|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра теоретической и прикладной информатики | | |
|  | | |
| Курсовой проект | | |
| по дисциплине «Численные методы» | | |
|  | | |
| **МКЭ для краевой задачи** | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМ-81 |
| Вариант: | 43 |
| Студенты: | Ефремов Артур |
|  |  |
|  |  |
| Преподаватели: | Патрушев Илья Игоревич  Задорожный Александр Геннадьевич  Рояк Михаил Эммануилович |
|  | | |
| Новосибирск | | |
| 2021 | | |

1. **Постановка задачи**
   1. **Условие задачи**

Вариант №43. МКЭ для двумерной краевой задачи для гиперболического уравнения в декартовой системе координат. Базисные функции биквадратичные на прямоугольниках. Краевые условия всех типов. Коэффициент диффузии разложить по билинейным базисным функциям. Матрицу СЛАУ сгенерировать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией. Четырехслойная неявная схема по времени.

* 1. **Решаемое уравнение**
     1. **В общем виде**
     2. **В декартовой системе координат**
  2. **Краевые условия**
  3. **Расчетная область**

Расчетная область задается с помощью описания подобластей прямоугольной формы.

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

Потребуем, чтобы невязка дифференциального уравнения была ортогональна некоторому пространству пробных функций .

Используя формулу Грина, получим:

Так как , преобразуем интегралы по границам и , воспользовавшись краевыми условиями 2 и 3:

Исключаем так как краевыми условиями не определяется значение . Тогда потребуем, чтобы содержало только - функции, которые принимают нулевые значения на границе . Тогда в качестве выберем - пространство функций, имеющие суммируемые с квадратом производные и равных нулю на границе .

Таким образом, получим вариационное уравнение вида

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

Разобьем Ω на подобласти, получим:

Функцию будем искать в виде разложения по базисным функциям с соответствующими весами :

Итак, используя базисные функции, принимающих нулевые значения во всех узлах сетки кроме одного, СЛАУ для вектора весов может быть записана в матричном виде:

где компоненты матрицы A и вектора b определяются соотношениями:

Матрица жесткости:

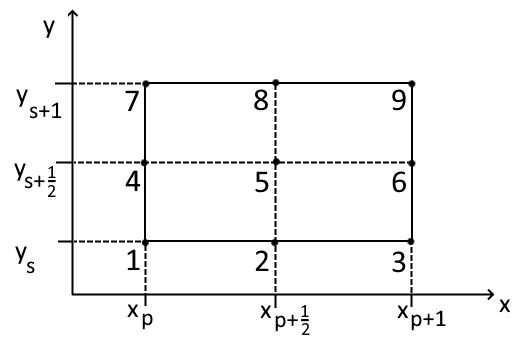
Матрица массы:

* + 1. **Построение базисных функций**

Биквадратичной называется функция вида:

Каждая биквадратичная функция фактически определяется значениями девяти коэффициентов. Поэтому на конечном элементе мы можем представить любую функцию в виде линейной комбинации некоторых девяти линейно-независимых биквадратичных функций.

Рассмотрим следующий способ определения явного вида биквадратичных базисных функций на прямоугольном конечном элементе. Представим каждую базисную функцию в виде произведений одномерных квадратичных базисных функций координат x и y:



Введем следующие обозначения:

Тогда локальные базисные функции на будут иметь следующий вид:

* + 1. **Локальные матрицы**

На локальных элементах базисные функции и можно выбрать следующими:

где или .

Так как коэффициент диффузии необходимо разложить по билинейным базисным функциям, выберем их в виде

На конечном элементе локальная матрица жесткости имеет следующий вид:

C учетом замены и разложением коэффициента диффузии по билинейному базису, локальная матрица жесткости принимает следующий вид:

Матрицу же массы удобнее всего вычислить через компоненты локальных матриц одномерных квадратичных элементов:

Локальный вектор правой части с учетом того что функция на конечном элементе представлена в виде биквадратичного интерполянта

может быть вычислен с помощью соотношения

где - вектор, составленный из значений правой части дифференциального уравнения в узлах элемента, – матрица массы:

Локальная матрица ребра длины с заданным на нем краевым условием третьего рода имеет вид:

а локальный вектор этого ребра при представлении на в виде разложения по одномерным квадратичным локальным базисным функциям имеет вид:

Локальный вектор ребра длины с заданным на нем краевым условием второго рода при представлении параметра на вычисляется аналогично и имеет вид:

Краевые условия первого рода учитываются после полной сборки глобальной матрицы и правой части путем фиксации соответствующих весов , при решении СЛАУ. Таким образом, из сгенерированной СЛАУ можно исключить уравнения с теми номерами, которые являются уравнениями узлов, лежащих на границе с краевыми условиями первого рода, а весам с этими номерами присвоить значения первого краевого условия в соответствующих узлах сетки. В работе реализован способ с занулением соответствующей строки глобальной матрицы системы, установкой единицы на главной диагонали и значения точного решения в соответствующей компоненте вектора правой части.

* 1. **Построение схемы по времени**

В четырехслойной неявной схеме по времени искомая функция может быть представлена в следующем виде:

где функции являются кубическими полиномами Лагранжа и имеют следующий вид:

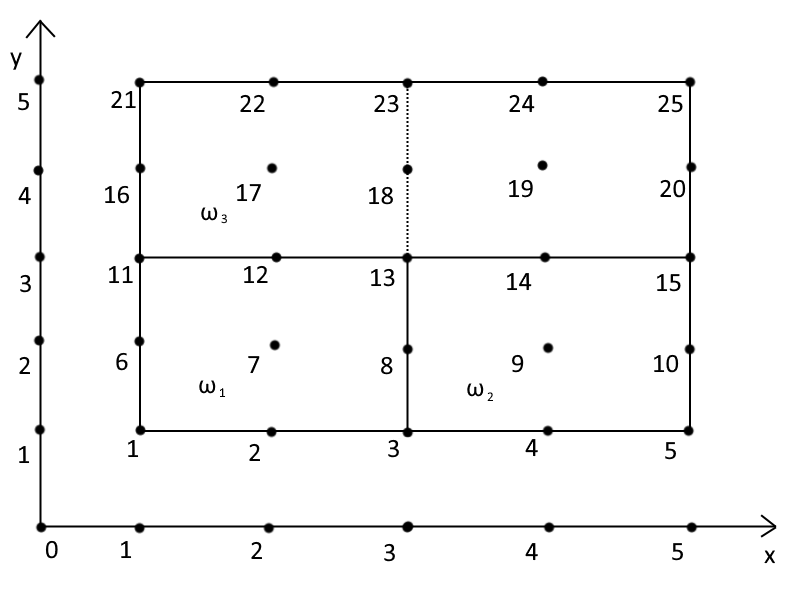
Вычислим производные функций по t:

Вычислим вторые производные функций по t:

Тогда задача сводится к решению следующей системы:

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

Разберем задание расчетной области и конечноэлементной сетки на следующем примере:



Расчетная область и конечноэлементная сетка задаются в файле **“regions.txt”** в следующем формате:

----X----

3

1 3 5

2 2

----Y----

3

1 3 5

2 2

-Regions-

3

0 1 0 1 5 1 2 2

1 2 0 1 5 1 0 0

0 2 1 2 5 1 1 1

Число **n** в первой строке соответствует количеству координат, необходимых для описания границ подобластей (регионов) по оси X, во второй строке идут **n** координат границ регионов по оси X (координатные линии). В третьей строке идут **n – 1** чисел, задающих количество разбиений сетки между координатными линиями. Разбиение строится и заносится в вектор **x\_nodes.**

Затем в строках 4, 5, 6 по схожему алгоритму описывается информация для построения сетки по Y, заполнения массива **y\_nodes.**

В строке 7 стоит число **m**, задающее количество регионов. В следующих **m** строках идут 4 индекса координат для левой, правой, нижней и верхней границы **i-ой** области, где , (**m** в примере равно 3), а также 4 числа, отвечающих за номер тестовой функции, номер функции для параметра λ, номер функции для параметра σ, номер функции для параметра χ соответственно.

Для хранения информации о регионах в программе используется массив элементов типа **Region**, который хранит поля **left, right** для хранения индексов границ региона в массиве **x\_nodes**, поля **bot, top** для хранения индексов региона в массиве **y\_nodes**. Хранение индексов таким образом позволяет добиться целочисленного сравнения при расчете подобласти, в которую попадает конечных элемент, зная индексы его центрального узла в массивах **x\_nodes** и **y\_nodes,** а также позволяет не выделять память под хранение координат каждого узла сетки, что позволяет сэкономить большое количество памяти, особенно при значительном увеличении числа дробления сетки. Также каждый **Region** хранит информацию о номерах функций для вычислений тестовых функций и коэффициентов уравнения.

Сетка по времени считывается из файла **“time.txt”** в котором хранится три числа: начальная временная точка, конечная временная точка и количество разбиений сетки. Сетка хранится в массиве **time\_grid.**

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

Формулу:

из

можно привести к виду:

где

и еще упростив, получим:

где

Матрица не зависит от размера конечного элемента, поэтому может быть вычислена заранее. Для удобства вычисления матрицы был написан скрипт на языке python:

import numpy as np

import math

def mu(i):

    return i % 3

def nu(i):

    return math.floor(i / 3)

M = np.array(

    [[4,2,-1],

    [2,16,2],

    [-1,2,4]])

M1 = np.zeros((9,9))

for i in range(9):

    for j in range(9):

        M1[i][j] = M[mu(i)][mu(j)] \* M[nu(i)][nu(j)]

def out(file\_name, mat):

    with open(file\_name, "w") as f:

     for i in range(9):

         f.write(str(mat[i][i]))

         f.write(" ")

     f.write("\n")

     for i in range(9):

for j in range(i):

             f.write(str(mat[i][j]))

             f.write(" ")

out("M.txt", M1)

Скрипт также осуществляет вывод диагоналей и нижних треугольников матриц в файл, для последующего чтения этих файлов основной программой.

* + - 1. **Матрица жесткости**

Формулу

можно привести к виду

где

Каждую из матриц и можно разделить на сумму 4 матриц и , которые будут отличаться лишь базисными функциями, отвечающими за разложение коэффициента диффузии в формулах для расчета интегралов. Представленные таким образом матрицы и не зависят от размера конечного элемента, поэтому могут быть вычислены заранее. Для удобства расчетов интегралов был написан скрипт на языке python:

from scipy.integrate import dblquad

import numpy as np

import math

def psi1(x):

    return 1 - x

def psi2(x):

    return x

flin = [psi1, psi2]

def phi1(x):

    return 2 \* (x - 0.5) \* (x - 1)

def phi2(x):

    return -4 \* x \* (x - 1)

def phi3(x):

    return 2 \* x \* (x - 0.5)

def dphi1dx(x):

    return 4 \* x - 3

def dphi2dx(x):

    return -8 \* x + 4

def dphi3dx(x):

    return 4 \* x – 1

fquad = [phi1, phi2, phi3]

dfquaddx = [dphi1dx, dphi2dx, dphi3dx]

Gl = np.zeros((4, 9, 9))

Gr = np.zeros((4, 9, 9))

for s in range(4):

    for i in range(9):

        for j in range(9):

            f = lambda x, y: flin[s % 2](x) \* flin[s // 2](y) \* fquad[i // 3](x) \* dfquaddx[i % 3](y) \* fquad[j // 3](x) \* dfquaddx[j % 3](y)

            Gl[s][i][j] = dblquad(f, 0, 1, 0, 1)[0]

            f = lambda x, y: flin[s % 2](x) \* flin[s // 2](y) \* dfquaddx[i // 3](x) \* fquad[i % 3](y) \* dfquaddx[j // 3](x) \* fquad[j % 3](y)

            Gr[s][i][j] = dblquad(f, 0, 1, 0, 1)[0]

def out(file\_name, mat):

    with open(file\_name, "w") as f:

        for i in range(9):

            f.write(str(mat[i][i]))

            f.write(" ")

        f.write("\n")

        for i in range(9):

            for j in range(i):

                f.write(str(mat[i][j]))

                f.write(" ")

out("data/Gl0.txt", Gl[0] \* 90)

out("data/Gl1.txt", Gl[1] \* 90)

out("data/Gl2.txt", Gl[2] \* 90)

out("data/Gl3.txt", Gl[3] \* 90)

out("data/Gr0.txt", Gr[0] \* 90)

out("data/Gr1.txt", Gr[1] \* 90)

out("data/Gr2.txt", Gr[2] \* 90)

out("data/Gr3.txt", Gr[3] \* 90)

Скрипт также осуществляет вывод диагоналей и нижних треугольников матриц в файл, для последующего чтения этих файлов основной программой.

