Shushu Qin

FEF HDG Poisson

This project presents the implementation of the HDG method for the poisson problem with Dirichlet, Neumann and Robin boundary conditions. Firstly, the strong and weak forms of the local and the global problems are derived. Secondly, the implementation of the Matlab codes is explained. At last the numerical studies are carried out with boundary conditions to reproduce a particular analytical solution.

The equation we want to solve is defined

$$\begin{cases} -\boldsymbol{\nabla}\cdot(\kappa\boldsymbol{\nabla}u) = s & \text{in } \Omega, \\ u = u_D & \text{on } \Gamma_D \\ \boldsymbol{n}\cdot(\kappa\boldsymbol{\nabla}u) = t & \text{on } \Gamma_N \\ \boldsymbol{n}\cdot(\kappa\boldsymbol{\nabla}u) + \gamma u = g & \text{on } \Gamma_R \end{cases} \tag{1}$$

In this example (Example 6), κ and γ is set be constants, 4 and 2.5 respectively. The analytical solution of u is $\log(a + \sin^2(\kappa\pi x)) + by^3$ with $a = 2$ and $b = 3$. The boundary conditions is shown in Figure 1.

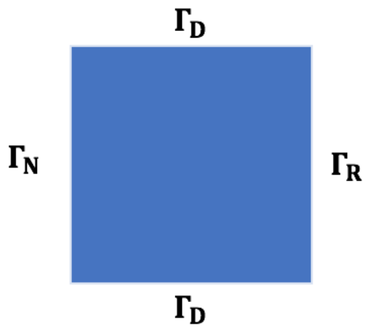


Figure 1: Boundary conditions

1 HDG formulation

The strong form written with mixed form over the broken computational domain reads as

$$\begin{cases} \boldsymbol{\nabla}\cdot\boldsymbol{q} = s & \text{in } \Omega_i, \text{ and for } i = 1,\dots,\texttt{n}_{\texttt{el}} \\ \boldsymbol{q} + \kappa\boldsymbol{\nabla}u = \boldsymbol{0} & \text{in } \Omega_i, \text{ and for } i = 1,\dots,\texttt{n}_{\texttt{el}} \\ u = u_D & \text{on } \Gamma_D \\ \boldsymbol{n}\cdot\boldsymbol{q} = -t & \text{on } \Gamma_N \\ \boldsymbol{n}\cdot\boldsymbol{q} - \gamma u = g & \text{on } \Gamma_R \\ [\![u\boldsymbol{n}]\!] = 0 & \text{on } \Gamma \\ [\![\boldsymbol{n}\cdot\boldsymbol{q}]\!] = 0 & \text{on } \Gamma \end{cases} \tag{2}$$

This can be rewrite as two equivalent problems. Firstly, the local problem

$$\begin{cases} \boldsymbol{\nabla} \cdot \boldsymbol{q}_i = s & \text{in } \Omega_i \\ \boldsymbol{q}_i + \kappa \boldsymbol{\nabla} u_i = \boldsymbol{0} & \text{in } \Omega_i \\ u = u_D & \text{on } \Gamma_D \\ u = \hat{u} & \text{on } \partial\Omega_i \setminus \Gamma_D \end{cases} \tag{3}$$

The global problem is defined as

$$\begin{cases} [\![u\boldsymbol{n}]\!] = 0 & \text{on } \Gamma \\ [\![\boldsymbol{n} \cdot \boldsymbol{q}]\!] = 0 & \text{on } \Gamma \\ \boldsymbol{n} \cdot \boldsymbol{q} = -t & \text{on } \Gamma_N \\ \boldsymbol{n} \cdot \boldsymbol{q} - \gamma \hat{u} = -g & \text{on } \Gamma_R \end{cases} \tag{4}$$

The first equation is automatically satisfied as $u = \hat{u}$ on Γ. Therefore, the local problem is simplified as

$$\begin{cases} [\![\boldsymbol{n} \cdot \boldsymbol{q}]\!] = 0 & \text{on } \Gamma \\ \boldsymbol{n} \cdot \boldsymbol{q} = -t & \text{on } \Gamma_N \\ \boldsymbol{n} \cdot \boldsymbol{q} - \gamma \hat{u} = -g & \text{on } \Gamma_R \end{cases} \tag{5}$$

After integrating by parts and substituting numerical fluxes with

$$\boldsymbol{n}_i \cdot \hat{\boldsymbol{q}}_i := \begin{cases} \boldsymbol{n}_i \cdot \boldsymbol{q}_i + \tau_i (u_i - u_D), & \text{on } \Omega_i \cap \Gamma_D \\ \boldsymbol{n}_i \cdot \boldsymbol{q}_i + \tau_i (u_i - \hat{u}), & \text{elsewhere} \end{cases} \tag{6}$$

The weak form of the local problem is the similar to that in [1] in the slides except that in the second equation the viscosity parameter is not equal to 1. In the following part, the superscript h is dropped out for readability.

