Министерство образования и науки РФ

Новосибирский государственный технический университет

Кафедра прикладной математики

Курсовая работа

по дисциплине

«Методы конечноэлементного анализа»

Факультет: ПМИ

Группа: ПММ-42

Студенты: Ряховский М.И., Сулейманова К.А.

Преподаватель: Персова М.Г.

Новосибирск

2015

# Цель работы

Разработать программу для решения стационарного уравнения диффузии Разрывным методом Галёркина (DG-методом) на тетраэдрах с использованием p-технологии.

# Математическая модель

Стационарное уравнение диффузии:

 в 

Краевые условия:



# Вариационная постановка

Для решения будем использовать вариационную постановку Баумана-Одена:

 ,

где:

*  – скачок функции на границе;
*  – среднее значение функции на границе;
*  – граница между элементами;
*  – часть внешней границы ;
* ;
*  – конечные элементы.

# Локальный базис на элементах

Использование p-технологии подразумевает наличие базисов разных порядков в одной конечноэлементной сетке. Для удобства работы построим базис по иерархическому принцу. Будем использовать базисы второго и третьего порядка.

Поскольку в качестве геометрического носителя элемента используются тетраэдры, базис будем определять через L-координаты тетраэдра. Базис третьего порядка будет состоять из следующих функций:

|  |  |  |  |
| --- | --- | --- | --- |
| Тип функции | Выражение для вычисления | Количество | Порядок базиса |
| 1 |  | 4 | 2, 3 |
| 2 |  | 6 | 2, 3 |
| 3 |  | 6 | 3 |
| 4 |  | 4 | 3 |

Базисные функции второго порядка – стандартные разрывные базисные функции на тетраэдре. Для построения дополнительных функций третьего порядка использовался метод, описанный в книге «Higher-Order Finite Element Methods», автор Pavel Solin. Способ аналогичен построению базисных функций третьего порядка на тетраэдрах, используемых в непрерывном методе.

Для расчёта матриц, необходимо интегрирование базисных функций их градиентов по тетраэдрам и треугольникам. Для вычисления интегралов использовались численные методы инициирования 8 порядка, 46 точек для тетраэдра и 16 точек для треугольника.

# Тестирование

Для двух крайних случаях (когда базис полностью второго или полностью третьего порядка) метод тестировался на полиномах соответствующего порядка. Решение точно совпало с искомым ответом.

# Исследования

Исследуем зависимость погрешности от количества элементов 3 порядка. В качестве тестовой функции будем использовать плотность нормального распределения:

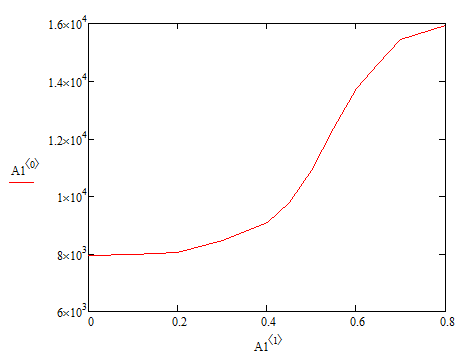


Ковариационная матрица  будет представляться в виде: . Данная функция симметричная относительно точки . И обладает следующим свойством: за пределами сферы  функция почти равна 0. Эти свойства позволяют нам варьирую параметр  делать функцию более резкой/пологой и приближать её к константе на удалении от центра симметрии.

Рассмотрим куб  в качестве расчётной области и введём величину . Если для центра тетраэдра  выполняется соотношение

,

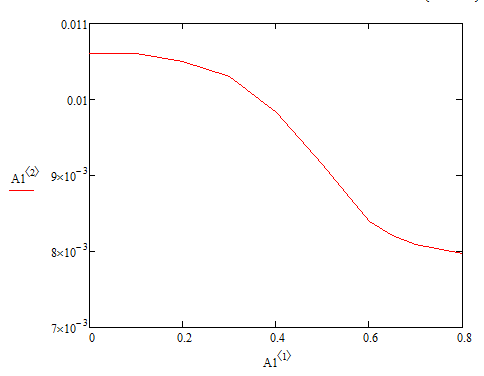
то на данном тетраэдре будем определять базис третьего порядка, иначе – базис второго.

