

# Coupling of discontinuous Galerkin and conforming nodal element for the *curl-curl* operator in $\mathbb{R}^2$

Chak Shing Lee

December 21, 2010

Supervised by Prof. Dr. Ralf Hiptmair

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Discretization</b>	<b>4</b>
<b>3</b>	<b>Auxiliary mixed form</b>	<b>5</b>
<b>4</b>	<b>Stability and Continuity</b>	<b>6</b>
<b>5</b>	<b>Error estimates of solution</b>	<b>10</b>
<b>6</b>	<b>Implementation</b>	<b>12</b>
<b>7</b>	<b>Numerical result</b>	<b>14</b>
<b>8</b>	<b>Concluding remarks</b>	<b>16</b>
<b>A</b>	<b>Other formulations</b>	<b>18</b>
<b>B</b>	<b>Matlab Codes</b>	<b>20</b>

# 1 Introduction

In this project the following model problem is investigated:

$$\begin{aligned} \mathbf{curl} \mathbf{curl} \mathbf{u} &= \mathbf{f} && \text{in } \Omega \\ \mathit{div} \mathbf{u} &= 0 && \text{in } \Omega \\ \mathbf{n} \times \mathbf{u} &= g && \text{on } \partial\Omega \end{aligned} \quad (1)$$

Here,  $\Omega$  is a bounded and simply-connected Lipschitz polygonal domain in  $\mathbb{R}^2$ ,  $\mathbf{n}$  is the unit outward normal to its boundary  $\partial\Omega$ . For a vector field  $\mathbf{u} = (u_1, u_2)$ ,  $\mathbf{curl} \mathbf{u} = \frac{\partial u_2}{\partial x} - \frac{\partial u_1}{\partial y}$ , while for a scalar field  $\phi$ ,  $\mathbf{curl} \phi = (\frac{\partial \phi}{\partial y}, -\frac{\partial \phi}{\partial x})$ . Since it is a problem in  $\mathbb{R}^2$ ,  $\mathbf{n} \times \mathbf{u} = \mathbf{u} \cdot \mathbf{t}$ , where  $\mathbf{t}$  is the counterclockwise oriented tangential unit vector to  $\partial\Omega$ . By taking divergence on both sides of the first equation in (1), one can see that the source function  $\mathbf{f}$  satisfies  $\mathit{div} \mathbf{f} = 0$  in  $\Omega$ . Besides, the Dirichlet boundary datum  $g$  is assumed to belong to  $L^2(\partial\Omega)$ . In fact, the **curl-curl** operator with a divergence free constraint is often encountered in electromagnetic applications, such as the time-harmonic Maxwell's problem in insulating materials (in this case  $g = 0$ ) or the magnetostatic problem in solving the vector potential with Coulomb's gauge. One way to treat the divergence free constraint is the so-called **grad-div** regularization [1, 4], in which case the term **grad div u** is added to the first equation of (1) so as to turn it into an elliptic problem. Following this idea, in the weak sense, problem (1) satisfies the following variational formulation: find  $\mathbf{u} \in \mathbf{V}$  such that  $\mathbf{u} \cdot \mathbf{t} = g$  on  $\partial\Omega$  and

$$\int_{\Omega} \mathbf{curl} \mathbf{u} \mathbf{curl} \mathbf{v} \, dx + \int_{\Omega} \mathit{div} \mathbf{u} \mathit{div} \mathbf{v} \, dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx \quad (2)$$

for all  $\mathbf{v} \in \mathbf{H}_0(\mathbf{curl}; \Omega) \cap \mathbf{H}(\mathit{div}; \Omega)$ . The following function spaces will be relevant:

$$\begin{aligned} \mathbf{H}(\mathit{div}; \Omega) &= \{\mathbf{v} \in L^2(\Omega)^2 : \mathit{div} \mathbf{v} \in L^2(\Omega)\}, \\ \mathbf{H}(\mathit{div}^0; \Omega) &= \{\mathbf{v} \in L^2(\Omega)^2 : \mathit{div} \mathbf{v} = 0 \text{ in } \Omega\}, \\ \mathbf{H}(\mathbf{curl}; \Omega) &= \{\mathbf{v} \in L^2(\Omega)^2 : \mathbf{curl} \mathbf{v} \in L^2(\Omega)\}, \\ \mathbf{H}_0(\mathbf{curl}; \Omega) &= \{\mathbf{v} \in \mathbf{H}(\mathbf{curl}; \Omega) : \mathbf{v} \cdot \mathbf{t} = 0\}, \\ \mathbf{V} &= \{\mathbf{v} \in \mathbf{H}(\mathbf{curl}; \Omega) \cap \mathbf{H}(\mathit{div}; \Omega) : \mathbf{v} \cdot \mathbf{t} \in L^2(\Omega)\}. \end{aligned}$$

If the domain is non-convex, the equation has singularities at the re-entrant corners. It is well-known that standard finite element methods will produce spurious solutions for the singularities due to the fact that the standard  $\mathbf{H}^1$  conforming nodal element space is not dense in  $\mathbf{V}$ , see [4]. For example, consider the exact solution given by the function  $\nabla [r^{2/3} \sin(\frac{2\theta}{3})]$  on a L-shaped domain with re-entrant corner at the origin, Figure 1 shows the graph of this function. In order to illustrate the inability of standard finite

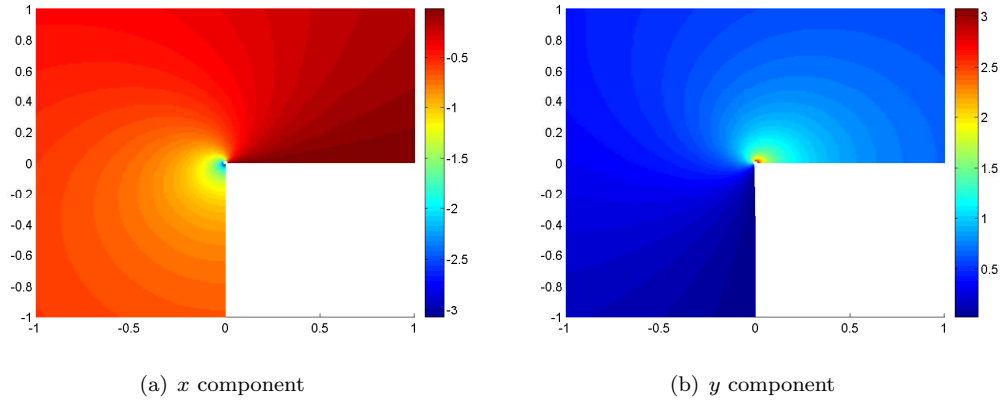


Figure 1: Graph of  $\nabla [r^{2/3} \sin(\frac{2\theta}{3})]$

element method based on the weak formulation (2), we provide also the numerical solution obtained by using conforming piecewise linear nodal elements in Figure 2. This approximation, which is completely different from the exact solution, is the so-called spurious solution. A remedy for such problem is to

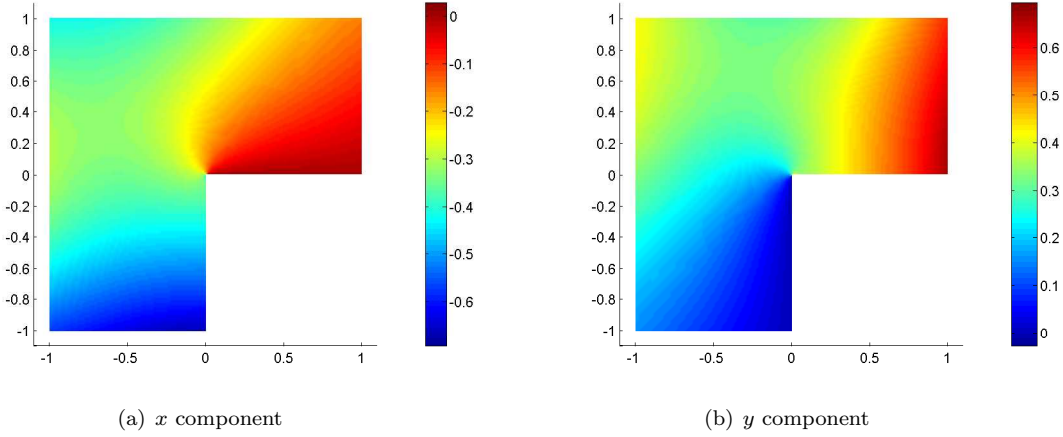


Figure 2: Numerical solution of standard finite element method: by Matlab script SReg\_LFE2.m

turn it into a mixed form in which the divergence free constraint is treated by introducing the Lagrange multiplier  $p$  and then use Nédélec's  $\mathbf{H}(\text{curl})$  conforming edge elements to approximate the vector field [4]. More precisely, note that the solution for (2) together with  $p = 0$  solves the following mixed problem: find  $\mathbf{u} \in \mathbf{H}(\text{curl}; \Omega)$  and  $p \in H_0^1(\Omega)$  such that  $\mathbf{u} \cdot \mathbf{t} = g$  on  $\partial\Omega$  and

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) &= l(\mathbf{v}) \\ b(\mathbf{u}, q) &= 0 \end{aligned} \quad (3)$$

for all  $(\mathbf{v}, q) \in \mathbf{H}_0(\text{curl}; \Omega) \times H_0^1(\Omega)$ , where the bilinear forms  $a$  and  $b$  and the linear form  $l$  are given by

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &= \int_{\Omega} \text{curl} \mathbf{u} \cdot \text{curl} \mathbf{v} \, d\mathbf{x} \\ b(\mathbf{v}, p) &= \int_{\Omega} \mathbf{v} \cdot \nabla p \, d\mathbf{x} \\ l(\mathbf{v}) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, d\mathbf{x}. \end{aligned}$$

Then the vector field  $\mathbf{u}$  and scalar function  $p$  are discretized by means of Nédélec's  $\mathbf{H}(\text{curl})$  conforming edge elements and standard  $\mathbf{H}^1$  nodal elements respectively. The disadvantage is that the order of convergence of edge elements is one order lesser than standard nodal elements for approximation of regular functions and it is hard to construct a basis for higher order edge elements. Hence, a scheme that produces good approximation for the singularities using nodal elements is desirable. The purpose of this project is to investigate whether coupling of discontinuous Galerkin and conforming nodal element based on (3) can serve as an alternative remedy for spurious solution while having an optimal convergence rate for both regular and singular functions.

**Remark** Solving (3) amounts to solve the following system:

$$\begin{aligned} \text{curl} \, \text{curl} \, \mathbf{u} + \nabla p &= \mathbf{f} && \text{in } \Omega \\ \text{div} \, \mathbf{u} &= 0 && \text{in } \Omega \\ \mathbf{n} \times \mathbf{u} &= g && \text{on } \partial\Omega \\ p &= 0 && \text{on } \partial\Omega \end{aligned} \quad (4)$$

It has been shown in [7] that problem (4) admits a unique solution  $(\mathbf{u}, p) \in \mathbf{H}_0(\text{curl}; \Omega) \cap \mathbf{H}(\text{div}^0; \Omega) \times H_0^1(\Omega)$ . Further, there exists  $\sigma = \sigma(\Omega) > \frac{1}{2}$  such that  $\mathbf{u} \in H^\sigma(\Omega)^2$ ,  $\text{curl } \mathbf{u} \in H^\sigma(\Omega)$  and

$$\|\mathbf{u}\|_{\sigma, \Omega} + \|p\|_{1, \Omega} \leq C_{reg} \|\mathbf{f}\|_{0, \Omega}.$$

for some constant  $C_{reg}$ . If  $\Omega$  is convex,  $\sigma$  can be chosen to be 1.

## 2 Discretization

In this section we introduce the discretization scheme that coupling the discontinuous Galerkin and conforming nodal element for the mixed problem (3). In this setting, the vector function  $\mathbf{u}$  will be approximated by functions in the space:

$$\mathbf{V}_h = \{\mathbf{v} \in L^2(\Omega)^2 : \mathbf{v}|_K \in P_1(K)^2 \forall K \in \mathcal{M}\}$$

while the scalar function  $p$  will be approximated by functions in the space:

$$Q_h = \{v \in C^0(\Omega) : v|_K \in P_1(K) \forall K \in \mathcal{M}, v|_{\partial\Omega} = 0\}.$$

Here  $P_1(K)$  is the space of polynomials of degree 1. The shape regular triangular meshes  $\mathcal{M}$  is used to partition  $\Omega$ , the diameter of each element  $K \in \mathcal{M}$  is denoted by  $h_K$ , and  $h = \max_K h_K$ . By shape regular we mean there exists a positive constant  $\kappa$  such that, for all  $K \in \mathcal{M}$ ,

$$\frac{h_K}{\rho_K} \leq \kappa$$

where  $\rho_K$  is the diameter of the largest ball contained in  $K$ , see Figure 3. Moreover, we assume the mesh

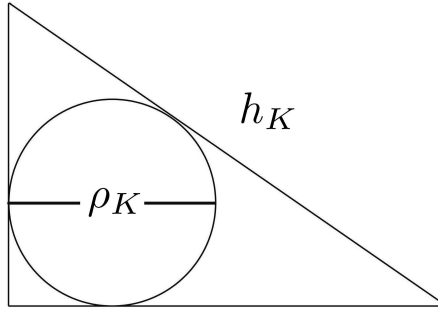


Figure 3: Shape regular element

satisfies bounded variation, that is, there exist constants  $C_1$  and  $C_2$  such that

$$C_1 h_K \leq h_{K'} \leq C_2 h_K$$

for any adjacent element  $K$  and  $K'$ . The set of interior edges is denoted by  $\varepsilon_I$  while that of boundary edges is denoted by  $\varepsilon_B$ , and  $\varepsilon = \varepsilon_I \cup \varepsilon_B$ . For each interior edge  $e$  shared by elements  $K^+$  and  $K^-$ , the jumps across  $e$  are defined as follows:  $[\mathbf{v}]_T = \mathbf{v}^+ \cdot \mathbf{t}^+ + \mathbf{v}^- \cdot \mathbf{t}^-$ ,  $[\mathbf{v}]_N = \mathbf{v}^+ \cdot \mathbf{n}^+ + \mathbf{v}^- \cdot \mathbf{n}^-$ , and the average is defined by  $\{\psi\} = (\psi^+ + \psi^-)/2$ , where  $\mathbf{v}^+$  and  $\mathbf{v}^-$  are the traces of  $\mathbf{v}$  in  $K^+$  and  $K^-$  respectively.  $\psi^+$  and  $\psi^-$  are defined similarly. Here  $\mathbf{n}^+$  and  $\mathbf{t}^+$  are the unit outward normal and the counterclockwise oriented unit tangent of  $K^+$  on  $e$ ,  $\mathbf{n}^-$  and  $\mathbf{t}^-$  are defined similarly, see Figure 4. For boundary edges,  $[\mathbf{v}]_T = \mathbf{v} \cdot \mathbf{t}$ ,  $\{\psi\} = \psi$ . Discretization will be based on the mixed formulation (3). Applying the discontinuous Galerkin method with interior penalty (IP-DGFEM) to the bilinear form  $a$  while keeping  $p$  conforming in  $H_0^1(\Omega)$  will result in the following discretized saddle point problem: find  $\mathbf{u}_h \in \mathbf{V}_h$  and  $p_h \in Q_h$  such that

$$\begin{aligned} a_h(\mathbf{u}_h, \mathbf{v}_h) + b_h(\mathbf{v}_h, p_h) &= l_h(\mathbf{v}_h) \\ b_h(\mathbf{u}_h, q_h) - d_h(p_h, q_h) &= 0 \end{aligned} \tag{5}$$

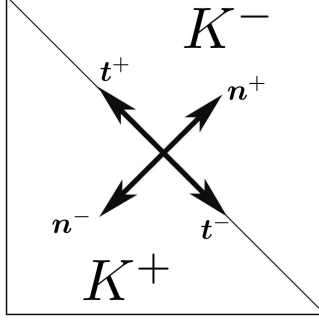


Figure 4: Normal and tangent vectors on interior edge

for all  $(\mathbf{v}_h, q_h) \in \mathbf{V}_h \times Q_h$ , where

$$\begin{aligned}
a_h(\mathbf{u}_h, \mathbf{v}_h) &= \int_{\Omega} \text{curl}_h \mathbf{u}_h \text{curl}_h \mathbf{v}_h \, d\mathbf{x} - \int_{\varepsilon} [\mathbf{u}_h]_T \{ \text{curl}_h \mathbf{v}_h \} ds - \int_{\varepsilon} [\mathbf{v}_h]_T \{ \text{curl}_h \mathbf{u}_h \} ds \\
&\quad + \int_{\varepsilon} \frac{\alpha}{|e|} [\mathbf{u}_h]_T [\mathbf{v}_h]_T \, ds + \beta h^2 \int_{\Omega} \text{div}_h \mathbf{u}_h \text{div}_h \mathbf{v}_h \, d\mathbf{x} + \int_{\varepsilon_I} \beta |e| [\mathbf{u}_h]_N [\mathbf{v}_h]_N \, ds \\
b_h(\mathbf{v}_h, p_h) &= \int_{\Omega} \mathbf{v}_h \cdot \nabla p_h \, d\mathbf{x} \\
d_h(p_h, q_h) &= \int_{\Omega} p_h q_h \, d\mathbf{x} \\
l(\mathbf{v}) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, d\mathbf{x} - \int_{\varepsilon_B} g \text{curl}_h \mathbf{v} \, ds + \int_{\varepsilon_B} \frac{\alpha}{|e|} g [\mathbf{v}]_T \, ds.
\end{aligned}$$

In the formulation  $\text{curl}_h$  and  $\text{div}_h$  are the elementwise  $\text{curl}$  and  $\text{div}$  operators,  $|e|$  denotes the length of the edge  $e$  where the integration takes place. The bilinear form  $d_h$  is added so that the auxiliary variable  $p$  can be canceled in the computation. One can use mass lumping for  $d_h$  so that the inversion becomes much easier and hence the computation cost is reduced. Note that  $d_h$  is equal to zero for  $p = 0$ , so (5) is consistent for the exact solution of (3). The inclusion of the  $h^2$  weighted  $\text{div-div}$  term and the penalty term of the normal jumps across interior edges is due to stability consideration, this will become clear in section 4. Also, it will be shown that the stabilization parameter  $\alpha$  in the penalty term has to be large enough to ensure stability.

### 3 Auxiliary mixed form

In order to get error estimates for the solution, an auxiliary mixed form will be introduced in this section. The resulting bilinear forms possess continuity and coercivity properties that favour the error analysis. First, the underlying function space is denoted by

$$\begin{aligned}
\mathbf{V}(h) &= \mathbf{V} + \mathbf{V}_h, \\
Q(h) &= H_0^1(\Omega) + Q_h = H_0^1(\Omega)
\end{aligned}$$

endowed with the DG-seminorm and DG-norms:

$$\begin{aligned}
|\mathbf{v}|_{\mathbf{V}(h)}^2 &= \|\text{curl} \, \mathbf{v}\|_{0,\Omega}^2 + h^2 \|\text{div} \, \mathbf{v}\|_{0,\Omega}^2 + \|h^{-\frac{1}{2}} [\mathbf{v}]_T\|_{0,\varepsilon}^2 + \|h^{\frac{1}{2}} [\mathbf{v}]_N\|_{0,\varepsilon_I}^2 \\
\|\mathbf{v}\|_{\mathbf{V}(h)}^2 &= \|\mathbf{v}\|_{0,\Omega}^2 + |\mathbf{v}|_{\mathbf{V}(h)}^2 \\
\|q\|_{Q(h)} &= \|q\|_{1,\Omega}
\end{aligned}$$

The broken Sobolev space is denoted by

$$H^s(\mathcal{M}) = \{v \in L^2(\Omega) : v|_K \in H^s(K) \, \forall K \in \mathcal{M}\}$$

endowed with the broken norm  $\|v\|_{s,\mathcal{M}}^2 = \sum_{K \in \mathcal{M}} \|v\|_{s,K}^2$ . Also, on the boundary, denote

$$H^s(\varepsilon_B) = \{v \in L^2(\partial\Omega) : v|_e \in H^s(e) \forall e \in \varepsilon_B\}$$

endowed with the broken norm  $\|v\|_{s,\varepsilon_B}^2 = \sum_{e \in \varepsilon_B} \|v\|_{s,e}^2$ .

Next, we defined the lifting operator  $\mathcal{L} : L^2(\varepsilon) \rightarrow V_h$  by

$$\int_{\Omega} \mathcal{L}(\mathbf{u}) \cdot v \, d\mathbf{x} = \int_{\varepsilon} [\mathbf{u}]_T \cdot \{v\} \, ds \quad \forall v \in V_h$$

where  $V_h = \{v \in L^2(\Omega) : v|_K \in P_1(K) \forall K \in \mathcal{M}\}$ .

Then, the auxiliary mixed formulation can be stated as: find  $(\mathbf{u}_h, \lambda) \in \mathbf{V}_h \times Q_h$  and  $p_h \in Q_h$  such that

$$\begin{aligned} A_h(\mathbf{u}_h, \lambda; \mathbf{v}_h, \eta) + B_h(\mathbf{v}_h, \eta; p_h) &= l_h(\mathbf{v}_h) \\ B_h(\mathbf{u}_h, \lambda; q_h) &= 0 \end{aligned} \quad (6)$$

for all  $(\mathbf{v}_h, \eta) \in \mathbf{V}_h \times Q_h$  and  $q_h \in Q_h$ . In which

$$\begin{aligned} A_h(\mathbf{u}_h, \lambda; \mathbf{v}_h, \eta) &= \int_{\Omega} \text{curl}_h \mathbf{u}_h \text{curl}_h \mathbf{v}_h \, d\mathbf{x} - \int_{\Omega} \mathcal{L}(\mathbf{u}_h) \cdot \text{curl} \mathbf{v}_h \, d\mathbf{x} - \int_{\Omega} \mathcal{L}(\mathbf{v}_h) \cdot \text{curl} \mathbf{u}_h \, d\mathbf{x} \\ &\quad + \int_{\varepsilon} \frac{\alpha}{|e|} [\mathbf{u}_h]_T [\mathbf{v}_h]_T \, ds + \beta h^2 \int_{\Omega} \text{div}_h \mathbf{u}_h \text{div}_h \mathbf{v}_h \, d\mathbf{x} + \int_{\varepsilon_T} \beta |e| [\mathbf{u}_h]_N [\mathbf{v}_h]_N \, ds + \int_{\Omega} \lambda \eta \, d\mathbf{x} \\ B_h(\mathbf{v}_h, \eta; p_h) &= \int_{\Omega} \mathbf{v}_h \cdot \nabla p_h \, d\mathbf{x} - \int_{\Omega} p_h \eta \, d\mathbf{x} \end{aligned}$$

Lastly, defined the product space  $\mathbf{W}(h) = \mathbf{V}(h) \times Q_h$ , endowed with the seminorm and norm:

$$\begin{aligned} |(\mathbf{v}, \eta)|_{\mathbf{W}(h)}^2 &= |\mathbf{v}|_{\mathbf{V}(h)}^2 + \|\eta\|_{0,\Omega}^2 \\ \|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)}^2 &= \|\mathbf{v}\|_{\mathbf{V}(h)}^2 + \|\eta\|_{0,\Omega}^2 \end{aligned}$$

In the following sections, the auxiliary mixed formulation will be analyzed using the above norm so that an error bound for the solution can be obtained.

## 4 Stability and Continuity

The continuity of the bilinear forms  $A_h$  and  $B_h$ , ellipticity of  $A_h$  on the kernel of  $B_h$ , and the inf-sup condition of  $B_h$  will be established in this section, which is essential for the stability of the auxiliary mixed formulation (6) and facilitates the error analysis in the next section. We begin by recalling some useful inverse inequalities [2] and the stability property of the lifting operator [7].

**Lemma 4.1** *For polynomials  $p \in P_{\ell}(K)$ , we have*

$$\begin{aligned} \|p\|_{0,\partial K} &\leq C_{inv} h_K^{-\frac{1}{2}} \|p\|_{0,K} \\ |p|_{1,K} &\leq C_{inv} h_K^{-1} \|p\|_{0,K} \end{aligned}$$

where  $C_{inv}$  is a constant depending on the shape regularity and the polynomial degree  $\ell$ .

**Lemma 4.2** *Let  $\mathcal{L}$  be the lifting operator defined above, then for all  $\mathbf{v}_h \in \mathbf{V}_h$*

$$\|\mathcal{L}([\mathbf{v}]_T)\|_{0,\Omega} \leq C_{lift} \|h^{-\frac{1}{2}} [\mathbf{v}]_T\|_{0,\varepsilon}$$

where  $C_{lift}$  is independent of the mesh size.

**Proof** By definition and the Cauchy Schwarz inequality,

$$\begin{aligned}
\|\mathcal{L}([\mathbf{v}]_T)\|_{0,\Omega} &= \sup_{w \in V_h} \frac{\int_{\Omega} \mathcal{L}([\mathbf{v}]_T) \cdot w \, dx}{\|w\|_{0,\Omega}} \\
&= \sup_{w \in V_h} \frac{\int_{\varepsilon} [\mathbf{v}]_T \cdot \{w\} \, ds}{\|w\|_{0,\Omega}} \\
&\leq \sup_{w \in V_h} \frac{\|h^{-\frac{1}{2}}[\mathbf{v}]_T\|_{0,\varepsilon} \|h^{\frac{1}{2}}\{w\}\|_{0,\varepsilon}}{\|w\|_{0,\Omega}}
\end{aligned}$$

Further, from the first inequality in Lemma 4.1, we get

$$\begin{aligned}
\|h^{\frac{1}{2}}\{w\}\|_{0,\varepsilon}^2 &\leq C \sum_{K \in \mathcal{M}} h_K \|w\|_{0,\partial K}^2 \\
&\leq C \sum_{K \in \mathcal{M}} \|w\|_{0,K}^2 \\
&= C \|w\|_{0,\Omega}^2.
\end{aligned}$$

Combining both inequalities, we get the result.  $\square$

The continuity of  $A_h$  and  $B_h$  follows easily from Cauchy Schwarz inequality, the definition of  $\|\cdot\|_{\mathbf{W}(h)}$  and Lemma 4.2.

