Министерство образования и науки Российской Федерации

Новосибирский государственный технический университет

Кафедра прикладной математики

Курсовой проект

по дисциплине: «**Уравнения математической физики**»

Факультет: ПМИ

Группа: ПМ-43

Студентка: Нестёркина А. А.

Преподаватели: Задорожный А. Г.

Персова М. Г.

Вариант: параболическая задача,

неявная трёхслойная схема по времени

Новосибирск

2017

Оглавление

[Цель работы 3](#_Toc485117293)

[Постановка задачи 3](#_Toc485117294)

[Теоретическая часть 4](#_Toc485117295)

[*Дискретизация по времени* 4](#_Toc485117296)

[*Вариационная постановка* 5](#_Toc485117297)

[*Конечноэлементная дискретизация* 7](#_Toc485117298)

[*Матричный вид постановки* 9](#_Toc485117299)

[Описание разработанных программ 11](#_Toc485117300)

[Структуры данных, используемые для задания расчётной области и конечноэлементной сетки 11](#_Toc485117301)

[Структуры основных модулей программы 12](#_Toc485117302)

[Тестирование 16](#_Toc485117303)

[Тестирование на стационарных задачах 16](#_Toc485117304)

[Тестирование на нестационарных задачах 19](#_Toc485117305)

[Исследования 22](#_Toc485117306)

[Исследования на стационарной задаче 22](#_Toc485117307)

[Исследования на нестационарной задаче 23](#_Toc485117308)

[Код программы 26](#_Toc485117309)

# Цель работы

Приобрести навыки численного решения начально-краевых задач для уравнений гиперболического и параболического типа в неоднородных одномерных, двумерных и трехмерных областях с помощью метода конечных элементов при использовании различных схем дискретизации по времени.

# Постановка задачи

Гиперболическая краевая задача для функции *u* определяется дифференциальным уравнением

заданным в области с границей , и краевыми условиями

в которых - значение искомой функции *u* на границе , а - значение на производной функции *u* по направлению внешней нормали к поверхности .

Вид разностной схемы по времени: неявная, трёхслойная.

Вид конечных элементов: прямоугольники.

Вид базисных функций: биквадратичные.

Формат хранения матрицы СЛАУ: разреженный строчно-столбцовый.

Решатель: ЛОС с неполной факторизацией.

# Теоретическая часть

## *Дискретизация по времени*

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

где – значения функции при соответственно, а – квадратичные полиномы Лагранжа.

Данные функции времени выглядят следующим образом:

где

Применим (5) для аппроксимации производной по времени в (1) на *j*-ом временном слое:

Вычислим первые производные от функций в точке :

С учётом (7) можно переписать (6) в виде:

После выполнения конечно-элементной аппроксимации получим СЛАУ вида:

## *Вариационная постановка*

Схема построения вариационной формулировки в форме уравнения Галёркина выглядит следующим образом. Пусть необходимо решить краевую задачу для дифференциального уравнения

Тогда обе части уравнения (8) домножим на функцию *v* из пространства пробных функций и проинтегрируем по Ω, что фактически соответствует скалярному умножению *Lu* и *f* на *v* в пространстве *L2*(Ω):

Уравнение (9) можно записать в виде

что означает ортогональность невязки уравнения (8) пространству .

Далее в уравнении (9) необходимо преобразовать его левую часть и использованием формулы Грина для понижения порядка дифференцирования функции *uj* и затем некоторые полученные после преобразования поверхностные интегралы преобразовать с использованием краевых условий.

Покажем более подробно*.*

Потребуем ортогональность невязки некоторому пространству пробных функций *v*:

Воспользовавшись формулой Грина, получим:

где (граница ).

Интегралы по границам и можно преобразовать с учётом краевых условий. На границе краевыми условиями значение не определяется. Поэтому это слагаемое исключается из уравнения на основании того требования, чтобы пространство пробных функций содержало только функции, которые принимали бы нулевые значения на границе (обозначим эти функции ).

В уравнение входят производные пробных функций *v*, поэтому в качестве пространства можем брать пространство .

Таким образом, получим вариационное уравнение вида:

При построении конечноэлементных аппроксимаций по методу Галёркина пространства и заменяются конечномерными пространствами и , которые являются элементами одного и того же конечномерного пространства . Это пространство аппроксимирует гильбертово пространство и имеет базис из кусочно-полиномиальных функций Элементами пространств и являются такие линейные базисных функций пространства , в которых зафиксированы значения коэффициентов перед базисными функциями, не равными нулю на границе (с заданными на ней краевыми условиями первого рода). Коэффициенты фиксируются так, чтобы в узлах сетки на границе выполнялось условие , а в узлах сетки на границе выполнялось условие

Построим аппроксимацию уравнения (7) на конечномерных подпространствах и , описанных выше. Для этого заменим в (7) функцию функцией , а функцию функцией :

справедливо представление:

где - множество индексов *i* таких, что базисные функции пространства являются и базисными функциями пространств и . Поэтому вариационное уравнение (8) эквивалентно системе уравнений:

МКЭ-решение удовлетворяет системе (12). Так как , то может быть представлено в виде:

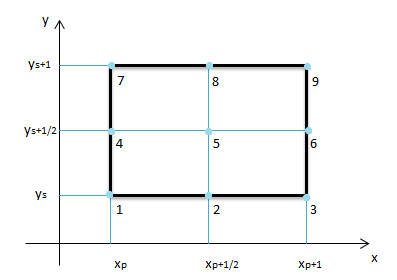
при этом *n-n0* (*n0 -* размерность пространств и ) компонент вектора **q**= должны быть фиксированы и могут быть определены из условия .

Подставив (13) в (12), получим СЛАУ для компонент вектора **q**, :

## *Конечноэлементная дискретизация*

Наиболее часто используемый способ определения явного вида биквадратичных базисных функций на прямоугольном конечном элементе – это представление каждой базисной функции в виде произведений одномерных квадратичных базисных функций координат x и y.

Расположение узлов на прямоугольном конечном элементе с биквадратичными базисными функциями:

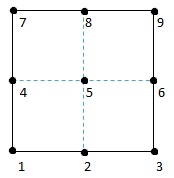


Локальные базисные функции строятся для шаблонного конечного элемента

Отображение шаблонного элемента в четырёхугольник , описанный выше, можно осуществить с помощью следующих соотношений:

Базис строится из одномерных квадратичных функций следующего вида:

; ;  
; ;  
;

Шаблонный конечный элемент имеет следующий вид:

Тогда локальные базисные функции на шаблонном элементе с показанной на рисунке нумерацией узлов имеют следующий вид:

Базисные функции на прямоугольнике можно получить, используя соотношения (15)-(16).

Приведём способ получения  на элементе из *x,y.* Якобиан преобразования координат (15)-(16):

и могут быть соответственно вычислены следующим образом :

## *Матричный вид постановки*

В матричном виде решаемая система имеет следующий вид:

**Aq=b**, где

Для удобства вычисление компонент матрицы разбивают на вычисления каждого из интегралов отдельно, то есть на каждом элементе вводятся локальные матрицы жёсткости (G), массы (M), компоненты которых вычисляются следующим образом.

Введём обозначение .

Элементы локальной матрицы массы вычисляются следующим образом:

Элементы локальной матрицы массы вычисляются следующим образом:

Используя правила дифференцирования сложной функции можно получить выражения для расчёта приведённых выше производных. Они имеют следующий вид:

Элементы локального вектора правой части вычисляются следующим образом:

В программе интегрирование реализовано численно с использованием квадратурных формул Гаусса-5.

Веса и точки одномерного интегрирования на [-1,1]:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Точки и веса для двумерного интегрирования получаются следующим образом:

# Описание разработанных программ

## Структуры данных, используемые для задания расчётной области и конечноэлементной сетки

Для заданной расчётной области используются следующие структуры данных:

* Структура **Area** (подобласть) содержит в себе целочисленные компоненты:  
    
  **leftX** – номер левой границы подобласти в массиве координатных линий по x;   
  **rightX**– номер правой границы подобласти в массиве координатных линий по x;  
  **lowY** – номер нижней границы подобласти в массиве координатных линий по у;   
  **upY** – номер нижней границы подобласти в массиве координатных линий по у.   
    
  Также структура содержит два целочисленных массива размерности, равной четырём:  
    
  **ku** – массив с типами краевых условий (1,2,3,-1),определённых на соответствующих границах(индексы массива определяют сами границы, нумерация идёт слева направо, снизу вверх);  
  **kuForm** – массив с номерами тестовых формул, по которым определяются краевые условия.  
  \*Информация считывается из файла
* Структура **AreasLines** (координатные линии подобластей) содержит в себе вещественные массивы значений всех вертикальных и горизонтальных границ подобластей:  
  **x** – массив значений вертикальных границ;  
  **y** – массив значений горизонтальных границ.  
    
  \*Информация считывается из файла
* Структура **Element** содержит в себе информацию об одном конечном элементе, о номере его подобласти, в которой располагается данный конечный элемент, о глобальной нумерации узлов конечного элемента в виде целочисленного массива размерности, равной четырём, о глобальной нумерации базисных функций в виде целочисленного массива размерности девять (степени свободы). Также в виде целочисленного массива представлены соседние элементы:  
    
  **numberOfArea** – номер подобласти, в которой лежит конечный элемент;  
  **nodes** – массив глобальной нумерации узлов конечного элемента;  
  **dofs** – массив глобальной нумерации базисных функций (степени свободы);  
  **neighbors** – массив номеров конечных элементов, являющихся соседними с данным конечным элементов по границам (индексы массива определяют сами границы, нумерация идёт слева направо, снизу вверх).
* Структура **BoundaryCondition** (краевое условие) содержит в себе следующую информацию о краевых условиях на границах для конечного элемента:  
    
  **elem** – номер конечного элемента, для которого заданы данные краевые условия  
  **edges** – целочисленный массив размерности четыре, содержащий информацию о том есть ли на соответствующей границе краевое условие (1) или нет (0);  
  **formNumber** – целочисленный массив размерности четыре, содержащий номера формул для расчёта краевых условий.
* Структура **Point** (точка) содержит в себе 2 целочисленных числа **x** и **y**, хранящих значения соответствующей точки в декартовых координатах.
* Класс **Grid** (сетка) содержит в себе следующие поля:  
    
  **areas** – вектор подобластей;  
  **areasLines** – координатные линии, образующие подобласти;  
  **elements** – вектор, содержащий все конечные элементы, после построения сетки;  
  **nodes** – вектор узлов, содержащий координаты всех точек, после построения сетки;

**time** – вектор узлов, содержащий значения всех временных слоёв, после построения сетки по времени;  
**ku** – массив (размерности три) краевых условий конечных элементов после построения сетки;  
**nx** – общее количество интервалов по x после построения сетки;  
**ny** – общее количество интервалов по y после построения сетки.

