DATA STRUCTURE OF SIMPLICIAL TRIANGULATIONS

LONG CHEN

ABSTRACT. This is extracted from "*i*FEM: an innovative finite element method package in Matlab" by Long Chen.

1. TRIANGULATION

In this section, we shall discuss triangulations used in finite element methods. We would like to distinguish two structures of a triangulation: one is the topology of a mesh which is determined by the combinatorial connectivity of vertices; another is the geometric shape which depends on the location of vertices. Correspondingly there are two basic data structure used to represents a triangulation. The data structures and corresponding algorithms on the topological/combinatorial structure of triangulations discussed here can be applied to other adaptive methods or other discretization methods. Note that the topological/combinatorial structure of triangulations is not thoroughly discussed in the literature.

1.1. **Geometric simplex and triangulation.** Let $\boldsymbol{x}_i = (x_{1,i}, \cdots, x_{d,i})^t, i = 1, \cdots, d+1$, be $d+1$ points in $\mathbb{R}^d, d \geq 1$, which do not all lie in one hyper-plane. The *convex hull* of the $d+1$ points $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{d+1}$,

$$\tau := \{\boldsymbol{x} = \sum_{i=1}^{d+1} \lambda_i \boldsymbol{x}_i \,|\, 0 \leq \lambda_i \leq 1, i = 1 : d+1, \sum_{i=1}^{d+1} \lambda_i = 1\} \tag{1}$$

is defined as a *geometric d-simplex* generated (or spanned) by the vertices $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{d+1}$. For example, a triangle is a 2-simplex and a tetrahedron is a 3-simplex. For the convenience of notation, we also call a point 0-simplex. For an integer $0 \leq m \leq d-1$, an m-dimensional face of τ is any m-simplex generated by $m+1$ of the vertices of τ. Zero-dimenisonal faces are called vertices or nodes and one-dimensional faces are called edges.

The numbers $\lambda_1(\boldsymbol{x}), \cdots, \lambda_{d+1}(\boldsymbol{x})$ are called *barycentric coordinates* of $\boldsymbol{x}$ with respect to the $d+1$ points $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{d+1}$. There is a simple geometric meaning of the barycentric coordinates. Given a $\boldsymbol{x} \in \tau$, let $\tau_i(\boldsymbol{x})$ be the simplex by replacing the vertex $\boldsymbol{x}_i$ of τ by $\boldsymbol{x}$. Then it can be shown that

$$\lambda_i(\boldsymbol{x}) = |\tau_i(\boldsymbol{x})|/|\tau|, \tag{2}$$

where $|\cdot|$ is the Lebesgure measure in $\mathbb{R}^d$, namely area in two dimensions and volume in three dimensions. From (2), it is easy to deduce that $\lambda_i(\boldsymbol{x})$ is an affine function of $\boldsymbol{x}$ and vanished on the $(d-1)$-face opposite to the vertex $\boldsymbol{x}_i$.

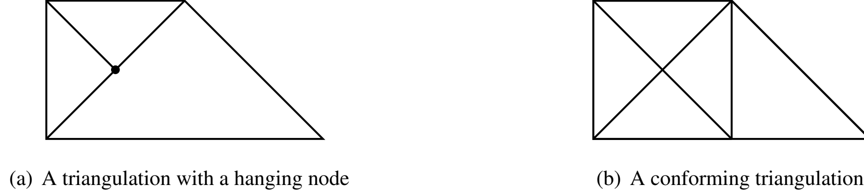
Let Ω be a polyhedral domain in $\mathbb{R}^d, d \geq 1$. A geometric triangulation $\mathcal{T}$ of Ω is a set of d-simplices such that

$$\cup_{\tau \in \mathcal{T}} \tau = \overline{\Omega}, \quad \text{and} \quad \mathring{\tau}_i \cap \mathring{\tau}_j = \varnothing, \text{ for any } \tau_i, \tau_j \in \mathcal{T}, i \neq j.$$

Remark 1.1. There are other type of meshes by partition the domain into quadrilateral (in 2-D), cube (in 3-D), hexahedron (in 3-D), and so on. In this paper, we restrict ourself to simplicial triangulations and thus will mix the usage of three words: grid, triangulation, and mesh. We also identify the words 'node' and 'vertex' since only linear element will be used in this paper.