* + 1. **Генерация глобальной матрицы**

Генерация глобальной матрицы происходит в цикле по конечным элементам. Для узлов каждого конечного элемента вычисляется соответствующий им индекс в глобальной нумерации узлов по следующей формуле:

где – количество конечных элементов по оси X, – номер конечного элемента, считая слева направо, снизу вверх.

Тогда для 9 узлов можно определить формулы их индексов в глобальной нумерации:

Координаты же узлов конечного элемента считаются по следующим формулам:

где и – массивы с координатами сетки по X и по Y соответственно.

Номер области, в которую попал конечный элемент легко найти – благодаря структуре хранения информации о каждом регионе (см. 3.1).

* + 1. **Генерация портрета глобальной матрицы**

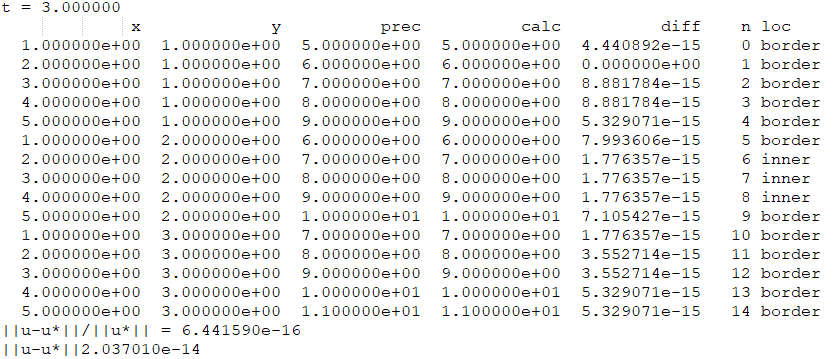
В цикле по конечным элементам вычисляются индексы узлов конечных элементов, формируется вспомогательный массив **help** размера 9 на 9, где для каждого элемента **helpij** лежит пара чисел – **i** и **j** в глобальной нумерации. Затем для каждого элемента из массива **help**, если в профиле **i** – той строки в портрете глобальной матрицы нет элемента, для которого **jg** равен **j**, добавим его, увеличим значения всех элементов начиная с **i** в массиве **ig** на 1.

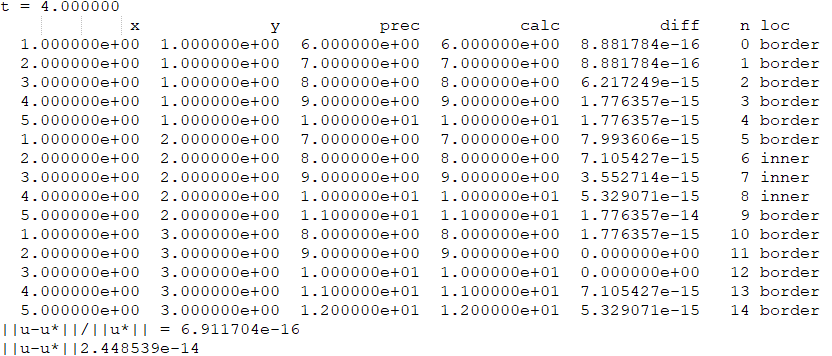
* + 1. **Решение СЛАУ**

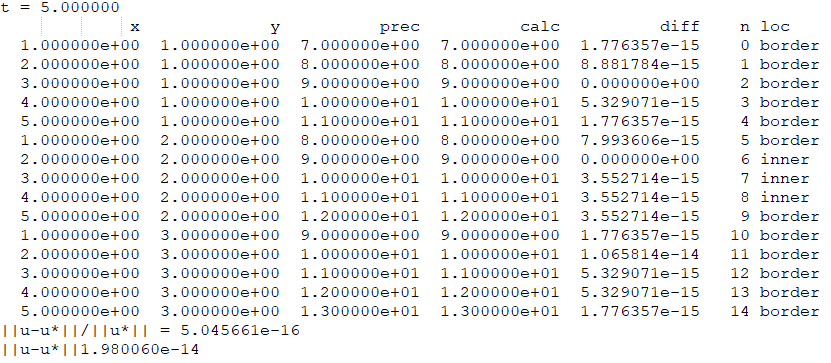
Решение СЛАУ осуществляется с помощью МСГ с неполной диагональной факторизацией.

1. **Описание тестирования программы**
   1. **Тестирование на прямоугольной расчетной области**

|  |  |
| --- | --- |
|  | **grid.txt** |
| ----X----  2  1 5  4  ----Y----  2  1 3  2  -Regions-  1  0 1 0 1 |
| **boundaries.txt** |
| 4  1 0 1 0 0  1 0 1 1 1  1 0 0 0 1  1 1 1 0 1 |

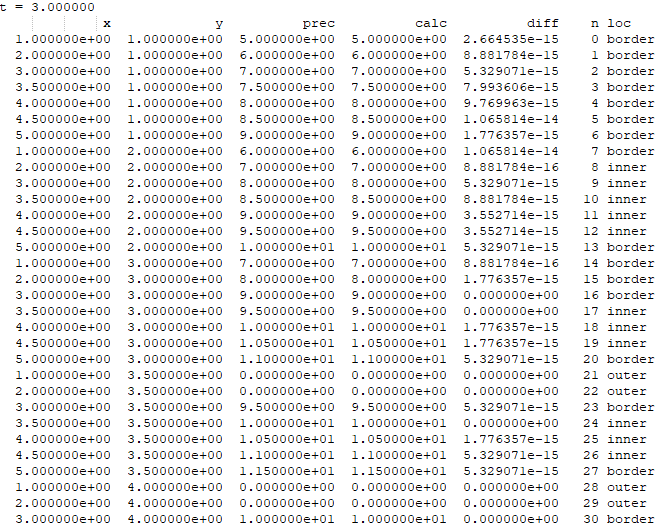


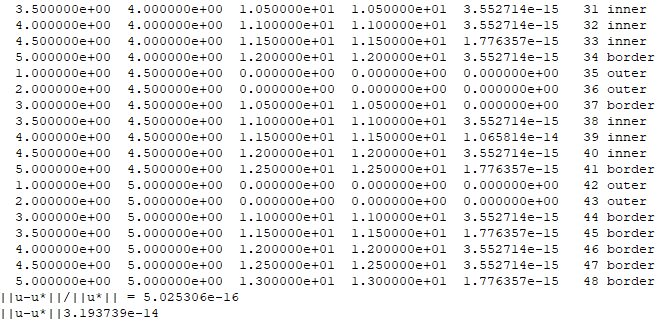




* 1. **Тестирование на произвольной расчетной области**

|  |  |
| --- | --- |
|  | **grid.txt** |
| ----X----  3  1 3 5  2 4  ----Y----  3  1 3 5  2 4  -Regions-  2  0 1 0 1  1 0 0 0  1 2 0 2  1 0 0 0 |
| **boundaries.txt** |
| 6  1 0 2 0 0  1 2 2 0 2  1 1 2 2 2  1 1 1 1 2  1 0 1 1 1  1 0 0 0 1 |

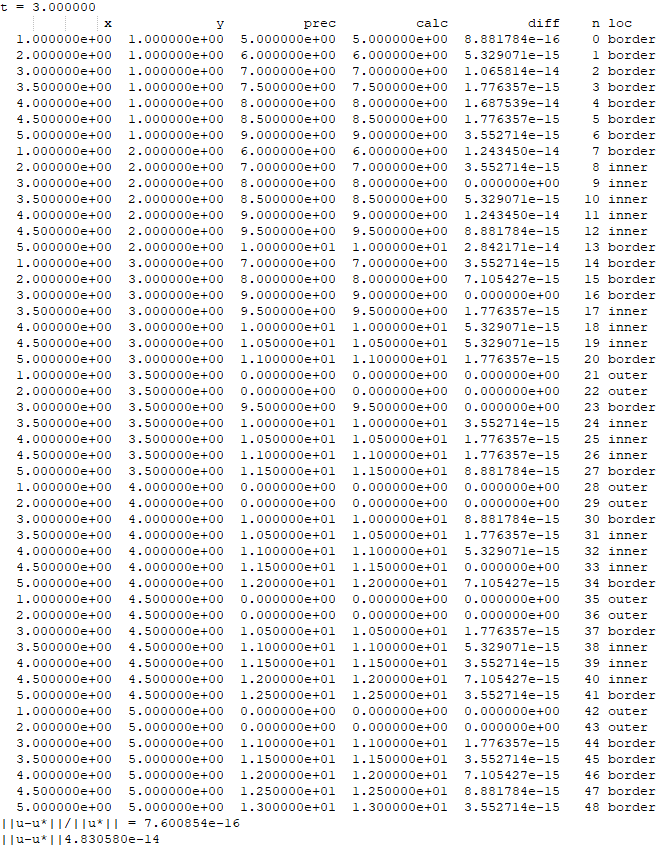
****

****

Обратим внимание на такие узлы как (1;4), (2;4). Программа адекватно интерпретировала их как внешние, при формировании портрета матрицы место под эти узлы не выделяется.

