iFEM: AN INNOVATIVE FINITE ELEMENT METHOD PACKAGE IN MATLAB

LONG CHEN

ABSTRACT. Sparse matrixlization, an innovative programming style for MATLAB, is introduced and used to develop an efficient software package, iFEM, on adaptive finite element methods. In this novel coding style, the sparse matrix and its operation is used extensively in the data structure and algorithms. Our main algorithms are written in one page long with compact data structure following the style "Ten digit, five seconds, and one page" proposed by Trefethen. The resulting code is simple, readable, and efficient. A unique strength of iFEM is the ability to perform three dimensional local mesh refinement and two dimensional mesh coarsening which are not available in existing MATLAB packages. Numerical examples indicate that iFEM can solve problems with size 10^5 unknowns in few seconds in a standard laptop. iFEM can let researchers considerably reduce development time than traditional programming methods.

1. INTRODUCTION

Finite element method (FEM) is a powerful and popular numerical method on solving partial differential equations (PDEs), with flexibility in dealing with complex geometric domains and various boundary conditions. MATLAB (Matrix Laboratory) is a powerful and popular software platform using matrix-based script language for scientific and engineering calculations. This paper is on the development of an finite element method package, with emphasis on adaptive finite element method (AFEM) through local mesh refinement, in MATLAB using an innovative programming style: sparse matrixlization. In this novel coding style, to make use of the unique strength of MATLAB on fast matrix operations, the sparse matrix and its operation is used extensively in the data structure and algorithms. iFEM, the resulting package, is a good balance between simplicity, readability, and efficiency. It will benefit not only education but also future research and algorithm development on finite element methods.

Finite element methods first decompose the domain into a grid (also indicated by mesh or triangulation) consisting of small elements. A family of grids are used to construct appropriate finite dimensional spaces. Then an appropriate form (so-called weak form) of the original equation is restricted to those finite dimensional spaces to get a set of algebraic equations. Solving these algebraic equations will give approximated solutions of original PDEs within certain accuracy.

A natural approach to improve the accuracy is to divide each element into small elements which is known as uniform refinement. However, uniform refinement will dramatically increase the computation effort including the physical memory as well as CPU time since the number of unknowns grows exponentially.

Intuitively only elements in the region where the solution changes dramatically need to be divided. Adaptive finite element method is such a methodology that adjust the element size according to the behavior of the solution automatically and systematically. Comparing with the uniform refinement, adaptive finite element methods are more preferred to locally increase mesh densities in the regions of interest, thus saving the computer resources. In this approach, the relation between accuracy and computational labor is optimized.

Date: November 15, 2008.

2000 *Mathematics Subject Classification.* 68N15, 65N30, 65M60.

Key words and phrases. Matlab program, adaptive Finite Element Method, sparse matrix.

The author was supported in part by NSF Grant DMS-0811272, and in part by NIH Grant P50GM76516 and R01GM75309.

Because of its wide application, the finite element method courses are usually offered to engineering and mathematics students in colleges. Many literature, see, for example, the books [20, 13], are devoted to the theoretical foundation on finite element methods. However, the programming of finite element method is not straightforward. Coding adaptive finite element methods requires sophisticated data structure on the grids and could be very time-consuming. Therefore it is necessary to have a software package with readable implementation of the basic components of adaptive finite element methods. iFEM is such a package implemented in MATLAB. It contains robust, efficient, and easy-following codes for the main building blocks of adaptive finite element methods.

We choose MATLAB because of its simplicity, readability, and popularity. MATLAB is a high-level programming language. Typically one line MATLAB code can replace ten lines of Fortran or C code. The matrix-based script is expressiveness and very close to the operator-based description of algorithms. MATLAB supports a range of operating systems and processor architectures, providing portability and flexibilty. Additionally, MATLAB provides its users with rich graphics capabilities for visualization. Today MATLAB has emerged as one of the predominant languages of education and technical computing.

These merits of using MATLAB in the scientific computing are best summarized in the beautiful little book "Spectral methods in MATLAB" [60] by Trefethen:

A new era in scientific computing has been ushered in by the development of MATLAB. One can now present advanced numerical algorithms and solutions of nontrivial problems in complete detail with great brevity, covering more applied mathematics in a few pages that would have been imaginable a few years ago. By sacrificing sometimes (not always!) a certain factor in machine efficiency compared with lower-level languages such as Fortran or C, one obtains with MATLAB a remarkable human efficiency – an ability to modify a program and try something new, then something new again, with unprecedented ease.

All this convenience came at a cost of performance. To be interactive, MATLAB is an interpret language. Indeed a common misperception is "MATLAB is slow for large finite element problems" [22]. This problem is typically due to an incorrect usage of sparse matrix as explained below.

The matrix in the algebraic equation obtained by FEM is a sparse matrix. Namely although the $N \times N$ matrix is of size N^2, there are only cN nonzero entries with a small constant c independent of N. Sparse matrix is the corresponding data structure to take advantage of this sparsity which makes the simulation of large systems possible. The basic idea of sparse matrix is to use a single array to store all nonzero entries and two additional integer arrays to store the location of nonzero entries.

The accessing and manipulating sparse matrices one element at a time requires searching the index arrays to find such nonzero entry. It takes time at least proportional to the logarithm of the length of the column of the matrix [29]. If the sparse pattern is changed, for example, inserting or removing a nonzero, it may require extensive data movement to reform the index and nonzero value arrays. On the other hand, since MATLAB is an interpret language, each line is compiled when it is going to be executed. If a loop or subroutine caused certain lines to be executed multiple times, they would be recompiled every time.

Therefore manipulating a sparse matrix element-by-element in a large `for` loop in MATLAB would quickly add significant overhead and slow down the performance. Unfortunately the straightforward implementation of main components of AFEM typically involves updating sparse matrices in a large loop over all elements. This is the main reason why most MATLAB implementation of finite element methods are slow for number of unknowns larger than thousands.

Therefore the development of an efficient MATLAB package on AFEM is not a simple translation of the code from existing packages using other low-level programming languages. This difference is not fully noticed in the existing implementation of finite element methods using MATLAB; See, for example, the books [49, 34, 35] and articles [30, 19, 14, 6, 3, 26, 10, 2]. Codes in some of these work are still written in the low-level fashion. It is simply a translation of other low-level languages, such

as Fortran or C, to make use of the easy access of MATLAB to public. Some of them aims to short implementation of algorithms for the education purpose. In a word, they are simple and readable but not efficient. [1]

We shall gain the efficiency by an innovative programming style: sparse matrixlization. We shall reformulate algorithms on AFEM in terms of matrix operations. To maintain the optimal complexity both in time and space, we shall use sparse matrix in most places. With such a methodology, we can make use of fast matrix operations build in the MATLAB. Our numerical examples indicate that iFEM can solve 2-D problems or 3-D problems with size 10^5 unknowns in 3 seconds or 8 seconds, respectively, in a standard laptop. Researchers then can easily conduct research and spend less effort on programming.

A unique strength of iFEM is the ability to perform three dimensional local mesh refinement and two dimensional mesh coarsening which are not available in existing MATLAB packages. Note that the three dimensional local mesh refinement is not easy to implement due to the complicated geometry and sophisticated data structure. The sparse matrixlization presents an innovative way for the elimination of hanging nodes and make an efficient implementation of three dimensional local refinement possible. On the coarsening, a unique feature is that only the current mesh instead of the whole refinement history is required. The algorithm can automatically extract a tree structure from the current grid.

Besides the efficiency, we still maintain the simplicity and readability. In iFEM, our main algorithms are written in one page long with compact data structure following the style "Ten digit, five seconds, and one page" proposed by Trefethen [61]. User can easily get overview and insight on the algorithms implemented, which is impossible to obtain when dealing with closed black-box routines such as the PDEtool box in MATLAB or more advanced commercial package FEMLAB.

We should mention that sparse matrixlization belongs to a more general coding style – vectorization. In the setting of MATLAB programming, vectorization can be understood as a way to replace `for` loops by matrix operations or other fast builtin functions. See Code Vectorization Guide at the MathWorks web page for more tools and techniques on the code vectorization. Sparse matrixlization is a vectorization technique tailored to adaptive finite element methods. The name sparse matrixlization is used to emphasis the extensive usage of sparse matrix for the data structures and algorithms.

To conclude the introduction, we present the layout of this paper. In Section 2, we shall discuss the data structure of sparse matrices and commands in MATLAB to generate and manipulate sparse matrices. In Section 3, we shall introduce triangulations and discuss efficient ways to construct data structures for geometric relations using sparse matrixlization. In Section 4, we shall improve the standard but not efficient assembling procedure of stiffness matrix to an efficient way. In Section 5, we shall discuss implementation of bisection methods in both two and three dimensions. In Section 6, we shall discuss the coarsening of bisection grids in two dimensions. In Section 7, we present a typical loop of AFEM and present numerical examples using iFEM to illustrate the efficiency of our package. In the last section, we shall summarize and present future working projects.

2. Sparse matrix in MATLAB

In this section, we shall explain basics on sparse matrix and corresponding operations in MATLAB, which will be used extensively later. The content presented here is mostly based on Gilbert, Moler and Schereiber [29] and is included here for the convenience of readers.

Sparse matrix is a data structure to take advantage of the sparsity of a matrix. Sparse matrix algorithms require less computational time by avoiding operations on zero entries and sparse matrix data structures require less computer memory by not storing many zero entries. We refer to books [44, 25, 23] for detailed description on sparse matrix data structure and [54] for a quick introduction on popular data

[1] Very recently Funken, Praetorius, and Wissgott [28] provide an efficient implementation of adaptive P1 finite element method in two dimensions in MATLAB. Our package is developed independently and includes adaptive finite element in three dimensions which is much harder than that in two dimensions.

structures of sparse matrix. In particular, the sparse matrix data structure and operations has been added to MATLAB by Gilbert, Moler and Schereiber and documented in [29].

2.1. **Storage scheme.** There are different types of data structures for the sparse matrix. All of them share the same basic idea: use a single array to store all nonzero entries and two additional integer arrays to store the indices of nonzero entries.

A natural scheme, known as *coordinate format*, is to store both the row and column indices. In the sequel, we suppose A is a $m \times n$ matrix containing only nnz nonzero elements. Let us look at the following simple example:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 4 \\ 0 & 0 & 0 \\ 0 & 9 & 0 \end{bmatrix}, \quad i = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 2 \end{bmatrix}, \quad j = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 3 \end{bmatrix}, \quad s = \begin{bmatrix} 1 \\ 2 \\ 9 \\ 4 \end{bmatrix}. \tag{1}$$

In this example, i vector stores row indices of non-zeros, j column indices, and s the value of non-zeros. All three vectors have the same length nnz. The two indices vectors i and j contains redundant information. We can compress the column index vector j to a column pointer vector with length $n+1$. The value $j(k)$ is the pointer to the beginning of k-th column in the vector of i and s, and $j(n+1) = nnz+1$. This scheme is known as *Compressed Sparse Column (CSC)* scheme and is used in MATLAB sparse matrices package. For example, in CSC formate, the vector to store the column pointer will be $j = [1\ 2\ 4\ 5]^t$. Comparing with coordinate formate, CSC formate saves storage for $nnz - n - 1$ integers which could be nonnegligilble when the number of nonzero is much larger than that of the column. In CSC formate it is efficient to extract a column of a sparse matrix. For example, the k-th column of a sparse matrix can be build from the index vector i and the value vector s ranging from $j(k)$ to $j(k+1)-1$. There is no need of searching index arrays. An algorithm that builds up a sparse matrix one column at a time can be also implemented efficiently [29].

Remark 2.1. CSC is an internal representation of sparse matrices in MATLAB. For user convenience, the coordinate scheme is presented as the interface. This allows users to create and decompose sparse matrices in a more straightforward way.

Comparing with the dense matrix, the sparse matrix lost the direct relation between the index `(i,j)` and the physical location to save the value `A(i,j)`. The accessing and manipulating matrices one element at a time requires the searching of the index vectors to find such nonzero entry. It takes time at least proportional to the logarithm of the length of the column; inserting or removing a nonzero may require extensive data movement [29]. Therefore, *do not manipulate a sparse matrix element-by-element in a `for` loop in MATLAB.*

Due to the lost of the link between the index and the value of entries, the operations on sparse matrices is delicate. One needs to write subroutines for standard matrix operations: multiplication of a matrix and a vector, addition of two sparse matrices, and transpose of sparse matrices etc. Since some operations will change the sparse pattern, typically there is a priori loop to set up the nonzero pattern of the resulting sparse matrix. Good sparse matrix algorithms should follow the "time is proportional to flops" rule [29]: *The time required for a sparse matrix operation should be proportional to the number of arithmetic operations on nonzero quantities*. The sparse package in MATLAB follows this rule; See [29] for details.

2.2. **Create and decompose sparse matrix.** To create a sparse matrix, we first form i, j and s vectors, i.e., a list of nonzero entries and their indices, and then call the function `sparse` using i, j, s as input. Several alternative forms of `sparse` (with more than one argument) allow this. The most commonly used one is

```
A = sparse(i,j,s,m,n).
```

This call generates an $m \times n$ sparse matrix, using $[i, j, s]$ as the coordinate formate. The first three arguments all have the same length. However, the indices in these three vectors need not be given in any particular order and could have duplications. If a pair of indices occurs more than once in i and j, `sparse` adds the corresponding values of s together. This nice summation property is very useful for finite element computation.