Date: October 26, 2008.

There are two conditions that we shall impose on triangulations that are important in the finite element computation. The first requirement is a topological property. A triangulation $\mathcal{T}$ is called *conforming* or *compatible* if the intersection of any two simplexes τ and τ' in $\mathcal{T}$ is either empty or a common lower dimensional simplex (nodes in two dimensions, nodes and edges in three dimensions). The node falls into the interior of a simplex is called a hanging node; See Figure 1 (a).



(a) A triangulation with a hanging node

(b) A conforming triangulation

FIGURE 1. Two triangulations. The left is non-conforming and the right is conforming.

The second important condition is on the geometric structure. A set of triangulations $\mathscr{T}$ is called *shape regular* if there exists a constant c_0 such that

$$\max_{\tau\in\mathcal{T}} \frac{\operatorname{diam}(\tau)^d}{|\tau|} \leq c_0, \quad \forall \mathcal{T} \in \mathscr{T}, \tag{3}$$

where $\operatorname{diam}(\tau)$ is the diameter of τ. In two dimensions, it is equivalent to the minimal angle of each triangulation is bounded below uniformly in the shape regular class.

Remark 1.2. In addition to (3), if there exists a constant c_1 such that

$$\frac{\max_{\tau\in\mathcal{T}} |\tau|}{\min_{\tau\in\mathcal{T}} |\tau|} \leq c_1, \quad \forall \mathcal{T} \in \mathscr{T}, \tag{4}$$

$\mathscr{T}$ is called *quasi-uniform*.

1.2. **Abstract simplex and simplicial complex.** To distinguish the topological structure with geometric one, we now understand the points as abstract entities and introduce *abstract simplex* or *combinatorial simplex* [**?**]. The set $\tau = \{v_1, \cdots, v_{d+1}\}$ of $d+1$ abstract points is called an abstract d-simplex. A *face* σ of a simplex τ is a simplex determined by a non-empty subset of τ. A *proper face* is any face different from τ.

Let $\mathcal{N} = \{v_1, v_2, \cdots, v_N\}$ be a set of N abstract points. An *abstract/combinatorial simplicial complex* $\mathcal{T}$ is a set of simplices formed by finite subsets of $\mathcal{N}$ such that

(1) if $\tau \in \mathcal{T}$ is a simplex, then any face of τ is also a simplex in $\mathcal{T}$;
(2) for two simplices $\tau_1, \tau_2 \in \mathcal{T}$, the intersection $\tau_1 \cap \tau_2$ is a face of both τ_1 and τ_2.

By the definition, a two dimensional combinatorial simplicial complex $\mathcal{T}$ contains not only triangles but also edges and vertices of these triangles. A geometric triangulation defined before is only a set of d-simplex but no faces. By including all faces, we shall get a simplicial complex if the triangulation is conforming which corresponds to the second requirement of a simplicial simplex.

A subset $\mathcal{M} \subset \mathcal{T}$ is a subcomplex of $\mathcal{T}$ if $\mathcal{M}$ is a simplicial complex itself. Important classes of subcomplex includes the *star* or *ring* of a simplex. That is for a simplex $\sigma \in \mathcal{T}$

$$\operatorname{star}(\sigma) = \{\tau \in \mathcal{T}, \sigma \subset \tau\}.$$

If two, or more, simplices of $\mathcal{T}$ share a common face, they are called *adjacent* or *neighbors*. The boundary of $\mathcal{T}$ is formed by any proper face that belongs to only one simplex, and its faces.

