|  |  |  |  |
| --- | --- | --- | --- |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
| Кафедра прикладной математики | | | |
|  | | | |
|  | | | |
|  | | | |
| Курсовой проект по курсу | | | |
| **«Численные методы»** | | | |
|  | | | |
| Группа ПМ-13  Студент Голубь Андрей Дмитриевич |  | |
|  |  | |
|  |  | |
|  | |
|  | |
|  |  | |
|  |  | |
|  |  | |
| |  | | --- | | Новосибирск |   2023 | | | |

1. **Постановка задачи**

МКЭ для двумерной краевой задачи для уравнения ротор-роторного типа в цилиндрической системе координат. Базисные функции биквадратичные на четырехугольниках. Краевые условия всех типов. Коэффициент диффузии λ зависит только от *z*. Матрицу СЛАУ генерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

* 1. **Решаемое уравнение**

Уравнение ротор-роторного типа для осесимметричных стационарных полей выглядит следующим образом:

φφ ,

где μ – магнитная проницаемость среды, - вектор потенциал с единственной ненулевой компонентой , являющейся функцией только двух пространственных переменных *r* и *z*: , φ – φ-компонента вектора плотности токов.

Так как λ зависит только от *z*, то осесимметричное стационарное магнитное поле в цилиндрических координатах может быть описано эллиптическим уравнением с коэффициентом диффузии , коэффициентом и правой частью φ

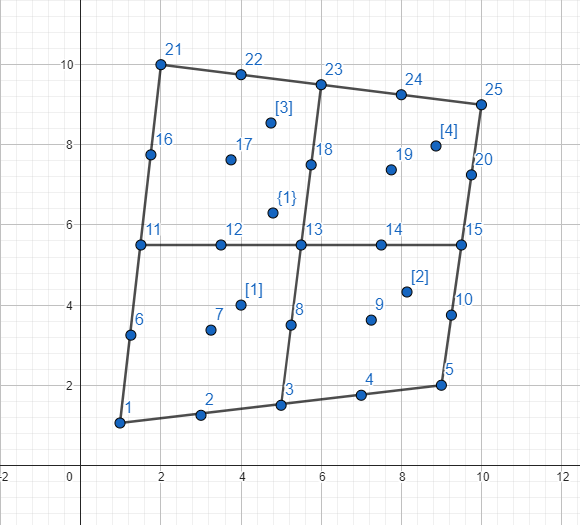
Дифференциальное уравнение для эллиптической осисимметричной краевой задачи может быть записано в виде

* 1. **Краевые условия**

Уравнение заданно в некоторой области Ω с границей , и краевыми условиями

где - значение исходной функции на границе , - значение на производной по направлению нормали к поверхности .

* 1. **Расчетная область**

****

На расчетной области числами 1, 2, 3, …, 25 обозначены номера узлов, числами в квадратных скобках [1], [2], [3], [4] обозначены номера конечных элементов, а число {1} в фигурных скобках – номер подобласти.

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

Пусть u – решение исходной краевой задачи.

Для всякой функции интегрируемой с квадратом, т.е. , дифференциальное уравнение имеет единственное решение , где – пространство функций, имеющие суммируемые с квадратом вторые производные и удовлетворяющие краевым условиям на границе области . Таким образом, краевую задачу можно представить в виде

где оператор есть взаимно однозначное отображение в , и вид оператора можно получить из сравнения уравнений предыдущего и исходного:

Пусть – финитные базисные функции пространства , являющегося подпространством . Тогда приближенное решение будем искать как проекцию на конечномерное подпространство гильбертова пространства , натянутого на систему базисных функций. Значит

где - приближенное решение уравнения в виде .

Скалярное произведение в выглядит следующим образом:

.

Преобразуем уравнение с использованием формулы Грина:

,

где , как и ранее, граница Ω , а значит

.

Учтем заданные краевые условия. Поскольку значит и = 0 , тогда

.

* 1. **Конечноэлементная дискретизация и переход к локальным матрицам**

Исходя из того, что , уравнение принимает вид:

Поскольку исходная задача рассматривается в цилиндрической системе координат, то , где элемент области равен , а элемент границ равен . Следовательно

Построим на четырехугольном конечном элементе Ωk локальные базисные функции с помощью биквадратичных функций, заданных на шаблонном элементе ΩE, являющимся единичным квадратом:

Отобразим ΩE в четырехугольник Ωk с вершинами с помощью следующих соотношений:

Введем на ΩE биквадратичные базисные функции

где

Тогда базисные функции на четырехугольнике Ωk можно определить с помощью соотношений

где и - функции, определяемые из вышестоящих соотношений и .

* 1. **Аналитические выражения для вычисления локальных матриц и схемы численного интегрирования**

Получим формулы для вычисления локальной матрицы массы.

Так как функция на элементе Ωk – это образ функции , определенной на шаблонном элементе ΩE , коэффициент диффузии зависит только от *z* и заменяется на свое точное значение, также как и , то

где

,

,

Вычислим якобиан преобразования координат:

где коэффициенты , и имеют следующий вид:

При этом следует учесть, что якобиан в силу взаимной однозначности отображения на ΩE нигде не обращается в ноль и поэтому не меняет знак. Тогда модуль якобиана может быть вычислен как

Поскольку при .

При вынесении общего множителя, локальная матрица имеет вид:

где элементы матрицы получаются путем численного интегрирования, используя метод Гаусса. Квадратурные формулы при интегрировании по шаблонному элементу ΩE (с четырьмя узлами по каждой из координат) имеют следующий вид:

где коэффициенты и определяются соотношениями:

Рассмотрим, как можно вычислить компоненты локальной матрицы жесткости:

Вычислим производные и . Используя правила дифференцирования сложной функции, получим

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

где

Тогда

Полученные интегралы вычисляются численно методом Гаусса (с четырьмя узлами по каждой из координат).

Локальный вектор правой части , будем вычислять с учетом того, что функция на конечном элементе представлена в виде разложения по базисным функциям , где - число локальных базисных функций конечного элемента . Тогда

При переходе на шаблонный элемент ΩE значения , равны значениям функции на узлах.

Интеграл же имеет вид:

Полученные интегралы вычисляются численно методом Гаусса (с четырьмя узлами по каждой из координат).

Элементы матрицы и вектора правой части конечноэлементной СЛАУ представляют собой сумму объемных и поверхностных интегралов

где – множество индексов узлов, на которых заданы краевые условия первого рода.

Фактически, решая задачу учета краевых условий второго и третьего рода, мы переходим к решению одномерной задачи на ребре. Базисными функциями ребра являются три ненулевые на данном ребре квадратичные базисные функции ,

Для учета краевых условий необходимо вычислить интегралы:

Будем считать, что параметр на постоянен, тогда для ребра , заданного координатами узлов и

где якобиан был представлен в виде линейной комбинации базисных функций.

Значит

где - длина ребра .

Параметр , будем раскладывать по трем квадратичным базисным функциям, определенным на этом ребре:

где – значения функции в узлах ребра.

Аналогично поступаем и при учете вторых краевых условий, раскладывая по базису ребра функцию .

Таким образом, для ребер и

1. **Описание разработанных программ**

На вход программы подается файл **grid\_coordinates.txt**, в котором первое значение – количество разбиений ребра (все ребра разбиваются одинаково), затем парами цилиндрические координаты и соответственно определяющие один узел (всего 4 узла).

* 1. **Структуры данных, используемые для задания расчетной области и конечноэлементной сетки**

Рассмотрим пример сетки, представленной в разделе «Расчетная область».