The function `[i,j,s]=find(A)` is the inverse of `sparse` function. It will extract the nonzero elements together with their indices. The indices set (i, j) are sorted in column major order and thus the nonzero `A(i,j)` is sorted in lexicographic order of `(j,i)` not `(i,j)`. See the example in (1).

Remark 2.2. There is a similar command `accumarray` to create a dense matrix A from indices and values. It is slightly different from `sparse`. We need to pair `[i j]` to form a subscript vector. So is the dimension `[m n]`. Since the accessing of a single element in a dense matrix is much faster than that in a sparse matrix, when m or n is small, say $n = 1$, it is better to use `accumarray` instead of `sparse`. A most commonly used command is

```
accumarray([i j], s, [m n]).
```

3. 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.

3.1. 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 $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_{d+1}$,

$$\tau := \{\boldsymbol{x} = \sum_{i=1}^{d+1} \lambda_i \boldsymbol{x}_i \,|\, 0 \leq \lambda_i \leq 1, i = 1 : d+1, \sum_{i=1}^{d+1} \lambda_i = 1\} \tag{2}$$

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

$$\lambda_i(\boldsymbol{x}) = |\tau_i(\boldsymbol{x})|/|\tau|, \tag{3}$$

where $|\cdot|$ is the Lebesgure measure in $\mathbb{R}^d$, namely area in two dimensions and volume in three dimensions. From (3), 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

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

Remark 3.1. 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.

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

$$\max_{\tau\in\mathcal{T}} \frac{\operatorname{diam}(\tau)^d}{|\tau|} \leq c_0, \quad \forall \mathcal{T} \in \mathscr{T}, \tag{4}$$

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 3.2. In addition to (4), if there exists a constant c_1 such that

$$\frac{\max_{\tau\in\mathcal{T}} |\tau|}{\min_{\tau\in\mathcal{T}} |\tau|} \leq c_1, \quad \forall \mathcal{T} \in \mathscr{T}, \tag{5}$$

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

3.2. **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* [48]. 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 $\mathbb{R}^3$.

3.3. **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.

3.3.1. *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.

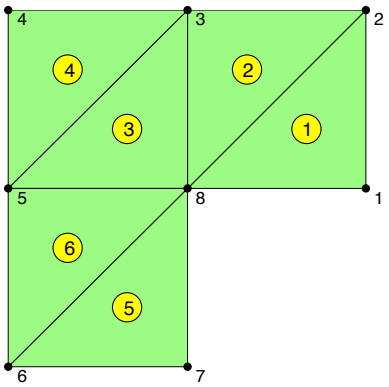
The matrix `node` gives the geometric realization of the simplicial complex. For example, for a 2-D triangulation, `node(k,1:2)` contain x- and y-coordinates of the k-th node. We shall always order the vertices of a simplex such that the signed volume is positive. That is in 2-D, three vertices of a 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).

3.3.2. *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.

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



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

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

1 2

node

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

1 2 3

elem

(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(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],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 [6], 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));
```

	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

t2v

	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

neighbor

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:

```
for t=1:NT
    for i=1:3
        valence(elem(t,i)) = valence(elem(t,i))+1;
    end
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`$\times$`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)`$\cap$`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);
```

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

3.3.3. *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.

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

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

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

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

The first line generates the incidence matrix between edges and vertices. The sparse matrix `e2v` is of dimension `NE`$\times$`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)`$\cap$`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`$\times$`1` vector `i` is reshaped to a `NT`$\times$`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`$\times$`2` dense

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

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

4. Assembling of Matrix Equation

In this section, we discuss how to obtain a matrix equation for the linear finite element method of solving the Poisson equation

$$-\Delta u = f \text{ in } \Omega, \qquad u = g_D \text{ on } \Gamma_D, \qquad \nabla u \cdot n = g_N \text{ on } \Gamma_N, \tag{6}$$

where $\partial\Omega = \Gamma_D \cup \Gamma_N$ and $\Gamma_D \cap \Gamma_N = \varnothing$. We assume Γ_D is closed and Γ_N open.

Denoted by $H^1_{g,D}(\Omega) = \{v \in L^2(\Omega), \nabla v \in L^2(\Omega) \text{ and } v|_{\Gamma_D} = g_D\}$. Using integration by parts, the weak form of the Poisson equation (6) is to find $u \in H^1_{g,D}(\Omega)$ such that for all $v \in H^1_{0_D}$

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v \,\mathrm{dxdy} = \int_\Omega f v \,\mathrm{dxdy} + \int_{\Gamma_N} g_N v \, dS. \tag{7}$$

Let $\mathcal{T}$ be a triangulation of Ω. We define the linear finite element space on $\mathcal{T}$ as

$$\mathbb{V}_\mathcal{T} = \{v \in C(\bar{\Omega}) : v|_\tau \in \mathcal{P}_1(\tau), \forall \tau \in \mathcal{T}\}.$$

For each vertex v_i of $\mathcal{T}$, let ϕ_i be the piecewise linear function such that $\phi_i(v_i) = 1$ and $\phi_i(v_j) = 0$ if $j \neq i$. Then it is easy to see $\mathbb{V}_\mathcal{T}$ is spanned by $\{\phi_i\}_{i=1}^N$. The linear finite element method for solving (6) is to find $u \in \mathbb{V}_\mathcal{T} \cap H^1_{g,D}(\Omega)$ such that (7) holds for all $v \in \mathbb{V}_\mathcal{T} \cap H^1_{0,D}(\Omega)$.

We shall discuss an efficient way to obtain the algebraic equation. It is an improved version, for the sake of efficiency, of that in the paper [2] by Alberty, Carstensen, and Funken. Recently in [28], Funken, Praetorious and Wissgott improved the assembling process in [2] by vectorization. Their approach use a special formula for two dimensional Poisson equation. Our approach presented here works for both two and three dimensions. The two dimensional case is already mentioned in our recent work [15, 17]. We also note that a large loop is avoid in the assembling procedure in [27] and [21].

The following subroutines will compute the linear finite element approximation in two and three dimensions:

```
u = Poisson(node,elem,bdEdge,@f,@g_D,@g_N);
u = Poisson3(node,elem,bdFace,@f,@g_D,@g_N);
```

In the input, the triangulation of the domain Ω is given by `node` and `elem`. The boundary Γ_D and Γ_N is build into `bdEdge` or `bdFace`. The data is given by handles of functions `@f,@g_D,@g_N`. The input `bdEdge` or `bdFace` can be omitted if $\Gamma_D = \varnothing$ or $\Gamma_N = \varnothing$, i.e., non-mixed boundary conditions. For example, the following codes

```
u = Poisson(node,elem,[],f,g_D,[])        % Dirichlet boundary condition
u = Poisson(node,elem,[],f,[],g_N)        % Neumann boundary condition
```

will compute the solution to the Poisson equation with Dirichlet or Neumann boundary condition in the whole boundary $\partial\Omega$. The boundary will be found by `find(sparse)` commands. See also Section 4.3. This provides more flexibility in mesh adaptation without tracking boundary edges or faces.

4.1. **Assembling the stiffness matrix.** For any function $v \in \mathbb{V}_{\mathcal{T}}$, there is a unique representation: $v = \sum_{i=1}^{N} v_i\phi_i$. We define an isomorphism $\mathbb{V}_{\mathcal{T}} \cong \mathbb{R}^N$ by

$$v = \sum_{i=1}^{N} v_i\phi_i \longleftrightarrow \boldsymbol{v} = (v_1, \cdots, v_N)^t, \tag{8}$$

and call $\boldsymbol{v}$ the vector representation of v (with respect to the basis $\{\phi_i\}_{i=1}^N$). We introduce the *stiffness matrix*

$$\boldsymbol{A} = (a_{ij})_{N\times N}, \text{ with } \quad a_{ij} = a(\phi_j, \phi_i).$$

In this subsection, we shall discuss how to form the matrix $\boldsymbol{A}$ efficiently in MATLAB.

4.1.1. *Standard assembling process.* By the definition, for $1 \le i, j \le N$,

$$a_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}\boldsymbol{x} = \sum_{\tau\in\mathcal{T}} \int_\tau \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}\boldsymbol{x}.$$

For each simplex τ, we define the local stiffness matrix $A^\tau = (a_{ij}^\tau)_{(d+1)\times(d+1)}$ as

$$a_{ij}^\tau = \int_\tau \nabla\lambda_j \cdot \nabla\lambda_i \,\mathrm{d}\boldsymbol{x}, \text{ for } 1 \le i, j \le d+1.$$

The computation of a_{ij} will be decomposed into the computation of local stiffness matrix, a_{ij}^τ and the summation over all elements. Note that the index set in a_{ij}^τ is local index while in a_{ij} it is global index. The assembling process is to distribute the quantity associated to the local index to that to the global index.

A standard procedure to compute the local stiffness matrix is to transfer the computation to a reference simplex through an affine map. We include the two dimensional case here for the comparison and completeness.

We call the triangle $\hat{\tau}$ spanned by $\hat{v}_1 = (1, 0), \hat{v}_2 = (0, 1)$ and $\hat{v}_3 = (0, 0)$ *a reference triangle* and use $\hat{\boldsymbol{x}} = (\hat{x}, \hat{y})^t$ for the vector in that coordinate. For any $\tau \in \mathcal{T}$, we treat it as the image of $\hat{\tau}$ under an affine map: $F : \hat{\tau} \to \tau$. One of such affine map is to match the local indices of three vertices, i.e., $F(\hat{v}_i) = v_i, i = 1, 2, 3$:

$$F(\hat{\boldsymbol{x}}) = B^t(\hat{\boldsymbol{x}}) + c,$$

where

$$B = \begin{bmatrix} x_1 - x_3 & y_1 - y_3 \\ x_2 - x_3 & y_2 - y_3 \end{bmatrix}, \text{ and } c = (x_3, y_3)^t.$$

We define $\hat{u}(\hat{\boldsymbol{x}}) = u(F(\hat{\boldsymbol{x}}))$. Then $\hat{\nabla}\hat{u} = B\nabla u$ and $\mathrm{dxdy} = |\det(B)|\mathrm{d\hat{x}d\hat{y}}$. We change the computation of the integral in τ to $\hat{\tau}$ by

$$\begin{aligned}\int_\tau \nabla\lambda_i \cdot \nabla\lambda_j \mathrm{dxdy} &= \int_{\hat{\tau}} (B^{-1}\hat{\nabla}\hat{\lambda}_i) \cdot (B^{-1}\hat{\nabla}\hat{\lambda}_j) |\det(B)| \mathrm{d\hat{x}d\hat{y}} \\ &= \frac{1}{2}|\det(B)| (B^{-1}\hat{\nabla}\hat{\lambda}_i) \cdot (B^{-1}\hat{\nabla}\hat{\lambda}_j).\end{aligned}$$

In the reference triangle, $\hat{\lambda}_1 = \hat{x}, \hat{\lambda}_2 = \hat{y}$ and $\hat{\lambda}_3 = 1 - \hat{x} - \hat{y}$. Thus

$$\hat{\nabla}\hat{\lambda}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \hat{\nabla}\hat{\lambda}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \text{ and } \hat{\nabla}\hat{\lambda}_3 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

We then end with the following subroutine to compute the local stiffness matrix in one triangle τ.

```
1  function [At,area] = localstiffness(p)
2  At = zeros(3,3);
3  B = [p(1,:)-p(3,:); p(2,:)-p(3,:)];
4  G = [[1,0]',[0,1]',[-1,-1]'];
5  area = 0.5*abs(det(B));
6  for i = 1:3
7     for j = 1:3
8        At(i,j) = area*((B\G(:,i))'*(B\G(:,j)));
9     end
10 end
```

The advantage of this approach is that by modifying the subroutine `localstiffness`, one can easily adapt to new elements and new equations.

To get the global stiffness matrix, we apply a `for` loop of all elements and distribute element-wise quantity to node-wise quantity. A straightforward MATLAB code is like

```
1  function A = assemblingstandard(node,elem)
2  N=size(node,1); NT=size(elem,1);
3  A=zeros(N,N); %A = sparse(N,N);
4  for t=1:NT
5      At=locatstiffness(node(elem(t,:),:));
6      for i=1:3
7          for j=1:3
8              A(elem(t,i),elem(t,j))=A(elem(t,i),elem(t,j))+At(i,j);
9          end
10     end
11 end
```

The above code is correct but not efficient. There are two reasons for the slow performance:

(1) The stiffness matrix `A` is a full matrix which needs $O(N^2)$ storage. It will be out of memory quickly when N is big (e.g., $N = 10^4$). Sparse matrix should be used for the sake of memory. Nothing wrong with MATLAB. Coding in other languages also need to use sparse matrix data structure.

(2) There is a large `for` loops with size of the number of elements. This can quickly add significant overhead when `NT` is large since each line in the loop will be interpreted in each iteration. This is a weak point of MATLAB. Vectorization should be applied for the sake of efficiency.

4.1.2. *Assembling using sparse matrix.* A straightforward modification of using sparse matrix is to replace the line 3 in the subroutine `assemblingstandard` by `A=sparse(N,N)`. Then MATLAB will use sparse matrix to store `A` and thus we solve the problem of storage. Thanks to the sparse matrix package in MATLAB, we can still access and operate the sparse `A` use standard format and thus keep other lines of code unchanged.

However, as we mentioned before, updating one single element of a sparse matrix in a large loop is very expensive since the nonzero indices and values vectors will be reformed and a large of data movement is involved. Therefore the code in line 8 of `assemblingstandard` will dominate the whole computation procedure. In this example, numerical experiments show that the subroutine `assemblingstandard` will take $O(N^2)$ time.

We should use `sparse` command to form the sparse matrix. The following subroutine is suggested by T. Davis [22].