By associating the set of abstract points with geometric points in $\mathbb{R}^n, n \geq d$, we obtain a geometric shape consisting of piecewise flat simplices. This is called a geometric realization of an abstract simplicial complex or, using the terminology of geometry, the embedding of $\mathcal{T}$ into $\mathbb{R}^n$. The embedding is uniquely determined by the identification of abstract and geometric vertices.

A planar triangulation is a two dimensional abstract simplicial complex which can be embedded into $\mathbb{R}^2$ and thus called 2-D triangulation. A 2-D simplicial complex could also be embedding into $\mathbb{R}^3$ and result a triangulation of a surface. Therefore the surface mesh in $\mathbb{R}^3$ is usually called $2\frac{1}{2}$-D triangulation. For these two different embedding, they many have the same combinatorary structure as an abstract simplicial complex but different geometric structure by representing a flat domain in $\mathbb{R}^2$ or a surface in $\mathbb{R}^3$.

1.3. **Data structure for triangulations.** We shall discuss the data structure to represent triangulations and facilitate the mesh adaptation procedure. There is a dilemma for the data structure in the implementation level. If more sophisticated data structure is used to easily traverse in the mesh, for example, to save the star of vertices or edges, it will simplify the implementation of most adaptive finite element subroutines. On the other hand, if the triangulation is changed, for example, a triangle is bisected, one has to update those data structure which in turn complicates the implementation.

Our solution is to maintain two basic data structure and construct auxiliary data structure inside each subroutine when it is necessary. It is not optimal in terms of the computational cost. But it will benefit the interface of accessing subroutines, simplify the coding and save the memory. Also as we shall see soon, the auxiliary data structure can be constructed by sparse matrixlization efficiently. This is an example we scarifies a small factor of efficiency to gain the simplicity.

1.3.1. *Basic data structure.* The matrices `node(1:N,1:d)` and `elem(1:NT,1:d+1)` are used to represent a d-dimensional triangulation embedded in $\mathbb{R}^d$, where N is the number of vertices and NT is the number of elements. These two matrices represent two different structure of a triangulation: `elem` for the topology and `node` for the embedding.

The matrix `elem` represents a set of abstract simplices. The index set $\{1, 2, \cdots, N\}$ is called the global index set of vertices. Here a vertex is thought as an abstract entity. By definition, `elem(t,1:d+1)` are the global indices of $d + 1$ vertices which form the abstract d-simplex `t`. Note that any permutation of vertices of t will represent the same abstract simplex.

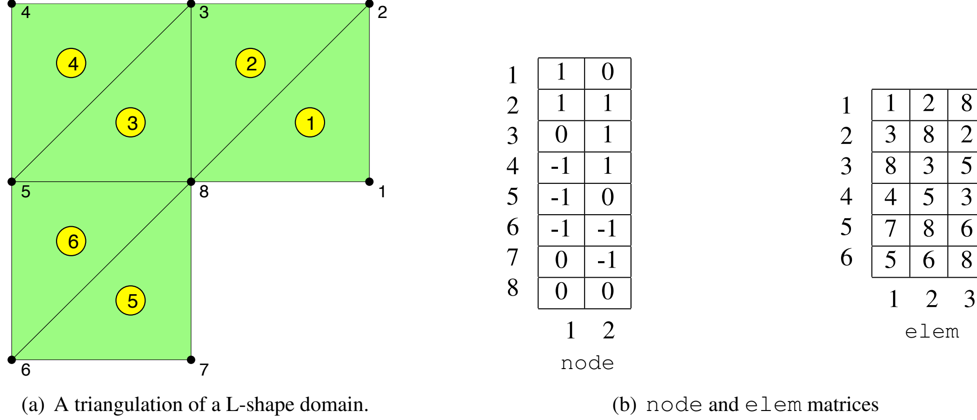
The matrix `node` gives the geometric realization of the simplicial complex. For example, for a 2-D triangulation, `node(k,1:2)` contain x- and y-coordinates of the k-th node. We shall always order the vertices of a simplex such that the signed volume is positive. That is in 2-D, three vertices of a triangle is ordered counter-clockwise and in 3-D, the ordering of vertices follows the right-hand rule. Note that even permutation of vertices is still allowed to represent the same element.