Информация о четырехугольниках сетки представлена массивом, состоящим из 9 целых чисел, являющихся номерами вершин (не учитываем номер подобласти, так как считаем, что на всем конечном элементе задана всего одна подобласть). Для предложенной сетки массив выглядит следующим образом:

1 2 3 6 7 8 11 12 13 3 4 5 8 9 10 13 14 15

11 12 13 16 17 18 21 22 23 13 14 15 18 19 20 23 24 25

Краевые условия второго и третьего родов хранятся в массиве четверками целых чисел, первое и третье число номера вершин ребра, второе – номер центрального узла, четвертое – номер формулы, по которой должно вычисляться значение соответствующих параметров. Для предложенной сетки массив выглядит следующим образом (если на нижнем ребре задано краевое условие второго рода, а на правом и левом ребрах заданно третье краевое условие):

* второй род:

1 2 3 1 3 4 5 1

* третий род:

5 10 15 1 15 20 25 1 1 6 11 2 11 16 21 2

Информация о краевых условиях первого рода может быть задана массивом пар целых чисел, в которых первое число – номер узла, а второе – номер формулы, по которой вычисляется значение заданного параметра. Для предложенной сетки массив выглядит следующим образом (если заданно на верхнем ребре):

21 1 22 1 23 1 24 1 25 1

Матрица СЛАУ, получаемая в ходе решения, хранится в разреженном строчно-столбцовом формате. Сама СЛАУ решается с использованием метода сопряженных градиентов с неполной факторизацией

Пары координат узлов сетки хранятся с помощью структуры std::pair, позволяющей обрабатывать два объекта как один, базисные функции и их производные используют шаблон std::function, позволяющий вызываться функцией лямбда-выражению, и построение портрета матрицы реализовано с помощью списка смежности для узлов, используя класс std::set, контейнера с уникальными автоматически упорядоченными элементами. Все остальные массивы реализованы с помощью контейнера std::vector.

* 1. **Структура основных модулей программы, в том числе генерация портрета СЛАУ, вычисление локальных матриц, генерация глобальных матриц, решение СЛАУ**

Программа представленна в виде модулей с названием соответствующим содержимому:

* Объявление глобальных переменных
* Функции краевых условий и задание этих функций
* Замена переменных
* Задание функции, лямбды и якобиана преобразования
* Базисные функции
* Гаусс
* Генерация сетки
* Создание массива конечных элементов
* Генерация портрета и подпр-мы добавления в глоб. матрицу
* Работа с краевыми условиями
* Работа с локальными матрицами и векторами
* Построение глобальной матрицы и вектора
* МСГ
* Тестирование

1. **Объявление глобальных переменных**

* num\_split\_edge - число узлов в ребре;
* num\_nodes – число узлов;
* num\_split – количество разбиений;
* beta – значение параметра , заданное константой;
* coef\_r, coef\_z – коэффициент разрядки по и соответственно;
* ia, ja, aal, di, b – вектора, предназначенные для хранения матрицы и вектора;
* choise – массив типов краевых условий, заданных на конкретном ребре в порядке: нижнее ребро, правое ребро, верхнее ребро и левое ребро;
* L\_sq, di\_sq – массивы матрицы и диагонали соответственно после неполной факторизации;
* normal – вектор нормали ребер, соответствующий их порядку при choice;
* basic\_func, deriv\_basic\_func\_xi, deriv\_basic\_func\_eta - массивы базисных функций и их производных по переменным соответственно;
* nodes - узлы сетки, заданные координатами;
* edge – массив ребер с соответствующими краевыми условиями;
* array\_quad - массив четырехугольников;
* A\_1, A\_2, A\_3 – массивы матриц, полученных при аналитическом интегрировании по ребрам.

1. Функции краквых условий и задание этих функций

* EnterBoundCondit - ввод краевых условий;
* u\_g - краевое условие первого рода. Передаем тип функции и координаты, на выходе получаем значение функции в этих координатах;
* theta - краевое условие второго рода. Передаем тип функции и координаты, на выходе получаем значение функции в этих координатах;
* u\_beta - краевое условие третьего рода. Передаем тип функции и координаты, на выходе получаем значение функции в этих координатах.

1. Замена переменных

* r – замена координаты при переходе в единичный квадрат. Передаем тип функции, координаты замены и сами значения . Функция меняется в зависимости от наличия внутренних узлов конечного элемента. На выходе получаем значение функции в заданных координатах;
* z – замена координаты при переходе в единичный квадрат. Передаем координаты замены и сами значения . На выходе получаем значение функции в заданных координатах.

1. Задание функции, лямбды и якобиана преобразования

* f – функция правой части дифференциального уравнения. Передаем координаты, на выходе получаем значение функции в этих координатах;
* lambda – функция . Передаем координаты, на выходе получаем значение функции в этих координатах;
* Jacobian – якобиан преобразования из четырехугольника в единичный квадрат. Передаем координаты и массив доп. значений, вычисленных на конкретном конечном элементе. На выходе получаем значение функции в заданных координатах.

1. Базисные функции

* FiniteFunc – финитная функция, из которой будем строить локальные базисные по разным координатам. Передаем тип финитной функции и значение, на выходе получаем значение необходимой функции в точке;
* DerivFiniteFunc – производная финитной функции. Передаем тип финитной функции и значение, на выходе получаем значение необходимой функции в точке;
* BasicFunc – локальная базисная функция. Передаем координаты, на выходе получаем значение функции в этих координатах;
* DerivBasicFuncXi, DerivBasicFuncEta – производные локальных базисных функций по и соответственно. Передаем координаты, на выходе получаем значение функции в этих координатах.

1. Гаусс

* Gauss4MatrMass, Gauss4MatrStiff, Gauss4VecB – функции вычисления интегралов матрицы массы, жесткости и вектора соответственно с помощью метода Гаусса-4. Передаем локальные индексы и конечного элемента и необходимые значения для вычисления интегралов. На выходе получаем значения данного интеграла.

1. Генерация сетки

* GenEndElGrid – генерация конечноэлементной сетки. На основе входного файла и коэффициентов разрядки получаем необходимую сетку.

1. Создание массива конечных элементов

* ArrayQuadrilaterals – массив четырехугольников.

1. Генерация портрета и подпр-мы добавления в глоб. матрицу

* GeneratePortrait - генерация портрета матрицы;
* AddLocalMatr – подпрограмма добавления матрицы в глобальную;
* AddLocalVec – подпрограмма добавления вектора в глобальный;
* AddLocalMartBound\_3 – подпрограмма добавления матрицы из третьего краевого в глобальную;
* AddLocalVecBound – подпрограмма добавления вектора из третьего или второго краевого в глобальный.

1. Работа с краевыми условиями

* GetStartPoint – функция, задающая начальную точку ребра для создания краевых условий. Передаем -й индекс массива choice;
* GetStepEndEl – функция, задающая шаг для след. конечного элемента. Передаем -й индекс массива choice;
* BoundCondit – генерация краевых условий в виде массива узлов, описанных в пункте **3.1**;
* BuildBoundMatrices – построение «краевых» матриц;
* GetVecNormal – подпрограмма, вычисляющая значения векторов нормали на необходимых ребрах;
* FindInd – функция определяющая наличие передаваемого индекса в векторах первого краевого условия;
* ConsiderBoundConditFirstType - учет краевых условий первого типа на необходимом узле;
* ConsiderBoundConditSecType - учет краевых условий второго типа на необходимом ребре;
* ConsiderBoundConditThirdType - учет краевых условий третьего типа на необходимом ребре;
* ConsiderBoundCondit – подпрограмма, вызывающая учет всех краевых условий.

1. Работа с локальными матрицами и векторами

* LocalMatrMass – вычисление локальной матрицы массы на конкретном конечном элементе;
* LocalMatrStiff - вычисление локальной матрицы жесткости на конкретном конечном элементе;
* localVecB - вычисление локального вектора на конкретном конечном элементе.

1. Построение глобальной матрицы и вектора

* BuildMatrA – построение матрицы СЛАУ без учета краевых условий;
* BuildVecB – построение вектора правой части без учета краевых условий.

1. МСГ

* 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.

1. Тестирование

* Test – подпрограмма, вычисляющая относительную норму вектора погрешности.

