

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

I	Abbreviations	3
1	Introduction	4
2	Theoretical aspects	6
2.1	Boundary Value Problem	6
2.2	Finite Elements	8
2.3	Regularization	10
2.3.1	grad-div Regularization	11
2.3.2	Discrete Regularization	12
2.4	Time stepping	13
3	Implementation	15
3.1	Code	15
3.1.1	Main Routines	16
3.1.1.1	run.m	16
3.1.1.2	leapfrog.m	18
3.1.1.3	assemMat_LFE.m	23
3.1.1.4	MASS_Lump_LFE.m	25
3.1.1.5	P706.m	25
3.1.1.6	shap_W1F.m	26
3.1.1.7	assemMat_W1F.m	27
3.1.1.8	MASS_W1F.m	29
3.1.1.9	STIMA_Curl_W1F.m	30
3.1.1.10	assemMat_WRegW1F_2.m	31
3.1.1.11	grad_shap_LFE.m	33
3.1.1.12	STIMA_WReg_W1Fb.m	34
3.1.1.13	assemLoad_W1F.m	35

3.1.1.14	shap_LFE2.m	37
3.1.1.15	assemMat_LFE3.m	37
3.1.1.16	MASS_LFE2.m	39
3.1.1.17	STIMA_Curl_LFE2.m	40
3.1.1.18	STIMA_Reg_LFE2.m	41
3.1.1.19	assemLoad_LFE3.m	41
3.2	Initial Value	43
3.2.0.20	initL.m	43
3.2.0.21	initSq.m	44
3.2.1	Plotting	45
3.2.1.1	plotfield1.m	45
3.2.1.2	plotiteratel.m	45
3.2.1.3	plot_LFE.m	46
3.2.1.4	plot_Mesh.m	48
3.2.1.5	plot_Norm.W1F.m	53
3.2.2	Mesh	55
3.2.2.1	sqr_str_gen.m	55
3.2.2.2	Lshap_str_gen.m	56
3.2.2.3	load_Mesh.m	57
3.2.2.4	refine_REG.m	58
3.2.3	Viewing results	65
3.2.3.1	replay.m	65
4	Numerical Experiments	69
4.1	Domain	69
4.2	Starting Conditions	70
4.2.1	Singular Starting Conditions	70
4.2.2	Smooth Starting Conditions	71
4.3	Time Stepping	72
4.4	Results	73
4.4.1	Energy Behaviour Using a Singular Initial Function . .	74
4.4.2	Energy Behaviour Using a Smooth Initial Function . .	76
4.4.3	Convergence results	78

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

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

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

$$\operatorname{curl} \mathbf{E} = -\frac{d}{dt} \mathbf{B} \quad (1.1)$$

$$\operatorname{curl} \mathbf{H} = \frac{d}{dt} \mathbf{D} + \mathbf{J} \quad (1.2)$$

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 and for 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)

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

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

$$\operatorname{curl} \operatorname{curl} \mathbf{E} = -\frac{d^2}{dt^2} \mathbf{E}. \quad (2.1)$$

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

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

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

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

where $\Omega \in \mathbb{R}^2$ is a bounded domain, $T \in \mathbb{R}_+$ and $\mathbf{E}_0 \in \mathbf{H}_0(\mathbf{curl}; \Omega) := \{\mathbf{v} \in \mathbf{L}^2(\Omega) : \mathbf{curl}_{2D} \mathbf{v} \in \mathbf{L}^2(\Omega), \mathbf{v} \times \mathbf{n} = 0\}$. 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.

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

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 $\mathbf{v}, \mathbf{E} \in \mathbf{H}_0(\mathbf{curl}; \Omega)$.

Proof. take $\nu := \mathbf{curl}_{2D} \mathbf{E}$ 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 $\eta, \omega \in \mathcal{DF}^{1,1}(\Omega)$.

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

From Stokes Theorem we obtain

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

To see that the right-hand-side (rhs.) of the last term vanishes, recall that $\gamma_t \mathbf{v} = 0$ 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 $\int_{\partial\Omega} \nu v_1 dx_1 + \nu v_2 dx_2 = \int_{\partial\Omega} \mathbf{curl}_{2D} \mathbf{E} \wedge \mathbf{v} = 0$ \square

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

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

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 and 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 $\mathbf{a}_j^i \in \Omega$, $j = 1, 2, 3$, 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

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

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

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

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

$$\begin{aligned} M \ddot{\vec{\mathbf{E}}} + C \vec{\mathbf{E}} &= 0 \\ \dot{\vec{\mathbf{E}}}^0 &= 0 \\ \vec{\mathbf{E}}^0 &= \Pi_{V_h} \mathbf{E}_0 \end{aligned} \quad (2.5)$$

This linear system has a leak, it will not ensure $\text{div } E(\cdot, t) = 0$ 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 $F_T : \hat{T} \rightarrow T \in \mathcal{T}_h$
- reference shape-functions $\hat{\mathbf{N}}$,

where $\hat{T} := (\hat{\mathbf{a}}_1 | \hat{\mathbf{a}}_2 | \hat{\mathbf{a}}_3)$, $\hat{\mathbf{a}}_i \in \mathbb{R}^2$, $i = 1, 2, 3$ (Figure 2.1).

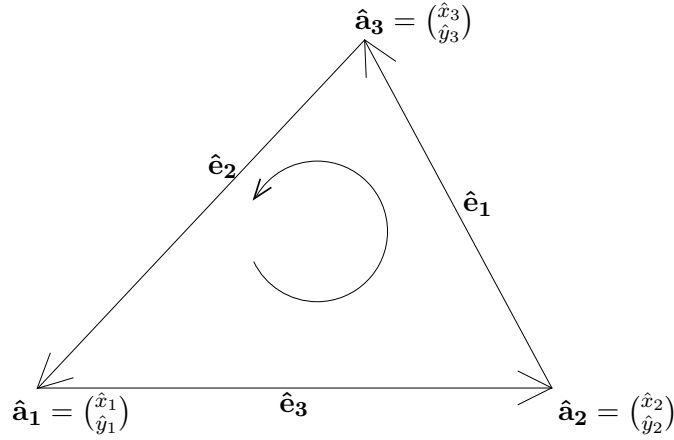


Figure 2.1: Reference element

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

$$\mathbf{x} = F_T(\hat{\mathbf{x}}) = \mathbf{a}_1 + \mathbf{B}_T \hat{\mathbf{x}}, \quad \text{where} \quad \mathbf{B}_T = [\mathbf{a}_2 - \mathbf{a}_1, \mathbf{a}_3 - \mathbf{a}_1]. \quad (2.6)$$

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

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

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 $\mathcal{B}_{\mathcal{N}_{\hat{T}}} = \left\{ \begin{pmatrix} \lambda_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \lambda_1 \end{pmatrix}, \begin{pmatrix} \lambda_2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \lambda_2 \end{pmatrix}, \begin{pmatrix} \lambda_3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \lambda_3 \end{pmatrix} \right\}$
- $\mathcal{E}_h :=$ lowest order Whitney 1-forms $\subset \mathbf{H}_0(\mathbf{curl}; \Omega)$,
with local basis $\mathcal{B}_{\mathcal{E}_{\hat{T}}} = \left\{ \begin{array}{cc} \lambda_2 \mathbf{grad} \lambda_3 & - \lambda_3 \mathbf{grad} \lambda_2, \\ \lambda_3 \mathbf{grad} \lambda_1 & - \lambda_1 \mathbf{grad} \lambda_3, \\ \lambda_1 \mathbf{grad} \lambda_2 & - \lambda_2 \mathbf{grad} \lambda_1 \end{array} \right\}$

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 $\hat{\mathbf{e}}_i$ be the edge of \hat{T} opposite to $\hat{\mathbf{a}}_i$ 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

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

i.e. the \mathbf{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 $\text{div} \mathbf{E}_0 = 0$, 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 $\mathbf{E}(\cdot, t)$ 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

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

It can be proven ([1, Ch. 4]) that the solution

$\mathbf{E} \in Z_0(I, \Omega) := \left\{ \mathbf{u} \in H_0(\mathbf{curl}; \Omega) \mid (\mathbf{u}, \mathbf{Z})_{L^2(\Omega)} = 0, \forall \mathbf{Z} \in H_0(\mathbf{curl}; \Omega) \right\}$. The reason why this is relevant for the continuous regularization is to be clarified with the next claim.