```
1  function A = assemblingsparse(node,elem)
2  N = size(node,1); NT = size(elem,1);
3  i = zeros(9*NT,1); j = zeros(9*NT,1); s = zeros(9*NT,1);
```

```
index = 0;
for t = 1:NT
    At = localstiffness(node(elem(t,:),:));
    for ti = 1:3
        for tj = 1:3
            index = index + 1;
            i(index) = elem(t,ti);
            j(index) = elem(t,tj);
            s(index) = At(ti,tj);
        end
    end
end
A = sparse(i, j, s, N, N);
```

In the subroutine `assemblingsparse`, we first record a list of index and nonzero entries in the loop and use build-in function `sparse` to form the sparse matrix outside the loop. By doing in this way, we avoid updating a sparse matrix inside a large loop. The subroutine `assemblingsparse` is much faster than `assemblingstandard`. Numerical test shows the computational complexity is improved from $O(N^2)$ to $O(N \log N)$.

4.1.3. *Vectorization of assembling.* There is still a large loop in the subroutine `aseemblingsparse`. We shall use the vectorization technique to avoid the outer large `for` loop.

Given a d-simplex τ, recall that the barycentric coordinates $\lambda_j(x), j = 1, \cdots, d+1$ are linear functions. Let `k=elem(t,j)`, i.e, the j-th vertices of a simplex τ is the k-th vertex, then the hat basis function ϕ_k restricted to a simplex τ will coincide with the barycentric coordinate λ_j. Note that the index $j = 1, \cdots, d+1$ is the local index set for the vertices of τ, while $k = 1, \cdots, N$ is the global index set of all vertices in the triangulation.

We shall derive a geometric formula for $\nabla\lambda_i, i = 1, \cdots, d+1$. Let F_i denote the $(d-1)$-face of τ opposite to the ith-vertex. Since $\lambda_i(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in F_i$, and $\lambda_i(\boldsymbol{x})$ is an affine function of $\boldsymbol{x}$, the gradient $\nabla\lambda_i$ is a normal vector of the face F_i with magnitude $1/h_i$, where h_i is the distance from the vertex x_i to the face F_i. Using the relation $|\tau| = \frac{1}{d}|F_i|h_i$, we end with the following formula

$$\nabla\lambda_i = \frac{1}{d!\,|\tau|}\boldsymbol{n}_i, \tag{9}$$

where $\boldsymbol{n}_i$ is an *inward* normal vector of the face F_i with magnitude $\|\boldsymbol{n}_i\| = (d-1)!|F_i|$. Therefore

$$a_{ij}^{\tau} = \int_{\tau} \nabla\lambda_i \cdot \nabla\lambda_j \,\mathrm{dxdy} = \frac{1}{d!^2|\tau|}\boldsymbol{n}_i \cdot \boldsymbol{n}_j.$$

In 2-D, the scaled normal vector $\boldsymbol{n}_i$ can be easily computed by a rotation of the edge vector. For a triangle spanned by $\boldsymbol{x}_1, \boldsymbol{x}_2$ and $\boldsymbol{x}_3$, we define $\boldsymbol{l}_i = \boldsymbol{x}_{i+1} - \boldsymbol{x}_{i-1}$ where the subscript is 3-cyclic. For a vector $\boldsymbol{v} = (x, y)$, we denoted by $\boldsymbol{v}^{\perp} = (-y, x)$. Then $\boldsymbol{n}_i = \boldsymbol{l}_i^{\perp}$ and $\boldsymbol{n}_i \cdot \boldsymbol{n}_j = \boldsymbol{l}_i \cdot \boldsymbol{l}_j$. The edge vector $\boldsymbol{l}_i$ for all triangles can be computed as a matrix and used to compute the area of all triangles.

We then end with the following compact and efficient code for the assembling of stiffness matrix in two dimensions.

```
function A = assembling(node,elem)
N = size(node,1); NT = size(elem,1); A = sparse(N,N);
ve(:,:,1) = node(elem(:,3),:)-node(elem(:,2),:);
ve(:,:,2) = node(elem(:,1),:)-node(elem(:,3),:);
ve(:,:,3) = node(elem(:,2),:)-node(elem(:,1),:);
area = 0.5*abs(-ve(:,1,3).*ve(:,2,2)+ve(:,2,3).*ve(:,1,2));
for i = 1:3
    for j = 1:3
```

```
9           Aij = dot(ve(:,:,i),ve(:,:,j),2)./(4*area);;
10          A = A + sparse(elem(:,i),elem(:,j),Aij,N,N);
11      end
12  end
```

In 3-D, the scaled normal vector $\boldsymbol{n}_i$ can be computed by the cross product of two edge vectors. We list the code below and explain it briefly.

```
1   function A = assembling3(node,elem)
2   face = [elem(:,[2 4 3]);elem(:,[1 3 4]);elem(:, [1 4 2]);elem(:, [1 2 3])];
3   v12 = node(face(:,2),:)-node(face(:,1),:);
4   v13 = node(face(:,3),:)-node(face(:,1),:);
5   allNormal = cross(v12,v13,2);
6   normal = zeros(NT,3,4);
7   normal(1:NT,:,1) = allNormal(1:NT,:);
8   normal(1:NT,:,2) = allNormal(NT+1:2*NT,:);
9   normal(1:NT,:,3) = allNormal(2*NT+1:3*NT,:);
10  normal(1:NT,:,4) = allNormal(3*NT+1:4*NT,:);
11  v12 = v12(3*NT+1:4*NT,:); v13 = v13(3*NT+1:4*NT,:);
12  v14 = node(elem(:,4),:)-node(elem(:,1),:);
13  volume = dot(cross(v12,v13,2),v14,2)/6;
14  for i = 1:4
15      for j = 1:4
16          Aij = dot(normal(:,:,i),normal(:,:,j),2)./(36*volume);
17          A = A + sparse(elem(:,i),elem(:,j),Aij,N,N);
18      end
19  end
```

The code in line 2 will collect all faces of the tetrahedron mesh. So the `face` is of dimension `4NT×3`. For each face, we form two edge vectors `v12` and `v13`, and apply the cross product to obtain the scaled normal vector in `allNormal` matrix. The code in line 6-10 is to reshape the `4NT×3` normal vector to a `NT×3×4` matrix. Line 13 use the mix product of three edge vectors to compute the volume and line 14–19 is similar to 2-D case. The introduction of the scaled normal vector $\boldsymbol{n}_i$ simplify the implementation and enable us to vectorize the code.

Remark 4.1. The computation of the local stiffness matrix, i.e., the subroutine `localstiffness`, can be written in a concise way by avoiding the two inner loops; See [2, 28]. Similarly the small loop for $i, j = 1, \cdots, d+1$ can be avoid by reshape the vector. But we shall not preform the vectorization for these small loops since the gained efficiency is marginal and cannot compensate the reduction of readability.

4.2. **Right hand side.** We define the vector $\boldsymbol{f} = (f_1, \cdots, f_N)^t$ by $f_i = \int_\Omega f\phi_i$, where ϕ_i is the hat basis at the vertex v_i. For quasi-uniform meshes, all simplices are in the same size, while in adaptive finite element method, some elements with large mesh size could remain unchanged. Therefore, although the 1-point quadrature is adequate for linear element on quasi-uniform meshes, to reduce the error introduced by the numerical quadrature, we compute the load term $\int_\Omega f\phi_i$ by 3-points quadrature rule in 2-D and 4-points rule in 3-D.

We list the 2-D code below as an example to emphasis again that the command `accumarray` is used to avoid the slow `for` loop over all elements.

```
1   mid1 = (node(elem(:,2),:)+node(elem(:,3),:))/2;
2   mid2 = (node(elem(:,3),:)+node(elem(:,1),:))/2;
3   mid3 = (node(elem(:,1),:)+node(elem(:,2),:))/2;
4   bt1 = area.*(f(mid2)+f(mid3))/6;
5   bt2 = area.*(f(mid3)+f(mid1))/6;
```

```
bt3 = area.*(f(mid1)+f(mid2))/6;
b = accumarray(elem(:),[bt1;bt2;bt3],[N 1]);
```

4.3. **Boundary condition.** To build the boundary condition into the matrix equation, we first discuss the data structure to represent Dirichlet boundary Γ_D and Neumann boundary Γ_N.

We shall use `bdEdge(1:NT,1:3)` to indicate which edge of each element is on the boundary. The value is the type of boundary condition: 1 for first type, i.e., Dirichlet boundary edges, 2 for second type, i.e., Neumann boundary edges, and 0 for non-boundary, i.e., interior edges. For example, `bdEdge(t,:)=[1 0 2]` means, the edge opposite to `elem(t,1)` is a Dirichlet boundary face, the one to `elem(t,3)` is of Neumann type, and the other is an interior edge. The third type of boundary condition, i.e., Robin boundary condition, can be easily added into `bdEdge` but is not discussed in this paper. We can extract boundary edges from `bdEdge` using the following code:

```
totalEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
isBdEdge = reshape(bdEdge,3*NT,1);
Dirichlet = totalEdge((isBdEdge == 1),:);
Neumann = totalEdge((isBdEdge == 2),:);
```

Similarly in 3-D, we use `bdFace(1:NT,1:4)` to indicate which face of each element is on the boundary and extract different type of boundary faces in this way.

Remark 4.2. The matrix `bdFace` is sparse but we use dense matrix to store it. It would save storage if we save boundary edges or faces only. The current form is more convenient for the local refinement and coarsening since the update on the boundary can be easily done with the bisection of elements.

We do not save `bdFace` as a sparse matrix since updating sparse matrix is time consuming. We set up the type of `bdEdge` to `int8` to minimize the waste of spaces.

Remark 4.3. When only Dirichlet or Neumann boundary condition is posed, we do not have to store the boundary edges or faces. The boundary will be found by

```
[bdNode,bdEdge] = findboundary(elem)
[bdNode,bdFace] = findboundary3(elem)
```

for 2-D or 3-D domains, respectively, using `find(sparse)` commands.

The boundary condition is then treated in the same way as that in [2]. We list the code for 2-D case and briefly explain it for the completeness.

```
%-------------------- Dirichlet boundary conditions------------------------
isBdNode = false(N,1); isBdNode(Dirichlet) = true;
bdNode = find(isBdNode);
freeNode = find(~isBdNode);
u = zeros(N,1); u(bdNode) = g_D(node(bdNode,:));
b = b - A*u;
%-------------------- Neumann boundary conditions ------------------------
Nve = node(Neumann(:,1),:) - node(Neumann(:,2),:);
edgeLength = sqrt(sum(Nve.^2,2));
mid = (node(Neumann(:,1),:) + node(Neumann(:,2),:))/2;
b = b + accumarray([Neumann(:),ones(2*size(Neumann,1),1)], ...
                   repmat(edgeLength.*g_N(mid)/2,2,1),[N,1]);
```

Line 1-3 will find all Dirichlet boundary nodes. The Dirichlet boundary condition is posed by assign the function values at Dirichlet boundary nodes. Note that the vector `u` is initialized as zero vector. Therefore after line 5, the vector `u` will represent a function $u_D \in H_{g_D,\Gamma_D}$. Writing $u = \tilde{u} + u_D$, finding u is equivalent to finding $\tilde{u} \in \mathbb{V}_{\mathcal{T}} \cap H_0^1(\Omega)$ such that $a(\tilde{u},v) = (f,v) - a(u_D,v) + (g_N,v)_{\Gamma_N}$ for all $v \in \mathbb{V}_{\mathcal{T}} \cap H_0^1(\Omega)$. The modification of the right hand side $(f,v) - a(u_D,v)$ is realized by the

code `b=b-A*u` in line 6. The boundary integral involving the Neumann boundary part is computed in line 8–12. Note that the code is speed up using `accumarray`.

Since u_D and $\tilde{u}$ use disjoint nodes set, one vector `u` is used to represent both. The addition of $\tilde{u}+u_D$ is realized by assign values to different node sets of the same vector `u`. We have assigned the value to boundary nodes in line 5. We will compute $\tilde{u}$, i.e., the value at other nodes (denoted by `freeNode`), by