As an example, `node` and `elem` matrices to represent the triangulation of the L-shape domain $(-1, 1) \times (-1, 1) \backslash ([0, 1] \times [0, -1])$ in the Figure 2 (a) and (b).

1.3.2. *Auxiliary data structure for 2-D triangulation.* We shall discuss how to extract the topological or combinatorial structure of a triangulation by using `elem` array only. The combinatorial structure will benefit the implementation of finite element methods.

edge. We first complete the 2-D simplicial complex by constructing the 1-dimensional simplex. In the matrix `edge(1:NE,1:2)`, the first and second rows contain indices of the starting and ending points. The column is sorted in the way that for the k-th edge, `edge(k,1)<edge(k,2)`. The following code will generate an `edge` matrix.

```
1 totalEdge = sort([elem(:,[1,2]); elem(:,[1,3]); elem(:,[2,3])],2);
2 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
3 edge = [j,i]; bdEdge = [j(s==1),i(s==1)];
```



(a) A triangulation of a L-shape domain.

(b) `node` and `elem` matrices

FIGURE 2. (a) is a triangulation of the L-shape domain $(-1,1) \times (-1,1) \backslash ([0,1] \times [0,-1])$ and (b) is its representation using `node` and `elem` matrices.

The first line collect all edges from the set of triangles and sort the column such that `totalEdge(k,1) <totalEdge(k,2)`. The interior edges are repeated twice in `totalEdge`. We use the summation property of `sparse` command to merge the duplicated indices. The nonzero vector s takes values 1 (for boundary edges) or 2 (for interior edges). We then use `find` to return the nonzero indices which forms the `edge` set. We can also find the boundary edges using the subset of indices pair corresponding to the nonzero value 1. Note that we switch the order of `(i,j)` in line 3 to sort the edge set row-wise since the output of `find(sparse)` is sorted column-wise.

To construct `edge` matrix only, the above 3 lines code can be further simplified to one line:

```
edge = unique(sort([elem(:,[1,2]);elem(:,[1,3]);elem(:,[2,3])],2),'rows');
```

The `unique` function provides more functionality which we shall explore more later. However, numerical test shows that the running time of `unique` is around 3 times of the combination `find(sparse)`.

Now we have three types of simplices for a 2-D simplicial complex:

0-simplex: `{1,2,...,N}`; 1-simplex: `edge`; 2-simplex: `elem`.

We shall discuss data structure to efficiently traverse in these simplices. These data structure use mainly the combinatorial property of a mesh, i.e., using the matrix `elem`. We do use some geometric properties of the 2-D planar triangulation. For example, we assume each edge is shared by at most two triangles, which may not hold for general abstract simplicial complex.

Following [?], we shall use the name convention `a2b` to represent the link form `a` to `b`. This link is usually the map from the local index set to the global index set. Throught out this paper, we denote the number of `node`, `elem`, and `edge` by

```
N = size(node,1); NT = size(elem,1); NE = size(edge,1);
```

`node` and `elem`. The `elem` matrix, by the definition, is a link from triangles to vertices, i.e., `elem` is `elem2node`. The link from vertices to triangles, namely given a vertex v, to find all triangles containing v, is stored in the sparse matrix:

```
t2v = sparse([1:NT,1:NT,1:NT], elem, 1, NT, N);
```

The `NT`$\times$ `N` matrix `t2v` is the incidence matrix between triangles and vertices. `t2v(t,i)=1` means the i-th node is a vertex of triangle t. If we look at `t2v` column-wise, the nonzero in the i-th column of `t2v(:,i)` will give all triangles containing the i-th node. Since sparse matrix is stored column-wise, the star of the i-th node can be efficiently found by

```
nodeStar = find(t2v(:,i));
```

t2v

	1	2	3	4	5	6	7	8
1	1	1	0	0	0	0	0	1
2	0	1	1	0	0	0	0	1
3	0	0	1	0	1	0	0	1
4	0	0	1	1	1	0	0	0
5	0	0	0	0	0	1	1	1
6	0	0	0	0	1	1	0	1

neighbor

	1	2	3
1	2	1	1
2	1	2	3
3	4	6	2
4	3	4	4
5	6	5	5
6	5	3	6

TABLE 1. `t2v` and `neighbor` matrices for the L-shape domain in Figure 2.

