|  |  |  |
| --- | --- | --- |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
|  | | |
| Кафедра прикладной математики | | |
| Курсовая проект по курсу | | |
| «Численные методы» | | |
|  | | |
|  | | |
|  |  |  |
|  | Группа пм-12 |
|  | Студент: Лойченко данила сергеевич |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Новосибирск | | |

2023

1. **Условие задания**

МКЭ для трехмерной краевой задачи для уравнения эллиптического типа в декартовой системе координат. Базисные функции линейные на прямых призмах с четырехугольным основанием (основание призмы в координатах (x, y)). Краевые условия всех типов. Коэффициент диффузии λ - кусочно-постоянная функция. Матрицу СЛАУ генерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

1. **Постановка задачи**

Эллиптическая краевая задача для функции u определяется дифференциальным уравнением

,

заданным в некоторой области Ω с границей , иA краевыми условиями

- значение искомой функции на границе

- значение на производной функции по направлению внешней нормали к поверхности

Дифференциальное уравнение для двумерной эллиптической краевой задачи в декартовой системе координат может быть записано в виде:

.

1. **Теоретическая часть**

*Вариационная постановка*

Уравнение можно также записать:

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

Преобразуем слагаемое с использованием формул Грина:

Где границы, на которых заданы соответствующие краевые условия.

Интегралы по границам можно преобразовать, воспользовавшись краевыми условиями:

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

В качестве мы можем выбрать .

Таким образом, получаем вариационное уравнение вида:

Получим аппроксимацию уравнения Галёркина на конечномерных подпространствах , аппроксимирующих исходные пространства

Поскольку любая функция ∈ может быть представлена в виде линейной комбинации

вариационное уравнение эквивалентно следующей системе уравнений:

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

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

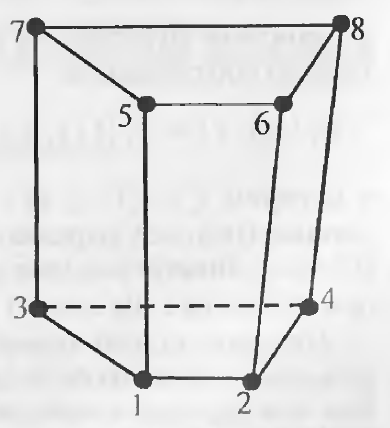
Поскольку исходная задача рассматривается в декартовой системе координат, то

и, соответственно:

Отсюда получаем:

*Конечноэлементная дискретизация и переход к локальным матрицами*

Определим на прямой призме, в основании которой лежит четырехугольник, область  с вершинами



и восемь линейных базисных функций , равных единице в одной вершине призмы с соответствующим номером и нулю во всех остальных, которые в свою очередь состоят из:

* Двумерных линейных базисных функции , заданных на двумерном четырехугольнике  с вершинами  являющимся основанием призмы , равные единице в одной из этих вершин и нулю в остальных
* Одномерных линейных базисных функций на отрезке , равные единице в одном из концов



Тогда функция , можно записать в следующем виде:



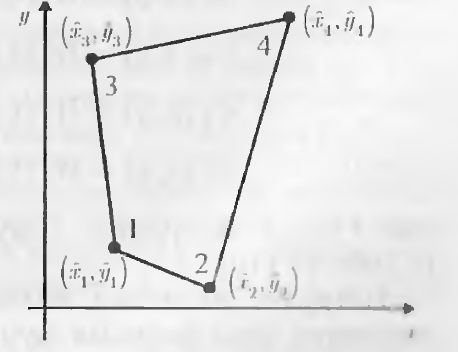
Где  и  определяются по таблице,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|  | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
|  | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |

либо по следующим формулам



Рассмотрим повнимательнее локальные базисные функции  на четырехугольнике с вершинами . При этом вершины должны быть пронумерованы так, чтобы на одной стороне четырехугольника не лежали первая и четвертая вершина (и соответственно вторая и третья)



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



Отобразим  в четырехугольник  с вершинами  с помощью следующих соотношений:



Вышеизложенные соотношения переводят границу  квадрата  в границу четырехугольника , заключенную между вершинами . При этом граница  переходит в границу , граница  - в границу между , а граница – в границу между 

Введем на  билинейные базисные функции

Где  и  - одномерные функции следующего вида:



Тогда базисные функции на четырехугольнике  можно определить с помощью соотношений



Где  и  - функции, неявно определяемые соотношением



Получим соотношения для вычисления элементов локальных матрицы массы и жесткости прямоугольной призмы с четырехугольным основанием. Основанная сложность заключается в том, что базисные функции  не представлены в аналитическом виде. Поэтому при вычислении интегралов мы воспользуемся заменной переменных  и  на  и , что позволит перейти от интегрирование по четырехугольнику  к интегрированию по квадрату , где аналитический вид базисный функций нам известен.

Для упрощения выкладок будем считать, что коэффициенты и постоянны на , поэтому, чтобы не загромождать выкладки они будут опущены в формулах.

Сначала получим формулы для вычисления локальной матрицы массы.

Так как функция на элементе  — это образ функции



Вычислим якобиан  для преобразования координат



Где коэффициенты 



При этом следует учесть, что якобиан  в силу взаимной однозначности отображения



не меняет знак. Тогда модуль якобиана  может быть записан как



В результате матрица массы для аппроксимации с использованием билинейный функций может быть представлена в виде:



Теперь получим формулы для вычисления 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



Обозначим матрицы жесткости и массы соответствующих элементов двумерных и одномерных элементов линейный элементов следующим образом:







Интеграл для будем вычислять численно. Для этого воспользуемся методом Гаусса. Квадратурные формулы для интегрирования прямоугольника (с тремя узлами по каждой из координат) имеют следующий вид:



Где



Для элементного шаблона эти формулы принимают вид:



Тогда компоненты локальной матрицы жесткости элемента могут быть вычислены через компоненты матриц следующим образом.



Нетрудно убедиться, что компоненты локальной матрицы массы элемента вычисляются с помощью матриц 



Для вычисления вектора правой части будем использовать интерполирование функции по локальным базисным функциям



Учет вторых краевых так же основывается на интерполяции функции  по локальным базисным функциям



Если краевое задано на боковой грани, то можно рассматривать локальные базисные функции по двум переменным, так как одна из координат зафиксировано (в силу того, что расчетная область -параллелепипед и его боковые грани параллельны осям координат). Тогда



Учет третьих краевых похож на случай с вторыми краевыми.

Для начала рассмотрим, как учитываются третьи краевые в глобальном матрицы.



А учет третях краевых в вектор правой части почти ничем не отличается случая с вторыми краевыми. Отличие заключается в том, что интерполируется функция уже  по локальным базисным функциям и интеграл домножается на коэффициент 

Учет первых краевых условий проводится после полной сборки глобальной матрицы и правой части (включая учет условий второго и третьего рода). В глобальной матрице и глобальном векторе находим соответствующую глобальному номеру краевого узла строку, и ставим вместо диагонального элемента глобальной матрицы на этой строке единицу и нули вместо внедиагольных элементов, а вместо элемента с таким номером в вектор правой части присваиваем значения первого краевого условия.

1. **Описание разработанных программ**

*Входные данные*

|  |  |
| --- | --- |
| Название файла | Содержание |
| FE.txt | Информация о списке КЭ содержащихся в расчетной области.  Первым числом в файле идет общее число КЭ в расчётной области.  Далее идут семерки целых чисел:  Первые 4 числа задают номер узлов в основании расчетной области, которым соответствуют узлы, находящиеся на более высоком уровне, следующие 2 числа номер верхнего и нижнего предел высоты КЭ, последнее число подобласть, к которой принадлежит КЭ. |
| XY.txt | Информация о списке точек в плоскости Oxy  Первым числом в файле идет общее число точек.  Далее идут пары вещественных чисел  Первое число первая координата точки, второе число вторая координата точки. |
| FirstBC.txt | Информация о списке первых краевых условий.  Первым числом в файле идет общее число первых краевых условий.  Далее идут пары чисел:  Первое целое число номер узла, второе вещественной число значение ug в узле. |
| SecondBC.txt | Информация о списке вторых краевых условий.  Первым числом в файле идет общее число вторых краевых условий.  Далее идут шестерки чисел:  Первые 4 числа по узлам задают грань, 5 число указывает номер формулы, 6 число указывает на то является сторона основанием или боковой гранью |
| ThirdBC.txt | Информация о списке третьих краевых условий.  Первым числом в файле идет общее число третьих краевых условий.  Далее идут семерки чисел:  Первые 4 числа по узлам задают грань, 5 число указывает номер формулы, 6 число численное значение бэты, 7 число указывает на то является сторона основанием или боковой гранью |