В главной функции main последовательно выполняются необходимые подпрограммы, а так же выделяется память под необходимые вектора для решения системы методом сопряженных градиентов и задается точность необходимого решения вместе с максимальным количеством итераций.

1. **Описание тестирования программ**
   1. **Текстовые примеры с пояснением и полученные результаты**

Тесты на порядок аппроксимации

* Тест 1 – степень полиномов равна степени базисных функций

, , ,

|  |  |  |
| --- | --- | --- |
| Номер узла | r | z |
| 0 | 1,000 | 1,000 |
| 1 | 3,000 | 1,250 |
| 2 | 5,000 | 1,500 |
| 3 | 7,000 | 1,750 |
| 4 | 9,000 | 2,000 |
| 5 | 1,250 | 3,250 |
| 6 | 3,250 | 3,375 |
| 7 | 5,250 | 3,500 |
| 8 | 7,250 | 3,625 |
| 9 | 9,250 | 3,750 |
| 10 | 1,500 | 5,500 |
| 11 | 3,500 | 5,500 |
| 12 | 5,500 | 5,500 |
| 13 | 7,500 | 5,500 |
| 14 | 9,500 | 5,500 |
| 15 | 1,750 | 7,750 |
| 16 | 3,750 | 7,625 |
| 17 | 5,750 | 7,500 |
| 18 | 7,750 | 7,375 |
| 19 | 9,750 | 7,250 |
| 20 | 2,000 | 10,000 |
| 21 | 4,000 | 9,750 |
| 22 | 6,000 | 9,500 |
| 23 | 8,000 | 9,250 |
| 24 | 10,000 | 9,000 |

|  |  |
| --- | --- |
| Номер конечного элемента | Узлы |
| 0 | 0 1 2 5 6 7 10 11 12 |
| 1 | 2 3 4 7 8 9 12 13 14 |
| 2 | 10 11 12 15 16 17 20 21 22 |
| 3 | 12 13 14 17 18 19 22 23 24 |

|  |  |  |
| --- | --- | --- |
|  | Узлы | Номер формулы |
| Нижняя грань - | 0 1 2 | 0 |
| 2 3 4 | 0 |
| Правая грань - | 4 9 14 | 0 |
| 14 19 24 | 0 |
| Верхняя грань - | 20 | 0 |
| 21 | 0 |
| 22 | 0 |
| 23 | 0 |
| 24 | 0 |
| Левая грань - | 0 5 10 | 1 |
| 10 15 20 | 1 |

Краевые условия:

* Тест 2 – степень полиномов больше степени базисных функций

, , ,

|  |  |  |
| --- | --- | --- |
| Номер узла | r | z |
| 0 | 1,000 | 1,000 |
| 1 | 3,000 | 1,250 |
| 2 | 5,000 | 1,500 |
| 3 | 7,000 | 1,750 |
| 4 | 9,000 | 2,000 |
| 5 | 1,250 | 3,250 |
| 6 | 3,250 | 3,375 |
| 7 | 5,250 | 3,500 |
| 8 | 7,250 | 3,625 |
| 9 | 9,250 | 3,750 |
| 10 | 1,500 | 5,500 |
| 11 | 3,500 | 5,500 |
| 12 | 5,500 | 5,500 |
| 13 | 7,500 | 5,500 |
| 14 | 9,500 | 5,500 |
| 15 | 1,750 | 7,750 |
| 16 | 3,750 | 7,625 |
| 17 | 5,750 | 7,500 |
| 18 | 7,750 | 7,375 |
| 19 | 9,750 | 7,250 |
| 20 | 2,000 | 10,000 |
| 21 | 4,000 | 9,750 |
| 22 | 6,000 | 9,500 |
| 23 | 8,000 | 9,250 |
| 24 | 10,000 | 9,000 |

|  |  |
| --- | --- |
| Номер конечного элемента | Узлы |
| 0 | 0 1 2 5 6 7 10 11 12 |
| 1 | 2 3 4 7 8 9 12 13 14 |
| 2 | 10 11 12 15 16 17 20 21 22 |
| 3 | 12 13 14 17 18 19 22 23 24 |

|  |  |  |
| --- | --- | --- |
|  | Узлы | Номер формулы |
| Нижняя грань - | 0 1 2 | 0 |
| 2 3 4 | 0 |
| Правая грань - | 4 9 14 | 0 |
| 14 19 24 | 0 |
| Верхняя грань - | 20 | 0 |
| 21 | 0 |
| 22 | 0 |
| 23 | 0 |
| 24 | 0 |
| Левая грань - | 0 5 10 | 1 |
| 10 15 20 | 1 |

Краевые условия:

**Вывод:** результат показал, что порядок аппроксимации равен двум, что совпадает с теоритическим значением, т.к. порядок аппроксимации равен порядку базисных функций.

1. **Проведенные исследования и выводы**

Определение порядка сходимости

, , ,

Краевые условия:

Шаг равен :

|  |  |  |
| --- | --- | --- |
| Номер узла | r | z |
| 0 | 1,000 | 1,000 |
| 1 | 3,000 | 1,500 |
| 2 | 5,000 | 2,000 |
| 3 | 1,500 | 2,500 |
| 4 | 3,000 | 3,000 |
| 5 | 4,500 | 3,500 |
| 6 | 2,000 | 4,000 |
| 7 | 3,000 | 4,500 |
| 8 | 4,000 | 5,000 |

|  |  |
| --- | --- |
| Номер конечного элемента | Узлы |
| 0 | 0 1 2 3 4 5 6 7 8 |

|  |  |  |
| --- | --- | --- |
|  | Узлы | Номер формулы |
| Нижняя грань - | 0 1 2 | 0 |
| Правая грань - | 2 5 8 | 0 |
| Верхняя грань - | 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| Левая грань - | 0 3 6 | 1 |

Шаг равен

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер узла | r | z | Номер узла | r | z |
| 0 | 1,000 | 1,000 | 13 | 3,750 | 3,250 |
| 1 | 2,000 | 1,250 | 14 | 4,500 | 3,500 |
| 2 | 3,000 | 1,500 | 15 | 1,750 | 3,250 |
| 3 | 4,000 | 1,750 | 16 | 2,375 | 3,500 |
| 4 | 5,000 | 2,000 | 17 | 3,000 | 3,750 |
| 5 | 1,250 | 1,750 | 18 | 3,625 | 4,000 |
| 6 | 2,125 | 2,000 | 19 | 4,250 | 4,250 |
| 7 | 3,000 | 2,250 | 20 | 2,000 | 4,000 |
| 8 | 3,875 | 2,500 | 21 | 2,500 | 4,250 |
| 9 | 4,750 | 2,750 | 22 | 3,000 | 4,500 |
| 10 | 1,500 | 2,500 | 23 | 3,500 | 4,750 |
| 11 | 2,250 | 2,750 | 24 | 4,000 | 5,000 |
| 12 | 3,000 | 3,000 |  |  |  |

|  |  |
| --- | --- |
| Номер конечного элемента | Узлы |
| 0 | 0 1 2 5 6 7 10 11 12 |
| 1 | 2 3 4 7 8 9 12 13 14 |
| 2 | 10 11 12 15 16 17 20 21 22 |
| 3 | 12 13 14 17 18 19 22 23 24 |

|  |  |  |
| --- | --- | --- |
|  | Узлы | Номер формулы |
| Нижняя грань - | 0 1 2 | 0 |
| 2 3 4 | 0 |
| Правая грань - | 4 9 14 | 0 |
| 14 19 24 | 0 |
| Верхняя грань - | 20 | 0 |
| 21 | 0 |
| 22 | 0 |
| 23 | 0 |
| 24 | 0 |
| Левая грань - | 0 5 10 | 1 |
| 10 15 20 | 1 |

