PROGRAMMING OF FINITE ELEMENT METHODS IN MATLAB

LONG CHEN

ABSTRACT. We discuss how to implement the linear finite element method for solving the Poisson equation. We begin with the data structure to represent the triangulation and boundary conditions, introduce the sparse matrix, and then discuss the assembling process. We pay special attention to an efficient programming style using sparse matrices in MATLAB.

1. DATA STRUCTURE OF TRIANGULATIONS

We shall discuss the data structure to represent triangulations and boundary conditions.

1.1. **Mesh 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 geometric embedding.

The matrix `elem` represents a set of abstract simplices. The index set $\{1, 2, \ldots, N\}$ is called the global index set of vertices. Here an vertex is thought as an abstract entity. For a simplex t, $\{1, 2, \ldots, d+1\}$ is the local index set of t. The matrix `elem` is the mapping (pointer) from the local index to the global one, i.e., `elem(t,1:d+1)` records the global indices of $d+1$ vertices which form the abstract d-simplex `t`. Note that any permutation of vertices of a simplex 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)` contains x- and y-coordinates of the k-th nodes, respectively.

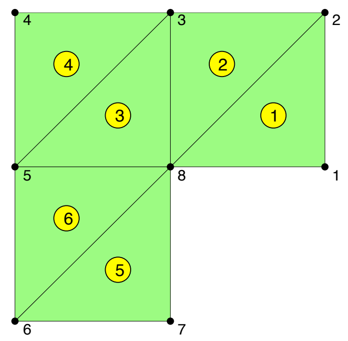
The geometric realization introduces an ordering of the simplex. For each `elem(t,:)`, we shall always order the vertices of a simplex such that the signed area is positive. That is in 2-D, three vertices of a triangle is ordered counter-clockwise and in 3-D, the ordering of vertices follows the right-hand rule.

Remark 1.1. Even with the orientation requirement, certain permutation of vertices is still allowed. Similarly any labeling of simplices in the triangulation, i.e. any permutation of rows of `elem` matrix will represent the same triangulation. The ordering of simplexes and vertices will be used to facilitate the implementation of the local mesh refinement and coarsening. See `bisect`, `coarsen`, `bisect3`, and `coarsen3` in iFEM [2]. □

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

1.2. **Boundary conditions.** We use `bdFlag(1:NT,1:d+1)` to record the type of boundary sides (edges in 2-D and faces in 3-D). The value is the type of boundary condition:

- 0 for non-boundary sides;
- 1 for the first type, i.e., Dirichlet boundary;
- 2 for the second type, i.e., Neumann boundary;



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

node

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

elem

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

(b) `node` and `elem` matrices

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

- 3 for the third type, i.e., Robin boundary.

For a d-simplex, we label its $(d-1)$-faces in the way so that the ith face is opposite to the ith vertex. For example, for a 2-D triangulation, `bdFlag(t,:) = [1 0 2]` means, the edge opposite to `elem(t,1)` is a Dirichlet boundary edge, the one to `elem(t,3)` is of Neumann type, and the other is an interior edge.

We may extract boundary edges for a 2-D triangulation from `bdFlag` by:

```
totalEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
Dirichlet = totalEdge(bdFlag(:) == 1,:);
Neumann = totalEdge(bdFlag(:) == 2,:);
```

Remark 1.2. The matrix `bdFlag` is sparse but we use a dense matrix to store it. It would save storage if we record boundary edges or faces only. The current form is convenient for the local refinement and coarsening since the boundary conditions can be easily updated along with the change of elements. We do not save `bdFlag` as a sparse matrix since updating a sparse matrix is time consuming. We set up the type of `bdFlag` to `int8` to minimize the waste of spaces. □

2. SPARSE MATRIX IN MATLAB

MATLAB is an interactive environment and high-level programming language for numeric scientific computation. One of its distinguishing features is that the main data type is the matrix. Matrices may be manipulated element-by-element, as in low-level languages like Fortran or C. But it is better to manipulate matrices at a time which will be called *high level* coding style. This style will result in more compact code and usually improve the efficiency.

We start with explanation of the sparse matrix and corresponding operations. The fast sparse matrix package and built-in functions in MATLAB will be used extensively later on. The content presented here is mostly based on Gilbert, Moler and Schereiber [4].

