ETH ZÜRICH

Spurious Solutions for transient Maxwell equations in 2D

Semester Work

presented by:
Andrés Rosero
Born 8 November 1978
citizen of Ambato, Ecuador

Supervision:
Prof. Ralf Hiptmair

April 8, 2011

Contents

Chapter I

Abbreviations

Ch.	Chapter
FEM	Finite elements methods
PDE	Partial differential equation
p.	Page
ODE	Ordinary differential equation
Th.	Theorem
rhs.	Right hand side
w.r.t.	With respect to

Chapter 1

Introduction

Electromagnetic phenomena is well described by the Maxwell Equations

$$\operatorname{div} \mathbf{D} = \rho$$

$$\operatorname{div} \mathbf{B} = 0$$

$$\operatorname{curl} \mathbf{E} = -\tfrac{d}{dt}\mathbf{B} \tag{1.1}$$

$$\operatorname{curl} \mathbf{H} = \tfrac{d}{dt}\mathbf{D} + \mathbf{J} \tag{1.2}$$

and $\mathbf{B} = \mu\mathbf{H}$, $\mathbf{D} = \epsilon\mathbf{E}$, where $\mathbf{E}$ and $\mathbf{B}$ are the electric and the magnetic field, ρ is the total charge density and $\mathbf{J}$ the total current density.
The first two equations describe how the sources generate the fields, whereas the last two describe the time evolution of the fields.

Numerical solutions to these equations can be found using FEM, however the inappropriate application of FEM could generate spurious solutions. These solutions are not physical, i.e. can not be observed in reality [1, p.323], [2].

This work illustrates and compares spurious solutions obtained using Nodal elements with the right solutions obtained using edge elements for transient Maxwell's equations in a 2D-domain.

In Chapter 2, we present the Cauchy problem and state its variational formulation. We also describe the application of FEM for nodal and edge elements furthermore we give some stability conditions to ensure the right choice for the time-step.

Chapter 3 illustrates the MATLAB implementation with the corresponding comments and discusses some difference between the two discretizations.
Finally in Chapter 4 we describe numerical experiments carried on on a

square and on an L-shaped domain and compare the results for the nodal and edge elements andfor different mesh-sizes.

The structure of this work is based on [3], however there are some parts that were quoted literally from [3] as we were not able to find an equivalent formulation.

Chapter 2

Theoretical aspects

Equation (1.1) is called Faraday law and describes how changes of the Magnetic field, induce an electric field. Equation (1.2) is called Ampere's law and describes how current flux and changes in the electrical field generate a magnetic field. To decouple this two equations we can apply the curl operator on (1.1)

$$\operatorname{curl}\operatorname{curl}\mathbf{E} = -\tfrac{d}{dt}\operatorname{curl}\mathbf{B}.$$

Let us assume that $\mathbf{J}$ is constant and insert (1.2) in the right hand side (rhs), so we obtain the electric wave equation

$$\operatorname{curl}\operatorname{curl}\mathbf{E} = -\tfrac{d^2}{dt^2}\mathbf{E}. \tag{2.1}$$

The two dimensional version of (2.1) can be used to compute the electrical field for translational symmetric systems. We consider numerical solutions for such version using FEM, on this purpose we start stating the Cauchy problem.

2.1 Boundary Value Problem

The curl operator is defined for functions $\mathbf{u} \in C^1(\mathbb{R}^3;\mathbb{R}^3)$. For our two dimensional problem we use the differential operators

$$\mathbf{curl}_{2D}\, u = \begin{pmatrix} -\tfrac{\partial u}{\partial y} & \tfrac{\partial u}{\partial x} \end{pmatrix}^T, \text{ for } u \in C^1(\mathbb{R}^2;\mathbb{R}),$$

and

$$\operatorname{curl}_{2D}\mathbf{u} = \frac{\partial u^1}{\partial y} - \frac{\partial u^2}{\partial x}, \text{ for } \mathbf{u} \in C^1(\mathbb{R}^2;\mathbb{R}^2).$$

Then the electric field $\mathbf{E}(\mathbf{x},t)$ in homogeneous, isotropic materials solves the boundary value problem

$$\begin{array}{llll} \frac{d^2}{dt^2}\mathbf{E} + \mathbf{curl}_{2D}\,\mathrm{curl}_{2D}\,\mathbf{E} & = 0 & \text{in} & \Omega \times (0,T) \\ \mathbf{E}(\cdot,t) \times \mathbf{n} & = 0 & \text{on} & \partial_\Omega \times (0,T) \\ \mathbf{E}(\mathbf{x},0) & = \mathbf{E}_0 & \text{in} & \Omega \\ \frac{d}{dt}\mathbf{E}(\mathbf{x},0) & = 0 & \text{in} & \Omega, \end{array} \tag{2.2}$$

where $\Omega \in \mathbb{R}^2$ is a bounded domain, $T \in \mathbb{R}_+$ and $\mathbf{E}_0 \in \mathbf{H}_0(\mathbf{curl};\Omega) := \left\{\mathbf{v} \in \mathbf{L^2}(\Omega) :\ \mathrm{curl}_{2D}\,\mathbf{v} \in \mathbf{L^2}(\Omega), \mathbf{v}\times\mathbf{n} = 0\right\}$. Let us also assume $\operatorname{div}\mathbf{E}_0 = 0$. Now we want to give the variational formulation to (2.2), to this end we proof first the following claim.

Claim 1.

$$(\boldsymbol{curl}_{2D}\ curl_{2D}\,\mathbf{E}, v)_{L^2(\Omega)} = (curl_{2D}\,\mathbf{E}, curl_{2D}\,\mathbf{v})_{L^2(\Omega)}$$

Where $(\mathbf{u},\mathbf{v})_{L^2(\Omega)} := \int_\Omega \langle \mathbf{u},\mathbf{v}\rangle dx$ *is the* L^2*-scalar product,* $\mathbf{v}$ *is any test function with* $\mathbf{v}$, $\mathbf{E} \in \mathbf{H}_0(\boldsymbol{curl};\Omega)$.

Proof. take $\nu := \mathrm{curl}_{2D}\,\mathbf{E}$ and $\eta := v_1\,dx_1 + v_2\,dx_2$, and define $\omega = \nu \wedge \eta$. Clearly $\nu \in \mathcal{DF}^{0,1}(\Omega)$ and $\eta,\omega \in \mathcal{DF}^{1,1}(\Omega)$.

$$\begin{array}{rcl} d\omega & = & d\nu \wedge \eta + \nu \wedge d\eta \\ & = & \left(\frac{\partial}{\partial x_1}\mathrm{curl}_{2D}\,\mathbf{E} + \frac{\partial}{\partial x_2}\mathrm{curl}_{2D}\,\mathbf{E}\right) \wedge (v_1\,dx_1 + v_2\,dx_2) \\ & + & \mathrm{curl}_{2D}\,\mathbf{E}\left(-\frac{\partial}{\partial v_1} + \frac{\partial}{\partial v_2}\right) dx_1 \wedge dx_2 \\ & = & \left(\frac{\partial}{\partial x_1}\mathrm{curl}_{2D}\,\mathbf{E}v_2 - \frac{\partial}{\partial x_2}\mathrm{curl}_{2D}\,\mathbf{E}v_1\right) dx_1 \wedge dx_2 \\ & - & \mathrm{curl}_{2D}\,\mathbf{E}\mathrm{curl}_{2D}\,\mathbf{v}dx_1 \wedge dx_2 \\ & = & (\langle \mathbf{curl}_{2D}\,\mathrm{curl}_{2D}\,\mathbf{E},\mathbf{v}\rangle - \mathrm{curl}_{2D}\,\mathbf{E}\mathrm{curl}_{2D}\,\mathbf{v})\,dx_1 \wedge dx_2. \end{array}$$

From Stokes Theorem we obtain

$$\begin{array}{rcl} \int_\Omega d\omega & = & \int_\Omega (\langle \mathbf{curl}_{2D}\,\mathrm{curl}_{2D}\,\mathbf{E},\mathbf{v}\rangle - \mathrm{curl}_{2D}\,\mathbf{E}\mathrm{curl}_{2D}\,\mathbf{v})\,dx_1 \wedge dx_2 \\ & = & \int_{\partial_\Omega} \omega = \int_{\partial_\Omega} \mathrm{curl}_{2D}\,\mathbf{E} \wedge \mathbf{v}. \end{array}$$

To see that the right-hand-side (rhs.) of the last term vanishes, recall that $\gamma_\mathbf{t}\mathbf{v} = 0$ for $\mathbf{v} \in \mathbf{H}_0(\mathbf{curl};\Omega)$, i.e. $\int_{\partial_\Omega} v_1\,dx_1 + v_2\,dx_2 = 0$ holds pointwise, but then $\int_{\partial_\Omega} \nu v_1 dx_1 + \nu v_2 dx_2 = \int_{\partial_\Omega} \mathrm{curl}_{2D}\,\mathbf{E} \wedge \mathbf{v} = 0$ □

Let $a(u,v) := (\mathrm{curl}_{2D}\,\mathbf{E}, \mathrm{curl}_{2D}\,\mathbf{v})_{L^2(\Omega)}$. The weak formulation of (2.2) reads: Find $\mathbf{E} \in C^2([0;T], \mathbf{H}_0(\mathbf{curl};\Omega))$ such that

$$\begin{array}{llll} \left(\frac{d^2}{dt^2}\mathbf{E},\mathbf{v}\right)_{L^2(\Omega)} + a(\mathbf{E},\mathbf{v}) & = 0 & \text{in} & \Omega \times (0,T) \\ (E_h(\mathbf{x},0),\mathbf{v})_{L^2(\Omega)} & = (E_0,\mathbf{v})_{L^2(\Omega)} & \text{in} & \Omega \\ \left(\frac{\partial}{\partial t}E_h(\mathbf{x},0),\mathbf{v}\right)_{L^2(\Omega)} & = 0 & \text{in} & \Omega\,, \end{array} \tag{2.3}$$

for any $\mathbf{v} \in \mathbf{H}_0(\mathbf{curl};\Omega)$.

Remark 1. *Note that the operator $a(\cdot,\cdot)$ is symmetric and satisfies an elliptical condition (it follows immediately from the Friedrichs Inequality). This fact implies the existence end uniqueness of Weak solutions of* (2.3)*. Weak solutions can be approximated using FEM. In the next section we describe the application of this method.*

2.2 Finite Elements

In this section we want to give a detailed procedure to approximate (2.3) using FEM. The starting point of every FE-algorithm is the discretization of the domain Ω. We choose a triangular mesh $\mathcal{T}_h = \{T_i\}_N$, where $T_i := (\mathbf{a}_1^i, \mathbf{a}_2^i, \mathbf{a}_3^i)$ is the i-th triangle with vertices $a_j^i \in \Omega$, $j = 1, 2, 3$, h is the mesh width, and $N(h) := |\mathcal{T}_h|$.

Let $\mathbf{V}_h \in \mathbf{H}_0(\mathbf{curl};\Omega)$ be a finite dimensional linear subspace with basis $\{w_i\}_N$. We define the FE-approximation $\mathbf{E}_h = \sum_{i=1}^{N} E_i(t)w_i(\mathbf{x})$ to $\mathbf{E} \in C^2([0;T], \mathbf{H}_0(\mathbf{curl};\Omega))$ by: Find $\mathbf{E}_h \in V_h$ such that

$$\begin{array}{lcll} \left(\frac{d^2}{dt^2}\mathbf{E}_h, \mathbf{v}\right)_{L^2(\Omega)} + a(\mathbf{E}_h, \mathbf{v}) & = & 0 & \text{in} \quad \Omega \times (0,T) \\ (E_h(\mathbf{x},0), \mathbf{v})_{L^2(\Omega)} & = & (E_0, \mathbf{v})_{L^2(\Omega)} & \text{in} \quad \Omega \\ \left(\frac{\partial}{\partial t}E_h(\mathbf{x},0), \mathbf{v}\right)_{L^2(\Omega)} & = & 0 & \text{in} \quad \Omega\,, \end{array} \tag{2.4}$$

for any $\mathbf{v} \in V_h$. Expanding $\mathbf{E}_h$ and $\mathbf{v}$ on its basis functions we obtain

$$\left(\sum_{i=1}^{N} \frac{d^2}{dt^2}E_i(t)w_i(\mathbf{x}), \sum_{j=1}^{N} v_j w_j(\mathbf{x})\right)_{L^2(\Omega)} + a\left(\sum_{i=1}^{N} E_i(t)w_i(\mathbf{x}), \sum_{j=1}^{N} v_j w_j(\mathbf{x})\right) =$$
$$\sum_{i=1}^{N}\sum_{j=1}^{N} v_j \left(w_i(\mathbf{x}), w_j(\mathbf{x})\right)_{L^2(\Omega)} \frac{d^2}{dt^2}E_i(t) + \sum_{i=1}^{N}\sum_{j=1}^{N} v_j \; a\left(w_i(\mathbf{x}), w_j(\mathbf{x})\right) E_i(t) \qquad .$$

We can write this expression using matrices as

$$\vec{\mathbf{v}}^t M\ddot{\vec{\mathbf{E}}} + \vec{\mathbf{v}}^t C\vec{\mathbf{E}} = 0$$

where $\vec{\mathbf{E}} := \{E_i\}_N$, $\vec{\mathbf{v}} := \{v_j\}_N$, $M_{ij} := (w_i, w_j)_{L^2(\Omega)}$ and $C_{ij} := a(w_i, w_j)$. Finally the ODE corresponding to (2.4) reads

$$\begin{array}{lcl} M\ddot{\vec{\mathbf{E}}} + C\vec{\mathbf{E}} & = & 0 \\ \dot{\vec{\mathbf{E}}}^0 & = & 0 \\ \vec{\mathbf{E}}^0 & = & \Pi_{V_h}\mathbf{E}_0 \end{array} \tag{2.5}$$

This linear system has a leak, it will not ensure $\operatorname{div} E(\cdot,t) = 0$ for all times. Regularization terms will solve the problem. This will be discussed in the next section. Now, before we take a closer look in the FE-spaces $\mathbf{V}_h$, we describe the FE-algorithm, which is mainly based on

- a reference element $\hat{T}$
- an element mapping $F_T : \hat{T} \to T \in \mathcal{T}_h$
- reference shape-functions $\hat{\mathbf{N}}$,

where $\hat{T} := (\hat{\mathbf{a}}_\mathbf{1}|\hat{\mathbf{a}}_\mathbf{2}|\hat{\mathbf{a}}_\mathbf{3})$, $\hat{\mathbf{a}}_\mathbf{i} \in \mathbb{R}^2$, $i = 1, 2, 3$ (Figure 2.1).

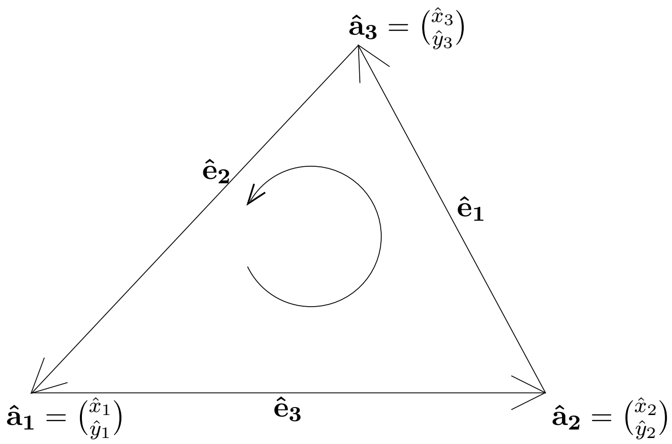


Figure 2.1: Reference element

The numerical approximation of the matrices in (2.5) is usually performed computing first the corresponding matrices locally, then assembling these local contributions to the corresponding global matrices. The local matrices can be computed evaluating the corresponding operators on the reference element and using the following affine map

$$\mathbf{x} = F_T(\hat{\mathbf{x}}) = \mathbf{a_1} + \mathbf{B_T}\,\hat{\mathbf{x}}, \quad \text{where} \quad \mathbf{B_T} = [\mathbf{a_2} - \mathbf{a_1}, \mathbf{a_2} - \mathbf{a_1}]\,. \qquad (2.6)$$

Considering the shape functions note that every point within $\hat{T}$ can be represented using barycentric coordinates $\lambda_i(\mathbf{x})\, i = 1, 2, 3$. They are linear and have the property $\lambda_i(\mathbf{a}_j) = \delta_{ij}$. They can be written as

$$\begin{aligned}
\lambda_1(\mathbf{x}) &= \frac{1}{2|\hat{T}|}(\mathbf{x} - \begin{pmatrix}\hat{x}_2\\ \hat{y}_2\end{pmatrix}) \cdot \begin{pmatrix}\hat{y}_2 - \hat{y}_3\\ \hat{x}_3 - \hat{x}_2\end{pmatrix}\ ,\\
\lambda_2(\mathbf{x}) &= \frac{1}{2|\hat{T}|}(\mathbf{x} - \begin{pmatrix}\hat{x}_3\\ \hat{y}_3\end{pmatrix}) \cdot \begin{pmatrix}\hat{y}_3 - \hat{y}_1\\ \hat{x}_1 - \hat{x}_3\end{pmatrix}\ ,\\
\lambda_3(\mathbf{x}) &= \frac{1}{2|\hat{T}|}(\mathbf{x} - \begin{pmatrix}\hat{x}_1\\ \hat{y}_1\end{pmatrix}) \cdot \begin{pmatrix}\hat{y}_1 - \hat{y}_2\\ \hat{x}_2 - \hat{x}_1\end{pmatrix}\ .
\end{aligned}$$

Their gradients are constant and read

$$
\begin{aligned}
\mathbf{grad}\,\lambda_1 &= \tfrac{1}{2|\hat{T}|}\begin{pmatrix}\hat{y}_2-\hat{y}_3\\ \hat{x}_3-\hat{x}_2\end{pmatrix}, \\
\mathbf{grad}\,\lambda_2 &= \tfrac{1}{2|\hat{T}|}\begin{pmatrix}\hat{y}_3-\hat{y}_1\\ \hat{x}_1-\hat{x}_3\end{pmatrix}, \\
\mathbf{grad}\,\lambda_3 &= \tfrac{1}{2|\hat{T}|}\begin{pmatrix}\hat{y}_1-\hat{y}_2\\ \hat{x}_2-\hat{x}_1\end{pmatrix},
\end{aligned} \tag{2.7}
$$

where $|\hat{T}|$ denotes the area of $\hat{T}$.
We will use the following FE-spaces

- $\mathcal{S}_h := \{v \in C^0(\Omega)\cap H_0^1(\Omega),\ v_{|T}\in \mathcal{P}_1(T)\,\forall T\in\mathcal{T}_h\}$,
  with local basis $\mathcal{B}_{\mathcal{S}_{\hat{T}}} = \{\lambda_1,\ \lambda_2,\ \lambda_3\}$,
- $\mathcal{N}_h := \{\mathbf{v}\in (C^0(\Omega))^2\cap \mathbf{H}_0(\mathbf{curl};\Omega),\ \mathbf{v}_{|T}\in(\mathcal{P}_1(T))^2\,\forall T\in\mathcal{T}_h\}$
  with local basis $\mathcal{B}_{\mathcal{N}_{\hat{T}}} = \left\{\binom{\lambda_1}{0},\ \binom{0}{\lambda_1},\ \binom{\lambda_2}{0},\ \binom{0}{\lambda_2},\ \binom{\lambda_3}{0},\ \binom{0}{\lambda_3}\right\}$
- $\mathcal{E}_h :=$ lowest order Whitney 1-forms $\subset \mathbf{H}_0(\mathbf{curl};\Omega)$,
  with local basis $\mathcal{B}_{\mathcal{E}_{\hat{T}}} = \left\{\begin{array}{lcl}\lambda_2\mathbf{grad}\,\lambda_3 & - & \lambda_3\mathbf{grad}\,\lambda_2,\\ \lambda_3\mathbf{grad}\,\lambda_1 & - & \lambda_1\mathbf{grad}\,\lambda_3,\\ \lambda_1\mathbf{grad}\,\lambda_2 & - & \lambda_2\mathbf{grad}\,\lambda_1\end{array}\right\}$

The Edge elements are a very powerful tool to discretize Maxwell's equations. The reason is that they present the same properties in discrete spaces as differential forms have in continuous spaces. Let for example $\hat{\mathbf{e}}_\mathbf{i}$ be the edge of $\hat{T}$ opposite to $\hat{\mathbf{a}}_\mathbf{i}$ and directed as shown in Figure 2.1, furthermore let $\phi_j \in \mathcal{B}_{\mathcal{E}_{\hat{T}}}$, then the local edge elements satisfy

$$
\frac{1}{|\hat{T}|}\int_T \mathrm{curl}_{2D}\,\phi_i d\mathbf{x} \underbrace{=}_{\text{Gauss}} -\frac{1}{|\hat{T}|}\int_{\partial T}\phi_i d\vec{\mathbf{s}} = -\frac{1}{|\hat{T}|}\sum_{j=1}^{3}\underbrace{\int_{\hat{\mathbf{e}}_\mathbf{j}}\phi_i d\vec{\mathbf{s}}}_{\delta_{ij}} = -\frac{1}{|\hat{T}|},
$$

i.e. the curl_{2D} of these vector fields are constant on $\hat{T}$. This fact will be useful for the generation of the local curl Matrix. Another property of edge elements is useful for the regularization of (2.4), it is described next.