$$\big\langle v, \tau_i u_i \big\rangle_{\partial\Omega_i} - \big(\boldsymbol{\nabla} v_i, \boldsymbol{q}_i\big)_{\Omega_i} + \big\langle v, \boldsymbol{n}_i \cdot \boldsymbol{q}_i \big\rangle_{\partial\Omega_i} = \big(v, s\big)_{\Omega_i} + \big\langle v, \tau_i u_D \big\rangle_{\Omega_i \cap \Gamma_D} + \big\langle v, \tau_i \hat{u} \big\rangle_{\Omega_i \setminus \Gamma_D} \tag{7a}$$

$$-\big(\boldsymbol{w}, \boldsymbol{q}_i\big)_{\Omega_i} + \kappa\big(\boldsymbol{\nabla} \cdot \boldsymbol{w}, u_i\big)_{\Omega_i} = \kappa\big\langle \boldsymbol{n}_i \cdot \boldsymbol{w}, u_D \big\rangle_{\Omega_i \cap \Gamma_D} + \kappa\big\langle \boldsymbol{n}_i \cdot \boldsymbol{w}, \hat{u} \big\rangle_{\Omega_i \setminus \Gamma_D} \tag{7b}$$

We can apply integrating by parts again on the second term of Equation 7a. This leads to the following local problem:

$$\big\langle v, \tau_i u_i \big\rangle_{\partial\Omega_i} + \big(v_i, \boldsymbol{\nabla} \cdot \boldsymbol{q}_i\big)_{\Omega_i} = \big(v, s\big)_{\Omega_i} + \big\langle v, \tau_i u_D \big\rangle_{\Omega_i \cap \Gamma_D} + \big\langle v, \tau_i \hat{u} \big\rangle_{\Omega_i \setminus \Gamma_D} \tag{8a}$$

$$-\big(\boldsymbol{w}, \boldsymbol{q}_i\big)_{\Omega_i} + \kappa\big(\boldsymbol{\nabla} \cdot \boldsymbol{w}, u_i\big)_{\Omega_i} = \kappa\big\langle \boldsymbol{n}_i \boldsymbol{w}, u_D \big\rangle_{\Omega_i \cap \Gamma_D} + \kappa\big\langle \boldsymbol{n}_i \boldsymbol{w}, \hat{u} \big\rangle_{\Omega_i \setminus \Gamma_D} \tag{8b}$$

The weak form the global problem is similar to that in [1]. We only need to add the Robin boundary condition, which gives us.

$$\sum_{e=1}^{\mathtt{n_{el}}} \big\langle \mu, \boldsymbol{n}_i \cdot \hat{\boldsymbol{q}} \big\rangle_{\partial\Omega_i \setminus \partial\Omega} + \sum_{e=1}^{\mathtt{n_{el}}} \big\langle \mu, \boldsymbol{n}_i \cdot \hat{\boldsymbol{q}} + t \big\rangle_{\partial\Omega_i \cap \Gamma_N} + \sum_{e=1}^{\mathtt{n_{el}}} \big\langle \mu, \boldsymbol{n}_i \cdot \hat{\boldsymbol{q}} - \gamma \hat{u} + g \big\rangle_{\partial\Omega_i \cap \Gamma_R} = 0 \tag{9}$$

Therefore, the final global weak form is

$$\sum_{e=1}^{\mathtt{n_{el}}}\Big\{\big\langle\mu,\boldsymbol{n}_i\cdot\boldsymbol{q}\big\rangle_{\partial\Omega_i\backslash\Gamma_D}+\big\langle\mu,\tau_i u_i\big\rangle_{\partial\Omega_i\backslash\Gamma_D}-\big\langle\mu\tau_i\hat{u}\big\rangle_{\partial\Omega_i\backslash\Gamma_D}-\big\langle\mu,\gamma\hat{u}\big\rangle_{\partial\Omega_i\backslash\Gamma_R} \\ =-\sum_{e=1}^{\mathtt{n_{el}}}\Big\{\big\langle\mu,t\big\rangle_{\partial\Omega_i\cap\Gamma_N}+\big\langle\mu,g\big\rangle_{\partial\Omega_i\cap\Gamma_R}\Big\} \tag{10}$$

2 Implementation in the MATLAB code

Several parts need to be changed in order to simulate this problem

1. Degree of freedom of the unknowns should be calculated according to the global face ID of the Dirichlet boundaries. (See appendix Degree of unknows)
2. In the function GetFaces, we add one more row to extFaces in order to store the information of face types of the external faces (See appendix Get faces).
3. In the function hdg_preprocess, we add one more variable called facetypes and define 0 for Dirichlet, 1 for Neumann, 2 for Robin, 3 for internal faces (See appendix Preprocess).
4. In the funtion hdgMatrixPoisson, we need to compute the matrix of global system ($\mathbf{A}_{\hat{u}\hat{u}}$ and $\mathbf{f}_{\hat{u}}$) differently with additional Neumann and Robin boundary conditions (See appendix hdgMatrixPoisson).
5. In the postprocess (See appendix postprocess), the viscosity is added because the equations for postprocess becomes:

$$\begin{aligned}-\boldsymbol{\nabla}\cdot(\kappa\boldsymbol{\nabla}u^\star)&=\boldsymbol{\nabla}\cdot\boldsymbol{q} &&\text{in }\Omega_i\\ \kappa\boldsymbol{n}\cdot\boldsymbol{\nabla}u^\star&=\boldsymbol{n}\cdot\boldsymbol{q} &&\text{on }\partial\Omega_i\\ (u^\star,1)_{\Omega_i}&=(u,1)_{\Omega_i} &&\text{in }\Omega_i\end{aligned}$$

6. The code for computing analytical solution, source conditions the boundary conditions of the problem is added to the folder *poisson*.
7. A file for computing the $\mathcal{L}_2$ norm of the error of $\boldsymbol{q}$ is added. It is similar to the file. The only change is that the variable is changed from scalar to vector. So instead of compute the square of variable, the inner product of the vector is computed in the contribution of the current integration point to the elemental $\mathcal{L}_2$ norm.

