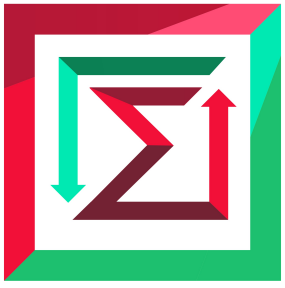
МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Новосибирский государственный технический университет»

НГТУ НЭТИ

Кафедра прикладной математики

Курсовой проект
по дисциплине «Численные методы»



ФПМИ

ГРУППА	ПМ-92
ВАРИАНТ	21
СТУДЕНТ	ГЛУШКО ВЛАДИСЛАВ
ПРЕПОДАВАТЕЛЬ	СОЛОВЕЙЧИК ЮРИЙ ГРИГОРЬЕВИЧ

Новосибирск

1 Условие задачи

Формулировка задачи

МКЭ для двумерной краевой задачи для эллиптического уравнения в декартовой системе координат. Базисные функции линейные на треугольниках. Краевые условия всех ти-пов. Коэффициент разложить по линейным базисным функциям. Матрицу СЛАУ генери-ровать в разреженном строчном формате. Для решения СЛАУ использовать МСГ или ЛОС с неполной факторизацией.

Постановка задачи

Эллиптическая краевая задача для функции *u* определяется дифференциальным уравнением:

$$-div(\lambda\ gradu) + \gamma u = f$$

заданным в некоторой области Ω с границей $S = S_1 \cup S_2 \cup S_3$ и краевыми условиями:

$$u|_{S_1} = u_g$$

$$\lambda \frac{\partial u}{\partial n}\bigg|_{S_2} = \theta$$

$$\lambda \frac{\partial u}{\partial n}\bigg|_{S_3} + \beta(u|_{S_3} - u_\beta) = 0$$

В декартовой системе координат x,y это уравнение может быть записано в виде

$$-\frac{\partial}{\partial x}\left(\lambda \frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(\lambda \frac{\partial u}{\partial y}\right) + \gamma u = f$$

Исходное уравнение можно переписать в виде: $Lu = f$, где $Lu = -div(\lambda gradu) + \gamma u$. Тогда чтобы решить исзодную задачу следует левую и праую части уравнения домножить на функцию v из пространства пробных функций и проинтегрировать по Ω. Фактически это соответсвует скалярному умножению Lu и f на v в пространтсве $L_2(\Omega)$:

$$(Lu, v) = (f, v)$$

$$(Lu - f, v) = 0$$

Это уравнение Галеркина в слабой форме.
В пространстве $L_2(\Omega)$ скалярное произведение вычисляется по формуле:

$$(u, v) = \int\limits_\Omega uvd\Omega$$

Перепишем уравнение Галеркина в явном виде:

$$-\int\limits_\Omega div(\lambda\ gradu)v\ d\Omega + \int\limits_\Omega \gamma uv\ d\Omega - \int\limits_\Omega fv\ d\Omega = 0$$

Воспользуемся формулой Грина:

$$\int\limits_{\Omega}(\lambda\ gradu\ gradv)\ d\Omega = -\int\limits_{\Omega} div(\lambda\ gradu)v\ d\Omega + \int\limits_{S}\lambda\frac{\partial u}{\partial n}v\ dS$$

Сделав соответствующую подстановку, получим уравнение вида:

$$\int\limits_{\Omega}(\lambda\ gradu\ gradv)d\Omega - \int\limits_{S}\lambda\frac{\partial u}{\partial n}vdS + \int\limits_{\Omega}\gamma uv\ d\Omega \int\limits_{\Omega} fv\ d\Omega = 0$$

Учитывая краевые условия $S = S_1 \cup S_2 \cup S_3$, получим:

$$\int\limits_{\Omega}(\lambda\ gradu\ gradv)d\Omega - \int\limits_{S_1}\lambda\frac{\partial u}{\partial n}vdS - \int\limits_{S_2}\theta vdS - \int\limits_{S_3}\beta(u|_{\beta} - u)vdS + \int\limits_{\Omega}\gamma uv\ d\Omega - \int\limits_{\Omega} fv\ d\Omega = 0$$

Так как $v|_{S_1} = 0$, то

$$\int\limits_{S_1}\lambda\frac{\partial u}{\partial n}v\ dS = 0$$

Уравнение примет вид:

$$\int\limits_{\Omega}(\lambda gradu\ gradv)d\Omega + \int\limits_{S_3}\beta uvdS + \int\limits_{\Omega}\gamma uvd\Omega = \int\limits_{S_2}\theta vdS + \int\limits_{S_3}\beta u|_{\beta}vdS + \int\limits_{\Omega} fvd\Omega = 0$$

Будем искать решение в виде:

$$u = \sum_{i=1}^{n} q_i\psi_i$$

где ψ_i - базисные функции. Функция v может быть представлена в таком же виде. Подставив, получим СЛАУ для компонент q_i:

$$\sum_{i=1}^{n} q_i \left(\int\limits_{\Omega}(\lambda\ grad\psi_i\ grad\psi_j)d\Omega + \int\limits_{S_3}\beta\psi_i\psi_j\ dS + \int\limits_{\Omega}\gamma\psi_i\psi_j\ d\Omega\right) = \\ = \int\limits_{S_2}\theta\psi_j\ dS + \int\limits_{S_3}\beta u_{\beta}\psi_j\ dS + \int\limits_{\Omega} f\psi_j\ d\Omega$$

Поскольку исходная задача рассматривается в декартовой системе координат, то $gradu = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right)$ и, соответсвенно: $gradu\ gradv = \frac{\partial u}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial u}{\partial y}$. Отсюда получаем уравнение в виде:

$$\sum_{j=1}^{n} q_j \int\limits_{\Omega}\lambda\left(\frac{\partial\psi_j}{\partial x}\frac{\partial\psi_i}{\partial x} + \frac{\partial\psi_j}{\partial y}\frac{\partial\psi_i}{\partial y}\right)dxdy + \sum_{j=1}^{n} q_j \int\limits_{\Omega}\gamma\psi_j\psi_i\ dxdy + \sum_{j=1}^{n} q_j \int\limits_{\Omega}\beta\psi_j\psi_i\ dxdy = \\ = \int\limits_{\Omega} f\psi_i\ dxdy + \int\limits_{S_2}\theta\psi_i\ dxdy + \int\limits_{S_3}\beta u_{\beta}\psi_i\ dxdy$$

