FEM tools in three dimensions

Coordinator : Francisco–Javier Sayas (University of Delaware)
Developers : Thomas Brown (UD, 2015-2018)
Matthew Hassell (UD, 2015-2016)
Allan Hungria (UD, 2015-)
Tianyu Qiu (UD, 2015-2016)
Tonatiuh Sánchez-Vizuet (UD, 2015-2016)
Hasan Eruslu (UD, Spring 2016-)
Shukai Du (UD, Spring 2016-)
Ben Civiletti (UD, Fall 2016)
Jake Jacavage (UD, Fall 2016)
Collaborators : Kevin Cotter (UD, Summer 2015)
Jaspal Nijjar (UD, Summer 2015)

Project started: Spring 2015.
Last update: November 22, 2019

Contents

1 Triangulation **2**
1.1 The reference element . . . 2
1.2 A tetrahedrization . . . 3
1.3 The data structure . . . 4

2 Counting degrees of freedom **10**
2.1 Local count by element . . . 10
2.2 Global count of degrees of freedom . . . 10
2.3 Local count on the reference face . . . 15
2.4 Boundary degrees of freedom . . . 15

3 Polynomial bases **19**
3.1 Trivariate polynomials . . . 19
3.2 Bivariate polynomials . . . 22
3.3 The finite element space . . . 23
3.4 Assembly by elements . . . 24
3.5 Assembly on Neumann or Dirichlet faces . . . 25

4 Quadrature **26**
4.1 Quadrature on tetrahedra . . . 27
4.2 Quadrature on faces . . . 28
4.3 Quadrature code . . . 29

5 Mass, stiffness, convection, and load **30**
5.1 The load vector . . . 30
5.2 The mass matrix . . . 31
5.3 The stiffness matrix . . . 32

5.4 Non-symmetric stiffness matrices . . . 35
5.5 Constant coefficient stiffness matrices . . . 37
5.6 The convection matrix . . . 39
5.7 The transport matrix . . . 41
5.8 The error function . . . 43

6 Boundary conditions **45**
6.1 Computations on the boundary . . . 45
6.2 Neumann boundary conditions . . . 50
6.3 Dirichlet Boundary Conditions . . . 52

7 Sub-tetrahedrization tools **53**
7.1 Working with interfaces . . . 57
7.2 Interface matrices . . . 59
7.3 Coupling Matrices . . . 61

8 Some tools for meshing **62**
8.1 A quadrilateral mesh data-structure . . . 62
8.2 Tetrahedrization of a quad partition . . . 66
8.3 Creating a partition of the unit cube . . . 67
8.4 Creating a mesh of a rectangular prism . . . 69
8.5 Creating a mesh of a prism . . . 71
8.6 Creating a partition of the reference tetrahedron . . . 73
8.7 Creating a mesh of a axisymmetric solid . . . 74
8.8 Creating a mesh of a donut . . . 76

9 Tools to handle meshes **78**
9.1 Burying a mesh . . . 78
9.2 Refining a Tetrahedral Mesh . . . 79
9.3 Measuring the shape-regularity of a mesh . . . 81

Foreword (for internal use). This is a 'column code,' which means that geometric information is typically stored in matrices where each column corresponds to a geometric element: the coordinates of a point, the vertices of a triangle. Some of the team's recent codes are 'row codes.' It happens that access to information is very often more efficient in column codes and several annoying transpositions are avoided.

1 Triangulation

1.1 The reference element

In the reference tetrahedron

$$\widehat{K} := \{(\widehat{x},\widehat{y},\widehat{z}) \,:\, 1-\widehat{x}-\widehat{y}-\widehat{z},\widehat{x},\widehat{y},\widehat{z} \geq 0\},$$

we have the following numering of its geometric elements:

- Vertices:
$$\widehat{\mathbf{v}}_1 = (0,0,0), \qquad \widehat{\mathbf{v}}_2 = (1,0,0), \qquad \widehat{\mathbf{v}}_3 = (0,1,0), \qquad \widehat{\mathbf{v}}_4 = (0,0,1).$$
- Faces (with orientation):
$$\widehat{F}_1 := [\widehat{\mathbf{v}}_1,\widehat{\mathbf{v}}_2,\widehat{\mathbf{v}}_3], \qquad \widehat{F}_2 := [\widehat{\mathbf{v}}_1,\widehat{\mathbf{v}}_2,\widehat{\mathbf{v}}_4], \qquad \widehat{F}_3 := [\widehat{\mathbf{v}}_1,\widehat{\mathbf{v}}_3,\widehat{\mathbf{v}}_4]. \qquad \widehat{F}_4 := [\widehat{\mathbf{v}}_4,\widehat{\mathbf{v}}_2,\widehat{\mathbf{v}}_3].$$

- Edges (with orientation):

$$\widehat{E}_1 := [\widehat{\mathbf{v}}_1, \widehat{\mathbf{v}}_2], \qquad \widehat{E}_2 := [\widehat{\mathbf{v}}_2, \widehat{\mathbf{v}}_3], \qquad \widehat{E}_3 := [\widehat{\mathbf{v}}_1, \widehat{\mathbf{v}}_3].$$
$$\widehat{E}_4 := [\widehat{\mathbf{v}}_2, \widehat{\mathbf{v}}_4], \qquad \widehat{E}_5 := [\widehat{\mathbf{v}}_3, \widehat{\mathbf{v}}_4], \qquad \widehat{E}_6 := [\widehat{\mathbf{v}}_1, \widehat{\mathbf{v}}_4].$$

In the reference triangle

$$\widehat{F} := \{(s,t) \,:\, 1-s-t, s, t \geq 0\},$$

we count as follows:

- The vertices

$$\widehat{\mathbf{v}}_1 := (0,0), \qquad \widehat{\mathbf{v}}_2 := (1,0), \qquad \widehat{\mathbf{v}}_3 := (0,1),$$

- The edges (with orientation):

$$\widehat{E}_1 := [\widehat{\mathbf{v}}_1, \widehat{\mathbf{v}}_2], \qquad \widehat{E}_2 := [\widehat{\mathbf{v}}_2, \widehat{\mathbf{v}}_3], \qquad \widehat{E}_3 := [\widehat{\mathbf{v}}_3, \widehat{\mathbf{v}}_1].$$

sec:1.2

1.2 A tetrahedrization

We consider a polyhedral domain Ω with boundary Γ. The faces of Γ are grouped into the Dirichlet and Neumann boundaries Γ_D and Γ_N. General Finite Element notation:

- $\mathcal{T}_h$ is a tetrahedral partition of Ω with $N_{\rm elt}$ elements. Elements will be given with positive orientation. This means that if $[\mathbf{v}_1^K, \mathbf{v}_2^K, \mathbf{v}_3^K, \mathbf{v}_4^K]$ are the vertices of $K \in \mathcal{T}_h$ and

$$\mathbf{v}_j^K = (x_j^K, y_j^K, z_j^K) \qquad j \in \{1,2,3,4\}$$

then the transformation

$$\mathrm{F}_K : \widehat{K} \to K,$$

given by

$$\mathrm{F}_K(\widehat{x},\widehat{y},\widehat{z}) = \begin{bmatrix} x_2^K - x_1^K & x_3^K - x_1^K & x_4^K - x_1^K \\ y_2^K - y_1^K & y_3^K - y_1^K & y_4^K - y_1^K \\ z_2^K - z_1^K & z_3^K - z_1^K & z_4^K - z_1^K \end{bmatrix} \begin{bmatrix} \widehat{x} \\ \widehat{y} \\ \widehat{z} \end{bmatrix} + \begin{bmatrix} x_1^K \\ y_1^K \\ z_1^K \end{bmatrix}$$

has positive determinant.

- $\mathcal{F}_h$ is the set of all the faces of the elements of $\mathcal{T}_h$. We count $\#\mathcal{F}_h = N_{\rm fc}$. A face $F \in \mathcal{F}_h$ is given by three vertices $[\mathbf{v}_1^F, \mathbf{v}_2^F, \mathbf{v}_3^F]$ with coordinates

$$\mathbf{v}_j^F = (x_j^F, y_j^F, z_j^F).$$

The associated parametrization is

$$\begin{aligned} \boldsymbol{\phi}_F(s,t) &= (1-s-t)\mathbf{v}_1^F + s\mathbf{v}_2^F + t\mathbf{v}_3^F \\ &= \begin{bmatrix} x_1^F \\ y_1^F \\ z_1^F \end{bmatrix} + s \begin{bmatrix} x_2^F - x_1^F \\ y_2^F - y_1^F \\ z_2^F - z_1^F \end{bmatrix} + t \begin{bmatrix} x_3^F - x_1^F \\ y_3^F - y_1^F \\ z_3^F - z_1^F \end{bmatrix}. \end{aligned}$$

This parametrization defines a normal vector,

$$\mathbf{n}_F = \frac{1}{2} \begin{bmatrix} x_2^F - x_1^F \\ y_2^F - y_1^F \\ z_2^F - z_1^F \end{bmatrix} \times \begin{bmatrix} x_3^F - x_1^F \\ y_3^F - y_1^F \\ z_3^F - z_1^F \end{bmatrix},$$

which is normalized so that

$$|\mathbf{n}|_F = |F|.$$

- $\mathcal{F}_h^{\text{dir}}$ is the set of Dirichlet faces and $\mathcal{F}_h^{\text{neu}}$ is the set of Neumann faces. We write $N_{\text{dir}} := \#\mathcal{F}_h^{\text{dir}}$ and $N_{\text{neu}} := \#\mathcal{F}_h^{\text{neu}}$. *On the boundary faces,* $\mathbf{n}_F$ *points outwards.*
- $\mathcal{E}_h$ is the set of all edges of the triangulation. An edge $E \in \mathcal{E}_h$ is given as a pair of vertices $E = [\mathbf{v}_1^E, \mathbf{v}_2^E]$. This sets an orientation to the edge itself.

We will need to be careful with orientation. For edges, we can have a mismatch of orientation between local an global orientation. We will keep this is an $6 \times N_{\text{elt}}$ array with ± 1 entries. The situation is much more complicated when we think of faces. Consider the following matrix

$$\Sigma := \begin{bmatrix} 1 & 1 & 3 & 3 & 2 & 2 \\ 2 & 3 & 1 & 2 & 3 & 1 \\ 3 & 2 & 2 & 1 & 1 & 3 \end{bmatrix}$$

Each column of this matrix corresponds to a permutation of [1,2,3]. Given an element K and one of its faces F_j^K for $j = 1, 2, 3, 4$. Up to permutations of indices, this face corresponds to a single $F \in \mathcal{F}_h$. Consider then the global and local description

$$\text{global}(F) = [n_1^F, n_2^F, n_3^F], \qquad \text{local}(F_j^K) = [n_1, n_2, n_3].$$

Note that the local ordering is inherited from the element ordering and the way we describe the four faces of an element. The orientation index of the face F_j^K is

$$p \in \{1, 2, 3, 4, 5, 6\} \quad \text{such that} \quad n_i^F = n_{\Sigma(i,p)}, \quad i = 1, 2, 3.$$

The orientation matrix is a $4 \times N_{\text{elt}}$ matrix containing p associated to the j-th face of the element K.

1.3 The data structure

The triangulation is stored using a data structure `T` with the following fields:

```
T =
    coordinates: [3x8 double]
       elements: [4x6 double]
        neumann: [3x4 double]
      dirichlet: [3x8 double]
```

This means that there are 8 nodes, 6 elements (tetrahedra), 4 Neumann faces and 8 Dirichlet faces.

Triangulation with added information. The triangulation may then be expanded to include edge and face information:

```
T =
    coordinates: [3x8 double]
       elements: [4x6 double]
      dirichlet: [3x4 double]
        neumann: [3x8 double]
          edges: [3x19 double]
     edgebyelt: [6x6 double]
         faces: [4x18 double]
     facebyelt: [4x6 double]
    faceorient: [4x6 double]
```

This is the meaning of all the new fields:

- The list of edges is contained in `T.edges`. Each column represents an edge $E = E_j$; the first two entries of `T.edges(:,j)` are the node number of the first and second node of the edge. The third entry is an indicator of whether the edge is interior (0), Dirichlet (1) or Neumann (2). Note that the numbering of nodes in each of the edges creates an orientation for the edge.

- The list `T.edgebyelt` is a signed matrix containing the global edge number of each of the 6 edges of a given tetrahedron. For the element $K = K_j$, `T.elements(:,j)` contains the vertices (n_1, n_2, n_3, n_4); then `T.edgebyelt(:,j)` contains the global edge numbers for the edges

$$\{n_1, n_2\}, \{n_2, n_3\}, \{n_1, n_3\}, \{n_2, n_4\}, \{n_3, n_4\} \quad \text{and} \quad \{n_1, n_4\}$$

  in that order. The sign of `T.edgebyelt(i,j)` is positive if the local orientation of this edge matches the global orientation given in `T.edges` and negative otherwise.

- `T.faces` lists the faces of the tetrahedrization. Each column represents a face $F = F_j$; the first three entries `T.faces(1:3,j)` are the three vertices, and the last entry `T.faces(4,j)` is an indicator of whether the face F is an interior (0), Dirichlet (1) or Neumann (2) face.

- The list `T.facebyelt` contains the global face number of each of the 4 faces of a given tetrahedron. For the element $K = K_j$, `T.elements(:,j)` contains the vertices (n_1, n_2, n_3, n_4); then `T.facebyelt(:,j)` contains the global face numbers for the faces

$$\{n_1, n_2, n_3\}, \{n_1, n_2, n_4\}, \{n_1, n_3, n_4\}, \quad \text{and} \quad \{n_4, n_2, n_3\}$$

  in that order.

- The matrix `T.faceorient` is the same size as `T.facebyelt`, with each entry corresponding to one of six possible permutations of the vertices that must be applied to the locally described face to get the entry in `T.faces`. The permutation number corresponds to the column number of the matrix Σ. The meaning of this matrix was explained in Section 1.2.

When this function has been applied to a Tetrahedrization, we refer to it as an **extended** data structure.

This function needs some explanations

```
function T=edgesAndFaces(T)

% T=edgesAndFaces(T)
% Input:
%     T    : basic tetrahedrization data structure (3D tets)
% Output:
%     T    : geometric fields added
%            T.edges, T.edgebyelt, T.faces,T.faceorient,T.facebyelt
%
% Last modified: May 4, 2015.

N=size(T.coordinates,2);
Nelt=size(T.elements,2);
Ndir = size(T.dirichlet,2);
Nneu = size(T.neumann,2);


if Ndir
    edgesDir = sparse(T.dirichlet,T.dirichlet([2 3 1],:),1); % all dir face edges
    [idir, jdir] = find(triu(edgesDir+edgesDir'));
    diredges=[idir';jdir'];
    edgesDir=sparse(diredges(1,:),diredges(2,:),1,N,N);
else
    edgesDir=sparse(1,1,0,N,N);
    idir=[];
    jdir=[];
end


if Nneu
    edgesNeu = sparse(T.neumann,T.neumann([2 3 1],:),1); % all neu face edges
```

```
    [ineu, jneu] = find(triu(edgesNeu+edgesNeu')==2);
    neuedges=[ineu';jneu'];
    edgesNeu=sparse(neuedges(1,:),neuedges(2,:),1,N,N);
else
    edgesNeu=sparse(1,1,0,N,N);
    ineu = [];
    jneu = [];
end


edgesAll=sparse(T.elements([1 2 1 2 3 1],:),T.elements([2 3 3 4 4 4],:),1,N,N);
edgesAll=sign(triu(edgesAll+edgesAll'));
edgesInt=edgesAll-edgesDir-edgesNeu;

[iint, jint]=find(edgesInt);
T.edges=[iint jint zeros(size(iint,1),1);...
         idir jdir ones(size(idir,1),1);...
         ineu jneu 2*ones(size(ineu,1),1)]';
Nedge=size(T.edges,2);
edgesAll=sparse(T.edges(1,:),T.edges(2,:),1:Nedge,N,N);
edgesAll=edgesAll-edgesAll';

T.edgebyelt=...
    [full(edgesAll(sub2ind([N,N],T.elements(1,:),T.elements(2,:))));...
     full(edgesAll(sub2ind([N,N],T.elements(2,:),T.elements(3,:))));...
     full(edgesAll(sub2ind([N,N],T.elements(1,:),T.elements(3,:))));...
     full(edgesAll(sub2ind([N,N],T.elements(2,:),T.elements(4,:))));...
     full(edgesAll(sub2ind([N,N],T.elements(3,:),T.elements(4,:))));...
     full(edgesAll(sub2ind([N,N],T.elements(1,:),T.elements(4,:))))];


shape=[1 1 1 4;...
       2 2 3 2;...
       3 4 4 3];
faces=reshape(T.elements(shape(:),:),3,4*Nelt);
faces=sort(faces,1)';
[allfaces,~,j]=unique(faces,'rows');

% Lists of interior and boundary faces with references
bdfaces=sort([T.dirichlet T.neumann],1)';
[intfaces,i]=setdiff(allfaces,bdfaces,'rows');
[bdfaces,ii,jj]=intersect(allfaces,bdfaces,'rows');

%sizes
nintfaces=size(intfaces,1);
nbdfaces =size(bdfaces,1);
nfaces   =nintfaces+nbdfaces;

T.faces=[intfaces' T.dirichlet T.neumann;
         zeros(1,nintfaces) ones(1,Ndir) 2*ones(1,Nneu)];

% Backward referencing to construct T.facebyelt
u=zeros(nfaces,1);
v=nintfaces+1:nintfaces+nbdfaces;
u(i)=1:nintfaces;
u(ii)=v(jj);
j=u(j);          % reconstructing list of all faces (4*Nelt) now with global numbering
T.facebyelt=reshape(j,[4 Nelt]);


% Matrix with permutation order
eq = @(u,v) sum(u==v,1)==3; % checks what rows are equal
rot=[1 1 3 3 2 2;...
     2 3 1 2 3 1;...
     3 2 2 1 1 3];       %permutations
```

```
T.faceorient=zeros(4,Nelt);
for f=1:4    % counter over faces
    faceGlobal=T.faces(1:3,T.facebyelt(f,:));
    faceLocal =T.elements(shape(:,f),:);
    for j=1:6
        T.faceorient(f,:)=T.faceorient(f,:)+j*eq(faceGlobal,faceLocal(rot(:,j),:));
    end
end

return
```

What now follows is a detailed explanation of how the above code creates the fields described above.

When `T.neumann` and/or `T.dirichlet` are nonempty, we create sparse $(0,1)$-matrices to represent the edges of the Neumann and Dirichlet faces. In the case of the Neumann faces, an edge is only considered a Neumann edge if it appears in the list twice, because otherwise the edge is shared by a Neumann face and a Dirichlet face and is considered a Dirichlet edge. For each type of edge, index lists are created consisting of the vertex numbers which serve as the two endpoints of the edge.

After these lists are created, a sparse $(0,1)$-matrix consisting of all edges in the tetrahedrization is created. Subtracting the Dirichlet and Neumann edges from this matrix gives us a list of all interior edges. The field `T.edges` is then created with the lists representing the three types of edges.

Having given a global numbering and orientation of edges with the creation of `T.edges` a new sparse matrix is created, indexed by vertices, containing the global edge number and orientation. That is if e_j is global edge number j, with endpoints $e_j^{(1)}$ and $e_j^{(2)}$ then in this new matrix, called `edgesAll` in the code, we have

$$\texttt{edgesAll}(e_j^{(1)}, e_j^{(2)}) = j \qquad \text{and} \qquad \texttt{edgesAll}(e_j^{(2)}, e_j^{(1)}) = -j.$$

This matrix is then used to create `T.edgebyelt` by reading its entries based on local edge orientation.

A list of faces is created from `T.elements` and all duplicate faces are removed by sorting. From this list of faces, the faces contained in `T.neumann` and `T.dirichlet` are removed, creating a list of interior faces. Using this list, along with `T.neumann` and `T.dirichlet`, the field `T.faces` is created giving each face a global numbering (by column) and global orientation (by order of the vertices in the first three rows).

At the same time the list of faces is being created, a vector, j, is created which contains information for placing the globally numbered faces back into local ordering. A vector, u is created with a global numbering of each face, and then $u(j)$ is vector of length $4 * N_{\text{elt}}$ which gives the global face numbers arranged locally by element. This vector is reshaped to create `T.facebyelt`.

To create the field `T.faceorient`, the following are used:

- an indicator function `eq` such that

$$\texttt{eq}(\mathbf{a}, \mathbf{b}) = \begin{cases} 1, & \text{if } \mathbf{a} = \mathbf{b} \\ 0, & \text{otherwise} \end{cases}.$$

- The permuation matrix Σ described in Section 1.2, called `rot` in the code
- The global orientation of each face
- The local orientation of the faces in each element

Orientations are assigned by permuting the local orientation of the face until it matches the global orientation of the face. For example, let $\{\sigma_j\}_{j=1}^6$ represent the columns of Σ, then a face represented by both global(F) and local(F_ℓ^K) is assigned an orientation of j if and only if

$$\text{global}(F) = \text{local}(F_\ell^K)(\sigma_j).$$

The triangulation may then be further expanded to include necessary geometric information:

```
T =
    coordinates: [3x8 double]
       elements: [4x6 double]
      dirichlet: [3x4 double]
        neumann: [3x8 double]
          edges: [3x19 double]
      edgebyelt: [6x6 double]
          faces: [4x18 double]
      facebyelt: [4x6 double]
     faceorient: [4x6 double]
         volume: [1x6 double]
           area: [1x18 double]
        normals: [3x18 double]
```

This is the meaning of all new fields:

- `T.volume` contains the volume of each element K, listed by the same numbering as `T.elements`.
- `T.area` contains the area of each face, listed by the same numbering as `T.faces`.
- `T.normals` is $3 \times$ `nFaces` and contains the normal vector for each face. The columns of `T.normals` correspond to the same ordering of faces in `T.faces`. That is, column k contains the normal to the face defined in column k of `T.faces` The length of the normal vector equals the area of the face. Boundary face normals point outwards.

When this function has been applied to a data structure, we refer to it as an **enhanced** data structure.

```
function [ T ] = enhanceGrid3D( T )
% function [ T ] = enhanceGrid3D( T )
%
%Input:
%     T : Extended tetrahedral data structure
%
% Output:
%     T : Enhanced tetrahedral data structure with T.volume, T.normal,
%           T.area
%
% Last modified: Apr 10, 2015

nelts = size(T.elements,2);

T.volume=(1/6)*...
 ((T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:))).*...
     (T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:))).*...
     (T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:)))-...
  (T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:))).*...
     (T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:))).*...
     (T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:)))-...
  (T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:))).*...
     (T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:))).*...
     (T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:)))+...
  (T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:))).*...
     (T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:))).*...
     (T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:)))+...
  (T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:))).*...
     (T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:))).*...
     (T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:)))-...
  (T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:))).*...
     (T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:))).*...
     (T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:))));

T.area=(1/2)*sqrt(...
 ((T.coordinates(2,T.faces(2,:))-T.coordinates(2,T.faces(1,:))).*...
  (T.coordinates(3,T.faces(3,:))-T.coordinates(3,T.faces(1,:)))-...
```

```
  (T.coordinates(3,T.faces(2,:))-T.coordinates(3,T.faces(1,:))).*...
  (T.coordinates(2,T.faces(3,:))-T.coordinates(2,T.faces(1,:)))).^2+...
 ((T.coordinates(1,T.faces(2,:))-T.coordinates(1,T.faces(1,:))).*...
  (T.coordinates(3,T.faces(3,:))-T.coordinates(3,T.faces(1,:)))-...
  (T.coordinates(3,T.faces(2,:))-T.coordinates(3,T.faces(1,:))).*...
  (T.coordinates(1,T.faces(3,:))-T.coordinates(1,T.faces(1,:)))).^2+...
 ((T.coordinates(1,T.faces(2,:))-T.coordinates(1,T.faces(1,:))).*...
  (T.coordinates(2,T.faces(3,:))-T.coordinates(2,T.faces(1,:)))-...
  (T.coordinates(2,T.faces(2,:))-T.coordinates(2,T.faces(1,:))).*...
  (T.coordinates(1,T.faces(3,:))-T.coordinates(1,T.faces(1,:)))).^2);

% Normals to the faces
XX = reshape(T.coordinates(1,T.faces(1:3,:)),size(T.faces(1:3,:)));
YY = reshape(T.coordinates(2,T.faces(1:3,:)),size(T.faces(1:3,:)));
ZZ = reshape(T.coordinates(3,T.faces(1:3,:)),size(T.faces(1:3,:)));

VX1 = XX(2,:)-XX(1,:);
VY1 = YY(2,:)-YY(1,:);
VZ1 = ZZ(2,:)-ZZ(1,:);

VX2 = XX(3,:)-XX(1,:);
VY2 = YY(3,:)-YY(1,:);
VZ2 = ZZ(3,:)-ZZ(1,:);

V1 = [VX1;VY1;VZ1];
V2 = [VX2;VY2;VZ2];

T.normals = (1/2)*cross(V1',V2')';

return
```

2 Counting degrees of freedom

From now on $k \geq 1$ will be the polynomial degree used for the FEM code.

2.1 Local count by element

sec:2.1

On the reference element we will count using a four-vector:

$$\boldsymbol{\alpha} := (\alpha_1, \alpha_2, \alpha_3, \alpha_4), \qquad \alpha_j \geq 0, \qquad \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = k.$$

This format is related to the Lagrangian points

$$\tfrac{1}{k}(\alpha_2, \alpha_3, \alpha_4),$$

although the basis that we will be using is not a Lagrange basis. We are going to need the following numbers

$$\begin{aligned}
d_k &:= \dim \mathbb{P}_k := \binom{k+3}{3} = \frac{(k+3)(k+2)(k+1)}{6}, \\
e_k &:= k-1, \\
f_k &:= \binom{k-1}{2} = \frac{(k-1)(k-2)}{2}, \\
t_k &:= \binom{k-1}{3} = \frac{(k-1)(k-2)(k-3)}{6} = \dim \mathbb{P}_k - 4 - 6e_k - 4f_k.
\end{aligned}$$

The quantity t_k is the number of internal degrees of freedom and the quantity f_k is the number of internal degrees of freedom on the faces. We number the multi-indices following the next table:

$(k,0,0,0)$			:	1
$(0,k,0,0)$			:	2
$(0,0,k,0)$			:	3
$(0,0,0,k)$			:	4
$(k-j,j,0,0)$,	$j=1,\ldots,k-1$		:	$4+\{1,\ldots,e_k\}$
$(0,k-j,j,0)$,	$j=1,\ldots,k-1$		:	$4+e_k+\{1,\ldots,e_k\}$
$(k-j,0,j,0)$,	$j=1,\ldots,k-1$		:	$4+2e_k+\{1,\ldots,e_k\}$
$(0,k-j,0,j)$,	$j=1,\ldots,k-1$		:	$4+3e_k+\{1,\ldots,e_k\}$
$(0,0,k-j,j)$,	$j=1,\ldots,k-1$		:	$4+4e_k+\{1,\ldots,e_k\}$
$(k-j,0,0,j)$,	$j=1,\ldots,k-1$		:	$4+5e_k+\{1,\ldots,e_k\}$
$(k-i-j,i,j,0)$,	$i=1,\ldots,k-2$,	$j=1,\ldots,k-1-i$	:	$4+6e_k+\{1,\ldots,f_k\}$
$(k-i-j,i,0,j)$,	$i=1,\ldots,k-2$,	$j=1,\ldots,k-1-i$	:	$4+6e_k+f_k+\{1,\ldots,f_k\}$
$(0,i,j,k-i-j)$,	$i=1,\ldots,k-2$,	$j=1,\ldots,k-1-i$	:	$4+6e_k+2f_k+\{1,\ldots,f_k\}$
$(k-i-j,0,i,j)$,	$i=1,\ldots,k-2$,	$j=1,\ldots,k-1-i$	:	$4+6e_k+3f_k+\{1,\ldots,f_k\}$
$(k-i-j-\ell,i,j,\ell)$,	$i=1,\ldots,k-2$,	$j=1,\ldots,k-1-i$	:	$4+6e_k+4f_k+\{1,\ldots,t_k\}$
	$\ell=1,\ldots,k-1-i-j$			

Note that we first count the vertices, then the edges (following orientation), then the faces (following internal orientation), and finally the interior of the element.