2.3 Regularization

The FE-discretization (2.4) may produce approximations to non-physical solutions, the so-called spurious solutions. The reason lays in the kernel of the operator $a(\cdot,\cdot)$ [1, p. 318, Ch. 6], [2]. Recall that in (2.2) we required $\mathrm{div}\,\mathbf{E}_0 = 0$, theoretically this conditions ensures that $\mathbf{E}(\cdot,t)$ behaves divergence-free for all $t \in [0,T]$. Unfortunately, "a slight perturbation of the initial value might lead to growing curl_{2D}-free components in $\mathbf{E}(.,t)$ that may eventually swamp the physically meaningful solution. A remedy is offered by regularization"[1].

2.3.1 grad-div Regularization

Consider the electric Maxwell's eigenvalue problem

$$(\text{curl}_{2D}\, E, \text{curl}_{2D}\, \mathbf{v})_{L^2(\Omega)} = \omega^2 \left(\mathbf{E}, \mathbf{v}\right)_{L^2(\Omega)} \;, \forall \mathbf{v} \in \mathbf{H}_0(\mathbf{curl};\Omega).$$

It can be proven ([1, Ch. 4]) that the solution
$\mathbf{E} \in Z_0(I,\Omega) := \left\{ \mathbf{u} \in H_0(\text{curl};\Omega) \middle| (\mathbf{u},\mathbf{Z})_{L^2(\Omega)} = 0, \forall \mathbf{z} \in H_0(\text{curl}0;\Omega) \right\}$. The reason why this is relevant for the continuous regularization is to be clarified with the next claim.

Claim 2. $\mathbf{E} \in Z_0(I,\Omega) \Rightarrow div\,\mathbf{E} = 0$

Proof. From Poincaré's theorem we know that a 0-form $\nu_{\mathbf{z}}$ exist, s.t.
$\omega_{\mathbf{z}} = d\nu_{\mathbf{z}}$, where $\omega_{\mathbf{u}} := u_1 dx_1 + u_2 dx_2$ is the 1-form induced by the vector field $\mathbf{u} \in C(\Omega;\mathbb{R}^2)$.
Let $\xi := \nu \wedge *\omega_{\mathbf{E}}$ then

$$\int_\Omega d\xi = \underbrace{\int_\Omega d\nu_{\mathbf{z}} \wedge *\omega_{\mathbf{E}}}_{=(\mathbf{E},\mathbf{z})_{L^2(\Omega)}=0} + \underbrace{\int_\Omega \nu_{\mathbf{z}} \wedge d * \omega_{\mathbf{E}}}_{\int_\Omega \nu_{\mathbf{z}} \text{div}\, \mathbf{E} d\mathbf{x}} \underbrace{=}_{\text{stoke's Thm.}} \int_{\partial\Omega} \nu_{\mathbf{z}} \wedge *\omega_{\mathbf{E}}.$$

A test function $\mathbf{z}$ s.t. $\nu_{\mathbf{z}} \in H_0^1(\Omega)$ yields the result. □

Our goal is to state a variational problem equivalent to (2.3). Using the last claim we end up with: seek $\mathbf{E} \in C^2([0;T], \mathbf{H}_0(\mathbf{curl};\Omega) \cap H(\text{div};\Omega))$ such that for all $\mathbf{v} \in \mathbf{H}_0(\mathbf{curl};\Omega) \cap H(\text{div};\Omega)$

$$\begin{array}{lcll} \left(\frac{d^2}{dt^2}\mathbf{E},\mathbf{v}\right)_{L^2(\Omega)} + a(\mathbf{E},\mathbf{v}) + (\text{div}\,\mathbf{E}, \text{div}\,\mathbf{v})_{L^2(\Omega)} & = & 0 & \text{in} \quad \Omega \times (0,T) \\ (E_h(\mathbf{x},0),\mathbf{v})_{L^2(\Omega)} & = & (E_0,\mathbf{v})_{L^2(\Omega)} & \text{in} \quad \Omega \\ \left(\frac{\partial}{\partial t}E_h(\mathbf{x},0),\mathbf{v}\right)_{L^2(\Omega)} & = & 0 & \text{in} \quad \Omega\,. \end{array} \tag{2.8}$$

We will use $\mathcal{N}_h$ to discretize (2.8). Proceeding the same way as in (2.4) we end up with the following ODE

$$\begin{array}{lcl} \hat{M}\ddot{\vec{\mathbf{E}}} + \hat{C}\vec{\mathbf{E}} + \hat{R}\vec{\mathbf{E}} & = & 0 \\ \dot{\vec{\mathbf{E}}}^0 & = & 0 \\ \vec{\mathbf{E}}^0 & = & \Pi_{V_h}\mathbf{E}_0\,, \end{array} \tag{2.9}$$

[1]Quoting from [3]

where $\hat{R}$ corresponds to the regularization term, $\hat{C}$ and $\hat{M}$ are the stiffness and mass matrices. Here we denote with $\hat{\cdot}$ a matrix w.r.t. $\mathcal{N}_h$. Note that, since $\mathcal{N}_h \in H^1(\Omega)$, we are looking for a FE-approximation $\mathbf{E}_h \in H^1_x := H^1(\Omega) \cap \mathbf{H}_0(\mathbf{curl};\Omega)$. Unfortunately $\mathbf{E}_h$ does not always converge to E. This is the statement of the following theorem from [1, Ch 6, Thm. 6.3, p.322].

Theorem 1. *The space $H^1_x(\Omega)$ is a closed subspace of $X_0(I,\Omega)$ and the inclusion is strict, if Ω has re-entrant edges or corners.*

We will illustrate this phenomena with an example, where an electromagnetic field on a square domain and on an L-shaped domain is approximated. A comparison's reference is delivered by edge elements using a discrete regularization.

2.3.2 Discrete Regularization

The variational problem (2.8) can not be discretized with edge elements because $\mathcal{E}_h \not\subseteq H(\mathrm{div}\,,\Omega)$. The way out is to regularise (2.4), exploiting the fact that $\mathrm{grad}\,\mathcal{S}_h \subset \mathcal{E}_h$, we obtain

$$(\mathbf{E}_h, \mathbf{grad}\, v_h)_{L^2(\Omega)} = 0 \quad \forall\, v_h \in \mathcal{S}_h, \tag{2.10}$$

for $\mathbf{E}_h \in \mathcal{E}_h$ solving (2.4). The last expression holds in a discrete level, but since Whitney forms behave as differential forms do on a continuous level, (2.10) can be justified, considering $\omega := v \wedge *\mathbf{E}$, and

$$\int_\Omega d\omega = \underbrace{\int_\Omega dv \wedge *\mathbf{E}}_{=\int_\Omega \langle \mathbf{grad}\, v, \mathbf{E}\rangle d\mathbf{x}} + \underbrace{\int_\Omega v \wedge d * \mathbf{E}}_{=\int_\Omega v \mathrm{div}\, \mathbf{E} d\mathbf{x}} \overset{\text{Stokes thm.}}{=} \underbrace{\int_{\partial\Omega} \omega}_{\int_{\partial\Omega} v \wedge *\mathbf{E}} = 0 \quad \forall v \in H^1_0(\Omega).$$

The discrete regularised weak formulation reads: Find $\mathbf{E}_h \in C^2([0,T],\mathcal{E}_h)$, such that

$$\begin{array}{lcll}
\left(\frac{d^2}{dt^2}\mathbf{E}_h, \mathbf{v}\right)_{L^2(\Omega)} + a(\mathbf{E}_h, \mathbf{v}) + (v_h, \mathbf{grad}\, p_h)_{L^2(\Omega)} & = & 0 & \text{in} \quad \Omega \times (0,T) \\
(\mathbf{E}_h, \mathbf{grad}\, q_h)_{L^2(\Omega)} - d(p_h, q_h) & = & 0 & \text{in} \quad \Omega \times (0,T) \\
(E_h(\mathbf{x},0), \mathbf{v})_{L^2(\Omega)} & = & (E_0, \mathbf{v})_{L^2(\Omega)} & \text{in} \quad \Omega \\
\left(\frac{\partial}{\partial t} E_h(\mathbf{x},0), \mathbf{v}\right)_{L^2(\Omega)} & = & 0 & \text{in} \quad \Omega\,,
\end{array} \tag{2.11}$$

for any $\mathbf{v}_h \in \mathcal{E}_h$, $q_h \in \mathcal{S}_h$, where $d(\cdot,\cdot)$ is an arbitrary symmetric positive definite (spd) bilinear form on $\mathcal{S}_h$ as $p_h = 0$ anyway. For numerical issues a practical choice is the lumped $L^2(\Omega)$ inner product, since its corresponding

matrix is diagonal.
Expanding $\mathbf{E}_h$, v_h on its respective basis functions, we obtain

$$(\mathbf{E}_h, \mathbf{grad}\, q_h)_{L^2(\Omega)} = \left(\sum_{i=1}^{N_\mathcal{E}} \mathbf{E}_h^i(t) w_i^\mathcal{E}, \mathbf{grad} \sum_{j=1}^{N_\mathcal{N}} w_j^\mathcal{S} \right)_{L^2(\Omega)} = \sum_{i=1}^{N_\mathcal{E}} \sum_{j=1}^{N_\mathcal{N}} \underbrace{\left(w_i^\mathcal{E}, \mathbf{grad}\, w_j^\mathcal{N} \right)_{L^2(\Omega)}}_{=:G_{ij}} \mathbf{E}_h^i(t).$$

The coupled ODE for (2.11) reads

$$\begin{array}{rcccccl} M\ddot{\vec{\mathbf{E}}} & + & C\vec{\mathbf{E}} & + & G\vec{\mathbf{p}} & = & 0 \\ & & G^t\vec{\mathbf{E}} & - & D\vec{\mathbf{p}} & = & 0, \end{array}$$

where D corresponding to $d(\cdot,\cdot)$ is diagonal, M and C are the mass and curl matrices w.r.t $\mathcal{E}_h$. Decoupling we obtain

$$\begin{array}{lcl} M\ddot{\vec{\mathbf{E}}} + (C + GD^{-1}G^t)\vec{\mathbf{E}} & = & 0 \\ \dot{\vec{\mathbf{E}}}^0 & = & 0 \\ \vec{\mathbf{E}}^0 & = & \Pi_{\mathcal{E}_h}\mathbf{E}_0\,. \end{array} \tag{2.12}$$

In the next section we discuss a way to compute approximations to the solutions of (2.12) and (2.9).

2.4 Time stepping

Clearly we are interested only in Runge-Kutta schemes conserving the total energy in the system. We choose the leapfrog scheme and apply it to (2.9),

$$\begin{array}{lcl} \vec{\mathbf{E}}^0 & = & \hat{M}^{-1}\Pi_{\mathcal{N}_h}\mathbf{E}_0 \\ \vec{\mathbf{E}}^1 & = & \vec{\mathbf{E}}^0 - 1/2\tau^2\hat{M}^{-1}(\hat{C} + \hat{R})\vec{\mathbf{E}}^0 \\ \vec{\mathbf{E}}^{n+1} & = & 2\vec{\mathbf{E}}^n - \vec{\mathbf{E}}^{n-1} - \tau^2\hat{M}^{-1}(\hat{C} + \hat{R})\vec{\mathbf{E}}^n\,, \end{array} \tag{2.13}$$

where the condition $\dot{\vec{\mathbf{E}}} = 0$ is interpreted as $\vec{\mathbf{E}}^1 = \vec{\mathbf{E}}^{-1}$ and used to obtain $\vec{\mathbf{E}}^1$.
The starting condition for the $\mathcal{E}_h$-discretization is a little more complicated, as we have to ensure that $\vec{\mathbf{E}}^0$ is divergence-free on the discrete level, ie

$$\left(\operatorname{div}\mathbf{E}_h^0, \phi_h\right)_{L^2(\Omega)} = \left(\mathbf{E}_h^0, \mathbf{grad}\,\phi_h\right)_{L^2(\Omega)} = 0 \quad \forall\,\phi_h \in \mathcal{S}_h. \tag{2.14}$$

This condition is fulfilled, if we find $\mathbf{E}_h^0 \in \mathcal{N}_h$, $\mathbf{u}_h \in \mathcal{N}_h$ for all $\mathbf{v}_h, \mathbf{w}_h \in \mathcal{N}_h$

$$\begin{array}{rcl} \left(\mathbf{E}_h^0, \mathbf{v}_h\right)_{L^2(\Omega)} + (\operatorname{curl}_{2D}\mathbf{u}_h, \operatorname{curl}_{2D}\mathbf{v}_h)_{L^2(\Omega)} & = & (\mathbf{E}_0, \mathbf{v}_h)_{L^2(\Omega)}\;, \\ \left(\operatorname{curl}_{2D}\mathbf{E}_h^0, \operatorname{curl}_{2D}\mathbf{w}_h\right)_{L^2(\Omega)} & = & \left(\operatorname{curl}_{2D}\mathbf{E}^0, \operatorname{curl}_{2D}\mathbf{w}_h\right)_{L^2(\Omega)}\;. \end{array}$$

Let Q_h denote the $L^2(\Omega)$-orthogonal projection onto the space $\mathcal{T}_h$ of piecewise constant functions, and $\Pi_{\mathcal{E}_h}$ the local edge elements interpolation operator, then the following diagram holds

$$\mathrm{curl}_{2D} \circ \Pi_{\mathcal{E}_h} = Q_h \circ \mathrm{curl}_{2D}$$

Note that $\mathrm{curl}_{2D}\, \mathbf{E}_h^0 = Q_h(\mathrm{curl}_{2D}\, \mathbf{E}_0)$, i.e. $\mathrm{curl}_{2D}\, (E_h^0 - \Pi_{\mathcal{E}_H} \mathbf{E}_0) = 0$, thus exists $\phi_h \in \mathcal{S}_1$ with $E_h^0 - \Pi_{\mathcal{E}} E_0 = \mathbf{grad}\, \phi_h$ and satisfies

$$(\mathbf{grad}\, \phi_h, \mathbf{grad}\, \psi_h)_{L^2(\Omega)} = (\mathbf{E}_0 - \Pi_h \mathbf{E}_0, \mathbf{grad}\, \psi_h)_{L^2(\Omega)} = -\left(\Pi_h \mathbf{E}_0, \mathbf{grad}\, \psi_h\right)_{L^2(\Omega)} .$$

Its matrix representation reads

$$G^t M G \vec{\phi} = G^t M \Pi_{\mathcal{E}_h} \vec{\mathbf{E}}_0$$

Substitution of $\vec{\phi}$ in $\vec{\mathbf{E}}_h^0 - \Pi_{\mathcal{E}_h} \mathbf{E}_0 = G\vec{\phi}$ yields the desired starting value. Thus the ODE to be considered reads

$$\begin{array}{lcl} \vec{\mathbf{E}}_h^0 & = & (I + G(G^t M G)^{-1} G^t M)\Pi_{\mathcal{E}_h} \mathbf{E}_0 \\ \vec{\mathbf{E}}^1 & = & \vec{\mathbf{E}}^0 - 1/2\tau^2 M^{-1}(C + G D^{-1} G^t)\vec{\mathbf{E}}^0 \\ \vec{\mathbf{E}}^{n+1} & = & 2\vec{\mathbf{E}}^n - \vec{\mathbf{E}}^{n-1} - \tau^2 M^{-1}(C + G D^{-1} G^t)\vec{\mathbf{E}}^n . \end{array} \qquad (2.15)$$

A CFL condition

$$\begin{array}{lcl} \left\| 1/2\tau^2 \hat{M}^{-1}(\hat{C} + \hat{R}) \right\| & \leq & 1 \quad \text{for (2.13), and} \\ \left\| 1/2\tau^2 M^{-1}(C + G D^{-1} G^t) \right\| & \leq & 1 \quad \text{for (2.15)} \end{array}$$

ensures the stability of the scheme as time evolves. An accurate estimation of the time step τ requires the computation of the largest eigenvalue of the corresponding operators. In our simulation we only ensure that the CFL condition is fulfilled, thus we just choose $\tau = Ch$, where h is the mesh width and the constant $0 \leq C \in \mathbb{R}$ small enough. Our implementation computes approximations to the solutions of (2.13) and (2.15). We give in the next chapter some details of the structure of the program.

Chapter 3

Implementation

In this chapter we present the implementation of a program computing approximations to the solutions for transient Maxwell's equations using FEM. The program was done in MATLAB using the "LehrFem" framework, so most of the code was already available. Very useful was the code of Prof. Hiptmair, the structure of the main function is actually based on this code.

