# Implementation of HDG in three dimensions

Zhixing Fu, Luis F Gatica, Allan Hungria, & Francisco-Javier Sayas

Last update: July 29, 2015

## Contents

Notation of this documentation follows strictly that of the paper

*Matlab tools for HDG in three dimensions*

## 1 Pseudo-Matlab notation

Assume that $\mathbf{u}$ is a column vector with $N$ components and A is an $M \times N$ matrix. We then consider the $M \times N$ matrix $\mathbf{u}^\top \odot \mathrm{A} := \mathrm{A}\,\mathrm{diag}(\mathbf{u})$ with elements

$$(\mathbf{u}^\top \odot \mathrm{A})_{ij} = u_j \mathrm{A}_{ij}.$$

Similarly, if $\mathbf{v}$ is a column vector with $M$ components, we consider the matrix $\mathbf{v} \odot \mathrm{A} := \mathrm{diag}(\mathbf{v})\mathrm{A}$, i.e.,

$$(\mathbf{v} \odot \mathrm{A})_{ij} = v_i \mathrm{A}_{ij}.$$

Assuming correct sizes for the matrices and vectors (vectors will be assumed to be column vectors throughout), both operations can be easily accomplished in Matlab

```
bsxfun(@times,u',A)   % row times matrix
bsxfun(@times,v,A)    % column times matrix
```

We will use Kronecker products in a very particular form. Assume that $\mathbf{c}$ is a column vector with $N$ components and that A is a $m \times n$ matrix. Then

$$\mathbf{c}^\top \otimes \mathrm{A} = \left[\ c_1\mathrm{A} \mid c_2\mathrm{A} \mid \cdots \mid c_N\mathrm{A}\ \right]$$

is a $m \times (nN)$ matrix, organized in $N$ blocks of size $m \times n$. It will be the case that we will want the result stored as a 3-dimensional $m \times n \times N$. This is easily programmed as follows

```
cA=kron(c',A):
cA=reshape(cA,[m,n,N]);
```

For convenience, we will also write

$$\mathbf{a}_i^\top := \mathrm{row}(\mathrm{A}, i)$$

to select the $i$-th row of a matrix A. Finally, in the last part of the code we will use the symbol $\bullet$ to represent the element by element multiplication of arrays (Matlab's `.*` operator).

## 2 Geometric elements

### 2.1 Reference elements

We first consider the reference tetrahedron $\widehat{K}$, with vertices

$$\widehat{\mathbf{v}}_1 := (0,0,0) \qquad \widehat{\mathbf{v}}_2 := (1,0,0), \qquad \widehat{\mathbf{v}}_3 := (0,1,0), \qquad \widehat{\mathbf{v}}_4 := (0,0,1).$$

Note that $|\widehat{K}| := \mathrm{vol}\,\widehat{K} = 1/6$. We also consider the two dimensional reference element $\widehat{K}_2 := \{(s,t) : s,t \ge 0, s+t \le 1\}$ with vertices

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

## 2.2 Tetrahedra

Given a tetrahedron with vertices $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4)$ (the order is relevant), we consider the affine mapping $\mathrm{F}_K : \widehat{K} \to K$

$$\mathrm{F}_K(\widehat{\mathbf{x}}) = \mathrm{B}_K \widehat{\mathbf{x}} + \mathbf{v}_1, \qquad \mathrm{B}_K = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{bmatrix}$$

so that $\mathrm{F}_K(\widehat{\mathbf{v}}_i) = \mathbf{v}_i$ for $i =\in \{1, 2, 3, 4\}$. All elements of the triangulation will be given with positive orientation, that is,

$$\det \mathrm{B}_K = 6|K| = \Big((\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)\Big) \cdot (\mathbf{v}_4 - \mathbf{v}_1) > 0.$$

## 2.3 Parametrization of triangles

A triangle $e$ in $\mathbb{R}^3$ with vertices $(\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3)$ (the order is relevant), will be parametrized via $\boldsymbol{\phi}_e : \widehat{K}_2 \to e$, given by

$$\boldsymbol{\phi}_e(s, t) := s\,(\mathbf{w}_2 - \mathbf{w}_1) + t\,(\mathbf{w}_3 - \mathbf{w}_1) + \mathbf{w}_1, \qquad |\partial_s \boldsymbol{\phi}_e \times \partial_t \boldsymbol{\phi}_e| = 2|e|,$$

so that $\boldsymbol{\phi}_e(\widehat{\mathbf{w}}_i) = \mathbf{w}_i$, for $i \in \{1, 2, 3\}$. The local orientation of the vertices of $e$ gives an orientation to the normal vector: we will define the normal vector so that its norm is proportional to the area of $e$, that is

$$\mathbf{n}_e := \tfrac{1}{2}\Big((\mathbf{w}_2 - \mathbf{w}_1) \times (\mathbf{w}_3 - \mathbf{w}_1)\Big).$$

## 2.4 Parametrization of the faces of a tetrahedron

Given a tetrahedron $K$ with vertices $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4)$ we will consider its four faces given in the following order (and with the inherited orientations):

$$\begin{array}{ccc} e_1^K & \longleftrightarrow & (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) \\ e_2^K & \longleftrightarrow & (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_4) \\ e_3^K & \longleftrightarrow & (\mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_4) \\ e_4^K & \longleftrightarrow & (\mathbf{v}_4, \mathbf{v}_2, \mathbf{v}_3). \end{array} \qquad \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 1 & 3 & 4 \\ 4 & 2 & 3 \end{bmatrix}$$

(Note that with this orientation of the faces, the normals of the second and fourth faces point outwards, while those of the first and third faces point inwards. This numbering is done for the sake of parametrization.)
The parametrizations of the faces $e_\ell^K \in \mathcal{E}(K)$

$$\boldsymbol{\phi}_\ell^K : \widehat{K}_2 \to e_\ell^K \qquad \ell \in \{1, 2, 3, 4\},$$

given by the formulas

$$\begin{aligned} \boldsymbol{\phi}_1^K(s, t) &:= (s, t, 0) \\ \boldsymbol{\phi}_2^K(s, t) &:= (s, 0, t) \\ \boldsymbol{\phi}_3^K(s, t) &:= (0, s, t) \\ \boldsymbol{\phi}_4^K(s, t) &:= (s, t, 1 - s - t) \end{aligned}$$

will be used for integrals on $\partial K$.

## 2.5 The permutation index

Consider the affine invertible maps $F_\mu : \widehat{K}_2 \to \widehat{K}_2$ given by the formulas

$$\begin{aligned} F_1(s,t) &:= (s,t) \\ F_2(s,t) &:= (t,s) \\ F_3(s,t) &:= (t, 1-s-t) \\ F_4(s,t) &:= (s, 1-s-t) \\ F_5(s,t) &:= (1-s-t, s) \\ F_6(s,t) &:= (1-s-t, t) \end{aligned} \qquad \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \mathbf{1} & 3 & 2 \\ 3 & 1 & 2 \\ 3 & \mathbf{2} & 1 \\ 2 & 3 & 1 \\ 2 & 1 & \mathbf{3} \end{bmatrix}.$$

The table on the right shows the indices of images of the vertices $(\widehat{\mathbf{w}}_1, \widehat{\mathbf{w}}_2, \widehat{\mathbf{w}}_3)$, with boldface font for those that stay fixed. We note that $F_2$, $F_4$ and $F_6$ change orientation.
Given a tetrahedron $K$, assume that the face $e = e_\ell^K$. We thus have six possible cases of how the parametrizations $\boldsymbol{\phi}_\ell^K$ and $\boldsymbol{\phi}_e$ match. We will encode this information in a matrix so that

$$\mu = \mathrm{perm}(K,\ell) \quad \text{and} \quad e = e_\ell^K \qquad \text{imply} \qquad \boldsymbol{\phi}_e \circ F_\mu = \boldsymbol{\phi}_\ell^K.$$

## 2.6 A tetrahedrization in basic and expanded forms

We are given a polyhedral domain $\Omega$, with faces grouped in two subsets $\Gamma_D$ and $\Gamma_N$ (for Dirichlet and Neumann boundary conditions), and a tetrahedrization of $\Omega$. The basic tetrahedrization is given through four fields:

- `T.coordinates` is an $N_{\mathrm{ver}} \times 3$ matrix with the coordinates of the vertices of the triangulation,
- `T.elements` is an $N_{\mathrm{elt}} \times 4$ matrix: the $K$-th row of the matrix contains the indices of the vertices of $K$; *positive orientation is assumed*,
- `T.dirichlet` is a $N_{\mathrm{dir}} \times 3$ matrix, with the vertex numbers for the Dirichlet faces,
- `T.neumann` is a $N_{\mathrm{neu}} \times 3$ matrix, with the vertex numbers for the Neumann faces.

For the boundary faces, it is assumed that *either all of them are given with positive orientation* (normals pointing outwards) or *all of them are given with negative orientation*. This is an example of a basic tetrahedral data structure, for a partition with 48 vertices, 108 tetrahedral elements, 36 Dirichlet faces and 48 Neumann faces.

```
T =
    coordinates: [48x3 double]
       elements: [108x4 double]
      dirichlet: [36x3 double]
        neumann: [48x3 double]
```

In **expanded form**, the tetrahedral data structure contains many more useful fields.

- In this form, the faces listed in the fields `T.dirichlet` and `T.neumann` are positively oriented.
- `T.faces` is a $N_{\mathrm{fc}} \times 4$ matrix with a list of faces: the first three columns contain the global vertex numbers for the faces (its order will give the intrinsic parametrization of the face); Dirichlet and Neumann faces are numbered exacly as in `T.dirichlet` and `T.neumann`, the fourth column contains an index:

  - 0 for interior faces
  - 1 for Dirichlet faces
  - 2 for Neumann faces

- `T.dirfaces` and `T.neufaces` are row vectors with the list of Dirichlet and Neumann faces (that is, they point out what rows of `T.faces` contain a 1 (resp a 2) in the last column)

- `T.facebyele` is an $N_{\rm elt} \times 4$ matrix: its $K$-th row contains the numbers of faces that make up $\partial K$; they are given in the order shown in Section 2.4, that is, locally

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 1 & 3 & 4 \\ 4 & 2 & 3 \end{bmatrix}$$

  This is the matrix we have described as $e_\ell^K$.

- `T.perm` is an $N_{\rm elt} \times 4$ matrix containing numbers from 1 to 6; the $K$-th row indicates what permutations are needed for each of the faces to get to the proper numbering of the face (see Section 2.5), i.e., this is just the matrix $\mathrm{perm}(K, \ell)$

- `T.volume` is a $N_{\rm elt} \times 1$ column vector with the volumes of the elements

- `T.area` is a $N_{\rm fc} \times 1$ column vector with the areas of the faces

- `T.normals` is a $N_{\rm elt} \times 12$ matrix with the *non-normalized* normal vectors for the faces of the ech; its $K$-th row contains four row vectors of three components each

$$\left[\ \mathbf{n}_1^\top \mid \mathbf{n}_2^\top \mid \mathbf{n}_3^\top \mid \mathbf{n}_4^\top\ \right]$$

  so that $\mathbf{n}_\ell$ is the normal vector to the face $e_\ell^K$, *pointing outwards* and such that $|\mathbf{n}_\ell| = |e_\ell^K|$.

---

```
T =
    coordinates: [48x3 double]
       elements: [108x4 double]
      dirichlet: [36x3 double]
        neumann: [48x3 double]
          faces: [258x4 double]
       dirfaces: [1x36 double]
       neufaces: [1x48 double]
      facebyele: [108x4 double]
    orientation: [108x4 double]
           perm: [108x4 double]
         volume: [108x1 double]
           area: [258x1 double]
        normals: [108x12 double]
```

---

In what follows we will frequently identify

$$\mathcal{T}_h \equiv \{1, \ldots, N_{\rm elt}\} \qquad \mathcal{E}_h \equiv \{1, \ldots, N_{\rm fc}\}.$$

# 3 Quadrature

## 3.1 Volume integrals

Quadrature rules will be given in the reference element $\widehat{K}$. They will be composed of quadrature points, given by their *barycentric coordinates*, and weights. Geometrically, we can think of points

$$\widehat{\mathbf{p}}_q := (\widehat{x}_q, \widehat{y}_q, \widehat{z}_q), \qquad q = 1, \ldots, N_{\mathrm{qd}}$$

and weights $\widehat{\omega}_q$, with the normalization

$$\sum_{q=1}^{N_{\mathrm{qd}}} \widehat{\omega}_q = 1, \qquad \text{so that} \qquad \int_{\widehat{K}} \widehat{\phi} \approx \tfrac{1}{6} \sum_q \widehat{\omega}_q \widehat{\phi}(\widehat{\mathbf{p}}_q).$$

For a general tetrahedron, we will approximate

$$\int_K \phi = \det \mathrm{B}_K \int_{\widehat{K}} \phi \circ \mathrm{F}_K \approx |K| \sum_q \widehat{\omega}_q \phi(\mathbf{p}_q^K) \qquad \text{with} \qquad \mathbf{p}_q^K = \mathrm{F}_K(\widehat{\mathbf{p}}_q).$$

For convenience, a quadrature formula will be stored in an $N_{\mathrm{qd}} \times 5$ matrix. The $q$-th row contains the barycentric coordinates of $\widehat{\mathbf{p}}_q$ and then the weight $\widehat{\omega}_q$, that is, we store

$$(1 - \widehat{x}_q - \widehat{y}_q - \widehat{z}_q, \widehat{x}_q, \widehat{y}_q, \widehat{z}_q, \widehat{\omega}_q)$$

as rows. We will also consider the $N_{\mathrm{qd}} \times 4$ matrix $\Lambda$ with the barycentric coordinates of the quadrature points.

## 3.2 Integrals on faces

Two dimensional quadrature rules will be given in the reference element $\widehat{K}_2$, using points and weights so that

$$\sum_{r=1}^{N_{\mathrm{qd2}}} \varpi_r = 1 \qquad \text{and thus} \qquad \int_{\widehat{K}_2} \widehat{\phi} \approx \tfrac{1}{2} \sum_r \varpi_r \widehat{\phi}(\widehat{\mathbf{q}}_r), \qquad \widehat{\mathbf{q}}_r = (\widehat{s}_r, \widehat{t}_r).$$

To compute an integral on $e \in \mathcal{E}_h$, we simply parametrize from $\widehat{K}_2$ and proceed accordingly:

$$\int_e \phi = 2|e| \int_{\widehat{K}_2} \phi \circ \boldsymbol{\phi}_e \approx |e| \sum_r \varpi_r \phi(\mathbf{q}_r^e) \qquad \text{with} \qquad \mathbf{q}_r^e := \boldsymbol{\phi}_e(\widehat{\mathbf{q}}_r).$$

A face quadrature formula will be stored in an $N_{\mathrm{qd2}} \times 4$ matrix, with the barycentric coordinates of the quadrature points in the first columns and the weights in the last one. The $r$-th row of this matrix is therefore

$$(1 - \widehat{s}_r - \widehat{t}_r, \widehat{s}_r, \widehat{t}_r, \varpi_r).$$

The $N_{\mathrm{qd2}} \times 3$ with the barycentric coordinates of the quadrature points will be denoted $\Xi$.

## 3.3 Integrals on boundaries of tetrahedra

In many cases we will be integrating on a face that is given with geometric information of an adjacent tetrahedron. The quadrature points $\widehat{\mathbf{q}}_r$ lead to four groups of quadrature points on the faces of $\widehat{K}$ (see Section 2.4):

$$\begin{aligned}
\widehat{\mathbf{q}}_r^1 &:= (\widehat{s}_r, \widehat{t}_r, 0) \\
\widehat{\mathbf{q}}_r^2 &:= (\widehat{s}_r, 0, \widehat{t}_r) \\
\widehat{\mathbf{q}}_r^3 &:= (0, \widehat{s}_r, \widehat{t}_r) \\
\widehat{\mathbf{q}}_r^4 &:= (\widehat{s}_r, \widehat{t}_r, 1 - \widehat{s}_r - \widehat{t}_r)
\end{aligned}$$

For a given $\psi : K \to \mathbb{R}$, we can approximate

$$\int_{e_\ell^K} \psi \approx |e_\ell^K| \sum_r \varpi_r \psi(\mathbf{q}_{r,\ell}^K) \qquad \text{with} \qquad \mathbf{q}_{r,\ell}^K = \mathrm{F}_K(\widehat{\mathbf{q}}_r^\ell) = \boldsymbol{\phi}_{e_\ell^K}(F_{\mathrm{perm}(K,\ell)}(\widehat{\mathbf{q}}_r))$$

and thus

$$\int_{\partial K} \psi \approx \sum_{\ell=1}^{4} |e_\ell^K| \sum_r \varpi_r \psi(\mathbf{q}_{r,\ell}^K).$$

### 3.4 Update - Stroud Quadrature

For any polynomial degree `k`, the function `computeQuadrature.m` creates four Stroud quadrature formulas, namely:

- `formula{1}` 3D Stroud quadrature formula of degree `3k` or `2k`
- `formula{2}` 3D Stroud quadrature formula of degree `2k`
- `formula{3}` 2D Stroud quadrature formula of degree `2k`
- `formula{4}` 2D Stroud quadrature formula of degree `2k+2`

See the Quadrature section of the FEM3D documentation for specifics.

## 4 Dubiner polynomial bases

### 4.1 The Dubiner basis in two variables

Let

$$d_2 = d_2(k) = \binom{k+2}{2} = \dim \mathcal{P}_k(\widehat{K}_2).$$

The Dubiner basis in two variables is a basis of the space of bivariate polynomials such that

$$\int_{\check{K}_2} \check{D}_i \, \check{D}_j = 0 \qquad i \neq j, \qquad \check{K}_2 = \{2\widehat{\mathbf{x}} - (1,1)^\top \,:\, \widehat{\mathbf{x}} \in \widehat{K}_2\}.$$

It is ordered in such a way that

$$\mathcal{P}_k(\check{K}_2) = \mathrm{span}\{\check{D}_j \,:\, j \leq d_2(k)\} \qquad \forall k \geq 0.$$

The Dubiner basis is evaluated using Jacobi polynomials (here we use a small variation of code by John Buckhart for the evaluation of the Jacobi polynomials). Details on how this is coded can be found in the documentation of the 2-dimensional HDG code. Given $N_{\text{points}}$ points $\mathbf{q}_r = (s_r, t_r)$, the function **dubiner2d** returns the matrices

$$\check{D}_j(\mathbf{q}_r), \qquad \partial_s \check{D}_j(\mathbf{q}_r), \qquad \partial_t \check{D}_j(\mathbf{q}_r).$$

Output is given as three $N_{\text{points}} \times d_2$ matrices.

```
function [db,dbx,dby]=dubiner2d(x,y,k)
% function [db,dbx,dby]=dubiner2d(x,y,k)
% input:
%       x,y: the coordinates at which we evaluate the Dubiner basis
%         k: the degree of the polynomial. Must be a column vector.
% output:
%        db: the value of dubiner basis at given coordinates, Nnodes x Nbasis
%       dbx: the derivative to x of the Dubiner basis, Nnodes x Nbasis
%      dby: the derivative to y of the Dubiner basis, Nnodes x Nbasis
%
% Last modified: June 5 2012

Nnodes=size(x,1);
Nbasis=nchoosek(k+2,2);
db=zeros(Nnodes,Nbasis);
dbx=zeros(Nnodes,Nbasis);
dby=zeros(Nnodes,Nbasis);
eta1(y≠1)=2*(1+x(y≠1))./(1-y(y≠1))-1;
eta1(y==1)=-1; eta1=eta1';
eta2=y;
% locate the index with p,q
a=zeros(k+1,k+1);
index=1;
for l=2:k+2
    for i=l-1:-1:1
        a(i,l-i)=index;
        index=index+1;
    end
end
loc=@(p,q) a(p+1,q+1);
%
% Dubiner Basis
JP=JacobiP(k,0,0,eta1);
for p=0:k
    A=JP(:,p+1).*((1-eta2)/2).^p;
    JPQ=JacobiP(k-p,2*p+1,0,eta2);
    for q=0:k-p
        db(:,loc(p,q))=A.*JPQ(:,q+1);
    end
end
%
% Derivative of Dubiner Basis
dbx(:,loc(0,0))=0.*x;           % DB_x^{0,0}
dby(:,loc(0,0))=0.*y;           % DB_y^{0,0}
if k>0
    dbx(:,loc(1,0))=1+0.*x;         % DB_x^{1,0}
    dby(:,loc(1,0))=1/2+0.*y;       % DB_y^{1,0}
end
for p=1:k-1
    dbx(:,loc(p+1,0))=(2*p+1)/(p+1)*(1+2.*x+y)/2.*dbx(:,loc(p,0))...
                      -p/(p+1)*(1-y).^2/4.*dbx(:,loc(p-1,0))...
                      +(2*p+1)/(p+1)*db(:,loc(p,0));
    dby(:,loc(p+1,0))=1/2*(2*p+1)/(p+1)*db(:,loc(p,0))...
                      +(2*p+1)/(p+1)*(1+2*x+y)/2.*dby(:,loc(p,0))...
                      -p/(p+1)*(1/2* (y-1).*db(:,loc(p-1,0))+(1-y).^2/4.*dby(:,loc(p-1,0)) );
end
for p=0:k-1
    dbx(:,loc(p,1))=dbx(:,loc(p,0)).*(1+2*p+(3+2*p)*y)/2;
    dby(:,loc(p,1))=dby(:,loc(p,0)).*(1+2*p+(3+2*p)*y)/2+(3+2*p)/2*db(:,loc(p,0));
end
for p=0:k-1
    for q=1:k-p-1
        a=(2*q+2*p+2)*(2*q+2*p+3)/2/(q+1)/(q+2*p+2);
        b=(2*q+2*p+2)*(2*p+1)^2/2/(q+1)/(q+2*p+2)/(2*q+2*p+1);
        c=(2*q+2*p+3)*(q+2*p+1)*q/(q+1)/(q+2*p+2)/(2*q+2*p+1);
        dbx(:,loc(p,q+1))=(a*y+b).*dbx(:,loc(p,q))-c*dbx(:,loc(p,q-1));
        dby(:,loc(p,q+1))=(a*y+b).*dby(:,loc(p,q))-c*dby(:,loc(p,q-1))+a*db(:,loc(p,q));
    end
end

return


function v = JacobiP(n,alpha,beta,x)
```

```
% Subfunction with evaluation of Jacobi polynomials
% taken from code by John Burkardt,
% distributed under the GNU LGPL license

v=zeros(size(x,1),n+1);
v(:,1) = 1.0;
if ( n == 0 )
    return
end

v(:,2) = ( 1.0 + 0.5 * ( alpha + beta ) ) * x(:)  + 0.5 * ( alpha - beta );

for i = 2 : n
    c1 = 2 * i * ( i + alpha + beta ) * ( 2 * i - 2 + alpha + beta );
    c2 = ( 2 * i - 1 + alpha + beta ) * ( 2 * i + alpha + beta ) ...
      * ( 2 * i - 2 + alpha + beta );
    c3 = ( 2 * i - 1 + alpha + beta ) * ( alpha + beta ) * ( alpha - beta );
    c4 = - 2 * ( i - 1 + alpha ) * ( i - 1 + beta )  * ( 2 * i + alpha + beta );
    v(:,i+1) = ( ( c3 + c2 * x(:) ) .* v(:,i) + c4 * v(:,i-1) ) / c1;
end

return
```

## 4.2 The Dubiner basis in three variables

We now denote

$$d_3 := d_3(k) = \binom{k+3}{3} = \dim \mathcal{P}_k(\widehat{K}).$$

A basis $\{\check{P}_j\}$ of the space of 3-variate polynomials is given with the orthogonality property

$$\int_{\check{K}} \check{P}_i \check{P}_j = 0 \qquad i \neq j, \qquad \check{K} = \{2\widehat{\mathbf{x}} - (1,1,1)^\top \,:\, \widehat{\mathbf{x}} \in \widehat{K}\}.$$

We also assume that

$$\mathcal{P}_k(\check{K}) = \operatorname{span}\{P_j \,:\, j \leq d_3(k)\} \qquad \forall k \geq 0.$$