One of the nice features of finite element methods is the sparsity of the matrix obtained via the discretization. Although the matrix is $N\times N = N^2$, there are only cN nonzero entries in the matrix with a small constant c. Sparse matrix is the corresponding data structure to take advantage of this sparsity. Sparse matrix algorithms require less computational time by avoiding operations on zero entries and sparse matrix data structures require less memory by not storing many zero entries. We refer to the book [5] for detailed description on sparse matrix data structure and [6] for a quick introduction on popular data 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 [4].

2.1. **Storage schemes.** 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.

An intuitive 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 $j(n+1) = nnz + 1$. For example, in CSC formate, the vector to store the column pointer will be $j = [1\ 2\ 4\ 5]^{\intercal}$. This scheme is known as *Compressed Sparse Column (CSC)* scheme and is used in MATLAB sparse matrices package.

Comparing with coordinate formate, CSC formate saves storage for $nnz - n - 1$ integers which could be non-negligilble 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 [4]. On the other hand, extract one row of a sparse matrix saved in CSC will involve a search in the whole vector i and j.

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

Comparing with the dense matrix, the sparse matrix lost the direct connection between the index `(i,j)` and the physical location to store the value `A(i,j)`. Then accessing and manipulating 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 [4]. Therefore, *do not manipulate or change a sparse matrix element-by-element in a large* `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 code specific subroutines for standard matrix operations: matrix times 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 rule "time is proportional to flops" [4]: The time required for a sparse matrix operation should be proportional to the number of arithmetic operations on nonzero quantities.

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, having one nonzero for each entry in the vectors i, j, and s. The first three arguments all have the same length. However, the indices in i and j 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 the assembling procedure in 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 usage is slightly different from `sparse`. The index `[i j]` should be paired together to form a subscript vectors. 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. Assembling of The Matrix Equation

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

$$(2) \qquad -\Delta u = f \text{ in } \Omega, \qquad u = g_D \text{ on } \Gamma_D, \qquad \nabla u \cdot n = g_N \text{ on } \Gamma_N,$$

where $\partial\Omega = \Gamma_D \cup \Gamma_N$ and $\Gamma_D \cap \Gamma_N = \emptyset$. 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 (2) is: find $u \in H^1_{g,D}(\Omega)$ such that

$$(3) \quad a(u,v) := \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\boldsymbol{x} = \int_\Omega f v \, \mathrm{d}\boldsymbol{x} + \int_{\Gamma_N} g_N v \, dS \quad \text{for all } v \in H^1_{0,D}(\Omega).$$

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

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

where $\mathcal{P}_1$ is the space of linear polynomials. 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 (2) is to find $u \in \mathbb{V}_\mathcal{T} \cap H^1_{g,D}(\Omega)$ such that (3) 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 [1].

3.1. **Assembling the stiffness matrix.** For a 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

$$v = \sum_{i=1}^{N} v_i\phi_i \longleftrightarrow \boldsymbol{v} = (v_1, \cdots, v_N)^{\intercal}, \tag{4}$$

and call $\boldsymbol{v}$ the coordinate vector of v relative to the basis $\{\phi_i\}_{i=1}^{N}$. Following the terminology in the linear elasticity, 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 discuss how to form the matrix $\boldsymbol{A}$ efficiently in MATLAB.

3.1.1. *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

$$a_{i_\tau j_\tau}^{\tau} = \int_{\tau} \nabla\lambda_{j_\tau} \cdot \nabla\lambda_{i_\tau} \, \mathrm{d}\boldsymbol{x}, \text{ for } 1 \le i_\tau, j_\tau \le d+1.$$

The computation of a_{ij} will then be decomposed into the computation of local stiffness matrix and the summation over all elements. Here we use the fact that restricted to one simplex, the basis ϕ_i is identical to the barycentric coordinate λ_{i_τ} and the subscript in $a_{i_\tau j_\tau}^{\tau}$ is the local index while in a_{ij} it is the global index. The assembling process is to distribute the quantity associated to the local index to that to the global index.

Suppose we have a subroutine `locatstiffness` to compute the local stiffness matrix, 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 at least two reasons for the slow performance.

(1) The stiffness matrix `A` created in line 3 is a dense matrix which needs $\mathcal{O}(N^2)$ 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 the sparse matrix data structure. Use `A = sparse(N,N)` in line 3 will solve this problem.