Конечноэлементная дискретизация

Так как для решения задачи используются линейные базисные функции, то на каждом конечном элементе Ω_k - треугольнике эти функции будут совпадать с функциями $L_1(x,y), L_2(x,y), L_3(x,y)$, такими, что $L_1(x,y)$ равна единице в вершине (x_1,y_1) и нулю во всех остальных вершинах, $L_2(x,y)$ равна единице в вершине (x_2,y_2) и нулю во всех остальных вершинах, $L_3(x,y)$ равна единице в вершине (x_3,y_3) и нулю во всех остальных вершинах. Любая линейная на Ω_k функция представима в виде линейной комбинации этих базисных линейных функций, коэффициентами будут значения функции в каждой из вершин треугольника Ω_k. Таким образом, на каждом конечном элементе нам понадобятся три узла – вершины треугольника.

$$\psi_1 = L_1(x,y)$$
$$\psi_2 = L_2(x,y)$$
$$\psi_3 = L_3(x,y)$$

Учитывая построение *L-функций*, получаем следующие соотношения:

$$\begin{cases} L_1 + L_2 + L_3 = 1 \\ L_1x_1 + L_2x_2 + L_3x_3 = x \\ L_1y_1 + L_2y_2 + L_3y_3 = y \end{cases}$$

Т.е. имеем систему:

$$\begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix} \cdot \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} = \begin{pmatrix} 1 \\ x \\ y \end{pmatrix}$$

Отсюда находим коэффициенты линейных функций $L_1(x,y), L_2(x,y), L_3(x,y)$

$$L_i = a_0^i + a_1^i x + a_2^i y, i = \overline{1,3}$$

$$\begin{pmatrix} \alpha_0^1 & \alpha_1^1 & \alpha_2^1 \\ \alpha_0^2 & \alpha_1^2 & \alpha_2^2 \\ \alpha_0^3 & \alpha_1^3 & \alpha_2^3 \end{pmatrix} = D^{-1} = \begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}^{-1}$$

$$D^{-1} = \frac{1}{|detD|} \begin{pmatrix} x_2y_3 - x_3y_2 & y_2 - y_3 & x_3 - x_2 \\ x_3y_1 - x_1y_3 & y_3 - y_1 & x_1 - x_3 \\ x_1y_2 - x_2y_1 & y_1 - y_2 & x_2 - x_1 \end{pmatrix}$$

Переход к локальным матрицам

Чтобы получить выражения для локальных матриц жёсткости G и массы M каждого конечного элемента Ω_K, перейдём к решению локальной задачи на каждом конечном элементе. Полученное уравнение для области Ω представим в виде суммы интегралов по областям Ω_k без учёта краевых условий. Тогда на каждом конечном элементе будем решать локальную задачу построения матриц жёсткости, массы и вектора правой части.

$$\int\limits_{\Omega_k} \lambda \left(\frac{\partial \psi_j}{\partial x} \frac{\partial \psi_i}{\partial x} + \frac{\partial \psi_j}{\partial y} \frac{\partial \psi_i}{\partial y} \right) dxdy + \int\limits_{\Omega_k} \gamma \psi_j \psi_i dsdy = \int\limits_{\Omega_k} f \psi_i dxdy$$

Локальная матрица будет представлять собой сумму матриц жёсткости и массы и будет иметь размерность 3×3 (по числу узлов на конечном элементе)

Построение матрицы массы

$$M_{ij} = \int\limits_{\Omega_m} \gamma Y_i Y_j d\Omega_m = \Big| \gamma = Y_1\gamma_1 + Y_2\gamma_2 + Y_3\gamma_3 \Big| = \int\limits_{\Omega_m} \left(Y_1\gamma_1 + Y_2\gamma_2 + Y_3\gamma_3 \right) Y_i Y_j d\Omega_m =$$

$$= \gamma_1 \int\limits_{\Omega_m} Y_1 Y_i Y_j d\Omega_m + \gamma_2 \int\limits_{\Omega_m} Y_2 Y_i Y_j d\Omega_m + \gamma_3 \int\limits_{\Omega_m} Y_3 Y_i Y_j d\Omega_m =$$

$$= \gamma_1 \int\limits_{\Omega_m} L_1 L_i L_j d\Omega_m + \gamma_2 \int\limits_{\Omega_m} L_2 L_i L_j d\Omega_m + \gamma_3 \int\limits_{\Omega_m} L_3 L_i L_j d\Omega_m$$

Построение матрицы жёсткости

Рассмотрим первый член в выражении для k-го конечного эдемента:

$$\int\limits_{\Omega_k} \lambda \left(\frac{\partial \psi_j}{\partial x} \frac{\partial \psi_i}{\partial x} + \frac{\partial \psi_j}{\partial y} \frac{\partial \psi_i}{\partial y} dxdy \right)$$

$$B_{i,j} = (\alpha_1^i \alpha_1^j + \alpha_2^i \alpha_2^j) \frac{\left| detD \right|}{2} \qquad i,j = \overline{0,2}$$

Построение вектора правой части

Рассмотрим правую часть выражения для k-го конечного элемента:

$$\int\limits_{\Omega_k} f \psi_i dxdy$$

представим f в виде $f_1 L_1 + f_2 L_2 + f_3 L_3$, где f_i - значения в вершинах треугольника. Получим:

$$\int\limits_{\Omega_k} f_q L_q L_i dxdy = f_q \int\limits_{\Omega_k} L_q L_i d\Omega_k$$

Таким образом:

$$G_i = \sum_{q=1}^{3} f_q \int\limits_{\Omega_k} L_q L_i d\Omega_k \qquad i = \overline{0,2}$$

Сборка глобальной матрицы и глобального вектора

При формировании глобальной матрицы из локальных, полученных суммированием соответствующих матриц массы и жесткости, учитываем соответствие локальной и глобальной нумераций каждого конечного элемента. Глобальная нумерация каждого конечного элемента однозначно определяет позиции вклада его локальной м атрицы в глобальную. Поэтому, зная глобальные номера соответствующих узлов конечного элемента, определяем и то, какие элементы глобальной матрицы изменятся при учете текущего конечного элемента. Аналогичным образом определяется вклад локального вектора правой части в глобальный. При учете текущего локального вектора изменятся те элементы глобального вектора правой части, номера которых совпадают с глобальными номерами узлов, присутствующих в этом конечном элементе.