**Proposition 4.3** *There exist constants  $C_{cont} > 0$  independent of mesh size such that*

$$\begin{aligned}
|A_h(\mathbf{u}_h, \lambda; \mathbf{v}_h, \eta)| &\leq C_{cont} \|(\mathbf{u}_h, \lambda)\|_{\mathbf{W}(h)} \|(\mathbf{v}_h, \eta)\|_{\mathbf{W}(h)} \\
|B_h(\mathbf{v}_h, \eta; q)| &\leq C_{cont} \|(\mathbf{v}_h, \eta)\|_{\mathbf{W}(h)} \|q\|_{1,\Omega}
\end{aligned}$$

for all  $(\mathbf{u}_h, \lambda), (\mathbf{v}_h, \eta) \in \mathbf{W}(h)$ ,  $q \in H_0^1$ . Moreover,

$$|l(\mathbf{v})| \leq C_{cont} \left( \|\mathbf{f}\|_{0,\Omega} + \|h^{-\frac{1}{2}}g\|_{0,\partial\Omega} \right) \|\mathbf{v}\|_{\mathbf{W}(h)}$$

Next, consider the kernel of  $B_h$

$$Ker(B_h) = \{(\mathbf{u}_h, \lambda) \in \mathbf{W}_h : B_h(\mathbf{u}_h, \lambda; q) = 0 \, \forall q \in Q_h\}$$

The following proposition gives the coercivity of  $A_h$  on the kernel of  $B_h$ .

**Proposition 4.4** *There exists  $\alpha_{min} > 0$  which is independent of the mesh size such that, for all stabilization parameter  $\alpha > \alpha_{min}$ ,  $\beta > 0$ , there exists a constant  $C_{coer}$ , which is also independent of the mesh size, such that for all  $(\mathbf{u}_h, \lambda) \in Ker(B_h)$ ,*

$$A_h(\mathbf{u}_h, \lambda; \mathbf{u}_h, \lambda) \geq C_{coer} \|(\mathbf{u}_h, \lambda)\|_{\mathbf{W}(h)}^2$$

**Proof** First, we have for  $\alpha$  large enough,

$$A_h(\mathbf{u}_h, \lambda; \mathbf{u}_h, \lambda) \geq C |(\mathbf{u}_h, \lambda)|_{\mathbf{W}(h)}^2 \quad (7)$$

for all  $(\mathbf{u}_h, \lambda) \in \mathbf{W}_h$ , where  $C$  is independent of mesh size. This can be seen by making use of Lemma 4.2, Cauchy Schwarz inequality and Young's inequality. Note that

$$\begin{aligned}
A_h(\mathbf{u}_h, \lambda; \mathbf{u}_h, \lambda) &= \|curl \, \mathbf{u}_h\|_{0,\Omega}^2 - 2 \int_{\Omega} \mathcal{L}(\mathbf{u}_h) \cdot curl \, \mathbf{u}_h \, dx + \alpha \|h^{-\frac{1}{2}}[\mathbf{u}_h]_T\|_{0,\varepsilon}^2 \\
&\quad + \beta \|h \, div \, \mathbf{u}_h\|_{0,\Omega}^2 + \beta \|h^{\frac{1}{2}}[\mathbf{u}_h]_N\|_{0,\varepsilon_I}^2 + \|\lambda\|_{0,\Omega}^2 \\
&\geq \|curl \, \mathbf{u}_h\|_{0,\Omega}^2 - \delta \|\mathcal{L}(\mathbf{u}_h)\|_{0,\Omega}^2 - \delta^{-1} \|curl \, \mathbf{u}_h\|_{0,\Omega}^2 + \alpha \|h^{-\frac{1}{2}}[\mathbf{u}_h]_T\|_{0,\varepsilon}^2 \\
&\quad + \beta \|h \, div \, \mathbf{u}_h\|_{0,\Omega}^2 + \beta \|h^{\frac{1}{2}}[\mathbf{u}_h]_N\|_{0,\varepsilon_I}^2 + \|\lambda\|_{0,\Omega}^2 \\
&\geq (1 - \delta^{-1}) \|curl \, \mathbf{u}_h\|_{0,\Omega}^2 + (\alpha - \delta C_{ift}^2) \|h^{-\frac{1}{2}}[\mathbf{u}_h]_T\|_{0,\varepsilon}^2 \\
&\quad + \beta \|h \, div \, \mathbf{u}_h\|_{0,\Omega}^2 + \beta \|h^{\frac{1}{2}}[\mathbf{u}_h]_N\|_{0,\varepsilon_I}^2 + \|\lambda\|_{0,\Omega}^2
\end{aligned}$$

where  $C_{lift}$  is the constant in Lemma 4.2, and  $\delta$  is a positive constant. Hence, if  $\alpha > \alpha_{min} = C_{lift}^2$ , one can pick  $\delta$  with  $1 < \delta < \alpha/C_{lift}^2$  and (7) follows.

Now we claim that for all  $(\mathbf{u}_h, \lambda) \in Ker(B_h)$ ,

$$|(\mathbf{u}_h, \lambda)|_{\mathbf{W}(h)} \geq C \|\mathbf{u}_h\|_{0, \Omega}.$$

Once this is proved, the result will follow immediately. To prove this, fix an arbitrary  $(\mathbf{u}_h, \lambda) \in Ker(B_h)$  and consider the solution  $(\mathbf{z}, \psi)$  of the dual problem

$$\begin{aligned} \mathbf{curl} \mathbf{curl} \mathbf{z} + \nabla \psi &= \mathbf{u}_h & \text{in } \Omega \\ \mathbf{div} \mathbf{z} &= 0 & \text{in } \Omega \\ \mathbf{n} \times \mathbf{z} &= \mathbf{g} & \text{on } \partial\Omega \\ \psi &= 0 & \text{on } \partial\Omega \end{aligned}$$

By the regularity of the solution, we have

$$\|\mathbf{curl} \mathbf{z}\|_{0, \Omega} + \|\mathbf{z}\|_{0, \Omega} + \|\nabla \psi\|_{0, \Omega} + \|\psi\|_{0, \Omega} \leq C \|\mathbf{u}_h\|_{0, \Omega}.$$

Set  $w = \mathbf{curl} \mathbf{z}$ , then  $\mathbf{curl} w \in L^2(\Omega)^2$ . From [3], there is  $w_0 \in H^1(\Omega)$  such that  $\mathbf{curl} w = \mathbf{curl} w_0$ ,

$$\|w_0\|_{1, \Omega} \leq C (\|w\|_{0, \Omega} + \|\mathbf{curl} w\|_{0, \Omega}) \leq \|\mathbf{u}_h\|_{0, \Omega} \quad (8)$$

Then we multiply the first equation of the dual problem by  $\mathbf{u}_h$ , perform integration by parts and obtain

$$\begin{aligned} \|\mathbf{u}_h\|_{0, \Omega}^2 &= \int_{\Omega} w_0 \cdot \mathbf{curl} \mathbf{u}_h \, d\mathbf{x} - \int_{\varepsilon} w_0 [\mathbf{u}_h]_T ds \\ &\quad - \int_{\Omega} \psi \cdot \mathbf{div} \mathbf{u}_h \, d\mathbf{x} + \int_{\varepsilon_I} \psi [\mathbf{u}_h]_N ds \end{aligned}$$

By making use the fact that  $(\mathbf{u}_h, \lambda) \in Ker(B_h)$ , we have  $B_h(\mathbf{u}_h, \lambda; \psi_h) = 0$  for all  $\psi_h \in Q_h$ . Therefore,

$$\begin{aligned} \|\mathbf{u}_h\|_{0, \Omega}^2 &= \int_{\Omega} w_0 \cdot \mathbf{curl} \mathbf{u}_h \, d\mathbf{x} - \int_{\varepsilon} w_0 [\mathbf{u}_h]_T ds \\ &\quad - \int_{\Omega} (\psi - \psi_h) \cdot \mathbf{div} \mathbf{u}_h \, d\mathbf{x} + \int_{\varepsilon_I} (\psi - \psi_h) [\mathbf{u}_h]_N ds - \int_{\Omega} \psi_h \lambda \, d\mathbf{x} \end{aligned}$$

Now we bound each terms on the right hand side. By (8), one can obtain

$$\begin{aligned} \left| \int_{\Omega} w_0 \cdot \mathbf{curl} \mathbf{u}_h \, d\mathbf{x} \right| &\leq C \|w_0\|_{1, \Omega} \|\mathbf{curl} \mathbf{u}_h\|_{0, \Omega} \\ &\leq C \|\mathbf{u}_h\|_{0, \Omega} |\mathbf{u}_h|_{\mathbf{V}(h)} \end{aligned}$$

Also, by (8) and trace inequalities, we get

$$\begin{aligned} \left| \int_{\varepsilon} w_0 [\mathbf{u}_h]_T ds \right| &\leq C \left( \sum_{K \in \mathcal{M}} h_K \|w_0\|_{0, \partial K}^2 \right)^{\frac{1}{2}} \|h^{-\frac{1}{2}} [\mathbf{u}_h]_T\|_{0, \varepsilon} \\ &\leq C \|w_0\|_{1, \Omega} \|h^{-\frac{1}{2}} [\mathbf{u}_h]_T\|_{0, \varepsilon} \\ &\leq C \|\mathbf{u}_h\|_{0, \Omega} |\mathbf{u}_h|_{\mathbf{V}(h)} \end{aligned}$$

Then, we will take  $\psi_h$  as the  $L^2$  projection of  $\psi$  on  $Q_h$  in the rest of the proof. Using Cauchy Schwarz inequality, the approximation properties of  $L^2$  projection, and (8) we get

$$\begin{aligned} \left| \int_{\Omega} (\psi - \psi_h) \cdot \mathbf{div} \mathbf{u}_h \, d\mathbf{x} \right| &\leq C h^{-1} \|\psi - \psi_h\|_{0, \Omega} h \|\mathbf{div} \mathbf{u}_h\|_{0, \Omega} \\ &\leq C \|\psi\|_{1, \Omega} h \|\mathbf{div} \mathbf{u}_h\|_{0, \Omega} \\ &\leq C \|\mathbf{u}_h\|_{0, \Omega} |\mathbf{u}_h|_{\mathbf{V}(h)} \end{aligned}$$

and

$$\begin{aligned}
\left| \int_{\varepsilon_I} (\psi - \psi_h) [\mathbf{u}_h]_N ds \right| &\leq C \left( \sum_{K \in \mathcal{M}} h_K^{-1} \|\psi - \psi_h\|_{0, \partial K}^2 \right)^{\frac{1}{2}} \|h^{\frac{1}{2}} [\mathbf{u}_h]_N\|_{0, \varepsilon_I} \\
&\leq C \left( \sum_{K \in \mathcal{M}} \|\psi\|_{1, K}^2 \right)^{\frac{1}{2}} |\mathbf{u}_h|_{\mathbf{V}(h)} \\
&\leq C \left( \sum_{K \in \mathcal{M}} \|\psi\|_{1, K}^2 \right)^{\frac{1}{2}} |\mathbf{u}_h|_{\mathbf{V}(h)} \\
&\leq C \|\psi\|_{1, \Omega} |\mathbf{u}_h|_{\mathbf{V}(h)} \\
&\leq C \|\mathbf{u}_h\|_{0, \Omega} |\mathbf{u}_h|_{\mathbf{V}(h)}
\end{aligned}$$

Lastly,

$$\begin{aligned}
\left| \int_{\Omega} \psi_h \lambda \, d\mathbf{x} \right| &\leq \|\psi_h\|_{0, \Omega} \|\lambda\|_{0, \Omega} \\
&\leq C \|\psi\|_{1, \Omega} \|\lambda\|_{0, \Omega} \\
&\leq C \|\mathbf{u}_h\|_{0, \Omega} \|\lambda\|_{0, \Omega}
\end{aligned}$$

So the claim is proved.  $\square$

The last proposition in this section is the inf-sup condition of  $B_h$ .

**Proposition 4.5** *There exists a constant  $C_{is} > 0$ , independent of the mesh size, such that*

$$\inf_{0 \neq q \in Q_h} \sup_{0 \neq (\mathbf{v}_h, \eta) \in \mathbf{W}_h} \frac{B_h(\mathbf{v}_h, \eta; q)}{\|q\|_{1, \Omega} \|(\mathbf{v}_h, \eta)\|_{\mathbf{W}(h)}} \geq C_{is}$$

**Proof** Fix  $0 \neq q \in Q_h$ , take  $\mathbf{v}_h = \nabla q \in \mathbf{V}_h \cap \mathbf{H}_0(\text{curl}; \Omega)$ , then  $\text{curl } \mathbf{v}_h = 0$  and

$$B_h(\mathbf{v}_h, 0; q) = \|\nabla q\|_{0, \Omega}^2 \geq \|q\|_{1, \Omega}^2$$

where we have used Poincaré Friedrich's inequality for  $q \in H_0^1(\Omega)$ .

On the other hand,

$$\begin{aligned}
\|(\mathbf{v}_h, 0)\|_{\mathbf{W}(h)}^2 &= \|\mathbf{v}\|_{0, \Omega}^2 + \|h^{\frac{1}{2}} [\mathbf{v}]_N\|_{0, \varepsilon_I}^2 + h^2 \|\text{div } \mathbf{v}\|_{0, \Omega}^2 \\
&\leq \|\nabla q\|_{0, \Omega}^2 + C \sum_{K \in \mathcal{M}} h \|\nabla q\|_{0, \partial \Omega}^2 + \|h \Delta q\|_{0, \Omega}^2 \\
&\leq \|\nabla q\|_{0, \Omega}^2 + C \sum_{K \in \mathcal{M}} \|\nabla q\|_{0, \Omega}^2 + C_{inv} \|\nabla q\|_{0, \Omega}^2 \\
&\leq C \|q\|_{1, \Omega}^2
\end{aligned}$$

where we have used Lemma 4.1 in the second last inequality. So

$$B_h(\mathbf{v}_h, 0; q) \geq C \|q\|_{1, \Omega} \|(\mathbf{v}_h, 0)\|_{\mathbf{W}(h)}$$

This completes the proof.  $\square$

## 5 Error estimates of solution

In order to give the error estimates of the solution, we define the residuals for the solution  $(\mathbf{u}, p)$  of (3) by

$$\begin{aligned} R_h^1(\mathbf{u}, p; \mathbf{v}, \eta) &= A_h(\mathbf{u}, 0; \mathbf{u}, \eta) + B_h(\mathbf{v}, \eta; p) - l_h(\mathbf{v}), \\ R_h^2(\mathbf{u}; q) &= B_h(\mathbf{u}, 0; q) \end{aligned}$$

for all  $(\mathbf{v}, \eta) \in \mathbf{W}_h$  and  $q \in Q_h$ , and

$$\begin{aligned} \mathcal{R}_h^1(\mathbf{u}, p) &= \sup_{0 \neq (\mathbf{v}, \eta) \in \mathbf{W}_h} \frac{|R_h^1(\mathbf{u}, p; \mathbf{v}, \eta)|}{\|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)}}, \\ \mathcal{R}_h^2(\mathbf{u}) &= \sup_{0 \neq q \in Q_h} \frac{|R_h^2(\mathbf{u}; q)|}{\|q\|_{1, \Omega}} \end{aligned}$$

Now we can get the estimates of error in terms of the discretization errors and the residual terms.

**Theorem 5.1** *Let  $\mathbf{u}, p$  be the solutions of (3) and  $(\mathbf{u}_h, \lambda_h)$ ,  $p$  be the solution of (5). Then there exists a constant  $C > 0$ , which depends on the constants  $C_{cont}$  and  $C_{is}$  but independent of mesh size, such that*

$$\|(\mathbf{u} - \mathbf{u}_h, \lambda_h)\|_{\mathbf{W}(h)} \leq C \left( \inf_{\mathbf{v} \in \mathbf{V}_h} \|\mathbf{u} - \mathbf{v}\|_{\mathbf{V}(h)} + \inf_{q \in Q_h} \|p - q\|_{1, \Omega} + \mathcal{R}_h^1(\mathbf{v}, p) + \mathcal{R}_h^2(\mathbf{u}) \right), \quad (9)$$

$$\|p - p_h\|_{1, \Omega} \leq C \left( \inf_{q \in Q_h} \|p - q\|_{1, \Omega} + \|(\mathbf{u} - \mathbf{u}_h, \lambda_h)\|_{\mathbf{W}(h)} + \mathcal{R}_h^1(\mathbf{v}, p) \right) \quad (10)$$

**Proof** We first prove (10). Let  $q \in Q_h$ . By definition and direct calculation, we have

$$A_h(\mathbf{u} - \mathbf{u}_h, \lambda_h; \mathbf{v}, \eta) + B_h(\mathbf{v}, \eta; p - q) + B_h(\mathbf{v}, \eta; q - p_h) = R_h^1(\mathbf{u}, p; \mathbf{v}, \eta),$$

for all  $(\mathbf{v}, \eta) \in \mathbf{W}_h$ . Then, from Lemma 4.5, we get

$$\begin{aligned} \|q - p_h\|_{1, \Omega} &\leq \frac{1}{C_{is}} \sup_{0 \neq (\mathbf{v}, \eta) \in \mathbf{W}_h} \frac{B_h(\mathbf{v}, \eta; q - p_h)}{\|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)}} \\ &= \frac{1}{C_{is}} \sup_{0 \neq (\mathbf{v}, \eta) \in \mathbf{W}_h} \frac{-A_h(\mathbf{u} - \mathbf{u}_h, \lambda_h; \mathbf{v}, \eta) - B_h(\mathbf{v}, \eta; p - q) + R_h^1(\mathbf{u}, p; \mathbf{v}, \eta)}{\|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)}} \\ &\leq \frac{C_{cont}}{C_{is}} \|(\mathbf{u} - \mathbf{u}_h, \lambda_h)\|_{\mathbf{W}(h)} + \frac{C_{cont}}{C_{is}} \|p - q\|_{1, \Omega} + \frac{1}{C_{is}} \mathcal{R}_h^1(\mathbf{u}, p) \end{aligned}$$

Since the triangle inequality  $\|p - p_h\|_{1, \Omega} \leq \|p - q\|_{1, \Omega} + \|q - p_h\|_{1, \Omega}$  holds true for all  $q \in Q_h$ , we have

$$\|p - p_h\|_{1, \Omega} \leq \frac{C_{cont}}{C_{is}} \|(\mathbf{u} - \mathbf{u}_h, \lambda_h)\|_{\mathbf{W}(h)} + \left(1 + \frac{C_{cont}}{C_{is}}\right) \|p - q\|_{1, \Omega} + \frac{1}{C_{is}} \mathcal{R}_h^1(\mathbf{u}, p)$$

for all  $q \in Q_h$ . This gives (10) after taking infimum on  $q$  on the right hand side.

Now to prove (9), we consider  $(\mathbf{v}, \eta) \in Ker(B_h)$ . Note that  $(\mathbf{v} - \mathbf{u}_h, \eta - \lambda_h) \in Ker(B_h)$ , so we have

$$\begin{aligned} C_{coer} \|(\mathbf{v} - \mathbf{u}_h, \eta - \lambda_h)\|_{\mathbf{W}(h)} &\leq A_h(\mathbf{v} - \mathbf{u}_h, \eta - \lambda_h; \mathbf{v} - \mathbf{u}_h, \eta - \lambda_h) \\ &= A_h(\mathbf{v} - \mathbf{u}, \eta; \mathbf{v} - \mathbf{u}_h, \eta - \lambda_h) - B_h(\mathbf{v} - \mathbf{u}_h, \eta - \lambda_h; p - q) \\ &\quad + R_h^1(\mathbf{u}, p; \mathbf{v} - \mathbf{u}_h, \eta - \lambda_h) \end{aligned}$$

for all  $q \in Q_h$ . Here we have used Proposition 4.4. Similarly, with the help of triangle inequality

$$\|(\mathbf{u} - \mathbf{u}_h, \lambda_h)\|_{\mathbf{W}(h)} \leq \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} + \|(\mathbf{v} - \mathbf{u}_h, \eta - \lambda_h)\|_{\mathbf{W}(h)}$$

and Proposition 4.3, we get

$$\begin{aligned} \|(\mathbf{v} - \mathbf{u}_h, \eta - \lambda_h)\|_{\mathbf{W}(h)} &\leq \left(1 + \frac{C_{cont}}{C_{coer}}\right) \inf_{(\mathbf{v}, \eta) \in Ker(B_h)} \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} \\ &\quad + \frac{C_{cont}}{C_{coer}} \inf_{q \in Q_h} \|p - q\|_{1, \Omega} + \frac{1}{C_{coer}} \mathcal{R}_h^1(\mathbf{u}, p). \end{aligned}$$

We claim that

$$\inf_{(\mathbf{v}, \eta) \in Ker(B_h)} \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} \leq \left(1 + \frac{C_{cont}}{C_{is}}\right) \inf_{(\mathbf{v}, \eta) \in \mathbf{W}_h} \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} + \frac{1}{C_{is}} \mathcal{R}_h^2(\mathbf{u}).$$

Let  $(\mathbf{v}, \eta) \in \mathbf{W}_h$  be arbitrary. By definition, we have

$$B_h(-\mathbf{v}, -\eta; q) = B_h(\mathbf{u} - \mathbf{v}, -\eta; q) - R_h^2(\mathbf{u}, q) \quad \forall q \in Q_h$$

Then, by Proposition 4.3 and 4.4, we get

$$\|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)} \leq \frac{C_{cont}}{C_{is}} \inf_{(\mathbf{v}, \eta) \in \mathbf{W}_h} \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} + \frac{1}{C_{is}} \mathcal{R}_h^2(\mathbf{u}). \quad (11)$$

Obviously  $(0, 0) \in Ker(B_h)$ , and

$$\|(\mathbf{u} - 0, 0)\|_{\mathbf{W}(h)} \leq \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} + \|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)},$$

for all  $(\mathbf{v}, \eta) \in \mathbf{W}_h$ . Hence, together with (11), the claim is proved. So we have

$$\|(\mathbf{u} - \mathbf{u}_h, \lambda_h)\|_{\mathbf{W}(h)} \leq C \left( \inf_{(\mathbf{v}, \eta) \in \mathbf{W}_h} \|(\mathbf{u} - \mathbf{v}, \eta)\|_{\mathbf{W}(h)} + \inf_{q \in Q_h} \|p - q\|_{1, \Omega} + \mathcal{R}_h^1(\mathbf{v}, p) + \mathcal{R}_h^2(\mathbf{u}) \right) \quad (12)$$

Taking  $\eta = 0$ , then (9) follows.  $\square$

Before giving the explicit error estimates of the solution, we first estimate the residuals.

**Proposition 5.2** *Assume the exact solution  $(\mathbf{u}, p)$  of (3) satisfies  $\mathbf{u} \in H^s(\mathcal{M})^2$  and  $\text{curl } \mathbf{u} \in H^s(\mathcal{M})^2$ ,  $s > \frac{1}{2}$ . Then*

$$\mathcal{R}_h^1(\mathbf{u}, p) + \mathcal{R}_h^2(\mathbf{u}) \leq Ch^{\min\{s, 2\}} (\|\mathbf{u}\|_{s, \mathcal{M}} + \|\text{curl } \mathbf{u}\|_{s, \mathcal{M}})$$

where  $C$  is a constant independent of mesh size.

**Proof** Since the exact solution  $(\mathbf{u}, p)$  satisfies  $\text{div } \mathbf{u} = 0$ ,  $p = 0$  in  $\Omega$ . Moreover, all the jump terms vanishes since the exact solution is continuous. Hence, using integration by parts, we have for any  $q \in Q_h$

$$|R_h^2(\mathbf{u}; q)| = \left| - \int_{\Omega} \text{div } \mathbf{u} \cdot q d\mathbf{x} + \int_{\partial\Omega} \mathbf{u} \cdot \mathbf{n} q ds \right|.$$

Since  $q|_{\partial\Omega} = 0$  by the definition of  $Q_h$ , we have  $|R_h^2(\mathbf{u}; q)| = 0$ . Also, by definition of  $L^2$  projection, integration by parts and using the first equation of (1), we have

$$|R_h^1(\mathbf{u}; q)| = \left| \int_{\varepsilon} \{\text{curl } \mathbf{u} - \prod_{\mathbf{v}_h}(\text{curl } \mathbf{u})\} \cdot [\mathbf{v}]_T ds \right|,$$

where  $\prod_{\mathbf{v}_h}$  is the  $L^2$  projection onto  $\mathbf{V}_h$ . Then by making use of the Cauchy Schwarz inequality and the standard approximation properties, we get

$$\begin{aligned} |R_h^1(\mathbf{u}; q)| &\leq C \left( \sum_{K \in \mathcal{M}} h_K \|\text{curl } \mathbf{u} - \prod_{\mathbf{v}_h}(\text{curl } \mathbf{u})\|_{0, \partial K}^2 \right)^{\frac{1}{2}} \|h^{-\frac{1}{2}} [\mathbf{u}_h]_T\|_{0, \varepsilon} \\ &\leq Ch^{\min\{s, 2\}} \|\text{curl } \mathbf{u}\|_{s, \mathcal{M}} \|(\mathbf{v}, \eta)\|_{\mathbf{W}(h)} \end{aligned}$$

This finishes the proof.  $\square$

The next Theorem provides an explicit error estimates for the solution. The estimates are optimal and require a minimal regularity assumption on the exact solution, which is satisfied in general polygonal domain.

**Theorem 5.3** *Let  $(\mathbf{u}, p)$  be the exact solution of (3) satisfying  $\mathbf{u} \in H^s(\mathcal{M})^2$ ,  $\text{curl } \mathbf{u} \in H^s(\mathcal{M})^2$ ,  $p \in H^{s+1}(\mathcal{M})$ , and  $g \in H^{s+\frac{1}{2}}(\varepsilon_B)$  with  $s > \frac{1}{2}$ . Let  $(\mathbf{u}_h, p_h)$  be the solution of the discrete problem (5). Then*

$$\|\mathbf{u} - \mathbf{u}_h\|_{\mathbf{V}(h)} + \|p - p_h\|_{1,\Omega} \leq Ch^{\min\{s,1\}} \left( \|\mathbf{u}\|_{s,\mathcal{M}} + \|\text{curl } \mathbf{u}\|_{s,\mathcal{M}} + \|p\|_{s+1,\mathcal{M}} + \|g\|_{s+\frac{1}{2},\varepsilon_B} \right),$$

where  $C$  is a constant independent of mesh size but the shape regularity of mesh and the stabilization parameters  $\alpha$  and  $\beta$ .

**Remark** In general the constant  $C$  depends also on the degree  $\ell$  of polynomials used in the approximation space, but in this project we are restricted to the case of piecewise linear polynomials, so this dependence is omitted.

