|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра прикладной математики | | | |
|  | | | |
|  | | | |
| Курсовой проект по курсу | | | |
| **«Численные методы»** | | | |
|  | | | |
|  | Группа | ПМ-12 |
|  |  |
| Студент | Овчинников и. р. |
|  |  |
|
|  |
| Новосибирск | | | |

2023

1. **Формулировка задания**

МКЭ для двумерной краевой задачи для эллиптического уравнения в декартовой системе координат. Базисные функции линейные на треугольниках. Краевые условия всех типов. Коэффициент диффузии λ разложить по линейным базисным функциям. Матрица СЛАУ генерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

1. **Постановка задачи**

Уравнение эллиптической краевой задачи функции u в области Ω с границей S = S1 ∪ S2 ∪ S3:



Краевые условия:

 - краевое условие 1-го рода

 - краевое условие 2-го рода

 - краевое условие 3-го рода

Уравнение в декартовой системе координат:



1. **Теоретическая часть**
   1. Вариационная постановка

Уравнение можно записать в виде:

Потребуем, чтобы невязка дифференциального уравнения была ортогональна некоторому пространству функций , которое мы будем называть пространством пробных функций, т. е.

Преобразуем слагаемое с использованием формул Грина:

Где границы, на которых заданы соответствующие краевые условия.

Интегралы по границам можно преобразовать, воспользовавшись краевыми условиями:

Поскольку на границе краевыми условиями не определяется значение , слагаемое следует исключить из уравнения, потребовав, чтобы пространство пробный функций содержало только функции, которые принимали бы нулевые значения на границе.

В качестве мы можем выбрать .

Таким образом, получаем вариационное уравнение вида:

Получим аппроксимацию уравнения Галёркина на конечномерных подпространствах , аппроксимирующих исходные пространства

Поскольку любая функция ∈ может быть представлена в виде линейной комбинации

вариационное уравнение эквивалентно следующей системе уравнений:

Поскольку , оно может быть представлено в виде линейной комбинации базисных функций пространства :

подставляя в прошлое уравнение, получим СЛАУ для компонент вектора весов q с индексами :

Поскольку исходная задача рассматривается в декартовой системе координат, то

и, соответственно:

Отсюда получаем:

* 1. Конечноэлементная дискретизация и переход к локальным матрицами

Рассмотрим треугольник с вершинами , и

Введем на нём три линейные функции

,

такие, что функция равна единице в вершине и нулю в двух остальных вершинах(аналогично для функций ).

Отсюда справедливо следующее:

=

И отсюда следует, что все коэффициенты функций являются компонентами матрицы **α**, обратной к определяемой только координатами вершин треугольника матрице

- матрица координат вершин треугольника

Через матрицу может быть вычислена площадь треугольника с вершинами , и

где – определитель матрицы . Значения и компонент , являющихся коэффициентами функций легко вычисляются из определения матрицы :

Функции , удобны тем, что для них существуют очень простые формулы, позволяющие вычислять интегралы от произведенийв произвольных степенях по треугольнику

На каждом элементе треугольной сетки определим три локальные базисные функции

Чтобы получить выражения для локальных матриц жёсткости и массы каждого конечного элемента , перейдём к решению локальной задачи на каждом конечном элементе. Полученное уравнение для области представим в виде суммы интегралов по областям без учёта краевых условий. Тогда на каждом конечном элементе будем решать локальную задачу построения матриц жёсткости и массы и вектора правой части.

Локальная матрица будет представлять собой сумму матриц жёсткости и массы и будет иметь размерность 3x3 (по числу узлов на конечном элементе).

Получим выражение для локальной матрицы жесткости ***G***.

В поставленной задаче требуется разложить по линейным базисным функциям:

Получим выражение для локальной матрицы масс ***M***.

Где осредненное значение на элементе.

Получим выражение для вектора правой части

представим в виде , где значения в вершинах треугольника. Получим:

Таким образом:

* 1. Учет краевых условий
* Учет первых краевых условий

Учет первых краевых условий проводится после полной сборки глобальной матрицы и правой части (включая учет условий второго и третьего рода). В глобальной матрице и глобальном векторе находится соответствующая глобальному номеру краевого узла строка, и ставятся единица на место диагонального элемента глобальной и нули на место внедиагольных элементов. Вместо элемента с соответствующим номером в вектор правой части присваивается значение первого краевого условия.