Учёт первых краевых условий

Для учета первых краевых условий, в глобальной матрице и глобальном векторе находим соответствующую глобальному номеру краевого узла строку и зануляем всё кроме диагонального элемента, которому присваиваем 1, а вместо элемента с таким номером в векторе правой части - значение краевого условия, заданное в исходной задаче.

Учёт вторых и третьих краевых условий

Рассмотрим краевые условия второго и третьего рода:

$$\lambda \frac{\partial u}{\partial n}\bigg|_{S_2} = \theta$$

$$\lambda \frac{\partial u}{\partial n}\bigg|_{S_3} + \beta(u|_{S_3} - u_\beta) = 0$$

Отсюда получаем, что для учёта краевых условий необходимо вычислить интегралы:

$$\int\limits_{S_2} \theta\psi_j dxdy, \qquad \int\limits_{S_3} \beta u_\beta \psi_j dxdy, \qquad \int\limits_{S_3} \beta\psi_i\psi_j dxdy$$

Краевые условия второго и третьего рода задаются на рёбрах, т.е. определяются двумя узлами, лежащими на ребру. Будем считать, что параметр β на S_3 постоянен, тогда параметр $_\beta$ будем раскладывать по двум базисным функциям, определённым на этом ребре:

$$u_\beta = u_{\beta 1}\phi_1 + u_{\beta 2}\phi_2$$

где $\phi_i,\ i = \overline{0,1}$ - локально занумерованные линейные базисные функции, которые имеют также свои глобальные номера во всей расчетной области, а $u_{\beta i}$ - значение функции u_β в узлах ребра.

Аналогично поступаем и при учете вторых краевых условий, раскладывая по базису ребра функцию $\theta = \theta_0\phi_0 + theta_1\phi_1$.
Тогда приведенные выше интегралы примут вид:

$$I_1 = \int\limits_{S_2} (\theta_0\phi_0 + \theta_1\phi_1)\phi_i dxdy$$

$$I_2 = \beta \int\limits_{S_3} (u_{\beta 1}\phi_0 + u_{\beta 2}\phi_1)\phi_i dxdy$$

$$I_3 = \beta \int\limits_{S_3} \phi_i\phi_j dxdy$$

Фактически, решая задачу учета краевых условий второго и третьего рода, мы переходим к решению одномерной задачи на ребре для того, чтобы занести соответствующие результаты в глобальную матрицу и вектор.

Базисными функциями ребра являются две ненулевые на данном ребре базисные функции из $\phi_i,\ i = \overline{0,1}$ конечного элемента.

Для учёта вклада вторых и третьих краевых условий рассчитываются 2 матрицы 2×2.

Игтегралы I_1, I_2, I_3 будем вычислять по формуле:

$$\int (L_i)^{v_i}(L_j)^{v_j} dS = \frac{v_i! v_j!}{(v_i + v_j + 1)!} mes\Gamma,\ i \neq j$$

где $mes\Gamma$ длина ребра. При этом независимо от того, что на каждом из ребер присутствуют свои функции, интегралы, посчитанные по приведенным выше формулам, будут равны.

$$I_1 = \begin{pmatrix} \int\limits_{S_2} L_1L_1 dxdy & \int\limits_{S_2} L_1L_2 dxdy \\ \int\limits_{S_2} L_2L_1 dxdy & \int\limits_{S_2} L_2L_2 dxdy \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} = \frac{1}{6} mes S_2 \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix}$$

Этот вектор поправок в правую часть позволяет учесть не только вторые краевые условия, но и часть βu_β из третьих. Осталось рассмотреть матрицу поправок в левую часть:

$$I_3 = \beta \int\limits_{S_3} \phi_i\phi_j dxdy$$

Очевидно, что получится та же матрица, только не умноженная на вектор констант.

$$I_3 = \frac{1}{6} mes S_3 \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Добавляя эту матрицу в левую часть, на места соответствующие номерам узлов, получаем учет третьих краевых условий.

2 Текст программы

Весь проект: https://github.com/ISTECTION/FEM

main.cpp

```cpp
#include "argparse/argparse.hpp"
#include "timer/cxxtimer.hpp"
#include "LOS/LOS.hpp"
#include "FEM.hpp"

#include <iostream>
#include <optional>
#include <fstream>

int main(int argc, char* argv[]) {
    using namespace    ::Log;
    using ::std::chrono::milliseconds;

    argparse::ArgumentParser _program("FEM", "1.0.0");
    _program.add_argument("-i", "--input")
        .help("path to input files" )
        .required();
    _program.add_argument("-o", "--output")
        .help("path to output files");

    try {
        _program.parse_args(argc, argv);

        std::optional _opt            = _program.present("-o");
        std::filesystem::path _input  = _program.get<std::string>("-i");
        std::filesystem::path _output =
            _opt.has_value() ?
                _program.get<std::string>("-o") :
                _input / "sparse";

        Function::setFunction(_input.string());

        cxxtimer::Timer _timer(true);           /// start timer
        FEM _FEM(_input);                       /// start FEM
        _FEM.writeFile(_output, 1E-14, 10000);  /// overwriting files
        LOS<double> _LOS(_output);              /// here is reading
        _LOS.solve(Cond::DIAGONAL, true);
        _timer.stop();                          /// stop timer

        #if DEBUG != 0
        _FEM.printAll();                        /// print input FEM data
        _FEM.printSparse();                     /// print sparse format
        _FEM.printAnalitics();                  /// print analicals solve
        #endif
```

```
45
46          std::cout << "SOLUTION: ";
47          print(_LOS.getX(), 14);                 /// print solution vector
48          std::cout << "Milliseconds: " << _timer.count<milliseconds>();
49      } catch(const std::runtime_error& err) {
50          Logger::append(getLog("argc != 2 (FEM --input ./input)"));
51          std::cerr << err.what();
52          std::cerr << _program;
53          std::exit(1);                           /// program error
54      }
55      return 0;
56  }
```