*Структура модулей программы*

*Структуры*

struct Point {} - структура для хранения точек

struct FirstBoundary {} - структура для хранения первых краевых

struct SecondBoundary {}- структура для хранения вторых краевых

struct ThirdBoundary {}- структура для хранения третьих краевых

class GridAndSLAE {} - структура для хранения информации о сетке и СЛАУ

Методы GridAndSLAE

void InputFromFile(FILE\* inFE, FILE\* inXY, FILE\* inZ, FILE\* inFirstBC, FILE\* inSecondBC, FILE\* inThirdBC); - считывания данных из файла

void GeneratePortrait(); - генерация портрета матрицы

void CalculateA\_b(); - подсчет глобальной матрицы и глобального вектора из локальных матриц массы и жесткости и локального вектора

void FirstBoundaryConditions(); - учет первых краевых

void SecondBoundaryConditions();- учет вторых краевых

void ThirdBoundaryConditions();- учет третьих краевых

void MSGForNonSymMatrixWithLuSqP(); - решение СЛАУ при помощи МСГ с LU(sq) факторизацией

void OutputSolutionQ(); - вывод вектора ответа

double Gauss3\_Gxy(int i, int j, double b1, double b2, double b3, double b4, double b5, double b6, double a0, double a1, double a2); - численное интегрирование матрицы жесткости для Oxy

double Phi(double e, double n, int i, int j, double b1, double b2, double b3, double b4, double b5, double b6, double a0, double a1, double a2); - функция в подынтегральном выражении матрицы жесткости для Oxy

double TETA(int number, double x, double y, double z); - функция для высчитывания тэты в узле

double UBETA(int number, double x, double y, double z); - функция для высчитывания ub в узле

double GAMMA(int number); - функция для получения гаммы на КЭ

double LAMBDA(int number); - функция для получения лямбды на КЭ

double FUN(int number, double x, double y, double z); - функция для высчитывания f в узле

void AllocateMemory(); - выделение памяти

1. **Тестирование программы**

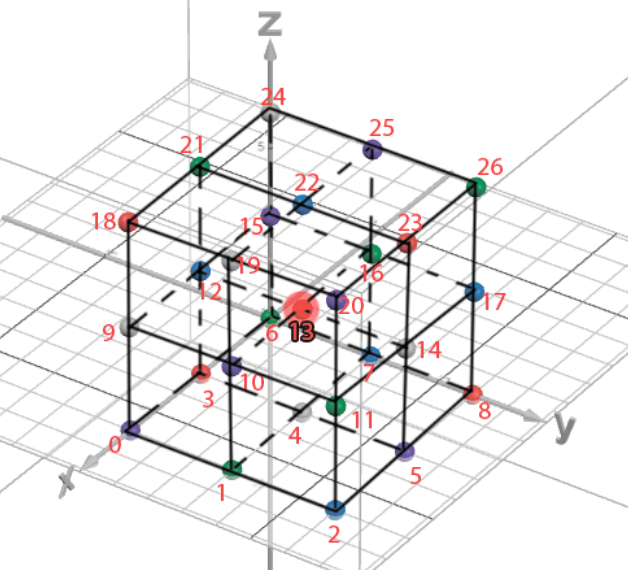
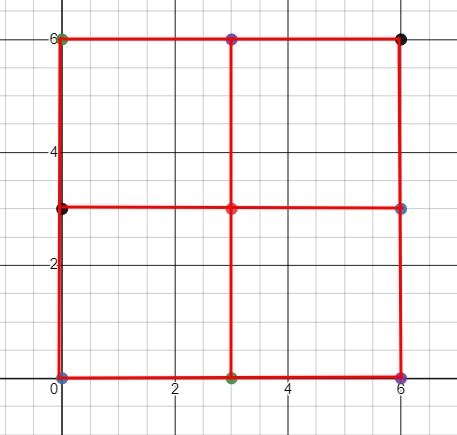
**Тест на первые краевые условия во всех узлах кроме центрального (в основании лежит прямоугольник)**



Входные данные:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FE | XY | Z | inFirstBC | inSecondBC | inThirdBC |
| 8  0 1 3 4 0 1 0  0 1 3 4 1 2 0  1 2 4 5 0 1 0  1 2 4 5 1 2 0  3 4 6 7 0 1 0  3 4 6 7 1 2 0  4 5 7 8 0 1 0  4 5 7 8 1 2 0 | 9  0.000000000000000 0.000000000000000  3.000000000000000 0.000000000000000  6.000000000000000 0.000000000000000  0.000000000000000 3.000000000000000  3.000000000000000 3.000000000000000  6.000000000000000 3.000000000000000  0.000000000000000 6.000000000000000  3.000000000000000 6.000000000000000  6.000000000000000 6.000000000000000 | 3  0.000000000000000  3.000000000000000  6.000000000000000 | 26  0 0  1 3  2 6  3 3  4 6  5 9  6 6  7 9  8 12  9 3  10 6  11 9  12 6  14 12  15 9  16 12  17 15  18 6  19 9  20 12  21 9  22 12  23 15  24 12  25 15  26 18 | 0 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| N узла | q | q\* | |q\*-q| |
| 0 | 0.00000000000000 | 0 | 0.00E+00 |
| 1 | 3.00000000000000 | 3 | 0.00E+00 |
| 2 | 6.00000000000000 | 6 | 0.00E+00 |
| 3 | 3.00000000000000 | 3 | 0.00E+00 |
| 4 | 6.00000000000000 | 6 | 0.00E+00 |
| 5 | 9.00000000000000 | 9 | 0.00E+00 |
| 6 | 6.00000000000000 | 6 | 0.00E+00 |
| 7 | 9.00000000000000 | 9 | 0.00E+00 |
| 8 | 12.00000000000000 | 12 | 0.00E+00 |
| 9 | 3.00000000000000 | 3 | 0.00E+00 |
| 10 | 6.00000000000000 | 6 | 0.00E+00 |
| 11 | 9.00000000000000 | 9 | 0.00E+00 |
| 12 | 6.00000000000000 | 6 | 0.00E+00 |
| 13 | 9.00000000000000 | 9 | 0.00E+00 |
| 14 | 12.00000000000000 | 12 | 0.00E+00 |
| 15 | 9.00000000000000 | 9 | 0.00E+00 |
| 16 | 12.00000000000000 | 12 | 0.00E+00 |
| 17 | 15.00000000000000 | 15 | 0.00E+00 |
| 18 | 6.00000000000000 | 6 | 0.00E+00 |
| 19 | 9.00000000000000 | 9 | 0.00E+00 |
| 20 | 12.00000000000000 | 12 | 0.00E+00 |
| 21 | 9.00000000000000 | 9 | 0.00E+00 |
| 22 | 12.00000000000000 | 12 | 0.00E+00 |
| 23 | 15.00000000000000 | 15 | 0.00E+00 |
| 24 | 12.00000000000000 | 12 | 0.00E+00 |
| 25 | 15.00000000000000 | 15 | 0.00E+00 |
| 26 | 18.00000000000000 | 18 | 0.00E+00 |
|  |  |  |  |



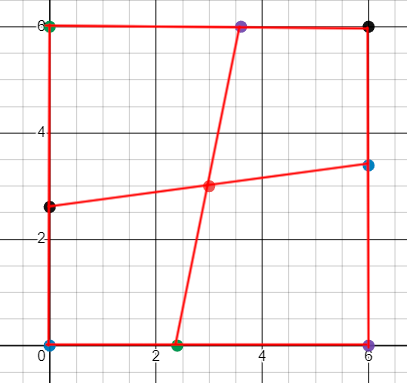
Основание расчетной область в Oxy

**Тест на первые краевые условия во всех узлах кроме центрального (в основании лежит четырехуголник)**



Входные данные:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FE | XY | Z | inFirstBC | inSecondBC | inThirdBC |
| 8  0 1 3 4 0 1 0  0 1 3 4 1 2 0  1 2 4 5 0 1 0  1 2 4 5 1 2 0  3 4 6 7 0 1 0  3 4 6 7 1 2 0  4 5 7 8 0 1 0  4 5 7 8 1 2 0 | 9  0.000000000000000 0.000000000000000  2.400000000000000 0.000000000000000  6.000000000000000 0.000000000000000  0.000000000000000 2.608695652173913  3.000000000000000 3.000000000000000  6.000000000000000 3.391304347826087  0.000000000000000 6.000000000000000  3.600000000000000 6.000000000000000  6.000000000000000 6.000000000000000 | 3  0  3  6 | 26  0 0.000000000000000  1 2.400000000000000  2 6.000000000000000  3 2.608695652173913  4 6.000000000000000  5 9.391304347826086  6 6.000000000000000  7 9.600000000000000  8 12.000000000000000  9 3.000000000000000  10 5.400000000000000  11 9.000000000000000  12 5.608695652173912  14 12.391304347826086  15 9.000000000000000  16 12.600000000000000  17 15.000000000000000  18 6.000000000000000  19 8.400000000000000  20 12.000000000000000  21 8.608695652173912  22 12.000000000000000  23 15.391304347826086  24 12.000000000000000  25 15.600000000000000  26 18.000000000000000 | 0 | 0 |



