МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ
НЭТИ | **Факультет прикладной математики и информатики**

Кафедра прикладной математики

Лабораторная работа № 2

по дисциплине «КТМиАД»

АЛГОРИТМЫ НУМЕРАЦИИ БАЗИСНЫХ ФУНКЦИЙ В МКЭ

Бригада 10 БАРАНОВ ЯРОСЛАВ

Группа ПММ-32 МАКАРЫЧЕВ СЕРГЕЙ

Вариант 2

Преподаватели КОШКИНА Ю.И.

Новосибирск, 2023

1. Задание

Цель: реализовать алгоритмы нумерации глобальных базисных функций в конечноэлементной сетке.

Вариант: пронумеровать грани в конечноэлементной сетке.

Подпрограммы для реализации:

1) Подпрограмма, выдающая по номеру конечного элемента номера его граней.
2) Подпрограмма, выдающая номер грани по набору её узлов.
3) Подпрограмма, выдающая по номеру грани номера соответствующих узлов и конечных элементов, которым принадлежит грань.

2. Теоретические основы и особенности реализации

Будем различать два вида граней расчётной области: граничные и негранничные.

Для учёта краевых условий второго и третьего рода необходимо составить два массива граней для каждого краевого условия. Так как тетраэдральная сетка была построена на основе кубической, которая была частично преобразована в шестигранную, то граничные грани тетраэдров – часть граней шестигранников. Поэтому построить эти два массива можно проходом по кубической сетке.

Алгоритм построения массивов граничных граней:

1) Вычисление лимитов ребра в кубической сетке.
2) Коррекция лимитов ребра для возможного обращения к крайним элементам. Также на этом этапе для совпадающих лимитов ребра проверяется есть ли узел слева (внизу или сзади, в зависимости от того какие лимиты совпали) в расчётной области (это необходимо для различия двух параллельных граней куба, так как разбиение на треугольники у них одно и тоже).
3) Далее в цикле по кубической сетке последовательно вычисляются номера узлов прямоугольника (грани куба) с последующей коррекцией с учётом пропущенных узлов (в сетке отсутствуют фиктивные узлы), который потом разбивается на треугольники согласно соответствующему паттерну разбиения куба на тетраэдры. Также вычисляется номер конечного элемента-шестигранника опять-таки с последующей коррекцией с учётом

пропущенных элементов шестигранников, которому принадлежат граничные грани (на один узел может приходится до 8 пропущенных элементов, поэтому необходимо отслеживать это отдельно). Для определения точного номера конечного элемента происходит линейный поиск грани в массиве из тетраэров, которые триангулируют шестигранник, номер которого был вычислен на предыдущем этапе, поэтому такой поиск весьма эффективен.

Далее рассмотрим построение массива граней, который включает в себя уже все грани, как граничные, так и неграничные.

Если граничная грань всегда принадлежит одному элементу, то неграничная грань принадлежит двум соприкасающимся по ней элементам. В случае если грань граничная, то номер второго конечного элемента, которому принадлежит грань, равен -1.

За основу был взят алгоритм описанный в [1] для наиболее общего случая, когда конечноэлементная сетка хранится поэлементно и каждый элемент задаётся набором своих узлов.

Исходный алгоритм:

1) В цикле по конечным элементам составить массив центров граней P для всех граней.
2) Отсортировать массив P по x, y, z (предполагается использование устойчивых методов сортировки за O(nlog(n)) в среднем).
3) Удалить дубликаты граней (O(n))
4) Создать массив ind длиной P.
5) В цикле по конечным элементам (m – номер элемента) для каждой грани найти её номер i в P (для этого снова нужно вычислить центр грани или сохранить его). То есть предполагается трёхэтапный поиск в упорядоченном массиве. Далее если ind[i] равен 0, то ind[i] = m, иначе ind[i] - номер элемента смежного по рассматриваемой грани и нужно внести информацию о номерах соседей.

Алгоритм был модернизирован так, что его главные минусы были нивелированы: затраты на сортировку центров граней и дополнительная память для хранения вещественных узлов.

Edges – выходной массив граней.

