Министерство образования и науки Российской Федерации

Новосибирский государственный технический университет

Кафедра прикладной математики

Методы конечноэлементного анализа

Лабораторные работы №2-4

Факультет	ПМИ
Группа	ПММ-42
Студенты	Александров М.Е.
	Жигалов П.С.
Преподаватели	Домников П.А.
	Персова М.Г.
Варианты	4, 2, 2

Новосибирск

2014

1. Цель работы

Реализовать алгоритмы нумерации глобальных базисных функций в конечноэлементной сетке. Реализовать алгоритм построения портрета и сборки конечноэлементной матрицы для разреженного строчного формата хранения. Протестировать написанную программу. Изучить методы построения изображений конечноэлементного решения при решении различных задач. Реализовать методы построения изображения решения для элементов высоких порядков.

2. Задание

Л/р 2. Написать программу нумерации лагранжевых базисных функций второго порядка в конечноэлементной сетке. При этом использовать конечноэлементную сетку, построенную в лабораторной работе 1.

Л/р 3. Написать подпрограмму построения портрета матрицы, возникающей при решении эллиптической краевой задачи методом конечных элементов с использованием лагранжевых базисных функций второго порядка.

Л/р 4. Написать программу рисования поля на четырехугольной сетке, разбивая каждый четырехугольник сначала на два треугольника, а затем на более мелкие треугольники посредством проведения линий через середины ребер.

3. Теоретическая часть

3.1. Лагранжевы базисные функции второго порядка

Одномерные лагранжевы квадратичные базисные функции на единичном отрезке:

$$\begin{aligned} \hat{\varphi}_1(\xi) &= 2(\xi-0.5)(\xi-1), \\ \hat{\varphi}_1(\xi) &= -4\xi(\xi-1), \\ \hat{\varphi}_1(\xi) &= 2\xi(\xi-0.5). \end{aligned} \tag{1}$$

Двумерные базисные функции на единичном квадрате:

$$\hat{\psi}_i(x,y) = \hat{\varphi}_{\mu(i)}(x)\hat{\varphi}_{\nu(i)}(y), \quad \mu(i) = ((i-1) \bmod 3) + 1, \quad \mu(i) = \left[\frac{i-1}{3}\right] + 1, \tag{2}$$

где mod - остаток от деления, $[\,]$ - целая часть числа.

3.2. Базисные функции на четырехугольниках

Отобразим единичный квадрат $\Omega^E = \{(\xi,\eta) \mid 0 \le \xi \le 1, 0 \le \eta \le 1\}$ в четырехугольник Ω_k с вершинами $(\hat{x}_i, \hat{y}_i)$ с помощью следующих соотношений:

$$\begin{aligned} x &= (1-\xi)(1-\eta)\hat{x}_1 + \xi(1-\eta)\hat{x}_2 + (1-\xi)\eta\hat{x}_3 + \xi\eta\hat{x}_4, \\ y &= (1-\xi)(1-\eta)\hat{y}_1 + \xi(1-\eta)\hat{y}_2 + (1-\xi)\eta\hat{y}_3 + \xi\eta\hat{y}_4. \end{aligned} \tag{3}$$

Будем использовать следующие обозначения:

$$\alpha_0=(\hat{x}_2-\hat{x}_1)(\hat{y}_3-\hat{y}_1)-(\hat{y}_2-\hat{y}_1)(\hat{x}_3-\hat{x}_1),$$
$$\alpha_1=(\hat{x}_2-\hat{x}_1)(\hat{y}_4-\hat{y}_3)-(\hat{y}_2-\hat{y}_1)(\hat{x}_4-\hat{x}_3), \quad (4)$$
$$\alpha_3=(\hat{y}_3-\hat{y}_1)(\hat{x}_4-\hat{x}_2)-(\hat{x}_3-\hat{x}_1)(\hat{y}_4-\hat{y}_2),$$

$$\beta_1=\hat{x}_3-\hat{x}_1, \quad \beta_2=\hat{x}_2-\hat{x}_1, \quad \beta_3=\hat{y}_3-\hat{y}_1, \quad \beta_4=\hat{y}_2-\hat{y}_1,$$
$$\beta_5=\hat{x}_1-\hat{x}_2-\hat{x}_3+\hat{x}_4, \quad \beta_6=\hat{y}_1-\hat{y}_2-\hat{y}_3+\hat{y}_4. \quad (5)$$

С учетом этих обозначений, матрицы массы и жесткости примут вид:

$$\hat{M}_{i,j}=\iint_{\Omega_k}\hat{\psi}_i(x,y)\hat{\psi}_j(x,y)\,dx\,dy=\iint_{\Omega^E}\hat{\psi}_i(\xi,\eta)\hat{\psi}_j(\xi,\eta)|J|\,d\xi\,d\eta, \quad (6)$$

$$\hat{G}_{i,j}=\iint_{\Omega_k}\nabla\hat{\psi}_i(x,y)\cdot\nabla\hat{\psi}_j(x,y)\,dx\,dy=\operatorname{sign}(\alpha_0)\iint_{\Omega^E}\frac{1}{|J|}*$$
$$*\left(\left[\frac{\partial\hat{\psi}_i(\xi,\eta)}{\partial\xi}(\beta_6\xi+\beta_3)-\frac{\partial\hat{\psi}_i(\xi,\eta)}{\partial\eta}(\beta_6\eta+\beta_4)\right]\left[\frac{\partial\hat{\psi}_j(\xi,\eta)}{\partial\xi}(\beta_6\xi+\beta_3)-\frac{\partial\hat{\psi}_j(\xi,\eta)}{\partial\eta}(\beta_6\eta+\beta_4)\right]+\right. \quad (7)$$
$$\left.+\left[\frac{\partial\hat{\psi}_i(\xi,\eta)}{\partial\eta}(\beta_5\eta+\beta_2)-\frac{\partial\hat{\psi}_i(\xi,\eta)}{\partial\xi}(\beta_5\xi+\beta_1)\right]\left[\frac{\partial\hat{\psi}_j(\xi,\eta)}{\partial\eta}(\beta_5\eta+\beta_2)-\frac{\partial\hat{\psi}_j(\xi,\eta)}{\partial\xi}(\beta_5\xi+\beta_1)\right]\right)d\xi\,d\eta$$

где $i,j=1..4$, $|J|=\alpha_0+\alpha_1\xi+\alpha_2\eta$.

Для перевода четырехугольника Ω_k в единичный квадрат Ω^E воспользуемся следующими соотношениями:

$$\xi=\frac{\beta_3(x-\hat{x}_1)-\beta_1(y-\hat{y}_1)}{\beta_2\beta_3-\beta_1\beta_4}, \quad \eta=\frac{\beta_2(y-\hat{y}_1)-\beta_4(x-\hat{x}_1)}{\beta_2\beta_3-\beta_1\beta_4} \quad \text{при } \alpha_1=\alpha_2=0;$$

$$\xi=\frac{\alpha_2(x-\hat{x}_1)+\beta_1 w(x,y)}{\alpha_2\beta_2-\beta_5 w(x,y)}, \quad \eta=-\frac{w(x,y)}{\alpha_2} \quad \text{при } \alpha_1=0 \text{ и } \alpha_2\neq 0;$$

$$\xi=\frac{w(x,y)}{\alpha_1}, \quad \eta=\frac{\alpha_1(y-\hat{y}_1)-\beta_4 w(x,y)}{\alpha_1\beta_3-\beta_6 w(x,y)} \quad \text{при } \alpha_1\neq 0 \text{ и } \alpha_2=0; \quad (8)$$

$$\beta_5\alpha_2\eta^2+(\alpha_2\beta_2+\alpha_1\beta_1+\beta_5 w(x,y))\eta+\alpha_1(\hat{x}_1-x)+\beta w(x,y)=0,$$

$$\xi=\frac{\alpha_2}{\alpha_1}\eta+\frac{w(x,y)}{\alpha_1} \quad \text{при } \alpha_1\neq 0 \text{ и } \alpha_2\neq 0;$$

где $w(x,y)=\beta_6(x-\hat{x}_1)-\beta_5(y-\hat{y}_1)$.

3.3. Алгоритм нумерации базисных функций

Построим упорядоченный список ребер по всей области. Ребра будем упорядочивать по возрастанию минимальных номеров определяющих их вершин.

Далее упорядочим глобальные базисные функции по геометрическому принципу. Для этого будем последовательно обходить все конечные элементы, а уже в них обходить все локальные ба-

зисные функции. Если глобальная базисная функция уже была пронумерована, то будем использовать старый номер, иначе выделим для базисной функции первый незанятый номер.

Таким образом, при правильной нумерации конечных элементов будет обеспечено расположение рядом как минимум четырех базисных функций с конечного элемента и еще двух со смещением не более чем 9. Смещение трех оставшихся базисных функции будет зависеть от расположения КЭ.

