|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра прикладной математики | | |
| Курсовой проект | | |
| по дисциплине «Численные методы» | | |
| . | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМ-13 |
| Студент: | Буданцев д. Е. |
| Вариант | 79, 11 |
|  |  |
|  |  |
| Преподаватели | сивенкова анастасия павловна |
|  |  |
| Новосибирск, 2024 | | |

1. **Постановка задачи:**

Решить методом конечных элементов двухмерную краевую задачу для гиперболического уравнения в цилиндрической системе координат (r, z). Вид уравнения:



(коэффициент диффузии) – зависит только от z;



 - известная функция двух переменных

 - известная функция двух переменных

Краевые условия всех типов:





Вид базисных функций: биквадратичные на четырёхугольниках.

Формат хранения матрицы СЛАУ: разреженный строчно-столбцовый.

Метод решения СЛАУ: метод сопряженных градиентов.

Используемая схема: четырёхслойная неявная схема для аппроксимации по времени.

1. **Теоритическая часть:**

**2.1 Дискретизация по времени**

Введём сетку по времени, разбив необходимый отрезок точками . Рассмотрим отрезок  и представим на нём нашу функцию в виде полиномов Лагранжа:

(1.2) ,

где  - значения функции  при  соответственно,

а  - кубические полиномы равные 1 при , и 0 при .

Данные функции времени выглядят следующим образом:

 где 

Подставим вид искомой функции (1.2) в (1.1). Для этого надо вычислить первые и вторые производные от функции  в точке  (т.к. схема неявная):

После подстановки уравнение (1.1) примет вид:

 Перенесём все известные компоненты в правую часть:



Для краткости выкладок обозначим

**2.2 Вариационная постановка:**

Учитывая (1.3) уравнение (1.1) принимает вид:

(1.4) 

Выполним вариационную постановку методом Бубнова-Галёркина. Введём гильбертово пространство . В нём скалярное произведение определяется следующим отбразом:



И норма, ассоциированная со скалярным произведением: 

В общем виде постановка Бубнова –Галёркина для операторного уравнения  записывается в следующем виде:

(1.5)  

Где . Для уравнения (1.4) постановка (1.5) примет вид:

Применив формулу Грина к этому выражению получим:

(1.6)  

* 1. **Конечноэлементная дискретизация:**

Разобьём область  на непересекающие подобласти – конечные элементы: . В соответсвии с заданием  - это выпуклый четырёхугольник и каждый четырёхугольник задан координатами его четырёх вершин , где . При этом вершины должны быть пронумерованны так, чтобы на одной стороне четырёхугольника не лежали первая и четвёртая вершины (и соответственно вторая и третья). Тогда формулу (1.5) можно переписать в виде:



Заменим пространство  на конечномерное пространство , которое определим как линейное пространство натянутое на базисные функции . Заменим в (1.5) функцию аппроксимирующей ее функцией , а функцию  - функцией  и получим аппроксимацию уравнения Галеркина

(1.7) 

Поскольку любая функция  может быть представлена в виде линейной комбинации , вариационное уравнение (1.7) эквивалентно следующей системе уравнений:

(1.8) 

может быть представлено в виде:  (1.9), причем  компонент вектора весов  могут быть фиксированы и определены из условия . Подставляя (1.9) в (1.8) получим СЛАУ для компонент  вектора весов :



Конечно элементная СЛАУ может быть записана в матричном виде: , где:

 (1.10)



В области  выберем биквадратичный финитный базис , обладающие следующими свойствами:

1. Базисная функция  ассоциирована с узлом конечноэлементной сетки , причём:



1. Базисная функция  отличная  от нуля только на тех конечных элементах, которые содержат точку 
   1. **Локальные матрицы и вектора конечных элементов**

Построим на четырёхугольном конечном элементе  локальные базисные функции с помощью биквадратичных функций, заданных на шаблонном элементе , являющемся единичным квадратом:

(2.1) 

Отобразим  в четырёхугольник  с вершинами  с помощью следующих соотношений:

(2.2) 

(2.3) 

где

 , 

, 

Введём на биквадратичные базисные функции:

, , 

, , 

, , 

Где,  где  - одномерные функции следующего вида:

, , 

Тогда базисные функции на четырёхугольниках  можно определить с помощью соотношений

,,

,,

,,

, ,



Вычислим Якобиан (1.9)  преобразования координат (2.2), (2.3):



где коэффициенты  имеют следующий вид:







При этом следует учесть, что якобиан в силу взаимной однозначности отображения (2.2) – (2.3) на нигде не отображается в ноль и поэтому не меняет знак. Тогда модуль якобиана может быть вычислен как



Так как используется цилиндрическая система координат, исходное уравнение может быть представлено в виде:



Выражения для вычисления компонент локальных матриц в этом случае принимают вид:

Так как функция на элементе  - это образ функции , определённой на шаблонном элементе , то





 Вычислим производные  и . Используя правила дифференцирования сложной функции, получим





Применяя формулы Крамера (их применения возможно в силу того что ), получаем





где



ТогдаЛокальный вектор правой части , будем вычислять с учетом того, что функция  на конечном элементе  представлена в виде разложения по базисным функциям , где n- число локальных базисных функций конечного элемента , а  - это значения f(r,z) на узлах. Тогда





Вычислять интегралы массы (2.4)-(2.5) и жёсткости (2.7) буду числено с помощью численного метода Гаусса. Квадратурные формулы при интегрировании по прямоугольнику (с тремя узлами по каждой из координат) имеют следующий вид:



где









* 1. **Локальные матрицы и вектора рёбер, на которых заданы краевые условия второго и третьего рода**

Локальная матрица ребра Г c заданным на нём краевым условием третьего рода имеет вид:



Будем считать что  , если нет то находим среднее значение 

Представим r в виде разложения по базисным функциям  тогда (3.1) можно переписать в виде:



где якобиан r был представлен в виде линейной комбинации базисных функций.



Локальный вектор  этого ребра Г при представлении  на  в виде разложения по локальным базисным функциям (т.е.  ) имеет вид:



Локальный вектор ребра  с заданным на нём условием второго рода получается аналогичным способом,  на  представим в виде разложения по базисным функциям (т.е.  ), получается:



В матричном виде (1.1) будет выглядеть:



1. **Описание разработанной программы:**

Описание входных данных приведены в таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| Имя файла | Данные | Описание способа  Задание данных | Пример |
| Nodes.txt | Граничные узлы | В каждой новой строке перечисляются парами координаты узлов, в порядке слево на право, снизу вверх. Количество пар 4. | 0 1  1 0  2 3  3 2 |
| El.txt | Коэффициент разрядки и количество разбиений | Указывается сначала коэфицент растяжки следом количество разбиений | 1,3 2 |
| time.txt | Промежуток времени, коэфициент разрядки и количество разбиений | В первой строчке указывается промежуток времени через пробел. Во второй строчке указывается сначала коэфициент разрядки следом количество разбиений | 0 2  1 10 |

Данные о значениях хранятся внутри программы в соответствующих функиях.

Использовались такие структуры данных, std::vector, std::set, как контейнер с автоматическим упорядочиванием. std::function для хранения базисных функций и их производных и передачи функций в подпрограмму решения интеграла методом Гаусса. std::tuple для более компактной передачи координат.

**3.1 Структура разработанной программы:**

Основные чисти программы:

* *Глобальная матрица и вектора правой части* – инициазизация переменных глобальной матрицы и векторов главной части
* *Конечноэлементная сетка -* инициазизация переменных и массивов храняжие данные о сетке
* *Прототипы –* Все прототипы функций, которые используются в программе
* *Реализация функций* – Реализация функций прототипов.
* *Тестирование –* Программа для получения относительной нормы вектора погрешности полученного решения. И для вывода нужной информации

**3.1.1 Реализация функций**

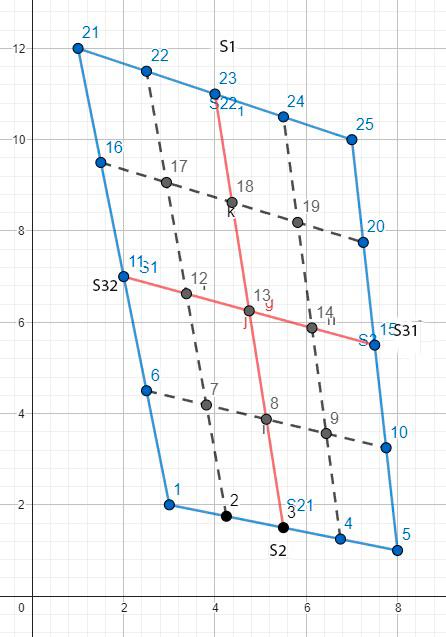
Перечислим основные функции этой части кода:

* *Реализация Загрузок и обработок сеток –* в этой части кода описаны методы загрузок данных с файлов, генерация сетки, дополнительные функции для работы с сеткой
  + *LoadNet –* подпрограмма, которая загружает данные из файлов и на их основе генерирует сетку
  + *LoadTNet – подпрограмма, которая загружает данные из файлов и на их основе генерирует сетку по времени*
  + *GetCordIK –* подпрограмма, которая возвращает координаты узла, нужна для подстраховки от проблем с генерацией сетки
  + *InitQJ – инициализация хранилища для векторов решений *
  + *InitFirstQ – заполнение первых векторов решений с помощью краевых условий по времени*
* *Реализация загрузок базисных функций-* инициализируются массивы, в которых хранятся базисные функции и их производные
  + *FiniteFunc –* возвращает значение финитной функции
  + *DerivFiniteFunc –* Возвращает значение производной финитной функции
  + *InitBaseFunction –* Инициализируют массив с базисными функциями
  + *InitDPerivativeBaseFunction –* Инициализируют массив с производными от базисных функций
* *Реализация функций краевых условий –* в этой части кода описаны функции  также учёт краевых условий.
  + *ug –* соответствующая функция
  + *teta –* соответствующая функция 
  + *beta –* соответствующая функция 
  + *u\_beta –* соответствующая функция 
  + *u0 – соответсвующая функция *
  + *u0 – соответсвующая функция *
  + *EnterBoundCondit –* ввод краевых условий с консоли
  + *FindInd –* поиск индекса в списке
  + *ConsiderBoundConditFirstType –* учёт первых краевых условий
  + *AddLocalVecBound –*  добавление локального вектора в вектор правой части
  + *AddLocalMartBound\_3 –* добавление локальной матрицы в глобальную для 3 краевых условий
  + *ConsiderBoundConditThirdType –* учёт третьих краевых условий
  + *ConsiderBoundConditionSecType -* учёт вторых краевых условий
  + *BuildBoundMetrices –* инициализация краевых матриц
  + *BoundCondid –* формирование списка в котором перечислены все раздроблённые рёбра для каждого краевого условия
  + *ConsiderBoundCondit –* запуск учёта всех трёх краевых условий
* *Реализация функций параметров уравнения –* в этой части кода присутствуют реализации функций 
  + *RightFunction –* выводит значение правой *функции *
  + *lambda –* выводит значение *лямбды *
  + *sigma – соответствующая функция *
  + *xi\_d – соответствующая функция *
* *Реализация построений Альфа и Мю –* в этой части кода расчитываются значения Альфы и Мю на конечном элементе
  + *CalcVecAlpha –* вычислить значения Альфа на конечном элементе
  + *CalcVecMyu –* вычислить значения Мю на конечном элементе
* *Реализация Гаусса –* содержит подпрограмму вычисления интегралла численным методом Гаусса
  + *Gauss2Rod –* вычисление интегралла заданной функции методом численного интегрирования Гаусса
* *Построение глобальной матрицы и вектора –* в этой части собраны функции для вычислений матриц масс, жёсткости локального вектора правой части, функции добавления локальных матриц и локального вектора к глобальным, генерация портрета матрицы. Также дополнительные к ним функции.
  + *GeneratePortrait –* функция генерирует портрет матрицы
  + *add\_local\_to\_global –* функция переводящая локальные матрицы к глобальной также переводит локальный вектор к глобальному вектору
  + *LocalCalcMK –* Вычисление матрицы массы
  + *LocalCalcGK –* Вычисление матрицы жёсткости
  + *RPsyEta –* возврачает значения r принимая параметры * и *
  + *ZPsyEta -* возврачает значения z принимая параметры * и *
  + *SumVector –* Суммирует заданные векторы
  + *SubVector – Разница* заданных векторов
  + *EqVector – Присвоить значения одного вектора другому вектору*
  + *LocalCalcVectorFEK –* вычисляет локальный вектор правой части
  + *InitVectors –* Инициализирует вектора для матрицы жёсткости, массы и локально вектора
  + *MultMKVector – Умножение матрицы массы на вектор*
  + *MultVectorConstant – Умножение вектора на константу*
  + *GetJLayer\_3SCHEME –* Создаёт глобальную матрицу A и глобальный вектор правой части без учёта краевых условий используя неявную трёхслойную схему
  + *GetJLayer\_4SCHEME –* Создаёт глобальную матрицу A и глобальный вектор правой части без учёта краевых условий используя неявную четырёхслойную схему
  + *Calc\_deltat\_3SCHEME – Вычисляет вектор хранящий значения  для неявной трёхслойной схемы*
  + *Calc\_deltat\_4SCHEME - Вычисляет вектор хранящий значения  для неявной четырёхслойной схемы*
  + *GetQpj – Получить часть вектора решений для заданного слоя и конечного элемента*
* *Реализация MSG –* в этой части кода используется реализация MSG или метод сопряжённых градиетов которая была получена на занятиях по численным методам
  + ScalarMult – функция, вычисляющая скалярное произведение передаваемых векторов в евклидовом пространстве;
  + MultMartVec – подпрограмма перемножающая матрицу СЛАУ на заданный вектор;
  + LU\_sq – подпрограмма, реализующая неполную факторизацию матрицы;
  + Ly\_f – подпрограмма нахождения вектора для нижнетреугольной матрицы и вектора правой части при решении СЛАУ;
  + Ly\_f\_transp f – подпрограмма нахождения вектора для верхнетреугольной матрицы и вектора правой части при решении СЛАУ;
  + PrMinusR – подпрограмма вычисляющая разность вектора правой части и передаваемого вектора;
  + FirstEqualSecond – подпрограмма равенства первого передаваемого вектора второму;
  + CulcResid – функция, вычисляющая относительную невязку;
  + CulcX – подпрограмма, вычисляющая вектор решения ;
  + CulcR – подпрограмма, вычисляющая вектор невязки ;
  + CulcZ – подпрограмма, вычисляющая вектор спуска ;
  + CulcAlpha – функция, вычисляющая параметр МСГ;
  + CulcBeta – функция, вычисляющая параметр МСГ;
  + CulcMult – подпрограмма, вычисляющая произведения матрицы на вектор невязки;
  + LU\_sq\_MSG – подпрограмма, осуществяющая решения системы методом сопряженных градиентов.
  + ClearVectorsLU – очищает векторы для возобновления вычисления

**3.1.2 Тестирование**

* *Test –* подпрограмма вычисляющая относительную норму вектора погрешности.

1. **Расчётная область**

****

На расчётной области числами 1, 2, 3, …, 25 обозначены номера ущлов, будем считать что все конечные элементы лежат в одной подобласти.

Узлы расчётной области:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|  | 3 | 4.25 | 5.5 | 6.75 | 8 | 2.5 | 3.8125 | 5.125 | 6.4375 | 7.75 | 2 |
|  | 2 | 1.75 | 1.5 | 1.25 | 1 | 4.5 | 4.1875 | 3.875 | 3.5625 | 3.25 | 7 |
|  | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|  | 3.375 | 4.75 | 6.125 | 7.5 | 1.5 | 2.9375 | 4.375 | 5.8125 | 7.25 | 1 | 2.5 |
|  | 6.625 | 6.25 | 5.875 | 5.5 | 9.5 | 9.0625 | 8.625 | 8.1875 | 7.75 | 12 | 11.5 |
|  | 23 | 24 | 25 |  |  |  |  |  |  |  |  |
|  | 3.375 | 4.75 | 6.125 |  |  |  |  |  |  |  |  |
|  | 6.625 | 6.25 | 5.875 |  |  |  |  |  |  |  |  |

1. **Описание тестирования программы:**

* Тест 1 – проверка учёта краевых условий второго и третьего рода



Краевые условия:



Относительная норма вектора погрешности полученных решений:

|  |  |
| --- | --- |
|  | Относительная погрешность |
| 0.57142857 | 3.48985683804144e-16 |
| 0.85714286 | 8.17267353527965e-16 |
| 1.14285714 | 1.52379977492264e-15 |
| 1.42857143 | 2.29065138282659e-15 |
| 1.71428571 | 2.64477623082479e-15 |
| 2 | 2.57134039645473e-15 |

1. **Проведенные исследования**
   * Исследование 1 – определение порядка аппроксимации по времени



Краевые условия:



Относительная норма вектора погрешности полученных решений:

|  |  |
| --- | --- |
|  | Относительная погрешность |
| 1.33333333 | 1.83060593881747e-16 |
| 2 | 2.63420846913993e-16 |



Краевые условия:



Относительная норма вектора погрешности полученных решений:

|  |  |
| --- | --- |
|  | Относительная погрешность |
| 2 | 1.94264551951282e-16 |



Краевые условия:



Относительная норма вектора погрешности полученных решений:

|  |  |
| --- | --- |
|  | Относительная погрешность |
| 2 | 0.0837109940085924 |

*Вывод:* Результат показал, что порядок аппроксимации равен трём, что совпадает с теоретическим значением, поскольку порядок аппроксимации равен порядку базисных функций (полинома Лагранжа). При использовании трёхслойной схемы для получения **** порядок падает до 2.

* + Исследование 2 – определение порядка сходимости по времени



Краевые условия:



**Зафиксируем **

|  |  |  |
| --- | --- | --- |
|  | Относительная погрешность (g) на 4 слое |  |
| *5* | 0.732895799010663 | - |
| *9* | 0.21418625883221 | 1.774742168130802 |
| *17* | 0.0632560706471841 | 1.7595900804889508 |
| *33* | 0.0178758042632062 | 1.8231957883447052 |
| *65* | 0.00482074546433969 | 1.890679990214828 |

*Вывод*: Как можно заметить из таблицы, порядок сходимости стремится к 2. Данные вычисления были произведены с использованием 3-x слойной схемы для получения значения на . Используя точное значение  порядок сходимости не меняется и по прежнему остаётся равным 2.