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

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

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

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

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 $\mathbf{v} \in \mathbf{H}_0(\mathbf{curl}; \Omega) \cap H(\text{div}; \Omega)$

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

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

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

¹Quoting 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 $\mathbf{E}_h \in H_x^1 := H^1(\Omega) \cap \mathbf{H}_0(\mathbf{curl}; \Omega)$. Unfortunately \mathbf{E}_h does not always converge to \mathbf{E} . This is the statement of the following theorem from [1, Ch 6, Thm. 6.3, p.322].

Theorem 1. *The space $H_x^1(\Omega)$ 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 $\mathcal{E}_h \not\subset H(\text{div}, \Omega)$. The way out is to regularise (2.4), exploiting the fact that $\text{grad } \mathcal{S}_h \subset \mathcal{E}_h$, we obtain

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

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 $\omega := v \wedge \ast \mathbf{E}$, and

$$\int_{\Omega} d\omega = \underbrace{\int_{\Omega} dv \wedge \ast \mathbf{E}}_{=\int_{\Omega} \langle \mathbf{grad } v, \mathbf{E} \rangle d\mathbf{x}} + \underbrace{\int_{\Omega} v \wedge d \ast \mathbf{E}}_{=\int_{\Omega} v \text{div } \mathbf{E} d\mathbf{x}} \stackrel{\text{Stokes thm.}}{=} \underbrace{\int_{\partial\Omega} \omega}_{=\int_{\partial\Omega} v \wedge \ast \mathbf{E}} = 0 \quad \forall v \in H_0^1(\Omega).$$

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

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

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

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

The coupled ODE for (2.11) reads

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

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

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

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),

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

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

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

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

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

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

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

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

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

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

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

A CFL condition

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

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 plotted results in avi-files.

3.1.1 Main Routines

3.1.1.1 run.m

```

1 function run(datapath,prb,scal)
2 % MATLAB-Script running the numerical experiments for
3 % the transient Maxwell's equations in a cavity
4
5 if (nargin < 1), datapath = './'; end