4. Исследования

4.1. Тест на линейной функции

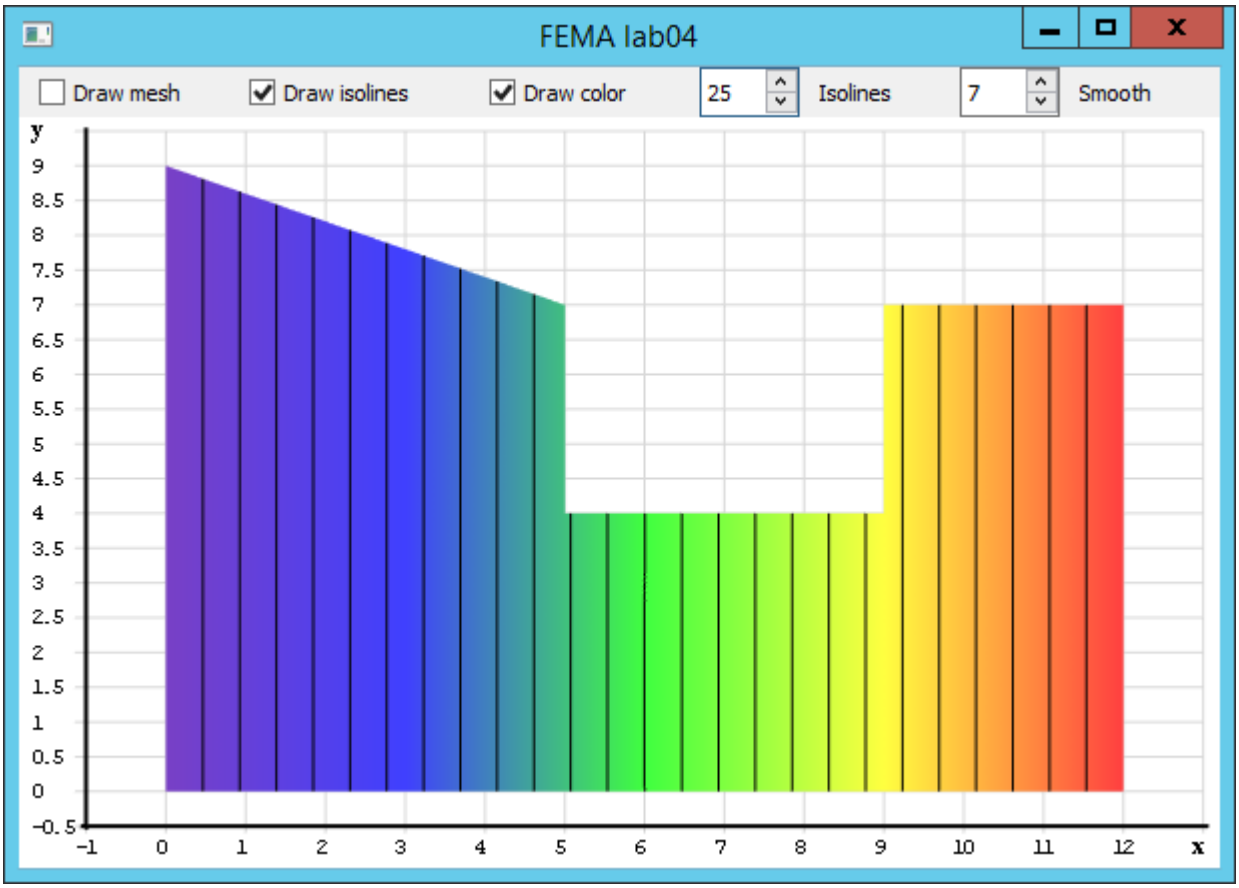
Уравнение: $-\Delta u + u = x$.

Аналитическое решение: $u = x$.

Краевые условия: первого рода на внешних вертикальных ребрах, второго рода на остальных.

Погрешность: $< 10^{-16}$.

Графическое представление решения:



4.2. Тест на проверку качества изолиний

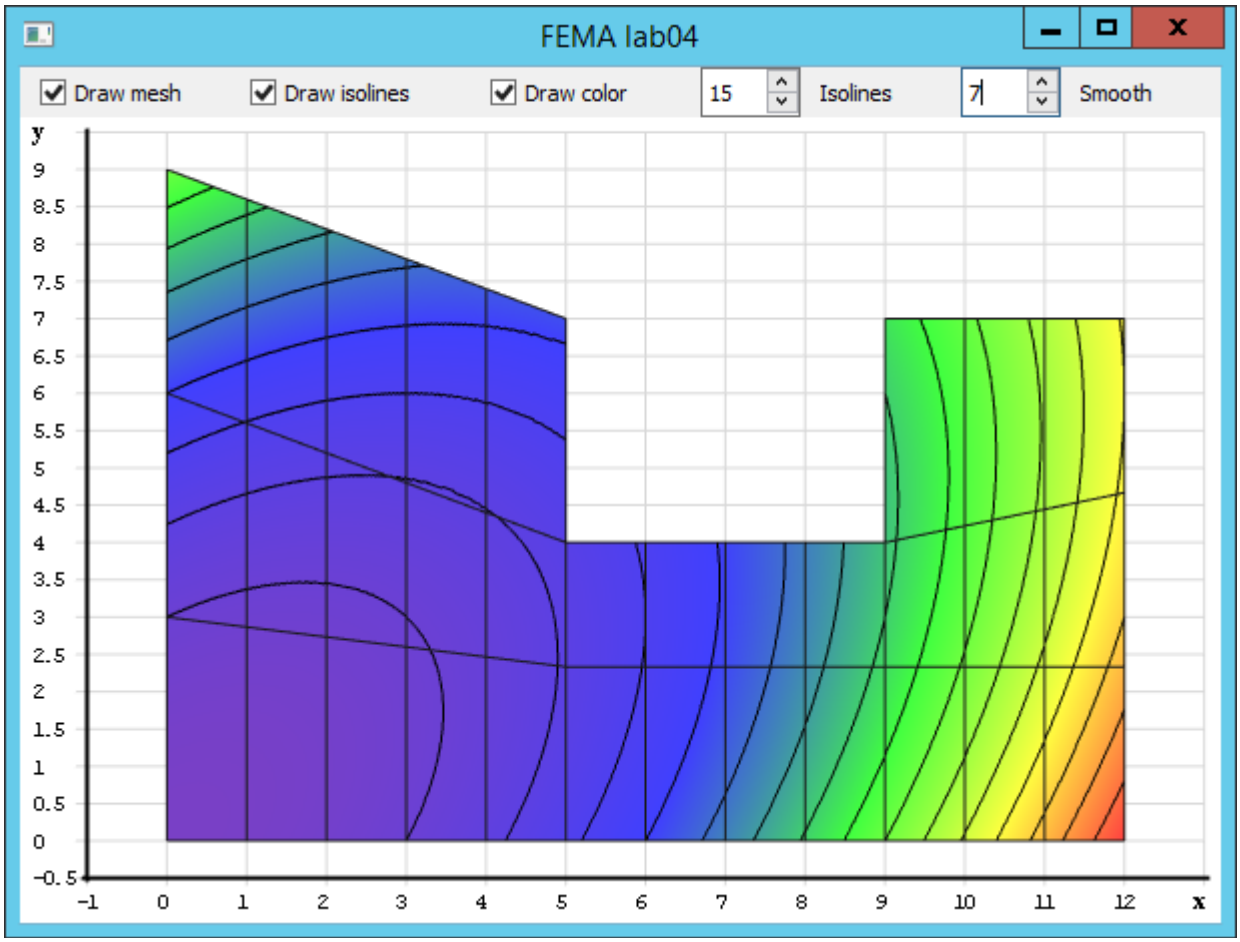
Уравнение: $-\Delta u+u=-4+x^2+y^2-xy$.

Аналитическое решение: $u=x^2+y^2-xy$.

Краевые условия: первого рода на всех внешних ребрах.

Погрешность: $<10^{-16}$.

Графическое представление решения:



4.3. Тест на порядок аппроксимации

Уравнение: $-\Delta u+u=-0.02e^{0.1(x+y)}+e^{0.1(x+y)}$.