Структуры данных для хранения информации о краевых условиях организованы вектором целых чисел, где сначала указано количество рёбер, затем первое число - номер узла начала ребра, второе – номер узла конца ребра, а третье – номер формулы, которая будет вычислять значение параметра ug.

* Учет вторых краевых условий:

Отсюда получаем, что для учета вторых краевых условий необходимо вычислить интеграл:

Введем на ребре локальную нумерацию его вершин и обозначим через базисную функцию, которая равна единице в первой вершине ребра, а через – во второй. Поскольку и на ребре – одномерные линейные базисные функции, они имеют вид:

где – длина ребра ( и – координаты вершин ребра ) и

Параметр заменяется на своим интерпеллянтом , где и -значения параметра в вершинах ребра .

Получаем:

* Учет третьих краевых условий:

Аналогично учету вторых краевых условий получаем вектор :

Также получаем матрицу :

1. **Описания разработанных программ**
   1. Структуры данных

|  |  |
| --- | --- |
| **Структура** | **Описание** |
| Vertex | Хранит координаты x и y узла |
| Triangle | Хранит номера узлов конечного элемента и область, к которой он относится |
| FirstBoundaryCondition | Хранит номер узла и номер выражения для вычисления значения |
| SecondBoundaryCondition | Хранит номера узлов ребра и номер выражения для вычисления значения θ |
| ThirdBoundaryCondition | Хранит номера узлов ребра и номера выражений для вычисления значений β и uβ |

* 1. Входные файлы

|  |  |
| --- | --- |
| **Название** | **Описание** |
| Vertices.txt | Первое значение – количество узлов, далее пары координат x и y |
| Triangles.txt | Первое значение – количество конечных элементов, далее для каждого элемента тройка номеров узлов и номер области |
| BoundaryConditions.txt | Первое значение – количество краевых условий, далее для каждого условия тип, ребро/узел и номера выражений для вычисления значений |

* 1. Структура модулей программы

vector<Vertex> vertices; // Узлы системы

vector<Triangle> tris; // Конечные элементы

vector<FirstBoundaryCondition> firstBoundary; // Первые краевые условия

vector<SecondBoundaryCondition> secondBoundary; // Вторые краевые условия

vector<ThirdBoundaryCondition> thirdBoundary; // Третьи краевые условия

double\* q; // Вектор решения

void Input(); // Ввод данных из файла

void Solve(); // Общая функция запуска решения

int regionsNum, globalN; // Количество областей и узлов

int\* ig, \* jg; // Глобальная матрица

double\* ggl, \* ggu, \* di, \* b;

double G[3][3]{}; // Пустая матрица G

double M[3][3]{}; // Пустая матрица M

const double pureM[3][3] = { {2, 1, 1}, {1, 2, 1}, {1, 1, 2} }; // Шаблон матрицы M для возвращения ее в исходное состояние на каждой итерации

double localB[3]{}; // Локальный вектор b

double Lamda(int vert, int region); // Вычисление значения лямбда

double Gamma(int vert, int region); // Вычисление значения гамма

double Function(int vert, int region); // Вычисление значения функции f

double Beta(int vert, int eqNum); // Вычисление значения бета

double Ubeta(int vert, int eqNum); // Вычисление значения U бета для 3 краевого условия

double Theta(int vert, int eqNum); // Вычисление значения тета для 2 краевого условия

double Ug(int vert, int eqNum); // Вычисление значения в узле для 1 краевого условия

double GetAverageLamda(Triangle tri); // Вычисление среднего лямбда на элементе

double GetAverageGamma(Triangle tri); // Вычисление среднего гамма на элементе

double DetD(Triangle tri); // Вычисление определителя D (удвоенной площади) элемента

double Alpha(Triangle tri, int k, int i); // Вычисление значения альфа для построения матрицы G

double EdgeLength(int vert1, int vert2); // Вычисление длины ребра

int IndexOfUnknown(Triangle tri, int i); // Получение глобального номера узла из локального у элемента