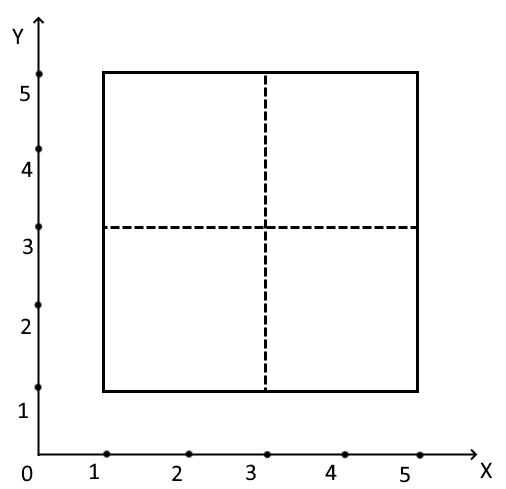
* 1. **Тестирование на произвольной расчетной области с разрывными параметрами**

|  |  |
| --- | --- |
|  | **grid.txt** |
| ----X----  3  1 3 5  2 4  ----Y----  3  1 3 5  2 4  -Regions-  2  0 1 0 1  1 0 0 0  1 2 0 2  1 0 1 1 |
| **boundaries.txt** |
| 6  1 0 2 0 0  1 2 2 0 2  1 1 2 2 2  1 1 1 1 2  1 0 1 1 1  1 0 0 0 1 |

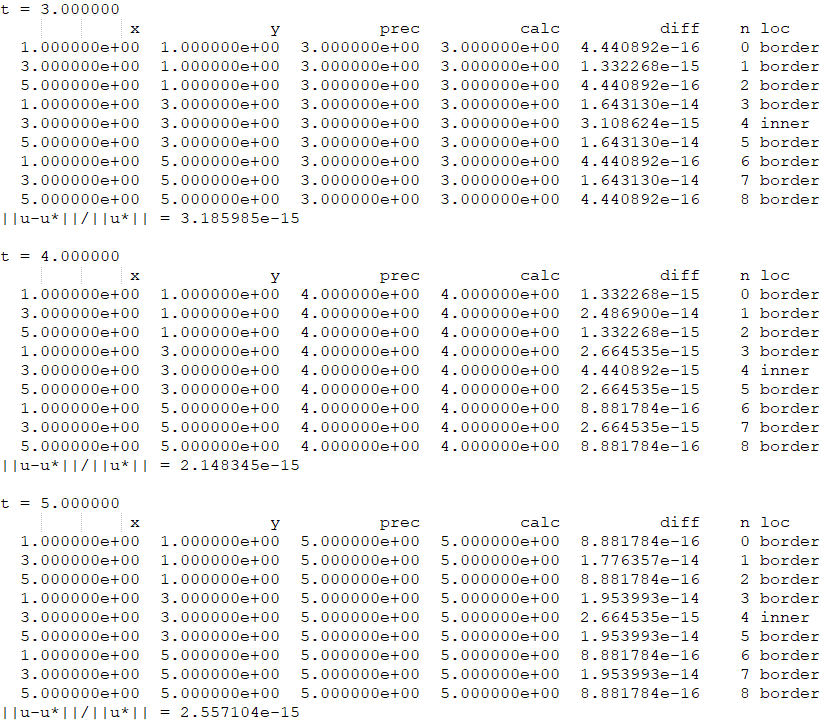


1. **Исследование порядка аппроксимации**

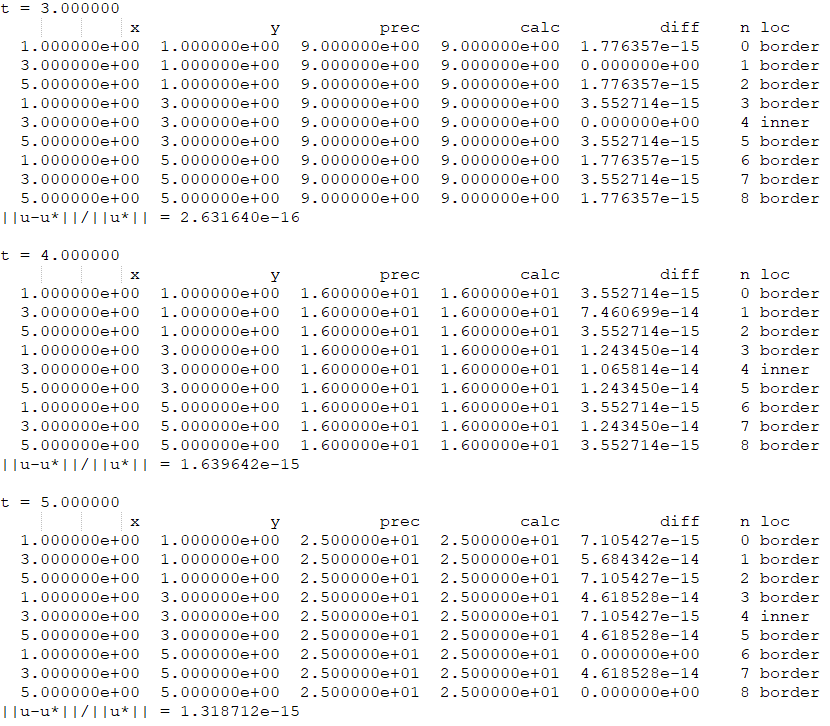
**Сетка по пространству для всех тестов:**

****

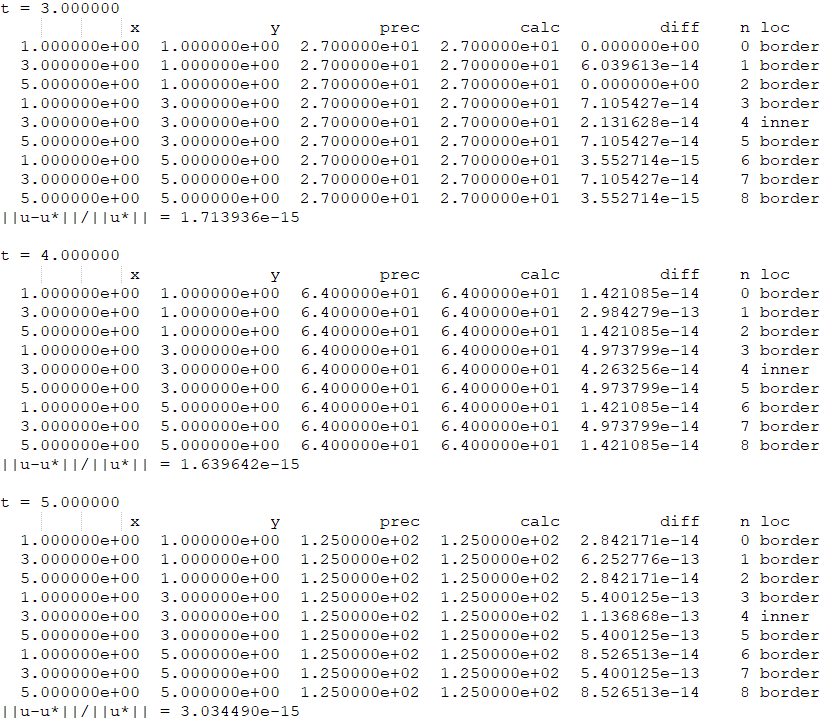
* 1. **Исследование порядка аппроксимации по времени**
     1. **Тест 1**



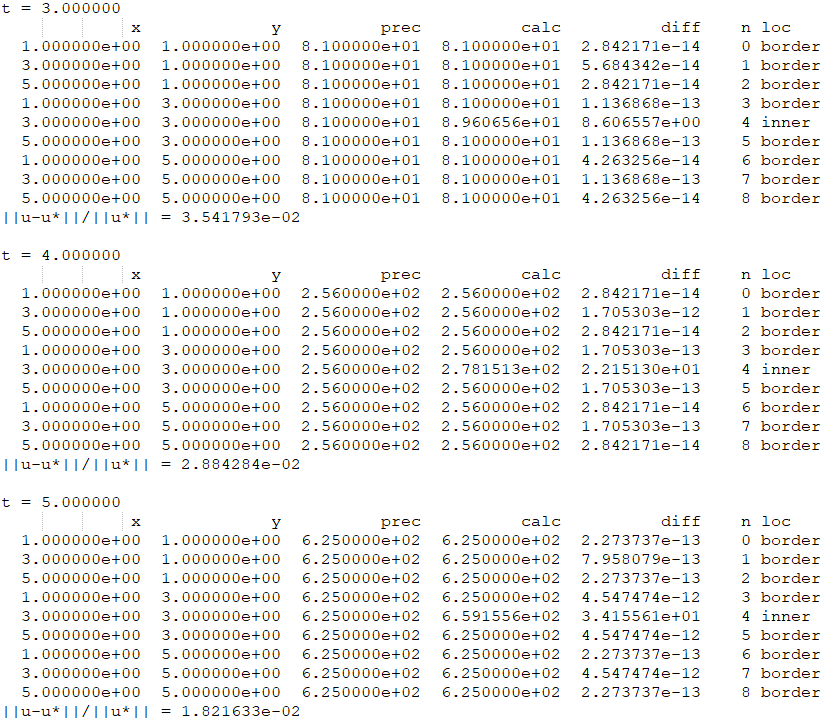
* + 1. **Тест 2**



* + 1. **Тест 3**



* + 1. **Тест 4**

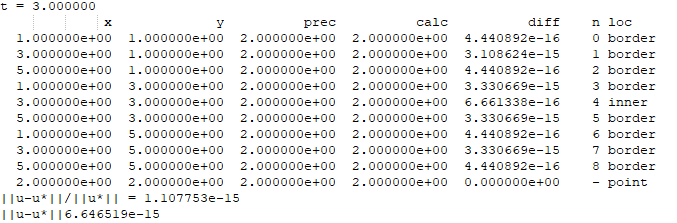


При увеличении степени t в искомой функции, начиная с , происходит увеличение погрешности. Следовательно, порядок аппроксимации схемы по времени = 3.

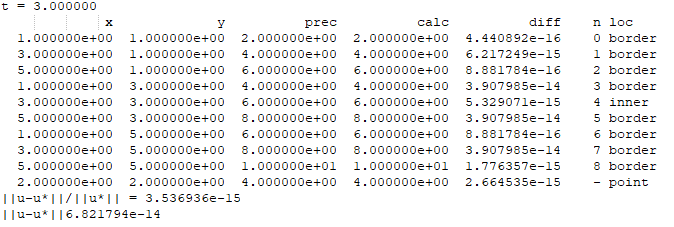
* 1. **Исследование порядка аппроксимации по пространству**

Помимо решения в узлах также выведем решение в произвольной точке расчетной области.

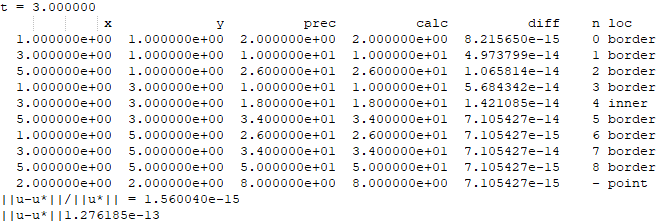
* + 1. **Тест 1**



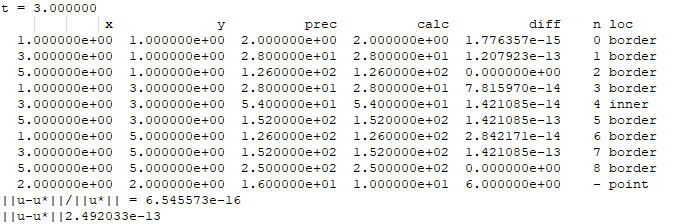
* + 1. **Тест 2**



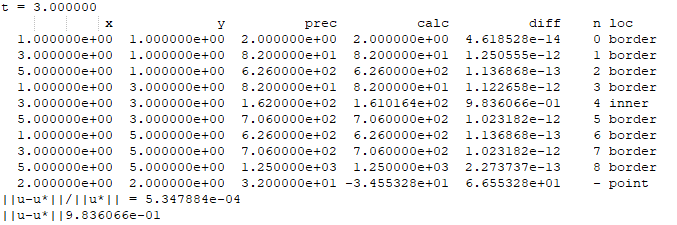
* + 1. **Тест 3**



* + 1. **Тест 4**



* + 1. **Тест 5**

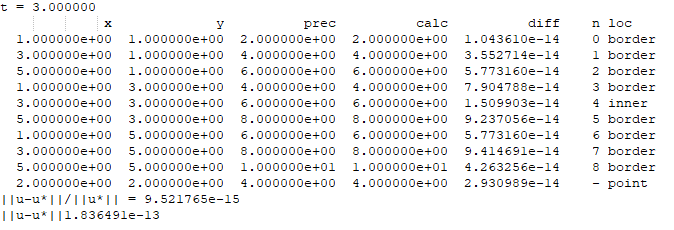


При увеличении степени x и y в искомой функции, начиная с , происходит увеличение погрешности в неузловой точке, а при и в узловой точке. Следовательно, порядок аппроксимации схемы по пространству = 2.