(2) There is a large `for` loops with size `NT` 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 the weak point of MATLAB. Vectorization should be applied for the sake of efficiency.

We now discuss the standard procedure: transfer the computation to a reference simplex through an affine map, on computing of the local stiffness matrix. 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 $\hat{\boldsymbol{x}} = (\hat{x}, \hat{y})^{\intercal}$ for the vector in the reference 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., $F(\hat{v}_i) = v_i, i = 1,2,3$:

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

where

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

We define $\hat{u}(\hat{\boldsymbol{x}}) = u(F(\hat{\boldsymbol{x}}))$. Then $\hat{\nabla}\hat{u} = B\nabla u$ and $\mathrm{dxdy} = |\det(B)|\mathrm{d\hat{x}d\hat{y}}$. By change of variables, the integral becomes

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

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 [1] to compute the local stiffness matrix in one triangle τ.

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

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

3.1.2. *Assembling using sparse matrix.* 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` behaves like an $\mathcal{O}(N^2)$ algorithm.

We should call `sparse` command once to form the sparse matrix. The following subroutine is suggested by T. Davis [3].

```
function A = assemblingsparse(node,elem)
N = size(node,1); NT = size(elem,1);
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 inside the loop and use built-in function `sparse` to form the sparse matrix outside of 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`. This simple modification is recommended when translating C or Fortran codes into MATLAB.

3.1.3. *Vectorization.* 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 $\lambda_j(\boldsymbol{x}), j = 1, \cdots, d+1$ are linear functions of $\boldsymbol{x}$. If the j-th vertex of a simplex τ is the k-th vertex of the triangulation, 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 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 height, i.e., 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

$$\nabla\lambda_i = \frac{1}{d!\,|\tau|}\boldsymbol{n}_i, \tag{5}$$

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

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

Note that we do not normalize the normal vector since the scaled one is easier to compute. 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 using matrix operations and can be used to calculate the area of all triangles.

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

```
function A = assembling(node,elem)
N = size(node,1); NT = size(elem,1);
ii = zeros(9*NT,1); jj = zeros(9*NT,1); sA = zeros(9*NT,1);
ve(:,:,3) = node(elem(:,2),:)-node(elem(:,1),:);
ve(:,:,1) = node(elem(:,3),:)-node(elem(:,2),:);
ve(:,:,2) = node(elem(:,1),:)-node(elem(:,3),:);
area = 0.5*abs(-ve(:,1,3).*ve(:,2,2)+ve(:,2,3).*ve(:,1,2));
index = 0;
for i = 1:3
    for j = 1:3
        ii(index+1:index+NT) = elem(:,i);
        jj(index+1:index+NT) = elem(:,j);
        sA(index+1:index+NT) = dot(ve(:,:,i),ve(:,:,j),2)./(4*area);
        index = index + NT;
    end
end
A = sparse(ii,jj,sA,N,N);
```

Remark 3.1. One can further improve the efficiency by using the symmetry of the matrix. For example, the inner loop can be changed to `for j = i:3`. □

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.

```
function A = assembling3(node,elem)
N = size(node,1); NT = size(elem,1);
ii = zeros(16*NT,1); jj = zeros(16*NT,1); sA = zeros(16*NT,1);
face = [elem(:,[2 4 3]);elem(:,[1 3 4]);elem(:, [1 4 2]);elem(:, [1 2 3])];
v12 = node(face(:,2),:)-node(face(:,1),:);
v13 = node(face(:,3),:)-node(face(:,1),:);
allNormal = cross(v12,v13,2);
normal(1:NT,:,4) = allNormal(3*NT+1:4*NT,:);
normal(1:NT,:,1) = allNormal(1:NT,:);
normal(1:NT,:,2) = allNormal(NT+1:2*NT,:);
normal(1:NT,:,3) = allNormal(2*NT+1:3*NT,:);
v12 = v12(3*NT+1:4*NT,:);
v13 = v13(3*NT+1:4*NT,:);
v14 = node(elem(:,4),:)-node(elem(:,1),:);
volume = dot(cross(v12,v13,2),v14,2)/6;
index = 0;
for i = 1:4
    for j = 1:4
        ii(index+1:index+NT) = elem(:,i);
        jj(index+1:index+NT) = elem(:,j);
        sA(index+1:index+NT) = dot(normal(:,:,i),normal(:,:,j),2)./(36*volume);
        index = index + NT;
    end
end
A = sparse(ii,jj,sA,N,N);
```

The code in line 4 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 8-11 is to reshape the `4NT×3` normal vector to a `NT×3×4` matrix. Note that in line 8, we assign the value to `normal(:,:,4)` first such that the MATLAB will allocate enough memory for the array `normal` when creating it. Line 15 use the mix product of three edge vectors to compute the volume and line 19–22 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.

3.2. **Right hand side.** We define the vector $\boldsymbol{f} = (f_1, \cdots, f_N)^{\intercal}$ 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 around 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 the 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. Arbitrary order quadrature will be discussed in the next section.

We list the 2-D code below to emphasize that the command `accumarray` is used to avoid the slow `for` loop over all elements.