void FormM(Triangle tri); // Формирование матрицы G

void FormG(Triangle tri); // Формирование матрицы M

void FormPortrait(); // Формирование портрета глобальной матрицы

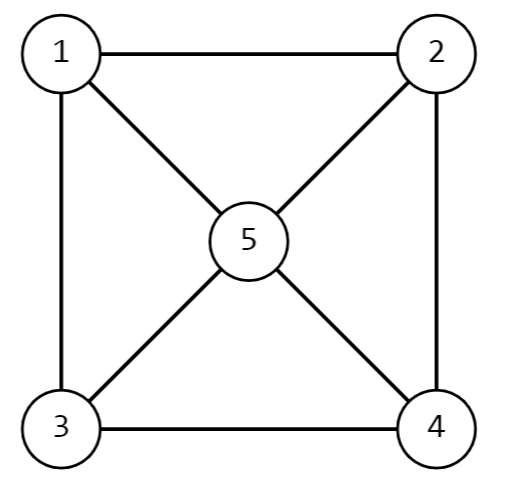
void ResolveBoundaries(); // Учет всех краевых условий

void AddToGlobal(int i, int j, double add); // Добавление значения в глобальную матрицу

void AllocateGlobalMatrix(); // Выделение памяти для глобальной матрицы

void FormB(Triangle tri); // Формирование локального вектора b

1. **Тесты**

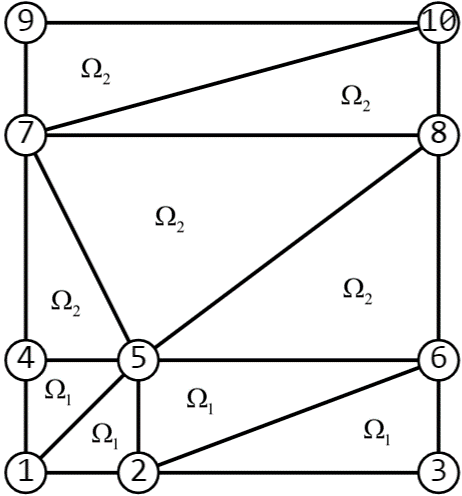
**Тест №1**

|  |  |  |
| --- | --- | --- |
| Параметры | Координаты вершин: | Краевые условия: |

Решение:

|  |  |  |  |
| --- | --- | --- | --- |
| Узел | q\* | q |  |
| 1 | 6 | 6.00000000000000e+00 | 0.000E+00 |
| 2 | 8 | 8.00000000000000e+00 | 0.000E+00 |
| 3 | 4 | 4.00000000000000e+00 | 0.000E+00 |
| 4 | 6 | 6.00000000000000e+00 | 0.000E+00 |
| 5 | 6 | 6.00000000000000e+00 | 0.000E+00 |

**Тест №2 (Все граничные условия)**

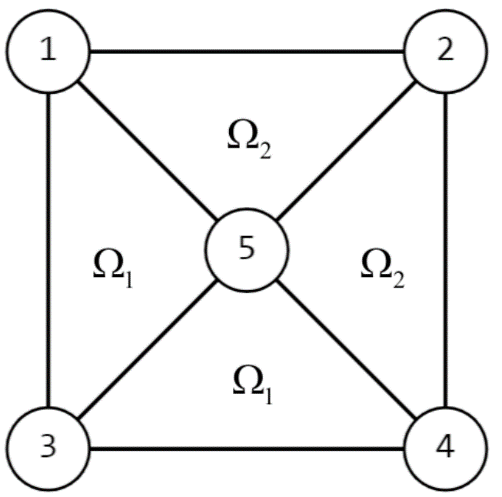


|  |  |  |  |
| --- | --- | --- | --- |
| Параметры | Координаты вершин: | Краевые условия: |  |