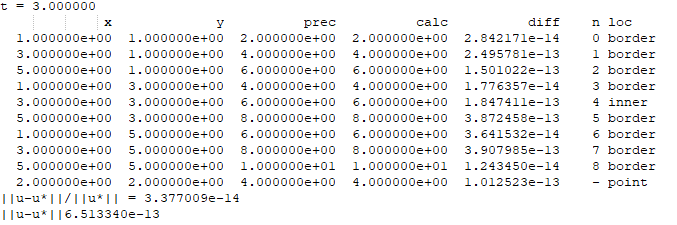
* 1. **Исследование порядка аппроксимации параметра диффузии**

Помимо решения в узлах также выведем решение в произвольной точке расчетной области.

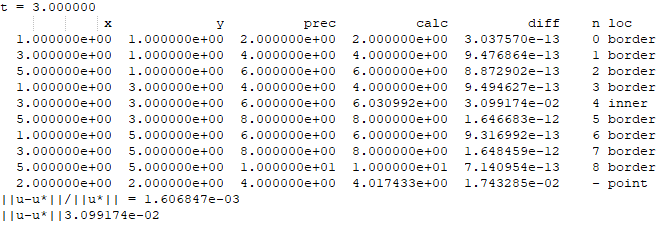
* + 1. **Тест 1**



* + 1. **Тест 2**



* + 1. **Тест 3**

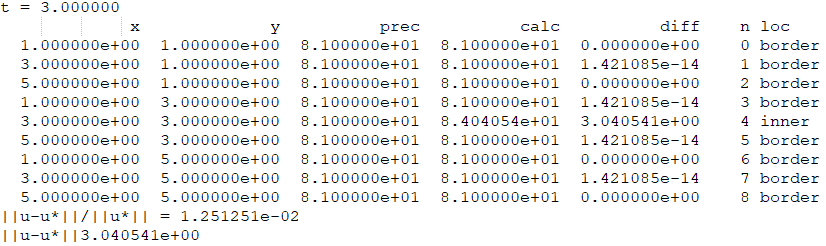


При увеличении степени x и y в коэффициенте диффузии функции, начиная с , происходит увеличение погрешности.

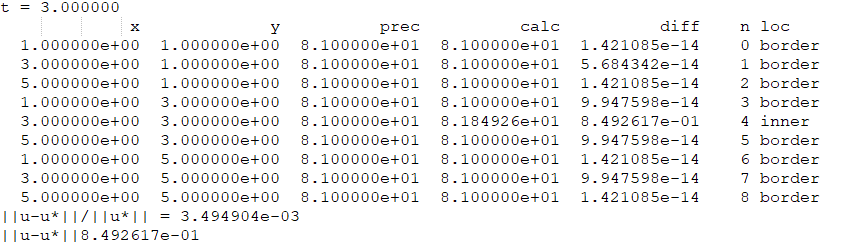
1. **Исследование порядка сходимости**
   1. **Исследование порядка сходимости по времени для параболической задачи**

Для удобства будем отображать только временные слои, которые были на изначальной временной сетке.

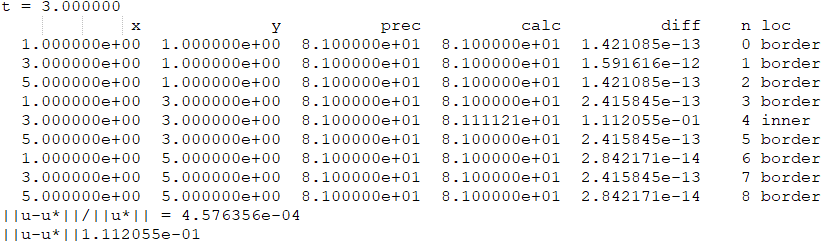
* + 1. **Тест 1**



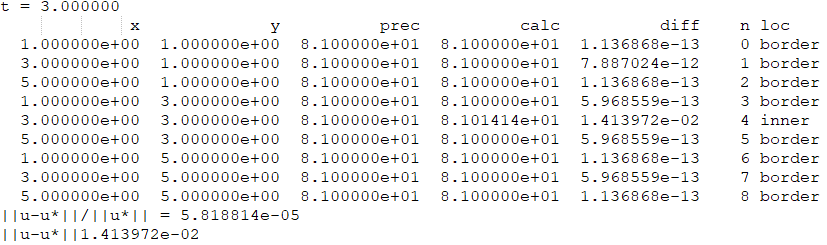
* + 1. **Тест 2**



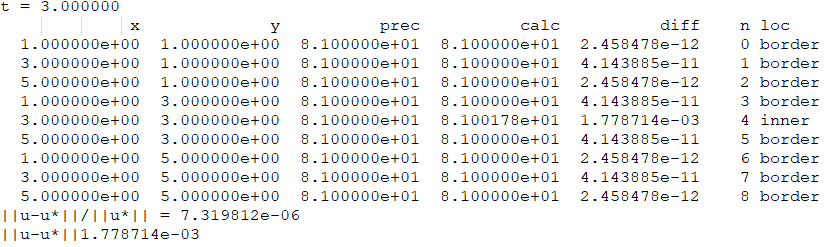
* + 1. **Тест 3**



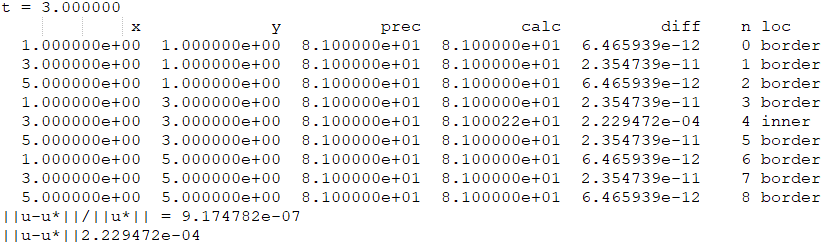
* + 1. **Тест 4**



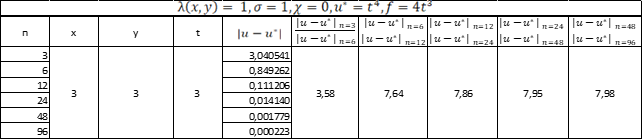
* + 1. **Тест 5**



* + 1. **Тест 6**



По результатам тестов можно составить таблицу для сравнения погрешностей во внутренней точке области:



и для сравнения нормы погрешности по всем точкам:

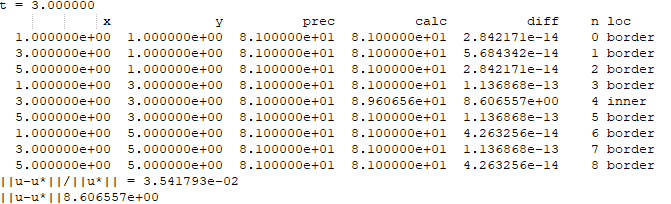


По таблицам видно, что порядок сходимости схемы для параболической задачи = 3.

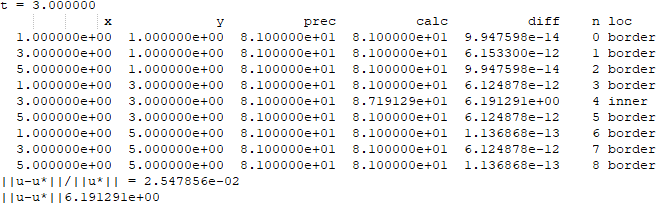
* 1. **Исследование порядка сходимости по времени для гиперболической задачи**

Для удобства будем отображать только временные слои, которые были на изначальной временной сетке.

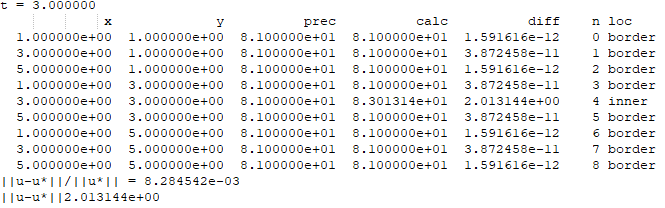
* + 1. **Тест 1**



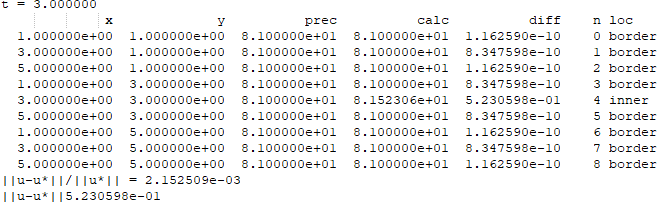
* + 1. **Тест 2**



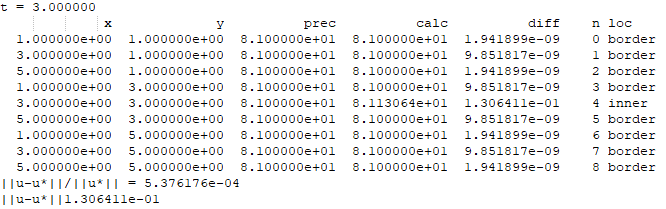
* + 1. **Тест 3**



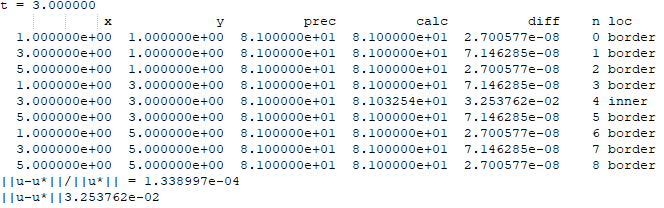
* + 1. **Тест 4**



* + 1. **Тест 5**



* + 1. **Тест 6**



По результатам тестов можно составить таблицу для сравнения погрешностей во внутренней точке области:



и для сравнения нормы погрешности по всем точкам:

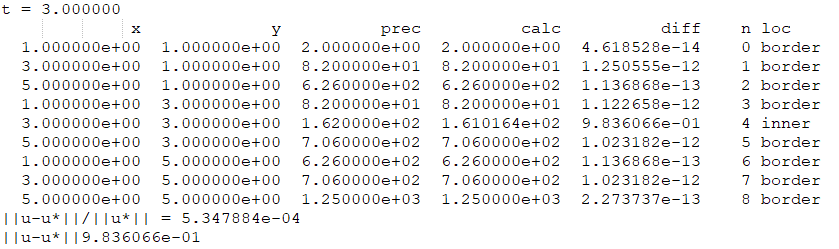


По таблицам видно, что порядок сходимости схемы для параболической задачи = 2.

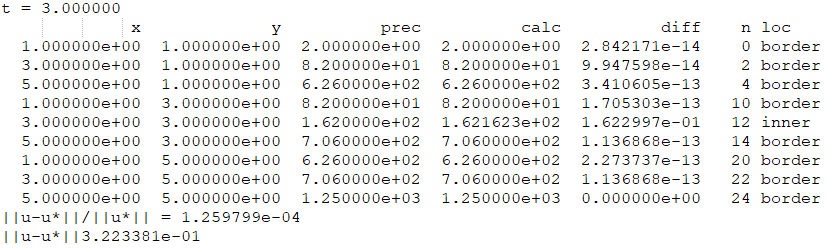
* 1. **Исследование порядка сходимости по пространству**

Для удобства будем отображать только узлы, которые были на изначальной пространственной сетке.