```
mid1 = (node(elem(:,2),:)+node(elem(:,3),:))/2;
mid2 = (node(elem(:,3),:)+node(elem(:,1),:))/2;
mid3 = (node(elem(:,1),:)+node(elem(:,2),:))/2;
bt1 = area.*(f(mid2)+f(mid3))/6;
bt2 = area.*(f(mid3)+f(mid1))/6;
bt3 = area.*(f(mid1)+f(mid2))/6;
b = accumarray(elem(:),[bt1;bt2;bt3],[N 1]);
```

3.3. **Boundary conditions.** We list the code for 2-D case and briefly explain it for the completeness. Recall that `Dirichlet` and `Neumann` are boundary edges which can be found using `bdFlag`. See Section 1.

```
1  %-------------------- Dirichlet boundary conditions------------------------
2  isBdNode = false(N,1);
3  isBdNode(Dirichlet) = true;
4  bdNode = find(isBdNode);
5  freeNode = find(¬isBdNode);
6  u = zeros(N,1);
7  u(bdNode) = g_D(node(bdNode,:));
8  b = b - A*u;
9  %-------------------- Neumann boundary conditions -------------------------
10 if (¬isempty(Neumann))
11     Nve = node(Neumann(:,1),:) - node(Neumann(:,2),:);
12     edgeLength = sqrt(sum(Nve.^2,2));
13     mid = (node(Neumann(:,1),:) + node(Neumann(:,2),:))/2;
14     b = b + accumarray([Neumann(:),ones(2*size(Neumann,1),1)], ...
15                    repmat(edgeLength.*g_N(mid)/2,2,1),[N,1]);
16 end
```

Line 2-4 will find all Dirichlet boundary nodes. The Dirichlet boundary condition is posed by assign the function values at Dirichlet boundary nodes `bdNode`. It could be found by using `bdNode = unique(Dirichlet)` but `unique` is very costly. So we use logic array to find all nodes on the Dirichlet boundary, denoted by `bdNode`. The other nodes will be denoted by `freeNode`.

The vector `u` is initialized as zero vector. Therefore after line 7, the vector `u` will represent a function $u_D \in H_{g,D}$. Writing $u = \tilde{u} + u_D$, the problem (3) 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 8. The boundary integral involving the Neumann boundary edges is computed in line 11–15 using the middle point quadrature. Note that it is vectorized 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 index 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 `freeNode`, by the direct solver

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

For the Poisson equation with pure Neumann boundary condition

$$-\Delta u = f \;\text{ in } \Omega, \quad \frac{\partial u}{\partial n} = g \text{ on } \Gamma,$$

there are two issues on the well posedness of the continuous problem:

(1) solutions are not unique. If u is a solution of Neumann problem, so is $u + c$ for any constant $c \in \mathbb{R}$. One more constraint is needed to determine this constant. A common choice is $\int_\Omega u \,\mathrm{d}x = 0$.
(2) a compatible condition for the existence of a solution. There is a compatible condition for f and g:

$$-\int_\Omega f\, dx = \int_\Omega \Delta u \,\mathrm{d}x = \int_{\partial\Omega} \frac{\partial u}{\partial n} \,\mathrm{d}S = \int_{\partial\Omega} g \,\mathrm{d}S. \tag{7}$$

We discuss the consequence of these two issues in the discretization. For Neumann problem, the stiffness matrix `A` is symmetric but only semi-definite. The kernel of `A` consists of constant vectors, i.e, the rank of `A` is `N-1`. Then `Au=b` is solvable if and only if

(8) `mean(b)=0`

which is the discrete compatible condition. If the integral is computed exactly, according to (7), (8) should hold in the discrete case. Since numerical quadrature is used to approximate the integral, (8) may hold exactly. We can enforce (8) by the modification `b = b - mean(b).`

To deal with the constant kernel of `A`, we can simply set `freeNode=2:N` and then use (6) to find values of `u` at `freeNode`. Since solution `u` is unique up to a constant, afterwards we can modify `u` to satisfy certain constraint. For example, to impose the zero average, i.e., $\int_\Omega u \,\mathrm{d}x = 0$, we could use the following code:

```
c = sum(mean(u(elem),2).*area)/sum(area);
u = u - c;
```

The H^1 error will not affect by the constant shift but when computing L^2 error, make sure the exact solution will satisfy the same constraint.

4. Numerical Quadrature

In the implementation, we need to compute various integrals on a simplex. In this section, we will present several numerical quadrature rules for simplexes in 1, 2 and 3 dimensions.

The numerical quadrature is to approximate an integral by weighted average of function values at sampling points p_i:

$$\int_\tau f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx I_n(f) := \sum_{i=1}^{n} f(p_i) w_i |\tau|.$$

The order of a numerical quadrature is defined as the largest integer k such that $\int_\tau f = I_n(f)$ when f is a polynomial of degree less than equal to k.

A numerical quadrature is determined by the quadrature points and corresponding weight: $(p_i, w_i), i = 1, \ldots, n$. For a d-simplex τ, let $\boldsymbol{x}_i, i = 1, \ldots, d+1$ be vertices of τ. The simplest one is the one point rule:

$$I_1(f) = f(c_\tau)|\tau|, \quad c_\tau = \frac{1}{d+1}\sum_{i=1}^{d+1} \boldsymbol{x}_i.$$

A very popular one is the trapezoidal rule:

$$I_1(f) = \frac{1}{d+1}\sum_{i=1}^{d+1} f(\boldsymbol{x}_i)|\tau|.$$

Both of them are of order one, i.e., exact for the linear polynomial only. For second order quadrature, in 1-D, the Simpson rule is quite popular

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\frac{1}{6}\left(f(a) + 4f((a+b)/2) + f(b)\right).$$

For a triangle, a second order quadrature, i.e., exact for quadratic polynomials, is using three middle points $m_i, i = 1, 2, 3$ of edges:

$$\int_\tau f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|\tau|}{3}\sum_{i=1}^{3} f(m_i).$$

These rules are popular due to the reason that the points and the weight are easy. No such second order rule exists for a tetrahedron in 3-D.

A criterion for choosing quadrature points is to attain a given precision with the fewest possible function evaluations. For the two (center v.s. vertices) first order quadrature rules given above, which one is more efficient? Restricting to one element, the answer is the center. When considering the evaluation over the whole triangulation, the trapezoidal rule is better since it only evaluates the function at N vertices while the center rule needs NT evaluation. It is a simple exercise to show $NT \approx 2N$ asymptotically in 2-D.

In 1-D, the Gauss quadrature use n points to achieve the order $2n-1$ which is the highest order for n points. The Gauss points are roots of orthogonal polynomials and can be found in almost all text books on numerical analysis. Quadrature rules for triangles and tetrahedron which is less well documented in the literature and we refer to [7] for a set of symmetric quadrature rules. 16 digits accurate quadrature points are included in iFEM. Type `quadpts` and `quadpts3` for the usage. We present the points in the barycentric coordinate $p = (\lambda_1, \ldots, \lambda_{d+1})$. The Cartesian coordinate of p is obtained by $\sum_{i=1}^{d+1} \lambda_i \boldsymbol{x}_i$.

REFERENCES

[1] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numerical Algorithms*, 20:117–137, 1999. 4, 6

[2] L. Chen. iFEM: An Integrated Finite Element Methods Package in MATLAB. *Technical Report, University of California at Irvine*, 2009. 1

[3] 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. 7

[4] 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. 2, 3, 4

[5] S. Pissanetsky. *Sparse matrix technology*. Academic Press, 1984. 3

[6] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003. 3

[7] L. Zhang, T. Cui, and H. Liu. A set of symmetric quadrature rules on triangles and tetrahedra. *Journal of Computational Mathematics*, 27(1):89–96, 2009. 11