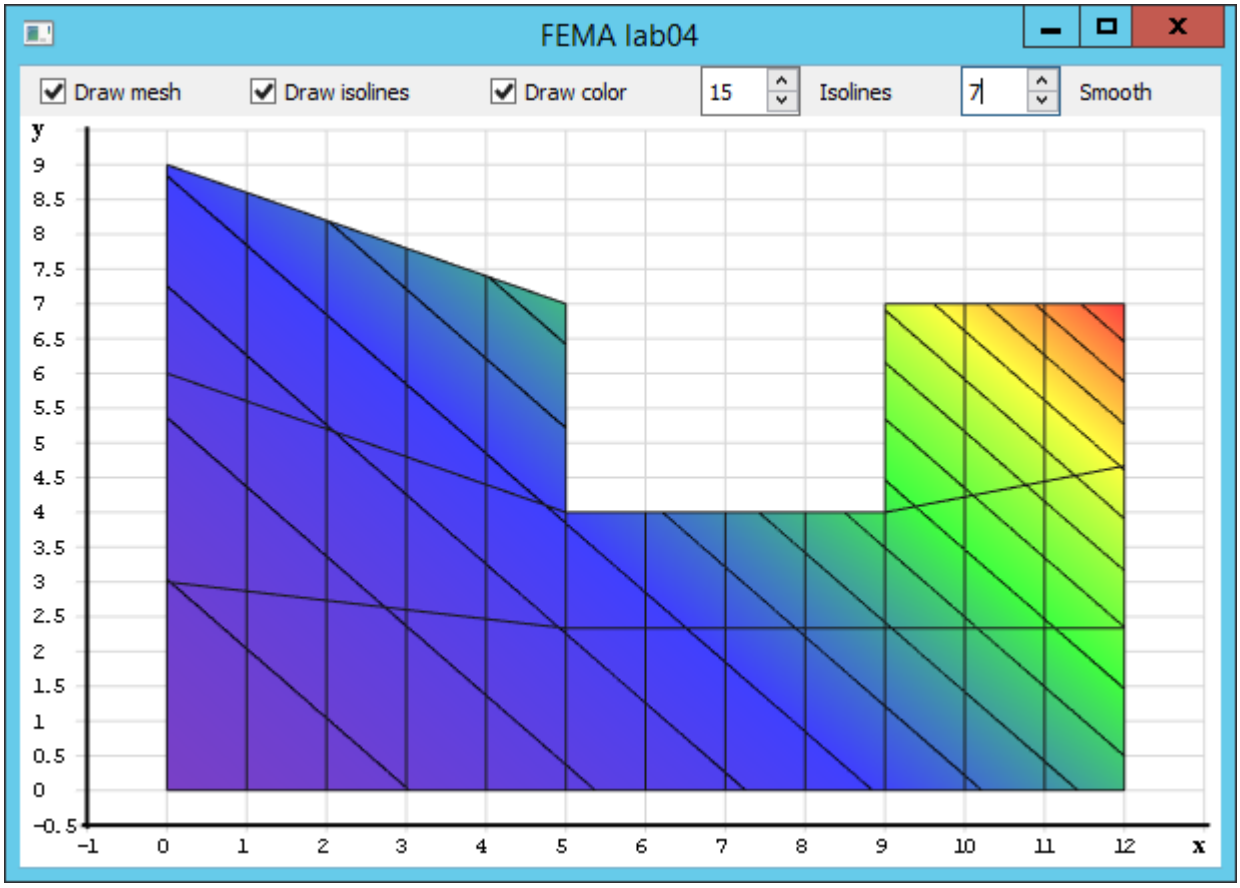
Аналитическое решение: $u=e^{0.1(x+y)}$.

Краевые условия: первого рода на всех внешних ребрах.

Погрешности и порядок аппроксимации

$\sqrt{\frac{\|u-u^h\|}{\|u\|}}$	$\sqrt{\frac{\|u-u^{h/2}\|}{\|u\|}}$	$\sqrt{\frac{\|u-u^{h/4}\|}{\|u\|}}$	$\log_2\sqrt{\frac{\|u-u^h\|}{\|u-u^{h/2}\|}}$	$\log_2\sqrt{\frac{\|u-u^{h/2}\|}{\|u-u^{h/4}\|}}$
7,801E-06	5,654E-07	3,559E-08	3,79	3,99

Графическое представление решения:



5. Выводы

Разработанная программа успешно решила все предоставленные ей тестовые задачи. Метод с использованием биквадратичных базисных функций показал четвертый порядок аппроксимации, что совпадает с теоретическим значением. Построенные изображения конечноэлементных решений также совпадают с их теоретическим видом.

6. Код программы (фрагменты)

Файл fem.h

```
// Класс точка
class point
{
public:
    double x;
    double y;
    size_t num;
    point();
    point(double x, double y);
    point(double x, double y, size_t num);
    point(const point & p);
    double & operator [] (size_t i);
    double operator [] (size_t i) const;
    bool operator < (const point & t) const;
    bool operator == (const point & t) const;
    point & operator = (const point & other);
    friend ostream & operator << (ostream & os, point a);
};

// Класс физическая область
class phys_area
{
public:
    double lambda;
    double gamma;
    size_t num;
    size_t gmsh_phys_num;
};

// Класс четырехугольник
class quadrilateral
{
public:
    friend class glwidget;
    point * nodes[4];
    phys_area * ph;
    void init();
    double get_local_G(size_t i, size_t j) const;
    double get_local_M(size_t i, size_t j) const;
    double get_local_rp(size_t i) const;
    bool inside(const point & p) const;
    double bfunc_2d(size_t num, const point & p) const;
    point grad_bfunc_2d(size_t num, const point & p) const;
    size_t degree_of_freedom[9];
private:
    point to_local(const point & p) const;
    point to_global(const point & p) const;
    point gauss_points[9];
    double gauss_weights[9];
    double det_J_local(const point & p) const;
    double alpha0, alpha1, alpha2;
    double beta1, beta2, beta3, beta4, beta5, beta6;
    double S;
    void two2one(size_t two, size_t & one1, size_t & one2) const;
    double bfunc_1d(size_t func_n, double ksi) const;
    double dbfunc_1d(size_t func_n, double ksi) const;
    double bfunc_2d_local(size_t num, const point & p) const;
    point grad_bfunc_2d_local(size_t num, const point & p) const;
};

// Класс СЛАУ
class SLAE
{
public:
    SLAE();
    ~SLAE();
    void solve(double eps);
    void alloc_all(size_t n_size, size_t gg_size);
    void add(size_t i, size_t j, double elem);
    double * gg, * di, * rp, * x;
    size_t * ig, * jg;
    size_t n;
private:
    CGM_LLT solver;
};

// Класс ребро
class edge
{
public:
    const point * nodes[2];
    size_t gmsh_phys_num;
    size_t degree_of_freedom[3];
    bool operator < (const edge & t) const;
    bool operator == (const edge & t) const;
    edge();
    edge(const point * begin, const point * end);
    point get_freedom_position(size_t num) const;
    double get_matrix_M(size_t i, size_t j);
    double bfunc_1d(size_t func_n, double x) const;
    size_t num;
    const quadrilateral * fes[2];
};

// Класс МКЭ
class FEM
{
public:
    friend class glwidget;
```

```
    FEM();
    ~FEM();
    SLAE slae;
    void input();
    void make_portrait();
    void assembling_global();
    void applying_bounds();
    double get_solution(const point & p) const;
    double get_solution(const point & p, const quadrilateral * fe) const;
    point get_grad(const point & p) const;
    point get_grad(const point & p, const quadrilateral * fe) const;
    const quadrilateral * get_fe(const point & p) const;
    // lab02
    vector<edge>::const_iterator get_edge_by_nodes(const point & begin, const point & end) const;
    vector<edge>::const_iterator get_edge_by_fe(const quadrilateral & fe, size_t edge_num) const;
    const point * get_nodes_by_edge(const edge & e, size_t node_num) const;
    const quadrilateral * get_fe_by_edge(const edge & e, size_t fe_num) const;
private:
    point * nodes;
    size_t nodes_num;
    quadrilateral * qls;
    size_t qls_num;
    phys_area * phs;
    size_t phs_num;
    edge * bounds;
    size_t bounds_num;
    size_t degree_of_freedom_num;
    vector<edge> edges_freedom;
    void add_edge_freedom(const point * begin, const point * end, size_t ql_num);
};

double func_rp(const point & p, const quadrilateral * r);
size_t get_type_of_bound(size_t gmsh_num);
double get_bound_value(const point & p, const edge & e);

#endif // FEM_H
```