Решение:

|  |  |  |  |
| --- | --- | --- | --- |
| Узел | q\* | q |  |
| 1 | 29 | 2.90000000000000e+01 | 0.000E+00 |
| 2 | 29.5 | 2.95000000000000e+01 | 0.000E+00 |
| 3 | 32 | 3.20000000000000e+01 | 0.000E+00 |
| 4 | 35 | 3.50000000000000e+01 | 0.000E+00 |
| 5 | 35.5 | 3.55000000000000e+01 | 0.000E+00 |
| 6 | 38 | 3.80000000000000e+01 | 0.000E+00 |
| 7 | 50 | 5.00000000000000e+01 | 0.000E+00 |
| 8 | 53 | 5.30000000000000e+01 | 0.000E+00 |
| 9 | 59 | 5.90000000000000e+01 | 0.000E+00 |
| 10 | 62 | 6.20000000000000e+01 | 0.000E+00 |

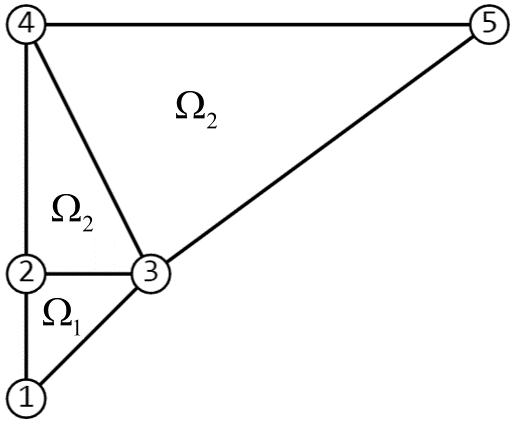
**Тест №3 (Порядок аппроксимации)**

Аналитическая функция – полином 1 порядка:

|  |  |  |
| --- | --- | --- |
| Параметры | Координаты вершин: | Краевые условия: |

Решение:

|  |  |  |  |
| --- | --- | --- | --- |
| Узел | q\* | q |  |
| 1 | 6 | 6.00000000000000e+00 | 0.000E+00 |
| 2 | 7 | 7.00000000000000e+00 | 0.000E+00 |
| 3 | 4 | 4.00000000000000e+00 | 0.000E+00 |
| 4 | 6 | 6.00000000000000e+00 | 0.000E+00 |
| 5 | 6 | 6.00000000000000e+00 | 0.000E+00 |

Аналитическая функция – полином 2 порядка

|  |  |  |
| --- | --- | --- |
| Параметры | Координаты вершин: | Краевые условия: |

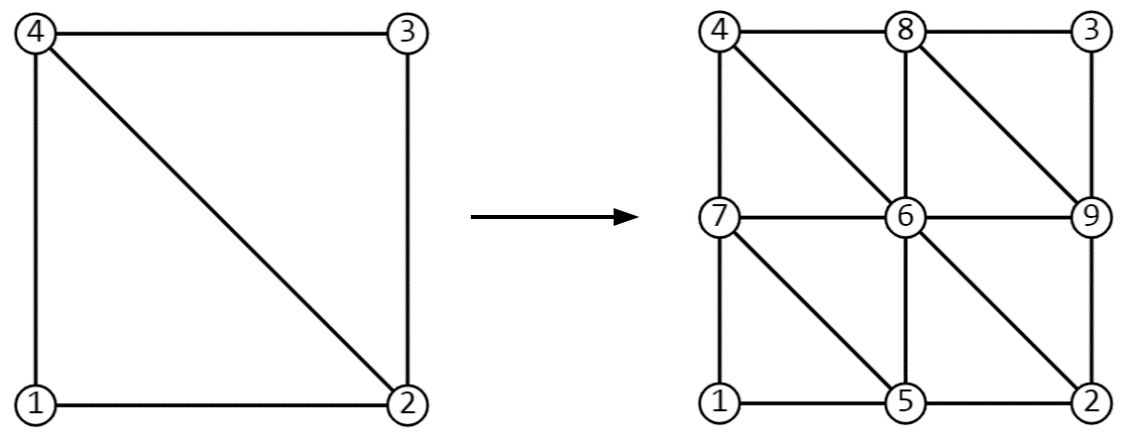
Решение:

|  |  |  |  |
| --- | --- | --- | --- |
| Узел | q\* | q |  |
| 1 | 0 | 1.31432708909994e-14 | 1.314E-14 |
| 2 | 1 | 1.14320069960650e+00 | 1.432E-01 |
| 3 | 1 | 1.00000000000004e+00 | 3.997E-14 |
| 4 | 61 | 6.10240489724530e+01 | 2.405E-02 |
| 5 | 61 | 6.10010931351115e+01 | 1.093E-03 |