3 Determine the analytical solution of the data

The analytical solution is calculated from the following codes:

```
syms  kappa gamma a b pi x y
u = log(a+(sin(kappa*pi*x))^2)+b*y^3;
du_x = diff(u,x)
du_y = diff(u,y)
s = -kappa*(diff(du_x,x)+diff(du_y,y))
t = -kappa*du_x
g = kappa*du_x+gamma*u
```

u_D is defined the same as the analytical solution.
The source term is

$$\begin{aligned} s = -\kappa(6by + (2\kappa^2\pi^2\cos^2(\pi\kappa x))/(a+\sin^2(\pi\kappa x)) - \\ (2\kappa^2\pi^2\sin^2(\pi\kappa x))/(a+\sin^2(\pi\kappa x)) \\ -(4\kappa^2\pi^2\sin^2(\pi\kappa x)\cos^2(\pi\kappa x))/(a+\sin^2(\kappa x\pi))) \end{aligned} \tag{11}$$

The traction term is

$$t = -(2\kappa^2\pi\sin(\pi\kappa x)\cos(\pi\kappa x))/(a+\sin^2(\pi\kappa x)) \tag{12}$$

The Robin term is

$$\begin{aligned} g = \gamma(log(a+\sin^2(\pi\kappa x)) + by^3) \\ + (2\kappa^2\pi\sin(\pi\kappa x)cos(\pi\kappa x))/(a+\sin^2(\pi\kappa x)) \end{aligned} \tag{13}$$

4 Accuracy of u and $u^\star$

The analytical solution of u is shown in Figure 2. As we can see from Figure 3 and Table 1, the accuracy of the postprocessed solution $u^\star$ is higher than that of the primal solution u with the same mesh and degree of approximation. The error of both variables decreases if we increase the degree of approximation or refine the mesh.

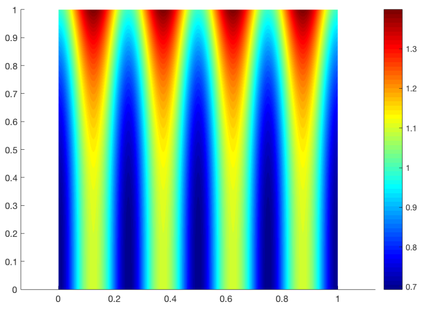
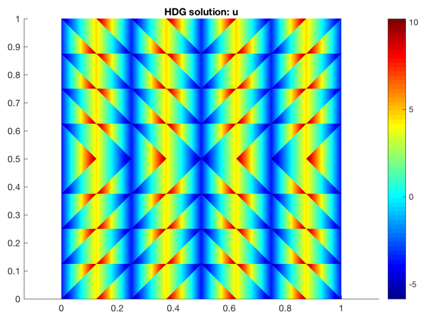
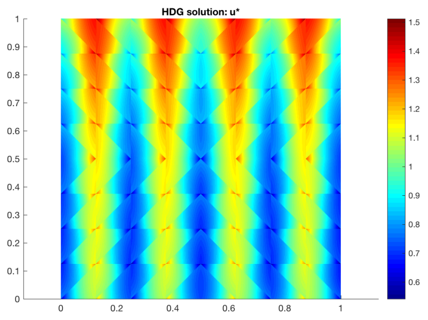