The cardinality of the node star, called *valence*, can be computed by the `accumarray` command. The following one line code

```
valence = accumarray(elem(:),ones(3*NT,1),[N 1]);
```

is equivalent to the double loop:

```
1 for t=1:NT
2     for i=1:3
3         valence(elem(t,i)) = valence(elem(t,i))+1;
4     end
5 end
```

When `NT` is big, the `for t=1:NT` loop is not efficient in MATLAB. As we mentioned early, `sparse` and `accumarray` are two most commonly used commands to replace the `for` loop.

`node` and `edge`. The `edge` matrix, by the definition, is a link from edges to vertices. Sometimes we know only vertices of an edge, say v_i, v_j, and want to find the edge using these two nodes. Namely an index map from $(v_i, v_j) \to k$ such that `edge(k,:)=[vi vj]` or `[vj vi]`. We shall construct such mapping by the sparse matrix

```
node2edge = sparse(edge(:,[1,2]),edge(:,[2,1]),[1:NE,1:NE],N,N);
```

Here we repeat the `edge` matrix with the reverse order in the indices set to allow `i<j` or `j<i` such that if `[i,j]=edge(k,:)` then `node2edge(i,j)=node2edge(j,i)=k`. Thus `node2edge` is a symmetric matrix.

There is another way to construct the link `node` $\to$ `edge` using the product of sparse matrices. Let us introduce the incidence matrix between edges and vertices as

```
e2v = sparse([1:NE,1:NE],[edge(:,1);edge(:,2)],1,NE,N);
```

The sparse matrix `e2v` is of dimension `NE`×`N` such that `e2v(e,v)=1` if `v` is a vertex of `e`. Then `e2v(:,i)` or `e2v(:,j)` contains all edges using the vertex `i` or `j`, respectively. The intersection of `e2v(:,i)`∩`e2v(:,j)` is the edge using `i,j` as two nodes, which can be found by

```
find(e2v(:,i).*e2v(:,j));
```

`edge` and `elem`. The edge matrix is constructed using element matrix. But there is no direct link between edges and triangles. One indirect link is through the path `elem` $\to$ `node` $\to$ `edge`. For example, `node2edge(elem(t,2),elem(t,3))` will give the index of the edge composed by the second and third vertices of the triangle `t`.

Since the access of a sparse matrix is not efficient especially in a large loop, we shall form a direct link `elem` $\to$ `edge`. We label three edges of a triangle such that the i-th edge is opposite to the i-th vertex. We define the matrix `elem2edge` as the map of local index of edges in each triangle to its

global index. The following three lines code will construct `elem2edge` using more output from `unique` function.

```
1 totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
2 [edge, i2, j] = unique(totalEdge,'rows');
3 elem2edge = reshape(j,NT,3);
```

Line 1 collects all edges element-wise. The size of `totalEdge` is thus `3NT×2`. By the construction, there is a natural index mapping from `totalEdge` to `elem`. In line 2, we apply `unique` function to obtain the edge matrix. The output index vectors `i2` and `j` contain the index mapping between `edge` and `totalEdge`. Here `i2` is a `NE×1` vector to index the last (2-nd in our case) occurrence of each unique value in `totalEdge` such that `edge = totalEdge(i2,:)`, while `j` is a `3NT×1` vector such that `totalEdge = edge(j,:)`. (Try `help unique` in MATLAB to learn more examples.) Then using the natural index mapping from `totalEdge` to `elem`, we reshape the `3NT×1` vector `j` to a `NT×3` matrix which is `elem2edge`.