Модернизированный алгоритм:

1) В цикле по конечным элементам заполнить массив граней Edges.
2) Отсортировать массив Edges по первому номеру вершины грани, по второму и по третьему. Благодаря тому, что в данном случае ключи сортировки целые числа, возможна замена трёх сортировок одной, также за O(nlog(n)) в среднем, по одному составному 64-битному ключу, который вычисляется с помощью побитовых операций следующим образом: `u64 key = v[0] | (v[1] « 21) | (v[2] « 42)`, где v – вершины грани.
3) Удалить дубликаты граней (O(n))
4) Создать массив ind длиной P.
5) В цикле по конечным элементам (m – номер элемента) для каждой грани найти её номер i в Edges (для этого используется бинарный поиск по составному ключу за O(log(n)), создание которого описано выше). Далее если ind[i] равен 0, то ind[i] = m, иначе ind[i] - номер элемента смежного по рассматриваемой грани и нужно внести информацию о номерах соседей.

Таким образом, вместо массива P, элементом которого является узел, состоящий из 3-ёх вещественных чисел, которые представляют центр треугольника, сразу используется массив Edges, элементом которого является грань, как три номера в массиве узлов. Благодаря этому экономится 3*8*4*k = 96k байт памяти, где k – количество конечных элементов. При этом не вычисляется центр грани. Кроме того, три сортировки заменяются одной, а трёхэтапный поиск в отсортированном массиве P – бинарным поиском в массиве граней по составному ключу.

Структура данных для граней:

```
struct Edge
{
   std::array<int, SIZE_EDGE> vertexes;
   std::array<int, 2> elemNums;
   bool operator!=(const Edge& edge) const { return edge.vertexes != vertexes; }
   bool operator==(const Edge& edge) const { return !(edge != *this); }
};
```

Реализация подпрограмм:

1) Подпрограмма, выдающая по номеру конечного элемента номера его граней:

```
const std::array<int, NUMBER_EDGES_ELEMENT>& Grid::GetNumbersEdges(int numElem)
const
{
   //номера граней в массиве вершин конечного элемента
   std::array<std::array<int, SIZE_EDGE>, NUMBER_EDGES_ELEMENT> numEdges{ {{0, 1,
2}, {0, 1, 3}, {0, 2, 3}, {1, 2, 3}} };
   std::array<int, NUMBER_EDGES_ELEMENT> numbersEdges;
```

```
   for (int j = 0; j < m_elements[numElem].vertexes.size(); j++)
      numbersEdges[j] = GetNumberEdge(Edge{ {
m_elements[numElem].vertexes[numEdges[j][0]],
m_elements[numElem].vertexes[numEdges[j][1]],
m_elements[numElem].vertexes[numEdges[j][2]] }, { numElem, -1 } });

   return numbersEdges;
    }
```

2) Подпрограмма, выдающая номер грани по набору её узлов:

```
int Grid::GetNumberEdge(const Edge& edge) const
{
   //поиск с помощью составного ключа (если ключи < 2^21)
   return binarySearch(m_edges, edge, 0, m_edges.size() - 1, [](const auto& edge_1,
const auto& edge_2)->bool
      {return (edge_1.vertexes[0] | (static_cast<int64_t>(edge_1.vertexes[1]) << 21)
| (static_cast<int64_t>(edge_1.vertexes[2]) << 42)) >
      (edge_2.vertexes[0] | (static_cast<int64_t>(edge_2.vertexes[1]) << 21) |
(static_cast<int64_t>(edge_2.vertexes[2]) << 42)); });
}
```