```

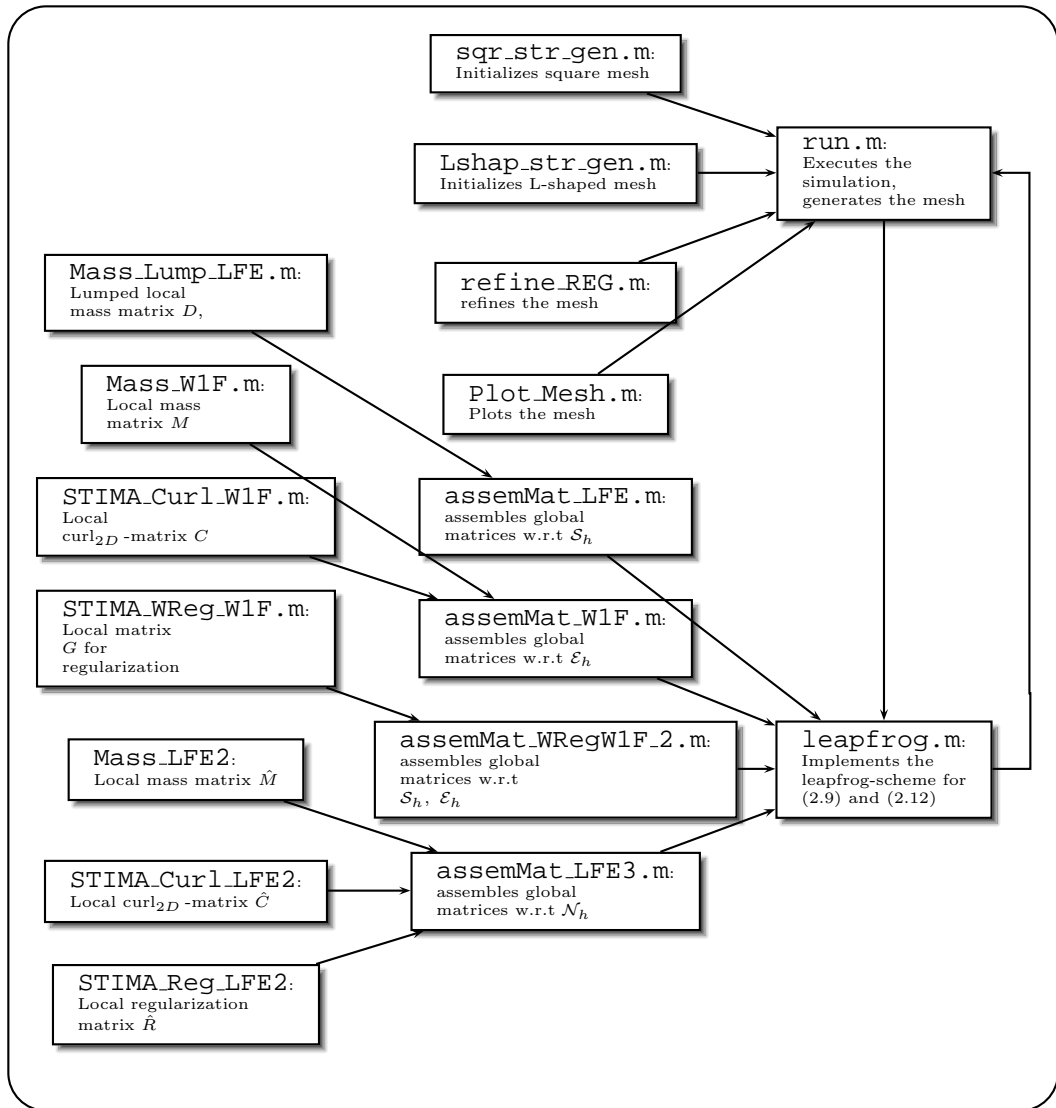


Figure 3.1: Structure of the Implementation

```

6  if (nargin < 2), prb = -1; end
7  if (nargin < 3), scal = 0.5; end
8
9  % Initialize constants
10
11 InitREF = 2; % size of the Initial Mesh
12 NREFSs = 5; % Number of uniform mesh refinements
13 finaltime = 3;
14 timestep = 0.01;
15
16 disp('MATLAB based numerical experiments for transient Maxwell equations');
17 disp('Goal is the comparison of nodal and edge element discretizations');
18 fprintf('Path for output files %s\n',datapath);
19 fprintf('Relative scaling of C and R: %d <=> %d\n',2*scal,2*(1-scal));
20
21 % Generate initial meshes, where the meshwidth depends on InitREF
22 %Square mesh
23 MeshS=sqr_str_gen(InitREF);
24 %Add to the mesh some useful information to handle edge elements
25 MeshS.ElemFlag = ones(size(MeshS.Elements,1),1);
26 MeshS = add_Edges(MeshS);
27 LocS = get_BdEdges(MeshS);
28 MeshS.BdFlags = zeros(size(MeshS.Edges,1),1);
29 MeshS.BdFlags(LocS) = -1;
30
31 %L-shaped mesh
32 MeshL=Lshap_str_gen(InitREF);
33 %Add to the mesh some useful information to handle edge elements
34 MeshL.ElemFlag = ones(size(MeshL.Elements,1),1);
35 MeshL = add_Edges(MeshL);
36 LocL = get_BdEdges(MeshL);
37 MeshL.BdFlags = zeros(size(MeshL.Edges,1),1);
38 MeshL.BdFlags(LocL) = -1;
39
40 % Do NREFS uniform refinement steps
41
42 for i = 1:NREFSs
43
44     % For the square mesh
45     % Refine Mesh
46     MeshS = refine_REG(MeshS);
47     % plot it
48     plot_Mesh(MeshS, 'as')
49
50     % start leapfrog with starting condition initSq
51     [en,sol_v,sol_e,times] = leapfrog(MeshS,@initSq,timestep*(2^(-i+1)),finaltime,2^(i-1));
52
53     %Saving Data
54     Sq_str=['Square' int2str(i)];
55     fprintf(['Finished on' Sq_str ' : Results stored in %s'],[datapath,[Sq_str,'_res']]);
56     save([datapath, Sq_str, '_res'],'en','sol_v','sol_e','times');
57
58     %plot energy evolution in time
59     figure; clf;

```

```

60     subplot(1,2,1);
61     plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
62     legend('Nodal scheme','Edge elements');
63     title([Sq_str,': Electric energy']);
64     xlabel('time');
65     subplot(1,2,2);
66     plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
67     legend('Nodal scheme','Edge elements');
68     title([Sq_str,': Magnetic energy']);
69     xlabel('time');
70     drawnow;
71     clear en solv sole times;
72     % end for the square mesh
73
74
75     %For the L-mesh
76     % Refine mesh
77     MeshL = refine_REG(MeshL);
78     plot_Mesh(MeshL,'as')
79
80     % start leapfrog with starting condition initL
81     [en,sol_v,sol_e,times] = leapfrog(MeshL,@initL,timestep*(2^(-i+1)),finaltime,2^(i-1),
82
83     %Saving Data
84     L_str=[ 'Lshape' int2str(i)];
85     fprintf(['Finished on' L_str ': Results stored in %s'],[datapath,[L_str,'_res']]);
86     save([datapath, L_str, '_res'],'en','sol_v','sol_e','times');
87
88     % Actualize energy plot
89     figure; clf;
90     subplot(1,2,1);
91     plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
92     legend('Nodal scheme','Edge elements');
93     title([L_str,': Electric energy']);
94     xlabel('time');
95     subplot(1,2,2);
96     plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
97     legend('Nodal scheme','Edge elements');
98     title([L_str,': Magnetic energy']);
99     xlabel('time');
100    drawnow;
101    clear en solv sole times;
102
103
104    end

```

3.1.1.2 leapfrog.m

```

1  function [energies,sol_v,sol_e,times] = leapfrog(Mesh,init_field,ts,T,grabstep,scal)
2  % leapfrog timestepping discretizations of Maxwell's equations
3  %
4  % Mesh          -> 2D unstructured mesh

```

```

5 % init_field    -> string designating the routine providing the initial
6 %               vectorfield
7 % ts           -> timestep
8 % T            -> final time
9 % grabstep     -> every #grabstep iterate will be sampled
10 % scal         -> governs strength of regularization
11 %              A = 2*(scal*C + (1-scal)*R)
12 %
13 % Result:
14 %
15 % energies -> trace of electric/magnetic energies during timestepping
16 %           energies(:,1) = time,
17 %           energies(:,2) = electric energy of nodal solution
18 %           energies(:,3) = magnetic energy of nodal solution
19 %           energies(:,4) = electric energy of edge element solution
20 %           energies(:,5) = magnetic energy of edge element solution
21 % sol_v -> sampled solutions for nodal FEM
22 % sol_e -> sampled solutions for edge elements
23 % times -> vector of sampling times
24 %
25 %%%%%%%%%%%
26
27 % Initialize some functions
28
29 U.Handle = @(x,varargin)ones(size(x,1),1);
30 F.Handle = @(x,varargin)[zeros(size(x,1),1) zeros(size(x,1),1)];
31 GD.Handle = @(x,varargin)[-x(:,2) x(:,1)];
32
33
34 %Add Boundary plot data to the mesh, this data will be used by
35 %plotiteratel.m
36 [Mesh.BdEdges_x Mesh.BdEdges_y]=dataBoundaryPlot(Mesh);
37 Mesh=setBdFlags(Mesh);
38 nCoordinates = size(Mesh.Coordinates,1);
39
40 %determine scalation and the number of steps to be saved, if they weren't
41 %specified as arguments
42 if (nargin < 6), scal = 0.5; end
43 if (nargin < 5), grabstep = 1; end
44
45
46 %%%%%%%%%% Edge elements matrix generation %%%%%%%%%%%
47 %%%%%%%%%%%
48 % Ap equals the stiffnes matrix for the curl without regularization and
49 % discretized with whithney-1 edge elements, Me is the corresponding
50 % Mass Matrix the error between the theoretical matrices is 2.2737e-13
51
52 D = assemMat_LFE(Mesh,@MASS_Lump_LFE);
53 G = assemMat_WRegW1F_2(Mesh,@STIMA_WReg_W1Fb,P7O6(),U.Handle);
54 Ap = assemMat_W1F(Mesh,@STIMA_Curl_W1F,U.Handle,P7O6());
55 Me = assemMat_W1F(Mesh,@MASS_W1F,U.Handle,P7O6());
56 %G=Me*G1;
57
58 % Determine degrees of freedom

```

```

59 Loc = get_BdEdges(Mesh);
60 DEdges = Loc(Mesh.BdFlags(Loc) == -1);
61 DNodes = unique([Mesh.Edges(DEdges,1); Mesh.Edges(DEdges,2)]);
62 VDofs = setdiff(1:nCoordinates,DNodes);
63 EDofs = setdiff(1:size(Mesh.Edges,1),DEdges);
64
65 % Curl-matrix with regularization
66 Ae=Ap(EDofs,EDofs)+G(EDofs,VDofs)*(D(VDofs,VDofs)\G(EDofs,VDofs)');
67
68
69
70 % Projection of initial value onto discrete space
71 ev = assemLoad_W1F(Mesh,P706(),init_field);
72
73 % non discrete divergence free
74 Me=Me(EDofs,EDofs);
75 ev = ev(EDofs);
76 ev = Me\ev; %Starting value
77
78 % Discrete divergence free %%%%%%%%%%%%%%%
79 % Is not working with smooth initial condition
80 % Topological gradient. Note that G=Me*G1.
81 G1=Gradmat(Mesh);
82 G1=G1(EDofs,VDofs);
83 p = G1*((G1'*Me*G1)\(G1'*Me*ev));
84 ev = ev + p; %Starting value
85
86
87
88
89 clear Ap D G
90
91 %%%%%%%%% Nodal elements matrix generation %%%%%%%%%
92 %%%%%%%%%
93 % An equals the stiffness matrix for the curl without regularization and
94 % discretized with linear finite Elements elements, Mn is the corresponding
95 % Mass Matrix
96 An = assemMat_LFE3(Mesh,@STIMA_Curl_LFE2)+assemMat_LFE3(Mesh,@STIMA_Reg_LFE2);
97 Mn = assemMat_LFE3(Mesh,@MASS_LFE2);
98
99 % Determine degrees of freedom
100 NDofs = [2*find(Mesh.VertBdFlags(:,1) == 0); 2*find(Mesh.VertBdFlags(:,2) == 0)-1];
101 An=An(NDofs,NDofs);
102 Mn=Mn(NDofs,NDofs);
103
104 % Projection of initial value onto discrete space
105 nv = assemLoad_LFE3(Mesh,P706(),init_field);
106 nv = nv(NDofs);
107 nv = Mn\nv; % Starting Value
108
109
110
111 % Memory allocation for lepa frog-scheme %%%%%%%%%
112 nv_new = zeros(size(nv));