**Proof** Let  $(\mathbf{u}_h, \lambda_h; p_h)$  be the solution of the auxiliary problem (6). Let  $\mathbf{v} = \prod_{\text{curl}} \mathbf{u} \in \mathbf{H}(\text{curl}; \Omega)$  be the conforming Nédélec interpolant of  $\mathbf{u}$  of the second type [6]. Then,  $[\mathbf{v}]_T = 0$  on  $\varepsilon_I$  and from [1], we have

$$\|\mathbf{u} - \prod_{\text{curl}} \mathbf{u}\|_{0,K} + \|\text{curl}(\mathbf{u} - \prod_{\text{curl}} \mathbf{u})\|_{0,K} \leq Ch_K^{\min\{s,1\}} (\|\mathbf{u}\|_{s,K} + \|\text{curl}(\mathbf{u})\|_{s,K})$$

with  $C$  independent of mesh size. Next, from standard scaling arguments, we have for all  $e \in \varepsilon_B$ ,

$$\|(\mathbf{u} - \prod_{\text{curl}} \mathbf{u}) \cdot \mathbf{t}\|_{0,e} \leq Ch_K^{\min\{s,1\} + \frac{1}{2}} \|g\|_{s+\frac{1}{2},e}$$

Hence,

$$\|h^{-\frac{1}{2}}(\mathbf{u} - \prod_{\text{curl}} \mathbf{u}) \cdot \mathbf{t}\|_{0,\varepsilon_B} \leq Ch^{\min\{s,1\}} \|g\|_{s+\frac{1}{2},\varepsilon_B}$$

Further, note that  $\text{div } \mathbf{u} = 0$ , from [7], we have

$$\|h^{\frac{1}{2}}[\mathbf{u} - \prod_{\text{curl}} \mathbf{u}]_N\|_{0,\varepsilon_I} \leq Ch^{\min\{s,1\}} (\|\mathbf{u}\|_{s,\mathcal{M}} + \|\text{curl}(\mathbf{u})\|_{s,\mathcal{M}})$$

and

$$\left( \sum_{K \in \mathcal{M}} h_K^2 \|\text{div}(\mathbf{u} - \prod_{\text{curl}} \mathbf{u})\|_{0,K} \right)^{\frac{1}{2}} \leq Ch^{\min\{s,1\}} (\|\mathbf{u}\|_{s,\mathcal{M}} + \|\text{curl}(\mathbf{u})\|_{s,\mathcal{M}}).$$

On the other hand, let  $q = \prod_{H^1} p$  be the standard  $H^1$  interpolant of  $p$ , then we have the following standard estimates

$$\|p - \prod_{H^1} p\|_{1,\Omega} \leq Ch^{\min\{s,1\}} \|p\|_{s+1,\mathcal{M}}$$

Together with Theorem 5.1, Proposition 5.2 and all the estimates above, the result follows.  $\square$

## 6 Implementation

The matlab codes used for numerical tests are based on LehrFEM, which is a exercise and demonstration software for education in numerics of partial differential equations developed in SAM, Department of Math, ETH, Zürich. Additional routines that are needed for this project will be included in appendix. In order to make it more clear, we will explain the structure of the element matrices in this section. Let  $K$  denote a single triangle with vertices  $a_i = (x_i, y_i)$ ,  $i = 1, 2, 3$ . Let  $\lambda_i$  denote the barycentric coordinate

functions, i.e., the linear functions with  $\lambda_i(a_j) = \delta_{ij}$ . If the vertices are arranged in counterclockwise fashion, they have the representation

$$\begin{aligned}\lambda_1(x) &= \frac{1}{2|K|} \left( x - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \right) \cdot \begin{pmatrix} y_2 - y_3 \\ x_3 - x_2 \end{pmatrix}, \\ \lambda_2(x) &= \frac{1}{2|K|} \left( x - \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} \right) \cdot \begin{pmatrix} y_3 - y_1 \\ x_1 - x_3 \end{pmatrix}, \\ \lambda_3(x) &= \frac{1}{2|K|} \left( x - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right) \cdot \begin{pmatrix} y_1 - y_2 \\ x_2 - x_1 \end{pmatrix}.\end{aligned}$$

Their gradients are constant and read

$$\nabla \lambda_1 = \frac{1}{2|K|} \begin{pmatrix} y_2 - y_3 \\ x_3 - x_2 \end{pmatrix}, \quad \nabla \lambda_2 = \frac{1}{2|K|} \begin{pmatrix} y_3 - y_1 \\ x_1 - x_3 \end{pmatrix}, \quad \nabla \lambda_3 = \frac{1}{2|K|} \begin{pmatrix} y_1 - y_2 \\ x_2 - x_1 \end{pmatrix}.$$

Given any bilinear form  $b_K$  and local basis functions  $\phi_1, \dots, \phi_m$  we aim to compute the element matrix  $\mathcal{S}_K \in \mathbb{R}^{m,m}$  with

$$\mathcal{S}_{ij} = b_K(\phi_j, \phi_i).$$

In the space  $Q_h$  we only need a lumped mass matrix, obtained from the local bilinear form

$$b_K(\phi, \psi) = \frac{1}{3}|K| \sum_{j=1}^3 \phi(a_j) \psi(a_j).$$

The lumped local mass matrix is diagonal and reads  $D_K = \frac{1}{3}|K| I_3$ .

For the space  $\mathbf{V}_h$ , the basis functions are (in the order we shall take) the 2-vector-valued functions given by

$$\phi_1 = (\lambda_1, 0), \quad \phi_2 = (0, \lambda_1), \quad \phi_3 = (\lambda_2, 0), \quad \phi_4 = (0, \lambda_2), \quad \phi_5 = (\lambda_3, 0), \quad \phi_6 = (0, \lambda_3).$$

## 6.1 Local curl-curl matrix

The local bilinear form is

$$b_K(\phi_i, \phi_j) = \int_K \text{curl}_h \phi_i \text{curl}_h \phi_j \, d\mathbf{x};,$$

where  $\text{curl}_h \phi = \frac{\partial \phi^2}{\partial x} - \frac{\partial \phi^1}{\partial y}$ . We get  $\hat{C}_K = |K| R^T R$ , where  $R$  is the  $1 \times 6$  matrix whose entries are  $\text{curl}_h \phi_i$ , i.e.,

$$\begin{aligned}R &= \left( -\frac{\partial \lambda_1}{\partial y}, \frac{\partial \lambda_1}{\partial x}, -\frac{\partial \lambda_2}{\partial y}, \frac{\partial \lambda_2}{\partial x}, -\frac{\partial \lambda_3}{\partial y}, \frac{\partial \lambda_3}{\partial x} \right) \\ &= \frac{1}{2|K|} (-x_3 + x_2, -y_3 + y_2, -x_1 + x_3, -y_1 + y_3, -x_2 + x_1, -y_2 + y_1).\end{aligned}$$

## 6.2 Local div-div matrix

The local bilinear form is

$$b_K(\phi_i, \phi_j) = \int_K \text{div}_h \phi_i \text{div}_h \phi_j \, d\mathbf{x};,$$

We form the row vector

$$\begin{aligned}R &= \left( \frac{\partial \lambda_1}{\partial x}, \frac{\partial \lambda_1}{\partial y}, \frac{\partial \lambda_2}{\partial x}, \frac{\partial \lambda_2}{\partial y}, \frac{\partial \lambda_3}{\partial x}, \frac{\partial \lambda_3}{\partial y} \right) \\ &= \frac{1}{2|K|} (x_3 - x_2, y_2 - y_3, x_1 - x_3, y_3 - y_1, x_2 - x_1, y_1 - y_2).\end{aligned}$$

Then we have  $\hat{D}_K = |K| L^T L$ .

### 6.3 Local $\nabla$ matrix

The local bilinear form is

$$b_K(\phi_i, \lambda_j) = \int_K \phi_i \nabla_h \lambda_j \, d\mathbf{x}; ,$$

Note that this is a  $6 \times 3$  matrix because  $j$  only run through 1 to 3. Also, since the integrand is not constant, so one cannot express it like the *curl-curl* matrix, but it is convenient use quadrature rules to evaluate the integral. The order of the quadrature rules do not have to be high(even first order accurate is fine) as we are just using linear polynomials.

### 6.4 Contribution from edges

Here we just explain one of the edge integrals. Consider the local bilinear form

$$b_K(\phi_i, \phi_j) = \int_e [\phi_i]_T \{curl_h \phi_j\} ds,$$

Note that for interior edges, the size of the matrix from local contribution is  $12 \times 12$  because the basis functions of both elements sharing the same edge are involved and there are 12 of them. On the other hand, for boundary edges, the size of the matrix from local contribution is  $6 \times 6$  because one boundary edge only belongs to one element. Other edge integrals have similar structure.

### 6.5 Right hand side

We do not show the details for the load vector as it can be obtained similarly by simply replacing the basis functions by the source function or the boundary datum, so results in a vector instead of a matrix. Usually one uses quadrature rule to compute the load vector as the source function or the boundary datum are complicated functions.

### 6.6 Solving the system

After assembling the local contributions, the system (3) takes the form:

$$\begin{aligned} S\vec{u}_h + G\vec{p} &= \vec{f} \\ G^T\vec{u}_h - D\vec{p} &= 0 \end{aligned}$$

where  $D$  is diagonal and therefore can be easily inverted,  $S$  is the stiff matrix consists of all the terms on the left hand side except the gradient and  $G$  is the discrete gradient. Upon canceling the multiplier  $p$ , the above system can be reduced to

$$(S + GD^{-1}G^T)\vec{u}_h = \vec{f}$$

The numerical solution is obtained by solving this system of equations.

## 7 Numerical result

In this section, results of numerical experiment using the method stated in section 2 will be shown for the following functions:

$$\begin{aligned} S_1(x, y) &= \begin{pmatrix} y(y+1) \\ x(x-1) \end{pmatrix} \\ S_2(x, y) &= \nabla \left[ r^{4/3} \sin\left(\frac{4\theta}{3}\right) \right] \\ S_3(x, y) &= \nabla \left[ r^{2/3} \sin\left(\frac{2\theta}{3}\right) \right]. \end{aligned}$$

For the vector field, the exact and computed solutions are denoted by  $\mathbf{u}$  and  $\mathbf{u}_h$  respectively, and the  $L^2$  norm is denoted by  $\|\cdot\|_{0,\Omega}$  and DG energy norm is denoted by  $\|\cdot\|_h^2$ . The square domain is  $(0,1)^2$  while the L-shaped domain is  $(-1,1)^2 \setminus ([0,1] \times [0,-1])$ . The parameter is set to be  $\alpha = 10$  and  $\beta = 1$ . Tables 1-3 give the details of the errors and the corresponding orders of convergence of  $\mathbf{u}$ . For the

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u}-\mathbf{u}_h\ _{0,\Omega}$	Order	$\ \mathbf{u}_h\ _h$	$\ \mathbf{u}-\mathbf{u}_h\ _h$	Order
0.3536	1.0346e+00	8.0101e-03	-	4.1223e+00	2.1944e-01	-
0.1768	1.0334e+00	2.1427e-03	1.90238	5.7459e+00	1.0725e-01	1.03289
0.0884	1.0330e+00	5.5442e-04	1.95039	8.0646e+00	5.2969e-02	1.01774
0.0442	1.0328e+00	1.4110e-04	1.97425	1.1360e+01	2.6316e-02	1.00918
0.0221	1.0328e+00	3.5600e-05	1.98676	1.6033e+01	1.3116e-02	1.00467

Table 1:  $S_1$  in square domain: results by Matlab script MIXDG.m

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u}-\mathbf{u}_h\ _{0,\Omega}$	Order	$\ \mathbf{u}_h\ _h$	$\ \mathbf{u}-\mathbf{u}_h\ _h$	Order
0.5000	2.0877e+00	4.2699e-02	-	4.8618e+00	1.9273e-01	-
0.2500	2.0917e+00	1.8021e-02	1.24451	6.5472e+00	8.2656e-02	1.22141
0.1250	2.0925e+00	7.2757e-03	1.30855	9.0193e+00	3.3565e-02	1.30017
0.0625	2.0926e+00	2.9009e-03	1.32656	1.2582e+01	1.3409e-02	1.32373
0.0312	2.0927e+00	1.1527e-03	1.33148	1.7671e+01	5.3320e-03	1.33048

Table 2:  $S_2$  in L-shaped domain: results by Matlab script MIXDG.m

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u}-\mathbf{u}_h\ _{0,\Omega}$	Order	$\ \mathbf{u}_h\ _h$	$\ \mathbf{u}-\mathbf{u}_h\ _h$	Order
0.5000	1.2751e+00	1.5689e-01	-	2.4092e+00	4.0585e-01	-
0.2500	1.3209e+00	9.5824e-02	0.71132	3.1442e+00	2.6888e-01	0.59399
0.1250	1.3411e+00	5.9113e-02	0.69692	4.2404e+00	1.7225e-01	0.64247
0.0625	1.3495e+00	3.6852e-02	0.68174	5.8433e+00	1.0911e-01	0.65867
0.0312	1.3528e+00	2.3108e-02	0.67335	8.1523e+00	6.8866e-02	0.66393

Table 3:  $S_3$  in L-shaped domain: results by Matlab script MIXDG.m

scalar function, the exact and computed solutions are denoted by  $p$  and  $p_h$  respectively, and the  $H^1$  norm is denoted by  $\|\cdot\|_{1,\Omega}$ . Tables 4-6 give the details of the errors and the corresponding orders of convergence of  $p$ . Since we are more concerned about the performance of the scheme for corner singular

$h$	$\ p-p_h\ _{0,\Omega}$	Order	$\ p-p_h\ _{1,\Omega}$	Order
0.3536	1.0183e-04	-	8.1208e-04	-
0.1768	2.1060e-05	2.27362	3.5805e-04	1.18147
0.0884	2.4584e-06	3.09870	7.5864e-05	2.23867
0.0442	2.4114e-07	3.34978	1.4373e-05	2.40007
0.0221	2.2290e-08	3.43540	2.6191e-06	2.45621

Table 4:  $S_1$  in square domain: results by Matlab script MIXDG.m

$h$	$\ p-p_h\ _{0,\Omega}$	Order	$\ p-p_h\ _{1,\Omega}$	Order
0.5000	9.6120e-03	-	6.7680e-02	-
0.2500	2.6180e-03	1.87635	2.9673e-02	1.18956
0.1250	5.9874e-04	2.12849	1.2005e-02	1.30554
0.0625	1.2791e-04	2.22685	4.7854e-03	1.32692
0.0312	2.6459e-05	2.27326	1.9011e-03	1.33183

Table 5:  $S_2$  in L-shaped domain: results by Matlab script MIXDG.m

$h$	$\ p-p_h\ _{0,\Omega}$	Order	$\ p-p_h\ _{1,\Omega}$	Order
0.5000	4.8355e-02	-	2.2350e-01	-
0.2500	2.4606e-02	0.97464	1.6163e-01	0.46764
0.1250	1.0806e-02	1.18723	1.0648e-01	0.60205
0.0625	4.4971e-03	1.26473	6.8180e-02	0.64317
0.0312	1.8294e-03	1.29765	4.3214e-02	0.65787

Table 6:  $S_3$  in L-shaped domain: results by Matlab script MIXDG.m

function  $S_3$ , which is not able to be solved by standard finite element method, we show in Figure 5 the

graph of the numerical solution obtained by the proposed scheme. As we can see in the figure, and also revealed from the numbers in Tables 1-6, the numerical solution by the proposed scheme is a very good approximation of the exact solution even for the highly singular function. In order to lucidly show a

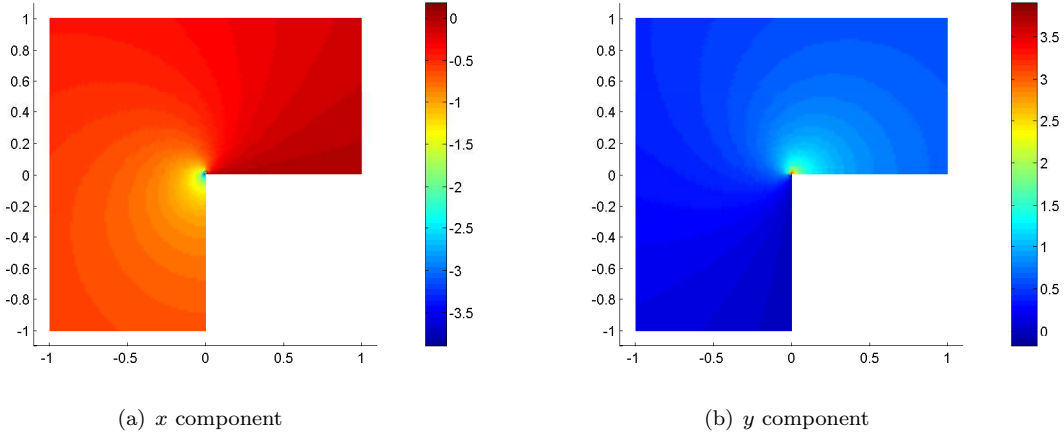


Figure 5: Numerical solution of the proposed scheme: by Matlab script MIXDG.m

comparison of the performance of different methods in each test case, we include a log-log plot of  $L^2$  error versus mesh size in Figure 7. In the legend, MIXDG refers to our scheme, obtained by Matlab script MIXDG.m, WRegW1F(weak regularization) refers to the method using edge elements(Whitney 1-form), obtained by Matlab script WReg\_mix.m, SRegLFE(strong regularization) refers to standard finite element method, obtained by Matlab script SReg\_LFE2.m As mentioned in the introduction, the  $\mathbf{H}(\text{curl})$  conforming edge elements can give a good approximation for the singular solution (see Figure 6(c)), but in general its order of convergence is not optimal, which is shown in Figure 6(a) and 6(b). On the other hand, standard Lagrangian elements give an optimal convergence rate for regular functions, but fail to deal with the corner singular functions. Our proposed scheme possesses the advantages of both methods, therefore is a good option for solving (1). Moreover, because the elements used in the scheme is Lagrangian nodal elements, it can be implemented using conventional finite element tool box easily. Lastly, we remark that the  $h^2$  weighted *div-div* regularization term and the normal jump penalty term cannot be dropped because then formulation (5) become unstable (see Proposition 4.4) and one can not even get convergence for smooth functions.

## 8 Concluding remarks

We have investigated the coupling of discontinuous Galerkin and conforming nodal element for the *curl-curl* operator in  $\mathbb{R}^2$ , this method provides a spurious free solution for the singularities at the re-entrant corners of non-convex domain. Following the same arguments as in [5], with an additional stabilization term in proving the ellipticity of  $A_h$  on  $\text{Ker}(B_h)$ , we have shown an optimal a priori error estimate for the solution in the DG energy norm. The numerical tests have been implemented for both smooth and singular functions on convex and non-convex domains, respectively. The results agree with the expected order of convergence. Note that this method is very similar to the method introduced by I.Perugia et al. [5], but the number of degrees of freedom is smaller in our method. Besides, by using mass lumping technique for the bilinear form  $d_h$ , the problem size can be further reduced. Moreover, this method should also work in  $\mathbb{R}^3$  and should be able to be generalized to high order using higher degree polynomials as the approximation space. If a different polynomial degree is used for the finite element approximation spaces, say,  $P_1$  for  $\mathbf{u}$  and  $P_2$  for  $p$ , the formulation may require less stabilization terms. In short, this scheme provides a good alternative as a remedy for solving the corner singularities of the *curl-curl* operator.

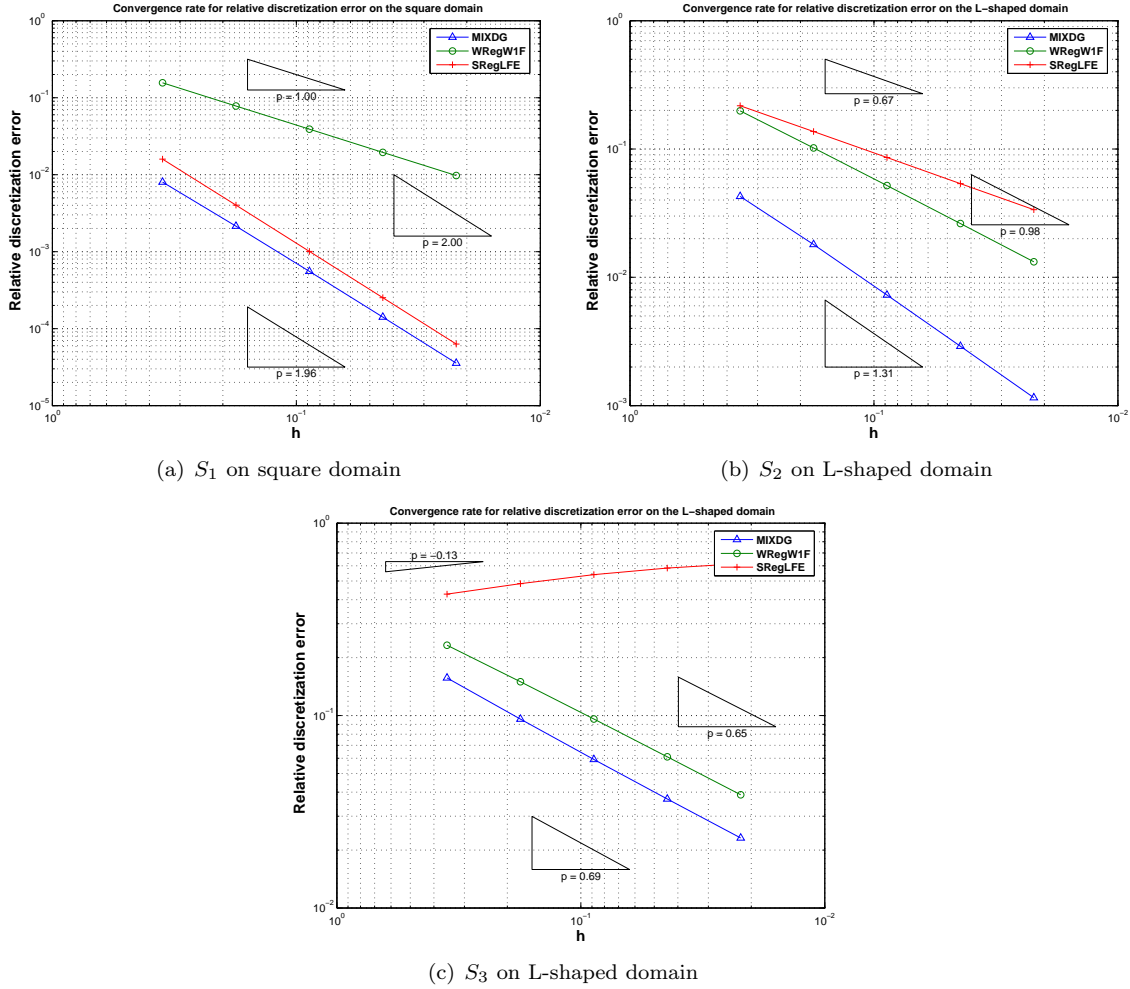


Figure 6: Log-log plot of  $L^2$  error versus mesh size: by Matlab script plot\_rate4.m

## References

- [1] A. Alonso and A. Valli, *An optimal domain decomposition preconditioner for low-frequency time-harmonic Maxwell equations*, Math. Comp., 68:607-631, 1999.
- [2] P. Ciarlet, *The Finite Element Method for Elliptic Problems*, North-Holland, Amsterdam, 1978.
- [3] P. Fernandes and G. Gilardi, *Magnetostatic and electrostatic problems in inhomogeneous anisotropic media with irregular boundary and mixed boundary conditions*, Math. Models Methods Appl. Sci., 7:957V991, 1997.
- [4] R. Hiptmair, *Finite elements in computational electromagnetics*, Acta Numerica, 11:237-339, 2002.
- [5] P. Houston, I. Perugia, and D. Schötzau, *Mixed discontinuous Galerkin approximation of the Maxwell operator*, SIAM J. Numer. Anal., 42:434-459, 2004.
- [6] J.C. Nédélec, *A new family of mixed finite elements in  $R^3$* , Numerische Mathematik, 50:57-81, 1986.
- [7] I. Perugia, D. Schötzau, and P. Monk, *Stabilized interior penalty methods for the time-harmonic Maxwell equations*, Comput. Meth. Appl. Mech. Eng., 191:4675-4697, 2002.