Output is given in a similar way to the `dubiner2d` code: given points $\mathbf{p}_q$, we evaluate

$$\check{P}_j(\mathbf{p}_q), \qquad \partial_x \check{P}_j(\mathbf{p}_q), \qquad \partial_y \check{P}_j(\mathbf{p}_q), \qquad \partial_z \check{P}_j(\mathbf{p}_q)$$

and output as four $N_{\text{points}} \times d_3$ matrices.

```
function [db,dbx,dby,dbz]=dubiner3d(x,y,z,k)
% function [db,dbx,dby,dbz]=dubiner3d(x,y,z,k)
% input:
%     x,y,z: the coordinates at which we evaluate the Dubiner basis
%         k: the degree of the polynomial. Must be a column vector.
% output:
%       db: the value of dubiner basis at given coordinates, Nnodes x Nbasis
%      dbx: the derivative to x of the Dubiner basis, Nnodes x Nbasis
%      dby: the derivative to y of the Dubiner basis, Nnodes x Nbasis
%      dbz: the derivative to y of the Dubiner basis, Nnodes x Nbasis
% Last modified: May 16 2012

Nnodes=size(x,1);
Nbasis=nchoosek(k+3,3);
db=zeros(Nnodes,Nbasis);
dbx=zeros(Nnodes,Nbasis);
dby=zeros(Nnodes,Nbasis);
dbz=zeros(Nnodes,Nbasis);
```

```
a=@(al,be,n) ( (2*n+1+al+be)*(2*n+2+al+be) )/ ( 2*(n+1)*(n+1+al+be) );
b=@(al,be,n) ( (al^2-be^2)*(2*n+1+al+be) ) / ( 2*(n+1)*(2*n+al+be)*...
                                                          (n+1+al+be)  );
c=@(al,be,n) ( (n+al)*(n+be)*(2*n+2+al+be) ) / ( (n+1)*(n+1+al+be)*...
                                                (2*n+al+be)  );

% Locate the basis with index p,q,r
index=zeros(k+1,k+1,k+1);
count=1;
%
for j=0:k
    for i1=j:-1:0
        for i2=j-i1:-1:0
            index(i1+1,i2+1,j-i1-i2+1)=count;
            count=count+1;
        end
    end
end
loc=@(i1,i2,i3)  index(i1+1,i2+1,i3+1);


F1= (2+2*x+y+z)/2;  F1x= 1+0*x;   F1y= 1/2+0*x;    F1z= 1/2+0*x;
F2= ((y+z)/2).^2;   F2x= 0.*x;    F2y= 1/2*(y+z);  F2z= 1/2*(y+z);
F3= (2+3*y+z)/2;    F3x= 0+0.*x;  F3y= 3/2+0.*x;   F3z= 1/2+0.*x;
F4= (1+2*y+z)/2;    F4x= 0+0.*x;  F4y= 1+0.*x;     F4z= 1/2+0.*x;
F5= (1-z)/2;        F5x= 0+0.*x;  F5y= 0+0.*x;     F5z= -1/2+0.*x;

db(:,loc(0,0,0))=1;
dbx(:,loc(0,0,0))=0*x;   dby(:,loc(0,0,0))=0*y;  dbz(:,loc(0,0,0))=0*z;
if k>0
    db(:,loc(1,0,0))=F1;
    dbx(:,loc(1,0,0))=F1x; dby(:,loc(1,0,0))=F1y; dbz(:,loc(1,0,0))=F1z;
end
for p=1:k-1
    db(:,loc(p+1,0,0))=(2*p+1)/(p+1)*F1.*db(:,loc(p,0,0))...
        -(p/(p+1))*F2.*db(:,loc(p-1,0,0));
    dbx(:,loc(p+1,0,0))=(2*p+1)/(p+1)*( F1x.*db(:,loc(p,0,0)) + ...
        F1.*dbx(:,loc(p,0,0)) ) - (p/(p+1))*(  F2x.*db(:,loc(p-1,0,0))+ ...
        F2.*dbx(:,loc(p-1,0,0))  );
    dby(:,loc(p+1,0,0))=(2*p+1)/(p+1)*( F1y.*db(:,loc(p,0,0)) + ...
        F1.*dby(:,loc(p,0,0)) ) - (p/(p+1))*(  F2y.*db(:,loc(p-1,0,0))+ ...
        F2.*dby(:,loc(p-1,0,0))  );
    dbz(:,loc(p+1,0,0))=(2*p+1)/(p+1)*( F1z.*db(:,loc(p,0,0)) + ...
        F1.*dbz(:,loc(p,0,0)) ) - (p/(p+1))*(  F2z.*db(:,loc(p-1,0,0))+ ...
        F2.*dbz(:,loc(p-1,0,0))  );
end

for p=0:k-1
    db(:,loc(p,1,0))=(p*(1+y)+F3).*db(:,loc(p,0,0));
    dbx(:,loc(p,1,0)) = F3x.*db(:,loc(p,0,0)) +...
        dbx(:,loc(p,0,0)).*(p*(1+y)+F3);
    dby(:,loc(p,1,0)) = (p+F3y).*db(:,loc(p,0,0)) +...
        dby(:,loc(p,0,0)).*(p*(1+y)+F3);
    dbz(:,loc(p,1,0)) = F3z.*db(:,loc(p,0,0)) +...
        dbz(:,loc(p,0,0)).*(p*(1+y)+F3);
end

for p=0:k-2
    for q=1:k-p-1
        db(:,loc(p,q+1,0))=(a(2*p+1,0,q)*F4+b(2*p+1,0,q)*F5).*db(:,loc(p,q,0))...
            -c(2*p+1,0,q)*F5.^2.*db(:,loc(p,q-1,0));
        dbx(:,loc(p,q+1,0))=(a(2*p+1,0,q)*F4x+b(2*p+1,0,q)*F5x).*db(:,loc(p,q,0))...
            +(a(2*p+1,0,q)*F4+b(2*p+1,0,q)*F5).*dbx(:,loc(p,q,0))...
            -2*c(2*p+1,0,q)*F5.*F5x.*db(:,loc(p,q-1,0))...
            -c(2*p+1,0,q)*F5.^2.*dbx(:,loc(p,q-1,0));
        dby(:,loc(p,q+1,0))=(a(2*p+1,0,q)*F4y+b(2*p+1,0,q)*F5y).*db(:,loc(p,q,0))...
```

```
            +(a(2*p+1,0,q)*F4+b(2*p+1,0,q)*F5).*dby(:,loc(p,q,0))...
            -2*c(2*p+1,0,q)*F5.*F5y.*db(:,loc(p,q-1,0))...
            -c(2*p+1,0,q)*F5.^2.*dby(:,loc(p,q-1,0));
        dbz(:,loc(p,q+1,0))=(a(2*p+1,0,q)*F4z+b(2*p+1,0,q)*F5z).*db(:,loc(p,q,0))...
            +(a(2*p+1,0,q)*F4+b(2*p+1,0,q)*F5).*dbz(:,loc(p,q,0))...
            -2*c(2*p+1,0,q)*F5.*F5z.*db(:,loc(p,q-1,0))...
            -c(2*p+1,0,q)*F5.^2.*dbz(:,loc(p,q-1,0));

    end
end

for p=0:k-1
    for q=0:k-p-1
        db(:,loc(p,q,1)) =(1+p+q+(2+q+p)*z).*db(:,loc(p,q,0));
        dbx(:,loc(p,q,1)) =(1+p+q+(2+q+p)*z).*dbx(:,loc(p,q,0));
        dby(:,loc(p,q,1)) =(1+p+q+(2+q+p)*z).*dby(:,loc(p,q,0));
        dbz(:,loc(p,q,1)) =(1+p+q+(2+q+p)*z).*dbz(:,loc(p,q,0))...
            +(2+p+q)*db(:,loc(p,q,0));
    end
end

for p=0:k-2
    for q=0:k-p-2
        for r=1:k-p-q-1
            db(:,loc(p,q,r+1))=(a(2*p+2*q+2,0,r)*z+b(2*p+2*q+2,0,r)).*...
                db(:,loc(p,q,r))-c(2*p+2*q+2,0,r)*db(:,loc(p,q,r-1));
            dbx(:,loc(p,q,r+1))=(a(2*p+2*q+2,0,r)*z+b(2*p+2*q+2,0,r)).*...
                dbx(:,loc(p,q,r))-c(2*p+2*q+2,0,r)*dbx(:,loc(p,q,r-1));
            dby(:,loc(p,q,r+1))=(a(2*p+2*q+2,0,r)*z+b(2*p+2*q+2,0,r)).*...
                dby(:,loc(p,q,r))-c(2*p+2*q+2,0,r)*dby(:,loc(p,q,r-1));
            dbz(:,loc(p,q,r+1))=(a(2*p+2*q+2,0,r)*z+b(2*p+2*q+2,0,r)).*...
                dbz(:,loc(p,q,r))-c(2*p+2*q+2,0,r)*dbz(:,loc(p,q,r-1))...
                + a(2*p+2*q+2,0,r)*db(:,loc(p,q,r));
        end
    end
end



return
```

## 5 Volume matrices and integrals

### 5.1 Representation of a piecewise polynomial function

Let $k$ be a fixed polynomial degree and consider

$$\widehat{P}_j := \check{P}_j(2\,\cdot\,-(1,1,1)^\top), \qquad \mathcal{P}_k(\widehat{K}) = \mathrm{span}\{\widehat{P}_j\,:\,1\le j\le d_3\}.$$

This basis is hierarchical. We then consider the following basis of $\mathcal{P}_k(K)$

$$P_i^K := \widehat{P}_i \circ \mathrm{F}_K^{-1}, \qquad i=1,\ldots,d_3.$$

A function in the space

$$W_h := \prod_{K\in\mathcal{T}_h} \mathcal{P}_k(K) = \{u_h:\Omega\to\mathbb{R}\,:\,u_h|_K\in\mathcal{P}_k(K)\quad \forall K\in\mathcal{T}_h\}$$

will be represented in two possible forms: as a $d_3\times N_{\mathrm{elt}}$ matrix, whose $K$-th column contains the coefficients of $u_h|_K$ in the basis $\{P_i^K\}$, or as a $d_3\,N_{\mathrm{elt}}$ column vector, with $d_3$-sized blocks containing the same values. *The matrix storage form will be preferred.* The `reshape` command moves from one to the other easily.

**Warning.** At every evaluation of the basis functions or their derivatives, we need to apply the following rule

$$\widehat{P}_i(\mathbf{x}) = \check{P}_i(2\mathbf{x}-\mathbf{1}), \qquad \partial_{\widehat{\star}}\widehat{P}_i(\mathbf{x}) = 2\partial_{\check{\star}}\check{P}_i(2\mathbf{x}-\mathbf{1}), \qquad \star \in \{x,y,z\}, \qquad \mathbf{1} := (1,1,1)^\top.$$

## 5.2 Computing all quadrature nodes at once

Let $\Lambda$ be the $N_{\rm qd}\times 4$ matrix with the barycentric coordinates of the nodes of a quadrature formula (see Section 3.1). Let

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

be the $4\times N_{\rm elt}$ matrices with the $(x,y,z)$ coordinates of the four vertices of all elements (we count elements by rows and vertices by columns). Then, the $N_{\rm qd}\times N_{\rm elt}$ matrices

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

contain the coordinates of all quadrature nodes on all the elements. Therefore, if $f$ is a vectorized function of three variables, the $N_{\rm qd}\times N_{\rm elt}$ matrix $f(\mathrm{X},\mathrm{Y},\mathrm{Z})$ contains the values of $f$ at all the quadrature nodes. The Matlab instruction to generate these points based on our data structure are:

```
x=T.coordinates(:,1); x=formula(:,[1 2 3 4])*x(T.elements');
y=T.coordinates(:,1); y=formula(:,[1 2 3 4])*y(T.elements');
z=T.coordinates(:,1); z=formula(:,[1 2 3 4])*z(T.elements');
```

## 5.3 Testing a function on elements

Given a vectorized function $f:\Omega\to\mathbb{R}$, we aim to compute the integrals

$$\int_K f P_i^K \qquad i=1,\ldots,d_3, \quad K\in\mathcal{T}_h,$$

and store them in a $d_3\times N_{\rm elt}$ matrix. Let

$$\mathrm{P}_{qj} := \widehat{P}_j(\widehat{\mathbf{p}}_q), \qquad q=1,\ldots,N_{\rm qd}, \qquad j=1,\ldots,d_3.$$

Then

$$\int_K f P_i^K \approx |K|\sum_q \widehat{\omega}_q f(\mathbf{p}_K^q)\widehat{P}_i(\widehat{\mathbf{p}}_q) = |K|\sum_q \widehat{\omega}_q \mathrm{P}_{qi} f(\mathbf{p}_K^q), \qquad i=1,\ldots,d_3, \quad K\in\mathcal{T}_h.$$

If **vol** is the column vector with the volumes of all elements if $\widehat{\boldsymbol\omega}$ is the column vector with the weights of the quadrature rule, this formula is

$$\mathbf{vol}^\top \odot \Big((\widehat{\boldsymbol\omega}\odot \mathrm{P})^\top f(\mathrm{X},\mathrm{Y},\mathrm{Z})\Big).$$

**Implementation notes.** The code admits a row cell array of functions as input. If the array contains only one function, the output is a matrix. If the input contains several functions, then the output is a row cell array with the matrices.

```
function Ints=testElem(f,T,k,formula)

%{Int1,Int2,...}=testElem({f1,f2,...},T,k,formula)
%             Int=testElem({f},T,k,formula)
%
%Input:
% {f1,f2...} : cell array with vectorized functions of three variables
%          T : expanded tetrahedrization
%    formula : 3d quadrature formula (Nnd x 5)
%          k : polynomial degree
%Output:
% {Int1,Int2,...}: each cell is d3 x Nelts (\int_K f{l} P_i^K)
%            Int : matrix \int_K f{l} P_i^K
%Last modified: March 14, 2013

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');
xhat=formula(:,2);
yhat=formula(:,3);
zhat=formula(:,4);

P=dubiner3d(2*xhat-1,2*yhat-1,2*zhat-1,k);
wP=bsxfun(@times,formula(:,5),P);

nInts=size(f,2);
if nInts==1
    Ints=bsxfun(@times,T.volume',wP'*f(x,y,z));
else
    Ints=cell(1,nInts);
    for n=1:nInts
        ff=f{n};
        Ints{n}=bsxfun(@times,T.volume',wP'*ff(x,y,z));
    end
end
return
```

## 5.4 Mass matrices

Given a vectorized function $m : \Omega \to \mathbb{R}$, we aim to compute the integrals

$$\int_K m\, P_i^K P_j^K \qquad i,j = 1,\ldots,d_3, \quad K \in \mathcal{T}_h,$$

and store them in a $d_3 \times d_3 \times N_{\rm elt}$ array. We first change to the reference element and apply there a quadrature rule:

$$\int_K m\, P_i^K P_j^K \approx |K| \sum_q \widehat{\omega}_q(\mathbf{p}_q^K)\widehat{P}_i(\widehat{\mathbf{p}}_q)\widehat{P}_j(\widehat{\mathbf{p}}_q).$$

Let X, Y, Z be the $N_{\rm qd} \times N_{\rm elt}$ matrices with the coordinates of all quadrature nodes and let **vol** be the column vector with the volumes of the elements. We thus consider the $N_{\rm qd} \times N_{\rm elt}$ matrix

$$\mathrm{M} = \mathbf{vol}^\top \odot m(\mathrm{X},\mathrm{Y},\mathrm{Z}),$$

compute

$$\sum_q \mathbf{m}_q^\top \otimes (\widehat{\omega}_q \mathbf{p}_q \mathbf{p}_q^\top), \qquad \mathbf{m}_q^\top = \mathrm{row}(\mathrm{M}, q), \qquad \mathbf{p}_q^\top = \mathrm{row}(\mathrm{P}, q),$$

and reshape this $d_3 \times (d_3 N_{\rm elt})$ matrix as a $d_3 \times d_3 \times N_{\rm elt}$ array.

**Implementation notes.** The code produces mass matrices associated to several coefficients. The input is a row cell array and the output a cell array whose elements are the mass matrices.

```
function Mass=MassMatrix(T,coeffs,k,formula)

%{M1,M2,...}=MassMatrix(T,{c1,c2,...},k,formula)
%Input:
%           T: expanded etrahedrization
% {c1,c2,...}: cell array with vectorized functions of three variables
%           k: polynomial degree
%     formula: quadrature formula in 3d (N x 5 matrix)
%
%Output:
% {M1,M2,...}: each cell is d3 x d3 x Nelts (\int_K c{1} P_i^K P_j^K )
%
%Last modified: March 14, 2013

Nnodes=size(formula,1);
Nelts=size(T.elements,1);
d3=nchoosek(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'); % Nnd x Nelts
xhat=formula(:,2);
yhat=formula(:,3);
zhat=formula(:,4);

P=dubiner3d(2*xhat-1,2*yhat-1,2*zhat-1,k);  % Nnd x d3

nMass=size(coeffs,2);
Mass=cell(1,nMass);
for n=1:nMass
    c=coeffs{n};
    C=bsxfun(@times,T.volume',c(x,y,z));           % Nnd x Nelts
    mass=zeros(d3,Nelts*d3);
    for q=1:Nnodes
        mass=mass+kron(C(q,:),formula(q,5)*P(q,:)'*P(q,:));
    end
    mass=reshape(mass,[d3,d3,Nelts]);
    Mass{n}=mass;
end
return
```

## 5.5 Convection matrices

The next aim is the computation of the matrices

$$\int_K P_i^K \partial_\star P_j^K, \qquad i,j=1,\dots,d_3, \quad K\in\mathcal{T}_h, \quad \star\in\{x,y,z\}$$

to be stored in $d_3\times d_3\times N_{\rm elt}$ arrays $\mathrm{C}^\star$. The first step will require the computation of matrices on the reference element by means of a quadrature formula of sufficiently high order:

$$\widehat{\mathrm{C}}^\star_{ij} := \int_{\widehat K} \widehat P_i \partial_{\widehat\star}\widehat P_j = \tfrac16 \sum_q \widehat\omega_q \widehat P_i(\widehat{\mathbf{p}}_q)\partial_{\widehat\star}\widehat P_j(\widehat{\mathbf{p}}_q), \qquad i,j=1,\dots,d_3, \qquad \star\in\{x,y,z\}.$$

If

$$\mathrm{P}^\star_{qi} := \partial_{\widehat\star}\widehat P_i(\widehat{\mathbf{p}}_q), \qquad q=1,\dots,N_{\rm qd}, \quad i=1,\dots,d_3, \qquad \star\in\{x,y,z\}$$

then

$$\widehat{\mathrm{C}}^\star = \tfrac16(\widehat{\boldsymbol\omega}\odot\mathrm{P})^\top\mathrm{P}^\star.$$

We next make a change of variables to the reference element

$$\begin{aligned}\int_K P_i^K \begin{bmatrix} \partial_x P_j^K \\ \partial_y P_j^K \\ \partial_z P_j^K \end{bmatrix} &= \det \mathrm{B}_K \int_{\widehat{K}} \widehat{P}_i \mathrm{B}_K^{-\top} \begin{bmatrix} \partial_{\widehat{x}} \widehat{P}_j^K \\ \partial_{\widehat{y}} \widehat{P}_j^K \\ \partial_{\widehat{z}} \widehat{P}_j^K \end{bmatrix} \\ &= \begin{bmatrix} a_{xx}^K & a_{xy}^K & a_{xz}^K \\ a_{yx}^K & a_{yy}^K & a_{yz}^K \\ a_{zx}^K & a_{zy}^K & a_{zz}^K \end{bmatrix} \begin{bmatrix} \widehat{\mathrm{C}}_{ij}^x \\ \widehat{\mathrm{C}}_{ij}^y \\ \widehat{\mathrm{C}}_{ij}^z \end{bmatrix}, \qquad \det \mathrm{B}_K \mathrm{B}_K^{-\top} = \begin{bmatrix} a_{xx}^K & a_{xy}^K & a_{xz}^K \\ a_{yx}^K & a_{yy}^K & a_{yz}^K \\ a_{zx}^K & a_{zy}^K & a_{zz}^K \end{bmatrix},\end{aligned}$$

with (the element index $K$ is omitted)

$$\begin{aligned} a_{xx} &= (y_3-y_1)(z_4-z_1)-(y_4-y_1)(z_3-z_1), \\ a_{xy} &= (y_4-y_1)(z_2-z_1)-(y_2-y_1)(z_4-z_1), \\ a_{xz} &= (y_2-y_1)(z_3-z_1)-(y_3-y_1)(z_2-z_1), \\ a_{yx} &= (x_4-x_1)(z_3-z_1)-(x_3-x_1)(z_4-z_1), \\ a_{yy} &= (x_2-x_1)(z_4-z_1)-(x_4-x_1)(z_2-z_1), \\ a_{yz} &= (x_3-x_1)(z_2-z_1)-(x_2-x_1)(z_3-z_1), \\ a_{zx} &= (x_3-x_1)(y_4-y_1)-(x_4-x_1)(y_3-y_1), \\ a_{zy} &= (x_4-x_1)(y_2-y_1)-(x_2-x_1)(y_4-y_1), \\ a_{zz} &= (x_2-x_1)(y_3-y_1)-(x_3-x_1)(y_2-y_1). \end{aligned}$$

If the previous nine quantities are computed for each of the elements and stored in nine *column* vectors with $N_{\text{elt}}$ components, it follows that

$$\begin{aligned} \mathrm{C}^x &= \mathbf{a}_{xx}^\top \otimes \widehat{\mathrm{C}}^x + \mathbf{a}_{xy}^\top \otimes \widehat{\mathrm{C}}^y + \mathbf{a}_{xz}^\top \otimes \widehat{\mathrm{C}}^z, \\ \mathrm{C}^y &= \mathbf{a}_{yx}^\top \otimes \widehat{\mathrm{C}}^x + \mathbf{a}_{yy}^\top \otimes \widehat{\mathrm{C}}^y + \mathbf{a}_{yz}^\top \otimes \widehat{\mathrm{C}}^z, \\ \mathrm{C}^z &= \mathbf{a}_{zx}^\top \otimes \widehat{\mathrm{C}}^x + \mathbf{a}_{zy}^\top \otimes \widehat{\mathrm{C}}^y + \mathbf{a}_{zz}^\top \otimes \widehat{\mathrm{C}}^z, \end{aligned}$$

that is

$$\mathrm{C}^\star = \sum_{\# \in \{x,y,z\}} \mathbf{a}_{\star\#}^\top \otimes \widehat{\mathrm{C}}^{\#}.$$

```
function [convx,convy,convz]=ConvMatrix(T,k,formula)

%[convx,convy,convz]=ConvMatrix(T,k,formula)
%
%Input:
%        T: expanded tetrahedrization
%        k: polynomial degree
%  formula: quadrature formula in 3d (N x 5 matrix)
%
%Output:
%    convx: d3 x d3 x Nelts  ( \int_K P_i^K \partial_x P_j^K )
%    convy: d3 x d3 x Nelts  ( \int_K P_i^K \partial_y P_j^K )
%    convz: d3 x d3 x Nelts  ( \int_K P_i^K \partial_z P_j^K )
%
%Last modified: March 14, 2013

Nelts=size(T.elements,1);
d3=nchoosek(k+3,3);
```

```
xhat=formula(:,2);
yhat=formula(:,3);
zhat=formula(:,4);

[p,px,py,pz]=dubiner3d(2*xhat-1,2*yhat-1,2*zhat-1,k);
px=2*px;
py=2*py;
pz=2*pz;
wp=bsxfun(@times,formula(:,5),p);

convhatx=1/6*wp'*px;
convhaty=1/6*wp'*py;
convhatz=1/6*wp'*pz;

x12=T.coordinates(T.elements(:,2),1)-T.coordinates(T.elements(:,1),1); %x2-x1
x13=T.coordinates(T.elements(:,3),1)-T.coordinates(T.elements(:,1),1); %x3-x1
x14=T.coordinates(T.elements(:,4),1)-T.coordinates(T.elements(:,1),1); %x4-x1
y12=T.coordinates(T.elements(:,2),2)-T.coordinates(T.elements(:,1),2); %y2-y1
y13=T.coordinates(T.elements(:,3),2)-T.coordinates(T.elements(:,1),2); %y3-y1
y14=T.coordinates(T.elements(:,4),2)-T.coordinates(T.elements(:,1),2); %y4-y1
z12=T.coordinates(T.elements(:,2),3)-T.coordinates(T.elements(:,1),3); %z2-z1
z13=T.coordinates(T.elements(:,3),3)-T.coordinates(T.elements(:,1),3); %z3-z1
z14=T.coordinates(T.elements(:,4),3)-T.coordinates(T.elements(:,1),3); %z4-z1

axx=y13.*z14-y14.*z13;
axy=y14.*z12-y12.*z14;
axz=y12.*z13-y13.*z12;
ayx=x14.*z13-x13.*z14;
ayy=x12.*z14-x14.*z12;
ayz=x13.*z12-x12.*z13;
azx=x13.*y14-x14.*y13;
azy=x14.*y12-x12.*y14;
azz=x12.*y13-x13.*y12;

convx=kron(axx',convhatx)+kron(axy',convhaty)+kron(axz',convhatz);
convy=kron(ayx',convhatx)+kron(ayy',convhaty)+kron(ayz',convhatz);
convz=kron(azx',convhatx)+kron(azy',convhaty)+kron(azz',convhatz);

convx=reshape(convx,[d3,d3,Nelts]);
convy=reshape(convy,[d3,d3,Nelts]);
convz=reshape(convz,[d3,d3,Nelts]);
return
```

## 5.6 A function for errors

Given a function $u:\Omega\to\mathbb{R}$ and a piecewise polynomial function $u_h\in W_h$, the aim of this part is the approximated computation of

$$\Big(\sum_{K\in\mathcal{T}_h}|u-u_h|^2\Big)^{1/2}.$$

As usual, let X, Y, Z be the $N_{\mathrm{qd}}\times N_{\mathrm{elt}}$ matrices with the coordinates of the quadrature nodes on all the elements and let

$$\mathrm{P}_{qi}:=\widehat{P}_i(\widehat{\mathbf{p}}_q)\qquad q=1,\ldots,N_{\mathrm{qd}},\quad i=1,\ldots,d_3.$$

Assuming that the coefficients of $u_h$ are given in a $d_3\times N_{\mathrm{elt}}$ matrix U, it follows that PU are the values of $u_h$ on all quadrature nodes, and therefore the $N_{\mathrm{qd}}\times N_{\mathrm{elt}}$ matrix

$$\mathrm{E}:=\mathrm{PU}-u(\mathrm{X},\mathrm{Y},\mathrm{Z})\qquad \mathrm{E}_{qK}:=u_h(\mathbf{p}_q^K)-u(\mathbf{p}_q^K)$$

contains the differences at all quadrature points. The computation of the error is just a vector-matrix-vector multiplication, after element-by-element squaring the values $\mathrm{E}_{qK}$:

$$\Big(\sum_{q,K} \widehat{\omega}_q \mathrm{E}_{qK}^2 |K|\Big)^{1/2}.$$

```
function error=errorElem(T,p,ph,k,formula)

%error=errorElem(T,p,ph,k,formula)
%
%Input:
%          T: expanded tetrahedrization
%          p: vectorized function of three variables
%         ph: discontinuous Pk function (d3 x Nelts)
%          k: polynomial degree
%    formula: quadrature formula in 3d (N x 5 matrix)
%
%Output:
%      error: \| p - ph \|_{L^2}
%
%Last modified: May 31, 2012

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=p(x,y,z);

xhat=formula(:,2);
yhat=formula(:,3);
zhat=formula(:,4);
B=dubiner3d(2*xhat-1,2*yhat-1,2*zhat-1,k);
ph=B*ph;

error=sqrt(formula(:,5)'*(p-ph).^2*T.volume);
return
```

## 6 Surface matrices and integrals

### 6.1 Piecewise polynomial functions on the skeleton

Given a face $e \in \mathcal{E}_h$, a basis for $\mathcal{P}_k(e)$ is defined by pushing forward the basis on $\widehat{K}_2$ using the parametrization $\boldsymbol{\phi}_e$, namely,

$$D_i^e \circ \boldsymbol{\phi}_e = \widehat{D}_i, \qquad i = 1, \ldots, d_2, \qquad e \in \mathcal{E}_h.$$

The skeleton of the triangulation $\partial\mathcal{T}_h$ is the set formed by joining all faces of all elements. A function in the space

$$M_h := \prod_{e\in\mathcal{E}_h} \mathcal{P}_k(e) = \{\widehat{u}_h : \partial\mathcal{T}_h \to \Omega \,:\, \widehat{u}_h|_e \in \mathcal{P}_k(e) \quad \forall e \in \mathcal{E}_h\}, \qquad \partial\mathcal{T}_h = \bigcup_{e\in\mathcal{E}_h} e,$$

can be stored in two ways: as a $d_2 \times N_{\mathrm{fc}}$ matrix, whose $e$-th column stores the coefficients of the function on the face $e$, or as a $d_2 N_{\mathrm{fc}}$ column vector, with blocks of $d_2$ elements corresponding to the faces.