The program is structured as shown in Figure 3.1. The starting point is the routine `run.m`. It computes and plots the electrical field $\vec{\mathbf{E}}$, and plots also the total energy, for both (2.12) and (2.9), giving a singular function as starting value $\mathbf{E}_0$, for an square mesh width an initial mesh width h_0 and initial time step $\tau_0 = 0.01$. The representation of the plots allows an easy comparison between the two discretizations. The final time is set to $T = 3$. A plot of the time evolution of the magnetic and electrical energy is displayed when T is reached. The same is performed for an L-shaped mesh. This process is repeated "NREFS= 5" times, refining in each step the mesh by $h_{n+1} = h_n/2$. The routine `run_smooth.m` performs the equivalent simulation, but for smooth starting conditions. More details about the numerical experiments are given in Chapter 4.

3.1 Code

In this section we list almost all the code. Some less important routines are not included, even though there is quite a lot of code. For this reason we have divided the routines in five groups. In the first group we have listed the most important functions, i.e the functions needed to compute (2.12) and (2.9). The second group contains the routines implementig the starting functions, the next section lists the routines used to plot the results, followed by the routines used for meshing and finally we list the routine used to plot the computed results again saving the ploted results in avi-files.

3.1.1 Main Routines

3.1.1.1 run.m

```
function run(datapath,prb,scal)
% MATLAB-Script running the numerical experiments for
% the transient Maxwell's equations in a cavity

if (nargin < 1), datapath = './'; end
```

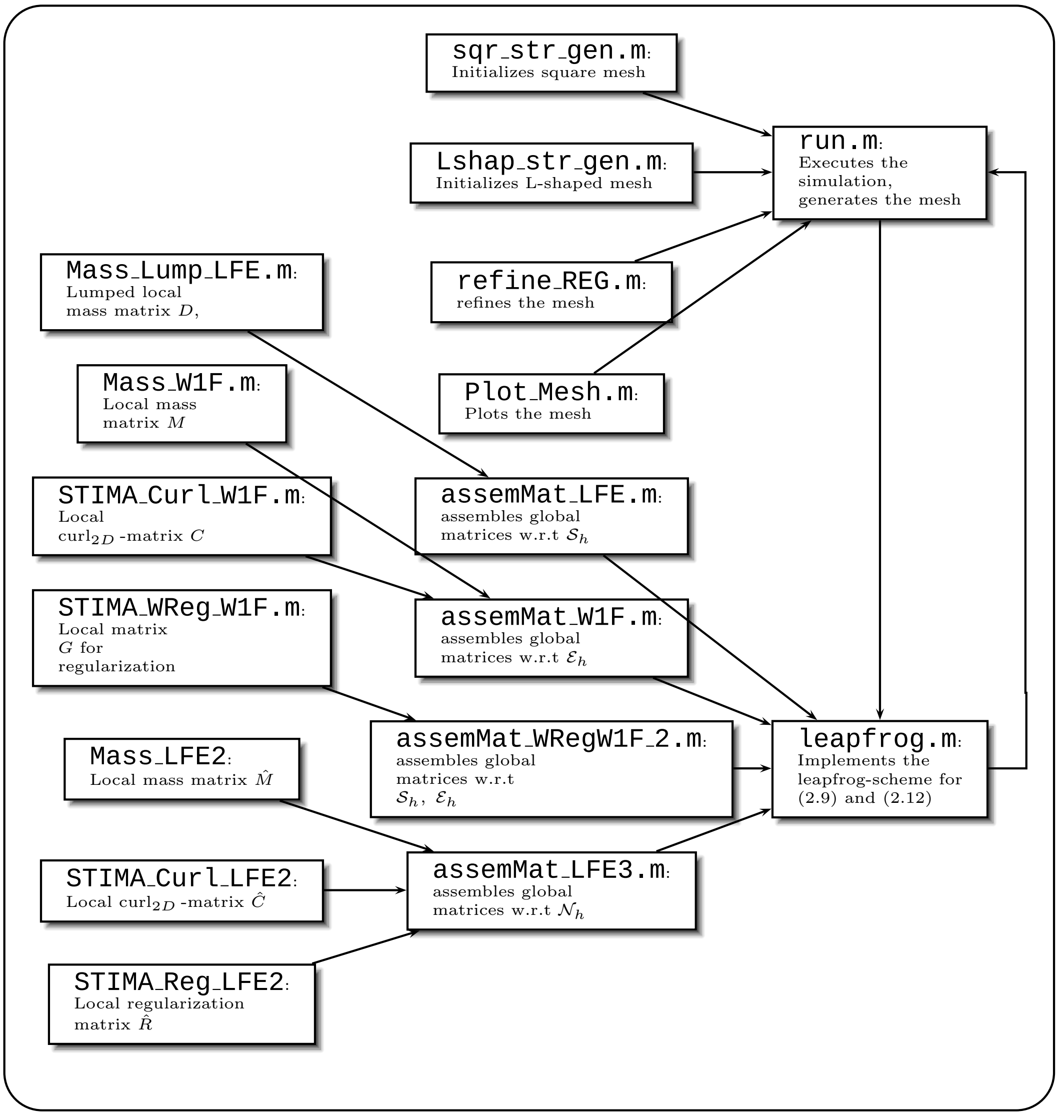


Figure 3.1: Structure of the Implementation

```matlab
if (nargin < 2), prb = -1; end
if (nargin < 3), scal = 0.5; end

% Initialize constants

InitREF = 2; % size of the Initial Mesh
NREFSs = 5;  % Number of unifrom mesh refinements
finaltime = 3;
timestep = 0.01;

disp('MATLAB based numerical experiments for transient Maxwell equations');
disp('Goal is the comparison of nodal and edge element discretizations');
fprintf('Path for output files %s\n',datapath);
fprintf('Relative scaling of C and R: %d <-> %d\n',2*scal,2*(1-scal));

% Generate initial meshes, where the meshwidth depends on InitREF
%Square mesh
MeshS=sqr_str_gen(InitREF);
%Add to the mesh some useful information to handle edge elements
MeshS.ElemFlag = ones(size(MeshS.Elements,1),1);
MeshS = add_Edges(MeshS);
LocS = get_BdEdges(MeshS);
MeshS.BdFlags = zeros(size(MeshS.Edges,1),1);
MeshS.BdFlags(LocS) = -1;

%L-shaped mesh
MeshL=Lshap_str_gen(InitREF);
%Add to the mesh some useful information to handle edge elements
MeshL.ElemFlag = ones(size(MeshL.Elements,1),1);
MeshL = add_Edges(MeshL);
LocL = get_BdEdges(MeshL);
MeshL.BdFlags = zeros(size(MeshL.Edges,1),1);
MeshL.BdFlags(LocL) = -1;

% Do NREFS uniform refinement steps

for i = 1:NREFSs

    % For the square mesh
    % Refine Mesh
    MeshS = refine_REG(MeshS);
    % plot it
    plot_Mesh(MeshS,'as')

    % start leapfrog with starting condition initSq
    [en,sol_v,sol_e,times] = leapfrog(MeshS,@initSq,timestep*(2^(-i+1)),finaltime,2^(i-1

    %Saving Data
    Sq_str=['Square' int2str(i)];
    fprintf(['Finished on' Sq_str ': Results stored in %s'],[datapath,[Sq_str,'_res']]);
    save([datapath, Sq_str, '_res'],'en','sol_v','sol_e','times');

    %plot energy evolution in time
    figure; clf;
```

```
    subplot(1,2,1);
    plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
    legend('Nodal scheme','Edge elements');
    title([Sq_str,': Electric energy']);
    xlabel('time');
    subplot(1,2,2);
    plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
    legend('Nodal scheme','Edge elements');
    title([Sq_str,': Magnetic energy']);
    xlabel('time');
    drawnow;
    clear en solv sole times;
    % end for the square mesh


    %For the L-mesh
    % Refine mesh
    MeshL = refine_REG(MeshL);
    plot_Mesh(MeshL,'as')

    % start leapfrog with starting condition initL
    [en,sol_v,sol_e,times] = leapfrog(MeshL,@initL,timestep*(2^(-i+1)),finaltime,2^(i-1),

    %Saving Data
    L_str=['Lshape' int2str(i)];
    fprintf(['Finished on' L_str ': Results stored in %s'],[datapath,[L_str,'_res']]);
    save([datapath, L_str, '_res'],'en','sol_v','sol_e','times');

    % Actualize energy plot
    figure; clf;
    subplot(1,2,1);
    plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
    legend('Nodal scheme','Edge elements');
    title([L_str,': Electric energy']);
    xlabel('time');
    subplot(1,2,2);
    plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
    legend('Nodal scheme','Edge elements');
    title([L_str,': Magnetic energy']);
    xlabel('time');
    drawnow;
    clear en solv sole times;


end
```

3.1.1.2 leapfrog.m

```
function [energies,sol_v,sol_e,times] = leapfrog(Mesh,init_field,ts,T,grabstep,scal)
% leapfrog timestepping discretizations of Maxwell's equations
%
% Mesh          -> 2D unstructured mesh
```

```
% init_field    -> string designating the routine providing the initial
%                  vectorfield
% ts            -> timestep
% T             -> final time
% grabstep      -> every #grabstep iterate will be sampled
% scal          ->  governs strength of regularization
%                   A = 2*(scal*C + (1-scal)*R)
%
% Result:
%
% energies -> trace of electric/magnetic energies during timestepping
%         energies(:,1) = time,
%         energies(:,2) = electric energy of nodal solution
%         energies(:,3) = magnetic energy of nodal solution
%         energies(:,4) = electric energy of edge element solution
%         energies(:,5) = magnetic energy of edge element solution
% sol_v -> sampled solutions for nodal FEM
% sol_e -> sampled solutions for edge elements
% times -> vector of sampling times
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Initialize some functions

U_Handle = @(x,varargin)ones(size(x,1),1);
F_Handle = @(x,varargin)[zeros(size(x,1),1) zeros(size(x,1),1)];
GD_Handle = @(x,varargin)[-x(:,2) x(:,1)];


%Add Boundary plot data to the mesh, this data will be used by
%plotiterate1.m
[Mesh.BdEdges_x Mesh.BdEdges_y]=dataBoundaryPlot(Mesh);
Mesh=setBdFlags(Mesh);
nCoordinates = size(Mesh.Coordinates,1);

%determine scalation and the number of steps to be saved, if they weren't
%specified as arguments
if (nargin < 6), scal = 0.5; end
if (nargin < 5), grabstep = 1; end


%%%%%%%%% Edge elements matrix generation  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ap equals the stiffnes matrix for the curl without regularization and
% discretized with whithney-1 edge elements, Me is the corresponding
% Mass Matrix the error between the theoretical matrices is 2.2737e-13

D = assemMat_LFE(Mesh,@MASS_Lump_LFE);
G = assemMat_WRegW1F_2(Mesh,@STIMA_WReg_W1Fb,P7O6(),U_Handle);
Ap = assemMat_W1F(Mesh,@STIMA_Curl_W1F,U_Handle,P7O6());
Me = assemMat_W1F(Mesh,@MASS_W1F,U_Handle,P7O6());
%G=Me*G1;

% Determine degrees of freedom
```

```
Loc = get_BdEdges(Mesh);
DEdges = Loc(Mesh.BdFlags(Loc) == -1);
DNodes = unique([Mesh.Edges(DEdges,1); Mesh.Edges(DEdges,2)]);
VDofs = setdiff(1:nCoordinates,DNodes);
EDofs = setdiff(1:size(Mesh.Edges,1),DEdges);

% Curl-matrix with regularization
Ae=Ap(EDofs,EDofs)+G(EDofs,VDofs)*(D(VDofs,VDofs)\G(EDofs,VDofs)');

% Projection of initial value onto discrete space
ev = assemLoad_W1F(Mesh,P7O6(),init_field);

%%% non discrete divergence free
Me=Me(EDofs,EDofs);
ev = ev(EDofs);
ev = Me\ev;     %Starting value

%%% Discrete divergence free  %%%%%%%%%%%%%%%%%%
% Is not working with smooth initial condition
%%% Topological gradient. Note that G=Me*G1.
G1=Gradmat(Mesh);
G1=G1(EDofs,VDofs);
p = G1*((G1'*Me*G1)\(G1'*Me*ev));
ev = ev + p;      %Starting value

clear Ap D G

%%%%%%%%% Nodal elements matrix generation  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% An equals the stiffnes matrix for the curl without regularization and
% discretized with linear finite Elements elements, Mn is the corresponding
% Mass Matrix
An = assemMat_LFE3(Mesh,@STIMA_Curl_LFE2)+assemMat_LFE3(Mesh,@STIMA_Reg_LFE2);
Mn = assemMat_LFE3(Mesh,@MASS_LFE2);

% Determine degrees of freedom
NDofs = [2*find(Mesh.VertBdFlags(:,1) == 0); 2*find(Mesh.VertBdFlags(:,2) == 0)-1];
An=An(NDofs,NDofs);
Mn=Mn(NDofs,NDofs);

% Projection of initial value onto discrete space
nv = assemLoad_LFE3(Mesh,P7O6(),init_field);
nv = nv(NDofs);
nv = Mn\nv;     % Starting Value

%%%%  Memory alocation for lepafrog-scheme   %%%%%%%%%%%%%%%%%%%%%%%%
nv_new = zeros(size(nv));
```

```
nv_mid = zeros(size(nv));
nv_tmp = zeros(size(nv));
ev_new = zeros(size(ev));
ev_mid = zeros(size(ev));
ev_tmp = zeros(size(ev));


%%% initialize the plot window
disp('Displaying initial iterates');
figure;
clf; figno = gcf;
%%% actualize the plot window
plotiterate1(Mesh,ev,nv,0,figno,NDofs,EDofs);

%%% precomputing some variables before the time-iteration
sol_v = nv;
sol_e = ev;
times = [0.0];
nv_tmp = An*nv;
ev_tmp = Ae*ev;
etot_v = dot(nv_tmp,nv);
etot_e = dot(ev_tmp,ev);

disp('Initial energies:');
fprintf('\t #### Nodal scheme : E_el = %f, E_mag = %f\n',...
    0,etot_v);
fprintf('\t #### Edge elements: E_el = %f, E_mag = %f\n',...
    0,etot_e);

% First step
nv_mid = nv - 0.5*ts*ts*(Mn\nv_tmp);
ev_mid = ev - 0.5*ts*ts*(Me\ev_tmp);

stp = 1;
if (grabstep == 1)
    sol_v = [sol_v nv_mid];
    sol_e = [sol_e ev_mid];
    times = [times ts];
end

plotiterate1(Mesh,ev_mid,nv_mid,ts,gcf,NDofs,EDofs);
energies = [0.0 0.0 etot_v 0.0 etot_e];

% % %%%% uncomment if memory lacks
save_idx=0;
%%release memory step. Stores results in harddisk every 128*grabstep steps
relMemStep=128*grabstep;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for t=2*ts:ts:T
    stp = stp + 1;
    fprintf('Iteration step %d at time %d\n',stp,t);
    nv_tmp = An*nv_mid;
    ev_tmp = Ae*ev_mid;
```

```
    nv_new = 2*nv_mid - nv - ts*ts*(Mn\nv_tmp);
    ev_new = 2*ev_mid - ev - ts*ts*(Me\ev_tmp);

    disp('Displaying new iterate');

    plotiterate1(Mesh,ev_new,nv_new,t,figno,NDofs,EDofs);


    if (mod(stp,grabstep) == 0)
        sol_v = [sol_v nv_new];
        sol_e = [sol_e ev_new];
        times = [times t];
        %%% uncomment if memory lacks
            if(mod(stp,relMemStep)==0)
                save_idx=save_idx+1;
                file_name=[int2str(save_idx) 'results.mat'];
                save(file_name, 'sol_v', 'sol_e', 'times');
                sol_v = [];
                sol_e = [];
                times = [];
            end

        %F(stp/grabstep) = getframe(figno);
    end

    % Computing energies
    nv = (nv_new - nv)/(2*ts);
    ev = (ev_new - ev)/(2*ts);
    el_en_v = dot(nv,Mn*nv);
    el_en_e = dot(ev,Me*ev);
    mag_en_v = dot(nv_tmp,nv_mid);
    mag_en_e = dot(ev_tmp,ev_mid);

    fprintf('\t Nodal scheme : E_el = %f, E_mag = %f, E_tot = %f\n',...
        el_en_v,mag_en_v,el_en_v+mag_en_v);
    fprintf('\t Edge elements: E_el = %f, E_mag = %f, E_tot = %f\n',...
        el_en_e,mag_en_e,el_en_e+mag_en_e);
    if (el_en_v+mag_en_v > 10*etot_v)
        disp('Instability of nodal scheme!');
        break;
    end
    if (el_en_e+mag_en_e > 10*etot_e)
        disp('Instability of edge element scheme!');
        break;
    end

    energies = [energies; t el_en_v mag_en_v el_en_e mag_en_e];
    nv = nv_mid;
    ev = ev_mid;
    ev_mid = ev_new;
    nv_mid = nv_new;
end

%%% Uncomment if memory lacks
```

```
save_idx=save_idx+1;
file_name=[int2str(save_idx) 'results.mat'];
save(file_name, 'sol_v', 'sol_e', 'times');
save('energies.mat','energies', 'save_idx');
clear variables;
%close all;
load energies;
load '1results'
sol_e_tmp=sol_e;
sol_v_tmp=sol_v;
times_tmp=times;
for i=2:save_idx
    filename=[int2str(i) 'results'];
    load(filename);
    sol_e_tmp=[sol_e_tmp sol_e];
    sol_v_tmp=[sol_v_tmp sol_v];
    times_tmp=[times_tmp times];
end
sol_e=sol_e_tmp;
sol_v=sol_v_tmp;
times=times_tmp;
%movie(F,1,10)
```

3.1.1.3 assemMat_LFE.m

```
function varargout = assemMat_LFE(Mesh,EHandle,varargin)
% ASSEMMAT_LFE Assemble linear FE contributions.
%
%   A = ASSEMMAT_LFE(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.
%
%   A = ASSEMMAT_LFE(MESH,EHANDLE,EPARAM) handles the variable length
%   argument list EPARAM to the function handle EHANDLE during the assembly
%   process.
%
%   [I,J,A] = ASSEMMAT_LFE(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.
%
%   Example:
%
%   Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
%   Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
%   EHandle = @STIMA_Lapl_LFE;
%   A = assemMat_LFE(Mesh,EHandle);
```

```
%
%   See also set_Rows, set_Cols.

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  nElements = size(Mesh.Elements,1);

  % Preallocate memory

  I = zeros(9*nElements,1);
  J = zeros(9*nElements,1);
  A = zeros(9*nElements,1);

  % Check for element flags

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

  % Assemble element contributions

  loc = 1:9;
  for i = 1:nElements

    % Extract vertices of current element

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

    % Compute element contributions

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

    % Add contributions to stiffness matrix

    I(loc) = set_Rows(idx,3);
    J(loc) = set_Cols(idx,3);
    A(loc) = Aloc(:);
    loc = loc+9;

  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
```