Основание расчетной область в Oxy

|  |  |  |  |
| --- | --- | --- | --- |
| N узла | q | q\* | |q\*-q| |
| 0 | 0.00000000000000 | 0.00000000000000 | 0.00E+00 |
| 1 | 2.40000000000000 | 2.40000000000000 | 0.00E+00 |
| 2 | 6.00000000000000 | 6.00000000000000 | 0.00E+00 |
| 3 | 2.60869565217391 | 2.60869565217391 | 0.00E+00 |
| 4 | 6.00000000000000 | 6.00000000000000 | 0.00E+00 |
| 5 | 9.39130434782608 | 9.39130434782608 | 0.00E+00 |
| 6 | 6.00000000000000 | 6.00000000000000 | 0.00E+00 |
| 7 | 9.60000000000000 | 9.60000000000000 | 0.00E+00 |
| 8 | 12.00000000000000 | 12.00000000000000 | 0.00E+00 |
| 9 | 3.00000000000000 | 3.00000000000000 | 0.00E+00 |
| 10 | 5.40000000000000 | 5.40000000000000 | 0.00E+00 |
| 11 | 9.00000000000000 | 9.00000000000000 | 0.00E+00 |
| 12 | 5.60869565217391 | 5.60869565217391 | 0.00E+00 |
| 13 | 9.00000000000000 | 9.00000000000000 | 0.00E+00 |
| 14 | 12.39130434782600 | 12.39130434782600 | 0.00E+00 |
| 15 | 9.00000000000000 | 9.00000000000000 | 0.00E+00 |
| 16 | 12.60000000000000 | 12.60000000000000 | 0.00E+00 |
| 17 | 15.00000000000000 | 15.00000000000000 | 0.00E+00 |
| 18 | 6.00000000000000 | 6.00000000000000 | 0.00E+00 |
| 19 | 8.40000000000000 | 8.40000000000000 | 0.00E+00 |
| 20 | 12.00000000000000 | 12.00000000000000 | 0.00E+00 |
| 21 | 8.60869565217391 | 8.60869565217391 | 0.00E+00 |
| 22 | 12.00000000000000 | 12.00000000000000 | 0.00E+00 |
| 23 | 15.39130434782600 | 15.39130434782600 | 0.00E+00 |
| 24 | 12.00000000000000 | 12.00000000000000 | 0.00E+00 |
| 25 | 15.60000000000000 | 15.60000000000000 | 0.00E+00 |
| 26 | 18.00000000000000 | 18.00000000000000 | 0.00E+00 |

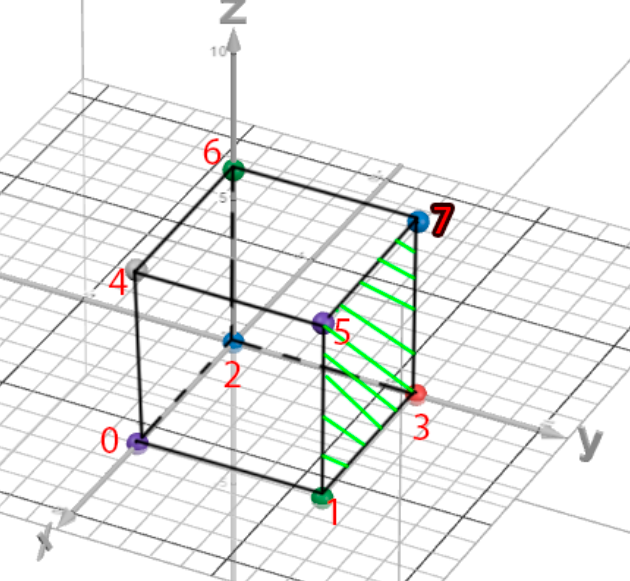
**Тест на учет вторых краевых условий (7 первых краевых и одно второе краевое)**



Входные данные:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FE | XY | Z | inFirstBC | inSecondBC | inThirdBC |
| 1  0 1 2 3 0 1 0 | 4  0 0  6 0  0 6  6 6 | 2  0  6 | 7  0 0  1 6  2 0  3 6  4 0  5 6  6 0 | 1  1 3 5 7 0 1 | 0 |

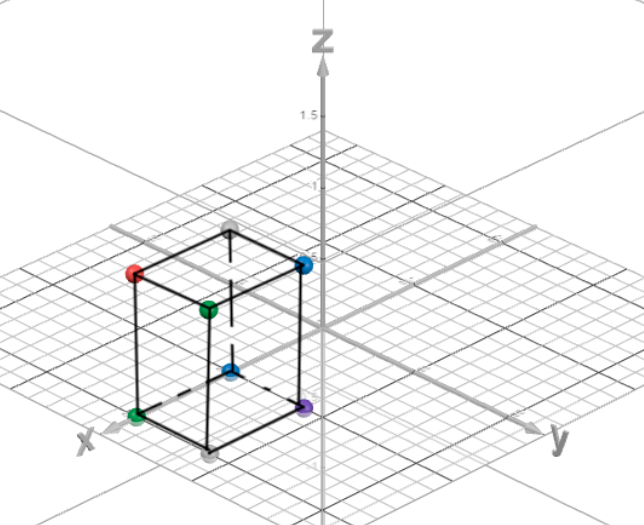
|  |  |  |  |
| --- | --- | --- | --- |
| N | q | q\* | |q\*-q| |
| 0 | 0.000000000000000 | 0 | 0 |
| 1 | 6.000000000000000 | 6 | 0 |
| 2 | 0.000000000000000 | 0 | 0 |
| 3 | 6.000000000000000 | 6 | 0 |
| 4 | 0.000000000000000 | 0 | 0 |
| 5 | 6.000000000000000 | 6 | 0 |
| 6 | 0.000000000000000 | 0 | 0 |
| 7 | 6.000000000000000 | 6 | 0 |



**Произвольный тест на одном конечном элементе**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FE | XY | Z | inFirstBC | inSecondBC | inThirdBC |
| 1  0 1 2 3 0 1 0 | 4  10 0  20 0  10 8  20 8 | 2  0  1 | 0 | 3  0 2 4 6 0 1  2 3 6 7 1 1  4 5 6 7 2 -1 | 3  1 3 5 7 0 10 1  0 1 4 5 1 10 1  0 1 2 3 2 10 -1 |



|  |  |  |  |
| --- | --- | --- | --- |
| N | q | q\* | |q\*-q| |
| 0 | 7.000000000000050 | 7 | 4.9738E-14 |
| 1 | 8.999999999999970 | 9 | 3.0198E-14 |
| 2 | 54.999999999999900 | 55 | 9.9476E-14 |
| 3 | 97.000000000000000 | 97 | 0 |
| 4 | 46.999999999999900 | 47 | 9.9476E-14 |
| 5 | 58.999999999999900 | 59 | 9.9476E-14 |
| 6 | 254.999999999999000 | 255 | 9.9476E-13 |
| 7 | 387.000000000000000 | 387 | 0 |

**Листинг  
mainprog.cpp**

//#include "GRID.h"

#include "TESTS/TEST\_x2/TEST\_x2.h"

#include "string"

string path = "TESTS\\TEST\_x2\\";

string FE = path + "FE.txt";

string XY = path + "XY.txt";

string Z = path + "Z.txt";

string FBC = path + "FirstBC.txt";

string SBC = path + "SecondBC.txt";

string TBC = path + "ThirdBC.txt";

int main()

{

FILE\* inFE, \* inXY, \* inZ, \* inFirstBC, \* inSecondBC, \* inThirdBC;

fopen\_s(&inFE, FE.c\_str(), "r");

fopen\_s(&inXY, XY.c\_str(), "r");

fopen\_s(&inZ, Z.c\_str(), "r");

fopen\_s(&inFirstBC, FBC.c\_str(), "r");

fopen\_s(&inSecondBC, SBC.c\_str(), "r");

fopen\_s(&inThirdBC, TBC.c\_str(), "r");

GridAndSLAE grid;

grid.InputFromFile(inFE, inXY, inZ, inFirstBC, inSecondBC, inThirdBC);

grid.CalculateA\_b();

grid.SecondBoundaryConditions();

grid.ThirdBoundaryConditions();

grid.FirstBoundaryConditions();

grid.OutputDense();

grid.MSGForNonSymMatrixWithLuSqP();

grid.OutputDense();

grid.OutputSolutionQ();

}

Grid.h

#pragma once

#include <iostream>

#include <vector>

#include <math.h>

#include <algorithm>

#define REALOUTD "%.4lf\t"

#define REALOUTDb "%.4lf\n"

using namespace std;

struct Point {

double x;

double y;

};

struct FiniteElement

{

int node1, node2, node3, node4; // x1y1, x2y1, x1y2, x2y2

int bottom, top; //z1, z2

int region;

};

struct FirstBoundary

{

int node;

double ug;

};

struct SecondBoundary

{

int node1, node2, node3, node4; // x1y1, x2y1, x1y2, x2y2

int num\_teta;

int side; // -1 - ���������, 1 - ������� �����

};

struct ThirdBoundary

{

int node1, node2, node3, node4; // x1y1, x2y1, x1y2, x2y2

int num\_ubeta;

double beta;

int side; // 0 - ���������, 1 - ������� �����

};

class GridAndSLAE

{

public:

// NumberOfNodes = NoN

int NoN; // ����� ���-�� �����

int NoN\_xy; // ����� ���-�� ����� � ��������� XY

int NoN\_z; // ����� ���-�� ����� �� ��� Z

int NoN\_fe; // ����� ���-�� �������� ��������� �� ��� Z

int Nof\_FisrtBC;

int Nof\_SecondBC;

int Nof\_ThirdBC;

vector<FiniteElement> fe; // ������ ���������� ���������� �� � �������� ��������

vector<Point> xy; // ������ � ������������ ����� � Oxy

vector<double> z; // ������ � ������������ ����� ����� ��� Z

vector<FirstBoundary> FirstBC;

vector<SecondBoundary> SecondBC;

vector<ThirdBoundary> ThirdBC;

void InputFromFile(FILE\* inFE, FILE\* inXY, FILE\* inZ, FILE\* inFirstBC, FILE\* inSecondBC, FILE\* inThirdBC);

void CalculateA\_b();

void FirstBoundaryConditions();

void SecondBoundaryConditions();

void ThirdBoundaryConditions();

void OutputDense();

void OutputLUDense();

