.precision(14); cout.setf(ios\_base::fixed);

for (auto elem : q)

{

cout << elem << endl;

}

return;

}

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 <string>

#include <stdexcept>

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";

const std::string s2\_edgesPath = filesPath + "s2\_edges.txt";

const std::string s3\_edgesPath = filesPath + "s3\_edges.txt";

}

#pragma region TYPEDEFINES

using Matrix = std::vector<std::vector<double>>;

struct Rectangle {

int a = 0;

int b = 0;

int c = 0;

int d = 0;

int regionNum = 0;

};

struct Node {

double r = 0.0;

double phi = 0.0;

};

struct S1\_node {

int node = 0;

int funcNum = 0;

};

struct S23\_edge {

int node1 = 0;

int node2 = 0;

int funcNum = 0;

};

#pragma endregion TYPEDEFINES

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

double ans = 0.0;

switch (regionNum)

{

case 0: ans = r \* phi; break;

//case 1: ans = 1.0; break;

default:

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

break;

}

return ans;

}

double f\_value(int regionNum, Node node) {

return f\_value(regionNum, node.r, node.phi);

}

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

double ans = 0.0;

switch (regionNum)

{

case 0: ans = 0.0; break;

//case 1: ans = 1.0; break;

default:

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

break;

}

return ans;

}

double lambda\_value(int regionNum, Node node) {

return lambda\_value(regionNum, node.r, node.phi);

}

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

double ans = 0.0;

switch (regionNum)

{

case 0: ans = r \* phi; break;

//case 1: ans = 1.0; break;

default:

throw std::runtime\_error("Значения функции gamma для региона с номером " + std::to\_string(regionNum) + " не найдено.");

break;

}

return ans;

}

double gamma\_value(int regionNum, Node node) {

return gamma\_value(regionNum, node.r, node.phi);

}

double s3\_beta\_value(int s3\_funcNum, Node node) {

double ans = 0.0;

switch (s3\_funcNum)

{

//case 0: ans = 1.0; break;

//case 1: ans = 1.0; break;

default:

throw std::runtime\_error("Значения функции beta для s3-краевого с номером " + std::to\_string(s3\_funcNum) + " не найдено.");

break;

}

return ans;

}

double s3\_u\_value(int s3\_funcNum, Node node) {

double ans = 0.0;

switch (s3\_funcNum)

{

//case 0: ans = 1.0; break;

//case 1: ans = 1.0; break;

default:

throw std::runtime\_error("Значения функции U\_beta для s3-краевого с номером " + std::to\_string(s3\_funcNum) + " не найдено.");

break;

}

return ans;

}

double s2\_theta\_value(int s2\_funcNum, Node node) {

double ans = 0.0;

switch (s2\_funcNum)

{

//case 0: ans = 0.0; break;

//case 1: ans = 0.0; break;

default:

throw std::runtime\_error("Значения функции theta для s2-краевого с номером " + std::to\_string(s2\_funcNum) + " не найдено.");

break;

}

return ans;

}

double s1\_u\_value(int s1\_funcNum, Node node) {

double ans = 0.0;

switch (s1\_funcNum)

{

case 0: ans = 1.0; break;

//case 1: ans = 1.0; break;

default:

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

break;

}

return ans;

}

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

uint16\_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 + 1]; 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);

}

/// <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 + 1]; 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;

}

// Конструкторы матрицы

SparseMatrix::SparseMatrix() {}

SparseMatrix::SparseMatrix(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 + 1);

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;

}

Файл SparseMatrix.h:

#pragma once

#include <vector>

#include <cmath>

#include <stdexcept>

#include <fstream>

#include <string>

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

/// <summary>

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

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

/// </summary>

class SparseMatrix {

// Переменные матрицы

public:

/// <summary>

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

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

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

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

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

/// </summary>

std::vector<uint32\_t> ig;

/// <summary>

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

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

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> jg: { 0, 1 } </para>

/// </summary>

std::vector<uint16\_t> jg;

/// <summary>

/// Массив элементов нижнего треугольника матрицы

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

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> ggl: { 3, 2 } </para>

/// </summary>

std::vector<double> ggl;

/// <summary>

/// Массив элементов верхнего треугольника матрицы

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

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> ggu: { 2, 1 } </para>

/// </summary>

std::vector<double> ggu;

/// <summary>

/// Массив элементов диагонали матрицы

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

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> di: { 1, 8, 4 } </para>

/// </summary>

std::vector<double> di;

// Методы матрицы

public:

uint16\_t Size() 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;

/// <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;

// Конструкторы матрицы

public:

SparseMatrix();

SparseMatrix(SparseMatrix& right);

// Конструктор перемещения (нужен для метода ReadFromFiles)

SparseMatrix(SparseMatrix&& right) noexcept;

// Статические методы матрицы

public:

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.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] + 1])

{

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 + 1]; 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 + 1]; 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 + 1]; 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 + 1]; 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 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.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;

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 LOS {

size\_t resetIter = 10;

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) {

uint16\_t size = x.size();

LU& lu = \*\_lu\_mat;

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

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

A.MultToVec(x, tmp);

for (uint16\_t 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() {

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

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

delete \_lu\_mat;

\_lu\_mat = nullptr;

}

};

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;

extern LU\* \_lu\_mat;

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

namespace LOS {

extern size\_t resetIter;

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();

};