Файл fem.cpp

```
void FEM::add_edge_freedom(const point * begin, const point * end, size_t ql_num)
{
    edge ed(begin, end);
    edge ei(end, begin);
    vector<edge>::iterator id = find(edges_freedom.begin(), edges_freedom.end(), ed);
    vector<edge>::iterator ii = find(edges_freedom.begin(), edges_freedom.end(), ei);
    if(id == edges_freedom.end() && ii == edges_freedom.end())
    {
        edges_freedom.push_back(ed);
        edges_freedom[edges_freedom.size()-1].fes[0] = qls + ql_num;
    }
    else
    {
        vector<edge>::iterator it;
        edge et;
        if(ii == edges_freedom.end())
        {
            it = id;
            et = ed;
        }
        else
        {
            it = ii;
            et = ei;
        }
        if(!it->fes[0])
            it->fes[0] = qls + ql_num;
        else if(!it->fes[1])
            it->fes[1] = qls + ql_num;
        else
            assert(!it->fes[0] || !it->fes[1]);
    }
}

void FEM::input()
{
    ifstream ifs;
    cout << "Reading data..." << endl;

    ifs.open("data/phys.txt", ios::in);
    assert(ifs.good());
    ifs >> phs_num;
    cout << " > detected " << phs_num << " physical areas" << endl;
    phs = new phys_area [phs_num];
    for(size_t i = 0; i < phs_num; i++)
    {
        ifs >> phs[i].gmsh_phys_num;
        ifs >> phs[i].lambda;
        ifs >> phs[i].gamma;
        phs[i].num = i;
    }
    ifs.close();

    string line;
    double garbage;
    ifstream gmsh_file;
    gmsh_file.open("data/mesh.msh", ios::in);
    assert(gmsh_file.good());
    do
        getline(gmsh_file, line);
    while(line.find("$Nodes") == string::npos && gmsh_file.good());

    gmsh_file >> nodes_num;
    cout << " > detected " << nodes_num << " nodes" << endl;
    nodes = new point[nodes_num];
    for(size_t i = 0; i < nodes_num; i++)
    {
```

```
    gmsh_file >> garbage >> nodes[i].x >> nodes[i].y >> garbage;
    nodes[i].num = i;
}

do
    getline(gmsh_file, line);
while(line.find("$Elements") == string::npos && gmsh_file.good());

size_t num_elem;
size_t type_elem;
vector<edge> tmp_bounds;
vector<quadrilateral> tmp_rects;
gmsh_file >> num_elem;
for(size_t i = 0; i < num_elem; i++)
{
    gmsh_file >> garbage >> type_elem;

    if(type_elem == 1)
    {
        edge tmp_edge;
        gmsh_file >> garbage >> tmp_edge.gmsh_phys_num >> garbage;
        for(int j = 0; j < 2; j++)
        {
            size_t tmp_node;
            gmsh_file >> tmp_node;
            tmp_edge.nodes[j] = nodes + tmp_node - 1;
        }
        if(tmp_edge.nodes[0]->num > tmp_edge.nodes[1]->num)
            swap(tmp_edge.nodes[0], tmp_edge.nodes[1]);
        tmp_bounds.push_back(tmp_edge);
    }
    else if(type_elem == 3)
    {
        quadrilateral tmp_rect;

        size_t gmsh_phys_num;
        gmsh_file >> garbage >> gmsh_phys_num >> garbage;
        tmp_rect.ph = NULL;
        for(size_t j = 0; j < phs_num; j++)
            if(phs[j].gmsh_phys_num == gmsh_phys_num)
                tmp_rect.ph = phs + j;

        for(int j = 0; j < 4; j++)
        {
            size_t tmp_node;
            gmsh_file >> tmp_node;
            tmp_rect.nodes[j] = nodes + tmp_node - 1;
        }
        swap(tmp_rect.nodes[2], tmp_rect.nodes[3]);

        tmp_rects.push_back(tmp_rect);
    }
}
gmsh_file.close();

qls_num = tmp_rects.size();
cout << " > detected " << qls_num << " rectangles" << endl;
qls = new quadrilateral [qls_num];
for(size_t i = 0; i < qls_num; i++)
    qls[i] = tmp_rects[i];
tmp_rects.clear();

bounds_num = tmp_bounds.size();
cout << " > detected " << bounds_num << " bounds" << endl;
bounds = new edge [bounds_num];
for(size_t i = 0; i < bounds_num; i++)
    bounds[i] = tmp_bounds[i];
tmp_bounds.clear();

// Вот тут будем делать степени свободы
for(size_t i = 0; i < qls_num; i++)
{
    add_edge_freedom(qls[i].nodes[0], qls[i].nodes[1], i);
    add_edge_freedom(qls[i].nodes[0], qls[i].nodes[2], i);
    add_edge_freedom(qls[i].nodes[1], qls[i].nodes[3], i);
    add_edge_freedom(qls[i].nodes[2], qls[i].nodes[3], i);
}
sort(edges_freedom.begin(), edges_freedom.end());
vector<edge>::iterator it;
it = unique(edges_freedom.begin(), edges_freedom.end());
edges_freedom.resize(distance(edges_freedom.begin(), it));

// Занумеруем ребра
for(size_t i = 0; i < edges_freedom.size(); i++)
    edges_freedom[i].num = i;

// У узлов пусть будут номера узлов
for(size_t i = 0; i < edges_freedom.size(); i++)
{
    edges_freedom[i].degree_of_freedom[0] = edges_freedom[i].nodes[0]->num;
    edges_freedom[i].degree_of_freedom[2] = edges_freedom[i].nodes[1]->num;
}
// Переменная, содержащая свободную степень свободы
size_t curr_freedom = nodes_num;
// Теперь заполним степени свободы на ребрах
for(size_t i = 0; i < edges_freedom.size(); i++)
{
    edges_freedom[i].degree_of_freedom[1] = curr_freedom;
    curr_freedom++;
}
// Дальше заполним центры и узлы, их сразу
for(size_t i = 0; i < qls_num; i++)
{
    qls[i].degree_of_freedom[4] = curr_freedom;
    curr_freedom++;
    qls[i].degree_of_freedom[0] = qls[i].nodes[0]->num;
    qls[i].degree_of_freedom[2] = qls[i].nodes[1]->num;
```

```
        qls[i].degree_of_freedom[6] = qls[i].nodes[2]->num;
        qls[i].degree_of_freedom[8] = qls[i].nodes[3]->num;
    }
    // Теперь и то, что на ребрах
    for (size_t i = 0; i < qls_num; i++)
    {
        vector<edge>::iterator it;
        it = find(edges_freedom.begin(), edges_freedom.end(), edge(qls[i].nodes[0], qls[i].nodes[1]));
        assert(it != edges_freedom.end());
        qls[i].degree_of_freedom[1] = it->degree_of_freedom[1];
        it = find(edges_freedom.begin(), edges_freedom.end(), edge(qls[i].nodes[0], qls[i].nodes[2]));
        assert(it != edges_freedom.end());
        qls[i].degree_of_freedom[3] = it->degree_of_freedom[1];
        it = find(edges_freedom.begin(), edges_freedom.end(), edge(qls[i].nodes[1], qls[i].nodes[3]));
        assert(it != edges_freedom.end());
        qls[i].degree_of_freedom[5] = it->degree_of_freedom[1];
        it = find(edges_freedom.begin(), edges_freedom.end(), edge(qls[i].nodes[2], qls[i].nodes[3]));
        assert(it != edges_freedom.end());
        qls[i].degree_of_freedom[7] = it->degree_of_freedom[1];
    }

    // Заполним также инфу о степенях свободы в ребрах для краевых
    for(size_t i = 0; i < bounds_num; i++)
    {
        vector<edge>::iterator it;
        it = find(edges_freedom.begin(), edges_freedom.end(), bounds[i]);
        if(it == edges_freedom.end()) // Здравствуй жопа новый год
        {
            swap(bounds[i].nodes[0], bounds[i].nodes[1]);
            it = find(edges_freedom.begin(), edges_freedom.end(), bounds[i]);
        }
        assert(it != edges_freedom.end());
        for(size_t j = 0; j < 3; j++)
            bounds[i].degree_of_freedom[j] = it->degree_of_freedom[j];
        // За одно и инфу о КЭ
        for(size_t j = 0; j < 2; j++)
            bounds[i].fes[j] = it->fes[j];
    }

    degree_of_freedom_num = curr_freedom;
    cout << " > detected " << degree_of_freedom_num << " degree of freedom" << endl;

    // Перенумерация степеней свободы
    map<size_t, size_t> convert;
    size_t curr_deg = 0;