//void SolveSLAE();

void MatrixVectorMultiplication(vector<double>& vectorMult, vector<double>& vectorOut);

void MSGForNonSymMatrixWithLuSqP();

void OutputSolutionQ();

vector<double> x;

vector<double> tmp;

protected:

int maxiter = 10000;

vector<int> ia;

vector<int> ja;

vector<double> al;

vector<double> au;

vector<double> di;

vector<double> b;

vector<double> r;

vector<double> z\_msg;

vector<double> x0;

vector<double> alLU;

vector<double> auLU;

vector<double> diLU;

vector<vector<int>> iaja;

double eps = 1e-20;

int maxIter = 10000;

void GeneratePortrait();

void VectorCopy(vector<double>& from, vector<double>& to);

void CalculateRelativeDiscrepancy(vector<double>& vectorMult, vector<double>& vectorOut);

void SolveForwardLU(vector<double>& lowerTringMat, vector<double>& diag, vector<double>& rightVector, vector<double>& vectorX);

void SolveBackwardLU(vector<double>& upperTringMat, vector<double>& diag, vector<double>& rightVector, vector<double>& vectorX);

void CalculateLUsq();

double CalculateRelativeDiscrepancyWithR(double norm);

double VectorNorm(vector<double>& vector);

double VectorScalarProduction(vector<double>& vector1, vector<double>& vector2);

void MatrixUVectorMultiplicationLU(vector<double>& upperTringMat, vector<double>& diag, vector<double>& vectorMult, vector<double>& vectorOut);

void TransposedMatrixVectorMultiplication(vector<double>& vectorMult, vector<double>& vectorOut);

void CalculateZ\_LUsq(vector<double>& vectorOut);

double CalculateRelativeDiscrepancy(double norm);

void VectorSubtract(vector<double>& first, vector<double>& second, vector<double>& result);

double Gauss3\_Gxy(int i, int j, double b1, double b2, double b3, double b4, double b5, double b6, double a0, double a1, double a2);

double Phi(double e, double n, int i, int j, double b1, double b2, double b3, double b4, double b5, double b6, double a0, double a1, double a2);

double TETA(int number, double x, double y, double z);

double UBETA(int number, double x, double y, double z);

double GAMMA(int number);

double LAMBDA(int number);

double FUN(int number, double x, double y, double z);

void AllocateMemory();

};

**Function.cpp**

#include "GRID.h"

void GridAndSLAE::InputFromFile(FILE\* inFE, FILE\* inXY, FILE\* inZ, FILE\* inFirstBC, FILE\* inSecondBC, FILE\* inThirdBC)

{

// Запись информации об узлах конечных элементах

fscanf\_s(inFE, "%d", &NoN\_fe);

fe.resize(NoN\_fe);

// Мб тут сделать специально сортировку, чтобы номер узлов всегда шли бы на увелечение

for (int i = 0; i < NoN\_fe; i++)

{

fscanf\_s(inFE, "%d", &fe[i].node1);

fscanf\_s(inFE, "%d", &fe[i].node2);

fscanf\_s(inFE, "%d", &fe[i].node3);

fscanf\_s(inFE, "%d", &fe[i].node4);

fscanf\_s(inFE, "%d", &fe[i].bottom);

fscanf\_s(inFE, "%d", &fe[i].top);

fscanf\_s(inFE, "%d", &fe[i].region);

}

// Запись информации об точках в Oxy

fscanf\_s(inXY, "%d", &NoN\_xy);

xy.resize(NoN\_xy);

for (int i = 0; i < NoN\_xy; i++)

{

fscanf\_s(inXY, "%lf", &xy[i].x);

fscanf\_s(inXY, "%lf", &xy[i].y);

}

// Запись информации об узлах вдоль оси Z

fscanf\_s(inZ, "%d", &NoN\_z);

z.resize(NoN\_z);

for (int i = 0; i < NoN\_z; i++)

{

fscanf\_s(inZ, "%lf", &z[i]);

}

NoN = NoN\_xy \* NoN\_z;

fscanf\_s(inFirstBC, "%d", &Nof\_FisrtBC);

FirstBC.resize(Nof\_FisrtBC);

for (int i = 0; i < Nof\_FisrtBC; i++)

{

fscanf\_s(inFirstBC, "%d", &FirstBC[i].node);

fscanf\_s(inFirstBC, "%lf", &FirstBC[i].ug);

}

fscanf\_s(inSecondBC, "%d", &Nof\_SecondBC);

SecondBC.resize(Nof\_SecondBC);

for (int i = 0; i < Nof\_SecondBC; i++)

{

fscanf\_s(inSecondBC, "%d", &SecondBC[i].node1);

fscanf\_s(inSecondBC, "%d", &SecondBC[i].node2);

fscanf\_s(inSecondBC, "%d", &SecondBC[i].node3);

fscanf\_s(inSecondBC, "%d", &SecondBC[i].node4);

fscanf\_s(inSecondBC, "%d", &SecondBC[i].num\_teta);

fscanf\_s(inSecondBC, "%d", &SecondBC[i].side);

}

fscanf\_s(inThirdBC, "%d", &Nof\_ThirdBC);

ThirdBC.resize(Nof\_ThirdBC);

for (int i = 0; i < Nof\_ThirdBC; i++)

{

fscanf\_s(inThirdBC, "%d", &ThirdBC[i].node1);

fscanf\_s(inThirdBC, "%d", &ThirdBC[i].node2);

fscanf\_s(inThirdBC, "%d", &ThirdBC[i].node3);

fscanf\_s(inThirdBC, "%d", &ThirdBC[i].node4);

fscanf\_s(inThirdBC, "%d", &ThirdBC[i].num\_ubeta);

fscanf\_s(inThirdBC, "%lf", &ThirdBC[i].beta);

fscanf\_s(inThirdBC, "%d", &ThirdBC[i].side);

}

}

double sign(double x)

{

if (x > 0)

return 1;

else if (x < 0)

return -1;

else

return 0;

}

void GridAndSLAE::AllocateMemory()

{

di.resize(NoN);

b.resize(NoN);

x.resize(NoN);

al.resize(ja.size());

au.resize(ja.size());

alLU.resize(al.size());

auLU.resize(au.size());

diLU.resize(NoN);

r.resize(NoN);

z.resize(NoN);

x0.resize(NoN);

tmp.resize(NoN);

iaja.resize(NoN);

}

void GridAndSLAE::CalculateA\_b()

{

GeneratePortrait();

AllocateMemory();

vector <int> nodes\_global;

nodes\_global.resize(8);

vector<double> b\_local;

b\_local.resize(8);

// нужен будет потом, когда будут разные функции на разных КЭ

vector<double> f\_local;

f\_local.resize(8);

//

// Хранится только нижний треугольник

//

// Локальные матрицы массы

//

vector<vector<double>> M0 = { {4}, {2, 4}, {2, 1, 4}, {1, 2, 2, 4} };

vector<vector<double>> M1 = { {2}, {2, 6}, {1, 1, 2}, {1, 2, 3, 6} };

vector<vector<double>> M2 = { {2}, {1, 2}, {2, 1, 6}, {1, 2, 3, 6} };

vector<vector<double>> Mxy;

Mxy.resize(4);

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

Mxy[i].resize(i + 1);

vector<vector<double>> Mz\_0 = { {2}, {1, 2} };

vector<vector<double>> Mz;

Mz.resize(2);

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

Mz[i].resize(i + 1);

vector<vector<double>> M\_local;

M\_local.resize(8);

for (int i = 0; i < 8; i++)

M\_local[i].resize(i + 1);

//

// Локальые матрицы жесткости

//

vector<vector<double>> Gxy;

Gxy.resize(4);

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

Gxy[i].resize(i + 1);

vector<vector<double>> Gz\_0 = { {1}, {-1, 1} };

vector<vector<double>> Gz;

Gz.resize(2);

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

Gz[i].resize(i + 1);

vector<vector<double>> G\_local;

G\_local.resize(8);

for (int i = 0; i < 8; i++)

G\_local[i].resize(i + 1);

//

// Сборка локальных матриц

//

for (int curFE = 0; curFE < NoN\_fe; )

{

nodes\_global[0] = fe[curFE].node1 + fe[curFE].bottom \* NoN\_xy;

nodes\_global[1] = fe[curFE].node2 + fe[curFE].bottom \* NoN\_xy;

nodes\_global[2] = fe[curFE].node3 + fe[curFE].bottom \* NoN\_xy;

nodes\_global[3] = fe[curFE].node4 + fe[curFE].bottom \* NoN\_xy;

int region\_cur = fe[curFE].region;

double x1 = xy[fe[curFE].node1].x;

double y1 = xy[fe[curFE].node1].y;

double x2 = xy[fe[curFE].node2].x;

double y2 = xy[fe[curFE].node2].y;

double x3 = xy[fe[curFE].node3].x;

double y3 = xy[fe[curFE].node3].y;

double x4 = xy[fe[curFE].node4].x;

double y4 = xy[fe[curFE].node4].y;

double a0 = ((x2 - x1) \* (y3 - y1) - (y2 - y1) \* (x3 - x1));

double a1 = ((x2 - x1) \* (y4 - y3) - (y2 - y1) \* (x4 - x3));

double a2 = ((y3 - y1) \* (x4 - x2) - (x3 - x1) \* (y4 - y2));

for (int i = 0; i < 4; i++) // Формирование локальной матрицы массы для Oxy

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

Mxy[i][j] = sign(a0) \* (a0 / 36. \* M0[i][j] + a1 / 72. \* M1[i][j] + a2 / 72. \* M2[i][j]);

double b1 = x3 - x1;

double b2 = x2 - x1;

double b5 = x1 - x2 - x3 + x4;

double b3 = y3 - y1;

double b4 = y2 - y1;

double b6 = y1 - y2 - y3 + y4;