Union.hpp

```
1   #ifndef _UNION_HPP_
2   #define _UNION_HPP_
3   #include <vector>
4   #include <array>
5
6   _UNION_BEGIN
7
8   struct XY {
9       double x;
10      double y;
11  };
12
13  struct Material {
14      double betta;
15      double gamma;
16  };
17
18  struct Element {
19      size_t area;
20      std::array<size_t, 3> nodeIdx;
21  };
22
23  struct Boundary {
24      size_t cond;
25      size_t type;
26      size_t area;
27      std::array<size_t, 2> nodeIdx;
28  };
29
30  struct Param {
31      size_t nodes;
32      size_t elems;
33      size_t areas;
```

```
    size_t conds;
};

_UNION_END
#endif /// _UNION_HPP_
```

FEM.hpp

```
#ifndef _FEM_HPP_
#define _FEM_HPP_
#include "utils/lightweight.hpp"
#include "utils/overload.hpp"
#include "utils/friendly.hpp"
#include "Function.hpp"
#include "Logger.hpp"
#include "Union.hpp"

#include <algorithm>
#include <cmath>
#include <set>

class FEM
{
private:
    Union::Param _size;

    std::vector<Union::XY>       nodes;
    std::vector<Union::Element>  elems;
    std::vector<Union::Boundary> boundarys;
    std::vector<Union::Material> materials;

    std::vector<double> gb;
    std::vector<double> gg;
    std::vector<double> di;
    std::vector<size_t> ig;
    std::vector<size_t> jg;

public:
    FEM(std::filesystem::path _path) {
        assert(readFile(_path));
        portrait(true);
        global();
        boundaryCondition();
    }
    ~FEM() { }

    void printAnalitics() {
        std::vector<size_t> ax;
```

```
41
42          for (const auto& _elem : elems) {
43              for (size_t i = 0; i < 3; i++)
44                  ax[_elem.nodeIdx[i]] =
45                      Function::analitics(
46                          nodes[_elem.nodeIdx[i]],
47                          _elem.area
48                      );
49          }
50          std::cout << "ANALITIC: "; print(ax, 14);
51      }
52      size_t    getNodes() { return _size.nodes; }
53
54  private:
55      void global();
56      void resize();
57
58      template<size_t N, typename _Struct>
59      void loc_A_to_global(
60          const std::array<std::array<double, N>, N>&,
61          const _Struct&
62      );
63
64      template<size_t N, typename _Struct>
65      void loc_b_to_global(const std::array<double, N>&, const _Struct& );
66
67      array::xxx localA(const std::array<Union::XY, 3>&, size_t) const;
68      array::x   buildF(const std::array<Union::XY, 3>&, size_t) const;
69
70      array::xxx G(const std::array<Union::XY, 3>&, size_t) const;
71      array::xxx M(const std::array<Union::XY, 3>&, size_t) const;
72
73      bool readFile(const std::filesystem::path&  );
74      void portrait(const bool isWriteList = false);
75
76      void boundaryCondition();
77      void first (const Union::Boundary& bound);
78      void second(const Union::Boundary& bound);
79      void third (const Union::Boundary& bound);
80  };
81
82  void FEM::global() {
83
84      std::array<Union::XY, 3> coords;
85
86      for (size_t i = 0; i < _size.elems; i++) {
87          for (size_t j = 0; j < 3; j++) {
88              size_t point = elems[i].nodeIdx[j];
89              coords[j].x = nodes[point].x;
```

```
            coords[j].y = nodes[point].y;
        }
        array::x   local_b = buildF(coords, elems[i].area);
        array::xxx local_A = localA(coords, elems[i].area);

        loc_A_to_global<3>(local_A, elems[i]);
        loc_b_to_global<3>(local_b, elems[i]);
    }
}

void FEM::boundaryCondition() {
    using namespace ::Log;

    for (size_t _count = 0; _count < _size.conds; _count++) {
        switch (boundarys[_count].cond)
        {
            case FIRST_BOUNDARY_COND:
                first(boundarys[_count]);
                break;
            case SECOND_BOUNDARY_COND:
                second(boundarys[_count]);
                break;
            case THIRD_BOUNDARY_COND:
                third(boundarys[_count]);
                break;
            default:
                Logger::append(getLog("There is no such condition"));
        }
    }
}

void FEM::first(const Union::Boundary& bound) {
    di[bound.nodeIdx[0]] = { 1 };
    di[bound.nodeIdx[1]] = { 1 };

    for (size_t i = 0; i < 2; i++)
        gb[bound.nodeIdx[i]] =
            Function::firstBound({
                nodes[bound.nodeIdx[i]].x,
                nodes[bound.nodeIdx[i]].y
            }, bound.type);

    for (size_t k = 0; k < 2; k++) {
        size_t node = bound.nodeIdx[k];
        for (size_t i = ig[node]; i < ig[node + 1]; i++) {
            if(di[jg[i]] != 1)
                gb[jg[i]] -= gg[i] * gb[node];
            gg[i] = 0;
        }
```

```
139
140         for(size_t i = node + 1; i < _size.nodes; i++) {
141             size_t lbeg = ig[i];
142             size_t lend = ig[i + 1];
143             for(size_t p = lbeg; p < lend; p++) {
144                 if(jg[p] == node) {
145                     if(di[i] != 1)
146                         gb[i] -= gg[p] * gb[node];
147                     gg[p] = 0;
148                 }
149             }
150         }
151     }
152 }
153
154 void FEM::second(const Union::Boundary& bound) {
155
156     std::array<Union::XY, 2>
157         coord_borders = {
158             nodes[bound.nodeIdx[0]],
159             nodes[bound.nodeIdx[1]]
160         };
161
162     double _koef = edgeLength(coord_borders) / 6;
163
164     std::array<double, 2> corr_b;
165     for (size_t i = 0; i < 2; i++)
166         corr_b[i] = _koef * (
167             2 * Function::secondBound({
168                     nodes[bound.nodeIdx[i]].x,
169                     nodes[bound.nodeIdx[i]].y
170                 }, bound.type) +
171                 Function::secondBound({
172                     nodes[bound.nodeIdx[1 - i]].x,
173                     nodes[bound.nodeIdx[1 - i]].y
174                 }, bound.type)
175             );
176
177     loc_b_to_global<2>(corr_b, bound);
178 }
179
180 void FEM::third(const Union::Boundary& bound) {
181
182     std::array<Union::XY, 2> coord_borders = {
183         nodes[bound.nodeIdx[0]],
184         nodes[bound.nodeIdx[1]]
185     };
186
187     double _koef =
```

```
188             materials[bound.area].betta *
189             edgeLength(coord_borders) / 6;
190
191         std::array<std::array<double, 2>, 2> corr_a;
192
193         std::array<double, 2> corr_b;
194         for (size_t i = 0; i < 2; i++) {
195
196             corr_b[i] = _koef * (
197                 2 * Function::thirdBound({
198                         nodes[bound.nodeIdx[i]].x,
199                         nodes[bound.nodeIdx[i]].y
200                     }, bound.type) +
201                     Function::thirdBound({
202                         nodes[bound.nodeIdx[1 - i]].x,
203                         nodes[bound.nodeIdx[1 - i]].y
204                     }, bound.type)
205                 );
206
207             for (size_t j = 0; j < 2; j++) {
208                 corr_a[i][j] =
209                     (i == j) ? (2 * _koef) :
210                                (    _koef) ;
211             }
212         }
213         loc_b_to_global<2>(corr_b, bound);
214         loc_A_to_global<2>(corr_a, bound);
215     }
216
217     template<size_t N, typename _Struct>
218     void FEM::loc_A_to_global(
219             const std::array<std::array<double, N>, N>& locA,
220             const _Struct& elem) {
221
222         using            ::std::vector;
223         using iterator = ::std::vector<size_t>::iterator;
224
225         for (size_t i = 0; i < N; i++) {
226             di[elem.nodeIdx[i]] += locA[i][i];
227
228             for (size_t j = 0; j < i; j++) {
229                 size_t a = elem.nodeIdx[i];
230                 size_t b = elem.nodeIdx[j];
231                 if (a < b) std::swap(a, b);
232
233                 if (ig[a + 1] > ig[a]) {
234                     iterator _beg = jg.begin() + ig[a];
235                     iterator _end = jg.begin() + ig[a + 1] - ig[0];
236
```

```
                auto _itr = std::lower_bound(_beg, _end, b);
                auto _idx = _itr - jg.begin();
                gg[_idx] += locA[i][j];
            }
        }
    }
}

template<size_t N, typename _Struct>
void FEM::loc_b_to_global(
        const std::array<double, N>& loc_b,
        const _Struct& elem) {

    for (size_t i = 0; i < N; i++)
        gb[elem.nodeIdx[i]] += loc_b[i];
}

array::x FEM::buildF(const std::array<Union::XY, 3>& elem, size_t area)
 ↪  const {
    std::array<double, 3> function {
        Function::f(elem[0], area),
        Function::f(elem[1], area),
        Function::f(elem[2], area)
    };

    double det_D = fabs(determinant(elem)) / 24;
    return {
        det_D * (2 * function[0] + function[1] + function[2]),
        det_D * (2 * function[1] + function[0] + function[2]),
        det_D * (2 * function[2] + function[0] + function[1]),
    };
}

array::xxx FEM::localA(const std::array<Union::XY, 3>& elem, size_t area)
 ↪  const {
    std::array<std::array<double, 3>, 3> G = FEM::G(elem, area);
    std::array<std::array<double, 3>, 3> M = FEM::M(elem, area);
    std::array<std::array<double, 3>, 3> A = G + M;
    return A;
}

array::xxx FEM::G(const std::array<Union::XY, 3>& elem, size_t area) const
 ↪  {
    double det   = fabs(determinant(elem));
    double _koef = Function::lambda(area) / (2 * det);

    std::array<std::array<double, 3>, 3> G;
    std::array<std::array<double, 2>, 3> a {

