УДК 681.3.06

СТРУКТУРЫ ДАННЫХ ДЛЯ АНАЛИЗА СЕТОК ТЕТРАЭДРАЛЬНЫХ КОНЕЧНЫХ ЭЛЕМЕНТОВ*

© 2014 г. **В.Л. Тарасов, П.Д. Чекмарев**

Нижегородский госуниверситет им. Н.И. Лобачевского

vl-tarasov@yandex.ru

Поступила в редакцию 21.04.2014

Рассмотрена задача преобразования сплошной сетки тетраэдральных конечных элементов в ажурную сетку. Средствами языка C++ разработаны структуры данных для представления узлов сетки, конечных элементов и их ребер. Использованы встроенные массивы и векторы стандартной библиотеки шаблонов STL. Реализован алгоритм построения ажурных сеток, удаляющий из сетки конечные элементы, но оставляющий все ребра, что обеспечивает сохранение связности сетки. Построенные ажурные сетки содержат в 2,5–3 раза меньше конечных элементов, чем сплошные. Время работы алгоритма линейно зависит от размера сетки.

Ключевые слова: метод конечных элементов, ажурная сетка, класс, стандартная библиотека шаблонов, вектор, структуры данных.

Введение

Методы конечных элементов (МКЭ) предполагают, как правило, что расчетная область заполняется конечными элементами сплошь – без промежутков и наложения друг на друга. В статьях [1, 2] предложены *ажурные* численные схемы метода конечных элементов решения трехмерных задач механики сплошных сред, в которых конечные элементы заполняют расчетную область с промежутками. Это позволяет более эффективно использовать информацию в узлах сетки о напряженно-деформированном состоянии сред и избегать лишних вычислений при описании процессов деформирования без ущерба для точности получаемых численных решений. Теоретические обоснования подобных схем даны в [3–5].

Существуют расчетные комплексы, использующие МКЭ, например [6], позволяющие строить сплошные сетки конечных элементов. В [7] рассмотрено построение изначально ажурных сеток. В настоящей статье описываются структуры данных, предназначенные для моделирования конечно-элементных сеток, и алгоритм, позволяющий удалять часть конечных элементов без потери связности сетки, в предположении, что дана исходная сплошная сетка конечных элементов.

* Выполнено при финансовой поддержке РФФИ (грант 13-01-97052 р_поволжье_а).

Модель конечно-элементной сетки

Будем считать, что имеется предварительно построенная сплошная сетка конечных элементов (КЭ) из тетраэдров, покрывающая некоторое трехмерное тело. Сетка задается тремя декартовыми координатами узлов и четверками номеров узлов, определяющих конечные элементы сетки. Каждая пара узлов, относящихся к некоторому конечному элементу, образует ребро. Конечный элемент в виде тетраэдра имеет шесть ребер, которые могут принадлежать сразу нескольким элементам. Будем называть число конечных элементов, которым принадлежит ребро, его *степенью*.

Необходимо исключить максимальное число КЭ так, чтобы в ажурной сетке сохранились все ребра элементов. Это значит, что если два узла исходной сетки принадлежат одному или нескольким конечным элементам, то в ажурной сетке они будут принадлежать хотя бы одному КЭ.

Для моделирования конечно-элементной сетки использован язык C++ [8] и средства его стандартной библиотеки шаблонов [9]. На рис. 1 показана диаграмма разработанных классов. Диаграмма создана с помощью инструментария, входящего в состав Microsoft Visual Studio. Для построения диаграммы классов, входящих в состав какого-либо проекта, надо выбрать этот проект в обозревателе решений Visual Studio и выполнить команду контекстного меню **Перейти к схеме классов**.