1. **Тексты основных модулей программ.**

#include <iostream>

#include <Windows.h>

#include <stdio.h>

#include <math.h>

#include <iomanip>

#include <functional>

#include <tuple>

#include <fstream>

#include <set>

#include <vector>

using namespace std;

// Реализация функции sign

template <typename T> int sign(T val) {

    return (T(0) < val) - (val < T(0));

}

template <typename T> int ElemInList(T\* List, T Elem, int n) {

    for (int i = 0; i < n; i++)

        if (List[i] == Elem) return i;

    return -1;

}

#pragma region Глобальная матрица и вектор правой части

vector<vector<vector<int>>> edge;

vector<vector<double>> A\_1, A\_2, A\_3;

vector<double> ggl, di, b, q, L\_sq, di\_sq;

vector<int> ig, jg;

//вектор весов, результат работы программы

int\* choise;

#pragma endregion

#pragma region Конечноэлементная сетка

int Wn, Dn; // Размерность массивов W, D

function<double(double, double)> \*W; //Базисные функции

function<double(double, double)>\* D[2]; //Производные от базисных функций

double koef\_r; //Коэффициент растяжения по r

int nr; // Количество разбиений по r и z  соответственно

int\* numfe[9];       // номера узлов конечных элементов

double\* rz[2];       // координаты узлов

int NodesCount;      //количество узлов

int FECount;         //количество КЭ

int NodesCountR;

#pragma endregion

#pragma region Локальные матрицы Жёсткости и Массы и вестора правой части

//Локальные матрицы жёсткости и массы

int GLen, MLen; //Размерность массивов G, M

double alpha[3]; //Элементы альфа которые участвуют при счёте Якоби

double Myu[6]; //Коэффициенты Мю которые участвуют после решения Крамера

double\* MkXi,

      \* MkSig,

      \* Gk;

double\* bk;

#pragma endregion

#pragma region Сетка по времени

int num\_t; // Количество узлов в сетке

vector<double> tj; //Массив с значениями tj

double koef\_t; // Коэффициент растяжения по оси времени

#pragma endregion

#pragma region Вектор решений или вектор qj и константы

vector<vector<double>> qj; // Размер данного вектора будет ограничен до 4 необходимых. Чтобы меньше тратить памяти.

#pragma endregion

#pragma region Прототипы

#pragma region Прототипы Гаусса

double Gauss2rod(function<double(double, double, tuple<int, int, int>, double)> Ifun, tuple<int, int, int> FuncIndex, double time);

double Gauss2rod(function<double(double, double, tuple<int, int, int>)> Ifun, tuple<int, int, int> FuncIndex);

#pragma endregion

#pragma region Прототипы параметров уравнения

double lambda(double z);

double RightFunction(double r, double z, double time);

#pragma endregion

#pragma region Прототипы краевых условий

double ug(int IndexFunc, double r, double z, double t);

double teta(int IndexFunc, double r, double z, double t);

double beta();

double sigma(double r, double z);

double xi\_d(double r, double z);

double u\_beta(int IndexFunc, double r, double z, double t);

double u0(double r, double z);

double u1(double r, double z);

double u2(double r, double z);

void EnterBoundCondit();

int GetStepEndEl(int i);

int GetStartPoint(int i);

bool FindInd(int i);

void ConsiderBoundConditFirstType(int node\_num, double t);

void ConsiderBoundConditSecType(int num\_edge, double t);

void AddLocalMartBound\_3(int num\_edge, vector<vector<double>> matr);

void AddLocalVecBound(int bound\_condit, int num\_edge, vector<double> vec);

void ConsiderBoundConditThirdType(int num\_edge, double t);

void BuildBoundMatrices();

void BoundCondit();

void ConsiderBoundCondit(double t);

#pragma endregion

#pragma region Прототипы загрузки и обработки сетки

void LoadNet(bool debag = false); // Загрузка сетки

void LoadTNet();

void InitQJ();

void InitFirstQ();

tuple<double, double> GetCordIK(int i, int IndexFE);

#pragma endregion

#pragma region Прототипы загрузки базисных функций и их производных

double FiniteFunc(int i, double var);

double DerivFiniteFunc(int i, double var);

void InitBaseFunction();

void InitDРerivativeBaseFunction();

#pragma endregion

#pragma region Прототипы построений векторов Альфа и Мю

void CalcVecAlpha(unsigned int indexFE);

void CalcVecMyu(unsigned int indexFE);

#pragma endregion

#pragma region Построение глобальной матрицы и вектора

void LocalCalcGK(int IndexFE);

double RPsyEta(double Psy, double Eta, int IndexFe); // r с заменой меременной

double ZPsyEta(double Psy, double Eta, int IndexFE);

void LocalCalcVectorFEK(int IndexFE, double time);

void SumVector(double\* &FirstVec, double \*SecondVec, int n);

void MultVectorConstant(double\*& FirstVec, double Constant, int n);

void InitVectors();

#pragma endregion

#pragma endregion

#pragma region Реализация функций

#pragma region Реализация Загрузок и обработок сеток

void LoadNet(bool debag) {

    ifstream NodesFile("Nodes.txt");

    if (!NodesFile.is\_open())

        throw "Файл Nodes.txt не удалось открыть";

    double\* rz\_area[2]; // Координаты области

    rz\_area[0] = new double[4]; rz\_area[1] = new double[4];

    for (int i = 0; i < 4; i++) {

        NodesFile >> rz\_area[0][i] >> rz\_area[1][i];

    }

    NodesFile.close();

    ifstream FEFile("fe.txt");

    if (!FEFile.is\_open()) {

        throw "Файл fe.txt не удалось открыть";

    }

    FEFile >> koef\_r;

    //Генерация сетки

    double\* r0[2], \* r1[2];

    FEFile >> nr;

    FEFile.close();

    NodesCountR = 2 \* nr + 1;

    NodesCount = NodesCountR \* NodesCountR;

    FECount = nr \* nr;

    for (int i = 0; i < 2; i++) {

        rz[i] = new double[NodesCount];

        r0[i] = new double[NodesCountR];

        r1[i] = new double[NodesCountR];

    }

    double rh, zh, max\_val;

    if (abs(koef\_r - 1) < 1e-15) {

        rh = (rz\_area[0][2] - rz\_area[0][0]) / (NodesCountR - 1);

        zh = (rz\_area[1][2] - rz\_area[1][0]) / (NodesCountR - 1);

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

            r0[0][i] = rz\_area[0][0] + rh \* i;

            r0[1][i] = rz\_area[1][0] + zh \* i;

        }

        rh = (rz\_area[0][3] - rz\_area[0][1]) / (NodesCountR - 1);

        zh = (rz\_area[1][3] - rz\_area[1][1]) / (NodesCountR - 1);

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

            r1[0][i] = rz\_area[0][1] + rh \* i;

            r1[1][i] = rz\_area[1][1] + zh \* i;

        }

        for (int i = 0, p = 0; i < NodesCountR; i++) {

            rh = (r1[0][i] - r0[0][i]) / (NodesCountR - 1);

            zh = (r1[1][i] - r0[1][i]) / (NodesCountR - 1);

            for (int j = 0; j < NodesCountR; j++, p++) {

                rz[0][p] = r0[0][i] + rh \* j;

                rz[1][p] = r0[1][i] + zh \* j;

            }

        }

    }

    else {

        rh = (rz\_area[0][2] - rz\_area[0][0]);

        zh = (rz\_area[1][2] - rz\_area[1][0]);

        if (koef\_r > 1) {

            max\_val = pow(koef\_r, NodesCountR - 2);

            r0[0][0] = rz\_area[0][0];

            r0[1][0] = rz\_area[1][0];

            for (int i = 0; i < NodesCountR - 1; i++) {

                r0[0][i + 1] = rz\_area[0][0] + rh \* pow(koef\_r, i) / max\_val;

                r0[1][i + 1] = rz\_area[1][0] + zh \* pow(koef\_r, i) / max\_val;

            }

            rh = (rz\_area[0][3] - rz\_area[0][1]);

            zh = (rz\_area[1][3] - rz\_area[1][1]);

            r1[0][0] = rz\_area[0][1];

            r1[1][0] = rz\_area[1][1];

            for (int i = 0; i < NodesCountR - 1; i++) {

                r1[0][i + 1] = rz\_area[0][1] + rh \* pow(koef\_r, i) / max\_val;

                r1[1][i + 1] = rz\_area[1][1] + zh \* pow(koef\_r, i) / max\_val;

            }

            int p = 0;

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

                rh = (r1[0][i] - r0[0][i]);

                zh = (r1[1][i] - r0[1][i]);

                rz[0][p] = r0[0][i];

                rz[1][p] = r1[1][i];

                p += 1;

                for (int j = 0; j < NodesCountR - 1; j++) {

                    rz[0][p] = r0[0][i] + rh \* pow(koef\_r, j) / max\_val;

                    rz[1][p] = r0[1][i] + zh \* pow(koef\_r, j) / max\_val;

                    p += 1;

                }

            }

        }

        else {

            koef\_r += 1;

            double max\_val = pow(koef\_r, NodesCountR - 2);

            r0[0][NodesCountR - 1] = rz\_area[0][2];

            r0[1][NodesCountR - 1] = rz\_area[1][2];

            rh \*= -1; zh \*= -1;

            for (int i = 0; i < NodesCountR - 1; i++) {

                r0[0][NodesCountR - i - 2] = rz\_area[0][2] + rh \* pow(koef\_r, i) / max\_val;

                r0[1][NodesCountR - i - 2] = rz\_area[1][2] + zh \* pow(koef\_r, i) / max\_val;