Шаг равен

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер узла | r | z | Номер узла | r | z |
| 0 | 1,00000 | 1,00000 | 41 | 3,37500 | 3,12500 |
| 1 | 1,50000 | 1,12500 | 42 | 3,75000 | 3,25000 |
| 2 | 2,00000 | 1,25000 | 43 | 4,12500 | 3,37500 |
| 3 | 2,50000 | 1,37500 | 44 | 4,50000 | 3,50000 |
| 4 | 3,00000 | 1,50000 | 45 | 1,62500 | 2,87500 |
| 5 | 3,50000 | 1,62500 | 46 | 1,96875 | 3,00000 |
| 6 | 4,00000 | 1,75000 | 47 | 2,31250 | 3,12500 |
| 7 | 4,50000 | 1,87500 | 48 | 2,65625 | 3,25000 |
| 8 | 5,00000 | 2,00000 | 49 | 3,00000 | 3,37500 |
| 9 | 1,12500 | 1,37500 | 50 | 3,34375 | 3,50000 |
| 10 | 1,59375 | 1,50000 | 51 | 3,68750 | 3,62500 |
| 11 | 2,06250 | 1,62500 | 52 | 4,03125 | 3,75000 |
| 12 | 2,53125 | 1,75000 | 53 | 4,37500 | 3,87500 |
| 13 | 3,00000 | 1,87500 | 54 | 1,75000 | 3,25000 |
| 14 | 3,46875 | 2,00000 | 55 | 2,06250 | 3,37500 |
| 15 | 3,93750 | 2,12500 | 56 | 2,37500 | 3,50000 |
| 16 | 4,40625 | 2,25000 | 57 | 2,68750 | 3,62500 |
| 17 | 4,87500 | 2,37500 | 58 | 3,00000 | 3,75000 |
| 18 | 1,25000 | 1,75000 | 59 | 3,31250 | 3,87500 |
| 19 | 1,68750 | 1,87500 | 60 | 3,62500 | 4,00000 |
| 20 | 2,12500 | 2,00000 | 61 | 3,93750 | 4,12500 |
| 21 | 2,56250 | 2,12500 | 62 | 4,25000 | 4,25000 |
| 22 | 3,00000 | 2,25000 | 63 | 1,87500 | 3,62500 |
| 23 | 3,43750 | 2,37500 | 64 | 2,15625 | 3,75000 |
| 24 | 3,87500 | 2,50000 | 65 | 2,43750 | 3,87500 |
| 25 | 4,31250 | 2,62500 | 66 | 2,71875 | 4,00000 |
| 26 | 4,75000 | 2,75000 | 67 | 3,00000 | 4,12500 |
| 27 | 1,37500 | 2,12500 | 68 | 3,28125 | 4,25000 |
| 28 | 1,78125 | 2,25000 | 69 | 3,56250 | 4,37500 |
| 29 | 2,18750 | 2,37500 | 70 | 3,84375 | 4,50000 |
| 30 | 2,59375 | 2,50000 | 71 | 4,12500 | 4,62500 |
| 31 | 3,00000 | 2,62500 | 72 | 2,00000 | 4,00000 |
| 32 | 3,40625 | 2,75000 | 73 | 2,25000 | 4,12500 |
| 33 | 3,81250 | 2,87500 | 74 | 2,50000 | 4,25000 |
| 34 | 4,21875 | 3,00000 | 75 | 2,75000 | 4,37500 |
| 35 | 4,62500 | 3,12500 | 76 | 3,00000 | 4,50000 |
| 36 | 1,50000 | 2,50000 | 77 | 3,25000 | 4,62500 |
| 37 | 1,87500 | 2,62500 | 78 | 3,50000 | 4,75000 |
| 38 | 2,25000 | 2,75000 | 79 | 3,75000 | 4,87500 |
| 39 | 2,62500 | 2,87500 | 80 | 4,00000 | 5,00000 |
| 40 | 3,00000 | 3,00000 |  |  |  |

|  |  |
| --- | --- |
| Номер конечного элемента | Узлы |
| 0 | 0 1 2 9 10 11 18 19 20 |
| 1 | 2 3 4 11 12 13 20 21 22 |
| 2 | 4 5 6 13 14 15 22 23 24 |
| 3 | 6 7 8 15 16 17 24 25 26 |
| 4 | 18 19 20 27 28 29 36 37 38 |
| 5 | 20 21 22 29 30 31 38 39 40 |
| 6 | 22 23 24 31 32 33 40 41 42 |
| 7 | 24 25 26 33 34 35 42 43 44 |
| 8 | 36 37 38 45 46 47 54 55 56 |
| 9 | 38 39 40 47 48 49 56 57 58 |
| 10 | 40 41 42 49 50 51 58 59 60 |
| 11 | 42 43 44 51 52 53 60 61 62 |
| 12 | 54 55 56 63 64 65 72 73 74 |
| 13 | 56 57 58 65 66 67 74 75 76 |
| 14 | 58 59 60 67 68 69 76 77 78 |
| 15 | 60 61 62 69 70 71 78 79 80 |

|  |  |  |
| --- | --- | --- |
|  | Узлы | Номер формулы |
| Нижняя грань - | 0 1 2 | 0 |
| 2 3 4 | 0 |
| 4 5 6 | 0 |
| 6 7 8 | 0 |
| Правая грань - | 8 17 26 | 0 |
| 26 35 44 | 0 |
| 44 53 62 | 0 |
| 62 71 80 | 0 |
| Верхняя грань - | 72 | 0 |
| 73 | 0 |
| 74 | 0 |
| 75 | 0 |
| 76 | 0 |
| 77 | 0 |
| 78 | 0 |
| 79 | 0 |
| 80 | 0 |
| Левая грань - | 0 9 18 | 1 |
| 18 27 36 | 1 |
| 36 45 54 | 1 |
| 54 63 72 | 1 |

**Вывод:** теоретическое значение порядка сходимости для заданных базисных функций примерно равно трем, что удовлетворяет только первой вложенной сетке. При дальнейшем дроблении порядок заметно падает. Это объясняется заданной функцией, для которой разбиение на равномерной сетке неэффективно, т.к. наибольшая погрешность возникает при максимальных значениях и . Данную проблему можно решить, используя неравномерные сетки.

Шаг равен (коэф. разрядки и равен 1.15)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер узла | r | z | Номер узла | r | z |
| 0 | 1,000 | 1,000 | 13 | 3,97959 | 4,5 |
| 1 | 2,714 | 1,429 | 14 | 4,14286 | 4,57143 |
| 2 | 4,429 | 1,857 | 15 | 1,92857 | 3,78571 |
| 3 | 4,714 | 1,929 | 16 | 2,84694 | 4,21429 |
| 4 | 5,000 | 2,000 | 17 | 3,76531 | 4,64286 |
| 5 | 1,429 | 2,286 | 18 | 3,91837 | 4,71429 |
| 6 | 2,776 | 2,714 | 19 | 4,07143 | 4,78571 |
| 7 | 4,122 | 3,143 | 20 | 2 | 4 |
| 8 | 4,347 | 3,214 | 21 | 2,85714 | 4,42857 |
| 9 | 4,571 | 3,286 | 22 | 3,71429 | 4,85714 |
| 10 | 1,857 | 3,571 | 23 | 3,85714 | 4,92857 |
| 11 | 2,837 | 4,000 | 24 | 4 | 5 |
| 12 | 3,816 | 4,429 |  |  |  |

|  |  |
| --- | --- |
| Номер конечного элемента | Узлы |
| 0 | 0 1 2 5 6 7 10 11 12 |
| 1 | 2 3 4 7 8 9 12 13 14 |
| 2 | 10 11 12 15 16 17 20 21 22 |
| 3 | 12 13 14 17 18 19 22 23 24 |

|  |  |  |
| --- | --- | --- |
|  | Узлы | Номер формулы |
| Нижняя грань - | 0 1 2 | 0 |
| 2 3 4 | 0 |
| Правая грань - | 4 9 14 | 0 |
| 14 19 24 | 0 |
| Верхняя грань - | 20 | 0 |
| 21 | 0 |
| 22 | 0 |
| 23 | 0 |
| 24 | 0 |
| Левая грань - | 0 5 10 | 1 |
| 10 15 20 | 1 |