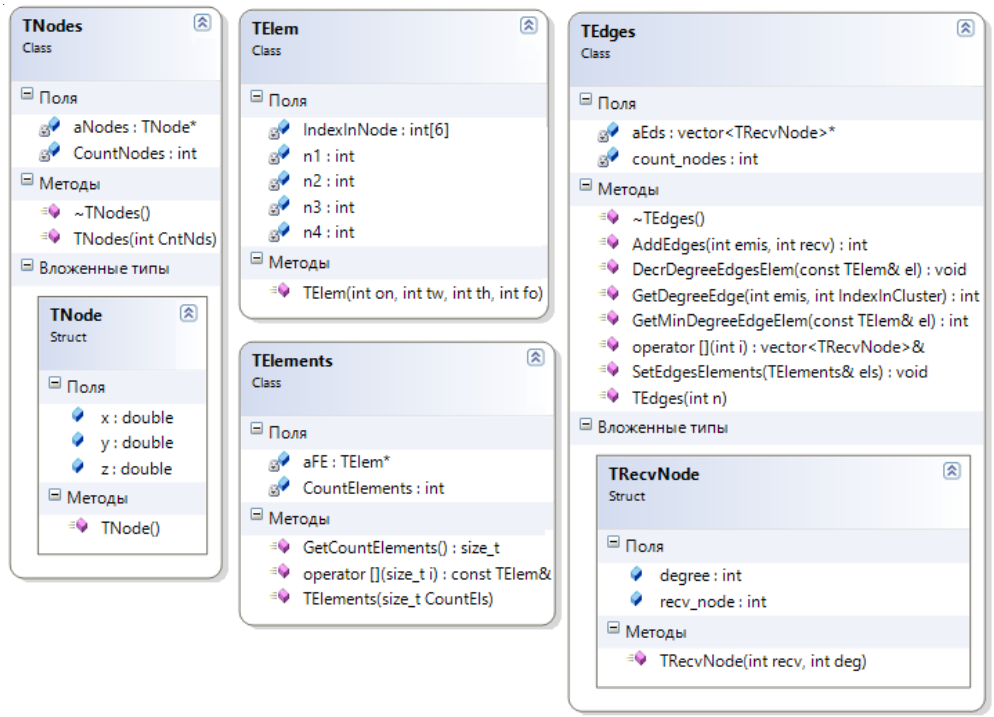

Рис. 1

Узлы сетки

Класс **`TNodes`** моделирует совокупность узлов конечно-элементной сетки. Отдельный узел описывается вложенной структурой **`TNode`**, содержащей три декар-

товы координаты x, y, z отдельного узла. Все узлы сетки хранятся в массиве **aNodes** размера **CountNodes**.

```
class TNodes{           // Узлы сетки
   struct TNode         // Узел сетки
   {
      double x, y, z;  // Координаты узла
      TNode(void){x = y = z = 0; }
   };
   TNode* aNodes;       // Массив узлов сетки. Индекс массива равен номеру узла минус 1.
   int CountNodes;      // Число узлов
public:
   TNodes(int CntNds)
   {
      CountNodes = CntNds;
      aNodes = new TNode[CountNodes];
   }
   ~TNodes() {delete[] aNodes;}
   friend istream& operator>>(istream& is, TNodes& nds);
   friend ostream& operator<<(ostream& os, const TNodes& nds);
};
```

При задании исходных данных принято нумеровать узлы последовательно, начиная с 1. Так как в C++ элементы массивов нумеруются с нуля, то индекс элемента массива, соответствующего некоторому узлу, равен номеру узла минус 1.

Реализация операторов ввода и вывода узлов сетки очевидна.