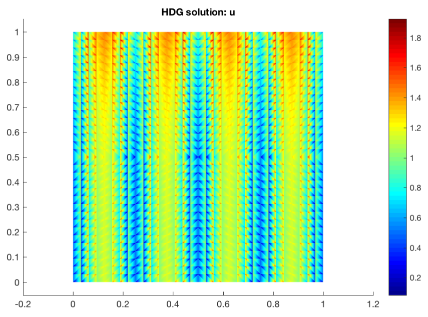
Figure 2: Analytical solution of u



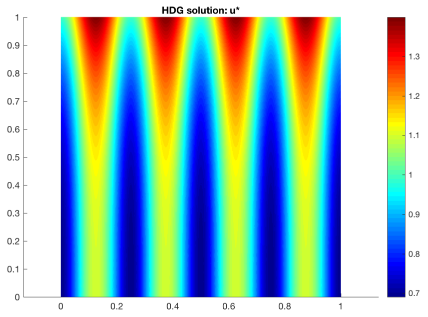
(a) mesh3 degree1 u



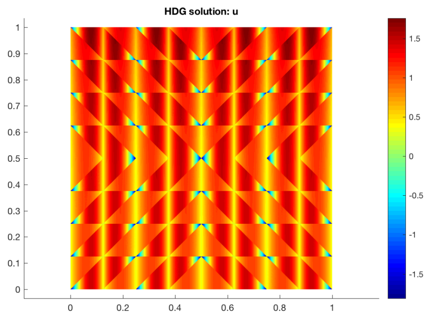
(b) mesh3 degree1 $u^\star$



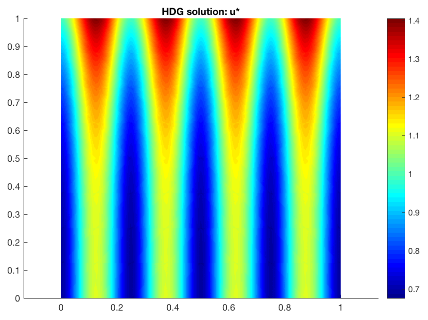
(c) mesh5 degree1 u



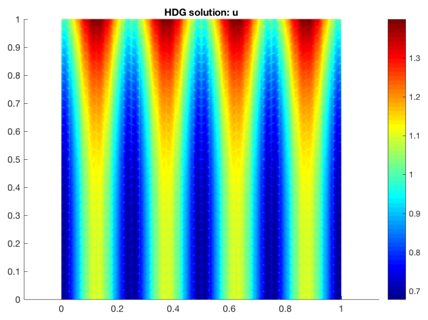
(d) mesh5 degree1 $u^\star$



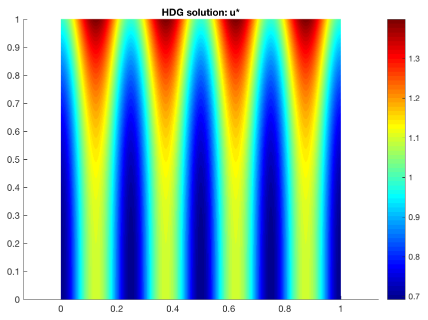
(e) mesh3 degree2 u



(f) mesh3 degree2 $u^\star$



(g) mesh5 degree2 u



(h) mesh5 degree2 $u^\star$

$Mesh$	degree	$\|u - u^h\|_{\mathcal{L}_2(\Omega)}$	$\|u^\star - u^{\star h}\|_{\mathcal{L}_2(\Omega)}$
3	1	2.732381e+00	2.860039e-02
5	1	1.626039e-01	4.343540e-04
3	2	3.572434e-01	2.750809e-03
5	2	1.256886e-02	2.012913e-05

Table 1: $\mathcal{L}_2$ error of u and $u^\star$

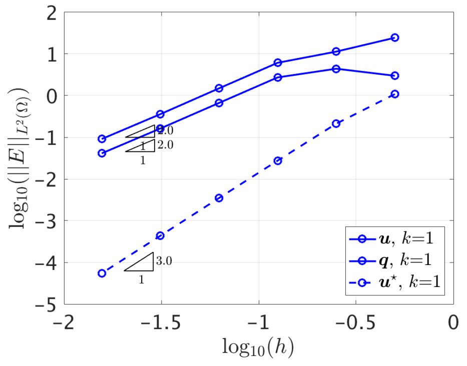
5 Convergence rate of HDG

The optimal convergence rate is studied with the $\mathcal{L}_2(\Omega)$ norm of the error, which is defined as

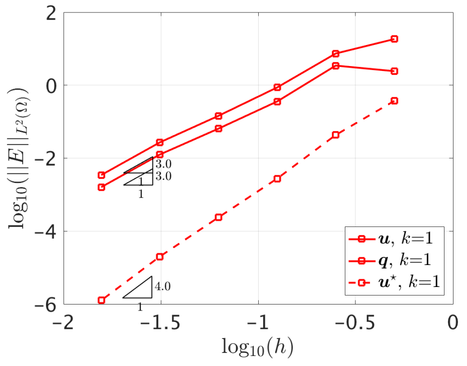
$$\|E_\omega\|_{\mathcal{L}_2(\Omega} = \left\{ \frac{\int_\Omega (\omega^h - \omega) \cdot (\omega^h - \omega) d\Omega}{\int_\Omega \omega \cdot \omega d\Omega} \right\}^{1/2} \tag{14}$$

where ω is the exact solution and ω^h is the numerical approximation. In order calculate the error for $\boldsymbol{q}$ in the $\mathcal{L}_2$-norm defined on the domain Ω, we introduce one more function qPoisson to calculate the analytical solution of $\boldsymbol{q}$ and computeL2NormQ for calculating the error. It is similar to the function computeL2Norm the only changes is in the contribution of the current integration point. It is changed to the inner product of the vector $\boldsymbol{q}$.