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

for (int j = 0; j <= i; j++) // Формирование локальной жесткости для Oxy

Gxy[i][j] = Gauss3\_Gxy(i, j, b1, b2, b3, b4, b5, b6, a0, a1, a2);

// Короче, поменял обход области (последовательнсти КЭ). До этого было так, я обходил сначала все конечные элементы,

// у которых высота 0-1, потом 1-2, потом 2-3, и тд

// Но я тут заметил, что по сути КЭ, которые "стоят" в одном столбце имеют одинаковое основание

// Поэтому достотаточно один раз посчитать Mxy, Gxy, а Mz, Gz, пересчитывать на каждом уровне

for (int lvl = 0; lvl < NoN\_z - 1 and curFE < NoN\_fe; lvl++, curFE++) // Здесь как раз за это и отвечает этот цикл

{

nodes\_global[4] = fe[curFE].node1 + fe[curFE].top \* NoN\_xy;

nodes\_global[5] = fe[curFE].node2 + fe[curFE].top \* NoN\_xy;

nodes\_global[6] = fe[curFE].node3 + fe[curFE].top \* NoN\_xy;

nodes\_global[7] = fe[curFE].node4 + fe[curFE].top \* NoN\_xy;

double z1 = z[fe[curFE].bottom];

double z2 = z[fe[curFE].top];

double h\_z = z2 - z1;

for (int i = 0; i < 2; i++) // Формирование локальной матрицы массы для оси Z

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

Mz[i][j] = h\_z / 6. \* Mz\_0[i][j];

for (int i = 0; i < 8; i++) // Сборка локальной матрицы массы

{

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

{

int mu\_i = i % 4;

int mu\_j = j % 4;

if (mu\_i < mu\_j)

swap(mu\_i, mu\_j);

int nu\_i = (int)(i / 4.);

int nu\_j = (int)(j / 4.);

/\*if (nu\_i < nu\_j)

swap(nu\_i, nu\_j);\*/

M\_local[i][j] = Mxy[mu\_i][mu\_j] \* Mz[nu\_i][nu\_j];

}

}

for (int i = 0; i < 2; i++) // Формирование локальной жесткости для оси Z

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

Gz[i][j] = Gz\_0[i][j] / h\_z;

for (int i = 0; i < 8; i++)

{

for (int j = 0; j <= i; j++) // Сборка локальной матрицы жесткости

{

int mu\_i = i % 4;

int mu\_j = j % 4;

if (mu\_i < mu\_j)

swap(mu\_i, mu\_j);

int nu\_i = (int)(i / 4.);

int nu\_j = (int)(j / 4.);

G\_local[i][j] = Gxy[mu\_i][mu\_j] \* Mz[nu\_i][nu\_j] + Mxy[mu\_i][mu\_j] \* Gz[nu\_i][nu\_j];

}

}

f\_local[0] = FUN(region\_cur, x1, y1, z1); // Вот здесь высока вероятность ошибки

f\_local[1] = FUN(region\_cur, x2, y2, z1); // так как я неявно считаю значения функции

f\_local[2] = FUN(region\_cur, x3, y3, z1); // в лок. узлах. Если бы нумерации узлов в КЭ поменялась

f\_local[3] = FUN(region\_cur, x4, y4, z1); // то может выскочить ошибка

f\_local[4] = FUN(region\_cur, x1, y1, z2);

f\_local[5] = FUN(region\_cur, x2, y2, z2);

f\_local[6] = FUN(region\_cur, x3, y3, z2);

f\_local[7] = FUN(region\_cur, x4, y4, z2);

for (int i = 0; i < 8; i++) //Нужно потом более оптимально умножение сделать

{

double sum = 0;

for (int j = 0; j < 8; j++)

{

if (i >= j)

sum += f\_local[j] \* M\_local[i][j];

else

sum += f\_local[j] \* M\_local[j][i];

}

b\_local[i] = sum;

}

double Aij;

double lambda = LAMBDA(region\_cur);

double gamma = GAMMA(region\_cur);

for (int i = 0; i < 8; i++)

{

Aij = gamma \* M\_local[i][i] + lambda \* G\_local[i][i];

di[nodes\_global[i]] += Aij;

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

{

for (int k = ia[nodes\_global[i]]; k < ia[nodes\_global[i]+1]; k++)

{

if (ja[k] == nodes\_global[j])

{

Aij = gamma \* M\_local[i][j] + lambda \* G\_local[i][j];

al[k] += Aij;

au[k] += Aij;

}

}

}

}

for (int i = 0; i < 8; i++)

b[nodes\_global[i]] += b\_local[i];

nodes\_global[0] = nodes\_global[4];

nodes\_global[1] = nodes\_global[5];

nodes\_global[2] = nodes\_global[6];

nodes\_global[3] = nodes\_global[7];

}

}

} void GridAndSLAE::SecondBoundaryConditions()

{

if (Nof\_SecondBC != 0)

{

vector<double> bS2\_local(4, 0);

vector<int> nodes\_global(4, 0);

vector<double> teta\_local(4, 0);

//

// Матрицы для боковой грани

//

vector<vector<double>> MXorY\_0{ {2}, {1, 2} };

vector<vector<double>> MXorY;

MXorY.resize(2);

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

MXorY[i].resize(i + 1);

vector<vector<double>> Mz\_0 = { {2}, {1, 2} };

vector<vector<double>> Mz;

Mz.resize(2);

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

Mz[i].resize(i + 1);

vector<vector<double>> M\_XYorXZorYZ;

M\_XYorXZorYZ.resize(4);

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

M\_XYorXZorYZ[i].resize(i + 1);

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

//

// Матрицы для основания

//

vector<vector<double>> M0 = { {4}, {2, 4}, {2, 1, 4}, {1, 2, 2, 4} };

vector<vector<double>> M1 = { {2}, {2, 6}, {1, 1, 2}, {1, 2, 3, 6} };

vector<vector<double>> M2 = { {2}, {1, 2}, {2, 1, 6}, {1, 2, 3, 6} };

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

for (int curSecondBC = 0; curSecondBC < Nof\_SecondBC; curSecondBC++)

{

nodes\_global[0] = SecondBC[curSecondBC].node1;

nodes\_global[1] = SecondBC[curSecondBC].node2;

nodes\_global[2] = SecondBC[curSecondBC].node3;

nodes\_global[3] = SecondBC[curSecondBC].node4;

int num\_teta\_local = SecondBC[curSecondBC].num\_teta;

// Определить что это - боковая грань или основание?

if (SecondBC[curSecondBC].side == 1)

{

// Значит боковая грань

// Будем обозначать матрицу массы для x/y как M\_XorY (смотря паралельно какой оси находится грань)

// Будем искать матрицу массы для краевого условия как M\_XorY \* Mz

double x1 = xy[nodes\_global[0] % NoN\_xy].x;

double x2 = xy[nodes\_global[1] % NoN\_xy].x;

double y1 = xy[nodes\_global[0] % NoN\_xy].y;

double y2 = xy[nodes\_global[1] % NoN\_xy].y;

double h\_XorY = 0;

div\_t result = div(nodes\_global[0], NoN\_xy);

double z1 = z[result.quot];

double z2 = z[result.quot + 1];

double h\_z = z2 - z1;

teta\_local[0] = TETA(num\_teta\_local, x1, y1, z1);

teta\_local[1] = TETA(num\_teta\_local, x2, y2, z1);

teta\_local[2] = TETA(num\_teta\_local, x1, y1, z2);

teta\_local[3] = TETA(num\_teta\_local, x2, y2, z2);

if ((nodes\_global[0] + 1) == nodes\_global[1]) // значит боковая грань поралельна Ox. Я предполагаю что нумерация у узлов корректная

h\_XorY = x2 - x1;

else if ((nodes\_global[0] % NoN\_xy) == (nodes\_global[2] % NoN\_xy)) // значит боковая грань поралельна Oy.

h\_XorY = y2 - y1;

else

throw "1 Словил какую-ту херь в 2ом краевом\n";

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

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

MXorY[i][j] = h\_XorY / 6. \* MXorY\_0[i][j];

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

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

Mz[i][j] = h\_z / 6. \* Mz\_0[i][j];

// стр 234 кирпича

// Вообще можно сделать сразу матрицу MXZorYZ размером 4х4 (стр 233 5.20), мб потом переделаю

// а пока тупо скатаю формулы (при учете того, что у меня нижний треугольник, а в кирпиче верхний)

// но лучше перепроверить все равно

// UPD: на стр 331 и 333 есть

M\_XYorXZorYZ[0][0] = MXorY[0][0] \* Mz[0][0];

M\_XYorXZorYZ[1][0] = MXorY[1][0] \* Mz[0][0];

M\_XYorXZorYZ[2][0] = MXorY[0][0] \* Mz[1][0];

M\_XYorXZorYZ[3][0] = MXorY[1][0] \* Mz[1][0];

M\_XYorXZorYZ[1][1] = MXorY[1][1] \* Mz[0][0];

M\_XYorXZorYZ[2][1] = MXorY[1][0] \* Mz[1][0];

M\_XYorXZorYZ[3][1] = MXorY[1][1] \* Mz[1][0];

M\_XYorXZorYZ[2][2] = MXorY[0][0] \* Mz[1][1];

M\_XYorXZorYZ[3][2] = MXorY[1][0] \* Mz[1][1];

M\_XYorXZorYZ[3][3] = MXorY[1][1] \* Mz[1][1];

}

else if (SecondBC[curSecondBC].side == -1)