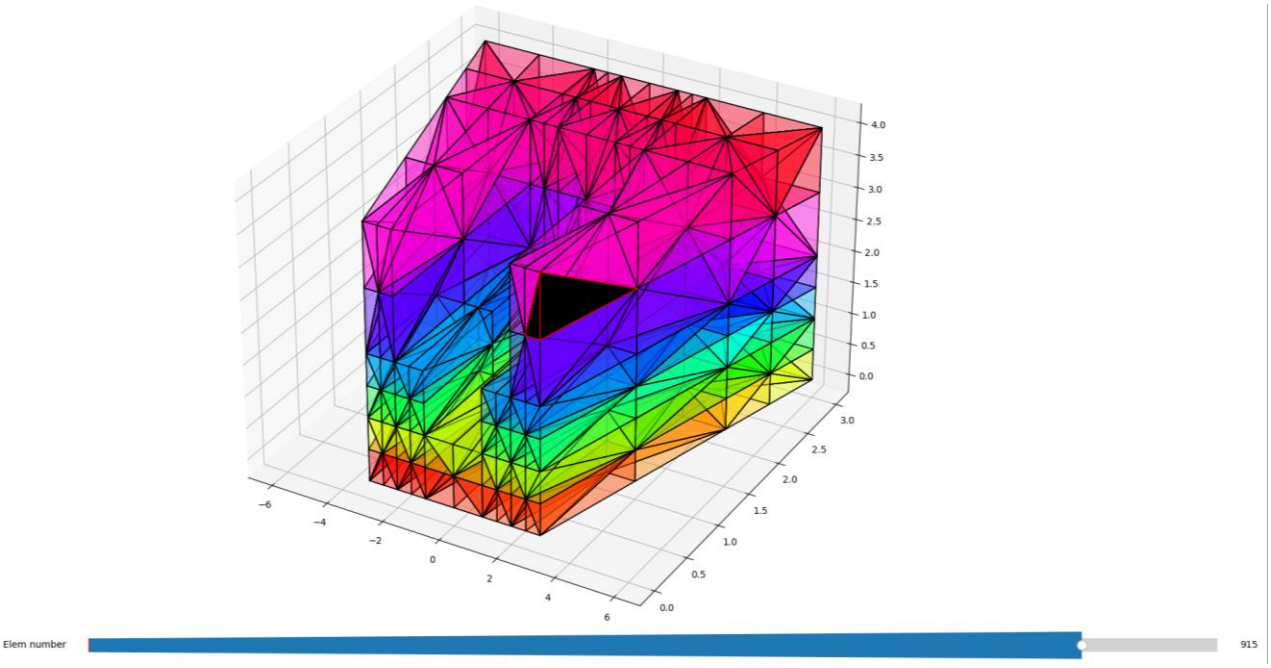
3) Подпрограмма, выдающая по номеру грани номера соответствующих узлов и конечных элементов, которым принадлежит грань:

```
const Edge& GetEdge(int i) const { return m_edges[i]; }
```

3. Тестирование

Произведём тестирование требуемых подпрограмм (последняя подпрограмма в виду её просты не тестируется, а вторая используется в первой).

Подпрограмма, выдающая по номеру конечного элемента номера его граней:



2 грани 915 элемента являются граничными.

Их номера в массиве Edges: 1777, 2044, 2045, 2046.

Номера для узлов в массиве узлов и информация о соседях:

Номер грани	Номера узлов	Номера элементов
1777	275, 276, 282	915, 778
2044	275, 276, 321	915, -1
2045	275, 282, 321	919, 915
2046	276, 282, 321	915, -1

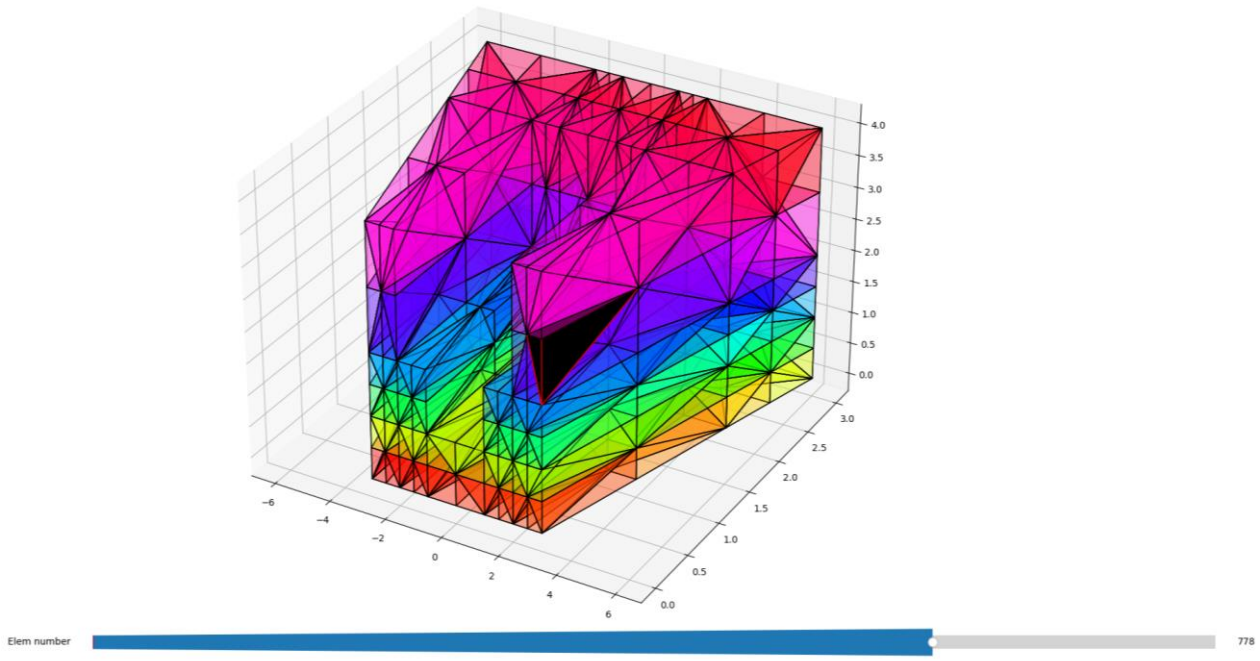
Как видно, 2 грани из 4 действительно являются граничными.