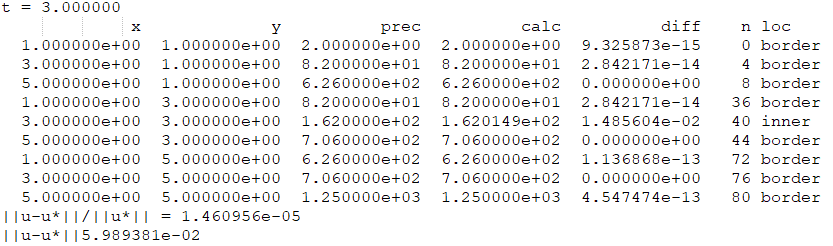
* + 1. **Тест 1**



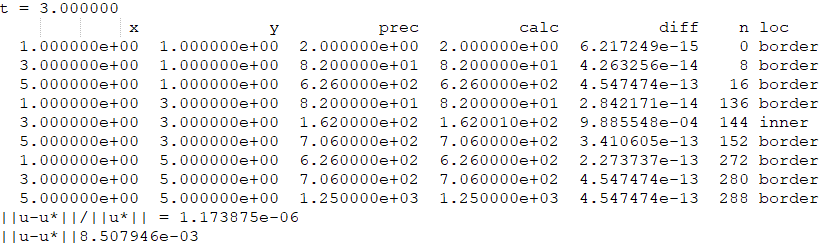
* + 1. **Тест 2**



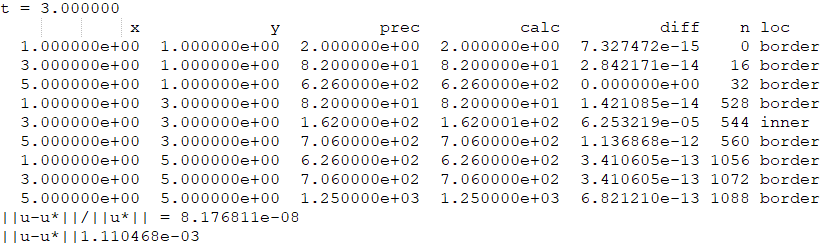
* + 1. **Тест 3**



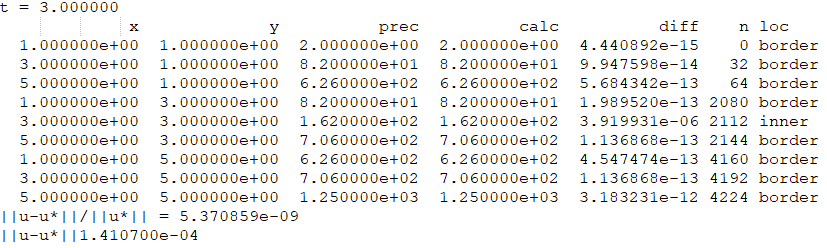
* + 1. **Тест 4**



* + 1. **Тест 5**



* + 1. **Тест 6**



По результатам тестов можно составить таблицу для сравнения погрешностей во внутренней точке области:



и для сравнения нормы погрешности по всем точкам:



По таблицам видно, что порядок сходимости схемы по пространству = 4.

1. **Тексты разработанных программ**
   1. **Файл *BoundaryValueProblem.h,* отвечающий за хранение основных данных и функций, реализующих решение краевой задачи гиперболического типа**

#pragma once

#include <iomanip>

#include "Matrix.h"

#include "SLAE.h"

#include "Test.h"

#include "Region.h"

using namespace std;

class BoundaryValueProblem

{

public:

Matrix global; // Глобальная матрица

Matrix fac\_global; // Неполная факторизация глобальной матрицы

Matrix stiff\_mat; // Матрица жесткости

Matrix sigma\_mass\_mat; // Матрица массы для параметра Сигма

Matrix chi\_mass\_mat; // Матрица массы для параметра Хи

Matrix M; // Вспомогательные матрицы для вычисления элементов

vector<Matrix> Gl, Gr; // локальных матриц и векторов правых частей

SLAE slae; // Решатель системы без предобуславливания

SLAE fac\_slae; // Решатель системы c предобуславливанием

vector<double> b; // Глобальный вектор правой части

vector<double> loc\_b; // Локальный вектор правой части

vector<double> solution; // Решение

vector<double> prev\_solution1; // Решение на предыдущей временной итерации

vector<double> prev\_solution2; // Решение на предпредыдущей временной итерации

vector<double> prev\_solution3; // Решение на предпредпредыдущей временной итерации

vector<double> true\_solution; // Точное решение для сравнения

vector<int> location; // Положение на сетке для каждого узла

int elems\_count; // Количество конечных элементов

int nodes\_count; // Количество узлов

vector<double> x\_nodes; // Координаты сетки по x

vector<double> y\_nodes; // Координаты сетки по y

int x\_nodes\_count; // Количество узлов по x

int y\_nodes\_count; // Количество узлов по y

vector<Region> regions; // Информация о подобластях (регионах)

int regions\_count; // Количество регионов

vector<int> x\_cords\_i; // Индексы исходных координатных линий в векторе сетки по x

vector<int> y\_cords\_i; // Индексы исходных координатных линий в векторе сетки по y

vector<vector<int>> boundaries; // Информация о краевых условиях

int bound\_count; // Количество краевых условий

vector<double> time\_grid; // Сетка по времени

int timesteps\_count; // Количество временных слоев

vector<double> Mnqj\_1; // Вспомогательный вектор для итерации по времени

vector<double> Mnqj\_2; // Вспомогательный вектор для итерации по времени

vector<double> Mnqj\_3; // Вспомогательный вектор для итерации по времени

vector<double> Moqj\_1; // Вспомогательный вектор для итерации по времени

vector<double> Moqj\_2; // Вспомогательный вектор для итерации по времени

vector<double> Moqj\_3; // Вспомогательный вектор для итерации по времени

vector<double> xk; // Вспомогательный вектор для МСГ

// Вспомогательный массив для вычисления индексов узлов в глобальной нумерации

vector<vector<vector<int>>> help;

// Считывание и формирование сетки по времени из файла file\_name

void ReadFormTimeGrid(const string& file\_name)

{

ifstream fin(file\_name);

double t\_first, t\_last;

fin >> t\_first >> t\_last >> timesteps\_count;

timesteps\_count++;

time\_grid = vector<double>(timesteps\_count);

double h = (t\_last - t\_first) / (timesteps\_count - 1);

time\_grid[0] = t\_first;

for(int i = 1; i < timesteps\_count; i++)

time\_grid[i] = h \* i;

fin.close();

}

// Считывание и формирование сетки из файла file\_name

void ReadFormGrid(const string& file\_name)

{

ifstream fin(file\_name);

// Переменная для считывания разделителей

string fake;

fin >> fake;

// Считывание координатных линий по x

int x\_coord\_count;

fin >> x\_coord\_count;

vector<double> x\_coords(x\_coord\_count);

for(int i = 0; i < x\_coord\_count; i++)

fin >> x\_coords[i];

// Формирование сетки по x

x\_nodes\_count = 1;

x\_nodes = vector<double>(1);

x\_nodes[0] = x\_coords[0];

vector<int> n\_x(x\_coord\_count - 1);

for(int i = 0; i < x\_coord\_count - 1; i++)

{

fin >> n\_x[i];

x\_nodes.resize(x\_nodes\_count + n\_x[i]);

double h = (x\_coords[i + 1] - x\_coords[i]) / n\_x[i];

for(int j = 0; j < n\_x[i]; j++)

x\_nodes[j + x\_nodes\_count] = x\_nodes[x\_nodes\_count - 1] + h \* (j + 1);

x\_nodes\_count += n\_x[i];

}

fin >> fake;

// Считывание координатных линий по y

int y\_coord\_count;

fin >> y\_coord\_count;

vector<double> y\_coords(y\_coord\_count);

for(int i = 0; i < y\_coord\_count; i++)

fin >> y\_coords[i];

// Формирование сетки по y

y\_nodes\_count = 1;

y\_nodes = vector<double>(1);

y\_nodes[0] = y\_coords[0];

vector<int> n\_y(y\_coord\_count - 1);

for(int i = 0; i < y\_coord\_count - 1; i++)

{

fin >> n\_y[i];

y\_nodes.resize(y\_nodes\_count + n\_y[i]);

double h = (y\_coords[i + 1] - y\_coords[i]) / n\_y[i];

for(int j = 0; j < n\_y[i]; j++)

y\_nodes[j + y\_nodes\_count] = y\_nodes[y\_nodes\_count - 1] + h \* (j + 1);

y\_nodes\_count += n\_y[i];

}

fin >> fake;

// Считывание информации о подобластях

fin >> regions\_count;

regions = vector<Region>(regions\_count);

for(int i = 0; i < regions\_count; i++)

{

fin >> regions[i].left >> regions[i].right >> regions[i].bot >> regions[i].top;

int test\_n;

fin >> test\_n;

regions[i].test = Test(test\_n);

fin >> regions[i].test.lambda\_n >> regions[i].test.sigma\_n >> regions[i].test.chi\_n;

}

fin.close();

// Пересчет индексов границ подобластей

for(int i = 1; i < n\_x.size(); i++)

n\_x[i] += n\_x[i - 1];

for(int i = 1; i < n\_y.size(); i++)

n\_y[i] += n\_y[i - 1];

x\_cords\_i = vector<int>(n\_x.size() + 1);

for(int i = 0; i < n\_x.size(); i++)

x\_cords\_i[i + 1] = n\_x[i];

y\_cords\_i = vector<int>(n\_y.size() + 1);

for(int i = 0; i < n\_y.size(); i++)

y\_cords\_i[i + 1] = n\_y[i];

for(int reg\_i = 0; reg\_i < regions\_count; reg\_i++)

{

regions[reg\_i].left = x\_cords\_i[regions[reg\_i].left];

regions[reg\_i].right = x\_cords\_i[regions[reg\_i].right];

regions[reg\_i].bot = x\_cords\_i[regions[reg\_i].bot];

regions[reg\_i].top = x\_cords\_i[regions[reg\_i].top];

}

nodes\_count = x\_nodes\_count \* y\_nodes\_count;

elems\_count += n\_x[n\_x.size() - 1] / 2 \* n\_y[n\_y.size() - 1] / 2;

}

// Считывание информации о краевых условиях из файла file\_name

void ReadBoundaries(const string& file\_name)

{

ifstream fin(file\_name);

fin >> bound\_count;

boundaries = vector<vector<int>>(bound\_count, vector<int>(5));

for(int bound\_i = 0; bound\_i < bound\_count; bound\_i++)

{

for(int i = 0; i < 5; i++)

fin >> boundaries[bound\_i][i];

boundaries[bound\_i][1] = x\_cords\_i[boundaries[bound\_i][1]];

boundaries[bound\_i][2] = x\_cords\_i[boundaries[bound\_i][2]];

boundaries[bound\_i][3] = y\_cords\_i[boundaries[bound\_i][3]];

boundaries[bound\_i][4] = y\_cords\_i[boundaries[bound\_i][4]];

}

fin.close();

}

// Считывание вспомогательных матриц для формирования

// матриц жесткости и массы

void ReadMatrices()

{

M = Matrix(9);

M.ReadDiTr("data/M1.txt");

Gl = vector<Matrix>(4, Matrix(9));

Gl[0].ReadDiTr("data/Gl0.txt");

Gl[1].ReadDiTr("data/Gl1.txt");

Gl[2].ReadDiTr("data/Gl2.txt");

Gl[3].ReadDiTr("data/Gl3.txt");

Gr = vector<Matrix>(4, Matrix(9));

Gr[0].ReadDiTr("data/Gr0.txt");

Gr[1].ReadDiTr("data/Gr1.txt");

Gr[2].ReadDiTr("data/Gr2.txt");