## A Other formulations

In this project, attempts have also been made for other discretization methods, but the results show that they are not good for solving (1). Here in this appendix section, we will state these discretization methods as well as the corresponding numerical results. The first method is to apply the discontinuous Galerkin method with interior penalty(IP-DGFEM) to (2). Then the discretized problem is: find  $\mathbf{u}_h \in \mathbf{V}_h$  such that  $a_1(\mathbf{u}_h, \mathbf{v}) = l_1(\mathbf{v})$  for all  $\mathbf{v} \in \mathbf{V}_h$ , where

$$\begin{aligned}
a_1(\mathbf{u}, \mathbf{v}) &= \int_{\Omega} \mathit{curl}_h \mathbf{u} \mathit{curl}_h \mathbf{v} \, d\mathbf{x} - \int_{\varepsilon} [\mathbf{u}]_T \{ \mathit{curl}_h \mathbf{v} \} ds - \int_{\varepsilon} [\mathbf{v}]_T \{ \mathit{curl}_h \mathbf{u} \} ds \\
&\quad + \mathit{div}_h \mathbf{u} \mathit{div}_h \mathbf{v} \, d\mathbf{x} - \int_{\varepsilon_I} [\mathbf{u}]_N \{ \mathit{div}_h \mathbf{v} \} ds - \int_{\varepsilon_I} [\mathbf{v}]_N \{ \mathit{div}_h \mathbf{u} \} ds \\
&\quad + \int_{\varepsilon} \frac{\alpha}{|e|} [\mathbf{u}]_T [\mathbf{v}]_T \, ds + \int_{\varepsilon_I} \frac{\beta}{|e|} [\mathbf{u}]_N [\mathbf{v}]_N \, ds \\
l_1(\mathbf{v}) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, d\mathbf{x} - \int_{\varepsilon_B} g \mathit{curl}_h \mathbf{v} \, ds + \int_{\varepsilon_B} \frac{\alpha}{|e|} g [\mathbf{v}]_T \, ds,
\end{aligned}$$

$\mathbf{V}_h = \{ \mathbf{v} \in L^2(\Omega)^2 : \mathbf{v}|_K \in P_1(K)^2 \, \forall K \in \mathcal{M} \}$ ,  $\mathit{curl}_h$  and  $\mathit{div}_h$  are the elementwise  $\mathit{curl}$  and  $\mathit{div}$  operators. The constants  $\alpha$  and  $\beta$  in the penalty terms have to be large enough to ensure stability. By taking  $\alpha = \beta = 10$ , we obtained the numerical results in Tables 7-9. The numerical results show that this method works well for regular functions, but it fails to recover the singular functions. Note that for  $S_3$  in the L-shaped domain, the numerical solution converges to the spurious solution. For the second

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u} - \mathbf{u}_h\ _{0,\Omega}$	Order
0.3536	1.0360e+00	1.0316e-02	-
0.1768	1.0339e+00	2.9125e-03	1.82452
0.0884	1.0331e+00	7.7026e-04	1.91886
0.0442	1.0329e+00	1.9789e-04	1.96065
0.0221	1.0328e+00	5.0144e-05	1.98055

Table 7:  $S_1$  in square domain: results by Matlab script DGcurl.m

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u} - \mathbf{u}_h\ _{0,\Omega}$	Order
0.5000	2.0473e+00	6.3444e-02	-
0.2500	2.0624e+00	4.2022e-02	0.59434
0.1250	2.0732e+00	2.7188e-02	0.62818
0.0625	2.0803e+00	1.7342e-02	0.64871
0.0312	2.0849e+00	1.0982e-02	0.65916

Table 8:  $S_2$  in L-shaped domain: results by Matlab script DGcurl.m

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u} - \mathbf{u}_h\ _{0,\Omega}$	Order
0.5000	8.5797e-01	5.9718e-01	-
0.2500	8.3561e-01	6.2971e-01	-0.07654
0.1250	8.2266e-01	6.4887e-01	-0.04323
0.0625	8.1470e-01	6.6068e-01	-0.02603
0.0312	8.0970e-01	6.6811e-01	-0.01613

Table 9:  $S_3$  in L-shaped domain: results by Matlab script DGcurl.m

method, note that for sufficiently smooth function  $\mathbf{u}$ , equations (1) are equivalent to:

$$\begin{aligned}
-\Delta \mathbf{u} &= \mathbf{f} && \text{in } \Omega \\
\mathbf{n} \times \mathbf{u} &= \mathbf{g} && \text{on } \partial\Omega
\end{aligned} \tag{13}$$

Write  $\mathbf{H}_T^1(\Omega) = \{\mathbf{v} \in H^1(\Omega)^2 : \mathbf{v} \cdot \mathbf{t} = 0 \text{ on } \partial\Omega\}$  and  $H^1(\Omega) = \{v \in L^2(\Omega) : \nabla v \in L^2(\Omega)^2\}$ , then the variational formulation for (13) is: find  $\mathbf{u} \in H^1(\Omega)^2$  such that  $\mathbf{u} \cdot \mathbf{t} = g$  on  $\partial\Omega$  and

$$\int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \, dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx \quad (14)$$

for all  $\mathbf{v} \in \mathbf{H}_T^1(\Omega)$ . So the second method is applying IP-DGFEM to (14), which yields the following discretized problem: find  $\mathbf{u}_h \in \mathbf{V}_h$  such that  $a_2(\mathbf{u}_h, \mathbf{v}) = l_2(\mathbf{v})$  for all  $\mathbf{v} \in \mathbf{V}_h$ , where

$$\begin{aligned} a_2(\mathbf{u}, \mathbf{v}) &= \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} \, dx - \int_{\varepsilon_I} [\mathbf{u}]_N \{\nabla \mathbf{v}\} ds - \int_{\varepsilon_I} [\mathbf{v}]_N \{\nabla \mathbf{u}\} ds \\ &\quad - \int_{\varepsilon_B} \mathbf{u}_T \cdot (\nabla \mathbf{v} \cdot \mathbf{n}) ds - \int_{\varepsilon_B} \mathbf{v}_T \cdot (\nabla \mathbf{u} \cdot \mathbf{n}) ds \\ &\quad + \int_{\varepsilon_I} \frac{\gamma}{|e|} [\mathbf{u}][\mathbf{v}] \, ds + \int_{\varepsilon_B} \frac{\gamma}{|e|} [\mathbf{u}]_T [\mathbf{v}]_T \, ds \\ l_2(\mathbf{v}) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \, dx - \int_{\varepsilon_B} (g\mathbf{t}) \cdot (\nabla \mathbf{v} \cdot \mathbf{n}) \, ds + \int_{\varepsilon_B} \frac{\alpha}{|e|} g [\mathbf{v}]_T \, ds - \int_{\varepsilon_B} \hat{h} \mathbf{v} \cdot \mathbf{n} \, ds. \end{aligned}$$

Here  $\mathbf{V}_h$  is same as the one in method 1 and  $\mathbf{u}_T = (\mathbf{u} \cdot \mathbf{t})\mathbf{t}$ , *i.e.* the tangential component of  $\mathbf{u}$ . Note that when one try to recover the original PDE (13) from the weak formulation (14), one will find a natural boundary condition for the PDE, namely,  $\mathbf{n}^T \mathbf{D}\mathbf{u} \cdot \mathbf{n} = \hat{h}$  on  $\partial\Omega$ . So in order to implement this method, it is assumed that we know the value of  $\hat{h}$  on the boundary, otherwise the it fails to be consistent. By taking  $\gamma = 10$ , we obtained the numerical results in Tables 10-12. The numerical results show that this

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u} - \mathbf{u}_h\ _{0,\Omega}$	Order
0.3536	1.0369e+00	1.1424e-02	-
0.1768	1.0340e+00	3.1035e-03	1.88014
0.0884	1.0331e+00	8.0643e-04	1.94427
0.0442	1.0329e+00	2.0542e-04	1.97298
0.0221	1.0328e+00	5.1832e-05	1.98666

Table 10:  $S_1$  in square domain: results by Matlab script DGlapl.m

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u} - \mathbf{u}_h\ _{0,\Omega}$	Order
0.5000	2.0963e+00	2.5162e-02	-
0.2500	2.0930e+00	1.0141e-02	1.31102
0.1250	2.0923e+00	4.0805e-03	1.31343
0.0625	2.0923e+00	1.6758e-03	1.28391
0.0312	2.0924e+00	7.2452e-04	1.20973

Table 11:  $S_2$  in L-shaped domain: results by Matlab script DGlapl.m

$h$	$\ \mathbf{u}_h\ _{0,\Omega}$	$\ \mathbf{u} - \mathbf{u}_h\ _{0,\Omega}$	Order
0.5000	8.0587e-01	6.7104e-01	-
0.2500	8.0212e-01	6.7827e-01	-0.01545
0.1250	8.0122e-01	6.8031e-01	-0.00435
0.0625	8.0105e-01	6.8084e-01	-0.00113
0.0312	8.0104e-01	6.8096e-01	-0.00024

Table 12:  $S_3$  in L-shaped domain: results by Matlab script DGlapl.m

method have similar performance as the first method, and the numerical solution also converges to the spurious solution for  $S_3$  in the L-shaped domain. Indeed, denote the solutions obtained from the two methods stated in this appendix section by  $\mathbf{u}^1$  and  $\mathbf{u}^2$ , we can see from Tables 13-15 that their differences are converging to zero, so the solutions of the two methods are actually converging to the same solution in all test cases.

$h$	$\ \mathbf{u}^1 - \mathbf{u}^2\ _{0,\Omega}$	Order
0.3536	4.1530e-03	-
0.1768	9.9064e-04	2.06774
0.0884	2.4009e-04	2.04480
0.0442	5.8978e-05	2.02530
0.0221	1.4609e-05	2.01334

Table 13:  $L^2$  difference of the two approximations of  $S_1$  in square domain

$h$	$\ \mathbf{u}^1 - \mathbf{u}^2\ _{0,\Omega}$	Order
0.5000	1.0393e-01	-
0.2500	6.7480e-02	0.62313
0.1250	4.2950e-02	0.65181
0.0625	2.7138e-02	0.66234
0.0312	1.7103e-02	0.66608

Table 14:  $L^2$  difference of the two approximations of  $S_2$  in L-shaped domain

$h$	$\ \mathbf{u}^1 - \mathbf{u}^2\ _{0,\Omega}$	Order
0.5000	7.8552e-02	-
0.2500	5.3783e-02	0.54651
0.1250	3.5621e-02	0.59441
0.0625	2.3123e-02	0.62338
0.0312	1.4833e-02	0.64059

Table 15:  $L^2$  difference of the two approximations of  $S_3$  in L-shaped domain

## B Matlab Codes

### MIXDG.m

```

% Run script for discontinuous Galerkin finite element solver
% Numerical results will be saved in the file MIXDG.txt

% Initialize constants
fid = fopen('MIXDG.txt','w');

alpha1 = 1; % parameter beta
alpha2 = 10; % parameter alpha
S = 1; % Symmetric (+1) or antisymmetric (-1) discretization
NREFS = 5; % Number of red refinement steps
clear Mesh
fprintf(fid,'Mixed DG with alpha = %10.5f, beta = %10.5f\n',alpha2,alpha1);
fprintf(fid,'\n');
SIGMA1 = @(P0,P1,varargin)alpha1*norm(P1-P0); % interior Edge weight function
SIGMA2 = @(P0,P1,varargin)alpha2/norm(P1-P0); % boundary Edge weight function

O_Handle = @(x,varargin)0*ones(size(x,1),1); % zero function
O_Handle2 = @(x,varargin)0*[ones(size(x,1),1) ones(size(x,1),1)];

for M_flag = 1:2
for F_flag = 1:3
clear Mesh

% Initialize source function and boundary datum for different exact solution
if F_flag == 1
G1 = @(x,varargin)-2*ones(size(x,1),1);
G2 = @(x,varargin)2*ones(size(x,1),1);
UD1 = @(x,varargin)x(:,2).*(1+x(:,2));
UD2 = @(x,varargin)x(:,1).*(1-x(:,1));
Curl_U = @(x,varargin)-2*(x(:,1)+x(:,2));
Div_U = @(x,varargin)0*ones(size(x,1),1);
fprintf(fid,'Regular function 1 ');
elseif F_flag == 2
G1 = @(x,varargin)0*ones(size(x,1),1);

```



```

T = B*D*transpose(B);
[IT, JT, T] = find(T);

[I2, J2, Jinn_n] = assemMat_Inn_DG2(Mesh, @STIMA_InnPen_Normal_DGLFE2, SIGMA1);
[I2, J2, Jinn_t] = assemMat_Inn_DG2(Mesh, @STIMA_InnPen_Tangent_DGLFE2, SIGMA2);

[I2, J2, Ainn_Curl] = assemMat_Inn_DG2(Mesh, @STIMA_Curl_Inn_DGLFE2, S);

[I3, J3, Jbnd] = assemMat_Bnd_DG2(Mesh, @STIMA_BndPen_Tangent_DGLFE2, SIGMA2);
[I3, J3, Abnd_Curl] = assemMat_Bnd_DG2(Mesh, @STIMA_Curl_Bnd_DGLFE2, S);

Lvols = assemLoad_Vol_DG2(Mesh, @LOAD_Vol_DGLFE2, QuadRule_2D, G1, G2);
Lbnds = assemLoad_Bnd_DG2(Mesh, @LOAD_Curl_Bnd_Tangent_DGLFE2, QuadRule_1D, S, SIGMA2, UD1,
    UD2);

h=get_MeshWidth(Mesh);

% Create system matrix

A = sparse([J1; J2; J3; JT], ...
           [I1; I2; I3; IT], ...
           [Avol_Curl+alpha*h^2*Avol_Div; Ainn_Curl+Jinn_t+Jinn_n; Abnd_Curl+Jbnd; T]);

L = Lvols+Lbnds;

U = A\L;

% Plot the numerical solution in the last refinement

if i == NREFS+1
    plot_DGLFE(U(1:2:end), Mesh);
    colorbar;
    plot_DGLFE(U(2:2:end), Mesh);
    colorbar;
end

% Compute and print the errors

MLW(i) = get_MeshWidth(Mesh);
Norm(i) = L2Err_DGLFE2(Mesh, U, P7O6(), O_Handle, O_Handle);
L2Err(i) = L2Err_DGLFE2(Mesh, U, P7O6(), UD1, UD2);
DGNorm(i) = EnergyErr_MIXDG(Mesh, U, P7O6, O_Handle, O_Handle, Curl_U, Div_U, SIGMA1, SIGMA2,
    alpha1, alpha2);
DGErr(i) = EnergyErr_MIXDG(Mesh, U, P7O6, UD1, UD2, Curl_U, Div_U, SIGMA1, SIGMA2, alpha1, alpha2
    );

% If one want to print the errors of multiplier p, use the following
%
%nv = size(Mesh.Coordinates, 1);
%p = zeros(nv, 1);
%p(FreeDofs) = D*B'*U;
%L2Err(i) = L2Err_LFE(Mesh, p, P7O6(), O_Handle);
%Norm(i) = L2Err(i);
%DGErr(i) = H1Err_LFE(Mesh, p, P7O6(), O_Handle, O_Handle2);
%DGNorm(i) = DGErr(i);

if i==1
    fprintf(fid, '%8.4f & %14.4e & %17.4e & - & %17.4e & %17.4e & \\\n', MLW(i), Norm(i), L2Err(i), DGNorm(i), DGErr(i));
else
    fprintf(fid, '%8.4f & %14.4e & %17.4e & %8.5f & %17.4e & %17.4e & \\\n', MLW(i), Norm(i), L2Err(i), log(L2Err(i-1)/L2Err(i))/log(2), DGNorm(i), DGErr(i), log(DGErr(i-1)/DGErr(i))/log(2));
end

% Refine the mesh after each computation
Mesh = refine_REG(Mesh);
clear A L U

```

```

end

% Save the errors and mesh width for plotting purpose
if M_flag==1
    if F_flag==1
        save MIXDG11.mat MW L2Err
    elseif F_flag==3
        save MIXDG13.mat MW L2Err
    elseif F_flag==4
        save MIXDG14.mat MW L2Err
    end
else
    if F_flag==1
        save MIXDG21.mat MW L2Err
    elseif F_flag==3
        save MIXDG23.mat MW L2Err
    elseif F_flag==4
        save MIXDG24.mat MW L2Err
    end
end

fprintf(fid, '\n');
end
end

fclose(fid);

% Clear memory

clear all

```

### SReg\_LFE2.m

```

% Run script for discontinuous Galerkin finite element solver
% Numerical results will be saved in the file SReg_LFE2.txt

% Initialize constants
fid = fopen('SReg_LFE2.txt', 'w');

NREFS = 5; % Number of red refinement steps
U_Handle = @(x, varargin) ones(size(x,1), 1);
O_Handle = @(x, varargin) 0*[ones(size(x,1), 1) ones(size(x,1), 1)];
clear Mesh
fprintf(fid, 'Strong regularization version with nodal element\n');
fprintf(fid, '\n');
for M_flag = 1:2
    for F_flag = 1:3
        clear Mesh
        if F_flag == 1
            G1 = @(x, varargin) -2*ones(size(x,1), 1);
            G2 = @(x, varargin) 2*ones(size(x,1), 1);
            UD1 = @(x, varargin) x(:, 2).*(1+x(:, 2));
            UD2 = @(x, varargin) x(:, 1).*(1-x(:, 1));
            fprintf(fid, 'Regular function 1 ');
        elseif F_flag == 2
            G1 = @(x, varargin) 0*ones(size(x,1), 1);
            G2 = @(x, varargin) 0*ones(size(x,1), 1);
            UD1 = @(x, varargin) 4/3*(x(:, 1).^2+x(:, 2).^2).^^(1/6).*sin(angle(x(:, 2), x(:, 1)))/3);
            UD2 = @(x, varargin) 4/3*(x(:, 1).^2+x(:, 2).^2).^^(1/6).*cos(angle(x(:, 2), x(:, 1)))/3);
            fprintf(fid, 'Singular function 1 ');
        elseif F_flag == 3
            G1 = @(x, varargin) 0*ones(size(x,1), 1);
            G2 = @(x, varargin) 0*ones(size(x,1), 1);
            UD1 = @(x, varargin) sing_fcn(x).*sin(-angle(x(:, 2), x(:, 1)))/3);
            UD2 = @(x, varargin) sing_fcn(x).*cos(-angle(x(:, 2), x(:, 1)))/3);
            fprintf(fid, 'Singular function 2 ');
        end
    end
end

```

```

F_Handle = @(x, varargin)[G1(x) G2(x)];
GD_Handle = @(x, varargin)[UD1(x) UD2(x)];

SIGMA = @(P0,P1, varargin)alpha/norm(P1-P0); % Edge weight function

% Initialize mesh
if M_flag == 1
Mesh.Coordinates = [0 0;1 0; 1 1; 0 1];
Mesh.Elements = [1 2 3; 1 3 4];
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;
fprintf(fid, 'in square domain\n');
else
Mesh = load_Mesh('Coord_LShap.dat', 'Elem_LShap.dat');
Mesh = add_Edges(Mesh);
Mesh = add_Edge2Elem(Mesh);
Loc = get_BdEdges(Mesh);
BdFlags = [-1 -2 -3 -4 -5 -6];
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
fprintf(fid, 'in L-shaped domain\n');
end
Mesh = refine_REG(Mesh);

fprintf(fid, 'Mesh Width      L2 Norm of exact solution      Relative L2 Error      Order      \n')
;
fprintf(fid, '=====      =====      =====      =====\n')
;

for i = 1:NREFS
Mesh = refine_REG(Mesh);
Mesh = add_Edge2Elem(Mesh);
Mesh = add_DGData(Mesh);

% Assemble matrices and load vectors (discontinuous Lagrangian elements)

QuadRule_1D = gauleg(0,1,2);
QuadRule_2D = P3O3();

nCoordinates = size(Mesh.Coordinates,1);
[IC,JC,C] = assemMat_LFE2(Mesh,@STIMA_Curl_LFE2,U_Handle,P7O6());
if M_flag == 1 | M_flag == 2
[ID,JD,D] = assemMat_LFE2(Mesh,@STIMA_Div_LFE2,U_Handle,P7O6());
else
[ID,JD,D] = assemMat_LFE2(Mesh,@STIMA_Div_Weighted2_LFE2,U_Handle,P7O6());
D = 2*D;
end
[U,g,FreeDofs,IB,JB,B] = assemDir_StrRegLFE2(Mesh,-1,GD_Handle);
A = sparse([IC;ID;IB+2*nCoordinates;JB],[JC;JD;JB;IB+2*nCoordinates],[C;D;B;B]);
l = assemLoad_LFE2(Mesh,P7O6(),F_Handle);
L = [1;g];
L = L - A*U;

% Solve the system

U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);
U = U(1:2*nCoordinates);

% if i ==NREFS
% figure
% plot_LFE2(U(1:end/2),Mesh);
% figure
% plot_LFE2(U(end/2+1:end),Mesh);
% end

```

```

        % Compute the errors
        L2Err(i) = L2Err_LFE2(Mesh,U,P7O6(),GD_Handle);
        MW(i) = get_MeshWidth(Mesh);
        Norm_Ex(i) = L2Err_LFE2(Mesh,zeros(size(U,1),1),P7O6(),GD_Handle);
        Norm(i) = L2Err_LFE2(Mesh,U,P7O6(),O_Handle);
        %L2Err(i) = L2Err(i)/Norm_Ex(i);
    if i==1
        fprintf(fid, '%8.4f & %20.4e & %17.4e & - \\\n',MW(i),Norm(i),L2Err
            (i));
    else
        fprintf(fid, '%8.4f & %20.4e & %17.4e & %6.5f\\n',MW(i),Norm(i),L2Err(i
            ),log(L2Err(i-1)/L2Err(i))/log(2));
    end
save SReg_Sol.mat Mesh U;
clear A L U
end

    if M_flag==1
        if F_flag==1
            save SReg11.mat MW L2Err
        elseif F_flag==3
            save SReg13.mat MW L2Err
        elseif F_flag==4
            save SReg14.mat MW L2Err
        end
    else
        if F_flag==1
            save SReg21.mat MW L2Err
        elseif F_flag==3
            save SReg23.mat MW L2Err
        elseif F_flag==4
            save SReg24.mat MW L2Err
        end
    end

fprintf(fid, '\n');
end
end
fclose(fid);

% Clear memory

clear all

```

### WReg\_mix.m

```

% Run script for discontinuous Galerkin finite element solver
% Numerical results will be saved in the file WReg_mix.txt

% Initialize constants
fid = fopen('WReg_mix.txt', 'w');

S = -1;          % Symmetric (+1) or antisymmetric (-1) discretization
NREFS = 5;      % Number of red refinement steps
U_Handle = @(x, varargin) ones(size(x,1),1);
O_Handle = @(x, varargin) 0*[ones(size(x,1),1) ones(size(x,1),1)];
clear Mesh
fprintf(fid, 'Weak regularization version with edge element\n');
fprintf(fid, '\n');
for M_flag = 1:2
for F_flag = 1:3
clear Mesh
if F_flag == 1
    G1 = @(x, varargin) -2*ones(size(x,1),1);
    G2 = @(x, varargin) 2*ones(size(x,1),1);
    UD1 = @(x, varargin) x(:,2).*(1+x(:,2));
    UD2 = @(x, varargin) x(:,1).*(1-x(:,1));

```

```

    fprintf(fid, 'Regular function 1 ');
elseif F_flag == 2
    G1 = @(x, varargin) 0*ones(size(x,1),1);
    G2 = @(x, varargin) 0*ones(size(x,1),1);
    UD1 = @(x, varargin) 4/3*(x(:,1).^2+x(:,2).^2).^(1/6).*sin(angle(x(:,2),x(:,1)))/3);
    UD2 = @(x, varargin) 4/3*(x(:,1).^2+x(:,2).^2).^(1/6).*cos(angle(x(:,2),x(:,1)))/3);
    fprintf(fid, 'Singular function 1 ');
elseif F_flag == 3
    G1 = @(x, varargin) 0*ones(size(x,1),1);
    G2 = @(x, varargin) 0*ones(size(x,1),1);
    UD1 = @(x, varargin) 2/3*(x(:,1).^2+x(:,2).^2).^(-1/6).*sin(-angle(x(:,2),x(:,1)))/3);
    UD2 = @(x, varargin) 2/3*(x(:,1).^2+x(:,2).^2).^(-1/6).*cos(-angle(x(:,2),x(:,1)))/3);
    fprintf(fid, 'Singular function 2 ');
end
F_Handle = @(x, varargin) [G1(x) G2(x)];
GD_Handle = @(x, varargin) [UD1(x) UD2(x)];

SIGMA = @(P0,P1, varargin) alpha/norm(P1-P0); % Edge weight function

% Initialize mesh
if M_flag == 1
    Mesh.Coordinates = [0 0;1 0; 1 1; 0 1];
    Mesh.Elements = [1 2 3; 1 3 4];
    fprintf(fid, 'in square domain\n');
else
    Mesh = load_Mesh('Coord_LShap.dat', 'Elem_LShap.dat');
    fprintf(fid, 'in L-shaped domain\n');
end
Mesh.ElemFlag = ones(size(Mesh.Elements),1);
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges),1);
Mesh.BdFlags(Loc) = -1;
Mesh = refine_REG(Mesh);

fprintf(fid, 'Mesh Width      L2 Norm of exact solution      Relative L2 Error      Order      \n')
;
fprintf(fid, '=====      =====      =====      =====\n')
;

for i = 1:NREFS
    Mesh = refine_REG(Mesh);
    Mesh = add_Edge2Elem(Mesh);
    Mesh = add_DGData(Mesh);

% Assemble matrices and load vectors (discontinuous Lagrangian elements)

QuadRule_1D = gauleg(0,1,10);
QuadRule_2D = P3O3();

[IC,JC,C] = assemMat_W1F(Mesh,@STIMA_Curl_W1F,U_Handle,P7O6());
B = assemMat_WRegW1F(Mesh,@STIMA_WReg_W1F);
D = assemMat_LFE(Mesh,@MASS_Lump_LFE);

nCoordinates = size(Mesh.Coordinates,1);
Loc = get_BdEdges(Mesh);
DEdges = Loc(Mesh.BdFlags(Loc) == -1);
DNodes = unique([Mesh.Edges(DEdges,1); Mesh.Edges(DEdges,2)]);
FreeDofs = setdiff(1:nCoordinates, DNodes);
B = B(:,FreeDofs);
D = D(FreeDofs,FreeDofs);

T = B*inv(D)*transpose(B);
[IT,JT,T] = find(T);
A = sparse([IC;IT],[JC;JT],[C;T]);
L = assemLoad_W1F(Mesh,P7O6(),F_Handle);

% Incorporate Dirichlet boundary data

```

```

[U,FreeDofs] = assemDir_W1F(Mesh,-1,GD_Handle,gauleg(0,1,1));
L = L - A*U;

% Solve the system

U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);

% Compute the errors
L2Err(i) = L2Err_W1F(Mesh,U,P7O6(),GD_Handle);
MW(i) = get_MeshWidth(Mesh);
Norm_Ex(i) = L2Err_W1F(Mesh,zeros(size(U,1),1),P7O6(),GD_Handle);
Norm(i) = L2Err_W1F(Mesh,U,P7O6(),O_Handle);
% L2Err(i) = L2Err(i)/Norm_Ex(i);
if i==1
    fprintf(fid,'%8.4f & %20.4e & %17.4e & - \\\ \n',MW(i),Norm(i),L2Err(i));
else
    fprintf(fid,'%8.4f & %20.4e & %17.4e & %6.5f\\ \n',MW(i),Norm(i),L2Err(i),log(L2Err(i-1)/L2Err(i))/log(2));
end
clear A L U
end

if M_flag==1
    if F_flag==1
        save WReg11.mat MW L2Err
    elseif F_flag==3
        save WReg13.mat MW L2Err
    elseif F_flag==4
        save WReg14.mat MW L2Err
    end
else
    if F_flag==1
        save WReg21.mat MW L2Err
    elseif F_flag==3
        save WReg23.mat MW L2Err
    elseif F_flag==4
        save WReg24.mat MW L2Err
    end
end

fprintf(fid,'\n');
end
end
fclose(fid);
% Clear memory

clear all

```

### DGcurl.m

```

% Run script for discontinuous Galerkin finite element solver
% Numerical results will be saved in the file DGcurl.txt

% Initialize constants
fid = fopen('DGcurl.txt','w');

alpha1 = 10;
alpha2 = 10;
S = 1; % Symmetric (+1) or antisymmetric (-1) discretization
NREFS = 3; % Number of red refinement steps
clear Mesh
SIGMA1 = @(P0,P1,varargin)alpha1/norm(P1-P0); % interior Edge weight function
SIGMA2 = @(P0,P1,varargin)alpha2/norm(P1-P0); % boundary Edge weight function
fprintf(fid,'DG version with IP-DGFEM with alpha = %10.5f, beta = %10.5f\n',alpha2,
alpha1);
fprintf(fid,'\n');