```
u(freeNode)=A(freeNode,freeNode)\b(freeNode).
```
(10)

Here `A\b` computes $A^{-1}b$ using MATLAB build int direct solver.

5. Bisection

In this section, we shall discuss bisection methods for the local mesh refinement and efficient implementations of newest vertex bisection in 2-D and longest edge bisection in 3-D. In short, the bisection method will divide one simplex into two children simplicies. Another class of mesh refinement method, known as regular refinement, which divide one simplex into 2^d children simplicies, will not be discussed in this paper.

5.1. Bisection methods.

In this subsection, we shall present an abstract definition of bisection methods and review existing bisection methods in 2-D and 3-D.

5.1.1. *Abstract definition.* Given a simplex τ, we shall choose one of its edges as *refinement edge*. A *labeled simplex* is a pair (τ, e) and a *labeled triangulation* is a set $(\mathcal{T},\mathcal{L}) := \{(\tau,e) : \tau \in \mathcal{T}\}$. Starting from an initial triangulation $\mathcal{T}_0$, a bisection method consists of:

(1) a rule to assign refinement edges for each element $\tau \in \mathcal{T}_0$;
(2) a rule to divide a simplex with a refinement edge into two children simplices;
(3) a rule to assign refinement edges to children simplices.

Rule 1 can be described by a mapping $\mathcal{T}_0 \to (\mathcal{T}_0,\mathcal{L})$ and called *initial labeling*. This rule is an essential ingredient of bisection methods. Once the initial labeling is given, the subsequent grids inherit labels according to Rule 3 such that the bisection can proceed. Mathematically, let τ_1 and τ_2 be two simplices such that $\tau_1 \cup \tau_2 = \tau$ is a simplex. Rule 2 and 3 can be described by a mapping

$$b_\tau : \{(\tau,e)\} \to \{(\tau_1,e_1),(\tau_2,e_2)\}.$$

For a labeled triangulation $(\mathcal{T},\mathcal{L})$, and a bisection b_τ for $\tau \in \mathcal{T}$, we define a formal addition

$$\mathcal{T} + b_\tau := \Big[(\mathcal{T},\mathcal{L})\backslash\{(\tau,e)\}\Big] \cup \{(\tau_1,e_1),(\tau_2,e_2)\}.$$

For a sequence of bisections $\mathcal{B} = (b_{\tau_1}, b_{\tau_2}, \cdots, b_{\tau_N})$, we define

$$\mathcal{T} + \mathcal{B} := ((\mathcal{T} + b_{\tau_1}) + b_{\tau_2}) + \cdots + b_{\tau_N}, \tag{11}$$

whenever the addition is well defined, i.e., τ_i should exists in the previous labeled triangulation. These additions are a convenient mathematical description of bisection methods on triangulations.

Given a labeled initial grid $\mathcal{T}_0$ and a bisection method, we define

$$\mathscr{F}(\mathcal{T}_0) = \{\mathcal{T} : \text{ there exists a bisection sequence } \mathcal{B} \text{ such that } \mathcal{T} = \mathcal{T}_0 + \mathcal{B}\},$$
$$\text{and} \qquad \mathscr{C}(\mathcal{T}_0) = \{\mathcal{T} \in \mathscr{F}(\mathcal{T}_0) : \mathcal{T} \text{ is conforming}\}.$$

Namely $\mathscr{F}(\mathcal{T}_0)$ contains all triangulations obtained from $\mathcal{T}_0$ using this bisection method. But a triangulation $\mathcal{T} \in \mathscr{F}(\mathcal{T}_0)$ could be non-conforming and thus we define $\mathscr{C}(\mathcal{T}_0)$ as a subset of $\mathscr{F}(\mathcal{T}_0)$ containing only conforming triangulations.

We then define a partial ordering in the set $\mathscr{F}(\mathcal{T}_0)$. We say $\mathcal{T}_2$ is a refinement of $\mathcal{T}_1$ and denoted by $\mathcal{T}_1 \le \mathcal{T}_2$ if there exists a bisection sequence $\mathcal{B}$ such that $\mathcal{T}_2 = \mathcal{T}_1 + \mathcal{B}$. Given a triangulation $\mathcal{T} \in \mathscr{F}(\mathcal{T}_0)$, if there exists a triangulation $\overline{\mathcal{T}} \in \mathscr{C}(\mathcal{T}_0)$ with minimal number of elements such that $\mathcal{T} \le \overline{\mathcal{T}}$, we call $\overline{\mathcal{T}}$ the *completion* of $\mathcal{T}$.

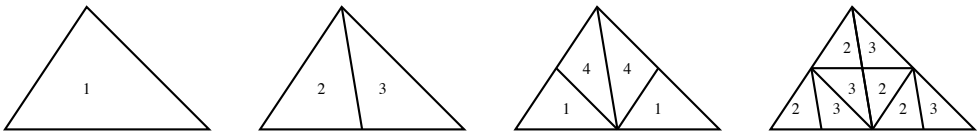


FIGURE 3. Four similarity classes of triangles generated by the newest vertex bisection

There are two main issues in designing a good bisection method.

(B1) Shape regularity: Prove $\mathscr{F}(\mathcal{T}_0)$ is shape regular.

(B2) Conformity: Construct an algorithm to find the completion $\overline{\mathcal{T}}$ for any $\mathcal{T} \in \mathscr{F}(\mathcal{T}_0)$.

An iterative algorithm of the completion is the following. Let M denote the set of elements to be bisected.

```
function completion(T,M)
while M is not empty
    bisect each element in M;
    let now M be the set of all non-conforming elements.
end
```

Since all additional bisection rather than the bisection of M is introduced for the conformity, the subroutine `completion(T,M)` will produce $\overline{\mathcal{T}}$. Therefore, to verify (B2), one needs only to show the subroutine `completion(T,M)` will terminate in finite steps.

5.1.2. *Existing bisection methods.* All existing bisection methods share the same Rule 2 described below. Given a labeled simplex (τ, e), the two children of τ are obtained by connecting the midpoint of e to other vertexes. The Rule 2 distinguish the bisection methods with another important class of local refinement method: regular refinement, which divides one simplex into 2^d children; See [8, 36, 11].

Different bisection methods differ in Rules 1 and 3. We shall divide them into two groups: longest edge bisection [47, 51, 52, 53, 50, 46, 45] and newest vertex bisection [57, 38, 40, 9, 33, 59, 4, 37]. The former make use of the geometric structure of the triangulation while the later mainly use the topological structure.

As the name *longest edge bisection* suggests, the refinement edge of a simplex is always its longest edge. This is used in both Rule 1 and 3. We treat the bisection methods based on the skeleton [46, 45] as a variant of longest edge bisection. The finite termination of the completion procedure `completion(T,M)` is obvious because when traversing from one simplex to another, one steps along paths of simplices with strictly longer longest edges. But (B1) is only proved for two dimensional triangulations [53]. Experiments show that the tetrahedra meshes produced by longest edge bisection satisfy (B1) in 3-D also, but it is an open problem to prove this fact.

Another class of bisection methods, the *newest vertex bisection* methods for two dimensional triangulations is to assign the edge opposite to the newest vertex of each child as its refinement edge. Once a initial labeling is assigned to $\mathcal{T}_0$, the refinement edges of all descendants of triangles in $\mathcal{T}_0$ determined by the combinatorial structure of the triangulation. It can be shown that all the descendants of a triangle in $\mathcal{T}_0$ fall into four similarity classes and hence (B1) holds. See Figure 5.1.2 for an illustration and the reference [57, 38] for a proof using graph theories.

To show (B2), a key observation is that no matter what the initial labeling is used, uniform bisection of a conforming triangulation even times always give a conforming triangulation. Let us introduce notation for uniform bisection by setting $D^0(\mathcal{T}) = \mathcal{T}, D(\mathcal{T}) = \mathcal{T} + \{b_\tau, \tau \in \mathcal{T}\}$, and $D^{k+1}(\mathcal{T}) = D(D^k(\mathcal{T}))$ for $k \geq 2$. For any conforming two dimensional triangulation $\mathcal{T}$, $D^2(\mathcal{T})$ will be conforming

although $D(\mathcal{T})$ may not. For example, uniform bisect the mesh in Figure 1 (b) using longest edge bisection leads to a non-conforming mesh. By the definition of completion

$$\overline{\mathcal{T}} \leq D^2(\mathcal{T}), \quad \forall \mathcal{T} \in \mathscr{F}(\mathcal{T}_0). \tag{12}$$

Since all additional bisection rather than the bisection of M is introduced for the conformity, the subroutine `completion(T,M)` will produce $\overline{\mathcal{T}}$.

There are several bisection methods proposed in three and higher dimensions which try to generalize the newest vertex bisection [9, 33, 59, 4, 37]. Indeed all of these methods are more or less equivalent in Rule 3. We shall not give detailed description of these bisection methods here since the description of Rules 1 and 3 is very technical for three and higher dimensions. In these methods, (B1) is relatively easy to prove by showing all descendants of a simplex in $\mathcal{T}_0$ fall into similarity classes. Usually (B2) requires special initial labeling, i.e., Rule 1.

For some special triangulations and initial labeling, longest edge bisection and newest vertex bisection are equivalent. For example, for triangulations composed by isosceles right triangles using the longest edge in Rule 1, then longest bisection and newest vertex bisection coincides. Similar fact holds for 3-D triangulations obtained by dividing one cube into six tetrahedron. But in general, these two methods are quite different. See [38] for a throughly comparison in 2-D.

Remark 5.1. If $D^k(\mathcal{T}_0)$ is conforming, for any $k \geq 0$, there is a recursive algorithm for the completion. But to ensure this condition, special initial labeling is needed. The requirement of such initial labeling is practical in 2-D [38, 40] and more restrictive in three and high dimensions [33, 12, 59, 37, 58].

5.2. **Implementation of newest vertex bisection in two dimensions.** In this subsection, we shall present an efficient completion algorithm which will terminate for arbitrary initial labeling. This completion algorithm is firstly proposed in our recent work [15].

The following subroutine

```
[node,elem,bdEdge] = bisect(node,elem,markedElem,bdEdge)
```

will refine the current triangulation by bisecting marked elements and minimal neighboring elements to get a conforming and shape regular triangulation. Newest vertex bisection is used. The input `markedElem` is a vector containing the index of elements to be bisected, and `bdEdge` stores boundary edges. The argument `bdEdge` in the input and output is optional and can be omitted.

5.2.1. *Refinement edge.* The first question is: how to represent a labeled triangulation? Of course we could introduce an additional vector to record the refinement edge of each element. But we have a better way to do that. Recall that our representation of a triangle: `elem(t,[1 2 3])` are global indices of three vertices of the triangle `t`. The only requirement on the ordering is that the orientation is counter-clockwise. A cyclical permutation of three indices still represents the same triangle. Namely `elem(t,[2 3 1]), elem(t,[3 1 2])` represent the same triangle as `elem(t,[1 2 3])`. We shall use the following rule to record the refinement edge:

The refinement edge of `t` *is* `elem(t,[2 3])`. (13)

That is `elem(t,1)` will be always the newest vertex of the triangle t. By following this rule, we build the labeled triangle into the ordering of the second dimension of the matrix `elem`.

5.2.2. *Completion.* We now give a more efficient algorithm, comparing with `completion(T,M)`, for the completion procedure. A key observation is that since $\overline{\mathcal{T}} \leq D^2(\mathcal{T})$, the new points added from $\mathcal{T}$ to $\overline{\mathcal{T}}$ are middle points of some edges of $\mathcal{T}$. Therefore instead of operating on triangles, we cut necessary edges first.

Let us denote the refinement edge of τ by $\mathrm{cutEdge}(\tau)$. For a triangle $\tau \in \mathcal{T}$, there are at most three neighbors of τ. We define the refinement neighbor of τ, denoted by $F(\tau)$, as the neighbor sharing the refinement edge of τ. When $\mathrm{cutEdge}(\tau)$ is on the boundary, we define $F(\tau) = \tau$. Note that

$\mathrm{cutEdge}(\tau)$ may not be the refinement edge of $F(\tau)$. To ensure the conformity, it suffices to satisfy the rule

If $\mathrm{cutEdge}(\tau)$ *is bisected, then* $\mathrm{cutEdge}(F(\tau))$ *should also be bisected.*

Given a triangle τ to be bisected, we bisect $\mathrm{cutEdge}(\tau)$ and check the refinement edge of the refinement neighbor of τ, i.e., $\mathrm{cutEdge}(F(\tau))$. If it is bisected already, we stop. Otherwise, we bisect $\mathrm{cutEdge}(F(\tau))$ and check $\mathrm{cutEdge}(F^2(\tau))$ and so on. Starting form a marked triangle τ, we then have a flow:

$$\mathrm{cutEdge}(\tau) \rightarrow \mathrm{cutEdge}(F(\tau)) \rightarrow \ldots \rightarrow \mathrm{cutEdge}(F^m(\tau)).$$

The flow will end if $\mathrm{cutEdge}(F^m(\tau))$ is already bisected. In the best senior, τ and $F(\tau)$ share a refinement edge and thus $m = 1$. In general, the length of the flow could be very long. But since this is a flow among all edges of $\mathcal{T}$, it will stop in finite steps (the worst case is that it traverse all edges).

Using our auxiliary data structure `neighbor` and `elem2edge`, this completion can be implemented efficiently; See the following self-explanatory code.