## Структуры основных модулей программы

* **void BuildGrid();**   
    
   Процедура, реализующая построение сетки путём разбиения каждой подобласти по введённой в файл «Intervals.txt» информации о количестве частей для каждого интервала по x и y и о том, во сколько раз сгущается сетка на интервале.  
   При вводе в специальных массивах xIntervals, yIntervals сохраняется информация о том, на сколько частей нужно разбить каждый интервал, а в массивах xCoefficient и yCoefficient – их соответствующие коэффициенты сгущения. Затем вычисляется суммарное количество интервалов по x и по y, определяется размерность сетки по x и по y и заносится в поля nx, ny соответственно структуры grid. Далее проходом по координатным линиям areasLines для х и y определяются координаты разбиения и заносятся в массивы xi, yj (вызов процедуры PartitionСoordinate). После из массивов xi и yj собираются узлы и заносятся в поле nodes в виде точек (процедура PushNode).
* **int GetGlobalNumber(int elementNumber, int localNumber);**  
   Функция, отвечающая за получение глобальной нумерации узлов сетки.  
   По числу интервалов по х определяем номер горизонтальной линии, которая является нижним ребром элемента, путём нахождения целой части от деления номера соответствующего элемента и числа интервалов. После определяем номер начального узла на ребре, умножая номер горизонтальной линии на число координат по x. Далее, находим отступ от начального узла (остаток от деления номера элемента на количество интервалов по х) для того, чтобы получить текущий номер левого нижнего узла конечного элемента. После получения глобального номера первого локального узла не трудно пронумеровать остальные путём добавления единицы для каждого последующего.
* **int GetGlobalFuncNumber(int elNum, int localFuncNum);**  
   Функция, отвечающая за получение глобальной нумерации узлов базисных функций конечных элементов.  
   По числу интервалов по х определяем номер горизонтальной линии, которая является нижней гранью элемента, путём нахождения целой части от деления номера соответствующего элемента и числа интервалов. После определяем количество узлов на одной горизонтальной линии (разница удвоенного числа координат и 1). Определив номер начального узла на ребре (произведение удвоенного числа узлов на горизонтальной линии и номера горизонтальной линии) и отступ от начального узла (удвоенный остаток от деления номера элемента на количество интервалов по х), их суммой получим глобальный номер первого локального узла конечного элемента. Далее, не трудно глобально пронумеровать и остальные локальные узлы.
* **int FindArea(double x, double y**);  
    
   Функция, возвращающая номер подобласти в которой находится заданная точка с координатами x и y.  
   Перебирая все подобласти, до тех пор, пока не нашли искомую подобласть и не перебрали их до конца, идёт проверка на то, лежит точка в i-ой подобласти или нет.
* **void FindNeighbors(int elementNumber);**  
    
   Процедура поиска соседних конечных элементов для заданного.   
   Формируется массив с четырьмя булевскими переменными, которые изначально принимают значение «ложь». Далее, до тех пор, пока не нашли 4 соседних элемента и не проверили все элементы, кроме заданного, сверяем узлы соответствующих рёбер элементов. Если ребро совпадает, то соответствующая ему булевская переменная принимает значение «истина» а в поле элемента neighbors на соответствующее место встанет номер соседнего элемента. Если нет соседнего элемента по какой-либо границе, то на соответствующее место поле neighbors кладётся значение «-1».
* **void ComputeElements();** Процедура, формирующая массив конечных элементов из массива узлов, полученных после построения сетки.   
   Вычисляем глобальные номера узлов и базисных функции каждого конечного элемента и заполняем список конечных элементов. Далее, для каждого конечного элемента находим номер его подобласти по координате, находящейся в центре прямоугольника. После для каждого конечного элемента находим соседние.
* **void FormKU();**  
   Процедура, реализующая формирование массива краевых условий.  
   Для каждого конечного элемента находим, какие краевые условия заданы на его границах. Каждому краевому условию ставим в соответствие номер элемента, массив, определяющий есть ли на определённой границе краевое условие или нет, и массив с номером формулы для краевого условия на этой границе. Далее, в ячейку с номером соответствующего краевого условия кладём полученные результаты.
* **void SearchElements(int dofsNumber, vector<int>& elList);**  
    
   Процедура, формирующая список элементов, содержащих глобальный номер базисной функции, равный dofsNumber.
* **void CreatePortret(int slaeSize, Grid grid);**  
    
   Процедура генерации портрета глобальной матрицы.  
   Осуществляется проход по всем базисным функциям. Генерируем список элементов, в которых присутствует i-ая базисная функция. Собираем список узлов этих элементов, по глобальному номеру меньших номера базисной функции. Сортируем этот список по возрастанию. Вычисляем размерность нижнего и верхнего треугольника, равную сумме числа элементов в каждом таком списке. Инициализируем матрицу (выделяем соответствующую память под все компоненты) и собираем портрет. Внешний цикл проходит по всем глобальным базисным функциям, в нём формируются элементы ig путём прибавления числа элементов в i-ом списке к значению предыдущего элемента ig . Внутренний цикл проходит по списку элементы в массив jg берутся из списка.
* **void CalculateG(int elementNumber);**  
    
   Процедура, позволяющая выполнить сборку локальной матрицы жёсткости.  
   Производится двойное численное интегрирование соответствующей подынтегральной функции с использованием квадратурных формул Гаусса-5 по каждому базисному узлу.
* **void CalculateM(int elementNumber) и void CalculateMg(int elementNumber);**  
    
   Процедура, позволяющая выполнить сборку локальной матрицы массы.  
   Производится двойное численное интегрирование соответствующей подынтегральной функции с использованием квадратурных формул Гаусса-5 по каждому базисному узлу.
* **void CalculateLocalF(int elementNumber);**  
   Процедура, позволяющая выполнить сборку локальной правой части.  
   Производится численное интегрирование соответствующей подынтегральной функции с использованием квадратурных формул Гаусса-5 по каждому базисному узлу.
* **void AddElementToGlobalMatrix(Matrix &B, int i, int j, double element);**  
    
   Процедура, позволяющая добавить вещественное значение на i-ую строку в j-ый столбец глобальной матрицы B.
* **void CalculateLocals(int elementNumber);**  
    
   Процедура сборки локальных матриц и правой части, добавление в глобальные.  
   Вычисляются локальные матрицы жёсткости и масс, а также локальный вектор правой части. После по глобальным номерам базисных функций идёт добавление локальных частей в глобальные. Также осуществляется сборка глобальных матриц масс для последующих добавок в вектор правой части.
* **void CalculateBoundaries1ForNode(int node, double gi, double weight);**  
    
   Процедура осуществляет вычисление первого краевого условия (главного) для одного узла.  
    
   Идёт зануление внедиагональных элементов строки node, диагональный элемент на этой строке становится равным weight, а соответствующая этой строке правая часть становится равной gi.
* **void CalculateBoundaries1(int number);**  
   Процедура, осуществляющая учёт первых краевых условий для заданного конечного элемента.  
   На границе элемента, где задано первое краевое условие, производится вызов процедуры CalculateBoundaries1ForNode по узлам базисных функций, лежащих на этой границе.
* **void GenerateSLAE()**

Процедура, осуществляющая сборку глобальной матрицы и правой части, через процедуру CalculateLocals. Совершает добавки в правую часть от предыдущих решений по времени, а так же учитывает краевые условия.

* **void LULOS();** Процедура, реализующая метод LOS с LU предобусловдиванием. Отвечает за решение СЛАУ.  
    
   Сначала происходит LU-разложение матрицы, а затем последовательными приближениями решается СЛАУ.

# Тестирование

## Тестирование на стационарных задачах

**Тест №1** (на полиноме 1 степени)

Цель теста: стационарная задача, решение – полином первой степени

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2x-3y+1 | | | | | |
| lambda: | 1 | sigma: | 1 | gamma: | 1 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 1 | 1 | 0.00E+00 |
|  | 1 | 0 | 3 | 3 | 0.00E+00 |
|  | 2 | 0 | 5 | 5 | 0.00E+00 |
|  | 0 | 1 | -2 | -2 | 0.00E+00 |
|  | 1 | 1 | 0 | 0 | 0.00E+00 |
|  | 2 | 1 | 2 | 2 | 0.00E+00 |
|  | 0 | 2 | -5 | -5 | 0.00E+00 |
|  | 1 | 2 | -3 | -3 | 0.00E+00 |
|  | 2 | 2 | -1 | -1 | 0.00E+00 |

Тест завершён успешно, погрешности не обнаружены.

**Тест №2** (на полиноме 2 степени)

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Цель теста: стационарная задача, решение –

полином второй степени

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2x^2-3y^2+1 | | | | | |
| lambda | 2 | sigma | 1 | gamma | 3 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 1 | 1 | 0.00E+00 |
|  | 1 | 0 | 3 | 3 | 0.00E+00 |
|  | 2 | 0 | 9 | 9 | 0.00E+00 |
|  | 0 | 1 | -2 | -2 | 0.00E+00 |
|  | 1 | 1 | 0 | 0 | 0.00E+00 |
|  | 2 | 1 | 6 | 6 | 0.00E+00 |
|  | 0 | 2 | -11 | -11 | 0.00E+00 |
|  | 1 | 2 | -9 | -9 | 0.00E+00 |
|  | 2 | 2 | -3 | -3 | 0.00E+00 |

Тест завершён успешно, погрешности не обнаружены.

**Тест №3** (на полиноме 3 степени)

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Цель теста: стационарная задача, решение –

полином третьей степени

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2x^3-3y^3+1 | | | | | |
| lambda | 1 | sigma | 1 | gamma | 1 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 1 | 1 | 0.00E+00 |
|  | 1 | 0 | 3 | 3 | 0.00E+00 |
|  | 2 | 0 | 17 | 17 | 0.00E+00 |
|  | 0 | 1 | -2 | -2 | 0.00E+00 |
|  | 1 | 1 | 0 | 0 | 0.00E+00 |
|  | 2 | 1 | 14 | 14 | 0.00E+00 |
|  | 0 | 2 | -23 | -23 | 0.00E+00 |
|  | 1 | 2 | -21 | -21 | 0.00E+00 |
|  | 2 | 2 | -7 | -7 | 0.00E+00 |

Тест завершён успешно, погрешности не обнаружены.