Расчётная сетка включает в себя 796 элементов. В зависимости от радиуса количество степеней свободы меняется следующим образом:

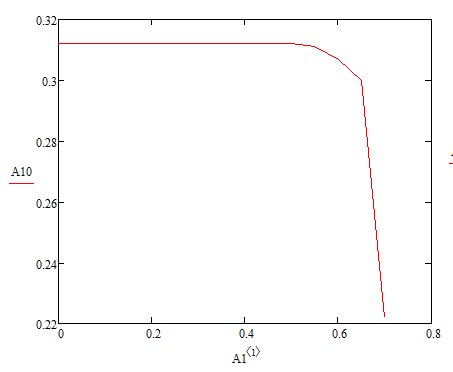
|  |  |
| --- | --- |
|  | Число степеней свободы |
| 0.00E+00 | 7960 |
| 1.00E-01 | 7970 |
| 2.00E-01 | 8060 |
| 3.00E-01 | 8450 |
| 4.00E-01 | 9090 |
| 4.50E-01 | 9800 |
| 5.00E-01 | 10900 |
| 5.50E-01 | 12340 |
| 6.00E-01 | 13740 |
| 6.50E-01 | 14620 |
| 7.00E-01 | 15460 |
| 8.00E-01 | 15920 |

Далее следую значения относительно погрешности в норме  для разных значений параметров и . Погрешность считается следующим образом:

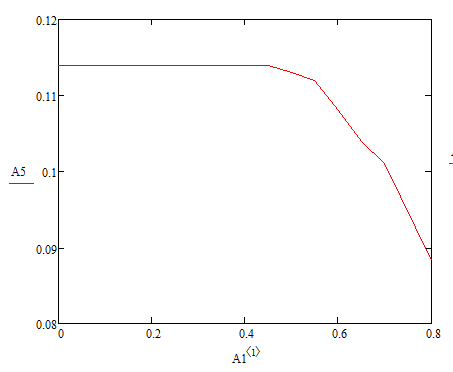




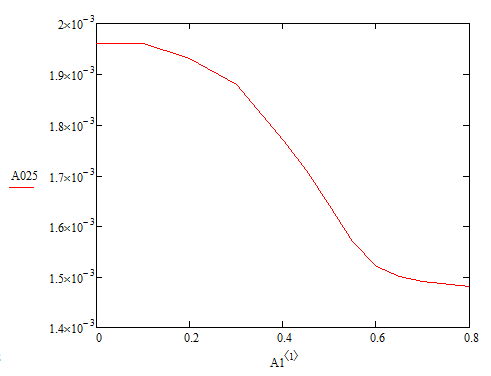
|  |  |
| --- | --- |
|  | Число степеней свободы |
| 0.00E+00 | 1.06E-02 |
| 1.00E-01 | 1.06E-02 |
| 2.00E-01 | 1.05E-02 |
| 3.00E-01 | 1.03E-02 |
| 4.00E-01 | 9.83E-03 |
| 4.50E-01 | 9.49E-03 |
| 5.00E-01 | 9.15E-03 |
| 5.50E-01 | 8.78E-03 |
| 6.00E-01 | 8.40E-03 |
| 6.50E-01 | 8.21E-03 |
| 7.00E-01 | 8.09E-03 |
| 8.00E-01 | 7.97E-03 |