    for(size_t i = 0; i < qls_num; i++)
    {
        for(size_t j = 0; j < 9; j++)
        {
            if(convert.find(qls[i].degree_of_freedom[j]) == convert.end())
            {
                convert[qls[i].degree_of_freedom[j]] = curr_deg;
                curr_deg++;
            }
        }
    }

    // Применение перенумерованных значений
    // В четырехугольниках
    for(size_t i = 0; i < qls_num; i++)
        for(size_t j = 0; j < 9; j++)
            qls[i].degree_of_freedom[j] = convert[qls[i].degree_of_freedom[j]];
    // В ребрах
    for(size_t i = 0; i < edges_freedom.size(); i++)
        for(size_t j = 0; j < 3; j++)
            edges_freedom[i].degree_of_freedom[j] = convert[edges_freedom[i].degree_of_freedom[j]];
    // В краевых
    for(size_t i = 0; i < bounds_num; i++)
        for(size_t j = 0; j < 3; j++)
            bounds[i].degree_of_freedom[j] = convert[bounds[i].degree_of_freedom[j]];

    for(size_t i = 0; i < qls_num; i++)
        qls[i].init();
}

void FEM::make_portrait()
{
    // Формирование профиля (портрета)
    cout << "Generating profile..." << endl;
    size_t gg_size = 0;
    // Создаем массив списков для хранения связей
    set<size_t> * profile = new set<size_t> [degree_of_freedom_num];

    // Связь есть, если узлы принадлежат одному КЭ
    // Поэтому обходим конечные элементы и добавляем в список общие вершины
    for(size_t i = 0; i < qls_num; i++)
        for(size_t j = 0; j < 9; j++)
            for(size_t k = 0; k < j; k++)
            {
                size_t ik = qls[i].degree_of_freedom[k];
                size_t ij = qls[i].degree_of_freedom[j];
                if(ik > ij) swap(ik, ij);
                profile[ik].insert(ij);
            }

    // Удаляем повторяющиеся записи в спиках и сортируем их, попутно считая размер матрицы
    for(size_t i = 0; i < degree_of_freedom_num; i++)
        gg_size += profile[i].size();
    slae.alloc_all(degree_of_freedom_num, gg_size);

    cout << " > slae.n_size = " << degree_of_freedom_num << endl;
    cout << " > slae.gg_size = " << gg_size << endl;

    // Заполнение профиля (портрета)
    slae.ig[0] = 0;
    slae.ig[1] = 0;
    size_t tmp = 0;
```

```
    for(size_t i = 0; i < slae.n; i++)
    {
        size_t k = 0;
        for(size_t j = 0; j <= i; j++)
        {
            // Если есть связь между i и j, значит в этом месте матрицы будет ненулевой элемент
            // занесем информацию об этом в jg
            if(profile[j].find(i) != profile[j].end())
            {
                slae.jg[tmp] = j;
                tmp++;
                k++;
            }
        }
        // a в ig занесем информацию о количестве ненулевых элементов в строке
        slae.ig[i + 1] = slae.ig[i] + k;
    }

    // Очистка списков
    for(size_t i = 0; i < degree_of_freedom_num; i++)
        profile[i].clear();
    delete [] profile;
}

void FEM::assembling_global()
{
    cout << "Assembling global matrix..." << endl;
    for(size_t k = 0; k < qls_num; k++)
    {
        for(size_t i = 0; i < 9; i++)
        {
            for(size_t j = 0; j <= i; j++)
            {
                slae.add(qls[k].degree_of_freedom[i], qls[k].degree_of_freedom[j],
                         qls[k].get_local_G(i, j) * qls[k].ph->lambda);
                slae.add(qls[k].degree_of_freedom[i], qls[k].degree_of_freedom[j],
                         qls[k].get_local_M(i, j) * qls[k].ph->gamma);
            }

            slae.rp[qls[k].degree_of_freedom[i]] += qls[k].get_local_rp(i);
        }
    }
}

void FEM::applying_bounds()
{
    cout << "Applying bounds..." << endl;
    for(size_t bi = 0; bi < bounds_num; bi++)
    {
        if(get_type_of_bound(bounds[bi].gmsh_phys_num) == 2)
        {
            double theta[3] =
            {
                get_bound_value(bounds[bi].get_freedom_position(0), bounds[bi]),
                get_bound_value(bounds[bi].get_freedom_position(1), bounds[bi]),
                get_bound_value(bounds[bi].get_freedom_position(2), bounds[bi])
            };

            double lambda = bounds[bi].fes[0]->ph->lambda;
            for(size_t i = 0; i < 3; i++)
            {
                double vfrr = 0.0;
                for(size_t j = 0; j < 3; j++)
                    vfrr += bounds[bi].get_matrix_M(i, j) * theta[j] / lambda;
                slae.rp[bounds[bi].degree_of_freedom[i]] += vfrr;
            }
        }
    }

    for(size_t bi = 0; bi < bounds_num; bi++)
    {
        if(get_type_of_bound(bounds[bi].gmsh_phys_num) == 1)
        {
            for(size_t j = 0; j < 3; j++)
            {
                size_t freedom = bounds[bi].degree_of_freedom[j];
                double val = get_bound_value(bounds[bi].get_freedom_position(j), bounds[bi]);
                // Учет первых краевых "по-хорошему"
                // В диагональ пишем 1
                slae.di[freedom] = 1.0;
                // В правую часть пишем значение краевого
                slae.rp[freedom] = val;
                // А вот тут все веселье
                // Нам надо занулить строку, а у нас симметричная матрица
                // Поэтому будем бегать по матрице, занулять стоки
                // А то, что было в столбцах - выкидывать в правую часть
                size_t i_s = slae.ig[freedom], i_e = slae.ig[freedom + 1];
                for(size_t i = i_s; i < i_e; i++)
                {
                    slae.rp[slae.jg[i]] -= slae.gg[i] * val;
                    slae.gg[i] = 0.0;
                }
                for(size_t p = freedom + 1; p < degree_of_freedom_num; p++)
                {
                    size_t i_s = slae.ig[p], i_e = slae.ig[p + 1];
                    for(size_t i = i_s; i < i_e; i++)
                    {
                        if(slae.jg[i] == freedom)
                        {
                            slae.rp[p] -= slae.gg[i] * val;
                            slae.gg[i] = 0.0;
                        }
                    }
                }
            }
        }
    }
```

```
}

vector<edge>::const_iterator FEM::get_edge_by_nodes(const point & begin, const point & end) const
{
    vector<edge>::const_iterator it;
    it = find(edges_freedom.begin(), edges_freedom.end(), edge(&begin, &end));
    if(it == edges_freedom.end())
        it = find(edges_freedom.begin(), edges_freedom.end(), edge(&end, &begin));
    return it;
}

vector<edge>::const_iterator FEM::get_edge_by_fe(const quadrilateral & fe, size_t edge_num) const
{
    if(edge_num == 0) return get_edge_by_nodes(* fe.nodes[0], * fe.nodes[1]);
    if(edge_num == 1) return get_edge_by_nodes(* fe.nodes[0], * fe.nodes[2]);
    if(edge_num == 2) return get_edge_by_nodes(* fe.nodes[1], * fe.nodes[3]);
    if(edge_num == 3) return get_edge_by_nodes(* fe.nodes[2], * fe.nodes[3]);
    assert(edge_num < 4);
    return edges_freedom.end();
}

const point * FEM::get_nodes_by_edge(const edge & e, size_t node_num) const
{
    assert(node_num < 2);
    return e.nodes[node_num];
}

const quadrilateral * FEM::get_fe_by_edge(const edge & e, size_t fe_num) const
{
    assert(fe_num < 2);
    return e.fes[fe_num];
}
```