```

```

113 nv_mid = zeros(size(nv));
114 nv_tmp = zeros(size(nv));
115 ev_new = zeros(size(ev));
116 ev_mid = zeros(size(ev));
117 ev_tmp = zeros(size(ev));
118
119
120 %%% initialize the plot window
121 disp('Displaying initial iterates');
122 figure;
123 clf; figno = gcf;
124 %%% actualize the plot window
125 plotiteratel(Mesh,ev,nv,0,figno,NDofs,EDofs);
126
127 %%% precomputing some variables before the time-iteration
128 sol_v = nv;
129 sol_e = ev;
130 times = [0.0];
131 nv_tmp = An*nv;
132 ev_tmp = Ae*ev;
133 etot_v = dot(nv_tmp,nv);
134 etot_e = dot(ev_tmp,ev);
135
136 disp('Initial energies:');
137 fprintf('\t #### Nodal scheme : E_el = %f, E_mag = %f\n',...
138         0,etot_v);
139 fprintf('\t #### Edge elements: E_el = %f, E_mag = %f\n',...
140         0,etot_e);
141
142 % First step
143 nv_mid = nv - 0.5*ts*ts*(Mn\nv_tmp);
144 ev_mid = ev - 0.5*ts*ts*(Me\ev_tmp);
145
146 stp = 1;
147 if (grabstep == 1)
148     sol_v = [sol_v nv_mid];
149     sol_e = [sol_e ev_mid];
150     times = [times ts];
151 end
152
153 plotiteratel(Mesh,ev_mid,nv_mid,ts,gcf,NDofs,EDofs);
154 energies = [0.0 0.0 etot_v 0.0 etot_e];
155
156 % % %%% uncomment if memory lacks
157 save_idx=0;
158 %%release memory step. Stores results in harddisk every 128*grabstep steps
159 relMemStep=128*grabstep;
160 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
161
162 for t=2*ts:ts:T
163     stp = stp + 1;
164     fprintf('Iteration step %d at time %d\n',stp,t);
165     nv_tmp = An*nv_mid;
166     ev_tmp = Ae*ev_mid;

```

```

167     nv_new = 2*nv_mid - nv - ts*ts*(Mn\nv_tmp);
168     ev_new = 2*ev_mid - ev - ts*ts*(Me\ev_tmp);
169
170     disp('Displaying new iterate');
171
172     plotiteratel(Mesh,ev_new,nv_new,t,figno,NDofs,EDofs);
173
174
175     if (mod(stp,grabstep) == 0)
176         sol_v = [sol_v nv_new];
177         sol_e = [sol_e ev_new];
178         times = [times t];
179         %% uncomment if memory lacks
180         if(mod(stp,relMemStep)==0)
181             save_idx=save_idx+1;
182             file_name=[int2str(save_idx) 'results.mat'];
183             save(file_name, 'sol_v', 'sol_e', 'times');
184             sol_v = [];
185             sol_e = [];
186             times = [];
187         end
188
189         %F(stp/grabstep) = getframe(figno);
190     end
191
192     % Computing energies
193     nv = (nv_new - nv)/(2*ts);
194     ev = (ev_new - ev)/(2*ts);
195     el_en_v = dot(nv,Mn*nv);
196     el_en_e = dot(ev,Me*ev);
197     mag_en_v = dot(nv_tmp,nv_mid);
198     mag_en_e = dot(ev_tmp,ev_mid);
199
200     fprintf('\t Nodal scheme : E_el = %f, E_mag = %f, E_tot = %f\n',...
201         el_en_v,mag_en_v,el_en_v+mag_en_v);
202     fprintf('\t Edge elements: E_el = %f, E_mag = %f, E_tot = %f\n',...
203         el_en_e,mag_en_e,el_en_e+mag_en_e);
204     if (el_en_v+mag_en_v > 10*etot_v)
205         disp('Instability of nodal scheme!');
206         break;
207     end
208     if (el_en_e+mag_en_e > 10*etot_e)
209         disp('Instability of edge element scheme!');
210         break;
211     end
212
213     energies = [energies; t el_en_v mag_en_v el_en_e mag_en_e];
214     nv = nv_mid;
215     ev = ev_mid;
216     ev_mid = ev_new;
217     nv_mid = nv_new;
218 end
219
220 %% Uncomment if memory lacks

```

```

221 save_idx=save_idx+1;
222 file_name=[int2str(save_idx) 'results.mat'];
223 save(file_name, 'sol_v', 'sol_e', 'times');
224 save('energies.mat','energies', 'save_idx');
225 clear variables;
226 %close all;
227 load energies;
228 load 'lresults'
229 sol_e_tmp=sol_e;
230 sol_v_tmp=sol_v;
231 times_tmp=times;
232 for i=2:save_idx
233     filename=[int2str(i) 'results'];
234     load(filename);
235     sol_e_tmp=[sol_e_tmp sol_e];
236     sol_v_tmp=[sol_v_tmp sol_v];
237     times_tmp=[times_tmp times];
238 end
239 sol_e=sol_e_tmp;
240 sol_v=sol_v_tmp;
241 times=times_tmp;
242 %movie(F,1,10)

```

3.1.1.3 assemMat_LFE.m

```

1 function varargout = assemMat_LFE(Mesh,EHandle,varargin)
2 % ASSEMMAT_LFE Assemble linear FE contributions.
3 %
4 % A = ASSEMMAT_LFE(MESH,EHANDLE) assembles the global matrix from the
5 % local element contributions given by the function handle EHANDLE and
6 % returns the matrix in a sparse representation.
7 %
8 % A = ASSEMMAT_LFE(MESH,EHANDLE,EPARAM) handles the variable length
9 % argument list EPARAM to the function handle EHANDLE during the assembly
10 % process.
11 %
12 % [I,J,A] = ASSEMMAT_LFE(MESH,EHANDLE) assembles the global matrix from
13 % the local element contributions given by the function handle EHANDLE
14 % and returns the matrix in an array representation.
15 %
16 % The struct MESH must at least contain the following fields:
17 % COORDINATES M-by-2 matrix specifying the vertices of the mesh.
18 % ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the
19 % mesh.
20 % ELEMFLAG N-by-1 matrix specifying additional element information.
21 %
22 % Example:
23 %
24 % Mesh = loadMesh('Coord.LShap.dat','Elem.LShap.dat');
25 % Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
26 % EHandle = @STIMA_Lapl_LFE;
27 % A = assemMat_LFE(Mesh,EHandle);