Структуры данных для конечных элементов

Класс **TElem** моделирует один конечный элемент. Он содержит 4 целочисленных номера узлов сетки, являющихся вершинами тетраэдрального КЭ. Номера узлов одного КЭ для удобства хранятся в упорядоченном виде: **n1 < n2 < n3 < n4.** Упорядочение выполняется в конструкторе. Ребрами конечного элемента являются 6 пар его вершин. Условимся считать первой вершиной ребра узел с меньшим номером, тогда ребра одного конечного элемента опишутся парами номеров узлов: **(n1, n2)**, **(n1, n3)**, **(n1, n4)**, **(n2, n3)**, **(n2, n4)**, **(n3, n4)**. Специальной структуры данных для хранения таких пар не создаем, но подразумеваем, что ребра представляются такими парами. Узел ребра с меньшим номером назовем «*выпускающим*», а с большим – «*принимающим*». Целочисленный массив **IndexInNode** класса **TElem** хранит индексы 6 ребер конечного элемента в кластерах ребер (описаны далее), соответствующих выпускающим узлам ребер.

```
class TElem               // Содержит четыре номера узлов - вершин конечного элемента.
{                         // Нумерация узлов - от 1
   friend class TEdges;      // Дружественный класс ребер
   int n1, n2, n3, n4;       // Номера узлов
   int IndexInNode[6];       // Номера ребер КЭ в кластерах ребер
public:
   // Конструктор. По умолчанию создается элемент с нулевыми номерами
   TElem(int on = 0, int tw = 0, int th = 0, int fo = 0);

   friend ostream& operator<<(ostream& os, const TElem& el)     // Вывод узлов элемента
   { return os << el.n1 << '\t' << el.n2 << '\t' << el.n3 << '\t' << el.n4 << endl; }
};

TElem::TElem(int on, int tw, int th, int fo)   // Конструктор
{
   if(on < tw){ n1 = on; n2 = tw; }            // Упорядочиваем первые два номера
   else{ n1 = tw; n2 = on; }                   // Упорядочиваем три номера
```

```
    if(n2 < th) n3 = th;
    else{ n3 = n2;
       if(n1 < th) n2 = th;
       else{ n2 = n1; n1 = th; }
    }
    if(n3 < fo) n4 = fo;                          // Упорядочиваем 4 номера
    else{ n4 = n3;
       if(n2 < fo) n3 = fo;
       else{ n3 = n2;
          if(n1 < fo) n2 = fo;
          else{ n2 = n1; n1 = fo; }
       }
    }
       IndexInNode[0] = IndexInNode[1] = IndexInNode[2] =
                        IndexInNode[3] = IndexInNode[4] = IndexInNode[5] = -1;
}
```

Класс **TElements** содержит массив **aFE**, содержащий все конечные элементы сетки. При задании исходных данных конечные элементы нумеруются, начиная с единицы, поэтому индекс массива **aFE** на единицу меньше номера соответствующего конечного элемента.

```
class TElements{
    friend class TEdges;
    TElem* aFE;            // Массив конечных элементов. Индекс равен номеру КЭ минус 1
    int CountElements;     // Число КЭ
public:
    TElements(int CountEls)
    {
       CountElements = CountEls;
       aFE = new TElem[CountEls];
    }
    int GetCountElements()     // Возвращает число КЭ
    { return CountElements; }
    const TElem& operator[](size_t i) const  // Доступ к КЭ
    { return aFE[i]; }
    friend istream& operator>>(istream& in, TElements& elms);       // Ввод конечн. элементов
    friend ostream& operator<<(ostream& out, const TElements& elms); // Вывод конечн. элементов
};
```

В конструкторе класса **TElements** создается динамический массив конечных элементов заданного размера, причем все номера узлов у всех КЭ вначале нулевые, как задано в конструкторе по умолчанию. При вводе массива конечных элементов вызывается конструктор отдельного КЭ, которому передается четверка введенных номеров узлов. Конструктор создает безымянный КЭ с упорядоченными номерами узлов, который присваивается КЭ из массива.

```
istream& operator>>(istream& in, TElements& elms)  // Ввод конечных элементов
{
    size_t on, tw, th, fr;
    for(size_t i = 0; i < elms.CountElements; i++){
       in >> on >> tw >> th >> fr;
       elms.aFE[i] = TElem(on, tw, th, fr);
    }
    return in;
}
```

Реализация оператора вывода конечных элементов очевидна.

Ребра конечных элементов

Для моделирования ребер конечных элементов возможны различные подходы. Например, совокупность ребер можно представить в виде матрицы смежности,

номерами строк и столбцов которой являются номера узлов конечно-элементной сетки, а на пересечении строк и столбцов стоит 1, если соответствующие узлы соединяются ребром, и 0, если ребра между узлами нет. Такое представление ребер позволяет быстро проверять наличие ребра между заданными узлами, но требует много памяти.