Вывод: из результатов тестов видно, что порядок аппроксимации равен 1. На полиноме более высокого порядка присутствует погрешность, которая не является ошибкой округления

**Тест №4 (Порядок сходимости)**

Дробление происходит следующим образом. Левая область является начальной для дробления



|  |  |
| --- | --- |
| Параметры | Краевые условия:  На всех “внешних” узлах заданы 1 краевые условия |

Результат:

|  |  |
| --- | --- |
| Количество дроблений | Норма погрешности |
| 3 | 1.953E-04 |
| 4 | 4.924E-05 |

, откуда порядок сходимости стремится к 2, что соответствует теоретическому значению

1. **Текст программы**

**FEM.h**

#pragma once

#include "stdafx.h"

using namespace std;

struct Vertex

{

double x;

double y;

};

struct Triangle

{

int vert1, vert2, vert3;

int region;

};

struct FirstBoundaryCondition

{

int vert;

int equationNum;

};

struct SecondBoundaryCondition

{

int vert1, vert2;

int equationNum;

};

struct ThirdBoundaryCondition

{

int vert1, vert2;

int betaEquationNum, UbetaEquationNum;

};

class FEM

{

public:

vector<Vertex> vertices; // Узлы системы

vector<Triangle> tris; // Конечные элементы

vector<FirstBoundaryCondition> firstBoundary; // Первые краевые условия

vector<SecondBoundaryCondition> secondBoundary; // Вторые краевые условия

vector<ThirdBoundaryCondition> thirdBoundary; // Третьи краевые условия

double\* q; // Вектор решения

void Input(); // Ввод данных из файла

void Solve(); // Общая функция запуска решения

void PrintSolution();

private:

int regionsNum, globalN; // Количество областей и узлов

int\* ig, \* jg; // Глобальная матрица

double\* ggl, \* ggu, \* di, \* b;

double G[3][3]{}; // Пустая матрица G

double M[3][3]{}; // Пустая матрица M

const double pureM[3][3] = { {2, 1, 1}, {1, 2, 1}, {1, 1, 2} }; // Шаблон матрицы M для возвращения ее в исходное состояние на каждой итерации

double localB[3]{}; // Локальный вектор b

double Lamda(int vert, int region); // Вычисление значения лямбда

double Gamma(int vert, int region); // Вычисление значения гамма

double Function(int vert, int region); // Вычисление значения функции f

double Beta(int vert, int eqNum); // Вычисление значения бета

double Ubeta(int vert, int eqNum); // Вычисление значения U бета для 3 краевого условия

double Theta(int vert, int eqNum); // Вычисление значения тета для 2 краевого условия

double Ug(int vert, int eqNum); // Вычисление значения в узле для 1 краевого условия

double GetAverageLamda(Triangle tri); // Вычисление среднего лямбда на элементе

double GetAverageGamma(Triangle tri); // Вычисление среднего гамма на элементе

double DetD(Triangle tri); // Вычисление определителя D (удвоенной площади) элемента

double Alpha(Triangle tri, int k, int i); // Вычисление значения альфа для построения матрицы G

double EdgeLength(int vert1, int vert2); // Вычисление длины ребра

int IndexOfUnknown(Triangle tri, int i); // Получение глобального номера узла из локального у элемента

void FormM(Triangle tri); // Формирование матрицы G

void FormG(Triangle tri); // Формирование матрицы M

void FormPortrait(); // Формирование портрета глобальной матрицы

void ResolveBoundaries(); // Учет всех краевых условий

void AddToGlobal(int i, int j, double add); // Добавление значения в глобальную матрицу

void AllocateGlobalMatrix(); // Выделение памяти для глобальной матрицы

void FreeMemory();

void FormB(Triangle tri); // Формирование локального вектора b

};

**FEM.cpp**

#include "stdafx.h"