```



```

28 %
29 % See also set_Rows, set_Cols.
30
31 % Copyright 2005–2005 Patrick Meury
32 % SAM – Seminar for Applied Mathematics
33 % ETH–Zentrum
34 % CH–8092 Zurich, Switzerland
35
36 % Initialize constants
37
38 nElements = size(Mesh.Elements,1);
39
40 % Preallocate memory
41
42 I = zeros(9*nElements,1);
43 J = zeros(9*nElements,1);
44 A = zeros(9*nElements,1);
45
46 % Check for element flags
47
48 if(isfield(Mesh,'ElemFlag')),
49     flags = Mesh.ElemFlag;
50 else
51     flags = zeros(nElements,1);
52 end
53
54 % Assemble element contributions
55
56 loc = 1:9;
57 for i = 1:nElements
58
59     % Extract vertices of current element
60
61     idx = Mesh.Elements(i,:);
62     Vertices = Mesh.Coordinates(idx,:);
63
64     % Compute element contributions
65
66     Aloc = EHandle(Vertices,flags(i),varargin{:});
67
68     % Add contributions to stiffness matrix
69
70     I(loc) = set_Rows(idx,3);
71     J(loc) = set_Cols(idx,3);
72     A(loc) = Aloc(:);
73     loc = loc+9;
74
75 end
76
77 % Assign output arguments
78
79 if(nargout > 1)
80     varargout{1} = I;
81     varargout{2} = J;

```

```

82     varargout{3} = A;
83     else
84         varargout{1} = sparse(I,J,A);
85     end
86
87 return

```

3.1.1.4 MASS_Lump_LFE.m

```

1 function Aloc = MASS_Lump_LFE(Vertices,varargin)
2 % MASS_LUMP_LFE element lumbda mass matrix.
3 %
4 %   ALOC = MASS_LUMP_LFE(VERTICES) computes the element mass matrix
5 %   using W1F finite elements.
6 %
7 %   VERTICES is 3-by-2 matrix specifying the vertices of the current
8 %   element in a row wise orientation.
9 %
10 %   Example:
11 %
12 %   Aloc = MASS_Lump_LFE(Vertices);
13
14 %   Copyright 2005–2005 Patrick Meury & Mengyu Wang
15 %   SAM – Seminar for Applied Mathematics
16 %   ETH–Zentrum
17 %   CH–8092 Zurich, Switzerland
18
19 % Compute the area of the element
20
21 BK = [Vertices(2,:)-Vertices(1,:);Vertices(3,:)-Vertices(1,:)];
22 det_BK = abs(det(BK));
23
24 % Compute local mass matrix
25
26 Aloc = 1/6*det_BK*eye(3);
27
28 return

```

3.1.1.5 P706.m

```

1 function QuadRule = P706()
2 % P706 2D Quadrature rule.
3 %
4 %   QUADRULE = P706() computes a 7 point Gauss quadrature rule of order 6
5 %   (exact for all polynomials up to degree 5) on the reference element.
6 %
7 %   QUADRULE is a struct containing the following fields:
8 %       w Weights of the quadrature rule
9 %       x Abscissae of the quadrature rule (in reference element)
10 %
11 %   To recover the barycentric coordinates xbar of the quadrature points

```

```

12 %     xbar = [QuadRule.x, 1-sum(QuadRule.x)'];
13 %
14 %     Example:
15 %
16 %     QuadRule = P706();
17
18 %     Copyright 2005–2005 Patrick Meury
19 %     SAM – Seminar for Applied Mathematics
20 %     ETH–Zentrum
21 %     CH–8092 Zurich, Switzerland
22
23     QuadRule.w = [          9/80; ...
24                   (155+sqrt(15))/2400; ...
25                   (155+sqrt(15))/2400; ...
26                   (155+sqrt(15))/2400; ...
27                   (155–sqrt(15))/2400; ...
28                   (155–sqrt(15))/2400; ...
29                   (155–sqrt(15))/2400 ];
30
31     QuadRule.x = [          1/3          1/3; ...
32                   (6+sqrt(15))/21 (6+sqrt(15))/21; ...
33                   (9–2*sqrt(15))/21 (6+sqrt(15))/21; ...
34                   (6+sqrt(15))/21 (9–2*sqrt(15))/21; ...
35                   (6–sqrt(15))/21 (9+2*sqrt(15))/21; ...
36                   (9+2*sqrt(15))/21 (6–sqrt(15))/21; ...
37                   (6–sqrt(15))/21 (6–sqrt(15))/21 ];
38
39     return

```

3.1.1.6 shap_W1F.m

```

1 function shap = shap_W1F(x)
2 % SHAP_W1F Shape functions.
3 %
4 %     SHAP = SHAP_W1F(X) computes the values of the shape functions for the
5 %     edge finite element (Whitney–1-Form) at the quadrature points X.
6 %
7 %     Example:
8 %
9 %     shap = shap_W1F([0 0]);
10 %
11 %     See also shap_LFE2.
12
13 %     Copyright 2005–2005 Patrick Meury and Mengyu Wang
14 %     SAM – Seminar for Applied Mathematics
15 %     ETH–Zentrum
16 %     CH–8092 Zurich, Switzerland
17
18     shap = zeros(size(x,1),6);
19
20     shap(:,1) = –x(:,2);
21     shap(:,2) = x(:,1);

```

```

22     shap(:,3) = -x(:,2);
23     shap(:,4) = x(:,1)-1;
24     shap(:,5) = 1-x(:,2);
25     shap(:,6) = x(:,1);
26
27     return

```

3.1.1.7 assemMat_WlF.m

```

1  function varargout = assemMat_WlF(Mesh,EHandle,varargin)
2  % ASSEMMAT_WlF Assembly for *edge elements* in 2D
3  %
4  %   A = ASSEMMAT_WlF(MESH,EHANDLE) assembles the global matrix from the
5  %   local element contributions given by the function handle EHANDLE and
6  %   returns the matrix in a sparse representation.
7  %
8  %   A = ASSEMMAT_WlF(MESH,EHANDLE,EPARAM) handles the variable length
9  %   argument list EPARAM to the function handle EHANDLE during the assembly
10 %   process.
11 %
12 %   [I,J,A] = ASSEMMAT_WlF(MESH,EHANDLE) assembles the global matrix from
13 %   the local element contributions given by the function handle EHANDLE
14 %   and returns the matrix in an array representation.
15 %
16 %   The struct MESH must at least contain the following fields:
17 %   COORDINATES  M-by-2 matrix specifying the vertices of the mesh.
18 %   ELEMENTS     N-by-3 or N-by-4 matrix specifying the elements of the
19 %               mesh.
20 %   ELEMFLAG     N-by-1 matrix specifying additional element information.
21 %   VERT2EDGE    Edge numbers associated with pairs of vertices
22 %               (sparse matrix)
23 %
24 %   Example:
25 %
26 %   Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
27 %   Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
28 %   EHandle = @STIMA_Curl_WlF;
29 %   A = assemMat_WlF(Mesh,EHandle);
30 %
31 %   See also SET_ROWS, SET_COLS.
32 %
33 %   Copyright 2005–2005 Patrick Meury & Mengyu Wang
34 %   SAM – Seminar for Applied Mathematics
35 %   ETH–Zentrum
36 %   CH–8092 Zurich, Switzerland
37
38 % Initialize constants
39
40 nElements = size(Mesh.Elements,1);
41
42 % Preallocate memory
43