            }

            rh = -(rz\_area[0][3] - rz\_area[0][1]);

            zh = -(rz\_area[1][3] - rz\_area[1][1]);

            r1[0][NodesCountR - 1] = rz\_area[0][3];

            r1[1][NodesCountR - 1] = rz\_area[1][3];

            for (int i = 0; i < NodesCountR - 1; i++) {

                r1[0][NodesCountR - i - 2] = rz\_area[0][3] + rh \* pow(koef\_r, i) / max\_val;

                r1[1][NodesCountR - i - 2] = rz\_area[1][3] + zh \* pow(koef\_r, i) / max\_val;

            }

            int p = NodesCount - 1;

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

                rh = -(r1[0][NodesCountR - i - 1] - r0[0][NodesCountR - i - 1]);

                zh = -(r1[1][NodesCountR - i - 1] - r0[1][NodesCountR - i - 1]);

                rz[0][p] = r0[0][NodesCountR - i - 1];

                rz[1][p] = r1[1][NodesCountR - i - 1];

                p--;

                for (int j = 0; j < NodesCountR - 1; j++, p--) {

                    rz[0][p] = r1[0][NodesCountR - i - 1] + rh \* pow(koef\_r, j) / max\_val;

                    rz[1][p] = r1[1][NodesCountR - i - 1] + zh \* pow(koef\_r, j) / max\_val;

                }

            }

        }

    }

    //Заполнение массива rz\_fe

    for (int i = 0; i < 9; i++) numfe[i] = new int[FECount];

    int xn = 0;

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

        if (i != 0 && i % nr == 0) xn += NodesCountR + 1;

        for (int j = 0; j < 3; j++) {

            numfe[j][i] = xn; numfe[j + 3][i] = xn + NodesCountR; numfe[j + 6][i] = xn + 2 \* NodesCountR;

            if (j == 2) continue;

            xn++;

        }

    }

    if (debag) {

        printf("Print Num Global Function\n");

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

            printf("%d %d %d\n", numfe[6][k], numfe[7][k], numfe[8][k]);

            printf("%d %d %d\n", numfe[3][k], numfe[4][k], numfe[5][k]);

            printf("%d %d %d\n", numfe[0][k], numfe[1][k], numfe[2][k]);

        }

    }

}

tuple<double, double> GetCordIK(int i, int IndexFE) {

    int NumGlobalFunc = numfe[i][IndexFE];

    return make\_tuple(rz[0][NumGlobalFunc], rz[1][NumGlobalFunc]);

}

void LoadTNet() {

    ifstream FileT("time.txt");

    if (!FileT.is\_open()) {

        throw "Файл time.txt не открылся";

    }

    double max\_t, min\_t;

    FileT >> min\_t >> max\_t >> koef\_t >> num\_t;

    FileT.close();

    if (koef\_t <= 0 || num\_t <= 0) {

        throw "Сетка по времени заданна не верно";

    }

    tj.resize(num\_t);

    if (abs(koef\_t - 1) < 1e-15) {

        double h0 = (max\_t - min\_t) / (num\_t - 1);

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

            tj[i] = min\_t + h0 \* i;

        }

    }

    else {

        double h0 = (max\_t - min\_t) \* (koef\_t - 1) / (pow(koef\_t, num\_t - 1) - 1);

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

            tj[i] = min\_t + h0 \* (pow(koef\_r, i) - 1) / (koef\_t - 1);

        }

    }

}

void InitQJ() {

    qj.resize(3);

    for (int i = 0; i < 3; i++) {

        qj[i].resize(NodesCount, 0);

    }

}

void InitFirstQ() {

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

        qj[0][i] = u0(rz[0][i], rz[1][i]);

        qj[1][i] = u1(rz[0][i], rz[1][i]);

        //qj[2][i] = u2(rz[0][i], rz[1][i]);

    }

}

#pragma endregion

#pragma region Реализация загрузок базисных функций и и производных

double FiniteFunc(int i, double var) {

    if (i == 0) {

        return 2 \* (var - 0.5) \* (var - 1.0);

    }

    else if (i == 1) {

        return -4 \* var \* (var - 1.0);

    }

    else return 2 \* var \* (var - 0.5);

}

double DerivFiniteFunc(int i, double var) { // производная финитной функции

    if (i == 0) {

        return 4 \* var - 3;

    }

    else if (i == 1) {

        return -8 \* var + 4;

    }

    else return 4 \* var - 1;

}

void InitBaseFunction() {

    /\*int Wn, Dn; // Размерность массивов W, D

      function<double(double)> \*W; //Базисные функции

      function<double(double)>\* D; //Производные от базисных функций

    \*/

    Wn = 9;

    W = new function<double(double, double)> [Wn];

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

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

            W[p] = [i, j](double Psy, double Eta) {return FiniteFunc(j, Psy) \* FiniteFunc(i, Eta);  };

        }

}

void InitDРerivativeBaseFunction() {

    Dn = Wn \* 2;

    D[0] = new function<double(double, double)>[9]; D[1] = new function<double(double, double)>[9];

    for (int i = 0, p = 0; i < 3; i++) {

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

            D[0][p] = [i, j](double Psy, double Eta) {return DerivFiniteFunc(j, Psy) \* FiniteFunc(i, Eta);  };

            D[1][p] = [i, j](double Psy, double Eta) {return FiniteFunc(j, Psy) \* DerivFiniteFunc(i, Eta);  };

        }

    }

}

#pragma endregion

#pragma region Реализация функций краевых условий

double ug(int IndexFunc, double r, double z, double t) {

    return exp(3 \* t);

}

double teta(int IndexFunc, double r, double z, double t) {

    if (IndexFunc == 0) {

        return (-2 \* r) / sqrt(26);

    }

    else {

        return (-2 \* r) / sqrt(26);

    }

}

double beta() {

    return 1;

}

double u0(double r, double z) {

    return exp(3 \* tj[0]);

}

double u1(double r, double z) {

    return exp(3 \* tj[1]);

}

double u2(double r, double z) {

    return exp(3 \* tj[2]);

}

double u\_beta(int IndexFunc, double r, double z, double t) {

    if (IndexFunc == 0) {

        return pow(r, 2) + t + 18 \* r / sqrt(82);

    }

    else {

        return pow(r, 2) + t - 10 \* r / sqrt(26);

    }

}

void EnterBoundCondit() { // ввод краевых условий

    cout << "Введите тип краевого условия на соответствующем ребре:" << endl;

    cout << "1 - первое краевое условие;" << endl;

    cout << "2 - второе краевое условие;" << endl;

    cout << "3 - третье краевое условие;" << endl << endl;

    choise = new int[4];

    cout << "Краевое условие нижнего ребра:" << endl;

    cin >> choise[0];

    cout << endl;

    cout << "Краевое условие правого ребра:" << endl;

    cin >> choise[1];

    cout << endl;

    cout << "Краевое условие верхнего ребра:" << endl;

    cin >> choise[2];

    cout << endl;

    cout << "Краевое условие левого ребра:" << endl;

    cin >> choise[3];

    cout << endl;

}

int GetStepEndEl(int i) { // задаем шаг до след конеч. эл.

    if (i == 0 || i == 2) {

        return 2;

    }

    else return 2 \* NodesCountR;

}

int GetStartPoint(int i) { // задаем начальную точку ребра

    switch (i)

    {

    case 0:

        return 0;

        break;

    case 1:

        return NodesCountR - 1;

        break;

    case 2:

        return NodesCount - NodesCountR;

        break;

    case 3:

        return 0;

        break;

    }

}

bool FindInd(int i) {

    for (int j = 0; j < edge[0].size(); j++) {

        if (edge[0][j][0] == i) {

            return true;

        }

    }

    return false;

}

void ConsiderBoundConditFirstType(int node\_num, double t) { // учет краевых условий первого типа

    double \_r, \_z;

    \_r = rz[0][edge[0][node\_num][0]], \_z = rz[1][edge[0][node\_num][0]];

    b[edge[0][node\_num][0]] = ug(edge[0][node\_num][1], \_r, \_z, t);

    di[edge[0][node\_num][0]] = 1;

    for (int i = ig[edge[0][node\_num][0]]; i < ig[edge[0][node\_num][0] + 1]; i++) {

        int \_i = jg[i];

        if (FindInd(\_i)) {

            ggl[i] = 0;

            continue;

        }

        b[\_i] -= b[edge[0][node\_num][0]] \* ggl[i];

        ggl[i] = 0;

    }

    for (int i = edge[0][node\_num][0]; i < NodesCount; i++) {

        int k = 0;

        for (int j = ig[i]; j < ig[i + 1]; j++) {

            if (jg[j] == edge[0][node\_num][0]) {

                if (FindInd(i)) {

                    ggl[j] = 0;

                    continue;

                }

                b[i] -= b[edge[0][node\_num][0]] \* ggl[j];

                ggl[j] = 0;

            }

        }

    }

}

void ConsiderBoundConditSecType(int num\_edge, double t) { // учет краевых условий второго типа

    vector<double> \_r(3), \_z(3);

    \_r[0] = rz[0][edge[1][num\_edge][0]], \_z[0] = rz[1][edge[1][num\_edge][0]];

    \_r[1] = rz[0][edge[1][num\_edge][1]], \_z[1] = rz[1][edge[1][num\_edge][1]];

    \_r[2] = rz[0][edge[1][num\_edge][2]], \_z[2] = rz[1][edge[1][num\_edge][2]];

    double h = sqrt(pow(\_r[0] - \_r[2], 2) + pow(\_z[0] - \_z[2], 2));

    vector<double> b\_s2(3, 0);

    vector<double> \_theta(3);

    \_theta[0] = teta(edge[1][num\_edge][3], \_r[0], \_z[0], t);