**Тест №4** (на полиноме 4 степени)

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Цель теста: стационарная задача, решение –

полином четвёртой степени

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2x^4-3y^4+1 | | | | | |
| lambda | 1 | sigma | 1 | gamma | 4 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 1 | 1 | 0.00E+00 |
|  | 1 | 0 | 3 | 3 | 0.00E+00 |
|  | 2 | 0 | 17 | 33 | 1.60E+01 |
|  | 0 | 1 | -2 | -2 | 0.00E+00 |
|  | 1 | 1 | 0 | 0.151846 | 1.52E-01 |
|  | 2 | 1 | 14 | 30 | 1.60E+01 |
|  | 0 | 2 | -23 | -47 | 2.40E+01 |
|  | 1 | 2 | -21 | -45 | 2.40E+01 |
|  | 2 | 2 | -7 | -15 | 8.00E+00 |

Тест завершён успешно, обнаружена погрешность во внутреннем узле. Соответствующие исследования на порядок аппроксимации и сходимости проведём позднее.

## Тестирование на нестационарных задачах

**Тест №1** (на полиноме 1 степени)

Цель теста: нестационарная задача, решение – полином первой степени по времени

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2t | | | | | |
| lambda | 1 | sigma | 1 | gamma | 1 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 4 | 4 | 0.00E+00 |
|  | 1 | 0 | 4 | 4 | 0.00E+00 |
|  | 2 | 0 | 4 | 4 | 0.00E+00 |
|  | 0 | 1 | 4 | 4 | 0.00E+00 |
|  | 1 | 1 | 4 | 4 | 0.00E+00 |
|  | 2 | 1 | 4 | 4 | 0.00E+00 |
|  | 0 | 2 | 4 | 4 | 0.00E+00 |
|  | 1 | 2 | 4 | 4 | 0.00E+00 |
|  | 2 | 2 | 4 | 4 | 0.00E+00 |
| 3 | 0 | 0 | 6 | 6 | 0.00E+00 |
|  | 1 | 0 | 6 | 6 | 0.00E+00 |
|  | 2 | 0 | 6 | 6 | 0.00E+00 |
|  | 0 | 1 | 6 | 6 | 0.00E+00 |
|  | 1 | 1 | 6 | 6 | 0.00E+00 |
|  | 2 | 1 | 6 | 6 | 0.00E+00 |
|  | 0 | 2 | 6 | 6 | 0.00E+00 |
|  | 1 | 2 | 6 | 6 | 0.00E+00 |
|  | 2 | 2 | 6 | 6 | 0.00E+00 |
| 4 | 0 | 0 | 8 | 8 | 0.00E+00 |
|  | 1 | 0 | 8 | 8 | 0.00E+00 |
|  | 2 | 0 | 8 | 8 | 0.00E+00 |
|  | 0 | 1 | 8 | 8 | 0.00E+00 |
|  | 1 | 1 | 8 | 8 | 0.00E+00 |
|  | 2 | 1 | 8 | 8 | 0.00E+00 |
|  | 0 | 2 | 8 | 8 | 0.00E+00 |
|  | 1 | 2 | 8 | 8 | 0.00E+00 |
|  | 2 | 2 | 8 | 8 | 0.00E+00 |

Тест завершён успешно, погрешности не обнаружены.

**Тест №2** (на полиноме 2 степени)

Цель теста: нестационарная задача, решение – полином второй степени по времени

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2t^2 | | | | | |
| lambda | 1 | sigma | 1 | gamma | 1 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 8 | 8 | 0.00E+00 |
|  | 1 | 0 | 8 | 8 | 0.00E+00 |
|  | 2 | 0 | 8 | 8 | 0.00E+00 |
|  | 0 | 1 | 8 | 8 | 0.00E+00 |
|  | 1 | 1 | 8 | 8 | 0.00E+00 |
|  | 2 | 1 | 8 | 8 | 0.00E+00 |
|  | 0 | 2 | 8 | 8 | 0.00E+00 |
|  | 1 | 2 | 8 | 8 | 0.00E+00 |
|  | 2 | 2 | 8 | 8 | 0.00E+00 |
| 3 | 0 | 0 | 18 | 18 | 0.00E+00 |
|  | 1 | 0 | 18 | 18 | 0.00E+00 |
|  | 2 | 0 | 18 | 18 | 0.00E+00 |
|  | 0 | 1 | 18 | 18 | 0.00E+00 |
|  | 1 | 1 | 18 | 18 | 0.00E+00 |
|  | 2 | 1 | 18 | 18 | 0.00E+00 |
|  | 0 | 2 | 18 | 18 | 0.00E+00 |
|  | 1 | 2 | 18 | 18 | 0.00E+00 |
|  | 2 | 2 | 18 | 18 | 0.00E+00 |
| 4 | 0 | 0 | 32 | 32 | 0.00E+00 |
|  | 1 | 0 | 32 | 32 | 0.00E+00 |
|  | 2 | 0 | 32 | 32 | 0.00E+00 |
|  | 0 | 1 | 32 | 32 | 0.00E+00 |
|  | 1 | 1 | 32 | 32 | 9.95E-14 |
|  | 2 | 1 | 32 | 32 | 0.00E+00 |
|  | 0 | 2 | 32 | 32 | 0.00E+00 |
|  | 1 | 2 | 32 | 32 | 0.00E+00 |
|  | 2 | 2 | 32 | 32 | 0.00E+00 |

Тест завершён успешно, погрешности не обнаружены.

**Тест №3** (на полиноме 3 степени)

Цель теста: нестационарная задача, решение – полином третьей степени по времени

x

y

0

1

2

0

1

2

0

1

2

3

4

5

6

7

8

Расчетная область по пространству: [0,2]x[0,2]

Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

1 конечный элемент, на всех границах заданы первые краевые условия.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| u = 2t^3 | | | | | |
| lambda | 1 | sigma | 1 | gamma | 1 |
| t | x | y | u\* | u | |u\*-u| |
| 2 | 0 | 0 | 16 | 16 | 0.00E+00 |
|  | 1 | 0 | 16 | 16 | 0.00E+00 |
|  | 2 | 0 | 16 | 16 | 0.00E+00 |
|  | 0 | 1 | 16 | 16 | 0.00E+00 |
|  | 1 | 1 | 16 | 16.83333 | 8.33E-01 |
|  | 2 | 1 | 16 | 16 | 0.00E+00 |
|  | 0 | 2 | 16 | 16 | 0.00E+00 |
|  | 1 | 2 | 16 | 16 | 0.00E+00 |
|  | 2 | 2 | 16 | 16 | 0.00E+00 |
| 3 | 0 | 0 | 54 | 54 | 0.00E+00 |
|  | 1 | 0 | 54 | 54 | 0.00E+00 |
|  | 2 | 0 | 54 | 54 | 0.00E+00 |
|  | 0 | 1 | 54 | 54 | 0.00E+00 |
|  | 1 | 1 | 54 | 55.05556 | 1.06E+00 |
|  | 2 | 1 | 54 | 54 | 0.00E+00 |
|  | 0 | 2 | 54 | 54 | 0.00E+00 |
|  | 1 | 2 | 54 | 54 | 0.00E+00 |
|  | 2 | 2 | 54 | 54 | 0.00E+00 |
| 4 | 0 | 0 | 128 | 128 | 0.00E+00 |
|  | 1 | 0 | 128 | 128 | 0.00E+00 |
|  | 2 | 0 | 128 | 128 | 0.00E+00 |
|  | 0 | 1 | 128 | 128 | 0.00E+00 |
|  | 1 | 1 | 128 | 129.0593 | 1.06E+00 |
|  | 2 | 1 | 128 | 128 | 0.00E+00 |
|  | 0 | 2 | 128 | 128 | 0.00E+00 |
|  | 1 | 2 | 128 | 128 | 0.00E+00 |
|  | 2 | 2 | 128 | 128 | 0.00E+00 |

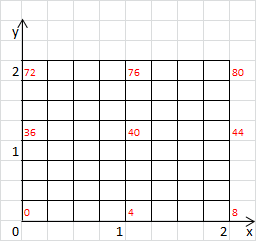
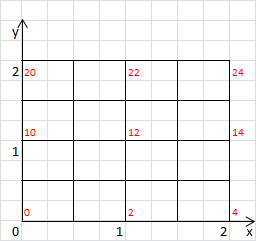
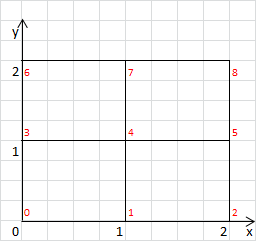
Тест завершён успешно, обнаружена погрешность во внутреннем узле. Соответствующие исследования на порядок сходимости проведём позднее.

# Исследования

## Исследования на стационарной задаче

**Исследование на вложенных равномерных сетках.**

Расчетная область по пространству: [0,2]x[0,2]



Расчетная область по времени: {0,1,2,3,4}

Искомое решение:

Правая часть СЛАУ:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | y | u\* | uh | uh/2 | uh/4 | |u\*-uh| | |u\*-uh/2| | |u\*-uh/4| |
| 0 | 0 | 1 | 1 | 1 | 1 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 1 | 0 | 3 | 3 | 3 | 3 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 2 | 0 | 33 | 33 | 33 | 33 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 0 | 1 | -2 | -2 | -2 | -2 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 1 | 1 | 0 | 0.1518465 | -0.050005682 | -0.020897203 | 1.52E-01 | 5.00E-02 | 2.09E-02 |
| 2 | 1 | 30 | 30 | 30 | 30 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 0 | 2 | -47 | -47 | -47 | -47 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 1 | 2 | -45 | -45 | -45 | -45 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 2 | 2 | -15 | -15 | -15 | -15 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| Погрешность ||u\*-ui||h | | | 1.52E-01 | log2(||u-u1/2|| / ||u1/2-u1/4||) = | | 2.79E+00 | |  |
| Погрешность ||u\*-ui||h/2 | | | 5.00E-02 |  |
| Погрешность ||u\*-ui||h/4 | | | 2.09E-02 |  |

Исследования показали приблизительно 2 порядок сходимости.

## Исследования на нестационарной задаче

**Исследование на вложенных равномерных сетках.**



Расчетная область по пространству: [0,2]x[0,2]

Искомое решение:



Правая часть СЛАУ:



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| t | x | y | u\* | uh | uh/2 | uh/4 | |u\*-uh| | |u\*-uh/2| | |u\*-uh/4| |
| 2 | 0 | 0 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 0 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 0 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 1 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 1 | 16 | 16.83333333333330 | 16.26577503429350 | 16.06523254422430 | 8.33E-01 | 2.66E-01 | 6.52E-02 |
|  | 2 | 1 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 2 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 2 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 2 | 16 | 16.00000000000000 | 16.00000000000000 | 16.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 3 | 0 | 0 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 0 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 0 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 1 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 1 | 54 | 55.05555555555550 | 54.26135603481850 | 54.06510001489640 | 1.06E+00 | 2.61E-01 | 6.51E-02 |
|  | 2 | 1 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 2 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 2 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 2 | 54 | 54.00000000000000 | 54.00000000000000 | 54.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| 4 | 0 | 0 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 0 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 0 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 1 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 1 | 128 | 129.05925925925900 | 128.26032731970400 | 128.06510424700700 | 1.06E+00 | 2.60E-01 | 6.51E-02 |
|  | 2 | 1 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 2 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 2 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 2 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| Погрешность ||u\*-ui||h | |  | 1.71E+00 | log2(||u-u1/2|| / ||u1/2-u1/4||) = | | 1.88E+00 | |  |  |
| Погрешность ||u\*-ui||h/2 | | | 4.55E-01 |  |  |
| Погрешность ||u\*-ui||h/4 | | | 1.13E-01 |  |  |

При исследованиях был получен 2 порядок сходимости. При дроблении сетки в 2 раза погрешность падает приблизительно в 4 раза.