Gr[3].ReadDiTr("data/Gr3.txt");

}

// Выделение памяти под массивы

void InitializeMemory()

{

slae = SLAE(nodes\_count, 10000, 1e-20);

fac\_slae = SLAE(nodes\_count, 10000, 1e-20);

global.ind = vector<int>(nodes\_count + 1);

b = vector<double>(nodes\_count);

solution = vector<double>(nodes\_count);

prev\_solution1 = vector<double>(nodes\_count);

prev\_solution2 = vector<double>(nodes\_count);

prev\_solution3 = vector<double>(nodes\_count);

true\_solution = vector<double>(nodes\_count);

location = vector<int>(nodes\_count);

Mnqj\_1 = vector<double>(nodes\_count);

Mnqj\_2 = vector<double>(nodes\_count);

Mnqj\_3 = vector<double>(nodes\_count);

Moqj\_1 = vector<double>(nodes\_count);

Moqj\_2 = vector<double>(nodes\_count);

Moqj\_3 = vector<double>(nodes\_count);

stiff\_mat = Matrix(nodes\_count);

sigma\_mass\_mat = Matrix(nodes\_count);

chi\_mass\_mat = Matrix(nodes\_count);

fac\_global = Matrix(nodes\_count, 0);

xk = vector<double>(nodes\_count);

help = vector<vector<vector<int>>>(9);

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

help[i].resize(9);

}

// Заполнение массива global\_indices индексами, соответствующими глобальной номерации

// узлов конечного элемента с номером elem\_i(индексация с нуля)

void CalcGlobalIndices(int elem\_i, vector<int>& global\_indices)

{

int n\_coords = x\_nodes\_count / 2 + 1;

int k = 2 \* floor((elem\_i) / (n\_coords - 1)) \* (2 \* n\_coords - 1) + 2 \* ((elem\_i) % (n\_coords - 1));

global\_indices[0] = k + 0;

global\_indices[1] = k + 1;

global\_indices[2] = k + 2;

global\_indices[3] = k + 2 \* n\_coords - 1;

global\_indices[4] = k + 2 \* n\_coords - 0;

global\_indices[5] = k + 2 \* n\_coords + 1;

global\_indices[6] = k + 2 \* (2 \* n\_coords - 1) + 0;

global\_indices[7] = k + 2 \* (2 \* n\_coords - 1) + 1;

global\_indices[8] = k + 2 \* (2 \* n\_coords - 1) + 2;

}

// Поиск региона по номеру конечного элемента

int CalcRegionIndex(const int& elem\_i)

{

int n\_coords = x\_nodes\_count / 2 + 1;

int x\_i = (elem\_i) % (n\_coords - 1) \* 2 + 1;

int y\_i = floor((elem\_i) / (n\_coords - 1)) \* 2 + 1;

return CalcRegionIndex(x\_i, y\_i);

}

// Поиск региона по индексам в сетке

int CalcRegionIndex(const int& x\_i, const int& y\_i)

{

int found\_reg\_i = -1;

for(int reg\_i = 0; reg\_i < regions\_count; reg\_i++)

{

if(regions[reg\_i].left <= x\_i && x\_i <= regions[reg\_i].right &&

regions[reg\_i].bot <= y\_i && y\_i <= regions[reg\_i].top)

{

found\_reg\_i = reg\_i;

break;

}

}

return found\_reg\_i;

}

// Вспомогательная функция для формирования портрета

void IncertToRow(const int& r, const int& c)

{

int i\_in\_jg = global.ind[r];

int prof\_len = global.ind[r + 1] - global.ind[r];

bool found = false;

for(int k = i\_in\_jg; k < i\_in\_jg + prof\_len; k++)

if(global.columns\_ind[k] == c)

{

found = true;

break;

}

if(!found)

{

for(int l = r + 1; l < global.ind.size(); l++)

global.ind[l]++;

int k = i\_in\_jg;

while((k < i\_in\_jg + prof\_len) && global.columns\_ind[k] < c)

k++;

global.columns\_ind.insert(global.columns\_ind.begin() + k, c);

}

}

// Формирование портрета глобальной матрицы

void FormPortrait()

{

global.ind[0] = global.ind[1] = 0;

for(int elem\_i = 0; elem\_i < elems\_count; elem\_i++)

{

int reg\_i = CalcRegionIndex(elem\_i);

if(reg\_i != -1)

{

vector<int> global\_indices(9);

CalcGlobalIndices(elem\_i, global\_indices);

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

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

help[i][j] = { global\_indices[i], global\_indices[j] };

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

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

IncertToRow(help[i][j][0], help[i][j][1]);

}

}

global.size = global.ind.size() - 1;

global.diag.resize(global.size);

global.tr\_size = global.columns\_ind.size();

global.bot\_tr.resize(global.tr\_size);

global.top\_tr.resize(global.tr\_size);

stiff\_mat = global;

sigma\_mass\_mat = global;

chi\_mass\_mat = global;

}

// Добавление элемента в матрицу в строку row\_i столбец col\_i

// val\_l добавляется в нижний треугольник, val\_u добавляется в верхний треугольник

void AddToMat(Matrix& mat, const int& row\_i, const int& col\_i, const double& val\_l, const double& val\_u)

{

int beg\_prof = mat.ind[row\_i];

int end\_prof = mat.ind[row\_i + 1];

for(int i\_in\_prof = beg\_prof; i\_in\_prof < end\_prof; i\_in\_prof++)

{

if(mat.columns\_ind[i\_in\_prof] == col\_i)

{

mat.bot\_tr[i\_in\_prof] += val\_l;

mat.top\_tr[i\_in\_prof] += val\_u;

break;

}

}

}

// Находит координаты узлов конечного элемента

// с номером elem\_i(индексация с нуля)

void CalcElemNodes(const int& elem\_i, vector<double>& x\_nodes\_elem, vector<double>& y\_nodes\_elem)

{

int n\_coords = x\_nodes\_count / 2 + 1;

int x0 = (elem\_i) % (n\_coords - 1) \* 2;

int y0 = floor((elem\_i) / (n\_coords - 1)) \* 2;

x\_nodes\_elem[0] = x\_nodes[x0];

x\_nodes\_elem[1] = x\_nodes[x0 + 1];

x\_nodes\_elem[2] = x\_nodes[x0 + 2];

y\_nodes\_elem[0] = y\_nodes[y0];

y\_nodes\_elem[1] = y\_nodes[y0 + 1];

y\_nodes\_elem[2] = y\_nodes[y0 + 2];

}

// Сборка матриц жесткости и массы

void BuildMatrices(const double& t)

{

vector<int> global\_indices(9);

for(int elem\_i = 0; elem\_i < elems\_count; elem\_i++)

{

int reg\_i = CalcRegionIndex(elem\_i);

CalcGlobalIndices(elem\_i, global\_indices);

if(reg\_i == -1)

{

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

{

stiff\_mat.diag[global\_indices[i]] = 1;

sigma\_mass\_mat.diag[global\_indices[i]] = 0;

chi\_mass\_mat.diag[global\_indices[i]] = 0;

b[global\_indices[i]] = 0;

location[global\_indices[i]] = 2;

}

}

else

{

Region r = regions[reg\_i];

vector<double> x\_nodes\_elem(3); // Координаты конечного элемента по x

vector<double> y\_nodes\_elem(3); // Координаты конечного элемента по y

CalcElemNodes(elem\_i, x\_nodes\_elem, y\_nodes\_elem);

double hx = x\_nodes\_elem[2] - x\_nodes\_elem[0];

double hy = y\_nodes\_elem[2] - y\_nodes\_elem[0];

vector<double> local\_f(9);

vector<double> lambda {

r.test.lambda(x\_nodes\_elem[0], y\_nodes\_elem[0]),

r.test.lambda(x\_nodes\_elem[0], y\_nodes\_elem[2]),

r.test.lambda(x\_nodes\_elem[2], y\_nodes\_elem[0]),

r.test.lambda(x\_nodes\_elem[2], y\_nodes\_elem[2]) };

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

{

double x = x\_nodes\_elem[i % 3];

double y = y\_nodes\_elem[floor(i / 3)];

stiff\_mat.diag[global\_indices[i]] += (1.0 / 90.0) \*

(hy / hx \* (Gl[0].diag[i] \* lambda[0] + Gl[1].diag[i] \* lambda[1] + Gl[2].diag[i] \* lambda[2] + Gl[3].diag[i] \* lambda[3]) +

hx / hy \* (Gr[0].diag[i] \* lambda[0] + Gr[1].diag[i] \* lambda[1] + Gr[2].diag[i] \* lambda[2] + Gr[3].diag[i] \* lambda[3]));

sigma\_mass\_mat.diag[global\_indices[i]] += (r.test.sigma() \* hx \* hy / 900.0) \* M.diag[i];

chi\_mass\_mat.diag[global\_indices[i]] += (r.test.chi() \* hx \* hy / 900.0) \* M.diag[i];

local\_f[i] = r.test.f(x, y, t);

true\_solution[global\_indices[i]] = r.test.u(x, y, t);

int beg\_prof = M.ind[i];

int end\_prof = M.ind[i + 1];

for(int i\_in\_prof = beg\_prof; i\_in\_prof < end\_prof; i\_in\_prof++)

{

int j = M.columns\_ind[i\_in\_prof];

double val\_l = (1.0 / 90.0) \*

(hy / hx \* (Gl[0].bot\_tr[i\_in\_prof] \* lambda[0] + Gl[1].bot\_tr[i\_in\_prof] \* lambda[1] + Gl[2].bot\_tr[i\_in\_prof] \* lambda[2] + Gl[3].bot\_tr[i\_in\_prof] \* lambda[3]) +

hx / hy \* (Gr[0].bot\_tr[i\_in\_prof] \* lambda[0] + Gr[1].bot\_tr[i\_in\_prof] \* lambda[1] + Gr[2].bot\_tr[i\_in\_prof] \* lambda[2] + Gr[3].bot\_tr[i\_in\_prof] \* lambda[3]));

double val\_u = (1.0 / 90.0) \*

(hy / hx \* (Gl[0].top\_tr[i\_in\_prof] \* lambda[0] + Gl[1].top\_tr[i\_in\_prof] \* lambda[1] + Gl[2].top\_tr[i\_in\_prof] \* lambda[2] + Gl[3].top\_tr[i\_in\_prof] \* lambda[3]) +

hx / hy \* (Gr[0].top\_tr[i\_in\_prof] \* lambda[0] + Gr[1].top\_tr[i\_in\_prof] \* lambda[1] + Gr[2].top\_tr[i\_in\_prof] \* lambda[2] + Gr[3].top\_tr[i\_in\_prof] \* lambda[3]));

AddToMat(stiff\_mat, global\_indices[i], global\_indices[j], val\_l, val\_u);

val\_l = (r.test.sigma() \* hx \* hy / 900.0) \* M.bot\_tr[i\_in\_prof];

val\_u = (r.test.sigma() \* hx \* hy / 900.0) \* M.top\_tr[i\_in\_prof];

AddToMat(sigma\_mass\_mat, global\_indices[i], global\_indices[j], val\_l, val\_u);

val\_l = (r.test.chi() \* hx \* hy / 900.0) \* M.bot\_tr[i\_in\_prof];

val\_u = (r.test.chi() \* hx \* hy / 900.0) \* M.top\_tr[i\_in\_prof];

AddToMat(chi\_mass\_mat, global\_indices[i], global\_indices[j], val\_l, val\_u);

}

}

vector<double> local\_b(9);

M.MatVecMult(local\_f, local\_b, M.bot\_tr, M.top\_tr);