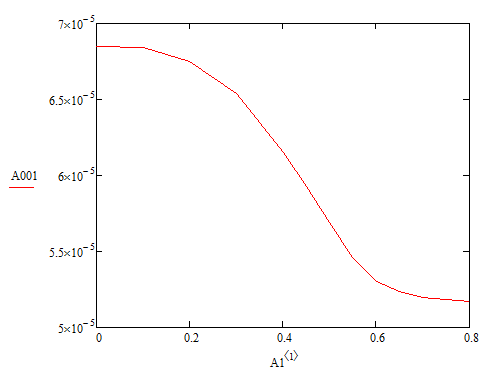
|  |  |
| --- | --- |
|  | Число степеней свободы |
| 0.00E+00 | 3.12E-01 |
| 1.00E-01 | 3.12E-01 |
| 2.00E-01 | 3.12E-01 |
| 3.00E-01 | 3.12E-01 |
| 4.00E-01 | 3.12E-01 |
| 4.50E-01 | 3.12E-01 |
| 5.00E-01 | 3.12E-01 |
| 5.50E-01 | 3.11E-01 |
| 6.00E-01 | 3.10E-01 |
| 6.50E-01 | 3.07E-01 |
| 7.00E-01 | 3.00E-01 |
| 8.00E-01 | 2.22E-01 |



|  |  |
| --- | --- |
|  | Число степеней свободы |
| 0.00E+00 | 1.14E-01 |
| 1.00E-01 | 1.14E-01 |
| 2.00E-01 | 1.14E-01 |
| 3.00E-01 | 1.14E-01 |
| 4.00E-01 | 1.14E-01 |
| 4.50E-01 | 1.14E-01 |
| 5.00E-01 | 1.13E-01 |
| 5.50E-01 | 1.12E-01 |
| 6.00E-01 | 1.08E-01 |
| 6.50E-01 | 1.04E-01 |
| 7.00E-01 | 1.01E-01 |
| 8.00E-01 | 8.83E-02 |



|  |  |
| --- | --- |
|  | Число степеней свободы |
| 0.00E+00 | 1.96E-03 |
| 1.00E-01 | 1.96E-03 |
| 2.00E-01 | 1.93E-03 |
| 3.00E-01 | 1.88E-03 |
| 4.00E-01 | 1.77E-03 |
| 4.50E-01 | 1.71E-03 |
| 5.00E-01 | 1.64E-03 |
| 5.50E-01 | 1.57E-03 |
| 6.00E-01 | 1.52E-03 |
| 6.50E-01 | 1.50E-03 |
| 7.00E-01 | 1.49E-03 |
| 8.00E-01 | 1.48E-03 |



|  |  |
| --- | --- |
|  | Число степеней свободы |
| 0.00E+00 | 6.85E-05 |
| 1.00E-01 | 6.84E-05 |
| 2.00E-01 | 6.75E-05 |
| 3.00E-01 | 6.54E-05 |
| 4.00E-01 | 6.15E-05 |
| 4.50E-01 | 5.93E-05 |
| 5.00E-01 | 5.69E-05 |
| 5.50E-01 | 5.46E-05 |
| 6.00E-01 | 5.30E-05 |
| 6.50E-01 | 5.24E-05 |
| 7.00E-01 | 5.20E-05 |
| 8.00E-01 | 5.17E-05 |

# Выводы

Использование p-техонологии для DG-метода по сравнению с классически МКЭ имеет свои положительные и отрицательные стороны.

Главной положительно стороной является то, что метод естественно позволяет работать с разными базисами на соседних элементах, что учитывается сразу в вариационной постановке. Поэтому нет необходимости в переходных элементах или других способах согласования, которые не являются частью метода.

Отрицательными качествами являются, как стандартные для DG-метода минусы – количество степеней свободы больше, чем для аналогичного непрерывного случая. Так и добавляется новое, как следствие положительной стороны: из-за того, что элементы не согласованы на границе раздела элементов (где второй порядок переходит в третий) ухудшается аппроксимация. А именно, требование непрерывности потока функции начинает выполняться слабее.

Для задач с не сильноконтрастными подобластями, p-технология разрывного Галёркина может быть излишней.