3.1.1.4 MASS_Lump_LFE.m

```
function Aloc = MASS_Lump_LFE(Vertices,varargin)
% MASS_LUMP_LFE element lumbda mass matrix.
%
%   ALOC = MASS_LUMP_LFE(VERTICES) computes the element mass matrix
%   using W1F finite elements.
%
%   VERTICES is 3-by-2 matrix specifying the vertices of the current
%   element in a row wise orientation.
%
%   Example:
%
%   Aloc = MASS_Lump_LFE(Vertices);

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % 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 = 1/6*det_BK*eye(3);

return
```

3.1.1.5 P706.m

```
function QuadRule = P706()
% P706 2D Quadrature rule.
%
%   QUADRULE = P706() computes a 7 point Gauss quadrature rule of order 6
%   (exact for all polynomials up to degree 5) on the reference element.
%
%   QUADRULE is a struct containing the following fields:
%    w Weights of the quadrature rule
%    x Abscissae of the quadrature rule (in reference element)
%
%   To recover the barycentric coordinates xbar of the quadrature points
```

```
%    xbar = [QuadRule.x, 1-sum(QuadRule.x)'];
%
%   Example:
%
%   QuadRule = P7O6();

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  QuadRule.w = [               9/80; ...
               (155+sqrt(15))/2400; ...
               (155+sqrt(15))/2400; ...
               (155+sqrt(15))/2400; ...
               (155-sqrt(15))/2400; ...
               (155-sqrt(15))/2400; ...
               (155-sqrt(15))/2400 ];

  QuadRule.x = [             1/3              1/3; ...
                 (6+sqrt(15))/21   (6+sqrt(15))/21; ...
               (9-2*sqrt(15))/21   (6+sqrt(15))/21; ...
                 (6+sqrt(15))/21 (9-2*sqrt(15))/21; ...
                 (6-sqrt(15))/21 (9+2*sqrt(15))/21; ...
               (9+2*sqrt(15))/21   (6-sqrt(15))/21; ...
                 (6-sqrt(15))/21   (6-sqrt(15))/21 ];

return
```

3.1.1.6 shap_W1F.m

```
function shap = shap_W1F(x)
% SHAP_W1F Shape functions.
%
%   SHAP = SHAP_W1F(X) computes the values of the shape functions for the
%   edge finite element (Whitney-1-Form) at the quadrature points X.
%
%   Example:
%
%   shap = shap_W1F([0 0]);
%
%   See also shap_LFE2.

%   Copyright 2005-2005 Patrick Meury and Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  shap = zeros(size(x,1),6);

  shap(:,1) = -x(:,2);
  shap(:,2) = x(:,1);
```

```
  shap(:,3) = -x(:,2);
  shap(:,4) = x(:,1)-1;
  shap(:,5) = 1-x(:,2);
  shap(:,6) = x(:,1);

return
```

3.1.1.7 assemMat_W1F.m

```
function varargout = assemMat_W1F(Mesh,EHandle,varargin)
% ASSEMMAT_W1F Assembly for *edge elements* in 2D
%
%   A = ASSEMMAT_W1F(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.
%
%   A = ASSEMMAT_W1F(MESH,EHANDLE,EPARAM) handles the variable length
%   argument list EPARAM to the function handle EHANDLE during the assembly
%   process.
%
%   [I,J,A] = ASSEMMAT_W1F(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.
%    VERT2EDGE    Edge numbers associated with pairs of vertices
%                 (sparse matrix)
%
%   Example:
%
%   Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
%   Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
%   EHandle = @STIMA_Curl_W1F;
%   A = assemMat_W1F(Mesh,EHandle);
%
%   See also SET_ROWS, SET_COLS.

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  nElements = size(Mesh.Elements,1);

  % Preallocate memory

```

```
  I = zeros(9*nElements,1);
  J = zeros(9*nElements,1);
  A = zeros(9*nElements,1);

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

  % Assemble element contributions

  loc = 1:9;
  for i = 1:nElements

    % Extract vertices of current element

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

    % Compute element contributions

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

    % Extract global edge numbers

    eidx = [Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3)) ...
            Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1)) ...
            Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))];

    % Determine the orientation

    if(Mesh.Edges(eidx(1),1)==vidx(2)),  p1 = 1;  else    p1 = -1;
end
    if(Mesh.Edges(eidx(2),1)==vidx(3)),  p2 = 1;  else    p2 = -1;
end
    if(Mesh.Edges(eidx(3),1)==vidx(1)),  p3 = 1;  else    p3 = -1;
end

    Peori = diag([p1 p2 p3]); % scaling matrix taking into account orientations
    Aloc = Peori*Aloc*Peori;

    % Add contributions to stiffness matrix

    I(loc) = set_Rows(eidx,3);
    J(loc) = set_Cols(eidx,3);
    A(loc) = Aloc(:);
    loc = loc+9;

  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
```

3.1.1.8 MASS_W1F.m

```
function Mloc = MASS_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule,varargin)
% MASS_W1F element mass matrix with weight mu for edge elements in 2D
%
%   MLOC = MASS_W1F(VERTICES) computes the element mass matrix using
%   Whitney 1-forms finite elements.
%
%   VERTICES is 3-by-2 matrix specifying the vertices of the current element
%   in a row wise orientation.
%
%   ElemInfo (not used)
%
%   MU_HANDLE handle to a functions expecting a matrix whose rows
%   represent position arguments. Return value must be a vector
%   (variable arguments will be passed to this function)
%
%   Example:
%
%   Mloc = MASS_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule);

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Compute element mapping

  P1 = Vertices(1,:);
  P2 = Vertices(2,:);
  P3 = Vertices(3,:);

  BK = [ P2 - P1 ; P3 - P1 ]; % transpose of transformation matrix
  det_BK = abs(det(BK));     % twice the area of the triagle

  % Compute constant gradients of barycentric coordinate functions
  g1 = [P2(2)-P3(2);P3(1)-P2(1)]/det_BK;
  g2 = [P3(2)-P1(2);P1(1)-P3(1)]/det_BK;
  g3 = [P1(2)-P2(2);P2(1)-P1(1)]/det_BK;

  % Get barycentric coordinates of quadrature points
  nPoints = size(QuadRule.w,1);
  baryc= [QuadRule.x,1-sum(QuadRule.x,2)];

  % Quadrature points in actual element
```

```
  % stored as rows of a matrix
  x = QuadRule.x*BK + ones(nPoints,1)*P1;

  % Evaluate coefficient function at quadrature nodes
  Fval = MU_HANDLE(x,ElemInfo,varargin{:});

  % Evaluate basis functions at quadrature points
  % the rows of b(i) store the value of the the i-th
  % basis function at the quadrature points
  b1 = baryc(:,2)*g3'-baryc(:,3)*g2';
  b2 = baryc(:,3)*g1'-baryc(:,1)*g3';
  b3 = baryc(:,1)*g2'-baryc(:,2)*g1';

  % Compute local mass matrix

  weights = QuadRule.w * det_BK;
  Mloc(1,1) = sum(weights.*Fval.*sum(b1.*b1,2));
  Mloc(2,2) = sum(weights.*Fval.*sum(b2.*b2,2));
  Mloc(3,3) = sum(weights.*Fval.*sum(b3.*b3,2));
  Mloc(1,2) = sum(weights.*Fval.*sum(b1.*b2,2)); Mloc(2,1) = Mloc(1,2);
  Mloc(1,3) = sum(weights.*Fval.*sum(b1.*b3,2)); Mloc(3,1) = Mloc(1,3);
  Mloc(2,3) = sum(weights.*Fval.*sum(b2.*b3,2)); Mloc(3,2) = Mloc(2,3);

  return
```

3.1.1.9 STIMA_Curl_W1F.m

```
function Aloc = STIMA_Curl_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule,varargin)
% STIMA_CURL_W1F element stiffness matrix for curl*curl-operator in 2D
% in the case of Galerkin discretization by means of edge elements
%
%   ALOC = STIMA_CURL_W1F(VERTICES,ELEMINFO,MU_HANDLE,QUADRULE) computes the
%   curl*\mu*curl element stiffness matrix using Whitney 1-forms finite elements.
%   The function \mu can be passed through the MU_HANDLE argument
%
%   VERTICES is 3-by-2 matrix specifying the vertices of the current element
%   in a row wise orientation.
%
%   ElemInfo (not used)
%
%   MU_HANDLE handle to a functions expecting a matrix whose rows
%   represent position arguments. Return value must be a vector
%   (variable arguments will be passed to this function)
%
%  QuadRule is a quadrature rule on the reference element
%
%   Example:
%
%   Aloc = STIMA_Curl_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule);

%   Copyright 2005-2006Patrick Meury & Mengyu Wang & Ralf Hiptmair
%   SAM - Seminar for Applied Mathematics
```

```
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constant

  nPoints = size(QuadRule.w,1);

  % Compute element mapping

  bK = Vertices(1,:); % row vector !
  BK = [Vertices(2,:)-bK; Vertices(3,:)-bK]; % Transpose of trafo matrix !
  det_BK = abs(det(BK)); % twice the area of the triangle

  % Quadrature points in actual element
  % stored as rows of a matrix
  x = QuadRule.x*BK + ones(nPoints,1)*bK;

  % Compute function value

  Fval = MU_HANDLE(x,ElemInfo,varargin{:});

  % Compute local curl-curl-matrix
  % Use that the curl of an edge element function is constant
  % and equals 1/area of triangle

  Aloc = 4/det_BK*sum(QuadRule.w.*Fval)*ones(3,3);
return
```

3.1.1.10 assemMat_WRegW1F_2.m

```
function varargout = assemMat_WRegW1F_2(Mesh,EHandle,varargin)
% ASSEMMAT_WREGW1F Assemble WREG W1F FE contributions.
%
%   A = ASSEMMAT_WREGW1F(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.
%
%   A = ASSEMMAT_WREGW1F(MESH,EHANDLE,EPARAM) handles the variable length
%   argument list EPARAM to the function handle EHANDLE during the assembly
%   process.
%
%   [I,J,A] = ASSEMMAT_WREGW1F(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.
%
%   Example:
```

```matlab
%
%   Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
%   Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
%   EHandle = @STIMA_WReg_W1F;
%   A = assemMat_WRegW1F(Mesh,EHandle);
%
%   See also SET_ROWS, SET_COLS.

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

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

  % Preallocate memory

  I = zeros(9*nElements,1);
  J = zeros(9*nElements,1);
  A = zeros(9*nElements,1);

  % Assemble element contributions

  loc = 1:9;

  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{:});


    % Extract global edge numbers

    eidx = [Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3)) ...
            Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1)) ...
            Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))];

    % Determine the orientation

    if(Mesh.Edges(eidx(1),1)==vidx(2))
        p1 = 1;
    else
        p1 = -1;
    end

```

```
    if(Mesh.Edges(eidx(2),1)==vidx(3))
        p2 = 1;
    else
        p2 = -1;
    end

    if(Mesh.Edges(eidx(3),1)==vidx(1))
        p3 = 1;
    else
        p3 = -1;
    end

    Peori = diag([p1 p2 p3]);
    Aloc = Peori*Aloc;

    % Add contributions to stiffness matrix

    I(loc) = set_Rows(eidx,3);
    J(loc) = set_Cols(vidx,3);
    A(loc) = Aloc(:);
    loc = loc+9;

  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
```

3.1.1.11 grad_shap_LFE.m

```
function grad_shap = grad_shap_LFE(x)
% GRAD_SHAP_LFE Gradient of shape functions.
%
%   GRAD_SHAP = GRAD_SHAP_LFE(X) computes the values of the gradient
%   of the shape functions for the Lagrangian finite element of order 1
%   at the quadrature points X.
%
%   Example:
%
%   grad_shap = grad_shap_LFE([0 0]);
%
%   See also shap_LFE.

%   Copyright 2005-2005 Patrick Meury and Kah Ling Sia
```

```
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  nPts = size(x,1);

  % Preallocate memory

  grad_shap = zeros(nPts,6);

  % Compute values of gradients

  grad_shap(:,1:2) = -ones(nPts,2);
  grad_shap(:,3) = ones(nPts,1);
  grad_shap(:,6) = ones(nPts,1);

return
```

3.1.1.12 STIMA_WReg_W1Fb.m

```
function Aloc = STIMA_WReg_W1F(Vertices,ElemInfo,QuadRule,varargin)
% Using QuadRule for the future work(space dependent version)

    % Initialize constant

    nGuass = size(QuadRule.w,1);

    % Preallocate memory

    Aloc = zeros(3,3);
    N_W1F = shap_W1F(QuadRule.x);
    grad_N = grad_shap_LFE(QuadRule.x);

    % Compute element mapping

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

    % Compute element entry

    N(:,1:2) = N_W1F(:,1:2)*TK;
    N(:,3:4) = N_W1F(:,3:4)*TK;
    N(:,5:6) = N_W1F(:,5:6)*TK;
    G(:,1:2) = grad_N(:,1:2)*TK;
```

```
    G(:,3:4) = grad_N(:,3:4)*TK;
    G(:,5:6) = grad_N(:,5:6)*TK;

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

return
```

3.1.1.13 assemLoad_W1F.m

```
function L = assemLoad_W1F(Mesh,QuadRule,FHandle,varargin)
% ASSEMLOAD_W1F Assemble W1F FE contributions.
%
%   L = ASSEMLOAD_W1F(MESH,QUADRULE,FHANDLE) assembles the global load
%   vector for the load data 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.
%    ELEMFLAG     N-by-1 matrix specifying additional element information.
%
%   QUADRULE is a struct, which specifies the Gauss qaudrature that is used
%   to do the integration:
%    W Weights of the Gauss quadrature.
%    X Abscissae of the Gauss quadrature.
%
%   L = ASSEMLOAD_W1F(COORDINATES,QUADRULE,FHANDLE,FPARAM) also handles the
%   additional variable length argument list FPARAM to the function handle
%   FHANDLE.
%
%   Example:
%
%   FHandle = @(x,varargin)[x(:,1).^2 x(:,2).^2];
%   L = assemLoad_W1F(Mesh,P706(),FHandle);
%
%   See also shap_W1F.

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  nPts = size(QuadRule.w,1);
```

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

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

% Preallocate memory

L = zeros(nEdges,1);

% Precompute shape functions

N = shap_W1F(QuadRule.x);

% Assemble element contributions

eidx = zeros(1,3);
for i = 1:nElements

  % Extract vertices

  vidx = Mesh.Elements(i,:);
  eidx(1) = Mesh.Vert2Edge(vidx(2),vidx(3));
  eidx(2) = Mesh.Vert2Edge(vidx(3),vidx(1));
  eidx(3) = Mesh.Vert2Edge(vidx(1),vidx(2));

  % Compute element mapping

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

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

  % Compute load data

  FVal = FHandle(x);

  % Determine the orientation

  if(Mesh.Edges(eidx(1),1)==vidx(2))
      p1 = 1;
  else
      p1 = -1;
  end

  if(Mesh.Edges(eidx(2),1)==vidx(3))
      p2 = 1;
  else
      p2 = -1;
  end
```

```
    if(Mesh.Edges(eidx(3),1)==vidx(1))
        p3 = 1;
    else
        p3 = -1;
    end

    % Add contributions to global load vector

    L(eidx(1)) = L(eidx(1)) + sum(QuadRule.w.*sum(FVal.*([N(:,1) N(:,2)]*TK),2))*det_BK*p
    L(eidx(2)) = L(eidx(2)) + sum(QuadRule.w.*sum(FVal.*([N(:,3) N(:,4)]*TK),2))*det_BK*p
    L(eidx(3)) = L(eidx(3)) + sum(QuadRule.w.*sum(FVal.*([N(:,5) N(:,6)]*TK),2))*det_BK*p

  end

return
```

3.1.1.14 shap_LFE2.m

```
function shap = shap_LFE2(x)
% SHAP_LFE2 Shape functions.
%
%   SHAP = SHAP_LFE2(X) computes the values of the shape functions for
%   the vector valued Lagrangian finite element of order 1 at the
%   quadrature points X.
%
%   Example:
%
%   shap = shap_LFE2([0 0]);
%
%   See also shap_LFE, shap_W1F.

%   Copyright 2005-2005 Patrick Meury and Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  shap = zeros(size(x,1),12);

  shap(:,1) = 1-x(:,1)-x(:,2);
  shap(:,4) = 1-x(:,1)-x(:,2);
  shap(:,5) = x(:,1);
  shap(:,8) = x(:,1);
  shap(:,9) = x(:,2);
  shap(:,12) = x(:,2);

return
```

3.1.1.15 assemMat_LFE3.m

```
function varargout = assemMat_LFE3(Mesh,EHandle,varargin)
```

```
% ASSEMMAT_LFE2 Assemble nodal FE contributions.
%
%   A = ASSEMMAT_LFE2(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.
%
%   A = ASSEMMAT_LFE2(MESH,EHANDLE,EPARAM) handles the variable length
%   argument list EPARAM to the function handle EHANDLE during the assembly
%   process.
%
%   [I,J,A] = ASSEMMAT_LFE2(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.
%
%   Example:
%
%   Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
%   Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
%   EHandle = @STIMA_Curl_LFE2;
%   A = assemMat_LFE2(Mesh,EHandle);
%
%   See also SET_ROWS, SET_COLS.

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

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

  % Preallocate memory

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

  % Assemble element contributions

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

    % Extract vertices of current element

    vidx = Mesh.Elements(i,:);
%     idx = [vidx(1) vidx(1)+nCoordinates ...
```

```
%            vidx(2) vidx(2)+nCoordinates ...
%            vidx(3) vidx(3)+nCoordinates];

idx = [2*vidx(1)-1 2*vidx(1)...
           2*vidx(2)-1 2*vidx(2) ...
           2*vidx(3)-1 2*vidx(3)];

    Vertices = Mesh.Coordinates(vidx,:);

    % Compute element contributions

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

    % Add contributions to stiffness matrix

    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
```

3.1.1.16 MASS_LFE2.m

```
function Mloc = MASS_LFE2(Vertices,varargin)
% MASS_LFE2 Element mass matrix.
%
%   MLOC = MASS_LFE2(VERTICES) computes the element mass matrix using
%   LFE2 finite elements.
%
%   VERTICES is 3-by-2 matrix specifying the vertices of the current element
%   in a row wise orientation.
%
%   Example:
%
%   Mloc = MASS_LFE2(Vertices);

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
```

```
%   CH-8092 Zurich, Switzerland

  % Compute element mapping

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

  % Compute local mass matrix

  Mloc = det_BK/24*[2 0 1 0 1 0;...
                    0 2 0 1 0 1;...
                    1 0 2 0 1 0;...
                    0 1 0 2 0 1;...
                    1 0 1 0 2 0;...
                    0 1 0 1 0 2];

return
```

3.1.1.17 STIMA_Curl_LFE2.m

```
function Aloc = STIMA_Curl_LFE2(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.
%
%   Example:
%
%   Aloc = STIMA_Curl_LFE2(Vertices);

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % 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
```

3.1.1.18 STIMA_Reg_LFE2.m

```
function Aloc = STIMA_WReg_W1Fa(Vertices,ElemInfo,varargin)
% STIMA_WREG_W1F Element stiffness matrix for the W1F finite element.
%
%   ALOC = STIMA_WREG_W1F(VERTICES,ELEMINFO) computes the element stiffness
%   matrix for the data given by function handle FHANDLE.
%
%   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.
%
%   Example:
%
%   Aloc = STIMA_WReg_W1F([0 0; 1 0; 0 1],0);
%
%   See also grad_shap_LFE.