    \_theta[1] = teta(edge[1][num\_edge][3], \_r[1], \_z[1], t);

    \_theta[2] = teta(edge[1][num\_edge][3], \_r[2], \_z[2], t);

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            b\_s2[i] += (A\_1[i][j] \* \_r[0] + A\_2[i][j] \* \_r[1] + A\_3[i][j] \* \_r[2]) \* \_theta[j];

        }

        b\_s2[i] \*= h \* beta() / double(420);

    }

    AddLocalVecBound(1, num\_edge, b\_s2);

}

void AddLocalMartBound\_3(int num\_edge, vector<vector<double>> matr) { // добавление матрицы из третьего краевого в глобальную

    for (int i = 0; i < 3; i++) {

        di[edge[2][num\_edge][i]] += matr[i][i];

    }

    for (int i = 0; i < 3; i++) {

        int i\_beg = ig[edge[2][num\_edge][i]];

        for (int j = 0; j < i; j++) {

            int i\_end = ig[edge[2][num\_edge][i] + 1];

            while (jg[i\_beg] != edge[2][num\_edge][j]) {

                int ind = (i\_beg + i\_end) / 2;

                if (jg[ind] <= edge[2][num\_edge][j]) i\_beg = ind;

                else i\_end = ind;

            }

            ggl[i\_beg] += matr[i][j];

            i\_beg++;

        }

    }

}

void AddLocalVecBound(int bound\_condit, int num\_edge, vector<double> vec) { // добавление вектора из третьего или второго краевого в глобальный

    for (int i = 0; i < 3; i++) {

        b[edge[bound\_condit][num\_edge][i]] += vec[i];

    }

}

void ConsiderBoundConditThirdType(int num\_edge, double t) { // учет краевых условий третьего типа

    vector<double> \_r(3), \_z(3);

    \_r[0] = rz[0][edge[2][num\_edge][0]], \_z[0] = rz[1][edge[2][num\_edge][0]];

    \_r[1] = rz[0][edge[2][num\_edge][1]], \_z[1] = rz[1][edge[2][num\_edge][1]];

    \_r[2] = rz[0][edge[2][num\_edge][2]], \_z[2] = rz[1][edge[2][num\_edge][2]];

    double h = sqrt(pow(\_r[0] - \_r[2], 2) + pow(\_z[0] - \_z[2], 2));

    vector<vector<double>> \_A(3);

    \_A[0].resize(1);

    \_A[1].resize(2);

    \_A[2].resize(3);

    vector<double> b\_s3(3, 0);

    vector<double> \_u\_beta(3);

    \_u\_beta[0] = u\_beta(edge[2][num\_edge][3], \_r[0], \_z[0], t);

    \_u\_beta[1] = u\_beta(edge[2][num\_edge][3], \_r[1], \_z[1], t);

    \_u\_beta[2] = u\_beta(edge[2][num\_edge][3], \_r[2], \_z[2], t);

    for (int i = 0; i < 3; i++) {

        for (int j = i; j < 3; j++) {

            \_A[j][i] = h \* beta() / double(420) \* (A\_1[j][i] \* \_r[0] + A\_2[j][i] \* \_r[1] + A\_3[j][i] \* \_r[2]);

        }

    }

    AddLocalMartBound\_3(num\_edge, \_A);

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < i; j++) {

            b\_s3[i] += \_A[i][j] \* \_u\_beta[j];

        }

        for (int j = i; j < 3; j++) {

            b\_s3[i] += \_A[j][i] \* \_u\_beta[j];

        }

    }

    AddLocalVecBound(2, num\_edge, b\_s3);

}

void BuildBoundMatrices() { // построение "краевых" матриц

    A\_1.assign(3, vector<double>(3));

    A\_2.assign(3, vector<double>(3));

    A\_3.assign(3, vector<double>(3));

    A\_1[0][0] = 39, A\_2[0][0] = 20, A\_3[0][0] = -3;

    A\_1[0][1] = A\_1[1][0] = 20, A\_2[0][1] = A\_2[1][0] = 16, A\_3[0][1] = A\_3[1][0] = -8;

    A\_1[0][2] = A\_1[2][0] = -3, A\_2[0][2] = A\_2[2][0] = -8, A\_3[0][2] = A\_3[2][0] = -3;

    A\_1[1][1] = 16, A\_2[1][1] = 192, A\_3[1][1] = 16;

    A\_1[1][2] = A\_1[2][1] = -8, A\_2[1][2] = A\_2[2][1] = 16, A\_3[1][2] = A\_3[2][1] = 20;

    A\_1[2][2] = -3, A\_2[2][2] = 20, A\_3[2][2] = 39;

}

void BoundCondit() { // краевые условия

    edge.resize(3);

    int num\_func\_1 = 0, num\_func\_2 = 0, num\_func\_3 = 0;

    for (int i = 0; i < 4; i++) {

        int start\_point = GetStartPoint(i);

        int step = GetStepEndEl(i);

        switch (choise[i])

        {

        case 1:

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

                edge[0].resize(NodesCountR \* (num\_func\_1 + 1));

                edge[0][j + NodesCountR \* num\_func\_1].push\_back(start\_point + j \* step / 2);

                edge[0][j + NodesCountR \* num\_func\_1].push\_back(num\_func\_1);

            }

            num\_func\_1++;

            break;

        case 2:

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

                edge[1].resize(nr \* (num\_func\_2 + 1));

                edge[1][j + nr \* num\_func\_2].push\_back(start\_point);

                edge[1][j + nr \* num\_func\_2].push\_back(start\_point + step / 2);

                edge[1][j + nr \* num\_func\_2].push\_back(start\_point + step);

                edge[1][j + nr \* num\_func\_2].push\_back(num\_func\_2);

                start\_point += step;

            }

            num\_func\_2++;

            break;

        case 3:

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

                edge[2].resize(nr \* (num\_func\_3 + 1));

                edge[2][j + nr \* num\_func\_3].push\_back(start\_point);

                edge[2][j + nr \* num\_func\_3].push\_back(start\_point + step / 2);

                edge[2][j + nr \* num\_func\_3].push\_back(start\_point + step);

                edge[2][j + nr \* num\_func\_3].push\_back(num\_func\_3);

                start\_point += step;

            }

            num\_func\_3++;

            break;

        }

    }

}

void ConsiderBoundCondit(double t) { // учет всех краевых

    for (int i = 0; i < edge[2].size(); i++) { // учет третьих краевых

        ConsiderBoundConditThirdType(i, t);

    }

    for (int i = 0; i < edge[1].size(); i++) { // учет вторых краевых

        ConsiderBoundConditSecType(i, t);

    }

    for (int i = 0; i < edge[0].size(); i++) { // учет первых краевых(не в верхнем цикле, тк должен быть в самом конце)

        ConsiderBoundConditFirstType(i, t);

    }

}

#pragma endregion

#pragma region Реализация функций параметров уравнения

double lambda(double z) {

    return 1;

}

double RightFunction(double r, double z, double time) {

    return 21 \* exp(3 \* time);

}

double sigma(double r, double z) {

    return 1;

}

double xi\_d(double r, double z) {

    return 2;

}

#pragma endregion

#pragma region Реализация построений Альфа и Мю

void CalcVecAlpha(unsigned int indexFE) {

    /\*

    int\* numfe[4],       // номера узлов конечных элементов

       \* numsubarea[2];  // номера подобластей конечных элементов

      double\* rz[2];         // координаты узлов

      int NodesCount;      //количество узлов

      int FECount;       //количество КЭ

    \*/

    if (FECount != NULL and NodesCount != NULL) {

        if (indexFE < FECount) {

            alpha[0] = (get<0>(GetCordIK(2, indexFE)) - get<0>(GetCordIK(0, indexFE))) \* (get<1>(GetCordIK(6, indexFE)) - get<1>(GetCordIK(0, indexFE))) -

                (get<1>(GetCordIK(2, indexFE)) - get<1>(GetCordIK(0, indexFE))) \* (get<0>(GetCordIK(6, indexFE)) - get<0>(GetCordIK(0, indexFE)));

            alpha[1] = (get<0>(GetCordIK(2, indexFE)) - get<0>(GetCordIK(0, indexFE))) \* (get<1>(GetCordIK(8, indexFE)) - get<1>(GetCordIK(6, indexFE))) -

                (get<1>(GetCordIK(2, indexFE)) - get<1>(GetCordIK(0, indexFE))) \* (get<0>(GetCordIK(8, indexFE)) - get<0>(GetCordIK(6, indexFE)));

            alpha[2] = (get<0>(GetCordIK(8, indexFE)) - get<0>(GetCordIK(2, indexFE))) \* (get<1>(GetCordIK(6, indexFE)) - get<1>(GetCordIK(0, indexFE))) -

                (get<1>(GetCordIK(8, indexFE)) - get<1>(GetCordIK(2, indexFE))) \* (get<0>(GetCordIK(6, indexFE)) - get<0>(GetCordIK(0, indexFE)));

        }

        else throw "Ошибка: указан индекс за диапазоном";

    }

    else throw "Ошибка: вычисление вектора альфа невозможно без инициализации области";

}

void CalcVecMyu(unsigned int indexFE) {

    /\*

    int\* numfe[4],       // номера узлов конечных элементов

       \* numsubarea[2];  // номера подобластей конечных элементов

      double\* rz[2];         // координаты узлов

      int NodesCount;      //количество узлов

      int FECount;       //количество КЭ

    \*/

    if (FECount != NULL and NodesCount != NULL) {

        if (indexFE < FECount) {

            Myu[0] = get<0>(GetCordIK(6, indexFE)) - get<0>(GetCordIK(0, indexFE));