2.2 Global count of degrees of freedom

sec:2.2

The dimension of the FEM space

$$V_h = \{u_h \in \mathcal{C}(\Omega) \,:\, u_h|_K \in \mathbb{P}_k(K) \quad \forall K \in \mathcal{T}_h\}$$

is

$$\begin{aligned}\dim V_h &= N_{\rm nd} + (k-1)N_{\rm edg} + \frac{(k-1)(k-2)}{2}N_{\rm fc} + \frac{(k-1)(k-2)(k-3)}{6}N_{\rm elt} \\ &= N_{\rm nd} + e_k N_{\rm edg} + f_k N_{\rm fc} + t_k N_{\rm elt}.\end{aligned}$$

For ease of access to this number, it is computed in the function `dimFEMspace.m`.

```
function d=dimFEMspace(T,k)

% d=dimFEMspace(T,k)
%
% Input:
%    T    :  tetrahedrization
%    k    :  polynomial degree
% Output:
%    d    :  dimension of the Pk FEM space
% Last modified: July 8, 2016

if ¬isfield(T,'edges')
    T=edgesAndFaces(T);
end

Nnod=size(T.coordinates,2);
Nelt=size(T.elements,2);
Nedg=size(T.edges,2);
Nfac=size(T.faces,2);

dimEdges=(k-1)*(k>1);
dimFaces=((k-1)*(k-2))/2*(k>2);
dimInt  =((k-1)*(k-2)*(k-3))/6*(k>3);

d = Nnod+Nedg*dimEdges+Nfac*dimFaces+Nelt*dimInt;

return
```

Globally we count:

- First all vertices, in the order set by `T.coordinates`. These will be the first $N_{\rm nd}$ degrees of freedom.
- Then $e_k = k-1$ degrees of freedom on each edge, in the order set by `T.edges`. The edges have a given orientation, which will be used to organize the assembly process. These are the degrees of freedom from $N_{\rm nd}+1$ to $N_{\rm nd} + e_k N_{\rm edg}$.
- Next $f_k = (k-1)(k-2)/2$ degrees of freedom on each face, in the order set by `T.faces`. The faces have a given orientation, which will be used to organize the assembly process. These are the degrees of freedom from $N_{\rm nd} + e_k N_{\rm edg} + 1$ to $N_{\rm nd} + e_k N_{\rm edg} + f_k N_{\rm fc}$.
- The t_k degrees of freedom on each element, in the order set by `T.elements`. The elements have an orientation, which will be used to organize the assembly process. These are the degrees of freedom from $N_{\rm nd} + e_k N_{\rm edg} + f_k N_{\rm fc} + 1$ to $N_{\rm nd} + e_k N_{\rm edg} + f_k N_{\rm fc} + t_k N_{\rm elt}$.

The d_k local DOF of an element are a subset of the global DOF. The function `DOF3D.m` returns a $d_k \times N_{\rm elt}$ matrix with the list of local to global DOF. The column number K of the matrix contains first the DOF corresponding to the vertices, then, the DOF corresponding to the edges (grouped by edge, and correctly oriented), and finally the DOF corresponding to the element.

A block matrix. We can think of DOF as a $d_k \times N_{\text{elt}}$ matrix containing the global degrees of freedom corresponding to each of the elements. This matrix can be broken into four blocks:

$$\text{DOF} = \begin{bmatrix} \text{vDOF} \\ \text{eDOF} \\ \text{fDOF} \\ \text{tDOF} \end{bmatrix}$$

where vDOF is $4 \times N_{\text{elt}}$, eDOF is $e_k \times N_{\text{elt}}$, vDOF is $f_k \times N_{\text{elt}}$ and tDOF is $t_k \times N_{\text{elt}}$. The second block first appears with $k = 2$, the third with $k = 3$ and the fourth with $k = 4$. We can think of each of the blocks as counting from 1 by doing the following decomposition

$$\text{eDOF} = N_{\text{nd}} + \text{eDOF}^{\circ}, \quad \text{vDOF} = N_{\text{nd}} + e_k N_{\text{edg}} + \text{vDOF}^{\circ}, \quad \text{tDOF} = N_{\text{nd}} + e_k N_{\text{edg}} + f_k N_{\text{fc}} + \text{tDOF}^{\circ}.$$

Here is how the blocks are produced:

- The vertex block is just `T.elements`.
- The element block tDOF° is a $t_k \times N_{\text{elt}}$ matrix with the numbers $\{1, \ldots, t_k N_{\text{elt}}\}$ written by columns of the matrix.
- To compute eDOF°, we start by creating the $6N_{\text{elt}}$ vector

  $$\mathbf{e} := e_k(\texttt{abs(T.edgebyelt(:))} - 1).$$

  We then create the $e_k \times (6N_{\text{elt}})$ matrix

  $$\begin{bmatrix} 1 + \mathbf{e}^{\top} \\ 2 + \mathbf{e}^{\top} \\ \vdots \\ e_k + \mathbf{e}^{\top} \end{bmatrix}.$$

  Next, the columns corresponding to negative orientation (which can be identified by having negative entries in the vector `T.edgebyelt(:)` are flipped. Finally, the matrix is reshaped to a $(6e_k) \times N_{\text{elt}}$ matrix, where each column contains the edge degrees of freedom of an element.
- We start the computation of fDOF° by creating a $4N_{\text{elt}}$ vector

  $$\mathbf{f} := f_k(\texttt{T.facebyelt(:)} - 1),$$

  and then creating the $f_k \times (4N_{\text{elt}})$ matrix

  $$\text{F} := \begin{bmatrix} 1 + \mathbf{f}^{\top} \\ 2 + \mathbf{f}^{\top} \\ \vdots \\ f_k + \mathbf{f}^{\top} \end{bmatrix}.$$

  We have next to go column by column a make some sort of permutation depending on the orientation of the face. (This is explained in the next paragraph.) Finally, the resulting matrix is reshaped to a $(4f_k) \times N_{\text{elt}}$ matrix, where each column contains the face degrees of freedom of an element.

How to reorient faces. We first create the matrix R with rows

$$r_{\ell} = \begin{bmatrix} k-i-j & i & j \end{bmatrix} \qquad \begin{matrix} i = 1, \ldots, k-2, \\ j = 1, \ldots, k-1-i \end{matrix} \qquad \ell = 1, \ldots, f_k.$$

For instance, for $k = 5$, this matrix is

$$\mathrm{R} := \begin{bmatrix} 3 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 1 & 3 \\ 2 & 2 & 1 \\ 1 & 2 & 2 \\ 1 & 3 & 1 \end{bmatrix}$$

The matrices R_p for $p \in \{1, 2, 3, 4, 5, 6\}$ are obtained by reading the columns of R using the permutation given by the p-th column of Σ. Obviously $\mathrm{R}_1 = \mathrm{R}$. For example, with $k = 5$ the 5-th permutation is $[2, 3, 1]$ and therefore

$$\mathrm{R}_5 = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 2 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \\ 3 & 1 & 1 \end{bmatrix}.$$

We now create six f_k-vectors $\mathbf{s}_p$ with the following information:

$$s_p(i) = j \text{ when the } i\text{-th row of R appears in the } j\text{-th row of } \mathrm{R}_p.$$

The subfunction `rotateFace` creates these six vectors are stored them in an $f_k \times 6$ matrix. For instance, for $k = 5$ these vectors are the columns of

$$\left[\begin{array}{c|c|c|c|c|c} 1 & 1 & 3 & 3 & 6 & 6 \\ 2 & 4 & 5 & 2 & 4 & 5 \\ 3 & 6 & 6 & 1 & 1 & 3 \\ 4 & 2 & 2 & 5 & 5 & 4 \\ 5 & 5 & 4 & 4 & 2 & 2 \\ 6 & 3 & 1 & 6 & 3 & 1 \end{array}\right]$$

Now, if a column $\mathbf{c}$ of F, corresponding to a face of an element with local orientation $p \in \{1, 2, 3, 4, 5, 6\}$ (we have this information in `T.faceorient`), then we reassign:

$$\mathbf{c}(\mathbf{s}_p) = \mathbf{c}.$$

Why is this so? The proper way of counting points to a face F follows the rows of the matrix R (understanding the indices as discrete barycentric coordinates). However, if this locally this face corresponds to an orientation p, we need to read the vector following $\sim_p$ to find where the points with the same discrete barycentric coordinates are.
We also deliver a row–assembly three dimensional array (called Cols), of size $d_k \times d_k \times N_{\mathrm{elt}}$. The $d_k \times d_k$ slice corresponding to the third index fixed to K contains d_k copies of a single row, with the same elements as the K column of the DOF matrix.

```
function [localDOF,Cols]=DOF3D(T,k)

% function [localDOF,Cols]=DOF3D(T,k)
% Input:
%      T        : Data structure, enhanced FEM tetrahedrization
% Output:
%      localDOF : dimPk x Nelt matrix with local DOF
%      Cols     : 3D array for assembly
% Last modified: May 1, 2015

Nnod=size(T.coordinates,2);
Nelt=size(T.elements,2);
Nedg=size(T.edges,2);
```

```
Nfac=size(T.faces,2);

fulldim=((k+3)*(k+2)*(k+1))/6;
dimEdges=(k-1)*(k>1);
dimFaces=((k-1)*(k-2))/2*(k>2);
dimInt  =((k-1)*(k-2)*(k-3))/6*(k>3);

vDOF=T.elements;
eDOF=[];
if k>1
    eDOF=(abs(T.edgebyelt)-1)*dimEdges+Nnod;
    eDOF=eDOF(:)';
    eDOF=bsxfun(@plus,(1:dimEdges)',eDOF);
    negative=find(sign(T.edgebyelt(:))==-1);
    eDOF(:,negative)=eDOF(dimEdges:-1:1,negative);
    eDOF=reshape(eDOF,[6*dimEdges,Nelt]);
end
fDOF=[];
if k>2
    fDOF=(T.facebyelt-1)*dimFaces+Nnod+dimEdges*Nedg;
    fDOF=fDOF(:)';
    fDOF=bsxfun(@plus,(1:dimFaces)',fDOF);
    rotation=rotateFace(k);
    for j=2:6
        perm=find(T.faceorient==j);
        fDOF(rotation(:,j),perm)=fDOF(:,perm);
    end
    fDOF=reshape(fDOF,[4*dimFaces,Nelt]);
end
tDOF=[];
if k>3
    tDOF=((1:Nelt)-1)*dimInt+Nnod+dimEdges*Nedg+dimFaces*Nfac;
    tDOF=bsxfun(@plus,(1:dimInt)',tDOF);
end
localDOF=[vDOF;eDOF;fDOF;tDOF];
Cols=repmat(localDOF(:)',[fulldim 1]);
Cols=reshape(Cols,[fulldim,fulldim,Nelt]);

return

function rotation=rotateFace(k)

index = 1;
for ind2 = 1:k-2
    for ind3 = 1:k-1-ind2
        ind1 = k-ind2-ind3;
        table(index,:) = [ind1 ind2 ind3];
        index = index +1;
    end
end
perm = [ 1 2 3;...
         1 3 2;...
         3 1 2;...
         3 2 1;...
         2 3 1;...
         2 1 3];
dim=((k-1)*(k-2))/2;
rotation=zeros(dim,6);
for j= 1:6
    [~,list] = ismember(table,table(:,perm(j,:)),'rows');
    rotation(:,j) = list;
end
return
```

2.3 Local count on the reference face

On the reference face we will count using a three-vector:

$$\boldsymbol{\alpha} := (\alpha_1, \alpha_2, \alpha_3), \qquad \alpha_j \geq 0, \qquad \alpha_1 + \alpha_2 + \alpha_3 = k.$$

This format is related to the Lagrangian points

$$\tfrac{1}{k}(\alpha_2, \alpha_3),$$

although the basis that we will be using is not a Lagrange basis. We are going to need the following numbers

$$\begin{aligned} \delta_k &:= \dim \mathbb{P}_k(\widehat{F}) := \binom{k+2}{2} = \frac{(k+2)(k+1)}{2}, \\ e_k &:= k-1, \\ f_k &:= \binom{k-1}{2} = \frac{(k-1)(k-2)}{2}. \end{aligned}$$

The quantity f_k is the number of internal degrees of freedom on the faces. We number the multi-indices following the next table:

$(k,0,0)$,	:	1
$(0,k,0)$,	:	2
$(0,0,k)$,	:	3
$(k-j,j,0)$, $\quad j=1,\ldots,k-1$	:	$3+\{1,\ldots,e_k\}$
$(0,k-j,j)$, $\quad j=1,\ldots,k-1$	:	$3+e_k+\{1,\ldots,e_k\}$
$(j,0,k-j)$, $\quad j=1,\ldots,k-1$	:	$3+2e_k+\{1,\ldots,e_k\}$
$(k-i-j,i,j)$, $\quad i=1,\ldots,k-2$, $\quad j=1,\ldots,k-1-i$	:	$3+3e_k+\{1,\ldots,f_k\}$

Note that we first count the vertices, then the edges (following orientation), and finally the interior of the face.

2.4 Boundary degrees of freedom

Given a data structure for a mesh `T`, a polynomial degree k, and a list of faces (not necessarily on the boundary, but that is the natural application), the function `computeBDDOF3D.m` returns two things :

- a matrix containing the global numbering of the degrees of freedom on each face in the list by column. That is the degrees of freedom located on F_i in the list are located in column i of `listDOF`.
- the three dimensional array of information needed to assemble a matrix using the degrees of freedom on the faces desired.

On each face, we count the degrees of freedom locally following the following scheme:

- First the degrees of freedom on the vertices, **in the order given by `T.faces`**
- Then the degrees of freedom on the edges, moving along the edges starting at the first vertex and moving towards the second, then the third one and closing the loop back in the first one **(according to the order given by `T.faces`)**.
- Finally the degrees of freedom in the faces. **Counting from bottom to top and left to right** with respect to the first edge (i.e. the one determined by the first two vertices on `T.faces`), as shown schematically in Figure 2.4.

`bdDOF`

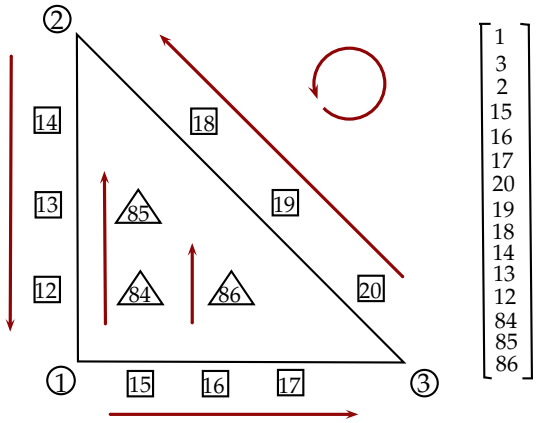

Figure 1: Example taken from a two-element tetrahedrization of a cube, with $k = 4$. The column of `listDOF` corresponding to this face is shown on the right. The degrees of freedom follow the order shown by the arrows, as discussed above. The orientation of the loop is determined by the order in which the vertices appear in `T.faces`, in this example, the face was given as `[1; 3 ; 2]`.

Note that this is just the same count on the physical element that was used on the reference element. Globally we will count in a very similar way to the global DOF on the elements. We will count

- First all of the vertices
- Then, all of the edges following their orientation
- Finally, the interior DOF for the faces.

The vertices of all faces in the list are taken as the first three rows of the `listDOF` matrix. If the polynomial degree is more than one, then the degrees of freedom on the edges are included following the orientation discussed in the previous paragraph.

A key part of the process is the formation of an auxiliary matrix `EdgebyFace` which gives, columnwise, the edges forming each face (i.e. $\texttt{EdgebyFace}_{ij}$ is the global number of the i-th edge of the j-th face) and as a companion, the matrix of orientations of each face, `Orient`, is created. This matrix indicates if the edge is traversed in positive order (increasing vertex numbering) or in negative order (decreasing vertex numbering).

Using only the columns of `EdgebyFace` corresponding to faces in the list, the DOF's on the edges are then added as rows of the matrix, following the global numbering convention, but ordered according to the orientation of the edge (i.e. increasingly if the edge is positively oriented or decreasing if the edge is negatively oriented). Finally, for polynomial degree greater than 2, the face DOFs are added as the last rows of the matrix.

The matrix `listDOF` gives, on each column, the degrees of freedom on each face in the list ordered and oriented as explained above and represented in figure 2.4.

To create the array `assembleDOF`, we first compute the number of degrees of freedom located on each face (note that this should also be the number of rows in `listDOF`), which we will denote $\dim P_k(F)$. Making `listDOF` into a long vector, we copy it $\dim P_k(F)$ times and then reshape these copies into a $\dim P_k(F) \times \dim P_k(F) \times$ (number of faces in our list) array of column information needed to assemble a (mass or other) matrix on these faces.

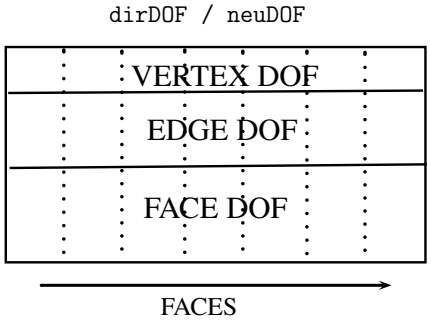

Figure 2: Scheme of the distribution of degrees of freedom in the matrix `listDOF`.

```
function [listDOF,assembleDOF]=computeBDDOF3D(T,k,list)

% [listDOF,asseDOF]=computeBDDOF3D(T,k,list)
%
% Input:
%    T       : full tetrahedrization
%    k       : polynomial degree
%    list    : row vector with numbers from 1 to Nfac
% Output:
%    listDOF : dim P_k(F) x #list matrix with global DOF for listed faces
%    asseDOF : dim P_k(F) x dim P_k(F) x #list COLUMN assembly array
%
% Last modified: January 18, 2017

Nnod = size(T.coordinates,2);
Nedg = size(T.edges,2);

%  Edgebyface
Edges = T.edges(1:2,:)';
[E1,I1] = sort(T.faces([1 2],:)',2);
[E2,I2] = sort(T.faces([2 3],:)',2);
[E3,I3] = sort(T.faces([3 1],:)',2);
[¬,EBF1] = ismember(E1,Edges,'rows');
[¬,EBF2] = ismember(E2,Edges,'rows');
[¬,EBF3] = ismember(E3,Edges,'rows');
EdgebyFace = [EBF1 EBF2 EBF3]';
Orient = [I1(:,2)-I1(:,1) I2(:,2)-I2(:,1) I3(:,2)-I3(:,1)]';
listEdges = EdgebyFace(:,list); listEdges = listEdges(:)';
listOrient = Orient(:,list); listOrient = listOrient(:)';

% Nodal DOF
listnodeDOF=T.faces(1:3,list);

% Edge DOF (orientation enforced in second line)
listedgeDOF = bsxfun(@plus,(k-1)*(listEdges-1) +Nnod,(1:k-1)');
listedgeDOF(:,listOrient<0) = flipud(listedgeDOF(:,listOrient<0));
listedgeDOF = reshape(listedgeDOF,3*(k-1),length(list));

% Face DOF (offset by nodal and edge DOF)
dimk=(k-1)*(k-2)/2; % number of internal/face DOF
listfaceDOF = bsxfun(@plus,dimk*(list-1)+Nnod+(k-1)*Nedg,(1:dimk)');
listfaceDOF = reshape(listfaceDOF,dimk,length(list));

listDOF     = [listnodeDOF; listedgeDOF; listfaceDOF];

% Assembly matrix (columns only)

dk = 3+3*(k-1)+(k-1)*(k-2)/2;   % dim P_k(F)
```

```
assembleDOF = repmat(listDOF(:)',[dk,1]);
assembleDOF = reshape(assembleDOF,[dk,dk,length(list)]);

end
```

What follows next is using the above function (`computeBDDOF3D`) applied to Dirichlet and Neumann faces. Using the convenient tag located in the fourth row of `T.faces` (Dirichlet faces are tagged with a 1, Neumann faces are tagged with a 2), we can extract the global face numbers of each kind of face. These global face numberings are stored in `dirFaces` and `neuFaces` respectively.
Now calling `computeBDDOF3D` twice (once with `dirFaces` and once with `neuFaces` as our 'list'), we obtain

- the $\dim P_k(F) \times N_{\text{dir}}$ and $\dim P_k(F) \times N_{\text{neu}}$ matrices which contain the global numbering of Dirichlet and Neumann degrees of freedom. Each column represents a different face.
- the $\dim P_k(F) \times \dim P_k(F) \times N_{\text{neu}}$ array of column information needed to assemble a boundary mass matrix on the Neumann faces.

The function also provides a list of global Dirichlet degrees of freedom in `dirlist`, where the global numbering of faces is determined implicitly by the columns of the matrix `T.faces`.

The vector `free` contains a list of **all the non-Dirichlet** degrees of freedom, which is created by making a list of numbers from 1 to the dimension of our finite element space and eliminating the numbers contained in `dirlist`.
Remark: The degrees of freedom lying on Dirichlet/Neumann transition edges and vertices are counted twice, and appear on both `DirDOF` and `NeuDOF` as well as in the lists, but those Neumann DOF's that are not on transition edges or vertices are considered free and are listed as such in `free`.

```
function [DirDOF,NeuDOF,NeuAssem,dirFaces,neuFaces,dirlist,free] = bdDOF3D(T,k)

% [DirDOF,NeuDOF,NeuAssem,dirFaces,neuFaces,dirlist,free] = bdDOF3D(T,k)
%
% Input:
% T             : Data structure. Enhanced 3D FEM tetrahedrization
% k             : Polynomial degree
% Output:
% DirDOF        : dim P_k(F) x NDirFaces. Matrix
%                 containing global boundary DOF per Dirichlet face counted
%                 columnwise.
% NeuDOF        : dim P_k(F) x NNeuFaces. Matrix
%                 containing global boundary DOF per Neumann face counted
%                 columnwise.
% NeuAssem      : dim P_k(F) x dim P_k(F) x NneuFaces. Neumann-assembly matrix
% dirFaces      : 1 x NdirFaces. Vector containing the global numbering of
%                 the Dirichlet faces.
% neuFaces      : 1 x NneuFaces. Vector containing the global numbering of
%                 the Neumann faces.
% dirlist       : NdirDOF x 1. List of Dirichlet DOF
% free          : Nfree x 1. vector containing the global numbering of the
%                 free DOF.
%
% NOTE: DOF on Dirichlet/Neumann transition nodes or edges are counted
% twice, appearing in both DirDOF and NeuDOF.
%
% Last modified: January 18, 2017

dirFaces = find(T.faces(4,:)==1);
neuFaces = find(T.faces(4,:)==2);
[DirDOF,¬] = computeBDDOF3D(T,k,dirFaces);
[NeuDOF,NeuAssem] = computeBDDOF3D(T,k,neuFaces);

dirlist=unique(DirDOF(:));
free   =(1:dimFEMspace(T,k))'; free(dirlist)=[];
```

```
return
```

3 Polynomial bases

3.1 Trivariate polynomials

Counting and evaluation. The basis in the reference element will be numbered using the four-index format of Section 2.1. The Bernstein-Bézier basis of degree $k \geq 0$ is defined as

$$B_{\boldsymbol{\alpha}}^{(k)} := \binom{k}{\boldsymbol{\alpha}} \boldsymbol{\lambda}^{\boldsymbol{\alpha}}, \qquad \boldsymbol{\alpha} \in \mathbb{N}^4, \qquad |\boldsymbol{\alpha}| = k.$$

In the reference element, this is just

$$\frac{k!}{\alpha_1!\alpha_2!\alpha_3!\alpha_4!}(1-\widehat{x}-\widehat{y}-\widehat{z})^{\alpha_1}\widehat{x}^{\alpha_2}\widehat{y}^{\alpha_3}\widehat{z}^{\alpha_4}.$$

In the current version of the code, this is evaluated in a naive way, without using the De Casteljau algorithm.

```
    function [P,indices]=bernstein3D(x,y,z,k)

% [P,ind]=bernstein3D(x,y,z,k)
%
% Input:
%    [x,y,z]: three M x 1 vectors with coordinates in the reference element
%    k      : polynomial degree
% Output:
%    P      : M x dk matrix with Pk basis evaluated at points
%    ind    : 4d array to find where four-index BB basis is located at
% Last modified: April 9, 2015

if k==0
    P=ones(size(x));
    indices=1;
    return
end

lambda1=bsxfun(@power,1-x-y-z,0:k);
lambda2=bsxfun(@power,x,0:k);
lambda3=bsxfun(@power,y,0:k);
lambda4=bsxfun(@power,z,0:k);

indices = location(k);
ind = @(i1,i2,i3,i4) indices(i1+1,i2+1,i3+1,i4+1);

P=zeros(size(x,1),((k+1)*(k+2)*(k+3))/6);
for a1=0:k
    for a2=0:k-a1
        for a3=0:k-a1-a2
            a4=k-a1-a2-a3;
            P(:,ind(a1,a2,a3,a4))=multinomial(k,a1,a2,a3,a4)*...
                    lambda1(:,a1+1).*lambda2(:,a2+1).*lambda3(:,a3+1).*lambda4(:,a4+1);
        end
    end
end

return

function matrix=location(k)
```

```
% 4D array with order indices for the BB basis in local FE form

matrix=zeros(k+1,k+1,k+1,k+1);

matrix(k+1,1,1,1)=1;
matrix(1,k+1,1,1)=2;
matrix(1,1,k+1,1)=3;
matrix(1,1,1,k+1)=4;

for j=1:k-1
    matrix(k-j+1,j+1,1,1)=4+j;
    matrix(1,k-j+1,j+1,1)=4+k-1+j;
    matrix(k-j+1,1,j+1,1)=4+2*(k-1)+j;
    matrix(1,k-j+1,1,j+1)=4+3*(k-1)+j;
    matrix(1,1,k-j+1,j+1)=4+4*(k-1)+j;
    matrix(k-j+1,1,1,j+1)=4+5*(k-1)+j;
end

sofar=4+6*(k-1);
dimfac=(k-1)*(k-2)/2;

for i=1:k-2
    di=(i-1)*(k-2)-(i-2)*(i-1)/2;
    for j=1:k-1-i
        matrix(k-i-j+1,i+1,j+1,1)=sofar+di+j;
        matrix(k-i-j+1,i+1,1,j+1)=sofar+dimfac+di+j;
        matrix(k-i-j+1,1,i+1,j+1)=sofar+2*dimfac+di+j;
        matrix(1,i+1,j+1,k-i-j+1)=sofar+3*dimfac+di+j;
    end
end

index = 4+6*(k-1)+4*dimfac;
for p=1:k-3
    for i=1:k-2-p
        for j=1:k-1-p-i
            index=index+1;
            matrix(k-i-j-p+1,p+1,i+1,j+1) = index;
        end
    end
end

return


function y=multinomial(k,i1,i2,i3,i4)
% Bad implementation of multinomial coefficients, only valid for small k
y=factorial(k)/(factorial(i1)*factorial(i2)*factorial(i3)*factorial(i4));
return
```