**Исследование на вложенных неравномерных сетках**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| t | x | y | u\* | uh | uh/2 | uh/4 | |u\*-uh| | |u\*-uh/2| | |u\*-uh/4| |
| 2.28223235504882 | 0 | 0 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 1 | 0 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 2 | 0 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 0 | 1 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 1 | 1 | 23.77440024 | 24.01599405893360 | 23.82360340125690 | 23.78568880929490 | 2.42E-01 | 4.92E-02 | 1.13E-02 |
|  | 2 | 1 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 0 | 2 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 1 | 2 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
|  | 2 | 2 | 23.77440024 | 23.77440024220010 | 23.77440024220010 | 23.77440024220020 | 8.88E-14 | 8.88E-14 | 1.07E-14 |
| 3.40574161156684 | 0 | 0 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 1 | 0 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 2 | 0 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 0 | 1 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 1 | 1 | 79.00691107 | 79.04655600634030 | 79.01529698990670 | 79.00884498490310 | 3.96E-02 | 8.39E-03 | 1.93E-03 |
|  | 2 | 1 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 0 | 2 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 1 | 2 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
|  | 2 | 2 | 79.00691107 | 79.00691106534190 | 79.00691106534190 | 79.00691106534190 | 2.98E-13 | 2.98E-13 | 2.98E-13 |
| 4.00000000000000 | 0 | 0 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 0 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 0 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 1 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 1 | 128 | 128.00371393497200 | 128.00078741869500 | 128.00018158487300 | 3.71E-03 | 7.87E-04 | 1.82E-04 |
|  | 2 | 1 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 0 | 2 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 1 | 2 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
|  | 2 | 2 | 128 | 128.00000000000000 | 128.00000000000000 | 128.00000000000000 | 0.00E+00 | 0.00E+00 | 0.00E+00 |
| Погрешность ||u\*-ui||h | |  | 2.45E-01 | log2(||u-u1/2|| / ||u1/2-u1/4||) = | | 2.34E+00 | |  |  |
| Погрешность ||u\*-ui||h/2 | | | 4.99E-02 |  |  |
| Погрешность ||u\*-ui||h/4 | | | 1.15E-02 |  |  |

При исследованиях на вложенных неравномерных сетках был получен второй порядок сходимости, что соответствует теоретическому.

**Выводы**: При исследованиях был подверждён второй порядок сходимости, как по пространству (за счёт биквадратичных базисных функций), так и по времени (за счёт трёхслойной схемы). При правильно подобранной неравномерной сетке погрешность может падать быстрее, чем на равномерной, как и получилось в исследованиях на нестационарной задаче. Это может сыграть значимую роль в затратах ресурсов, если, допустим, известно, где решение просчитывается без необходимости внедрения дополнительных узлов, а где нужно сгустить сетку для точности решения. В равномерной же сетке нам потребуется больше памяти под хранение всех тех узлов, которыми можно было бы пренебречь в неравномерном случае.

# Код программы

/\*grid.h\*/

#pragma once

#include <stdio.h>

#include <vector>

#include <fstream>

#include <functional>

#include <array>

using namespace std;

namespace grid

{

//Нумерация идёт следующим образом:

//слева,справа, снизу, сверху

//кэ - конечный элемент

//б.ф. - базисная функция

//Область

struct Area

{

//x-координата левой границы

int leftX;

//x-координата правой границы

int rightX;

//у-координата нижней границы

int lowY;

//у-координата верхней границы

int upY;

//1 - 1 рода

//2 - 2 рода

//3 - 3 рода

//-1 - нет краевых условий

array<int, 4> ku;

//Cпособ расчёта краевых условий

//-1 - нет краевых условий

array<int, 4> kuForm;

void Input(FILE \*fo);

};

//Координатные линии областей

struct AreasLines

{

//x,y - массивы значений всех вертикальных

//и горизонтальных границ подоблостей Wi

vector <double> x;

vector <double> y;

AreasLines();

};

//Конечный элемент

struct Element

{

//Узлы

array<int, 4> nodes;

//Степени свободы

array<int, 9> dof;

//Номер области

int numberOfArea;

//Соседние элементы

array<int, 4> neighbors;

Element& operator=(Element element)

{

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

nodes[i] = element.nodes[i];

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

dof[i] = element.dof[i];

numberOfArea = element.numberOfArea;

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

neighbors[i] = element.neighbors[i];

return \*this;

}

//Поиск глобального номера б.ф. dofsNumber в элементе

bool SearchDof(int dofsNumber);

};

//Краевые условия для конечного элемента

struct BoundaryCondition

{

//Номер элемента

int elem;

//Краевое условие(на ребре): 1 - есть, 0 - нет

array<int, 4> edges;

//Номер формулы для расчёта

array<int, 4> formNumber;

};

struct Point

{

double x;

double y;

Point() {};

~Point() {};

Point(double xx, double yy)

{

x = xx;

y = yy;

}

bool operator==(Point point)

{

if (point.x == x && point.y == y)

return true;

else

return false;

}

};

//Разбиение общей области

class Grid

{

//Генерация координаты с учётом разбиения на всех интервалах

void PartitionСoordinate(vector <double> &ci, vector <double> areasLines,

vector <double> coefficient, vector <int> nIntervals);

void CreateGridTime();

//Добавление узла

void PushNode(double x, double y);

//Построение сетки

void BuildGrid();

//Получение глобального номера

int GetGlobalNumber(int elementNumber, int localNumber);

//Нахождение номера области, в которой лежит заданная точка

int FindArea(double x, double y);

//Нахождение соседних конечных элементов

void FindNeighbors(int elementNumber);

//Вычисление конечных элементов

void ComputeElements();

//Формирование массивов краевых условий

void FormKU();

public:

Grid();

//Массив областей

vector <Area> areas;

//Координатные линии

AreasLines areasLines;

//Массив конечных элементов

vector <Element> elements;

//Массив узлов

vector <Point> nodes;

vector <double> time;

//Массив краевых условий

array<vector <BoundaryCondition>, 3> ku;

//Общее количество интервалов по x и по y

//после построения сетки

int nx, ny;

//Получение глобального номера для базисных функций

int GetGlobalFuncNumber(int elNum, int localFuncNum);

//Образование множества кэ

void DoPartition();

//Формирование списка элементов, содержащих глобальный номер б.ф.

//равный dofsNumber

void SearchElements(int dofsNumber, vector<int>& elList);

~Grid();

};

}

/\*grid.cpp\*/

#include "grid.h"

namespace grid

{

void Area::Input(FILE \*fo)

{

fscanf\_s(fo, "%d", &leftX);

fscanf\_s(fo, "%d", &rightX);

fscanf\_s(fo, "%d", &lowY);

fscanf\_s(fo, "%d", &upY);

fscanf\_s(fo, "%d", &ku[0]);

fscanf\_s(fo, "%d", &ku[1]);

fscanf\_s(fo, "%d", &ku[2]);

fscanf\_s(fo, "%d", &ku[3]);

fscanf\_s(fo, "%d", &kuForm[0]);

fscanf\_s(fo, "%d", &kuForm[1]);

fscanf\_s(fo, "%d", &kuForm[2]);

fscanf\_s(fo, "%d", &kuForm[3]);

}

AreasLines::AreasLines()

{

int n;

double tmp;

FILE \*fo;

fopen\_s(&fo,"AreasLines.txt", "r");

fscanf\_s(fo, "%d", &n);

x.reserve(n);

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

{

fscanf\_s(fo, "%lf", &tmp);

x.push\_back(tmp);

}

fscanf\_s(fo, "%d", &n);

y.reserve(n);

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

{

fscanf\_s(fo, "%lf", &tmp);

y.push\_back(tmp);

}

fclose(fo);

}

//Поиск глобального номера б.ф. dofsNumber в элементе

bool Element::SearchDof(int dofsNumber)

{

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

if (dofsNumber == dof[i]) return true;

return false;

}

Grid::Grid()

{

int n;

FILE \*fo;

fopen\_s(&fo,"Areas.txt", "r");

fscanf\_s(fo, "%d", &n);

areas.resize(n);

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

areas[i].Input(fo);

fclose(fo);

}

Grid::~Grid() {}

//Генерация координаты с учётом разбиения на всех интервалах

void Grid::PartitionСoordinate(vector <double> &x, vector <double> areasLines,

vector <double> coefficient, vector <int> nIntervals)

{

int count;

//длина интервала

double l;

//шаг

double h;

//число интервалов

int nLines = areasLines.size();

count = 0;

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

{

x.push\_back(areasLines[i]);

count++;

//длина интервала

l = abs(areasLines[i + 1] - areasLines[i]);

//рассчитываем первый шаг

//равномерная

if (abs(1.0 - coefficient[i]) < 1E-14)

h = l / nIntervals[i];

else //сгущаем вправо

if (coefficient[i] < 1)

{

h = l \* (1.0 - coefficient[i]);

h /= 1.0 - pow(coefficient[i], nIntervals[i]);

}

else //сгущаем влево

{

h = l \* (coefficient[i] - 1.0);

h /= coefficient[i] \* (pow(coefficient[i], nIntervals[i] - 1.0) - 1.0);

h += coefficient[i] - 1.0;

}

//получаем сетку внутри интервала

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

{

if (j != 1) h \*= coefficient[i];

x.push\_back(x[count - 1] + h);

count++;

}

}

x.push\_back(areasLines[nLines - 1]);

}

void Grid::CreateGridTime()

{

int count = 0;

//длина интервала

double l;

//шаг

double h;

FILE \*fo;

int tIntervals;

double t1, tn, tCoefficient;

fopen\_s(&fo, "time.txt", "r");

fscanf\_s(fo, "%lf", &t1);

fscanf\_s(fo, "%lf", &tn);

fscanf\_s(fo, "%d", &tIntervals);

fscanf\_s(fo, "%lf", &tCoefficient);

fclose(fo);

time.push\_back(t1);

count++;