{

// Иначе основание

double x1 = xy[nodes\_global[0] % NoN\_xy].x;

double y1 = xy[nodes\_global[0] % NoN\_xy].y;

double x2 = xy[nodes\_global[1] % NoN\_xy].x;

double y2 = xy[nodes\_global[1] % NoN\_xy].y;

double x3 = xy[nodes\_global[2] % NoN\_xy].x;

double y3 = xy[nodes\_global[2] % NoN\_xy].y;

double x4 = xy[nodes\_global[3] % NoN\_xy].x;

double y4 = xy[nodes\_global[3] % NoN\_xy].y;

div\_t result = div(nodes\_global[0], NoN\_xy);

double zlvl = z[result.quot];

double a0 = ((x2 - x1) \* (y3 - y1) - (y2 - y1) \* (x3 - x1));

double a1 = ((x2 - x1) \* (y4 - y3) - (y2 - y1) \* (x4 - x3));

double a2 = ((y3 - y1) \* (x4 - x2) - (x3 - x1) \* (y4 - y2));

teta\_local[0] = TETA(num\_teta\_local, x1, y1, zlvl);

teta\_local[1] = TETA(num\_teta\_local, x2, y2, zlvl);

teta\_local[2] = TETA(num\_teta\_local, x3, y3, zlvl);

teta\_local[3] = TETA(num\_teta\_local, x4, y4, zlvl);

for (int i = 0; i < 4; i++) // Формирование локальной матрицы массы для Oxy

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

M\_XYorXZorYZ[i][j] = sign(a0) \* (a0 / 36. \* M0[i][j] + a1 / 72. \* M1[i][j] + a2 / 72. \* M2[i][j]);

}

else

throw "2 Словил какую-ту херь в 2ом краевом\n";

for (int i = 0; i < 4; i++) //Нужно потом более оптимально умножение сделать

{

double sum = 0;

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

{

if (i >= j)

sum += teta\_local[j] \* M\_XYorXZorYZ[i][j];

else

sum += teta\_local[j] \* M\_XYorXZorYZ[j][i];

}

bS2\_local[i] = sum;

}

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

b[nodes\_global[i]] += bS2\_local[i];

}

}

}

void GridAndSLAE::ThirdBoundaryConditions() // ctrl+c -> ctrl+v из SecondBoundaryConditions.

{ // Так что если и есть ошибки, то они могли всплыть из-за невдумчивого копирования

if (Nof\_ThirdBC != 0)

{

vector<double> bS3\_local(4, 0);

vector<int> nodes\_global(4, 0);

vector<double> ubeta\_local(4, 0);

//

// Матрицы для боковой грани

//

vector<vector<double>> MXorY\_0{ {2}, {1, 2} };

vector<vector<double>> MXorY;

MXorY.resize(2);

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

MXorY[i].resize(i + 1);

vector<vector<double>> Mz\_0 = { {2}, {1, 2} };

vector<vector<double>> Mz;

Mz.resize(2);

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

Mz[i].resize(i + 1);

vector<vector<double>> M\_XYorXZorYZ;

M\_XYorXZorYZ.resize(4);

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

M\_XYorXZorYZ[i].resize(i + 1);

//

// Матрицы для основания

//

vector<vector<double>> M0 = { {4}, {2, 4}, {2, 1, 4}, {1, 2, 2, 4} };

vector<vector<double>> M1 = { {2}, {2, 6}, {1, 1, 2}, {1, 2, 3, 6} };

vector<vector<double>> M2 = { {2}, {1, 2}, {2, 1, 6}, {1, 2, 3, 6} };

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

for (int curThirdBC = 0; curThirdBC < Nof\_ThirdBC; curThirdBC++)

{

nodes\_global[0] = ThirdBC[curThirdBC].node1;

nodes\_global[1] = ThirdBC[curThirdBC].node2;

nodes\_global[2] = ThirdBC[curThirdBC].node3;

nodes\_global[3] = ThirdBC[curThirdBC].node4;

int num\_ubeta\_local = ThirdBC[curThirdBC].num\_ubeta;

// Определить что это - боковая грань или основание?

if (ThirdBC[curThirdBC].side == 1)

{

// Значит боковая грань

// Будем обозначать матрицу массы для x/y как M\_XorY (смотря паралельно какой оси находится грань)

// Будем искать матрицу массы для краевого условия как M\_XorY \* Mz

double x1 = xy[nodes\_global[0] % NoN\_xy].x;

double x2 = xy[nodes\_global[1] % NoN\_xy].x;

double y1 = xy[nodes\_global[0] % NoN\_xy].y;

double y2 = xy[nodes\_global[1] % NoN\_xy].y;

double h\_XorY = 0;

div\_t result = div(nodes\_global[0], NoN\_xy);

double z1 = z[result.quot];

double z2 = z[result.quot + 1];

double h\_z = z2 - z1;

ubeta\_local[0] = UBETA(num\_ubeta\_local, x1, y1, z1);

ubeta\_local[1] = UBETA(num\_ubeta\_local, x2, y2, z1);

ubeta\_local[2] = UBETA(num\_ubeta\_local, x1, y1, z2);

ubeta\_local[3] = UBETA(num\_ubeta\_local, x2, y2, z2);

if ((nodes\_global[0] + 1) == nodes\_global[1]) // значит боковая грань поралельна Ox. Я предполагаю что нумерация у узлов корректная

h\_XorY = x2 - x1;

else if ((nodes\_global[0] % NoN\_xy) == (nodes\_global[2] % NoN\_xy)) // значит боковая грань поралельна Oy.

h\_XorY = y2 - y1;

else

throw "1 Словил какую-ту херь в 3ем краевом\n";

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

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

MXorY[i][j] = h\_XorY / 6. \* MXorY\_0[i][j];

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

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

Mz[i][j] = h\_z / 6. \* Mz\_0[i][j];

// стр 234 кирпича

// Вообще можно сделать сразу матрицу MXZorYZ размером 4х4 (стр 233 5.20), мб потом переделаю

// а пока тупо скатаю формулы (при учете того, что у меня нижний треугольник, а в кирпиче верхний)

// но лучше перепроверить все равно

M\_XYorXZorYZ[0][0] = MXorY[0][0] \* Mz[0][0];

M\_XYorXZorYZ[1][0] = MXorY[1][0] \* Mz[0][0];

M\_XYorXZorYZ[2][0] = MXorY[0][0] \* Mz[1][0];

M\_XYorXZorYZ[3][0] = MXorY[1][0] \* Mz[1][0];

M\_XYorXZorYZ[1][1] = MXorY[1][1] \* Mz[0][0];

M\_XYorXZorYZ[2][1] = MXorY[1][0] \* Mz[1][0];

M\_XYorXZorYZ[3][1] = MXorY[1][1] \* Mz[1][0];

M\_XYorXZorYZ[2][2] = MXorY[0][0] \* Mz[1][1];

M\_XYorXZorYZ[3][2] = MXorY[1][0] \* Mz[1][1];

M\_XYorXZorYZ[3][3] = MXorY[1][1] \* Mz[1][1];

}

else if (ThirdBC[curThirdBC].side == -1)

{

// Иначе основание

double x1 = xy[nodes\_global[0] % NoN\_xy].x;

double y1 = xy[nodes\_global[0] % NoN\_xy].y;

double x2 = xy[nodes\_global[1] % NoN\_xy].x;

double y2 = xy[nodes\_global[1] % NoN\_xy].y;

double x3 = xy[nodes\_global[2] % NoN\_xy].x;

double y3 = xy[nodes\_global[2] % NoN\_xy].y;

double x4 = xy[nodes\_global[3] % NoN\_xy].x;

double y4 = xy[nodes\_global[3] % NoN\_xy].y;

div\_t result = div(nodes\_global[0], NoN\_xy);

double zlvl = z[result.quot];

double a0 = ((x2 - x1) \* (y3 - y1) - (y2 - y1) \* (x3 - x1));

double a1 = ((x2 - x1) \* (y4 - y3) - (y2 - y1) \* (x4 - x3));

double a2 = ((y3 - y1) \* (x4 - x2) - (x3 - x1) \* (y4 - y2));

ubeta\_local[0] = UBETA(num\_ubeta\_local, x1, y1, zlvl);

ubeta\_local[1] = UBETA(num\_ubeta\_local, x2, y2, zlvl);

ubeta\_local[2] = UBETA(num\_ubeta\_local, x3, y3, zlvl);

ubeta\_local[3] = UBETA(num\_ubeta\_local, x4, y4, zlvl);

for (int i = 0; i < 4; i++) // Формирование локальной матрицы массы для Oxy

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

M\_XYorXZorYZ[i][j] = sign(a0) \* (a0 / 36. \* M0[i][j] + a1 / 72. \* M1[i][j] + a2 / 72. \* M2[i][j]);

}

else

throw "2 Словил какую-ту херь в 3ем краевом\n";

// Добавка в глоб матрицу

double beta = ThirdBC[curThirdBC].beta;

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

{

double AijS3;

AijS3 = beta \* M\_XYorXZorYZ[i][i];

di[nodes\_global[i]] += AijS3;

for (int j = i - 1; j >= 0; j--)

for (int k = ia[nodes\_global[i]]; k < ia[nodes\_global[i] + 1]; k++)

if (ja[k] == nodes\_global[j])

{

AijS3 = beta \* M\_XYorXZorYZ[i][j];

al[k] += AijS3;

au[k] += AijS3;

}

}

for (int i = 0; i < 4; i++) //Нужно потом более оптимально умножение сделать

{

double sum = 0;