Evaluation of derivatives. The formula for the derivative of the elements of the Bernstein basis is

$$\nabla B^{(k)}_{\boldsymbol{\alpha}} = k \sum_{\ell=1}^{4} B^{(k-1)}_{\boldsymbol{\alpha}-\mathbf{e}_\ell} \nabla \lambda_\ell,$$

where $\mathbf{e}_\ell$ are the vectors of the canonical basis of $\mathbb{R}^4$, and where polynomials with a negative entry in the multiindex have to be understood as zero. In the reference element, we have

$$k \left(B^{(k-1)}_{\boldsymbol{\alpha}-(1,0,0,0)} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} + B^{(k-1)}_{\boldsymbol{\alpha}-(0,1,0,0)} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + B^{(k-1)}_{\boldsymbol{\alpha}-(0,0,1,0)} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + B^{(k-1)}_{\boldsymbol{\alpha}-(0,0,0,1)} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right).$$

```
function [Px,Py,Pz]=bernsteinDer3D(x,y,z,k)

% [Px,Py,Pz]=bernsteinDer3D(x,y,z,k)
%
% Input:
%    [x,y,z]  : three M x 1 vectors with coordinates in the reference element
%    k        : polynomial degree
% Output:
%    Px,Py,Pz : M x dk matrices with gradient of Pk basis evaluated at points
% Last modified: April 9, 2015

if k==1
    dim=size(x);
    Px=[-ones(dim) ones(dim) zeros(dim) zeros(dim)];
    Py=[-ones(dim) zeros(dim) ones(dim) zeros(dim)];
    Pz=[-ones(dim) zeros(dim) zeros(dim) ones(dim)];
    return
end

[P,locm1]=bernstein3D(x,y,z,k-1);
[~,loc]  =bernstein3D(0,0,0,k);
indm1 = @(i1,i2,i3,i4) locm1(i1+1,i2+1,i3+1,i4+1);
ind   = @(i1,i2,i3,i4) loc(i1+1,i2+1,i3+1,i4+1);

PxA=zeros(size(x,1),((k+1)*(k+2)*(k+3))/6);
PxB=PxA;
PyA=PxA;
PyB=PxA;
PzA=PxA;
PzB=PzA;

for a1=1:k
    for a2=0:k-a1
        for a3=0:k-a1-a2;
            a4=k-a1-a2-a3;
            PxA(:,ind(a1,a2,a3,a4))=-P(:,indm1(a1-1,a2,a3,a4));
            PyA(:,ind(a1,a2,a3,a4))=-P(:,indm1(a1-1,a2,a3,a4));
            PzA(:,ind(a1,a2,a3,a4))=-P(:,indm1(a1-1,a2,a3,a4));
        end
    end
end

for a1=0:k
    for a2=1:k-a1
        for a3=0:k-a1-a2;
            a4=k-a1-a2-a3;
            PxB(:,ind(a1,a2,a3,a4))=P(:,indm1(a1,a2-1,a3,a4));
        end
    end
end

for a1=0:k
    for a2=0:k-a1
        for a3=1:k-a1-a2;
            a4 = k-a1-a2-a3;
                PyB(:,ind(a1,a2,a3,a4))=P(:,indm1(a1,a2,a3-1,a4));
        end
    end
end

for a1=0:k
    for a2=0:k-a1
        for a3=0:k-a1-a2;
            a4 = k-a1-a2-a3;
            if a4>0
                PzB(:,ind(a1,a2,a3,a4))=P(:,indm1(a1,a2,a3,a4-1));
```

```
            end
        end
    end
end

Px=k*(PxA+PxB);
Py=k*(PyA+PyB);
Pz=k*(PzA+PzB);
return
```

From this moment on, the reference element basis will be then numbered on a single index as P_j, with $j = 1,\ldots,d_k$. This basis is pushed forward to the physical elements $K \in \mathcal{T}_h$ with the transformations $\mathrm{F}_K : \widehat{K} \to K$

$$P_i^K \circ \mathrm{F}_K = P_i \qquad i = 1,\ldots,d_k.$$

3.2 Bivariate polynomials

On the two dimensional reference triangle

$$\widehat{E} := \{(\widehat{x},\widehat{y}) \,:\, \widehat{x},\widehat{y}, 1-\widehat{x}-\widehat{y} \geq 0\}$$

we consider the associated Bernstein-Bézier basis

$$\begin{aligned} P_{\boldsymbol{\alpha}} &= \binom{k}{\boldsymbol{\alpha}} \boldsymbol{\lambda}^{\boldsymbol{\alpha}} \\ &= \frac{k!}{\alpha_1!\alpha_2!\alpha_3!}(1-\widehat{x}-\widehat{y})^{\alpha_1}\widehat{x}^{\alpha_2}\widehat{y}^{\alpha_3} \qquad \boldsymbol{\alpha} \in \mathbb{N}^3, \quad |\boldsymbol{\alpha}| = k. \end{aligned}$$

The basis is numbered from 1 to $\binom{k+2}{2}$ in the following particular way:

$$\begin{array}{lll} (k,0,0) & : & 1 \\ (0,k,0) & : & 2 \\ (0,0,k) & : & 3 \\ (k-j,j,0), \quad j=1,\ldots,k-1 & : & 3+\{1,\ldots,k-1\} \\ (0,k-j,j), \quad j=1,\ldots,k-1 & : & 3+(k-1)+\{1,\ldots,k-1\} \\ (j,0,k-j), \quad j=1,\ldots,k-1 & : & 3+2(k-1)+\{1,\ldots,k-1\} \\ (k-i-j,i,j), \quad i=1,\ldots,k-2, \quad j=1,\ldots,k-1-i & : & 3+3(k-1)+\{1,\ldots,\binom{k-1}{2}\} \end{array}$$

```
function [P,indices] = bernstein2D(x,y,k)

% [P,ind] = bernstein2D(x,y,k)
%
% Input:
%    [x,y]  : two M x 1 vectors with coordinates in the reference element
%    k      : polynomial degree
% Output:
%    P      : M x dk matrix with Pk basis evaluated at points
%    ind    : 3d array to find where three-index BB basis is located at
% Last modified: April 3, 2015

lambda1=bsxfun(@power,1-x-y,0:k);
lambda2=bsxfun(@power,x,0:k);
```

```
lambda3=bsxfun(@power,y,0:k);

indices = location(k);
ind = @(i1,i2,i3) indices(i1+1,i2+1,i3+1);

P=zeros(size(x,1),((k+1)*(k+2))/2);
for a1=0:k
    for a2=0:k-a1
        a3=k-a1-a2;
        P(:,ind(a1,a2,a3))=multinomial(k,a1,a2,a3)*...
                    lambda1(:,a1+1).*lambda2(:,a2+1).*lambda3(:,a3+1);
    end
end

return

function matrix=location(k)
% 3D array with order indices for the BB basis in local FE form

matrix=zeros(k+1,k+1,k+1);
if k==0
    matrix=[1];
    return
end
matrix(k+1,1,1)=1;
matrix(1,k+1,1)=2;
matrix(1,1,k+1)=3;
for j=1:k-1
    matrix(k-j+1,j+1,1)=3+j;
    matrix(1,k-j+1,j+1)=3+k-1+j;
    matrix(j+1,1,k-j+1)=3+2*(k-1)+j;
end
index=3+3*(k-1);
for i=1:k-2
    for j=1:k-1-i
        index=index+1;
        matrix(k-i-j+1,i+1,j+1)=index;
    end
end
return

function y=multinomial(k,i1,i2,i3)
% Bad implementation of multinomial coefficients, onlu valid for small k
y=factorial(k)/(factorial(i1)*factorial(i2)*factorial(i3));
return
```

The bivariate basis will be denoted $\{Q_i : i = 1, \ldots, \delta_k\}$. This basis will be pushed forward to the faces F using the parametrizations $\boldsymbol{\phi}_F : \widehat{F} \to F$:

$$Q_i^F \circ \boldsymbol{\phi}_F = Q_i \qquad i = 1, \ldots, \delta_k, \qquad F \in \mathcal{F}_h.$$

3.3 The finite element space

The global basis of the Finite Element space will be denoted $\{\varphi_i\}$.

The full space. We consider the space

$$V_h = \{u_h \in \mathcal{C}(\Omega) : u_h|_K \in \mathbb{P}_k(K) \quad \forall K \in \mathcal{T}_h\}.$$

The full dimension of this space is

$$\dim V_h = N_{\rm nd} + (k-1)N_{\rm edg} + \frac{(k-1)(k-2)}{2}N_{\rm fc} + \frac{(k-1)(k-2)(k-3)}{6}N_{\rm elt}.$$

There is a basis of V_h which is counted as the global DOF (more on this to follow). Note that the first $N_{\rm nd}$ coefficients of a decomposition

$$V_h \ni u_h = \sum_{j=1}^{\dim V_h} u_j \varphi_j$$

are

$$u_j = u(\mathbf{v}_j) \qquad j = 1, \dots, N_{\rm nd},$$

where $\{\mathbf{v}_j \,:\, j = 1, \dots, N_{\rm nd}\}$ are the vertices of the tetrahedrization given in the order of `T.coordinates`. The easy explanation of the local-to-global ordering of the basis is: for each element, we are given a function (a vector)

$$\mathrm{dof}(\cdot, K) : \{1, \dots, d_k\} \to \{1, \dots, \dim V_h\},$$

which is stored in the K-th column of the output of `DOF3D.m`. Then, if $\{P_i^K \,:\, i = 1, \dots, K\}$ is the local basis on K

$$P_\ell^K = \varphi_{\mathrm{dof}(\ell,K)}|_K.$$

Dirichlet nodes. A Dirichlet DOF is:

- any vertex on the Dirichlet boundary, i.e., any vertex of a Dirichlet face (transition vertices, shared by Dirichlet and Neumann faces are Dirichlet vertices),
- any DOF corresponding to an edge of a Dirichlet face (again, edges shared by a Dirichlet and a Neumann face give their DOF to the Dirichlet list), and
- any interior DOF for a Dirichlet face.

If we consider the list Dir with all Dirichlet DOF and the list Free $= \{1, \dots, \dim V_h\} \setminus$ Dir, then $\{\varphi_i \,:\, i \in \text{Free}\}$ is a basis for

$$\{u_h \in V_h \,:\, u_h|_{\Gamma_D} = 0\}.$$

3.4 Assembly by elements

sec:3.4

Vector assembly. Assume that we have computed quantities (a $d_k \times N_{\rm elt}$ matrix)

$$c_i^K \qquad i = 1, \dots, d_k \qquad K \in \mathcal{T}_h.$$

We want to accummulate (assembly) these local vectors in a single V_h-vector. In a first stage this can be understood as the creation of some expanded versions of the vectors $\mathbf{c}^K$ as

$$\mathbf{d}^K \in \mathbb{R}^{\dim V_h} \qquad \text{given by} \quad d_\ell^K = \begin{cases} c_i^K, & \ell = \mathrm{dof}(i,K) \quad i = 1, \dots, d_k \\ 0, & \text{otherwise.} \end{cases}$$

Finally, the expanded vectors are added in a single vector

$$\mathbf{d} = \sum_K \mathbf{d}^K.$$

This can be done in an easy way using `accumarray` and the output of `DOF3D.m`.

```
loc2glob=DOF3D(T,k);    % k is the polynomial degree
d=accumarray(loc2glob(:),c(:));
```

Matrix assembly. Assume now that we have matrices (a $d_k \times d_k \times N_{\text{elt}}$ three dimensional array)

$$a_{ij}^K \qquad i,j = 1,\dots,d_k, \qquad K \in \mathcal{T}_h.$$

We expand these matrices to $d_k \times d_k$ matrices immersed in a $\dim V_h \times \dim V_h$

$$b_{\ell m} = \begin{cases} a_{ij}^K, & \text{if } \ell = \text{dof}(i,K) \text{ and } m = \text{dof}(j,K), \\ 0, & \text{otherwise,} \end{cases}$$

and then add the results

$$\mathbf{B} = \sum_K \mathbf{B}^K.$$

The result is a sparse matrix that can be easily constructed using `sparse` and the second output of `DOF3D.m`.

```
[¬,Cols]=DOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
B=sparse(Rows(:),Cols(:),A(:));
```

Some explanations might be necessary. The second output of `DOF3D.m` is just a three dimensional $d_k \times d_k \times N_{\text{elt}}$ array containing $d_k \times d_k$ slices (cards). The card corresponding to $K \in \mathcal{T}_h$ contains a $d_k \times d_k$ where all the columns are equal to the $\text{dof}(\cdot, K)$. Note that each column of this matrix contains the same number repeated d_k times. Let us the call $\mathbf{Cols}^K \in \mathbb{R}^{K\times K}$ to this matrix. Then

$$\mathbf{Rows}^K = (\mathbf{Cols}^K)^\top \qquad \forall K \in \mathcal{T}_h.$$

Finally `sparse` takes care of reorganising the entries of the local matrices and accummulating the expanded matrices for all the elements.

3.5 Assembly on Neumann or Dirichlet faces

Vector assembly. Assume now we have compute quantities (a $\delta_k \times N_{\text{bd}}$ matrix, where $\text{bd} \in \{\text{Dir}, \text{Neu}\}$

$$c_i^F \qquad i = 1,\dots,\delta_k, \qquad F \in \mathcal{F}_h^{\text{bd}}.$$

We want to accummulate (assembly) these local vectors in a single V_h-vector. In a first stage this can be understood as the creation of some expanded versions of the vectors $\mathbf{c}^F$ as

$$\mathbf{d}^F \in \mathbb{R}^{\dim V_h} \qquad \text{given by} \quad d_\ell^F = \begin{cases} c_i^F, & \ell = \text{dof}(i,F) \quad i = 1,\dots,\delta_k \\ 0, & \text{otherwise.} \end{cases}$$

Finally, the expanded vectors are added in a single vector

$$\mathbf{d} = \sum_F \mathbf{d}^F.$$

This can be done in an easy way using `accumarray` and the part of the output of `bdDOF3D.m`. For the case of a vector assembly on Neumann faces we do

```
[¬,NeuDOF]=bdDOF3D(T,k);     % k is the polynomial degree
d=accumarray(NeuDOF(:),c(:),[dimVh,1]);
```

while for Dirichlet faces we do

```
DirDOF=bdDOF3D(T,k);     % k is the polynomial degree
d=accumarray(DirDOF(:),c(:),[dimVh,1]);
```

Matrix assembly. Assume now that we have matrices (a $\delta_k \times \delta_k \times N_{\rm elt}$ three dimensional array)

$$a_{ij}^F \qquad i,j = 1,\ldots,\delta_k, \qquad F \in \mathcal{F}_h^{\rm bd}, \qquad {\rm bd} \in \{{\rm Dir}, {\rm Neu}\}.$$

We expand these matrices to $\delta_k \times \delta_k$ matrices immersed in a $\dim V_h \times \dim V_h$

$$b_{\ell m} = \begin{cases} a_{ij}^F, & \text{if } \ell = {\rm dof}(i,F) \text{ and } m = {\rm dof}(j,F), \\ 0, & \text{otherwise,} \end{cases}$$

and then add the results

$$\mathbf{B} = \sum_F \mathbf{B}^F.$$

The result is a sparse matrix that can be easily constructed using `sparse` and part of the output of `bdDOF3D.m`. This is strightforward in the case of Neumann faces. (The explanations are similar to those of the matrix assembly by tetrahedra.)

```
[¬,¬,Cols]=bdDOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
B=sparse(Rows(:),Cols(:),A(:),dimVh,dimVh);
```

The case of assembly on Dirichlet faces needs some additional work, since the list for Dirichlet assembly is not produced by `bdDOF3D.m`. This can be done in a couple of lines though:

```
DirDOF=bdDOF3D(T,k);
delk=3+3*(k-1)+0.5*(k-1)*(k-2);   % dim P_k(F)
Cols=repmat(DirDOF(:)',[delk,1]);
Cols=reshape(Cols,[delk,delk,Ndir]);  % Ndir=number of Dir faces
```

4 Quadrature

At this stage, we will be using Stroud quadrature on the reference tetrahedron $\widehat{K}$ and on the reference triangle $\widehat{F}$. The function `gaussJacobi.m` was adapted from the open source function `cdgqf.m` by John Burkhardt in order to generate Gauss-Jacobi quadrature rules in the interval $[-1,1]$ (see below). To generate the GJ formula of N nodes in $[-1,1]$ to approximate the integral

$$\int_{-1}^{1} f(t)(1-t)^m {\rm d}t$$

we use the command

```
gaussJacobi(N,m)
```

```
function [ t, wts ] = gaussJacobi(nt,alpha)

%function [ t, wts ] = cdgqf( nt,alpha)
%
%  Adapted from code written by John Burkardt, which in turn was adapted
%  from code written by Sylvan Elhay and Jaroslav Kautsky.
%
% Inputs:
%       nt    : number of desired points
%       alpha : power of (1-t) as in \int_{-1}^1 f(t) (1-t)^alpha dt
%
```

```
% Outputs:
%       t     : quadrature nodes
%       wts   : quadrature weights  (both for Gauss-Jacobi)
%
% Last Modified: March 24, 2016

bj = zeros(nt,1);
aj = zeros(nt,1);

aj(1) = -alpha / (alpha +2);
bj(1) = sqrt(4*(1 + alpha)/((alpha + 3)*(alpha + 2)^2));

aj(2:nt) = -alpha^2./((2*(2:nt) + alpha - 2).*(2*(2:nt) + alpha));
bj(2:nt) = sqrt((2*(2:nt).*((2:nt) + alpha)).^2 ...
           ./(((2*(2:nt) + alpha).^2 -1).*(2*(2:nt) + alpha).^2));

%   Diagonalize the Jacobi matrix.

A = diag(aj) + diag(bj(1:nt-1),-1) + diag(bj(1:nt-1),1);
[V,D] = eig(A);

%  Compute the knots and weights.

t = diag(D);

wts = zeros(nt, 1);
wts(1) = sqrt(2^(alpha + 1)*gamma(alpha + 1)*1/gamma(alpha +2));
wts = V'*wts;
wts = wts.^2;

  return
end
```

4.1 Quadrature on tetrahedra

Quadrature on the reference tetrahedron. We want to approximate

$$6\int_{\widehat{K}}\phi = \frac{1}{|\widehat{K}|}\int_{\widehat{K}}\phi \approx \sum_q \widehat{\omega}_q \phi(\widehat{\mathbf{p}}_q)$$

with exact quadrature for polynomials of degree m. Note that this normalization implies that

$$\sum_q \widehat{\omega}_q = 1.$$

Stroud quadrature. We use the change of variables:

$$\widehat{K} \ni (\widehat{x},\widehat{y},\widehat{z}) \longmapsto (x,y,z) := \left(x, \frac{y}{1-x}, \frac{z}{1-x-y}\right) \in [0,1]^3.$$

The inverse transformation is

$$\widehat{x} = x, \quad \widehat{y} = (1-x)y, \quad \widehat{z} = (1-x)(1-y)z,$$

which implies that the Jacobian is

$$(1-x)^2(1-y).$$

We thus transform

$$\int_{\widehat{K}}\phi = \int_{[0,1]^3}\phi(x,(1-x)y,(1-y)(1-y)z)(1-x)^2(1-y)\mathrm{d}x\mathrm{d}y\mathrm{d}z.$$

Finally, we apply a Gauss-Jacobi formula with $N = \lceil \frac{m+1}{2} \rceil$ points on each of the variables. Since

$$\phi \in \mathbb{P}_k \qquad \Longleftrightarrow \qquad \phi(x, (1-x)y, (1-y)(1-y)z) \in \mathbb{P}_k(x, y, z),$$

the resulting formula will have degree m.

Gauss-Jacobi formulas. We specifically need the following kind of GJ formulas

$$\int_0^1 (1-t)^m f(t) \mathrm{d}t \approx \sum_{q=1}^{N} \omega_q^m f(t_q^m).$$

GJ formulas are typically obtained in the form

$$\int_{-1}^1 (1-x)^\alpha (1-x)^\beta f(x) \mathrm{d}x \approx \sum_q \varpi_q^{\alpha,\beta} f(x_q^{\alpha,\beta}),$$

so we will need to transform the weights and nodes according to

$$t_q^m = \frac{1 + x_q^{m,0}}{2}, \qquad \omega_q^m = \frac{\varpi_q^{m,0}}{2^{m+1}}.$$

Quadrature on a physical element. On K we do as follows:

$$\begin{aligned} \int_K \phi &= \frac{|K|}{|\widehat{K}|} \int_{\widehat{K}} \phi \circ \mathrm{F}_K \\ &\approx |K| \sum_q \widehat{\omega}_q \phi(\mathbf{p}_q^K), \end{aligned}$$

where

$$\mathbf{p}_q^K = \mathrm{F}(\widehat{\mathbf{p}}_q) = (1 - \widehat{x}_q - \widehat{y}_q - \widehat{z}_q)\mathbf{v}_1^K + \widehat{x}_q \mathbf{v}_2^q + \widehat{y}_q \mathbf{v}_3^q + \widehat{z}_q \mathbf{v}_4^q.$$

(Explain here how to compute all quadrature points at the same time.)

4.2 Quadrature on faces

Quadrature on the parametric domain. Integrals on a face $F \in \mathcal{F}_h$ will be done by using the parametrization $\boldsymbol{\phi}_F : \widehat{F} \to F$. We first build a quadrature formula

$$2 \int_{\widehat{F}} \phi = \frac{1}{|\widehat{F}|} \int_{\widehat{F}} \phi \approx \sum_q \omega_q \phi(\widehat{\mathbf{q}}_q)$$

with weights normalized so that

$$\sum_q \omega_q = 1.$$

The formula is assumed to have degree m. We proceed like in the three dimensional case, using the transformation

$$\widehat{F} \ni (\widehat{x}, \widehat{y}) \longmapsto (x, y) := \left(x, \frac{y}{1-x} \right) \in [0,1]^2,$$

with

$$\widehat{x} = x, \quad \widehat{y} = (1-x)y, \qquad \mathrm{d}\widehat{x}\mathrm{d}\widehat{y} = (1-x)\mathrm{d}x\mathrm{d}y.$$

We then write

$$\int_{\widehat{F}} \phi = \int_{[0,1]^2} \phi(x, (1-x)y)(1-x)\mathrm{d}x\, \mathrm{d}y,$$

apply a GJ formula with $N = \lceil \frac{m+1}{2} \rceil$ on each variable and notcies that the change of variables preserves polynomial degrees.

Quadrature on a face. We just parametrize and apply the formula on the reference element

$$\int_F \phi = \int_{\widehat{F}} \phi \circ \boldsymbol{\phi}_F |\partial_{\widehat{x}} \boldsymbol{\phi}_F \times \partial_{\widehat{y}} \boldsymbol{\phi}_F| = \frac{|F|}{|\widehat{F}|} \int_{\widehat{F}} \phi \circ \boldsymbol{\phi}_F \approx |F| \sum_q \omega_q \phi(\boldsymbol{\phi}_F(\widehat{\mathbf{q}}_q)).$$

4.3 Quadrature code

The code for the computation of the two and three dimensional Stroud formulas is joint for two and three dimensions. The code is based on an open source implementation of several Gaussian formulas that includes the Gauss-Jacobi formulas. The rest of the work is the tensorization of the one dimensional formulas.

```
function formula=quadratureFEM(deg,dim)

% form=quadratureFEM(deg,d)
% Input:
%     deg    : degree of precision of the quad formula
%     d      : dimension
% Output:
%     form   : Stroud Quadrature formula on the d-dimensional simplex
%              (with d=2 or 3) - Nquad x (d+2) matrix
%              with barycentric coordinates and weights
%              (weights are not normalized yet)
% Last modified: June 10, 2016

N=ceil((deg+1)/2);
[t3,w3]=gaussJacobi(N,0);
[t2,w2]=gaussJacobi(N,1);
[t1,w1]=gaussJacobi(N,2);
t3=(t3+1)/2;
t2=(t2+1)/2;
t1=(t1+1)/2;
w3=w3/2;
w2=w2/4;
w1=w1/8;

if dim==3
    table=zeros(N,N,N);
    table(:)=1:N^3;
    x=zeros(N^3,1);
    y=x;z=x;w=x;
    for q1=1:N
        for q2=1:N
            for q3=1:N
                x(table(q1,q2,q3))=t1(q1);
                y(table(q1,q2,q3))=(1-t1(q1))*t2(q2);
                z(table(q1,q2,q3))=(1-t1(q1))*(1-t2(q2))*t3(q3);
                w(table(q1,q2,q3))=w1(q1)*w2(q2)*w3(q3);
            end
        end
    end
    formula=[1-x-y-z x y z 6*w];
elseif dim==2
    table=zeros(N,N);
    table(:)=1:N^2;
    x=zeros(N^2,1);
    y=x;w=x;
    t1=t2;w1=w2;
    t2=t3;w2=w3;
    for q1=1:N
        for q2=1:N
                x(table(q1,q2))=t1(q1);
                y(table(q1,q2))=(1-t1(q1))*t2(q2);
```

```
                w(table(q1,q2))=w1(q1)*w2(q2);
        end
    end
    formula=[1-x-y x y 2*w];
end
return
```

5 Mass, stiffness, convection, and load

5.1 The load vector

The load vector for a function f is

$$\int_\Omega f\,\varphi_i, \qquad i=1,\dots,\dim V_h.$$

It is computed from local load vector

$$\int_K f P_i^K, \qquad i=1,\dots,d_k, \qquad K\in\mathcal{T}_h\equiv\{1,\dots,N_{\rm elt}\}$$

and the vector assembly process explained in Section 3.4.