В итоге

**Вывод:** для неравномерной сетки значение порядка сходимости совпало с теоретическим.

1. **Тексты основных модулей программ**

#include <iostream>

#include <fstream>

#include <vector>

#include <functional>

#include <set>

#include <iomanip>

#include <math.h>

using namespace std;

#pragma region Объявление глобальных переменных

int num\_split\_edge, num\_nodes, num\_split; //число узлов в ребре, число узлов, количество разбиений

double beta = 1, coef\_z = 1.15, coef\_r = 1.15; // параметр бета(константа), коэф. разрядки z, коэф. разрядки r

vector<int> ia, ja, choice;

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

vector<function<double(double, double)>> basic\_func, deriv\_basic\_func\_xi, deriv\_basic\_func\_eta;

vector<pair<double, double>> nodes; // узлы сетки, заданные координатами

vector<vector<vector<int>>> edge; // массив ребер с краевыми соответствующего рода

vector<vector<int>> array\_quad; // массив четырехугольников

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

#pragma endregion

#pragma region Функции краевых условий и задание этих функций

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;

choice.resize(4);

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

cin >> choice[0];

cout << endl;

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

cin >> choice[1];

cout << endl;

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

cin >> choice[2];

cout << endl;

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

cin >> choice[3];

cout << endl;

}

double u\_g(int var, double r, double z) { // краевое условие первого рода

if (var == 0) {

return exp(r \* z);

}

else return exp(r \* z);

}

double theta(int var, double r, double z) { // краевое условие второго рода

if (var == 0) {

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

if (choice[i] != 2) continue;

return (exp(r \* z) \* z \* normal[i \* 2] + exp(r \* z) \* r \* normal[i \* 2 + 1]);

}

}

else {

bool first\_condit = 1;

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

if (choice[i] != 2) continue;

if (first\_condit) {

first\_condit = 0;

continue;

}

return (exp(r \* z) \* z \* normal[i \* 2] + exp(r \* z) \* r \* normal[i \* 2 + 1]);

}

}

}

double u\_beta(int var, double r, double z) { // краевое условие третьего рода

if (var == 0) {

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

if (choice[i] != 3) continue;

return (exp(r \* z) \* z \* normal[i \* 2] + exp(r \* z) \* r \* normal[i \* 2 + 1]) + exp(r \* z);

}

}

else {

bool first\_condit = 1;

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

if (choice[i] != 3) continue;

if (first\_condit) {

first\_condit = 0;

continue;

}

return (exp(r \* z) \* z \* normal[i \* 2] + exp(r \* z) \* r \* normal[i \* 2 + 1]) + exp(r \* z);

}

}

}

#pragma endregion

#pragma region Замена переменных

double r(int var, double xi, double eta, vector<double> \_r) {

if (var == 0) {

return (1 - xi) \* (1 - eta) \* \_r[0] + xi \* (1 - eta) \* \_r[1] + (1 - xi) \* eta \* \_r[2] + xi \* eta \* \_r[3]; // нет внутренних узлов

}

else {

return (1 - xi) \* (1 - eta) \* \_r[0] + xi \* (1 - eta) \* \_r[2] + (1 - xi) \* eta \* \_r[6] + xi \* eta \* \_r[8]; // есть внутренние узлы

}

}

double z(double xi, double eta, vector<double> \_z) {

return (1 - xi) \* (1 - eta) \* \_z[0] + xi \* (1 - eta) \* \_z[1] + (1 - xi) \* eta \* \_z[2] + xi \* eta \* \_z[3];

}

#pragma endregion

#pragma region Задание функции, лямбды и якобиана преобразования

double f(double r, double z) {

return exp(r \* z) \* (- z / r - z \* z - r \* r + 1 / r \* r);

}

double lambda(double z) {

return 1;

}

double Jacobian(double xi, double eta, vector<double> a) {

return a[0] + a[1] \* xi + a[2] \* eta;

}

#pragma endregion

#pragma region Базисные функции

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 BasicFunc() {

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

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

basic\_func.push\_back([i, j](double xi, double eta) { return FiniteFunc(j, xi) \* FiniteFunc(i, eta); });

}

}

}

void DerivBasicFuncXi() {

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

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

deriv\_basic\_func\_xi.push\_back([i, j](double xi, double eta) { return DerivFiniteFunc(j, xi) \* FiniteFunc(i, eta); });

}

}

}

void DerivBasicFuncEta() {

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

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

deriv\_basic\_func\_eta.push\_back([i, j](double xi, double eta) { return FiniteFunc(j, xi) \* DerivFiniteFunc(i, eta); });

}

}

}

#pragma endregion

#pragma region Гаусс

double Gauss4MatrMass(vector<double> a, vector<double> \_r, vector<double> \_z, int \_i, int \_j) {

double w[4], t[4];

w[0] = 0.6521451548625; w[1] = w[0], w[2] = 0.3478548451375, w[3] = w[2];

t[0] = -0.3399810435849, t[1] = -t[0], t[2] = -0.8611363115941, t[3] = -t[2];

double integral = 0;

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

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

double point\_i = (1.0 + t[i]) / 2.0;

double point\_j = (1.0 + t[j]) / 2.0;

double point\_z = z(point\_i, point\_j, \_z);

integral += w[i] \* w[j] \* Jacobian(point\_i, point\_j, a) \* lambda(point\_z) \* basic\_func[\_i](point\_i, point\_j)

\* basic\_func[\_j](point\_i, point\_j) / r(0, point\_i, point\_j, \_r);

}

}

return integral / double(4);

}

double Gauss4MatrStiff(vector<double> a, vector<double> \_r, vector<double> \_z, vector<double> beta, int \_i, int \_j) {

double w[4], t[4];

w[0] = 0.6521451548625; w[1] = w[0], w[2] = 0.3478548451375, w[3] = w[2];

t[0] = -0.3399810435849, t[1] = -t[0], t[2] = -0.8611363115941, t[3] = -t[2];

double integral = 0;

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

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

double point\_i = (1.0 + t[i]) / 2.0;

double point\_j = (1.0 + t[j]) / 2.0;

double point\_z = z(point\_i, point\_j, \_z);

integral += (w[i] \* w[j]) \* lambda(point\_z) \* r(0, point\_i, point\_j, \_r) \* (

(deriv\_basic\_func\_xi[\_i](point\_i, point\_j) \* (beta[5] \* point\_i + beta[2])

- deriv\_basic\_func\_eta[\_i](point\_i, point\_j) \* (beta[5] \* point\_j + beta[3])) \*

(deriv\_basic\_func\_xi[\_j](point\_i, point\_j) \* (beta[5] \* point\_i + beta[2])

- deriv\_basic\_func\_eta[\_j](point\_i, point\_j) \* (beta[5] \* point\_j + beta[3])) +

(deriv\_basic\_func\_eta[\_i](point\_i, point\_j) \* (beta[4] \* point\_j + beta[1])

- deriv\_basic\_func\_xi[\_i](point\_i, point\_j) \* (beta[4] \* point\_i + beta[0])) \*

(deriv\_basic\_func\_eta[\_j](point\_i, point\_j) \* (beta[4] \* point\_j + beta[1])

- deriv\_basic\_func\_xi[\_j](point\_i, point\_j) \* (beta[4] \* point\_i + beta[0]))

) / Jacobian(point\_i, point\_j, a);

}

}

return integral / double(4);

}

double Gauss4VecB(vector<double> a, vector<double> \_r, int \_i, int \_j) {

double w[4], t[4];

w[0] = 0.6521451548625; w[1] = w[0], w[2] = 0.3478548451375, w[3] = w[2];

t[0] = -0.3399810435849, t[1] = -t[0], t[2] = -0.8611363115941, t[3] = -t[2];

double integral = 0;