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

b[global\_indices[i]] += hx \* hy / 900.0 \* local\_b[i];

}

}

}

// Сборка глобальной матрицы

void AssembleGlobalMatrix()

{

for(int i = 0; i < nodes\_count; i++)

global.diag[i] = stiff\_mat.diag[i] + sigma\_mass\_mat.diag[i] + chi\_mass\_mat.diag[i];

for(int i = 0; i < global.tr\_size; i++)

{

global.bot\_tr[i] = stiff\_mat.bot\_tr[i] + sigma\_mass\_mat.bot\_tr[i] + chi\_mass\_mat.bot\_tr[i];

global.top\_tr[i] = stiff\_mat.top\_tr[i] + sigma\_mass\_mat.top\_tr[i] + chi\_mass\_mat.top\_tr[i];

}

}

// Учет первых краевых условий u в точке (x,y) на временном слое t на строке с номером line\_i

void FirstBoundOnLine(const int& line\_i, const double& x, const double& y, const double& t, const double& u)

{

global.diag[line\_i] = 1;

true\_solution[line\_i] = u;

b[line\_i] = u;

for(int prof\_i = global.ind[line\_i]; prof\_i < global.ind[line\_i + 1]; prof\_i++)

{

global.bot\_tr[prof\_i] = 0;

}

for(int i = 0; i < nodes\_count; i++)

{

for(int prof\_i = global.ind[i]; prof\_i < global.ind[i + 1]; prof\_i++)

{

if(global.columns\_ind[prof\_i] == line\_i)

global.top\_tr[prof\_i] = 0;

}

}

}

// Учет первых краевых условий в СЛАУ на временном слое t

void AccountFirstBound(const double& t)

{

for(int x\_i = 0; x\_i < x\_nodes\_count; x\_i++)

{

for(int y\_i = 0; y\_i < y\_nodes\_count; y\_i++)

{

int reg\_i = CalcRegionIndex(x\_i, y\_i);

if(reg\_i != -1)

{

for(int bound\_i = 0; bound\_i < bound\_count; bound\_i++)

{

if(boundaries[bound\_i][1] <= x\_i && x\_i <= boundaries[bound\_i][2] &&

boundaries[bound\_i][3] <= y\_i && y\_i <= boundaries[bound\_i][4])

{

int i = x\_i + y\_i \* x\_nodes\_count;

FirstBoundOnLine(i, x\_nodes[x\_i], y\_nodes[y\_i], t, regions[reg\_i].test.u(x\_nodes[x\_i], y\_nodes[y\_i], t));

location[i] = 1;

break;

}

}

}

}

}

}

// Нахождение решения СЛАУ

void Solve()

{

slae.b = b;

global.DiagFact(fac\_global);

cout << slae.ConjGradPredMethod(xk, solution, global, fac\_slae, fac\_global) << endl;

//slae.ConjGradMethod(xk, solution, global);

}

// Итерации по времени, вывод в поток fout

void TimeIterations(ofstream& fout)

{

for(int y\_i = 0; y\_i < y\_nodes\_count; y\_i++)

{

for(int x\_i = 0; x\_i < x\_nodes\_count; x\_i++)

{

int reg\_i = CalcRegionIndex(x\_i, y\_i);

if(reg\_i != -1)

{

prev\_solution3[x\_i + y\_i \* x\_nodes\_count] =

regions[reg\_i].test.u(x\_nodes[x\_i], y\_nodes[y\_i], time\_grid[0]);

prev\_solution2[x\_i + y\_i \* x\_nodes\_count] =

regions[reg\_i].test.u(x\_nodes[x\_i], y\_nodes[y\_i], time\_grid[1]);

prev\_solution1[x\_i + y\_i \* x\_nodes\_count] =

regions[reg\_i].test.u(x\_nodes[x\_i], y\_nodes[y\_i], time\_grid[2]);

}

}

}

for(int time\_i = 3; time\_i < timesteps\_count; time\_i++)

{

// Временные слои

double t3 = time\_grid[time\_i - 3];

double t2 = time\_grid[time\_i - 2];

double t1 = time\_grid[time\_i - 1];

double t0 = time\_grid[time\_i - 0];

// Числители дробей

double den3 = (t3 - t2) \* (t3 - t1) \* (t3 - t0);

double den2 = (t2 - t3) \* (t2 - t1) \* (t2 - t0);

double den1 = (t1 - t3) \* (t1 - t2) \* (t1 - t0);

double den0 = (t0 - t3) \* (t0 - t2) \* (t0 - t1);

// Первые произодные

double n3 = (3 \* t0 \* t0 - 2 \* t0 \* t2 - 2 \* t0 \* t1 - 2 \* t0 \* t0 +

t2 \* t1 + t2 \* t0 + t1 \* t0) / den3;

double n2 = (3 \* t0 \* t0 - 2 \* t0 \* t3 - 2 \* t0 \* t1 - 2 \* t0 \* t0 +

t3 \* t1 + t3 \* t0 + t1 \* t0) / den2;

double n1 = (3 \* t0 \* t0 - 2 \* t0 \* t3 - 2 \* t0 \* t2 - 2 \* t0 \* t0 +

t3 \* t2 + t3 \* t0 + t2 \* t0) / den1;

double n0 = (3 \* t0 \* t0 - 2 \* t0 \* t3 - 2 \* t0 \* t2 - 2 \* t0 \* t1 +

t3 \* t2 + t3 \* t1 + t2 \* t1) / den0;

// Вторые производные

double o3 = (6 \* t0 - 2 \* t2 - 2 \* t1 - 2 \* t0) / den3;

double o2 = (6 \* t0 - 2 \* t3 - 2 \* t1 - 2 \* t0) / den2;

double o1 = (6 \* t0 - 2 \* t3 - 2 \* t2 - 2 \* t0) / den1;

double o0 = (6 \* t0 - 2 \* t3 - 2 \* t2 - 2 \* t1) / den0;

sigma\_mass\_mat.ResetValues();

chi\_mass\_mat.ResetValues();

stiff\_mat.ResetValues();

global.ResetValues();

for(int i = 0; i < nodes\_count; i++)

b[i] = 0;

BuildMatrices(t0);

sigma\_mass\_mat.MatVecMult(prev\_solution1, Mnqj\_1, sigma\_mass\_mat.bot\_tr, sigma\_mass\_mat.top\_tr);

sigma\_mass\_mat.MatVecMult(prev\_solution2, Mnqj\_2, sigma\_mass\_mat.bot\_tr, sigma\_mass\_mat.top\_tr);

sigma\_mass\_mat.MatVecMult(prev\_solution3, Mnqj\_3, sigma\_mass\_mat.bot\_tr, sigma\_mass\_mat.top\_tr);

chi\_mass\_mat.MatVecMult(prev\_solution1, Moqj\_1, chi\_mass\_mat.bot\_tr, chi\_mass\_mat.top\_tr);

chi\_mass\_mat.MatVecMult(prev\_solution2, Moqj\_2, chi\_mass\_mat.bot\_tr, chi\_mass\_mat.top\_tr);

chi\_mass\_mat.MatVecMult(prev\_solution3, Moqj\_3, chi\_mass\_mat.bot\_tr, chi\_mass\_mat.top\_tr);

for(int i = 0; i < nodes\_count; i++)

{

Mnqj\_1[i] \*= n1;

Mnqj\_2[i] \*= n2;

Mnqj\_3[i] \*= n3;

Moqj\_1[i] \*= o1;

Moqj\_2[i] \*= o2;

Moqj\_3[i] \*= o3;

}

sigma\_mass\_mat.Mult(n0);

chi\_mass\_mat.Mult(o0);

AssembleGlobalMatrix();

for(int i = 0; i < nodes\_count; i++)

b[i] -= Mnqj\_1[i] + Mnqj\_2[i] + Mnqj\_3[i] + Moqj\_1[i] + Moqj\_2[i] + Moqj\_3[i];

AccountFirstBound(t0);

Solve();

PrintSolution(fout, t0);

fout << endl;

prev\_solution3 = prev\_solution2;

prev\_solution2 = prev\_solution1;

prev\_solution1 = solution;

}

}

// Базисные функции для расчета решения не в узле сетки

double phi1(double t)

{

return 2 \* (t - 0.5) \* (t - 1);

}

double phi2(double t)

{

return -4 \* t \* (t - 1);

}

double phi3(double t)

{

return 2 \* t \* (t - 0.5);

}

// Вывод решения на временном слое t в поток fout

void PrintSolution(ofstream& fout, const double& t)

{

fout << "t = " << fixed << t << endl;

fout << setw(14) << "x" << setw(14) << "y";

fout << setw(14) << "prec" << setw(14) << "calc" << setw(14) << "diff" << setw(5) << "n" << " loc" << endl;

double norm = 0, norm\_u = 0;

for(int y\_i = 0; y\_i < y\_nodes\_count; y\_i++)

{

for(int x\_i = 0; x\_i < x\_nodes\_count; x\_i++)

{

int i = x\_i + y\_i \* x\_nodes\_count;

double prec = true\_solution[i];

double calc = solution[i];

//if(x\_i % 32 == 0 && y\_i % 32 == 0)

{

fout << scientific;

fout << setw(14) << x\_nodes[x\_i];

fout << setw(14) << y\_nodes[y\_i];

fout << setw(14) << prec;

fout << setw(14) << calc;

fout << setw(14) << abs(true\_solution[i] - solution[i]);

fout << fixed << setw(5) << i;

if(location[i] == 2)

fout << " outer";

else if(location[i] == 1)

fout << " border";

else

fout << " inner";

fout << endl;

}

norm\_u += prec \* prec;

norm += abs(prec - calc) \* abs(prec - calc);

}

}

// Блок вывода для вывода решения в произвольной точке расчетной области

/\* double x = 0.25;

double y = 0.25;

vector<double> s = solution;

double calc = 0;

calc += phi1(x) \* phi1(y) \* s[0];

calc += phi2(x) \* phi1(y) \* s[1];

calc += phi3(x) \* phi1(y) \* s[2];

calc += phi1(x) \* phi2(y) \* s[3];

calc += phi2(x) \* phi2(y) \* s[4];

calc += phi3(x) \* phi2(y) \* s[5];

calc += phi1(x) \* phi3(y) \* s[6];

calc += phi2(x) \* phi3(y) \* s[7];

calc += phi3(x) \* phi3(y) \* s[8];

double prec = regions[0].test.u(2, 2, 3);

fout << scientific;

fout << setw(14) << 2.0;

fout << setw(14) << 2.0;

fout << setw(14) << prec;

fout << setw(14) << calc;

fout << setw(14) << abs(prec - calc);

fout << fixed << setw(5) << "-";

fout << " point" << endl;\*/

fout << "||u-u\*||/||u\*|| = " << scientific << sqrt(norm) / sqrt(norm\_u) << endl;

fout << "||u-u\*||" << scientific << sqrt(norm) << endl;

}

};

* 1. **Файл *SLAE.h,* отвечающий за хранение основных данных и функций, реализующих решение системы методом сопряженных градиентов с диагональным предобуславливанием**

#pragma once

#include "Vector.h"

#include "Matrix.h"

using namespace std;

class SLAE

{

public:

int maxiter; // Максимальное количество итераций

double eps; // Велечина требуемой относительной невязки

vector<double> b; // Вектор правой части

vector<double> t; // Вспомогательный вектор для МСГ

vector<double> tt; // Вспомогательный вектор для МСГ

vector<double> rk1; // Вектор невязки на перд. итерации МСГ

vector<double> zk1; // Вектор спуска на пред. итерации МСГ

vector<double> AtAzk1; // Вспомогательный вектор для МСГ

SLAE(int size, int \_maxiter, double \_eps)