Local load vectors. Given a function f, we first want to compute the integrals To compute the local load vector (matrix) we proceed as follows:

$$\int_K f P_i^K = 6|K|\int_{\widehat{K}}(f\circ \mathrm{F}_K)P_i \approx \sum_q |K|\widehat{\omega}_q P_i(\widehat{\mathbf{p}}_q) f(\mathbf{p}_q^K), \qquad \mathbf{p}_q^K=\mathrm{F}_K(\widehat{\mathbf{p}}_q).$$

We start by computing all quadrature points at once. Let Λ be the $N_{\rm quad}\times 4$ matrix with the barycentric coordinates of all quadrature nodes. Let $\mathrm{X}^{\mathcal{T}}$, $\mathrm{Y}^{\mathcal{T}}$, and $\mathrm{Z}^{\mathcal{T}}$ be the $4\times N_{\rm elt}$ matrices with the x, y, and z coordinates of all 4 vertices of all triangles. Then

$$\mathrm{X}:=\Lambda\mathrm{X}^{\mathcal{T}}, \qquad \mathrm{Y}:=\Lambda\mathrm{Y}^{\mathcal{T}}, \qquad \mathrm{Z}:=\Lambda\mathrm{Z}^{\mathcal{T}}$$

are $N_{\rm quad}\times N_{\rm elt}$ with the coordinates of all quadrature points on all triangles. Let us finally consider the $N_{\rm quad}\times d_k$ matrix

$$(\mathrm{P}_\omega)_{q,i}:=\widehat{\omega}_q P_i(\widehat{\mathbf{p}}_q),$$

and the column vector **area** with the areas of all the elements (sotred in `T.area`). Then we just need to compute

$$\mathbf{area}^\top\odot(\mathrm{P}_\omega^\top f(\mathrm{X},\mathrm{Y})),$$

where $\odot$ is used to represent the `bsxfun(@times,....)` command, which, in this case, multiplies element by element the row vector $\mathbf{area}^\top$ by each of the rows ot the $d_k\times N_{\rm elt}$ matrix $\mathrm{P}_\omega^\top f(\mathrm{X},\mathrm{Y})$.
It should be noted that `loadVector3D.m` can take an array of vectorized functions `f={f1,f2,...,fN}` to produce an array `fh={fh1,fh2,...,fhN}` of load vectors as well as the single function case. This is particularly useful for vector field computations such as elasticity.

```
function fh=loadVector3D(f,T,k)

% fh=loadVector3D(f,T,k)
%
% Input:
%    f  : vectorized function of three variables
%                        or
%         array of vectorized functions of three vars {f1,f2,f3,...}
```

```
%    T  : enhanced triangulation
%    k  : polynomial degree
% Output:
%    fh : dim P_k(T_h) load vector
%                 or matrix of load vectors
% Last modified: July 16, 2015

formula=quadratureFEM(3*k,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);
Pw=bsxfun(@times,formula(:,5),bernstein3D(formula(:,2),formula(:,3),formula(:,4),k));
loc2glob=DOF3D(T,k);
dimVh=max(loc2glob(:));
if iscell(f)
    fh=zeros(dimVh,length(f));
    for i=1:length(f)
        fhi=bsxfun(@times,T.volume,Pw'*f{i}(x,y,z));
        fh(:,i)=accumarray(loc2glob(:),fhi(:));
    end
else
    fh=bsxfun(@times,T.volume,Pw'*f(x,y,z));% dk x Nelt matrix with local tests
    fh=accumarray(loc2glob(:),fh(:));
end
return
```

5.2 The mass matrix

The mass matrix for a coefficient/weight $c : \Omega \to \mathbb{R}$ is the matrix with entries

$$\int_\Omega c\,\varphi_i\,\varphi_j \qquad i,j = 1,\dots,\dim V_h.$$

It can be compute with the local mass matrices

$$\int_K c P_i^K P_j^K, \qquad i,j = 1,\dots,d_k, \qquad K \in \mathcal{T}_h$$

and the matrix assembly process explained in Section 3.4.

Local mass matrices. The collection of all local mass matrices will be produced as a single $d_k \times (d_k N_{\text{elt}})$ matrix with N_{elt} row blocks of $d_k \times d_k$ size containing the information of each element. (We can think that this matrix is momentarily be reshaped to a $d_k \times d_k \times N_{\text{elt}}$ before assembly, in order to fit in the description of the assembly process of Section 3.4. This reshaping step is not needed though.)
The integrals are approximated using the formula

$$\int_K c\,P_i^K P_j^K \approx \sum_q |K| f(\mathbf{p}_q^K)\big(\widehat{\omega}_q P_i(\widehat{\mathbf{p}}_q) P_j(\widehat{\mathbf{p}}_q)\big)$$

Consider then the $N_{\text{quad}} \times d_k$ matrix

$$\mathrm{P}_{q,i} = P_i(\widehat{\mathbf{p}}_q),$$

the $d_k \times d_k$ matrices

$$\mathrm{PP}^q := \widehat{\omega}_q \mathrm{row}(\mathrm{P},q)^\top \mathrm{row}(\mathrm{P},q),$$

and the $N_{\text{quad}} \times N_{\text{elt}}$ matrix $\mathrm{C} := c(\mathrm{X},\mathrm{Y})$ with the values of the coefficient c on all quadrature points. The $d_k \times (d_k N_{\text{elt}})$ matrix we are looking for is

$$\sum_q \big(\mathbf{area}^\top * \mathrm{row}(\mathrm{C},q)\big) \otimes \mathrm{PP}^q,$$

where $\otimes$ is the symbol for the Kronecker product and $*$ is used for the element by element product of arrays (the `.*` operator in Matlab).

```
function Mh=massMatrix3D(c,T,k)

% Mh=massMatrix3D(c,T,k)
%
% Input:
%    c  : vectorized function of three variables
%    T  : enhanced triangulation
%    k  : polynomial degree
% Output:
%    Mh : dim P_k(T_h) x dim P_k(T_h) mass matrix
% Last modified: April 17, 2015

formula=quadratureFEM(4*k+3,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);
c=bsxfun(@times,T.volume,c(x,y,z));
P=bernstein3D(formula(:,2),formula(:,3),formula(:,4),k);

dk=size(P,2);
Nelt=size(T.elements,2);
Mh=zeros(dk,dk*Nelt);

for q=1:size(formula,1)
    Mh=Mh+kron(c(q,:),formula(q,5)*P(q,:)'*P(q,:));
end
[¬,Cols]=DOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
Mh=sparse(Rows(:),Cols(:),Mh(:));

return
```

stiffy

5.3 The stiffness matrix

The goal of this section is the computation of

$$\int_\Omega (\kappa \nabla \varphi_j) \cdot \nabla \varphi_i \qquad i,j = 1,\dots,\dim V_h,$$

where

$$\kappa = \begin{bmatrix} \kappa_{11} & \kappa_{12} & \kappa_{13} \\ \kappa_{12} & \kappa_{22} & \kappa_{23} \\ \kappa_{13} & \kappa_{23} & \kappa_{33} \end{bmatrix} : \Omega \to \mathbb{R}^{3\times 3}_{\text{sym}}.$$

With the view of dealing more general problems, we will compute six matrices

$$\int_\Omega \kappa_{ij} \partial_{x_j} \varphi_b \partial_{x_i} \varphi_a \qquad a,b = 1,\dots,\dim V_h, \qquad 1 \le i \le j \le 3.$$

These matrices will be computed from the local stiffness matrices

$$\int_K \kappa_{ij} \partial_i P^K_\alpha \partial_j P^K_\beta, \qquad \alpha,\beta = 1,\dots,d_k, \quad K \in \mathcal{T}_h,$$

using a matrix assembly process.

Local stiffness matrices. With a change to the reference element, we obtain

$$\begin{aligned}\int_K \kappa_{ij}\partial_i P_\alpha^K \partial_j P_\beta^K &= 6|K|\int_{\widehat{K}} (\kappa_{ij}\circ \mathrm{F}_K)\big((\partial_i P_\alpha^K)\circ \mathrm{F}_K\big)\big((\partial_j P_\beta^K)\circ \mathrm{F}_K\big)\\ &= 6\int_{\widehat{K}} (\kappa_{ij}\circ \mathrm{F}_K)(c_{1i}^K\partial_1 P_\alpha + c_{2i}^K\partial_2 P_\alpha + c_{3i}^K\partial_3 P_\alpha)(c_{1j}^K\partial_1 P_\alpha + c_{2j}^K\partial_2 P_\beta + c_{3j}^K\partial_3 P_\beta)\\ &= \sum_{l,m\in\{1,2,3\}} c_{li}^K c_{mj}^K \Big(6\int_{\widehat{K}} (\kappa_{ij}\circ \mathrm{F}_K)\partial_l P_\alpha \partial_j P_\beta\Big)\\ &\approx \sum_q \sum_{l,m\in\{1,2,3\}} c_{li}^K c_{mj}^K \kappa_{ij}(\mathbf{p}_q^K)\big(\underbrace{\widehat{\omega}_q \partial_l P_\alpha(\widehat{\mathbf{p}}_q)\partial_m P_\beta(\widehat{\mathbf{p}}_q)}_{\mathrm{P}_{lm}^q}\big),\end{aligned}$$

where

$$\begin{aligned}\mathrm{C}^K &= |K|^{1/2}\mathrm{B}_K^{-1} = \frac{1}{\sqrt{6}}(\det \mathrm{B}_K)^{1/2}\mathrm{B}_K^{-1} = \begin{bmatrix} c_{11}^K & c_{12}^K & c_{13}^K \\ c_{21}^K & c_{22}^K & c_{23}^K \\ c_{31}^K & c_{32}^K & c_{33}^K \end{bmatrix}\\ &= \frac{1}{6\sqrt{|K|}}\begin{bmatrix} y_{13}z_{14}-y_{14}z_{13} & z_{13}x_{14}-z_{14}x_{13} & x_{13}y_{14}-x_{14}y_{13} \\ y_{14}z_{12}-y_{12}z_{14} & z_{14}x_{12}-z_{12}x_{14} & x_{14}y_{12}-x_{12}y_{14} \\ y_{12}z_{13}-y_{13}z_{12} & z_{12}x_{13}-z_{13}x_{12} & x_{12}y_{13}-x_{13}y_{12} \end{bmatrix}\end{aligned}$$

and

$$x_{1j} = x_j^K - x_1^K, \qquad y_{1j} = y_j^K - y_1^K, \qquad z_{1j} = z_j^K - z_1^K$$

are the entries of B_K.

The 'scalar' diffusion case. If κ is a symmetric matrix valued function, we need to plug $\kappa_{11}, \kappa_{12}, \kappa_{13}, \kappa_{21}, \kappa_{23}, \kappa_{33}$ into `stiffnessMatrices3D` function. We will also work out the case where all six functions are equal. This is done in a slightly inefficient way by making six copies of the scalar function κ. The case where all six functions are equal outputs six matrices

$$\int_\Omega \kappa \partial_{x_j}\varphi_b \partial_{x_i}\varphi_a \qquad a,b = 1,\dots,\dim V_h, \qquad 1\le i\le j\le 3.$$

Three of them (for the values $i=j=1,2,3$) would be added when dealing with isotropic diffusion, i.e., when we want to assemble the matrix

$$\int_\Omega \kappa \nabla\varphi_i\cdot\varphi_j \qquad i,j = 1,\dots,\dim V_h.$$

The other three are not used for this diffusion matrix, but they are needed for heterogeneous isotropic elasticity.
The code is organized so that it accepts a variable number of arguments:

- if there are eight incoming arguments, they correspond (in this order) to $\kappa_{11}, \kappa_{12}, \kappa_{13}, \kappa_{21}, \kappa_{23}, \kappa_{33}$ and then $\mathcal{T}_h$ (the tetrahedrization) and k (the polynomial degree);
- if there are three incoming arguments, they correspond to κ, $\mathcal{T}_h$ and k.

```
function S=stiffnessMatrices3D(varargin)

% Sh=stiffnessMatrices3DNew(kxx,kxy,kxz,kyy,kyz,kzz,T,k);
% Sh=stiffnessMatrices3DNew(kxx,T,k);
%
% Input:
%    kxx,kxy,kxz,kyy,kyz,kzz : vectorized functions of three variables
```

```
%    T  : enhanced triangulation
%    k  : polynomial degree
% Output:
%    Sh : 3 x 3 Cell array dim FE x dim FE stiffness matrices
%         (only upper blocks of cell array are non-empty)
% Last modified: September 30, 2016

% Evaluations of coefficients and basis functions

switch nargin
    case 8
        kxx = varargin{1};
        kxy = varargin{2};
        kxz = varargin{3};
        kyy = varargin{4};
        kyz = varargin{5};
        kzz = varargin{6};
        T = varargin{7};
        k = varargin{8};
    case 3
        kxx = varargin{1};
        kxy = kxx; kxz = kxx; kyy = kxx; kyz = kxx; kzz = kxx;
        T = varargin{2};
        k = varargin{3};
end

formula=quadratureFEM(3*k+1,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);

K{1,1}=kxx(x,y,z);
K{1,2}=kxy(x,y,z);
K{1,3}=kxz(x,y,z);
K{2,2}=kyy(x,y,z);
K{2,3}=kyz(x,y,z);
K{3,3}=kzz(x,y,z);
[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k);

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK

sqdet=1./(6*sqrt(T.volume));
c{1,1}=sqdet.*(y13.*z14 - y14.*z13);
c{1,2}=sqdet.*(x14.*z13 - x13.*z14);
c{1,3}=sqdet.*(x13.*y14 - x14.*y13);
c{2,1}=sqdet.*(y14.*z12 - y12.*z14);
c{2,2}=sqdet.*(x12.*z14 - x14.*z12);
c{2,3}=sqdet.*(x14.*y12 - x12.*y14);
c{3,1}=sqdet.*(y12.*z13 - z12.*y13);
c{3,2}=sqdet.*(x13.*z12 - x12.*z13);
c{3,3}=sqdet.*(x12.*y13 - x13.*y12);

% Loop over quadrature points
```

```
dk=size(Px,2);
Nelt=size(T.elements,2);
for i=1:3
    for j=i:3
        S{i,j}=zeros(dk,dk*Nelt);
    end
end

for q=1:size(formula,1)
    P{1,1}=formula(q,5)*Px(q,:)'*Px(q,:);
    P{1,2}=formula(q,5)*Px(q,:)'*Py(q,:);
    P{1,3}=formula(q,5)*Px(q,:)'*Pz(q,:);
    P{2,1}=P{1,2}';%formula(q,5)*Py(q,:)'*Px(q,:);
    P{2,2}=formula(q,5)*Py(q,:)'*Py(q,:);
    P{2,3}=formula(q,5)*Py(q,:)'*Pz(q,:);
    P{3,1}=P{1,3}';%formula(q,5)*Pz(q,:)'*Px(q,:);
    P{3,2}=P{2,3}';%formula(q,5)*Pz(q,:)'*Py(q,:);
    P{3,3}=formula(q,5)*Pz(q,:)'*Pz(q,:);

    for i=1:3
        for j=i:3
            for l=1:3
                for m=1:3
                    S{i,j}=S{i,j}+kron(c{l,j}.*c{m,i}.*K{i,j}(q,:),P{m,l});
                end
            end
        end
    end
end


% Assembly
[¬,Cols]=DOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
for i=1:3
    for j=i:3
        S{i,j}=sparse(Rows(:),Cols(:),S{i,j}(:));
    end
end
return
```

5.4 Non-symmetric stiffness matrices

For some non-symmetric diffusion problems the nine matrices

$$\int_\Omega \kappa_{ij}\partial_{x_j}\varphi_b\partial_{x_i}\varphi_a \qquad a,b=1,\dots,\dim V_h, \qquad 1\le i,j\le 3$$

are needed. The code is very similar to the one developed for stiffness matrices. The output of the function is a 3×3 cell array with the nine FEM matrices above.

```
function S=stiffnessMatricesNonSymmetric3D(kxx,kxy,kxz,kyx,kyy,kyz,...
                                           kzx,kzy,kzz,T,k)

% S=stiffnessMatricesNonSymmetric3D(kxx,kxy,kxz,kyx,kyy,kyz,...
%                                   kzx,kzy,kzz,T,k)
% Input:
%    kxx,kxy,kxz, : vectorized functions of three variables
%    kyx,kyy,kyz,
%    kzx,kzy,kzz
%    T            : enhanced triangulation
%    k            : polynomial degree
% Output:
```

```
%    S            : 3 x 3 Cell array dim FE x dim FE stiffness matrices
%
% Last modified: June 24, 2016

% Evaluations of coefficients and basis functions

formula=quadratureFEM(3*k+1,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);

K{1,1}=kxx(x,y,z);
K{1,2}=kxy(x,y,z);
K{1,3}=kxz(x,y,z);
K{2,1}=kyx(x,y,z);
K{2,2}=kyy(x,y,z);
K{2,3}=kyz(x,y,z);
K{3,1}=kzx(x,y,z);
K{3,2}=kzy(x,y,z);
K{3,3}=kzz(x,y,z);
[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k);

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK

sqdet=1./(6*sqrt(T.volume));
c{1,1}=sqdet.*(y13.*z14 - y14.*z13);
c{1,2}=sqdet.*(x14.*z13 - x13.*z14);
c{1,3}=sqdet.*(x13.*y14 - x14.*y13);
c{2,1}=sqdet.*(y14.*z12 - y12.*z14);
c{2,2}=sqdet.*(x12.*z14 - x14.*z12);
c{2,3}=sqdet.*(x14.*y12 - x12.*y14);
c{3,1}=sqdet.*(y12.*z13 - z12.*y13);
c{3,2}=sqdet.*(x13.*z12 - x12.*z13);
c{3,3}=sqdet.*(x12.*y13 - x13.*y12);

% Loop over quadrature points

dk=size(Px,2);
Nelt=size(T.elements,2);
for i=1:3
    for j=1:3
        S{i,j}=zeros(dk,dk*Nelt);
    end
end

for q=1:size(formula,1)
    P{1,1}=formula(q,5)*Px(q,:)'*Px(q,:);
    P{1,2}=formula(q,5)*Px(q,:)'*Py(q,:);
    P{1,3}=formula(q,5)*Px(q,:)'*Pz(q,:);
    P{2,1}=P{1,2}';%formula(q,5)*Py(q,:)'*Px(q,:);
    P{2,2}=formula(q,5)*Py(q,:)'*Py(q,:);
    P{2,3}=formula(q,5)*Py(q,:)'*Pz(q,:);
    P{3,1}=P{1,3}';%formula(q,5)*Pz(q,:)'*Px(q,:);
    P{3,2}=P{2,3}';%formula(q,5)*Pz(q,:)'*Py(q,:);
```

```
    P{3,3}=formula(q,5)*Pz(q,:)'*Pz(q,:);

    for i=1:3
        for j=1:3
            for l=1:3
                for m=1:3
                    S{i,j}=S{i,j}+kron(c{l,j}.*c{m,i}.*K{i,j}(q,:),P{m,l});
                end
            end
        end
    end
end


% Assembly
[¬,Cols]=DOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
for i=1:3
    for j=1:3
        S{i,j}=sparse(Rows(:),Cols(:),S{i,j}(:));
    end
end
return
```

5.5 Constant coefficient stiffness matrices

The goal of this function is the computation of six matrices

$$\int_\Omega \partial_{x_j}\varphi_b \partial_{x_i}\varphi_a \qquad a,b = 1,\dots,\dim V_h, \qquad 1 \le i \le j \le 3.$$

These matrices will be computed from the local stiffness matrices

$$\int_K \partial_i P_\alpha^K \partial_j P_\beta^K, \qquad \alpha,\beta = 1,\dots,d_k, \quad K \in \mathcal{T}_h,$$

using the same matrix assembly process that we used before. With the usual change to the reference element, we obtain

$$\begin{aligned}
\int_K \kappa_{ij}\partial_i P_\alpha^K \partial_j P_\beta^K &= 6|K| \int_{\widehat{K}} \big((\partial_i P_\alpha^K)\circ \mathrm{F}_K\big)\big((\partial_j P_\beta^K)\circ \mathrm{F}_K\big) \\
&= 6\int_{\widehat{K}} (c_{1i}^K\partial_1 P_\alpha + c_{2i}^K\partial_2 P_\alpha + c_{3i}^K\partial_3 P_\alpha)(c_{1j}^K\partial_1 P_\alpha + c_{2j}^K\partial_2 P_\beta + c_{3j}^K\partial_3 P_\beta) \\
&= \sum_{l,m\in\{1,2,3\}} c_{li}^K c_{mj}^K \Big(6\int_{\widehat{K}} \partial_l P_\alpha \partial_j P_\beta\Big) \\
&= \sum_{l,m\in\{1,2,3\}} c_{li}^K c_{mj}^K \underbrace{\Big(\sum_q \widehat{\omega}_q \partial_l P_\alpha(\widehat{\mathbf{p}}_q)\partial_m P_\beta(\widehat{\mathbf{p}}_q)\Big)}_{\mathrm{P}_{lm}},
\end{aligned}$$

where C^K is the same as in Section 5.3. This shows that there is no need to loop over quadrature points any longer and the computation at the local level is much simpler.

```
function S=stiffnessMatricesCC3D(T,k)

% Sh=stiffnessMatricesCC3D(T,k);
%
% Input:
%    T  : enhanced triangulation
```

```
%    k  : polynomial degree
% Output:
%    Sh : 3 x 3 Cell array dim FE x dim FE stiffness matrices
%         (only upper blocks of cell array are non-empty)
% Last modified: September 30, 2016

% Matrices on the reference element

formula=quadratureFEM(2*k-1,3);

[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k);
wPx=bsxfun(@times,formula(:,5),Px);
wPy=bsxfun(@times,formula(:,5),Py);
wPz=bsxfun(@times,formula(:,5),Pz);
P{1,1}=Px'*wPx;
P{1,2}=Px'*wPy;
P{1,3}=Px'*wPz;
P{2,1}=P{1,2}';
P{2,2}=Py'*wPy;
P{2,3}=Py'*wPz;
P{3,1}=P{1,3}';
P{3,2}=P{2,3}';
P{3,3}=Pz'*wPz;

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK

sqdet=1./(6*sqrt(T.volume));
c{1,1}=sqdet.*(y13.*z14 - y14.*z13);
c{1,2}=sqdet.*(x14.*z13 - x13.*z14);
c{1,3}=sqdet.*(x13.*y14 - x14.*y13);
c{2,1}=sqdet.*(y14.*z12 - y12.*z14);
c{2,2}=sqdet.*(x12.*z14 - x14.*z12);
c{2,3}=sqdet.*(x14.*y12 - x12.*y14);
c{3,1}=sqdet.*(y12.*z13 - z12.*y13);
c{3,2}=sqdet.*(x13.*z12 - x12.*z13);
c{3,3}=sqdet.*(x12.*y13 - x13.*y12);

% Matrices on the physical elements

dk=size(Px,2);
Nelt=size(T.elements,2);
for i=1:3
    for j=i:3
        S{i,j}=zeros(dk,dk*Nelt);
        for el=1:3
            for m=1:3
                S{i,j}=S{i,j}+kron(c{el,j}.*c{m,i},P{m,el});
            end
        end
    end
end

% Assembly
```

```
[¬,Cols]=DOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
for i=1:3
    for j=i:3
        S{i,j}=sparse(Rows(:),Cols(:),S{i,j}(:));
    end
end
return
```

ction_matrix

5.6 The convection matrix

The goal of this section is the computation of

$$\int_\Omega (\mathbf{b}\cdot\nabla\varphi_j)\varphi_i \qquad i,j=1,\dots,\dim V_h,$$

where

$$\mathbf{b} = \left[\begin{array}{ccc} b_1 & b_2 & b_3 \end{array}\right] : \Omega \to \mathbb{R}^3.$$

With the view of dealing more general problems, we will compute three matrices

$$\int_\Omega b_i\varphi_a\partial_{x_i}\varphi_b \qquad a,b=1,\dots,\dim V_h, \qquad 1\le i\le 3.$$

These matrices will be computed from the local convection matrices

$$\int_K b_i P_\alpha^K \partial_i P_\beta^K, \qquad \alpha,\beta=1,\dots,d_k, \quad K\in\mathcal{T}_h,$$

using a matrix assembly process.

Local stiffness matrices. With a change to the reference element, we obtain

$$\begin{aligned}
\int_K b_i P_\alpha^K \partial_i P_\beta^K &= 6|K| \int_{\widehat{K}} (b_i\circ \mathrm{F}_K)\, P_\alpha\left((\partial_i P_\beta^K)\circ \mathrm{F}_K\right) \\
&= 6\int_{\widehat{K}} (b_i\circ \mathrm{F}_K) P_\alpha (d_{1i}^K\partial_1 P_\alpha + d_{2i}^K\partial_2 P_\beta + d_{3i}^K\partial_3 P_\beta) \\
&= \sum_{j=1}^3 d_{ji}^K \Big(6\int_{\widehat{K}} (b_i\circ \mathrm{F}_K) P_\alpha \partial_j P_\beta\Big) \\
&\approx \sum_q \sum_{j=1}^3 d_{ji}^K b_i(\mathbf{p}_q^K)\big(\underbrace{\widehat{\omega}_q P_\alpha(\widehat{\mathbf{p}}_q)\partial_j P_\beta(\widehat{\mathbf{p}}_q)}_{\mathrm{Q}_j^q}\big),
\end{aligned}$$

where (compare with the section on the stiffness matrix)

$$\begin{aligned}
\mathrm{D}^K &= |K|\,\mathrm{B}_K^{1/2} = |K|^{1/2}\mathrm{C}^K = \left[\begin{array}{ccc} d_{11}^K & d_{12}^K & d_{13}^K \\ d_{21}^K & d_{22}^K & d_{23}^K \\ d_{31}^K & d_{32}^K & d_{33}^K \end{array}\right] \\
&= \frac{1}{6}\left[\begin{array}{ccc} y_{13}z_{14}-y_{14}z_{13} & z_{13}x_{14}-z_{14}x_{13} & x_{13}y_{14}-x_{14}y_{13} \\ y_{14}z_{12}-y_{12}z_{14} & z_{14}x_{12}-z_{12}x_{14} & x_{14}y_{12}-x_{12}y_{14} \\ y_{12}z_{13}-y_{13}z_{12} & z_{12}x_{13}-z_{13}x_{12} & x_{12}y_{13}-x_{13}y_{12} \end{array}\right]
\end{aligned}$$

and

$$x_{1j}=x_j^K-x_1^K, \qquad y_{1j}=y_j^K-y_1^K, \qquad z_{1j}=z_j^K-z_1^K$$

are the entries of B_K.

```
function Ch=convectionMatrices3D(bx, by, bz,T,k)

% Ch=convectionMatrices3D(bx, by, bz,T,k);
%
% Input:
%    bx,by,bz : vectorized functions of three variables
%    T  : enhanced triangulation
%    k  : polynomial degree
% Output:
%    Ch : 1 x 3 Cell array dim FE x dim FE convection matrices
%         \int b_i P_a \partial_i P_b
%
% Last modified: June 3, 2016

% Evaluations of coefficients and basis functions

formula=quadratureFEM(3*k+1,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);

B{1}=bx(x,y,z);
B{2}=by(x,y,z);
B{3}=bz(x,y,z);
P=bernstein3D(formula(:,2),formula(:,3),formula(:,4),k);
[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k);

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK

d{1,1}=1/6.*(y13.*z14 - y14.*z13);
d{1,2}=1/6.*(x14.*z13 - x13.*z14);
d{1,3}=1/6.*(x13.*y14 - x14.*y13);
d{2,1}=1/6.*(y14.*z12 - y12.*z14);
d{2,2}=1/6.*(x12.*z14 - x14.*z12);
d{2,3}=1/6.*(x14.*y12 - x12.*y14);
d{3,1}=1/6.*(y12.*z13 - z12.*y13);
d{3,2}=1/6.*(x13.*z12 - x12.*z13);
d{3,3}=1/6.*(x12.*y13 - x13.*y12);

% Loop over quadrature points

dk=size(Px,2);
Nelt=size(T.elements,2);
for i=1:3
    Ch{i}=zeros(dk,dk*Nelt);
end

for q=1:size(formula,1)
    Q{1}=formula(q,5)*P(q,:)'*Px(q,:);
    Q{2}=formula(q,5)*P(q,:)'*Py(q,:);
    Q{3}=formula(q,5)*P(q,:)'*Pz(q,:);

    for i=1:3
        for j=1:3
```

```
            Ch{i}=Ch{i}+kron(d{j,i}.*B{i}(q,:),Q{j});
        end
    end
end


% Assembly
[¬,Cols]=DOF3D(T,k);
Rows=permute(Cols,[2 1 3]);
for i=1:3
    Ch{i}=sparse(Rows(:),Cols(:),Ch{i}(:));
end
return
```

5.7 The transport matrix

The goal of this section is the computation of the transportation matrix of size $d_k \times d_{k+1}$ in the form of

$$\int_\Omega (\mathbf{b}\cdot\varphi_i)\nabla\varphi_j \qquad i=1,\dots,\dim V_h, \quad j=1,\dots,\dim V_{h+1},$$

where

$$\mathbf{b} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} : \Omega \to \mathbb{R}^3.$$

The code is similar to the case of the convection matrix (see 5.6) we will compute three matrices

$$\int_\Omega b_i \varphi_a \partial_{x_i}\varphi_b \qquad a=1,\dots,\dim V_h, \quad b=1,\dots,\dim V_{h+1}, \quad 1\le i\le 3.$$

These matrices will be computed from the local convection matrices

$$\int_K b_i P_\alpha^K \partial_i P_\beta^K, \qquad \alpha=1,\dots,d_k, \quad \beta=1,\dots,d_{k+1}, \quad K\in\mathcal{T}_h,$$

using a matrix assembly process.