//длина интервала

l = tn - t1;

//рассчитываем первый шаг

//равномерная

if (abs(1.0 - tCoefficient) < 1E-14)

h = l / tIntervals;

else //сгущаем вправо

if (tCoefficient< 1)

{

h = l \* (1.0 - tCoefficient);

h /= 1.0 - pow(tCoefficient, tIntervals);

}

else //сгущаем влево

{

h = l \* (tCoefficient - 1.0);

h /= tCoefficient \* (pow(tCoefficient, tIntervals - 1.0) - 1.0);

h += tCoefficient - 1.0;

}

//получаем сетку внутри интервала

for (int j = 1; j < tIntervals; j++)

{

if (j != 1) h \*= tCoefficient;

time.push\_back(time[count - 1] + h);

count++;

}

time.push\_back(tn);

}

//Добавление узла

void Grid::PushNode(double x, double y)

{

Point p;

p.y = y;

p.x = x;

//кладём узел в конец массива узлов

nodes.push\_back(p);

}

//Построение сетки

void Grid::BuildGrid()

{

//для iого интервала и каждой координаты

//коэффициент разрядки

vector <double> xCoefficient;

vector <double> yCoefficient;

//число подинтервалов

vector <int> xIntervals;

vector <int> yIntervals;

//количество координатных линий по x и y

int xLines = areasLines.x.size();

int yLines = areasLines.y.size();

//геометрические линии разбиения по х и y

vector <double> xi;

vector <double> yj;

//временная переменная для считывания

int tmp1;

//временная переменная для считывания

double tmp2;

//число узлов и число элементов

int nNodes, nElements;

xIntervals.reserve(xLines - 1); xCoefficient.reserve(xLines - 1);

yIntervals.reserve(yLines - 1); yCoefficient.reserve(yLines - 1);

FILE \*fo;

fopen\_s(&fo,"Intervals.txt", "r");

//ввод количества интервалов по х и у и коэффициентов разрядки

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

{

fscanf\_s(fo, "%d", &tmp1);

xIntervals.push\_back(tmp1);

fscanf\_s(fo, "%lf", &tmp2);

xCoefficient.push\_back(tmp2);

}

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

{

fscanf\_s(fo, "%d", &tmp1);

yIntervals.push\_back(tmp1);

fscanf\_s(fo, "%lf", &tmp2);

yCoefficient.push\_back(tmp2);

}

fclose(fo);

nx = 0;

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

//общее количество интервалов по х

nx += xIntervals[i];

nx++;

ny = 0;

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

//общее количество интервалов по у

ny += yIntervals[j];

ny++;

xi.reserve(nx); yj.reserve(ny);

//построение сеток по х и у

PartitionСoordinate(xi, areasLines.x, xCoefficient, xIntervals);

PartitionСoordinate(yj, areasLines.y, yCoefficient, yIntervals);

xIntervals.clear(); xCoefficient.clear();

yIntervals.clear(); yCoefficient.clear();

nElements = (nx - 1) \* (ny - 1);

nNodes = xi.size() \* yj.size();

elements.reserve(nElements);

nodes.reserve(nNodes);

//заполняем список узлов

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

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

PushNode(xi[i], yj[j]);

}

//Получение глобального номера

int Grid::GetGlobalNumber(int elementNumber, int localNumber)

{

//число интервалов по x

int nxint = nx - 1;

//номер горизонтальной линии, которая является нижним ребром элемента

int nGorLine = elementNumber / nxint;

//начальный номер узна на ребре

int nodeOnLine = nGorLine \* (nxint + 1);

//отступ от начального узла, чтобы получить текущий номер

//левого нижнего узла элемента;

int offsetX = elementNumber % nxint;

//номер нижнего левого узла

int nodeElem = nodeOnLine + offsetX;

if (localNumber == 0)

return nodeElem;

if (localNumber == 1)

return nodeElem + 1;

if (localNumber == 2)

return nodeElem + (nxint + 1);

if (localNumber == 3)

return nodeElem + (nxint + 1) + 1;

return -1;

}

//Получение глобальных базисных номеров

int Grid::GetGlobalFuncNumber(int elNum, int localFuncNum)

{

//число интервалов по x

int nxint = nx - 1;

//номер горизонтальной линии, которая является нижним ребром элемента

int nGorLine = elNum / nxint;

//Количество узлов на одной горизонтальной линии

int nGorNodes = 2 \* nx - 1;

//отступ от начального узла, чтобы получить текущий номер

//левого нижнего узла элемента;

int offsetX = elNum % nxint;

//глобальный номер нижнего левого узла

int res = 2 \* nGorLine \* nGorNodes + 2 \* offsetX;

//в нумерации от единицы 1-3

if (localFuncNum <= 2)

return res + localFuncNum;

else //... 4-6

if (localFuncNum <= 5)

return res + nGorNodes + (localFuncNum - 3);

else //... 7-9

return res + 2 \* nGorNodes + (localFuncNum - 6);

}

//Нахождение номера области, в которой лежит заданная точка

int Grid::FindArea(double x, double y)

{

bool xInterval, yInterval;

int size = areas.size();

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

{

xInterval = x > areasLines.x[areas[i].leftX] && x < areasLines.x[areas[i].rightX];

yInterval = y > areasLines.y[areas[i].lowY] && y < areasLines.y[areas[i].upY];

//если точка попадает в iую область, возвращаем номер этой области

if (xInterval && yInterval) return i;

}

}

//Нахождение соседних конечных элементов

void Grid::FindNeighbors(int elementNumber)

{

int count = 0;

//количество конечных элементов

int nElements = elements.size();

Element element;

element = elements[elementNumber];

bool tmp[4] = { false, false, false, false };

for (int i = 0; count < 4 && i < nElements; i++)

{

if (i != elementNumber)

{

//сосед по левому ребру

if (nodes[element.nodes[0]] == nodes[elements[i].nodes[1]] && nodes[element.nodes[2]] == nodes[elements[i].nodes[3]])

{

element.neighbors[0] = i;

tmp[0] = true;

count++;

}

//сосед по правому ребру

if (nodes[element.nodes[1]] == nodes[elements[i].nodes[0]] && nodes[element.nodes[3]] == nodes[elements[i].nodes[2]])

{

element.neighbors[1] = i;

tmp[1] = true;

count++;

}

//сосед по нижнему ребру

if (nodes[element.nodes[0]] == nodes[elements[i].nodes[2]] && nodes[element.nodes[1]] == nodes[elements[i].nodes[3]])

{

element.neighbors[2] = i;

tmp[2] = true;

count++;

}

//сосед по верхнему ребру

if (nodes[element.nodes[2]] == nodes[elements[i].nodes[0]] && nodes[element.nodes[3]] == nodes[elements[i].nodes[1]])

{

element.neighbors[3] = i;

tmp[3] = true;

count++;

}

}

}

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

//отсутствие соседа на соответствующей стороне

if (tmp[i] == false) element.neighbors[i] = -1;

elements[elementNumber] = element;

}

//Вычисление конечных элементов

void Grid::ComputeElements()

{

Element tmpEl;

double x, y;

//вычисляем глобальные номера узлов и базисных функций

//каждого кэ и заполняем список кэ

for (int i = 0; i < elements.capacity(); i++)

{

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

tmpEl.nodes[j] = GetGlobalNumber(i, j);

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

tmpEl.dof[j] = GetGlobalFuncNumber(i, j);

elements.push\_back(tmpEl);

}

//находим номер подобласти для каждого кэ

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

{

//берём центральный узел

x = (nodes[elements[i].nodes[0]].x + nodes[elements[i].nodes[1]].x) / 2;

y = (nodes[elements[i].nodes[0]].y + nodes[elements[i].nodes[2]].y) / 2;

elements[i].numberOfArea = FindArea(x, y);

}

//находим соседние кэ

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

FindNeighbors(i);

}

//Формирование массивов краевых условий

void Grid::FormKU()

{

int size = elements.size();

BoundaryCondition tmp;

//для каждого элемента находим,какие ку на его границах

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

{

bool left[3] = { false, false, false }, right[3] = { false, false, false },

low[3] = { false, false, false }, up[3] = { false, false, false };

if (elements[i].neighbors[0] == -1)

{

if (areas[elements[i].numberOfArea].ku[0] == 1) left[0] = true;

else

if (areas[elements[i].numberOfArea].ku[0] == 2) left[1] = true;

else

if (areas[elements[i].numberOfArea].ku[0] == 3) left[2] = true;

}

if (elements[i].neighbors[1] == -1)

{

if (areas[elements[i].numberOfArea].ku[1] == 1) right[0] = true;

else

if (areas[elements[i].numberOfArea].ku[1] == 2) right[1] = true;

else

if (areas[elements[i].numberOfArea].ku[1] == 3) right[2] = true;

}

if (elements[i].neighbors[2] == -1)

{

if (areas[elements[i].numberOfArea].ku[2] == 1) low[0] = true;

else

if (areas[elements[i].numberOfArea].ku[2] == 2) low[1] = true;

else

if (areas[elements[i].numberOfArea].ku[2] == 3) low[2] = true;

}

if (elements[i].neighbors[3] == -1)

{

if (areas[elements[i].numberOfArea].ku[3] == 1) up[0] = true;

else

if (areas[elements[i].numberOfArea].ku[3] == 2) up[1] = true;

else

if (areas[elements[i].numberOfArea].ku[3] == 3) up[2] = true;

}

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

if (left[j] || right[j] || low[j] || up[j])

{

tmp.elem = i;

if (left[j])

{

tmp.edges[0] = 1;

tmp.formNumber[0] = areas[elements[i].numberOfArea].kuForm[0];

}

else

{

tmp.edges[0] = 0;

tmp.formNumber[0] = -1;

}

if (right[j])

{

tmp.edges[1] = 1;

tmp.formNumber[1] = areas[elements[i].numberOfArea].kuForm[1];

}

else

{

tmp.edges[1] = 0;

tmp.formNumber[1] = -1;

}

if (low[j])

{

tmp.edges[2] = 1;

tmp.formNumber[2] = areas[elements[i].numberOfArea].kuForm[2];

}

else

{

tmp.edges[2] = 0;

tmp.formNumber[2] = -1;

}

if (up[j])

{

tmp.edges[3] = 1;

tmp.formNumber[3] = areas[elements[i].numberOfArea].kuForm[3];

}

else

{

tmp.edges[3] = 0;

tmp.formNumber[3] = -1;

}

ku[j].push\_back(tmp);

}

}

}

//Образование множества кэ

void Grid::DoPartition()

{

//Построение сетки

BuildGrid();

CreateGridTime();

//Вычисление конечных элементов

ComputeElements();

//Формирование массивов краевых условий

FormKU();

}

//Формирование списка элементов, содержащих глобальный номер б.ф.