```



```

QuadRule_1D = gauleg(0,1,10);
QuadRule_2D = P3O3();

[IC1,JC1,Avol_Curl] = assemMat_Vol_DG2(Mesh,@STIMA_Curl_LFE2);
[ID1,JD1,Avol_Div] = assemMat_Vol_DG2(Mesh,@STIMA_Div_LFE2);

[It2,Jt2,Jinn_t] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_Tangent_DGLFE2,SIGMA2);
[In2,Jn2,Jinn_n] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_Normal_DGLFE2,SIGMA1);
[IC2,JC2,Ainn_Curl] = assemMat_Inn_DG2(Mesh,@STIMA_Curl_Inn_DGLFE2,S);
[ID2,JD2,Ainn_Div] = assemMat_Inn_DG2(Mesh,@STIMA_Div_Inn_DGLFE2,S);

[It3,Jt3,Jbnd] = assemMat_Bnd_DG2(Mesh,@STIMA_BndPen_Tangent_DGLFE2,SIGMA2);
[IC3,JC3,Abnd_Curl] = assemMat_Bnd_DG2(Mesh,@STIMA_Curl_Bnd_DGLFE2,S);

Lvol = assemLoad_Vol_DG2(Mesh,@LOAD_Vol_DGLFE2,QuadRule_2D,G1,G2);
Lbnd = assemLoad_Bnd_DG2(Mesh,@LOAD_Curl_Bnd_Tangent_DGLFE2,QuadRule_1D,S,SIGMA2,UD1,
    UD2);
h=get_MeshWidth(Mesh);

% Create system matrix
A = sparse([JC1; JD1; JC2; JD2; JC3; Jt2; Jn2; Jt3], ...
    [IC1; ID1; IC2; ID2; IC3; It2; In2; It3],...
    [Avol_Curl; Avol_Div; Ainn_Curl; Ainn_Div; Abnd_Curl; Jinn_t; Jinn_n; Jbnd]);
L = Lvol + Lbnd;

% Solve the linear system
U = A\L;

M_Curl = sparse([JC1], ...
    [IC1], ...
    [Avol_Curl]);
M_Div = sparse([JD1], ...
    [ID1], ...
    [Avol_Div]);
X = sparse([Jt2;Jn2;Jt3], ...
    [It2;Jn2;It3], ...
    [1/alpha2*Jinn_t;1/alpha1*Jinn_n;0/alpha2*Jbnd]);
Y = sparse([Jn2;Jt3], ...
    [In2;It3], ...
    [0/alpha2*Jinn_t;1/alpha2*Jbnd]);

% Interpolation of exaction solution
UE = L2_interp_DGLFE(Mesh,P7O6(),UD1,UD2);

% Interpolation of exaction solution
if i == NREFS+1
plot_DGLFE(U(1:2:end),Mesh);
colorbar;

plot_DGLFE(U(2:2:end),Mesh);
colorbar;
end

% Compute the errors
MW(i) = get_MeshWidth(Mesh);
Norm(i) = L2Err_DGLFE2(Mesh,U,P7O6(),O_Handle,O_Handle);
L2Err(i) = L2Err_DGLFE2(Mesh,U,P7O6(),UD1,UD2);
DGNorm(i) = EnergyErr_DGcurl(Mesh,U,P7O6,O_Handle,O_Handle,Curl_U,Div_U,SIGMA1,SIGMA2,
    alpha1,alpha2);
DGErr(i) = EnergyErr_DGcurl(Mesh,U,P7O6,UD1,UD2,Curl_U,Div_U,SIGMA1,SIGMA2,alpha1,
    alpha2);

```

```

if i==1
    fprintf(fid, '%8.4f & %14.4e & %17.4e & - & %17.4e & %17.4e &
- \\\ \n', MW(i), Norm(i), L2Err(i), DGNorm(i), DGErr(i));
else
    fprintf(fid, '%8.4f & %14.4e & %17.4e & %8.5f & %17.4e & %17.4e &
%8.5f \\\ \n', MW(i), Norm(i), L2Err(i), log(L2Err(i-1)/L2Err(i))/log(2), DGNorm(i),
DGErr(i), log(DGErr(i-1)/DGErr(i))/log(2));
end

save Soln_Curl.mat U Mesh
Mesh = refine_REG(Mesh);
clear A L U
end

if M_flag==1
    if F_flag==1
        save DGcurl11.mat MW L2Err
    elseif F_flag==3
        save DGcurl13.mat MW L2Err
    elseif F_flag==4
        save DGcurl14.mat MW L2Err
    end
else
    if F_flag==1
        save DGcurl21.mat MW L2Err
    elseif F_flag==3
        save DGcurl23.mat MW L2Err
    elseif F_flag==4
        save DGcurl24.mat MW L2Err
    end
end

fprintf(fid, '\n');
end
end

fclose(fid);

% Clear memory
clear all

```

### DGlapl.m

```

% Run script for discontinuous Galerkin finite element solver
% Numerical results will be saved in the file DGlapl.txt

% Initialize constants

fid = fopen('DGlapl.txt', 'w');

alpha = 10;
S = 1; % Symmetric (+1) or antisymmetric (-1) discretization
NREFS = 3; % Number of red refinement steps
fprintf(fid, 'DG Laplacian version with IP-DGFEM\n');
fprintf(fid, '\n');
O_Handle = @(x, varargin) 0*ones(size(x,1), 1);

for M_flag = 1:2
    for F_flag = 1:3
        clear Mesh
        if F_flag == 1
            G1 = @(x, varargin) -2*ones(size(x,1), 1);
            G2 = @(x, varargin) 2*ones(size(x,1), 1);
            UD1 = @(x, varargin) x(:, 2).*(1+x(:, 2));
            UD2 = @(x, varargin) x(:, 1).*(1-x(:, 1));
            grad11 = @(x, varargin) 0*ones(size(x,1), 1);

```



```

QuadRule_1D = gauleg(0,1,10);
QuadRule_2D = P3O3();

[I1 ,J1 ,Avol] = assemMat_Vol_DG2(Mesh,@STIMA_Lapl_Vol_DGLFE2);

[I2 ,J2 ,Jinn] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_DGLFE2,SIGMA);
[I2 ,J2 ,Ainn] = assemMat_Inn_DG2(Mesh,@STIMA_Grad_Inn_DGLFE2,S);

[I3 ,J3 ,Jbnd] = assemMat_Bnd_DG2(Mesh,@STIMA_BndPen_Tangent_DGLFE2,SIGMA);
[I3 ,J3 ,Abnd] = assemMat_Bnd_DG2(Mesh,@STIMA_Grad_Bnd_Tangent_DGLFE2,S);

Lv1 = assemLoad_Vol_DG2(Mesh,@LOAD_Vol_DGLFE2,QuadRule_2D,G1,G2);
Lbnd = assemLoad_Bnd_DG2(Mesh,@LOAD_Grad_Bnd_Tangent_DGLFE2,QuadRule_1D,S,SIGMA,UD1,UD2);
Lbnd_Du = assemLoad_Bnd_DG2(Mesh,@LOAD_Du_Bnd_Normal_DGLFE2,QuadRule_1D,S,SIGMA,grad11,grad12,grad21,grad22);

% Create system matrix

A = sparse([J1; J2; J3; J2; J3], ...
           [I1; I2; I3; I2; I3], ...
           [Avol; Ainn; Abnd; Jinn; Jbnd]);
L = Lv1 + Lbnd - Lbnd_Du;

% Solve the linear system

U = A\L;

if i == NREFS+1
plot_DGLFE(U(1:2:end),Mesh);
colorbar;

plot_DGLFE(U(2:2:end),Mesh);
colorbar;
end

% Interpolation of exaction solution

UE = L2_interp_DGLFE(Mesh,P7O6(),UD1,UD2);

% Compute the errors
L2Err1 = L2Err_DGLFE(Mesh,U(1:2:end),P7O6(),UD1);
L2Err2 = L2Err_DGLFE(Mesh,U(2:2:end),P7O6(),UD2);
norm_Ex1 = L2Err_DGLFE(Mesh,zeros(size(U,1)/2,1),P7O6(),UD1);
norm_FE1 = L2Err_DGLFE(Mesh,U(1:2:end),P7O6(),O_Handle);
norm_Ex2 = L2Err_DGLFE(Mesh,zeros(size(U,1)/2,1),P7O6(),UD2);
norm_FE2 = L2Err_DGLFE(Mesh,U(2:2:end),P7O6(),O_Handle);
MW(i) = get_MeshWidth(Mesh);
Norm_Ex(i) = sqrt(norm_Ex1^2+norm_Ex2^2);
Norm(i) = sqrt(norm_FE1^2+norm_FE2^2);
L2Err(i) = sqrt(L2Err1^2+L2Err2^2);
A1 = sparse([J1;J3], ...
            [I1;I3], ...
            [Avol;0*Jbnd]);
A2 = sparse([J2], ...
            [I2], ...
            [Jinn]);
A3 = sparse([J3;J1], ...
            [I3;I1], ...
            [Jbnd;0*Avol]);
DGNorm(i) = sqrt(U'*A1*U+U'*A2*U+U'*A3*U);
DGErr(i) = sqrt((U-UE)'*A1*(U-UE)+(U-UE)'*A2*(U-UE)+(U-UE)'*A3*(U-UE));

if i==1
fprintf(fid, '%8.4f & %20.4e & %17.4e & - & %17.4e & %17.4e & \\\n',MW(i),Norm(i),L2Err(i),DGNorm(i),DGErr(i));
else

```

```

    fprintf(fid, '%8.4f & %20.4e & %17.4e & %8.5f & %17.4e & %17.4e &
      %8.5f\\ \\ \\ \\n', MLW(i), Norm(i), L2Err(i), log(L2Err(i-1)/L2Err(i))/log(2), DGNorm(i),
      DGErr(i), log(DGErr(i-1)/DGErr(i))/log(2));
end
Mesh = refine_REG(Mesh);
clear A L U
end
fprintf(fid, '\\n');
end
end

fclose(fid);

% Clear memory

clear all

```

### assemLoad\_Bnd\_DG2.m

```

function L = assemLoad_Bnd_DG2(Mesh, EHandle, varargin)
% ASSEMBLOAD_BND_DG2 Assemble vectorial discontinuous Lagrangian nodal FE boundary edge
% contributions.
%
% A = ASSEMBLOAD_BND_DG2(MESH, EHANDLE) assembles the global load vector
% from the local element contributions given by the function handle
% EHANDLE.
%
% The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 matrix specifying the elements of the mesh.
% EDGES N-by-2 matrix specifying all edges of the mesh.
% BDFLAGS P-by-1 matrix specifying the boundary type of each
% boundary edge in the mesh.
% VERT2EDGE M-by-M sparse matrix which specifies whether the two
% vertices i and j are connected by an edge with number
% VERT2EDGE(i, j).
% EDGE2ELEM P-by-2 matrix connecting edges to elements. The first
% column specifies the left hand side element where the
% second column specifies the right hand side element.
% EDGELOC P-by-3 matrix connecting edges to local edges of elements.
% NORMALS P-by-2 matrix specifying the normals on each edge. The
% normals on interior edges are chosen such that they point
% from the element with the lower number to the element
% with the higher number and on boundary edges such that
% they point outside the domain.
% MATCH P-by-2 matrix specifying wheter the edge orientation of
% the current edge matches the orientation of the left and
% right hand side element.

% Initialize constants

nEdges = size(Mesh.Edges, 1); % Number of edges
nElements = size(Mesh.Elements, 1); % Number of elements

% Preallocate memory

L = zeros(6*nElements, 1);

% Check for element flags

if(isfield(Mesh, 'ElemFlag')),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements, 1);
end
BdFlags = Mesh.BdFlags;

% Assemble element contributions

```

```

for i = 1:nEdges

    % Check for boundary edge

    if(BdFlags(i) < 0)

        Edge = Mesh.Coordinates(Mesh.Edges(i,:),:);
        Normal = Mesh.Normals(i,:);

        % Extract left or right hand side element data

        if(Mesh.Edge2Elem(i,1) > 0)
            Data.Element = Mesh.Edge2Elem(i,1);
            Data.ElemFlag = ElemFlag(Data.Element);
            Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
            Data.EdgeLoc = Mesh.EdgeLoc(i,1);
            Data.Match = Mesh.EdgeOrient(i,1);
        else
            Data.Element = Mesh.Edge2Elem(i,2);
            Data.ElemFlag = ElemFlag(Data.Element);
            Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
            Data.EdgeLoc = Mesh.EdgeLoc(i,2);
            Data.Match = Mesh.EdgeOrient(i,2);
        end

        % Compute element contributions

        Lloc = EHandle(Edge,Normal,BdFlags(i), ...
            Data,varargin{:});

        % Add element contributions to load vector

        idx = 6*(Data.Element-1) + [1 2 3 4 5 6];

        L(idx) = L(idx) + Lloc;

    end
end
return

```

#### assemLoad\_Vol\_DG2.m

```

function L = assemLoad_Vol_DG2(Mesh,EHandle,varargin)
% ASSEMBLOAD_VOL_DG2 Assemble vectorial discontinuous Lagrangian nodal FE volume
% contributions.
%
% L = ASSEMBLOAD_VOL_DG2(MESH,EHANDLE) assembles the global load vector
% from the local element contributions given by the function handle
% EHANDLE.
%
% A = ASSEMBLOAD_VOL_DG2(MESH,EHANDLE,EPARAM) handles the variable length
% argument list EPARAM to the function handle EHANDLE during the assembly
% process.
%
% The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 matrix specifying the elements of the mesh.
% ELEMFLAG N-by-1 matrix specifying additional element information.

% Initialize constants

nElements = size(Mesh.Elements,1); % Number of elements

% Preallocate memory

L = zeros(6*nElements,1);

```

```

% Check for element flags
if (isfield(Mesh, 'ElemFlag')),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements,1);
end
BdFlags = Mesh.BdFlags;

% Assemble element contributions

for i = 1:nElements

    % Extract vertices

    vidx = Mesh.Elements(i,:);

    % Compute element contributions

    Lloc = EHandle(Mesh.Coordinates(vidx,:), ElemFlag(i), varargin{:});

    % Add contributions to global load vector

    idx = 6*(i-1)+[1 2 3 4 5 6];

    L(idx) = L(idx)+Lloc;

end

return

```

#### assemMat\_Bnd\_DG2.m

```

function varargout = assemMat_Bnd_DG2(Mesh, EHandle, varargin)
% ASSEMMAT_BND_DG2 Assemble vectorial discontinuous Lagrangian nodal FE boundary edge
% contributions.
%
% A = ASSEMMAT_BND_DG2(MESH, EHANDLE) assembles the global matrix from the
% local element contributions given by the function handle EHANDLE and
% returns the matrix in a sparse representation.
%
% [I, J, A] = ASSEMMAT_BND_DG(MESH, EHANDLE) assembles the global matrix
% from the local element contributions given by the function handle
% EHANDLE and returns the matrix in an array representation.
%
% The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 matrix specifying the elements of the mesh.
% EDGES N-by-2 matrix specifying all edges of the mesh.
% BDFLAGS P-by-1 matrix specifying the boundary type of each
% boundary edge in the mesh.
% VERT2EDGE M-by-M sparse matrix which specifies whether the two
% vertices i and j are connected by an edge with number
% VERT2EDGE(i, j).
% EDGE2ELEM P-by-2 matrix connecting edges to elements. The first
% column specifies the left hand side element where the
% second column specifies the right hand side element.
% EDGELOC P-by-3 matrix connecting edges to local edges of elements.
% NORMALS P-by-2 matrix specifying the normals on each edge. The
% normals on interior edges are chosen such that they point
% from the element with the lower number to the element
% with the higher number and on boundary edges such that
% they point outside the domain.
% MATCH P-by-2 matrix specifying wheter the edge orientation of
% the current edge matches the orientation of the left and
% right hand side element.
%

```

```

% Initialize constants
nElements = size(Mesh.Elements,1); % Number of elements in the mesh
nEdges = size(Mesh.Edges,1); % Number of edges in the mesh

% Allocate memory
I = zeros(36*nEdges,1);
J = zeros(36*nEdges,1);
A = zeros(36*nEdges,1);

% Check for element flags
if(isfield(Mesh,'ElemFlag')),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements,1);
end
BdFlags = Mesh.BdFlags;

% Assemble element contributions

loc = 1:36;
last = 0;
for i = 1:nEdges

    % Check for boundary edge

    if(BdFlags(i) < 0)

        Edge = Mesh.Coordinates(Mesh.Edges(i,:),:);
        Normal = Mesh.Normals(i,:);

        % Extract left or right hand side element data

        if(Mesh.Edge2Elem(i,1) > 0)
            Data.Element = Mesh.Edge2Elem(i,1);
            Data.ElemFlag = ElemFlag(Data.Element);
            Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
            Data.EdgeLoc = Mesh.EdgeLoc(i,1);
            Data.Match = Mesh.EdgeOrient(i,1);
        else
            Data.Element = Mesh.Edge2Elem(i,2);
            Data.ElemFlag = ElemFlag(Data.Element);
            Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
            Data.EdgeLoc = Mesh.EdgeLoc(i,2);
            Data.Match = Mesh.EdgeOrient(i,2);
        end

        % Compute element contributions

        Aloc = EHandle(Edge,Normal,BdFlags(i), ...
            Data,varargin{:});

        % Add element contributions to stiffness matrix

        idx = 6*(Data.Element-1)+[1 2 3 4 5 6];

        I(loc) = set_Rows(idx,6);
        J(loc) = set_Cols(idx,6);
        A(loc) = Aloc(:);

        loc = loc + 36;
        last = last + 36;

    end

end
end

```

```

% Assign output arguments

loc = 1:last;
if(nargout > 1)
    varargout{1} = I(loc);
    varargout{2} = J(loc);
    varargout{3} = A(loc);
else
    varargout{1} = sparse(I(loc),J(loc),A(loc),6*nElements,6*nElements);
end
return

```

### assemMat\_Inn\_DG2.m

```

function varargout = assemMat_Inn_DG2(Mesh,EHandle,varargin)
% ASSEMMAT_INN_DG2 Assemble vectorial discontinuous Lagrangian nodal FE interior edge
% contributions.
%
% A = ASSEMMAT_INN_DG2(MESH,EHANDLE) assembles the global matrix from the
% local element contributions given by the function handle EHANDLE and
% returns the matrix in a sparse representation.
%
% [I,J,A] = ASSEMMAT_INN_DG2(MESH,EHANDLE) assembles the global matrix
% from the local element contributions given by the function handle
% EHANDLE and returns the matrix in an array representation.
%
% The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 matrix specifying the elements of the mesh.
% EDGES N-by-2 matrix specifying all edges of the mesh.
% BDFLAGS P-by-1 matrix specifying the boundary type of each
% boundary edge in the mesh.
% VERT2EDGE M-by-M sparse matrix which specifies whether the two
% vertices i and j are connected by an edge with number
% VERT2EDGE(i,j).
% EDGE2ELEM P-by-2 matrix connecting edges to elements. The first
% column specifies the left hand side element where the
% second column specifies the right hand side element.
% EDGELOC P-by-3 matrix connecting edges to local edges of elements.
% NORMALS P-by-2 matrix specifying the normals on each edge. The
% normals on interior edges are chosen such that they point
% from the element with the lower number to the element
% with the higher number and on boundary edges such that
% they point outside the domain.
% MATCH P-by-2 matrix specifying wheter the edge orientation of
% the current edge matches the orientation of the left and
% right hand side element.

% Initialize constants
nEdges = size(Mesh.Edges,1); % Number of edges
nElements = size(Mesh.Elements,1); % Number of elements

% Allocate memory
I = zeros(144*nEdges,1);
J = zeros(144*nEdges,1);
A = zeros(144*nEdges,1);

% Check for element and boundary flags
if(isfield(Mesh,'ElemFlag')),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements,1);
end

```

```

BdFlags = Mesh.BdFlags;

% Assemble element contributions

loc = 1:144;
last = 0;
for i = 1:nEdges

    % Check for interior edge

    if(BdFlags(i) >= 0)

        % Extract edge data

        Edge = Mesh.Coordinates(Mesh.Edges(i,:),:);
        Normal = Mesh.Normals(i,:);

        % Extract left and right hand side element data

        LData.Element = Mesh.Edge2Elem(i,1);
        LData.ElemFlag = ElemFlag(LData.Element);
        LData.Vertices = Mesh.Coordinates(Mesh.Elements(LData.Element,:),:);
        LData.EdgeLoc = Mesh.EdgeLoc(i,1);
        LData.Match = Mesh.EdgeOrient(i,1);

        RData.Element = Mesh.Edge2Elem(i,2);
        RData.ElemFlag = ElemFlag(RData.Element);
        RData.Vertices = Mesh.Coordinates(Mesh.Elements(RData.Element,:),:);
        RData.EdgeLoc = Mesh.EdgeLoc(i,2);
        RData.Match = Mesh.EdgeOrient(i,2);

        % Compute element contributions

        Aloc = EHandle(Edge,Normal,BdFlags(i), ...
            LData,RData,varargin{:});

        % Add contributions to stiffness matrix

        idx_l = 6*(LData.Element-1)+[1 2 3 4 5 6];
        idx_r = 6*(RData.Element-1)+[1 2 3 4 5 6];
        idx = [idx_l idx_r];

        I(loc) = set_Rows(idx,12);
        J(loc) = set_Cols(idx,12);
        A(loc) = Aloc(:);

        loc = loc+144;
        last = last+144;

    end

end

% Assign output arguments

loc = 1:last;
if(nargout > 1)
    varargout{1} = I(loc);
    varargout{2} = J(loc);
    varargout{3} = A(loc);
else
    varargout{1} = sparse(I(loc),J(loc),A(loc));
end

return

```

```

function varargout = assemMat_MIXDG(Mesh,EHandle,varargin)
% ASSEMMAT_MIXDG Assemble vectorial discontinuous Lagrangian nodal FE volume
% contributions.

% A = ASSEMMAT_MIXDG(MESH,EHANDLE) assembles the global matrix from the
% local element contributions given by the function handle EHANDLE and
% returns the matrix in a sparse representation.
%
% [I,J,A] = ASSEMMAT_MIXDG(MESH,EHANDLE) assembles the global matrix from
% the local element contributions given by the function handle EHANDLE
% and returns the matrix in an array representation.
%
% The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the
% mesh.
% ELEMFLAG N-by-1 matrix specifying additional element information.

% Initialize constants

nElements = size(Mesh.Elements,1);
nCoordinates = size(Mesh.Coordinates,1);

% Preallocate memory

I = zeros(18*nElements,1);
J = zeros(18*nElements,1);
A = zeros(18*nElements,1);

if(isfield(Mesh,'ElemFlag'),
    ElemFlags = Mesh.ElemFlag;
else
    ElemFlags = zeros(nElements,1);
end

% Assemble element contributions

loc = 1:18;
for i = 1:nElements

    % Extract vertices of current element

    vidx = Mesh.Elements(i,:);

    Vertices = Mesh.Coordinates(vidx,:);

    % Compute element contributions

    Aloc = EHandle(Vertices,Mesh.ElemFlag(i),varargin{:});

    % Add contributions to stiffness matrix

    J(loc) = set_Cols(vidx,6);
    idx = 6*(i-1)+[1 2 3 4 5 6];
    I(loc) = set_Rows(idx,3);
    A(loc) = Aloc(:);
    loc = loc+18;

end

% Assign output arguments

if(nargout > 1)
    varargout{1} = I;
    varargout{2} = J;
    varargout{3} = A;
else
    varargout{1} = sparse(I,J,A);

```

```

end
return

```

### assemMat\_Vol\_DG2.m

```

function varargout = assemMat_Vol_DG2(Mesh,EHandle,varargin)
% ASSEMMAT_VOL_DG2 Assemble vectorial discontinuous Lagrangian nodal FE volume
% contributions.
%
% A = ASSEMMAT_VOL_DG2(MESH,EHANDLE) assembles the global matrix from the
% local element contributions given by the function handle EHANDLE and
% returns the matrix in a sparse representation.
%
% [I,J,A] = ASSEMMAT_VOL_DG2(MESH,EHANDLE) assembles the global matrix
% from the local element contributions given by the function handle
% EHANDLE and returns the matrix in an array representation.
%
% The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 matrix specifying the elements of the mesh.

% Initialize constants

nElements = size(Mesh.Elements,1); % Number of elements

% Allocate memory

I = zeros(36*nElements,1);
J = zeros(36*nElements,1);
A = zeros(36*nElements,1);

% Check for element flags

if(isfield(Mesh,'ElemFlag')),
    ElemFlags = Mesh.ElemFlag;
else
    ElemFlags = zeros(nElements,1);
end

% Assemble element contributions

loc = 1:36;
for i = 1:nElements

    % Extract vertices of current element

    Vertices = Mesh.Coordinates(Mesh.Elements(i,:),:);

    % Compute element contributions

    Aloc = EHandle(Vertices,varargin{:});

    % Add contributions to stiffness matrix

    idx = 6*(i-1)+[1 2 3 4 5 6];

    I(loc) = set_Rows(idx,6);
    J(loc) = set_Cols(idx,6);
    A(loc) = Aloc(:);

    loc = loc+36;

end

% Assign output arguments

if(nargout > 1)

```

```

    varargout{1} = I;
    varargout{2} = J;
    varargout{3} = A;
    else
        varargout{1} = sparse(I,J,A);
    end
return

```

### EnergyErr\_DGcurl.m

```

function err = EnergyErr_DGcurl(Mesh,u,QuadRule,UHandle1,UHandle2,CurlUHandle,
    DivUHandle,SIGMA1,SIGMA2,alpha1,alpha2,varargin)
%EnergyErr_DGcurl DG energy-norm of discretization error of the discontinuous galerkin
  solution
%
% ERR = ENERGYERR_DGcurl(Mesh,u,QuadRule,UHANDLE1,UHANDLE2,CurlUHANDLE,DivUHANDLE,
% SIGMA1,SIGMA2,alpha1,alpha2,varargin)
% computes the DG energy-norm of discretization error of the
% discontinuous galerkin solution given by the vector U
% compared to the exact solution given by the
% function handles UHANDLE1 and UHANDLE2.
%
% The struct MESH should contain at least the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the
% mesh.
% ELEMDATA N-by-1 structure array containing at least the fields:
% NDOFS The number of degrees of freedom on the corresponding
% element.
% DIR A NDOFS-by-2 matrix containing the propagation
% directions of the plane wave basis functions in
% its rows.
%
% QUADRULE is a struct, which specifies the Gauss quadrature that is used
% to do the integration:
% W Weights of the Gauss quadrature.
% X Abscissae of the Gauss quadrature.
%
% UHANDLE1 is a function handle for the first component of the exact solution and
% UHANDLE2 is a
% function handle for the second component of the exact solution.
%
% CurlUHANDLE and DivUHANDLE are function handles for the curl and
% divergence of the exact solution, respectively.
%
% SIGMA1 and SIGMA2 are function handles for the stabilization term in
% the normal and tangential jump, respectively.
%
% Intialize constants
nPts = size(QuadRule.w,1);
nElements = size(Mesh.Elements,1);
nCoordinates = size(Mesh.Coordinates,1);
nEdges = size(Mesh.Edges,1);
%nVert = size(Mesh.Elements,2);
%nDofs = [Mesh.ElemData.nDofs]; % Number of degrees of freedom on element
%nDofsSum = cumsum([0,nDofs]); % Number of degrees of freedom on all preceding
    elements
%
% Compute volume terms of discretization error
%
if(isfield(Mesh,'ElemFlag'),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements,1);
end
% Precompute shape function values at the quadrature points