# Примеры программного кода

## Сборка матрицы

template<typename elementT, typename faceT> template<typename func\_t> void BaseElement<elementT, faceT>::generate\_matrix\_with\_out\_bound(vector<func\_t> equation\_right\_part) {

rp.resize(dofs\_n);

solutions.resize(dofs\_n);

for(int i = 0; i < dofs\_n; i++) {

rp[i] = new double [local\_dof\_n];

solutions[i] = new double [local\_dof\_n];

}

// Обнуление

for(int i = 0; i < local\_dof\_n; i++) {

di[i] = 0;

for(int k = 0; k < dofs\_n; k++) {

rp[k][i] = 0;

solutions[k][i] = 0.1;

}

}

#ifndef NOTSYM

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

gg[i] = 0;

}

#else

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

gu[i] = 0;

gl[i] = 0;

}

#endif

// Собрка

for(int el\_i = 0; el\_i < elements\_n; el\_i++) {

auto el\_dof = elements[el\_i].get\_dofs();

int el\_dof\_n = el\_dof.size();

double lambda = get\_lambda(elements[el\_i]);

auto A\_loc = elements[el\_i].get\_local\_matrix(lambda);

// Для каждого уравнения будет своя правая часть

for(int k = 0; k < dofs\_n; k++) {

auto b\_loc = elements[el\_i].get\_local\_right\_part(equation\_right\_part[k]);

for(int i = 0; i < el\_dof\_n; i++)

rp[k][el\_dof[i]] += b\_loc[i];

}

for(int i = 0; i < el\_dof\_n; i++) {

int i\_dof = el\_dof[i];

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

int pos = find\_pos(i\_dof,el\_dof[j]);

gu[pos] += A\_loc[i][j];

gl[pos] += A\_loc[j][i];

}

di[i\_dof] += A\_loc[i][i];

}

}

// Собрка

for(int el\_i = 0; el\_i < faces\_n; el\_i++) {

auto el\_dof = elements\_faces[el\_i].get\_dofs();

int el\_dof\_n = el\_dof.size();

double lambda = 1;

// Расчитаем матрицу для грани

auto A\_loc = elements\_faces[el\_i].get\_local\_matrix(lambda);

for(int i = 0; i < el\_dof\_n; i++) {

int i\_dof = el\_dof[i];

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

int pos = find\_pos(i\_dof,el\_dof[j]);

gu[pos] += A\_loc[i][j];

gl[pos] += A\_loc[j][i];

}

di[i\_dof] += A\_loc[i][i];

}

// Если грань - внешняя, то расчитаем вклады в правую часть

if (elements\_faces[el\_i].get\_el\_number() == 1) {

auto b\_loc = elements\_faces[el\_i].get\_local\_right\_part(equation\_right\_part[1]);

for(int i = 0; i < el\_dof\_n; i++) {

int i\_dof = el\_dof[i];

rp[0][i\_dof] += b\_loc[i];

}

}

}

cout << "Matrix assablation over\n";

}

## Вычисление локальных матриц

dyn\_matrix trface::get\_local\_matrix(double lambda) {

dyn\_matrix A\_loc;

A\_loc.resize(dofs\_number);

for(int i = 0; i < dofs\_number; i++)

A\_loc[i].resize(dofs\_number);

// Диагональные блоки

double mull = 1;

if (el\_count == 1)

mull = 2;

int count\_dof = 0;

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

for(int dof\_i = 0; dof\_i < elements\_dofs[el\_i]; dof\_i++) {

for(int dof\_j = 0; dof\_j < elements\_dofs[el\_i]; dof\_j++) {

//cout << dofs\_number << " " << count\_dof << " " << dof\_i << " " << dof\_j << " " << el\_count << " "<< el\_i << endl;

A\_loc[count\_dof+dof\_i][count\_dof+dof\_j] = integrate([&](double x, double y, double z)->double {

double phi1, phi2;

vec3d v1, v2;

phi1 = face\_elements[el\_i]->scalar\_basis\_v(dof\_i, x, y, z);

phi2 = face\_elements[el\_i]->scalar\_basis\_v(dof\_j, x, y, z);

v1 = face\_elements[el\_i]->scalar\_basis\_grad\_v(dof\_i, x, y, z);

v2 = face\_elements[el\_i]->scalar\_basis\_grad\_v(dof\_j, x, y, z);

vec3d nv = normal\_vector;

double res = lambda \* (phi1 \* v2 - phi2 \* v1) \* normals[el\_i] \* mull / 2.0;

return res;