Файл geometry.cpp

```
point quadrilateral::to_local(const point & p) const
{
    // Кирпич, страница 305
    double w = beta6 * (p.x - nodes[0]->x) - beta5 * (p.y - nodes[0]->y);
    double ksi = 0.0;
    double eta = 0.0;
    double eps = 1e-10;
    if(fabs(alpha1) < eps && fabs(alpha2) < eps) {
        ksi = (beta3 * (p.x - nodes[0]->x) - beta1 * (p.y - nodes[0]->y))/(beta2 * beta3 - beta1 * beta4);
        eta = (beta2 * (p.y - nodes[0]->y) - beta4 * (p.x - nodes[0]->x))/(beta2 * beta3 - beta1 * beta4);
    }
    else {
        if(fabs(alpha1) < eps) {
            ksi = (alpha2 * (p.x - nodes[0]->x) + beta1 * w)/(alpha2 * beta2 - beta5 * w);
            eta = -w/alpha2;
        }
        else {
            if(fabs(alpha2) < eps) {
                ksi = w / alpha1;
                eta = (alpha1 * (p.y - nodes[0]->y) - beta4 * w)/(alpha1 * beta3 + beta6 * w);
            }
            else {
                double a = beta5 * alpha2;
                double b = alpha2 * beta2 + alpha1 * beta1 + beta5 * w;
                double c = alpha1 * (nodes[0]->x - p.x) + beta2 * w;
                double D = b * b - 4.0 * a * c;
                double eta1 = (-b - sqrt(D)) / (2.0 * a);
                double eta2 = (-b + sqrt(D)) / (2.0 * a);
                double ksi1 = alpha2 / alpha1 * eta1 + w / alpha1;
                double ksi2 = alpha2 / alpha1 * eta2 + w / alpha1;
                if(eta1 + eps >= 0.0 && eta1 - eps <= 1.0 && ksi1 + eps >= 0.0 && ksi1 - eps <= 1.0) {
                    ksi = ksi1;
                    eta = eta1;
                }
                else {
                    if(eta2 + eps >= 0.0 && eta2 - eps <= 1.0 && ksi2 + eps >= 0.0 && ksi2 - eps <= 1.0) {
                        ksi = ksi2;
                        eta = eta2;
                    }
                    else
                    {
                        cerr << "Error: Target point is outside of element" << endl;
                        assert(eta2 + eps >= 0.0 && eta2 - eps <= 1.0 && ksi2 + eps >= 0.0 && ksi2 - eps <= 1.0);
                    }
                }
            }
        }
    }
    return point(ksi, eta);
}

point quadrilateral::to_global(const point & p) const
{
    // Кирпич, страница 298, формулы 5.109-5.110
    double x = (1 - p.x) * (1 - p.y) * nodes[0]->x + p.x * (1 - p.y) * nodes[1]->x + (1 - p.x) * p.y * nodes[2]->x + p.x
* p.y * nodes[3]->x;
    double y = (1 - p.x) * (1 - p.y) * nodes[0]->y + p.x * (1 - p.y) * nodes[1]->y + (1 - p.x) * p.y * nodes[2]->y + p.x
* p.y * nodes[3]->y;
    return point(x, y);
}

double quadrilateral::det_J_local(const point & p) const
{
    // Кирпич, страница 300, формула после 5.116
    double jacobian = alpha0 + alpha1 * p.x + alpha2 * p.y;
    return jacobian;
}

void quadrilateral::init()
{
    // Кирпич, страница 301, формула 5.117
```

```
    alpha0 = (nodes[1]->x - nodes[0]->x) * (nodes[2]->y - nodes[0]->y) - (nodes[1]->y - nodes[0]->y) * (nodes[2]->x -
nodes[0]->x);
    alpha1 = (nodes[1]->x - nodes[0]->x) * (nodes[3]->y - nodes[2]->y) - (nodes[1]->y - nodes[0]->y) * (nodes[3]->x -
nodes[2]->x);
    alpha2 = (nodes[2]->y - nodes[0]->y) * (nodes[3]->x - nodes[1]->x) - (nodes[2]->x - nodes[0]->x) * (nodes[3]->y -
nodes[1]->y);
    // Кирпич, страница 302, формула 5.118
    beta1 = nodes[2]->x - nodes[0]->x;
    beta2 = nodes[1]->x - nodes[0]->x;
    beta3 = nodes[2]->y - nodes[0]->y;
    beta4 = nodes[1]->y - nodes[0]->y;
    beta5 = nodes[0]->x - nodes[1]->x - nodes[2]->x + nodes[3]->x;
    beta6 = nodes[0]->y - nodes[1]->y - nodes[2]->y + nodes[3]->y;

    // Площадь через площадь двух треугольников
    double S1 = 0.5 * fabs((nodes[0]->x - nodes[2]->x) * (nodes[1]->y - nodes[2]->y) - (nodes[1]->x - nodes[2]->x) *
(nodes[0]->y - nodes[2]->y));
    double S2 = 0.5 * fabs((nodes[1]->x - nodes[3]->x) * (nodes[2]->y - nodes[3]->y) - (nodes[2]->x - nodes[3]->x) *
(nodes[1]->y - nodes[3]->y));
    S = S1 + S2;

    // Точки Гаусса в локальной с.к.
    double gauss_points_local[2][9] =
    {
        {0.0, 0.0, 0.0, sqrt(3.0 / 5.0), -sqrt(3.0 / 5.0), sqrt(3.0 / 5.0), sqrt(3.0 / 5.0), -sqrt(3.0 / 5.0),
-sqrt(3.0 / 5.0)},
        {0.0, sqrt(3.0 / 5.0), -sqrt(3.0 / 5.0), 0.0, 0.0, sqrt(3.0 / 5.0), -sqrt(3.0 / 5.0), sqrt(3.0 / 5.0),
-sqrt(3.0 / 5.0)}
    };

    // Веса Гаусса
    gauss_weights[0] = 64.0 / 81.0;
    gauss_weights[1] = gauss_weights[2] = gauss_weights[3] = gauss_weights[4] = 40.0 / 81.0;
    gauss_weights[5] = gauss_weights[6] = gauss_weights[7] = gauss_weights[8] = 25.0 / 81.0;

    // Перевод с Гауссова мастер-элемента на базисный
    for(int i = 0; i < 9; i++)
    {
        // Базисный мастер-элемент у нас [0,1]x[0,1], Гауссов - [-1,1]x[-1,1]
        gauss_points[i] = point((gauss_points_local[0][i] + 1.0) / 2.0, (gauss_points_local[1][i] + 1.0) / 2.0);
    }
}

// Получние номеров одномерных БФ из номера двумерной
void quadrilateral::two2one(size_t two, size_t & one1, size_t & one2) const
{
    one1 = two % 3;
    one2 = two / 3;
}

// Одномерные лагранжевы базисные функции
double quadrilateral::bfunc_1d(size_t func_n, double ksi) const
{
    switch(func_n)
    {
    case 0:
        return 2.0 * (ksi - 0.5) * (ksi - 1.0);
    case 1:
        return - 4.0 * ksi * (ksi - 1.0);
    case 2:
        return 2.0 * ksi * (ksi - 0.5);
    };
    assert(func_n < 3);
    return 0.0;
}

// Производные одномерных лагранжевых базисных функций
double quadrilateral::dbfunc_1d(size_t func_n, double ksi) const
{
    switch(func_n)
    {
    case 0:
        return 4.0 * (ksi - 0.75);
    case 1:
        return 4.0 - 8.0 * ksi;
    case 2:
        return 4.0 * (ksi - 0.25);
    };
    assert(func_n < 3);
    return 0.0;
}

// Двумерные лагранжевы базисные функции
double quadrilateral::bfunc_2d_local(size_t num, const point & p) const
{
    size_t nx = 0, ny = 0;
    two2one(num, nx, ny);
    return bfunc_1d(nx, p.x) * bfunc_1d(ny, p.y);
}

double quadrilateral::bfunc_2d(size_t num, const point & p) const
{
    return bfunc_2d_local(num, to_local(p));
}

// Градиент двумерных лагранжевых базисных функций
point quadrilateral::grad_bfunc_2d_local(size_t num, const point & p) const
{
    size_t nx = 0, ny = 0;
    two2one(num, nx, ny);
    return point(dbfunc_1d(nx, p.x) * bfunc_1d(ny, p.y),
                 bfunc_1d(nx, p.x) * dbfunc_1d(ny, p.y));
}

point quadrilateral::grad_bfunc_2d(size_t num, const point & p) const
{
    return grad_bfunc_2d_local(num, to_local(p));
}
```

```
// Получение элемента матрицы жесткости
double quadrilateral::get_local_G(size_t i, size_t j) const
{
    double sign_alpha0 = 0.0;
    if(alpha0 > 0.0) sign_alpha0 = 1.0;
    if(alpha0 < 0.0) sign_alpha0 = -1.0;
    // Кирпич, страницы 302-303, формула 5.119
    double result = 0.0;
    for(int g = 0; g < 9; g++)
    {
        point grad_i = grad_bfunc_2d_local(i, gauss_points[g]);
        point grad_j = grad_bfunc_2d_local(j, gauss_points[g]);
        double J = det_J_local(gauss_points[g]);
        double ksi = gauss_points[g].x;
        double eta = gauss_points[g].y;
        double dfii_dksi = grad_i.x;
        double dfii_deta = grad_i.y;
        double dfij_dksi = grad_j.x;
        double dfij_deta = grad_j.y;
        double func = (dfii_dksi * (beta6 * ksi + beta3) - dfii_deta * (beta6 * eta + beta4)) *
                      (dfij_dksi * (beta6 * ksi + beta3) - dfij_deta * (beta6 * eta + beta4)) +
                      (dfii_deta * (beta5 * eta + beta2) - dfii_dksi * (beta5 * ksi + beta1)) *
                      (dfij_deta * (beta5 * eta + beta2) - dfij_dksi * (beta5 * ksi + beta1));
        func /= J;
        result += gauss_weights[g] * func;
    }
    return sign_alpha0 * result / 4.0;
}

// Получение элемента матрицы массы
double quadrilateral::get_local_M(size_t i, size_t j) const
{
    double result = 0.0;
    for(int g = 0; g < 9; g++)
    {
        result += gauss_weights[g] * det_J_local(gauss_points[g]) *
                  bfunc_2d_local(i, gauss_points[g]) * bfunc_2d_local(j, gauss_points[g]);
    }
    return result / 4.0;
}

// Получение элемента правой части
double quadrilateral::get_local_rp(size_t i) const
{
    double result = 0.0;
    for(int g = 0; g < 9; g++)
    {
        result += gauss_weights[g] * det_J_local(gauss_points[g]) *
                  bfunc_2d_local(i, gauss_points[g]) * func_rp(to_global(gauss_points[g]), this);
    }
    return result / 4.0;
}

// Определение, внутри четырехугольника точка или нет
bool quadrilateral::inside(const point & p) const
{
    double eps = 1e-10;
    double S1 = 0.5 * fabs((nodes[0]->x - p.x) * (nodes[1]->y - p.y) - (nodes[1]->x - p.x) * (nodes[0]->y - p.y));
    double S2 = 0.5 * fabs((nodes[0]->x - p.x) * (nodes[2]->y - p.y) - (nodes[2]->x - p.x) * (nodes[0]->y - p.y));
    double S3 = 0.5 * fabs((nodes[1]->x - p.x) * (nodes[3]->y - p.y) - (nodes[3]->x - p.x) * (nodes[1]->y - p.y));
    double S4 = 0.5 * fabs((nodes[3]->x - p.x) * (nodes[2]->y - p.y) - (nodes[2]->x - p.x) * (nodes[3]->y - p.y));
    if(fabs(S - S1 - S2 - S3 - S4) < eps)
        return true;
    return false;
}

bool edge::operator < (const edge & t) const
{
    if(nodes[0]->num < t.nodes[0]->num) return true;
    if(nodes[0]->num > t.nodes[0]->num) return false;
    if(nodes[1]->num < t.nodes[1]->num) return true;
    return false;
}

bool edge::operator == (const edge &t) const
{
    if(nodes[0]->num == t.nodes[0]->num &&
       nodes[1]->num == t.nodes[1]->num) return true;
    return false;
}

point edge::get_freedom_position(size_t num) const
{
    if(num == 0) return point(*(nodes[0]));
    if(num == 2) return point(*(nodes[1]));
    return point((nodes[0]->x + nodes[1]->x) / 2.0,
            (nodes[0]->y + nodes[1]->y) / 2.0);
}

double edge::bfunc_1d(size_t func_n, double x) const
{
    switch(func_n)
    {
    case 0:
        return 2.0 * (x - 0.5) * (x - 1.0);
    case 1:
        return - 4.0 * x * (x - 1.0);
    case 2:
        return 2.0 * x * (x - 0.5);
    };
    assert(func_n < 3);
    return 0.0;
}

double edge::get_matrix_M(size_t i, size_t j)
{
```

```
    static double gauss_points[3] = {-sqrt(3.0 / 5.0) , 0.0, sqrt(3.0 / 5.0)};
    static double gauss_weights[3] = {5.0 / 9.0, 8.0 / 9.0, 5.0 / 9.0};
    double dx = nodes[1]->x - nodes[0]->x;
    double dy = nodes[1]->y - nodes[0]->y;
    double h = sqrt(dx * dx + dy * dy);
    double jacobian = h / 2.0;

    double result = 0.0;
    for(int g = 0; g < 3; g++)
    {
        double gauss_point_basis = (gauss_points[g] + 1.0) / 2.0;
        result += gauss_weights[g] *
                bfunc_1d(i, gauss_point_basis) * bfunc_1d(j, gauss_point_basis);
    }
    return result * jacobian;
}
```