```
isCutEdge = false(NE,1);
while sum(markedElem)>0
    isCutEdge(elem2edge(markedElem,1)) = true;
    refineNeighbor = neighbor(markedElem,1);
    markedElem = refineNeighbor(~isCutEdge(elem2edge(refineNeighbor,1)));
end
edge2newNode = zeros(NE,1);
edge2newNode(isCutEdge) = N+1:N+sum(isCutEdge);
node(N+1:N+sum(isCutEdge),:) = (node(edge(isCutEdge,1),:) + node(edge(isCutEdge,2),:))/2;
```

5.2.3. *Bisections of marked triangles.* Since we have added all necessary nodes from $\mathcal{T}$ to $\overline{\mathcal{T}}$, we only need to bisect the triangle whose refinement edge is bisected.

```
for k=1:2
    t = find(edge2newNode(elem2edge(:,1))>0);
    newNT = length(t);
    if (newNT == 0), break; end
    L = t; R = NT+1:NT+newNT;
    p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
    p4 = edge2newNode(elem2edge(t,1));
    elem(L,:) = [p4, p1, p2];
    elem(R,:) = [p4, p3, p1];
    bdEdge(R,[1 3]) = bdEdge(t,[2 1]);
    bdEdge(L,[1 2]) = bdEdge(t,[3 1]);
    elem2edge(L,1) = elem2edge(t,3);
    elem2edge(R,1) = elem2edge(t,2);
    NT = NT + newNT;
end
```

We labeled vertices of t as p_1, p_2, p_3 and the new node added on its refinement edge as p_4 (Fig. 6). We use the current location of t to store the left child of t and append a new element to `elem` matrix to store the right child. Here left and right is uniquely determined by the direction from p_4 to p_1. In children elements, the first node is changed to the newest vertex p_4 following the rule (13). The boundary edges is updated in line 10 and 11.

It is possible that the other two edges of t are also bisected due to the conformity. For example, they could be refinement edges of corresponding neighbors. Thus we need to recheck the `elem` matrix and apply further bisections if necessary. In line 12 and 13, we assign the correct refinement edge to the children elements. Note that we only need to repeat the process twice since $D^2(\mathcal{T})$ will be conforming.

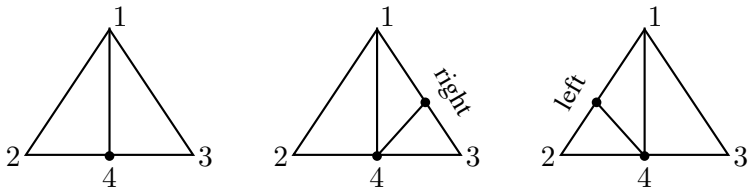


FIGURE 4. Divide a triangle according to the marked edges

5.2.4. *Initial labeling.* Although the completion procedure we proposed will terminate for arbitrary initial labeling, for the sake of shape regularity, we shall use the longest edge rule in Rule 1. Namely we assign the longest edge as the refinement edge of each triangle in the initial triangulation $\mathcal{T}_0$. The subroutine `label` is only need for the initial triangulation and thus called outside of the function `bisect`. It is interesting to note that if we call `label` inside `bisect`, then `bisect` becomes the longest edge bisection.

The following code is self-explanatory.

```
function elem = label(node,elem)
totalEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
ve = node(totalEdge(:,1),:)-node(totalEdge(:,2),:);
edgeLength = reshape(sum(ve.^2,2),size(elem,1),3);
[temp,I] = max(edgeLength,[],2);
elem((I==2),[1 2 3]) = elem((I==2), [2 3 1]);
elem((I==3),[1 2 3]) = elem((I==3), [3 1 2]);
```

5.3. **Implementation of longest edge bisection in three dimensions.** We shall implement the longest edge bisection in three dimensions. For this method, the Rule 1 and 3 are pretty simple: we always bisect the longest edge of a tetrahedron. The completion procedure is the standard one `completion(T,M)`. The function

```
[node,elem,bdFace] = bisect3(node,elem,markedElem,bdFace)
```

will refine the current triangulation by bisecting marked elements and minimal neighboring elements to get a conforming and shape regular triangulation. Longest edge bisection is used in the bisection. The input `markedElem` is an array containing the index of elements to be bisected, and `bdFace` stores boundary edges. The argument `bdFace` in input and output is optional and can be omitted.

5.3.1. *Bisection of marked elements.* For each element in the marked set M, we can get its refinement edge by finding its longest edge. If the refinement edge is not cut before, we add its middle point as a new node. Otherwise, we need to find out the index of this already added middle point. In 2-D, this is solved by a pointer vector `edge2newNode` and the link matrix `elem2edge`; See the code in Section 5.2.2. In 3-D, we use the idea in the construction `elem2edge` without generating `elem2edge` matrix.

The following code will find out the refinement edge of each marked element is cut or not.

```
elem = label3(node,elem,markedElem);
p1 = elem(markedElem,1); p2 = elem(markedElem,2);
ncEdge = find(~isConforming(1:Ncut));
nv2v = sparse([cutEdge(ncEdge,3);cutEdge(ncEdge,3)],...
              [cutEdge(ncEdge,1);cutEdge(ncEdge,2)],1,N,N);
[i,j] = find(nv2v(:,p1).*nv2v(:,p2));
isNewCutEdge = true(NM,1);
isNewCutEdge(j) = false;
ix = find(isNewCutEdge);
```

In line 1, the subroutine `label3` is similar to `label` in 2-D. It will compute edge lengths of each tetrahedron and switch the index such that the refinement edge of `t` is given by `[elem(t,1),elem(t,2)]`. Note that for the longest edge, the labeling is called inside `bisect3`.

In line 3, we use the logical vector `isConforming` to find out which edge is nonconforming and restrict the computation to the set of nonconforming edges. In line 4-5, we form a sparse matrix `nv2v` representing the indices matrix of new vertices and vertices. That is `nv2v(m,i)=1` and `nv2v(m,j)=1` if `m` is the middle point of `i,j`. Here the value `cutEdge(:,3)` stores the index of new nodes and `cutEdge(:,1:2)` are two parent nodes.

We perform the product of `nv2v(:,p1(t)).*nv2v(:,p2(t))` and use `find` to locate indices set of all non zeros. If the result is zero, then the middle point of `p1(t)` and `p2(t)` is not added before and thus the refinement edge of `t` is a new cut edge. Otherwise the i-th index of the nonzero value is the index of the middle point which is already added.

After the new cut edges are added to `cutEdge` and the middle points are added to `node`, we can form the sparse matrix `nv2v` again and use the same procedure to find out the indices of middle points of refinement edges for all marked elements.

The bisection of marked element and the update of boundary faces is then straightforward.

```
1 elem(markedElem,:) = [p1 p5 p3 p4];
2 elem(NT+1:NT+NM,:) = [p5 p2 p3 p4];
3 bdFace(NT+1:NT+NM,[1 3 4]) = bdFace(markedElem, [1 3 4]);
4 bdFace(markedElem,2:4) = bdFace(markedElem,2:4);
```

5.3.2. *Finding elements containing hanging nodes.* In this subsection, we shall discuss how to find elements containing hanging nodes in the second step of `completion(T,M)` using sparse matrixlization.

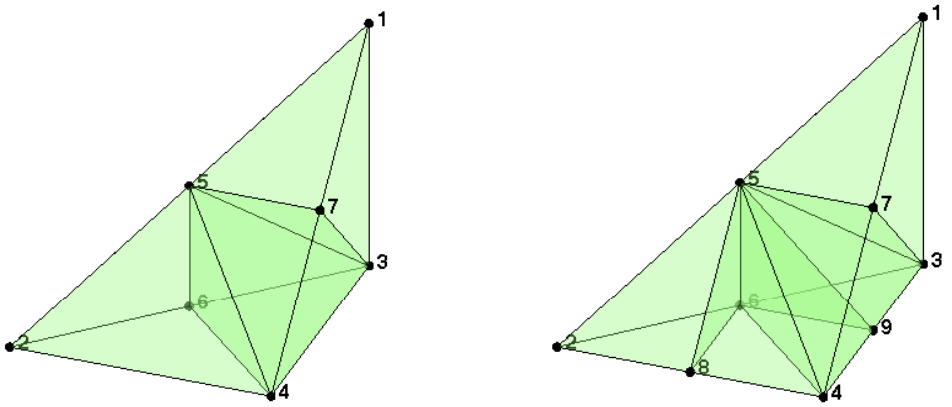
```
1 t2v = sparse([1:NT,1:NT,1:NT,1:NT], elem(1:NT,:), 1, NT, N);
2 suspect = find(~isConforming(1:Ncut));
3 [i,j] = find(t2v(:,cutEdge(suspect,1)).*t2v(:,cutEdge(suspect,2)));
4 markedElem = unique(i);
5 isConforming(suspect) = true;
6 isConforming(suspect(j)) = false;
```

We first construct the incidence matrix between tetrahedron and vertices such that `t2v(t,v)=1` if `v` is a vertex of `t`. Since each tetrahedron consists of 4 vertices, each row of `t2v` contains four non-zeros. The nonzero in the i-th column is the indices of elements use i as a vertex, i.e., the star of the node i.

Let us take a bisected edge with ending nodes i and j. The middle point of i and j is added as a new node. We compute the intersection of their stars, i.e., `nodeStar(i)`$\cap$`nodeStar(j)`. If the middle point of i and j is not a hanging node, then i and j are separated and thus the intersection is empty. Otherwise the intersection contains elements using i and j as vertexes and thus containing the middle point of i and j. We then put them into the marked elements.

Instead using a `for` loop to search bisected edges one by one, we use sparse matrixlization to find the intersection. The Hadamard product of `nodeStar1` and `nodeStar2` will leave the intersection location as nonzero, which is the star of the edge. We then use `find` to decompose the resulting matrix. The index vector i contains row indices of nonzero values which are indices of elements containing hanging nodes. The index vector j will tell us the non-conforming edges. We set `isConforming(j)=false` to indicate that we still need to check these edges next time. The computation is restricted to non-conforming edge only.

An example of this procedure is illustrated by Figure 5 and Table 2.



(a) Input triangulation (b) node 9 is an hanging node

FIGURE 5. The input triangulation is shown in (a). After bisections of two elements, the hanging node 9 is presented in (b).

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	1	1	0	1	0
2	1	0	1	0	1	0	1	0	0
3	0	0	1	0	1	1	0	0	1
4	0	0	1	1	1	0	1	0	0
5	0	0	0	1	1	1	0	1	0
6	0	0	0	1	1	1	0	0	1

t2v

	1	2		1	2		1	2
1	1	0		0	0		0	0
2	0	1		0	0		0	0
3	0	1	·	0	0	=	0	0
4	0	1		1	1		0	1
5	0	0		1	1		0	0
6	0	0		1	1		0	0

t2v(:,[2 3]).*t2v(:,[4 4])

TABLE 2. The left table is the `t2v` matrix for the triangulation in Figure 5 (b). The edge `[2,4]` and `[3,4]` in (a) are bisected. The nonzero in `t2v(:,[2 3]).*t2v(:,[4 4])` indicates the second edge `[3,4]` and the 4-th element `elem(4,:)` contains a hanging node.

6. COARSENING

In this section, we shall discuss the coarsening algorithm for newest vertex bisection in two dimensions. The algorithm is firstly presented in our recent work [18]. We include the content here briefly for the completeness and present a new efficient implementation using sparse matrixlization. The generalization of this coarsening algorithm to 3-D bisection is possible but more combinatorially complicated.

6.1. Decomposition of bisection grids.

Our coarsening algorithm is based on a novel decomposition of bisection grids in any dimensions. This decomposition is introduced in our recent work [16].

Given a labeled triangulation $(\mathcal{T}, \mathcal{L})$, an edge e of $\mathcal{T}$ is called a *compatible edge* if e is the refinement edge of each element in the edge star S_e. For a compatible edge, the star S_e is called a *compatible star*. A *compatible bisection* b_e consists of bisections of all element in S_e, i.e., $b_e := (b_{\tau_1}, \cdots, b_{\tau_{\#S_e}}), \tau_i \in S_e$ for $i = 1, \cdots, \#S_e$. We then define a formal addition

$$\mathcal{T} + b_e := \mathcal{T} + (b_{\tau_1}, \cdots, b_{\tau_{\#S_e}}).$$

For a sequence of compatible bisections $\mathcal{B}$, the addition $\mathcal{T} + \mathcal{B}$ is defined similarly as in (11). Note that if $\mathcal{T}$ is conforming and b_e is compatible, then $\mathcal{T} + b_e$ is conforming. Namely compatible bisections

preserve the conformity of triangulations. Thus it is more fundamental in the theoretical analysis and practical implementation.

In two dimensions, a compatible bisection b_e has only two possible configurations. One is bisecting an interior compatible edge in which case, the star S_e forms a quadrilateral. Another case is bisecting a boundary compatible edge and S_e forms a triangle; see Figure 6. In three dimensions, the configuration cases of compatible bisections depends on the initial labeling.

The bisection of paired triangles was firstly introduced by Mitchell [38, 39]. The idea was generalized by Kossaczký [33] to three dimensions, and Maubach [37] to any dimensions. We also note that the compatible bisection is called atomic refinement in ALBERTA [55]. In the aforementioned references, compatible bisection is introduced to construct efficient recursive completion procedures of bisection methods. We shall use them to characterize the conforming mesh obtained by bisection methods and construct coarsening algorithm.

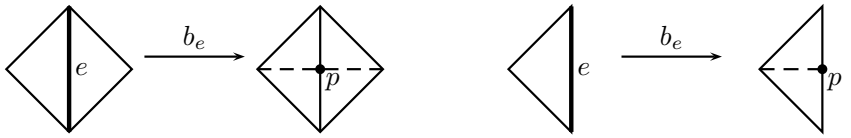


FIGURE 6. Two compatible bisections. Left: interior edge; right: boundary edge. The edge with boldface is the compatible refinement edge, and the dash-line represents the bisection.

We now present a theorem on the decomposition of triangulations in the conforming triangulation class $\mathscr{C}(\mathcal{T}_0)$ using compatible bisections. The proof can be found at [16].

Theorem 6.1. *Suppose the bisection method and the initial triangulation $\mathcal{T}_0$ satisfy the assumption:*

(U) for any $k \geq 0$, the k-th uniform refinements $D^k(\mathcal{T}_0)$ is conforming.

Then for any $\mathcal{T} \in \mathscr{C}(\mathcal{T}_0)$, there exists a compatible bisection sequence $\mathcal{B} = (b_1, b_2, \cdots, b_N)$, with N is the difference of number of nodes of $\mathcal{T}$ and $\mathcal{T}_0$, such that

$$\mathcal{T} = \mathcal{T}_0 + \mathcal{B}. \tag{14}$$

Remark 6.2. The assumption (U) is a sufficient condition for (14) holds. It is not by no means a necessary condition. For example, (U) does not hold for the triangulation in Figure 1 (b) using longest edge bisection. But the decomposition (14) still holds.

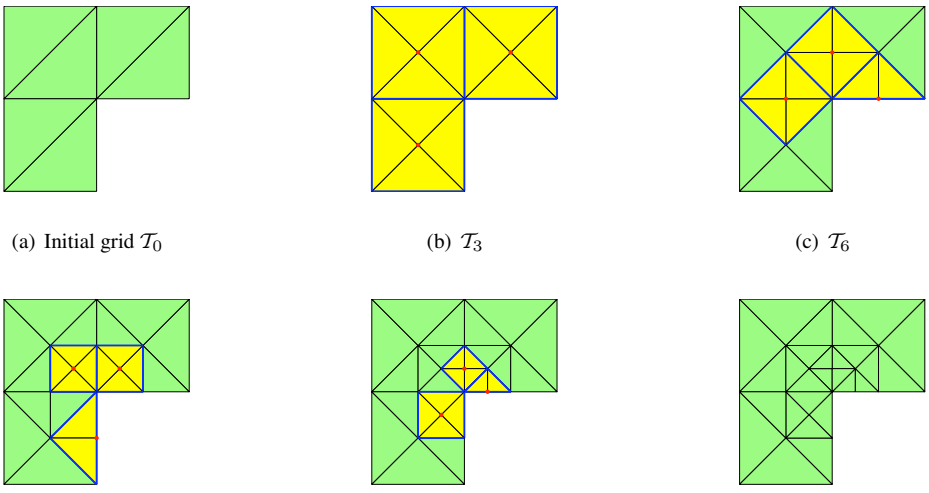
We use Figure 7 to illustrate the decomposition of a bisection grid obtained by newest vertex bisection. In these figures, we denoted by $\mathcal{T}_i := \mathcal{T}_0 + (b_1, b_2, \cdots, b_i)$ for $1 \leq i \leq 12$.

6.2. **Nodal-wise coarsening algorithm.** In this subsection, we shall present our coarsening algorithm for the adaptive grids obtained by newest vertex bisection. Unlike the existing element-wise coarsening algorithms in the literature [33, 56], our new coarsening algorithm is node-wise.

A key observation is that the inverse of a compatible bisection can be thought as a coarsening process. It is restricted to a compatible star and thus no conformity issue arises. See Figure 6 for an illustration. For a triangulation $\mathcal{T} \in \mathscr{C}(\mathcal{T}_0)$, a node p is called a *good-for-coarsening node*, or a *good node* in short, if there exist a compatible bisection b_e such that p is the middle point of e. The set of all good nodes in the grid $\mathcal{T}$ will be denoted by $\mathcal{G}(\mathcal{T})$.

Our new coarsening algorithm is simply read as the following:

```
function coarsen(T)
Find all good nodes of T;
Replace the star S_x by b_e^{-1}(S_x) for each good node x.
```



(a) Initial grid $\mathcal{T}_0$ (b) $\mathcal{T}_3$ (c) $\mathcal{T}_6$

(d) $\mathcal{T}_9$ (e) $\mathcal{T}_{12}$ (f) Fine grid $\mathcal{T}$

FIGURE 7. Decomposition of bisection grids obtained by longest edge bisection. The initial grid and the fine grid are presented in (a) and (f), respectively. In (b)-(e), each triangulation is obtained by performing three compatible bisections on the previous one. The compatible star is indicated by yellow region and the new vertices introduced by bisections are marked by red dots

The following theorem proves the existence of good nodes and gives a useful characterization of good nodes for 2-D newest vertex bisection. In [18], we prove that we can finally obtain the initial grid back by applying the coarsening algorithm `coarsen` repeatedly. To present this theorem, we call a labeled triangulation $(\mathcal{T}, \mathcal{L})$ *compatible labeled*, if $\mathcal{T}$ can be decomposed as the union of compatible stars.

Theorem 6.3. *Let $\mathcal{T}_0$ be a compatible labeled conforming triangulation. Then for any $\mathcal{T} \in \mathscr{C}(\mathcal{T}_0)$ and $\mathcal{T} \neq \mathcal{T}_0$, the set of good nodes $\mathcal{G}(\mathcal{T})$ is not empty. Furthermore $x \in \mathcal{G}(\mathcal{T})$ if and only if*

(1) *it is not a vertex of the initial grid $\mathcal{T}_0$;*
(2) *it is the newest vertex of all elements in the star of S_x.*
(3) *$\#S_x = 4$ for an interior node x or $\#S_x = 2$ for a boundary node x.*

Remark 6.4. The assumption, $\mathcal{T}_0$ is compatible labeled, is not restrictive. Indeed Mitchell [38] proved that for any conforming triangulation $\mathcal{T}$, there exist a compatible labeling. Recently Biedl, Bose, Demaine, and Lubiw [12] give an $O(N)$ algorithm to find a compatible labeling for a triangulation with N elements.

Remark 6.5. The assumption, $\mathcal{T}_0$ is compatible labeled, could be further relaxed by using the longest edge of each triangle as its refinement edge for the initial triangulation $\mathcal{T}_0$; see Kossaczký [33].

Remark 6.6. It is possible that `coarsen(T)` applied to the current grid $\mathcal{T}$ gives a coarse grid which is not in the adaptive history. Indeed our coarsening algorithm may remove nodes added in several different stages of the adaptive procedure.

6.3. **Implementation of coarsening algorithm.** In this subsection, we shall present an efficient implementation of the coarsening algorithm developed in the previous section. A unique feature of our

algorithm is that we only maintain `node` and `elem` matrices. At a first glance, one may wonder how can we go back without store the history. Our answer is: a tree structure of the adaptive procedure can be build into the `elem` matrix by the ordering. In principal, the LEFT and RIGHT direction in the binary tree will be recorded by BEFORE and AFTER ordering.

More precisely, we shall impose three assumptions on the ordering of `node` and `elem` matrices which can be easily build into the bisection. First the order of vertices will be used to represent a labeling of an element. The second assumption is on the order of triangles in `elem` matrix. The last one is on the ordering of `node` matrix. It enforces the algorithm not to coarsen the grid points in the initial triangulation.

(O1) `elem(t,1)` stores the newest vertex of element `t`.

(**O2**) When an element is bisected, its left child is stored in a prior to its right child in `elem` matrix.

(**O3**) The nodes in the initial triangulation $\mathcal{T}_0$ is stored in the range `node(1:N0,:)`.

6.3.1. *Finding good nodes.* The code to find good nodes is simply the translation of three conditions in Theorem 6.3. The only difference is that we only check the newest vertices of marked elements. The marked element vector is similar in the bisection case. It is introduced to control the error in the adaptive procedure. Note that in line 3, the valence is restricted to the newest vertex while in line 2 is for all nodes.

```
1 N = size(node,1); NT = size(elem,1);
2 valence = accumarray(elem(:),ones(3*NT,1),[N 1]);
3 markedVal = accumarray(elem(markedElem,1),ones(length(markedElem),1),[N 1]);
4 isIntGoodNode = ((markedVal==valence) & (valence==4));
5 isIntGoodNode(1:N0) = false;
6 isBdGoodNode = ((markedVal==valence) & (valence==2));
7 isBdGoodNode(1:N0) = false;
```

6.3.2. *Coarsening the compatible star.* The node star of a good node can be obtained by extracting columns from the sparse matrix `t2v`. For boundary good nodes, due to (O2), the two triangles in a compatible star appears in the type (L,R). So the parent triangle is uniquely determined. For interior good nodes, there could be several possibilities. We switch the four triangles in the node start such that (`t1,t2`) and (`t3,t4`) come from the same parent and in the type (L,R), respectively.

```
1  t2v = sparse([1:NT,1:NT,1:NT], elem(1:NT,:), 1, NT, N);
2  % interior node
3  [ii,jj] = find(t2v(:,isIntGoodNode)); clear jj
4  nodeStar = reshape(ii,4,sum(isIntGoodNode));
5  ix = (elem(nodeStar(1,:),3) == elem(nodeStar(4,:),2));
6  iy = (elem(nodeStar(2,:),2) == elem(nodeStar(3,:),3));
7  nodeStar(:,ix & iy) = nodeStar([1 2 3 4],ix & iy);
8  nodeStar(:,ix & ~iy) = nodeStar([1 3 2 4],ix & ~iy);
9  nodeStar(:,~ix & iy) = nodeStar([1 4 2 3],~ix & iy);
10 nodeStar(:,~ix & ~iy) = nodeStar([1 4 3 2],~ix & ~iy);
11 t1 = nodeStar(1,:); t4 = nodeStar(4,:);
12 t2 = nodeStar(2,:); t3 = nodeStar(3,:);
13 p2 = elem(t1,3); p3 = elem(t2,2); p4 = elem(t1,2); p5 = elem(t3,2);
14 elem(t1,:) = [p4 p2 p3]; elem(t2,1) = 0;
15 elem(t3,:) = [p5 p3 p2]; elem(t4,1) = 0;
16 % boundary nodes
17 [ii,jj] = find(t2v(:,isBdGoodNode));
18 nodeStar = reshape(ii,2,sum(isBdGoodNode));
19 t5 = nodeStar(1,:); t6 = nodeStar(2,:);
```

```
20 p2 = elem(t5,3); p3 = elem(t6,2); p4 = elem(t5,2);
21 elem(t5,:) = [p4 p2 p3]; elem(t6,1) = 0;
```

Remark 6.7. The correct configuration for the node star of an interior good node can be constructed by using `neighbor` matrix. But since `auxstructure` subroutine calls a `unique` command, the current implementation using only one `sparse` command is more efficient.

6.3.3. *Clearance of the empty entries.* To efficiently use the memory, we shall empty unused memory in the `node` and `elem` matrices.

```
1 elem((elem(:,1) == 0),:) = [];
2 node(isIntGoodNode | isBdGoodNode,:) = [];
3 indexMap = zeros(N,1);
4 indexMap(~(isIntGoodNode | isBdGoodNode))= 1:size(node,1);
5 elem = indexMap(elem);
```

The clearance of `node,elem` matrices is relatively easy; See the code line 1 and 2. But we should also shift the indices in `elem` to reflect to the change of node indices. So we build an index map from old node index to the new and shortened node index. Then `elem` is shifted by the index map in line 5.

6.3.4. *Update of boundary edges.* Since some points are deleted and the node index is shifted, we need to update the boundary edges accordingly. The following code is for this purpose.