**Warning.** Similarly to what we do in $\mathcal{P}_k(\widehat{K})$, for two dimensional functions we will be using a Dubiner basis, that is orthogonal in $\check{K}_2 := 2\widehat{K}_2 - \mathbf{1}$. Therefore, at each evaluation of basis functions we have to apply the substitution

$$\widehat{D}_i(\mathbf{x}) = \check{D}_i(2\mathbf{x} - \mathbf{1}).$$

## 6.2 Testing on faces

Let $f : \Omega \to \mathbb{R}$. The aim of this function is the approximation of

$$\int_e f D_i^e \qquad i = 1, \ldots, d_2, \qquad e \in \mathcal{E}_h,$$

which will be stored as a $d_2 \times N_{\rm fc}$ matrix. This process is very similar to the one explained in Section 5.3 for testing on elements. We first write the approximations in the form

$$\int_e f D_i^e \approx |e| \sum_r \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r) f(\mathbf{q}_r^e), \qquad \text{where} \qquad \mathbf{q}_r^e = \boldsymbol{\phi}_e(\widehat{\mathbf{q}}_r).$$

To evaluate $f$ at all quadrature points, we start by constructing three $3 \times N_{\rm fc}$ matrices $\mathrm{X}^{\mathcal{E}}$, $\mathrm{Y}^{\mathcal{E}}$, and $\mathrm{Z}^{\mathcal{E}}$, with the respective coordinates of the three vertices of each of the faces. If $\Xi$ is the $N_{\rm qd2} \times 3$ matrix with the barycentric coordinates of all quadrature points (the first three columns of the matrix where we store the quadrature formula), then

$$\mathrm{X} = \Xi\, \mathrm{X}^{\mathcal{E}}, \qquad \mathrm{Y} = \Xi\, \mathrm{Y}^{\mathcal{E}}, \qquad \mathrm{Z} = \Xi\, \mathrm{Z}^{\mathcal{E}}$$

are $N_{\rm qd2} \times N_{\rm fc}$ matrices with the coordinates of all quadrature points on the faces, as mapped from the reference element with the intrinsic parametrization of each element. With the given data structure, this construction is easily accomplished. For instance:

```
x=T.coordinates(:,1); x=formula(:,1:3)*x(T.faces(:,1:3)');
```

(Recall that the last column of `T.faces` contains information about the location of the face in the interior domain, Dirichlet boundary or Neumann boundary.) If

$$\mathrm{D}_{rj} = \widehat{D}_j(\widehat{\mathbf{q}}_r), \qquad r = 1, \ldots, N_{\rm qd}, \quad j = 1, \ldots, d_2,$$

$\boldsymbol{\varpi}$ is the column vector with the weights of the quadrature formula, and **area** is the column vector with the areas of the elements, then the result is just

$$\mathbf{area} \odot \Big( (\boldsymbol{\varpi} \odot \mathrm{D})^\top f(\mathrm{X}, \mathrm{Y}, \mathrm{Z}) \Big),$$

reshaped as a three dimensional array.

**Implementation notes.** For input/output, see `testElem.m` in Section 5.3

```
function Ints=testFaces(f,T,k,formula)

%{Int1,Int2,...}=testFaces({f1,f2,...},T,k,formula)
%            Int=testFaces({f1},T,k,formula)
%
%Input:
%  {f1,f2,...}: each cell is a vectorized function of three variables
%            T: expanded tetrahedrization
%      formula: quadrature formula in 2d (N x 4 matrix)
%            k: polynomial
%
%Output:
%  {Int1,Ints2,...}: each cell is a d2 x Nfaces matrix
%                    ( \int_e f{l} D_i^e )
%
%Last modified: March 14, 2013

x=T.coordinates(:,1); x=formula(:,1:3)*x(T.faces(:,[1 2 3])');
```

```
y=T.coordinates(:,2); y=formula(:,1:3)*y(T.faces(:,[1 2 3])');
z=T.coordinates(:,3); z=formula(:,1:3)*z(T.faces(:,[1 2 3])');

DB=dubiner2d(2*formula(:,2)-1,2*formula(:,3)-1,k);
DB=bsxfun(@times,formula(:,4),DB);

nInts=size(f,2);
if nInts==1
    Ints=bsxfun(@times,T.area',DB'*f(x,y,z));
else
    Ints=cell(1,nInts);
    for n=1:nInts
        ff=f{n};
        Ints{n}=bsxfun(@times,T.area',DB'*ff(x,y,z));
    end
end
return
```

## 6.3 The penalization parameter $\tau$

Piecewise constant functions on the boundaries of the elements,

$$\xi \in \mathcal{P}_0(\partial\mathcal{T}_h) := \prod_{K\in\mathcal{T}_h} \prod_{e\in\mathcal{E}(K)} \mathcal{P}_0(e),$$

will be stored as $4\times N_{\rm elt}$ matrices and denoted $\xi_\ell^K$. These functions can be double valued on internal faces. Some simple but relevant piecewise constant functions, that can be taken from geometric information, are

```
T.area(T.facebyele')
nx=T.normals(:,[1 4 7 10])';
ny=T.normals(:,[2 5 8 11])';
nz=T.normals(:,[3 6 9 12])';
```

The first one has been denoted so far $e_\ell^K$. The one with components of the normal vectors will be denoted $n_\star \in \mathcal{P}_0(\partial\mathcal{T}_h)$, for $\star \in \{x,y,z\}$.
An important part of the HDG scheme is a penalization function $\tau \in \mathcal{P}_0(\partial\mathcal{T}_h)$. Currently, the code supports three choices: constant $\tau \equiv 1$, the first face choice

$$\tau_1^K = 1, \quad K \in \mathcal{T}_h, \qquad \text{and} \qquad \tau_\ell^K = 0, \quad \ell \in \{2,3,4\}, \quad K \in \mathcal{T}_h,$$

and the one random face choice (using a discrete uniform random distribution to choose one face on each element)

$$\tau_\ell^K = \delta_{\ell(K)}^K \qquad \ell(K) \in \{1,2,3,4\}.$$

```
function tau=createTau3d(Nelts,option)

% tau=createTau3d(Nelts,option)
%
% Input:
%    Nelts     : number of elements
%    option    : (1) tau = 1
%                (2) SFHDG on the first face
%                (3) SFHDG on a random face
% Output:
%    tau       : 4 x Nelts
%
% Last modified: March 14, 2013
```

```
switch option
    case 1
        tau=ones(4,Nelts);
    case 2
        tau=[ones(1,Nelts);...
             zeros(3,Nelts)];
    case 3
        where=ceil(4*rand(1,Nelts));
        tau=sparse(where,1:Nelts,1);
        tau=full(tau);
end
return
```

## 6.4 Three types of surface matrices

### Type (a)

The aim of this section is the computation of

$$\int_{\partial K} P_i^K P_j^K \qquad i,j=1,\ldots,d_3, \qquad K\in\mathcal{T}_h,$$

and its storage as a $d_3\times d_3\times N_{\rm elt}$ array. Using a quadrature formula of sufficiently high order, we can write

$$\int_{\partial K} P_i^K P_j^K = \sum_{\ell=1}^4 \tau_\ell^K |e_\ell^K| \sum_r \widehat{P}_i(\widehat{\mathbf{q}}_r^\ell)\varpi_r \widehat{P}_j(\widehat{\mathbf{q}}_r^\ell).$$

We first compute four $N_{\rm qd}\times d_3$ matrices

$$\mathrm{P}_{ri}^\ell = \widehat{P}_i(\widehat{\mathbf{q}}_r^\ell), \qquad r=1,\ldots,N_{\rm qd2}, \quad i=1,\ldots,d_3, \qquad \ell\in\{1,2,3,4\},$$

and then

$$\sum_{\ell=1}^4 \mathbf{t}_\ell^\top \otimes \Big( (\boldsymbol{\varpi}\odot \mathrm{P}^\ell)^\top \mathrm{P}^\ell\Big), \qquad \mathbf{t}_\ell^\top = \mathrm{row}(\mathrm{T},\ell), \qquad \mathrm{T}_K^\ell := \tau_K^\ell |e_\ell^K|.$$

The result is a $d_3\times(d_3N_{\rm elt})$ matrix that is then reshaped as a $d_3\times d_3\times N_{\rm elt}$ array.

### Type (b)

The second group of matrices is

$$\tau_\ell^K \int_{e_\ell^K} D_i^{e_\ell^K} D_j^{e_\ell^K} \qquad \begin{array}{c} i=1,\ldots,d_2 \\ j=1,\ldots,d_3 \end{array} \qquad \begin{array}{c} K\in\mathcal{T}_h \\ \ell\in\{1,2,3,4\}, \end{array}$$

to be stored as the diagonal blocks of a block-diagonal $4d_2\times 4d_2\times N_{\rm elt}$ array. Using quadrature and the fact that we are free to choose the parametrization $\boldsymbol{\phi}_e$ when $e=e_\ell^K$, we arrive at

$$\int_{e_\ell^K} \tau D_i^{e_\ell^K} D_j^{e_\ell^K} = |e_\ell^K|\tau_\ell^K \sum_r \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r)\widehat{D}_j(\widehat{\mathbf{q}}_r),$$

so the $\ell$-th block is

$$\mathbf{t}_\ell^\top \otimes \Big( (\boldsymbol{\varpi}\odot \mathrm{P})^\top \mathrm{P}\Big).$$

**Type (c)**

The third group of boundary matrices produces (simultaneously) matrices

$$\frac{\xi_K^\ell}{|e_\ell^K|}\int_{e_\ell^K} D_i^{e_\ell^K} P_i^K, \qquad \begin{array}{l} i=1,\ldots,d_2 \\ j=1,\ldots,d_3 \end{array} \qquad \begin{array}{c} K\in\mathcal{T}_h \\ \ell\in\{1,2,3,4\} \end{array}$$

that will be stored as $4d_2\times d_3\times N_{\rm elt}$ arrays, in blocks of $d_2\times d_3\times N_{\rm elt}$ arrays corresponding to the four values of $\ell$. Here $\xi$ is a piecewise constant function. In practice we need

$$\xi_\ell^K = \mathrm{T}_\ell^K = \tau_\ell^K |e_\ell^K|, \qquad \xi_\ell^K = n_{\ell,\star}^K \qquad \star\in\{x,y,z\},$$

where $n_{\ell,\star}^K$ is the $\star$ component of the non-normalized outward normal vector on $e_\ell^K$. Using the notation of Section 3.3 and a quadrature formula of sufficiently high order, the integrals are then

$$\begin{aligned}\frac{\xi_K^\ell}{|e_\ell^K|}\int_{e_\ell^K} D_i^{e_\ell^K} P_i^K &= \xi_\ell^K \sum_r \widehat{D}_i(F_{\mathrm{perm}(K,\ell)}(\widehat{\mathbf{q}}_r))\varpi_r \widehat{P}_j(\widehat{\mathbf{q}}_r^\ell) \\ &= \sum_{\mu=1}^6 \xi_\ell^K \mathbf{1}_{\mathrm{perm}(K,\ell)=\mu}\Big(\sum_r \widehat{D}_i(F_\mu(\widehat{\mathbf{q}}_r)\varpi_r \widehat{P}_j(\widehat{\mathbf{q}}_r^\ell)\Big)\end{aligned}$$

Let then $\boldsymbol{\xi}_\mu\in\mathcal{P}_0(\partial\mathcal{T}_h)$ be given by

$$\xi_{\ell,\mu}^K = \xi_\ell^K \mathbf{1}_{\mathrm{perm}(K,\ell)=\mu} = \begin{cases} \xi_\ell^K & \text{if } \mathrm{perm}(K,\ell)=\mu, \\ 0, & \text{otherwise.}\end{cases}$$

Then, we just need to compute

$$\sum_{\mu=1}^6 \boldsymbol{\xi}_{\ell,\mu}^\top \otimes \big((\boldsymbol{\varpi}\odot \mathrm{D}^\mu)^\top \mathrm{P}^\ell\big), \qquad \boldsymbol{\xi}_{\ell,\mu}^\top = \mathrm{row}(\boldsymbol{\xi}_\mu,\ell),$$

where

$$\begin{aligned} \mathrm{P}_{ri}^\ell &= \widehat{P}_i(\widehat{\mathbf{q}}_r^\ell), & r&=1,\ldots,N_{\rm qd2}, & i&=1,\ldots,d_3, \\ \mathrm{D}_{ri}^\mu &= \widehat{D}_i(F_\mu(\widehat{\mathbf{q}}_r)), & r&=1,\ldots,N_{\rm qd2}, & i&=1,\ldots,d_2. \end{aligned}$$

**Implementation notes.** The matrices $(\boldsymbol{\varpi}\odot\mathrm{D}^\mu)^\top\mathrm{P}^\ell$ are stored in a single $6d_2\times 4d_3$ array. Two local anonymous functions are defined for easy access to local degrees of freedom. The `block2` and `block3` functions produce the lists

$$(\ell-1)d_2+\{1,2,\ldots,d_2\} \qquad \text{and} \qquad (\ell-1)d_3+\{1,2,\ldots,d_3\}$$

respectively. These lists allow for easy location of blocks in the $6d_2\times 4d_3$ array that stores them.

```
function [tauPP,tauDP,nxDP,nyDP,nzDP,tauDD]=matricesFace(T,tau,k,formula)

%[tauPP,tauDP,nxDP,nyDP,nzDP,tauDD]=matricesFace(T,tau,k,formula)
%
%Input:
%             T: expanded terahedrization
%           tau: penalization parameter for HDG (Nelts x 4)
%             k: polynomial degree
%       formula: quadrature formula in 2d (N x 4 matrix)
%Output:
%        tauPP :   d3 x d3    x Nelts, with <tau P_i,P_j>_{\partial K}
%        tauDP : 4*d2 x d3    x Nelts, with <tau D_i,P_j>_e, e\in E(K)
%          nxDP : 4*d2 x d3    x Nelts, with <nx D_i,P_j>_e, e\in E(K)
%          nyDP : 4*d2 x d3    x Nelts, with <ny D_i,P_j>_e, e\in E(K)
%          nzDP : 4*d2 x d3    x Nelts, with <nz D_i,P_j>_e, e\in E(K)
%         tauDD : 4*d2 x 4*d2 x Nelts, block diag <tau D_i, D_j>_e, e\in E(K)
%
%Last modified: March 14, 2013

Nelts=size(T.elements,1);
Nnodes=size(formula,1);
TauArea=T.area(T.facebyele').*tau;    % 4 x Nelts
T.perm=T.perm';                       % 4 x Nelts

d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);

s=formula(:,2);
t=formula(:,3);
weights=formula(:,4);

% Computation <tau*Pi,Pj>

O=zeros(size(s));
points3d=[s,t,O;...
          s,O,t;...
          O,s,t;...
          s,t,1-s-t];

pb=dubiner3d(2*points3d(:,1)-1,2*points3d(:,2)-1,2*points3d(:,3)-1,k);  %4*Nnodes x d3
pbweights=bsxfun(@times,[formula(:,4);...
                         formula(:,4);...
                         formula(:,4);...
                         formula(:,4)],pb);
pbpb1=pbweights(1:Nnodes,:)'*pb(1:Nnodes,:);
pbpb2=pbweights(Nnodes+1:2*Nnodes,:)'*pb(Nnodes+1:2*Nnodes,:);
pbpb3=pbweights(2*Nnodes+1:3*Nnodes,:)'*pb(2*Nnodes+1:3*Nnodes,:);
pbpb4=pbweights(3*Nnodes+1:4*Nnodes,:)'*pb(3*Nnodes+1:4*Nnodes,:);

tauPP=kron(TauArea(1,:),pbpb1)+kron(TauArea(2,:),pbpb2)...
      +kron(TauArea(3,:),pbpb3)+kron(TauArea(4,:),pbpb4);
tauPP=reshape(tauPP,[d3,d3,Nelts]);

% Computation <alpha*D,P>, alpha=tau,nx,ny,nz,

pb=[pb(1:Nnodes,:),pb(Nnodes+1:2*Nnodes,:),...
    pb(2*Nnodes+1:3*Nnodes,:),pb(3*Nnodes+1:4*Nnodes,:)];  % Nnodes x 4*d3
points2d=[s,t;...
          t,s;...
          1-s-t,s;...
          s,1-s-t;...
          t,1-s-t;...
          1-s-t,t];
db=dubiner2d(2*points2d(:,1)-1,2*points2d(:,2)-1,k);    % 6*Nnodes x d2
db=[db(1:Nnodes,:),db(Nnodes+1:2*Nnodes,:),...
    db(2*Nnodes+1:3*Nnodes,:),db(3*Nnodes+1:4*Nnodes,:),...
    db(4*Nnodes+1:5*Nnodes,:),db(5*Nnodes+1:6*Nnodes,:)]; % Nnodes x 6*d2
db=bsxfun(@times,weights,db);
allproducts=db'*pb;             %6*d2 x 4*d3

block2=@(x) (1+(x-1)*d2):(x*d2);
block3=@(x) (1+(x-1)*d3):(x*d3);

tauDP=zeros(4*d2,d3*Nelts);
nxDP=zeros(4*d2,d3*Nelts);
nyDP=zeros(4*d2,d3*Nelts);
nzDP=zeros(4*d2,d3*Nelts);
for l=1:4
```

```
    Nx=T.normals(:,3*(l-1)+1)';
    Ny=T.normals(:,3*(l-1)+2)';
    Nz=T.normals(:,3*(l-1)+3)';
    for mu=1:6
        taumu=TauArea(l,:).*(T.perm(l,:)==mu);
        tauDP(block2(l),:)=tauDP(block2(l),:)+...
            kron(taumu,allproducts(block2(mu),block3(l)));

        nxmu=Nx.*(T.perm(l,:)==mu);
        nxDP(block2(l),:)=nxDP(block2(l),:)+...
            kron(nxmu,allproducts(block2(mu),block3(l)));

        nymu=Ny.*(T.perm(l,:)==mu);
        nyDP(block2(l),:)=nyDP(block2(l),:)+...
            kron(nymu,allproducts(block2(mu),block3(l)));

        nzmu=Nz.*(T.perm(l,:)==mu);
        nzDP(block2(l),:)=nzDP(block2(l),:)+...
            kron(nzmu,allproducts(block2(mu),block3(l)));
    end
end
tauDP=reshape(tauDP,[4*d2,d3,Nelts]);
nxDP=reshape(nxDP,[4*d2,d3,Nelts]);
nyDP=reshape(nyDP,[4*d2,d3,Nelts]);
nzDP=reshape(nzDP,[4*d2,d3,Nelts]);

% Computation tauDD

d=dubiner2d(2*s-1,2*t-1,k);
dweights=bsxfun(@times,d,weights);
dwd=dweights'*d;
tauDD=zeros(4*d2,4*d2,Nelts);
for l=1:4
    tauDD(block2(l),block2(l),:)=reshape(kron(TauArea(l,:),dwd),...
                                         [d2,d2,Nelts]);
end
return
```

## 6.5 Another function for errors

Given $u:\Omega\to\mathbb{R}$ and $\widehat{u}_h\in M_h$ (stored as a $d_2\times N_{\rm fc}$ matrix), we want to compute

$$\|\widehat{u}_h-u\|_h:=\Big(\sum_{e\in\mathcal{E}_h}|e|\int_e|\widehat{u}_h-u|^2\Big)^{1/2}.$$

Note that

$$\sum_{e\in\mathcal{E}_h}|e|\int_e|\widehat{u}_h-u|^2\approx\sum_{e\in\mathcal{E}_h}\sum_r\varpi_r|\widehat{u}_h(\mathbf{q}_r^e)-u(\mathbf{q}_r^e)|^2|e|^2.$$

What is left is very similar to what was done in Section 5.6 to compute $L^2$ errors. We start computing three $N_{\rm qd2}\times N_{\rm fc}$ matrices $\mathrm{X},\mathrm{Y},\mathrm{Z}$, with the coordinates of all quadrature nodes $\mathbf{q}_r^e$, and the $N_{\rm qd2}\times d_2$ matrix

$$\mathrm{D}_{ri}:=\widehat{D}_i(\widehat{\mathbf{q}}_r)\qquad r=1,\ldots,N_{\rm qd2},\qquad i=1,\ldots,d_2.$$

If U is the $d_2\times N_{\rm fc}$ matrix with the coefficients of $\widehat{u}_h$, then

$$\mathrm{E}:=\mathrm{DU}-u(\mathrm{X},\mathrm{Y},\mathrm{Z})\qquad \mathrm{E}_{re}:=\widehat{u}_h(\mathbf{q}_r^e)-u(\mathbf{q}_r^e)$$

are the pointwise errors and the total error is just

$$\Big(\sum_{r,e}\varpi_r\mathrm{E}_{re}^2|e|^2\Big)^{1/2}.$$

```
function error=errorFaces(T,p,ph,k,formula)

%error =errorFaces(T,p,ph,k,formula)
%
%Input:
%          T: expanded tetrahedrization
%          p: vectorized function of three variables
%         ph: Pk function on skeleton (d2 x Nfaces)
%          k: polynomial degree
%    formula: quadrature formula in 2d (N x 4 matrix)
%
%Output:
%      error: \| p - ph \|_{h,\partial T_h}
%
%Last modified: March 14, 2013

x=T.coordinates(:,1);x=formula(:,[1 2 3])*x(T.faces(:,[1 2 3])');
y=T.coordinates(:,2);y=formula(:,[1 2 3])*y(T.faces(:,[1 2 3])');
z=T.coordinates(:,3);z=formula(:,[1 2 3])*z(T.faces(:,[1 2 3])');
p=p(x,y,z);  %Nnodes x Nfaces

DB=dubiner2d(2*formula(:,2)-1,2*formula(:,3)-1,k);
ph=DB*ph;                          %Nnodes x Nfaces

error=sqrt(formula(:,4)'*(p-ph).^2*(T.area).^2);
return
```

## 7 Local solvers

The local solvers that we next defined are related to the pair of first order PDEs

$$\kappa^{-1}\boldsymbol{q} + \nabla u = 0, \qquad \text{and} \qquad \nabla\cdot\boldsymbol{q} + cu = f \qquad \text{in } \Omega.$$

### 7.1 Matrices and bilinear forms

In order to recognize the matrices that we have computed with terms in the bilinear forms of the HDG method, we need some notation. We will write

$$(u,v)_K := \int_K u\,v, \qquad \langle u,v\rangle_{\partial K} := \int_{\partial K} u\,v$$

and we will consider the space

$$\mathcal{R}_k(\partial K) := \prod_{e\in\mathcal{E}(K)} \mathcal{P}_k(e), \qquad \dim \mathcal{R}_k(\partial K) = 4d_2.$$

The degrees of freedom for this last space are organized by taking one face at a time in the order they are given by `T.facebyele`.
For (non-symmetric) bilinear forms we will use the convention that the bilinear form $b(u,v)$ is related to the matrix $b(U_j,V_i)$, where $\{U_j\}$ is a basis of the space of $u$ and $\{V_i\}$ is a basis of the space for $v$. This is equivalent to saying that the unknown will always be placed as the left-most argument in the bilinear form and the test function will occupy the right-most location.

**Volume terms.** We start by computing mass matrices associated to two functions ($\kappa^{-1}$ and $c$), and the three convection matrices:

$$\mathrm{M}^K_{\kappa^{-1}}, \qquad \mathrm{M}^K_c, \qquad \mathrm{C}^K_x, \qquad \mathrm{C}^K_y, \qquad \mathrm{C}^K_z,$$

where

$$(\mathrm{M}^K_m)_{ij} = \int_K m\, P^K_i P^K_j, \qquad (\mathrm{C}^K_\star)_{ij} = \int_K P^K_i \partial_\star P^K_j.$$

Each of these matrices is $d_3 \times d_3 \times N_{\mathrm{elt}}$. They correspond to the bilinear forms

$$(m\, u_h, v_h)_K \qquad (\partial_\star u_h, v_h)_K \qquad u_h, v_h \in \mathcal{P}_k(K).$$

**Surface terms.** We next compute all matrices related to integrals on interfaces:

$$\tau\mathrm{PP}^K, \qquad \tau\mathrm{DP}^K, \qquad n_x\mathrm{DP}^K, \qquad n_y\mathrm{DP}^K, \qquad n_z\mathrm{DP}^K, \qquad \tau\mathrm{DD}^K.$$

The first of these arrays is $d_3 \times d_3 \times N_{\mathrm{elt}}$, the next four are $4d_2 \times d_3 \times N_{\mathrm{elt}}$ and the last one is $4d_2 \times 4d_2 \times N_{\mathrm{elt}}$. The first matrix and associated bilinear form is

$$\tau\mathrm{PP}^K_{ij} = \int_{\partial K} \tau\, P^K_i P^K_j, \qquad \langle \tau u_h, v_h\rangle_{\partial K} \qquad u_h, v_h \in \mathcal{P}_k(K).$$

The second one corresponds to the bilinear forms

$$\langle \tau u_h, \widehat{v}_h\rangle_{\partial K} \qquad u_h \in \mathcal{P}_k(K), \quad \widehat{v}_h \in \mathcal{R}_k(\partial K)$$

or equivalently to

$$\langle \tau u_h, \widehat{v}_h\rangle_e \qquad u_h \in \mathcal{P}_k(K), \quad \widehat{v}_h \in \mathcal{P}_k(e), \qquad e \in \mathcal{E}(K).$$

The matrices associated to the components of the normal vector $\boldsymbol{\nu} = (\nu_x, \nu_y, \nu_z)$ are related to the bilinear forms

$$\langle \nu_\star u_h, \widehat{v}_h\rangle_{\partial K} \qquad u_h \in \mathcal{P}_k(K), \quad \widehat{v}_h \in \mathcal{R}_k(\partial K), \qquad \star \in \{x, y, z\}.$$

The last matrix corresponds to

$$\langle \tau \widehat{u}_h, \widehat{v}_h\rangle_{\partial K} \qquad \widehat{u}_h, \widehat{v}_h \in \mathcal{R}_k(\partial K)$$

and is therefore block diagonal.
Finally we compute the vectors of tests of $f$ with the basis elements of $\mathcal{P}_k(K)$: $\mathbf{f}^K \in \mathbb{R}^{d_3}$.