Local stiffness matrices. With a change to the reference element, we obtain

$$\int_K b_i P_\alpha^K \partial_i P_\beta^K \approx \sum_q \sum_{j=1}^{3} d_{ji}^K b_i(\mathbf{p}_q^K)\big(\underbrace{\widehat{\omega}_q P_\alpha(\widehat{\mathbf{p}}_q)\partial_j P_\beta(\widehat{\mathbf{p}}_q)}_{\mathrm{Q}_j^q}\big),$$

where d_{ij}^K is the entires of D^K defined in section 5.6

Note: If you have a constant vector function $\mathbf{b}$, you can call `transportMatrices3D(T,k)` and multiply the resulting matrices with $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ respectively. This way you avoid looping through the quadratures points and obtain the matrices a lot faster.

```
function Ch=transportMatrices3D(varargin)

% Ch=transportMatrices3D(bx,by,bz,T,k);
% Ch=transportMatrices3D(T,k);
%
% Input:
%    bx,by,bz : vectorized functions of three variables
%    T  : enhanced triangulation
%    k  : polynomial degree
% Output:
%    Ch : 1 x 3 Cell array dim FE(P_k) x dim FE(P_{k+1}) convection matrices
```

```
%         \int b_i P_a \partial_i P_b
%
% Last modified: December 9, 2016

% Evaluations of coefficients and basis functions

if nargin == 2
    T=varargin{1};
    k=varargin{2};
elseif nargin == 5
    T=varargin{4};
    k=varargin{5};
end

formula=quadratureFEM(3*k+2,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);
P=bernstein3D(formula(:,2),formula(:,3),formula(:,4),k);
[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k+1);

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK

d{1,1}=1/6.*(y13.*z14 - y14.*z13);
d{1,2}=1/6.*(x14.*z13 - x13.*z14);
d{1,3}=1/6.*(x13.*y14 - x14.*y13);
d{2,1}=1/6.*(y14.*z12 - y12.*z14);
d{2,2}=1/6.*(x12.*z14 - x14.*z12);
d{2,3}=1/6.*(x14.*y12 - x12.*y14);
d{3,1}=1/6.*(y12.*z13 - z12.*y13);
d{3,2}=1/6.*(x13.*z12 - x12.*z13);
d{3,3}=1/6.*(x12.*y13 - x13.*y12);

% Loop over quadrature points

dk=size(P,2);
dkp1 = size(Px,2);
Nelt=size(T.elements,2);
for i=1:3
    Ch{i}=zeros(dk,dkp1*Nelt);
end

if nargin == 2
    P=bsxfun(@times,formula(:,5),P);
    Q{1} = P'*Px;
    Q{2} = P'*Py;
    Q{3} = P'*Pz;
    for i=1:3
        for j=1:3
            Ch{i}=Ch{i}+kron(d{j,i},Q{j});
        end
    end
elseif nargin == 5
    bx=varargin{1};
```

```
    by=varargin{2};
    bz=varargin{3};
    B{1}=bx(x,y,z);
    B{2}=by(x,y,z);
    B{3}=bz(x,y,z);
    for q=1:size(formula,1)
        Q{1}=formula(q,5)*P(q,:)'*Px(q,:);
        Q{2}=formula(q,5)*P(q,:)'*Py(q,:);
        Q{3}=formula(q,5)*P(q,:)'*Pz(q,:);
        for i=1:3
            for j=1:3
                Ch{i}=Ch{i}+kron(d{j,i}.*B{i}(q,:),Q{j});
            end
        end
    end
end

% Assembly

[¬,Colskp1]=DOF3D(T,k+1);
Colskp1(dk+1:dkp1,:,:) = [];

[¬,Cols]=DOF3D(T,k);
Cols = repmat(Cols(1,:,:),[dkp1 1 1]);
Rows=permute(Cols,[2 1 3]);

for i=1:3
    Ch{i}=sparse(Rows(:),Colskp1(:),Ch{i}(:));
end

return
```

5.8 The error function

error

Goal. With this function we compute

$$\left(\int_\Omega |u-u_h|^2\right)^{1/2} \qquad \text{and} \qquad \left(\int_\Omega |\nabla u-\nabla u_h|^2\right)^{1/2}$$

for given functions $(u,u_x,u_y,u_z):\Omega\to\mathbb{R}$ and $u_h\in V_h$.

Disassembling u_h. A discrete function $u_h\in V_h$ is going to be given as a $\dim V_h$-vector, say `uh`. If we write

```
Uh=uh(DOF3D(T,k))
```

we obtain a $d_k\times N_{\text{elt}}$ matrix with the local degrees of freedom. We will refer to this matrix as U_h.

Computation of the L^2 error. We proceed as follows

$$\begin{aligned}\int_\Omega |u-u_h|^2 &= \sum_K \int_K |u-u_h|^2\\ &\approx \sum_K\sum_q \omega_q |u(\mathbf{p}_q^K)-u_h(\mathbf{p}_q^K)|^2|K|.\end{aligned}$$

We create the $N_{\text{quad}}\times N_{\text{elt}}$ matrices X, Y, Z with the coordinates of all the quadrature points of all the elements. This will allow us to evaluate u easily. Let now P be the $N_{\text{quad}}\times d_k$ matrix

$$\mathrm{P}_{q\ell}=P_\ell(\widehat{\mathbf{p}}_q)\qquad q=1,\dots,N_{\text{quad}},\quad \ell=1,\dots,d_k.$$

Then

$$u(\mathrm{X},\mathrm{Y},\mathrm{Z}) - \mathrm{P}\,\mathrm{U}_h$$

is the $N_{\mathrm{quad}} \times N_{\mathrm{elt}}$ matrix with the values of $u - u_h$ at all quadrature points.

Computation of the H^1 seminorm. We decompose and approximate

$$\int_\Omega |\partial_j u - \partial_j u_h|^2 \approx \sum_K \sum_q \omega_q |\partial_j u(\mathbf{p}_q^K) - \partial_j u_h(\mathbf{p}_q^K)|^2 |K|.$$

The evaluation of $\partial_j u$ at all the quadrature points is standard. To differentiate u_h not that if

$$\begin{aligned} \mathrm{C}^K &= (\mathrm{B}^K)^{-1} = \begin{bmatrix} c_{11}^K & c_{12}^K & c_{13}^K \\ c_{21}^K & c_{22}^K & c_{23}^K \\ c_{31}^K & c_{32}^K & c_{33}^K \end{bmatrix} \\ &= \frac{1}{6|K|} \begin{bmatrix} y_{13}z_{14} - y_{14}z_{13} & z_{13}x_{14} - z_{14}x_{13} & x_{13}y_{14} - x_{14}y_{13} \\ y_{14}z_{12} - y_{12}z_{14} & z_{14}x_{12} - z_{12}x_{14} & x_{14}y_{12} - x_{12}y_{14} \\ y_{12}z_{13} - y_{13}z_{12} & z_{12}x_{13} - z_{13}x_{12} & x_{12}y_{13} - x_{13}y_{12} \end{bmatrix}, \end{aligned}$$

(the matrix C^K used in the computation of the stiffness matrices is scaled slightly differently), then

$$(\nabla u_h|_K)(\mathbf{p}_q^K) = \sum_{\ell=1}^{d_k} u_\ell^K \mathbf{C}_K^\top \nabla P_\ell(\widehat{\mathbf{p}}_q)$$

and therefore, denoting

$$\partial_j u_h|_K(\mathbf{p}_q^K) = \sum_{\ell=1}^{d_k} u_\ell^K \sum_{i=1}^{3} c_{ij}^K \partial_i P_\ell(\widehat{\mathbf{p}}_q).$$

Storing the coefficients c_{ik}^K in *row vectors* $\mathbf{c}_{ij} \in \mathbb{R}^{N_{\mathrm{elt}}}$ and evaluating the derivatives of the basis functions in matrices

$$\mathrm{P}_{q\ell}^i = \partial_i P_\ell(\widehat{\mathbf{p}}_q) \qquad q = 1,\ldots,N_{\mathrm{quad}}, \quad \ell = 1,\ldots,d_k, \qquad i = 1,2,3,$$

we can compute everything quite fast. The $N_{\mathrm{quad}} \times N_{\mathrm{elt}}$ matrix with the values of $\partial_j u_h$ at all the quadrature points is given by

$$\sum_{i=1}^{3} \mathrm{P}^i(\mathbf{c}_{ji} \odot \mathrm{U}_h).$$

```
function [eh,hh]=errorFEM3D(u,uh,T,k)

% eh=errorFEM3D(u,uh,T,k)
% [eh,hh]=errorFEM3D({u,ux,uy,uz},uh,T,k)
%
% Input:
%    u    : vectorized function of two variables, or
%    u,ux,uy,uz  : vectorized functions of three variables
%    uh   : dim FE-vector for a FE function
%    T    : basic triangulation
%    k    : polynomial degree
% Output:
%    eh   : L2 error
%    hh   : H1 error
% Last modified: May 1, 2015

T=edgesAndFaces(T);
T=enhanceGrid3D(T);
formula=quadratureFEM(4*k,3);
```

```
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);
if iscell(u)
    U=u{1}(x,y,z);
    Ux=u{2}(x,y,z);
    Uy=u{3}(x,y,z);
    Uz=u{4}(x,y,z);
else
    U=u(x,y,z);
end
Uh=uh(DOF3D(T,k));
uh=bernstein3D(formula(:,2),formula(:,3),formula(:,4),k)*Uh;
eh=sqrt(formula(:,5)'*abs(U-uh).^2*T.volume.');
if ~iscell(u)
    hh=0;
    return
end

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK

sqdet=1./(6*T.volume);
c11=sqdet.*(y13.*z14 - y14.*z13);
c12=sqdet.*(x14.*z13 - x13.*z14);
c13=sqdet.*(x13.*y14 - x14.*y13);
c21=sqdet.*(y14.*z12 - y12.*z14);
c22=sqdet.*(x12.*z14 - x14.*z12);
c23=sqdet.*(x14.*y12 - x12.*y14);
c31=sqdet.*(y12.*z13 - z12.*y13);
c32=sqdet.*(x13.*z12 - x12.*z13);
c33=sqdet.*(x12.*y13 - x13.*y12);


[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k);
uhx=Px*bsxfun(@times,c11,Uh)+Py*bsxfun(@times,c21,Uh)+Pz*bsxfun(@times,c31,Uh);
uhy=Px*bsxfun(@times,c12,Uh)+Py*bsxfun(@times,c22,Uh)+Pz*bsxfun(@times,c32,Uh);
uhz=Px*bsxfun(@times,c13,Uh)+Py*bsxfun(@times,c23,Uh)+Pz*bsxfun(@times,c33,Uh);
hh=sqrt(formula(:,5)'*(abs(Ux-uhx).^2+abs(Uy-uhy).^2 +abs(Uz-uhz).^2)*T.volume.');

return
```

6 Boundary conditions

6.1 Computations on the boundary

The function `bdWork3D.m` carries out a series of computations on parts of the boundary, depending on a number of input data. Let us first explain what these possible computations are and how they are carried out. Let us assume that we have a collection of faces (counting with the global face number) `list`, which

we will assume are boundary faces, although this is not noticed by the code. We will write

$$\mathcal{F}_h^{\text{list}} = \{F_\ell \in \mathcal{F}_h \,:\, \ell \in \texttt{list}\} \equiv \texttt{list}, \qquad \Gamma_{\text{list}} := \bigcup_{F \in \mathcal{F}_h^{\text{list}}} F.$$

We will compute several kinds of vectors and matrices related to Γ_{list}:

(a) Traction vectors for scalar and vector fields (boundary tests)

$$\int_{\Gamma_{\text{list}}} g\varphi_i \qquad \text{or} \qquad \int_{\Gamma_{\text{list}}} (\mathbf{g}\cdot\boldsymbol{\nu})\varphi_i, \qquad i = 1, \dots, \dim V_h,$$

(b) Traction vectors for given stress tensors

$$\left[\int_{\Gamma_{\text{list}}} (\sigma^{xx}, \sigma^{xy}, \sigma^{xz})\cdot\boldsymbol{\nu}\varphi_i \qquad \int_{\Gamma_{\text{list}}} (\sigma^{xy}, \sigma^{yy}, \sigma^{yz})\cdot\boldsymbol{\nu}\varphi_i \qquad \int_{\Gamma_{\text{list}}} (\sigma^{xz}, \sigma^{yz}, \sigma^{zz})\cdot\boldsymbol{\nu}\varphi_i\right],$$

(c) Boundary mass matrices with unit mass

$$\int_{\Gamma_{\text{list}}} \varphi_i\varphi_j \qquad i,j = 1, \dots, \dim V_h,$$

or with variable mass

$$\int_{\Gamma_{\text{list}}} \lambda\varphi_i\varphi_j \qquad \text{or} \qquad \int_{\Gamma_{\text{list}}} (\boldsymbol{\lambda}\cdot\boldsymbol{\nu})\varphi_i\varphi_j \qquad i,j = 1, \dots, \dim V_h,$$

(d) Boundary projections. A boundary projection of a function u_D is a V_h vector u_h such that all DOF that are not on Γ_{list} vanish while the other ones are computed using an $L^2(\Gamma_{\text{list}})$ projection, so that

$$\int_{\Gamma_{\text{list}}} u_h w_h = \int_{\Gamma_{\text{list}}} u_D w_h \qquad \forall w_h \in V_h.$$

This means that we have to solve a linear system

$$\sum_{j \in \text{DOF(list)}} \left(\int_{\Gamma_{\text{list}}} \varphi_i\,\varphi_j\right) u_j = \int_{\Gamma_{\text{list}}} u_D\varphi_i, \qquad i \in \text{DOF(list)},$$

where DOF(list) is the list of all degrees of freedom contained in $\mathcal{F}_h^{\text{list}}$.

Traction vectors. We first compute the local traction vectors

$$\int_F g\,Q_\alpha^F \qquad \text{or} \qquad \int_F (\mathbf{g}\cdot\boldsymbol{\nu}_F)\,Q_\alpha^F, \qquad \alpha = 1, \dots, \delta_k, \quad F \in \mathcal{F}_h^{\text{list}}.$$

To do this, we first evaluate

$$g_q^F = \begin{cases} |F| g(\mathbf{q}_q^F), & q = 1, \dots, N_{\text{quad}}, \\ \mathbf{g}(\mathbf{q}_q^F)\cdot\boldsymbol{\nu}_F, & q = 1, \dots, N_{\text{quad}}, \end{cases} \qquad \text{where} \qquad \mathbf{q}_q^F = \boldsymbol{\phi}_F(\widehat{\mathbf{q}}_q).$$

(Note that $|\boldsymbol{\nu}_F| = |F|$.) The construction of all quadrature points can be done in a similar way to what was done in the assembly of the load vector. The normal vectors are stored in the field `T.normals`. They have to be used only on the `list` faces. The local computation is finished with

$$\sum_q \omega_q Q_\alpha(\widehat{\mathbf{q}}_q) g_q^F.$$

After that, we apply an assembly process on the $\mathcal{F}_h^{\text{list}}$ faces. The result is a $\dim V_h \times 1$ column vector. The code allows for an input where instead of a single scalar input g, we bring functions $g_1, \dots, g_L$ and compute test vectors

$$\int_{\Gamma_{\text{list}}} g_\ell\,\varphi_i \qquad i = 1, \dots, \dim V_h, \quad \ell = 1, \dots, L,$$

outputting a $\dim V_h \times L$ matrix.

Tractions vectors for given stress. We will also accept that we bring six functions corresponding to the upper triangular components of a symmetric tensor

$$\begin{bmatrix} \sigma_1 & \sigma_2 & \sigma_3 \\ \sigma_2 & \sigma_4 & \sigma_5 \\ \sigma_3 & \sigma_5 & \sigma_6 \end{bmatrix}.$$

The process to compute the traction created by each of the three rows (or columns) of this tensor is exactly the same as the one to compute the traction (flux) created by a vector field $\mathbf{g}$. The output is a $\dim V_h \times 3$ matrix.

Boundary mass matrix with variable mass. We need to compute the local matrices

$$\int_F \lambda\, Q_\alpha^F Q_\beta^F \qquad \text{or} \qquad \int_F (\boldsymbol{\lambda}\cdot\boldsymbol{\nu}_F)\, Q_\alpha^F Q_\beta^F, \qquad\qquad \alpha,\beta = 1,\dots,\delta_k, \quad F \in \mathcal{F}_h^{\text{list}}.$$

The computations are similar to what has been done previously. We start by evaluating

$$\lambda_q^F = \begin{cases} |F|\lambda(\mathbf{q}_q^F), & q = 1,\dots,N_{\text{quad}}, \\ \boldsymbol{\lambda}(\mathbf{q}_q^F)\cdot\boldsymbol{\nu}_F, & q = 1,\dots,N_{\text{quad}}, \end{cases} \qquad \text{where} \qquad \mathbf{q}_q^F = \boldsymbol{\phi}_F(\widehat{\mathbf{q}}_q),$$

and then compute the matrices

$$\sum_q \lambda_q^F \omega_q Q_\alpha(\widehat{\mathbf{q}}_q) Q_\beta(\widehat{\mathbf{q}}_q).$$

To do this, we loop in q and use Kronecker products in F. Finally, a matrix assembly process on $\mathcal{F}_h^{\text{list}}$ faces is applied.

Boundary mass matrix with unit mass. This is a much simpler computation, which does not require looping over quadrature points. We compute

$$\int_F Q_\alpha^F Q_\beta^F = |F| \sum_q \omega_q Q_\alpha(\widehat{\mathbf{q}}_q) Q_\beta(\widehat{\mathbf{q}}_q) \qquad \alpha,\beta = 1,\dots,\delta_k, \qquad F \in \mathcal{F}_h^{\text{list}},$$

which can be done with evaluations of the local basis functions and matrix-matrix multiplications, with a final Kronecker product to scale the fixed matrix

$$\sum_q \omega_q Q_\alpha(\widehat{\mathbf{q}}_q) Q_\beta(\widehat{\mathbf{q}}_q)$$

by the element areas.

Projections. Assume now that F is a $\dim V_h \times M$ matrix produced by testing as above. Note that $F_i = 0$ (the i-th row of F) if $i \notin$ L:=DOF(List), the set of all degrees of freedom associated to faces in $\mathcal{F}_h^{\text{list}}$. We have also computed a boundary mass matrix M, whose only non-zero block, $\mathrm{M}_{\mathrm{L}\times\mathrm{L}}$, is symmetric and positive definite. The computation

$$\mathrm{U}_{\mathrm{L}} = \mathrm{M}_{\mathrm{L}\times\mathrm{L}}^{-1}\mathrm{F}_{\mathrm{L}}$$

provides the values of the DOF(List) degrees of freedom for each of the columns of the matrix U, whose other entries will be assigned zero.

Modes of operation. Thus far we have explained what the function `bdWork3D` is capable of producing, and spent little time explaining how to get the function to perform the desired computation. Given a vectorized function (or array of vectorized functions), enhanced data structure for a tetrahedral mesh, polynomial degree for the polynomial basis functions, and the list of faces where we want to perform the computations, we need to indicate what we want computed. The first decision to make is with the input `goal`, which can take one of three values:

- `'test'`, this tells the function that the desired output is to be a traction vector or matrix whose columns are traction vectors.
- `'proj'`, which indicates that the output is to be an $L^2(\Gamma_{\text{list}})$ projection computed as discussed above.

In addition to a goal, `bdWork` needs to know how to treat the input functions. This information is communicated through the string `testmode`, which can also take three values:

- `'sc'` indicates one or many scalar functions to be tested or projected separately,
- `'fl'` indicates three functions which will be treated as the components of a vector function (think gradient) to be dotted with the normal vectors of the `list` faces.
- `'tr'` indicates six functions which represent the upper triangular components of a 3×3 symmetric tensor. These six functions must be entered in the order

$$\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6\},$$

so that they will grouped appropriately by row according to

$$\begin{bmatrix} \sigma_1 & \sigma_2 & \sigma_3 \\ \sigma_2 & \sigma_4 & \sigma_5 \\ \sigma_3 & \sigma_5 & \sigma_6 \end{bmatrix}.$$

```
function [fh,M,dof,ML]=bdWork3D(f,T,k,list,goal,testmode,lambda)

% [fh,M,dof]=bdWork3D(f,T,k,list,goal,tstmod,lambda)
%
% Input:
%     f      : cell array with vectorized functions
%     T      : enhanced tetrahedrization
%     k      : polynomial degree
%     list   : row vector (sublist of the set of faces)
%     goal   : 'test', 'proj', 'eval'
%     tstmod : 'sc' (scalar), 'fl' (flux), 'tr' (traction)
%     lambda : array of vectorized functions of three variables
%              of one vectorized function of three variables
%              ALTERNATIVELY, any non-functional input (treated as empty)
% Output:
%     fh     : dim V_h long vector (see later to see what it does)
%     M      : dim V_h x dim V_h matrix
%                  int_{\Gamma_list} P_i P_j,
%                  \Gamma_list=\cup_{i\in list} F_i
%     dof    : dim P_k(F) x #list matrix with DOF for faces F_i, i\in list
%     ML      : dim V_h x dim V_h matrix
%                  int_{\Gamma_list} lambda P_i P_j,
%                  of int_{\Gamma_list} (lambda\dot n) P_i P_j
%
% Last modified: January 19, 2017

% Quadrature points and geometric quantities
```

```
form = quadratureFEM(3*k,2);

x = T.coordinates(1,:);
y = T.coordinates(2,:);
z = T.coordinates(3,:);
x = form(:,1:3)*x(T.faces(1:3,list));
y = form(:,1:3)*y(T.faces(1:3,list));
z = form(:,1:3)*z(T.faces(1:3,list));

P = bernstein2D(form(:,2),form(:,3),k);
Pw = bsxfun(@times,form(:,4),P);

normx = T.normals(1,list);
normy = T.normals(2,list);
normz = T.normals(3,list);
areas = T.area(list);

[dof,Assem]=computeBDDOF3D(T,k,list);
dimVh=dimFEMspace(T,k);

% Testing on separate faces

if testmode=='sc'
    for ell=1:length(f)
        fh{ell}=bsxfun(@times,areas,f{ell}(x,y,z));
    end
elseif testmode=='fl'
    fh{1}=bsxfun(@times,normx,f{1}(x,y,z))...
            +bsxfun(@times,normy,f{2}(x,y,z))...
            +bsxfun(@times,normz,f{3}(x,y,z));
elseif testmode=='tr'
    fh{1} = bsxfun(@times,normx,f{1}(x,y,z))...
                +bsxfun(@times,normy,f{2}(x,y,z))...
                +bsxfun(@times,normz,f{3}(x,y,z));
    fh{2} = bsxfun(@times,normx,f{2}(x,y,z))...
                +bsxfun(@times,normy,f{4}(x,y,z))...
                +bsxfun(@times,normz,f{5}(x,y,z));
    fh{3} = bsxfun(@times,normx,f{3}(x,y,z))...
                +bsxfun(@times,normy,f{5}(x,y,z))...
                +bsxfun(@times,normz,f{6}(x,y,z));
end
for ell=1:length(fh)
    fh{ell}=Pw'*fh{ell};
    fh{ell}=accumarray(dof(:),fh{ell}(:),[dimVh,1]);
end
if testmode=='sc'
    FF=sparse(dimVh,length(f));
    for ell=1:length(f);
        FF(:,ell)=fh{ell};
    end
    fh=FF;
elseif testmode=='fl'
    fh=fh{1};
elseif testmode=='tr'
    fh=[fh{1} fh{2} fh{3}];
end

% Boundary mass matrix with unit coefficient

M=kron(areas,P'*Pw);
Rows=permute(Assem,[2 1 3]);
M=sparse(Rows(:),Assem(:),M(:),dimVh,dimVh);


% L2 projection: only used when goal='proj' mode
```

```
if goal=='proj'
    active=unique(dof(:));
    fh(active,:)=M(active,active)\fh(active,:);
end

% Boundary mass matrix with variable coefficient

if iscell(lambda) || isa(lambda,'function_handle')
    if iscell(lambda)
        Lambda=bsxfun(@times,normx,lambda{1}(x,y,z))...
               +bsxfun(@times,normy,lambda{2}(x,y,z))...
               +bsxfun(@times,normz,lambda{3}(x,y,z));
    else
        Lambda=bsxfun(@times,areas,lambda(x,y,z));
    end
    dk=size(P,2);
    ML=zeros(dk,dk*length(list));
    for q=1:size(form,1)
        ML=ML+kron(Lambda(q,:),form(q,4)*P(q,:)'*P(q,:));
    end
    ML=sparse(Rows(:),Assem(:),ML(:),dimVh,dimVh);
else
    ML = 0;
end

return
```

6.2 Neumann boundary conditions

NBC

Given the appropriate data, the function `neumannBC3D.m` computes a traction vector or matrix whose columns are traction vectors on the Neumann faces of a 3D tetrahedral mesh. As data, the function can take a single vectorized function, trio of vectorized functions or sextet of vectorized functions. The resulting computations represent:

- traction vectors for scalar and vector fields

$$\int_{\Gamma_N} g\varphi_i \qquad \text{or} \qquad \int_{\Gamma_N} (\mathbf{g}\cdot\boldsymbol{\nu})\varphi_i, \qquad i = 1,\ldots,\dim V_h,$$

- traction vectors for given symmetric stress tensors

$$\left[\int_{\Gamma_N} (\sigma^{xx},\sigma^{xy},\sigma^{xz})\cdot\boldsymbol{\nu}\varphi_i \qquad \int_{\Gamma_N} (\sigma^{xy},\sigma^{yy},\sigma^{yz})\cdot\boldsymbol{\nu}\varphi_i \qquad \int_{\Gamma_N} (\sigma^{xz},\sigma^{yz},\sigma^{zz})\cdot\boldsymbol{\nu}\varphi_i\right],$$

- traction vectors for given 'non-symmetric stress tensors.'

$$\left[\int_{\Gamma_N} (\sigma^{xx},\sigma^{xy},\sigma^{xz})\cdot\boldsymbol{\nu}\varphi_i \qquad \int_{\Gamma_N} (\sigma^{yx},\sigma^{yy},\sigma^{yz})\cdot\boldsymbol{\nu}\varphi_i \qquad \int_{\Gamma_N} (\sigma^{zx},\sigma^{zy},\sigma^{zz})\cdot\boldsymbol{\nu}\varphi_i\right].$$

In addition, given a vectorized function λ as additional input, `neummanBC3D` produces a boundary mass matrix using the input function as a variable mass.
Everything is computed using the function `bdWork3D` passing in the list of Neumann faces as the `list` variable (see the above documentation for `bdWork3D`). The input functions come as a cell-array except in the scalar case. We admit input of the following forms:

- a scalar function handle;
- a 3-cell array with function handles;

- a 6-cell array with function handles, corresponding to a symmetric tensor numbered in the following way

$$\begin{bmatrix} \sigma_1 & \sigma_2 & \sigma_3 \\ \sigma_2 & \sigma_4 & \sigma_5 \\ \sigma_3 & \sigma_5 & \sigma_6 \end{bmatrix}$$

- a 3×3 cell array.