#include "FEM.h"

#include "slae.h"

double FEM::Lamda(int vert, int region)

{

// Зависит от конкретной задачи

}

double FEM::Gamma(int vert, int region)

{

// Зависит от конкретной задачи

}

double FEM::Function(int vert, int region)

{

// Зависит от конкретной задачи

}

double FEM::Beta(int vert, int eqNum)

{

// Зависит от конкретной задачи

}

double FEM::Ubeta(int vert, int eqNum)

{

// Зависит от конкретной задачи

}

double FEM::Theta(int vert, int eqNum)

{

// Зависит от конкретной задачи

}

double FEM::Ug(int vert, int eqNum)

{

// Зависит от конкретной задачи

}

void FEM::Input()

{

// Считывание узлов

FILE\* file;

fopen\_s(&file, "Vertices.txt", "r");

int num;

fscanf\_s(file, "%d", &num);

Vertex tempVert{};

globalN = num;

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

{

fscanf\_s(file, "%lf %lf", &tempVert.x, &tempVert.y);

vertices.push\_back(tempVert);

}

fclose(file);

// Считывание треугольников

fopen\_s(&file, "Triangles.txt", "r");

fscanf\_s(file, "%d", &num);

Triangle tempTri{};

regionsNum = num;

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

{

fscanf\_s(file, "%d %d %d %d", &tempTri.vert1, &tempTri.vert2, &tempTri.vert3, &tempTri.region);

tempTri.vert1--;

tempTri.vert2--;

tempTri.vert3--;

tris.push\_back(tempTri);

}

fclose(file);

// Считывание краевых

fopen\_s(&file, "BoundaryConditions.txt", "r");

fscanf\_s(file, "%d", &num);

FirstBoundaryCondition firstBoundTemp{};

SecondBoundaryCondition secondBoundTemp{};

ThirdBoundaryCondition thirdBoundTemp{};

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

{

int type;

fscanf\_s(file, "%d", &type);

switch (type)

{

case 1:

fscanf\_s(file, "%d %d", &firstBoundTemp.vert, &firstBoundTemp.equationNum);

firstBoundTemp.vert--;

firstBoundary.push\_back(firstBoundTemp);

break;

case 2:

fscanf\_s(file, "%d %d %d", &secondBoundTemp.vert1, &secondBoundTemp.vert2, &secondBoundTemp.equationNum);

secondBoundTemp.vert1--;

secondBoundTemp.vert2--;

secondBoundary.push\_back(secondBoundTemp);

break;

case 3:

fscanf\_s(file, "%d %d %d %d", &thirdBoundTemp.vert1, &thirdBoundTemp.vert2, &thirdBoundTemp.betaEquationNum, &thirdBoundTemp.UbetaEquationNum);

thirdBoundTemp.vert1--;

thirdBoundTemp.vert2--;

thirdBoundary.push\_back(thirdBoundTemp);

break;

default:

break;

}

}

fclose(file);

delete file;

}

void FEM::Solve()

{

FormPortrait();

AllocateGlobalMatrix();

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

{

Triangle currTri = tris[i];

FormG(currTri);

FormM(currTri);

FormB(currTri);

int globalBasis[3] = { currTri.vert1, currTri.vert2, currTri.vert3 };

for (int j = 0; j < 3; j++)

{

b[globalBasis[j]] += localB[j];

for (int k = 0; k < 3; k++)

AddToGlobal(globalBasis[j], globalBasis[k], G[j][k] + M[j][k]);

}

}

ResolveBoundaries();

SLAE slae;

slae.Input(globalN, 100000, 1e-30, ig, jg, ggl, ggu, di, b);

slae.OutputDense();

slae.MethodOfConjugateGradientsForNonSymMatrixWithDiagP();

q = slae.x;

PrintSolution();

FreeMemory();

}

void FEM::PrintSolution()

{

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

printf\_s("%.14lf\n", q[i]);

}

double FEM::GetAverageLamda(Triangle tri)

{

return (Lamda(tri.vert1, tri.region) + Lamda(tri.vert2, tri.region) + Lamda(tri.vert3, tri.region)) / 3.0;