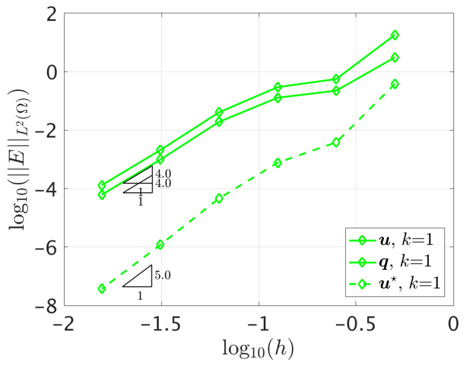
Figure 4 shows the convergence rate of the $\mathcal{L}_2$ norm error of u, $\boldsymbol{q}$ and $u^\star$ for a polynomial degree of approximation $k = 1, 2, 3, 4$. For all the degrees of approximation considered, the optimal convergence rate (i.e. k+1) of u and $\boldsymbol{q}$ is obtained. The results also illustrate the benefits of using high-order approximations. For instance, similar accuracy 10^{-1} in u is obtained with the 6th level mesh for $k = 1$ while 4th level mesh for $k = 3$. Moreover, super convergence rate $k + 2$ is obtained for the postprocessed variable $u^\star$. The substantial gain in accuracy introduced by the postprocessing technique is also illustrated in the figure.



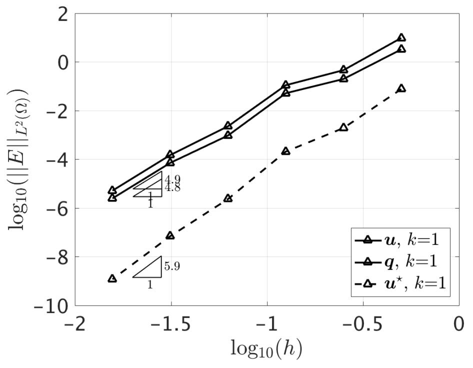
(a) k=1



(b) k=2



(c) k=3



(d) k=4

Figure 4: Convergence rate

Appendix

Degree of unknowns

```
%Dirichlet BC
%Dirichlet face nodal coordinates
nOfFaceNodes = degree+1;
nOfInteriorFaces = size(infoFaces.intFaces,1);
nOfExteriorFaces = size(infoFaces.extFaces,1);
aux0 = 1:nOfFaceNodes;
aux1 = find(infoFaces.faceTypes==0); % Dirichlet face IDs
aux2 = find(infoFaces.faceTypes~=0); % Other face IDs
nOfDirichletFaces = size(aux1,1);
nOfNonDirichletFaces = nOfExteriorFaces-nOfDirichletFaces;
dofDirichlet = zeros(nOfDirichletFaces*nOfFaceNodes,1);
for i=1:size(aux1,1)
    dofDirichlet(1+(i-1)*nOfFaceNodes:i*nOfFaceNodes) = (aux1(i)-1)*
   nOfFaceNodes + aux0;
```

```
end
dofUnknown  = zeros((nOfNonDirichletFaces+nOfInteriorFaces)*nOfFaceNodes,1);
for i=1:size(aux2,1)
    dofUnknown(1+(i-1)*nOfFaceNodes:i*nOfFaceNodes) = (aux2(i)-1)*nOfFaceNodes
    + aux0;
end
```

Get faces

```
function [intFaces,extFaces] = GetFaces(X,T)
%
% [intFaces,extFaces] = GetFaces(T)
% (only for triangles and tetrahedrons)
%
% For every face i:
% intFaces(i,:)=[element1 nface1 element2 nface2 node1] for interior faces
% extFaces(i,:)=[element1 nface1 typeOfFaces] for exterior faces
% typeOfFaces: 0 for Dirichlet, 1 for Neumann, 2 for Robin
% element1, element2:   number of the elements
% nface1, nface2:       number of face in each element
% node1:  number of node in the 2nd element that matches with the 1st node
%         in the 1st element (with the numbering of the face)
%
% Input:
% T: connectivity of the mesh
%
% Output:
% intFaces,extFaces: interior and exterior faces
%

[nElem,nen]=size(T);
nfaceel = nen;

nNodes = max(max(T));
N = zeros(nNodes,10);
nn = ones(nNodes,1);
for ielem = 1:size(T,1)
    Te = T(ielem,:);
    nn_Te = nn(Te);
    for kk = 1:3
        N(Te(kk),nn_Te(kk)) = ielem;
    end
    nn(Te) = nn(Te) +1;
end
N(:,max(nn):end) = [];

markE = zeros(nElem,nfaceel);
intFaces = zeros(3/2*size(T,1),5);
extFaces = zeros(size(T,1),3);

%Definition of the faces in the reference element
```

```
switch nen
    case 3, %triangle
        Efaces = [1 2; 2 3; 3 1];
    case 4, %tetrahedra
        error('Tetrahedra not implemented yet');
end
intF = 1;
extF = 1;
for iElem=1:nElem
    for iFace=1:nfaceel
        if(markE(iElem,iFace)==0)
            markE(iElem,iFace)=1;
            nodesf = T(iElem,Efaces(iFace,:));

            jelem = FindElem(iElem,nodesf);

            if(jelem~=0)
                [jface,node1]=FindFace(nodesf,T(jelem,:),Efaces);
                intFaces(intF,:)=[iElem,iFace,jelem,jface,node1];
                intF = intF +1;
                markE(jelem,jface)=1;
            else
                xx = X(nodesf,1);
                yy = X(nodesf,2);
                if xx(1)==0 && xx(2)==0 % Nuemann faces
                    extFaces(extF,:)=[iElem,iFace,1];
                elseif xx(1)==1 && xx(2)==1 % Robin faces
                    extFaces(extF,:)=[iElem,iFace,2];
                else   % Dirichlet faces
                    extFaces(extF,:)=[iElem,iFace,0];
                end
                extF = extF + 1;
            end
        end
    end
end

intFaces = intFaces(intFaces(:,1)~=0,:);
extFaces = extFaces(extFaces(:,1)~=0,:);

%Auxiliar functions
    function jelem = FindElem(iElem,nodesf)

        nen = length(nodesf);

        % [elems,aux] = find(T==nodesf(1));
        elems = N(nodesf(1),(N(nodesf(1),:)~=0));
        elems=elems(elems~=iElem);
        Ti=T(elems,:);
        for i=2:nen
```

```
            if(~isempty(elems))
                [aux,aux2] = find(Ti==nodesf(i));
                elems = elems(aux);
                Ti=Ti(aux,:);
            end
        end

        if(isempty(elems))
            jelem=0;
        else
            jelem=elems(1);
        end

    end

    function [jface,node1]=FindFace(nodesf,nodesE,Efaces)

        nFaces = size(Efaces,1);
        for j=1:nFaces
            nodesj = nodesE(Efaces(j,:));
            if (nodesj(1)==nodesf(1) || nodesj(1)==nodesf(2)) && ...
                    (nodesj(2)==nodesf(1) || nodesj(2)==nodesf(2))
                jface = j;
                node1 = find(nodesj==(nodesf(1)));
                break;
            end
        end

    end

end
```