## 7.2 Matrices related to local solvers

The $4d_3 \times 4d_3 \times N_{\mathrm{elt}}$ array with slices

$$\mathbb{A}^K_1 := \left[\begin{array}{cccc} \mathrm{M}^K_{\kappa^{-1}} & \mathrm{O} & \mathrm{O} & -(\mathrm{C}^K_x)^\top \\ \mathrm{O} & \mathrm{M}^K_{\kappa^{-1}} & \mathrm{O} & -(\mathrm{C}^K_y)^\top \\ \mathrm{O} & \mathrm{O} & \mathrm{M}^K_{\kappa^{-1}} & -(\mathrm{C}^K_z)^\top \\ \mathrm{C}^K_x & \mathrm{C}^K_y & \mathrm{C}^K_y & \mathrm{M}^K_c + \tau\mathrm{PP}^K \end{array}\right],$$

is the matrix representation of the bilinear form $\Big(\mathcal{P}_k(K)^3 \times \mathcal{P}_k(K)\Big) \times \Big(\mathcal{P}_k(K)^3 \times \mathcal{P}_k(K)\Big) \to \mathbb{R}$:

$$\begin{array}{lcl} (\boldsymbol{q}_h, u_h) \quad , \quad (\boldsymbol{r}_h, w_h) & \longmapsto & (\kappa^{-1}\boldsymbol{q}_h, \boldsymbol{r}_h)_K - (u_h, \nabla\cdot\boldsymbol{r}_h)_K \\ & & +(\nabla\cdot\boldsymbol{q}_h, w_h)_K + (c\, u_h, w_h)_K + \langle \tau u_h, v_h\rangle_{\partial K} \end{array}$$

The $4d_3 \times 4d_2 \times N_{\mathrm{elt}}$ array with slices

$$\mathbb{A}^K_2 := \left[\begin{array}{c} (n_x\mathrm{DP}^K)^\top \\ (n_y\mathrm{DP}^K)^\top \\ (n_z\mathrm{DP}^K)^\top \\ -(\tau\mathrm{DP}^K)^\top \end{array}\right]$$

is the matrix representation of the bilinear form $\mathcal{R}_k(\partial K) \times \Big(\mathcal{P}_k(K)^3 \times \mathcal{P}_k(K)\Big) \to \mathbb{R}$:

$$\widehat{u}_h \quad , \quad (\boldsymbol{r}_h, w_h) \quad \longmapsto \quad \langle \widehat{u}_h, \boldsymbol{r}_h \cdot \boldsymbol{\nu} \rangle_{\partial K} - \langle \tau \widehat{u}_h, w_h \rangle_{\partial K}.$$

If $\widehat{u}_h \in M_h$ is known, we can solve the local problems looking for $\boldsymbol{q}_h \in \boldsymbol{V}_h := W_h^3$ and $u_h \in W_h$ such that

$$\begin{aligned} (\kappa^{-1}\boldsymbol{q}_h, \boldsymbol{r}_h)_K - (u_h, \nabla \cdot \boldsymbol{r}_h)_K + \langle \widehat{u}_h, \boldsymbol{r}_h \cdot \boldsymbol{\nu} \rangle_{\partial K} &= 0 \qquad \forall \boldsymbol{r}_h \in \boldsymbol{V}_h \\ (\nabla \cdot \boldsymbol{q}_h, w_h)_K + (c\, u_h, w_h)_K + \langle \tau(u_h - \widehat{u}_h), w_h \rangle_{\partial K} &= 0 \qquad \forall w_h \in W_h. \end{aligned}$$

If $\widehat{u}_h|_{\partial K} \in \mathbb{R}_k(\partial K)$ is represented with a vector $\mathbf{u}_{\partial K} \in \mathbb{R}^{4d_2}$, then the matrix representation of this local solution is

$$\begin{bmatrix} \mathbf{q}_K \\ \mathbf{u}_K \end{bmatrix} = -(\mathbb{A}_1^K)^{-1}\mathbb{A}_2^K \mathbf{u}_{\partial K} \in \mathbb{R}^{3d_3+d_3}$$

If we consider the $4d_3 \times N_{\rm elt}$ matrix with columns

$$\mathbb{A}_f^K := \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{f}^K \end{bmatrix},$$

then

$$\begin{bmatrix} \mathbf{q}_K \\ \mathbf{u}_K \end{bmatrix} = (\mathbb{A}_1^K)^{-1}\mathbb{A}_f^K$$

are the coefficients of the solution of

$$\begin{aligned} (\kappa^{-1}\boldsymbol{q}_h, \boldsymbol{r}_h)_K - (u_h, \nabla \cdot \boldsymbol{r}_h)_K &= 0 \qquad \forall \boldsymbol{r}_h \in \boldsymbol{V}_h \\ (\nabla \cdot \boldsymbol{q}_h, w_h)_K + (c\, u_h, w_h)_K + \langle \tau u_h, w_h \rangle_{\partial K} &= (f, w_h)_K \qquad \forall w_h \in W_h. \end{aligned}$$

## 7.3 Flux operators

Consider now the $4d_2 \times 4d_3 \times N_{\rm elt}$ array with slices

$$\mathbb{A}_3^K := \begin{bmatrix} n_x \mathrm{DP}^K & n_y \mathrm{DP}^K & n_z \mathrm{DP}^K & \tau \mathrm{DP}^K \end{bmatrix},$$

the $4d_2 \times 4d_2 \times N_{\rm elt}$ array with slices

$$\mathbb{C}^K := \mathbb{A}_3^K (\mathbb{A}_1^K)^{-1} \mathbb{A}_2^K + \tau \mathrm{DD}^K$$

and the $4d_2 \times N_{\rm elt}$ matrix with columns

$$\mathbb{C}_f^K := \mathbb{A}_3^K (\mathbb{A}_1^K)^{-1} \mathbb{A}_f^K.$$

The meaning of these matrices can be made clear by looking at boundary fluxes.

**Flux due to $\widehat{u}_h$.** Given $(\boldsymbol{q}_h, u_h, \widehat{u}_h) \in \boldsymbol{V}_h \times W_h \times M_h$, the HDG is based on the construction of the function

$$\boldsymbol{q}_h \cdot \boldsymbol{\nu} + \tau(u_h - \widehat{u}_h) : \partial K \to \mathbb{R}.$$

Instead of this quantity, we pay attention to how it creates a linear form

$$\mathcal{R}_k(\partial K) \ni \widehat{v}_h \longmapsto -\langle \boldsymbol{q}_h \cdot \boldsymbol{\nu} + \tau(u_h - \widehat{u}_h), \widehat{v}_h \rangle_{\partial K} = -\langle \boldsymbol{q}_h \cdot \boldsymbol{\nu} + \tau u_h, \widehat{v}_h \rangle_{\partial K} + \langle \tau \widehat{u}_h, \widehat{v}_h \rangle_{\partial K},$$

whose matrix representation is

$$-\mathbb{A}_3^K \begin{bmatrix} \mathbf{q}_K \\ \mathbf{u}_K \end{bmatrix} + (\tau \mathrm{DD})^K \mathbf{u}_{\partial K} = \mathbb{A}_3^K (\mathbb{A}_1^K)^{-1} \mathbb{A}_2^K \mathbf{u}_{\partial K} + (\tau \mathrm{DD})^K \mathbf{u}_{\partial K} = \mathbb{C}^K \mathbf{u}_{\partial K}$$

**Flux due to sources.** If we take $(\boldsymbol{q}_h, u_h)$ by solving the local equations due to sources and consider the linear form

$$\mathcal{R}_k(\partial K) \ni \widehat{v}_h \longmapsto \langle \boldsymbol{q}_h \cdot \boldsymbol{\nu} + \tau u_h, \widehat{v}_h \rangle_{\partial K},$$

($\widehat{u}_h$ does not appear here), then the matrix representation of this is

$$\mathbb{A}_3^K \begin{bmatrix} \mathbf{q}_K \\ \mathbf{u}_K \end{bmatrix} = \mathbb{A}_3^K (\mathbb{A}_1^K)^{-1} \mathbb{A}_f^K = \mathbb{C}_f^K.$$

If we solve the local problems ($\widehat{u}_h \in M_h$ and $f$ are given)

$$\begin{aligned} (\kappa^{-1}\boldsymbol{q}_h, \boldsymbol{r}_h)_K - (u_h, \nabla \cdot \boldsymbol{r}_h)_K + \langle \widehat{u}_h, \boldsymbol{r}_h \cdot \boldsymbol{\nu} \rangle_{\partial K} &= 0 \qquad \forall \boldsymbol{r}_h \in \boldsymbol{V}_h \\ (\nabla \cdot \boldsymbol{q}_h, w_h)_K + (c\, u_h, w_h)_K + \langle \tau(u_h - \widehat{u}_h), w_h \rangle_{\partial K} &= (f, w_h)_K \qquad \forall w_h \in W_h, \end{aligned}$$

then the functional

$$\mathcal{R}_k(\partial K) \ni \widehat{v}_h \longmapsto \langle \boldsymbol{q}_h \cdot \boldsymbol{\nu} + \tau u_h, \widehat{v}_h \rangle_{\partial K},$$

has the matrix representation

$$\mathbb{C}^K \mathbf{u}_{\partial K} - \mathbb{C}_f^K,$$

where $\mathbf{u}_{\partial K}$ is the vector of degrees of freedom of $\widehat{u}_h|_{\partial K}$.

**Implementation notes.** Note that the Matlab expression `A3/A1*A2` corresponds to $A_3 A_1^{-1} A_2$ (no inversion of $A_1$ is required in the process). *The loop over elements can be parallelized in a very simple way.*

```
function [C,Cf,A1,A2,Af]=localsolvers3d(km,c,f,tau,T,k,formulas)

%[C,Cf,A1,A2,Af]=localsolvers3d(km,c,f,tau,T,k,{for1,for2,for3})
%
%Input:
%      km, c, f: vectorized functions of three variables
%           tau: penalization parameter for HDG (Nelts x 4)
%             T: expanded tetrahedrization
%             k: polynomial degree
%          for1: quadrature formula 3d (for mass matrix)
%          for2: quadrature formula 3d (for convection matrices)
%          for3: quadrature formula 2d
%
%Output:
%             C:   4*d2 x 4*d2 x Nelts
%            Cf:   4*d2 x Nelts
%            A1:   4*d3 x 4*d3 x Nelts
%            A2:   4*d3 x 4*d2 x Nelts
%            Af:   4*d3 x Nelts
%
%Last modified: Nov 29, 2012

Nelts=size(T.elements,1);
d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);

f=testElem(f,T,k,formulas{1});
Af=zeros(4*d3,Nelts);
Af(3*d3+1:4*d3,:)=f;

Mass=MassMatrix(T,{km,c},k,formulas{1});
Mk=Mass{1};Mc=Mass{2};
[Cx,Cy,Cz]=ConvMatrix(T,k,formulas{2});
[tauPP,tauDP,nxDP,nyDP,nzDP,tauDD]=matricesFace(T,tau,k,formulas{3});
```

```
A1=zeros(4*d3,4*d3,Nelts);
A2=zeros(4*d3,4*d2,Nelts);
C=zeros(4*d2,4*d2,Nelts);
Cf=zeros(4*d2,Nelts);

O=zeros(d3,d3,Nelts);
A1=[Mk        ,O        ,O        ,-permute(Cx,[2 1 3]);...
    O         ,Mk       ,O        ,-permute(Cy,[2 1 3]);...
    O         ,O        ,Mk       ,-permute(Cz,[2 1 3]);...
    Cx        ,Cy       ,Cz       ,Mc+tauPP];
A2=[permute(nxDP,[2 1 3]);...
    permute(nyDP,[2 1 3]);...
    permute(nzDP,[2 1 3]);...
    -permute(tauDP,[2 1 3])];

parfor i=1:Nelts
    C(:,:,i)=[nxDP(:,:,i) nyDP(:,:,i) nzDP(:,:,i) tauDP(:,:,i)]/A1(:,:,i)*A2(:,:,i)...
              +tauDD(:,:,i);
    Cf(:,i) =[nxDP(:,:,i) nyDP(:,:,i) nzDP(:,:,i) tauDP(:,:,i)]/A1(:,:,i)*Af(:,i);
end
return
```

## 8 Boundary conditions

### 8.1 Dirichlet boundary conditions

The discretization of the Dirichlet boundary condition is given by projecting the Dirichlet data $u_D$ on the space

$$M_h^{\mathrm{dir}} := \prod_{e\in\mathcal{E}_h^{\mathrm{dir}}} \mathcal{P}_k(e) \qquad \dim M_h^{\mathrm{dir}} = d_2 N_{\mathrm{dir}}.$$

We have to find values

$$u_j^e \quad j=1,\ldots,d_2, \qquad e\in\mathcal{E}_h^{\mathrm{dir}} \qquad \widehat{u}_h|_e = \sum_j u_j^e D_j^e$$

such that

$$\sum_j \left(\int_e D_i^e D_j^e\right) u_j^e = \int_e D_i^e \widehat{u}_h = \int_e D_i^e u_D, \qquad i=1,\ldots,d_2, \qquad e\in\mathcal{E}_h^{\mathrm{dir}}.$$

Using a quadrature rule, this is equivalent (up to quadrature error) to solving the system

$$|e|\sum_{j=1}^{d_2}\Big(\sum_r \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r)\widehat{D}_j(\widehat{\mathbf{q}}_r)\Big)u_j^e = |e|\sum_r \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r) u_D(\widehat{\mathbf{q}}_r), \qquad i=1,\ldots,d_2, \quad e\in\mathcal{E}_h^{\mathrm{dir}}.$$

If we compute three $N_{\mathrm{qd}}\times N_{\mathrm{dir}}$ matrices $\mathrm{X}^{\mathrm{dir}}$, $\mathrm{Y}^{\mathrm{dir}}$, $\mathrm{Z}^{\mathrm{dir}}$ and the basic $N_{\mathrm{qd}}\times d_2$ matrices on the reference element

$$\mathrm{D}_{ri} := \widehat{D}_i(\widehat{\mathbf{q}}_r)$$

then the entire computation is reduced to

$$\big((\boldsymbol{\varpi}\odot \mathrm{D})^\top \mathrm{D}\big)^{-1}(\boldsymbol{\varpi}\odot\mathrm{D})^\top u_D(\mathrm{X}^{\mathrm{dir}},\mathrm{Y}^{\mathrm{dir}},\mathrm{Z}^{\mathrm{dir}}).$$

### 8.2 Neumann boundary conditions

We will be imposing boundary conditions in the form

$$\kappa\nabla u\cdot\boldsymbol{\nu} = \boldsymbol{u}_N\cdot\boldsymbol{\nu} \qquad \text{or equivalently} \qquad -\boldsymbol{q}\cdot\boldsymbol{\nu} = \boldsymbol{u}_N\cdot\boldsymbol{\nu}.$$

This is done in this non-standard form in order to simplify the inclusion of complicated solutions. The aim of this part of the code is the computation of the $d_2 \times N_{\rm neu}$ matrix

$$\int_e (\boldsymbol{u}_N \cdot \boldsymbol{\nu}) D_i^e \qquad i = 1, \ldots, d_2, \qquad e \in \mathcal{E}_h^{\rm neu}.$$

Using a quadrature rule we can approximate

$$\int_e (\boldsymbol{u}_N \cdot \boldsymbol{\nu}) D_i^e \approx \sum_r \boldsymbol{u}_N(\mathbf{q}_r^e) \cdot \mathbf{n}^e\, \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r).$$

If $\boldsymbol{u}_N = (u_x, u_y, u_z)$ and $\mathbf{n}_\star$ are column vectors with the $\star$ components of the normal vectors of each of the $N_{\rm neu}$ Neumann faces, then, we just need to compute

$$\sum_{\star \in \{x,y,z\}} \mathbf{n}_\star^\top \odot \big( (\boldsymbol{\varpi} \odot \mathrm{D})^\top u_\star(\mathrm{X}^{\rm neu}, \mathrm{Y}^{\rm neu}, \mathrm{Z}^{\rm neu}) \big),$$

where $\mathrm{X}^{\rm neu}, \mathrm{Y}^{\rm neu}, \mathrm{Z}^{\rm neu}$ are $N_{\rm qd2} \times N_{\rm neu}$ matrices with the coordinates of all quadrature points on the Neumann faces.

```
function [uhd,qhn]=BC3d(uD,gx,gy,gz,T,k,formula)

%[uhd,qhn]=BC3d(uD,gx,gy,gz,T,k,formula)
%
% Input:
%          uD: Dirichlet data; vectorized function of three variables
%    gx,gy,gz: Neumann data (corresponds to kappa*grad(u))
%              vectorized functions of three variables
%           T: Full tetrahedrization
%           k:  polynomial degree
%     formula: quadrature formula in 2d (N x 4 matrix)
%
% Output:
%         uhd: d2 x Ndir,   Ndir, number of Dirichlet faces
%         qhn: d2 x Nneu,   Nneu, number of Neumann faces
%
% Last modified: March 21, 2013

x=T.coordinates(:,1);
y=T.coordinates(:,2);
z=T.coordinates(:,3);

%Dirichlet
xx=formula(:,[1 2 3])*x(T.dirichlet');  % Nnodes x Ndir
yy=formula(:,[1 2 3])*y(T.dirichlet');  % Nnodes x Ndir
zz=formula(:,[1 2 3])*z(T.dirichlet');  % Nnodes x Ndir
D=dubiner2d(2*formula(:,2)-1,2*formula(:,3)-1,k);  % Nnodes x d2
wD=bsxfun(@times,formula(:,4),D);             % Nnodes x d2
uhd=((wD'*D)\wD')*uD(xx,yy,zz);

%Neumann
x12=x(T.neumann(:,2))-x(T.neumann(:,1)); %x2-x1
y12=y(T.neumann(:,2))-y(T.neumann(:,1)); %y2-y1
z12=z(T.neumann(:,2))-z(T.neumann(:,1)); %z2-z1
x13=x(T.neumann(:,3))-x(T.neumann(:,1)); %x3-x1
y13=y(T.neumann(:,3))-y(T.neumann(:,1)); %y3-y1
z13=z(T.neumann(:,3))-z(T.neumann(:,1)); %z3-z1

Neu_normals=0.5*[y12.*z13-z12.*y13,...
                 z12.*x13-x12.*z13,...
                 x12.*y13-x13.*y12]; % Nneu x 3,  ||n^e||=|e|

xn=formula(:,[1 2 3])*x(T.neumann');  % Nnodes x Nneu
```

```
yn=formula(:,[1 2 3])*y(T.neumann');  % Nnodes x Nneu
zn=formula(:,[1 2 3])*z(T.neumann');  % Nnodes x Nneu

qhn=bsxfun(@times,Neu_normals(:,1)',wD'*gx(xn,yn,zn))+...
    bsxfun(@times,Neu_normals(:,2)',wD'*gy(xn,yn,zn))+...
    bsxfun(@times,Neu_normals(:,3)',wD'*gz(xn,yn,zn));  %d2 x Nneu
return
```

## 9 HDG

For the correct definition of the HDG method, we need to consider the space

$$M_h^{\mathrm{free}} := \{\widehat{v}_h \in M_h \,:\, \widehat{v}_h|_{\Gamma D} = 0\} = \prod_{e\in\mathcal{E}_h^{\mathrm{free}}} \mathcal{P}_k(e), \quad \text{where} \qquad \text{free} := \{1,\dots,N_{\mathrm{fc}}\}\setminus \text{dir}.$$

The HDG method looks for $\widehat{u}_h \in M_h$ such that

$$\langle \widehat{u}_h, \widehat{v}_h\rangle_{\Gamma_D} = \langle u_D, \widehat{v}_h\rangle_{\Gamma_D} \qquad \forall \widehat{v}_h \in M_h^{\mathrm{dir}}$$

(this is the discrete Dirichlet BC), and

$$-\sum_{K\in\mathcal{T}_h} \langle \boldsymbol{q}_h\cdot\boldsymbol{\nu} + \tau(u_h - \widehat{u}_h), \widehat{v}_h\rangle_{\partial K} = \langle \boldsymbol{u}_N\cdot\boldsymbol{\nu}, \widehat{v}_h\rangle_{\Gamma_N} \qquad \forall \widehat{v}_h \in M_h^{\mathrm{free}},$$

where $(\boldsymbol{q}_h, u_u) \in \boldsymbol{V}_h \times W_h$ is the solution of the local problems:

$$\begin{aligned}
(\kappa^{-1}\boldsymbol{q}_h, \boldsymbol{r}_h)_K - (u_h, \nabla\cdot\boldsymbol{r}_h)_K + \langle \widehat{u}_h, \boldsymbol{r}_h\cdot\boldsymbol{\nu}\rangle_{\partial K} &= 0 \qquad \forall \boldsymbol{r}_h \in \mathcal{P}_k(K)^3,\\
(\nabla\cdot\boldsymbol{q}_h, w_h)_K + (c\,u_h, w_h)_K + \langle \tau(u_h - \widehat{u}_h), w_h\rangle_{\partial K} &= (f, w_h)_K \qquad \forall w_h \in \mathcal{P}_k(K)
\end{aligned}$$

for all $K \in \mathcal{T}_h$. The global equation can be decomposed in two groups: for interior faces $e = K \cap \tilde{K}$, we can write

$$-\langle \boldsymbol{q}_K\cdot\boldsymbol{\nu}_K + \tau_K(u_K - \widehat{u}_e), \widehat{v}_e\rangle_e - \langle \boldsymbol{q}_{\tilde{K}}\cdot\boldsymbol{\nu}_{\tilde{K}} + \tau_{\tilde{K}}(u_{\tilde{K}} - \widehat{u}_e), \widehat{v}_e\rangle_e = 0 \qquad \forall \widehat{v}_e \in \mathcal{P}_k(e),$$

(this is equality of discrete fluxes across inter-element faces), and for Neumann faces $e \in \mathcal{E}_h^{\mathrm{neu}}$, $e \subset \partial K$, we have a discretized version of the Neumann BC

$$-\langle \boldsymbol{q}_K\cdot\boldsymbol{\nu}_K + \tau_K(u_K - \widehat{u}_e), \widehat{v}_e\rangle_e = \langle \boldsymbol{u}_N\cdot\boldsymbol{\nu}_e, \widehat{v}_e\rangle_e \qquad \forall \widehat{v}_e \in \mathcal{P}_k(e).$$

Since we are testing with the space $M_h^{\mathrm{free}}$, all integrals on Dirichlet faces are ignored after assembly, while values of $\widehat{u}_h$ on Dirichlet faces are substituted from the Dirichlet BC.

### 9.1 Assembly process

The local solvers produce a $4d_2 \times 4d_2 \times N_{\mathrm{elt}}$ array $\mathbb{C}$. We now use the `sparse` Matlab builder to assembly the global matrix. The degrees of freedom associated to face $e \in \{1,\dots,N_{\mathrm{fc}}\}$ are

$$\text{list}(e) = (e-1)d_2 + \{1,\dots,d_2\}.$$

The degrees of freedom associated to the faces of $K$ are thus

$$\text{dof}(K) := \{\text{list}(e_1^K), \text{list}(e_2^K), \text{list}(e_3^K), \text{list}(e_4^K)\}.$$

We then create two new $4d_2 \times 4d_2 \times N_{\mathrm{elt}}$ arrays

$$\text{Row}_{ij}^K = \text{dof}(K)_i \qquad \text{Col}_{ij}^K = \text{dof}(K)_j.$$

Note that the element $(i,j)$ of $\mathbb{C}^K$ has to be assembled at the location $(\text{Row}_{ij}^K, \text{Col}_{ij}^K) = (\text{dof}(K)_i, \text{dof}(K)_j)$. The assembly of the **source term**, given in the matrix $\mathbb{C}_f$, can be carried out using the `accumarray` command. The element $(\mathbb{C}_f^K)_i$ has to be added to the location $\text{dof}(K)_i$.

## 9.2 The global system

**1.** The assembly process of the $\mathbb{C}^K$ matrices produces an $N_{\rm fc}\times N_{\rm fc}$ matrix. Similarly, the accumulation of the $\mathbb{C}_f^K$ vectors produces a vector with $N_{\rm fc}$ components. On the Neumann components of this vector, we have to add a vector , with the tests of $\boldsymbol{u}_N\cdot\boldsymbol{\nu}$ on Neumann faces.

**2.** The value of the unknown on Dirichlet faces is substituted from the data given by the Dirichlet BC and the corresponding part of the system is send to the right hand side. The rows associated to the Dirichlet faces are eliminated from the system (this is the same process that is applied to Dirichlet BC in the Finite Element Method). The system is solved on the free degrees of freedom.