Файл glwidget.h

```
// Класс треугольник
class triangle
{
public:
    point nodes[3];
    double color[3];
    double solution[3];
    triangle() {}
    triangle(const point & node1, const point & node2, const point & node3)
    {
        nodes[0] = node1;
        nodes[1] = node2;
        nodes[2] = node3;
    }
};

// Класс OpenGL виджет
class glwidget : public QGLWidget
{
    Q_OBJECT
protected:
    // Событие таймера
    void timerEvent(QTimerEvent *);
public:
    // Инициализация сцены
    void initializeGL();
    // Действие при изменении размеров виджета
    void resizeGL(int nWidth, int nHeight);
    // Отрисовка сцены
    void paintGL();
    // Конструктор
    glwidget(QWidget* parent = 0);

    // Флаг отрисовки конечноэлементной сетки
    bool draw_mesh;
    // Флаг отрисовки изолиний
    bool draw_isolines;
    // Флаг закраски цветом
    bool draw_color;
    // Пересчет значений изолиний
    void set_isolines_num(size_t isolines_num);
    // Изменение количества сегментов, на которые разбивается каждый КЭ
    void set_div_num(size_t num);
private:
    // Мьютекс чтобы не случилось странностей при изменении данных извне
    QMutex mtx;
    // Класс МКЭ
    FEM fem;

    // Минимальные и максимальные значения геометрии + размер
    double min_x, max_x, size_x;
    double min_y, max_y, size_y;
    // Количество шагов координатной сетки
    size_t num_ticks_x, num_ticks_y;
    // Подгонка осей под реальность и вычисление шагов координатной сетки
    void adjustAxis(double & min, double & max, size_t & numTicks);

    // Минимальное и максимальное значения решения
    double min_u, max_u;
    // Значения изолиний
    set<double> isolines;
    // Вспомогательные шаги по цвету для закраски
    double step_u_big, step_u_small;

    // Треугольники, которые будем рисовать
    vector<triangle> triangles;
};
```