}

double FEM::GetAverageGamma(Triangle tri)

{

return (Gamma(tri.vert1, tri.region) + Gamma(tri.vert2, tri.region) + Gamma(tri.vert3, tri.region)) / 3.0;

}

double FEM::DetD(Triangle tri)

{

return (vertices[tri.vert2].x - vertices[tri.vert1].x) \* (vertices[tri.vert3].y - vertices[tri.vert1].y) - (vertices[tri.vert3].x - vertices[tri.vert1].x) \* (vertices[tri.vert2].y - vertices[tri.vert1].y);

}

double FEM::Alpha(Triangle tri, int k, int i)

{

if (k == 1) {

switch (i)

{

case 0:

return vertices[tri.vert2].y - vertices[tri.vert3].y;

case 1:

return vertices[tri.vert3].y - vertices[tri.vert1].y;

case 2:

return vertices[tri.vert1].y - vertices[tri.vert2].y;

default:

return NAN;

}

}

else if (k == 2) {

switch (i)

{

case 0:

return vertices[tri.vert3].x - vertices[tri.vert2].x;

case 1:

return vertices[tri.vert1].x - vertices[tri.vert3].x;

case 2:

return vertices[tri.vert2].x - vertices[tri.vert1].x;

default:

return NAN;

}

}

else

return NAN;

}

double FEM::EdgeLength(int vert1, int vert2)

{

double x = vertices[vert1].x - vertices[vert2].x;

double y = vertices[vert1].y - vertices[vert2].y;

return sqrt(x\*x + y\*y);

}

int FEM::IndexOfUnknown(Triangle tri, int i)

{

switch (i)

{

case 0:

return tri.vert1;

case 1:

return tri.vert2;

case 2:

return tri.vert3;

default:

return NAN;

}

}

void FEM::FormM(Triangle tri)

{

std::copy(&pureM[0][0], &pureM[0][0] + 9, &M[0][0]);

double factor = (GetAverageGamma(tri) \* fabs(DetD(tri))) / 24.0;

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

M[i][j] \*= factor;

}

}

void FEM::FormG(Triangle tri)

{

double factor = 1.0 / (fabs(DetD(tri)) \* 6.0);

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++) {

G[i][j] = 0;

int helper[3] = { tri.vert1, tri.vert2, tri.vert3 };

for (int k = 0; k < 3; k++)

{

G[i][j] += Lamda(helper[k], tri.region) \* factor \* (Alpha(tri, 1, i) \* Alpha(tri, 1, j) + Alpha(tri, 2, i) \* Alpha(tri, 2, j));

}

}

}

}

void FEM::FormPortrait()

{

ig = new int[globalN + 1];

int\* list[2]{};

list[0] = new int[2 \* globalN \* (globalN - 2)];

list[1] = new int[2 \* globalN \* (globalN - 2)];

int\* listbeg = new int[globalN];

int listsize = -1;

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

listbeg[i] = -1;

for (int ielem = 0; ielem < regionsNum; ielem++)

{

for (int i = 0; i < 3; i++)

{

int k = IndexOfUnknown(tris[ielem], i);

for (int j = i + 1; j < 3; j++)

{

int ind1 = k;

int ind2 = IndexOfUnknown(tris[ielem], j);

if (ind2 < ind1) {

ind1 = ind2;

ind2 = k;

}

int iaddr = listbeg[ind2];

if (iaddr == -1) {

listsize++;

listbeg[ind2] = listsize;

list[0][listsize] = ind1;

list[1][listsize] = -1;

}

else {

while (list[0][iaddr] < ind1 && list[1][iaddr] > 0)

iaddr = list[1][iaddr];

if (list[0][iaddr] > ind1) {

listsize++;

list[0][listsize] = list[0][iaddr];

list[1][listsize] = list[1][iaddr];

list[0][iaddr] = ind1;

list[1][iaddr] = listsize;

}

else {

if (list[0][iaddr] < ind1) {

listsize++;

list[1][iaddr] = listsize;

list[0][listsize] = ind1;

list[1][listsize] = -1;

}

}

}

}

}

}

jg = new int[listsize + 1];

ig[0] = 0;