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

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

double point\_i = (1.0 + t[i]) / 2.0;

double point\_j = (1.0 + t[j]) / 2.0;

integral += w[i] \* w[j] \* Jacobian(point\_i, point\_j, a) \* r(1, point\_i, point\_j, \_r)

\* basic\_func[\_i](point\_i, point\_j) \* basic\_func[\_j](point\_i, point\_j);

}

}

return integral / double(4);

}

#pragma endregion

#pragma region Генерация сетки

void GenEndElGrid() { // создание конечноэлементной сетки

ifstream file\_in("grid\_coordinates.txt");

file\_in >> num\_split;

num\_split\_edge = 2 \* num\_split + 1; //число узлов в ребре

num\_nodes = pow(num\_split\_edge, 2); //число узлов

nodes.resize(num\_nodes);

vector<pair<double, double>> step(4); // шаг разбиений на конкретном ребре

double a, b;

int l\_d, r\_d, l\_u, r\_u; //левый нижный, правный нижний, правый верхний, левый верхний соответственно

l\_d = 0;

r\_d = num\_split\_edge - 1;

l\_u = num\_nodes - num\_split\_edge;

r\_u = num\_nodes - 1;

file\_in >> a >> b;

nodes[l\_d] = { a, b };

file\_in >> a >> b;

nodes[r\_d] = { a, b };

file\_in >> a >> b;

nodes[r\_u] = { a, b };

file\_in >> a >> b;

nodes[l\_u] = { a, b };

file\_in.close();

int temp\_1, temp\_2;

if (coef\_r > 1) temp\_1 = 1 / (coef\_r - 1);

else temp\_1 = 1 / coef\_r;

if (coef\_z > 1) temp\_2 = 1 / (coef\_z - 1);

else temp\_2 = 1 / coef\_z;

int num\_eq\_parts\_r, num\_eq\_parts\_z;

if (coef\_r != 1) {

num\_eq\_parts\_r = (1 - pow(temp\_1, num\_split)) / (1 - temp\_1); // количество равных частей(по формуле геом. прог.)

}

else num\_eq\_parts\_r = num\_split;

if (coef\_z != 1) {

num\_eq\_parts\_z = (1 - pow(temp\_2, num\_split)) / (1 - temp\_2); // количество равных частей(по формуле геом. прог.)

}

else num\_eq\_parts\_z = num\_split;

step[0] = { (nodes[r\_d].first - nodes[l\_d].first) / num\_eq\_parts\_r, (nodes[r\_d].second - nodes[l\_d].second) / num\_eq\_parts\_z }; //шаг(разбиение) для нижнего ребра

step[1] = { (nodes[r\_u].first - nodes[r\_d].first) / num\_eq\_parts\_r, (nodes[r\_u].second - nodes[r\_d].second) / num\_eq\_parts\_z }; //шаг(разбиение) для правого ребра

step[2] = { (nodes[r\_u].first - nodes[l\_u].first) / num\_eq\_parts\_r, (nodes[r\_u].second - nodes[l\_u].second) / num\_eq\_parts\_z }; //шаг(разбиение) для верхнего ребра

step[3] = { (nodes[l\_u].first - nodes[l\_d].first) / num\_eq\_parts\_r, (nodes[l\_u].second - nodes[l\_d].second) / num\_eq\_parts\_z }; //шаг(разбиение) для левого ребра

if (coef\_r > 1) {

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

step[i].first \*= temp\_1;

}

}

if (coef\_z > 1) {

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

step[i].second \*= temp\_2;

}

}

for (int j = 2; j < num\_split\_edge - 1; j++) {

if (j % 2 != 0) continue;

nodes[j].first = nodes[j - 2].first + step[0].first; //нефиктивные узлы нижнего ребра

nodes[j].second = (nodes[r\_d].second - nodes[l\_d].second) \* ((nodes[j].first - nodes[l\_d].first) / (nodes[r\_d].first - nodes[l\_d].first)) + nodes[l\_d].second; //нефиктивные узлы нижнего ребра

if (coef\_r > 1) step[0].first /= temp\_1;

else step[0].first \*= temp\_1;

}

for (int j = 1; j < num\_split\_edge - 1; j++) {

if (j % 2 == 0) continue;

nodes[j] = { (nodes[j - 1].first + nodes[j + 1].first) / 2, (nodes[j - 1].second + nodes[j + 1].second) / 2 }; //фиктивные узлы нижнего ребра

}

for (int j = 2; j < num\_split\_edge - 1; j++) {

if (j % 2 != 0) continue;

nodes[l\_u + j].first = nodes[l\_u + j - 2].first + step[2].first; //нефиктивные узлы верхнего ребра

nodes[l\_u + j].second = (nodes[r\_u].second - nodes[l\_u].second) \* ((nodes[l\_u + j].first - nodes[l\_u].first) / (nodes[r\_u].first - nodes[l\_u].first)) + nodes[l\_u].second; //нефиктивные узлы верхнего ребра

if (coef\_r > 1) step[2].first /= temp\_1;

else step[2].first \*= temp\_1;

}

for (int j = 1; j < num\_split\_edge - 1; j++) {

if (j % 2 == 0) continue;

nodes[l\_u + j] = { (nodes[l\_u + j - 1].first + nodes[l\_u + j + 1].first) / 2, (nodes[l\_u + j - 1].second + nodes[l\_u + j + 1].second) / 2 }; //фиктивные узлы верхнего ребра

}

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

if (i == l\_d || i == l\_u) {

i += num\_split\_edge;

continue;

}

if (i % 2 \* num\_split\_edge != 0) {

i += num\_split\_edge;

continue;

}

double in\_step\_r;

nodes[i].second = nodes[i - 2 \* num\_split\_edge].second + step[3].second; //начальная вершина в строке по z

nodes[i].first = (nodes[l\_u].first - nodes[l\_d].first) \* ((nodes[i].second - nodes[l\_d].second) / (nodes[l\_u].second - nodes[l\_d].second)) + nodes[l\_d].first; //начальная вершина в строке по r

nodes[i + num\_split\_edge - 1].second = nodes[i - num\_split\_edge - 1].second + step[1].second; //конечная вершина в строке по z

nodes[i + num\_split\_edge - 1].first = (nodes[r\_u].first - nodes[r\_d].first) \* ((nodes[i + num\_split\_edge - 1].second - nodes[r\_d].second) / (nodes[r\_u].second - nodes[r\_d].second)) + nodes[r\_d].first; //конечная вершина в строке по r

in\_step\_r = (nodes[i + num\_split\_edge - 1].first - nodes[i].first) / num\_eq\_parts\_r;

if (coef\_r > 1) in\_step\_r \*= temp\_1;

for (int j = 2; j < num\_split\_edge - 1; j++) {

if (j % 2 != 0) continue;

nodes[i + j].first = nodes[i + j - 2].first + in\_step\_r;

nodes[i + j].second = (nodes[i + num\_split\_edge - 1].second - nodes[i].second) \* ((nodes[i + j].first - nodes[i].first) / (nodes[i + num\_split\_edge - 1].first - nodes[i].first)) + nodes[i].second;

if (coef\_r > 1) in\_step\_r /= temp\_1;

else in\_step\_r \*= temp\_1;

}

for (int j = 1; j < num\_split\_edge - 1; j++) {

if (j % 2 == 0) continue;

nodes[i + j] = { (nodes[i + j - 1].first + nodes[i + j + 1].first) / 2, (nodes[i + j - 1].second + nodes[i + j + 1].second) / 2 };

}

i += num\_split\_edge;

if (coef\_z > 1) step[3].second /= temp\_2, step[1].second /= temp\_2;

else step[3].second \*= temp\_2, step[1].second \*= temp\_2;

}

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

if (i == l\_d || i == l\_u) {

i += num\_split\_edge;

continue;