**3.** The solution of the resulting system is $\widehat{u}_h\in M_h$. Reconstruction of the other variables $(\boldsymbol{q}_h,u_h)$ is done by solving local problems. In matrix form, we have to solve on each $K\in\mathcal{T}_h$ the system

$$\mathbb{A}_1^K\left[\begin{array}{c}\mathbf{q}_K\\ \mathbf{u}_K\end{array}\right]=\mathbb{A}_f^K-\mathbb{A}_2^K\mathbf{u}_{\partial K}.$$

**Implementation notes.** The necessary quadrature formulas are brought in a cell array:

```
{formula1,formula2,formula3,formula4}
```

as follows:

- `formula1` is used for mass matrices; it is assumed to be of degree $3k$ at least; it is also used for all error computations
- `formula2` is used for convection matrices (with constant coefficients); it is assumed to be of degree $2k$ at least
- `formula3` is a 2-dimensional formula used for integrals on faces; it is assumed to be of degree $2k$ at least
- `formula4` is a higer order 2-dimensional formula used for errors

For input we bring in:

- the coefficients and source term: $\kappa^{-1},c,f$;
- the penalization parameter function $\tau$ in the form of a $N_{\rm elt}\times 4$ matrix; the choice of a matrix with all unit entries works fine
- the functions for the boundary conditions $u_D$ and $(g_x,g_y,g_z)$ (recall that we impose a boundary condition $-\boldsymbol{q}\cdot\boldsymbol{\nu}=\kappa\nabla u\cdot\boldsymbol{\nu}=\boldsymbol{g}\cdot\boldsymbol{\nu}$)
- an optional parameter:
  - if the parameter is not present or its value is zero, then the problem is solved and the solution is exported in five variables corresponding to matrix forms for $u_h$, the components of $\boldsymbol{q}_h$ and $\widehat{u}_h$
  - if the parameter is not zero, then the system is not solved, but the elements of the system (the matrix, the right hand side and the lists of degrees of freedom (not faces) that are free and Dirichlet) are exported, as well as the local solvers $\mathbb{A}_1$, $\mathbb{A}_2$ and $\mathbb{A}_f$, that are needed to reconstruct the solution.

```
function [Uh,Qxh,Qyh,Qzh,Uhat,system,solvers]=HDG3d(km,c,f,tau,T,k,formulas,uD,gx,gy,gz,varargin)


%[Uh,Qxh,Qyh,Qzh,Uhat]=HDG3d(km,c,f,tau,T,k,formulas,uD,gx,gy,gz)
%[Uh,Qxh,Qyh,Qzh,Uhat]=HDG3d(km,c,f,tau,T,k,formulas,uD,gx,gy,gz,0)
%[¬,¬,¬,¬,¬,system,solvers]=HDG3d(km,c,f,tau,T,k,formulas,uD,gx,gy,gz,1)
%
%Input:
%        km   : vectorized function (kappa^{-1}; kappa=diffusion parameter)
%        c    : vectorized function (reaction parameter)
%        f    : vectorized function (source term)
%        tau  : penalization parameter for HDG (Nelts x 4)
%        T    : expanded tetrahedrization
%        k    : polynomial degree
%     formulas: {for1,for2,for3}
%               (quadrature formulas as used by localsolvers3d)
%        uD   : Dirichlet data; vectorized function
%    gx,gy,gz : Neumann data (corresponds to kappa*grad(u)); vectorized fns
%
%Output:
%      Uh     : d3 x Nelts,   matrix with uh
%      Qxh    : d3 x Nelts,   matrix with qxh
%      Qyh    : d3 x Nelts,   matrix with qyh
%      Qzh    : d3 x Nelts,   matrix with qzh
%      Uhat   : d2 x Nelts    matrix with uhhat
%    system   : {HDGmatrix,HDGrhs,list of free d.o.f.,list of dir d.o.f.}
%    solvers  : {A1,A2,Af}    local solvers
%
%Last modified: April 11, 2013

if nargin==12
    export=varargin{1};
else
    export=0;
end

d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);
block3=@(x) (1+(x-1)*d3):(x*d3);
Nelts =size(T.elements,1);
Nfaces=size(T.faces,1);
Ndir  =size(T.dirichlet,1);
Nneu  =size(T.neumann,1);

%Matrices for assembly process

face=T.facebyele';     % 4 x Nelts
face=(face(:)-1)*d2;   % First degree of freedom of each face by element
face=bsxfun(@plus,face,1:d2);     %4*Nelts x d2 (d.o.f. for each face)
face=reshape(face',4*d2,Nelts);   %d.o.f. for the 4 faces of each element

[J,I]=meshgrid(1:4*d2);
R=face(I(:),:); R=reshape(R,4*d2,4*d2,Nelts);
C=face(J(:),:); C=reshape(C,4*d2,4*d2,Nelts);
      % R_ij^K d.o.f. for local (i,j) d.o.f. in element K ; R_ij^K=C_ji^K
RowsRHS=reshape(face,4*d2*Nelts,1);

dirfaces=(T.dirfaces(:)-1)*d2;
dirfaces=bsxfun(@plus,dirfaces,1:d2);
dirfaces=reshape(dirfaces',d2*Ndir,1);

free=((1:Nfaces)'-1)*d2;
free=bsxfun(@plus,free,1:d2);
free=reshape(free',d2*Nfaces,1);
free(dirfaces)=[];
```

```
neufaces=(T.neufaces(:)-1)*d2;
neufaces=bsxfun(@plus,neufaces,1:d2);
neufaces=reshape(neufaces',d2*Nneu,1);

%Local solvers and global system

[M1,Cf,A1,A2,Af]=localsolvers3d(km,c,f,tau,T,k,formulas);
M=sparse(R(:),C(:),M1(:));
phif=accumarray(RowsRHS,Cf(:));

[uhatD,qhatN]=BC3d(uD,gx,gy,gz,T,k,formulas{3});

%Dirichlet BC
Uhatv=zeros(d2*Nfaces,1);
Uhatv(dirfaces)=uhatD;       %uhat stored as a vector: d2*Nfaces

%RHS
rhs=zeros(d2*Nfaces,1);
rhs(free)=phif(free);
qhatN=reshape(qhatN,d2*Nneu,1);   % qhatN stored as a vector: d2*Nneu
rhs(neufaces)=rhs(neufaces)+qhatN;
rhs=rhs-M(:,dirfaces)*Uhatv(dirfaces);

if export
    system={M,rhs,free,dirfaces};
    solvers={A1,A2,Af};
    Uh=[];
    Qxh=[];
    Qyh=[];
    Qzh=[];
    Uhat=[];
    return
else
    system=[];
    solvers=[];
end

Uhatv(free)=M(free,free)\rhs(free);
Uhat=reshape(Uhatv,d2,Nfaces);

%Uh Qxh Qyh Qzh

faces=T.facebyele'; faces=faces(:);
uhhataux=reshape(Uhat(:,faces),[4*d2,Nelts]);
sol=zeros(4*d3,Nelts);
parfor K=1:Nelts
    sol(:,K)=A1(:,:,K)\(Af(:,K)-A2(:,:,K)*uhhataux(:,K));
end

Qxh=sol(block3(1),:);
Qyh=sol(block3(2),:);
Qzh=sol(block3(3),:);
Uh =sol(block3(4),:);
return
```

## 10 Local projections

### 10.1 $L^2$ projection on elements

Given $u : \Omega \to \mathbb{R}$ we look for $u_h$ (given by coefficients $u_j^K$) such that

$$\sum_j \Big( \int_K P_i^K P_j^K \Big) u_j^K = \int_K P_i^K u_h = \int_K P_i^K u \qquad i = 1, \ldots, d_3, \qquad K \in \mathcal{T}_h.$$

Using a quadrature rule, we can compute

$$\int_K P_i^K P_j^K = |K| \sum_q \widehat{\omega}_q \widehat{P}_i(\widehat{\mathbf{p}}_q)\widehat{P}_j(\widehat{\mathbf{p}}_q),$$

and approximate

$$\int_K P_i^K u \approx |K| \sum_q \widehat{\omega}_q \widehat{P}_i(\widehat{\mathbf{p}}_q) u(\mathbf{p}_q^K)$$

so that, with the $N_{\mathrm{qd}} \times N_{\mathrm{elt}}$ matrices of coordinates of all quadrature points on all elements, we can just compute

$$((\widehat{\boldsymbol{\omega}} \odot \mathrm{P})^\top \mathrm{P})^{-1} (\widehat{\boldsymbol{\omega}} \odot \mathrm{P})^\top u(\mathrm{X}, \mathrm{Y}, \mathrm{Z}).$$

```
    function [fh,error]=L2proj3d(f,T,k,formula1,formula2)

%[fh,error]=L2proj3d(f,T,k,formula1,formula2)
%
%Input:
%           f: vectorized function of three variables
%           T: full tetrahedrization
%           k: polynomial degree
%    formula1: quadrature formula in 3d
%    formula2: quadrature formula in 3d to compute errors
%
%Output:
%          fh: L2 projection of f ; disc P_k function (d3 x Nelts)
%       error: L^2 error
%
%Last modified: April 2, 2013

P =dubiner3d(2*formula2(:,2)-1,2*formula2(:,3)-1,2*formula2(:,4)-1,k);
wP=bsxfun(@times,formula2(:,5),P);
x=T.coordinates(:,1); x=formula2(:,1:4)*x(T.elements');
y=T.coordinates(:,2); y=formula2(:,1:4)*y(T.elements');
z=T.coordinates(:,3); z=formula2(:,1:4)*z(T.elements');

fh=((wP'*P)\wP')*f(x,y,z);
error=errorElem(T,f,fh,k,formula1);
return
```

## 10.2 $L^2$ projection on the skeleton

Given $u : \Omega \to \mathbb{R}$, we look for $\widehat{u}_h \in M_h$ (given by coefficients $u_j^e$) such that

$$\sum_j \Big( \int_e D_i^e D_j^e \Big) u_j^e = \int_e D_i^e \widehat{u}_h = \int_e D_i^e u \qquad i = 1, \ldots, d_2, \qquad e \in \mathcal{E}_h.$$

Using a quadrature formula we compute

$$\int_e D_i^e D_j^e = |e| \sum_r \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r)\widehat{D}_j(\widehat{\mathbf{q}}_r)$$

and approximate

$$\int_e D_i^e\, u \approx |e| \sum_r \varpi_r \widehat{D}_i(\widehat{\mathbf{q}}_r) u(\widehat{\mathbf{q}}_r^e).$$

Therefore, we just have to compute

$$((\boldsymbol{\varpi} \odot \mathrm{D})^\top \mathrm{D})^{-1} (\boldsymbol{\varpi} \odot \mathrm{D})^\top u(\mathrm{X}, \mathrm{Y}, \mathrm{Z}),$$

after computing the matrices with the coordinates of all quadrature points in all elements.

```
function [fh,error]=L2projskeleton3d(f,T,k,formula1,formula2)

%[fh,error]=L2projskeleton3d(f,T,k,formula1,formula2)
%
%Input:
%            f: vectorized function of three variables
%            T: expanded tetrahedrization
%            k: polynomial degree
%     formula1: quadrature formula in 2d
%     formula2: quadrature formula in 2d to compute errors
%
%Output:
%           fh: L2 projection on the skeleton (d2 x Nfaces)
%        error: h-weighted error
%
%Last modified: April 2, 2013

D =dubiner2d(2*formula1(:,2)-1,2*formula1(:,3)-1,k);   %Nnodes x d2
wD=bsxfun(@times,formula1(:,4),D);
x=T.coordinates(:,1); x=formula1(:,1:3)*x(T.faces(:,1:3)');
y=T.coordinates(:,2); y=formula1(:,1:3)*y(T.faces(:,1:3)');
z=T.coordinates(:,3); z=formula1(:,1:3)*z(T.faces(:,1:3)');

fh=((wD'*D)\wD')*f(x,y,z);

error=errorFaces(T,f,fh,k,formula2);
return
```

## 10.3 The HDG projection

Given a pair $(\boldsymbol{q}, u)$ (a vector field and a function), the HDG projection is the discrete pair $(\boldsymbol{q}_h, u_h) \in \boldsymbol{V}_h \times W_h$ such that

$$\begin{aligned}
(\boldsymbol{q}_h, \boldsymbol{r}_h)_K &= (\boldsymbol{q}, \boldsymbol{r}_h)_K & \forall \boldsymbol{r}_h \in \mathcal{P}_{k-1}(K)^3,\\
(u_h, w_h)_K &= (u, w_h)_K & \forall w_h \in \mathcal{P}_{k-1}(K),\\
\langle \boldsymbol{q}_h \cdot \boldsymbol{\nu} + \tau u_h, \widehat{v}_h\rangle_{\partial K} &= \langle \boldsymbol{q} \cdot \boldsymbol{\nu} + \tau u, \widehat{v}_h\rangle_{\partial K} & \forall \widehat{v}_h \in \mathcal{R}_k(\partial K),
\end{aligned}$$

for all $K \in \mathcal{T}_h$. This problem can be decomposed in a sequence of $N_{\rm elt}$ linear systems of order $4d_3$. We first compute a constant mass matrix $\mathrm{M}^K$ (in the usual $d_3 \times d_3 \times N_{\rm elt}$ format) and drop the last $d_2$ rows, to get a $(d_3 - d_2) \times d_3 \times N_{\rm elt}$ array with slices $\widetilde{\mathrm{M}}^K$. (Note that $\dim \mathcal{P}_k(K) - \dim \mathcal{P}_{k-1}(K) = d_2$. In the case $k = 0$, $\widetilde{\mathrm{M}}$ is an empty matrix.)
The matrix to that appears in the left hand side of each of the local projectors is the $(4(d_3-d_2)+4d_2)\times 4d_3$ block matrix

$$\begin{bmatrix}
\widetilde{\mathrm{M}}^K & \mathrm{O} & \mathrm{O} & \mathrm{O}\\
\mathrm{O} & \widetilde{\mathrm{M}}^K & \mathrm{O} & \mathrm{O}\\
\mathrm{O} & \mathrm{O} & \widetilde{\mathrm{M}}^K & \mathrm{O}\\
\mathrm{O} & \mathrm{O} & \mathrm{O} & \widetilde{\mathrm{M}}^K\\
n_x \mathrm{DP}^K & n_y \mathrm{DP}^K & n_z \mathrm{DP}^K & \tau \mathrm{DP}^K
\end{bmatrix}$$

The first four blocks of the right hand side (for the element $K$) correspond to the tests

$$\int_K q_x P_i^K, \qquad \int_K q_y P_i^K, \qquad \int_K q_z P_i^K, \qquad \int_K u P_i^K, \qquad i = 1, \ldots, d_3 - d_2.$$

```
function [Pqx,Pqy,Pqz,Pu]=projectHDG3d(T,coeffs,k,tau,formulas)
```

```
%[Pqx,Pqy,Pqz,Pu]=projectHDG3d(T,coeffs,k,tau,formulas)
%
%Input:
%          T: Tetrahedrization
%     coeffs: each cell is a vectorized function of three variables
%          k: degree of polynomial
%   formulas: quadrature matrix for mass matrix and convection matrix
%
%Output:
%        Pqx: \Pi qx,
%        Pqy: \Pi qy,
%        Pqz: \Pi qz,
%        Pu: \Pi u,

%Last modified: April 9, 2013

d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3); %Nbasis
if k>0
    d4=nchoosek(k+2,3); %dim P_{k-1}(K)
else
    d4=0;
end
block3=@(x) (1+(x-1)*d3):(x*d3);
a=@(x,y,z) 1+0.*x;

Mass=MassMatrix(T,{a},k,formulas{1});

Mass=Mass{1};
rows=1+d4:d3;
Mass(rows,:,:)=[];

[tauPP,tauDP,nxDP,nyDP,nzDP]=matricesFace(T,tau,k,formulas{3});
Nelts=size(T.elements,1);
O=zeros(d4,d3,Nelts);
M=[Mass O    O    O   ;
   O    Mass O    O   ;
   O    O    Mass O   ;
   O    O    O    Mass;
   nxDP nyDP nzDP tauDP];

if k>0
    Ints=testElem(coeffs,T,k-1,formulas{2});
    qxP=Ints{1};
    qyP=Ints{2};
    qzP=Ints{3};
    uP=Ints{4};
else
    qxP=zeros(0,Nelts);
    qyP=zeros(0,Nelts);
    qzP=zeros(0,Nelts);
    uP=zeros(0,Nelts);
end

Ints=testFaces(coeffs,T,k,formulas{3});
qxD=Ints{1};
qyD=Ints{2};
qzD=Ints{3};
 ud=Ints{4};

nx=T.normals(:,[1 4 7 10]);
ny=T.normals(:,[2 5 8 11]);
nz=T.normals(:,[3 6 9 12]);
nx=nx./T.area(T.facebyele);
ny=ny./T.area(T.facebyele);
```

```
nz=nz./T.area(T.facebyele);

QxD=zeros(d2,4,Nelts);
QyD=QxD;
QzD=QxD;
UD =QxD;

Pqx=zeros(d3,Nelts);
Pqy=Pqx;
Pqz=Pqx;
Pu =Pqx;

parfor i=1:Nelts
    QxD(:,:,i)=bsxfun(@times,qxD(:,T.facebyele(i,:)),nx(i,:));
    QyD(:,:,i)=bsxfun(@times,qyD(:,T.facebyele(i,:)),ny(i,:));
    QzD(:,:,i)=bsxfun(@times,qzD(:,T.facebyele(i,:)),nz(i,:));
     UD(:,:,i)=bsxfun(@times,ud(:,T.facebyele(i,:)),tau(:,i)');
    qxv=QxD(:,:,i);
    qyv=QyD(:,:,i);
    qzv=QzD(:,:,i);
     uv=UD(:,:,i);

    rhs=[qxP(:,i);
         qyP(:,i);
         qzP(:,i);
          uP(:,i);
          qxv(:)+qyv(:)+qzv(:)+uv(:)];
     vect=M(:,:,i)\rhs;

     Pqx(:,i)=vect(block3(1));
     Pqy(:,i)=vect(block3(2));
     Pqz(:,i)=vect(block3(3));
      Pu(:,i)=vect(block3(4));
end
return
```

## 11 Preprocessing of the triangulation

The first part of the code is quite technical and could possibly be optimized. We start by constructing `T.faces`, the list of all faces. The way this is constructed, the list contains:

- first all the interior faces; for each of the faces, the nodes are given in increasing order;
- then the Dirichlet faces, preserving the ordering that was given as input
- then the Neumann faces, preserving the ordering that was given as input.

The HDG code does not use these particularities of the construction of the list of faces. We next construct the list `T.facebyele` by backward referencing from the original lists. The last delicate part of the code is the construction of `T.perm`. Other than that, everything is relatively straightforward.

```
function T=HDGgrid3d(T,positive)
%
% T=HDGgrid3d(T,p)
%
% Input:
%     T : Basic tetrahedral data structure
%     p : 1 if boundary faces are positively oriented
%         -1 otherwise
% Output:
%     T : Expanded tetrahedral data structure
%
```

```
% Last modified: May 2, 2012
%

if positive≠1
    T.dirichlet(:,[2 3])=T.dirichlet(:,[3 2]);
    T.neumann(:,[2 3])=T.neumann(:,[3 2]);
end

% Construction of a list of all faces

shape=[1 3 2;1 2 4;1 4 3;2 3 4];
nelts=size(T.elements,1);
faces=zeros(4*nelts,3);
 for k=1:nelts
     nodes=T.elements(k,:); %1x4
     faces(4*(k-1)+(1:4),:)=nodes(shape); % 1x3
 end
copyoffaces=faces;  % 4 nelts x 3 (in local positive orientation)
faces=sort(faces,2);
[allfaces,i,j]=unique(faces,'rows');

% Lists of interior and boundary faces with references

bdfaces=sort([T.dirichlet;T.neumann],2);
[intfaces,i]=setdiff(allfaces,bdfaces,'rows');
[bdfaces,ii,jj]=intersect(allfaces,bdfaces,'rows');

nintfaces=size(intfaces,1);
ndirfaces=size(T.dirichlet,1);
nneufaces=size(T.neumann,1);
nbdfaces =size(bdfaces,1);
nfaces   =nintfaces+nbdfaces;

T.faces=[intfaces zeros(nintfaces,1);...
         T.dirichlet ones(ndirfaces,1);...
         T.neumann 2*ones(nneufaces,1)];
T.dirfaces=(nintfaces+1):(nintfaces+ndirfaces);
T.neufaces=(nintfaces+ndirfaces+1):(nintfaces+ndirfaces+nneufaces);

% Backward referencing to construct T.facebyele

u=zeros(nfaces,1);
v=nintfaces+1:nintfaces+nbdfaces;
u(i)=1:nintfaces;
u(ii)=v(jj);
j=u(j);                 % pointer from T.faces to the copyoffaces
faces=T.faces(j,1:3);   % 4 nelts x 3, with global numbering of nodes
j=reshape(j,[4 nelts]);
T.facebyele=j';

% Matrix with orientations

A=T.facebyele';
faces=T.faces(A(:),1:3);
t=sum(faces==copyoffaces,2)==ones(4*nelts,1);
t=1-2*t;t=reshape(t,[4,nelts]);
T.orientation=t';

% Matrix with permutation order

eq = @(u,v) sum(u==v,2)==3; % checks what rows are equal
rot=[1 2 3;...              % permutations
     1 3 2;...
     3 1 2;...
     3 2 1;...
     2 3 1;...
     2 1 3];
pattern=[1 2 3;...    % (s,t,0)
         1 2 4;...    % (s,0,t)
         1 3 4;...    % (0,s,t)
         4 2 3];      % (s,t,1-s-t)
orient=zeros(nelts,4);

for f=1:4    % counter over faces
    faceGlobal=T.faces(T.facebyele(:,f),1:3);
    faceLocal =T.elements(:,pattern(f,:));
    for j=1:6
        orient(:,f)=orient(:,f)+j*eq(faceGlobal,faceLocal(:,rot(j,:)));
    end
end
T.perm=orient;

% Volumes and areas

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


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

%Normals to the faces

for f=1:4
    oneface=T.faces(T.facebyele(:,f),1:3);
    x=T.coordinates(oneface(:),1);
    y=T.coordinates(oneface(:),2);
    z=T.coordinates(oneface(:),3);
%
    x12=x(nelts+1:2*nelts)-x(1:nelts); %x_2-x_1
    x13=x(2*nelts+1:end)-x(1:nelts);   %x_3-x_1
    y12=y(nelts+1:2*nelts)-y(1:nelts); %y_2-y_1
    y13=y(2*nelts+1:end)-y(1:nelts);   %y_3-y_1
    z12=z(nelts+1:2*nelts)-z(1:nelts); %z_2-z_1
    z13=z(2*nelts+1:end)-z(1:nelts);    %z_3-z_1
 %
    normals=(1/2)*[y12.*z13-y13.*z12,...
                   z12.*x13-z13.*x12,...
                   x12.*y13-x13.*y12];
    normals=bsxfun(@times,normals,T.orientation(:,f)); % Give the normals correctly orientated
%
    T.normals(:,(f-1)*3+(1:3))=normals;
end

return
```

## 12 Matrices for convection-diffusion

### 12.1 Variable convection

In order to compute convection matrices with a variable coefficient

$$\int_K c\, P_i^K \partial_\star P_j^K \qquad i,j,=1,\ldots,d_3, \qquad K\in\mathcal{T}_h, \qquad \star\in\{x,y,z\},$$

we proceed like in Section 5.5, but looping on quadrature nodes. Using notation of Section 5.5, we can write

$$\begin{aligned}\int_K c\, P_i^K P_{j,\star}^K &= \sum_{\#=1}^{3} a_{\star\#}^K \int_{\widehat{K}} (c\circ \mathrm{F}_K)\widehat{P}_i \widehat{P}_{j,\#} \\ &\approx \tfrac{1}{6}\sum_{\#=1}^{3}\sum_q a_{\star\#}^q c(\mathbf{p}_q^K)(\widehat{\mathrm{C}}_q^{\#})_{ij}, \qquad (\widehat{\mathrm{C}}_q^{\#})_{ij} := \widehat{\omega}_q \widehat{P}_i(\widehat{\mathbf{p}}_q)\widehat{P}_{j,\#}(\widehat{\mathbf{p}}_q).\end{aligned}$$

This means that we need to compute $3N_{\mathrm{qd}}$ $(d_3\times d_3)$ matrices on the reference element (the matrices $\widehat{\mathrm{C}}_q^{\#}$) and use them for $3N_{\mathrm{qd}}$ Kronecker products in order to compute a particular value of $\star$. In total, there will be $9N_{\mathrm{qd}}$ Kronecker products.
As for the vector involved in the Kronecker products, they are the columns of $\mathbf{a}_{\star\#}\odot c(\mathrm{X},\mathrm{Y},\mathrm{Z})$, where $\mathbf{a}_{\star\#}$ are the *column* vectors with the geometric coefficients of the change of variables and $\mathrm{X},\mathrm{Y},\mathrm{Z}$ are the $N_{\mathrm{qd}}\times N_{\mathrm{elt}}$ matrices with the coordinates of the quadrature points.

**Implementation notes.** Thinking of convection-diffusion problems, the code provides convection matrices with different parameter functions for each of the variables, that is

$$\int_K v_x P_i^K \partial_x P_j^K \qquad \int_K v_y P_i^K \partial_y P_j^K \qquad \int_K v_z P_i^K \partial_z P_j^K.$$

The variables `a, b, c,...` are used to tag the nine components of $\det \mathrm{B}_K\, \mathrm{B}_K^{-\top}$ read by rows (see Section 5.5) as $N_{\mathrm{elt}}\times 1$ vectors. The capitalized forms `A, B, ...` correspond to $N_{\mathrm{elt}}\times N_{\mathrm{qd}}$ arrays with the values of the variable coefficient at all quadrature points. For instance `a, b, c` corresponding to elements of the first row (and thus to a $\mathrm{C}^x$ matrix), they have to be multiplied by the value of $v_x$ at quadrature points.

```
function [convx,convy,convz]=VariableConv(T,vx,vy,vz,k,formula)

%[convx,convy,convz]=VariableConv(T,vx,vy,vz,k,formula)
%
%Input:
%         T: expanded tetrahedrization
%  vx,vy,vz: vectorized functions of three variables
%         k: polynomial degree
%   formula: quadrature formula in 3d (N x 5 matrix)
%
%Output:
%     convx: d3 x d3 x Nelts ( \int_K vx P_i^K \partial_x P_j^K )
%     convy: d3 x d3 x Nelts ( \int_K vy P_i^K \partial_y P_j^K )
%     convz: d3 x d3 x Nelts ( \int_K vz P_i^K \partial_z P_j^K )
%
%Last modified: May 31, 2012

Nelts=size(T.elements,1);
Nnodes=size(formula,1);
d3=nchoosek(k+3,3);

x=T.coordinates(:,1);x=x(T.elements)*formula(:,1:4)'; %Nelts x Nnodes
y=T.coordinates(:,2);y=y(T.elements)*formula(:,1:4)';
z=T.coordinates(:,3);z=z(T.elements)*formula(:,1:4)';

convx=zeros(d3,Nelts*d3);
convy=zeros(d3,Nelts*d3);
convz=zeros(d3,Nelts*d3);

xhat=formula(:,2);
yhat=formula(:,3);
zhat=formula(:,4);

[p,px,py,pz]=dubiner3d(2*xhat-1,2*yhat-1,2*zhat-1,k); % Nnodes x d3
px=2*px;
py=2*py;
pz=2*pz;

x12=T.coordinates(T.elements(:,2),1)-T.coordinates(T.elements(:,1),1); %x2-x1
x13=T.coordinates(T.elements(:,3),1)-T.coordinates(T.elements(:,1),1); %x3-x1
x14=T.coordinates(T.elements(:,4),1)-T.coordinates(T.elements(:,1),1); %x4-x1
y12=T.coordinates(T.elements(:,2),2)-T.coordinates(T.elements(:,1),2); %y2-y1
y13=T.coordinates(T.elements(:,3),2)-T.coordinates(T.elements(:,1),2); %y3-y1
y14=T.coordinates(T.elements(:,4),2)-T.coordinates(T.elements(:,1),2); %y4-y1
z12=T.coordinates(T.elements(:,2),3)-T.coordinates(T.elements(:,1),3); %z2-z1
z13=T.coordinates(T.elements(:,3),3)-T.coordinates(T.elements(:,1),3); %z3-z1
z14=T.coordinates(T.elements(:,4),3)-T.coordinates(T.elements(:,1),3); %z4-z1

a=y13.*z14-y14.*z13;
b=y14.*z12-y12.*z14;
c=y12.*z13-y13.*z12;
d=x14.*z13-x13.*z14;
e=x12.*z14-x14.*z12;
f=x13.*z12-x12.*z13;
g=x13.*y14-x14.*y13;
h=x14.*y12-x12.*y14;
i=x12.*y13-x13.*y12;

A=bsxfun(@times,a,vx(x,y,z));
B=bsxfun(@times,b,vx(x,y,z));
C=bsxfun(@times,c,vx(x,y,z));
D=bsxfun(@times,d,vy(x,y,z));
E=bsxfun(@times,e,vy(x,y,z));
F=bsxfun(@times,f,vy(x,y,z));
G=bsxfun(@times,g,vz(x,y,z));
H=bsxfun(@times,h,vz(x,y,z));
I=bsxfun(@times,i,vz(x,y,z));

for i=1:Nnodes
    convx=convx+kron(A(:,i)',1/6*formula(i,5)*p(i,:)'*px(i,:))+...
                kron(B(:,i)',1/6*formula(i,5)*p(i,:)'*py(i,:))+...
                kron(C(:,i)',1/6*formula(i,5)*p(i,:)'*pz(i,:));
    convy=convy+kron(D(:,i)',1/6*formula(i,5)*p(i,:)'*px(i,:))+...
                kron(E(:,i)',1/6*formula(i,5)*p(i,:)'*py(i,:))+...
                kron(F(:,i)',1/6*formula(i,5)*p(i,:)'*pz(i,:));
    convz=convz+kron(G(:,i)',1/6*formula(i,5)*p(i,:)'*px(i,:))+...
                kron(H(:,i)',1/6*formula(i,5)*p(i,:)'*py(i,:))+...
                kron(I(:,i)',1/6*formula(i,5)*p(i,:)'*pz(i,:));

end
convx=reshape(convx,[d3,d3,Nelts]);
convy=reshape(convy,[d3,d3,Nelts]);
convz=reshape(convz,[d3,d3,Nelts]);
return
```

## 12.2 Face matrices related to convection

The first collection of matrices is a generalization of what we called the Type (c) matrices in Section 6.4. The goal is the approximate computation of

$$\frac{\xi_\ell^K}{|e_\ell^K|}\int_{e_\ell^K} \alpha\, D_i^{e_\ell^K} P_j^K \qquad \begin{array}{l} i=1,\ldots,d_2 \\ j=1,\ldots,d_3 \end{array} \qquad \ell\in\{1,2,3,4\}, \qquad K\in\mathcal{T}_h,$$

where $\boldsymbol{\xi}\in\mathcal{P}_0(\partial\mathcal{T}_h)$, and $\alpha$ is a function of three variables. The result will be presented as a $(4d_2)\times d_3\times N_{\mathrm{elt}}$ array, by stacking the blocks for $\ell\in\{1,2,3,4\}$ on top of each other ($\ell=1$ on top). Most of what we next explain is an easy generalization of what appears in Section 6.4.
We first compute the evaluation matrices (see Section 3.3 for the quadrature nodes on the boundary of the reference tetrahedron)

$$\begin{array}{rlll} \mathrm{P}^\ell_{ri} := \widehat{P}_i(\widehat{\mathbf{q}}^\ell_r) & i=1,\ldots,d_3, & r=1,\ldots,N_{\mathrm{qd2}}, & \ell\in\{1,2,3,4\}, \\ \mathrm{D}^\mu_{ri} := \widehat{D}_i(F_\mu(\widehat{\mathbf{q}}_r)) & i=1,\ldots,d_2, & r=1,\ldots,N_{\mathrm{qd2}}, & \mu\in\{1,2,3,4,5,6\}. \end{array}$$

We next consider the piecewise constant functions $\boldsymbol{\xi}_\mu\in\mathcal{P}_0(\partial\mathcal{T}_h)$ given by

$$\xi^K_{\ell,\mu} = \xi^K_\ell \mathbf{1}_{\mathrm{perm}(K,\ell)=\mu} = \begin{cases} \xi^K_\ell & \text{if } \mathrm{perm}(K,\ell)=\mu, \\ 0, & \text{otherwise.} \end{cases}$$

If $\mathrm{X}^\mathcal{T},\mathrm{Y}^\mathcal{T},\mathrm{Z}^\mathcal{T}$ are the $4\times N_{\mathrm{elt}}$ matrices with the coordinates of the vertices of the elements and $\Omega_\ell$ is the $N_{\mathrm{qd2}}\times 4$ matrix with the barycentric coordinates of the points $\{\widehat{\mathbf{q}}^\ell_r\}$, we then construct the $N_{\mathrm{qd2}}\times N_{\mathrm{elt}}$ matrices

$$\mathrm{A}_\ell := \alpha(\Omega_\ell \mathrm{X}^\mathcal{T}, \Omega_\ell \mathrm{X}^\mathcal{T}, \Omega_\ell \mathrm{X}^\mathcal{T}) \qquad \ell\in\{1,2,3,4\},$$

containing the values of $\alpha$ in all quadrature points. The computation is then

$$\sum_{\mu=1}^{6} (\mathbf{a}^\top_{\ell,r}\bullet\boldsymbol{\xi}^\top_{\mu,r})\otimes\Big((\mathrm{D}^\mu)^\top(\boldsymbol{\varpi}\odot\mathrm{P}^\ell)\Big), \qquad \mathbf{a}^\top_{\ell,r} := \mathrm{Row}(\mathrm{A}_\ell, r), \qquad \boldsymbol{\xi}^\top_{\mu,r} := \mathrm{Row}(\boldsymbol{\xi}_\mu, r),$$

where the symbol $\bullet$ has been used to denote the element-by-element muyltiplication of two arays.