```
1 bdEdge(t1,:) = [bdEdge(t1,2) bdEdge(t2,1) bdEdge(t1,1)];
2 bdEdge(t3,:) = [bdEdge(t3,2) bdEdge(t4,1) bdEdge(t3,1)];
3 bdEdge(t5,:) = [bdEdge(t5,2) bdEdge(t6,1) bdEdge(t5,1)];
```

7. Adaptive Finite Element Methods

Standard adaptive finite element methods (AFEM) based on local mesh refinement can be written as loops of the form

$$\textbf{SOLVE} \rightarrow \textbf{ESTIMATE} \rightarrow \textbf{MARK} \rightarrow \textbf{REFINE/ COARSEN}. \tag{15}$$

More precisely, starting from an initial triangulation $\mathcal{T}_0$, to get $\mathcal{T}_{k+1}$ from $\mathcal{T}_k$ for $k \geq 0$, we first solve the equation to get the solution u_k. The error between the exact solution u and the computed approximation u_k is estimated using u_k, $\mathcal{T}_k$, and possibly the data f. The error estimator is distributed to element-wise error indicator which is used to mark a set of of triangles in $\mathcal{T}_k$. Marked triangles and possible more neighboring triangles are then refined or coarsened in such a way that the triangulation is still shape regular and conforming.

7.1. **The module SOLVE.** We shall not discuss the step **SOLVE** in this paper. We use the build in director solver of MATLAB. The performance is acceptable for systems with size less than 1 million. We do implement the most efficient iterative solver: using multigrid as a preconditioner in the Preconditioned Conjugate Gradient method in 2-D. For details, we refer to our recent paper [18].

7.2. **The module ESTIMATE.** The *a posteriori* error estimators and indicator are essential part of the **ESTIMATE** step. We use residual type error estimator as an example to illustrate the idea. For any $\tau \in \mathcal{T}$ and any $v_{\mathcal{T}} \in \mathbb{V}_{\mathcal{T}}$, we define

$$\eta(v_{\mathcal{T}}, \tau) = \left(\|h_\tau f\|_{0,\tau}^2 + \sum_{e \in \partial\tau} \|h_\tau^{1/2} [\nabla v_{\mathcal{T}} \cdot n_e]\|_{0,e}^2 \right)^{1/2}, \tag{16}$$

where $h_\tau = |\tau|^{1/d}$ represents the size of the element and the jump of flux across the side e (edge in 2-D or face in 3-D) is defined as

$$[\nabla v_{\mathcal{T}} \cdot \boldsymbol{\nu}_e] = \nabla v_{\mathcal{T}} \cdot \boldsymbol{\nu}_e|_{\tau_1} - \nabla v_{\mathcal{T}} \cdot \boldsymbol{\nu}_e|_{\tau_2},$$

if $e = \tau_1 \cap \tau_2$ and $[\nabla u \cdot \boldsymbol{\nu}_e] = 0$ if $e \in \partial\Omega$. Here $\boldsymbol{\nu}_e$ is a fixed unit normal direction of side e.

Let u and $u_{\mathcal{T}}$ be the exact solution and the linear finite element approximation of the Poisson equation with pure Dirichilet boundary condition. Then one can prove the error estimator

$$|u - u_{\mathcal{T}}|_1^2 \leq C \sum_{\tau \in \mathcal{T}} \eta^2(u_\tau, \tau). \tag{17}$$

The element wise quantity $\eta(u_\tau, \tau)$ is called *error indicator* of τ. Note that it may not be a upper bound of the local error in τ.

The computation of the residual type error indicator is easy using our auxiliary data structure `neighbor`. Here we take part of codes in `estimateresidual3` to explain how to compute the jump of ∇u. The element residual is computed using 4-points quadrature rule.

```
1 [Du,volume,normal] = gradient3(node,elem,u);
2 h = volume.^(1/3);
3 neighbor = auxstructure3(elem);
4 faceJump = dot((Du-Du(neighbor(:,1),:)),normal(:,:,1),2).^2 ...
5          + dot((Du-Du(neighbor(:,2),:)),normal(:,:,2),2).^2 ...
6          + dot((Du-Du(neighbor(:,3),:)),normal(:,:,3),2).^2 ...
7          + dot((Du-Du(neighbor(:,4),:)),normal(:,:,4),2).^2;
8 faceJump = h.^(-1).*faceJump;
```

Line 1 calls a subroutine `gradient3` to compute the gradient of a finite element function u using the formula of $\nabla\lambda_i$ in (9). The output `normal` is the scaled normal. Therefore the `faceJump` is scaled accordingly in line 8.

Remark 7.1. In *i*FEM, we also implement recovery type error estimators; See `estimateW21` and `recoveryZZ` subroutines.

7.3. **The module MARK.** The *a posteriori* error estimator is split into local error indicators and they are then employed to make local modifications by dividing the elements whose error indicator is large and possibly coarsening the elements whose error indicator is small. The way we mark these elements influences the efficiency of the adaptive algorithm. The traditional maximum marking strategy proposed in the pioneering work of Babuška and Vogelius [5] is to mark triangles τ^* for refinement such that

$$\eta(u_{\mathcal{T}}, \tau^*) \geq \theta \max_{\tau \in \mathcal{T}} \eta(u_{\mathcal{T}}, \tau), \quad \text{for some } \theta \in (0, 1). \tag{18}$$

Similar marking by reversing the inequality in (18) is used for coarsening. Such marking strategy is designed to evenly equi-distribute the error. We may leave some exceptional elements and focus on the overall amounts of the error. This leads to the bulk criterion firstly proposed by Dörfler [24] in order to prove the convergence of the local refinement strategy. With such strategy, one defines the marking set $\mathcal{M} \subset \mathcal{T}$ such that

$$\sum_{\tau \in \mathcal{M}} \eta^2(u_{\mathcal{T}}, \tau) \geq \theta \sum_{\tau \in \mathcal{T}} \eta^2(u_{\mathcal{T}}, \tau), \quad \text{for some } \theta \in (0, 1). \tag{19}$$

We implement these two popular marking strategies in `mark` subroutine where we call (18) as 'MAX' type and (19) as 'L2' type. The default one is the later.

```
1 function markedElem = mark(elem,eta,theta,method)
2 NT = size(elem,1); isMark = false(NT,1);
3 if (nargin < 4), method = 'L2'; end  % default marking is L2 based
4 switch upper(method)
5     case 'MAX'
6         isMark(eta>theta*max(eta))=1;
7     case 'COARSEN'
8         isMark(eta<theta*max(eta))=1;
```

```
    case 'L2'
        [sortedEta,ix] = sort(eta.^2,'descend');
        x = cumsum(sortedEta);
        isMark(ix(x < theta* x(NT))) = 1;
        isMark(ix(1)) = 1;
end
markedElem = find(isMark==true);
```

7.4. **The module REFINE/COARSENING.** After choosing a set of marked elements, we use bisection or coarsening algorithms discussed in previous sections to refine or coarsen elements where the error indicator is big or small, respectively.

7.5. **Numerical examples.** Due to the page limit, we provide only two examples. Extensive examples, as well as concise proves of convergence of AFEM, can be found at our book: *Introduction to adaptive finite element methods using MATLAB*.

Crack problem in 2-D. The first example is the following 2-D crack problem consider in [42]. Let $\Omega = \{|x| + |y| < 1\} \backslash \{0 \leq x \leq 1, y = 0\}$ with a crack and the solution u satisfies the Poisson equation

$$-\Delta u = 1, \text{ in } \Omega \quad \text{and } u = g_D \text{ on } \partial\Omega.$$

We choose g_D such that the exact solution u in polar coordinates is

$$u(r,\theta) = r^{\frac{1}{2}} \sin\frac{\theta}{2} - \frac{1}{4}r^2.$$

We use piecewise linear and global continuous finite element to solve this Poisson equation. The initial grid is given by hand. For general domains, one can use MATLAB's PDE tool box or use distmesh [43], a simple mesh generator written in MATLAB, to generate an initial mesh and call `label(node,elem)` for the initial labeling.

Cube problem in 3-D. The second example is the following Poisson equation in the cube $\Omega = (-1,1)^3$:

$$-\Delta u = f, \text{ in } \Omega, \quad u = g_D \text{ on } \partial\Omega. \tag{20}$$

We choose f and g_D such that the exact solution $u = e^{-10(x^2+y^2+z^2)}$. The function u presents a point singularity at original. It is also an example tested in [42]. We use piecewise linear and global continuous finite element to solve this Poisson equation. The initial grid is given by hand.

To save space, we only list the code for 3-D cube problem. The code is almost self-explantory. One remark is that, to test the performance, we control the number of the problem size instead of the error.

```
function cube
close all; clear all;
%---------------------- Parameters and Preallocation ----------------------
theta=0.35; maxN=1e5; maxIt = 50; N = zeros(maxIt,1);
errL2 = zeros(maxIt,1); errH1 = zeros(maxIt,1);
%---------------------- Generate initial mesh-----------------------------
node = [-1,-1,-1; 1,-1,-1; 1,1,-1; -1,1,-1; -1,-1,1; 1,-1,1; 1,1,1; -1,1,1];
elem = [1,2,3,7; 1,6,2,7; 1,5,6,7; 1,8,5,7; 1,4,8,7; 1,3,4,7];
for k = 1:2
    [node,elem] = uniformbisect3(node,elem);
end
for k=1:maxIt
    t=cputime;
    u = Poisson3(node,elem,[],@f,@g_D,[]);          % SOLVE
    eta = estimateresidual3(node,elem,u,@f);        % ESTIMATE
    markedElem = mark3(elem,eta.^2,theta);          % MARK
```