An alternative but more cost way to construct `elem2edge` using the product of sparse matrices will be discussed for 3-D mesh.

We then define a `NE×4` matrix `edge2elem` such that `edge2elem(k,1)` and `edge2elem(k,2)` are two triangles sharing the k-th edge for an interior edge. If the k-th edge is on the boundary, then we set `edge2elem(k,1) = edge2elem(k,2)`. Furthermore, we shall record the local indices in `edge2elem(k,3:4)` such that `elem2edge(edge2elem(k,1),edge2elem(k,3))=k`. Similarly `edge2elem(k,4)` is the local index of k-th edge in `edge2elem(k,2)`.

To construct `edge2elem` matrix, we need to find out the index map from `edge` to `elem`. The following code is a continuation of the code constructing `elem2edge`.

```
1 i1(j(3*NT:-1:1)) = 3*NT:-1:1; i1=i1';
2 k1 = ceil(i1/NT); t1 = i1 - NT*(k1-1);
3 k2 = ceil(i2/NT); t2 = i2 - NT*(k2-1);
4 edge2elem = [t1,t2,k1,k2];
```

The code in line 1 use `j` to find the first occurrence of each unique edge in the `totalEdge`. In MATLAB, when assign values using an index vector with duplication, the value at the repeated index will be the last one assigned to this location. Obvious `j` contains duplication of edge indices. For example, `j(1)=j(2)=4` which means `totalEdge(1,:)=totalEdge(2,:)=edge(4,:)`. We reverse the order of `j` such that `i1(4)=1` which is the first occurrence.

Using the natural index mapping from `totalEdge` to `elem`, for an index `i` between `1:N`, the formula `k=ceil(i/NT)` computes the local index of i-th edge, and `t=i-NT*(k-1)` is the global index of the triangle which `totalEdge(i,:)` belongs to. The `edge2elem` is just composed by `t1,t2,k1` and `k2`.

`elem` and `elem`. We use the matrix `neighbor(1:NT,1:3)` to record the neighboring triangles for each triangle. By definition, `neighbor(t,i)` is opposite to the i-th vertex of the t-th triangle. If i is opposite to the boundary, then we set `neighbor(t,i)=t`. Using the index map between edge and elem, we can easily form the neighbor matrix by the following 2 lines code.

```
1 ix = (i1 ~= i2);
2 neighbor = accumarray([[t1(ix),k1(ix)];[t2,k2]],[t2(ix);t1],[NT 3]);
```

In line 1, to avoid the duplication in the index array, we find the index set of interior edges by noting that if `e` is a boundary edge, then `i1(e)=i2(e)`. Since `t1` and `t2` share the same edge, we form the `neighbor` matrix by using `t1,k1` and `t2,k2` as indices set and `t2,t1` as the value in line 2.

We summarize the construction of these auxiliary data structure in a subroutine `auxstructure.m`.

```
1 function [neighbor,elem2edge,edge2elem,edge,bdEdge]=auxstructure(elem)
2 totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
```

```
3  [edge, i2, j] = unique(totalEdge,'rows');
4  NT = size(elem,1);
5  elem2edge = reshape(j,NT,3);
6  i1(j(3*NT:-1:1)) = 3*NT:-1:1; i1=i1';
7  k1 = ceil(i1/NT); t1 = i1 - NT*(k1-1);
8  k2 = ceil(i2/NT); t2 = i2 - NT*(k2-1);
9  ix = (i1 ~= i2);
10 neighbor = accumarray([[t1(ix),k1(ix)];[t2,k2]],[t2(ix);t1],[NT 3]);
11 bdEdge = edge((i1 == i2),:);
12 edge2elem = [t1,t2,k1,k2];
```

1.3.3. *Auxiliary data structure for 3-D triangulation.* Most codes discussed for 2-D triangulations can be generalized to 3-D triangulations in a straightforward way. Due to the page limit, we pick up the following important data structures to explain in detail.