```

```

44 I = zeros(9*nElements,1);
45 J = zeros(9*nElements,1);
46 A = zeros(9*nElements,1);
47
48 % Check for element flags
49 if (isfield(Mesh,'ElemFlag')), flags = Mesh.ElemFlag;
50 else flags = zeros(nElements,1); end
51
52 % Assemble element contributions
53
54 loc = 1:9;
55 for i = 1:nElements
56
57     % Extract vertices of current element
58
59     vidx = Mesh.Elements(i,:);
60     Vertices = Mesh.Coordinates(vidx,:);
61
62     % Compute element contributions
63
64     Aloc = EHandle(Vertices,flags(i),varargin{:});
65
66     % Extract global edge numbers
67
68     eidx = [Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3)) ...
69             Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1)) ...
70             Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))];
71
72     % Determine the orientation
73
74     if(Mesh.Edges(eidx(1),1)==vidx(2)), p1 = 1; else p1 = -1;
end
75     if(Mesh.Edges(eidx(2),1)==vidx(3)), p2 = 1; else p2 = -1;
end
76     if(Mesh.Edges(eidx(3),1)==vidx(1)), p3 = 1; else p3 = -1;
end
77
78     Peori = diag([p1 p2 p3]); % scaling matrix taking into account orientations
79     Aloc = Peori*Aloc*Peori;
80
81     % Add contributions to stiffness matrix
82
83     I(loc) = set_Rows(eidx,3);
84     J(loc) = set_Cols(eidx,3);
85     A(loc) = Aloc(:);
86     loc = loc+9;
87
88 end
89
90 % Assign output arguments
91
92 if(nargout > 1)
93     varargout{1} = I;
94     varargout{2} = J;

```

```

95     varargout{3} = A;
96     else
97         varargout{1} = sparse(I,J,A);
98     end
99
100 return

```

3.1.1.8 MASS_W1F.m

```

1 function Mloc = MASS_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule,varargin)
2 % MASS_W1F element mass matrix with weight mu for edge elements in 2D
3 %
4 %   MLOC = MASS_W1F(VERTICES) computes the element mass matrix using
5 %   Whitney 1-forms finite elements.
6 %
7 %   VERTICES is 3-by-2 matrix specifying the vertices of the current element
8 %   in a row wise orientation.
9 %
10 %   ElemInfo (not used)
11 %
12 %   MU_HANDLE handle to a functions expecting a matrix whose rows
13 %   represent position arguments. Return value must be a vector
14 %   (variable arguments will be passed to this function)
15 %
16 %   Example:
17 %
18 %   Mloc = MASS_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule);
19
20 %   Copyright 2005–2005 Patrick Meury & Mengyu Wang
21 %   SAM – Seminar for Applied Mathematics
22 %   ETH–Zentrum
23 %   CH–8092 Zurich, Switzerland
24
25 % Compute element mapping
26
27 P1 = Vertices(1,:);
28 P2 = Vertices(2,:);
29 P3 = Vertices(3,:);
30
31 BK = [ P2 - P1 ; P3 - P1 ]; % transpose of transformation matrix
32 det_BK = abs(det(BK));      % twice the area of the triangle
33
34 % Compute constant gradients of barycentric coordinate functions
35 g1 = [P2(2)-P3(2);P3(1)-P2(1)]/det_BK;
36 g2 = [P3(2)-P1(2);P1(1)-P3(1)]/det_BK;
37 g3 = [P1(2)-P2(2);P2(1)-P1(1)]/det_BK;
38
39 % Get barycentric coordinates of quadrature points
40 nPoints = size(QuadRule.w,1);
41 baryc= [QuadRule.x,1-sum(QuadRule.x,2)];
42
43 % Quadrature points in actual element

```

```

44 % stored as rows of a matrix
45 x = QuadRule.x*BK + ones(nPoints,1)*P1;
46
47 % Evaluate coefficient function at quadrature nodes
48 Fval = MU_HANDLE(x,ElemInfo,varargin{:});
49
50 % Evaluate basis functions at quadrature points
51 % the rows of b(i) store the value of the the i-th
52 % basis function at the quadrature points
53 b1 = baryc(:,2)*g3'-baryc(:,3)*g2';
54 b2 = baryc(:,3)*g1'-baryc(:,1)*g3';
55 b3 = baryc(:,1)*g2'-baryc(:,2)*g1';
56
57 % Compute local mass matrix
58
59 weights = QuadRule.w * det_BK;
60 Mloc(1,1) = sum(weights.*Fval.*sum(b1.*b1,2));
61 Mloc(2,2) = sum(weights.*Fval.*sum(b2.*b2,2));
62 Mloc(3,3) = sum(weights.*Fval.*sum(b3.*b3,2));
63 Mloc(1,2) = sum(weights.*Fval.*sum(b1.*b2,2)); Mloc(2,1) = Mloc(1,2);
64 Mloc(1,3) = sum(weights.*Fval.*sum(b1.*b3,2)); Mloc(3,1) = Mloc(1,3);
65 Mloc(2,3) = sum(weights.*Fval.*sum(b2.*b3,2)); Mloc(3,2) = Mloc(2,3);
66
67 return

```

3.1.1.9 STIMA_Curl_W1F.m

```

1 function Aloc = STIMA_Curl_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule,varargin)
2 % STIMA_CURL_W1F element stiffness matrix for curl*curl-operator in 2D
3 % in the case of Galerkin discretization by means of edge elements
4 %
5 % ALOC = STIMA_CURL_W1F(VERTICES,ELEMINFO,MU_HANDLE,QUADRULE) computes the
6 % curl*\mu*curl element stiffness matrix using Whitney 1-forms finite elements.
7 % The function \mu can be passed through the MU_HANDLE argument
8 %
9 % VERTICES is 3-by-2 matrix specifying the vertices of the current element
10 % in a row wise orientation.
11 %
12 % ElemInfo (not used)
13 %
14 % MU_HANDLE handle to a functions expecting a matrix whose rows
15 % represent position arguments. Return value must be a vector
16 % (variable arguments will be passed to this function)
17 %
18 % QuadRule is a quadrature rule on the reference element
19 %
20 % Example:
21 %
22 % Aloc = STIMA_Curl_W1F(Vertices,ElemInfo,MU_HANDLE,QuadRule);
23
24 % Copyright 2005–2006Patrick Meury & Mengyu Wang & Ralf Hiptmair
25 % SAM – Seminar for Applied Mathematics

```

```

26 %   ETH-Zentrum
27 %   CH-8092 Zurich, Switzerland
28
29 % Initialize constant
30
31 nPoints = size(QuadRule.w,1);
32
33 % Compute element mapping
34
35 bK = Vertices(1,:); % row vector !
36 BK = [Vertices(2,:)-bK; Vertices(3,:)-bK]; % Transpose of trafo matrix !
37 det_BK = abs(det(BK)); % twice the area of the triangle
38
39 % Quadrature points in actual element
40 % stored as rows of a matrix
41 x = QuadRule.x*BK + ones(nPoints,1)*bK;
42
43 % Compute function value
44
45 Fval = MU_HANDLE(x,ElemInfo,varargin{:});
46
47 % Compute local curl-curl-matrix
48 % Use that the curl of an edge element function is constant
49 % and equals 1/area of triangle
50
51 Aloc = 4/det_BK*sum(QuadRule.w.*Fval)*ones(3,3);
52 return