Можно представить набор ребер в виде одного вектора пар номеров узлов, между которыми есть ребро. Размер такого вектора будет равен числу ребер, но при работе с таким представлением потребуется много времени для поиска ребра, принадлежащего тому или иному элементу. В настоящей статье предложено представление для ребер, совмещающее небольшой размер памяти для хранения данных и быстрый доступ к ребрам. Ребра конечных элементов не имеют направления, однако условимся считать, что ребро направлено от узла с меньшим номером (*выпускающего* узла) к узлу с *бóльшим* номером (*принимающему* узлу). Это позволяет упорядочить ребра по меньшему номеру узла.

Класс **TEdges** описывает ребра конечно-элементной сетки. Вложенная структура **TRecvNode** хранит номер принимающего узла и степень ребра, соединяющего данный принимающий узел с выпускающим узлом этого ребра. Совокупность ребер, выходящих из некоторого выпускающего узла, назовем кластером ребер, который представим вектором из элементов типа **TRecvNode**. Размер этого вектора заранее не известен и определяется числом ребер, выпущенных из данного узла, и, как показывают рассмотренные примеры конечно-элементных сеток, не превышает двух десятков. Все кластеры ребер представляются массивом **aEds**, элементами которого являются векторы из элементов типа **TRecvNode**. Индекс массива **aEds**, увеличенный на единицу, является номером выпускающего узла ребра, то есть ребро (i,j), $i<j$, где i,j – номера узлов сетки, входит в кластер **aEds[i - 1]**. По номеру принимающего узла j легко найти ребро в кластере ребер.

```
class TEdges{                                  // Совокупность ребер
   struct TRecvNode{
      int recv_node,                           // Больший (принимающий) узел ребра
          degree;                              // Степень ребра, входящего в данный узел
                                               // из некоторого выпускащего узла
      TRecvNode(int recv = 0, int deg = 0)     // Конструктор
      {
         recv_node = recv;
         degree = deg;
      }
   };

   vector<TRecvNode>*  aEds;                   // Массив векторов из принимающих узлов
   int count_nodes;                            // Число узлов, из которых выпущены ребра
public:
   TEdges(int n)                               // Конструктор
   {
      count_nodes = n;
      aEds = new vector<TRecvNode>[n];
   }

   ~TEdges()                                   // Деструктор
   { delete[] aEds; }

   int AddEdges(int emis, int recv);           // Добавить ребро (emis, recv)

   vector<TRecvNode> & operator[](int i)       // Кластер ребер, исходящих из узла i + 1
   { return aEds[i]; }
```

```
    int GetDegreeEdge(int emis, int IndexInCluster)    // Степень ребра с выпускающим узлом emis
    { return aEds[emis - 1][IndexInCluster].degree; }  // и номером в кластере IndexInCluster

    int GetMinDegreeEdgeElem(const TElem& el);  // Минимальная степень ребра КЭ el

    void SetEdgesElements(TElements& els);      // Включает в набор ребра конечн. элементов els

    void DecrDegreeEdgesElem(const TElem& el);  // Уменьшить на 1 степень ребер элемента el

};
```

Метод, добавляющий ребро с заданными выпускающим и принимающим узлами:

```
int TEdges::AddEdges(int emis, int recv) // Добавление ребра (emis, recv) в кластер ребер,
{                           // emis - выпускающий узел, recv - принимающий, emis, < recv.
                            // Возвращается номер ребра в кластере
    bool Exist = false;                             // Ребро НЕ существует
    int i = 0, sz = aEds[emis - 1].size();          // sz - число ребер в кластере
    for(; i < sz; i++)
       if(recv == aEds[emis - 1][i].recv_node){
          Exist = true;
          break;
       }
    if(!Exist)                        // Ребра (emis, recv) пока нет в кластере ребер,
       aEds[emis - 1].push_back(TRecvNode(recv, 1));         // ребро добавляется.
    else
       aEds[emis - 1][i].degree++;                  // Иначе увеличивается степень ребра
    return i;                                       // Индекс ребра в кластере
}
```