**Implementation notes.** Instead of doing the computation for a single function $\alpha$ paired with a single piecewise constant function $\boldsymbol{\xi}$, the code runs for a cell array of functions $\alpha$ paired with a cell array of piecewise constant functions $\boldsymbol{\xi}$.

```
function mat=matricesVariableFaceA(T,al,pw,k,formula)

% {m1,m2,...}=matricesFaceVariable(T,{a1,a2,...},{pw1,pw2...},k,form)
%
% Input:
```

```
%      T           : full tetrahedrization
%      a1,a2,...   : vectorized function of three variables
%      pw1,pw2,... : piecewise constant function on skeleton 4 x Nfaces
%      k           : polynomial degree
%      form        : 2-dimensional quadrature rule
% Output:
%      m1,m2,...   : 4d2 x d3 x Nelts
%
% Last modified: April 5, 2013

% Evaluations of Dubiner3d functions on all faces

Nelts=size(T.elements,1);
Nnodes=size(formula,1);
T.perm=T.perm';                       % 4 x Nelts

d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);

s=formula(:,2);
t=formula(:,3);
O=zeros(size(s));
points3d=[s,t,O;...
          s,O,t;...
          O,s,t;...
          s,t,1-s-t];

P =dubiner3d(2*points3d(:,1)-1,2*points3d(:,2)-1,2*points3d(:,3)-1,k);
wP=bsxfun(@times,[formula(:,4);...
                  formula(:,4);...
                  formula(:,4);...
                  formula(:,4)],P);

% Evaluations of alpha on all faces

x=T.coordinates(:,1);
y=T.coordinates(:,2);
z=T.coordinates(:,3);
points3d=[1-sum(points3d,2) points3d];  % barycentric coordinates
x=points3d*x(T.elements');
y=points3d*y(T.elements');
z=points3d*z(T.elements');

% Evaluations of Dubiner2d polynomials on six different configurations

points2d=[s,t;...
          t,s;...
          1-s-t,s;...
          s,1-s-t;...
          t,1-s-t;...
          1-s-t,t];
D=dubiner2d(2*points2d(:,1)-1,2*points2d(:,2)-1,k);    % 6*Nnodes x d2

% Construction of the matrix

for c=1:length(al)
    alpha=al{c};
    alpha=alpha(x,y,z);       % 4Nnodes X Nelts
    pwct =pw{c};
    matrix=zeros(4*d2,d3*Nelts);
    for l=1:4
        rows=1+(l-1)*d2:(l*d2);
        for mu=1:6
            pwctmu=pwct(l,:).*(T.perm(l,:)==mu);
            for r=1:Nnodes
                rowP=(l-1)*Nnodes+r;
                rowD=(mu-1)*Nnodes+r;
                matrix(rows,:)=matrix(rows,:)...
                    +kron(pwctmu.*alpha(rowP,:),D(rowD,:)'*wP(rowP,:));
            end
        end
    end
    mat{c}=reshape(matrix,[4*d2,d3,Nelts]);
end
```

The following step concenrs the generalization of the Type (b) matrices of Section 6.4 to variable coefficients. We show how to approximate

$$\frac{\xi_\ell^K}{|e_\ell^K|}\int_{e_\ell^K} \alpha\, D_i^{e_\ell^K} D_j^{e_\ell^K} \qquad i,j=1,\ldots,d_2 \qquad \ell\in\{1,2,3,4\}, \qquad K\in\mathcal{T}_h,$$

where $\boldsymbol{\xi}\in\mathcal{P}_0(\partial\mathcal{T}_h)$, and $\alpha$ is a function of three variables. The result will be given as a block diagonal $(4d_2)\times(4d_2)\times N_{\rm elt}$ array, by daigonally stacking the blocks for different values of $\ell$. We start by computing the values of $\alpha$ on all quadrature nodes: the $N_{\rm qd2}\times N_{\rm fc}$ matrix

$$\mathrm{A}:=\alpha(\Xi\,\mathrm{X}^{\mathcal{E}},\Xi\,\mathrm{Z}^{\mathcal{E}},\Xi\,\mathrm{Y}^{\mathcal{E}})$$

(where $\mathrm{X}^{\mathcal{E}},\mathrm{Y}^{\mathcal{E}},\mathrm{Z}^{\mathcal{E}}$ are $3\times N_{\rm fc}$ with the coordinates of all vertices of all faces and $\Xi$ is the $N_{\rm qd2}\times 3$ matrix with the barycentric coordinates of the quadrature points $\widehat{\mathbf{q}}_r$), is organized by global number of faces. We can then construct (by choosing columns in a proper way) for $N_{\rm qd2}\times N_{\rm elt}$ matrices $\mathrm{A}_\ell$ such that

$$\mathrm{A}^K_{\ell,r}=\mathrm{A}_{r,e_\ell^K} \qquad r=1,\ldots,N_{\rm qd2}, \qquad K\in\mathcal{T}_h, \qquad \ell\in\{1,2,3,4\}.$$

(The information needed for this is contained in `T.facebyele(:,l)`.) Finally we compute

$$\sum_r(\mathbf{a}_{\ell,r}^\top\bullet\boldsymbol{\xi}_\ell^\top)\otimes(\varpi_r\mathbf{d}_r\mathbf{d}_r^\top), \qquad \mathbf{a}_{\ell,r}^\top:=\mathrm{Row}(\mathrm{A}_\ell,r), \quad \boldsymbol{\xi}_\ell^\top:=\mathrm{Row}(\boldsymbol{\xi},\ell), \quad \mathbf{d}_r^\top:=\mathrm{Row}(\mathrm{D},r).$$

**Implementation notes.** Instead of doing the computation for a single function $\alpha$ paired with a single piecewise constant function $\boldsymbol{\xi}$, the code runs for a cell array of functions $\alpha$ paired with a cell array of piecewise constant functions $\boldsymbol{\xi}$.

```
function mat=matricesVariableFaceB(T,al,pw,k,formula)

% {m1,m2,...}=matricesFaceVariable(T,{a1,a2,...},{pw1,pw2,...},k,form)
%
% Input:
%       T             : full tetrahedrization
%       a1,a2,...     : vectorized function of three variables
%       pw1,pw2,...   : piecewise constant function on skeleton 4 x Nfaces
%       k             : polynomial degree
%       form          : 2-dimensional quadrature rule
% Output:
%       m1,m2,...     : 4d2 x 4d2 x Nelts
%
% Last modified: April 2, 2013

% Parameters

Nelts=size(T.elements,1);
Nnodes=size(formula,1);
d2=nchoosek(k+2,2);
O=zeros(d2,d2,Nelts);

% Evaluations of alpha and Dubiner 2d on all faces
```

```
x=T.coordinates(:,1);x=formula(:,1:3)*x(T.faces(:,1:3)');
y=T.coordinates(:,2);y=formula(:,1:3)*y(T.faces(:,1:3)');
z=T.coordinates(:,1);z=formula(:,1:3)*z(T.faces(:,1:3)');
D=dubiner2d(2*formula(:,2)-1,2*formula(:,3)-1,k);

% Construction of the matrix

for c=1:length(al);
    alpha=al{c};
    alpha=alpha(x,y,z);       % Nnodes X Nfaces
    pwct=pw{c};
    for l=1:4
        dgm{l}=zeros(d2,d2*Nelts);
        for r=1:Nnodes
            dgm{l}=dgm{l}+...
                kron(pwct(l,:).*alpha(r,T.facebyele(:,l)),...
                     formula(r,4)*D(r,:)'*D(r,:));
        end
        dgm{l}=reshape(dgm{l},[d2,d2,Nelts]);
    end
    mat{c}=[dgm{1} 0      0      0      ;...
            0      dgm{2} 0      0      ;...
            0      0      dgm{3} 0      ;...
            0      0      0      dgm{4}];
end
return
```

# 13 Postprocessing

## 13.1 Local stiffness matrices

The goal of this section is the computation of the matrices

$$\int_K \nabla P_i^K \cdot \nabla P_j^K \qquad i,j = 1,\dots,d_3, \qquad K \in \mathcal{T}_h.$$

For differentiation indices, let us identify the sets $(x,y,z) \equiv (1,2,3)$. We consider the canonical basis of the space of matrices

$$\mathrm{E}_{\star\#} = \mathbf{e}_\star\,\mathbf{e}_\#^\top, \qquad \star,\# \in \{x,y,z\}.$$

For instance

$$\mathrm{E}_{xy} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

We need to compute six $3\times 3$ geometric matrices:

$$\mathrm{G}^K_{\star\#} = |\det \mathrm{B}_K|^2 \mathrm{B}_K^{-1}\mathrm{E}_{\star\#}\mathrm{B}_K^{-\top}, \qquad \star,\# \in \{x,y,x\}, \qquad K \in \mathcal{T}_h.$$

We will give explicit formulas for these matrices below. Note that

$$(\mathrm{G}^K_{\star\#})^\top = |\det \mathrm{B}_K|^2(\mathrm{B}_K^{-1}\mathrm{E}_{\star\#}\mathrm{B}_K^{-\top})^\top = |\det \mathrm{B}_K|^2\,\mathrm{B}_K^{-1}\mathrm{E}_{\star\#}^\top\mathrm{B}_K^{-\top} = \mathrm{G}^K_{\#\star},$$

which is the reason why only six of them are needed/computed. Besides, the matrices $\mathrm{G}^T_{\star\star}$ are symmetric. Once we have computed these matrices, we can obtain

$$\begin{aligned}
(\mathrm{S}^K_{\star\#})_{ij} &= \int_K \partial_\star P^K_i \partial_\# P^K_j = \int_K (\nabla P^K_i)\cdot(\mathrm{E}_{\star\#}\nabla P^K_j) \\
&= |\det \mathrm{B}_K| \int_{\widehat{K}} (\mathrm{B}_K^{-\top}\nabla \widehat{P}_i)\cdot(\mathrm{E}_{\star\#}\mathrm{B}_K^{-\top}\nabla \widehat{P}_j) \\
&= |\det \mathrm{B}_K|^{-1} \int_{\widehat{K}} (\nabla \widehat{P}_i)\cdot(\mathrm{G}^K_{\star\#}\nabla \widehat{P}_j) \\
&= |\det \mathrm{B}_K|^{-1} \sum_{\alpha,\beta\in\{x,y,z\}} (\mathrm{G}^K_{\star\#})_{\alpha,\beta} \int_{\widehat{K}} (\nabla \widehat{P}_i)\cdot(\mathrm{E}_{\alpha,\beta}\nabla \widehat{P}_j) \\
&= |\det \mathrm{B}_K|^{-1} \sum_{\alpha,\beta\in\{x,y,z\}} (\mathrm{G}^K_{\star\#})_{\alpha,\beta} \int_{\widehat{K}} \partial_\alpha \widehat{P}_i\, \partial_\beta \widehat{P}_j.
\end{aligned}$$

The elements of the matrices $\mathrm{G}^K_{\star\#}$ are polynomial expressions of the elements of $\mathrm{B}_K$.

$$\mathrm{G}^K_{xx} = \begin{bmatrix} (y_{13}\,z_{14} - z_{13}\,y_{14})^2 & (y_{13}\,z_{14} - z_{13}\,y_{14})(-y_{12}\,z_{14} + z_{12}\,y_{14}) & (y_{13}\,z_{14} - z_{13}\,y_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) \\ (y_{13}\,z_{14} - z_{13}\,y_{14})(-y_{12}\,z_{14} + z_{12}\,y_{14}) & (-y_{12}\,z_{14} + z_{12}\,y_{14})^2 & (-y_{12}\,z_{14} + z_{12}\,y_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) \\ (y_{13}\,z_{14} - z_{13}\,y_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) & (-y_{12}\,z_{14} + z_{12}\,y_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) & (y_{12}\,z_{13} - z_{12}\,y_{13})^2 \end{bmatrix}$$

$$\mathrm{G}^K_{xy} = \begin{bmatrix} (y_{13}\,z_{14} - z_{13}\,y_{14})(-x_{13}\,z_{14} + z_{13}\,x_{14}) & (y_{13}\,z_{14} - z_{13}\,y_{14})(x_{12}\,z_{14} - z_{12}\,x_{14}) & (y_{13}\,z_{14} - z_{13}\,y_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) \\ (-x_{13}\,z_{14} + z_{13}\,x_{14})(-y_{12}\,z_{14} + z_{12}\,y_{14}) & (-y_{12}\,z_{14} + z_{12}\,y_{14})(x_{12}\,z_{14} - z_{12}\,x_{14}) & (-y_{12}\,z_{14} + z_{12}\,y_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) \\ (-x_{13}\,z_{14} + z_{13}\,x_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) & (x_{12}\,z_{14} - z_{12}\,x_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) & (y_{12}\,z_{13} - z_{12}\,y_{13})(-x_{12}\,z_{13} + z_{12}\,x_{13}) \end{bmatrix}$$

$$\mathrm{G}^K_{xz} = \begin{bmatrix} (y_{13}\,z_{14} - z_{13}\,y_{14})(x_{13}\,y_{14} - y_{13}\,x_{14}) & (y_{13}\,z_{14} - z_{13}\,y_{14})(-x_{12}\,y_{14} + y_{12}\,x_{14}) & (y_{13}\,z_{14} - z_{13}\,y_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) \\ (x_{13}\,y_{14} - y_{13}\,x_{14})(-y_{12}\,z_{14} + z_{12}\,y_{14}) & (-y_{12}\,z_{14} + z_{12}\,y_{14})(-x_{12}\,y_{14} + y_{12}\,x_{14}) & (-y_{12}\,z_{14} + z_{12}\,y_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) \\ (x_{13}\,y_{14} - y_{13}\,x_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) & (-x_{12}\,y_{14} + y_{12}\,x_{14})(y_{12}\,z_{13} - z_{12}\,y_{13}) & (y_{12}\,z_{13} - z_{12}\,y_{13})(x_{12}\,y_{13} - y_{12}\,x_{13}) \end{bmatrix}$$

$$\mathrm{G}^K_{yy} = \begin{bmatrix} (-x_{13}\,z_{14} + z_{13}\,x_{14})^2 & (-x_{13}\,z_{14} + z_{13}\,x_{14})(x_{12}\,z_{14} - z_{12}\,x_{14}) & (-x_{13}\,z_{14} + z_{13}\,x_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) \\ (-x_{13}\,z_{14} + z_{13}\,x_{14})(x_{12}\,z_{14} - z_{12}\,x_{14}) & (x_{12}\,z_{14} - z_{12}\,x_{14})^2 & (x_{12}\,z_{14} - z_{12}\,x_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) \\ (-x_{13}\,z_{14} + z_{13}\,x_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) & (x_{12}\,z_{14} - z_{12}\,x_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) & (-x_{12}\,z_{13} + z_{12}\,x_{13})^2 \end{bmatrix}$$

$$\mathrm{G}^K_{yz} = \begin{bmatrix} (-x_{13}\,z_{14} + z_{13}\,x_{14})(x_{13}\,y_{14} - y_{13}\,x_{14}) & (-x_{13}\,z_{14} + z_{13}\,x_{14})(-x_{12}\,y_{14} + y_{12}\,x_{14}) & (-x_{13}\,z_{14} + z_{13}\,x_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) \\ (x_{13}\,y_{14} - y_{13}\,x_{14})(x_{12}\,z_{14} - z_{12}\,x_{14}) & (x_{12}\,z_{14} - z_{12}\,x_{14})(-x_{12}\,y_{14} + y_{12}\,x_{14}) & (x_{12}\,z_{14} - z_{12}\,x_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) \\ (x_{13}\,y_{14} - y_{13}\,x_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) & (-x_{12}\,y_{14} + y_{12}\,x_{14})(-x_{12}\,z_{13} + z_{12}\,x_{13}) & (-x_{12}\,z_{13} + z_{12}\,x_{13})(x_{12}\,y_{13} - y_{12}\,x_{13}) \end{bmatrix}$$

$$\mathrm{G}^K_{zz} = \begin{bmatrix} (x_{13}\,y_{14} - y_{13}\,x_{14})^2 & (x_{13}\,y_{14} - y_{13}\,x_{14})(-x_{12}\,y_{14} + y_{12}\,x_{14}) & (x_{13}\,y_{14} - y_{13}\,x_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) \\ (x_{13}\,y_{14} - y_{13}\,x_{14})(-x_{12}\,y_{14} + y_{12}\,x_{14}) & (-x_{12}\,y_{14} + y_{12}\,x_{14})^2 & (-x_{12}\,y_{14} + y_{12}\,x_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) \\ (x_{13}\,y_{14} - y_{13}\,x_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) & (-x_{12}\,y_{14} + y_{12}\,x_{14})(x_{12}\,y_{13} - y_{12}\,x_{13}) & (x_{12}\,y_{13} - y_{12}\,x_{13})^2 \end{bmatrix}$$

The stiffness matrix is the sum

$$\mathrm{S}^K_{xx} + \mathrm{S}^K_{yy} + \mathrm{S}^K_{zz}.$$

```
function S=Stiffness(T,k,formula)

% S=stiffness(T,k,formula)
%
% Input:
%   T       : tetrahedrization
%   k       : polynomial degree
%   formula : quadrature formula
% Output:
%   S       : d3 x d3 x Nelts stiffness matrix
% Last modified: March 27, 2013

% Computations in the reference element
```

```
xhat=formula(:,2);
yhat=formula(:,3);
zhat=formula(:,4);

[¬,px,py,pz]=dubiner3d(2*xhat-1,2*yhat-1,2*zhat-1,k);
px=2*px; wpx=bsxfun(@times,formula(:,5),px);
py=2*py; wpy=bsxfun(@times,formula(:,5),py);
pz=2*pz; wpz=bsxfun(@times,formula(:,5),pz);

Kxx=(1/6)*px'*wpx;
Kxy=(1/6)*px'*wpy;
Kxz=(1/6)*px'*wpz;
Kyy=(1/6)*py'*wpy;
Kyz=(1/6)*py'*wpz;
Kzz=(1/6)*pz'*wpz;

% Geometric constructions

x12=T.coordinates(T.elements(:,2),1)-T.coordinates(T.elements(:,1),1);
y12=T.coordinates(T.elements(:,2),2)-T.coordinates(T.elements(:,1),2);
z12=T.coordinates(T.elements(:,2),3)-T.coordinates(T.elements(:,1),3);
x13=T.coordinates(T.elements(:,3),1)-T.coordinates(T.elements(:,1),1);
y13=T.coordinates(T.elements(:,3),2)-T.coordinates(T.elements(:,1),2);
z13=T.coordinates(T.elements(:,3),3)-T.coordinates(T.elements(:,1),3);
x14=T.coordinates(T.elements(:,4),1)-T.coordinates(T.elements(:,1),1);
y14=T.coordinates(T.elements(:,4),2)-T.coordinates(T.elements(:,1),2);
z14=T.coordinates(T.elements(:,4),3)-T.coordinates(T.elements(:,1),3);
x12=x12'; x13=x13'; x14=x14';
y12=y12'; y13=y13'; y14=y14';
z12=z12'; z13=z13'; z14=z14'; % row vectors with differences of coords

idetB=1./(x12.*y13.*z14-x12.*y14.*z13-y12.*x13.*z14...
         +y12.*x14.*z13+z12.*x13.*y14-z12.*x14.*y13);

dxdx=kron(idetB.*(y13.*z14-z13.*y14).^2,Kxx)...
     +kron(idetB.*(y13.*z14-z13.*y14).*(-y12.*z14+z12.*y14),Kxy+Kxy')...
     +kron(idetB.*(y13.*z14-z13.*y14).*(y12.*z13-z12.*y13),Kxz+Kxz')...
     +kron(idetB.*(-y12.*z14+z12.*y14).^2,Kyy)...
     +kron(idetB.*(-y12.*z14+z12.*y14).*(y12.*z13-z12.*y13),Kyz+Kyz')...
     +kron(idetB.*(y12.*z13-z12.*y13).^2,Kzz);
dydy=kron(idetB.*(-x13.*z14+z13.*x14).^2,Kxx)...
     +kron(idetB.*(-x13.*z14+z13.*x14).*(x12.*z14-z12.*x14),Kxy+Kxy')...
     +kron(idetB.*(-x13.*z14+z13.*x14).*(-x12.*z13+z12.*x13),Kxz+Kxz')...
     +kron(idetB.*(x12.*z14-z12.*x14).^2,Kyy)...
     +kron(idetB.*(x12.*z14-z12.*x14).*(-x12.*z13+z12.*x13),Kyz+Kyz')...
     +kron(idetB.*(-x12.*z13+z12.*x13).^2,Kzz);
dzdz=kron(idetB.*(x13.*y14-y13.*x14).^2,Kxx)...
     +kron(idetB.*(x13.*y14-y13.*x14).*(-x12.*y14+y12.*x14),Kxy+Kxy')...
     +kron(idetB.*(x13.*y14-y13.*x14).*(x12.*y13-y12.*x13),Kxz+Kxz')...
     +kron(idetB.*(-x12.*y14+y12.*x14).^2,Kyy)...
     +kron(idetB.*(-x12.*y14+y12.*x14).*(x12.*y13-y12.*x13),Kyz+Kyz')...
     +kron(idetB.*(x12.*y13-y12.*x13).^2,Kzz);

 S=dxdx+dydy+dzdz;
 d3=nchoosek(k+3,3);
 Nelts=size(T.elements,1);
 S=reshape(S,[d3 d3 Nelts]);

 return

 % These matrices are not needed

 dxdy=kron(idetB.*(y13.*z14-z13.*y14).*(-x13.*z14+z13.*x14),Kxx)...
     +kron(idetB.*(y13.*z14-z13.*y14).*(x12.*z14-z12.*x14),Kxy)...
     +kron(idetB.*(y13.*z14-z13.*y14).*(-x12.*z13+z12.*x13),Kxz)...
     +kron(idetB.*(-x13.*z14+z13.*x14).*(-y12.*z14+z12.*y14),Kxy')...
     +kron(idetB.*(-y12.*z14+z12.*y14).*(x12.*z14-z12.*x14),Kyy)...
     +kron(idetB.*(-y12.*z14+z12.*y14).*(-x12.*z13+z12.*x13),Kyz)...
     +kron(idetB.*(-x13.*z14+z13.*x14).*(y12.*z13-z12.*y13),Kxz')...
     +kron(idetB.*(x12.*z14-z12.*x14).*(y12.*z13-z12.*y13),Kyz')...
     +kron(idetB.*(y12.*z13-z12.*y13).*(-x12.*z13+z12.*x13),Kzz);
dxdz=kron(idetB.*(y13.*z14-z13.*y14).*(x13.*y14-y13.*x14),Kxx)...
     +kron(idetB.*(y13.*z14-z13.*y14).*(-x12.*y14+y12.*x14),Kxy)...
     +kron(idetB.*(y13.*z14-z13.*y14).*(x12.*y13-y12.*x13),Kxz)...
     +kron(idetB.*(x13.*y14-y13.*x14).*(-y12.*z14+z12.*y14),Kxy')...
     +kron(idetB.*(-y12.*z14+z12.*y14).*(-x12.*y14+y12.*x14),Kyy)...
     +kron(idetB.*(-y12.*z14+z12.*y14).*(x12.*y13-y12.*x13),Kyz)...
     +kron(idetB.*(x13.*y14-y13.*x14).*(y12.*z13-z12.*y13),Kxz')...
     +kron(idetB.*(-x12.*y14+y12.*x14).*(y12.*z13-z12.*y13),Kyz')...
     +kron(idetB.*(y12.*z13-z12.*y13).*(x12.*y13-y12.*x13),Kzz);
dydz=kron(idetB.*(-x13.*z14+z13.*x14).*(x13.*y14-y13.*x14),Kxx)...
     +kron(idetB.*(-x13.*z14+z13.*x14).*(-x12.*y14+y12.*x14),Kxy)...
     +kron(idetB.*(-x13.*z14+z13.*x14).*(x12.*y13-y12.*x13),Kxz)...
     +kron(idetB.*(x13.*y14-y13.*x14).*(x12.*z14-z12.*x14),Kxy')...
     +kron(idetB.*(x12.*z14-z12.*x14).*(-x12.*y14+y12.*x14),Kyy)...
     +kron(idetB.*(x12.*z14-z12.*x14).*(x12.*y13-y12.*x13),Kyz)...
     +kron(idetB.*(x13.*y14-y13.*x14).*(-x12.*z13+z12.*x13),Kxz')...
     +kron(idetB.*(-x12.*y14+y12.*x14).*(-x12.*z13+z12.*x13),Kyz')...
     +kron(idetB.*(-x12.*z13+z12.*x13).*(x12.*y13-y12.*x13),Kzz);
```