```

```
            elem[1].y - elem[2].y,
            elem[2].x - elem[1].x,

            elem[2].y - elem[0].y,
            elem[0].x - elem[2].x,

            elem[0].y - elem[1].y,
            elem[1].x - elem[0].x
    };

    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        G[i][j] = _koef * (
            a[i][0] * a[j][0] +
            a[i][1] * a[j][1]
        );

    return G;
}

array::xxx FEM::M(const std::array<Union::XY, 3>& elem, size_t area) const
 ↪  {
    double det = fabs(determinant(elem));
    double gammaKoef = materials[area].gamma * det / 24;
    std::array<std::array<double, 3>, 3> M;
    for (size_t i = 0; i < 3; i++)
    for (size_t j = 0; j < 3; j++) {
        M[i][j] =
            (i == j) ? (2 * gammaKoef) :
                       (    gammaKoef) ;
    }
    return  M;
}

void FEM::portrait(const bool isWriteList) {

    const size_t N {    _size.nodes     };
    std::vector<std::set<size_t>> list(N);

    for (size_t el = 0; el < _size.elems; el++)
    for (size_t point = 0; point < 3; point++) {
        for (size_t i = point + 1; i < 3; i++) {
            size_t idx1 = { elems[el].nodeIdx[point] };
            size_t idx2 = { elems[el].nodeIdx[  i  ] };
            idx1 > idx2 ?
                list[idx1].insert(idx2) :
                list[idx2].insert(idx1) ;
        }
    }
```

```
331
332     for (size_t i = 2; i < ig.size(); i++)
333         ig[i] = ig[i - 1] + list[i - 1].size();
334
335     jg.resize(ig[N] - ig[0]);
336     gg.resize(ig[N] - ig[0]);
337
338     for (size_t index = 0, i = 1; i < list.size(); i++)
339     for (size_t value : list[i])
340         jg[index++] = value;
341
342     #if DEBUG != 0
343     if (isWriteList) {
344         std::cout << "list: " << '\n';
345         for (size_t i = 0; i < list.size(); i++) {
346             std::cout << i << ':' << ' ';
347             for (size_t j : list[i])
348                 std::cout << j << ' ';
349             std::cout << std::endl;
350         }
351     }
352     #endif
353 }
354
355 bool FEM::readFile(const std::filesystem::path& path) {
356     using namespace ::Log;
357     bool isError { true };
358
359     std::ifstream fin(path / "params.txt");
360     isError &= is_open(fin, getLog("Error - params.txt"));
361     fin >> _size.nodes
362         >> _size.elems
363         >> _size.areas
364         >> _size.conds;
365     fin.close();
366
367     resize();
368     std::fill_n(ig.begin(), 2, 0);
369
370     fin.open(path / "nodes.txt");
371     isError &= is_open(fin, getLog("Error - nodes.txt"));
372     for (size_t i = 0; i < _size.nodes; i++)
373         fin >> nodes[i].x >> nodes[i].y;
374     fin.close();
375
376     fin.open(path / "elems.txt");
377     isError &= is_open(fin, getLog("Error - elems.txt"));
378     for (size_t i = 0; i < _size.elems; i++) {
379         fin >> elems[i].nodeIdx[0]
```

```
            >> elems[i].nodeIdx[1]
            >> elems[i].nodeIdx[2];
    }
    fin.close();

    fin.open(path / "areas.txt");
    isError &= is_open(fin, getLog("Error - areas.txt"));
    for (size_t i = 0; i < _size.areas; i++)
        fin >> materials[i].gamma
            >> materials[i].betta;

    for (size_t i = 0; i < _size.elems; i++)
        fin >> elems[i].area;
    fin.close();

    fin.open(path / "bords.txt");
    isError &= is_open(fin, getLog("Error - bords.txt"));
    for (size_t i = 0; i < _size.conds; i++)
        fin >> boundarys[i].area
            >> boundarys[i].nodeIdx[0]
            >> boundarys[i].nodeIdx[1]
            >> boundarys[i].cond
            >> boundarys[i].type;
    fin.close();

    std::sort(
        boundarys.begin(),
        boundarys.end(),
        [](Union::Boundary& _left, Union::Boundary& _right){
            return _left.cond > _right.cond;
        }
    );

    return isError;
}

void FEM::resize() {
    nodes.    resize(  _size.nodes  );
    elems.    resize(  _size.elems  );
    boundarys.resize(  _size.conds  );
    materials.resize(  _size.areas  );

    gb.resize(  _size.nodes  );
    di.resize(  _size.nodes  );
    ig.resize(_size.nodes + 1);
}
#endif /// _FEM_HPP_
```