//равный dofsNumber

void Grid::SearchElements(int dofsNumber, vector <int> &elList)

{

int count;

int size = elements.size();

elList.reserve(4);

count = 0;

for (int i = 0; i < size && count < 4; i++)

{

if (elements[i].SearchDof(dofsNumber))

{

count++;

elList.push\_back(i);

}

}

}

}

/\*basis.h\*/

#pragma once

#include "grid.h"

using namespace grid;

namespace basis

{

const int nFunc = 9;

const int nFunc1D = 3;

struct Basis

{

//Указатели на функции вычисления базисных функций в точке

array<function<double(double, double)>, nFunc> phi;

//Указатели на функции вычисления d/dksi базисных функций в точке

array<function<double(double, double)>, nFunc> dphiksi;

//Указатели на функции вычисления d/detta базисных функций в точке

array<function<double(double, double)>, nFunc> dphietta;

Basis();

};

}

/\*basis.cpp\*/

#include "basis.h"

namespace basis

{

Basis::Basis()

{

array <function<double(double)>, nFunc1D> phi\_;

phi\_[0] = [](double ksi) { return 2 \* (ksi - 0.5) \* (ksi - 1); };

phi\_[1] = [](double ksi) { return -4 \* ksi \* (ksi - 1); };

phi\_[2] = [](double ksi) { return 2 \* ksi \* (ksi - 0.5); };

array <function<double(double)>, nFunc1D> dphi\_ksi;

dphi\_ksi[0] = [](double ksi) { return 4 \* ksi - 3; };

dphi\_ksi[1] = [](double ksi) { return -8 \* ksi + 4; };

dphi\_ksi[2] = [](double ksi) { return 4 \* ksi - 1; };

phi[0] = [phi\_](double ksi, double etta) { return phi\_[0](ksi) \* phi\_[0](etta); };

phi[1] = [phi\_](double ksi, double etta) { return phi\_[1](ksi) \* phi\_[0](etta); };

phi[2] = [phi\_](double ksi, double etta) { return phi\_[2](ksi) \* phi\_[0](etta); };

phi[3] = [phi\_](double ksi, double etta) { return phi\_[0](ksi) \* phi\_[1](etta); };

phi[4] = [phi\_](double ksi, double etta) { return phi\_[1](ksi) \* phi\_[1](etta); };

phi[5] = [phi\_](double ksi, double etta) { return phi\_[2](ksi) \* phi\_[1](etta); };

phi[6] = [phi\_](double ksi, double etta) { return phi\_[0](ksi) \* phi\_[2](etta); };

phi[7] = [phi\_](double ksi, double etta) { return phi\_[1](ksi) \* phi\_[2](etta); };

phi[8] = [phi\_](double ksi, double etta) { return phi\_[2](ksi) \* phi\_[2](etta); };

dphiksi[0] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[0](ksi) \* phi\_[0](etta); };

dphiksi[1] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[1](ksi) \* phi\_[0](etta); };

dphiksi[2] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[2](ksi) \* phi\_[0](etta); };

dphiksi[3] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[0](ksi) \* phi\_[1](etta); };

dphiksi[4] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[1](ksi) \* phi\_[1](etta); };

dphiksi[5] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[2](ksi) \* phi\_[1](etta); };

dphiksi[6] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[0](ksi) \* phi\_[2](etta); };

dphiksi[7] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[1](ksi) \* phi\_[2](etta); };

dphiksi[8] = [phi\_, dphi\_ksi](double ksi, double etta) { return dphi\_ksi[2](ksi) \* phi\_[2](etta); };

dphietta[0] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[0](ksi) \* dphi\_ksi[0](etta); };

dphietta[1] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[1](ksi) \* dphi\_ksi[0](etta); };

dphietta[2] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[2](ksi) \* dphi\_ksi[0](etta); };

dphietta[3] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[0](ksi) \* dphi\_ksi[1](etta); };

dphietta[4] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[1](ksi) \* dphi\_ksi[1](etta); };

dphietta[5] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[2](ksi) \* dphi\_ksi[1](etta); };

dphietta[6] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[0](ksi) \* dphi\_ksi[2](etta); };

dphietta[7] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[1](ksi) \* dphi\_ksi[2](etta); };

dphietta[8] = [phi\_, dphi\_ksi](double ksi, double etta) { return phi\_[2](ksi) \* dphi\_ksi[2](etta); };

}

}

/\*integration.h\*/

#pragma once

#include "matrix.h"

namespace integration

{

const int n\_ip = 25;

const int n\_ip1D = 5;

struct GaussIntegration

{

//Точки гаусса

array<array<double, n\_ip>, 2> gaussPoints;

//Веса гаусса

array<double, n\_ip> gaussWeights;

//Точки гаусса

array<double, n\_ip1D> gaussPoints1;

//Веса гаусса

array<double, n\_ip1D> gaussWeights1;

GaussIntegration();

};

}

/\*integration.cpp\*/

#include "integration.h"

namespace integration

{

GaussIntegration::GaussIntegration()

{

gaussPoints1[0] = 0;

gaussPoints1[1] = (1.0 / 3.0) \* sqrt(5.0 - 2.0 \* sqrt(10.0 / 7.0));

gaussPoints1[2] = -gaussPoints1[1];

gaussPoints1[3] = (1.0 / 3.0) \* sqrt(5.0 + 2.0 \* sqrt(10.0 / 7.0));

gaussPoints1[4] = -gaussPoints1[3];

gaussWeights1[0] = 128.0 / 225.0;

gaussWeights1[1] = (322.0 + 13.0 \* sqrt(70.0)) / 900.0;

gaussWeights1[2] = gaussWeights1[1];

gaussWeights1[3] = (322.0 - 13.0 \* sqrt(70.0)) / 900.0;

gaussWeights1[4] = gaussWeights1[3];

for (size\_t i = 0; i < n\_ip1D; i++)

{

for (size\_t j = 0; j < n\_ip1D; j++)

{

gaussPoints[0][i \* n\_ip1D + j] = gaussPoints1[j];

gaussPoints[1][i \* n\_ip1D + j] = gaussPoints1[i];

gaussWeights[i \* n\_ip1D + j] = gaussWeights1[i] \* gaussWeights1[j];

}

}

}

}

/\*matrix.h\*/

#pragma once

#include <algorithm>

#include "basis.h"

namespace matrix

{

struct Matrix

{

//Матрица СЛАУ

//Указатели начала строк(столбцов)

vector <int> ig;

//Номера столбцов внедиагональных элементов

vector <int> jg;

//Диагональные элементы матрицы

vector <double> di;

//Внедиагональные элементы нижнего треугольника матрицы

vector <double> ggl;

//Внедиагональные элементы верхнего треугольника матрицы

vector <double> ggu;

//Размерность матрицы

int n;

//Размер векторов ggl, ggu, jg

int size;

//Генерация портрета матрицы

void CreatePortret(int slaeSize, Grid grid);

//Инициализация матрицы после генерации портрета

void Initialize(int size1, int size2);

//Умножение матрицы на вектор

void MultiplyAx(const vector <double> a, vector<double>& result);

//Умножение транспонированной матрицы на вектор

void MultiplyATx(vector<double> a, vector<double>& result);

};

}

/\*matrix.cpp\*/

#include "matrix.h"

namespace matrix

{

//Генерация портрета матрицы

void Matrix::CreatePortret(int slaeSize, Grid grid)

{

vector <int> elList, unzeroNumbersList;

vector<vector <int>> list;

list.reserve(slaeSize);

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

{

//нашли,в каких элементах присутствует данная i-я степень свободы

grid.SearchElements(i, elList);

//собрать список узлов этих элементов, меньших по номеру

for (unsigned int j = 0; j < elList.size(); j++)

{

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

{

//если такого номера ещё нет и он меньше по номеру,то добавляем

if (find(unzeroNumbersList.begin(), unzeroNumbersList.end(), grid.elements[elList[j]].dof[k])

== unzeroNumbersList.end() && grid.elements[elList[j]].dof[k] < i)

unzeroNumbersList.push\_back(grid.elements[elList[j]].dof[k]);

}

}

sort(unzeroNumbersList.begin(), unzeroNumbersList.end());

list.push\_back(unzeroNumbersList);

unzeroNumbersList.clear();

elList.clear();

}

//вычисляем размерность ggl,ggu

int gg\_size = 0;

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

{

if (!list[i].empty())

gg\_size += list[i].size();

}

//инициализируем матрицу и собираем портрет

Initialize(slaeSize, gg\_size);

ig[0] = 0;

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

{

if (!list[i].empty())

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

else

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

}

int k = 0;

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

{

if (!list[i].empty())

{

for (unsigned int j = 0; j < list[i].size(); j++)

{

jg[k] = list[i][j];

k++;

}

list[i].clear();

}

}

list.clear();

}

//Инициализация матрицы после генерации портрета

void Matrix::Initialize(int size1, int size2)

{

n = size1; size = size2;

ggl.resize(size);

ggu.resize(size);

di.resize(n);

ig.resize(n + 1);

jg.resize(size);

}

//Умножение матрицы на вектор

void Matrix::MultiplyAx(const vector <double> a, vector <double> &result)

{

int i, j, l, ik, iend, k;

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

{

//начало i-ой строки(столбца)

l = ig[i];

//начало (i+1)-ой строки(столбца)

iend = ig[i + 1];

//количество элементов в i строке(столбце)

ik = iend - l;

result[i] = di[i] \* a[i];

//проходим по всем элементам i строки (столбца)

for (k = 0; k < ik; k++, l++)

{

j = jg[l];

result[i] += ggl[l] \* a[j];

result[j] += ggu[l] \* a[i];

}

}

}

//Умножение транспонированной матрицы на вектор

void Matrix::MultiplyATx(vector <double> a, vector <double> &result)

{

int i, j, l, ik, iend, k;

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

{

//начало i-ой строки(столбца)

l = ig[i];

//начало (i+1)-ой строки(столбца)

iend = ig[i + 1];

//количество элементов в i строке(столбце)

ik = iend - l;

result[i] = di[i] \* a[i];

//проходим по всем элементам i строки (столбца)

for (k = 0; k < ik; k++, l++)

{

j = jg[l];

result[i] += ggu[l] \* a[j];

result[j] += ggl[l] \* a[i];

}

}

}

}

/\*slae.h\*/

#pragma once

#include "tests.h"

using namespace matrix;

using namespace basis;

using namespace integration;

using namespace tests;

namespace slae

{

class SLAE : private Basis, private GaussIntegration

{

//Размерность задачи

int n;

//Максимальное количество итераций в решателе

int maxiter = 10000;