Метод **`SetEdgesElements()`** создает совокупность всех ребер конечно-элементной сетки, распределяя ребра по кластерам, определяемым меньшими номерами узлов.

```
void TEdges::SetEdgesElements(TElements& els) // Создать совокупность ребер по массиву КЭ els
{
   for(size_t i = 0; i < els.GetCountElements(); i++){
      els.aFE[i].IndexInNode[0] = AddEdges(els.aFE[i].n1, els.aFE[i].n2);
      els.aFE[i].IndexInNode[1] = AddEdges(els.aFE[i].n1, els.aFE[i].n3);
      els.aFE[i].IndexInNode[2] = AddEdges(els.aFE[i].n1, els.aFE[i].n4);
      els.aFE[i].IndexInNode[3] = AddEdges(els.aFE[i].n2, els.aFE[i].n3);
      els.aFE[i].IndexInNode[4] = AddEdges(els.aFE[i].n2, els.aFE[i].n4);
      els.aFE[i].IndexInNode[5] = AddEdges(els.aFE[i].n3, els.aFE[i].n4);
   }
}
```

Метод **`GetMinDegreeEdgeElem()`** возвращает минимальную степень ребра, принадлежащего заданному конечному элементу.

```
int TEdges::GetMinDegreeEdgeElem(const TElem& el)     // Минимальная степень ребра
{                                                     // в конечном элементе el
   int min = GetDegreeEdge(el.n1, el.IndexInNode[0]);    // Степень 1-го ребра
   int d = GetDegreeEdge(el.n1, el.IndexInNode[1]);      // Степень 2-го ребра
   if(d < min) min = d;
   d = GetDegreeEdge(el.n1, el.IndexInNode[2]);          // Степень 3-го ребра
   if(d < min) min = d;
   d = GetDegreeEdge(el.n2, el.IndexInNode[3]);          // Степень 4-го ребра
   if(d < min) min = d;
   d = GetDegreeEdge(el.n2, el.IndexInNode[4]);          // Степень 5-го ребра
   if(d < min) min = d;
   d = GetDegreeEdge(el.n3, el.IndexInNode[5]);          // Степень 6-го ребра
   if(d < min) min = d;
   return min;
}
```

Метод **DecrDegreeEdgesElem()** уменьшает на единицу степень всех ребер заданного конечного элемента, вызывается при исключении конечного элемента из сетки.

```
void TEdges::DecrDegreeEdgesElem(const TElem& el)  // Уменьшить на 1 степень ребер элемента el
{
    aEds[el.n1 - 1][el.IndexInNode[0]].degree--;
    aEds[el.n1 - 1][el.IndexInNode[1]].degree--;
    aEds[el.n1 - 1][el.IndexInNode[2]].degree--;
    aEds[el.n2 - 1][el.IndexInNode[3]].degree--;
    aEds[el.n2 - 1][el.IndexInNode[4]].degree--;
    aEds[el.n3 - 1][el.IndexInNode[5]].degree--;
}
```