275 уз. = (2.5; 0; 3), 276 уз. = (3; 0; 3), 282 уз. = (4; 1; 3), 321 уз. = (3; 0; 4).

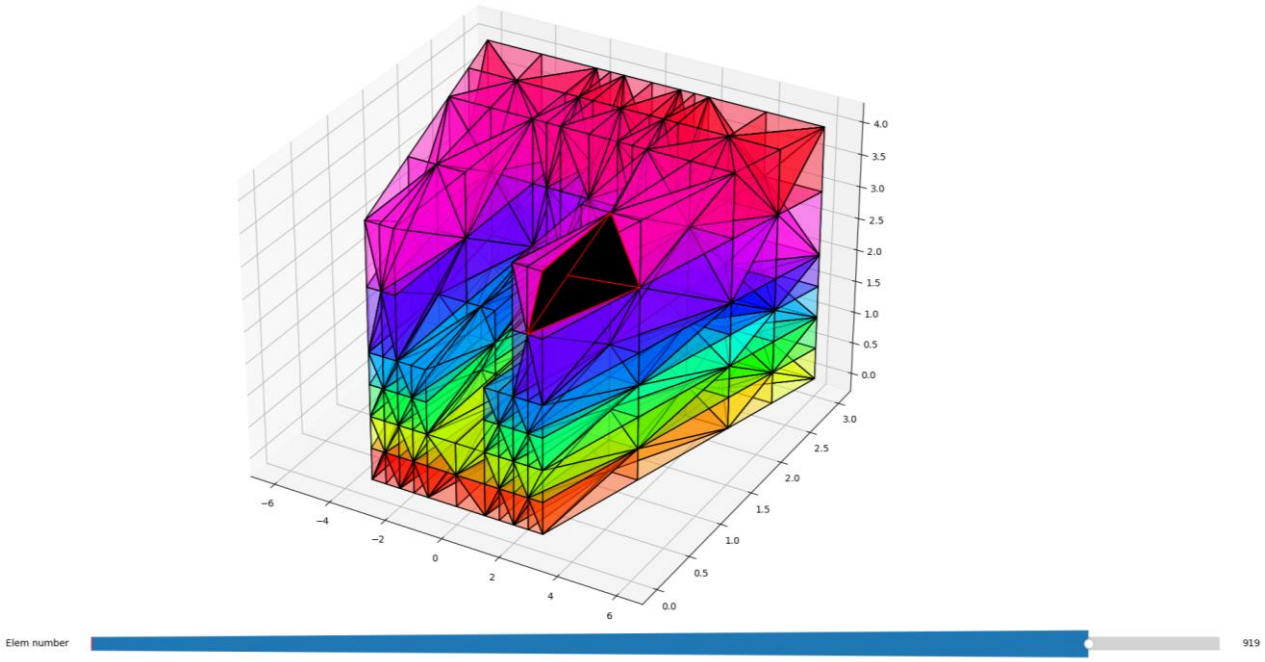
Сравнивая значения узлов граней с рисунком, можно сделать вывод, что номер узлов верны.

По всех номерах элементов присутствует 915.

Посмотрим соседа первой грани:



Затем третьей грани:



Как видно, соседи также найдены верно.

4. Выводы

Был разработан алгоритм для получения массивов граничных граней для учёта второго и третьего краевых условий, а также улучшен алгоритм нумерации граней как по временной, так и по пространственной сложности.