{

maxiter = \_maxiter;

eps = \_eps;

b.resize(size);

t.resize(size);

tt.resize(size);

rk1.resize(size);

zk1.resize(size);

AtAzk1.resize(size);

}

SLAE()

{

}

// Метод сопряженных градиентов, возвращает количество итераций

int ConjGradMethod(vector<double>& xk1, vector<double>& res, Matrix& mat)

{

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

res[i] = xk1[i] = 0;

mat.MatVecMult(xk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* x0

mat.MatVecMult(b - t, rk1, mat.top\_tr, mat.bot\_tr); // r0 = AT(f - A \* x0)

zk1 = rk1;

int k = 1;

while(k < maxiter)

{

mat.MatVecMult(zk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* zk-1

mat.MatVecMult(t, AtAzk1, mat.top\_tr, mat.bot\_tr); // AtAzk1 = At \* A \* zk-1

double ak = (rk1 \* rk1) / (AtAzk1 \* zk1);

xk1 = xk1 + ak \* zk1;

double bk = rk1 \* rk1;

rk1 = rk1 - ak \* AtAzk1;

bk = (rk1 \* rk1) / bk;

zk1 = rk1 + bk \* zk1;

double disc = Norm(rk1) / Norm(b); // Относительная невязка

if(disc < eps)

break;

else

k++;

}

res = xk1;

return k;

}

// Метод сопряженных градиентов с предобусловденной неполной

// факторизацией матрицей, возвращает количество итераций

int ConjGradPredMethod(vector<double>& xk1, vector<double>& res, Matrix& mat, SLAE& fac\_slae, Matrix& fac\_mat)

{

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

res[i] = xk1[i] = 0;

mat.MatVecMult(xk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* x0

mat.MatVecMult(b - t, rk1, mat.top\_tr, mat.bot\_tr); // r0 = AT(f - A \* x0)

// Решаем z0 = M-1 \* r0

fac\_slae.b = rk1;

fac\_slae.ConjGradMethod(t, zk1, fac\_mat);

int k = 1;

while(k < maxiter)

{

// Решаем tt = M-1 \* rk-1

fac\_slae.b = rk1;

fac\_slae.ConjGradMethod(t, tt, fac\_mat);

mat.MatVecMult(zk1, t, mat.bot\_tr, mat.top\_tr); // t = A \* zk-1

mat.MatVecMult(t, AtAzk1, mat.top\_tr, mat.bot\_tr); // AtAzk1 = At \* A \* zk-1

double ak = (tt \* rk1) / (AtAzk1 \* zk1);

xk1 = xk1 + ak \* zk1;

double bk = tt \* rk1;

rk1 = rk1 - ak \* AtAzk1;

// Решаем tt = M-1 \* rk

fac\_slae.b = rk1;

fac\_slae.ConjGradMethod(t, tt, fac\_mat);

bk = (tt \* rk1) / bk;

zk1 = tt + bk \* zk1;

double disc = Norm(rk1) / Norm(b);

if(disc < eps)

break;

else

k++;

}

res = xk1;

return k;

}

};

* 1. **Файл *Matrix.h,* отвечающий за хранение основных данных и функций, реализующих структуру матрицы в разреженном строчно-столбцовом виде**

#pragma once

#include <vector>

#include <fstream>

using namespace std;

class Matrix

{

public:

int size; // Размер матрицы

int tr\_size; // Количество элементов в треугольнике

vector<int> ind; // Указатели начала строк

vector<int> columns\_ind; // Номера столбцов внедиагональных элементов

vector<double> top\_tr; // Верхний треугольник

vector<double> bot\_tr; // Нижний треугольник

vector<double> diag; // Диагональ

// Конструктор для матрицы с известным количеством элементов в треуголниках

Matrix(const int& t\_size, const int& t\_tr\_size) : size(t\_size), tr\_size(t\_tr\_size)

{

top\_tr = vector<double>(tr\_size);

bot\_tr = vector<double>(tr\_size);

columns\_ind = vector<int>(tr\_size);

diag = vector<double>(size);

ind = vector<int>(size + 1);

}

// Конструктор для матрицы с полным заполнением треугольников

Matrix(const int& t\_size) : size(t\_size)

{

tr\_size = size \* (size - 1) / 2;

top\_tr.resize(tr\_size);

bot\_tr.resize(tr\_size);

columns\_ind.resize(tr\_size);

diag.resize(size);

ind.resize(size + 1);

ind[0] = ind[1] = 0;

diag[0] = 1.0;

for (int i = 1; i < size; i++)

{

int i0 = ind[i + 0];

int i1 = ind[i + 1] = ind[i + 0] + i;

for (int j = 0, k = i0; j < i; j++, k++)

columns\_ind[k] = j;

}

}

Matrix(const Matrix& mat)

{

size = mat.size;

tr\_size = mat.tr\_size;

top\_tr = mat.top\_tr;

bot\_tr = mat.bot\_tr;

diag = mat.diag;

ind = mat.ind;

columns\_ind = mat.columns\_ind;

}

Matrix()

{

}

// Получение диагональной факторизации матрицы

void DiagFact(Matrix& fact)

{

fact.diag = diag;

for (int i = 0; i < size + 1; i++)

fact.ind[i] = 0;

}

// Функция умножения матрицы на вектор vec, результат в res

void MatVecMult(const vector<double>& vec, vector<double>& res,

const vector<double>& bot\_tr, const vector<double>& top\_tr)

{

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

res[i] = 0;

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

{

res[i] += vec[i] \* diag[i];

int prof\_len = ind[i + 1] - ind[i];

for (int k = 0; k < prof\_len; k++)

{

int i\_in\_gg = ind[i] + k;

int j = columns\_ind[i\_in\_gg];

res[i] += vec[j] \* bot\_tr[i\_in\_gg];

res[j] += vec[i] \* top\_tr[i\_in\_gg];

}

}

}

// Функция умножения матрицы на вектор vec, результат в res

void MatVecMult(const vector<double>& vec, vector<double>& res)

{

MatVecMult(vec, res, bot\_tr, top\_tr);

}

// Функция считывания диагонали и треугольников из файлов

void ReadDiTr(const string& file\_name)

{

ifstream fin;

fin.open(file\_name);

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

fin >> diag[i];

for(int i = 0; i < tr\_size; i++)

{

double t;

fin >> t;

top\_tr[i] = bot\_tr[i] = t;

}

fin.close();

}

// Зануление всех элементов матрицы

void ResetValues()

{

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

diag[i] = 0;

for(int i = 0; i < tr\_size; i++)

{

bot\_tr[i] = 0;

top\_tr[i] = 0;

}

}

// Умножение матрицы на число

void Mult(const double& val)

{

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

diag[i] \*= val;

for(int i = 0; i < tr\_size; i++)

{

bot\_tr[i] \*= val;

top\_tr[i] \*= val;

}

}

};

* 1. **Файл *Region.h,* отвечающий за хранение информации о подобласти расчетной области**

class Region

{

public:

int left, right, bot, top; // Индексы границы региона в массивах сетки

Test test; // Информация о тестовой функции

};

* 1. **Файл *Vector.h,* отвечающий за хранение перегрузок операторов работы с векторами**

#pragma once

#include <vector>

using namespace std;

// Умножение вектора на число

vector<double> operator \* (const double& val, vector<double> vec)

{

for (size\_t i = 0; i < vec.size(); i++)

vec[i] \*= val;

return vec;

}

// Сложение векторов

vector<double> operator + (vector<double> vec1, const vector<double>& vec2)

{

for (size\_t i = 0; i < vec1.size(); i++)

vec1[i] += vec2[i];

return vec1;

}

// Вычитание векторов

vector<double> operator - (vector<double> vec1, const vector<double>& vec2)

{

for(size\_t i = 0; i < vec1.size(); i++)

vec1[i] -= vec2[i];

return vec1;

}

// Скалярное произведение векторов

double ScalarMult(const vector<double>& vec1, const vector<double>& vec2)

{

double res = 0;

for(int i = 0; i < vec1.size(); i++)

res += vec1[i] \* vec2[i];

return res;

}

// Скалярное произведение векторов

double operator \* (const vector<double>& vec1, const vector<double>& vec2)

{

return ScalarMult(vec1, vec2);

}

// Норма вектора

double Norm(const vector<double>& vec)

{

return sqrt(ScalarMult(vec, vec));

}

* 1. **Файл *Test.h,* отвечающий за хранение информации о тестовых функциях и коэффициентов уравнения на подобласти**

#pragma once

#pragma once

using namespace std;

class Test

{

public:

int test\_n = 0; // Номер теста

int lambda\_n = 0; // Номер формулы для коэффициента lambda

int sigma\_n = 0; // Номер формулы для коэффициента sigma

int chi\_n = 0; // Номер формулы для коэффициента chi

Test(const int& t\_N) : test\_n(t\_N) {};

Test() { };

double f(const double& x, const double& y, const double& t)

{

return -1 \* divlambdagrad(x, y, t) +

sigma() \* dudt(x, y, t) + chi() \* d2udt2(x, y, t);

}

double lambda(const double& x, const double& y)

{

switch(lambda\_n)

{

case 0: return 1;

case 1: return 2;

}

}

double sigma()

{

switch(sigma\_n)

{

case 0: return 1;

case 1: return 4;

}

}

double chi()

{

switch(chi\_n)

{

case 0: return 1;

case 1: return 0.3;

}

}

// Точное решение

double u(const double& x, const double& y, const double& t)

{

switch(test\_n)

{

case 0: return 2.0;

case 1: return x + y;

case 2: return x \* x + y \* y;

case 3: return x \* x \* x + y \* y \* y;

case 4: return x \* x \* x \* x + y \* y \* y \* y;

case 5: return t \* t \* t \* t;

case 6: return t \* t \* t;

case 7: return x \* x \* t \* t \* t \* t;

};

}

double divlambdagrad(const double& x, const double& y, const double& t)

{

switch(test\_n)

{

case 0: return 0;

case 1: return 0;

case 2: return 4;

case 3: return 6 \* x + 6 \* y;

case 4: return 12 \* x \* x + 12 \* y \* y;

case 5: return 0;

case 6: return 0;

case 7: return 2 \* t \* t \* t \* t;

};

}

// Производная точного решения по t

double dudt(const double& x, const double& y, const double& t)

{

switch(test\_n)

{

case 0: return 0;

case 1: return 0;

case 2: return 0;

case 3: return 0;

case 4: return 0;

case 5: return 4 \* t \* t \* t;

case 6: return 3 \* t \* t;

case 7: return 4 \* t \* t \* t \* x \* x;

};

}

// Вторая производная точного решения по t

double d2udt2(const double& x, const double& y, const double& t)

{

switch(test\_n)

{

case 0: return 0;

case 1: return 0;

case 2: return 0;

case 3: return 0;

case 4: return 0;

case 5: return 12 \* t \* t;

case 6: return 6 \* t;

case 7: return 12 \* t \* t \* x \* x;

};

}

};

* 1. **Файл *Main.h,* отвечающий за последовательное выполнение необходимых для решения задачи функций**

#include <iostream>

#include "BoundaryValueProblem.h"

using namespace std;

int main()

{

BoundaryValueProblem bvp;

bvp.ReadFormGrid("data/grid.txt");

bvp.ReadBoundaries("data/boundaries.txt");

bvp.ReadFormTimeGrid("data/time.txt");

bvp.ReadMatrices();

bvp.InitializeMemory();

bvp.FormPortrait();

ofstream fout("result.txt");

bvp.TimeIterations(fout);

fout.close();

}