```

3.1.1.10 assemMat_WRegWlF_2.m

```

1 function varargout = assemMat_WRegWlF_2(Mesh,EHandle,varargin)
2 % ASSEMMAT_WREGWlF Assemble WREG WlF FE contributions.
3 %
4 %   A = ASSEMMAT_WREGWlF(MESH,EHANDLE) assembles the global matrix from the
5 %   local element contributions given by the function handle EHANDLE and
6 %   returns the matrix in a sparse representation.
7 %
8 %   A = ASSEMMAT_WREGWlF(MESH,EHANDLE,EPARAM) handles the variable length
9 %   argument list EPARAM to the function handle EHANDLE during the assembly
10 %   process.
11 %
12 %   [I,J,A] = ASSEMMAT_WREGWlF(MESH,EHANDLE) assembles the global matrix
13 %   from the local element contributions given by the function handle
14 %   EHANDLE and returns the matrix in an array representation.
15 %
16 %   The struct MESH must at least contain the following fields:
17 %   COORDINATES   M-by-2 matrix specifying the vertices of the mesh.
18 %   ELEMENTS      N-by-3 or N-by-4 matrix specifying the elements of the
19 %                 mesh.
20 %   ELEMFLAG      N-by-1 matrix specifying additional element information.
21 %
22 %   Example:

```



```

23 %
24 % Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
25 % Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
26 % EHandle = @STIMA_WReg_WlF;
27 % A = assemMat_WRegWlF(Mesh,EHandle);
28 %
29 % See also SET_ROWS, SET_COLS.
30
31 % Copyright 2005–2005 Patrick Meury & Mengyu Wang
32 % SAM – Seminar for Applied Mathematics
33 % ETH–Zentrum
34 % CH–8092 Zurich, Switzerland
35
36 % Initialize constants
37
38 nElements = size(Mesh.Elements,1);
39 nCoordinates = size(Mesh.Coordinates,1);
40
41 % Preallocate memory
42
43 I = zeros(9*nElements,1);
44 J = zeros(9*nElements,1);
45 A = zeros(9*nElements,1);
46
47 % Assemble element contributions
48
49 loc = 1:9;
50
51 for i = 1:nElements
52
53     % Extract vertices of current element
54
55     vidx = Mesh.Elements(i,:);
56     Vertices = Mesh.Coordinates(vidx,:);
57
58     % Compute element contributions
59
60     Aloc = EHandle(Vertices,Mesh.ElemFlag(i),varargin{:});
61
62
63     % Extract global edge numbers
64
65     eidx = [Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3)) ...
66             Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1)) ...
67             Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))];
68
69     % Determine the orientation
70
71     if(Mesh.Edges(eidx(1),1)==vidx(2))
72         p1 = 1;
73     else
74         p1 = -1;
75     end
76

```

```

77     if(Mesh.Edges(eidx(2),1)==vidx(3))
78         p2 = 1;
79     else
80         p2 = -1;
81     end
82
83     if(Mesh.Edges(eidx(3),1)==vidx(1))
84         p3 = 1;
85     else
86         p3 = -1;
87     end
88
89     Peori = diag([p1 p2 p3]);
90     Aloc = Peori*Aloc;
91
92     % Add contributions to stiffness matrix
93
94     I(loc) = set_Rows(eidx,3);
95     J(loc) = set_Cols(vidx,3);
96     A(loc) = Aloc(:);
97     loc = loc+9;
98
99 end
100
101 % Assign output arguments
102
103 if(nargout > 1)
104     varargout{1} = I;
105     varargout{2} = J;
106     varargout{3} = A;
107 else
108     varargout{1} = sparse(I,J,A);
109 end
110
111 return

```

3.1.1.11 grad_shap_LFE.m

```

1 function grad_shap = grad_shap_LFE(x)
2 % GRAD_SHAP_LFE Gradient of shape functions.
3 %
4 % GRAD_SHAP = GRAD_SHAP_LFE(X) computes the values of the gradient
5 % of the shape functions for the Lagrangian finite element of order 1
6 % at the quadrature points X.
7 %
8 % Example:
9 %
10 % grad_shap = grad_shap_LFE([0 0]);
11 %
12 % See also shap_LFE.
13
14 % Copyright 2005–2005 Patrick Meury and Kah Ling Sia

```

```

15 % SAM – Seminar for Applied Mathematics
16 % ETH-Zentrum
17 % CH-8092 Zurich, Switzerland
18
19 % Initialize constants
20
21 nPts = size(x,1);
22
23 % Preallocate memory
24
25 grad_shap = zeros(nPts,6);
26
27 % Compute values of gradients
28
29 grad_shap(:,1:2) = -ones(nPts,2);
30 grad_shap(:,3) = ones(nPts,1);
31 grad_shap(:,6) = ones(nPts,1);
32
33 return

```

3.1.1.12 STIMA.WReg.WlFb.m

```

1 function Aloc = STIMA.WReg.WlF(Vertices,ElemInfo,QuadRule,varargin)
2 % Using QuadRule for the future work(space dependent version)
3
4 % Initialize constant
5
6 nGuass = size(QuadRule.w,1);
7
8 % Preallocate memory
9
10 Aloc = zeros(3,3);
11 N_WlF = shap_WlF(QuadRule.x);
12 grad_N = grad_shap_LFE(QuadRule.x);
13
14 % Compute element mapping
15
16 P1 = Vertices(1,:);
17 P2 = Vertices(2,:);
18 P3 = Vertices(3,:);
19 bK = P1;
20 BK = [P2-bK;P3-bK];
21 inv_BK = inv(BK);
22 det_BK = abs(det(BK));
23 TK = transpose(inv_BK);
24
25 % Compute element entry
26
27 N(:,1:2) = N_WlF(:,1:2)*TK;
28 N(:,3:4) = N_WlF(:,3:4)*TK;
29 N(:,5:6) = N_WlF(:,5:6)*TK;
30 G(:,1:2) = grad_N(:,1:2)*TK;

```

```

31     G(:,3:4) = grad_N(:,3:4)*TK;
32     G(:,5:6) = grad_N(:,5:6)*TK;
33
34     Aloc(1,1) = sum(QuadRule.w.*sum(N(:,1:2).*G(:,1:2),2))*det_BK;
35     Aloc(1,2) = sum(QuadRule.w.*sum(N(:,1:2).*G(:,3:4),2))*det_BK;
36     Aloc(1,3) = sum(QuadRule.w.*sum(N(:,1:2).*G(:,5:6),2))*det_BK;
37     Aloc(2,1) = sum(QuadRule.w.*sum(N(:,3:4).*G(:,1:2),2))*det_BK;
38     Aloc(2,2) = sum(QuadRule.w.*sum(N(:,3:4).*G(:,3:4),2))*det_BK;
39     Aloc(2,3) = sum(QuadRule.w.*sum(N(:,3:4).*G(:,5:6),2))*det_BK;
40     Aloc(3,1) = sum(QuadRule.w.*sum(N(:,5:6).*G(:,1:2),2))*det_BK;
41     Aloc(3,2) = sum(QuadRule.w.*sum(N(:,5:6).*G(:,3:4),2))*det_BK;
42     Aloc(3,3) = sum(QuadRule.w.*sum(N(:,5:6).*G(:,5:6),2))*det_BK;
43
44     return