The 'testmode' needed for `bdWork3D` is determined by the number of data functions passed in: `'sc'` for one function, `'fl'` for three, and `'tr'` for six. When we bring in nine data, we invoke `bdWork3D` three times, each of them with a row of the cell array.

Important points about empty Neumann boundaries.

- In the case that `T.neumann` is empty (i.e. all boundary faces have been selected as Dirichlet), the output for `gh` should be a
  1. $\dim V_h \times 1$ matrix of zeros if the input `g` is one vectorized function or if `g` is three vectorized functions (to yield $\mathbf{g} \cdot \mathbf{n}$)
  2. $\dim V_h \times 3$ matrix of zeros if the input `g` are six or 3×3 vectorized functions.
- The boundary mass matrix `MBd` is computed only if there is a fourth input parameter. Otherwise, the output for the boundary mass is just a zero value.

```
function [gh,MBd]=neumannBC3D(g,T,k,varargin)

% [gh]=neumannBC3D(g,T,k)
% [gh,MBd]=neumannBC3D(g,T,k,lambda)
% Input:
%     g      : a single vectorized function of three variables
%                      or
%              array of three vectorized functions of three vars {gx,gy,gz}
%                      or
%              array of six vectorized function of three variables
%                      {sigmaxx,sigmaxy,sigmaxz,sigmayy,sigmayz,sigmazz}
%              3x3 cell array of vectorized functions of three variables
%     T      : enhanced triangulation
%     k      : polynomial degree
%     lambda : array of vectorized functions of three variables
%                      or
%              a single vectorized function of three variables
%                (optional)
% Output:
%     gh     : dimVh x 1boundary vector int_{Gamma}(g or g dot n) \phi_i
%                   or
%              dimVh x 3 matrix \int_{Gamma} sigma n \phi_i
%     MBd    : boundary mass matrix int_{Gamma}lambda \phi_i \phi_j for LHS
%                (only when lambda is specified)
%
% Last modified: November 21, 2017

if ¬iscell(g)
    g={g};
end
param=length(g(:));   % 1, 3, 6, or 9 (for 3x3 case)
if size(T.neumann,2)==0
    dimVh=dimFEMspace(T,k);
    switch param
        case {6,9}
            gh=zeros(dimVh,3);
```

```
        case {1,3}
            gh=zeros(dimVh,1);
    end
    MBd=sparse(dimVh,dimVh);
    return
end

neuFaces=find(T.faces(4,:)==2);
switch param
    case 1
        testmode='sc';
    case {3,9}
        testmode='fl';
    case 6
        testmode='tr';
end
if nargin==4
    lambda=varargin{1};
else
    lambda=0;
end
if param==9
    [gh(:,1),¬,¬,MBd]=bdWork3D({g{1,:}},T,k,neuFaces,'test',testmode,lambda);
    gh(:,2)=bdWork3D({g{2,:}},T,k,neuFaces,'test',testmode,lambda);
    gh(:,3)=bdWork3D({g{3,:}},T,k,neuFaces,'test',testmode,lambda);
else
    [gh,¬,¬,MBd]=bdWork3D(g,T,k,neuFaces,'test',testmode,lambda);
end

return
```

6.3 Dirichlet Boundary Conditions

The function `dirichletBC3D.m` is designed to calculate a V_h vector u_h which such that all non-Dirichlet DOF for u_h vanish and

$$\int_{\Gamma_D} u_h w_h = \int_{\Gamma_D} u_D w_h \qquad \forall w_h \in V_h.$$

for some condition u_D on the Dirichlet boundary Γ_D. We do not impose the Dirichlet BC by interpolation, but by using an $L^2(\Gamma_D)$ projection. We have to solve a linear system

$$\sum_{j \in \mathrm{Dir}} \left(\int_{\Gamma_D} \varphi_i \, \varphi_j \right) u_j = \int_{\Gamma_D} u_D \varphi_i, \qquad i \in \mathrm{Dir},$$

where Dir is the list of Dirichlet nodes. The function `dirichletBC3D.m` is designed, if necessary, to handle an array of functions `u{i}`, and output a matrix such that each column is the test vector for each function in the input array.
For non-empty Dirichlet boundary, all of the computations are carried out by the function `bdWork.m`, where the list of faces are the Dirichlet faces, the goal is set to `'proj'`, and the testmode is set to `'sc'` (see above for the documentation of `bdWork3D`).

An important point about empty Dirichlet boundaries. In the case that `T.dirichlet` is empty (i.e. all boundary faces have been selected as Neumann), the output for `uh` should be a

1. $\dim V_h \times 1$ matrix of zeros if the input `u` is one vectorized function
2. $\dim V_h \times M$ matrix of zeros if the input `u` is an array of M vectorized functions.

DOF lists. In addition to the vector or matrix output, `dirichletBC3D.m` outputs a list of Dirichlet and Non-Dirichlet degrees of freedom. For empty Dirichlet boundaries, the Dirichlet list is empty, and the 'free' list consists of all numbers from 1 to $\dim V_h$. For non-empty Dirichlet boundary, the list of Dirichlet nodes by face are computed in a matrix, `dirDOF`, by `bdWork3D.m`. The 'dir' list is then made by extracting the unique numbers from this matrix, and the 'free' list by taking the set difference of the numbers 1 to $\dim V_h$ and the 'dir' list.

```
function [uh,dir,free]=dirichletBC3D(u,T,k)

% [uh,dir,free]=dirichletBC3D(u,T,k)
%
% Input:
%     u    : vectorized function of three variables
%              or cell array with M vectorized functions of three vars
%     T    : enhanced triangulation
%     k    : polynomial degree
% Output:
%    uh    : dim FE_h column vector with assigned Dirichlet DOF
%               or dim FE_h x M matrix
%    dir   : List of Dirichlet degrees of freedom
%    free  : List of ALL non Dirichlet DOF
%
% Last modified: January 24, 2017

dimVh=dimFEMspace(T,k);
if size(T.dirichlet,2)==0
    if iscell(u)
        uh=zeros(dimVh,length(u));
    else
    uh=zeros(dimVh,1);
    end
    dir=unique([]); free=(1:dimVh)';
    return
end
dirFaces=find(T.faces(4,:)==1);
if isa(u,'function_handle')
    u={u};
end
[uh,¬,dirDOF,¬]=bdWork3D(u,T,k,dirFaces,'proj','sc',0);
dir  =unique(dirDOF(:));
free =(1:dimVh)'; free(dir)=[];

return
```

7 Sub-tetrahedrization tools

The subtet data structure. A subtetrahedrization data structure has four fields that tell which vertices (`V`), edges (`E`), faces (`F`), and elements (`EL`) of a given tetrahedrization `T` are kept when choosing a sub-collection of elements.
This structure is easily produced from a tetrahedrization and a list of elements that we want to select to build the sub-tetrahedrization. The four fields in the subtetrahedrization data structure are row vectors with indices given in increasing order.

```
SubT =
     V: [1x30 double]
     E: [1x108 double]
     F: [1x129 double]
    EL: [1x50 double]
```

In the above example, we are choosing 50 elements, that involve 129 faces, 108 edges, and 30 vertices of an original tetrahedrization.

Creating a subtetrahedrization. Imagine that we are given an enhanced tetrahedrization and a list of marked elements, telling us which elements we want to keep. The function `subtet` returns the subtetrahedrization with:

- the elements, vertices, edges, and faces given in increasing order from the original list (nothing is rearranged);
- the reduced lists of Dirichlet and Neumann faces; note that the new boundary faces (corresponding to recently introduced interfaces) do not appear in any list and are not identified at all;
- we also output the subtetrahedrization data structure that will allow us to recover the location of the degrees of freedom from the subtetrahedrization in the global tetrahedrization.

The process to create the subtetrahedrization is as follows:

- We make lists of vertices, edges, and faces that are kept in the tetrahedrization. All this information is stored in the subtet data structure `SubT` that is exported. (It is needed to create the embedding operator for the finite element spaces.)
- We chop the basic fields `coordinates`, `elements`, `edges`, `faces`, `edgebyelt`, `facebyelt` and `faceorient` from `T` and copy them in the new tetrahedrization `Tnew`.
- In the case of `Tnew.edgebyelt` we copy only the absolute value, so momentarily we lose the orientation of the edge.
- Next we renumber all the entries of the above matrices (except `T.faceorientation` which contains entries from 1 to 3). This is done with three transfer vectors. For instance `transV` is a $1 \times N_{\mathrm{nd}}$ vector containing zeros for vertices of `T` that are not in `Tnew` and the numbers 1 to $N_{\mathrm{nd}}(\mathcal{T}_h^{\mathrm{new}})$ in increasing order, in the location of the vertices that are kept.
- In the case of `Tnew.edgebyelt`, we recover the orientation after renumbering, by using the corresponding columns of the large tetrahedrization.
- Finally, the Dirichlet and Neumann faces are recovered from `Tnew.faces` by looking at the last row (1 for Dirichlet, 2 for Neumann)

```
function [Tnew,SubT]=subtet(T,Elts)

% [Tnew,SubT]=subtet(T,Elts)
%
% Input:
%     T    : fully enhanced tetrahedrization
%     Elts : a list of integers which represents the 'marked' elements
% Output:
%     Tnew : fully enhanced tetrahedrization
%     SubT : subtet data structure
%              SubT.V : (row) list of marked vertices
%              SubT.E : (row) list of marked edges
%              SubT.F : (row) list of marked faces
%              SubT.EL: (row) list of marked elements
%
% Last modified: October 28, 2016

V = T.elements(:,Elts);
SubT.V = unique(V(:)');
E = abs(T.edgebyelt(:,Elts));
```

```
SubT.E = unique(E(:)');
F = T.facebyelt(:,Elts);
SubT.F = unique(F(:)');
SubT.EL = sort(Elts);

Nvert = size(T.coordinates,2);
Nedg  = size(T.edges,2);
Nface = size(T.faces,2);
Nelts = size(T.elements,2);

NNvert = length(SubT.V);
NNedg  = length(SubT.E);
NNface = length(SubT.F);
NNelts = length(SubT.EL);

Tnew.coordinates=T.coordinates(:,SubT.V);
Tnew.elements   =T.elements(:,SubT.EL);
Tnew.edges      =T.edges(:,SubT.E);
Tnew.faces      =T.faces(:,SubT.F);
Tnew.facebyelt  =T.facebyelt(:,SubT.EL);
Tnew.edgebyelt  =abs(T.edgebyelt(:,SubT.EL));
Tnew.faceorient =T.faceorient(:,SubT.EL);

transV=zeros(1,Nvert);
transV(SubT.V)=1:NNvert;
transE=zeros(1,Nedg);
transE(SubT.E)=1:NNedg;
transF=zeros(1,Nface);
transF(SubT.F)=1:NNface;

Tnew.elements(:) =transV(Tnew.elements(:));
Tnew.edges(1:2,:)=transV(Tnew.edges(1:2,:));
Tnew.faces(1:3,:)=transV(Tnew.faces(1:3,:));
Tnew.facebyelt(:)=transF(Tnew.facebyelt(:));
Tnew.edgebyelt(:)=transE(Tnew.edgebyelt(:));
Tnew.edgebyelt   =sign(T.edgebyelt(:,SubT.EL)).*Tnew.edgebyelt;

Tnew.dirichlet = Tnew.faces(1:3, find(Tnew.faces(4,:) == 1));
Tnew.neumann   = Tnew.faces(1:3, find(Tnew.faces(4,:) == 2));

Tnew=enhanceGrid3D(Tnew);

return
```

Embedding Finite Element Spaces. We will always assume that in a subtetrahedrization the orientations of edges, faces, and elements is the same one as the orientations in the original tetrahedrization. Let us denote $\mathcal{T}_h$ and $\mathcal{T}_h^{\text{new}}$ for a tetrahedrization and a subtetrahedrization of the same domain. The embedding operator is a vector with $\dim V_h(\mathcal{T}_h^{\text{new}})$ entries corresponding to numbers between 1 and $\dim V_h(\mathcal{T}_h)$. The construction is very similar to what is used in Section 2.2 to count all degrees of freedom in `DOF3D.m`. Say that the subtetrahedrization data structure `SubT` contains the following vectors:

$$V = \left[\begin{array}{cccc} V_1 & V_2 & \dots & V_{\text{nv}} \end{array}\right]$$
$$E = \left[\begin{array}{cccc} E_1 & E_2 & \dots & E_{\text{ne}} \end{array}\right]$$
$$F = \left[\begin{array}{cccc} F_1 & F_2 & \dots & F_{\text{nf}} \end{array}\right]$$
$$EL = \left[\begin{array}{cccc} K_1 & K_2 & \dots & K_{\text{nel}} \end{array}\right].$$

Let us denote, as usual,

$$e_k = k-1, \qquad f_k = \tfrac{1}{2}(k-1)(k-2), \qquad t_k = \tfrac{1}{6}(k-1)(k-2)(k-3).$$

- The degrees of freedom for the vertices are just listed in V

$$\left[\begin{array}{cccc} V_1 & V_2 & \dots & V_{\text{nv}} \end{array}\right].$$

- The degrees of freedom for the edges are the elements of this matrix

$$N_{\mathrm{nd}} + \begin{bmatrix} 1 \\ 2 \\ \vdots \\ e_k \end{bmatrix} \oplus \left(e_k \begin{bmatrix} E_1 - 1 & E_2 - 1 & \dots & E_{ne} - 1 \end{bmatrix} \right),$$

  where we are using $\oplus$ for the vectorized sum of a column times a row (that outputs a matrix with all the possible crossed sums).

- The degrees of freedom for the faces are the elements of this matrix

$$N_{\mathrm{nd}} + e_k N_{\mathrm{edg}} + \begin{bmatrix} 1 \\ 2 \\ \vdots \\ f_k \end{bmatrix} \oplus \left(f_k \begin{bmatrix} F_1 - 1 & F_2 - 1 & \dots & F_{nf} - 1 \end{bmatrix} \right).$$

- The degrees of freedom for the interior of the tetrahedra are

$$N_{\mathrm{nd}} + e_k N_{\mathrm{edg}} + f_k N_{\mathrm{fc}} + \begin{bmatrix} 1 \\ 2 \\ \vdots \\ t_k \end{bmatrix} \oplus \left(t_k \begin{bmatrix} K_1 - 1 & K_2 - 1 & \dots & K_{nel} - 1 \end{bmatrix} \right).$$

Here N_{nd}, N_{edg}, and N_{fc} refer to the large triangulation) Reading these matrices by columns, in order (first vertices, then edges, then faces, then elements) provides the embedding operator.

```
function Where=embed(Tbig,SubT,k)

% Where=embed(Tbig,SubT,k)
%
% Input:
%    Tbig   : full enhanced tetrahedrization
%    SubT   : SubTet data structure
%    k      : polynomial degree
% Output:
%    Where  : dimVh(Tsmall) column vector with embedding operator from
%               Vh(Tsmall) to Vh(Tbig)
%
% Last modified: September 16, 2016

Nvert=size(Tbig.coordinates,2);
Nedge=size(Tbig.edges,2);
Nface=size(Tbig.faces,2);

dimE  =k-1;
dimF  =(k-1)*(k-2)/2;
dimEL =(k-1)*(k-2)*(k-3)/6;

Vert =SubT.V;
Edges=bsxfun(@plus,(1:dimE)',dimE*(SubT.E-1));
Faces=bsxfun(@plus,(1:dimF)',dimF*(SubT.F-1));
Elts =bsxfun(@plus,(1:dimEL)',dimEL*(SubT.EL-1));
Where=[Vert(:);...
       Edges(:)+Nvert;...
       Faces(:)+Nvert+Nedge*dimE;...
       Elts(:)+Nvert+Nedge*dimE+Nface*dimF];
return
```

Cutting a mesh one or more planes. The idea behgind the function `planeCut` is that given an enhanced tetrahedrization, `T`, and one or more planes in the form $Ax+By+Cz+D=0$, one would want to create a subtetrahedrization of elements in `T` whose barycenters lie strictly above the chosen planes. The planes are passed into the function as a $N_{\text{plane}} \times 4$ matrix where N_{plane} is the number of planes and each of the four columns correspond to the coefficents A, B, C, and D respectively.
To mark the elements which lie above the chosen planes, the barycenters of each element are computed and stored in a $3 \times N_{\text{elt}}$ matrix. This matrix is then augmented with a fourth row comprised entirely of ones, so that matrix multiplication between the plane matrix and the barycenter matrix results in the computation $Ax + By + Cz + D$ for each barycenter (x, y, z) and each plane. The resulting matrix is turned into a $(0, 1)$-matrix with 1s marking where the matrix had positive entries. The entries of this matrix are multiplied column-wise and resulting non-zero entries are saved as "marked" elements.
Note that this function does not create the resulting subtetrahedrization, but only the list of elements needed for the function `subtetrahedrization`.

```
function Elts = planeCut( T, planes )

% Elts = planeCut( T, planes )
%
% Input:
%       T       : an enhanced tetrahedrization
%       planes  : a number of planes X 4 where each row contains the
%                 coefficients of a plane
%
% Output:
%       Elts    : the 'marked' elements whose barycenters are above all of
%                 the planes
%
% Last Modified: September 13, 2016

bary = (T.coordinates(:, T.elements(1,:))...
          + T.coordinates(:, T.elements(2,:))...
          + T.coordinates(:, T.elements(3,:))...
          + T.coordinates(:, T.elements(4,:)))/4;

bary = [bary; ones(1, size(bary,2))];

test = planes*bary;
test = test>0;
Elts = prod(test,1);
Elts = find(Elts==1);

end
```

7.1 Working with interfaces

When a subtetrahedrization has been created, Dirichlet and Neumann faces keep their status, but some former interior faces have become boundary (interface) faces. In `T.faces` they are still tagged with the index 0 and the orientation of their inherent normal vector might be pointing in. The goal of the function `interface` is multiple:

- It locates the faces on the new boundary (which we call the interface). This is done by locating the faces with a 0 index which appear only once in `T.facebyelt`.
- It changes the index of those faces in `T.faces` to 3, so that, from now one: 0 is interior, 1 is Dirichlet, 2 is Neumann, and 3 is Interface.
- It creates a new field `T.interface` with a $3 \times N_{\text{intfac}}$ matrix containing the node indices for the interfaces. This matrix is like `T.dirichlet` or `T.neumann`.

- It locates the interface faces whose orientation is negative (pointing towards the interior domain) and changes the sign of the normal there. **Important.** Before running this function, it is important to have *enhanced* the tetrahedrization so that it contains the `T.normals` field and these are not computed later.

To locate which faces are interior, the command

```
accummarray(T.facebyelt(:),1)'
```

assembles a vector adding a unit to each face, counting by element. The positions of this vector containing the number 2 tell the interior faces, while those with 1 indicate the boundary faces. Intersecting this list with the list of faces that are interior or interface (those with a 0 index) gives the list of interface faces. The next delicate issue is locating which interface faces are positively oriented. First we look locally at all the faces of all the elements. The $4 \times N_{\text{elt}}$ matrix A contains

- a 1 for any 1st or 3rd faces with permutation number in $\{2, 4, 6\}$
- a 1 for any 2nd or 4th faces with permutation number in $\{1, 3, 5\}$
- a 0, otherwise.

This has to be read as follows: if the face i of the element K marks a 1 index, this means that the inherent normal of that face (counted globally) points outwards from the element. We now restrict our attention to interface faces and locate those with a 0 index, since for those faces, the inherent normal is pointing inwards from the point of view of the element, and therefore, from the point of view of the domain.

This needs more detail

```
function T = interface(T)

%function T = interface(T)
%
% Input:
%       T: data structure of a subtetrahedrization of a larger
%          tetrahedrization
%
% Output:
%       T: updated data structure with the following changes:
%          (a) a new field T.interface which identifies false "boundary" faces
%          (b) an option of three in the last row in T.faces of the above
%              faces
%          (c) T.normals has been updated so that all normal vectors on the
%              interface point in the same direction
%
% Last Modified: October 21, 2016

freq = accumarray(T.facebyelt(:),1)';
nonintface = find(freq==1);
nonbdface = find(T.faces(4,:)==0);
interfaces = intersect(nonintface,nonbdface);
T.interface = T.faces(1:3,interfaces);
T.faces(4,interfaces) = 3;

A=T.faceorient;
for row=[1 3]
    for perm=[2 4 6]
        A(row,A(row,:)==perm)=10; % 10 is fine, 0 is not fine
    end
    A(row,A(row,:)≠10)=0;
end
for row=[2 4]
    for perm=[1 3 5]
```

```
        A(row,A(row,:)==perm)=10; % 10 is fine, 0 is not fine
    end
    A(row,A(row,:)≠10)=0;
end
A(A==10)=1;

[¬,WhichFaceIsIn]=ismember(T.facebyelt,interfaces);
Wrong=WhichFaceIsIn.*(1—A); % marks is the face is in and has the wrong orientation
Wrong=unique(Wrong(:));
if Wrong(1)==0;
    Wrong=Wrong(2:end);
end
change=interfaces(Wrong);
T.normals(:,change) = —T.normals(:,change);
end
```

7.2 Interface matrices

The goal is the computation of the matrices

$$\int_\Sigma P_i^- P_j^+ \rho \qquad i = 1, \ldots, \dim V_h^-, j = 1, \ldots, \dim V_h^+,$$

where

$$\rho \in \{1, n_x, n_y, n_z\}.$$

These matrices are the boundary Mass and Neumann matrices which are needed to couple two tetrahedrizations sharing a boundary. In addition, this function also has the ability to compute the vectors

$$\int_\Sigma \mathrm{G}\mathbf{n} P_i^- \quad i = 1, \ldots, \dim V_h^-,$$

and

$$\int_\Sigma \mathbf{g} \cdot \mathbf{n} P_j^+ \quad j = 1, \ldots, \dim V_h^+,$$

where G and $\mathbf{g}$ are data for the transmission conditions from a coupled wave-structure interaction problem.

Computing the matrices for V_h^-. Given a mesh for the interior domain, T^-, we use the function `interface.m` to find the faces which lie on Σ, the interface connecting T^- and the exterior mesh T^+. Now we use the function `neumannBC3D.m` to compute four boundary mass matrices:

$$\int_\Sigma \phi_i \phi_j \rho \qquad i, j = 1, \ldots, \dim V_h^-, \quad \rho \in \{1, n_x, n_y, n_z\}.$$

This requires (see Section 6.2) that we pass in a dummy function (which we call `zerofunc`) the mesh, polynomial degree of our finite element space, and the function which we call ρ above. For $\rho = 1$, we just want a boundary mass matrix. In order to compute the matrices for $\rho = n_x, n_y, n_z$, we cell arrays which represent the vectorized function versions of $[1,0,0]^\top, [0,1,0]^\top$, and $[0,0,1]^\top$ respectively. Note that given a vector, `neumannBC3D` will dot the given vector with the normal vector.

Fixing the dimensions. The dimensions of the matrices which we are trying to compute will be $\dim V_h^- \times \dim V_h^+$, however the dimensions of the mass boundary matrices which we have computed are $\dim V_h^- \times \dim V_h^-$. In order to make sure that all of the elements of each matrix are placed in the correct location of the final matrix we do the following:

- create an empty intermediate matrix $B \in \mathbb{R}^{\dim V_h^- \times \dim V_h}$,

- store the mass boundary matrix computed above in B using the embedding operator for T^- (see the above documentation **Embedding Finite Element Spaces**). This ensure that the entries of the mass boundary matrix are mapped to the columns of B corresponding to the nodes in the larger parent mesh, T.
- map the entries of B column-wise to the nodes of T^+ using the embedding operator for T^+.

This is the procedure used to create all all four matrices. The matrices are stored as the first four entries in a cell array with the entry number corresponding to the ρ that was used in the computation.

Computing the vectors If the data **g** and G are used as input into the function, we use `neumannBC3D` to create the load vectors that we specified above. There is nothing special done here with two exceptions:

- given the input G, `neumannBC3D` produces a matrix, which we then turn into a vector by stacking the columns.
- because we are testing **g** with basis functions from the exterior domain we use a similar process as with the mass boundary matrices using the embedding operators to ensure that the coefficients of the vector correspond to the correct degrees of freedom. This process is necessary because we only pass T^-, mesh information related to the interior domain, into the function. We could avoid using the embedding operators if the function also took T^+ as an input.