5. Код программы

Формирование массивов граничных граней:

```
void Grid::FormBC(const std::array<std::vector<double>, SIZE_NODE>& xyz, const
std::vector<int>& missingNodes, const std::vector<int>& missingElements)

{
   auto&& [x, y, z] = xyz;

   for (int h = 0; h < m_BC.size(); h++)
   {
      std::array<std::pair<int, int>, SIZE_NODE> limitsBoundaryEdge =
CalculationLimitsBoundaryEdge(m_BC[h].boundaries);

      switch (m_BC[h].typeBC)
      {
         case TYPE_BOUNDARY_CONDITION::FIRST:

            for (int k = limitsBoundaryEdge[2].first; k <=
limitsBoundaryEdge[2].second; k++)
               for (int j = limitsBoundaryEdge[1].first; j <=
limitsBoundaryEdge[1].second; j++)
                  for (int i = limitsBoundaryEdge[0].first; i <=
limitsBoundaryEdge[0].second; i++)
                  {
                        int number = k * x.size() * y.size() + j * x.size() + i;
                     //коррекция номеров вершин с учётом пропущенных вершин
                     number -= missingNodes[number];
                     m_BC_1.push_back(number);
                  }
            //удаление дубликатов из массива первых краевых(дубликаты расположены на
рёбрах, где грани соприкасаются)
            removeDuplicates(m_BC_1);
            break;

         case TYPE_BOUNDARY_CONDITION::SECOND:
         case TYPE_BOUNDARY_CONDITION::THIRD:
            std::vector<BoundaryEdge>& BC = (m_BC[h].typeBC ==
TYPE_BOUNDARY_CONDITION::SECOND) ? m_BC_2 : m_BC_3;

            bool reducelimit = false;
            int coordinateMatching = 0;
            //коррекция лимитов циклов
            for (int i = 0; i < limitsBoundaryEdge.size(); i++)
               if (limitsBoundaryEdge[i].first != limitsBoundaryEdge[i].second)

                  limitsBoundaryEdge[i].second--;
               else
               {
                  coordinateMatching = i;
                  if (0 < limitsBoundaryEdge[i].first)
                  {
                     std::array<double, SIZE_NODE> leftNode{0, 0, 0};
                     leftNode[i]--;
                     for (int j = 0; j < leftNode.size(); j++)
                        leftNode[j] = (xyz[j][limitsBoundaryEdge[j].first +
leftNode[j]] + xyz[j][limitsBoundaryEdge[j].second + leftNode[j]]) / 2.0;
                     if (InDomain(leftNode).first)
                        reducelimit = true;
                  }
               }

            for (int k = limitsBoundaryEdge[2].first; k <=
limitsBoundaryEdge[2].second; k++)
            {
               int kxy_0 = k * x.size() * y.size();
               int kxy_1 = (k + 1) * x.size() * y.size();
```

```
            for (int j = limitsBoundaryEdge[1].first; j <=
limitsBoundaryEdge[1].second; j++)
            {
               int jx_0 = j * x.size();
               int jx_1 = (j + 1) * x.size();

               for (int i = limitsBoundaryEdge[0].first; i <=
limitsBoundaryEdge[0].second; i++)
               {
                  std::array<int, NUMBER_NODES_CUBE / 2> vRect;
                  int elementOffset = k * (x.size() - 1) * (y.size() - 1) + j *
(x.size() - 1) + i;
                  //вычисление вершин прямоугольника, который будет разбит на
грани(треугольники)
                  switch (coordinateMatching)
                  {
                  case 0:
                     vRect = { kxy_0 + jx_0 + i, kxy_0 + jx_1 + i, kxy_1 + jx_0 + i,
kxy_1 + jx_1 + i };
                     if(reducelimit) elementOffset--;
                     break;
                  case 1:
                     vRect = { kxy_0 + jx_0 + i, kxy_0 + jx_0 + i + 1, kxy_1 + jx_0
+ i, kxy_1 + jx_0 + i + 1 };
                     if (reducelimit) elementOffset -= x.size() - 1;

                     break;
                  case 2:
                     vRect = { kxy_0 + jx_0 + i, kxy_0 + jx_0 + i + 1, kxy_0 + jx_1
+ i, kxy_0 + jx_1 + i + 1 };
                     if (reducelimit) elementOffset -= (x.size() - 1) * (y.size() -
1);
                     break;
                  }

                  //коррекция номеров вершин с учётом пропущенных вершин
                  for (int l = 0; l < vRect.size(); l++)
                     vRect[l] -= missingNodes[vRect[l]];

                  //коррекция номера элемента-куба с учётом пропущенных элементов-
кубов
                  elementOffset -= missingElements[elementOffset];
                  //каждый куб был разбит на тетраэдры
                  elementOffset *= m_gridPattern;

                  std::array<int, SIZE_EDGE> vertexes;
                  if (m_gridPattern == GRID_PATTERN::FIVE)
                  {
                     if ((i + j + k) % 2 == 0)
                     {
                        vertexes = { vRect[0], vRect[1], vRect[3] };
                        BC.push_back({ vertexes, SearchElement(vertexes,
elementOffset, elementOffset + m_gridPattern - 1) });
                        vertexes = { vRect[0], vRect[2], vRect[3] };
                        BC.push_back({ vertexes, SearchElement(vertexes,
elementOffset, elementOffset + m_gridPattern - 1) });
                     }
                     else
                     {
                        vertexes = { vRect[0], vRect[1], vRect[2] };
                        BC.push_back({ vertexes, SearchElement(vertexes,
elementOffset, elementOffset + m_gridPattern - 1) });
                        vertexes = { vRect[1], vRect[2], vRect[3] };
                        BC.push_back({ vertexes, SearchElement(vertexes,
elementOffset, elementOffset + m_gridPattern - 1) });
```

```
                    }
                }
                else
                {
                    vertexes = { vRect[0], vRect[1], vRect[2] };
                    BC.push_back({ vertexes, SearchElement(vertexes, elementOffset,
elementOffset + m_gridPattern - 1) });
                    vertexes = { vRect[1], vRect[2], vRect[3] };
                    BC.push_back({ vertexes, SearchElement(vertexes, elementOffset,
elementOffset + m_gridPattern - 1) });
                }

            }
        }
    }

    break;
  }
 }

}
```