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

{

ig[i + 1] = ig[i];

int iaddr = listbeg[i];

while (iaddr != -1) {

jg[ig[i + 1]] = list[0][iaddr];

ig[i + 1]++;

iaddr = list[1][iaddr];

}

}

delete[] list[0];

delete[] list[1];

delete[] listbeg;

}

void FEM::ResolveBoundaries()

{

// Учет 3 краевых

const double localA[2][2] = { {2.0, 1.0}, {1.0, 2.0} };

for (int i = 0; i < thirdBoundary.size(); i++)

{

ThirdBoundaryCondition temp = thirdBoundary[i];

double factor = (((Beta(temp.vert1, temp.betaEquationNum) + Beta(temp.vert2, temp.betaEquationNum)) / 2.0) \* EdgeLength(temp.vert1, temp.vert2)) / 6.0;

b[temp.vert1] += factor \* (2 \* Ubeta(temp.vert1, temp.UbetaEquationNum) + Ubeta(temp.vert2, temp.UbetaEquationNum));

b[temp.vert2] += factor \* (Ubeta(temp.vert1, temp.UbetaEquationNum) + 2 \* Ubeta(temp.vert2, temp.UbetaEquationNum));

int globalBasis[2] = { temp.vert1, temp.vert2 };

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

{

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

AddToGlobal(globalBasis[i], globalBasis[j], localA[i][j] \* factor);

}

}

// Учет 2 краевых

for (int i = 0; i < secondBoundary.size(); i++)

{

SecondBoundaryCondition temp = secondBoundary[i];

double factor = EdgeLength(temp.vert1, temp.vert2) / 6.0;

double t1 = factor \* (2 \* Theta(temp.vert1, temp.equationNum) + Theta(temp.vert2, temp.equationNum));

b[temp.vert1] += factor \* (2 \* Theta(temp.vert1, temp.equationNum) + Theta(temp.vert2, temp.equationNum));

b[temp.vert2] += factor \* (Theta(temp.vert1, temp.equationNum) + 2 \* Theta(temp.vert2, temp.equationNum));

}

// Учет 1 краевых

for (int i = 0; i < firstBoundary.size(); i++)

{

FirstBoundaryCondition temp = firstBoundary[i];

b[temp.vert] = Ug(temp.vert, temp.equationNum);

di[temp.vert] = 1;

for (int k = ig[temp.vert]; k < ig[temp.vert + 1]; k++) {

ggl[k] = 0;

}

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

{

int ind;

bool exist = false;

for (ind = ig[k]; ind < ig[k + 1]; ind++)

{

if (jg[ind] == temp.vert) {

exist = true;

break;

}

}

if (exist)

ggu[ind] = 0;

}

}

}

void FEM::AddToGlobal(int i, int j, double add)

{

if (i == j)

di[i] += add;

else if (i < j) {

int ind;

for (ind = ig[j]; ind < ig[j + 1]; ind++)

{

if (jg[ind] == i)

break;

}

ggu[ind] += add;

}

else {

int ind;

for (ind = ig[i]; ind < ig[i + 1]; ind++)

{

if (jg[ind] == j)

break;

}

ggl[ind] += add;

}

}

void FEM::AllocateGlobalMatrix()

{

di = new double[globalN]();

b = new double[globalN]();

ggl = new double[ig[globalN] - ig[0]]();

ggu = new double[ig[globalN] - ig[0]]();

}

void FEM::FreeMemory()

{

delete[] di;

delete[] b;

delete[] ggl;

delete[] ggu;

delete[] ig;

delete[] jg;

}

void FEM::FormB(Triangle tri)

{

double factor = fabs(DetD(tri)) / 24.0;

localB[0] = factor \* (2.0 \* Function(tri.vert1, tri.region) + Function(tri.vert2, tri.region) + Function(tri.vert3, tri.region));

localB[1] = factor \* (Function(tri.vert1, tri.region) + 2.0 \* Function(tri.vert2, tri.region) + Function(tri.vert3, tri.region));

localB[2] = factor \* (Function(tri.vert1, tri.region) + Function(tri.vert2, tri.region) + 2.0 \* Function(tri.vert3, tri.region));

}