            Myu[1] = get<0>(GetCordIK(2, indexFE)) - get<0>(GetCordIK(0, indexFE));

            Myu[2] = get<1>(GetCordIK(6, indexFE)) - get<1>(GetCordIK(0, indexFE));

            Myu[3] = get<1>(GetCordIK(2, indexFE)) - get<1>(GetCordIK(0, indexFE));

            Myu[4] = get<0>(GetCordIK(0, indexFE)) - get<0>(GetCordIK(2, indexFE)) - get<0>(GetCordIK(6, indexFE)) + get<0>(GetCordIK(8, indexFE));

            Myu[5] = get<1>(GetCordIK(0, indexFE)) - get<1>(GetCordIK(2, indexFE)) - get<1>(GetCordIK(6, indexFE)) + get<1>(GetCordIK(8, indexFE));

        }

        else throw "Ошибка: указан индекс за диапазоном";

    }

    else throw "Ошибка: вычисление вектора мю невозможно без инициализации области";

}

#pragma endregion

#pragma region Реализация Гаусса

double Gauss2rod(function<double(double, double, tuple<int, int, int>, double)> Ifun, tuple<int, int, int> FuncIndex, double time) {

    // Расчёт констант

    double Tau[3]; Tau[0] = 8.0 / 9; Tau[1] = Tau[2] = 5.0 / 9;

    double t[3] = {0, 0.77459666924148337, -0.77459666924148337 };

    double result = 0.0;

    //Расчёт метода Гаусса

    for (int i = 0; i < 3; i++) {

        double PsyI = (1 + t[i]) / 2;

        for (int j = 0; j < 3; j++) {

            double EtaJ = (1 + t[j]) / 2;

            result += Tau[i] \* Tau[j] \* Ifun(PsyI, EtaJ, FuncIndex, time);

        }

    }

    return result/double(4);

}

double Gauss2rod(function<double(double, double, tuple<int, int, int>)> Ifun, tuple<int, int, int> FuncIndex) {

    // Расчёт констант

    double Tau[3]; Tau[0] = 8.0 / 9; Tau[1] = Tau[2] = 5.0 / 9;

    double t[3] = { 0, 0.77459666924148337, -0.77459666924148337 };

    double result = 0.0;

    //Расчёт метода Гаусса

    for (int i = 0; i < 3; i++) {

        double PsyI = (1 + t[i]) / 2;

        for (int j = 0; j < 3; j++) {

            double EtaJ = (1 + t[j]) / 2;

            result += Tau[i] \* Tau[j] \* Ifun(PsyI, EtaJ, FuncIndex);

        }

    }

    return result / double(4);

}

#pragma endregion

#pragma region Построение глобальной матрицы и вектора

void GeneratePortrait() { // генерация портрета матрицы

    vector<set<int>> list(NodesCount);

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

        for (int j = 8; j >= 0; j--) {

            for (int m = j - 1; m >= 0; m--) {

                list[numfe[j][i]].insert(numfe[m][i]);

            }

        }

    }

    ig.resize(NodesCount + 1, 0);

    for (int i = 2; i <= NodesCount; i++) {

        ig[i] = ig[i - 1] + list[i - 1].size();

    }

    jg.resize(ig[NodesCount]);

    auto iter = jg.begin();

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

        copy(list[i].begin(), list[i].end(), iter);

        iter += list[i].size();

    }

    ggl.resize(ig[NodesCount]);

    di.resize(NodesCount);

    b.resize(NodesCount);

    L\_sq.resize(ig[NodesCount]);

    di\_sq.resize(NodesCount);

}

void add\_local\_to\_global(int IndexFE, int debag = false)//занесение локальной матрицы и локального вектора правой части в  глобальные

{

    for (int i = 0; i < 9; i++)

    {

        int ind = numfe[i][IndexFE];

        di[ind] += Gk[i];

        b[ind] += bk[i];

    }

    for (int i = 0, p = 9; i < 9; i++)

    {

        int d = numfe[i][IndexFE];//глобальный номер узла ii-го КЭ с локальным номером i

        int ibeg = ig[d];

        for (int j = 0; j < i; j++, p++)

        {

            int iend = ig[d + 1] - 1;

            while (jg[ibeg] != numfe[j][IndexFE])//дихотомия

            {

                int ind = (ibeg + iend) / 2 + 1;

                if (jg[ind] <= numfe[j][IndexFE]) ibeg = ind;

                else iend = (iend == ind) ? ind - 1 : ind;

            }

            ggl[ibeg] += Gk[p];

            ibeg++;

        }

    }

}

void LocalCalcMKXi(int IndexFE) {

    if (FECount == NULL || NodesCount == NULL)

        throw "Ошибка: Нельзя посчитать локальную матрицу не зная область";

    //Создаём функцию по которой интегрируем

    function<double(double, double, tuple<int, int, int> Index)> IntegratedFunction = [](double Psy, double Eta, tuple<int, int, int> IndexFunc) { return RPsyEta(Psy, Eta, get<2>(IndexFunc)) \* xi\_d(RPsyEta(Psy, Eta, get<2>(IndexFunc)), ZPsyEta(Psy, Eta, get<2>(IndexFunc))) \* W[get<0>(IndexFunc)](Psy, Eta) \* W[get<1>(IndexFunc)](Psy, Eta) \* (alpha[0] + alpha[1] \* Psy + alpha[2] \* Eta); };

    //Вычисляем M для первого конечного элемента

    for (int i = 0; i < 9; i++)

        MkXi[i] = sign(alpha[0]) \* Gauss2rod(IntegratedFunction, make\_tuple(i, i, IndexFE));

    for (int i = 1, p = 9; i < 9; i++)

        for (int j = 0; j < i; j++, p++)

            MkXi[p] = sign(alpha[0]) \* Gauss2rod(IntegratedFunction, make\_tuple(i, j, IndexFE));

}

void LocalCalcMKSig(int IndexFE) {

    if (FECount == NULL || NodesCount == NULL)

        throw "Ошибка: Нельзя посчитать локальную матрицу не зная область";

    //Создаём функцию по которой интегрируем

    function<double(double, double, tuple<int, int, int> Index)> IntegratedFunction = [](double Psy, double Eta, tuple<int, int, int> IndexFunc) { return RPsyEta(Psy, Eta, get<2>(IndexFunc)) \* sigma(RPsyEta(Psy, Eta, get<2>(IndexFunc)), ZPsyEta(Psy, Eta, get<2>(IndexFunc))) \* W[get<0>(IndexFunc)](Psy, Eta) \* W[get<1>(IndexFunc)](Psy, Eta) \* (alpha[0] + alpha[1] \* Psy + alpha[2] \* Eta); };

    //Вычисляем M для первого конечного элемента

    for (int i = 0; i < 9; i++)

        MkSig[i] = sign(alpha[0]) \* Gauss2rod(IntegratedFunction, make\_tuple(i, i, IndexFE));

    for (int i = 1, p = 9; i < 9; i++)

        for (int j = 0; j < i; j++, p++)

            MkSig[p] = sign(alpha[0]) \* Gauss2rod(IntegratedFunction, make\_tuple(i, j, IndexFE));

}

void LocalCalcGK(int IndexFE) {

    if (FECount == NULL || NodesCount == NULL)

        throw "Ошибка: Нельзя посчитать локальную матрицу не зная область";

    //Создаём функцию по которой интегрируем

    function<double(double, double, tuple<int, int, int>)> IntegretedFunc = [](double Psy, double Eta, tuple<int, int, int> IndexFunc) { return lambda(ZPsyEta(Psy, Eta, get<2>(IndexFunc))) \* RPsyEta(Psy, Eta, get<2>(IndexFunc)) \* ((D[0][get<0>(IndexFunc)](Psy, Eta) \* (Myu[5] \* Psy + Myu[2]) - D[1][get<0>(IndexFunc)](Psy, Eta) \* (Myu[5] \* Eta + Myu[3])) \* (D[0][get<1>(IndexFunc)](Psy, Eta) \* (Myu[5] \* Psy + Myu[2]) - D[1][get<1>(IndexFunc)](Psy, Eta) \* (Myu[5] \* Eta + Myu[3])) + (D[1][get<0>(IndexFunc)](Psy, Eta) \* (Myu[4] \* Eta + Myu[1]) - D[0][get<0>(IndexFunc)](Psy, Eta) \* (Myu[4] \* Psy + Myu[0])) \* (D[1][get<1>(IndexFunc)](Psy, Eta) \* (Myu[4] \* Eta + Myu[1]) - D[0][get<1>(IndexFunc)](Psy, Eta) \* (Myu[4] \* Psy + Myu[0]))) / (alpha[0] + alpha[1]\*Psy + alpha[2]\*Eta); };

    //Выделяем память

    Gk = new double[45];

    //Вычисляем G для IndexFE конечного элемента

    for (int i = 0; i < 9; i++)

        Gk[i] = sign(alpha[0]) \* Gauss2rod(IntegretedFunc, make\_tuple(i, i, IndexFE));

    for (int i = 1, p = 9; i < 9; i++)

        for (int j = 0; j < i; j++, p++)

            Gk[p] = sign(alpha[0]) \* Gauss2rod(IntegretedFunc, make\_tuple(i, j, IndexFE));

}

double RPsyEta(double Psy, double Eta, int IndexFE) {

    return (1 - Psy) \* (1 - Eta) \* get<0>(GetCordIK(0, IndexFE)) + Psy \* (1 - Eta) \* get<0>(GetCordIK(2, IndexFE)) +