**Local postprocessing** Once we have computed the solution $(\boldsymbol{q}_h, u_h, \widehat{u}_h) \in \boldsymbol{V}_h \times W_h \times M_h$ for the HDG equations, we can use some of the known superconvergence property to justify why (for $k \geq 1$) the following local postprocessing is superconvergent. We compute

$$u_h^\star \in \prod_{K\in\mathcal{T}_h} \mathcal{P}_{k+1}(K)$$

satisfying

$$\begin{array}{rcll}(\nabla u_h^\star, \nabla w_h)_K & = & -(\kappa^{-1}\boldsymbol{q}_h, \nabla w_h)_K & \qquad \forall w_h \in \mathcal{P}_{k+1}(K),\\ (u_h^\star, 1)_K & = & (u_h, 1)_K. & \end{array}$$

The HDG code provides decompositions:

$$u_h|_K = \sum_{j=1}^{d_3(k)} u_j^K P_j^K, \qquad q_{\#,h}|_K = \sum_{j=1}^{d_3(k)} q_{\#,j}^K P_j^K$$

We then use the fact that

$$\int_K D_1^K D_i^K = 0 \qquad \forall i \geq 2, \qquad \forall K \in \mathcal{T}_h,$$

to decompose

$$u_h^\star|_K = u_1^K P_1^K + \sum_{j=2}^{d_3(k+1)} w_j^K,$$

and write the system

$$\sum_{j=2}^{d_3(k+1)} \Big(\int_K \nabla P_j^K \cdot \nabla P_i^K\Big) w_j^K = -\sum_{\#\in\{x,y,z\}} \sum_{j=1}^{d_3(k)} \Big(\int_K \kappa^{-1} P_i^K\, \partial_\# P_j^K\Big) q_{\#,j}^K \qquad i = 2, \ldots, d_3(k+1).$$

Therefore, we only need to locally solve systems with the stiffness matrix (eliminating first row and columns) using variable convection matrices to build the right hand side. *This is the only place in the code where we explicitly use orthogonality properties of the polynomial basis.*

```
function uhstar=postprocessing(T,km,qhx,qhy,qhz,uh,k,formula)

% uhstar=postprocessing(T,km,qhx,qhy,qhz,uh,k,formula)
%
% Last modified: April 9, 2013


% Dimensions

d3plus=nchoosek(k+1+3,3);
d3    =nchoosek(k+3,3);
Nelts =size(T.elements,1);

% Matrices

S=Stiffness(T,k+1,formula);
S(1,:,:)=[];
S(:,1,:)=[];
[Cx,Cy,Cz]=VariableConv(T,km,km,km,k+1,formula);
Cx=permute(Cx,[2 1 3]);
Cy=permute(Cy,[2 1 3]);
Cz=permute(Cz,[2 1 3]);
Cx(:,d3+1:end,:)=[];
Cy(:,d3+1:end,:)=[];
Cz(:,d3+1:end,:)=[];
Cx(1,:,:)=[];
Cy(1,:,:)=[];
Cz(1,:,:)=[];

% Solution of local problems

uhstar=zeros(d3plus-1,Nelts);
parfor K=1:Nelts
    uhstar(:,K)=-S(:,:,K)\(Cx(:,:,K)*qhx(:,K)+...
                           Cy(:,:,K)*qhy(:,K)+...
                           Cz(:,:,K)*qhz(:,K));
end
uhstar=[uh(1,:); uhstar];
```

## 14 Convection-diffusion

The convection-diffusion equations are written as

$$\kappa^{-1}\boldsymbol{q} + \nabla u - 0, \qquad \text{and} \qquad \nabla\cdot(\boldsymbol{q} + u\,\boldsymbol{\beta}) + c\,u = f \qquad \text{in } \Omega.$$

On the Neumann boundary, the condition is given as $-(\boldsymbol{q}+u\boldsymbol{\beta})\cdot\boldsymbol{\nu} = \boldsymbol{u}_N\cdot\boldsymbol{\nu}$.
The local HDG equations are

$$\begin{aligned}
(\kappa^{-1}\boldsymbol{q}_h,\boldsymbol{r}_h)_K - (u_h,\nabla\cdot\boldsymbol{r}_h)_K + \langle\widehat{u}_h,\boldsymbol{r}_h\cdot\boldsymbol{\nu}\rangle_{\partial K} &= 0 & \forall \boldsymbol{r}_h\in\boldsymbol{V}_h,\\
(\nabla\cdot\boldsymbol{q}_h,w_h)_K - (u_h,\boldsymbol{\beta}\cdot\nabla w_h)_K + (cu_h,w_h)_K &\\
+\langle\tau(u_h-\widehat{u}_h)+(\boldsymbol{\beta}\cdot\boldsymbol{\nu})\widehat{u}_h,w_h\rangle_{\partial K} &= (f,w_h)_K & \forall w_h\in W_h.
\end{aligned}$$

The induced flux on the boundary is

$$\boldsymbol{q}_h\cdot\boldsymbol{\nu} + \tau(u_h-\widehat{u}_h) + (\boldsymbol{\beta}\cdot\boldsymbol{\nu})\widehat{u}_h.$$

Most of the derivations are straightforward extensions of the pure diffusion problem. The local solvers contain the $4d_3\times 4d_3\times N_{\mathrm{elt}}$ array

$$\mathbb{A}_1^K := \begin{bmatrix} \mathrm{M}^K_{\kappa^{-1}} & \mathrm{O} & \mathrm{O} & -(\mathrm{C}^K_x)^\top\\ \mathrm{O} & \mathrm{M}^K_{\kappa^{-1}} & \mathrm{O} & -(\mathrm{C}^K_y)^\top\\ \mathrm{O} & \mathrm{O} & \mathrm{M}^K_{\kappa^{-1}} & -(\mathrm{C}^K_z)^\top\\ \mathrm{C}^K_x & \mathrm{C}^K_y & \mathrm{C}^K_y & \mathrm{M}^K_c + \tau\mathrm{PP}^K - \sum_{\star\in\{x,y,z\}}\beta_\star\mathrm{C}^K_\star \end{bmatrix},$$

where $\beta_\star C_\star^K$ is the transpose of the variable convection matrix with entries

$$\int_K \beta_\star P_i^K \partial_x P_j^K \qquad i,j=1,\ldots,d_3.$$

The $4d_3 \times 4d_2 \times N_{\text{elt}}$ array with slices

$$\mathbb{A}_2^K := \left[\begin{array}{l} (n_x \mathrm{DP}^K)^\top \\ (n_y \mathrm{DP}^K)^\top \\ (n_z \mathrm{DP}^K)^\top \\ -(\tau \mathrm{DP}^K)^\top + \sum_{\star\in\{x,y,z\}} (n_\star \beta_\star \mathrm{DP}^K)^\top \end{array}\right],$$

where $n_\star\beta_\star\mathrm{DP}^K$ is the matrix with entries

$$\int_{e_\ell^K} \beta_\star \nu_\star D_i^{e_\ell^K} P_j^K \qquad i=1,\ldots,d_2, \quad j=1,\ldots,d_3, \quad \ell\in\{1,2,3,4\}.$$

The matrices $\mathbb{A}_3^K$ and $\mathbb{A}_f^K$ are the same as in the purely diffusive problem. Finally, the local solvers include the $4d_2 \times 4d_2 \times N_{\text{elt}}$ array with slices

$$\mathbb{C}^K := \mathbb{A}_3^K (\mathbb{A}_1^K)^{-1} \mathbb{A}_2^K + \tau \mathrm{DD}^K - \sum_{\star\in\{x,y,z\}} n_\star \beta_\star \mathrm{DD}^K,$$

where $n_\star\beta_\star\mathrm{DD}^K$ has elements

$$\int_{e_\ell^K} \beta_\star \nu_\star D_i^{e_\ell^K} D_j^{e_\ell^K} \qquad i,j=1,\ldots,d_2, \quad \ell\in\{1,2,3,4\}.$$

The local solver related to source terms $\mathbb{C}_f^K$ are defined with the same formulas as in the diffusive case. Everything else is taken verbatim from the diffusive HDG code.

```
function [C,Cf,A1,A2,Af]=localsolvers3dCD(km,c,f,beta,tau,T,k,formulas)

% [C,Cf,A1,A2,Af]=localsolvers3dCD(km,c,f,{betax,betay,betaz},...
%                                  tau,T,k,{for1,for2,for3,for4})
%
%Input:
%      km, c, f: vectorized functions of three variables
% {betax,betay,betaz}: vectorized functions of three variables
%           tau: penalization parameter for HDG (Nelts x 4)
%             T: expanded tetrahedrization
%             k: polynomial degree
%          for1: quadrature formula 3d (for mass matrix)
%          for2: quadrature formula 3d (for convection matrices)
%          for3: quadrature formula 2d
%          for4: quadrature formula 2d (variable coefficients and errors)
%
%Output:
%             C:   4*d2 x 4*d2 x Nelts
%            Cf:   4*d2 x Nelts
%            A1:   4*d3 x 4*d3 x Nelts
%            A2:   4*d3 x 4*d2 x Nelts
%            Af:   4*d3 x Nelts
%
%Last modified: April 11, 2013

Nelts=size(T.elements,1);
d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);
```

```
f=testElem(f,T,k,formulas{1});
Af=zeros(4*d3,Nelts);
Af(3*d3+1:4*d3,:)=f;

Mass=MassMatrix(T,{km,c},k,formulas{1});
Mk=Mass{1};Mc=Mass{2};
[Cx,Cy,Cz]=ConvMatrix(T,k,formulas{2});
[convx,convy,convz]=VariableConv(T,beta{1},beta{2},beta{3},k,formulas{1});
convbeta=permute(convx+convy+convz,[2 1 3]);

[tauPP,tauDP,nxDP,nyDP,nzDP,tauDD]=matricesFace(T,tau,k,formulas{3});
nx=T.normals(:,[1 4 7 10])';
ny=T.normals(:,[2 5 8 11])';
nz=T.normals(:,[3 6 9 12])';
bnDP=matricesVariableFaceA(T,beta,{nx,ny,nz},k,formulas{4});
bnDP=bnDP{1}+bnDP{2}+bnDP{3};
bnDD=matricesVariableFaceB(T,beta,{nx,ny,nz},k,formulas{4});
bnDD=bnDD{1}+bnDD{2}+bnDD{3};

O=zeros(d3,d3,Nelts);
A1=[Mk        ,O        ,O        ,-permute(Cx,[2 1 3]);...
    O         ,Mk       ,O        ,-permute(Cy,[2 1 3]);...
    O         ,O        ,Mk       ,-permute(Cz,[2 1 3]);...
    Cx        ,Cy       ,Cz       ,Mc+tauPP-convbeta];
A2=[permute(nxDP,[2 1 3]);...
    permute(nyDP,[2 1 3]);...
    permute(nzDP,[2 1 3]);...
    -permute(tauDP,[2 1 3])+permute(bnDP,[2 1 3])];

C=zeros(4*d2,4*d2,Nelts);
Cf=zeros(4*d2,Nelts);

parfor i=1:Nelts
    C(:,:,i)=[nxDP(:,:,i) nyDP(:,:,i) nzDP(:,:,i) tauDP(:,:,i)]/...
             A1(:,:,i)*A2(:,:,i)+tauDD(:,:,i)-bnDD(:,:,i);
    Cf(:,i) =[nxDP(:,:,i) nyDP(:,:,i) nzDP(:,:,i) tauDP(:,:,i)]/...
             A1(:,:,i)*Af(:,i);
end
return
```

```
function [Uh,Qxh,Qyh,Qzh,Uhat,system,solvers]=...
          HDG3dCD(km,c,f,beta,tau,T,k,formulas,uD,gx,gy,gz,varargin)

%[Uh,Qxh,Qyh,Qzh,Uhat]=HDG3dCD(km,c,f,{bx,by,bz},...
%                       tau,T,k,formulas,uD,gx,gy,gz)
%[Uh,Qxh,Qyh,Qzh,Uhat]=HDG3dCD(km,c,f,{bx,by,bz},...
%                       tau,T,k,formulas,uD,gx,gy,gz,0)
%[¬,¬,¬,¬,¬,system,solvers]=HDG3dCD(km,c,f,{bx,by,bz},...
%                       tau,T,k,formulas,uD,gx,gy,gz,1)
%
%Input:
%        km   : vectorized function (kappa^{-1}; kappa=diffusion parameter)
%        c    : vectorized function (reaction parameter)
%        f    : vectorized function (source term)
%  {bx,by,bz} : vectorized functions (convection field)
%        tau  : penalization parameter for HDG (Nelts x 4)
%        T    : expanded tetrahedrization
%        k    : polynomial degree
%     formulas: {for1,for2,for3,for4}
%               (quadrature formulas as used by localsolvers3dCD)
%        uD   : Dirichlet data; vectorized function
%    gx,gy,gz : Neumann data (corresponds to kappa*grad(u)); vectorized fns
%
%Output:
%      Uh     : d3 x Nelts,   matrix with uh
%      Qxh    : d3 x Nelts,   matrix with qxh
%      Qyh    : d3 x Nelts,   matrix with qyh
%      Qzh    : d3 x Nelts,   matrix with qzh
%      Uhat   : d2 x Nelts    matrix with uhat
%    system   : {HDGmatrix,HDGrhs,list of free d.o.f.,list of dir d.o.f.}
%    solvers  : {A1,A2,Af}    local solvers
%
%Last modified: April 11, 2013

if nargin==13
    export=varargin{1};
else
    export=0;
end

d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);
block3=@(x) (1+(x-1)*d3):(x*d3);
Nelts =size(T.elements,1);
Nfaces=size(T.faces,1);
Ndir  =size(T.dirichlet,1);
Nneu  =size(T.neumann,1);

%Matrices for assembly process

face=T.facebyele';    % 4 x Nelts
face=(face(:)-1)*d2;  % First degree of freedom of each face by element
face=bsxfun(@plus,face,1:d2);    %4*Nelts x d2 (d.o.f. for each face)
face=reshape(face',4*d2,Nelts);  %d.o.f. for the 4 faces of each element

[J,I]=meshgrid(1:4*d2);
R=face(I(:),:); R=reshape(R,4*d2,4*d2,Nelts);
C=face(J(:),:); C=reshape(C,4*d2,4*d2,Nelts);
      % R_ij^K d.o.f. for local (i,j) d.o.f. in element K ; R_ij^K=C_ji^K
RowsRHS=reshape(face,4*d2*Nelts,1);

dirfaces=(T.dirfaces(:)-1)*d2;
dirfaces=bsxfun(@plus,dirfaces,1:d2);
dirfaces=reshape(dirfaces',d2*Ndir,1);

free=((1:Nfaces)'-1)*d2;
free=bsxfun(@plus,free,1:d2);
free=reshape(free',d2*Nfaces,1);
free(dirfaces)=[];

neufaces=(T.neufaces(:)-1)*d2;
neufaces=bsxfun(@plus,neufaces,1:d2);
neufaces=reshape(neufaces',d2*Nneu,1);

%Local solvers and global system

[M1,Cf,A1,A2,Af]=localsolvers3dCD(km,c,f,beta,tau,T,k,formulas);
M=sparse(R(:),C(:),M1(:));
phif=accumarray(RowsRHS,Cf(:));

[uhatD,qhatN]=BC3d(uD,gx,gy,gz,T,k,formulas{3});

%Dirichlet BC
Uhatv=zeros(d2*Nfaces,1);
Uhatv(dirfaces)=uhatD;       %uhat stored as a vector: d2*Nfaces

%RHS
rhs=zeros(d2*Nfaces,1);
rhs(free)=phif(free);
qhatN=reshape(qhatN,d2*Nneu,1);   % qhatN stored as a vector: d2*Nneu
rhs(neufaces)=rhs(neufaces)+qhatN;
```

```
rhs=rhs-M(:,dirfaces)*Uhatv(dirfaces);

if export
    system={M,rhs,free,dirfaces};
    solvers={A1,A2,Af};
    Uh=[];
    Qxh=[];
    Qyh=[];
    Qzh=[];
    Uhat=[];
    return
else
    system=[];
    solvers=[];
end

Uhatv(free)=M(free,free)\rhs(free);
Uhat=reshape(Uhatv,d2,Nfaces);

%Uh Qxh Qyh Qzh

faces=T.facebyele'; faces=faces(:);
uhhataux=reshape(Uhat(:,faces),[4*d2,Nelts]);
sol=zeros(4*d3,Nelts);
parfor K=1:Nelts
    sol(:,K)=A1(:,:,K)\(Af(:,K)-A2(:,:,K)*uhhataux(:,K));
end

Qxh=sol(block3(1),:);
Qyh=sol(block3(2),:);
Qzh=sol(block3(3),:);
Uh =sol(block3(4),:);
return
```

## 15 Linear elasticity with strong symmetric stress

### 15.1 The system

Let $\mathcal{T}_h$ be a standard tetrahedrization of the polyhedron $\Omega$ with boundary $\Gamma$, and let $P_k(K,S)$ be the space of symmetric matrices of polynomials over an element $K$. Define the following spaces:

$$V_h := \prod_{K\in\mathcal{T}_h} P_k(K,S) \qquad W_h := \prod_{K\in\mathcal{T}_h} (P_k(K))^3 \qquad M_h^0 := \prod_{F\in\partial\mathcal{T}_h\setminus\Gamma} (P_k(F))^3 \qquad M_h^\Gamma := \prod_{F\in\Gamma} (P_k(F))^3$$

as well as $M_h = M_h^0 \oplus M_h^\Gamma$. Let $\Gamma_D$ be the Dirichlet boundary and $\Gamma_N$ the Neumann boundary. Also, suppose we have two Lamé parameters $\lambda, \mu > 0$, which may vary in space. Set

$$c_1 = \frac{\mu}{2} \qquad \text{and} \qquad c_2 = -\frac{\lambda}{2\mu(2\mu+3\lambda)}.$$

The HDG formulation for linear elasticity (for Hookean materials, with strong symmetric stresses) is then the following:
For some body force function $f : \Omega \to \mathbb{R}^3$, some a Dirichlet boundary condition $u_0 : \Gamma_D \to \mathbb{R}^3$, and a Neumann boundary condition $g : \Gamma_N \to \mathbb{R}^3$, and further defining the operator

$$\mathcal{A} : \sigma_h \mapsto c_1\sigma_h + c_2\mathrm{trace}(\sigma_h)$$

find $(\sigma_h, u_h, \hat{u}_h) \in V_h \times W_h \times M_h$ such that $\sigma_h n_F = g$ for all $F \in \Gamma_N$ and

$$\begin{aligned}
(\mathcal{A}\sigma_h, v)_{\mathcal{T}_h} + (u_h, \nabla \cdot v)_{\mathcal{T}_h} - \langle \hat{u}_h, vn \rangle_{\partial \mathcal{T}_h} &= 0 & \forall v \in V_h \qquad (1)\\
-(\nabla \cdot \sigma_h, w)_{\mathcal{T}_h} + \langle \tau(u_h - \hat{u}_h), w \rangle_{\partial \mathcal{T}_h} &= (f, w)_{\mathcal{T}_h} & \forall w \in W_h \qquad (2)\\
\langle \sigma_h n + \tau(u_h - \hat{u}_h), \mu \rangle_{\partial \mathcal{T}_h \setminus \Gamma} &= 0 & \forall \mu \in M_h^o \qquad (3)\\
\langle \hat{u}_h, \mu \rangle_{\Gamma_D} &= \langle u_0, \mu \rangle_{\Gamma_D} & \forall \mu \in M_h^{\Gamma} \qquad (4)
\end{aligned}$$

## 15.2 Matrix components and local solvers

In order to yield the matrices for the larger system, we need the following components, which may be taken from the functions already given in the above sections. This is the list of the necessary elementwise and/or facewise slices with dimensions written on the right:

$$\begin{aligned}
\mathtt{Mc1}_{ij}^K &= \int_K c_1 P_i^K P_j^K & d_3 \times d_3 \times N_{\text{elt}}\\
\mathtt{Mc2}_{ij}^K &= \int_K c_2 P_i^K P_j^K & d_3 \times d_3 \times N_{\text{elt}}\\
\mathtt{Cx}_{ij}^K &= \int_K P_i^K \partial_x P_j^K & d_3 \times d_3 \times N_{\text{elt}}\\
\mathtt{Cy}_{ij}^K &= \int_K P_i^K \partial_y P_j^K & d_3 \times d_3 \times N_{\text{elt}}\\
\mathtt{Cz}_{ij}^K &= \int_K P_i^K \partial_z P_j^K & d_3 \times d_3 \times N_{\text{elt}}\\
\mathtt{tauPP}_{ij}^K &= \int_{\partial K} \tau P_i^K P_j^K & d_3 \times d_3 \times N_{\text{elt}}\\
\mathtt{tauDP}_{ij}^K &= \int_{\partial K} \tau R_i^K P_j^K & 4d_2 \times d_3 \times N_{\text{elt}}\\
\mathtt{tauDD}_{ij}^K &= \int_{\partial K} \tau R_i^K R_j^K & 4d_2 \times 4d_2 \times N_{\text{elt}}\\
\mathtt{nxDP}_{ij}^F &= \int_F \tau n_x R_i^K P_j^K & 4d_2 \times d_3 \times N_{\text{fc}}\\
\mathtt{nyDP}_{ij}^F &= \int_F \tau n_y R_i^K P_j^K & 4d_2 \times d_3 \times N_{\text{fc}}\\
\mathtt{nzDP}_{ij}^F &= \int_F \tau n_z R_i^K P_j^K & 4d_2 \times d_3 \times N_{\text{fc}}
\end{aligned}$$

where $\{P_i\}_1^{d_3}$ is a basis for $\mathcal{P}_k(K)$ and $\{R_i\}_1^{4d_2}$ is a basis for $\mathcal{R}_k(\partial K) = \prod_{F \in \mathcal{E}(K)} \mathcal{P}(F)$.
We may therefore assemble the elementwise-slices:

$$\mathtt{A11} = \begin{bmatrix} \mathtt{Mc1}+\mathtt{Mc2} &&&&& \\ & \mathtt{Mc1}+\mathtt{Mc2} &&&& \\ && \mathtt{Mc1}+\mathtt{Mc2} &&& \\ &&& 2\mathtt{Mc1} && \\ &&&& 2\mathtt{Mc1} & \\ &&&&& 2\mathtt{Mc1} \end{bmatrix} \qquad \mathtt{A12} = \begin{bmatrix} \mathtt{Cx} && \\ & \mathtt{Cy} & \\ && \mathtt{Cz} \\ \mathtt{Cy} & \mathtt{Cx} & \\ \mathtt{Cz} && \mathtt{Cx} \\ & \mathtt{Cz} & \mathtt{Cy} \end{bmatrix}$$

$$\mathtt{A21} = -\mathtt{A12}^T = \begin{bmatrix} -\mathtt{Cx} &&& -\mathtt{Cy} & -\mathtt{Cz} & \\ & -\mathtt{Cy} && -\mathtt{Cx} && -\mathtt{Cz} \\ && -\mathtt{Cz} && -\mathtt{Cx} & -\mathtt{Cy} \end{bmatrix} \qquad \mathtt{A22} = \begin{bmatrix} \mathtt{tauPP} && \\ & \mathtt{tauPP} & \\ && \mathtt{tauPP} \end{bmatrix}$$

Taking $\mathtt{A} = \begin{bmatrix} \mathtt{A11} & \mathtt{A12} \\ \mathtt{A21} & \mathtt{A22} \end{bmatrix}$, we note for clarity that the expression $\mathtt{A}(\sigma_h^{xx}, \sigma_h^{yy}, \sigma_h^{zz}, \sigma_h^{xy}, \sigma_h^{xz}, \sigma_h^{yz}, u_h^x, u_h^y, u_h^z)^T$ corresponds precisely to setting $\hat{u} = 0$ in the left-hand sides of equations 1 and 2 when stacked and expanded in terms of the basis functions. Keeping this in mind, we then assemble the face-related matrices $\mathtt{B}, \mathtt{C}, \mathtt{D}$

$$\mathtt{A} = \begin{bmatrix} \mathtt{A11} & \mathtt{A12} \\ \mathtt{A21} & \mathtt{A22} \end{bmatrix} \qquad \mathtt{B} = \begin{bmatrix} -\mathtt{nxDP} & & \\ & -\mathtt{nyDP} & \\ & & -\mathtt{nzDP} \\ -\mathtt{nyDP} & -\mathtt{nxDP} & \\ -\mathtt{nzDP} & & -\mathtt{nxDP} \\ & -\mathtt{nzDP} & -\mathtt{nyDP} \\ -\mathtt{tauDP} & & \\ & -\mathtt{tauDP} & \\ & & -\mathtt{tauDP} \end{bmatrix}$$

$$\mathtt{C} = \begin{bmatrix} \mathtt{nxDP} & & & \mathtt{nyDP} & \mathtt{nzDP} & & -\mathtt{tauDP} & & \\ & \mathtt{nyDP} & & \mathtt{nxDP} & & \mathtt{nzDP} & & -\mathtt{tauDP} & \\ & & \mathtt{nzDP} & & \mathtt{nxDP} & \mathtt{nyDP} & & & -\mathtt{tauDP} \end{bmatrix} \quad \mathtt{D} = \begin{bmatrix} \mathtt{tauDD} & & \\ & \mathtt{tauDD} & \\ & & \mathtt{tauDD} \end{bmatrix}$$

Letting $\mathtt{F}$ be $(0, 0, [F_1, F_2, F_3]^T)^T$ where $F_1, F_2, F_3$ are test values of the components $f^x, f^y, f^z$ of the body force function $f$, we now solve 1 and 2 as if $\hat{u} = 0$ on all faces, i.e.

$$\begin{aligned} (\mathcal{A}\sigma_h^f, v)_{\mathcal{T}_h} + (u_h^f, \nabla \cdot v)_{\mathcal{T}_h} &= 0 & \forall v \in V_h \\ -(\nabla \cdot \sigma_h^f, w)_{\mathcal{T}_h} + \langle \tau u_h^f, w \rangle_{\partial \mathcal{T}_h} &= (f, w)_{\mathcal{T}_h} & \forall w \in W_h \end{aligned}$$