```
    [node,elem] = bisect3(node,elem,markedElem);      % REFINE
    N(k) = size(elem,1);
    cost(k) = cputime-t;
    if (N(k)>maxN), break; end
end
```

We compute the H^1 and L^2 norm of the error $u - u_h$ using 1-point and 3-points (in 2-D) or 4-points (in 3-D) quadrature rules, respectively. Figure 8 and Figure 9 evidently show our mesh adaptation achieves the optimal convergent order.

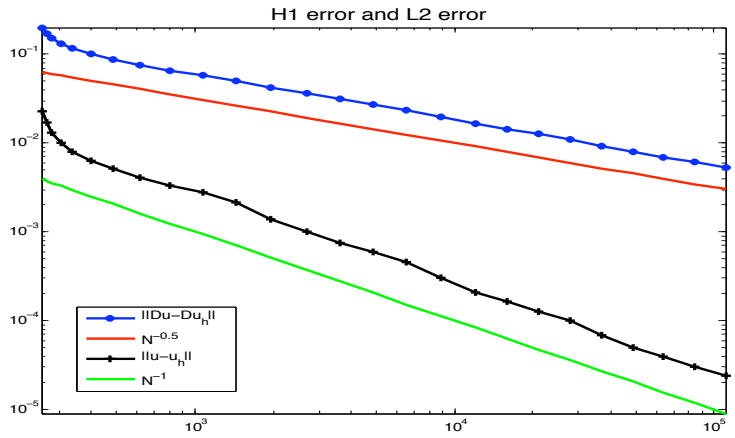


FIGURE 8. Convergent rate of 2-D crack problem.

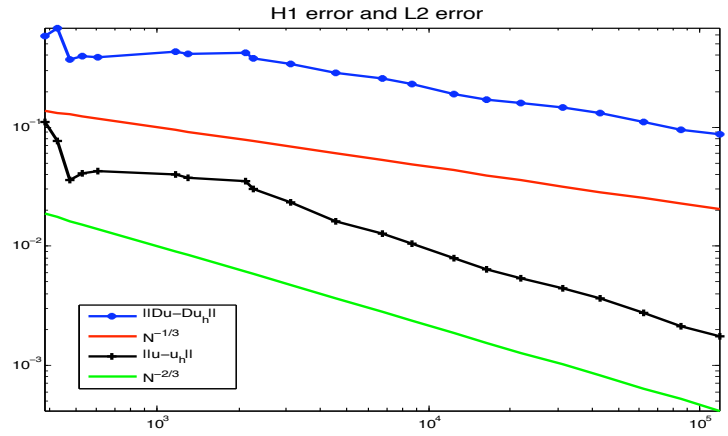


FIGURE 9. Convergent rate of 3-D Cube problem

To show our package can handle large size computation, we list the computational cost in Table 3 and Table 4. The time is measured in seconds. The simulation is done using MATLAB 7.6 in a MacBook Pro laptop with configuration: 2.16 GHz Intel Core 2 Duo and 2 GB 667 MHz DDR2 SDRAM. In short, in a standard laptop, our code can simulate a 2-D problem with size 10^5 in 3 seconds and a 3-D problem in 8 seconds.

Table 4 shows that the complexity for the 3-D cube problem is indeed linear for small size of unknowns but when the number of unknowns is big, it increases faster than linear rate. In contrast, Table 3 shows that up the scale of 10^5, the complexity for the 2-D problem behaves linear with respect to the number unknowns. Detailed profile shows that the solver part takes more than 50% CPU time when the size becomes large. Indeed the director solver build in MATLAB is not of linear complexity. Since the stiffness matrix in 3-D is more dense than that in 2-D, the solver issue is more serious for 3-D problems. We shall address this issue by using multigrid elsewhere. Also the 3-D refinement subroutine is 2 times slower than the counterpart in 2-D.

NT	6534	8839	11990	15842	21140	27903	36959	48846	63940	84038	110598
Time	0.11	0.13	0.19	0.25	0.53	0.61	0.74	1.01	1.29	1.74	2.25

TABLE 3. Computational cost of 2-D crack problem. The first row is the number of elements in each loop and the second row is the CPU time for each loop. The time is measured in second.

NT	1296	2208	3168	4848	8320	12614	19872	31584	49488	74400	119088
Time	0.15	0.21	0.30	0.36	0.60	0.76	1.06	1.72	2.90	5.21	7.98

TABLE 4. Computational cost of 3-D cube problem. The first row is the number of elements in each loop and the second row is the CPU time for each loop. The time is measured in second.

8. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel programming style, sparse matrixlization, and developed an efficient MATLAB package, iFEM, for the adaptive finite element methods. Our package can let researchers considerably reduce development time than traditional programming methods. The package can be downloaded from the website `http://ifem.wordpress.com`.

We should be cautious on the sparse matrixlization of the code. Writing efficient matrixlization code requires a higher level of abstraction and a different way of thinking: *Think in Matrix*. The code is often less readable and thus affect the easy access of package and understanding of the algorithm. It is not worth to spend two hours to optimize the code while only marginal saving on the running time, say one second over total one minute, is achieved [1].

Contrast to many existing MATLAB packages on finite element methods, iFEM can solve middle size problems in few seconds. Since our package heavily relies on the sparse matrix package, we could further improve the performance of iFEM by combining other sparse matrix packages.

We should also mention the limitation of our package. Only linear element and fixed quadrature rule is implemented. The code is still one magnitude slower than packages written by Fortran or C such as ALBERTA [56], MC [31], PHG [63], and PLTMG [7]. Therefore researchers should be aware on the trade-off between the considerably shorter development time and the slightly lower performance.

The data structure introduced in this paper can be used to implement other methods such as quadratic finite element, mixed finite element methods, non-conforming finite element methods, and edge element for electromangnetics. These elements will be added into our package in a near future. A h-p adaptive finite element package is also possible, but requiring further design of data structure.

One nice benefit of sparse matrixlization is the potential scaling characteristics of codes implemented in iFEM. Except the codes involving sparse matrix operations, other parts, e.g., the bisection of marked

elements, are embarrassingly parallel (the tasks really or never communicate). Provided the parallelization of sparse matrix package, the bisection and coarsening can then be easily parallelized, which in contrast may not be easy for code writing in traditional (sequential) styles. We mention that a parallelization of newest vertex bisection can be found in [41] for 2-D and in [63] for 3-D.

It is worth noting that 3-D coarsening is not discussed in this paper. Our decomposition of bisection grids presented in Section 6 holds for bisection methods in any dimensions. But the characterization of compatible star in three dimensions is missing. To do so, we need newest vertex type bisection not longest edge bisection. Additional data structure like the type of elements should be introduced to reduce the combinatory complication. This will be studied and reported elsewhere.

We also note that the computational cost is not scaled linearly with respect to the size of the problem. This is due to the direct solver in MATLAB which is not of optimal (linear) complexity. We have implemented 2-D multigrid based on our coarsening algorithm which is of optimal complexity. Efficient 3-D multigrid will be implemented when coarsening in 3-D is available or by introducing additional data structure. This will be also studied and reported elsewhere.

Historical remark. The author has written a short package [15] of AFEM for elliptic partial differential equations in early 2006 and successfully used it to teach graduate students in 2006 summer school at the Peking University. In late 2006, Chensong Zhang and the author improved it into a more completed package for two dimensional elliptic problems: AFEM@matlab [17], which has already been used in several recent publications [17, 62, 32]. iFEM is different with AFEM@matlab in several aspects: the data structure is updated and constructed more efficiently, the main subroutines are rewritten using new data structure and sparse matrixlization to improve the efficiency. More importantly, three dimensional adaptive finite element method is included.

Acknowledgement. The author would like to thank Professor Michael Holst in University of California at San Diego for the discussion on the data structure for three dimensional mesh refinement and finite element computation, Professor Ludmil Zikatanov in Pennsylvania State University for the discussion on the usage of sparse matrix in the data structure, and also Dr. Chensong Zhang in Pennsylvania State University for the effort in the development of AFEM@matlab, the early version of iFEM.

REFERENCES

[1] P. J. Acklam. MATLAB array manipulation tips and tricks. Notes, 2003.
[2] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numerical Algorithms*, 20:117–137, 1999.
[3] J. Alberty, C. Carstensen, S. A. Funken, and R. Klose. Matlab implementation of the finite element method in elasticity. *Computing*, 69(3):239–263, 2002.
[4] D. N. Arnold, A. Mukherjee, and L. Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM Journal of Scientific Computing*, 22(2):431–448, 2000.
[5] I. Babuška and M. Vogelius. Feeback and adaptive finite element solution of one-dimensional boundary value problems. *Numerische Mathematik*, 44:75–102, 1984.
[6] C. Bahriawati and C. Carstensen. Three matlab implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control. *Computational Methods In Applied Mathematics*, 5(4):333–361, 2005.
[7] R. E. Bank. PLTMG: A software package for solving elliptic partial differential equations users' guide 9.0. *Department of Mathematics, University of California at San Diego*, 2004.
[8] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing*, pages 3–17. IMACS/North-Holland Publishing Company, Amsterdam, 1983.
[9] E. Bänsch. Local mesh refinement in 2 and 3 dimensions. *Impact of Computing in Science and Engineering*, 3:181–191, 1991.
[10] S. Bartels, C. Carstensen, and A. Hecht. P2q2iso2d=2d isoparametric fem in matlab. *Journal of Computational and Applied Mathematics*, 192(2):219–250, 2006.
[11] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.
[12] T. C. Biedl, P. Bose, E. D. Demaine, and A. Lubiw. Efficient algorithms for Petersen's matching theorem. In *Symposium on Discrete Algorithms*, pages 130–139, 1999.

[13] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 2002.
[14] C. Carstensen and R. Klose. Elastoviscoplastic finite element analysis in 100 lines of Matlab. *J. Numer. Math.*, 10(3):157–192, 2002.
[15] L. Chen. Short implementation of bisection in MATLAB. In P. Jorgensen, X. Shen, C.-W. Shu, and N. Yan, editors, *Recent Advances in Computational Sciences – Selected Papers from the International Workship on Computational Sciences and Its Education*, pages 318 – 332, 2008.
[16] L. Chen, R. H. Nochetto, and J. Xu. Multilevel methods on graded bisection grids I: H^1 system. *Preprint*, 2007.
[17] L. Chen and C.-S. Zhang. AFEM@matlab: a Matlab package of adaptive finite element methods. *Technique Report, Department of Mathematics, University of Maryland at College Park*, 2006.
[18] L. Chen and C.-S. Zhang. A coarsening algorithm and multilevel methods on adaptive grids by newest vertex bisection. *Technique Report, Department of Mathematics, University of Maryland at College Park*, 2007.
[19] J. Chessa. Programing the finite element method with matlab. 2002.
[20] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 4 of *Studies in Mathematics and its Applications*. North-Holland Publishing Co., Amsterdam-New York-Oxford, 1978.
[21] M. Dabrowski, M. Krotkiewski, and D. Schmid. MILAMIN: MATLAB-based finite element method solver for large problems. *Geochemistry Geophysics Geosystems*, 9(4), 2008.
[22] T. Davis. Creating sparse Finite-Element matrices in MATLAB. http://blogs.mathworks.com/loren/2007/03/01/creating-sparse-finite-element-matrices-in-matlab/, 2007.
[23] T. A. Davis. *Direct methods for sparse linear systems*, volume 2 of *Fundamentals of Algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2006.
[24] W. Dörfler. A convergent adaptive algorithm for Poisson's equation. *SIAM Journal on Numerical Analysis*, 33:1106–1124, 1996.
[25] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Monographs on Numerical Analysis. The Clarendon Press Oxford University Press, New York, second edition, 1989. Oxford Science Publications.
[26] H. C. Elman, A. Ramage, and D. J. Silvester. Algorithm 866: IFISS, a Matlab toolbox for modelling incompressible flow. *ACM Transactions on Mathematical Software*, 33(2):14, June 2007. Article 14, 18 pages.
[27] H. C. Elman, A. Ramage, and D. J. Silvester. Algorithm 866: Ifiss, a matlab toolbox for modelling incompressible flow. *ACM Trans. Math. Softw.*, 33(2):14, 2007.
[28] S. Funken, D. Praetorius, and P. Wissgott. Efficient implementation of adaptive P1-FEM in MATLAB. *Preprint*, 2008.
[29] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.
[30] M. Holst. MCLite: An adaptive multilevel finite element MATLAB package for scalar nonlinear elliptic equations in the plane. *UCSD Technical report and guide to the MCLite software package*, 2000.
[31] M. Holst. Adaptive numerical treatment of elliptic systems on manifolds. *Advances in Computational Mathematics*, 15:139–191, 2001.
[32] R. H. W. Hoppe, G. Kanschat, and T. Warburton. Convergence analysis of an adaptive interior penalty discontinuous galerkin method. Technical report, Texas A & M University, 2007.
[33] I. Kossaczký. A recursive approach to local mesh refinement in two and three dimensions. *Journal of Computational and Applied Mathematics*, 55:275–288, 1994.
[34] Y. W. Kwon and H. Bang. *The finite element method using MATLAB*. CRC Mechanical Engineering Series. Chapman & Hall/CRC, Boca Raton, FL, second edition, 2000. With 1 CD-ROM (Windows 95, 98 or NT4.0).
[35] J. Li and Y.-T. Chen. *Computational Partial Differential Equations Using MATLAB*. Applied Mathematics & Nonlinear Science. Chapman & Hall/CRC, 2008.
[36] A. Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM Journal on Scientific Computing*, 16(6):1269–1291, 1995.
[37] J. Maubach. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM Journal of Scientific Computing*, 16(1):210–227, 1995.
[38] W. F. Mitchell. *Unified Multilevel Adaptive Finite Element Methods for Elliptic Problems*. PhD thesis, University of Illinois at Urbana-Champaign, 1988.
[39] W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Transactions on Mathematical Software (TOMS) archive*, 15(4):326 – 347, 1989.
[40] W. F. Mitchell. Optimal multilevel iterative methods for adaptive grids. *SIAM Journal on Scientific and Statistical Computing*, 13:146–167, 1992.
[41] W. F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67(4):417–429, 2007.
[42] P. Morin, R. H. Nochetto, and K. G. Siebert. Convergence of adaptive finite element methods. *SIAM Review*, 44(4):631–658, 2002.
[43] P.-O. Persson and G. Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, 2004.

[44] S. Pissanetsky. *Sparse matrix technology*. Academic Press, 1984.
[45] A. Plaza and G. F. Carey. Local refinement of simplicial grids based on the skeleton. *Applied Numerical Mathematics*, 32(2):195–218, 2000.
[46] A. Plaza, M. A. Padron, and G. F. Carey. A 3d refinement/derefinement algorithm for solving evolution problems. *Applied Numerical Mathematics*, 32(4):401–418, 2000.
[47] A. Plaza and M.-C. Rivara. Mesh refinement based on the 8-tetrahedra longest-edge partition. *12th meshing roundtable*, 2003.
[48] K. Polthier. Computational aspects of discrete minimal surfaces. In *Global theory of minimal surfaces*, volume 2 of *Clay Math. Proc.*, pages 65–111. Amer. Math. Soc., Providence, RI, 2005.
[49] C. Pozrikidis. *Introduction To Finite And Spectral Element Methods Using Matlab*. Chapman & Hall/CRC, 2005.
[50] M. C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21:604–613, 1984.
[51] M.-C. Rivara and P. Inostroza. Using longest-side bisection techniques for the automatic refinement fo Delaunay triangulations. *International Journal for Numerical Methods in Engineering*, 40:581–597, 1997.
[52] M. C. Rivara and G. Iribarren. The 4-triangles longest-side partition of triangles and linear refinement algorithms. *Mathemathics of Computation*, 65(216):1485–1501, 1996.
[53] M. C. Rivara and M. Venere. Cost analysis of the longest-side (triangle bisection) refinement algorithms for triangulations. *Engineering with Computers*, 12:224–234, 1996.
[54] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.
[55] A. Schmidt and K. G. Siebert. *Design of adaptive finite element software*, volume 42 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin, 2005. The finite element toolbox ALBERTA, With 1 CD-ROM (Unix/Linux).
[56] A. Schmidt and K. G. Siebert. *The finite element toolbox ALBERTA*. Springer-Verlag, Berlin, 2005.
[57] E. G. Sewell. Automatic generation of triangulations for piecewise polynomial approximation. In *Ph. D. dissertation*. Purdue Univ., West Lafayette, Ind., 1972.
[58] R. Stevenson. The completion of locally refined simplicial partitions created by bisection. *Mathemathics of Computation*, 77:227–241, 2008.
[59] C. T. Traxler. An algorithm for adaptive mesh refinement in n dimensions. *Computing*, 59(2):115–137, 1997.
[60] L. N. Trefethen. *Spectral methods in MATLAB*, volume 10 of *Software, Environments, and Tools*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.
[61] L. N. Trefethen. Ten digit algorithms. Mitchell Lecture, June 2005.
[62] J. Xu, Y. Zhu, and Q. Zou. Convergence analysis of adaptive finite volume element methods for general elliptic equations. (submitted), 2006.
[63] L. Zhang. A parallel algorithm for adaptive local refinement of tetrahedral meshes using bisection. *NUMERICAL MATHEMATICS: Theory, Methods and Applications*, 2008.

DEPARTMENT OF MATHEMATICS, UNIVERSITY OF CALIFORNIA AT IRVINE, IRVINE, CA 92697
E-mail address: chenlong@math.uci.edu