        (1 - Psy) \* Eta \* get<0>(GetCordIK(6, IndexFE)) + Psy \* Eta \* get<0>(GetCordIK(8, IndexFE));

}

double ZPsyEta(double Psy, double Eta, int IndexFE) {

    return (1 - Psy) \* (1 - Eta) \* get<1>(GetCordIK(0, IndexFE)) + Psy \* (1 - Eta) \* get<1>(GetCordIK(2, IndexFE)) +

        (1 - Psy) \* Eta \* get<1>(GetCordIK(6, IndexFE)) + Psy \* Eta \* get<1>(GetCordIK(8, IndexFE));

}

void SumVector(double\*& FirstVec, double\* SecondVec, int n) {

    for (int i = 0; i < n; i++) {

        FirstVec[i] += SecondVec[i];

    }

}

void SubVector(double\*& FirstVec, double\* SecondVec, int n) {

    for (int i = 0; i < n; i++) {

        FirstVec[i] -= SecondVec[i];

    }

}

void EqVector(double\*& FirstVec, double\* SecondVec, int n) {

    for (int i = 0; i < n; i++) {

        FirstVec[i] = SecondVec[i];

    }

}

void MultMKVector(double\*& FirstVec, double\* SecVector, double \* Mk) {

    for (int i = 0; i < 9; i++) {

        FirstVec[i] = Mk[i] \* SecVector[i];

    }

    for (int i = 0, p = 9; i < 9; i++)

        for (int j = 0; j < i; j++, p++) {

            FirstVec[i] += Mk[p] \* SecVector[j];

            FirstVec[j] += Mk[p] \* SecVector[i];

        }

}

void MultVectorConstant(double\*& FirstVec, double Constant, int n) {

    for (int i = 0; i < n; i++) FirstVec[i] \*= Constant;

}

void LocalCalcVectorFEK(int IndexFE, double time) { // реализовано

    if (FECount == NULL || NodesCount == NULL)

        throw "Ошибка: Нельзя посчитать вектор правой части без заданной области";

    //Создаём функцию по которой интегрируем

    function<double(double, double, tuple<int, int, int>, double)> IntegretedFunc = [] (double Psy, double Eta, tuple<int, int, int> IndexFunc, double time) { return RightFunction(RPsyEta(Psy, Eta, get<2>(IndexFunc)), ZPsyEta(Psy, Eta, get<2>(IndexFunc)), time) \* RPsyEta(Psy, Eta, get<2>(IndexFunc)) \* W[get<1>(IndexFunc)](Psy, Eta) \* W[get<0>(IndexFunc)](Psy, Eta) \* (alpha[0] + alpha[1] \* Psy + alpha[2] \* Eta); };

    for (int i = 0; i < 9; i++) bk[i] = 0.;

    CalcVecAlpha(IndexFE);

    for (int i = 0; i < 9; i++)

        for (int j = 0; j < 9; j++)

            bk[i] += sign(alpha[0]) \* Gauss2rod(IntegretedFunc, make\_tuple(i, j, IndexFE), time);

}

void InitVectors() {

    MkXi = new double[45];

    MkSig = new double[45];

    Gk = new double[45];

    bk = new double[9];

}

static void Calc\_deltat\_3SCHEME(double\*& Vector, int j) {

    if (j < 2) {

        throw "j < 2";

    }

    if (num\_t < 2 || j >= num\_t) {

        throw "Wrong j";

    }

    Vector[0] = tj[j] - tj[j - 1];

    Vector[1] = tj[j - 1] - tj[j - 2];

    Vector[2] = tj[j] - tj[j - 2];

}

void GetQpj(double \*& FirstVec, int IndexFe, int mj) {

    int index;

    for (int i = 0; i < 9; i++) {

        index = numfe[i][IndexFe];

        FirstVec[i] = qj[mj][index];

    }

}

void GetJLayer\_3SCHEME(bool debag, int j) {

    if (j < 2) {

        throw "j < 2";

    }

    InitVectors();

    double\* TempM = new double[45];

    double\* deltat = new double[3];

    Calc\_deltat\_3SCHEME(deltat, j);

    double con1 = 2 / (deltat[2] \* deltat[0]);

    double con2 = 1 / deltat[0] + 1 / deltat[2];

    double con3 = 2 / (deltat[1] \* deltat[0]);

    double con4 = 2 / (deltat[2] \* deltat[1]);

    double con5 = deltat[2] / (deltat[1] \* deltat[0]);

    double con6 = deltat[0] / (deltat[1] \* deltat[2]);

    double\* qp1 = new double[9];

    double\* qp2 = new double[9];

    for (int p = 0; p < FECount; p++)//проходим по всем КЭ

    {

        CalcVecAlpha(p);

        CalcVecMyu(p);

        LocalCalcGK(p);

        LocalCalcMKXi(p);

        LocalCalcMKSig(p);

        if (debag) {

            printf("Gk Mk до суммирования\nGk: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", Gk[i]);

            printf("\nMkXi: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", MkXi[i]);

            printf("\nMkSig: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", MkSig[i]);

            printf("\n");

        }

        EqVector(TempM, MkXi, 45);

        MultVectorConstant(TempM, con1, 45);

        SumVector(Gk, TempM, 45);

        EqVector(TempM, MkSig, 45);

        MultVectorConstant(TempM, con2, 45);

        SumVector(Gk, TempM, 45);

        if (debag) {

            printf("Gk после суммирования\nGk: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", Gk[i]);

            printf("\n");

        }

        LocalCalcVectorFEK(p, tj[j]);

        GetQpj(qp1, p, 1);

        MultVectorConstant(qp1, con3, 9);

        GetQpj(qp2, p, 0);

        MultVectorConstant(qp2, con4, 9);

        SubVector(qp1, qp2, 9);

        MultMKVector(qp2, qp1, MkXi);

        SumVector(bk, qp2, 9);

        GetQpj(qp1, p, 1);

        MultVectorConstant(qp1, con5, 9);

        GetQpj(qp2, p, 0);

        MultVectorConstant(qp2, con6, 9);

        SubVector(qp1, qp2, 9);

        MultMKVector(qp2, qp1, MkSig);

        SumVector(bk, qp2, 9);

        if (debag) {

            printf("bk: ");

            for (int i = 0; i < 9; i++)

                printf("%f ", bk[i]);

            printf("\n");

        }

        add\_local\_to\_global(p);

    }

    delete[] MkXi; delete[] MkSig; delete[] Gk; delete[] bk;

}

void Calc\_deltat\_4SCHEME(double\*& Vector, int j) {

    if (j < 3) {

        throw "j < 3";

    }

    if (num\_t < 3 || j >= num\_t) {

        throw "Wrong j";

    }

    Vector[0] = tj[j - 3] - tj[j - 2];

    Vector[1] = tj[j - 3] - tj[j - 1];

    Vector[2] = tj[j - 3] - tj[j];

    Vector[3] = tj[j - 2] - tj[j - 1];

    Vector[4] = tj[j - 2] - tj[j];

    Vector[5] = tj[j - 1] - tj[j];

}

void GetJLayer\_4SCHEME(bool debag, int j) {

    if (j < 3) {

        throw "j < 3";

    }

    InitVectors();

    double\* TempM = new double[45];

    double\* deltat = new double[6];

    Calc\_deltat\_4SCHEME(deltat, j);

    double con1 = deltat[4] \* deltat[5]/(deltat[0] \* deltat[1] \* deltat[2]);

    double con2 = -deltat[2] \* deltat[5] / (deltat[0] \* deltat[3] \* deltat[4]);

    double con3 = deltat[2]\*deltat[4]/(deltat[1]\*deltat[3]\*deltat[5]);

    double con4 = -(1 / deltat[2] + 1 / deltat[4] + 1 / deltat[5]);

    double con5 = -2 \* (deltat[4] + deltat[5]) / (deltat[0] \* deltat[1] \* deltat[2]);

    double con6 = 2 \* (deltat[2] + deltat[5]) / (deltat[0] \* deltat[3] \* deltat[4]);

    double con7 = -2 \* (deltat[2] + deltat[4]) / (deltat[1] \* deltat[3] \* deltat[5]);

    double con8 = 2 \* (deltat[2] + deltat[5] + deltat[4]) / (deltat[2] \* deltat[4] \* deltat[5]);

    double\* qp1 = new double[9];

    double\* qp2 = new double[9];

    for (int p = 0; p < FECount; p++)//проходим по всем КЭ

    {

        CalcVecAlpha(p);

        CalcVecMyu(p);

        LocalCalcGK(p);

        LocalCalcMKXi(p);

        LocalCalcMKSig(p);

        if (debag) {

            printf("Gk Mk до суммирования\nGk: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", Gk[i]);

            printf("\nMkXi: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", MkXi[i]);

            printf("\nMkSig: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", MkSig[i]);

            printf("\n");

        }

        EqVector(TempM, MkXi, 45);

        MultVectorConstant(TempM, con8, 45);

        SumVector(Gk, TempM, 45);

        EqVector(TempM, MkSig, 45);

        MultVectorConstant(TempM, con4, 45);

        SumVector(Gk, TempM, 45);

        if (debag) {

            printf("Gk после суммирования\nGk: ");

            for (int i = 0; i < 45; i++)

                printf("%f ", Gk[i]);

            printf("\n");

        }

        LocalCalcVectorFEK(p, tj[j]);

        GetQpj(qp1, p, 1);