Considering $\alpha^f = (\sigma_h^{xx,f}, \sigma_h^{yy,f}, \sigma_h^{zz,f}, \sigma_h^{xy,f}, \sigma_h^{xz,f}, \sigma_h^{yz,f}, u_h^{x,f}, u_h^{y,f}, u_h^{z,f})^T$ as the vector of unknowns that solves this problem, note that the previous problem is exactly equivalent to

$$\mathtt{A}\alpha^f = \mathtt{F}.$$

Therefore $\alpha^f = \mathtt{A}^{-1}\mathtt{F}$ and so, taking this on the faces, we define the flux due to sources $\mathtt{phiF} = C\alpha^f = \mathtt{CA}^{-1}\mathtt{F}$, and moreover note that this can be given as a purely local quantity. A list of the vectors **`phiF`** for each element is one output of **`localsolversElasticity3d.m`**.
The last piece of the local solvers puzzle is the displacement-to-stress operator $\mathtt{G}$. Now suppose $f = 0$ in 1 and 2 and consider that the system to be solved is now

$$\begin{aligned} (\mathcal{A}\sigma_h, v)_{\mathcal{T}_h} + (u_h, \nabla \cdot v)_{\mathcal{T}_h} - \langle \hat{u}_h, vn \rangle_{\partial \mathcal{T}_h} &= 0 & \forall v \in V_h \\ -(\nabla \cdot \sigma_h, w)_{\mathcal{T}_h} + \langle \tau(u_h - \hat{u}_h), w \rangle_{\partial \mathcal{T}_h} &= 0 & \forall w \in W_h \end{aligned}$$

Since the system is determined once $\hat{u}_h$ is fixed, we may consider the solution operators $\mathcal{L}^\sigma : \hat{u}_h \mapsto \sigma_h^L$ and $\mathcal{L}^u : \hat{u}_h \mapsto u_h^L$. Indeed, collecting the components of $\hat{u}_h$ and defining

$$\alpha^L = (\sigma_h^{xx,L}, \sigma_h^{yy,L}, \sigma_h^{zz,L}, \sigma_h^{xy,L}, \sigma_h^{xz,L}, \sigma_h^{yz,L}, u_h^{x,L}, u_h^{y,L}, u_h^{z,L})^T$$

we see that the above equations correspond to $\mathtt{A}\alpha^L = -\mathtt{B}\hat{u}_h$, or equivalently $\alpha^L = -\mathtt{A}^{-1}\mathtt{B}\hat{u}_h$. Thus, if we want a matrix $\mathtt{G}$ to represent the displacement-to-flux operator $\phi_h : \hat{u}_h \mapsto \hat{\sigma}_h^L n = \sigma_h^L - \tau(u_h^L - \hat{u}_h)$, note that $-\mathtt{CA}^{-1}\mathtt{B}\hat{u}_h = \sigma_h^L n - \tau u_h^L$, hence

$$\mathtt{G}\hat{u}_h = (-\mathtt{CA}^{-1}\mathtt{B} + \mathtt{D})\hat{u}_h = \sigma_h^L n - \tau(u_h - \hat{u}_h).$$

The following code outputs the quantities `A, B, C, D, F, phiF, G` for use in solving the global system for $\hat{u}$. It should be noted that $\mathtt{A}^{-1}$ is never computed, bypassing the computation using MATLAB's **`mrdivide`**.

```
function [A,B,C,D,F,phiF,G]=...
    localsolversElasticity3d(f,lambda,mu,tau,T,k,formulas)
```

```
% function [A11,A12,A21,A22,C1,C2]=...
%     localsolversElasticity3d(f,lambda,mu,tau,T,k,formulas)
%
% input:
%       f         : the source function for the problem
%       lambda    : first Lame parameter, function of spacial variable
%       mu        : second Lame parameter, function of spacial variable
%       tau       : T.N_elt x 4 matrix of penalization terms
%       T         : Tetrahedrization
%       k         : Polynomial degree
%       formulas  : 1-D array of quadrature formulas for approx integrals
% output:
%       A     : [ A11 A12 ]
%               [ A21 A22 ]   where
%               A11: 6*d3 x 6*d3 x Nelt (test stress sigma)
%               A12: 6*d3 x 3*d3 x Nelt (test displacement u)
%               A21: 3*d3 x 6*d3 x Nelt (test div sigma) = -A12*
%               A22: 3*d3 x 3*d3 x Nelt (test tau (u - uhat))
%       B     : [ B1 ]
%               [ B2 ] where
%               B1: 6*d3 x 3*(4*d2) x Nelt = -C1*
%               B2: 3*d3 x 3*(4*d2) x Nelt = C2*
%       C     : [ C1 C2 ]
%               C1: 3*(4*d2) x 6*d3 x Nelt (flux matrix 1) = -B1*
%               C2: 3*(4*d2) x 3*d3 x Nelt (flux matrix 2) = B2*
%       D     : 3*(4*d2) x 3*(4*d2) x Nelt (tau face product matrix)
%       F     : 4*(3*d3) x Nelt (test f on elements in 4th block only)
%       phiF  : flux for muhat=0, namely phiF = [ C A^-1 F ]
%       G     : displacement-to-stress operator for f = 0,
%                   namely G = [ D - C A^-1 B ]
% Last modified: July 14, 2015

%necessary sizes
Nelt=size(T.elements,1);
d3=nchoosek(k+3,3);
d2=nchoosek(k+2,2);

%define constants for inverted constitutative law:
c1 = @(x,y,z) 0.5./mu(x,y,z);
c2 = @(x,y,z) -0.5*lambda(x,y,z)...
                    ./(mu(x,y,z).*(2*mu(x,y,z)+3*lambda(x,y,z)));

%pull all mass matrices corresponding to "full" coeffs
formula = formulas{1};
M = MassMatrix(T,{c1,c2},k,formula);
Mc1=M{1};
Mc2=M{2};

%shave some time
TwoMc1=2*Mc1;

%make room for A11, a 6dim(P_k) x 6dim(P_k) x N_elt matrix
O=zeros(d3,d3,Nelt);

A11= [ Mc1 O O O O O;...
       O Mc1 O O O O;...
       O O Mc1 O O O;...
       O O O TwoMc1 O O;...
       O O O O TwoMc1 O;...
       O O O O O TwoMc1];

%add this to the top left of A11
A11(1:3*d3,1:3*d3,:)=A11(1:3*d3,1:3*d3,:)+repmat(Mc2,3,3);

%pull all convection matrices
```

```
[Cx,Cy,Cz]=ConvMatrix(T,k,formula);

%build negative A21
A21= [Cx O O Cy Cz O;...
      O Cy O Cx O Cz;...
      O O Cz O Cx Cy];

%transpose to get A12:
A12=permute(A21,[2 1 3]);

%turn negative A21 into A21
A21=-A21;

%pull face-related matrices
formula=formulas{3};
[tauPP,tauDP,nxDP,nyDP,nzDP,tauDD]=matricesFace(T,tau',k,formula);

%assembling A22:
A22= [tauPP O O;...
      O tauPP O;...
      O O tauPP];

%assemble the full A now that we have it
A=[A11 A12;...
   A21 A22];

%assembling C1 and C2:
O=zeros(4*d2,d3,Nelt);

C1= [ nxDP  O       O       nyDP    nzDP    O;...
      O     nyDP    O       nxDP    O       nzDP;...
      O     O       nzDP    O       nxDP    nyDP];

C2= [ tauDP O       O;...
      O     tauDP   O;...
      O     O       tauDP];

%assembling B and C
B=[-permute(C1,[2 1 3]);...
   -permute(C2,[2 1 3])];
C=[C1 -C2];

%assembling D:
O=zeros(4*d2,4*d2,Nelt);
D= [tauDD O O;...
    O tauDD O;...
    O O tauDD];

%test f on the elements
formula=formulas{1};
f=testElem(f,T,k,formula);
O=zeros(3*d3,Nelt);
F= [O;O;f{1};f{2};f{3}];

%calculating phiF and G
for n = 1:Nelt
    G(:,:,n)    = -C(:,:,n)/A(:,:,n)*B(:,:,n)+D(:,:,n);
    phiF(:,:,n)=  C(:,:,n)/A(:,:,n)*F(:,n);
end
```

## 15.3 Assembly, boundary conditions, and the global system

We turn to the now-decoupled equations 3 and 4 to solve the system only for $\hat{u}$, which may be written as

$$\begin{aligned} \langle \phi_h(\hat{u}_h) + \phi_h^f, \mu\rangle_{\partial\mathcal{T}_h\setminus\Gamma} &= 0 & \forall \mu \in M_h^o \\ \langle \hat{u}_h, \mu\rangle_{\Gamma_D} &= \langle u_0, \mu\rangle_{\Gamma_D} & \forall \mu \in M_h^\Gamma, \end{aligned}$$

along with fixed Neumann conditions, after which may define the numerical solution $(\sigma_h, u_h)^T = \alpha = -\mathtt{A}^{-1}\mathtt{B}\hat{u}_h$.
To solve this, our matrix $G$ is expanded into a $\dim V_h \times \dim V_h$ matrix $\mathtt{M}$.
Second, we first test the Dirichlet and Neumann conditions against our basis–this is done three times, exactly as it is done once with HDG3D using `BC3d.m`. Two matrices are created, namely

$$\mathtt{uhatD} = \begin{bmatrix} \left(\int_F u_0^x R_i\right)_{i=1}^{d_2} \\ \left(\int_F u_0^y R_i\right)_{i=1}^{d_2} \\ \left(\int_F u_0^z R_i\right)_{i=1}^{d_2} \end{bmatrix}, F \in \Gamma_D, (3d_2 \times N_{\text{dir}}), \qquad \text{and}$$

$$\mathtt{beta} = \begin{bmatrix} \left(\int_F (\sigma_h^{xx}, \sigma_h^{xy}, \sigma_h^{xz}) \cdot nR_i\right)_{i=1}^{d_2} \\ \left(\int_F (\sigma_h^{xy}, \sigma_h^{yy}, \sigma_h^{yz}) \cdot nR_i\right)_{i=1}^{d_2} \\ \left(\int_F (\sigma_h^{xz}, \sigma_h^{yz}, \sigma_h^{zz}) \cdot nR_i\right)_{i=1}^{d_2} \end{bmatrix}, F \in \Gamma_N, (3d_2 \times N_{\text{neu}})$$

Next we create a $3d_3$ vector `Uhat` and set it to the values of `uhatD` on all of the Dirichlet faces. Then, a vector `rhs` of the same size is created. Free faces receive the values of −`phiF`, and the Neumann faces also accumulate the values of −`beta`. Lastly for the right-hand side, the Dirichlet conditions are subtracted from `rhs`.
If we think of rearranging $\mathtt{M} = [\mathtt{G}; I]$ by rearranging the system into [free;dirichlet] parts, then the system is equivalent to:

$$\mathtt{M}\hat{u} \sim \begin{bmatrix} \mathtt{G} \\ \mathtt{I} \end{bmatrix} \hat{u} = \begin{bmatrix} -\mathtt{phiF} \\ u_0 \end{bmatrix}.$$

This is solved via the line `M(free,free)` \ `rhs(free)`. Finally, a $12d_2$ length vector `Uhataux` is created, which is a listing of the values of `Uhat` in the order of the face of the element it belongs to, so that it can be handled elementwise in parallel.
Finally, the values of $\sigma_h$ and $u_h$ as

$$\alpha = (\sigma_h^{xx}, \sigma_h^{yy}, \sigma_h^{zz}, \sigma_h^{xy}, \sigma_h^{xz}, \sigma_h^{yz}, u_h^x, u_h^y, u_h^z)^T$$

are recovered locally (in parallel, if selected), using the fact that the first equations 1 and 2 may be expressed as

$$\mathtt{A}\alpha + \mathtt{B}\hat{u} = F \qquad \Longleftrightarrow \qquad \alpha = \mathtt{A}^{-1}(\mathtt{F} - \mathtt{B}\hat{u}).$$

This is the process covered by `HDGElasticity3d.m`

```
function [Uhx,Uhy,Uhz,Sigmaxx,Sigmayy,Sigmazz,Sigmaxy,Sigmaxz,Sigmayz,...
          qhxn,qhyn,qhzn,beta,Uhataux]=...
    HDGElasticity3d(ux,uy,uz,sigmaxx,sigmayy,sigmazz,sigmaxy,sigmaxz,sigmayz,...
         f,lambda,mu,tau,T,k,formulas,parallel)

% [Uhx,Uhy,Uhz,Sigmaxx,Sigmayy,Sigmazz,Sigmaxy,Sigmaxz,Sigmayz,qhxn,qhyn,qhzn,beta]=...
%     HDGElasticity3d(ux,uy,uz,sigmaxx,sigmayy,sigmazz,sigmaxy,sigmaxz,sigmayz,...
%          f,lambda,mu,tau,T,k,formulas,parallel)
%Input:
```

```
% : Dirichlet data for boundary displacement (vec funcs):
%          ux      : first component
%          uy      : second component
%          uz      : third component
%        sigma''   : Neumann boundary stress (vec funcs)
%                    ordered: xx,yy,zz,xy,xz,yz
%
%        f         : the source function for the problem
%        lambda    : first Lame parameter, function of spacial variable
%        mu        : second Lame parameter, function of spacial variable
%        tau       : T.N_elt x 4 matrix of penalization terms
%        T         : Tetrahedrization
%        k         : Polynomial degree
%        formulas  : 1-D array of quadrature formulas for approx integrals
%        parallel  : loop in parallel?  1 yes; no otherwise.
%
%Output:
%       Uhx     : d3 x Nelts,   matrix with uhx values
%       Uhy     : d3 x Nelts,   matrix with uhy values
%       Uhz     : d3 x Nelts,   matrix with uhz values
%       Sigmaxx : d3 x Nelts,   matrix with Sigmaxx values
%       Sigmayy : d3 x Nelts,   matrix with Sigmayy values
%       Sigmazz : d3 x Nelts,   matrix with Sigmazz values
%       Sigmaxy : d3 x Nelts,   matrix with Sigmaxy values
%       Sigmaxz : d3 x Nelts,   matrix with Sigmaxz values
%       Sigmayz : d3 x Nelts,   matrix with Sigmayz values
%       qhxn    : d2 x Nneu,    matrix of Neumann tests (x component)
%       qhyn    : d2 x Nneu,    matrix of Neumann tests (y component)
%       qhzn    : d2 x Nneu,    matrix of Neumann tests (z component)
%
%Last modified: July 21, 2015

%important sizes
d2=nchoosek(k+2,2);
d3=nchoosek(k+3,3);
Nelt =size(T.elements,1);
Nfc=size(T.faces,1);
Ndir=size(T.dirichlet,1);
Nneu=size(T.neumann,1);
dim=d2*Nfc;

%Index matrices for assembly process
    %now transpose T.facebyele and fill each number with its DoF vector
face=T.facebyele';
face=(face(:)-1)*d2;
face=bsxfun(@plus,face,1:d2);
face=reshape(face',4*d2,Nelt);

Dof=[face;face+dim;face+2*dim];
Row=repmat(Dof,[12*d2,1]);
Row=reshape(Row,[12*d2,12*d2,Nelt]);

%transpose for C
Col=permute(Row,[2 1 3]);

%get our local solvers
[A,B,¬,¬,F,phiF,G]=...
    localsolversElasticity3d(f,lambda,mu,tau,T,k,formulas);

%assembly of LHS
M=sparse(Row(:),Col(:),G(:));

%RHS:
phiF=accumarray(Dof(:),phiF(:));

%list DoF for dir faces:
```

```
dirfaces=(T.dirfaces(:)-1)*d2;
dirfaces=bsxfun(@plus,dirfaces,1:d2);
dirfaces=reshape(dirfaces',d2*Ndir,1);
dirfaces=[dirfaces;dirfaces+dim;dirfaces+2*dim];

%list DoF for neu faces:
free=(1:3*dim)';  %triple the unknowns  (this too)
free(dirfaces)=[];
neufaces=(T.neufaces(:)-1)*d2;
neufaces=bsxfun(@plus,neufaces,1:d2);
neufaces=reshape(neufaces',d2*Nneu,1);
neufaces=[neufaces;neufaces+dim;neufaces+2*dim];

%Dirichlet + Neumann testing
formula=formulas{3};
[uhxd,qhxn]=BC3d(ux,sigmaxx,sigmaxy,sigmaxz,T,k,formula);
[uhyd,qhyn]=BC3d(uy,sigmaxy,sigmayy,sigmayz,T,k,formula);
[uhzd,qhzn]=BC3d(uz,sigmaxz,sigmayz,sigmazz,T,k,formula);

uhatD=[uhxd(:);uhyd(:);uhzd(:)];
beta=[qhxn(:);qhyn(:);qhzn(:)];

%Dirichlet BC
Uhat=zeros(3*dim,1);
Uhat(dirfaces)=uhatD;

%RHS
rhs=zeros(3*dim,1);
rhs(free)=-phiF(free);
rhs(neufaces)=rhs(neufaces)-beta;
rhs=rhs-M(:,dirfaces)*Uhat(dirfaces);

Uhat(free)=M(free,free)\rhs(free);
Uhat=reshape(Uhat,d2,Nfc*3);

%Uh Qhx Qhy Qhz
solution=zeros(9*d3,Nelt);
Uhataux=zeros(12*d2,Nelt);
TwoNfc=2*Nfc;
for i =1:Nelt
    Uhataux(:,i)=[  Uhat(:,T.facebyele(i,1));...
                    Uhat(:,T.facebyele(i,2));...
                    Uhat(:,T.facebyele(i,3));...
                    Uhat(:,T.facebyele(i,4));...
                    Uhat(:,T.facebyele(i,1)+Nfc);...
                    Uhat(:,T.facebyele(i,2)+Nfc);...
                    Uhat(:,T.facebyele(i,3)+Nfc);...
                    Uhat(:,T.facebyele(i,4)+Nfc);...
                    Uhat(:,T.facebyele(i,1)+TwoNfc);...
                    Uhat(:,T.facebyele(i,2)+TwoNfc);...
                    Uhat(:,T.facebyele(i,3)+TwoNfc);...
                    Uhat(:,T.facebyele(i,4)+TwoNfc)];
end

if parallel %then loop in parallel
    parfor i=1:Nelt
        solution(:,i)=A(:,:,i)\(F(:,i)-B(:,:,i)*Uhataux(:,i));
    end
else % don't
    for i=1:Nelt
        solution(:,i)=A(:,:,i)\(F(:,i)-B(:,:,i)*Uhataux(:,i));
    end
end

Sigmaxx=solution(1:d3,:);
Sigmayy=solution(d3+1:2*d3,:);
```

```
Sigmazz=solution(2*d3+1:3*d3,:);
Sigmaxy=solution(3*d3+1:4*d3,:);
Sigmaxz=solution(4*d3+1:5*d3,:);
Sigmayz=solution(5*d3+1:6*d3,:);
Uhx=solution(6*d3+1:7*d3,:);
Uhy=solution(7*d3+1:8*d3,:);
Uhz=solution(8*d3+1:9*d3,:);

return
```

### 15.4 Elasticity with multiple solutions and/or multiple forcing functions

Included as well are the functions `testFelasticityQV`, `localsolversElasticity3dQV`, and `HDGElasticity3dQV`. The QV stands for quick-vary, in that the three functions are designed to perform the same tasks as `localsolversElasticity3d` and `HDGElasticity3d` for multiple arrays of exact solutions and forcing functions without losing efficiency by re-calculating many things. The code is essentially the same as the above, however, as the forcing functions and boundary conditions are now arrays, so some things must be handled differently. Firstly, note that `f, u1, u2, u3, sigmaxx, sigmayy, sigmazz, sigmaxy, sigmaxz, sigmayz` must be arrays of the same size.
Secondly, the idea is to run `localsolversElasticity3dQV` to get the matrices `A,B,C,D,` and `G` from the previous sections, none of which depend on forcing functions or boundary conditions, and then separately (and, if possible, in parallel) give the values of `F` and `phiF` using `testFelasticityQV`. These are combined simply in `HDGElasticity3dQV`.

## 16 Quadrature rules

The scripts and `TableQuadForm.m` include `TablesQuadForm3d.m` matrices with quadrature formulas in the reference elements in two and three dimensions respectively. The are stored as $N_{\rm qd} \times 5$ matrices in the three dimensional case and as $N_{\rm qd2} \times 4$ matrices in the two dimensional case. The collection of formulas that are stored in those files is given in Tables 1 and 2. Once the polynomial degree $k$ is chosen, the function `checkQuadrature3d` chooses formulas for all two and three dimensional integrals. We avoid using the two dimensional quadrature formulas that use nodes on the edges of the reference triangle.

| name | degree of precision | number of quadrature nodes = $N_{\rm qd}$ |
|---|---|---|
| tetra1 | 1 | 1 |
| tetra3 | 2 | 4 |
| tetra5 | 5 | 14 |
| tetra7 | 6 | 25 |
| tetra9 | 8 | 45 |

Table 1: Quadrature formulas on $\widehat{K}$

```
function formulas=checkQuadrature3d(k,constantmass)

%formulas=checkQuadrature3d(k,constantmass)
%Imput:
%              k: degree of polynomials
%   constantmass: 1 constant coefficients in mass matrices
%                 0 non-costant
%Output:
%       formulas{1} : 3d quadrature formula for errors and var coeff
%       formulas{2} : 3d quadrature formula for constant coefficients
%       formulas{3} : 2d quadrature formula
%       formulas{4} : 2d quadrature formula for errors
```

| name | degree of precision | number of nodes $N_{\mathrm{qd2}}$ | nodes on edges |
|---|---|---|---|
| matrix0 | 1 | 1 | No |
| matrix2 | 2 | 3 | No |
| matrix4 | 4 | 6 | No |
| matrix5 | 5 | 10 | Yes |
| matrix7 | 7 | 15 | Yes |
| matrix9 | 9 | 21 | No |
| matrix11 | 11 | 28 | Yes |
| matrix13 | 13 | 36 | No |
| matrix14 | 14 | 45 | Yes |
| matrix16 | 16 | 55 | Yes |
| matrix18 | 18 | 66 | No |
| matrix20 | 20 | 78 | No |
| matrix21 | 21 | 91 | No |
| matrix23 | 23 | 105 | No |
| matrix25 | 25 | 120 | Yes |

Table 2: Quadrature formulas on $\widehat{K}_2$

```
%Last update: March 14, 2013

% degrees = {3k,2k,2k,2k+2} if constant mass = 0
%           {2k,2k,2k,2k+2} if constant mass = 1
% for k=0, take {2,0,0,2}

TablesQuadForm3d
TablesQuadForm
switch constantmass
    case 1
        switch k
            case 0
                formulas={tetra5,tetra1,matrix0,matrix4};
            case 1
                formulas={tetra3,tetra3,matrix4,matrix4};
            case 2
                formulas={tetra7,tetra7,matrix9,matrix9};
            case 3
                formulas={tetra9,tetra9,matrix11,matrix11};
        end
    case 0
        switch k
            case 0
                formulas={tetra5,tetra1,matrix0,matrix4};
            case 1
                formulas={tetra5,tetra3,matrix4,matrix4};
            case 2
                formulas={tetra7,tetra7,matrix9,matrix9};
            case 3
                formulas={tetra9,tetra9,matrix11,matrix11};
        end
end
formulas{3}(:,4)=formulas{3}(:,4)/2;
formulas{4}(:,4)=formulas{4}(:,4)/2;
return
```

## 17 Main programs and their dependences

**Routines needed to start working.** Note that implementation of the Dubiner polynomials in two dimensions uses code for the Jacobi polynomials as a subfunction. This has been adapted from a piece of code by John Burkardt, distributed under hte GNU LGPL license. Quadrature rules have been pre-stored in the form that is needed for the code. Instead of having an m–file with the formulas, there is a script that defines them one by one, and a function that chooses the ones that are going to be used for a given polynomial degree. This is done only once, so this part of the code can be easily modified. The three dimensional formulas that have been included can deal with all cases up to $k = 3$. For higher order, once can easily construct more formulas and modify the corresponding files.
**Update: Quadrature rules have been stored up to $k = 3$, but `computeQuadrature.m` may be used to compute Stroud quadrature rules of arbitrary order.

- `HDGgrid3d`
- `createTau3d`
- `checkQuadrature3d`
  - `TablesQuadFrom3d`
  - `TablesQuadForm`
- ** `computeQuadrature`
  - `quadratureHDG`

**Main functions.**

- `HDG3d`
  - `localsolvers3d`
    - `MassMatrix`
      - `dubiner3d`
    - `ConvMatrix`
      - `dubiner3d`
    - `matricesFace`
      - `dubiner2d`
      - `dubiner3d`
  - `BC3d`
    - `dubiner2d`
- `HDG3dCD`
  - `localsolvers3dCD`
    - `MassMatrix`
      - `dubiner3d`
    - `ConvMatrix`
      - `dubiner3d`
    - `VariableConv`
      - `dubiner3d`
    - `matricesVariableFaceA`
      - `dubiner2d`

            - dubiner3d
        - matricesVariableFaceB
            - dubiner2d
    - BC3d
        - dubiner2d

- HDGElasticity3d
    - localsolversElasticity3d
        - MassMatrix
            - dubiner3d
        - ConvMatrix
            - dubiner3d
        - VariableConv
            - dubiner3d
        - matricesFace
            - dubiner2d
            - dubiner3d

**Projections and error functions for testing**

- L2proj3d
    - dubiner3d
    - errorElem
        - dubiner3d
- L2projskeleton3d
    - dubiner2d
    - errorFaces
        - dubiner2d
- projectionHDG3d
    - MassMatrix
        - dubiner3d
    - matricesFace
        - dubiner2d
        - dubiner3d
    - testElem
        - dubiner3d
    - testFaces
        - dubiner2d
- errorElem
    - dubiner3d
- errorFaces

  - dubiner2d
- postprocessing
  - variableConv
    - dubiner3d
  - Stiffness
    - dubiner3d

## 18 Alphabetical list of all programs

**Main programs.**

- BC3d
- checkQuadrature3d
- ConvMatrix
- createTau3d
- dubiner2d
- dubiner3d
- errorElem
- errorFaces
- HDG3d
- HDG3dCD
- HDGElasticity3d
- HDGgrid3d
- L2proj3d
- L2projskeleton3d
- localsolvers3d
- localsolvers3dCD
- localsolversElasticity3d
- MassMatrix
- matricesFace
- matricesVariableFaceA
- matricesVariableFaceB
- postprocessing
- projectHDG3d
- Stiffness

- `TablesQuadForm`
- `TablesQuadForm3d`
- `testElem`
- `testFaces`
- `VariableConv`

**Examples of fully developed meshes**

- `4Tchimney`
- `Corner`
- `FicheraCorner1`
- `FicheraCorner2`
- `sixT3dDir`

**Scripts**

- `Script_HDG3DCD`
- `scriptHDG3dpaper`
- `scriptHDGElasticity3d`