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Preallocate memory

  Aloc = zeros(3,3);

  % Compute element mapping

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


  L = [ P2(2)-P3(2),P3(1)-P2(1),P3(2)-P1(2),P1(1)-P3(1),P1(2)-P2(2),P2(1)-P1(1) ]/(det_BK
  Aloc = det_BK/2*L'*L;

  return
```

3.1.1.19 assemLoad_LFE3.m

```
function L = assemLoad_LFE3(Mesh,QuadRule,FHandle,varargin)
% ASSEMLOAD_LFE Assemble nodal FE contributions.
%
```

```
%   L = ASSEMLOAD_LFE2(MESH,QUADRULE,FHANDLE) assembles the global load
%   vector for the load data given by the function handle FHANDLE.
%
%   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.
%
%   QUADRULE is a struct, which specifies the Gauss qaudrature that is used
%   to do the integration:
%    W Weights of the Gauss quadrature.
%    X Abscissae of the Gauss quadrature.
%
%   L = ASSEMLOAD_LFE2(COORDINATES,QUADRULE,FHANDLE,FPARAM) also handles the
%   additional variable length argument list FPARAM to the function handle
%   FHANDLE.
%
%   Example:
%
%   FHandle = @(x,varargin)x(:,1).^2+x(:,2).^2;
%   L = assemLoad_LFE2(Mesh,P706(),FHandle);
%
%   See also shap_LFE2.

%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

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

  % Preallocate memory

  L = zeros(2*nCoordinates,1);

  % Precompute shape functions

  N = shap_LFE2(QuadRule.x);

  % Assemble element contributions

  for i = 1:nElements

    % Extract vertices

    vidx = Mesh.Elements(i,:);

    % Compute element mapping

    bK = Mesh.Coordinates(vidx(1),:);
```

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

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

    % Compute load data

    %FVal = FHandle(x,Mesh.ElemFlag(i),varargin{:});
    FVal = FHandle(x,Mesh.ElemFlag(i),varargin{:});

    % Add contributions to global load vector

    L(2*vidx(1)-1) = L(2*vidx(1)-1) + sum(QuadRule.w.*FVal(:,1).*N(:,1))*det_BK;
    L(2*vidx(2)-1) = L(2*vidx(2)-1) + sum(QuadRule.w.*FVal(:,1).*N(:,5))*det_BK;
    L(2*vidx(3)-1) = L(2*vidx(3)-1) + sum(QuadRule.w.*FVal(:,1).*N(:,9))*det_BK;

    L(2*vidx(1)) = L(2*vidx(1)) + sum(QuadRule.w.*FVal(:,2).*N(:,4))*det_BK;
    L(2*vidx(2)) = L(2*vidx(2)) + sum(QuadRule.w.*FVal(:,2).*N(:,8))*det_BK;
    L(2*vidx(3)) = L(2*vidx(3)) + sum(QuadRule.w.*FVal(:,2).*N(:,12))*det_BK;

  end

return
```

3.2 Initial Value

3.2.0.20 initL.m

```
function y = initL(x,omega,phioffs)
% Initial electric field
% omega = 3/2*pi for L-shaped domain
% omega = pi/2 for square
omega = 3*pi/2;
phioffs=pi/2;%0.5*pi;
ep = pi/omega;
phi = atan2(x(:,2),x(:,1)) + phioffs;
rad = sqrt(x(:,1).*x(:,1)+x(:,2).*x(:,2));

sgty=find(rad < eps);
rad(sgty)=eps*ones(size(sgty));

p = rad.^(ep).*cos(ep.*phi);
cpx = ep*rad.^(ep-1).*(cos(ep.*phi).*(-x(:,2)) + ...
            sin(ep*phi).*x(:,1))./rad;
cpy = ep*rad.^(ep-1).*(cos(ep.*phi).*x(:,1) + ...
            sin(ep*phi).*x(:,2))./rad;
cp=[cpx cpy];


y=zeros(size(x));
cf=y;
```

```
f=ones(size(x(:,1)));
Loc1 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) < 0.5));

Loc2= find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) < 0.5));
  f(Loc2) = sin(pi*x(Loc2,1)).^2;
  cf(Loc2,2) = pi*sin(2*pi*x(Loc2,1));
Loc3 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) >= 0.5));
  f(Loc3) = sin(pi*x(Loc3,2)).^2;
  cf(Loc3,1) = pi*(-sin(2*pi*x(Loc3,2)));
Loc4 = find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) >= 0.5));
  f(Loc4) = (sin(pi*x(Loc4,1)).*sin(pi*x(Loc4,2))).^2;
  cf(Loc4,:) = pi*[-(sin(pi*x(Loc4,1)).^2).*sin(2*pi*x(Loc4,2)) ...
    sin(2*pi*x(Loc4,1)).*(sin(pi*x(Loc4,2)).^2)];

y = [f.*cpx f.*cpy] + [p.*cf(:,1) p.*cf(:,2)];
```

3.2.0.21 initSq.m

```
function y = initSq(x,omega,phioffs)
% Initial electric field
% omega = 3/2*pi for L-shaped domain
% omega = pi/2 for square
omega = pi/2;
phioffs=0;
ep = pi/omega;
phi = atan2(x(:,2),x(:,1)) + phioffs;
rad = sqrt(x(:,1).*x(:,1)+x(:,2).*x(:,2));

sgty=find(rad < eps);
rad(sgty)=eps*ones(size(sgty));

p = rad.^(ep).*cos(ep.*phi);
cpx = ep*rad.^(ep-1).*(cos(ep.*phi).*(-x(:,2)) + ...
            sin(ep*phi).*x(:,1))./rad;
cpy = ep*rad.^(ep-1).*(cos(ep.*phi).*x(:,1) + ...
            sin(ep*phi).*x(:,2))./rad;
cp=[cpx cpy];


y=zeros(size(x));
cf=y;
f=ones(size(x(:,1)));
Loc1 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) < 0.5));

Loc2= find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) < 0.5));
  f(Loc2) = sin(pi*x(Loc2,1)).^2;
  cf(Loc2,2) = pi*sin(2*pi*x(Loc2,1));
Loc3 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) >= 0.5));
  f(Loc3) = sin(pi*x(Loc3,2)).^2;
  cf(Loc3,1) = pi*(-sin(2*pi*x(Loc3,2)));
Loc4 = find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) >= 0.5));
  f(Loc4) = (sin(pi*x(Loc4,1)).*sin(pi*x(Loc4,2))).^2;
```

```
  cf(Loc4,:) = pi*[-(sin(pi*x(Loc4,1)).^2).*sin(2*pi*x(Loc4,2)) ...
    sin(2*pi*x(Loc4,1)).*(sin(pi*x(Loc4,2)).^2)];

y = [f.*cpx f.*cpy] + [p.*cf(:,1) p.*cf(:,2)];
```

3.2.1 Plotting

3.2.1.1 plotfield1.m

```
function plotfield1(Mesh,vals,BBox,titstr)

% Creates an arrow plot of a vectorfield whose values are stored
% in the vals column vector
%
% Mesh -> Data for 2D unstructured mesh
% vals -> column vector whose length must agree with mesh.Nv
%


nVertices=size(Mesh.Coordinates,1);
if (size(vals,1) ~= 2*nVertices) error('Size mismatch for argument vector'); end
if (size(vals,2) ~= 1), error('Vals must be a colun vector'); end
if (nargin < 3), titstr = 'Arrowplot of vectorfield'; end

hold on;
title(titstr);
bb = [ 0 0 0 0 ];

% Generates plot

  plot(Mesh.BdEdges_x,Mesh.BdEdges_y,'r-');

% Plot arrows
vx = vals(1:2:2*nVertices,1);
vy = vals(2:2:2*nVertices,1);
quiver(Mesh.Coordinates(:,1),Mesh.Coordinates(:,2),vx,vy,0.75,'b-');
axis(BBox*1.01);
hold off;
```

3.2.1.2 plotiterate1.m

```
function F = plotiterate1(Mesh,ev,nv,t,figno,NDofs,EDofs,mesh)
% Plots the current iterate during the leapfrog iteration
%
% mesh -> 2D triangulation
% ev -> vector of edge dofs of length #of active edges
% nv -> vector of nodal dofs of length #of active vertices
% t -> time (for title)
nVertices=size(Mesh.Coordinates,1);
evf = zeros(size(Mesh.Edges(:,1)));
nvf = zeros(2*size(Mesh.Coordinates(:,1),1),1);
```

```
evf(EDofs) = ev;
nvf(NDofs) = nv;
nvfm=sqrt((nvf(1:2:2*nVertices-1,1).^2+nvf(2:2:2*nVertices,1).^2));
s = sprintf('Time = %f',t);

BBox=[-1 1 -1 1];
figure(figno);
clf;
h1=subplot(2,2,1);
h2=subplot(2,2,3);
plot_Norm_W1F(evf,Mesh,h1,h2,BBox,'Edge Elements');
subplot(h2);
axis([BBox 0 3]);
title(s);
view([-30 70]);
subplot(2,2,2);
plotfield1(Mesh,nvf,BBox,'Nodal elements');
h=subplot(2,2,4);
plot_LFE(nvfm,Mesh,h);
axis([BBox 0 3]);
title(s);
view([-30 70]);
```

3.2.1.3 plot_LFE.m

```
function varargout = plot_LFE(U,Mesh,fig)
% PLOT_LFE Plot finite element solution.
%
%   PLOT_LFE(U,MESH) generates a plot of the finite element solution U on
%   the mesh MESH.
%
%   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.
%
%   H = PLOT_LFE(U,MESH) also returns the handle to the figure.
%
%   Example:
%
%   plot_LFE(U,MESH);

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  OFFSET = 0.05;

  % Compute axes limits
```

```matlab

XMin = min(Mesh.Coordinates(:,1));
XMax = max(Mesh.Coordinates(:,1));
YMin = min(Mesh.Coordinates(:,2));
YMax = max(Mesh.Coordinates(:,2));
XLim = [XMin XMax] + OFFSET*(XMax-XMin)*[-1 1];
YLim = [YMin YMax] + OFFSET*(YMax-YMin)*[-1 1];

% Generate figure

if(isreal(U))

  % Compute color axes limits

  CMin = min(U);
  CMax = max(U);
  if(CMin < CMax)          % or error will occur in set function
    CLim = [CMin CMax] + OFFSET*(CMax-CMin)*[-1 1];
  else
    CLim = [1-OFFSET 1+OFFSET]*CMin;
  end

  % Plot real finite element solution
  % Create new figure, if argument 'fig' is not specifiied
  % Otherwise this argument is supposed to be a figure handle
  if (nargin < 3),  fig = figure('Name','Linear finite elements');
  else %figure(fig);
  subplot(fig);
  end

  patch('faces', Mesh.Elements, ...
        'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) U], ...
        'CData', U, ...
        'facecolor', 'interp', ...
        'edgecolor', 'none');
  %set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 4]);

  if(nargout > 0)
    varargout{1} = fig;
  end

else

  % Compute color axes limits

  CMin = min([real(U); imag(U)]);
  CMax = max([real(U); imag(U)]);
  CLim = [CMin CMax] + OFFSET*(CMax-CMin)*[-1 1];

  % Plot imaginary finite element solution

  fig_1 = figure('Name','Linear finite elements');
  patch('faces', Mesh.Elements, ...
        'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) real(U)], ...
```

```
          'CData', real(U), ...
          'facecolor', 'interp', ...
          'edgecolor', 'none');
    set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 4]);
    fig_2 = figure('Name','Linear finite elements');
    patch('faces', Mesh.Elements, ...
          'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) imag(U)], ...
          'CData', imag(U), ...
          'facecolor', 'interp', ...
          'edgecolor', 'none');
    %set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 1]);
    set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 4]);
    if(nargout > 0)
      varargout{1} = fig_1;
      varargout{2} = fig_2;
    end

  end

return
```

3.2.1.4 plot_Mesh.m

```
function varargout = plot_Mesh(Mesh,varargin)
% PLOT_MESH Mesh plot.
%
%   PLOT_MESH(MESH) generate 2D plot of the mesh.
%
%   PLOT(MESH,OPT) adds labels to the plot, where OPT is a character string
%   made from one element from any or all of the following characters:
%    p Add vertex labels to the plot.
%    e Add edge labels/flags to the plot.
%    t Add element labels/flags to the plot.
%    a Dipslay axes on the plot.
%    s Add title and axes labels to the plot.
%    f Do NOT create new window for the mesh plot
%    [c add patch color to elements according to their flags] TODO !
%
%   H = PLOT_MESH(MESH,OPT) also returns the handle to the figure.
%
%   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 or N-by-4 matrix specifying the elements of the
%                mesh.
%
%   Example:
%
%   plot_Mesh(Mesh,'petas');
%
%   See also get_BdEdges, add_Edges.

%   Copyright 2005-2005 Patrick Meury
```

```
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

if(nargin > 1)
  opt = varargin{1};
else
  opt = ' ';
end
  % Initialize constants

  OFFSET = 0.05;      % Offset parameter
  EDGECOLOR = 'b';    % Interior edge color
  BDEDGECOLOR = 'r';  % Boundary edge color

  % Check mesh data structure and add necessary fields

  if(~isfield(Mesh,'Edges'))
    Mesh = add_Edges(Mesh);
  end
  nCoordinates = size(Mesh.Coordinates,1);
  nElements = size(Mesh.Elements,1);
  nEdges = size(Mesh.Edges,1);

  % Compute axes limits

  X = Mesh.Coordinates(:,1);
  Y = Mesh.Coordinates(:,2);
  XMin = min(X);
  XMax = max(X);
  YMin = min(Y);
  YMax = max(Y);
  XLim = [XMin XMax] + OFFSET*(XMax-XMin)*[-1 1];
  YLim = [YMin YMax] + OFFSET*(YMax-YMin)*[-1 1];

  % Compute boundary edges for piecewise linear boundaries

  Loc = get_BdEdges(Mesh);
  BdEdges_x = zeros(2,size(Loc,1));
  BdEdges_y = zeros(2,size(Loc,1));
  BdEdges_x(1,:) = Mesh.Coordinates(Mesh.Edges(Loc,1),1)';
  BdEdges_x(2,:) = Mesh.Coordinates(Mesh.Edges(Loc,2),1)';
  BdEdges_y(1,:) = Mesh.Coordinates(Mesh.Edges(Loc,1),2)';
  BdEdges_y(2,:) = Mesh.Coordinates(Mesh.Edges(Loc,2),2)';

  % Generate plot

  if(isempty(findstr('f',opt)))
    fig = figure('Name','Mesh plot');
  end

  if(~ishold)
    hold on;
  end
```

```
patch('Faces', Mesh.Elements, ...
      'Vertices', Mesh.Coordinates, ...
      'FaceColor', 'none', ...
      'EdgeColor', EDGECOLOR);
plot(BdEdges_x,BdEdges_y,[BDEDGECOLOR '-']);
hold off;
set(gca,'XLim',XLim, ...
        'YLim',YLim, ...
        'DataAspectRatio',[1 1 1], ...
        'Box','on', ...
        'Visible','off');

% Add labels/flags according to the string OPT


  % Add vertex labels

  if(~isempty(findstr('p',opt)))
    add_VertLabels(Mesh.Coordinates);
  end

  % Add element labels/flags to the plot

  if(~isempty(findstr('t',opt)))
    if(isfield(Mesh,'ElemFlag'))
      add_ElemLabels(Mesh.Coordinates,Mesh.Elements,Mesh.ElemFlag);
    else
      add_ElemLabels(Mesh.Coordinates,Mesh.Elements,1:nElements);
    end
  end

  % Add edge labels/flags to the plot

  if(~isempty(findstr('e',opt)))
    if(isfield(Mesh,'BdFlags'))
      add_EdgeLabels(Mesh.Coordinates,Mesh.Edges,Mesh.BdFlags);
    else
      add_EdgeLabels(Mesh.Coordinates,Mesh.Edges,1:nEdges);
    end
  end

  % Turn on axes, titles and labels

  if(~isempty(findstr('a',opt)))
    set(gca,'Visible','on');
    if(~isempty(findstr('s',opt)))
      if(size(Mesh.Elements,2) == 3)
        title(['{\bf 2D triangular mesh}']);
      else
        title(['{\bf 2D quadrilateral mesh}']);
      end
      xlabel(['{\bf # Vertices  :  ', int2str(nCoordinates), ...
              ',      # Elements  :  ', int2str(nElements), ...
              ',      # Edges  :  ',int2str(nEdges),'}']);
```

```matlab
      end
    end

  drawnow;

  % Assign output arguments

  if(nargout > 0)
    varargout{1} = fig;
  end

return


%%% Add vertex labels %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [] = add_VertLabels(Coordinates)
% ADD_VERTLABELS Add vertex labels to the plot.
%
%   ADD_VERTLABELS(COORDINATES) adds vertex labels to the current
%   figure.
%
%   Example:
%
%   add_VertLabels(Mesh.Coordinates);

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  WEIGHT = 'bold';
  SIZE = 8;
  COLOR = 'k';

  % Add vertex labels to the plot

  nCoordinates = size(Coordinates,1);
  for i = 1:nCoordinates
    text(Coordinates(i,1),Coordinates(i,2),int2str(i), ...
         'HorizontalAlignment','Center', ...
         'VerticalAlignment','Middle', ...
         'Color',COLOR, ...
         'FontWeight',WEIGHT, ...
         'FontSize',SIZE);
  end

return

%%% Add element labels %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [] = add_ElemLabels(Coordinates,Elements,Labels)
```

```
% ADD_ELEMLABELS Add element labels to the plot.
%
%   ADD_ELEMLABELS(COORDINATES,ELEMENTS,LABELS) adds the element labels
%   LABELS to the current figure.
%
%   Example:
%
%   add_ElemLabels(Mesh.Coordinates,Mesh.Elements,Labels);

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  WEIGHT = 'bold';
  SIZE = 8;
  COLOR = 'k';

  % Add element labels to the plot

  [nElements,nVert] = size(Elements);
  for i = 1:nElements
    CoordMid = sum(Coordinates(Elements(i,:),:),1)/nVert;
    text(CoordMid(1),CoordMid(2),int2str(Labels(i)), ...
         'HorizontalAlignment','Center', ...
         'VerticalAlignment','Middle', ...
         'Color',COLOR, ...
         'FontWeight',WEIGHT, ...
         'FontSize',SIZE);
  end

return

%%% Add edge labels %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [] = add_EdgeLabels(Coordinates,Edges,Labels)
% ADD_EDGELABELS Add edge labels to the plot.
%
%   ADD_EDGELABELS(COORDINATES,EDGES,LABELS) adds the edge labels LABELS to
%   the current figure.
%
%   Example:
%
%   add_EdgeLabels(Coordinates,Edges,Labels);

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