//Точность решения СЛАУ

const double eps = 1e-10;

//Сетка

Grid grid;

//Хранилище тестовых функций

Tests tests;

//Глобальная матрица

Matrix A;

Matrix globalM1;

Matrix globalM2;

//Локальные матрицы

//Матрица жёсткости

array<array<double, 9>, 9> G;

//Матрица массы

array<array<double, 9>, 9> M;

array<array<double, 9>, 9> Mg;

//Локальный вектор правой части

array <double, 9> locF;

//Глобальный вектор правой части

vector <double> F;

array <double, 9> newF;

//Вектор приближенного решения на предыдущей итерации

//по нелинейности

vector <double> u\_n;

//Вектора приближенного решения на предыдущих итерациях

//по времени

vector <double> u\_t1;

vector <double> u\_t2;

//Вектор приближенного решения

vector <double> u;

//Норма вектора правой части

double normF;

//Сборка локальных матриц жёсткости

void CalculateG(int elementNumber);

//Сборка локальных матриц масс

void CalculateM(int elementNumber);

void CalculateMg(int elementNumber);

//Сборка локальных правых частей

void CalculateLocalF(int elementNumber);

//Добавка локального элемента в глобальный

void AddElementToGlobalMatrix(Matrix &B, int i, int j, double element);

//Сборка локальных матриц(векторов) и добавление в глобальные

void CalculateLocals(int elementNumber);

//Вектор праввой части для первого краевого

array<double, 3> g;

//Нахождение правой части для 1ого краевого условия

void Calculate\_g(int formNumber, int orientation, int elNumber);

//Вычисление 1ого краевого условия для одного узла

void CalculateBoundaries1ForNode(int node, double gi, double weight);

//Учёт первого краевого условия

void CalculateBoundaries1(int number);

//Компоненты матрицы с факторизацией

vector <double> L;

vector <double> D;

vector <double> U;

//Текущее значение времени

double t;

double deltat;

double deltat0;

double deltat1;

//Вектор невязки

vector <double> r;

//Вектор спуска

vector <double> z;

//Вычисление нормы вектора

double Norm(const vector<double>& x);

//Скалярное произведение векторов

double Scalar(const vector<double>& x, const vector<double>& y);

//Генерация СЛАУ на i-ой итерации по времени

void GenerateSLAE();

//LU-факторизация

void LU();

//Вспомогательные функции для решателя

void LYF(const vector<double>& C, vector<double>& yl);

void UXY(const vector<double>& C, vector<double>& yu);

double Rel\_Discrepancy();

//Решатель ЛОС с LU-факторизацией

void LULOS();

double StopIteration();

void CalculateTimeVector(double time, vector<double> &result);

public:

SLAE();

void TSolve();

~SLAE() {};

};

}

/\*slae.cpp\*/

#include "slae.h"

namespace slae

{

SLAE::SLAE()

{

//Построение сетки

grid.DoPartition();

//Размерность задачи соответствует общему числу базисных функций

n = (2 \* grid.nx - 1)\*(2 \* grid.ny - 1);

F.resize(n);

u.resize(n);

u\_n.resize(n);

u\_t1.resize(n);

u\_t2.resize(n);

r.resize(n);

z.resize(n);

//Генерация портрета матрицы и её инициализация

A.CreatePortret(n, grid);

globalM1.CreatePortret(n, grid);

globalM2.CreatePortret(n, grid);

}

//Сборка локальных матриц жёсткости

void SLAE::CalculateG(int elementNumber)

{

Element element = grid.elements[elementNumber];

double g1, g2, ksi, etta, x\_, y\_, lambda,

x1 = grid.nodes[element.nodes[0]].x, x3 = grid.nodes[element.nodes[1]].x,

y1 = grid.nodes[element.nodes[0]].y, y3 = grid.nodes[element.nodes[2]].y,

hx = x3 - x1, hy = y3 - y1,

hx2 = hx \* hx, hy2 = hy \* hy,

jacobian = hx \* hy / 4.0;

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

{

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

{

g1 = 0; g2 = 0;

for (int k = 0; k < 25; k++)

{

ksi = 0.5 + 0.5 \* gaussPoints[0][k]; etta = 0.5 + 0.5 \* gaussPoints[1][k];

x\_ = x1 + ksi\*hx; y\_ = y1 + etta\*hy;

lambda = tests.Lambda(x\_, y\_, t);

g1 += gaussWeights[k] \* dphiksi[i](ksi, etta) \* dphiksi[j](ksi, etta) \* lambda;

g2 += gaussWeights[k] \* dphietta[i](ksi, etta) \* dphietta[j](ksi, etta) \* lambda;

}

G[i][j] = g1 \* jacobian / hx2 + g2 \* jacobian / hy2;

}

}

//матрица симметричная, заполняем нижний треугольник

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

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

G[i][j] = G[j][i];

}

//Сборка локальных матриц масс

void SLAE::CalculateM(int elementNumber)

{

Element element = grid.elements[elementNumber];

double g, ksi, etta, sigma, x\_, y\_,

x1 = grid.nodes[element.nodes[0]].x, x3 = grid.nodes[element.nodes[1]].x,

y1 = grid.nodes[element.nodes[0]].y, y3 = grid.nodes[element.nodes[2]].y,

hx = x3 - x1, hy = y3 - y1,

jacobian = hx \* hy / 4.0;

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

{

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

{

g = 0;

for (int k = 0; k < 25; k++)

{

ksi = 0.5 + 0.5 \* gaussPoints[0][k]; etta = 0.5 + 0.5 \* gaussPoints[1][k];

x\_ = x1 + ksi\*hx; y\_ = y1 + etta\*hy;

sigma = tests.Sigma(x\_, y\_, t);

g += sigma \* gaussWeights[k] \* phi[i](ksi, etta) \* phi[j](ksi, etta);

}

M[i][j] = g \* jacobian;

}

}

//матрица симметричная, заполняем нижний треугольник

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

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

M[i][j] = M[j][i];

}

void SLAE::CalculateMg(int elementNumber)

{

Element element = grid.elements[elementNumber];

double g, ksi, etta, gamma, x\_, y\_,

x1 = grid.nodes[element.nodes[0]].x, x3 = grid.nodes[element.nodes[1]].x,

y1 = grid.nodes[element.nodes[0]].y, y3 = grid.nodes[element.nodes[2]].y,

hx = x3 - x1, hy = y3 - y1,

jacobian = hx \* hy / 4.0;

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

{

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

{

g = 0;

for (int k = 0; k < 25; k++)

{

ksi = 0.5 + 0.5 \* gaussPoints[0][k]; etta = 0.5 + 0.5 \* gaussPoints[1][k];

x\_ = x1 + ksi\*hx; y\_ = y1 + etta\*hy;

gamma= tests.Gamma(x\_, y\_, t);

g += gamma \* gaussWeights[k] \* phi[i](ksi, etta) \* phi[j](ksi, etta);

}

Mg[i][j] = g \* jacobian;

}

}

//матрица симметричная, заполняем нижний треугольник

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

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

Mg[i][j] = Mg[j][i];

}

//Сборка локальных правых частей

void SLAE::CalculateLocalF(int elementNumber)

{

Element element = grid.elements[elementNumber];

double ksi, etta, x\_, y\_,

x1 = grid.nodes[element.nodes[0]].x, x3 = grid.nodes[element.nodes[1]].x,

y1 = grid.nodes[element.nodes[0]].y, y3 = grid.nodes[element.nodes[2]].y,

hx = x3 - x1, hy = y3 - y1,

jacobian = hx \* hy / 4.0;

//интегрирование(Гаусс 5)

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

{

locF[i] = 0;

for (int k = 0; k < 25; k++)

{

ksi = 0.5 + 0.5 \* gaussPoints[0][k]; etta = 0.5 + 0.5 \* gaussPoints[1][k];

x\_ = x1 + ksi\*hx; y\_ = y1 + etta\*hy;

locF[i] += tests.Fi(x\_, y\_, t) \* gaussWeights[k] \* phi[i](ksi, etta);

}

locF[i] \*= jacobian;

}

}

//Добавка локального элемента в глобальный

void SLAE::AddElementToGlobalMatrix(Matrix &B, int i, int j, double element)

{

int id;

bool flag;

if (i == j)

B.di[i] += element;

else

{

if (i < j)

{

flag = false;

for (id = B.ig[j]; !flag && id < B.ig[j + 1]; id++)

if (B.jg[id] == i) flag = true;

if (flag) B.ggu[id - 1] += element;

}

else

{

flag = false;

for (id = B.ig[i]; !flag && id < B.ig[i + 1]; id++)

if (B.jg[id] == j) flag = true;

if (flag) B.ggl[id - 1] += element;

}

}

}

//Сборка локальных матриц(векторов) и добавление в глобальные

void SLAE::CalculateLocals(int elementNumber)

{

Element element = grid.elements[elementNumber];

int ki, kj;

double tmpM;

//вычисление локальных матриц

CalculateG(elementNumber);

CalculateM(elementNumber);

CalculateMg(elementNumber);

CalculateLocalF(elementNumber);

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

{

ki = element.dof[i];

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

{

kj = element.dof[j];

tmpM = (deltat + deltat0) / deltat0 / deltat \* M[i][j];

//добавка в глобальную матрицу А

AddElementToGlobalMatrix(A, ki, kj, G[i][j] + tmpM + Mg[i][j]);

//добавка в глобальные матрицы масс

tmpM = deltat0 / deltat1 / deltat \* M[i][j];

AddElementToGlobalMatrix(globalM2, ki, kj, tmpM);

tmpM = deltat / deltat1 / deltat0 \* M[i][j];

AddElementToGlobalMatrix(globalM1, ki, kj, tmpM);

}

//добавка в глобальную правую часть

F[ki] += locF[i];

}

}

//Генерация СЛАУ

void SLAE::GenerateSLAE()

{

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

A.di[i] = F[i] = globalM1.di[i] = globalM2.di[i] = 0;

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

A.ggl[i] = A.ggu[i] = globalM1.ggl[i] = globalM1.ggu[i] =

globalM2.ggl[i] = globalM2.ggu[i]= 0;

//Высчитывание локальных матриц(векторов) и добавление в глобальные

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

CalculateLocals(i);

vector<double> b;

b.resize(n);

//добавка от матриц масс по времени

globalM2.MultiplyAx(u\_t2, b);

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

F[i] -= b[i];

globalM1.MultiplyAx(u\_t1, b);

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

F[i] += b[i];

//Учёт краевых условий

for (int i = 0; i < grid.ku[0].size(); i++)

CalculateBoundaries1(i);

normF = Norm(F);

}

//Нахождение правой части для 1ого краевого условия

void SLAE::Calculate\_g(int formNumber, int orientation, int elNumber)