}

if (i % 2 \* num\_split\_edge == 0) {

i += num\_split\_edge;

continue;

}

double in\_step\_r;

nodes[i] = { (nodes[i + num\_split\_edge].first + nodes[i - num\_split\_edge].first) / 2, (nodes[i + num\_split\_edge].second + nodes[i - num\_split\_edge].second) / 2 }; //начальная вершина в строке

nodes[i + num\_split\_edge - 1] = { (nodes[i + 2 \* num\_split\_edge - 1].first + nodes[i - 1].first) / 2, (nodes[i + 2 \* num\_split\_edge - 1].second + nodes[i - 1].second) / 2 }; //конечная вершина в строке

in\_step\_r = (nodes[i + num\_split\_edge - 1].first - nodes[i].first) / num\_eq\_parts\_r;

if (coef\_r > 1) in\_step\_r \*= temp\_1;

for (int j = 2; j < num\_split\_edge - 1; j++) {

if (j % 2 != 0) continue;

nodes[i + j].first = nodes[i + j - 2].first + in\_step\_r;

nodes[i + j].second = (nodes[i + num\_split\_edge - 1].second - nodes[i].second) \* ((nodes[i + j].first - nodes[i].first) / (nodes[i + num\_split\_edge - 1].first - nodes[i].first)) + nodes[i].second;

if (coef\_r > 1) in\_step\_r /= temp\_1;

else in\_step\_r \*= temp\_1;

}

for (int j = 1; j < num\_split\_edge - 1; j++) {

if (j % 2 == 0) continue;

nodes[i + j] = { (nodes[i + j - 1].first + nodes[i + j + 1].first) / 2, (nodes[i + j - 1].second + nodes[i + j + 1].second) / 2 };

}

i += num\_split\_edge;

}

}

#pragma endregion

#pragma region Создание массива конечных элементов

void ArrayQuadrilaterals() { // массив четырехугольников

array\_quad.resize(num\_split \* num\_split);

int k = 0;

for (int i = 0; i < num\_split \* num\_split;) {

for (int j = 0; j < num\_split; j++) {

int l\_d\_node = k + 2 \* j; // нижний левый номер узла конечного элемента

array\_quad[i + j].push\_back(l\_d\_node);

array\_quad[i + j].push\_back(l\_d\_node + 1);

array\_quad[i + j].push\_back(l\_d\_node + 2);

array\_quad[i + j].push\_back(l\_d\_node + num\_split\_edge);

array\_quad[i + j].push\_back(l\_d\_node + 1 + num\_split\_edge);

array\_quad[i + j].push\_back(l\_d\_node + 2 + num\_split\_edge);

array\_quad[i + j].push\_back(l\_d\_node + 2 \* num\_split\_edge);

array\_quad[i + j].push\_back(l\_d\_node + 1 + 2 \* num\_split\_edge);

array\_quad[i + j].push\_back(l\_d\_node + 2 + 2 \* num\_split\_edge);

}

i += num\_split;

k += 2 \* num\_split\_edge;

}

}

#pragma endregion

#pragma region Генерация портрета и подпр-мы добавления в глоб. матрицу

void GeneratePortrait() { // генерация портрета матрицы

vector<set<int>> list(num\_nodes);

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

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

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

list[array\_quad[i][j]].insert(array\_quad[i][m]);

}

}

}

ia.resize(num\_nodes + 1, 0);

for (int i = 2; i <= num\_nodes; i++) {

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

}

ja.resize(ia[num\_nodes]);

auto iter = ja.begin();

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

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

iter += list[i].size();

}

aal.resize(ia[num\_nodes]);

di.resize(num\_nodes);

b.resize(num\_nodes);

L\_sq.resize(ia[num\_nodes]);

di\_sq.resize(num\_nodes);

}

void AddLocalMatr(vector<int> node\_num, vector<vector<double>> loc\_matr) { // добавление матрицы в глобальную

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

di[node\_num[i]] += loc\_matr[i][i];

}

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

int i\_beg = ia[node\_num[i]];

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

int i\_end = ia[node\_num[i] + 1];

while (ja[i\_beg] != node\_num[j]) {

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

if (ja[ind] <= node\_num[j]) i\_beg = ind;

else i\_end = ind;

}

aal[i\_beg] += loc\_matr[i][j];

i\_beg++;

}

}

}

void AddLocalVec(vector<int> node\_num, vector<double> loc\_vec) { // добавление вектора в глобальный

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

b[node\_num[i]] += loc\_vec[i];

}

}

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 = ia[edge[2][num\_edge][i]];

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

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

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

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

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

else i\_end = ind;

}

aal[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];

}

}

#pragma endregion

#pragma region Работа с краевыми условиями

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

switch (i)

{

case 0:

return 0;

break;

case 1:

return num\_split\_edge - 1;

break;

case 2:

return num\_nodes - num\_split\_edge;

break;

case 3:

return 0;

break;

}

}

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

if (i == 0 || i == 2) {

return 2;

}

else return 2 \* num\_split\_edge;

}

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 (choice[i])

{

case 1:

for (int j = 0; j < num\_split\_edge; j++) {

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

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

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

}

num\_func\_1++;

break;

case 2:

for (int j = 0; j < num\_split; j++) {

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

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

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

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

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

start\_point += step;

}

num\_func\_2++;

break;

case 3:

for (int j = 0; j < num\_split; j++) {

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

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

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

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

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

start\_point += step;

}

num\_func\_3++;

break;

}

}

}

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 GetVecNormal() { // получаем значения векторов нормали на необходимых ребрах

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

if (choice[i] == 1) {

normal.push\_back(0);

normal.push\_back(0);

continue;

}

double coef\_r, coef\_z, sq;

switch (i)

{

case 0:

coef\_r = 1 / (nodes[num\_split\_edge - 1].first - nodes[0].first);

coef\_z = 1 / (nodes[0].second - nodes[num\_split\_edge - 1].second);

if (coef\_z > 0) coef\_r \*= -1, coef\_z \*= -1;

break;

case 1:

coef\_r = 1 / (nodes[num\_nodes - 1].first - nodes[num\_split\_edge - 1].first);

coef\_z = 1 / (nodes[num\_split\_edge - 1].second - nodes[num\_nodes - 1].second);

if (coef\_r < 0) coef\_r \*= -1, coef\_z \*= -1;

break;

case 2:

coef\_r = 1 / (nodes[num\_nodes - 1].first - nodes[num\_nodes - num\_split\_edge].first);

coef\_z = 1 / (nodes[num\_nodes - num\_split\_edge].second - nodes[num\_nodes - 1].second);

if (coef\_z < 0) coef\_r \*= -1, coef\_z \*= -1;

break;

case 3:

coef\_r = 1 / (nodes[num\_nodes - num\_split\_edge].first - nodes[0].first);

coef\_z = 1 / (nodes[0].second - nodes[num\_nodes - num\_split\_edge].second);

if (coef\_r > 0) coef\_r \*= -1, coef\_z \*= -1;

break;

}

sq = sqrt(coef\_r \* coef\_r + coef\_z \* coef\_z);

normal.push\_back(coef\_r / sq);

normal.push\_back(coef\_z / sq);

}

}

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) { // учет краевых условий первого типа

double \_r, \_z;

\_r = nodes[edge[0][node\_num][0]].first, \_z = nodes[edge[0][node\_num][0]].second;

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

di[edge[0][node\_num][0]] = 1;

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

int \_i = ja[i];

if (FindInd(\_i)) {

aal[i] = 0;

continue;

}

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

aal[i] = 0;

}

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

int k = 0;

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

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

if (FindInd(i)) {

aal[j] = 0;

continue;

}

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

aal[j] = 0;

}

}

}

}

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

vector<double> \_r(3), \_z(3);

\_r[0] = nodes[edge[1][num\_edge][0]].first, \_z[0] = nodes[edge[1][num\_edge][0]].second;