```

```
  WEIGHT = 'bold';
  SIZE = 8;
  COLOR = 'k';

  % Add edge labels to the plot

  nEdges = size(Edges,1);
  for i = 1:nEdges
    CoordMid = (Coordinates(Edges(i,1),:)+Coordinates(Edges(i,2),:))/2;
    text(CoordMid(1),CoordMid(2),int2str(Labels(i)), ...
         'HorizontalAlignment','Center', ...
         'VerticalAlignment','Middle', ...
         'Color',COLOR, ...
         'FontWeight',WEIGHT, ...
         'FontSize',SIZE);
  end

return
```

3.2.1.5 plot_Norm_W1F.m

```
function varargout = plot_Norm_W1F(U,Mesh,fig1,fig2,BBox,titstr)
% PLOT_NORM_W1F Plot routine for the modulus of W1F results.
%
%   FIG = PLOT_NORM_W1F(U,MESH) generates a plot of the modulus for the
%   velocity field which is represented by the W1F solution U on the struct
%   MESH and returns the handle FIG to the figure.
%
%   The struct should at least contain the following fields:
%    COORDINATES M-by-2 matrix specifying the vertices of the mesh, where
%                M is equal to the number of vertices contained in the
%                mesh.
%    ELEMENTS    M-by-3 matrix specifying the elements of the mesh, where M
%                is equal to the number of elements contained in the mesh.
%    EDGES       P-by-2 matrix specifying the edges of 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).
%
%   Example:
%
%   fig = plot_Norm_W1F(U,Mesh);

%   Copyright 2005-2006 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

    % Initialize constant

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

```matlab

    % Preallocate memory

    ux = zeros(nElements,1);
    uy = zeros(nElements,1);
    PU = zeros(nCoordinates,1);
    PUx = zeros(nCoordinates,1);
    PUy = zeros(nCoordinates,1);

    % Calculate modulus

    for i = 1:nElements

        vidx = Mesh.Elements(i,:);
        P1 = Mesh.Coordinates(vidx(1),:);
        P2 = Mesh.Coordinates(vidx(2),:);
        P3 = Mesh.Coordinates(vidx(3),:);

        bK = P1;
        BK = [P2-P1;P3-P1];
        TK = transpose(inv(BK));

        % Locate barycenter

        Bar_Node = 1/3*[P1 + P2 + P3];

        % Compute velocity field at barycenters

        eidx = [Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3)) ...
                Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1)) ...
                Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))];

        % Determine edge orientation

        if(Mesh.Edges(eidx(1),1) == vidx(2))
            p1 = 1;
        else
            p1 = -1;
        end

        if(Mesh.Edges(eidx(2),1) == vidx(3))
            p2 = 1;
        else
            p2 = -1;
        end

        if(Mesh.Edges(eidx(3),1) == vidx(1))
            p3 = 1;
        else
            p3 = -1;
        end

        % Compute velocity field at barycenters

```

```
        N = shap_W1F(Bar_Node);
        NS(1:2) = N(1:2)*TK;
        NS(3:4) = N(3:4)*TK;
        NS(5:6) = N(5:6)*TK;

        ux(i) = U(eidx(1))*p1*NS(1) + ...
                U(eidx(2))*p2*NS(3) + ...
                U(eidx(3))*p3*NS(5);

        uy(i) = U(eidx(1))*p1*NS(2) + ...
                U(eidx(2))*p2*NS(4) + ...
                U(eidx(3))*p3*NS(6);

    end

    % Calculate value on each vertice

    Mesh = add_Patches(Mesh);

    for i = 1:nCoordinates
        L_patch = Mesh.AdjElements(i,:);
        loc = find(L_patch>0);
        Eidx = L_patch(loc);
        PUx(i) = sum(ux(Eidx))/Mesh.nAdjElements(i);
        PUy(i) = sum(uy(Eidx))/Mesh.nAdjElements(i);
    end

    PU = sqrt(PUx.^2+PUy.^2);

    % Plot solution

    %%%%%%%%%%Added by me%%%%%%%%%%%%%%%%%%%%
    subplot(fig1);
    hold on
    title(titstr);
    quiver(Mesh.Coordinates(:,1),Mesh.Coordinates(:,2),PUx,PUy,0.75,'b-');
    plot(Mesh.BdEdges_x,Mesh.BdEdges_y,'r-');
    axis(BBox*1.01);
    hold off
    %fig = plot_LFE(PU,Mesh,h);
    plot_LFE(PU,Mesh,fig2);
    % Assign output arguments

    if(nargout > 0)
      varargout{1} = fig;
    end

return
```

3.2.2 Mesh

3.2.2.1 sqr_str_gen.m

```
function Mesh=sqr_str_gen(NREFS)
% Generates a square structured mesh of the unit square

%   Copyright 2005-2005 Patrick Meury & Kah-Ling Sia
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  %NREFS = 4;  % Number of unifrom red refinements

  % Load mesh from file

  Mesh = load_Mesh('Coord_Sqr.dat','Elem_Sqr.dat');

  % Add edge data structure

  Mesh = add_Edges(Mesh);
  Loc = get_BdEdges(Mesh);
  Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
  Mesh.BdFlags(Loc) = -1;

  % Do NREFS uniform refinement steps

  for i = 1:NREFS
    Mesh = refine_REG(Mesh);
  end
```

3.2.2.2 Lshap_str_gen.m

```
function Mesh=Lshap_str_gen(NREFS)
% Generates a triangular structured mesh of a L-shaped domain

%   Copyright 2005-2005 Patrick Meury & Kah-Ling Sia
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  %NREFS = 4;  % Number of unifrom red refinements

  % Load mesh from file

  Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');

  % Add element flags

 % Mesh.ElemFlag = [1 2 3 4]';

  % Add edge data structure
```

```matlab

  Mesh = add_Edges(Mesh);
  Loc = get_BdEdges(Mesh);
  Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
  Mesh.BdFlags(Loc) = -1;

  % Do NREFS uniform refinement steps

  for i = 1:NREFS
    Mesh = refine_REG(Mesh);
  end
```

3.2.2.3 load_Mesh.m

```matlab
function Mesh = load_Mesh(CoordFile,ElemFile)
% LOAD_MESH Load mesh from file.
%
%   MESH = LOAD_MESH(COORDFILE,ELEMFILE) loads a mesh from the files COORDFILE
%   (list of vertices) and ELEMFILE (list of elements).
%
%   The struct MESH contains the followng 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.
%
%   Example:
%
%   Mesh = load_Mesh('Coordinates.dat','Elements.dat');

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Load mesh from files

  Mesh.Coordinates = load_Coordinates(CoordFile);
  Mesh.Elements = load_Elements(ElemFile);

return

%%%

function Coordinates = load_Coordinates(File)
% LOAD_COORDINATES Load vertex coordinates from file.
%
%   COORDINATES = LOAD_COORDINATES(FILE) loads the vertex coordinates from
%   the .dat file FILE.
%
%   Example:
%
%   Coordinates = load_Coordinates('Coordinates.dat');
%
```

```
%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  Coordinates = load(File);
  Coordinates(:,1) = [];

return

%%%

function Elements = load_Elements(File)
% LOAD_ELEMENTS Load elements from a file.
%
%   ELEMENETS = LOAD_ELEMENTS(FILE) load the elements of a mesh from the
%   .dat file FILE.
%
%   Example:
%
%   Elements = load_Elements('Elements.dat');
%

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  Elements = load(File);
  Elements(:,1) = [];

return
```

3.2.2.4 refine_REG.m

```
function  New_Mesh = refine_REG(Old_Mesh,varargin)
% REFINE_REG Regular refinement.
%
%   MESH = REFINE_REG(MESH) performs one regular red refinement step with the
%   struct MESH.
%
%   MESH = REFINE_REG(MESH,DHANDLE) performs one regular red refinement step
%   with the struct MESH. During red refinement the signed distance function
%   DHANDLE (function handle/inline onject) is used to project the new vertices
%   on the boundary edges onto the boundary of the domain.
%
%   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 or N-by-4 matrix specifying the elements of the mesh.
%    EDGES        P-by-2 matrix specifying the 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).
%
%   MESH = REFINE_REG(MESH,DHANDLE,DPARAM) also handles the variable argument
%   list DPARAM to the signed distance function DHANDLE.
%
%   Example:
%
%   Mesh = refine_REG(Mesh,@dist_circ,[0 0],1);
%
%   See also ADD_EDGES.

%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

nCoordinates = size(Old_Mesh.Coordinates,1);
[nElements,nVert] = size(Old_Mesh.Elements);
nEdges = size(Old_Mesh.Edges,1);
nBdEdges = size(find(Old_Mesh.BdFlags < 0),1);

% Extract input arguments

if(nargin > 1)
DHandle = varargin{1};
DParam = varargin(2:nargin-1);
else
DHandle = [];
end

% Red refinement for triangular or triangular elements

if(nVert == 3)

% Preallocate memory

New_Mesh.Coordinates = zeros(nCoordinates+nEdges,2);
New_Mesh.Elements = zeros(4*nElements,3);
New_Mesh.BdFlags = zeros(2*nEdges+3*nElements,1);
if(isfield(Old_Mesh,'ElemFlag'))
    New_Mesh.ElemFlag = zeros(4*nElements,1);
end

% Do regular red refinement

New_Mesh.Coordinates(1:nCoordinates,:) = Old_Mesh.Coordinates;
Bd_Idx = 0;
Aux = zeros(nBdEdges,4);
for i = 1:nElements

    % Get vertex numbers of the current element

```

```
    i1 = Old_Mesh.Elements(i,1);
    i2 = Old_Mesh.Elements(i,2);
    i3 = Old_Mesh.Elements(i,3);

    % Compute vertex numbers of new vertices localized on edges

    j1 = nCoordinates+Old_Mesh.Vert2Edge(i2,i3);
    j2 = nCoordinates+Old_Mesh.Vert2Edge(i3,i1);
    j3 = nCoordinates+Old_Mesh.Vert2Edge(i1,i2);

    % Generate new elements

    New_Mesh.Elements(4*(i-1)+1,:) = [i1 j3 j2];
    New_Mesh.Elements(4*(i-1)+2,:) = [j3 i2 j1];
    New_Mesh.Elements(4*(i-1)+3,:) = [j2 j1 i3];
    New_Mesh.Elements(4*(i-1)+4,:) = [j1 j2 j3];

    % Generate new vertex on edge 1, project to boundary if necessary

    BdFlag_1 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i2,i3));
    if(BdFlag_1 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i2,:)-...
                Old_Mesh.Coordinates(i3,:));
            x = 1/2*(Old_Mesh.Coordinates(i2,:)+Old_Mesh.Coordinates(i3,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j1,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i2,:)...
                +Old_Mesh.Coordinates(i3,:));
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_1 i2 j1 i3];
    else
        New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i2,:)...
            +Old_Mesh.Coordinates(i3,:));
    end

    % Generate new vertex on edge 2, project to boundary if necessary

    BdFlag_2 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i3,i1));
    if(BdFlag_2 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i3,:)-...
                Old_Mesh.Coordinates(i1,:));
            x = 1/2*(Old_Mesh.Coordinates(i3,:)+Old_Mesh.Coordinates(i1,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j2,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i3,:)...
```

```
                +Old_Mesh.Coordinates(i1,:));
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_2 i3 j2 i1];
    else
        New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+Old_Mesh.Coordinates
    end

    % Generate new vertex on edge 3, project to boundary if necessary

    BdFlag_3 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i1,i2));
    if(BdFlag_3 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i1,:)-Old_Mesh.Coordinates(i2,:));
            x = 1/2*(Old_Mesh.Coordinates(i1,:)+Old_Mesh.Coordinates(i2,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j3,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i1,:)...
                +Old_Mesh.Coordinates(i2,:));
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_3 i1 j3 i2];
    else
        New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i1,:)+Old_Mesh.Coordinates
    end

    % Update element flag if defined

    if(isfield(Old_Mesh,'ElemFlag'))
        New_Mesh.ElemFlag(4*(i-1)+1) = Old_Mesh.ElemFlag(i);
        New_Mesh.ElemFlag(4*(i-1)+2) = Old_Mesh.ElemFlag(i);
        New_Mesh.ElemFlag(4*(i-1)+3) = Old_Mesh.ElemFlag(i);
        New_Mesh.ElemFlag(4*(i-1)+4) = Old_Mesh.ElemFlag(i);
    end
end

% Add edges to new mesh

New_Mesh = add_Edges(New_Mesh);

% Update boundary flags

for i = 1:nBdEdges
    New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,2),Aux(i,3))) = Aux(i,1);
    New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,3),Aux(i,4))) = Aux(i,1);
end

else

% Preallocate memory

```

```
New_Mesh.Coordinates = zeros(nCoordinates+nEdges,2);
New_Mesh.Elements = zeros(4*nElements,4);
New_Mesh.BdFlags = zeros(2*nEdges+4*nElements,1);
if(isfield(Old_Mesh,'ElemFlag'))
    New_Mesh.ElemFlag = zeros(4*nElements,1);
end

% Do regular red refinement

New_Mesh.Coordinates(1:nCoordinates,:) = Old_Mesh.Coordinates;
Bd_Idx = 0;
Aux = zeros(nBdEdges,4);
for i = 1:nElements

    % Get vertex numbers of the current element

    i1 = Old_Mesh.Elements(i,1);
    i2 = Old_Mesh.Elements(i,2);
    i3 = Old_Mesh.Elements(i,3);
    i4 = Old_Mesh.Elements(i,4);

    % Compute vertex numbers of new vertices localized on edges

    j1 = nCoordinates+Old_Mesh.Vert2Edge(i1,i2);
    j2 = nCoordinates+Old_Mesh.Vert2Edge(i2,i3);
    j3 = nCoordinates+Old_Mesh.Vert2Edge(i3,i4);
    j4 = nCoordinates+Old_Mesh.Vert2Edge(i4,i1);

    % Compute vertex number of new vertex in the interior of each element

    jc = nCoordinates+nEdges+i;

    % Generate new elements

    New_Mesh.Elements(4*(i-1)+1,:) = [i1 j1 jc j4];
    New_Mesh.Elements(4*(i-1)+2,:) = [j1 i2 j2 jc];
    New_Mesh.Elements(4*(i-1)+3,:) = [j4 jc j3 i4];
    New_Mesh.Elements(4*(i-1)+4,:) = [jc j2 i3 j3];

    % Generate new vertex on edge 1, project to boundary if necessary

    BdFlag_1 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i1,i2));
    if(BdFlag_1 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i2,:)-...
                Old_Mesh.Coordinates(i1,:));
            x = 1/2*(Old_Mesh.Coordinates(i1,:)+Old_Mesh.Coordinates(i2,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j1,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i1,:)+...
                Old_Mesh.Coordinates(i2,:));
```

```
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_1 i1 j1 i2];
    else
        New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i1,:)+...
            Old_Mesh.Coordinates(i2,:));
    end

    % Generate new vertex on edge 2, project to boundary if necessary

    BdFlag_2 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i2,i3));
    if(BdFlag_2 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i3,:)-...
                Old_Mesh.Coordinates(i2,:));
            x = 1/2*(Old_Mesh.Coordinates(i2,:)+Old_Mesh.Coordinates(i3,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j2,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i2,:)...
                +Old_Mesh.Coordinates(i3,:));
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_2 i2 j2 i3];
    else
        New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i2,:)...
            +Old_Mesh.Coordinates(i3,:));
    end

    % Generate new vertex on edge 3, project to boundary if necessary

    BdFlag_3 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i3,i4));
    if(BdFlag_3 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i4,:)-...
                Old_Mesh.Coordinates(i3,:));
            x = 1/2*(Old_Mesh.Coordinates(i3,:)+Old_Mesh.Coordinates(i4,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j3,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+...
                Old_Mesh.Coordinates(i4,:));
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_3 i3 j3 i4];
    else
        New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+...
            Old_Mesh.Coordinates(i4,:));
    end

```

```
    % Generate new vertex on egde 4, prohject to boundary if necessary

    BdFlag_4 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i4,i1));
    if(BdFlag_4 < 0)
        if(~isempty(DHandle))
            DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i1,:)-...
                Old_Mesh.Coordinates(i4,:));
            x = 1/2*(Old_Mesh.Coordinates(i4,:)+Old_Mesh.Coordinates(i1,:));
            dist = feval(DHandle,x,DParam{:});
            grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
                feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
            New_Mesh.Coordinates(j4,:) = x-dist*grad_dist;
        else
            New_Mesh.Coordinates(j4,:) = 1/2*(Old_Mesh.Coordinates(i4,:)+...
                Old_Mesh.Coordinates(i1,:));
        end
        Bd_Idx = Bd_Idx+1;
        Aux(Bd_Idx,:) = [BdFlag_4 i4 j4 i1];
    else
        New_Mesh.Coordinates(j4,:) = 1/2*(Old_Mesh.Coordinates(i4,:)+...
            Old_Mesh.Coordinates(i1,:));
    end

    % Generate new vertex in the interior

    New_Mesh.Coordinates(jc,:) = 1/4*(New_Mesh.Coordinates(j1,:)+...
        New_Mesh.Coordinates(j2,:)+New_Mesh.Coordinates(j3,:)+...
        New_Mesh.Coordinates(j4,:));

    % Update element flag if defined

    if(isfield(Old_Mesh,'ElemFlag'))
        New_Mesh.ElemFlag(4*(i-1)+1) = Old_Mesh.ElemFlag(i);
        New_Mesh.ElemFlag(4*(i-1)+2) = Old_Mesh.ElemFlag(i);
        New_Mesh.ElemFlag(4*(i-1)+3) = Old_Mesh.ElemFlag(i);
        New_Mesh.ElemFlag(4*(i-1)+4) = Old_Mesh.ElemFlag(i);
    end
end

% Add edges to new mesh

New_Mesh = add_Edges(New_Mesh);

% Update boundary flags

for i = 1:nBdEdges
    New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,2),Aux(i,3))) = Aux(i,1);
    New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,3),Aux(i,4))) = Aux(i,1);
end
end

return
```

3.2.3 Viewing results

3.2.3.1 replay.m

```

% Rendering of nodal/edge element solution for
% transient Maxwell problem in a cavity

% Initialize constants

InitREF = 2; % size of the Initial Mesh
NREFSs = 5;  % Number of unifrom mesh refinements
finaltime = 3;
timestep = 0.01;
step=1;
makemovie =0; % set to 1 to make avi files
rect=[100 100 1024 768];   % movie size (not length)