{

Element element = grid.elements[elNumber];

switch (orientation)

{

//левое ребро

case 0:

{

double x = grid.nodes[element.nodes[0]].x;

double y1 = grid.nodes[element.nodes[0]].y;

double y3 = grid.nodes[element.nodes[2]].y;

double y2 = (y1 + y3) / 2;

g[0] = tests.Ug(x, y1,t);

g[1] = tests.Ug(x, y2, t);

g[2] = tests.Ug(x, y3, t);

}

break;

//правое ребро

case 1:

{

double x = grid.nodes[element.nodes[1]].x;

double y1 = grid.nodes[element.nodes[1]].y;

double y3 = grid.nodes[element.nodes[3]].y;

double y2 = (y1 + y3) / 2;

g[0] = tests.Ug(x, y1, t);

g[1] = tests.Ug(x, y2, t);

g[2] = tests.Ug(x, y3, t);

}

break;

//нижнее ребро

case 2:

{

double y = grid.nodes[element.nodes[0]].y;

double x1 = grid.nodes[element.nodes[0]].x;

double x3 = grid.nodes[element.nodes[1]].x;

double x2 = (x1+x3)/2;

g[0] = tests.Ug(x1, y, t);

g[1] = tests.Ug(x2, y, t);

g[2] = tests.Ug(x3, y, t);

}

break;

//верхнее ребро

case 3:

{

double y = grid.nodes[element.nodes[2]].y;

double x1 = grid.nodes[element.nodes[2]].x;

double x3 = grid.nodes[element.nodes[3]].x;

double x2 = (x1 + x3) / 2;

g[0] = tests.Ug(x1, y, t);

g[1] = tests.Ug(x2, y, t);

g[2] = tests.Ug(x3, y, t);

}

break;

default:; break;

}

}

//Вычисление 1ого краевого условия для одного узла

void SLAE::CalculateBoundaries1ForNode(int node, double gi, double weight)

{

int id;

F[node] = gi;

A.di[node] = weight;

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

if (node < j)

{

bool flag = false;

for (id = A.ig[j]; !flag && id <= A.ig[j + 1] - 1; id++)

if (A.jg[id] == node) flag = true;

if (flag) A.ggu[id - 1] = 0.0;

}

else

{

bool flag = false;

for (id = A.ig[node]; !flag && id <= A.ig[node + 1] - 1; id++)

if (A.jg[id] == j) flag = true;

if (flag) A.ggl[id - 1] = 0.0;

}

}

//Учёт первого краевого условия

void SLAE::CalculateBoundaries1(int number)

{

Element element = grid.elements[grid.ku[0][number].elem];

if (grid.ku[0][number].edges[0] == 1)

{

int indexes[3] = { element.dof[0], element.dof[3], element.dof[6] };

Calculate\_g(grid.ku[0][number].formNumber[0], 0, grid.ku[0][number].elem);

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

CalculateBoundaries1ForNode(indexes[i], g[i], 1);

}

if (grid.ku[0][number].edges[1] == 1)

{

int indexes[3] = { element.dof[2], element.dof[5], element.dof[8] };

Calculate\_g(grid.ku[0][number].formNumber[1], 1, grid.ku[0][number].elem);

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

CalculateBoundaries1ForNode(indexes[i], g[i], 1);

}

if (grid.ku[0][number].edges[2] == 1)

{

int indexes[3] = { element.dof[0], element.dof[1], element.dof[2] };

Calculate\_g(grid.ku[0][number].formNumber[2], 2, grid.ku[0][number].elem);

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

CalculateBoundaries1ForNode(indexes[i], g[i], 1);

}

if (grid.ku[0][number].edges[3] == 1)

{

int indexes[3] = { element.dof[6], element.dof[7], element.dof[8] };

Calculate\_g(grid.ku[0][number].formNumber[3], 3, grid.ku[0][number].elem);

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

CalculateBoundaries1ForNode(indexes[i], g[i], 1);

}

}

#pragma region LU\_LOS

//Вычисление нормы вектора

double SLAE::Norm(const vector<double> &x)

{

double norm = 0;

int size = x.size();

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

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

return sqrt(norm);

}

//Скалярное произведение векторов

double SLAE::Scalar(const vector<double> &x, const vector<double> &y)

{

double sum = 0;

int size = x.size();

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

sum += x[i] \* y[i];

return sum;

}

//LU-факторизация

void SLAE::LU()

{

int i, i0, j0, iend, num, ki, kj, jend;

double suml, sumu, sumdg;

L.resize(A.ggl.size());

L = A.ggl;

U.resize(A.ggu.size());

U = A.ggu;

D.resize(A.di.size());

D = A.di;

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

{

i0 = A.ig[i];

iend = A.ig[i + 1];

for (num = i0, sumdg = 0; num < iend; num++)

{

j0 = A.ig[A.jg[num]];

jend = A.ig[A.jg[num] + 1];

ki = i0;

kj = j0;

//для num учитываются все предыдущие элементы

for (suml = 0, sumu = 0, ki = i0; ki < num; ki++)

{

for (int m = kj; m < jend; m++)

//ищем соответствующие ненулевые элементы для умножения

if (A.jg[ki] == A.jg[m])

{

suml += L[ki] \* U[m];

sumu += L[m] \* U[ki];

}

}

L[num] -= suml;

U[num] = (U[num] - sumu) / D[A.jg[num]];

//умножаются симметричные элементы

sumdg += L[num] \* U[num];

}

D[i] -= sumdg;

}

}

void SLAE::LYF(const vector <double> &C, vector <double> &yl)

{

int i, i0, iend; //i0-адрес начала строки, iend-адрес конца строки

double sum;

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

{

i0 = A.ig[i]; iend = A.ig[i + 1];

yl[i] = C[i];

for (i0, sum = 0; i0 < iend; i0++)

yl[i] -= yl[A.jg[i0]] \* L[i0];

yl[i] /= D[i];

}

}

void SLAE::UXY(const vector <double> &C, vector <double> &yu)

{

int i, i0, iend;

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

yu[i] = 0.0;

for (i = n - 1; i >= 0; i--)//проход по столбцам с конца

{

yu[i] += C[i];

i0 = A.ig[i]; iend = A.ig[i + 1]; iend--;

for (; iend >= i0; iend--)//идём по столбцу с конца

yu[A.jg[iend]] -= yu[i] \* U[iend];

}

}

double SLAE::Rel\_Discrepancy()

{

double dis1, dis2;

dis1 = Scalar(r, r);

dis2 = Scalar(F, F);

double dis = dis1 / dis2;

return sqrt(dis);

}

void SLAE::LULOS()

{

double a, b, pp, dis, rr;

int i,k;

vector <double> Ax(n), C(n), p(n);

LU();

//Ax0

A.MultiplyAx(u, Ax);

//f-Ax0

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

r[i] = F[i] - Ax[i];

//r0=L^(-1)(f-Ax0)

LYF(r, r);

//z0=U^(-1)r0->r0=Uz0

UXY(r, z);

//p0=L^(-1)Az0

A.MultiplyAx(z, Ax);//Az0

LYF(Ax, p);

rr = Scalar(r, r);

dis = Scalar(r, r) / rr;

dis = sqrt(dis);

k = 0;

for (k = 1; dis > eps && k <= maxiter; k++)

{

//Аk

pp = Scalar(p, p);

a = Scalar(p, r) / pp;

//Xk, Rk

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

{

u[i] = u[i] + a\*z[i];

r[i] = r[i] - a\*p[i];

}

//UY=rk->Y=U^(-1)rk

UXY(r, C);

//AU^(-1)rk=Ax

A.MultiplyAx(C, Ax);

//L^(-1)AU^(-1)rk=Y2->L^(-1)B=Y2->LY2=B->Y2=L^(-1)AU^(-1)rk

LYF(Ax, Ax);

//bk

b = -Scalar(p, Ax) / pp;

//zk=U^(-1)rk+bkz[k-1]

//pk

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

{

z[i] = C[i] + b \* z[i];

p[i] = Ax[i] + b \* p[i];

}

dis = Scalar(r, r) / rr;

dis = sqrt(dis);

}

dis = Rel\_Discrepancy();

}

#pragma endregion

void SLAE::CalculateTimeVector(double time, vector<double> &result)

{

//определение вектора решения на 0 временном слое

double x\_0, x\_1, x\_2, y\_0, y\_1, y\_2;

for (auto i : grid.elements)

{

x\_0 = grid.nodes[i.nodes[0]].x;

x\_2 = grid.nodes[i.nodes[1]].x;

x\_1 = (x\_2 - x\_0) / 2;

y\_0 = grid.nodes[i.nodes[0]].y;

y\_2 = grid.nodes[i.nodes[2]].y;

y\_1 = (y\_2 - y\_0) / 2;

result[i.dof[0]] = tests.Ug(x\_0, y\_0, time);

result[i.dof[1]] = tests.Ug(x\_1, y\_0, time);

result[i.dof[2]] = tests.Ug(x\_2, y\_0, time);

result[i.dof[3]] = tests.Ug(x\_0, y\_1, time);

result[i.dof[4]] = tests.Ug(x\_1, y\_1, time);

result[i.dof[5]] = tests.Ug(x\_2, y\_1, time);

result[i.dof[6]] = tests.Ug(x\_0, y\_2, time);

result[i.dof[7]] = tests.Ug(x\_1, y\_2, time);

result[i.dof[8]] = tests.Ug(x\_2, y\_2, time);

}

}

void SLAE::TSolve()

{

FILE \*fo;

fopen\_s(&fo, "result.txt", "w");

CalculateTimeVector(grid.time[0], u\_t2);

CalculateTimeVector(grid.time[1], u\_t1);

CalculateTimeVector(grid.time[2], u);

for (auto i:grid.time)

fprintf(fo, "%.14lf\t", i);

fprintf(fo, "\n\n");

//цикл по времени

for (int i = 2; i < grid.time.size(); i++)

{

t = grid.time[i];

deltat = t - grid.time[i - 2];

deltat1 = grid.time[i - 1] - grid.time[i - 2];

deltat0 = t - grid.time[i - 1];

printf("Calculating......([t] = %f)\n", t);

//находим новое решение

GenerateSLAE();

LULOS();

//сохраняем решение на предыдущих временных слоях

u\_t2 = u\_t1;

u\_t1 = u;

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

fprintf(fo, "%.14lf\n", u[i]);

fprintf(fo, "%.14lf\t", t);

fprintf(fo, "\n\n");

printf("Vector:");

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

printf("\t\t%.14lf\n", u[i]);

printf("\n\n");

}

fclose(fo);

}

}

/\*main.cpp\*/

#include "slae.h"

void main()

{

slae::SLAE A;

A.TSolve();

}