```

```

N = shap_DGLFE(QuadRule.x);

err = 0;
for i = 1:nElements

    % Extract vertex and basis function numbers
    vidx = Mesh.Elements(i,:);
    idx = 6*(i-1)+[1 2 3 4 5 6];

    % Compute element mapping
    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
    det_BK = abs(det(BK));

    dNC = -[ Mesh.Coordinates(vidx(3),:) - Mesh.Coordinates(vidx(2),:) ...
            Mesh.Coordinates(vidx(1),:) - Mesh.Coordinates(vidx(3),:) ...
            Mesh.Coordinates(vidx(2),:) - Mesh.Coordinates(vidx(1),:) ]/det_BK;

    dND = [ Mesh.Coordinates(vidx(2),2) - Mesh.Coordinates(vidx(3),2) ...
            Mesh.Coordinates(vidx(3),1) - Mesh.Coordinates(vidx(2),1) ...
            Mesh.Coordinates(vidx(3),2) - Mesh.Coordinates(vidx(1),2) ...
            Mesh.Coordinates(vidx(1),1) - Mesh.Coordinates(vidx(3),1) ...
            Mesh.Coordinates(vidx(1),2) - Mesh.Coordinates(vidx(2),2) ...
            Mesh.Coordinates(vidx(2),1) - Mesh.Coordinates(vidx(1),1) ]/det_BK;

    % Transform quadrature points
    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions

    u_EX1 = UHandle1(x,ElemFlag(i),varargin{:});
    u_EX2 = UHandle2(x,ElemFlag(i),varargin{:});
    u_FE1 = u(idx(1))*N(:,1) + u(idx(3))*N(:,2) + u(idx(5))*N(:,3);
    u_FE2 = u(idx(2))*N(:,1) + u(idx(4))*N(:,2) + u(idx(6))*N(:,3);

    % Evaluate curl and divergence of solutions
    Curl_u_FE = u(idx(1))*dNC(1) + u(idx(2))*dNC(2) + u(idx(3))*dNC(3) + u(idx(4))*dNC(4)
                + u(idx(5))*dNC(5) + u(idx(6))*dNC(6);
    Curl_u_EX = Curl_UHandle(x,varargin{:});

    Div_u_FE = u(idx(1))*dND(1) + u(idx(2))*dND(2) + u(idx(3))*dND(3) + u(idx(4))*dND(4)
                + u(idx(5))*dND(5) + u(idx(6))*dND(6);
    Div_u_EX = Div_UHandle(x,varargin{:});

    % Compute error on current element
    err = err+sum(QuadRule.w.*abs(u_EX1-u_FE1).^2)*det_BK+sum(QuadRule.w.*abs(u_EX2-u_FE2)
                ).^2)*det_BK+sum(QuadRule.w.*abs(Curl_u_EX-Curl_u_FE).^2)*det_BK+sum(QuadRule.w.*
                abs(Div_u_EX-Div_u_FE).^2)*det_BK;

end

% Handle interior jump terms
[I2,J2,Jinn_n] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_Normal_DGLFE2,SIGMA1);
[I2,J2,Jinn_t] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_Tangent_DGLFE2,SIGMA2);
[I3,J3,Jbnd] = assemMat_Bnd_DG2(Mesh,@STIMA_BndPen_Tangent_DGLFE2,SIGMA2);
X = sparse([J2;J3], ...
           [I2;I3], ...
           [1/alpha2*Jinn_t+1/alpha1*Jinn_n;0*Jbnd]);
err = err+u'*X*u;

% Handle boundary terms
for j = 1:nEdges

    if(Mesh.BdFlags(j) < 0)
        if(Mesh.Edge2Elem(j,1) > 0)
            Data.Element = Mesh.Edge2Elem(j,1);

```

```

        Data.ElemFlag = ElemFlag(Data.Element);
        Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
        Data.EdgeLoc = Mesh.EdgeLoc(j,1);
        Data.Match = Mesh.EdgeOrient(j,1);
    else
        Data.Element = Mesh.Edge2Elem(j,2);
        Data.ElemFlag = ElemFlag(Data.Element);
        Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
        Data.EdgeLoc = Mesh.EdgeLoc(j,2);
        Data.Match = Mesh.EdgeOrient(j,2);
    end

    Normal = MeshNormals(j,:);
    QuadRule_Edge = gauleg(0,1,15);

    P0 = Mesh.Coordinates(Mesh.Edges(j,1),:);
    P1 = Mesh.Coordinates(Mesh.Edges(j,2),:);

    bK = Data.Vertices;
    BK = [bK(2,:)-bK(1,:); ...
          bK(3,:)-bK(1,:)];

    idx = 6*(Data.Element-1)+[1 2 3 4 5 6];

    nPts_Edge = size(QuadRule_Edge.w,1);
    x = ones(nPts_Edge,1)*P0 + QuadRule_Edge.x*(P1-P0);
    x_Ref = (x-ones(nPts_Edge,1)*bK(1,:))*inv(BK);
    N = shap_DGLFE(x_Ref);
    u_FE1 = u(idx(1))*N(:,1) + u(idx(3))*N(:,2) + u(idx(5))*N(:,3);
    u_FE2 = u(idx(2))*N(:,1) + u(idx(4))*N(:,2) + u(idx(6))*N(:,3);
    u_EX1 = UHandle1(x,ElemFlag(i),varargin{:});
    u_EX2 = UHandle2(x,ElemFlag(i),varargin{:});

    % Add jump terms to error
    % err = err + eta*abs(u'*B*u);
    err = err + sum(QuadRule_Edge.w.*abs(Normal(2)*(u_EX1-u_FE1)-Normal(1)*(u_EX2-u_FE2))
        .^2);

    end
end
% Calculate actual error: square root of above sum
err = sqrt(err);

return

```

### EnergyErr\_MIXDG.m

```

function err = EnergyErr_MIXDG(Mesh,u,QuadRule,UHandle1,UHandle2,Curl_UHandle,Div_UHandle
,SIGMA1,SIGMA2,alpha1,alpha2,varargin)
%EnergyErr_MIXDG DG energy-norm of discretization error of the discontinuous galerkin
solution
%
% ERR = ENERGYERR_MIXDG(Mesh,u,QuadRule,UHANDLE1,UHANDLE2,Curl_UHANDLE,Div_UHANDLE,
SIGMA1,SIGMA2,alpha1,alpha2,varargin)
% computes the DG energy-norm of discretization error of the
% discontinuous galerkin solution given by the vector U
% compared to the exact solution given by the
% function handles UHANDLE1 and UHANDLE2.
%
% The struct MESH should contain at least the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the
% mesh.
% ELEMDATA N-by-1 structure array containing at least the fields:
% NDOFS The number of degrees of freedom on the corresponding
% element.
% DIR A NDOFS-by-2 matrix containing the propagation
% directions of the plane wave basis functions in

```

```

%
%           its rows.
%
% QUADRULE is a struct, which specifies the Gauss quadrature that is used
% to do the integration:
% W Weights of the Gauss quadrature.
% X Abscissae of the Gauss quadrature.
%
% UHANDLE1 is a function handle for the first component of the exact solution and
% UHANDLE2 is a
% function handle for the second component of the exact solution.
%
% Curl_UHANDLE and Div_UHANDLE are function handles for the curl and
% divergence of the exact solution, respectively.
%
% SIGMA1 and SIGMA2 are function handles for the stabilization term in
% the normal and tangential jump, respectively.

% Intialize constants
nPts = size(QuadRule.w,1);
nElements = size(Mesh.Elements,1);
nCoordinates = size(Mesh.Coordinates,1);
nEdges = size(Mesh.Edges,1);
h = get_MeshWidth(Mesh);

% Compute volume terms of discretization error
if (isfield(Mesh,'ElemFlag')),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements,1);
end

% Precompute shape function values at the quadrature points
N = shap_DGLFE(QuadRule.x);

err = 0;
for i = 1:nElements

    % Extract vertex and basis function numbers
    vidx = Mesh.Elements(i,:);
    idx = 6*(i-1)+[1 2 3 4 5 6];

    % Compute element mapping
    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
    det_BK = abs(det(BK));

    % Transform quadrature points
    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions
    u_EX1 = UHandle1(x,ElemFlag(i),varargin{:});
    u_EX2 = UHandle2(x,ElemFlag(i),varargin{:});
    u_FE1 = u(idx(1))*N(:,1) + u(idx(3))*N(:,2) + u(idx(5))*N(:,3);
    u_FE2 = u(idx(2))*N(:,1) + u(idx(4))*N(:,2) + u(idx(6))*N(:,3);
    dNC = -[ Mesh.Coordinates(vidx(3),:) - Mesh.Coordinates(vidx(2),:) ...
             Mesh.Coordinates(vidx(1),:) - Mesh.Coordinates(vidx(3),:) ...
             Mesh.Coordinates(vidx(2),:) - Mesh.Coordinates(vidx(1),:) ]/det_BK;
    dND = [ Mesh.Coordinates(vidx(2),2) - Mesh.Coordinates(vidx(3),2) ...
            Mesh.Coordinates(vidx(3),1) - Mesh.Coordinates(vidx(2),1) ...
            Mesh.Coordinates(vidx(3),2) - Mesh.Coordinates(vidx(1),2) ...
            Mesh.Coordinates(vidx(1),1) - Mesh.Coordinates(vidx(3),1) ...
            Mesh.Coordinates(vidx(1),2) - Mesh.Coordinates(vidx(2),2) ...
            Mesh.Coordinates(vidx(2),1) - Mesh.Coordinates(vidx(1),1) ]/det_BK;

    % Evaluate curl and divergence of solutions

```

```

Curl_u_FE = u(idx(1))*dNC(1) + u(idx(2))*dNC(2) + u(idx(3))*dNC(3) + u(idx(4))*dNC(4)
            + u(idx(5))*dNC(5) + u(idx(6))*dNC(6);
Curl_u_EX = Curl_UHandle(x, varargin{:});

Div_u_FE = u(idx(1))*dND(1) + u(idx(2))*dND(2) + u(idx(3))*dND(3) + u(idx(4))*dND(4)
            + u(idx(5))*dND(5) + u(idx(6))*dND(6);
Div_u_EX = Div_UHandle(x, varargin{:});

% Compute error on current element
err = err+sum(QuadRule.w.*abs(u_EX1-u_FE1).^2)*det_BK+sum(QuadRule.w.*abs(u_EX2-u_FE2)
            .^2)*det_BK+sum(QuadRule.w.*abs(Curl_u_EX-Curl_u_FE).^2)*det_BK+sum(h^2*QuadRule
            .w.*abs(Div_u_EX-Div_u_FE).^2)*det_BK;

end

% Handle interior jump terms
[I2,J2,Jinn_n] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_Normal_DGLFE2,SIGMA1);
[I2,J2,Jinn_t] = assemMat_Inn_DG2(Mesh,@STIMA_InnPen_Tangent_DGLFE2,SIGMA2);
[I3,J3,Jbnd] = assemMat_Bnd_DG2(Mesh,@STIMA_BndPen_Tangent_DGLFE2,SIGMA2);
X = sparse([J2;J3], ...
            [I2;I3], ...
            [1/alpha2*Jinn_t+1/alpha1*Jinn_n;0*Jbnd]);
err = err+u'*X*u;

% Handle boundary terms
for j = 1:nEdges
    if(Mesh.BdFlags(j) < 0)
        if(Mesh.Edge2Elem(j,1) > 0)
            Data.Element = Mesh.Edge2Elem(j,1);
            Data.ElemFlag = ElemFlag(Data.Element);
            Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
            Data.EdgeLoc = Mesh.EdgeLoc(j,1);
            Data.Match = Mesh.EdgeOrient(j,1);
        else
            Data.Element = Mesh.Edge2Elem(j,2);
            Data.ElemFlag = ElemFlag(Data.Element);
            Data.Vertices = Mesh.Coordinates(Mesh.Elements(Data.Element,:),:);
            Data.EdgeLoc = Mesh.EdgeLoc(j,2);
            Data.Match = Mesh.EdgeOrient(j,2);
        end

% Extract normal and quadrature rule
Normal = MeshNormals(j,:);
QuadRule_Edge = gauleg(0,1,15);

% Extract vertex and basis function numbers
P0 = Mesh.Coordinates(Mesh.Edges(j,1),:);
P1 = Mesh.Coordinates(Mesh.Edges(j,2),:);
idx = 6*(Data.Element-1)+[1 2 3 4 5 6];

% Compute element mapping
bK = Data.Vertices;
BK = [bK(2,:)-bK(1,:); ...
      bK(3,:)-bK(1,:)];

% Transform quadrature points
nPts_Edge = size(QuadRule_Edge.w,1);
x = ones(nPts_Edge,1)*P0 + QuadRule_Edge.x*(P1-P0);
x_Ref = (x-ones(nPts_Edge,1)*bK(1,:))*inv(BK);
N = shap_DGLFE(x_Ref);

% Evaluate solutions
u_FE1 = u(idx(1))*N(:,1) + u(idx(3))*N(:,2) + u(idx(5))*N(:,3);
u_FE2 = u(idx(2))*N(:,1) + u(idx(4))*N(:,2) + u(idx(6))*N(:,3);
u_EX1 = UHandle1(x, ElemFlag(i), varargin{:});
u_EX2 = UHandle2(x, ElemFlag(i), varargin{:});

% Compute error on current edge

```

```

err = err + sum(QuadRule.Edge.w.*abs(Normal(2)*(u_EX1-u_FE1)-Normal(1)*(u_EX2-u_FE2))
.^2);

end
end

% Calculate actual error: square root of above sum
err = sqrt(err);

return

```

### L2Err\_DGLFE2.m

```

function err = L2Err_DGLFE2(Mesh,u,QuadRule,FHandle1,FHandle2,varargin)
% L2ERR_DGLFE2 Discretization error measured in the L2 norm for vectorial discontinuous
% galerkin finite elements.
%
% ERR = L2ERR_DGLFE2(MESH,U,QUADRULE,FHANDLE1,FHANDLE2) computes the discretization
% error between the exact solution given by the function handle FHANDLE1
% FHANDLE2 and the finite element solution U on the struct MESH.
%
% The struct MESH should at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 matrix specifying the elements of the mesh.
%
% QUADRULE is a struct, which specifies the Gauss quadrature that is used
% to do the integration:
% W Weights of the Gauss quadrature.
% X Abscissae of the Gauss quadrature.
%
% ERR = L2ERR_DGLFE(MESH,U,QUADRULE,FHANDLE,FPARAM) also handles the
% variable length argument list FPARAM to the exact solution FHANDLE.

% Intialize constants

nPts = size(QuadRule.w,1);
nCoordinates = size(Mesh.Coordinates,1);
nElements = size(Mesh.Elements,1);
% Check for element flags

if(isfield(Mesh,'ElemFlag')),
    ElemFlag = Mesh.ElemFlag;
else
    ElemFlag = zeros(nElements,1);
end
% Precompute shape function values at the quadrature points

N = shap_DGLFE(QuadRule.x);

% Compute discretization error

err = 0;
for i = 1:nElements

    % Extract vertex and edge numbers

    vidx = Mesh.Elements(i,:);
    idx = 6*(i-1)+[1 2 3 4 5 6];

    % Compute element mapping

    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
    det_BK = abs(det(BK));

    % Transform quadrature points

    x = QuadRule.x*BK+ones(nPts,1)*bK;

```

```

% Evaluate solutions

u_EX1 = FHandle1(x, ElemFlag(i), varargin{:});
u_EX2 = FHandle2(x, ElemFlag(i), varargin{:});
u_FE1 = u(idx(1))*N(:,1) + u(idx(3))*N(:,2) + u(idx(5))*N(:,3);
u_FE2 = u(idx(2))*N(:,1) + u(idx(4))*N(:,2) + u(idx(6))*N(:,3);
% Compute error on current element

err = err+sum(QuadRule.w.*abs(u_EX1-u_FE1).^2)*det_BK+sum(QuadRule.w.*abs(u_EX2-u_FE2).^2)*det_BK;

end

err = sqrt(err);

return

```

### LOAD\_Curl\_Bnd\_Tangent\_DGLFE2.m

```

function Lloc = LOAD_Curl_Bnd_Tangent_DGLFE2(Edge, Normal, BdFlag, Data, QuadRule, s, SHandle,
...
        FHandle1, FHandle2, varargin)
% LOAD_CURL_BND_TANGENT_DGLFE2 Element load vector for tangential component of boundary
% load data.
%
% LLOC = LOAD_CURL_BND_TANGENT_DGLFE2(EDGE, NORMAL, BDFLAG, DATA, QUADRULE, S, SHANDLE,
% FHANDLE)
% computes the entries of the element load vector for the tangential component of the
% boundary load data.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The struct DATA contains the left or right hand side element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% The integer S can specifies wheter the diffusive fluxes are discretized
% in a symmetric or anti-symmetric way:
% -1 Antisymmetric discretization of diffusive fluxes
% +1 Symmetric discretization of diffusive fluxes
%
% QUADRULE is a struct, which specifies the Gauss quadrature that is used
% to do the integration:
% w Weights of the Gauss quadrature.
% x Abscissae of the Gauss quadrature.
%
% SHANDLE is a function pointer to the edge weight function.
%
% FHANDLE is a function pointer to the load data.
%
% LLOC = LOAD_CURL_BND_TANGENT_DGLFE2(EDGE, NORMAL, BDFLAG, DATA, QUADRULE, SHANDLE, FHANDLE,
...

```

```

% PARAM) also handles the variable length argumet list PARAM to the
% function pointers SHANDLE and FHANDLE.

% Initialize constants
nPts = size(QuadRule.x,1);

% Preallocate memory
Lloc = zeros(6,1);

% Compute value of jump weight function
P0 = Edge(1,:);
P1 = Edge(2,:);
sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions
shap = shap_DGLFE([QuadRule.x zeros(nPts,1)]);

bK = Data.Vertices(1,:);
BK = [Data.Vertices(2,:)-bK; ...
      Data.Vertices(3,:)-bK];
area = abs(det(BK));

N = zeros(nPts,3);
dN = zeros(1,6);
switch(Data.EdgeLoc)
case 1
    if(Data.Match == 1)
        N(:,1) = shap(:,3);
        N(:,2) = shap(:,1);
        N(:,3) = shap(:,2);
    else
        N(:,1) = shap(:,3);
        N(:,2) = shap(:,2);
        N(:,3) = shap(:,1);
    end
case 2
    if(Data.Match == 1)
        N(:,1) = shap(:,2);
        N(:,2) = shap(:,3);
        N(:,3) = shap(:,1);
    else
        N(:,1) = shap(:,1);
        N(:,2) = shap(:,3);
        N(:,3) = shap(:,2);
    end
case 3
    if(Data.Match == 1)
        N(:,1) = shap(:,1);
        N(:,2) = shap(:,2);
        N(:,3) = shap(:,3);
    else
        N(:,1) = shap(:,2);
        N(:,2) = shap(:,1);
        N(:,3) = shap(:,3);
    end
end
end

dN = -[Data.Vertices(3,:) - Data.Vertices(2,:) ...
      Data.Vertices(1,:) - Data.Vertices(3,:) ...
      Data.Vertices(2,:) - Data.Vertices(1,:) ]/(area);

% Compute element map
x = ones(nPts,1)*P0 + QuadRule.x*(P1-P0);

```

```

dS = norm(P1-P0);

% Compute function values
FVal1 = FHandle1(x, BdFlag, varargin{:});
FVal2 = FHandle2(x, BdFlag, varargin{:});

% Compute the tangential component
FVal = -Normal(2)*FVal1+Normal(1)*FVal2;

% Compute entries of element load vector
Lloc(1) = -Normal(2)*sigma*sum(QuadRule.w.*FVal.*N(:,1))*dS ...
+ s*dN(1)*sum(QuadRule.w.*FVal)*dS;
Lloc(2) = Normal(1)*sigma*sum(QuadRule.w.*FVal.*N(:,1))*dS ...
+ s*dN(2)*sum(QuadRule.w.*FVal)*dS;
Lloc(3) = -Normal(2)*sigma*sum(QuadRule.w.*FVal.*N(:,2))*dS ...
+ s*dN(3)*sum(QuadRule.w.*FVal)*dS;
Lloc(4) = Normal(1)*sigma*sum(QuadRule.w.*FVal.*N(:,2))*dS ...
+ s*dN(4)*sum(QuadRule.w.*FVal)*dS;
Lloc(5) = -Normal(2)*sigma*sum(QuadRule.w.*FVal.*N(:,3))*dS ...
+ s*dN(5)*sum(QuadRule.w.*FVal)*dS;
Lloc(6) = Normal(1)*sigma*sum(QuadRule.w.*FVal.*N(:,3))*dS ...
+ s*dN(6)*sum(QuadRule.w.*FVal)*dS;

return

```

### LOAD\_Du\_Bnd\_Normal\_DGLFE2.m

```

function Lloc = LOAD_Du_Bnd_Normal_DGLFE2(Edge, Normal, BdFlag, Data, QuadRule, s, SHandle, ...
    FHandle1, FHandle2, FHandle3, FHandle4, varargin)
% LOAD_DU_BND_NORMAL_DGLFE2 Element load vector for normal derivative boundary load data.
%
% LLOC = LOAD_DU_BND_NORMAL_DGLFE2(EDGE, NORMAL, BDFLAG, DATA, QUADRULE, S, SHANDLE, FHANDLE)
% computes the entries of the element load vector for the boundary load
% data.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The struct DATA contains the left or right hand side element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% The integer S can specifies wheter the diffusive fluxes are discretized
% in a symmetric or anti-symmetric way:
% +1 Antisymmetric discretization of diffusive fluxes
% -1 Symmetric discretization of diffusive fluxes
%
% QUADRULE is a struct, which specifies the Gauss quadrature that is used
% to do the integration:
% w Weights of the Gauss quadrature.
% x Abscissae of the Gauss quadrature.

```

```

%
% SHANDLE is a function pointer to the edge weight function.
%
% FHANDLE is a function pointer to the load data.
%
% LLOC = LOAD_BND_DGLFE(EDGE,NORMAL,BDFLAG,DATA,QUADRULE,SHANDLE,FHANDLE, ...
% PARAM) also handles the variable length argumet list PARAM to the
% function pointers SHANDLE and FHANDLE.

% Initialize constants
nPts = size(QuadRule.x,1);

% Preallocate memory
Lloc = zeros(6,1);

% Compute value of jump weight function

P0 = Edge(1,:);
P1 = Edge(2,:);
sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions

shap = shap_DGLFE([QuadRule.x zeros(nPts,1)]);

bK = Data.Vertices(1,:);
BK = [Data.Vertices(2,:)-bK; ...
      Data.Vertices(3,:)-bK];
area = abs(det(BK));

N = zeros(nPts,3);
dN = zeros(1,6);
switch(Data.EdgeLoc)
  case 1
    if(Data.Match == 1)
      N(:,1) = shap(:,3);
      N(:,2) = shap(:,1);
      N(:,3) = shap(:,2);
    else
      N(:,1) = shap(:,3);
      N(:,2) = shap(:,2);
      N(:,3) = shap(:,1);
    end
  case 2
    if(Data.Match == 1)
      N(:,1) = shap(:,2);
      N(:,2) = shap(:,3);
      N(:,3) = shap(:,1);
    else
      N(:,1) = shap(:,1);
      N(:,2) = shap(:,3);
      N(:,3) = shap(:,2);
    end
  case 3
    if(Data.Match == 1)
      N(:,1) = shap(:,1);
      N(:,2) = shap(:,2);
      N(:,3) = shap(:,3);
    else
      N(:,1) = shap(:,2);
      N(:,2) = shap(:,1);
      N(:,3) = shap(:,3);
    end
end
end

dN = [ Data.Vertices(3,:) - Data.Vertices(2,:) ...