Поиск грани в массиве конечных элементов:

```
int Grid::SearchElement(const std::array<int, SIZE_EDGE>& vertexes, int l, int r)
{
 for (int i = l; i <= r; i++)
 {
  bool edgeInElement = true;
  for (int j = 0; j < vertexes.size(); j++)
   if (binarySearch(m_elements[i].vertexes, vertexes[j], 0,
m_elements[i].vertexes.size() - 1) == -1)
    edgeInElement = false;
  if (edgeInElement) return i;
 }
 return -1;
}
```

Формирование массива граней:

```
void Grid::FormEdges()
{
 //номера граней в массиве вершин конечного элемента
 std::array<std::array<int, SIZE_EDGE>, NUMBER_EDGES_ELEMENT> numEdges{ {{0, 1,
2}, {0, 1, 3}, {0, 2, 3}, {1, 2, 3}} };

 //формирование массива граней
 for (int i = 0; i < m_elements.size(); i++)
  for (int j = 0; j < m_elements[i].vertexes.size(); j++)
   m_edges.push_back({ { m_elements[i].vertexes[numEdges[j][0]],
m_elements[i].vertexes[numEdges[j][1]], m_elements[i].vertexes[numEdges[j][2]] },
{i, -1} });

 //сортировка по трём вершинам с первой по третью с помощью составного ключа (если
ключи < 2^21)
 std::sort(m_edges.begin(), m_edges.end(), [](const auto& edge_1, const auto&
edge_2)
  {return (edge_1.vertexes[0] | (static_cast<int64_t>(edge_1.vertexes[1]) << 21)
| (static_cast<int64_t>(edge_1.vertexes[2]) << 42)) <
  (edge_2.vertexes[0] | (static_cast<int64_t>(edge_2.vertexes[1]) << 21) |
(static_cast<int64_t>(edge_2.vertexes[2]) << 42)); });

 //удаление дубликатов граней
 removeDuplicatesFromSortedArray(m_edges);

 std::vector<int> ind(m_edges.size(), 0);
```

```
   for (int i = 0; i < m_elements.size(); i++)
      for (int j = 0; j < m_elements[i].vertexes.size(); j++)
      {
         int num = GetNumberEdge(Edge{ {m_elements[i].vertexes[numEdges[j][0]],
m_elements[i].vertexes[numEdges[j][1]], m_elements[i].vertexes[numEdges[j][2]]}, {i,
-1} });
         if (ind[num])
         {
            m_edges[num].elemNums[1] = ind[num];
            m_edges[ind[num]].elemNums[1] = i;
         }
         else
            ind[num] = i;
      }
}
```

6. Используемые источники

1) Соловейчик Ю.Г., Рояк М.Э., Персова М.Г. Метод конечных элементов для решения скалярных и векторных задач: учеб. пособие. – Новосибирск: Изд-во НГТУ, 2007. – 896 с. («Учебники НГТУ»).