% Generate initial meshes, where the meshwidth depends on InitREF
%Square mesh
MeshS=sqr_str_gen(InitREF);
%Add to the mesh some useful information to handle edge elements
MeshS.ElemFlag = ones(size(MeshS.Elements,1),1);
MeshS = add_Edges(MeshS);
LocS = get_BdEdges(MeshS);
MeshS.BdFlags = zeros(size(MeshS.Edges,1),1);
MeshS.BdFlags(LocS) = -1;

%L-shaped mesh
MeshL=Lshap_str_gen(InitREF);
%Add to the mesh some useful information to handle edge elements
MeshL.ElemFlag = ones(size(MeshL.Elements,1),1);
MeshL = add_Edges(MeshL);
LocL = get_BdEdges(MeshL);
MeshL.BdFlags = zeros(size(MeshL.Edges,1),1);
MeshL.BdFlags(LocL) = -1;

% Do NREFS uniform refinement steps

for i = 1:NREFSs



    % For the square mesh
    % Refine Mesh
    MeshS = refine_REG(MeshS);
    % plot it
    plot_Mesh(MeshS,'as')

    % Write some necesary information in the mesh
    [MeshS.BdEdges_x MeshS.BdEdges_y]=dataBoundaryPlot(MeshS);
    MeshS=setBdFlags(MeshS);
```

```
Loc = get_BdEdges(MeshS);
NDofs = [2*find(MeshS.VertBdFlags(:,1) == 0); 2*find(MeshS.VertBdFlags(:,2) == 0)-1];
DEdges = Loc(MeshS.BdFlags(Loc) == -1);
EDofs = setdiff(1:size(MeshS.Edges,1),DEdges);

%Loading Data
Sq_str1=['Square' int2str(i)];
Sq_str=['Square' int2str(i) '_res'];
load(Sq_str);
N=length(times);

if (size(sol_v,2) ~= N), error('Wrong number of samples in sol_v'); end
if (size(sol_e,2) ~= N), error('Wrong number of samples in sol_e'); end

if(makemovie) moviename=[Sq_str '.avi'];
    aviobj = avifile(moviename,'fps',10); end;
figno=figure('position',rect);
set(gcf,'nextplot','replace');
axis off;
if(makemovie) Sqr = getframe(figno); aviobj = addframe(aviobj,Sqr); end;

for j=1:step:N/30

    plotiterate1(MeshS,sol_e(:,j),sol_v(:,j),times(j),figno,NDofs,EDofs);

    % add Energy information
    [a,I]=min(abs(times(j)-en(:,1)));
    subplot(2,2,3,'Parent',figno);
    title(sprintf('Time = %f     Etot = %f ',times(j),en(I,4)+en(I,5) ));

    subplot(2,2,4,'Parent',figno);
    title(sprintf('Time = %f     Etot = %f ',times(j),en(I,2)+en(I,3)));
    if(makemovie)Sqr = getframe(gcf); aviobj = addframe(aviobj,Sqr); end;

end
if(makemovie)aviobj = close(aviobj); end;


%plot energy evolution in time
figure; clf;
subplot(1,2,1);
plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
legend('Nodal scheme','Edge elements');
title([Sq_str1,': Electric energy']);
xlabel('time');
subplot(1,2,2);
plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
legend('Nodal scheme','Edge elements');
title([Sq_str1,': Magnetic energy']);
xlabel('time');
%   Sq_str1=['../Bericht/En_smoo_' Sq_str1 '.eps'];
%   saveas(gcf,Sq_str1,'psc2');
drawnow;
clear en sol_v sol_e times;
```

```
% end for the square mesh


%For the L-mesh
% Refine mesh
MeshL = refine_REG(MeshL);
plot_Mesh(MeshL,'as')

[MeshL.BdEdges_x MeshL.BdEdges_y]=dataBoundaryPlot(MeshL);
MeshL=setBdFlags(MeshL);
Loc = get_BdEdges(MeshL);
NDofs = [2*find(MeshL.VertBdFlags(:,1) == 0); 2*find(MeshL.VertBdFlags(:,2) == 0)-1];
DEdges = Loc(MeshL.BdFlags(Loc) == -1);
EDofs = setdiff(1:size(MeshL.Edges,1),DEdges);
%Loading Data

L_str1=['Lshape' int2str(i)];
L_str=['Lshape' int2str(i) '_res'];
load(L_str);
N=length(times);

if(makemovie) moviename=[L_str '.avi']; aviobj = avifile(moviename,'fps',10); end;
figno=figure('position',rect);
set(gcf,'nextplot','replace');
axis off;
if(makemovie)L_shape = getframe(figno); aviobj = addframe(aviobj,L_shape);end

for j=1:step:N/10
    plotiterate1(MeshL,sol_e(:,j),sol_v(:,j),times(j),figno,NDofs,EDofs);

    % add Energy information
    [a,I]=min(abs(times(j)-en(:,1)));
    subplot(2,2,3,'Parent',figno);
    title(sprintf('Time = %f     Etot = %f ',times(j),en(I,4)+en(I,5) ));

    subplot(2,2,4,'Parent',figno);
    title(sprintf('Time = %f     Etot = %f ',times(j),en(I,2)+en(I,3)));

    if(makemovie)L_shape = getframe(figno); aviobj = addframe(aviobj,L_shape); end;
end
if(makemovie) aviobj = close(aviobj); end;

% Actualize energy plot
figure; clf;
subplot(1,2,1);
plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
legend('Nodal scheme','Edge elements');
title([L_str1,': Electric energy']);
xlabel('time');
subplot(1,2,2);
plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
legend('Nodal scheme','Edge elements');
title([L_str1,': Magnetic energy']);
xlabel('time');
```