Data.hpp

```cpp
#ifndef _DATA_HPP_
#define _DATA_HPP_
#include "../utils/friendly.hpp"
#include "../Logger.hpp"

#include <filesystem>
#include <cassert>
#include <fstream>
#include <sstream>
#include <vector>
#include <cmath>

_SYMMETRIC_BEG

struct Param {
    size_t n;
    double epsilon;
    size_t max_iter;
};

enum class Cond {
    NONE,
    DIAGONAL,
    HOLLESKY
};

template <class T>
class Data
{
protected:
    Param param;
    std::vector<size_t> ig;
    std::vector<size_t> jg;
    std::vector<T> di;
    std::vector<T> gg;
    std::vector<T> b;
    std::vector<T> x;

    std::vector<T> di_l;
    std::vector<T> gg_l;
    std::vector<T> y;
    size_t iter{ 0 };

public:
    Data(std::filesystem::path _path) { assert(loadData(_path)); }
    Data(Friendly* _friend, size_t _n, T _eps, size_t _max_iter) {
        param.n       = _n;
        param.epsilon  = _eps;
```

```
49          param.max_iter = _max_iter;
50
51          ig = _friend->ig;
52          jg = _friend->jg;
53          gg = _friend->gg;
54          di = _friend->di;
55          b  = _friend->gb;
56
57          x.resize(_n);
58          delete _friend;
59      }
60      ~Data() { }
61
62      std::vector<T>& getX() const { return x;    }
63      size_t  getIteration() const { return iter; }
64
65      void convertToLU();
66      std::vector<T> normal (std::vector<T> b);
67      std::vector<T> reverse(std::vector<T> y);
68      std::vector<T> mult(const std::vector<T>& _vec);
69
70  private:
71      bool loadData(std::filesystem::path _path);
72  };
73
74  template <class T>
75  void Data<T>::convertToLU() {
76      di_l = di;
77      gg_l = gg;
78
79      for (size_t i = 0; i < param.n; i++) {
80          T sum_diag = 0;
81          for (size_t j = ig[i]; j < ig[i + 1] ; j++) {
82              T sum = 0;
83              size_t jk = ig[jg[j]];
84              size_t ik = ig[i];
85              while ((ik < j) && (jk < ig[jg[j] + 1]))
86              {
87                  size_t l = jg[jk] - jg[ik];
88                  if (l == 0) {
89                      sum += gg_l[jk] * gg_l[ik];
90                      ik++; jk++;
91                  }
92                  jk += (l < 0);
93                  ik += (l > 0);
94              }
95              gg_l[j]  -= sum;
96              gg_l[j]  /= di_l[jg[j]];
97              sum_diag += gg_l[j] * gg_l[j];
```

```
        }
        di_l[i] -= sum_diag;
        di_l[i] = sqrt(fabs(di_l[i]));
    }
}

template <class T>
std::vector<T> Data<T>::normal(std::vector<T> b) {
    for (size_t i = 0; i < param.n; i++) {
        for (size_t j = ig[i]; j < ig[i + 1]; j++)
            b[i] -= gg_l[j] * b[jg[j]];

        b[i] = b[i] / di_l[i];
    }
    return b;
}

template <class T>
std::vector<T> Data<T>::reverse(std::vector<T> x) {
    for (int j = param.n - 1; j >= 0; j--) {
        x[j] = x[j] / di_l[j];

        for (size_t i = ig[j]; i < ig[j + 1]; i++)
            x[jg[i]] -= gg_l[i] * x[j];
    }
    return x;
}

template <class T>
std::vector<T> Data<T>::mult(const std::vector<T>& _vec) {
    std::vector<T> pr(_vec.size());

    int jj = 0;
    for (size_t i = 0; i < _vec.size(); i++) {
        pr[i] = di[i] * _vec[i];

        for (size_t j = ig[i]; j < ig[i + 1]; j++, jj++) {
            pr[i] += gg[jj] * _vec[jg[jj]];
            pr[jg[jj]] += gg[jj] * _vec[i];
        }
    }
    return pr;
}

template <typename T>
bool read(std::filesystem::path _path, std::vector<T>& _vec) {
    using namespace ::Log;
    std::ifstream fin(_path);
    if (not
```

```
147         is_open(fin, "Error - " + _path.filename().string()))
148         return false;
149     for (size_t i = 0; i < _vec.size(); i++)
150             fin >> _vec[i];
151     fin.close(); return true;
152 }
153 
154 template <class T>
155 bool Data<T>::loadData(std::filesystem::path _path) {
156     using namespace ::Log;
157     std::ifstream fin(_path / "kuslau.txt");
158     if (not is_open(fin, "Error - kuslau.txt"))
159         return false;
160     fin >> param.n
161         >> param.epsilon
162         >> param.max_iter;
163     fin.close();
164 
165     bool is_cor {  true  };
166     ig.resize(param.n + 1);
167 
168     is_cor &= read(_path / "ig.txt", ig);
169 
170     gg.resize(ig.back());
171     jg.resize(ig.back());
172     di.resize( param.n );
173 
174     b.resize (param.n);
175     x.resize (param.n);
176 
177     is_cor &= read(_path / "gg.txt", gg);
178     is_cor &= read(_path / "di.txt", di);
179     is_cor &= read(_path / "jg.txt", jg);
180     is_cor &= read(_path / "gb.txt", b);
181     return is_cor;
182 }
183 _SYMMETRIC_END
184 #endif /// _DATA_HPP_
```