Preprocess

```
function [F infoFaces] = hdg_preprocess(X,T)

% create infoFaces
[intFaces extFaces] = GetFaces(X,T(:,1:3));

nOfElements = size(T,1);
nOfInteriorFaces = size(intFaces,1);
nOfExteriorFaces = size(extFaces,1);
faceTypes = zeros(nOfInteriorFaces+nOfExteriorFaces,1);
F = zeros(nOfElements,3);
for iFace = 1:nOfInteriorFaces
    infoFace = intFaces(iFace,:);
    F(infoFace(1),infoFace(2)) = iFace;
    F(infoFace(3),infoFace(4)) = iFace;
    faceTypes(iFace) = 3;
end

```

```
for iFace = 1:nOfExteriorFaces
    infoFace = extFaces(iFace,:);
    F(infoFace(1),infoFace(2)) = iFace + nOfInteriorFaces;
    faceTypes(iFace+ nOfInteriorFaces) = infoFace(3);
end

infoFaces.intFaces = intFaces;
infoFaces.extFaces = extFaces;
infoFaces.faceTypes = faceTypes;
```

hdgMatrixPoisson

```
function [KK, f, QQ, UU, Qf, Uf] = hdgMatrixPoisson(muElem,X,T,F,
    referenceElement,infoFaces,tau)

nOfFaces = max(max(F));
nOfElements = size(T,1);
nOfInteriorFaces = size(infoFaces.intFaces,1);
nOfFaceNodes = size(referenceElement.NodesCoord1d,1);
nDOF = nOfFaces*nOfFaceNodes;
f = zeros(nDOF,1);
QQ = cell(nOfElements,1); UU = cell(nOfElements,1); Qf = cell(nOfElements,1);
    Uf = cell(nOfElements,1);

% loop in elements
indK = 1; n = nOfElements*(3*nOfFaceNodes)^2; ind_i  = zeros(1,n); ind_j  =
    zeros(1,n); coef_K = zeros(1,n);
for iElem = 1:nOfElements
    Te = T(iElem,:);
    Xe = X(Te,:);
    Fe = F(iElem,:); %Global face ID of iElem
    % Get information of the type of the face
    isFeNeumann = (infoFaces.faceTypes(Fe) == 1);
    isFeRobin = (infoFaces.faceTypes(Fe) == 2);
    isFeInterior = (infoFaces.faceTypes(Fe) == 3); %Boolean (1 for yes)
    % elemental matrices
    [Qe,Ue,Qfe,Ufe,Alq,Alu,All,fl] = KKeElementalMatricesIsoParametric(muElem(
    iElem),Xe,referenceElement,tau(iElem,:),isFeNeumann,isFeRobin);

    % Interior faces seen from the second element are flipped to have
    % proper orientation
    flipFace = zeros(1,3); %Boolean (1=face to be flipped)
    flipFace(isFeInterior)=any(infoFaces.intFaces(Fe(isFeInterior),[1 3])<
    iElem,2);
    indL=1:3*nOfFaceNodes;
    aux=nOfFaceNodes:-1:1; indflip=[aux,nOfFaceNodes+aux,2*nOfFaceNodes+aux];
    aux=ones(1,nOfFaceNodes); aux = [flipFace(1)*aux, flipFace(2)*aux,
    flipFace(3)*aux];
    indL(aux==1)=indflip(aux==1); %permutation for local numbering

    Qe=Qe(:,indL);     Ue=Ue(:,indL);
    Alq=Alq(indL,:);   Alu=Alu(indL,:);    All=All(indL,indL);

    %The local problem solver is stored for postprocess
    QQ{iElem} = sparse(Qe);  UU{iElem} = sparse(Ue);
    Qf{iElem} = sparse(Qfe); Uf{iElem} = sparse(Ufe);

    %Elemental matrices to be assembled
    KKe = Alq*Qe + Alu*Ue + All;
    ffe = -fl-(Alq*Qfe + Alu*Ufe);

    aux = (1:nOfFaceNodes);
    indRC = [(Fe(1)-1)*nOfFaceNodes + aux,(Fe(2)-1)*nOfFaceNodes + aux,(Fe(3)
    -1)*nOfFaceNodes + aux];
    f(indRC) = f(indRC) + ffe;
    for irow = 1:(3*nOfFaceNodes)
        for icol = 1:(3*nOfFaceNodes)
            ind_i(indK)  = indRC(irow); ind_j(indK)  = indRC(icol);
            coef_K(indK) = KKe(irow,icol); indK = indK+1;
        end
    end
end
KK = sparse(ind_i,ind_j,coef_K);


%%
%% ELEMENTAL MATRIX
function [Q,U,Qf,Uf,Alq,Alu,All,fl] = KKeElementalMatricesIsoParametric(mu,Xe,
    referenceElement,tau,isFeNeumann,isFeRobin)
nOfElementNodes = size(referenceElement.NodesCoord,1);