```

```

        Data.Vertices(1,:) - Data.Vertices(3,:) ...
        Data.Vertices(2,:) - Data.Vertices(1,:) ]/(area);

% Compute element map

x = ones(nPts,1)*P0 + QuadRule.x*(P1-P0);
dS = norm(P1-P0);

% Compute function values

FVal1 = FHandle1(x, BdFlag, varargin{:});
FVal2 = FHandle2(x, BdFlag, varargin{:});
FVal3 = FHandle3(x, BdFlag, varargin{:});
FVal4 = FHandle4(x, BdFlag, varargin{:});

FVal = Normal(1)^2*FVal1+Normal(1)*Normal(2)*FVal2+Normal(1)*Normal(2)*FVal3+Normal(2)
^2*FVal4;

% Compute entries of element load vector

Lloc(1) = Normal(1)*sum(QuadRule.w.*FVal.*N(:,1))*dS;
Lloc(2) = Normal(2)*sum(QuadRule.w.*FVal.*N(:,1))*dS;
Lloc(3) = Normal(1)*sum(QuadRule.w.*FVal.*N(:,2))*dS;
Lloc(4) = Normal(2)*sum(QuadRule.w.*FVal.*N(:,2))*dS;
Lloc(5) = Normal(1)*sum(QuadRule.w.*FVal.*N(:,3))*dS;
Lloc(6) = Normal(2)*sum(QuadRule.w.*FVal.*N(:,3))*dS;

return

```

### LOAD\_Grad\_Bnd\_Tangent\_DGLFE2.m

```

function Lloc = LOAD_Grad_Bnd_Tangent_DGLFE2(Edge, Normal, BdFlag, Data, QuadRule, s, SHandle,
...
        FHandle1, FHandle2, varargin)
% LOAD_GRAD_BND_TANGENT_DGLFE2 Element load vector for tangential boundary load data.
%
% LLOC = LOAD_GRAD_BND_TANGENT_DGLFE2(EDGE, NORMAL, BDFLAG, DATA, QUADRULE, S, SHANDLE,
% FHANDLE)
% computes the entries of the element load vector for the boundary load
% data.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The struct DATA contains the left or right hand side element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
%

```

```

% The integer S can specifies wheter the diffusive fluxes are discretized
% in a symmetric or anti-symmetric way:
% +1 Antisymmetric discretization of diffusive fluxes
% -1 Symmetric discretization of diffusive fluxes
%
% QUADRULE is a struct, which specifies the Gauss quadrature that is used
% to do the integration:
% w Weights of the Gauss quadrature.
% x Abscissae of the Gauss quadrature.
%
% SHANDLE is a function pointer to the edge weight function.
%
% FHANDLE is a function pointer to the load data.
%
% LLOC = LOAD_BND_DGLFE(EDGE,NORMAL,BDFLAG,DATA,QUADRULE,SHANDLE,FHANDLE, ...
% PARAM) also handles the variable length argumet list PARAM to the
% function pointers SHANDLE and FHANDLE.

% Initialize constants
nPts = size(QuadRule.x,1);

% Preallocate memory
Lloc = zeros(6,1);

% Compute value of jump weight function
P0 = Edge(1,:);
P1 = Edge(2,:);
sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions
shap = shap_DGLFE([QuadRule.x zeros(nPts,1)]);
grad_shap = grad_shap_DGLFE([0 0]);

bK = Data.Vertices(1,:);
BK = [Data.Vertices(2,:)-bK; ...
      Data.Vertices(3,:)-bK];
inv_BK_t = transpose(inv(BK));

N = zeros(nPts,3);
dN = zeros(1,3);
switch(Data.EdgeLoc)
case 1
    if(Data.Match == 1)
        N(:,1) = shap(:,3);
        N(:,2) = shap(:,1);
        N(:,3) = shap(:,2);
    else
        N(:,1) = shap(:,3);
        N(:,2) = shap(:,2);
        N(:,3) = shap(:,1);
    end
case 2
    if(Data.Match == 1)
        N(:,1) = shap(:,2);
        N(:,2) = shap(:,3);
        N(:,3) = shap(:,1);
    else
        N(:,1) = shap(:,1);
        N(:,2) = shap(:,3);
        N(:,3) = shap(:,2);
    end
case 3

```

```

    if(Data.Match == 1)
        N(:,1) = shap(:,1);
        N(:,2) = shap(:,2);
        N(:,3) = shap(:,3);
    else
        N(:,1) = shap(:,2);
        N(:,2) = shap(:,1);
        N(:,3) = shap(:,3);
    end
end
end
dN(1) = sum((grad_shap(1:2)*inv_BK_t).*Normal,2);
dN(2) = sum((grad_shap(3:4)*inv_BK_t).*Normal,2);
dN(3) = sum((grad_shap(5:6)*inv_BK_t).*Normal,2);

% Compute element map

x = ones(nPts,1)*P0 + QuadRule.x*(P1-P0);
dS = norm(P1-P0);
% Compute function values

FVal1 = FHandle1(x,BdFlag,varargin{:});
FVal2 = FHandle2(x,BdFlag,varargin{:});

FVal3 = Normal(2)*FVal1-Normal(1)*FVal2;
FVal1 = Normal(2)*FVal3;
FVal2 = -Normal(1)*FVal3;

% Compute entries of element load vector

Lloc(1) = Normal(2)*sigma*sum(QuadRule.w.*FVal3.*N(:,1))*dS ...
    + s*dN(1)*sum(QuadRule.w.*FVal1)*dS;
Lloc(2) = -Normal(1)*sigma*sum(QuadRule.w.*FVal3.*N(:,1))*dS ...
    + s*dN(1)*sum(QuadRule.w.*FVal2)*dS;
Lloc(3) = Normal(2)*sigma*sum(QuadRule.w.*FVal3.*N(:,2))*dS ...
    + s*dN(2)*sum(QuadRule.w.*FVal1)*dS;
Lloc(4) = -Normal(1)*sigma*sum(QuadRule.w.*FVal3.*N(:,2))*dS ...
    + s*dN(2)*sum(QuadRule.w.*FVal2)*dS;
Lloc(5) = Normal(2)*sigma*sum(QuadRule.w.*FVal3.*N(:,3))*dS ...
    + s*dN(3)*sum(QuadRule.w.*FVal1)*dS;
Lloc(6) = -Normal(1)*sigma*sum(QuadRule.w.*FVal3.*N(:,3))*dS ...
    + s*dN(3)*sum(QuadRule.w.*FVal2)*dS;

return

```

### LOAD\_Vol\_DGLFE2.m

```

function Lloc = LOAD_Vol_DGLFE2(Vertices,ElemFlag,QuadRule,FHandle1,FHandle2,varargin)
% LOAD_VOL_DGLFE2 Element load vector for volume load data.
%
%   LLOC = LOAD_VOL_DGLFE2(EDGE,NORMAL,BDFLAG,DATA,QUADRULE,S,FHANDLE)
%   computes the entries of the element load vector for the boundary load
%   data.
%
%   VERTICES is 3-by-2 matrix whose rows contain the vertices of the
%   current element.
%
%   The integer ELEMFLAG denotes the element flag of the current element.
%
%   QUADRULE is a struct, which specifies the Gauss quadrature that is used
%   to do the integration:
%   w Weights of the Gauss quadrature.
%   x Abscissae of the Gauss quadrature.
%
%   FHANDLE is a function pointer to the load data.

% Initialize constants

```

```

nPts = size(QuadRule.x,1);

% Preallocate memory

Lloc = zeros(6,1);

% Compute values of shape functions

N = shap_DGLFE(QuadRule.x);

% Compute element mapping

bK = Vertices(1,:);
BK = [ Vertices(2,:)-bK; ...
      Vertices(3,:)-bK];
det_BK = abs(det(BK));

x = QuadRule.x*BK+ones(nPts,1)*bK;

% Compute function values

FVal1 = FHandle1(x,ElemFlag,varargin{:});
FVal2 = FHandle2(x,ElemFlag,varargin{:});
FVal = [FVal1 FVal2];

% Compute entries of element load vector

Lloc(1) = sum(QuadRule.w.*FVal(:,1).*N(:,1))*det_BK;
Lloc(2) = sum(QuadRule.w.*FVal(:,2).*N(:,1))*det_BK;
Lloc(3) = sum(QuadRule.w.*FVal(:,1).*N(:,2))*det_BK;
Lloc(4) = sum(QuadRule.w.*FVal(:,2).*N(:,2))*det_BK;
Lloc(5) = sum(QuadRule.w.*FVal(:,1).*N(:,3))*det_BK;
Lloc(6) = sum(QuadRule.w.*FVal(:,2).*N(:,3))*det_BK;

return

```

MASS\_Lump\_Inv\_LFE.m

```

function Aloc = MASS_Lump_Inv_LFE(Vertices,varargin)
% MASS_LUMPLFE element lump mass matrix.
%
% ALOC = MASS_LUMP_Inv_LFE(VERTICES) computes the inverse of element mass matrix
% using nodal finite elements.
%
% VERTICES is 3-by-2 matrix specifying the vertices of the current
% element in a row wise orientation.

% Compute the area of the element

BK = [ Vertices(2,:)-Vertices(1,:); Vertices(3,:)-Vertices(1,:) ];
det_BK = abs(det(BK));

% Compute local mass matrix

Aloc = 6/(det_BK)*eye(3);

return

```

plot\_rate4.m

```

% Run script for plotting the errors versus mesh size in a log-log scale
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% u1 sq %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

MW=[.3536 .1768 .0884 .0442 .0221];

load MIXDG11.mat L2Err

```

```

Err_L2_1=L2Err;
load WReg11.mat L2Err
Err_L2_3=L2Err;
load SReg11.mat L2Err
Err_L2_4=L2Err;
figure

    plot(MW,Err_L2_1,'-^',MW,Err_L2_3,'-o',MW,Err_L2_4,'-+');
    grid on
    set(gca,'XScale','log','YScale','log','XDir','reverse');
    p = polyfit(log(MW),log(abs(Err_L2_1)),1);
    add_Slope(gca,'South',p(1));
    p = polyfit(log(MW),log(abs(Err_L2_3)),1);
    add_Slope(gca,'North',p(1));
    p = polyfit(log(MW),log(abs(Err_L2_4)),1);
    add_Slope(gca,'West',p(1));
    ylabel('\bf\fontsize{14} Relative discretization error')
    xlabel('\bf\fontsize{14}h')
    title('\bfConvergence rate for relative discretization error on the square domain')
    legend('\bf\fontsize{10}MIXDG','\bf\fontsize{10}WRegWIF','\bf\fontsize{10}SRegLFE')
    % Generate .eps files

    print('-depsc','rate_ulsq.eps');
    !gv rate_ulsq.eps &

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% u2 ls %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
load MIXDG23.mat L2Err
Err_L2_1=L2Err;
load WReg23.mat L2Err
Err_L2_3=L2Err;
load SReg23.mat L2Err
Err_L2_4=L2Err;

```

```

figure

    plot(MW,Err_L2_1,'-^',MW,Err_L2_3,'-o',MW,Err_L2_4,'-+');
    grid on
    set(gca,'XScale','log','YScale','log','XDir','reverse');
    p = polyfit(log(MW),log(abs(Err_L2_1)),1);
    add_Slope(gca,'South',p(1));
    p = polyfit(log(MW),log(abs(Err_L2_3)),1);
    add_Slope(gca,'West',p(1));
    p = polyfit(log(MW),log(abs(Err_L2_4)),1);
    add_Slope(gca,'North',p(1));
    ylabel('\bf\fontsize{14} Relative discretization error')
    xlabel('\bf\fontsize{14}h')
    title('\bfConvergence rate for relative discretization error on the L-shaped domain')
    legend('\bf\fontsize{10}MIXDG','\bf\fontsize{10}WRegWIF','\bf\fontsize{10}SRegLFE')
    % Generate .eps files

    print('-depsc','rate_u2ls.eps');
    !gv rate_u2ls.eps &

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% u3 ls %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
load MIXDG24.mat L2Err
Err_L2_1=L2Err;
load WReg24.mat L2Err
Err_L2_3=L2Err;
load SReg24.mat L2Err
Err_L2_4=L2Err;

```

```

figure

    plot(MW,Err_L2_1,'-^',MW,Err_L2_3,'-o',MW,Err_L2_4,'-+');
    grid on
    set(gca,'XScale','log','YScale','log','XDir','reverse');
    p = polyfit(log(MW),log(abs(Err_L2_1)),1);
    add_Slope(gca,'South',p(1));

```

```

p = polyfit(log(MW),log(abs(Err_L2_3)),1);
add_Slope(gca,'West',p(1));
p = polyfit(log(MW),log(abs(Err_L2_4)),1);
add_Slope(gca,'NorthEast',p(1));
ylabel('\bf\fontsize{14} Relative discretization error')
xlabel('\bf\fontsize{14}h')
title('\bfConvergence rate for relative discretization error on the L-shaped domain')
legend('\bf\fontsize{10}MXDG', '\bf\fontsize{10}WRegWIF', '\bf\fontsize{10}SRegLFE')

% Generate .eps files

print('-depsc', 'rate_u3ls.eps');
!gv rate_u3ls.eps &

```

#### sing\_fcn.m

```

function fval = sing_fcn(x)
%SING_FCN calculate the values of the singular function and force it to be
%zeros at the origin where the function has value NaN.

fval = 0*ones(size(x,1),1);
for j = 1:size(x,1)
    if norm(x(j,:))>0
        fval(j) = 2/3*(x(j,1).^2+x(j,2).^2).^(-1/6);
    end
end

```

#### STIMA\_BndPen\_Tangent\_DGLFE2.m

```

function Jloc = STIMA_BndPen_Tangent_DGLFE2(Edge, Normal, BdFlag, Data, SHandle, varargin)
% STIMA_BNDPEN_TANGENT_DGLFE2 Element stiffness matrix for boundary tangential penalty
% term.
%
% JLOC = STIMA_BNDPEN_TANGENT_DGLFE2(EDGE, NORMAL, BDFLAG, DATA, SHANDLE) computes the
% entries of the element stiffness matrix for the boundary tangential penalty term for
% vector valued function.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The struct DATA contains the left or right hand side element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% SHANDLE is a function pointer to the edge weight function.
%
% JLOC = STIMA_BNDPEN_TANGENT_DGLFE2(EDGE, NORMAL, BDFLAG, LDATA, RDATA, SHANDLE, SPARAM)
% also handles the variable length argument list SPARAM to the function
% pointer SHANDLE.

% Preallocate memory

Jloc = zeros(6,6);
Jtmp = zeros(3,3);
% Compute jump weight

```

```

P0 = Edge(1,:);
P1 = Edge(2,:);

sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions

dS = norm(P1-P0);
tmp = [1/3 1/6 0; 1/6 1/3 0; 0 0 0];

switch(Data.EdgeLoc)
case 1
    if(Data.Match == 1)
        Perm = [3 1 2];
    else
        Perm = [3 2 1];
    end
case 2
    if(Data.Match == 1)
        Perm = [2 3 1];
    else
        Perm = [1 3 2];
    end
case 3
    if(Data.Match == 1)
        Perm = [1 2 3];
    else
        Perm = [2 1 3];
    end
end

Jtmp = sigma*tmp(Perm,Perm)*dS;

for j = 1:3
    for i = 1:3
        Jloc(2*i-1:2*i,2*j-1:2*j) = Jtmp(i,j);
    end
end

% Take only the tangential component

for i = 1:2:6
    Jloc(i,:) = -Normal(2)*Jloc(i,:);
    Jloc(i+1,:) = Normal(1)*Jloc(i+1,:);
    Jloc(:,i) = -Normal(2)*Jloc(:,i);
    Jloc(:,i+1) = Normal(1)*Jloc(:,i+1);
end

return

```

### STIMA\_Curl\_Bnd\_DGLFE2.m

```

function Aloc = STIMA_Curl_Bnd_DGLFE2(Edge,Normal,BdFlag,Data,s,varargin)
% STIMA_CURL_BND_DGLFE2 Element stiffness matrix for boundary terms.
%
% ALOC = STIMA_CURL_BND_DGLFE2(EDGE,NORMAL,BDFLAG,DATA,S) computes the entries
% of the element stiffness matrix for the boundary terms for vector valued function.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
%

```

```

% The structs DATA contains the left or right hand side element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.

% Preallocate memory

Aloc = zeros(6,6);

% Compute discrete curl and tangential component of shape function on the boundary
% element

dS = norm(Edge(2,:)-Edge(1,:));

bK = Data.Vertices(1,:);
BK = [Data.Vertices(2,:)-bK; ...
      Data.Vertices(3,:)-bK];
area = abs(det(BK));

IN = zeros(1,6);
dN = zeros(1,6);

dN = -[ Data.Vertices(3,:) - Data.Vertices(2,:) ...
       Data.Vertices(1,:) - Data.Vertices(3,:) ...
       Data.Vertices(2,:) - Data.Vertices(1,:) ]/(area);

switch(Data.EdgeLoc)
case 1
    IN(1) = 0;
    IN(2) = 0;
    IN(3) = -Normal(2)*dS/2;
    IN(4) = Normal(1)*dS/2;
    IN(5) = -Normal(2)*dS/2;
    IN(6) = Normal(1)*dS/2;
case 2
    IN(1) = -Normal(2)*dS/2;
    IN(2) = Normal(1)*dS/2;
    IN(3) = 0;
    IN(4) = 0;
    IN(5) = -Normal(2)*dS/2;
    IN(6) = Normal(1)*dS/2;
case 3
    IN(1) = -Normal(2)*dS/2;
    IN(2) = Normal(1)*dS/2;
    IN(3) = -Normal(2)*dS/2;
    IN(4) = Normal(1)*dS/2;
    IN(5) = 0;
    IN(6) = 0;
end

% Compute entries of element penalty matrix

for i = 1:6
    for j = 1:6
        Aloc(i,j) = dN(i)*IN(j)+s*IN(i)*dN(j);
    end
end

return

```

STIMA\_Curl\_Inn\_DGLFE2.m

```

function Aloc = STIMA_Curl_Inn_DGLFE2(Edge, Normal, BdFlag, LData, RData, s, varargin)
% STIMA_CURL_INN_DGLFE2 Element stiffness matrix for interior terms.
%
% ALOC = STIMA_CURL_INN_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,S) computes the
% entries of the element stiffness matrix for the interior terms for vector valued
% function.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The structs LDATA and RDATA contain the left and right hand side
% element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.

% Preallocate memory

Aloc = zeros(12,12);

% Compute discrete curl and tangential component of shape function
% on the left and right hand side element

dS = norm(Edge(2,:)-Edge(1,:));

LbK = LData.Vertices(1,:);
LBK = [LData.Vertices(2,:)-LbK; ...
      LData.Vertices(3,:)-LbK];
Larea = abs(det(LBK));

INL = zeros(1,6);
dNL = zeros(1,6);

dNL = -[ LData.Vertices(3,:) - LData.Vertices(2,:) ...
        LData.Vertices(1,:) - LData.Vertices(3,:) ...
        LData.Vertices(2,:) - LData.Vertices(1,:) ]/(Larea);

switch(LData.EdgeLoc)
case 1
    INL(1) = 0;
    INL(2) = 0;
    INL(3) = -Normal(2)*dS/2;
    INL(4) = Normal(1)*dS/2;
    INL(5) = -Normal(2)*dS/2;
    INL(6) = Normal(1)*dS/2;
case 2
    INL(1) = -Normal(2)*dS/2;
    INL(2) = Normal(1)*dS/2;
    INL(3) = 0;
    INL(4) = 0;
    INL(5) = -Normal(2)*dS/2;
    INL(6) = Normal(1)*dS/2;
case 3
    INL(1) = -Normal(2)*dS/2;
    INL(2) = Normal(1)*dS/2;
    INL(3) = -Normal(2)*dS/2;

```

```

    INL(4) = Normal(1)*dS/2;
    INL(5) = 0;
    INL(6) = 0;
end

RbK = RData.Vertices(1,:);
RBK = [RData.Vertices(2,:)-RbK; ...
       RData.Vertices(3,:)-RbK];
Rarea = abs(det(RBK));

INR = zeros(1,6);
dNR = zeros(1,6);

dNR = -[ RData.Vertices(3,:) - RData.Vertices(2,:) ...
        RData.Vertices(1,:) - RData.Vertices(3,:) ...
        RData.Vertices(2,:) - RData.Vertices(1,:) ]/(Rarea);

switch(RData.EdgeLoc)
case 1
    INR(1) = 0;
    INR(2) = 0;
    INR(3) = -Normal(2)*dS/2;
    INR(4) = Normal(1)*dS/2;
    INR(5) = -Normal(2)*dS/2;
    INR(6) = Normal(1)*dS/2;
case 2
    INR(1) = -Normal(2)*dS/2;
    INR(2) = Normal(1)*dS/2;
    INR(3) = 0;
    INR(4) = 0;
    INR(5) = -Normal(2)*dS/2;
    INR(6) = Normal(1)*dS/2;
case 3
    INR(1) = -Normal(2)*dS/2;
    INR(2) = Normal(1)*dS/2;
    INR(3) = -Normal(2)*dS/2;
    INR(4) = Normal(1)*dS/2;
    INR(5) = 0;
    INR(6) = 0;
end

% Compute entries of element penalty matrix

if(LData.Element < RData.Element)
    gamma = 1;
else
    gamma = -1;
end

for i = 1:6
    for j = 1:6
        Aloc(i,j) = gamma*(dNL(i)*INL(j)+s*INL(i)*dNL(j))/2;
        Aloc(i,6+j) = gamma*(-dNL(i)*INR(j)+s*INL(i)*dNR(j))/2;
        Aloc(6+i,j) = gamma*(dNR(i)*INL(j)-s*INR(i)*dNL(j))/2;
        Aloc(6+i,6+j) = gamma*(-dNR(i)*INR(j)-s*INR(i)*dNR(j))/2;
    end
end

end

return

```

#### STIMA\_CurlLFE2.m

```

function Aloc = STIMA_CurlLFE2(Vertices, varargin)
% STIMA_CURL_LFE2 element stiffness matrix.
%
% ALOC = STIMA_CURL_LFE2(VERTICES) computes the element stiffness matrix
% using nodal finite elements.
%

```

```

% VERTICES is 3-by-2 matrix specifying the vertices of the current
% element in a row wise orientation.

% Compute the area of the element

BK = [ Vertices(2,:) - Vertices(1,:); Vertices(3,:) - Vertices(1,:) ];
det_BK = abs(det(BK));

% Compute local mass matrix

K = [ Vertices(3,:) - Vertices(2,:) ...
      Vertices(1,:) - Vertices(3,:) ...
      Vertices(2,:) - Vertices(1,:) ];

Aloc = 1/(2*det_BK)*(K')*K;

return

```

### STIMA\_Div\_Inn\_DGLFE2.m

```

function Aloc = STIMA_Div_Inn_DGLFE2(Edge,Normal,BdFlag,LData,RData,s,varargin)
% STIMA_DIV_INN_DGLFE2 Element stiffness matrix for interior terms.
%
% ALOC = DIV_INN_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,S) computes the
% entries of the element stiffness matrix for the interior terms for vector valued
% function.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The structs LDATA and RDATA contain the left and right hand side
% element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% The integer S can specify whether the diffusive fluxes are discretized
% in a symmetric or anti-symmetric way:
% -1 Antisymmetric discretization of diffusive fluxes
% +1 Symmetric discretization of diffusive fluxes

% Preallocate memory

Aloc = zeros(12,12);

% Compute normal derivatives on the left and right hand side element

dS = norm(Edge(2,:)-Edge(1,:));

LbK = LData.Vertices(1,:);
LBK = [LData.Vertices(2,:)-LbK; ...
       LData.Vertices(3,:)-LbK];
Larea = abs(det(LBK));

INL = zeros(1,6);

```

```

dNL = zeros(1,6);

dNL = [ LData.Vertices(2,2) - LData.Vertices(3,2) ...
        LData.Vertices(3,1) - LData.Vertices(2,1) ...
        LData.Vertices(3,2) - LData.Vertices(1,2) ...
        LData.Vertices(1,1) - LData.Vertices(3,1) ...
        LData.Vertices(1,2) - LData.Vertices(2,2) ...
        LData.Vertices(2,1) - LData.Vertices(1,1) ]/(Larea);

switch(LData.EdgeLoc)
case 1
    INL(1) = 0;
    INL(2) = 0;
    INL(3) = Normal(1)*dS/2;
    INL(4) = Normal(2)*dS/2;
    INL(5) = Normal(1)*dS/2;
    INL(6) = Normal(2)*dS/2;
case 2
    INL(1) = Normal(1)*dS/2;
    INL(2) = Normal(2)*dS/2;
    INL(3) = 0;
    INL(4) = 0;
    INL(5) = Normal(1)*dS/2;
    INL(6) = Normal(2)*dS/2;
case 3
    INL(1) = Normal(1)*dS/2;
    INL(2) = Normal(2)*dS/2;
    INL(3) = Normal(1)*dS/2;
    INL(4) = Normal(2)*dS/2;
    INL(5) = 0;
    INL(6) = 0;
end

RbK = RData.Vertices(1,:);
RBK = [RData.Vertices(2,:)-RbK; ...
        RData.Vertices(3,:)-RbK];
Rarea = abs(det(RBK));

INR = zeros(1,6);
dNR = zeros(1,6);

dNR = [ RData.Vertices(2,2) - RData.Vertices(3,2) ...
        RData.Vertices(3,1) - RData.Vertices(2,1) ...
        RData.Vertices(3,2) - RData.Vertices(1,2) ...
        RData.Vertices(1,1) - RData.Vertices(3,1) ...
        RData.Vertices(1,2) - RData.Vertices(2,2) ...
        RData.Vertices(2,1) - RData.Vertices(1,1) ]/(Rarea);

switch(RData.EdgeLoc)
case 1
    INR(1) = 0;
    INR(2) = 0;
    INR(3) = Normal(1)*dS/2;
    INR(4) = Normal(2)*dS/2;
    INR(5) = Normal(1)*dS/2;
    INR(6) = Normal(2)*dS/2;
case 2
    INR(1) = Normal(1)*dS/2;
    INR(2) = Normal(2)*dS/2;
    INR(3) = 0;
    INR(4) = 0;
    INR(5) = Normal(1)*dS/2;
    INR(6) = Normal(2)*dS/2;
case 3
    INR(1) = Normal(1)*dS/2;
    INR(2) = Normal(2)*dS/2;
    INR(3) = Normal(1)*dS/2;
    INR(4) = Normal(2)*dS/2;

```

```

    INR(5) = 0;
    INR(6) = 0;
end

% Compute entries of element penalty matrix

if(LData.Element < RData.Element)
    gamma = 1;
else
    gamma = -1;
end

for i = 1:6
    for j = 1:6
        Aloc(i,j) = gamma*(dNL(i)*INL(j)+s*INL(i)*dNL(j))/2;
        Aloc(i,6+j) = gamma*(-dNL(i)*INR(j)+s*INL(i)*dNR(j))/2;
        Aloc(6+i,j) = gamma*(dNR(i)*INL(j)-s*INR(i)*dNL(j))/2;
        Aloc(6+i,6+j) = gamma*(-dNR(i)*INR(j)-s*INR(i)*dNR(j))/2;
    end
end

return

```

### STIMA\_Div\_LFE2.m

```

function Aloc = STIMA_Div_LFE2(Vertices, varargin)
% STIMA_DIV_LFE2 element stiffness matrix.
%
% ALOC = STIMA_DIV_LFE2(VERTICES,ELEMINFO,F_HANDLE,QUADRULE) computes the
% element stiffness matrix using nodal finite elements.
%
% VERTICES is 3-by-2 matrix specifying the vertices of the current element
% in a row wise orientation.

% Compute the area of the element

BK = [Vertices(2,:) - Vertices(1,:); Vertices(3,:) - Vertices(1,:)];
det_BK = abs(det(BK));

% Compute local mass matrix

L = [ Vertices(2,2) - Vertices(3,2) ...
      Vertices(3,1) - Vertices(2,1) ...
      Vertices(3,2) - Vertices(1,2) ...
      Vertices(1,1) - Vertices(3,1) ...
      Vertices(1,2) - Vertices(2,2) ...
      Vertices(2,1) - Vertices(1,1) ];

Aloc = 1/(2*det_BK)*L'*L;

return

```

### STIMA\_Grad\_Bnd\_Tangent\_DGLFE2.m

```

function Aloc = STIMA_Grad_Bnd_Tangent_DGLFE2(Edge, Normal, BdFlag, Data, s, varargin)
% STIMA_GRAD_BND_TANGENT_DGLFE2 Element stiffness matrix for boundary terms.
%
% ALOC = STIMA_GRAD_BND_TANGENT_DGLFE2(EDGE,NORMAL,BDFLAG,DATA,S) computes the entries
% of the element stiffness matrix for the boundary terms for vector valued function.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.