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

{

if (i > j)

sum += ubeta\_local[j] \* M\_XYorXZorYZ[i][j];

else

sum += ubeta\_local[j] \* M\_XYorXZorYZ[j][i];

}

bS3\_local[i] = sum;

}

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

b[nodes\_global[i]] += beta \* bS3\_local[i];

}

}

}

void GridAndSLAE::FirstBoundaryConditions()

{

for (int curFisrtBC = 0; curFisrtBC < Nof\_FisrtBC; curFisrtBC++)

{

int node = FirstBC[curFisrtBC].node;

double ug = FirstBC[curFisrtBC].ug;

di[node] = 1;

b[node] = ug;

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

{

al[k] = 0;

}

for (int curNode = node + 1; curNode < NoN; curNode++)

{

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

{

int j = ja[k];

if (ja[k] == node)

{

au[k] = 0;

}

}

}

//OutputDense();

}

}

void GridAndSLAE::GeneratePortrait()

{

iaja.resize(NoN);

vector<int> nodes\_local(8, 0);

for (int curFE = 0; curFE < NoN\_fe; curFE++)

{

nodes\_local[0] = fe[curFE].node1 + fe[curFE].bottom \* NoN\_xy;

nodes\_local[1] = fe[curFE].node2 + fe[curFE].bottom \* NoN\_xy;

nodes\_local[2] = fe[curFE].node3 + fe[curFE].bottom \* NoN\_xy;

nodes\_local[3] = fe[curFE].node4 + fe[curFE].bottom \* NoN\_xy;

nodes\_local[4] = fe[curFE].node1 + fe[curFE].top \* NoN\_xy;

nodes\_local[5] = fe[curFE].node2 + fe[curFE].top \* NoN\_xy;

nodes\_local[6] = fe[curFE].node3 + fe[curFE].top \* NoN\_xy;

nodes\_local[7] = fe[curFE].node4 + fe[curFE].top \* NoN\_xy;

for (int i = 0; i < 8; i++)

{

int node\_i = nodes\_local[i];

for (int j = 0; j < 8; j++)

{

int node\_j = nodes\_local[j];

if (node\_i > node\_j)

{

int flag = 0;

for (int k = 0; k < iaja[node\_i].size() and flag != 1; k++)

{

if (iaja[node\_i][k] == node\_j)

{

flag = 1;

continue;

}

}

if (!flag)

iaja[node\_i].push\_back(node\_j);

}

}

}

}

// Сортировка

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

sort(iaja[i].begin(), iaja[i].end());

ia.resize(NoN + 1);

ia[0] = 0;

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

{

int size = iaja[i].size();

ia[i + 1] = ia[i] + size;

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

ja.push\_back(iaja[i][j]);

}

}

double GridAndSLAE::Phi(double e, double n, int i, int j, double b1, double b2, double b3, double b4, double b5, double b6, double a0, double a1, double a2) // Ммммммм список аргументов

{

// Формула интеграла

// SS (phi\_i\_1 \* phi\_j\_1 + phi\_i\_2 \* phi\_j\_2)/J dedn

double J = 0;

double value = 0;

double phi\_i\_1 = 0, phi\_j\_1 = 0, phi\_i\_2 = 0, phi\_j\_2 = 0;

switch (i) // Не стал делать для всяких "b6\*e + b3" отдельные переменные, чтобы легче можно было формулам проверять

{

case 0:

// phi = (1 - e)\*(1 - n)

// d(phi)/de = -1 + n

// d(phi)/dn = -1 + e

phi\_i\_1 = (n - 1.) \* (b6 \* e + b3) - (e - 1.) \* (b6 \* n + b4);

phi\_i\_2 = (e - 1.) \* (b5 \* n + b2) - (n - 1.) \* (b5 \* e + b1);

break;

case 1:

// phi = e\*(1 - n)

// d(phi)/de = 1 - n

// d(phi)/dn = -e

phi\_i\_1 = (1. - n) \* (b6 \* e + b3) - (-e) \* (b6 \* n + b4);

phi\_i\_2 = (-e) \* (b5 \* n + b2) - (1. - n) \* (b5 \* e + b1);

break;

case 2:

// phi = (1 - e)\*n

// d(phi)/de = -n

// d(phi)/dn = 1 - e

phi\_i\_1 = (-n) \* (b6 \* e + b3) - (1. - e) \* (b6 \* n + b4);

phi\_i\_2 = (1. - e) \* (b5 \* n + b2) - (-n) \* (b5 \* e + b1);

break;

case 3:

// phi = e\*n

// d(phi)/de = n

// d(phi)/dn = e

phi\_i\_1 = n \* (b6 \* e + b3) - e \* (b6 \* n + b4);

phi\_i\_2 = e \* (b5 \* n + b2) - n \* (b5 \* e + b1);

break;

}

switch (j)

{

case 0:

// phi = (1 - e)\*(1 - n)

// d(phi)/de = -1 + n

// d(phi)/dn = -1 + e

phi\_j\_1 = (n - 1.) \* (b6 \* e + b3) - (e - 1.) \* (b6 \* n + b4);

phi\_j\_2 = (e - 1.) \* (b5 \* n + b2) - (n - 1.) \* (b5 \* e + b1);

break;

case 1:

// phi = e\*(1 - n)

// d(phi)/de = 1 - n

// d(phi)/dn = -e

phi\_j\_1 = (1. - n) \* (b6 \* e + b3) - (-e) \* (b6 \* n + b4);

phi\_j\_2 = (-e) \* (b5 \* n + b2) - (1. - n) \* (b5 \* e + b1);

break;

case 2:

// phi = (1 - e)\*n

// d(phi)/de = -n

// d(phi)/dn = 1 - e

phi\_j\_1 = (-n) \* (b6 \* e + b3) - (1. - e) \* (b6 \* n + b4);

phi\_j\_2 = (1. - e) \* (b5 \* n + b2) - (-n) \* (b5 \* e + b1);

break;

case 3:

// phi = e\*n

// d(phi)/de = n

// d(phi)/dn = e

phi\_j\_1 = n \* (b6 \* e + b3) - e \* (b6 \* n + b4);

phi\_j\_2 = e \* (b5 \* n + b2) - n \* (b5 \* e + b1);

break;

}

J = a0 + a1 \* e + a2 \* n;

value = (phi\_i\_1 \* phi\_j\_1 + phi\_i\_2 \* phi\_j\_2)/J;

return value;

}

double GridAndSLAE::Gauss3\_Gxy(int i, int j, double b1, double b2, double b3, double b4, double b5, double b6, double a0, double a1, double a2)

{

double sum = 0;

double tauK = 0, tauL = 0;

double tK = 0, tL = 0;

double q = sqrt(3. / 5.);

sum += (8. / 9.) \* (8. / 9.) \* Phi((1. + 0) / 2., (1. + 0) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (8. / 9.) \* (5. / 9.) \* Phi((1. + 0) / 2., (1. + q) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (8. / 9.) \* (5. / 9.) \* Phi((1. + 0) / 2., (1. - q) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (5. / 9.) \* (8. / 9.) \* Phi((1. + q) / 2., (1. + 0) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (5. / 9.) \* (5. / 9.) \* Phi((1. + q) / 2., (1. + q) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (5. / 9.) \* (5. / 9.) \* Phi((1. + q) / 2., (1. - q) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (5. / 9.) \* (8. / 9.) \* Phi((1. - q) / 2., (1. + 0) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (5. / 9.) \* (5. / 9.) \* Phi((1. - q) / 2., (1. + q) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum += (5. / 9.) \* (5. / 9.) \* Phi((1. - q) / 2., (1. - q) / 2., /\*\*/ i, j, /\*\*/ b1, b2, b3, b4, b5, b6, a0, a1, a2);

sum /= 4.;

return sum;

}

SlaeSolver.cpp

#include "GRID.h"

void GridAndSLAE::OutputDense()

{

int flagfound = 0;

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

{

int k = ia[i + 1] - ia[i];

if (k == 0)

{

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

{

printf(REALOUTD, 0.0);

}

}

else

{

int lastj = 0;

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

{

for (int p = lastj; p < ja[j]; p++)

{

printf(REALOUTD, 0.0);

}

printf(REALOUTD, al[j]); // ��� ������� 216

//printf(REALOUTD, 216 \* al[j]); // ��� ������� 216

lastj = ja[j] + 1;

}

for (int j = lastj; j < i; j++)

{

printf(REALOUTD, 0.0);

}

}

printf(REALOUTD, di[i]); // ��� ������� 216

//printf(REALOUTD, 216 \* di[i]); // ��� ������� 216

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

{

k = ia[j + 1] - ia[j];

if (k == 0) {

printf(REALOUTD, 0.0);

}

else

{

flagfound = 0;

for (k = ia[j]; k < ia[j + 1]; k++)

{

if (ja[k] == i)

{

printf(REALOUTD, au[k]); // ��� ������� 216

//printf(REALOUTD, 216 \* au[k]); // ��� ������� 216

flagfound = 1;

break;

}

}

if (flagfound == 0)

printf(REALOUTD, 0.0);

}

}

printf("| %.4lf\n", b[i]); // ��� ������� 216

//printf("| %.4lf\n", 216 \* b[i]); // ��� ������� 216

printf("\n");

}

printf("\n");

}

void GridAndSLAE::OutputLUDense()

{

int flagfound = 0;

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

{

int k = ia[i + 1] - ia[i];

if (k == 0)

{

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

{

printf(REALOUTD, 0.0);

}

}

else

{

int lastj = 0;