nOfFaceNodes = size(referenceElement.NodesCoord1d,1);
faceNodes = referenceElement.faceNodes;
nOfFaces = 3; %triangles
gamma = 2.5;
% Information of the reference element
N = referenceElement.N;
Nxi = referenceElement.Nxi; Neta = referenceElement.Neta;
N1d = referenceElement.N1d; Nx1d = referenceElement.N1dxi;
%Numerical quadrature
IPw_f = referenceElement.IPweights1d; ngf = length(IPw_f);
IPw = referenceElement.IPweights; ngauss = length(IPw);

%%Volume computations
% Jacobian
J11 = Nxi*Xe(:,1); J12 = Nxi*Xe(:,2);
J21 = Neta*Xe(:,1); J22 = Neta*Xe(:,2);
detJ = J11.*J22-J12.*J21;
%maybe we should use bsxfun instead of diagonal matrices...
dvolu = spdiags(referenceElement.IPweights.*detJ,0,ngauss,ngauss);
invJ11 = spdiags(J22./detJ,0,ngauss,ngauss);
invJ12 = spdiags(-J12./detJ,0,ngauss,ngauss);
```

```
invJ21 = spdiags(-J21./detJ,0,ngauss,ngauss);
invJ22 = spdiags(J11./detJ,0,ngauss,ngauss);
% xy-derivatives
Nx = invJ11*Nxi + invJ12*Neta;
Ny = invJ21*Nxi + invJ22*Neta;

%Computation of r.h.s. source term (analytical laplacian)
Xg = N*Xe;
sourceTerm = sourcePoisson(Xg,mu);
fe = N'*(dvolu*sourceTerm);
%Elemental matrices
Me = N'*(dvolu*N);
Aqq = zeros(2*size(Me));
aux = 1:2:2*nOfElementNodes; aux2 = 2:2:2*nOfElementNodes;
Aqq(aux,aux)=Me; Aqq(aux2,aux2)=Me;
Auq = zeros(nOfElementNodes,2*nOfElementNodes);
Auq(:,aux) = N'*(dvolu*Nx); %x derivatives & 1st component of q
Auq(:,aux2)= N'*(dvolu*Ny); %y derivatives & 2nd component of q

%% Faces computations
Alq = zeros(3*nOfFaceNodes,2*nOfElementNodes);
Auu = zeros(nOfElementNodes,nOfElementNodes);
Alu = zeros(3*nOfFaceNodes,nOfElementNodes);
All = zeros(3*nOfFaceNodes,3*nOfFaceNodes);
fl = zeros(3*nOfFaceNodes,1);
%Is it possible to remove this loop?
for iface = 1:nOfFaces
    tau_f = tau(iface);
    nodes = faceNodes(iface,:); Xf = Xe(nodes,:); % Nodes in the face
    dxdxi = Nx1d*Xf(:,1); dydxi = Nx1d*Xf(:,2);
    dxdxiNorm = sqrt(dxdxi.^2+dydxi.^2);
    dline = dxdxiNorm.*IPw_f';
    nx = dydxi./dxdxiNorm; ny=-dxdxi./dxdxiNorm;
    %Face matrices
    ind_face = (iface-1)*nOfFaceNodes + (1:nOfFaceNodes);
    Alq(ind_face,2*nodes-1) = N1d'*(spdiags(dline.*nx,0,ngf,ngf)*N1d);
    Alq(ind_face,2*nodes) = N1d'*(spdiags(dline.*ny,0,ngf,ngf)*N1d);
    Auu_f = N1d'*(spdiags(dline,0,ngf,ngf)*N1d)*tau_f;
    Auu(nodes,nodes) = Auu(nodes,nodes) + Auu_f;
    Alu(ind_face,nodes) = Alu(ind_face,nodes) + Auu_f;
    All(ind_face,ind_face) = -Auu_f;
    if isFeNeumann(iface)==1
        Xfg = N1d*Xf;
        traction = tractionPoisson(Xfg,mu);
        for ig = 1:size(N1d,1)
            fl(ind_face) = fl(ind_face) + N1d(ig,:)'*dline(ig)*traction(ig);
        end
    elseif isFeRobin(iface)==1
        All(ind_face,ind_face) = All(ind_face,ind_face)-N1d'*(spdiags(dline,0,
    ngf,ngf)*N1d)*gamma;
```

```
            Xfg = N1d*Xf;
            g= RobinPoisson(Xfg,mu);
            for ig = 1:size(N1d,1)
                fl(ind_face) = fl(ind_face) + N1d(ig,:)'*dline(ig)*g(ig);
            end
        else

        end

end

% Elemental mapping
Aqu = mu*Auq'; Aul = Alu'; Aql = mu*Alq';

A = [Auu Auq; Aqu -Aqq];
UQ = A\[Aul;Aql];

fUQ= A\[fe;zeros(2*nOfElementNodes,1)];

U = UQ(1:nOfElementNodes,:);
Uf=fUQ(1:nOfElementNodes); % maps lamba into U

Q = UQ(nOfElementNodes+1:end,:);
Qf=fUQ(nOfElementNodes+1:end); % maps lamba into Q
```