        MultVectorConstant(qp1, con6, 9);

        GetQpj(qp2, p, 0);

        MultVectorConstant(qp2, con5, 9);

        SumVector(qp1, qp2, 9);

        GetQpj(qp2, p, 2);

        MultVectorConstant(qp2, con7, 9);

        SumVector(qp1, qp2, 9);

        MultMKVector(qp2, qp1, MkXi);

        SubVector(bk, qp2, 9);

        GetQpj(qp1, p, 1);

        MultVectorConstant(qp1, con2, 9);

        GetQpj(qp2, p, 0);

        MultVectorConstant(qp2, con1, 9);

        SumVector(qp1, qp2, 9);

        GetQpj(qp2, p, 2);

        MultVectorConstant(qp2, con3, 9);

        SumVector(qp1, qp2, 9);

        MultMKVector(qp2, qp1, MkSig);

        SubVector(bk, qp2, 9);

        if (debag) {

            printf("bk: ");

            for (int i = 0; i < 9; i++)

                printf("%f ", bk[i]);

            printf("\n");

        }

        add\_local\_to\_global(p);

    }

    delete[] MkXi; delete[] MkSig; delete[] Gk; delete[] bk;

}

#pragma endregion

#pragma region Реалиация MSG function

double ScalarMult(vector<double>& first\_vec, vector<double>& second\_vec, int n) {

    double res = 0;

    for (int i = 0; i < n; i++) {

        res += first\_vec[i] \* second\_vec[i];

    }

    return res;

}

void MultMartVec(vector<double>& input\_vec, vector<double>& res\_vec, int n) {

    for (int i = 0; i < n; i++) {

        res\_vec[i] = di[i] \* input\_vec[i];

        for (int k = ig[i]; k < ig[i + 1]; k++) {

            int j = jg[k];

            res\_vec[i] += ggl[k] \* input\_vec[j];

            res\_vec[j] += ggl[k] \* input\_vec[i];

        }

    }

}

void LU\_sq(int n) {

    for (int i = 0; i < n; i++) {

        double s\_d = 0;

        for (int k = ig[i]; k < ig[i + 1]; k++) {

            double s\_l = 0;

            int j = jg[k];

            int j0 = ig[j];

            int j1 = ig[j + 1];

            int ki = ig[i];

            int kj = j0;

            for (; ki < k && kj < j1;) {

                int jl = jg[ki];

                int ju = jg[kj];

                if (jl == ju) {

                    s\_l += L\_sq[kj] \* L\_sq[ki];

                    ki++, kj++;

                }

                else if (jl < ju) ki++;

                else kj++;

            }

            L\_sq[k] = (ggl[k] - s\_l) / di\_sq[j];

            s\_d += L\_sq[k] \* L\_sq[k];

        }

        di\_sq[i] = sqrt(di[i] - s\_d);

    }

}

void Ly\_f(vector<double>& f, int n) {

    for (int i = 0; i < n; i++) {

        double y = 0;

        for (int k = ig[i]; k < ig[i + 1]; k++) {

            int j = jg[k];

            y += L\_sq[k] \* f[j];

        }

        f[i] = (f[i] - y) / di\_sq[i];

    }

}

void Ly\_f\_transp(vector<double>& f, int n) {

    for (int i = n - 1; i >= 0; i--) {

        f[i] /= di\_sq[i];

        for (int k = ig[i]; k < ig[i + 1]; k++) {

            int j = jg[k];

            f[j] -= L\_sq[k] \* f[i];

        }

    }

}

void PrMinusR(vector<double>& r, int n) {

    for (int i = 0; i < n; i++) {

        r[i] = b[i] - r[i];

    }

}

void FirstEqualSecond(vector<double>& z, vector<double>& r, int n) {

    for (int i = 0; i < n; i++) {

        z[i] = r[i];

    }

}

double CulcResid(vector<double>& r, double norma\_f, int n) {

    return sqrt(ScalarMult(r, r, n)) / norma\_f;

}

void CulcX(vector<double>& x, vector<double>& z, double alpha, int n) {

    for (int i = 0; i < n; i++) {

        x[i] += alpha \* z[i];

    }

}

void CulcR(vector<double>& r, vector<double>& Az, double alpha, int n) {

    for (int i = 0; i < n; i++) {

        r[i] -= alpha \* Az[i];

    }

}

void CulcZ(vector<double>& Mult, vector<double>& z, double beta, int n) {

    for (int i = 0; i < n; i++) {

        z[i] = Mult[i] + beta \* z[i];

    }

}

double CulcAlpha(vector<double>& Az, vector<double>& z, vector<double>& Mult, vector<double>& r, int n) {

    return ScalarMult(Mult, r, n) / ScalarMult(Az, z, n);

}

double CulcBeta(double numerator, double denominator) {

    return numerator / denominator;

}

void CulcMult(vector<double>& Mult, int n) {

    Ly\_f(Mult, n);

    Ly\_f\_transp(Mult, n);

}

void LU\_sq\_MSG(vector<double>& x, vector<double>& r, vector<double>& z, vector<double>& Az, vector<double>& Mult, int n, double eps, int max\_iter) {

    double resid, norma\_f;

    int counter = 1;

    LU\_sq(n);

    MultMartVec(x, r, n);

    PrMinusR(r, n);

    norma\_f = sqrt(ScalarMult(b, b, n));

    resid = CulcResid(r, norma\_f, n);

    cout << "Start resid: " << resid << endl;

    FirstEqualSecond(Mult, r, n);

    CulcMult(Mult, n);

    FirstEqualSecond(z, Mult, n);

    for (; counter < max\_iter && resid > eps; counter++) {

        double alpha, beta, numerator, denominator;

        resid = 0;

        MultMartVec(z, Az, n);

        alpha = CulcAlpha(Az, z, Mult, r, n);

        denominator = ScalarMult(Mult, r, n);

        CulcX(x, z, alpha, n);

        CulcR(r, Az, alpha, n);

        FirstEqualSecond(Mult, r, n);

        CulcMult(Mult, n);

        numerator = ScalarMult(Mult, r, n);

        beta = CulcBeta(numerator, denominator);

        CulcZ(Mult, z, beta, n);

        resid = sqrt(ScalarMult(r, r, n)) / norma\_f;

        cout << setprecision(15) << resid << " " << counter << endl;

    }

    MultMartVec(x, r, n);

    PrMinusR(r, n);

    resid = CulcResid(r, norma\_f, n);

    cout << "End resid: " << resid << endl;

}

void ClearVectorsLU() {

    ggl.resize(ig[NodesCount]);

    di.resize(NodesCount);

    b.resize(NodesCount);

    L\_sq.resize(ig[NodesCount]);

    di\_sq.resize(NodesCount);

    for (int i = 0; i < ig[NodesCount]; i++) {

        ggl[i] = 0.0;

        L\_sq[i] = 0.0;

    }

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

        di[i] = 0.0;

        b[i] = 0.0;

        di\_sq[i] = 0.0;

    }

}

#pragma endregion

#pragma endregion

#pragma region Тестирование

void Test(vector<double> q, int layer) {

    vector<double> q\_u(NodesCount, 0);

    for (int i = 0, p = 0; i < NodesCountR; i++) {

        for (int j = 0; j < NodesCountR; j++, p++) {

            q\_u[p] = ug(0, rz[0][p], rz[1][p], tj[layer]);

        }

    }

    double norm\_vec\_err = 0, norm\_vec\_q\_u = 0; // норма вектора погрешности и q\_u

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

        norm\_vec\_err += (q[i] - q\_u[i]) \* (q[i] - q\_u[i]);

        norm\_vec\_q\_u += (q\_u[i]) \* (q\_u[i]);

    }

    cout << endl;

    cout << "Относительная норма вектора погрешности полученного решения на слое " << layer + 1 << " при t = " << tj[layer] << " равна: " << endl;

    cout << sqrt(norm\_vec\_err) / sqrt(norm\_vec\_q\_u) << endl;

    cout.precision(15);

    /\*

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

        cout << q[i] << " " << q\_u[i] << endl;

    \*/

}

#pragma endregion

int main()

{

    setlocale(LC\_ALL, "Russian");

    EnterBoundCondit();

    InitBaseFunction();

    InitDРerivativeBaseFunction();

    LoadNet();

    LoadTNet();

    InitQJ();

    InitFirstQ();

    BoundCondit();

    BuildBoundMatrices();

    GeneratePortrait();

    GetJLayer\_3SCHEME(false, 2);

    ConsiderBoundCondit(tj[2]);

    int max\_iter = 1000;

    double eps = 1e-15;

    vector<double> r(NodesCount);

    vector<double> z(NodesCount);

    vector<double> Mult(NodesCount);

    vector<double> Az(NodesCount);

    vector<double> q(NodesCount, 0);

    LU\_sq\_MSG(q, r, z, Az, Mult, NodesCount, eps, max\_iter);

    qj[2] = q;

    Test(q, 2);

    for (int j = 3; j < num\_t; j++) {

        ClearVectorsLU();

        GetJLayer\_4SCHEME(false, j);

        ConsiderBoundCondit(tj[j]);

        vector<double> r(NodesCount);

        vector<double> z(NodesCount);

        vector<double> Mult(NodesCount);

        vector<double> Az(NodesCount);

        vector<double> q(NodesCount, 0);

        LU\_sq\_MSG(q, r, z, Az, Mult, NodesCount, eps, max\_iter);

        qj[0] = qj[1]; qj[1] = qj[2]; qj[2] = q;

        Test(q, j);

    }

    return 0;

}