Тестирование

Ажурную сетку строит функция **MakeTracery()**, которой передаются два файловых потока **inf** и **fout**. Поток **inf** связан со входным файлом, из которого читаются исходные данные, поток **fout** связан с выходным файлом, в который записываются параметры ажурной сетки. Функция возвращает количество конечных элементов, оставшихся в ажурной сетке.

```
int MakeTracery(ifstream& inf, ofstream& fout)
{
    size_t CountNodes, CountFE;          // Число узлов сетки и число конечных элементов
    inf >> CountNodes >> CountFE;              // Ввод числа узлов и числа КЭ
    TNodes Nodes(CountNodes);                  // Узлы сетки
    inf >> Nodes;                              // Ввод трех координат всех узлов
    TElements Elements(CountFE);               // Конечные элементы
    inf >> Elements;                           // Ввод четверок номеров узлов КЭ
    TEdges Edges(CountNodes);                  // Ребра сетки
    Edges.SetEdgesElements(Elements);          // Формирование кластеров ребер
    int CountDeletedElements = 0;              // Счетчик удаленных элементов
    int minDegree;                             // Минимальная степень ребра КЭ
    vector<bool> abFE(CountFE);                // Признаки включенности КЭ в ажурную сетку
    //   Перебор конечных элементов и удаление лишних элементов
    for(int indElem = 0; indElem < CountFE;  indElem++){           // Прямой обход
//  for(int indElem = CountFE - 1; indElem >= 0;indElem--){        // обратный обход

        abFE[indElem] = true;                                  // КЭ в ажурной сетке
        minDegree = Edges.GetMinDegreeEdgeElem(Elements[indElem]);
        if(minDegree > 1) {                                    // КЭ можно исключить из сетки
            Edges.DecrDegreeEdgesElem(Elements[indElem]);      // Уменьшение степени ребер КЭ
            CountDeletedElements++;                            // Увеличение счетчика удаленных КЭ
            abFE[indElem] = false;                             // КЭ не входит в ажурную сетку
        }
    }
    size_t cFE = CountFE - CountDeletedElements;           // Число КЭ в ажурной сетке
    fout << CountNodes << endl << cFE << endl;
    fout << Nodes;                                         // Вывод трех координат всех узлов
// Вывод номеров КЭ ажурной сетки и номеров узлов этих КЭ
    for(size_t indElem = 0; indElem < CountFE;  indElem++)
        if(abFE[indElem])
            fout << (indElem + 1) << " " << Elements[indElem];
    return cFE;
}
```

Входной и выходной файлы предполагаются текстовыми. В начале входного файла находятся два целых числа – количество узлов сетки и количество конечных элементов. Функция **MakeTracery()** считывает их соответственно в переменные **CountNodes** и **CountFE**. Далее во входном файле расположены **CountNodes** троек вещественных чисел – координат узлов сетки. После координат узлов перечисляются **CountFE** четверок целых чисел – номеров узлов сетки, являющихся верши-

нами конечных элементов. Координаты узлов считываются в объект **Nodes**, а номера узлов конечных элементов – в объект **Elements**. Объект **Edges**, представляющий совокупность ребер, сразу после создания содержит **CountNodes** пустых кластеров ребер. Число кластеров ребер принято равным общему числу узлов сетки. Кластеры заполняются ребрами всех конечных элементов функцией **Edges.SetEdgesElements(Elements)**, при этом определяются степени всех ребер.

Затем в цикле перебираются все конечные элементы, и если минимальная степень ребер элемента больше 1, этот конечный элемент удаляется из расчетной сетки, что фиксируется установкой в **false** соответствующего члена вектора признаков **abFE**. По завершении обработки конечных элементов в выходной файл выводятся число узлов сетки, число конечных элементов в ажурной сетке, координаты узлов, номера конечных элементов, образующих ажурную сетку, и номера узлов этих КЭ. Результаты тестирования программы на нескольких сетках приведены в таблице 1.

Таблица 1

Число узлов **CountNodes**	Число КЭ **CountFE**	Число КЭ ажурной сетки **cFE**			$\frac{\textbf{CountFE}}{\textbf{cFE}}$	Время работы T, с	$\frac{10000\ T}{\textbf{CountFE}}$
		прямой обход	обратный обход	разница, %			
233	727	285	282	1,05	2,55	0,109	1,50
3504	12529	4623	4426	4,26	2,71	2	1,60
12463	57924	19138	19029	0,57	3,03	8,531	1,47
73427	300082	102541	98482	3,96	2,93	46,065	1,54
168125	821544	264586	269085	1,70	3,11	123,944	1,51

Обход конечных элементов проводился в порядке возрастания их номеров (прямой обход) и в порядке убывания номеров (обратный обход). Для реализации нужного варианта обхода в функции **MakeTracery()** оставлялась соответствующая версия цикла. При разных направлениях обхода получаются разные ажурные сетки, но число оставленных в сетке КЭ отличается несущественно – менее чем на 5%. Важно, что число конечных элементов в ажурной сетке в 2,5–3 раза меньше, чем в исходной сетке. Это приводит к пропорциональному сокращению времени вычислений при расчете тел на таких сетках.