`elem` and `face`. The `face` matrix, which represents the 2-D simplex, can be generated by the `unique` function of all element-wise faces. The link `elem2face`, `faceStar`, and `neighbor` can be constructed similarly using the index map. We list `auxstructure3.m` below and skip the explanation.

```
1  function [neighbor,elem2face,face2elem,face,bdFace] = auxstructure3(elem)
2  face = [elem(:,[2 4 3]);elem(:,[1 3 4]);elem(:, [1 4 2]);elem(:, [1 2 3])];
3  [face, i2, j] = unique(sort(face,2),'rows');
4  NT = size(elem,1);
5  elem2face = reshape(j,NT,4);
6  i1(j(4*NT:-1:1)) = 4*NT:-1:1; i1 = i1';
7  k1 = ceil(i1/NT); t1 = i1 - NT*(k1-1);
8  k2 = ceil(i2/NT); t2 = i2 - NT*(k2-1);
9  ix = (i1 ~= i2);
10 neighbor = accumarray([[t1(ix),k1(ix)];[t2,k2]],[t2(ix);t1],[NT 4]);
11 bdFace = face((i1 == i2),:);
12 face2elem = [t1,t2,k1,k2];
```

`elem` and `edge`. The `edge` matrix can be generated using `find(sparse)` commands as in the 2-D case. The vector `edgeValence` is used to denote the number of elements sharing each edge.

```
1  totalEdge = sort([elem(t,[1 2]); elem(t,[1 3]); elem(t,[1 4]); ...
2                    elem(t,[2 3]); elem(t,[2 4]); elem(t,[3 4])],2);
3  [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
4  edge = [j,i]; edgeValence = s;
```

We then find the link `elem2edge` which is useful for the high order elements and edge element.

```
1  e2v = sparse([1:NE,1:NE],[edge(:,1);edge(:,2)],1,NE,N);
2  [i,j,s] = find(e2v(:,totalEdge(:,1)).*e2v(:,totalEdge(:,2)));
3  elem2edge = reshape(i,NT,6);
```

The first line generates the incidence matrix between edges and vertices. The sparse matrix `e2v` is of dimension `NE×N` such that `e2v(e,v)=1` if `v` is a vertex of `e`. Then `e2v(:,p1)` or `e2v(:,p2)` contains all edges using the vertex `p1` or `p2`, respectively. The intersection of `e2v(:,p1)∩e2v(:,p2)` is the edge using `p1` and `p2` as two vertices. The intersection is found by the Hadamard product, i.e., item wise product, of two sparse matrices `e2v(:,totalEdge(:,1))` and `e2v(:,totalEdge(:,2))`, and recorded in the index set `i`. In line 3 the `6NT×1` vector `i` is reshaped to a `NT×6` matrix which is what we want.

We now discuss the construction of `edgeStar`. This link from `edge` to `elem` is important since the 3-D local mesh refinement is always cutting edges. Unlike the 2-D case, we cannot use a `NE×2` dense

matrix for edgeStar since the number of elements sharing one edge varies a lot. Again we shall resort to the sparse matrix.

```
1 t2v = sparse([1:NT,1:NT,1:NT,1:NT], elem(1:NT,:), 1, NT, N);
2 nodeStar1 = t2v(1:NT,edge(:,1));
3 nodeStar2 = t2v(1:NT,edge(:,2));
4 edgeStar = nodeStar1.*nodeStar2;
```

The elements containing an edge is characterized as the intersection two stars of the ending nodes of this edge. The first line generates the incidence matrix t2v. Line 2 and 3 extract columns from t2v. The intersection is found by the Hadamard product of two sparse matrix nodeStar1 and nodeStar2. The resulting sparse matrix edgeStar is a NT×NE sparse matrix and find(edgeStar(:,i)) will give the element indices containing the i-th edge.

In the construction of elem2edge and edgeStar, we use Hadamard product of sparse matrices to find the quantity associated two index sets. This technique is crucial in 3-D refinement.