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

{

for (int p = lastj; p < ja[j]; p++)

{

printf(REALOUTD, 0.0);

}

printf(REALOUTD, alLU[j]);

lastj = ja[j] + 1;

}

for (int j = lastj; j < i; j++)

{

printf(REALOUTD, 0.0);

}

}

printf(REALOUTD, diLU[i]);

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

{

k = ia[j + 1] - ia[j];

if (k == 0) {

printf(REALOUTD, 0.0);

}

else

{

flagfound = 0;

for (k = ia[j]; k < ia[j + 1]; k++)

{

if (ja[k] == i)

{

printf(REALOUTD, auLU[k]);

flagfound = 1;

break;

}

}

if (flagfound == 0)

printf(REALOUTD, 0.0);

}

}

printf("\n");

}

printf("\n");

}

void GridAndSLAE::VectorCopy(vector<double>& from, vector<double>& to)

{

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

to[i] = from[i];

}

void GridAndSLAE::CalculateRelativeDiscrepancy(vector<double>& vectorMult, vector<double>& vectorOut)

{

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

{

vectorOut[i] = di[i] \* vectorMult[i];

vectorOut[i] = b[i] - vectorOut[i];

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

{

int j = ja[k];

vectorOut[i] += al[k] \* vectorMult[j];

vectorOut[j] += au[k] \* vectorMult[i];

}

}

}

void GridAndSLAE::MatrixVectorMultiplication(vector<double>& vectorMult, vector<double>& vectorOut)

{

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

{

vectorOut[i] = di[i] \* vectorMult[i];

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

{

int j = ja[k];

vectorOut[i] += al[k] \* vectorMult[j];

vectorOut[j] += au[k] \* vectorMult[i];

}

}

}

void GridAndSLAE::SolveForwardLU(vector<double>& lowerTringMat, vector<double>& diag, vector<double>& rightVector, vector<double>& vectorX) {

for (int i0, i1, i = 0; i < NoN; i++)

{

double sum = 0;

i0 = ia[i];

i1 = ia[i + 1];

//int j = i - (i1 - i0);

for (int k = i0; k < i1; k++)

{

int j = ja[k];

sum += lowerTringMat[k] \* vectorX[j];

}

vectorX[i] = (rightVector[i] - sum) / diag[i];

}

}

void GridAndSLAE::SolveBackwardLU(vector<double>& upperTringMat, vector<double>& diag, vector<double>& rightVector, vector<double>& vectorX) {

for (int i0, i1, i = NoN - 1; i >= 0; i--)

{

i0 = ia[i];

i1 = ia[i + 1];

vectorX[i] = rightVector[i] / diag[i];

for (int j, k = i0; k < i1; k++)

{

j = ja[k];

rightVector[j] -= upperTringMat[k] \* vectorX[i];

}

}

}

double GridAndSLAE::VectorNorm(vector<double>& vector)

{

double norm = 0;

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

norm += vector[i] \* vector[i];

return sqrt(norm);

}

double GridAndSLAE::VectorScalarProduction(vector<double>& vector1, vector<double>& vector2)

{

double prod = 0;

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

{

prod += vector1[i] \* vector2[i];

}

return prod;

}

double GridAndSLAE::CalculateRelativeDiscrepancyWithR(double norm)

{

return VectorNorm(r) / norm;

}

void GridAndSLAE::CalculateLUsq()

{

for (int i = 0; i < NoN; i++) //i = 7

{

int i0 = ia[i]; //15

int i1 = ia[i + 1]; //19

double sumD = 0;

for (int k = i0; k < i1; k++)

{

double sumL = 0, sumU = 0;

int j = ja[k];

int j0 = ia[j];

int j1 = ia[j + 1];

int ik = i0;

int kj = j0;

for (; ik < k and kj < j1; )

{

if (ja[ik] < ja[kj])

ik++;

if (ja[ik] > ja[kj])

kj++;

if (ja[ik] == ja[kj]) {

sumL += alLU[ik] \* auLU[kj];

sumU += alLU[kj] \* auLU[ik];

ik++; kj++;

}

}

alLU[k] = (al[k] - sumL) / diLU[j];

auLU[k] = (au[k] - sumU) / diLU[j];

sumD += alLU[k] \* auLU[k];

}

diLU[i] = sqrt(di[i] - sumD);

}

}

void GridAndSLAE::MatrixUVectorMultiplicationLU(vector<double>& upperTringMat, vector<double>& diag, vector<double>& vectorMult, vector<double>& vectorOut)

{

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

{

vectorOut[i] = vectorMult[i] \* diag[i];

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

{

j = ja[k];

vectorOut[j] += upperTringMat[k] \* vectorMult[i];

}

}

}

void GridAndSLAE::TransposedMatrixVectorMultiplication(vector<double>& vectorMult, vector<double>& vectorOut)

{

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

{

vectorOut[i] = di[i] \* vectorMult[i];

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

{

int j = ja[k];

vectorOut[i] += au[k] \* vectorMult[j];

vectorOut[j] += al[k] \* vectorMult[i];

}

}

}

void GridAndSLAE::CalculateZ\_LUsq(vector<double>& vectorOut)

{

SolveBackwardLU(auLU, diLU, tmp, tmp);

MatrixVectorMultiplication(tmp, x0);

SolveForwardLU(alLU, diLU, x0, tmp);

SolveBackwardLU(alLU, diLU, tmp, tmp);

TransposedMatrixVectorMultiplication(tmp, x0);

SolveForwardLU(auLU, diLU, x0, vectorOut);

}

void GridAndSLAE::VectorSubtract(vector<double>& first, vector<double>& second, vector<double>& result)

{

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

{

result[i] = first[i] - second[i];

}

}

double GridAndSLAE::CalculateRelativeDiscrepancy(double norm)

{

MatrixVectorMultiplication(x, tmp);

VectorSubtract(b, tmp, tmp);

return VectorNorm(tmp) / norm;

}

void GridAndSLAE::OutputSolutionQ()

{

printf("q = {\n");

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

{

printf("%.15lf\n", x[i]);

}

printf("%.15lf\n }\n", x[NoN - 1]);

}

void GridAndSLAE::MSGForNonSymMatrixWithLuSqP()

{

CalculateLUsq();

MatrixUVectorMultiplicationLU(auLU, diLU, x0, x);

CalculateRelativeDiscrepancy(x0, tmp);

SolveForwardLU(alLU, diLU, tmp, tmp);

SolveBackwardLU(alLU, diLU, tmp, tmp);

TransposedMatrixVectorMultiplication(tmp, x0);

SolveForwardLU(auLU, diLU, x0, tmp);

VectorCopy(tmp, r);

VectorCopy(tmp, z);

double r\_rPrev, r\_rCur, Newz\_zPrev, ak, bk;

r\_rPrev = VectorScalarProduction(r, r);

double normB = VectorNorm(b);

double RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

for (int curIt = 0; curIt < maxiter and RelDiscrepancy > eps; curIt++)

{

VectorCopy(z, tmp);

CalculateZ\_LUsq(tmp);

Newz\_zPrev = VectorScalarProduction(tmp, z);

ak = r\_rPrev / Newz\_zPrev;

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

{

x[i] = x[i] + ak \* z[i];

r[i] = r[i] - ak \* tmp[i];

}

r\_rCur = VectorScalarProduction(r, r);

bk = r\_rCur / r\_rPrev;

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

z[i] = r[i] + bk \* z[i];

RelDiscrepancy = CalculateRelativeDiscrepancyWithR(normB);

r\_rPrev = r\_rCur;

printf("Iteration: %d, RelDiscrepancy of r: %.15lf\n", curIt + 1, RelDiscrepancy);

}

SolveBackwardLU(auLU, diLU, x, x);

printf("%.15lf\n", CalculateRelativeDiscrepancy(normB));

MatrixVectorMultiplication(x, tmp);

}

**Test.h**

#pragma once

#include "GRID.h"

double GridAndSLAE::FUN(int number, double x, double y, double z)

{

switch (number)

{

case 0:

return 0.4 \* (5. + 0.2 \* x + y + 30. \* z + 0.5 \* x \* y + x \* z + 10. \* y \* z + x \* y \* z);

break;

default:

throw "Ошибка в FUN";

break;

}

}

double GridAndSLAE::LAMBDA(int number)

{

switch (number)

{

case 0:

return 5.;

break;

default:

throw "Ошибка в LAMBDA";

break;

}

}

double GridAndSLAE::GAMMA(int number)

{

switch (number)

{

case 0:

return 0.4;

break;

default:

throw "Ошибка в GAMMA";

break;

}

}

double GridAndSLAE::TETA(int number, double x, double y, double z)

{

switch (number)

{

case 0: // Для теста с 1им КЭ

return -1. - 2.5 \* y - 5. \* z - 5. \* y \* z;

break;

case 1:

return 5. + 2.5 \* x + 50. \* z + 5. \* x \* z;

break;

case 2:

return 150. + 5. \* x + 50. \* y + 5. \* x \* y;

break;

default:

throw "Ошибка в TETA";

break;

}

}

double GridAndSLAE::UBETA(int number, double x, double y, double z)

{

switch (number)

{

case 0:// Для теста с 1им КЭ

return 9.1 + 11.25 \* y + 50.5 \* z + 30.5 \* y \* z;

break;

case 1:

return 4.5 - 0.05 \* x + 25. \* z + 0.5 \* x \* z;

break;

case 2:

return -10. - 0.3 \* x - 4 \* y;

default:

throw "Ошибка в UBETA";

break;

}

}