Интересен показатель **10000*T*/CountFE**, пропорциональный отношению времени вычислений к числу конечных элементов. Видно, что его значение составляет около 1,5 и практически не зависит от числа КЭ. Таким образом, время работы алгоритма линейно зависит от размера сетки.

Заключение

Рассмотрено построение ажурных сеток конечных элементов на основе сплошных сеток, получаемых сторонними сеточными генераторами. Разработаны структуры данных, использующие встроенные массивы языка C++ и шаблон **vector** стандартной библиотеки шаблонов. Как показали численные эксперименты, количество конечных элементов в ажурной сетке в 2,5–3 раза меньше, чем в исходной сплошной сетке. Использование ажурной сетки позволяет в соответствующее число раз снизить время расчетов. Численные эксперименты показывают, что время работы предложенного алгоритма построения ажурной сетки линейно зависит от размера задачи, который характеризуется числом конечных элементов.

Предложенные структуры данных и алгоритм «прореживания» можно применять для конечных элементов, отличающихся от тетраэдральных. Для этого надо использовать в программе другое количество узлов и ребер конечных элементов, модифицировав функцию **`SetEdgesElements()`**, которая формирует ребра одного конечного элемента, и функцию **`DecrDegreeEdgesElem()`**, уменьшающую степени ребер. Например, для двумерных сеток, состоящих из треугольников, количество узлов – 4, количество ребер – 3; для пространственных шестигранников количество узлов – 8, количество ребер – 12.

Список литературы

1. *Чекмарев Д.Т.* Ажурные схемы метода конечного элемента // Прикладные проблемы прочности и пластичности: Межвуз. сб. / Нижегород. ун-т, 1997. Вып. 55. С. 157–159.

2. *Чекмарев Д.Т.* Численные схемы метода конечного элемента на «ажурных» сетках // Вопросы атомной науки и техники, Сер. Математическое моделирование физических процессов. 2009. Вып. 2. С. 49–54.

3. *Баженов В.Г.*, *Чекмарев Д.Т.* Об индексной коммутативности численного дифференцирования // Ж. вычисл. математики и мат. физики. 1989. Т. 29, №5. С. 662–674.

4. *Баженов В.Г.*, *Чекмарев Д.Т.* Вариационно-разностные схемы в нестационарных волновых задачах динамики пластин и оболочек. Н. Новгород: Изд-во Нижегород. ун-та, 1992. 159 с.

5. *Баженов В.Г.*, *Чекмарев Д.Т.* Решение задач нестационарной динамики пластин и оболочек вариационно-разностным методом. Н.Новгород: Изд-во Нижегород. ун-та, 2000. 118 с.

6. Применение системы ANSYS к решению задач механики сплошной среды: Практич. руководство / Под ред. А.К. Любимова. Н. Новгород: Изд-во Нижегород. ун-та, 2006. 277 с.

7. *Чекмарев Д.Т.*, *Кастальская К.А.* О построении трехмерных ажурных сеток // Супервычисления и математическое моделирование: Труды XII Междунар. семинара. Саров, 2011. С. 374–381.

8. The C++ Resources Network [Электронный ресурс]. – Режим доступа: http://www.cplusplus.com/

9. *Джосьютис Н.* C++ Стандартная библиотека. Для профессионалов. СПб.: Питер, 2004. 730 с.

DATA STRUCTURES FOR ANALYSIS MESHES OF TETRAHEDRAL FINITE ELEMENTS

V.L. Tarasov, P.D. Chekmarev

The problem of transformation of a continuous mesh of tetrahedral finite elements in the rare mesh is considered. With means of language C++ are developed data structures for representation of nodes of a mesh, finite elements and their edges. The native arrays and the vectors of standard library of templates STL are used. The algorithm of construction of the rare meshes, deleting from a mesh a part of finite elements, but leaving all edges, that provides preservations of connectivity of a mesh is realized. The constructed rare meshes contain in 2,5–3 times less finite elements, than continuous meshes. The time of work of the algorithm linearly depends on the size of a mesh.

Keywords: finite element method, rare mesh, class, standard template library, vector, data structures.