});

}

}

count\_dof += elements\_dofs[el\_i];

}

// Внедиагональные блоки

for(int dof1\_i = 0; dof1\_i < elements\_dofs[0]; dof1\_i++) {

for(int dof2\_j = 0; dof2\_j < elements\_dofs[1]; dof2\_j++) {

//cout << dofs\_number << " " << elements\_dofs[0] << " " << dof1\_i << " " << dof2\_j << endl;

A\_loc[dof1\_i][elements\_dofs[0]+dof2\_j] = integrate([&](double x, double y, double z)->double {

double phi1, phi2;

vec3d v1, v2;

phi1 = face\_elements[0]->scalar\_basis\_v(dof1\_i, x, y, z);

phi2 = face\_elements[1]->scalar\_basis\_v(dof2\_j, x, y, z);

v1 = face\_elements[0]->scalar\_basis\_grad\_v(dof1\_i, x, y, z);

v2 = face\_elements[1]->scalar\_basis\_grad\_v(dof2\_j, x, y, z);

double res = lambda \* (phi1 \* v2 \* normals[0] - phi2 \* v1 \* normals[1]) / 2.0;

return res;

});

A\_loc[elements\_dofs[0]+dof2\_j][dof1\_i] = -A\_loc[dof1\_i][elements\_dofs[0]+dof2\_j];

}

}

return A\_loc;

}

## Вычисление базисных функций в точке

vec3d tetelement::scalar\_basis\_grad\_v(int i, double x, double y, double z) {

point p\_glob(x, y, z);

vec3d result;

// То, что заместо первого порядка

if (i <= 3) {

double lambda\_i = lambda(i, p\_glob);

vec3d lambda\_g = grad\_lambda(i);

result = (2\*lambda\_i - 0.5) \* lambda\_g;

}

// Рёберные функции второго порядка

else if(i <= 9) {

int shift = i - 4;

double lambda\_1 = lambda(edge\_lambdas[shift][0], p\_glob);

double lambda\_2 = lambda(edge\_lambdas[shift][1], p\_glob);

vec3d lambda\_1\_g = grad\_lambda(edge\_lambdas[shift][0]);

vec3d lambda\_2\_g = grad\_lambda(edge\_lambdas[shift][1]);

result = 2 \* (lambda\_1 \* lambda\_2\_g + lambda\_2 \* lambda\_1\_g);

}

// Рёберные функции третьего порядка

else if (i <= 15) {

int shift = i - 10;

double lambda\_1 = lambda(edge\_lambdas[shift][0], p\_glob);

double lambda\_2 = lambda(edge\_lambdas[shift][1], p\_glob);

double ker\_val = kernel(1, lambda\_1 - lambda\_2);

vec3d lambda\_1\_g = grad\_lambda(edge\_lambdas[shift][0]);

vec3d lambda\_2\_g = grad\_lambda(edge\_lambdas[shift][1]);

double d\_ker = kernel\_d(1, lambda\_1 - lambda\_2);

result = lambda\_1 \* ker\_val \* lambda\_2\_g + lambda\_2 \* ker\_val\* lambda\_1\_g + lambda\_1 \* lambda\_2 \* d\_ker \* (lambda\_1\_g - lambda\_2\_g);

}

// Функции третьего порядка, ассациированные с гранями

else if (i <= 19) {

int shift = i - 16;

array<double, 3> face\_lambda;

array<vec3d, 3> face\_grads;

for(int f\_i = 0; f\_i < 3; f\_i++) {

face\_lambda[f\_i] = lambda(face\_lambdas[shift][f\_i], p\_glob);

face\_grads[f\_i] = grad\_lambda(face\_lambdas[shift][f\_i]);

}

result = face\_lambda[0] \* face\_lambda[1] \* face\_grads[2] + face\_lambda[0] \* face\_lambda[2] \* face\_grads[1] + face\_lambda[1] \* face\_lambda[2] \* face\_grads[0];

}

return result;

}