```
    drawnow;
    clear en sol_v sol_e times;
%     L_str1=['../Bericht/En_smoo_' L_str1 '.eps'];
%     saveas(gcf,L_str1,'psc2');
end
```

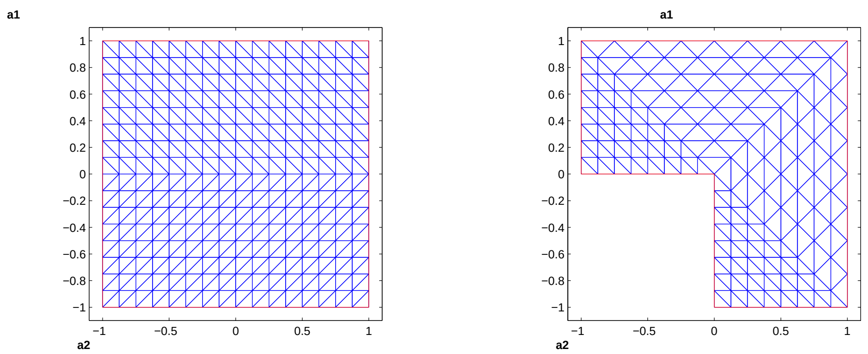
Chapter 4

Numerical Experiments

Now we describe the experiments performed with our implementation. Approximations to the electrical fields solving (2.12) and (2.9) were computed using for the starting value $\mathbf{E}_0$ both, singular and smooth functions. Convergence could be observed in all cases, excluding the case where the electrical field was computed on an L-shaped domain using a singular starting function. Next we present the meshes we used.

4.1 Domain

We tried several mesh-types available in "LehrFem", though as the results were similar, and the mesh-refinement's time using regular or large-edge-bisection (LEB) algorithms are considerable, we decided to use simple structured meshes. The initial meshes are shown in Figure 4.1.



(a) Square: $h_1 = 0.17678$ (b) L-shape: $h_1 = 0.25000$

Figure 4.1: Initial meshes.

			Vertices	Edges	Elements
h_1	=	0.1768	289	800	512
h_2	=	0.0884	1089	3136	2048
h_3	=	0.0442	4225	12416	8192
h_4	=	0.0221	16641	49408	32768
h_5	=	0.0110	66049	197120	131072

Table 4.1: Attributes of the square meshes

			Vertices	Edges	Elements
h_1	=	0.250	153	408	256
h_2	=	0.125	561	1584	1024
h_3	=	0.0625	2145	6240	4096
h_4	=	0.0312	8385	24768	16384
h_5	=	0.0156	33153	98688	65536

Table 4.2: Attributes of the L-shaped meshes

The Meshes were refined four times. Table 4.1 and 4.2 shows the size of the meshes we used.

4.2 Starting Conditions

The starting condition is a very important topic in this work. Recall that we require for the initial electrical field $\operatorname{div} E_0 = 0$, furthermore the starting condition will determine whether the solution converges or not. Let us first consider the L-shaped domain.

4.2.1 Singular Starting Conditions[1]

In the domain $\Omega =]-1,1[\times]0;1[\cup[0;1]\times]-1;0[$ we choose an initial field that contains a singular contribution in the point $(0,0)$, to this end let us define

$$u(r,\varphi) := r^{\pi/\omega}\sin(\frac{\pi}{\omega}\varphi) \quad r \geq 0, 0 < \varphi < \frac{3}{2}\pi \,.$$

For the L-shaped domain we choose $\omega = \frac{3}{2}\pi$. Note also that the laplacian of this function is singular. The associated vector field reads

$$\mathbf{grad}\, u(r,\varphi) = \frac{u}{r} \cdot \vec{\mathbf{e}}_r + \frac{1}{r}\frac{u}{\varphi} \cdot \vec{\mathbf{e}}_\varphi \,.$$

[1] taked from [3]

Then, we introduce

$$p(r,\varphi) := r^{\pi/\omega} \cos(\frac{\pi}{\omega}\varphi) \ ,$$

and see

$$\left.\begin{array}{rcl} \frac{p}{r} & = & \frac{1}{r}\frac{u}{\varphi} \\ -\frac{1}{r}\frac{p}{\varphi} & = & \frac{u}{r} \end{array}\right\} \quad \Rightarrow \quad \mathbf{curl}_{2D}\, p := \frac{p}{r}\cdot \vec{\mathbf{e}}_\varphi - \frac{1}{r}\frac{p}{\varphi}\cdot \vec{\mathbf{e}}_r = \mathbf{grad}\, u \ .$$

Now, let us introduce the cut-off function,

$$g(t) := \begin{cases} 1 & \text{if } 0 \le |t| \le \frac{1}{2} \ , \\ \sin^2(\pi\, t) & \text{if } \frac{1}{2} \le |t| \le 1 \end{cases} \quad \Rightarrow \quad g'(t) = \begin{cases} 0 & \text{if } 0 \le |t| \le \frac{1}{2} \ , \\ \pi \sin(2\pi\, t) & \text{if } \frac{1}{2} \le |t| \le 1 \end{cases} \ .$$

We write $f(x,y) = g(x)g(y)$ and define

$$\mathbf{E}^0(x,y) := \mathbf{curl}_{2D}\,(p(r,\varphi)\cdot f(x,y)) = f\,\mathbf{curl}_{2D}\, p + p\,\mathbf{curl}_{2D}\, f \ . \tag{4.1}$$

Note that

$$\mathbf{curl}_{2D}\, f(x,y) = \begin{pmatrix} -g(x)g'(y) \\ g'(x)g(y) \end{pmatrix} = \begin{cases} \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \text{if } 0 \le |x|,|y| \le \frac{1}{2} \ , \\ \begin{pmatrix} 0 \\ \pi\sin(2\pi\, x) \end{pmatrix} & \text{if } 0 \le |y| \le \frac{1}{2}, |x| \ge \frac{1}{2} \ , \\ \begin{pmatrix} -\pi\sin(2\pi\, y) \\ 0 \end{pmatrix} & \text{if } 0 \le |x| \le \frac{1}{2}, |y| \ge \frac{1}{2} \ , \\ \begin{pmatrix} -\pi\sin^2(\pi x)\sin(2\pi y) \\ \pi\sin(2\pi x)\sin^2(\pi y) \end{pmatrix} & \text{if } \frac{1}{2} \le |x|,|y| \le 1 \end{cases} \ .$$

Figure 4.2 shows the discretization $\vec{\mathbf{E}}^0$ of this field using $\mathcal{E}_{h_1}$ (left), and $\mathcal{N}_{h_1}$ (right).

Considering the square domain $\Omega = [-1,1]\times[-1,1]$, we can choose either $\omega = \frac{\pi}{2}$ or $\omega = 2\pi$, both gives a square. We have chosen the first as the graphical representation is more appealing. Its discretization is shown in Figure 4.3

4.2.2 Smooth Starting Conditions

For smooth starting conditions we expect convergence in all treated cases. We choose the following function

$$\mathbf{E}_0 = \begin{pmatrix} \sin(\pi x_1)\sin(\pi x_2) \\ \sin(\pi x_1)\sin(\pi x_2) \end{pmatrix} \ , \tag{4.2}$$

which fulfils our divergence-free requirement. Plots of the discretization to this field are represented in Figure 4.5 and 4.4.
The condition that ensures that $\vec{\mathbf{E}}^0$ is divergence-free on the discrete level considered in (2.15), could not be applied to (2.12) using edge elements. The correction term to the initial field was almost as big as the initial field itself. The reason for this behaviour and the way how it can be corrected is unknown to us.

4.3 Time Stepping

In Chapter 2.4 we have already mentioned that the time-step needs to fulfil a CFL-condition. In our case we obtain that the time step

$$\tau \leq \sqrt{\frac{2}{\underbrace{\lambda_{\max}}_{\approx \frac{C_1}{|\mathbf{T}_{h_i}|^2}}}} = C_2|\mathbf{T}_{h_i}| \leq C_3 h_i^2 \leq C\left(2^{-(i-1)}h_1\right)^2 \leq C\frac{h_1^2}{16}2^{-i+1} \leq 2^{-i+1}0.001\,,$$

for suitable constants C_1, C_2, C_3, C, and $i = 1, \ldots, 5$. In the implementation we choose the time-step in this sense. The constant time step= 0.001 works with the used meshes for 5 refinements. For further refinements this constant should be reduced. Note also that this time step is not efficient for the smaller meshes. The reason why we use it anyway, is that we want to evaluate the time evolution at some fix times on all meshes. In this way comparisons between results can be carried out for different meshes and at different times.

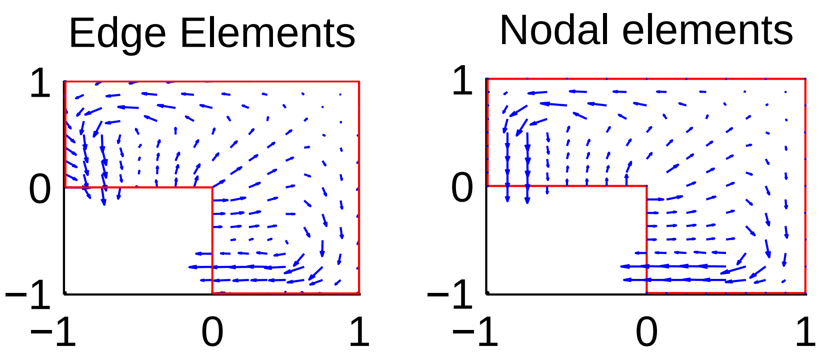


Figure 4.2: Discrete singular initial field on the L-shaped domain

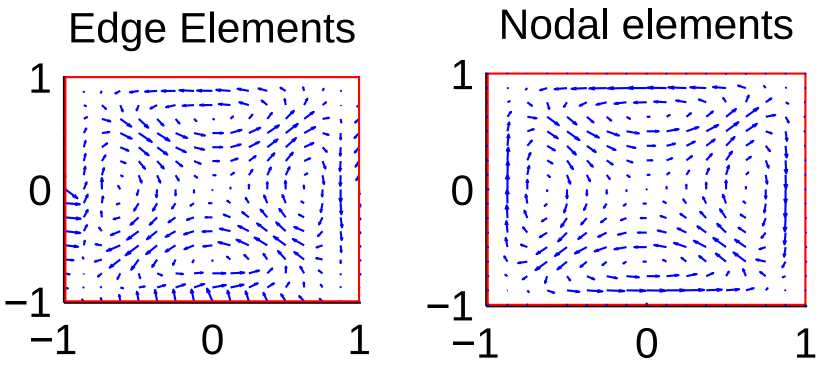


Figure 4.3: Discrete singular initial field on the square domain

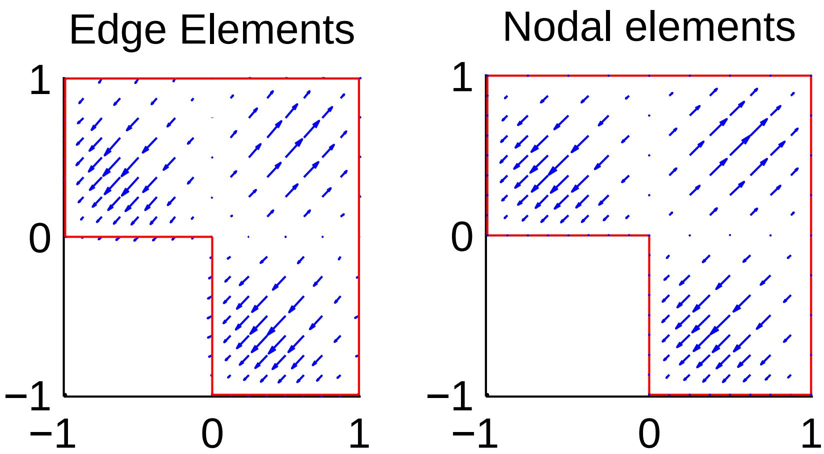


Figure 4.4: Discrete smooth initial field on the L-shaped domain

4.4 Results

First we want to present the energy behaviour of the approximations computed with (2.15) and (2.13) for every mesh. The energy is computed using

$$\vec{\mathbf{E}^n}^t M \vec{\mathbf{E}}^n \quad \text{for} \quad \text{the electrical energy, and}$$
$$\vec{\mathbf{E}^n}^t A \vec{\mathbf{E}}^n \quad \text{for} \quad \text{the magnetic energy,}$$

for the discretization using $\mathcal{E}_h$. In the nodal case, we use $\hat{M}$ and $\hat{A}$ instead of M and A.

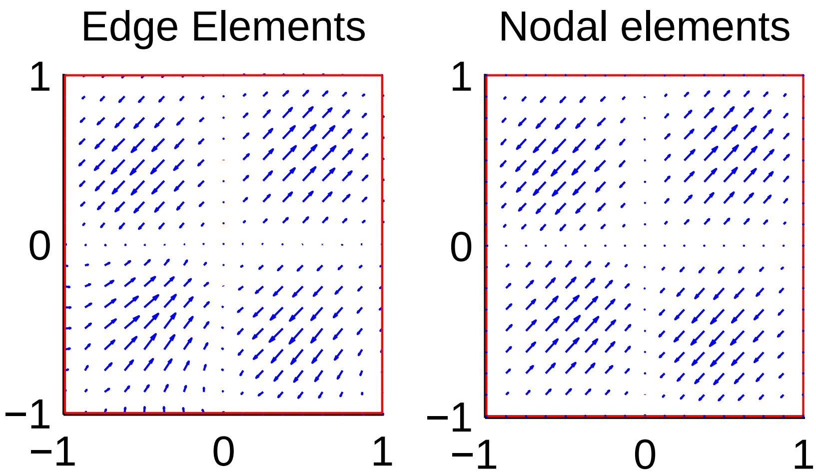
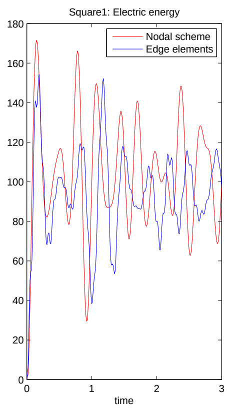
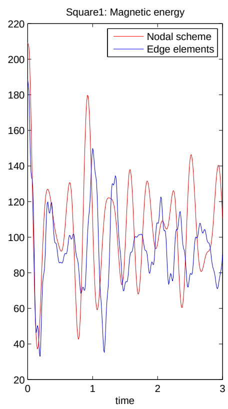
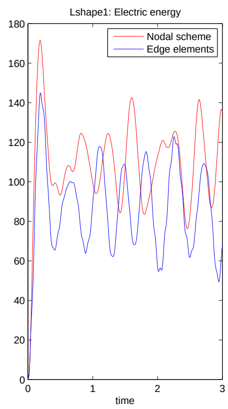
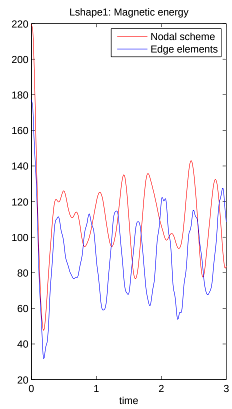
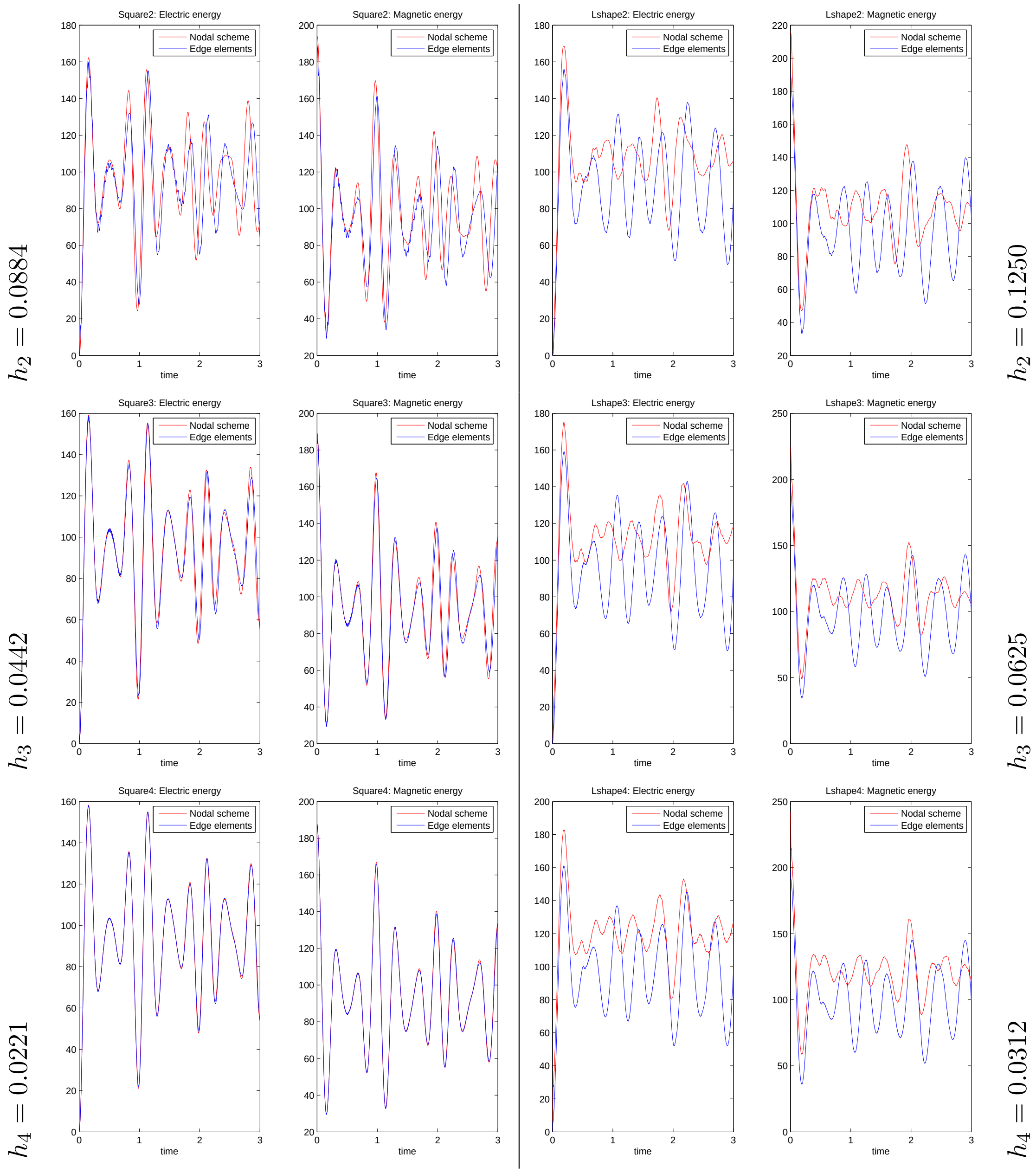


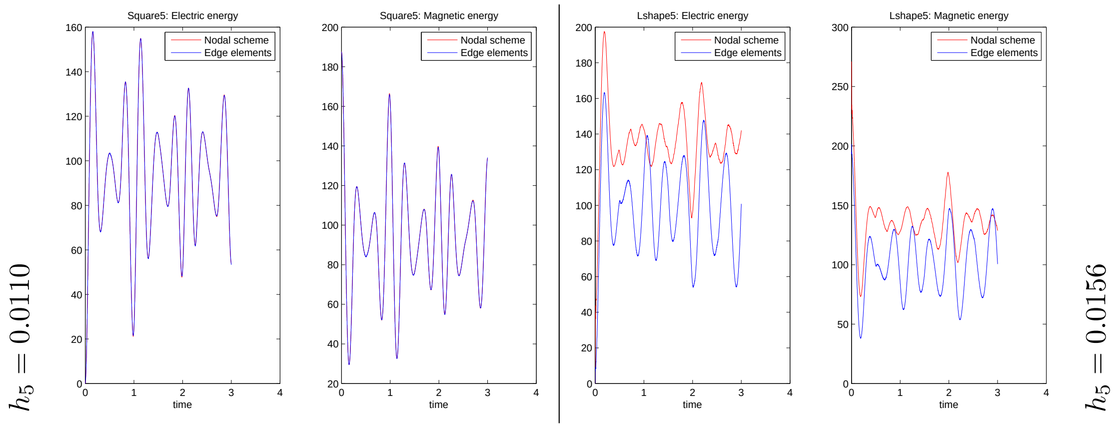
Figure 4.5: Discrete smooth initial field on the square domain

4.4.1 Energy Behaviour Using a Singular Initial Function

The results are listed in the table below. Note how the graphs of the energy evolution overlap for an square domain whereas the graphs for the L-shaped domain do not converge at all

	Square Mesh		L-shaped Mesh		
$h_1 = 0.1768$					$h_1 = 0.2500$

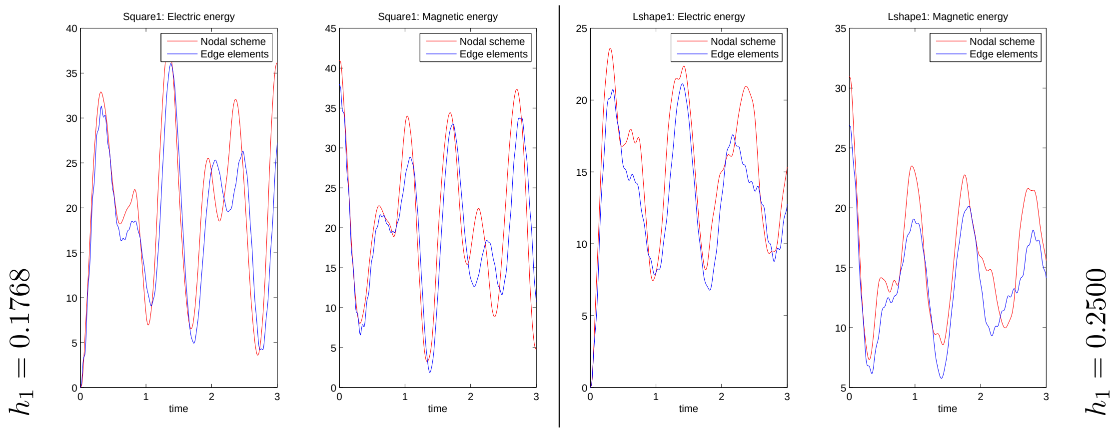

Square2: Electric energy
Square2: Magnetic energy
Lshape2: Electric energy
Lshape2: Magnetic energy
Square3: Electric energy
Square3: Magnetic energy
Lshape3: Electric energy
Lshape3: Magnetic energy
Square4: Electric energy
Square4: Magnetic energy
Lshape4: Electric energy
Lshape4: Magnetic energy
Nodal scheme
Edge elements
time
$h_2 = 0.0884$
$h_3 = 0.0442$
$h_4 = 0.0221$
$h_2 = 0.1250$
$h_3 = 0.0625$
$h_4 = 0.0312$

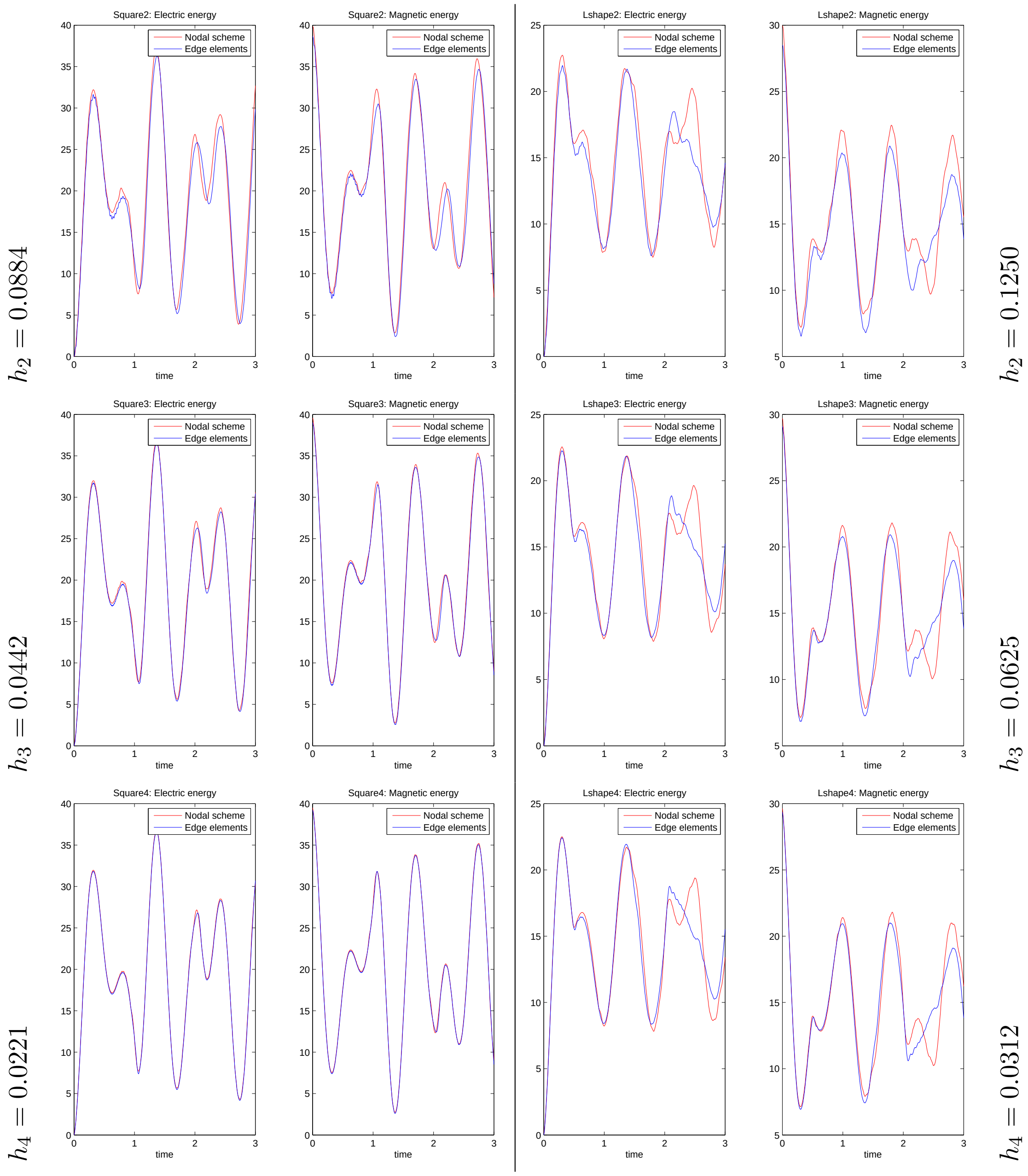


4.4.2 Energy Behaviour Using a Smooth Initial Function

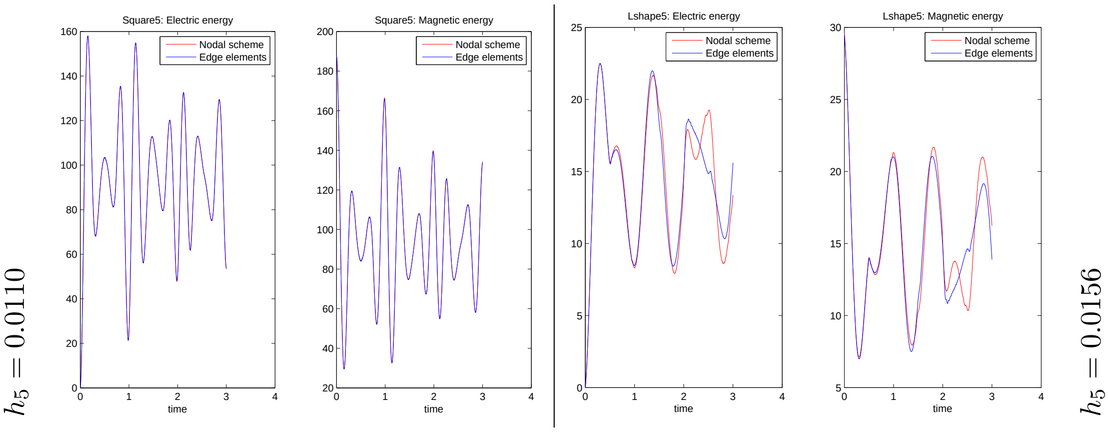
In this case we observe that the graphs for the square mesh and for the L-shaped mesh seem to converge at least for the time $t \leq 2$. In the L-shaped mesh after time $t = 2$ we observe a difference between the Edge and nodal discretization, the nature of this behaviour is not clear to us.

Square Mesh	L-shaped Mesh





Square2: Electric energy
Square2: Magnetic energy
Lshape2: Electric energy
Lshape2: Magnetic energy
Square3: Electric energy
Square3: Magnetic energy
Lshape3: Electric energy
Lshape3: Magnetic energy
Square4: Electric energy
Square4: Magnetic energy
Lshape4: Electric energy
Lshape4: Magnetic energy
Nodal scheme
Edge elements
time
$h_2 = 0.0884$
$h_3 = 0.0442$
$h_4 = 0.0221$
$h_2 = 0.1250$
$h_3 = 0.0625$
$h_4 = 0.0312$



4.4.3 Convergence results

The convergence rate of our approximations can be usually obtained comparing the approximations with the corresponding exact solution evaluated at the final-time T. Furthermore the time-step should be carefully determined, as the error behaves additive, i.e. $\mathcal{O}(h^n + \tau^m)$ where h is the mesh width, n depends on the smoothness of the polynomials, on the dimension of the underlying problem and on the grad of the shape functions (Statement of a suitable approximation proposition). τ is the time-step and m depends on the numerical scheme used to solve the ODE, in the case of leapfrog $m = 4$.

As we only want to answer the question if our approximations converges or not, we proceed simply comparing the approximations obtained for $h_1, \ldots, h_4$ with the approximation obtained for h_5.

$$\text{error}_i := \left\| \mathbf{E}^T_{h_5} - \mathbf{E}^T_{h_i} \right\|_2 \quad \text{for } i = 1, \ldots, 4\,.$$

The results are shown in Figure 4.6. Note that the discretization with nodal elements on the L-shaped domain using singular initial conditions seems to converge, however the solution to which i converges is not the right one.

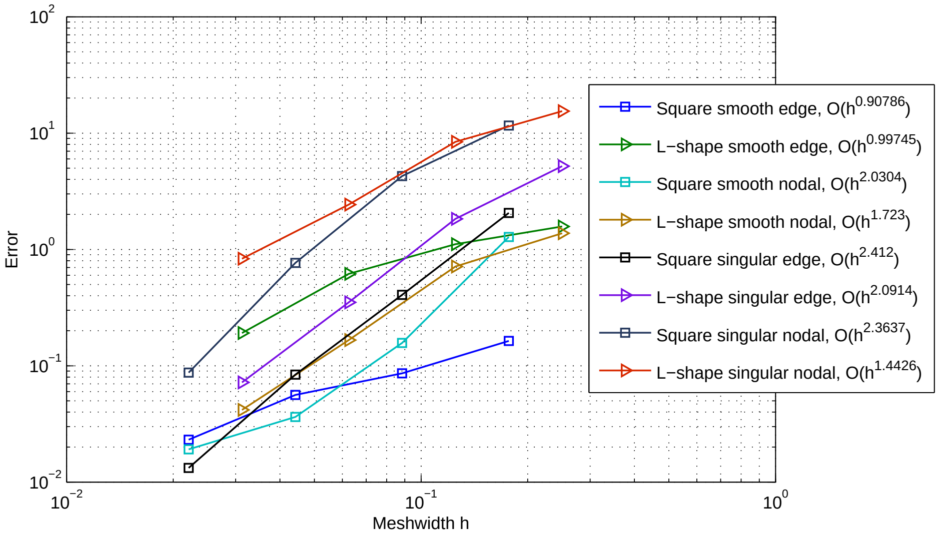


Figure 4.6: Convergence comparison between the approximations using different domains, elements and initial conditions

Bibliography

[1] R. Hiptmair. *Finite Elements in computational electromagnetism*, Universität Tübingen, Acta Numerica (2002), pp.237-339

[2] D. Sun, J. Manges, X. Yuan, Z. Cendes. *Spurious Modes in Finite-Element Methods*, IEEE Antennas and Propagation Magazine, Vol. 37, No. 5, October 1995.

[3] D. Arnold, R. Hiptmair. *Discretizing transient Maxwell's equations*,