Postprocess

```
function u_star = HDGpostprocess(X,T,u,q,referenceElementStar,mu)
% q is assumed to be grad(u) in two columns

nOfElements = size(T,1);
nOfElementNodes = size(T,2);
coordRef_star = referenceElementStar.NodesCoord;
npoints = size(coordRef_star,1);

% Compute shape functions at interpolation points
nDeg = referenceElementStar.degree-1;
shapeFunctions=evaluateNodalBasisTriwithoutDerivatives(referenceElementStar.
    NodesCoord,referenceElementStar.NodesCoordGeo,nDeg);

% u star initialization
u_star = zeros(npoints*nOfElements,1);

% Loop in elements
for iElem = 1:nOfElements

    ind = (iElem-1)*nOfElementNodes+1:iElem*nOfElementNodes;
    ind_star = (iElem-1)*npoints+1:iElem*npoints;
    ue = shapeFunctions*u(ind);
    qe = shapeFunctions*q(ind,:);
    [Ke,Beqe,int_ue_star, int_ue] = ElementalMatrices(X(T(iElem,:),:),
    referenceElementStar,ue,qe,mu);

    % Lagrange multipliers
    K = [Ke int_ue_star; int_ue_star' 0];
    f = [Beqe;int_ue];

    % elemental solution
    sol = K\f;

    % postprocessed solution
    u_star(ind_star) = sol(1:end-1);
end

%%
%% ELEMENTAL MATRIX

function [K,Bq,int_u_star,int_u] = ElementalMatrices(Xe,referenceElement_star,
    ue,qe,mu)

nOfElementNodes_star = size(referenceElement_star.NodesCoord,1);
K = zeros(nOfElementNodes_star,nOfElementNodes_star);
Bq = zeros(nOfElementNodes_star,1);
int_u_star = zeros(nOfElementNodes_star,1);
int_u = 0;

% Information of the reference element
IPw = referenceElement_star.IPweights;
N = referenceElement_star.N;
Nxi = referenceElement_star.Nxi;
NxiGeo=referenceElement_star.dNGeodxi;
Neta = referenceElement_star.Neta;
NetaGeo = referenceElement_star.dNGeodeta;
NN_xy = zeros(1,2*nOfElementNodes_star);

% Number of Gauss points in the interior
ngauss = length(IPw);

% x and y coordinates of the element nodes
xe = Xe(:,1); ye = Xe(:,2);

% VOLUME COMPUTATIONS: LOOP IN GAUSS POINTS
for g = 1:ngauss
    %Shape functions and derivatives at the current integration point
    N_g = N(g,:);
    Nxi_g = Nxi(g,:);
    Neta_g = Neta(g,:);

    %Jacobian
    J = [NxiGeo(g,:)*xe    NxiGeo(g,:)*ye
```

```
        NetaGeo(g,:)*xe  NetaGeo(g,:)*ye];

    %Integration weight
    dvolu=IPw(g)*det(J);

    %x and y derivatives
    invJ = inv(J);
    Nx_g = invJ(1,1)*Nxi_g + invJ(1,2)*Neta_g;
    Ny_g = invJ(2,1)*Nxi_g + invJ(2,2)*Neta_g;
    NN_xy(1:2:end) = Nx_g;
    NN_xy(2:2:end) = Ny_g;

    % u and q at gauss points
    u_g = N_g*ue;
    qx_g = N_g*qe(:,1);
    qy_g = N_g*qe(:,2);

    %Contribution of the current integration point to the elemental matrix
    Bq = Bq + (Nx_g'*qx_g + Ny_g'*qy_g)*dvolu;
    K = K + mu*(Nx_g'*Nx_g + Ny_g'*Ny_g)*dvolu;
    int_u_star = int_u_star + N_g'*dvolu;
    int_u = int_u + u_g*dvolu;
end
```

References

[1] R. Sevilla and A. Huerta, "Tutorial on hybridizable discontinuous galerkin (hdg) for second-order elliptic problems," in *Advanced Finite Element Technologies*, pp. 105–129, Springer, 2016.