Файл glwidget.cpp

```
// Конструктор
glwidget::glwidget(QWidget* parent) : QGLWidget(parent)
{
    // Пусть будем рисовать все
    draw_mesh = true;
    draw_isolines = true;
    draw_color = true;

    // Решаем МКЭ задачу
    fem.input();
    fem.make_portrait();
    fem.assembling_global();
    fem.applying_bounds();
    fem.slae.solve(1e-16);
```

```
    // Ищем минимальные и максимальные значения координат
    max_x = min_x = fem.nodes[0].x;
    max_y = min_y = fem.nodes[0].y;
    for(size_t i = 1; i < fem.nodes_num; i++)
    {
        if(fem.nodes[i].x > max_x) max_x = fem.nodes[i].x;
        if(fem.nodes[i].y > max_y) max_y = fem.nodes[i].y;
        if(fem.nodes[i].x < min_x) min_x = fem.nodes[i].x;
        if(fem.nodes[i].y < min_y) min_y = fem.nodes[i].y;
    }
    size_x = max_x - min_x;
    size_y = max_y - min_y;
    min_x -= size_x * 0.01;
    max_x += size_x * 0.01;
    min_y -= size_y * 0.01;
    max_y += size_y * 0.01;

    // Поправляем значения мин / макс чтобы влазило в сетку
    adjustAxis(min_x, max_x, num_ticks_x);
    adjustAxis(min_y, max_y, num_ticks_y);
    size_x = max_x - min_x;
    size_y = max_y - min_y;

    // Ищем максимальные и минимальные значения, чтобы нормировать цвет
    max_u = min_u = fem.slae.x[0];
    for(size_t i = 1; i < fem.slae.n; i++)
    {
        if(fem.slae.x[i] > max_u) max_u = fem.slae.x[i];
        if(fem.slae.x[i] < min_u) min_u = fem.slae.x[i];
    }

    // Рассчитываем изолинии
    const size_t isolines_num = 25;
    set_isolines_num(isolines_num);

    // Шаги для разбиения по цветовым областям
    step_u_big = (max_u - min_u) / 4.0;
    step_u_small = step_u_big / 256.0;

    // Число разбиений КЭ на сегменты
    set_div_num(1);

    // Запускаем таймер отрисовки
    startTimer(500);
}

// Пересчет значений изолиний
void glwidget::set_isolines_num(size_t isolines_num)
{
    mtx.lock();
    isolines.clear();
    double isolines_step = (max_u - min_u) / (double)(isolines_num + 1);
    for(size_t i = 0; i < isolines_num; i++)
        isolines.insert(min_u + isolines_step * (double)(i + 1));
    mtx.unlock();
}

// Изменение количества сегментов, на которые разбивается каждый КЭ
void glwidget::set_div_num(size_t num)
{
    mtx.lock();

    vector<triangle> tmp1;
    vector<triangle> tmp2;
    // Посчитаем в локальных координатах
    tmp1.push_back(triangle(point(0.0, 0.0), point(1.0, 0.0), point(0.0, 1.0)));
    tmp1.push_back(triangle(point(1.0, 0.0), point(1.0, 1.0), point(0.0, 1.0)));
    for(size_t i = 0; i < num; i++)
    {
        for(size_t j = 0; j < tmp1.size(); j++)
        {
            point middles[3] =
            {
                point((tmp1[j].nodes[0].x + tmp1[j].nodes[1].x) / 2.0, (tmp1[j].nodes[0].y + tmp1[j].nodes[1].y) / 2.0),
                point((tmp1[j].nodes[0].x + tmp1[j].nodes[2].x) / 2.0, (tmp1[j].nodes[0].y + tmp1[j].nodes[2].y) / 2.0),
                point((tmp1[j].nodes[1].x + tmp1[j].nodes[2].x) / 2.0, (tmp1[j].nodes[1].y + tmp1[j].nodes[2].y) / 2.0)
            };
            tmp2.push_back(triangle(tmp1[j].nodes[0], middles[0], middles[1]));
            tmp2.push_back(triangle(middles[0], tmp1[j].nodes[1], middles[2]));
            tmp2.push_back(triangle(middles[1], middles[2], tmp1[j].nodes[2]));
            tmp2.push_back(triangle(middles[0], middles[2], middles[1]));
        }
        tmp2.swap(tmp1);
        tmp2.clear();
    }

    // Заполняем вектор из треугольников переводя координаты в глобальные и считая цвет
    triangles.clear();
    for(size_t i = 0; i < fem.qls_num; i++)
    {
        for(size_t j = 0; j < tmp1.size(); j++)
        {
            triangle tmp_tr;
            // Переводим координаты в глобальные
            for(size_t k = 0; k < 3; k++)
                tmp_tr.nodes[k] = fem.qls[i].to_global(tmp1[j].nodes[k]);
            // Занесем значение решения в узлах
            for(size_t k = 0; k < 3; k++)
                tmp_tr.solution[k] = fem.get_solution(tmp_tr.nodes[k], fem.qls + i);

            // Барицентр треугольника
            double cx = 0.0, cy = 0.0;
            for(size_t k = 0; k < 3; k++)
            {
                cx += tmp_tr.nodes[k].x;
                cy += tmp_tr.nodes[k].y;
            }
```

```
            point center(cx / 3.0, cy / 3.0);
            // Решение в барицентре
            double center_u = fem.get_solution(center, fem.qls + i);

            // Ищем цвет решения по алгоритму заливки радугой (Rainbow colormap)
            unsigned short r_color = 0, g_color = 0, b_color = 0;
            if(center_u > min_u + step_u_big * 3.0)
            {
                r_color = 255;
                g_color = 255 - (unsigned short)((center_u - (min_u + step_u_big * 3.0)) / step_u_small);
                b_color = 0;
            }
            else if(center_u > min_u + step_u_big * 2.0)
            {
                r_color = (unsigned short)((center_u - (min_u + step_u_big * 2.0)) / step_u_small);
                g_color = 255;
                b_color = 0;
            }
            else if(center_u > min_u + step_u_big)
            {
                unsigned short tmp = (unsigned short)((center_u - (min_u + step_u_big)) / step_u_small);
                r_color = 0;
                g_color = tmp;
                b_color = 255 - tmp;
            }
            else
            {
                unsigned short tmp = 76 - (unsigned short)((center_u - min_u) / (step_u_small * (255.0 / 76.0)));
                r_color = tmp;
                g_color = 0;
                b_color = 255 - tmp;
            }

            // Приглушаем кислотные цвета
            r_color = r_color * 3 / 4 + 64;
            g_color = g_color * 3 / 4 + 64;
            b_color = b_color * 3 / 4 + 64;

            // Задаем посчитанный цвет
            tmp_tr.color[0] = (double)r_color / 255.0;
            tmp_tr.color[1] = (double)g_color / 255.0;
            tmp_tr.color[2] = (double)b_color / 255.0;

            // И заносим в вектор
            triangles.push_back(tmp_tr);
        }
    }

    mtx.unlock();
}

// Подгонка осей под реальность и вычисление шагов координатной сетки
void glwidget::adjustAxis(double & min, double & max, size_t & numTicks)
{
    static const double axis_epsilon = 1.0 / 10000.0;
    if(max - min < axis_epsilon)
    {
        min -= 2.0 * axis_epsilon;
        max += 2.0 * axis_epsilon;
    }

    static const size_t MinTicks = 10;
    double grossStep = (max - min) / MinTicks;
    double step = pow(10, floor(log10(grossStep)));

    if (5 * step < grossStep)
        step *= 5;
    else if (2 * step < grossStep)
        step *= 2;

    numTicks = (size_t)(ceil(max / step) - floor(min / step));
    min = floor(min / step) * step;
    max = ceil(max / step) * step;
}

// Отрисовка сцены
void glwidget::paintGL()
{
    if(!mtx.tryLock()) return;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Координатные оси
    glColor3d(0.0, 0.0, 0.0);
    glLineWidth(2.0f);
    glBegin(GL_LINES);
    glVertex3d(0.0, -0.005, -0.1);
    glVertex3d(0.0, 1.005, -0.1);
    glVertex3d(-0.005, 0.0, -0.1);
    glVertex3d(1.005, 0.0, -0.1);
    glEnd();

    // Подписи осей
    QFont fnt_mono("Courier", 8);
    QFont fnt_serif("Times", 10);
    fnt_mono.setLetterSpacing(QFont::PercentageSpacing, 75.0);
    fnt_serif.setBold(true);
    renderText(0.99f, -0.04f, 0.0f, trUtf8("x"), fnt_serif);
    renderText(-0.05f, 0.99f, 0.0f, trUtf8("y"), fnt_serif);

    // Координатная сетка
    glColor3d(0.85, 0.85, 0.85);
    glLineWidth(1.0f);
    for(size_t i = 0; i <= num_ticks_x; i++)
```

```
    {
        double x = (double)i / (double)num_ticks_x;
        glBegin(GL_LINES);
        glVertex3d(x, -0.01, -0.2);
        glVertex3d(x, 1.0, -0.2);
        glEnd();
    }
    for(size_t i = 0; i <= num_ticks_y; i++)
    {
        double y = (double)i / (double)num_ticks_y;
        glBegin(GL_LINES);
        glVertex3d(-0.01, y, -0.2);
        glVertex3d(1.0, y, -0.2);
        glEnd();
    }

    // Отрисовка шкалы
    glColor3d(0.0, 0.0, 0.0);
    glLineWidth(2.0f);
    for(size_t i = 0; i < num_ticks_x; i++)
    {
        double x = (double)i / (double)num_ticks_x;
        double x_real = (double)(floor((x * size_x + min_x) * 10000.0 + 0.5)) / 10000.0;
        QString st = QString::number(x_real);
        renderText((float)x - 0.01f, -0.04f, 0.001f, st, fnt_mono);
    }
    for(size_t i = 0; i < num_ticks_y; i++)
    {
        double y = (double)i / (double)num_ticks_y;
        double y_real = (double)(floor((y * size_y + min_y) * 10000.0 + 0.5)) / 10000.0;
        QString st = QString::number(y_real);
        renderText(-0.05f, (float)y - 0.01f, 0.001f, st, fnt_mono);
    }

    // Отрисовка конечноэлементной сетки
    if(draw_mesh)
    {
        glColor3d(0.1, 0.1, 0.1);
        glLineWidth(1.0f);
        glBegin(GL_LINES);
        for(vector<edge>::const_iterator i = fem.edges_freedom.begin(); i != fem.edges_freedom.end(); i++)
        {
            glVertex3d((i->nodes[0]->x - min_x) / size_x, (i->nodes[0]->y - min_y) / size_y, 0.1);
            glVertex3d((i->nodes[1]->x - min_x) / size_x, (i->nodes[1]->y - min_y) / size_y, 0.1);
        }
        glEnd();
    }

    // Отрисовка всех треугольников
    for(size_t i = 0; i < triangles.size(); i++)
    {
        // Раскрашивать будем если запрошено сие, иначе зальем белым цветом
        if(draw_color)
            // Задаем посчитанный цвет
            glColor3dv(triangles[i].color);
        else
            // Задаем белый цвет
            glColor3d(1.0, 1.0, 1.0);
        // Рисуем
        glBegin(GL_TRIANGLES);
        for(size_t k = 0; k < 3; k++)
            glVertex3d((triangles[i].nodes[k].x - min_x) / size_x, (triangles[i].nodes[k].y - min_y) / size_y, 0.01);
        glEnd();
    }

    for(size_t i = 0; i < triangles.size(); i++)
    {
        // Изолинии рисуем только если оно нам надо
        if(draw_isolines)
        {
            // Теперь рисуем изолинии
            // Будем искать наименьшее значение, большее или равное решению
            // Если значения на разных концах ребра будут разными, значит изолиния проходит через это ребро
            set<double>::const_iterator segment_isol[3];
            for(size_t k = 0; k < 3; k++)
                segment_isol[k] = isolines.lower_bound(triangles[i].solution[k]);

            // А теперь нарисуем, согласно вышеприведенному условию
            glColor3d(0.0, 0.0, 0.0);
            glLineWidth(1.0f);
            glBegin(GL_LINE_STRIP);
            if(segment_isol[0] != segment_isol[1])
                glVertex3d(((triangles[i].nodes[1].x + triangles[i].nodes[0].x) * 0.5 - min_x) / size_x,
((triangles[i].nodes[1].y + triangles[i].nodes[0].y) * 0.5 - min_y) / size_y, 0.02);
            if(segment_isol[0] != segment_isol[2])
                glVertex3d(((triangles[i].nodes[2].x + triangles[i].nodes[0].x) * 0.5 - min_x) / size_x,
((triangles[i].nodes[2].y + triangles[i].nodes[0].y) * 0.5 - min_y) / size_y, 0.02);
            if(segment_isol[1] != segment_isol[2])
                glVertex3d(((triangles[i].nodes[2].x + triangles[i].nodes[1].x) * 0.5 - min_x) / size_x,
((triangles[i].nodes[2].y + triangles[i].nodes[1].y) * 0.5 - min_y) / size_y, 0.02);
            glEnd();
        }
    }

    mtx.unlock();
}
```