\_r[1] = nodes[edge[1][num\_edge][1]].first, \_z[1] = nodes[edge[1][num\_edge][1]].second;

\_r[2] = nodes[edge[1][num\_edge][2]].first, \_z[2] = nodes[edge[1][num\_edge][2]].second;

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

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

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

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 ConsiderBoundConditThirdType(int num\_edge) { // учет краевых условий третьего типа

vector<double> \_r(3), \_z(3);

\_r[0] = nodes[edge[2][num\_edge][0]].first, \_z[0] = nodes[edge[2][num\_edge][0]].second;

\_r[1] = nodes[edge[2][num\_edge][1]].first, \_z[1] = nodes[edge[2][num\_edge][1]].second;

\_r[2] = nodes[edge[2][num\_edge][2]].first, \_z[2] = nodes[edge[2][num\_edge][2]].second;

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]);

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

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

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 ConsiderBoundCondit() { // учет всех краевых

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

ConsiderBoundConditThirdType(i);

}

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

ConsiderBoundConditSecType(i);

}

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

ConsiderBoundConditFirstType(i);

}

}

#pragma endregion

#pragma region Работа с локальными матрицами и векторами

void LocalMatrMass(vector<double> \_r, vector<double> \_z, vector<double> a, vector<int> node\_num, int sign\_a0) { // локальная матрица массы

vector<vector<double>> loc\_matr\_mass(9);

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

loc\_matr\_mass[i].resize(i + 1);

}

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

for (int j = i; j < 9; j++) {

if (i == j) {

loc\_matr\_mass[i][i] = sign\_a0 \* Gauss4MatrMass(a, \_r, \_z, i, i);

}

else {

loc\_matr\_mass[j][i] = sign\_a0 \* Gauss4MatrMass(a, \_r, \_z, j, i);

}

}

}

AddLocalMatr(node\_num, loc\_matr\_mass);

}

void LocalMatrStiff(vector<double> \_r, vector<double> \_z, vector<double> a, vector<int> node\_num, int sign\_a0) { // локаланая матрица жесткости

vector<double> beta(6);

beta[0] = \_r[2] - \_r[0];

beta[1] = \_r[1] - \_r[0];

beta[2] = \_z[2] - \_z[0];

beta[3] = \_z[1] - \_z[0];

beta[4] = \_r[0] - \_r[1] - \_r[2] + \_r[3];

beta[5] = \_z[0] - \_z[1] - \_z[2] + \_z[3];

vector<vector<double>> loc\_matr\_stiff(9);

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

loc\_matr\_stiff[i].resize(i + 1);

}

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

for (int j = i; j < 9; j++) {

if (i == j) {

loc\_matr\_stiff[i][i] = sign\_a0 \* Gauss4MatrStiff(a, \_r, \_z, beta, i, i);

}

else {

loc\_matr\_stiff[j][i] = sign\_a0 \* Gauss4MatrStiff(a, \_r, \_z, beta, j, i);

}

}

}

AddLocalMatr(node\_num, loc\_matr\_stiff);

}

void localVecB(vector<double> \_r, vector<double> \_z, vector<double> p, vector<double> a, vector<int> node\_num, int sign\_a0) {

vector<double> loc\_vec\_b(9, 0);

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

for (int j = 0; j < 9; j++) {

loc\_vec\_b[i] += p[j] \* Gauss4VecB(a, \_r, j, i);

}

loc\_vec\_b[i] \*= sign\_a0;

}

AddLocalVec(node\_num, loc\_vec\_b);

}

#pragma endregion

#pragma region Построение глобальной матрицы и вектора

void BuildMatrA() {

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

vector<int> node\_num(9);

for (int j = 0; j < 9; j++) {

node\_num[j] = array\_quad[i][j];

}

vector<double> \_r(4), \_z(4);

\_r[0] = nodes[node\_num[0]].first, \_z[0] = nodes[node\_num[0]].second;

\_r[1] = nodes[node\_num[2]].first, \_z[1] = nodes[node\_num[2]].second;

\_r[2] = nodes[node\_num[6]].first, \_z[2] = nodes[node\_num[6]].second;

\_r[3] = nodes[node\_num[8]].first, \_z[3] = nodes[node\_num[8]].second;

vector<double> a(3);

a[0] = (\_r[1] - \_r[0]) \* (\_z[2] - \_z[0]) - (\_z[1] - \_z[0]) \* (\_r[2] - \_r[0]);

a[1] = (\_r[1] - \_r[0]) \* (\_z[3] - \_z[2]) - (\_z[1] - \_z[0]) \* (\_r[3] - \_r[2]);

a[2] = (\_r[3] - \_r[1]) \* (\_z[2] - \_z[0]) - (\_z[3] - \_z[1]) \* (\_r[2] - \_r[0]);

int sign\_a0;

if (a[0] > 0) sign\_a0 = 1;

else sign\_a0 = -1;

LocalMatrMass(\_r, \_z, a, node\_num, sign\_a0);

LocalMatrStiff(\_r, \_z, a, node\_num, sign\_a0);

}

}

void BuildVecB() {

for (int k = 0; k < num\_split \* num\_split; k++) {

vector<int> node\_num(9);

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

node\_num[i] = array\_quad[k][i];

}

vector<double> \_r(9), \_z(9);

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

\_r[i] = nodes[node\_num[i]].first, \_z[i] = nodes[node\_num[i]].second;

}

vector<double> a(3);

a[0] = (\_r[2] - \_r[0]) \* (\_z[6] - \_z[0]) - (\_z[2] - \_z[0]) \* (\_r[6] - \_r[0]);

a[1] = (\_r[2] - \_r[0]) \* (\_z[8] - \_z[6]) - (\_z[2] - \_z[0]) \* (\_r[8] - \_r[6]);

a[2] = (\_r[8] - \_r[2]) \* (\_z[6] - \_z[0]) - (\_z[8] - \_z[2]) \* (\_r[6] - \_r[0]);

vector<double> p(9);

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

p[i] = f(\_r[i], \_z[i]);

}

int sign\_a0;

if (a[0] > 0) sign\_a0 = 1;

else sign\_a0 = -1;

localVecB(\_r, \_z, p, a, node\_num, sign\_a0);

}

}

#pragma endregion

#pragma region МСГ

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 = ia[i]; k < ia[i + 1]; k++) {

int j = ja[k];

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

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

}

}

}

void LU\_sq(int n) {

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

double s\_d = 0;

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

double s\_l = 0;

int j = ja[k];

int j0 = ia[j];

int j1 = ia[j + 1];

int ki = ia[i];

int kj = j0;

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

int jl = ja[ki];

int ju = ja[kj];

if (jl == ju) {

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

ki++, kj++;

}

else if (jl < ju) ki++;

else kj++;

}

L\_sq[k] = (aal[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 = ia[i]; k < ia[i + 1]; k++) {

int j = ja[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 = ia[i]; k < ia[i + 1]; k++) {

int j = ja[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;

}

#pragma endregion

#pragma region Тестирование

void Test() {

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

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

q\_u[i] = u\_g(0, nodes[i].first, nodes[i].second);

}

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

for (int i = 0; i < num\_nodes; 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:" << endl;

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

}

#pragma endregion

int main()

{

setlocale(LC\_ALL, "Russian");

EnterBoundCondit();

GenEndElGrid();

ArrayQuadrilaterals();

BasicFunc();

DerivBasicFuncXi();

DerivBasicFuncEta();

BoundCondit();

GetVecNormal();

BuildBoundMatrices();

GeneratePortrait();

BuildMatrA();

BuildVecB();

ConsiderBoundCondit();

q.resize(num\_nodes, 0);

vector<double> r(num\_nodes);

vector<double> z(num\_nodes);

vector<double> Mult(num\_nodes);

vector<double> Az(num\_nodes);

int max\_iter = 1000;

double eps = 1e-15;

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

Test();

}