These two vectors, if computed, are stored as the fifth (for data G) and sixth (for data **g**) entries in the returned cell array.

```
function I = interfaceMatrices3D(Tm, ep, em, dim, k, varargin)

% I = interfaceMatrix3D(Tm, ep, em, [d,dp,dm], k)
% I = interfaceMatrix3D(Tm, ep, em, [d,dp,dm], k, g , G)
%
% Inputs:
%       Tm : enhanced mesh of interior domain
%       ep : vector embedding Tp into T
%       em : vector embedding Tm into T
%       [d,dp,dm]: dimensions of Pk FEM spaces Vh, Vhp, Vhm
%       k  : polynomial degree
%       g  : 1x3 cell array with 3 vect fns of 3 variables
%       G  : 1x6 cell array with 6 vect fns of 3 variables
% Output:
%       I  : Cell Array of four interface matrices
%               \int_Gint P_i^- P_j^+ \tau \, \tau\in {1,nx,ny,nz}
%            ... or two added interface vectors
%               \int_Gint (G n) P_i^-
%               \int_Gint (g.n) P_i^+
%
% Last Modified: October 28, 2016

d = dim(1);
dp = dim(2);
dm = dim(3);

% copy on Tm.interface onto Tm.neumann
Tm = interface(Tm);
Tm.faces(4,Tm.faces(4,:)==2) = 5;
Tm.faces(4,Tm.faces(4,:)==3) = 2;
Tm.oldneumann = Tm.neumann;
Tm.neumann = Tm.interface;

onefunc = @(x,y,z) 1 + 0*x;
zerofunc = @(x,y,z) 0*x;
```

```
I{1} = sparse(dm,dp); I{2} = I{1}; I{3} = I{1}; I{4} = I{1};
[¬,A] = neumannBC3D(zerofunc,Tm,k,onefunc);
B = sparse(dm,d);
B(:,em) = A;
I{1} = B(:,ep);

[¬,A] = neumannBC3D(zerofunc,Tm,k,{onefunc,zerofunc,zerofunc});
B = sparse(dm,d);
B(:,em) = A;
I{2} = B(:,ep);

[¬,A] = neumannBC3D(zerofunc,Tm,k,{zerofunc,onefunc,zerofunc});
B = sparse(dm,d);
B(:,em) = A;
I{3} = B(:,ep);

[¬,A] = neumannBC3D(zerofunc,Tm,k,{zerofunc,zerofunc,onefunc});
B = sparse(dm,d);
B(:,em) = A;
I{4} = B(:,ep);

if nargin==7
    bG = neumannBC3D(varargin{2},Tm,k); I{5}=bG(:);
    bpg=zeros(d,1);
    bpg(em)=neumannBC3D(varargin{1},Tm,k);
    I{6} = bpg(ep);
end
```

7.3 Coupling Matrices

```
function [MAT,b] = couplingMatrices(T,Tp,Tm,ep,em,rho,lam,mu,k,s,g,G,vinc)

% [Mh,b]= couplingMatrices(T,Tp,Tm,ep,em,rho,lam,mu,k,s,g,G,vinc)
% Input:
%   Tp, Tm          : Data structure. Enhanced FEM triangulation
%   ep, em          : Embeding operators
%   mu,lam          : Vect function of 3 vars (elastic parameters)
%   rho             : Vect function of 3 vars (density)
%   g               : 1x3 cellarray w vect fns of 3 vars (jump in velocity)
%   G               : 1x6 cellarray w vect fnd of 3 vars (jump in stress)
%                         Order: xx,xy,xz,yy,yz,zz
%   k               : Polynomial degree
%   s               : Complex frequency in Laplace domain
%   vinc            : 1x4 cell array w vect fns of 3 vers (inc wave & grad)
%
% Output:
%   Mh,b            :
%
% Last modified : November 18, 2016

% Elasticity in Tm

Sm = stiffnessMatrices3D(mu,Tm,k);
Sl = stiffnessMatrices3D(lam,Tm,k);
Mel = massMatrix3D(rho,Tm,k);
Sel = [2*Sm{1,1}+Sl{1,1}+Sm{2,2}+Sm{3,3}, Sl{1,2}+Sm{1,2}.',...
                                                    Sl{1,3}+Sm{1,3}.';...
        Sm{1,2}+Sl{1,2}.', 2*Sm{2,2}+Sm{1,1}+Sl{2,2}+Sm{3,3},...
                                                    Sl{2,3}+Sm{2,3}.';...
        Sl{1,3}.'+Sm{1,3}, Sl{2,3}.'+Sm{2,3},...
                                    2*Sm{3,3}+Sm{1,1}+Sm{2,2}+Sl{3,3}];
O = sparse(size(Mel,1),size(Mel,2));
Mel = [Mel O O;...
```

```
       O Mel O;...
       O O Mel];
EL = s^2*Mel + Sel;

% Acoustics in Tp

unit = @(x,y,z) 0*x+1;
Mac = massMatrix3D(unit,Tp,k);
Sac = stiffnessMatricesCC3D(Tp,k);
AC = s^2*Mac + Sac{1,1}+Sac{2,2}+Sac{3,3};
[¬,MBd] = neumannBC3D(unit,Tp,k,unit);  % b.form impedance condition
AC = AC+s*MBd;

% Coupling terms

d = dimFEMspace(T,k);
dm = dimFEMspace(Tm,k);
dp = dimFEMspace(Tp,k);
I = interfaceMatrices3D(Tm,ep,em,[d,dp,dm],k,g,G);
C = [I{2};I{3};I{4}];

MAT = [EL   s*C;
       -s*C'  AC];

% Right-hand side

bext = s*neumannBC3D(vinc{1},Tp,k)+neumannBC3D({vinc{2:4}},Tp,k);
b = [I{5}; -I{6} + bext];


end
```

8 Some tools for meshing

8.1 A quadrilateral mesh data-structure

The basic data structure has the following components:

```
Q =
       elements: [8x60 double]
    coordinates: [3x120 double]
         bdface: {1x6 cell}
          bdtag: {1x6 cell}
      dirichlet: [4x27 double]
        neumann: [4x67 double]
```

This means that:

- There are 120 nodes: the coordinates of each node are given in a column of `Q.coordinates`
- There are 60 elements: the local nodes of each (8 nodes) are given in a column of `Q.elements` (see later for local numbering)
- There are 27 Dirichlet boundary faces, organized by columns again (see later for local numbering)
- There are 67 Neumann boundary faces, organized by columns and the same fashion as the Dirichlet faces.
- The domain has 6 faces. The partition for the i-th face is stored in a $3 \times N_i$ matrix `Q.bdface`$\{i\}$, where N_i is the number of quadrilaterals on the face. The field `Q.bdtag`$\{i\}$ contains a text description of what the i-th face is. The fields `Q.bdface` and `Q.bdtag` are optional.

All the elements have to be numbered in the same geographical way. To explain this, take a single element a look at the local numbering in Figure 8.1. In a partition with several quadrilateral we will demand that all of them are numbered locally in a coherent way. If you are going to use the quad partition directly, there's no need to have them respecting orientation, but when we subdivide them in tetrahedra we will have to be careful with this when creating the diagonals that break the element. Even if this is advancing news, let's clarify the concepts.

The twelve edges of the quad are given by the indices in the following matrix: each column marks an edge

$$\left[\begin{array}{cccc|cccc|cccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 & 6 & 7 & 8 & 5 & 5 & 6 & 7 & 8 \end{array}\right]$$

The tetrahedral partition (of which we will speak later on) will leave the vertices 1 and 7 untouched, will create two interior triangles

$$\left[\begin{array}{ccc} 2 & 4 & 5 \end{array}\right] \qquad \left[\begin{array}{ccc} 3 & 6 & 8 \end{array}\right]$$

and connect the vertices 4 and 6. As can be seen in the right part of Figure 8.1, from each of the vertices in the list $\{2, 3, 4, 5, 6, 8\}$ we will take two diagonals on faces (never three!). The interior connection of the pair $\{4, 6\}$ could be easily substituted by $\{2, 8\}$. That effect would be appreciable from the boundary, which is where the elements will have to match after subdivision.

fig1

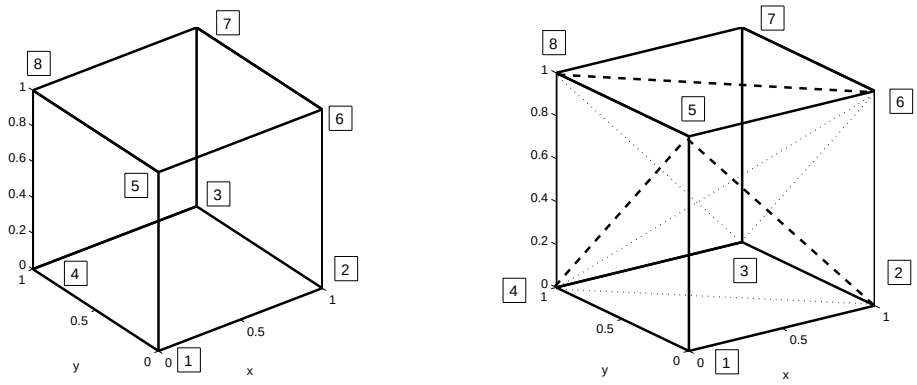

Figure 3: One quadrilateral element (quad). On the right, what the quad will look like once it has been subdivided into six tetrahedra.

With boundary faces we will adopt the following convention:

- the orientation will be positive (counterclockwise) seen from inside;
- we will always start on a vertex that will not suffer subdivision, when the face is divided into two triangles.

With the example of Figure 8.1, the six faces would be

$$\left[\begin{array}{cc|cc|cc} 1 & 5 & 2 & 1 & 1 & 4 \\ 2 & 8 & 6 & 4 & 5 & 3 \\ 3 & 7 & 7 & 8 & 6 & 7 \\ 4 & 6 & 3 & 5 & 2 & 8 \end{array}\right]$$

As long as we keep the internal ordering, the numbering of each face can be started in the third node. For instance, the top face admits the forms

$$\left[\begin{array}{cccc} 5 & 8 & 7 & 6 \end{array}\right] \quad \text{and} \quad \left[\begin{array}{cccc} 7 & 6 & 5 & 8 \end{array}\right]$$

The function `meshPartition.m` provides a uniform refinement of a quadrilateral mesh. Each element of the original mesh is subdivided into eight elements of the new mesh. This function does not respect the structure of the the quadrilateral mesh fields with respect to the fields `Q.bdface` and `Q.bdtag`. The output of `meshPartition` contains only the fields `Q.coordinates, Q.elements, Q.dirichlet`, and `Q.neumann`. This can be fixed in the future by altering the function (i.e. this is a fun 30 minute project for someone)

```
function [Qnew]=meshQuadRefinement(Q)

% [Qnew]=meshQuadRefinement(Q)
%
% Input:
%   Q      : basic quad partition
% Output:
%   Qnew     : uniformly refined quad partition
%
% Last Modified: July 1, 2016

Ncubes=size(Q.elements,2);
Nnodes=size(Q.coordinates,2);

% Edge connectivity

edges=[1 2;2 3;3 4;4 1;...
       5 6;6 7;7 8;8 5;...
       1 5; 2 6;3 7; 4 8]';
econ=reshape(Q.elements(edges(:),:),2,12*Ncubes);
econ=sort(econ);
whichedge=zeros(12,Ncubes);
[econ,i,whichedge(:)]=unique(econ','rows');  % econ = Nedges x 2
% whichedge=whichedge';     % whichedge(e,:) = edges for element #e
Nedges=size(econ,1);
econ = econ';

% Face connectivity

faces=[1 2 3 4;...
       5 8 7 6;...
       1 5 6 2;...
       4 3 7 8;...
       1 4 8 5;...
       2 6 7 3]';
fcon=reshape(Q.elements(faces(:),:),4,6*Ncubes);
ffcon=fcon';
fcon=sort(fcon);
whichface=zeros(6,Ncubes);
[fcon,i,whichface(:)]=unique(fcon','rows');  % fcon = Nfaces x 4
% whichface=whichface';     % whichface(e,:) = faces for element #e
Nfaces=size(fcon,1);
ffcon=ffcon(i,[2 4]);
fcon = fcon';


% Coordinates of new points

Qnew.coordinates=[Q.coordinates ...
   ( Q.coordinates(:,econ(1,:))+Q.coordinates(:,econ(2,:)))/2 ...           % edges
   ( Q.coordinates(:,fcon(1,:))+Q.coordinates(:,fcon(2,:))...               % faces
    +Q.coordinates(:,fcon(3,:))+Q.coordinates(:,fcon(4,:)))/4 ...
   ( Q.coordinates(:,Q.elements(1,:))+Q.coordinates(:,Q.elements(2,:))...   % elements
    +Q.coordinates(:,Q.elements(3,:))+Q.coordinates(:,Q.elements(4,:))...
    +Q.coordinates(:,Q.elements(5,:))+Q.coordinates(:,Q.elements(6,:))...
    +Q.coordinates(:,Q.elements(7,:))+Q.coordinates(:,Q.elements(8,:)))/8];

% Element subdivision
```

```
elt=[Q.elements;...
        Nnodes+whichedge;...
        Nnodes+Nedges+whichface;...
        Nnodes+Nedges+Nfaces+(1:Ncubes)];
pattern=[1 9 21 12 17 23 27 25;...
    9 2 10 21 23 18 26 27;...
    12 21 11 4 25 27 24 20;...
    21 10 3 11 27 26 19 24;...
    17 23 27 25 5 13 22 16;...
    25 27 24 20 16 22 15 8;...
    23 18 26 27 13 6 14 22;...
    27 26 19 24 22 14 7 15]';
Qnew.elements=reshape(elt(pattern,:),8,8*Ncubes);

% PARTITION OF BOUNDARY FACES

edges=[1 2 3 4;...
       2 3 4 1];
pattern=[1 5 9 8;...
    5 2 6 9;...
    8 9 7 4;...
    9 6 3 7];

% Location of edges and faces of Dirichlet elements

Ndir=size(Q.dirichlet,2);
if Ndir>0
    listDir=reshape(Q.dirichlet(edges(:),:),2,4*Ndir);
    listDir=sort(listDir',2);       % edges of Dirichlet faces (with repetitions)
    [listDir,i,j]=unique(listDir,'rows');
    [aux,ii,jj]=intersect(econ',listDir,'rows');
    whichDiredge=reshape(ii(j),4,Ndir);
    [aux,i,j]=intersect(fcon',sort(Q.dirichlet',2),'rows');
    whichDirface(j) = i;
    elt=[Q.dirichlet;...
        Nnodes+whichDiredge;...
        Nnodes+Nedges+whichDirface];
    Qnew.dirichlet=reshape(elt(pattern',:),4,4*Ndir);
else
    Qnew.dirichlet=[];
end

% Location of edges and faces of Neumann elements

Nneu=size(Q.neumann,2);
if Nneu>0
    listNeu=reshape(Q.neumann(edges(:),:),2,4*Nneu);
    listNeu=sort(listNeu',2);       % edges of Neumann faces (with repetitions)
    [listNeu,i,j]=unique(listNeu,'rows');
    [aux,ii,jj]=intersect(econ',listNeu,'rows');
    whichNeuedge=reshape(ii(j),4,Nneu);
    [aux,i,j]=intersect(fcon',sort(Q.neumann',2),'rows');
    whichNeuface(j)=i;
    elt=[Q.neumann;...
        Nnodes+whichNeuedge;...
        Nnodes+Nedges+whichNeuface];
    Qnew.neumann=reshape(elt(pattern',:),4,4*Nneu);
else
    Qnew.neumann=[];
end

if isfield(Q, 'bdface')
    disp('WARNING: at this time partition.m is not equipped to update');
    disp('the fields bdface and bdtag.  As such, they no longer exist.');
end
```

```
return
```

8.2 Tetrahedrization of a quad partition

The idea is to break each quadrilateral into six tetrahedra by making simultaneously three cuts of the quad into pairs of prisms with triangular bases. This is the data structure for a basic tetrahedral partition, corresponding to the example given for quadrilateral partitions: note that there are 6 times the number of elements and twice the number of faces.

```
T =
    coordinates: [3x120 double]
       elements: [4x360 double]
      dirichlet: [3x54 double]
        neumann: [3x134 double]
         bdface: {1x6 cell}
          bdtag: {1x6 cell}
```

The structure is very similar to a quadrilateral partition. The tetrahedra are positively oriented and so are the boundary faces.

fig4

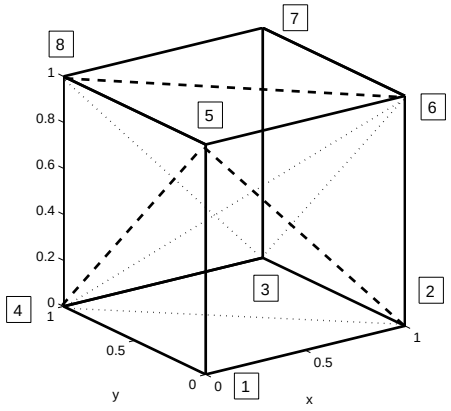

Figure 4: The tetrahedral division of a quad.

```
function T=quad2tet(Q)

% T=quad2tet(Q)
%
% Input:
%    Q : basic quad partition
% Output:
%    T : tetrahedral partition with the same nodes and boundary
% Last modified: July 15, 2015

T.coordinates=Q.coordinates;

% partition of each quad into 6 tetrahedra

tetra=[1 2 4 5;...
    6 4 5 8;...
    6 5 4 2;...
    2 3 4 6;...
    8 7 6 3;...
```

```
    3 8 4 6]';
Ncubes=size(Q.elements,2);
T.elements=Q.elements(tetra(:),:);
T.elements=reshape(T.elements,[4,6*Ncubes]);

% partition of each face into 2 triangles

tri=[1 4 2;2 4 3]';

% Dirichlet faces

NDir=size(Q.dirichlet,2);
if isempty(Q.dirichlet)
    T.dirichlet=[];
else
    T.dirichlet=Q.dirichlet(tri(:),:);
    T.dirichlet=reshape(T.dirichlet,[3,2*NDir]);
end
% Neumann faces

NNeu=size(Q.neumann,2);
if isempty(Q.neumann)
    T.neumann=[];
else
    T.neumann=Q.neumann(tri(:),:);
    T.neumann=reshape(T.neumann,[3,2*NNeu]);
end

% Separated face

if isfield(Q,'bdface')
    for i=1:length(Q.bdface)
        Ni=size(Q.bdface{i},2);
        T.bdface{i}=Q.bdface{i}(tri(:),:);
        T.bdface{i}=reshape(T.bdface{i},[3,2*Ni]);
    end
    T.bdtag=Q.bdtag;
end

return
```

8.3 Creating a partition of the unit cube

The goal of the next function is the construction of a tetrahedrization of the unit cube

$$[0,1] \times [0,1] \times [0,1].$$

The default is that the top and bottom faces ($z = 0$ and $z = 1$) are Dirichlet and the other ones are Neumann. Otherwise, the collection of Dirichlet faces can be given as an additional parameter. The faces are numbered in the following way:

1. Bottom ($z = 0$)
2. Top ($z = 1$)
3. Left ($y = 0$)
4. Right ($y = 1$)
5. Back ($x = 0$)
6. Front ($x = 1$)

The names for the six faces correspond to a standard view of the unit cube.
We first subdivide the cube into quadrilateral elements of the same size, with N_x, N_y and N_z partitions in the respective x, y, z directions. Once this process is finished, the function `quad2tet.m` is called to make the tetrahedral subdivision. The quadrilateral partition is also exported as a second output argument.

```
function [T,Q]=meshCube(Nx,Ny,Nz,varargin)

% [T,Q]=meshCube(Nx,Ny,Nz)
% [T,Q]=meshCube(Nx,Ny,Nz,listDir)
% input
%    Nx  : number of partitions in the x direction
%    Ny  : number of partitions in the y direction
%    Nz  : number of partitions in the z direction
%    listDir : list of Dirichlet faces
% output
%    T   : tetrahedral partition data structure
%          default: the top and bottom faces are Dirichlet
%    Q   : quadrilateral mesh of the same data structure
%          default: the top and bottom faces are Diriclet
% last modified: March 4, 2016

Nx=Nx+1;
Ny=Ny+1;
Nz=Nz+1;

% creating the cubic partition

list=reshape(1:Nx*Ny*Nz,Nx,Ny,Nz);
c=list(1:Nx-1,1:Ny-1,1:Nz-1);
q=[c(:) c(:)+1 c(:)+Nx+1 c(:)+Nx];
Q.elements=[q q+Nx*Ny]';

% coordinates of the nodes
% you can change the cube to another paralellepiped here

x=linspace(0,1,Nx);
y=linspace(0,1,Ny);
z=linspace(0,1,Nz);
[y,x,z]=meshgrid(y,x,z);
Q.coordinates=[x(:) y(:) z(:)]';

reverse=[1 4 3 2];

% faces 1 (bottom z=0) & 2 (top z=1)
c=list(1:Nx-1,1:Ny-1,1);
face{1}=[c(:) c(:)+1 c(:)+Nx+1 c(:)+Nx];
face{2}=(Nz-1)*Nx*Ny+face{1};
face{2}=face{2}(:,reverse);

% faces 3 (left y=0) & 4 (right y=1)
c=list(1:Nx-1,1,1:Nz-1);
face{3}=[c(:) c(:)+Nx*Ny c(:)+Nx*Ny+1 c(:)+1];
face{4}=(Ny-1)*Nx+face{3};
face{4}=face{4}(:,reverse);

% faces 5 (back x=0) & 6 (front x=1)
c=list(1,1:Ny-1,1:Nz-1);
face{5}=[c(:) c(:)+Nx c(:)+Nx*Ny+Nx c(:)+Nx*Ny];
face{6}=(Nx-1)+face{5};
face{6}=face{6}(:,reverse);

for i=1:6
    Q.bdface{i}=face{i}';
end
```

```
Q.bdtag{1}='bottom (z=0)';
Q.bdtag{2}='top (z=1)';
Q.bdtag{3}='left (y=0)';
Q.bdtag{4}='right (y=1)';
Q.bdtag{5}='back (x=0)';
Q.bdtag{6}='front (x=1)';

% Dirichlet/Neumann division

if nargin==3
    listDir=[1 2];
    listNeu=[3 4 5 6];
else
    listDir=varargin{1};
    listNeu=1:6;
    listNeu(listDir)=[];
end
Q.dirichlet=[];
Q.neumann=[];
for i=listDir
    Q.dirichlet=[Q.dirichlet face{i}'];
end
for i=listNeu
    Q.neumann=[Q.neumann face{i}'];
end
T = quad2tet(Q);
return
```

8.4 Creating a mesh of a rectangular prism

The function `meshRectPrism` is a modification of `meshCube` and it creates a partition of the solid

$$[0, d_x] \times [0, d_y] \times [0, d_z].$$

The function refines this solid m times by partitioning it into $6 \times m\,d_x \times m\,d_y \times m\,d_z$ many identical tetrahedra, so that each edge is equi-partitioned into $m\,d_\alpha$ many edges for $\alpha \in \{x, y, z\}$. Face tags are same with `meshCube`.

```
function [T,Q]=meshRectPrism(dx,dy,dz,lev,varargin)

% [T,Q]=meshRectPrism(dx,dy,dz,lev)
% [T,Q]=meshRectPrism(dx,dy,dz,lev,listDir)
% Input
%    dx  : dimension of the prism in the x direction
%    dy  : dimension of the prism in the y direction
%    dz  : dimension of the prism in the z direction
%    lev : nof refinements per unit dimension
%    listDir : list of Dirichlet faces: bottom(1), top(2), left(3),
%              right(4), back(5), front(6)
% Output
%    T   : tetrahedral partition data structure
%          default: the top and bottom faces are Dirichlet
%    Q   : quadrilateral mesh of the same data structure
%          default: the top and bottom faces are Diriclet
% Last modified: January 10, 2019

Nx=lev*dx;Ny=lev*dy;Nz=lev*dz;
Nx=Nx+1;
Ny=Ny+1;
Nz=Nz+1;

% creating the cubic partition
```

```
list=reshape(1:Nx*Ny*Nz,Nx,Ny,Nz);
c=list(1:Nx-1,1:Ny-1,1:Nz-1);
q=[c(:) c(:)+1 c(:)+Nx+1 c(:)+Nx];
Q.elements=[q q+Nx*Ny]';

% coordinates of the nodes
% you can change the cube to another paralellepiped here

x=linspace(0,dx,Nx);
y=linspace(0,dy,Ny);
z=linspace(0,dz,Nz);
[y,x,z]=meshgrid(y,x,z);
Q.coordinates=[x(:) y(:) z(:)]';

reverse=[1 4 3 2];

% faces 1 (bottom z=0) & 2 (top z=1)
c=list(1:Nx-1,1:Ny-1,1);
face{1}=[c(:) c(:)+1 c(:)+Nx+1 c(:)+Nx];
face{2}=(Nz-1)*Nx*Ny+face{1};
face{2}=face{2}(:,reverse);

% faces 3 (left y=0) & 4 (right y=1)
c=list(1:Nx-1,1,1:Nz-1);
face{3}=[c(:) c(:)+Nx*Ny c(:)+Nx*Ny+1 c(:)+1];
face{4}=(Ny-1)*Nx+face{3};
face{4}=face{4}(:,reverse);

% faces 5 (back x=0) & 6 (front x=1)
c=list(1,1:Ny-1,1:Nz-1);
face{5}=[c(:) c(:)+Nx c(:)+Nx*Ny+Nx c(:)+Nx*Ny];
face{6}=(Nx-1)+face{5};
face{6}=face{6}(:,reverse);

for i=1:6
    Q.bdface{i}=face{i}';
end
Q.bdtag{1}='bottom (z=0)';
Q.bdtag{2}=['top (z=' num2str(Nz-1) ')'];
Q.bdtag{3}='left (y=0)';
Q.bdtag{4}=['right (y=' num2str(Ny-1) ')'];
Q.bdtag{5}='back (x=0)';
Q.bdtag{6}=['front (x=' num2str(Nx-1) ')'];

% Dirichlet/Neumann division

if nargin==4
    listDir=[1 2];
    listNeu=[3 4 5 6];
else
    listDir=varargin{1};
    listNeu=1:6;
    listNeu(listDir)=[];
end
Q.dirichlet=[];
Q.neumann=[];
for i=listDir
    Q.dirichlet=[Q.dirichlet face{i}'];
end
for i=listNeu
    Q.neumann=[Q.neumann face{i}'];
end
T = quad2tet(Q);
return
```

8.5 Creating a mesh of a prism

The function `meshPrism` creates a partition of the prism

$$\{(x,y) : x,y \geq 0, \quad x+y \leq 1\} \times [0,1].$$

This partition is created by deforming the partition of a cube, instead of cutting half of it. (The aim of this process is the construction of a partition that can be deformed to a quarter cylinder.) The default is that the top and bottom faces are Dirichlet. We number faces as follows:

1. Bottom ($z=0$)
2. Top ($z=1$)
3. Left ($x=0$)
4. Front ($x+y=1$)
5. Back ($y=0$)

Here are some basic ideas of the process. First of all, we build a quadrilateral partition of the unit cube. Then, we deform the partition with the following transformation of the (x,y) variables:

$$x_{\text{new}} = x - \tfrac{1}{2}\min\{x,y\} = \begin{cases} x - \frac{1}{2}y & y < x, \\ \frac{1}{2}x & x < y, \end{cases}$$

$$y_{\text{new}} = y - \tfrac{1}{2}\min\{x,y\} = \begin{cases} \frac{1}{2}y & y < x, \\ y - \frac{1}{2}x & x < y. \end{cases}$$

Some of the original quadrilaterals are deformed into prisms. This forces a slightly different tetrahedral partition and `quad2tet` cannot be used. In particular, if the quads are numbered as usual, the internal tetrahedral partition is built with the table

$$\begin{bmatrix} 1 & 2 & 3 & 5 \\ 6 & 3 & 5 & 7 \\ 6 & 5 & 3 & 2 \\ 1 & 3 & 4 & 5 \\ 7 & 8 & 4 & 5 \\ 7 & 4 & 3 & 5 \end{bmatrix}$$

```
function T=meshPrism(Nx,Ny,Nz,varargin)

%T=meshPrism(Nx,Ny,Nz)
%T=meshPrism(Nx,Ny,Nz,listDir)
%
% input
%    Nx  : number of partitions in the x direction
%    Ny  : number of partitions in the y direction
%    Nz  : number of partitions in the z direction
%    listDir : row vector with Dir BC {1:bottom, 2:top, 3:left(x=0), 4
%              4:front(x+y≤1), 5:back(y=0)}.
%              default listDir=[1 2];
% output
%    T   : tetrahedrization of a prism
%
% Last Modified: March 18, 2016

[¬,Q] = meshCube(Nx,Ny,Nz);
```

```
L = min(Q.coordinates(1,:)/2,Q.coordinates(2,:)/2);
Q.coordinates(1,:)=Q.coordinates(1,:)-L;
Q.coordinates(2,:)=Q.coordinates(2,:)-L;

split = size(Q.bdface{4},2);
Q.bdface{4}=[Q.bdface{4} Q.bdface{6}];
Q.bdface(6)=[];

Q.bdtag{1}='bottom (z=0)';
Q.bdtag{2}='top (z=1)';
Q.bdtag{3}='left (x=0)';
Q.bdtag{4}='front (x+y≤1)';
Q.bdtag{5}='back (y=0)';
Q.bdtag(6)=[];

% Dirichlet/Neumann division

if nargin==3
    listDir=[1 2];
    listNeu=[3 4 5];
else
    listDir=varargin{1};
    listNeu=1:5;
    listNeu(listDir)=[];
end

T.coordinates=Q.coordinates;

% partition of each prism into 6 tetrahedra

tetra=[1 2 3 5;...
    6 3 5 7;...
    6 5 3 2;...
    1 3 4 5;...
    7 8 4 5;...
    7 4 3 5]';
Ncubes=size(Q.elements,2);
T.elements=Q.elements(tetra(:),:);
T.elements=reshape(T.elements,[4,6*Ncubes]);

% partition of each face into 2 triangles

tri1=[1 4 2;2 4 3]';
tri2=[1 4 3;2 1 3]';

% Separated faces
for i=1:2
    Ni=size(Q.bdface{i},2);
    T.bdface{i}=Q.bdface{i}(tri2(:),:);
    T.bdface{i}=reshape(T.bdface{i},[3,2*Ni]);
end

Ni=size([Q.bdface{4}],2);
T.bdface{4}=[Q.bdface{4}(tri2(:),1:split) Q.bdface{4}(tri1(:),split+1:end)];
T.bdface{4}=reshape(T.bdface{4},[3,2*Ni]);

for i=[3 5]
    Ni=size(Q.bdface{i},2);
    T.bdface{i}=Q.bdface{i}(tri1(:),:);
    T.bdface{i}=reshape(T.bdface{i},[3,2*Ni]);
end

% Dirichlet & Neumann faces
T.dirichlet=[];
T.neumann=[];
for i=listDir
```

```
    T.dirichlet=[T.dirichlet T.bdface{i}];
end
for i=listNeu
    T.neumann=[T.neumann T.bdface{i}];
end

for i=1:5
    T.bdtag{i}=Q.bdtag{i};
end
return
```

8.6 Creating a partition of the reference tetrahedron

The function `meshTetrahedron` creates a partition of the reference tetrahedron

$$\widehat{K} = \{(x, y, z) : x, y, z, 1 - (x + y + z) \geq 0\}$$

into N^3 smaller tetrahedra. This is done by subdividing the unit cube into $6N^3$ tetrahedra and then picking the ones that fit in the unit tetrahedron. The algorithm is not very refined. The main difficulty is in adding the element faces on $x + y + z = 1$, since they were not part of the original tetrahedrization of the cube. The tags for the faces are:

1. Left $(y = 0)$
2. Bottom $(z = 0)$
3. Back $(x = 0)$
4. Lid $(x + y + z = 1)$

The default is all faces being Dirichlet faces.

```
function T=meshTetrahedron(N,varargin)

% T=reftetMesh(N)
% T=reftetMesh(N,listDir)
% input
%    N  : number of partitions in the x, y and z directions
%    listDir : list of Dirichlet faces
% output
%    T   : tetrahedral partition data structure
%          default: all Dirichlet
%
% last modified: March 4, 2016

TQ=meshCube(N,N,N);
ep=0.5/N;
stay=find(sum(TQ.coordinates)≤1+ep);
nvert=size(TQ.coordinates,2);
nvertTet=length(stay);
corresp=zeros(nvert,1);
corresp(stay)=1:nvertTet;
todelete=@(A) find(prod(A)==0);

% Coordinates and elements

T.coordinates=TQ.coordinates(:,stay);
T.elements   =corresp(TQ.elements);
T.elements(:,todelete(T.elements))=[];

% Coordinate faces
```

```
T.bdface{1}=corresp(TQ.bdface{3});
T.bdface{2}=corresp(TQ.bdface{1});
T.bdface{3}=corresp(TQ.bdface{5});
T.bdface{1}(:,todelete(T.bdface{1}))=[];
T.bdface{2}(:,todelete(T.bdface{2}))=[];
T.bdface{3}(:,todelete(T.bdface{3}))=[];

T.bdtag{1}='left (y=0)';
T.bdtag{2}='bottom (z=0)';
T.bdtag{3}='back (x=0)';

% Lid

ep=0.05/N;
notonlid=find(abs(sum(T.coordinates)-1)>ep);
corresp=(1:nvertTet)';
corresp(notonlid)=0;
group1=corresp(T.elements([1 2 3],:));
group2=corresp(T.elements([2 3 4],:));
group3=corresp(T.elements([3 4 1],:));
group4=corresp(T.elements([4 1 2],:));
group1(:,todelete(group1))=[];
group2(:,todelete(group2))=[];
group3(:,todelete(group3))=[];
group4(:,todelete(group4))=[];
T.bdface{4}=[group1 group2 group3 group4];
T.bdtag{4}='lid (x+y+z=1)';

% Dirichlet/Neumann faces

if nargin==1
    listDir=[1 2 3 4];
    listNeu=[];
else
    listDir=varargin{1};
    listNeu=1:4;
    listNeu(listDir)=[];
end
T.dirichlet=[];
T.neumann=[];
for i=listDir
    T.dirichlet=[T.dirichlet T.bdface{i}];
end
for i=listNeu
    T.neumann=[T.neumann T.bdface{i}];
end

end
```

8.7 Creating a mesh of a axisymmetric solid

The function `meshAxisymmetric` creates a mesh of a solid whose lateral face is determined by the input $r(z)$ which is a function of the height variable. If $r(z)$ is a constant, the resulting solid is a cylinder.

This function first creates a mesh on a triangular prism, then creates a rectangular prism by copying and rotating the triangular prism without changing the coordinates on the shared middle face. The lateral faces are then deformed in to single face defined by $r(z)$ as mentioned above.

A list of Dirichlet faces can be passed in. If no list is passed in, the default is that the top and bottom faces are Dirichlet.

The fields `T.bdface` and `T.bdtag` contain information about the faces. These are numbered as follows:

- `T.bdtag{1}` is "bottom"

- `T.bdtag{2}` is "top"
- `T.bdtag{3}` is "all around", which refers to the lateral surfaces.

```
function T = meshAxisymmetric(Nx,Nz,H,r,varargin)

% T=Cylinder(Nx,Nz,H,r)
% T=Cylinder(Nx,Nz,H,r,Dir)
% Input:
%      Nx   :  8 Nx is the number of radial divisions
%      Nz   :  divisions in the z direction
%      r    :  function handle giving shape to the lateral face
%      H    :  height of cylinder in the z-direction
%      Dir  :  row vector with Dir BC (1:bottom, 2:top, 3:sides)
%              default Dir=[1 2];
% Output:
%      T    :  tetrahedrization of a cylinder
% Last modified: March 11, 2016

% Original prism

T=meshPrism(Nx,Nx,Nz,1:5);
T.coordinates(3,:)=H.*T.coordinates(3,:);
Nvert=size(T.coordinates,2);

% Symmetrized prism

TT=T;
TT.coordinates(1:2,:)=1-TT.coordinates([2 1],:);
TT.elements=TT.elements([1 2 4 3],:);
for i=1:5
    TT.bdface{i}=TT.bdface{i}([1 3 2],:);
end

% Counter for new coordinate numbers

onInterface=unique(TT.bdface{4}(:));
notOnInterface=1:Nvert; notOnInterface(onInterface)=[];
new=1:Nvert;
new(notOnInterface)=Nvert+(1:size(notOnInterface,2));

% Merger

T.coordinates(:,new)=TT.coordinates;
T.elements=[T.elements new(TT.elements)];
T.bdface{1}=[T.bdface{1} new(TT.bdface{1})];
T.bdtag{1} ='bottom';
T.bdface{2}=[T.bdface{2} new(TT.bdface{2})];
T.bdtag{2} ='top';
T.bdface{3}=[T.bdface{3} T.bdface{5} new(TT.bdface{3}) new(TT.bdface{5})];
T.bdtag{3} ='all around';
T.bdface([4 5])=[];
T.bdtag([4 5])=[];

% Cylinder

T.coordinates(1:2,:)=2*T.coordinates(1:2,:)-1;
transf = @(X,Y,Z) r(Z).*max(abs(X),abs(Y))./sqrt(X.^2+Y.^2);
rad    = transf(T.coordinates(1,:),T.coordinates(2,:),T.coordinates(3,:));
rad(isnan(rad))=0;
T.coordinates(1,:)=T.coordinates(1,:).*rad+1;
T.coordinates(2,:)=T.coordinates(2,:).*rad+1;

% Dirichlet and Neumann boundaries
```

```
if nargin==4
    listDir=[1 2];
    listNeu=3;
else
    listDir=varargin{1};
    listNeu=1:3;
    listNeu(listDir)=[];
end
T.dirichlet=[];
T.neumann=[];
for i=listDir
    T.dirichlet=[T.dirichlet T.bdface{i}];
end
for i=listNeu
    T.neumann=[T.neumann T.bdface{i}];
end

return
```

8.8 Creating a mesh of a donut

The function `meshDonut` creates tetrahedron and quadrahedron mesh for a donut shape geometry:

$$\{(x,y) \,:\, 0.5 \le \max\{|x|,|y|\} \le 1.5\} \times [-0.5, 0.5].$$

The donut geometry can be regarded as adhering 8 cubes in a circle. When the output is tetrahedron, each cube has been further divided into 6 tetrahedrons.

The face of the donut is divided into 4 parts:

- Top ($z = 0.5,\ 0.5 \le \max\{|x|,|y|\} \le 1.5$)
- Bottom ($z = -0.5,\ 0.5 \le \max\{|x|,|y|\} \le 1.5$)
- Inside ($-0.5 \le z \le 0.5,\ \max\{|x|,|y|\} = 0.5$)
- Outside ($-0.5 \le z \le 0.5,\ \max\{|x|,|y|\} = 1.5$)

A list of Dirichlet faces can be passed in. If no list is passed in, all faces are by default Dirichlet faces.

An integer for refinement level (lev) need to be passed in. The number of refinment is equal to lev-1. For each time of refinment, `meshQuadRefinement` is called for the quadrahedron data structure, `meshTetRefinment` is called for tetrahedron data structure.

```
function [T,Q] = meshDonut(lev,varargin)

% [T,Q] = meshDonut(lev)
% [T,Q] = meshDonut(lev,Dirichlet)
%
% Input:
%     lev       : refinement level, 1 for no refinement
%     Dirichlet : vector with indices for Dirichlet faces
%                   1 (top), 2 (bottom), 3 (inside), 4 (outside)
% Output:
%     T   : tetrahedron partition of a polyhedral donut
%           default is all faces Dirichlet
%     Q   : quad partition of a polyhedral donut
%           default is all faces Dirichlet
% Last modified: June 28, 2016

% initialize parameter
```

```
N= 4; index= 1:N^2; Δ= 1;

% generate coordinates
xinitial= -3/2;
yinitial= -3/2;
zinitial= -1/2;
x= (mod(index-1,N))*Δ + xinitial;
y= (floor((index-1)/N))*Δ + yinitial;
z= ones(1,N^2) * zinitial;
A1= [x' y' z'];
A2= A1; A2(:,3)= Δ + A2(:,3);
Q.coordinates= [A1', A2'];

% generate elements
cubes_first_row= [1 2 3 7 11 10 9 5];
cube_matrix_upper_half= [cubes_first_row; cubes_first_row+1; ...
    cubes_first_row+5; cubes_first_row+4];
cube_matrix_lower_half= cube_matrix_upper_half + 16;
Q.elements= [cube_matrix_upper_half;cube_matrix_lower_half];

% generate boundary face
Q.bdface{1}= cube_matrix_lower_half;
Q.bdtag{1} = 'top';

Q.bdface{2}= cube_matrix_upper_half;
Q.bdface{2}(:,:) = Q.bdface{2}([1 4 3 2],:);
Q.bdtag{2} = 'bottom';

externalface_first_row= [1 2 3 4 8 12 16 15 14 13 9 5];
externalface_second_row= [externalface_first_row(2:12) 1];
externalface_matrix= [externalface_first_row; externalface_second_row;...
    externalface_second_row+16; externalface_first_row+16];

Q.bdface{4}=externalface_matrix;
Q.bdface{4}(:,7:12) = Q.bdface{4}([4 1 2 3],7:12);
Q.bdtag{4} = 'outside';

internalface_first_row= [6 7 11 10];
internalface_last_row= [internalface_first_row(2:4) 6];
internalface_matrix= [internalface_first_row; internalface_first_row+16;...
    internalface_last_row+16;internalface_last_row];
Q.bdface{3}= internalface_matrix;
Q.bdface{3}(:,3:4) = Q.bdface{3}([4 1 2 3],3:4);
Q.bdtag{3} = 'inside';

% turn outward to inward
Q.bdface{1}(:,:)= Q.bdface{1}([1 4 3 2],:);
Q.bdface{2}(:,:)= Q.bdface{2}([1 4 3 2],:);
Q.bdface{3}(:,:)= Q.bdface{3}([1 4 3 2],:);
Q.bdface{4}(:,:)= Q.bdface{4}([1 4 3 2],:);

if nargin==2
    listDir=varargin{1};
    listNeu=1:4;
    listNeu(listDir)=[];
else
    listDir=1:4;
    listNeu=[];
end
Q.dirichlet=[];
Q.neumann=[];
for k=listDir
    Q.dirichlet=[Q.dirichlet Q.bdface{k}];
end
for k=listNeu
    Q.neumann=[Q.neumann Q.bdface{k}];
```

```
end

T = quad2tet(Q);
for i=1:lev-1
    T = meshTetRefinement(T);
    Q = meshQuadRefinement(Q);
end

end
```

9 Tools to handle meshes

9.1 Burying a mesh

Given a tetrahedral mesh T, and a parameter L, the function `meshBury.m` looks at all of the faces on the boundary and recategorizes them based on the location of the barycenter of the face with respect to L. Going through each global boundary surface, the barycenters of the boundary faces are computed and the z-coordinate is compared to L. The face is then recategorized as either 'above' or 'below' L. The number of cells in the fields `T.bdface` and `T.bdtag` is changed to reflect this new catagorization.

The purpose of this division is to change the type of boundary conditions that are implemented on the mesh. All faces that get recatagorized as 'above' the parameter become (or remain) Neumann faces and are placed in the field `T.neumann`. Similarly all faces catagorized as 'below' are placed in the field `T.dirichlet`.

```
function T = meshBury(T,L)

% T=meshBury(T,L)
%
% Input:
%    T   : tetrahedral mesh
%    L   : level
% Output:
%    T   : tetrahedral mesh - boundary face fields have been subdivided
%          based on the location of the z-coordinates of the barycenters
%          with respect to L
% Last modified: July 8, 2016

Facenum = size(T.bdface,2);

for i=1:Facenum
    FaceCoords = T.coordinates(:,T.bdface{i});
    n = size(FaceCoords,2);

    F1 = FaceCoords(:,1:3:n);
    F2 = FaceCoords(:,2:3:n);
    F3 = FaceCoords(:,3:3:n);

    BaryC = (F1+F2+F3)/3;

    M = find(BaryC(3,:)<=L);

    T.bdface{i+Facenum} = T.bdface{i}(:,M);
    T.bdface{i}(:,M)=[];
    T.bdtag{i+Facenum}=[ 'below ' T.bdtag{i}];
    T.bdtag{i}=['above ' T.bdtag{i}];
end

T.dirichlet=[];
T.neumann=[];
for i=Facenum+1:2*Facenum
```

```
    T.dirichlet=[T.dirichlet T.bdface{i}];
end
for i=1:Facenum
    T.neumann=[T.neumann T.bdface{i}];
end

empty=[];
for i=1:2*Facenum
    if size(T.bdface{i},1) == 0 || size(T.bdface{i},2) == 0
        if  (i+Facenum)≤size(T.bdface,2) && size(T.bdface{i+Facenum},1)≥1
            T.bdface([i,i+Facenum]) = T.bdface([i+Facenum,i]);
            T.bdtag([i,i+Facenum]) = T.bdtag([i+Facenum,i]);
            empty = [empty i+Facenum];
        else
            empty = [empty i];
        end
    end
end
T.bdface(empty)=[];
T.bdtag(empty)=[];

return
```

9.2 Refining a Tetrahedral Mesh

Given a tetrahedral mesh T, on the cube $[0,1] \times [0,1] \times [0,1]$ (we can change this!), the function `meshTetRefinement` creates a new tetrahedral mesh where each element of the original has been broken into eight new elements. This function assumes that each boundary face of the original mesh has been categorized as either a Dirichlet face or a Neumann face. We will go through how each field is updated.

To update the coordinates, we first use `edgesAndFaces` to create an extended data structure. With this information, we find the midpoint of each edge and make it a vertex. In this way, the new coordinates field consists of the old coordinates field and the midpoints of all of the edges.

The first step in updating the elements is to identify which new nodes belong to each old element. This is accomplished by adding the number of old vertices to the field `T.edgebyelt`, as these are the new vertices. We concatenate the the vertices in each element as well as the new numbering for the midpoints of each edge. We then use a pattern of combining these vertices that was found in the paper, "Quality Local Refinement of Tetrahedral Meshes Based on 8-Subtetrahedron Subdivision" by Liu and Joe (perhaps we need a real citation?). This pattern orders the vertices and creates 8 elements where previously there was one.

The same process is used to update both the Dirichlet and Neumann faces. First, we create a matrix of which edges (which are also the new vertices) are contained in each of the original faces. The idea is that we want to use the old face vertices and the new vertices (edge midpoints) to break the old face into four new faces. The idea is simple, but unfortunately things get a bit more complicated. This numbering is based on the orientation of the face in the element. In order to refine the faces correctly, we must first map the face as given in the field `T.Dirichlet` or `T.Neumann` to its local orientation in the element. To do this, we need the rotation matrix defined in Section 1.2 and the orientation of the faces in each element found in `T.faceorient`. Once we have located which faces are Dirichlet or Neumann (depending on which we are updating), we permute the face based on its orientation. We then refine the faces, and permute the faces back to their original orientation. This second permutation is necessary in order to preserve the outward orientation of each boundary face.

Updating the field `T.bdtag` is trivial. We update each part of the array `T.bdface` by looking at all boundary faces and using the fact that the sum of a particular coordinate (either x, y, or z) is always either 0 or 3 depending on the face we are looking at. Again, we are assuming that we have a mesh on the unit cube, otherwise, this would not be true.

If we try to refine a 3-D tetrahedral mesh which is not filling the unit cube, this function will successfully update the first four fields and then break when it comes to updating the boundary face field. This can be changed by adding options, or by commenting out this section of the code.

```
function [ S ] = meshTetRefinement( T )

% function [ S ] = meshTetRefinement( T )
%
% Input :
%        T : a basic Tetrahedral mesh
%
% Output :
%        S : a new mesh where each element has been refined into 8 new
%            tetrahedra
%
% Last Modified: May 20, 2016

Nvert = size(T.coordinates,2);
Nelt = size(T.elements,2);
NDir = size(T.dirichlet,2);
NNeu = size(T.neumann,2);
T = edgesAndFaces(T);

S.coordinates = [T.coordinates   (T.coordinates(:,T.edges(1,:))...
                                  + T.coordinates(:, T.edges(2,:))) /2];
newVertoldElt = abs(T.edgebyelt) + Nvert;
allVert = [T.elements; newVertoldElt];

% following pattern taken from Liu and Joe '96, Quality Local Refinement of
% Tetrahedral Meshes Based on 8-subtetrahedron subdivision
newEltPattern = [1 5 7 10 5 2 6 8 7 6 3 9 10 8 9 4 ...% these are good
                 5 7 10 8 5 8 6 7 9 7 6 8 9 10 7 8]'; % these can be changed
                                                      % based on
                                                      % deformation of the
                                                      % tets (maybe even
                                                      % element by element)
S.elements = reshape(allVert(newEltPattern,:),4, 8*Nelt);

faceByEdges = abs(T.edgebyelt([1 2 3 1 4 6 3 5 6 4 2 5],:));
faceByEdges = reshape(faceByEdges,3,4*Nelt) + Nvert;
x = T.facebyelt(:);
orients = T.faceorient(:);

rot = [1 1 3 3 2 2;...
       2 3 1 2 3 1;...
       3 2 2 1 1 3];
if NDir
    dir = find(T.faces(4,:) ==1);
    dirPos = find(ismember(x,dir) ==1);
    dirFaceCopy = T.faces;
    for i = 1:length(dirPos)
        dirFaceCopy(rot(:, orients(dirPos(i))), x(dirPos(i)))...
            = dirFaceCopy(1:3, x(dirPos(i)));
    end

    dirInElts = dirFaceCopy(1:3, x(dirPos));
    newDir = [dirInElts(1,:); faceByEdges(1,dirPos); ...
              dirInElts(2,:); faceByEdges(2,dirPos); ...
              dirInElts(3,:); faceByEdges(3,dirPos)];
    newDir = newDir([1 2 6 2 3 4 6 4 5 2 4 6],:);
    newDir = reshape(newDir,3,4*length(dirPos));

    for i = 1:length(dirPos)
        S.dirichlet(:,4*(i-1) + 1:4*i) = ...
            newDir(rot(:, orients(dirPos(i))), 4*(i-1) + 1:4*i);
```

```
    end
else
    S.dirichlet = [];
end

if NNeu
    neu = find(T.faces(4,:) == 2);
    neuPos = find(ismember(x,neu) ==1);
    neuFaceCopy = T.faces;
    for i = 1:length(neuPos)
        neuFaceCopy(rot(:, orients(neuPos(i))), x(neuPos(i)))...
            = neuFaceCopy(1:3, x(neuPos(i)));
    end

    neuInElts = neuFaceCopy(1:3, x(neuPos));
    newNeu = [neuInElts(1,:); faceByEdges(1,neuPos); ...
              neuInElts(2,:); faceByEdges(2,neuPos); ...
              neuInElts(3,:); faceByEdges(3,neuPos)];
    newNeu = newNeu([1 2 6 2 3 4 6 4 5 2 4 6],:);
    newNeu = reshape(newNeu,3,4*length(neuPos));

    for i = 1:length(neuPos)
        S.neumann(:,4*(i-1) + 1:4*i) = ...
            newNeu(rot(:, orients(neuPos(i))), 4*(i-1) + 1:4*i);
    end
else
    S.neumann = [];
end

S.bdtag = T.bdtag;

% to be changed...
% AllBDFaces = [S.dirichlet, S.neumann];
% AllFaceCoords = S.coordinates(:,AllBDFaces);
% AllFaceCoords = reshape(AllFaceCoords, 3,3, size(AllBDFaces,2));
% S.bdface{1} = AllBDFaces(:,sum(AllFaceCoords(3,:,:))==0);
% S.bdface{2} = AllBDFaces(:,sum(AllFaceCoords(3,:,:))==3);
% S.bdface{3} = AllBDFaces(:,sum(AllFaceCoords(2,:,:))==0);
% S.bdface{4} = AllBDFaces(:,sum(AllFaceCoords(2,:,:))==3);
% S.bdface{5} = AllBDFaces(:,sum(AllFaceCoords(1,:,:))==0);
% S.bdface{6} = AllBDFaces(:,sum(AllFaceCoords(1,:,:))==3);

end
```

9.3 Measuring the shape-regularity of a mesh

Given a triangulation $\mathcal{T}_h$, we compute two parameters measuring the shape-regularity of the triangulation

$$d_F := \max_{K\in\mathcal{T}_h} \frac{\max_{F\in\mathcal{F}_K} |F|}{\min_{F\in\mathcal{F}_K} |F|} \qquad d_L := \max_{K\in\mathcal{T}_h} \frac{\max_{e\in\mathcal{E}_K} |e|}{\min_{e\in\mathcal{E}_K} |e|}$$

and a parameter mesuring uniformity

$$d_V := \frac{\max_{K\in\mathcal{T}_h} |K|}{\min_{K\in\mathcal{T}_h} |K|}$$

```
function [ dF,dL,dV ] = meshQualityTest( T )

%function [ dF,dL,dV ] = meshQualityTest( T )
%
% Inputs:
%       T : a basic tetrahedrization
```

```
%
% Outputs:
%       dF : maximum face deformation
%       dL : maximum edge deformation
%       dV : ratio of largest/smallest volumes of elements
%
% Last Modified: February 19, 2016

T = edgesAndFaces(T);
T = enhanceGrid3D(T);

F = T.area(T.facebyelt);

lengths = sqrt(sum((T.coordinates(:, T.edges(1,:))...
                    - T.coordinates(:, T.edges(2,:))).^2));

L = lengths(abs(T.edgebyelt));

dF = max(max(F)/min(F));
dL = max(max(L)/min(L));
dV = max(T.volume)/min(T.volume);

end
```