```

```

%
% The structs DATA contains the left or right hand side element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% The integer S can specifies wheter the diffusive fluxes are discretized
% in a symmetric or anti-symmetric way:
% -1 Antisymmetric discretization of diffusive fluxes
% +1 Symmetric discretization of diffusive fluxes

% Preallocate memory

Aloc = zeros(6,6);

% Compute normal derivatives and integrated shape functions

dS = norm(Edge(2,:)-Edge(1,:));

grad_shap = grad_shap_DGLFE([0 0]);

bK = Data.Vertices(1,:);
BK = [Data.Vertices(2,:)-bK; ...
      Data.Vertices(3,:)-bK];
inv_BK_t = transpose(inv(BK));

IN = zeros(1,6);
dN = zeros(1,6);
switch(Data.EdgeLoc)
case 1
    IN(1) = 0;
    IN(2) = 0;
    IN(3) = dS/2;
    IN(4) = dS/2;
    IN(5) = dS/2;
    IN(6) = dS/2;
case 2
    IN(1) = dS/2;
    IN(2) = dS/2;
    IN(3) = 0;
    IN(4) = 0;
    IN(5) = dS/2;
    IN(6) = dS/2;
case 3
    IN(1) = dS/2;
    IN(2) = dS/2;
    IN(3) = dS/2;
    IN(4) = dS/2;
    IN(5) = 0;
    IN(6) = 0;
end
dN(1) = sum((grad_shap(1:2)*inv_BK_t).*Normal,2);
dN(2) = sum((grad_shap(1:2)*inv_BK_t).*Normal,2);
dN(3) = sum((grad_shap(3:4)*inv_BK_t).*Normal,2);
dN(4) = sum((grad_shap(3:4)*inv_BK_t).*Normal,2);
dN(5) = sum((grad_shap(5:6)*inv_BK_t).*Normal,2);
dN(6) = sum((grad_shap(5:6)*inv_BK_t).*Normal,2);

% Compute entries of element penalty matrix
for i = 1:6
    for j = 1:6
        Aloc(i,j) = dN(i)*IN(j)+s*IN(i)*dN(j);
    end
end

```

```

    end
end
for i = 1:2:6
    Aloc(i,:) = -Normal(2)*Aloc(i,:);
    Aloc(i+1,:) = Normal(1)*Aloc(i+1,:);
    Aloc(:,i) = -Normal(2)*Aloc(:,i);
    Aloc(:,i+1) = Normal(1)*Aloc(:,i+1);
end
return

```

### STIMA\_Grad\_Inn\_DGLFE2.m

```

function Aloc = STIMA_Grad_Inn_DGLFE2(Edge,Normal,BdFlag,LData,RData,s,varargin)
% STIMA_GRAD_NN_DGLFE2 Element stiffness matrix for interior terms.
%
% ALOC = STIMA_GRAD_NN_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,S) computes the
% entries of the element stiffness matrix for the interior terms for vector valued
% function.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The structs LDATA and RDATA contain the left and right hand side
% element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% The integer S can specify whether the diffusive fluxes are discretized
% in a symmetric or anti-symmetric way:
% -1 Antisymmetric discretization of diffusive fluxes
% +1 Symmetric discretization of diffusive fluxes

% Preallocate memory
Aloc = zeros(12,12);

% Compute normal derivatives on the left and right hand side element
dS = norm(Edge(2,:)-Edge(1,:));

grad_shap = grad_shap_DGLFE([0 0]);

bK = LData.Vertices(1,:);
BK = [LData.Vertices(2,:)-bK; ...
      LData.Vertices(3,:)-bK];
inv_BK_t = transpose(inv(BK));

INL = zeros(1,3);
dNL = zeros(1,3);
switch(LData.EdgeLoc)
case 1
    INL(1) = 0;
    INL(2) = dS/2;
    INL(3) = dS/2;

```

```

case 2
    INL(1) = dS/2;
    INL(2) = 0;
    INL(3) = dS/2;
case 3
    INL(1) = dS/2;
    INL(2) = dS/2;
    INL(3) = 0;
end
dNL(1) = sum((grad_shap(1:2)*inv_BK_t).*Normal,2);
dNL(2) = sum((grad_shap(3:4)*inv_BK_t).*Normal,2);
dNL(3) = sum((grad_shap(5:6)*inv_BK_t).*Normal,2);

bK = RData.Vertices(1,:);
BK = [RData.Vertices(2,:)-bK; ...
      RData.Vertices(3,:)-bK];
inv_BK_t = transpose(inv(BK));

INR = zeros(1,3);
dNR = zeros(1,3);
switch(RData.EdgeLoc)
case 1
    INR(1) = 0;
    INR(2) = dS/2;
    INR(3) = dS/2;
case 2
    INR(1) = dS/2;
    INR(2) = 0;
    INR(3) = dS/2;
case 3
    INR(1) = dS/2;
    INR(2) = dS/2;
    INR(3) = 0;
end
dNR(1) = sum((grad_shap(1:2)*inv_BK_t).*Normal,2);
dNR(2) = sum((grad_shap(3:4)*inv_BK_t).*Normal,2);
dNR(3) = sum((grad_shap(5:6)*inv_BK_t).*Normal,2);

% Compute entries of element penalty matrix

if(LData.Element < RData.Element)
    gamma = 1;
else
    gamma = -1;
end

Aloc(1,1) = gamma*( dNL(1)*INL(1)+s*INL(1)*dNL(1))/2;
Aloc(1,3) = gamma*( dNL(1)*INL(2)+s*INL(1)*dNL(2))/2;
Aloc(1,5) = gamma*( dNL(1)*INL(3)+s*INL(1)*dNL(3))/2;

Aloc(1,7) = gamma*(-dNL(1)*INR(1)+s*INL(1)*dNR(1))/2;
Aloc(1,9) = gamma*(-dNL(1)*INR(2)+s*INL(1)*dNR(2))/2;
Aloc(1,11) = gamma*(-dNL(1)*INR(3)+s*INL(1)*dNR(3))/2;

Aloc(2,2) = gamma*( dNL(1)*INL(1)+s*INL(1)*dNL(1))/2;
Aloc(2,4) = gamma*( dNL(1)*INL(2)+s*INL(1)*dNL(2))/2;
Aloc(2,6) = gamma*( dNL(1)*INL(3)+s*INL(1)*dNL(3))/2;

Aloc(2,8) = gamma*(-dNL(1)*INR(1)+s*INL(1)*dNR(1))/2;
Aloc(2,10) = gamma*(-dNL(1)*INR(2)+s*INL(1)*dNR(2))/2;
Aloc(2,12) = gamma*(-dNL(1)*INR(3)+s*INL(1)*dNR(3))/2;

Aloc(3,1) = gamma*( dNL(2)*INL(1)+s*INL(2)*dNL(1))/2;
Aloc(3,3) = gamma*( dNL(2)*INL(2)+s*INL(2)*dNL(2))/2;
Aloc(3,5) = gamma*( dNL(2)*INL(3)+s*INL(2)*dNL(3))/2;

Aloc(3,7) = gamma*(-dNL(2)*INR(1)+s*INL(2)*dNR(1))/2;
Aloc(3,9) = gamma*(-dNL(2)*INR(2)+s*INL(2)*dNR(2))/2;

```

$Aloc(3,11) = \text{gamma} * (-dNL(2) * INR(3) + s * INL(2) * dNR(3)) / 2;$   
 $Aloc(4,2) = \text{gamma} * (dNL(2) * INL(1) + s * INL(2) * dNL(1)) / 2;$   
 $Aloc(4,4) = \text{gamma} * (dNL(2) * INL(2) + s * INL(2) * dNL(2)) / 2;$   
 $Aloc(4,6) = \text{gamma} * (dNL(2) * INL(3) + s * INL(2) * dNL(3)) / 2;$   
 $Aloc(4,8) = \text{gamma} * (-dNL(2) * INR(1) + s * INL(2) * dNR(1)) / 2;$   
 $Aloc(4,10) = \text{gamma} * (-dNL(2) * INR(2) + s * INL(2) * dNR(2)) / 2;$   
 $Aloc(4,12) = \text{gamma} * (-dNL(2) * INR(3) + s * INL(2) * dNR(3)) / 2;$   
 $Aloc(5,1) = \text{gamma} * (dNL(3) * INL(1) + s * INL(3) * dNL(1)) / 2;$   
 $Aloc(5,3) = \text{gamma} * (dNL(3) * INL(2) + s * INL(3) * dNL(2)) / 2;$   
 $Aloc(5,5) = \text{gamma} * (dNL(3) * INL(3) + s * INL(3) * dNL(3)) / 2;$   
 $Aloc(5,7) = \text{gamma} * (-dNL(3) * INR(1) + s * INL(3) * dNR(1)) / 2;$   
 $Aloc(5,9) = \text{gamma} * (-dNL(3) * INR(2) + s * INL(3) * dNR(2)) / 2;$   
 $Aloc(5,11) = \text{gamma} * (-dNL(3) * INR(3) + s * INL(3) * dNR(3)) / 2;$   
 $Aloc(6,2) = \text{gamma} * (dNL(3) * INL(1) + s * INL(3) * dNL(1)) / 2;$   
 $Aloc(6,4) = \text{gamma} * (dNL(3) * INL(2) + s * INL(3) * dNL(2)) / 2;$   
 $Aloc(6,6) = \text{gamma} * (dNL(3) * INL(3) + s * INL(3) * dNL(3)) / 2;$   
 $Aloc(6,8) = \text{gamma} * (-dNL(3) * INR(1) + s * INL(3) * dNR(1)) / 2;$   
 $Aloc(6,10) = \text{gamma} * (-dNL(3) * INR(2) + s * INL(3) * dNR(2)) / 2;$   
 $Aloc(6,12) = \text{gamma} * (-dNL(3) * INR(3) + s * INL(3) * dNR(3)) / 2;$   
 $Aloc(7,1) = \text{gamma} * (dNR(1) * INL(1) - s * INR(1) * dNL(1)) / 2;$   
 $Aloc(7,3) = \text{gamma} * (dNR(1) * INL(2) - s * INR(1) * dNL(2)) / 2;$   
 $Aloc(7,5) = \text{gamma} * (dNR(1) * INL(3) - s * INR(1) * dNL(3)) / 2;$   
 $Aloc(7,7) = \text{gamma} * (-dNR(1) * INR(1) - s * INR(1) * dNR(1)) / 2;$   
 $Aloc(7,9) = \text{gamma} * (-dNR(1) * INR(2) - s * INR(1) * dNR(2)) / 2;$   
 $Aloc(7,11) = \text{gamma} * (-dNR(1) * INR(3) - s * INR(1) * dNR(3)) / 2;$   
 $Aloc(8,2) = \text{gamma} * (dNR(1) * INL(1) - s * INR(1) * dNL(1)) / 2;$   
 $Aloc(8,4) = \text{gamma} * (dNR(1) * INL(2) - s * INR(1) * dNL(2)) / 2;$   
 $Aloc(8,6) = \text{gamma} * (dNR(1) * INL(3) - s * INR(1) * dNL(3)) / 2;$   
 $Aloc(8,8) = \text{gamma} * (-dNR(1) * INR(1) - s * INR(1) * dNR(1)) / 2;$   
 $Aloc(8,10) = \text{gamma} * (-dNR(1) * INR(2) - s * INR(1) * dNR(2)) / 2;$   
 $Aloc(8,12) = \text{gamma} * (-dNR(1) * INR(3) - s * INR(1) * dNR(3)) / 2;$   
 $Aloc(9,1) = \text{gamma} * (dNR(2) * INL(1) - s * INR(2) * dNL(1)) / 2;$   
 $Aloc(9,3) = \text{gamma} * (dNR(2) * INL(2) - s * INR(2) * dNL(2)) / 2;$   
 $Aloc(9,5) = \text{gamma} * (dNR(2) * INL(3) - s * INR(2) * dNL(3)) / 2;$   
 $Aloc(9,7) = \text{gamma} * (-dNR(2) * INR(1) - s * INR(2) * dNR(1)) / 2;$   
 $Aloc(9,9) = \text{gamma} * (-dNR(2) * INR(2) - s * INR(2) * dNR(2)) / 2;$   
 $Aloc(9,11) = \text{gamma} * (-dNR(2) * INR(3) - s * INR(2) * dNR(3)) / 2;$   
 $Aloc(10,2) = \text{gamma} * (dNR(2) * INL(1) - s * INR(2) * dNL(1)) / 2;$   
 $Aloc(10,4) = \text{gamma} * (dNR(2) * INL(2) - s * INR(2) * dNL(2)) / 2;$   
 $Aloc(10,6) = \text{gamma} * (dNR(2) * INL(3) - s * INR(2) * dNL(3)) / 2;$   
 $Aloc(10,8) = \text{gamma} * (-dNR(2) * INR(1) - s * INR(2) * dNR(1)) / 2;$   
 $Aloc(10,10) = \text{gamma} * (-dNR(2) * INR(2) - s * INR(2) * dNR(2)) / 2;$   
 $Aloc(10,12) = \text{gamma} * (-dNR(2) * INR(3) - s * INR(2) * dNR(3)) / 2;$   
 $Aloc(11,1) = \text{gamma} * (dNR(3) * INL(1) - s * INR(3) * dNL(1)) / 2;$   
 $Aloc(11,3) = \text{gamma} * (dNR(3) * INL(2) - s * INR(3) * dNL(2)) / 2;$   
 $Aloc(11,5) = \text{gamma} * (dNR(3) * INL(3) - s * INR(3) * dNL(3)) / 2;$   
 $Aloc(11,7) = \text{gamma} * (-dNR(3) * INR(1) - s * INR(3) * dNR(1)) / 2;$   
 $Aloc(11,9) = \text{gamma} * (-dNR(3) * INR(2) - s * INR(3) * dNR(2)) / 2;$   
 $Aloc(11,11) = \text{gamma} * (-dNR(3) * INR(3) - s * INR(3) * dNR(3)) / 2;$   
 $Aloc(12,2) = \text{gamma} * (dNR(3) * INL(1) - s * INR(3) * dNL(1)) / 2;$   
 $Aloc(12,4) = \text{gamma} * (dNR(3) * INL(2) - s * INR(3) * dNL(2)) / 2;$

```

Aloc(12,6) = gamma*( dNR(3)*INL(3)-s*INR(3)*dNL(3))/2;
Aloc(12,8) = gamma*(-dNR(3)*INR(1)-s*INR(3)*dNR(1))/2;
Aloc(12,10) = gamma*(-dNR(3)*INR(2)-s*INR(3)*dNR(2))/2;
Aloc(12,12) = gamma*(-dNR(3)*INR(3)-s*INR(3)*dNR(3))/2;
return

```

### STIMA\_Grad\_MIXDG.m

```

function Aloc = STIMA_Grad_MIXDG(Vertices, ElemInfo, varargin)
% STIMA_GRAD_MIXDG Element stiffness matrix for the gradient of multiplier p.
%
% ALOC = STIMA_GRAD_MIXDG(VERTICES,ELEMINFO) computes the element stiffness
% matrix for the gradient of multiplier p coupling discontinuous and
% conforming nodal elements
%
% VERTICES is a 3-by-2 matrix specifying the vertices of the current
% element in a row wise orientation.
%
% ELEMINFO is an integer parameter which is used to specify additional
% element information on each element.

% Preallocate memory
QuadRule=P3O3();
Aloc = zeros(6,3);

% Compute element mapping

P1 = Vertices(1,:);
P2 = Vertices(2,:);
P3 = Vertices(3,:);
bK = P1;
BK = [P2-bK;P3-bK];
det_BK = abs(det(BK));

N = shap_DGLFE(QuadRule.x);

% Compute the discrete gradient

gradient(1,1:2) = [P2(2)-P3(2) -P2(1)+P3(1)];
gradient(2,1:2) = [P3(2)-P1(2) -P3(1)+P1(1)];
gradient(3,1:2) = [P1(2)-P2(2) -P1(1)+P2(1)];

% Compute element stiffness matrix

for i = 1:3
    for j = 1:3
        Aloc(2*i-1,j) = sum(QuadRule.w.*N(:,i))*gradient(j,1);
        Aloc(2*i,j) = sum(QuadRule.w.*N(:,i))*gradient(j,2);
    end
end
return

```

### STIMA\_InnPen\_DGLFE2.m

```

function Jloc = STIMA_InnPen_DGLFE2(Edge, Normal, BdFlag, LData, RData, SHandle, varargin)
% STIMA_INNPEN_DGLFE2 Element stiffness matrix for interior penalty term.
%
% JLOC = STIMA_INNPEN_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,SHANDLE)
% computes the entries of the element stiffness matrix for the interior
% penalty term.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 marix which contains the unit normal with respect to
% the current edge EDGE.

```

```

%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The structs LDATA and RDATA contain the left and right hand side
% element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% SHANDLE is a function pointer to the edge weight function.
%
% JLOC = STIMA_INNPEN_DGLFE(EDGE, NORMAL, BDFLAG, LDATA, RDATA, SHANDLE, SPARAM)
% also handles the variable length argument list SPARAM to the function
% pointer SHANDLE.

% Preallocate memory

Jloc = zeros(12,12);
Jtmp = zeros(6,6);

% Compute jump weight

P0 = Edge(1,:);
P1 = Edge(2,:);

sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions

dS = norm(P1-P0);
tmp = [1/3 1/6 0; 1/6 1/3 0; 0 0 0];

switch(LData.EdgeLoc)
case 1
    if(LData.Match == 1)
        LPerm = [3 1 2];
    else
        LPerm = [3 2 1];
    end
case 2
    if(LData.Match == 1)
        LPerm = [2 3 1];
    else
        LPerm = [1 3 2];
    end
case 3
    if(LData.Match == 1)
        LPerm = [1 2 3];
    else
        LPerm = [2 1 3];
    end
end

switch(RData.EdgeLoc)
case 1
    if(RData.Match == 1)
        RPerm = [3 1 2];
    else
        RPerm = [3 2 1];
    end
case 2

```

```

    if(RData.Match == 1)
        RPerm = [2 3 1];
    else
        RPerm = [1 3 2];
    end
case 3
    if(RData.Match == 1)
        RPerm = [1 2 3];
    else
        RPerm = [2 1 3];
    end
end
Jtmp = sigma*[ tmp(LPerm,LPerm) -tmp(LPerm,RPerm);
               -tmp(RPerm,LPerm) tmp(RPerm,RPerm)]*dS;

for j = 1:6
    for i = 1:6
        Jloc(2*i-1,2*j-1) = Jtmp(i,j);
        Jloc(2*i,2*j) = Jtmp(i,j);
    end
end
return

```

### STIMA\_InnPen\_Normal\_DGLFE2.m

```

function Jloc = STIMA_InnPen_Normal_DGLFE2(Edge,Normal,BdFlag,LData,RData,SHandle,
    varargin)
% STIMA_INNPEN_NORMAL_DGLFE2 Element stiffness matrix for interior normal penalty term.
%
% JLOC = STIMA_INNPEN_NORMAL_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,SHANDLE)
% computes the entries of the element stiffness matrix for the interior
% normal penalty term.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The structs LDATA and RDATA contain the left and right hand side
% element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% SHANDLE is a function pointer to the edge weight function.
%
% JLOC = STIMA_INNPEN_NORMAL_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,SHANDLE,SPARAM)
% also handles the variable length argument list SPARAM to the function
% pointer SHANDLE.
%
% Preallocate memory
Jloc = zeros(12,12);
Jtmp = zeros(6,6);
% Compute jump weight

```

```

P0 = Edge(1,:);
P1 = Edge(2,:);

sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions
dS = norm(P1-P0);
tmp = [1/3 1/6 0; 1/6 1/3 0; 0 0 0];

switch(LData.EdgeLoc)
case 1
    if(LData.Match == 1)
        LPerm = [3 1 2];
    else
        LPerm = [3 2 1];
    end
case 2
    if(LData.Match == 1)
        LPerm = [2 3 1];
    else
        LPerm = [1 3 2];
    end
case 3
    if(LData.Match == 1)
        LPerm = [1 2 3];
    else
        LPerm = [2 1 3];
    end
end

switch(RData.EdgeLoc)
case 1
    if(RData.Match == 1)
        RPerm = [3 1 2];
    else
        RPerm = [3 2 1];
    end
case 2
    if(RData.Match == 1)
        RPerm = [2 3 1];
    else
        RPerm = [1 3 2];
    end
case 3
    if(RData.Match == 1)
        RPerm = [1 2 3];
    else
        RPerm = [2 1 3];
    end
end

Jtmp = sigma*[ tmp(LPerm,LPerm) -tmp(LPerm,RPerm);
               -tmp(RPerm,LPerm) tmp(RPerm,RPerm)]*dS;

for j = 1:6
    for i = 1:6
        Jloc(2*i-1:2*i,2*j-1:2*j) = Jtmp(i,j);
    end
end

% Take only the normal component
for i = 1:2:12
    Jloc(i,:) = Normal(1)*Jloc(i,:);
    Jloc(i+1,:) = Normal(2)*Jloc(i+1,:);
    Jloc(:,i) = Normal(1)*Jloc(:,i);
end

```

```

        Jloc(:,i+1) = Normal(2)*Jloc(:,i+1);
    end

    if(LData.Element < RData.Element)
        Jloc=Jloc;
    end

return

```

### STIMA\_InnPen\_Tangent\_DGLFE2.m

```

function Jloc = STIMA_InnPen_Tangent_DGLFE2(Edge,Normal,BdFlag,LData,RData,SHandle,
    varargin)
% STIMA_INNPEN_TANGENT_DGLFE2 Element stiffness matrix for interior tangential penalty
% term.
%
% JLOC = STIMA_INNPEN_TANGENT_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,SHANDLE)
% computes the entries of the element stiffness matrix for the interior
% tangential penalty term.
%
% EDGE is 2-by-2 matrix whose rows contain the start and end node of the
% current edge.
%
% NORMAL is 1-by-2 matrix which contains the unit normal with respect to
% the current edge EDGE.
%
% The integer BDFLAG denotes the boundary flag of the current edge. Note
% that for interior edges only values larger than are allowed.
%
% The structs LDATA and RDATA contain the left and right hand side
% element data:
% ELEMENT Integer specifying the neighbouring element.
% ELEMFLAG Integer specifying the element flag of the neighbouring
% element or zero.
% VERTICES 3-by-2 matrix specifying the vertices of the neighbouring
% element.
% EDGELOC Integer specifying the local edge number on the neighbouring
% element.
% MATCH Integer specifying the relative orientation of the edge with
% respect to the orientation of the neighbouring element.
%
% SHANDLE is a function pointer to the edge weight function.
%
% JLOC = STIMA_INNPEN_TANGENT_DGLFE2(EDGE,NORMAL,BDFLAG,LDATA,RDATA,SHANDLE,SPARAM)
% also handles the variable length argument list SPARAM to the function
% pointer SHANDLE.

% Preallocate memory

Jloc = zeros(12,12);
Jtmp = zeros(6,6);

% Compute jump weight

P0 = Edge(1,:);
P1 = Edge(2,:);

sigma = SHandle(P0,P1,varargin{:});

% Compute values of shape functions

dS = norm(P1-P0);
tmp = [1/3 1/6 0; 1/6 1/3 0; 0 0 0];

switch(LData.EdgeLoc)
    case 1
        if(LData.Match == 1)
            LPerm = [3 1 2];

```

```

    else
        LPerm = [3 2 1];
    end
case 2
    if(LData.Match == 1)
        LPerm = [2 3 1];
    else
        LPerm = [1 3 2];
    end
case 3
    if(LData.Match == 1)
        LPerm = [1 2 3];
    else
        LPerm = [2 1 3];
    end
end

switch(RData.EdgeLoc)
case 1
    if(RData.Match == 1)
        RPerm = [3 1 2];
    else
        RPerm = [3 2 1];
    end
case 2
    if(RData.Match == 1)
        RPerm = [2 3 1];
    else
        RPerm = [1 3 2];
    end
case 3
    if(RData.Match == 1)
        RPerm = [1 2 3];
    else
        RPerm = [2 1 3];
    end
end

Jtmp = sigma*[ tmp(LPerm,LPerm) -tmp(LPerm,RPerm);
               -tmp(RPerm,LPerm) tmp(RPerm,RPerm) ]*dS;

for j = 1:6
    for i = 1:6
        Jloc(2*i-1:2*i,2*j-1:2*j) = Jtmp(i,j);
    end
end

% Take only the tangential component
for i = 1:2:12
    Jloc(i,:) = -Normal(2)*Jloc(i,:);
    Jloc(i+1,:) = Normal(1)*Jloc(i+1,:);
    Jloc(:,i) = -Normal(2)*Jloc(:,i);
    Jloc(:,i+1) = Normal(1)*Jloc(:,i+1);
end

if(LData.Element < RData.Element)
    Jloc=Jloc;
end

return

```

### STIMA\_Lapl\_Vol\_DGLFE2.m

```

function Aloc = STIMA_Lapl_Vol_DGLFE2(Vertices, varargin)
% STIMA_LAPL_VOL_DGLFE2 Element stiffness matrix for the vectorial Laplacian.
%
% ALOC = STIMA_LAPL_VOL_DGLFE2(VERTICES) computes the element stiffness

```

```

% matrix for the Laplacian using discontinuous vector valued linear Lagrangian finite
% elements.
%
% VERTICES is 3-by-2 matrix specifying the vertices of the current element
% in a row wise orientation.

% Preallocate memory

Aloc = zeros(6,6);

% Analytic computation of matrix entries using barycentric coordinates

a = norm(Vertices(3,:)-Vertices(2,:));
b = norm(Vertices(3,:)-Vertices(1,:));
c = norm(Vertices(2,:)-Vertices(1,:));
s = (a+b+c)/2;
r = sqrt((s-a)*(s-b)*(s-c)/s);
cot_1 = cot(2*atan(r/(s-a)));
cot_2 = cot(2*atan(r/(s-b)));
cot_3 = cot(2*atan(r/(s-c)));

Aloc(1,1) = 1/2*(cot_3+cot_2);
Aloc(1,3) = 1/2*(-cot_3);
Aloc(1,5) = 1/2*(-cot_2);
Aloc(2,2) = 1/2*(cot_3+cot_2);
Aloc(2,4) = 1/2*(-cot_3);
Aloc(2,6) = 1/2*(-cot_2);
Aloc(3,3) = 1/2*(cot_3+cot_1);
Aloc(3,5) = 1/2*(-cot_1);
Aloc(4,4) = 1/2*(cot_3+cot_1);
Aloc(4,6) = 1/2*(-cot_1);
Aloc(5,5) = 1/2*(cot_2+cot_1);
Aloc(6,6) = 1/2*(cot_2+cot_1);

% Update lower triangular part

Aloc = Aloc+Aloc'-diag(diag(Aloc));

return

```