LOS.hpp

```
1 #ifndef _LOS_HPP_
2 #define _LOS_HPP_
3 #include "Data.hpp"
4 #include "LOS_Function.hpp"
5 
6 using namespace Symmetric;
7 
```

```
#define LOGGER if (isLog)   \
    printLog(this->iter, eps);

template <class T>
class LOS : public Data<T>
{
public:
    LOS(std::filesystem::path _path) : Data<T>(_path) { }
    LOS(Friendly* _friend, size_t _n,  T _eps, size_t _max_iter)
        : Data<T>(_friend, _n, _eps, _max_iter) { }

    ~LOS() { }

    void solve(Cond _cond, bool isLog = true);
private:
    void none    (bool);
    void diagonal(bool);
    void hollesky(bool);
};

template <class T>
void LOS<T>::solve(Cond _cond, bool isLog) {
    using namespace ::Log;
    std::streamsize p = std::cout.precision();
    std::cout.precision(2);
    std::cout.setf(std::ios::uppercase);
    switch (_cond) {
        case Cond::NONE:     none(  isLog  ); break;
        case Cond::DIAGONAL: diagonal(isLog); break;
        case Cond::HOLLESKY: hollesky(isLog); break;
        default:
            Logger::append(getLog("this conditional non exist"));
            std::exit(1);
    }
    std::cout.unsetf(std::ios::scientific);
    std::cout.unsetf(std::ios::uppercase);
    std::cout.precision(p);
}

template <class T>
void LOS<T>::none(bool isLog) {
    std::vector<T> r (this->param.n),
                   z (this->param.n),
                   p (this->param.n),
                   Ar(this->param.n);

    r = this->b - this->mult(this->x);
    z = r;
    p = this->mult(z);
```

```
57
58      T alpha, betta, eps;
59      do {
60          betta   = scalar(p, p);
61          alpha   = scalar(p, r) / betta;
62          this->x = this->x + alpha * z;
63          r       = r - alpha * p;
64          Ar      = this->mult(r);
65          betta   = scalar(p, Ar) / betta;
66          z       = r  - betta * z;
67          p       = Ar - betta * p;
68          eps     = scalar(r, r);
69
70          this->iter++;
71          LOGGER
72
73      } while(
74          this->iter < this->param.max_iter
75              && eps > this->param.epsilon);
76  }
77
78  template <class T>
79  void LOS<T>::diagonal(bool isLog) {
80      std::vector<T> r (this->param.n),
81                     z (this->param.n),
82                     p (this->param.n),
83                     Ar(this->param.n);
84
85      std::vector<T> L(this->param.n, 1);
86      for (size_t i = 0; i < L.size(); i++)
87          L[i] /= sqrt(this->di[i]);
88
89      r = L * (this->b - this->mult(this->x));
90      z = L * r;
91      p = L * this->mult(z);
92
93      T alpha, betta, eps;
94      do {
95          betta   = scalar(p, p);
96          alpha   = scalar(p, r) / betta;
97          this->x = this->x + alpha * z;
98          r       = r - alpha * p;
99          Ar      = L * this->mult(L * r);
100         betta   = scalar(p, Ar) / betta;
101         z       = L * r - betta * z;
102         p       = Ar - betta * p;
103         eps     = scalar(r, r);
104
105         this->iter++;
```

```
        LOGGER

    } while(
        this->iter < this->param.max_iter
            && eps > this->param.epsilon);
}

template <class T>
void LOS<T>::hollesky(bool isLog) {
    std::vector<T> r  (this->param.n),
                   z  (this->param.n),
                   p  (this->param.n),
                   Ar (this->param.n),
                   LAU(this->param.n);

    this->convertToLU();
    r = this->normal(this->b - this->mult(this->x));
    z = this->reverse(r);
    p = this->normal(this->mult(z));

    T alpha, betta, eps;
    do {
        betta   = scalar(p, p);
        alpha   = scalar(p, r) / betta;
        this->x = this->x + alpha * z;
        r       = r - alpha * p;
        LAU     = this->normal(this->mult(this->reverse(r)));
        betta   = scalar(p, LAU) / betta;
        z       = this->reverse(r) - betta * z;
        p       = LAU - betta * p;
        eps     = scalar(r, r);

        this->iter++;
        LOGGER
    }
    while(
        this->iter < this->param.max_iter
            && eps > this->param.epsilon);
}
#undef LOGGER
#endif /// _LOS_HPP_
```

LOS_Function.hpp

```
#ifndef _LOS_FUNCTION_HPP_
#define _LOS_FUNCTION_HPP_

#include <numeric>
```

```cpp
#include <vector>
#include <cmath>

template <typename T>
inline void printLog(size_t _iter, T _eps) {
    std::cout << "Iteration = "   << std::fixed      << _iter << "\t\t" <<
                 "Discrepancy = " << std::scientific << _eps  << std::endl;
}

template <typename T>
T scalar(const std::vector<T>& _v1, const std::vector<T>& _v2) {
    T _res = 0;
    for (size_t i = 0; i < _v1.size(); i++)
        _res += _v1[i] * _v2[i];
    return _res;
}

template <typename T>
T norm (const std::vector<T>& _v) {
    return sqrt(std::accumulate(_v.begin(), _v.end(), 0.0,
        [] (double _S, const double &_El) { return _S + _El * _El; }));
}

template <typename T>
std::vector<T> operator* (std::vector<T> _v1, const std::vector<T>& _v2) {
    for (size_t i = 0; i < _v1.size(); i++)
        _v1[i] *= _v2[i];
    return _v1;
};

template <typename T>
std::vector<T> operator- (std::vector<T> _v1, const std::vector<T>& _v2) {
    for (size_t i = 0; i < _v1.size(); i++)
        _v1[i] -= _v2[i];
    return _v1;
};

template <typename T>
std::vector<T> operator+ (std::vector<T> _v1, const std::vector<T>& _v2) {
    for (size_t i = 0; i < _v1.size(); i++)
        _v1[i] += _v2[i];
    return _v1;
};

template <typename T>
std::vector<T> operator* (T _alpha, std::vector<T> _v1) {
    for (size_t i = 0; i < _v1.size(); i++)
        _v1[i] *= _alpha;
    return _v1;
```

```
};
#endif // _LOS_FUNCTION_HPP_
```

lightweight.hpp

```
#ifndef _LIGHTWEIGHT_HPP_
#define _LIGHTWEIGHT_HPP_
#include "../Union.hpp"

#include <array>
#include <cmath>

double
determinant(const std::array<Union::XY, 3>& elem) {
    return (
        (elem[1].x - elem[0].x) * (elem[2].y - elem[0].y) -
        (elem[1].y - elem[0].y) * (elem[2].x - elem[0].x)
    );
}

double
edgeLength(const std::array<Union::XY, 2>& elem) {
    return (
        sqrt (
            pow(elem[1].x - elem[0].x ,2) +
            pow(elem[1].y - elem[0].y, 2)
        )
    );
}
#endif // _LIGHTWEIGHT_HPP_
```

overload.hpp

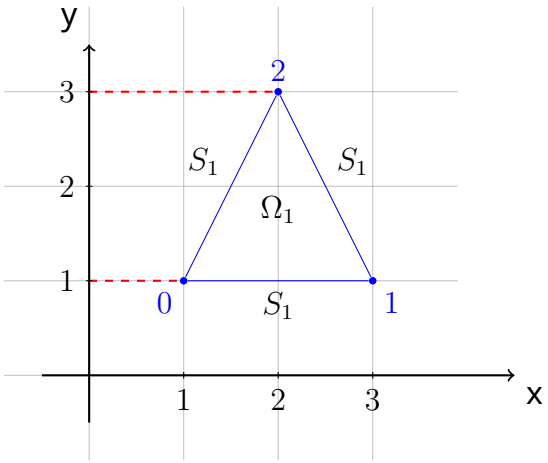
```
#ifndef _OVERLOAD_HPP_
#define _OVERLOAD_HPP_
#include <iostream>
#include <vector>
#include <array>

namespace array {
    using x   = std::array<double, 3>;
    using xxx = std::array<std::array<double, 3>, 3>;
}

array::xxx
operator+ (const array::xxx& G, const array::xxx& M) {
```

```
14      array::xxx _res;
15      for (size_t i = 0; i < G.size(); i++)
16      for (size_t j = 0; j < G.size(); j++)
17          _res[i][j] = G[i][j] + M[i][j];
18      return _res;
19  }
20  #endif // _OVERLOAD_HPP_
```

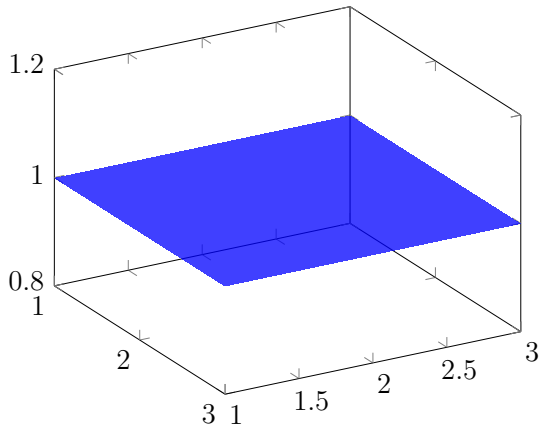
3 Тестирование

Тест №1

$$u(x,y) = 1$$
$$f(x,y) = 0$$
$$\lambda = 1$$
$$\gamma = 0$$
$$\beta = 0$$



$$I_0 = 1$$



nodes	elems	area	bords
1 1	0 1 2	0	0 0 1 1 0
3 1			0 1 2 1 0
2 3			0 2 0 1 0

x	x^*	$x^* - x$	$\|x^* - x\|$
1.000	1.000	0.00E+00	
1.000	1.000	0.00E+00	0.00E+00
1.000	1.000	0.00E+00	

Тест №2

Тест №3

Тест №4

Тест №5

4 Выводы

Выводы