```

3.1.1.13 assemLoad_WlF.m

```

1  function L = assemLoad_WlF(Mesh,QuadRule,FHandle,varargin)
2  % ASSEMBLOAD_WlF Assemble WlF FE contributions.
3  %
4  %   L = ASSEMBLOAD_WlF(MESH,QUADRULE,FHANDLE) assembles the global load
5  %   vector for the load data given by the function handle EHANDLE.
6  %
7  %   The struct MESH must at least contain the following fields:
8  %   COORDINATES  M-by-2 matrix specifying the vertices of the mesh.
9  %   ELEMENTS     N-by-3 matrix specifying the elements of the mesh.
10 %   ELEMFLAG     N-by-1 matrix specifying additional element information.
11 %
12 %   QUADRULE is a struct, which specifies the Gauss quadrature that is used
13 %   to do the integration:
14 %   W Weights of the Gauss quadrature.
15 %   X Abscissae of the Gauss quadrature.
16 %
17 %   L = ASSEMBLOAD_WlF(COORDINATES,QUADRULE,FHANDLE,FPARAM) also handles the
18 %   additional variable length argument list FPARAM to the function handle
19 %   FHANDLE.
20 %
21 %   Example:
22 %
23 %   FHandle = @(x,varargin)[x(:,1).^2 x(:,2).^2];
24 %   L = assemLoad_WlF(Mesh,P706(),FHandle);
25 %
26 %   See also shap_WlF.
27
28 %   Copyright 2005–2005 Patrick Meury & Mengyu Wang
29 %   SAM – Seminar for Applied Mathematics
30 %   ETH–Zentrum
31 %   CH–8092 Zurich, Switzerland
32
33 % Initialize constants
34
35 nPts = size(QuadRule.w,1);

```

```

36 nCoordinates = size(Mesh.Coordinates,1);
37 nElements = size(Mesh.Elements,1);
38 nEdges = size(Mesh.Edges,1);
39
40 % Check for element flags
41 if (isfield(Mesh,'ElemFlag')), flags = Mesh.ElemFlag;
42 else flags = zeros(nElements,1); end
43
44 % Preallocate memory
45
46 L = zeros(nEdges,1);
47
48 % Precompute shape functions
49
50 N = shap_WlF(QuadRule.x);
51
52 % Assemble element contributions
53
54 eidx = zeros(1,3);
55 for i = 1:nElements
56
57     % Extract vertices
58
59     vidx = Mesh.Elements(i,:);
60     eidx(1) = Mesh.Vert2Edge(vidx(2),vidx(3));
61     eidx(2) = Mesh.Vert2Edge(vidx(3),vidx(1));
62     eidx(3) = Mesh.Vert2Edge(vidx(1),vidx(2));
63
64     % Compute element mapping
65
66     bK = Mesh.Coordinates(vidx(1),:);
67     BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
68     det_BK = abs(det(BK));
69     TK = transpose(inv(BK));
70
71     x = QuadRule.x*BK + ones(nPts,1)*bK;
72
73     % Compute load data
74
75     FVal = FHandle(x);
76
77     % Determine the orientation
78
79     if(Mesh.Edges(eidx(1),1)==vidx(2))
80         p1 = 1;
81     else
82         p1 = -1;
83     end
84
85     if(Mesh.Edges(eidx(2),1)==vidx(3))
86         p2 = 1;
87     else
88         p2 = -1;
89     end

```

```

90
91     if(Mesh.Edges(eidx(3),1)==vidx(1))
92         p3 = 1;
93     else
94         p3 = -1;
95     end
96
97     % Add contributions to global load vector
98
99     L(eidx(1)) = L(eidx(1)) + sum(QuadRule.w.*sum(FVal.*([N(:,1) N(:,2)]*TK),2))*det_BK*p
100     L(eidx(2)) = L(eidx(2)) + sum(QuadRule.w.*sum(FVal.*([N(:,3) N(:,4)]*TK),2))*det_BK*p
101     L(eidx(3)) = L(eidx(3)) + sum(QuadRule.w.*sum(FVal.*([N(:,5) N(:,6)]*TK),2))*det_BK*p
102
103 end
104
105 return

```

3.1.1.14 shap_LFE2.m

```

1 function shap = shap_LFE2(x)
2 % SHAP_LFE2 Shape functions.
3 %
4 %   SHAP = SHAP_LFE2(X) computes the values of the shape functions for
5 %   the vector valued Lagrangian finite element of order 1 at the
6 %   quadrature points X.
7 %
8 %   Example:
9 %
10 %   shap = shap_LFE2([0 0]);
11 %
12 %   See also shap_LFE, shap_W1F.
13
14 %   Copyright 2005–2005 Patrick Meury and Mengyu Wang
15 %   SAM – Seminar for Applied Mathematics
16 %   ETH–Zentrum
17 %   CH–8092 Zurich, Switzerland
18
19 shap = zeros(size(x,1),12);
20
21 shap(:,1) = 1-x(:,1)-x(:,2);
22 shap(:,4) = 1-x(:,1)-x(:,2);
23 shap(:,5) = x(:,1);
24 shap(:,8) = x(:,1);
25 shap(:,9) = x(:,2);
26 shap(:,12) = x(:,2);
27
28 return

```

3.1.1.15 assemMat_LFE3.m

```

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

```

```

2 % ASSEMMAT_LFE2 Assemble nodal FE contributions.
3 %
4 % A = ASSEMMAT_LFE2(MESH,EHANDLE) assembles the global matrix from the
5 % local element contributions given by the function handle EHANDLE and
6 % returns the matrix in a sparse representation.
7 %
8 % A = ASSEMMAT_LFE2(MESH,EHANDLE,EPARAM) handles the variable length
9 % argument list EPARAM to the function handle EHANDLE during the assembly
10 % process.
11 %
12 % [I,J,A] = ASSEMMAT_LFE2(MESH,EHANDLE) assembles the global matrix from
13 % the local element contributions given by the function handle EHANDLE
14 % and returns the matrix in an array representation.
15 %
16 % The struct MESH must at least contain the following fields:
17 % COORDINATES M-by-2 matrix specifying the vertices of the mesh.
18 % ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the
19 % mesh.
20 % ELEMFLAG N-by-1 matrix specifying additional element information.
21 %
22 % Example:
23 %
24 % Mesh = load.Mesh('Coord.LShap.dat','Elem.LShap.dat');
25 % Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
26 % EHandle = @STIMA_Curl_LFE2;
27 % A = assemMat_LFE2(Mesh,EHandle);
28 %
29 % See also SET_ROWS, SET_COLS.
30
31 % Copyright 2005–2005 Patrick Meury & Mengyu Wang
32 % SAM – Seminar for Applied Mathematics
33 % ETH–Zentrum
34 % CH–8092 Zurich, Switzerland
35
36 % Initialize constants
37
38 nElements = size(Mesh.Elements,1);
39 nCoordinates = size(Mesh.Coordinates,1);
40
41 % Preallocate memory
42
43 I = zeros(36*nElements,1);
44 J = zeros(36*nElements,1);
45 A = zeros(36*nElements,1);
46
47 % Assemble element contributions
48
49 loc = 1:36;
50 for i = 1:nElements
51
52     % Extract vertices of current element
53
54     vidx = Mesh.Elements(i,:);
55     % idx = [vidx(1) vidx(1)+nCoordinates ...

```

```

56 %             vidx(2) vidx(2)+nCoordinates ...
57 %             vidx(3) vidx(3)+nCoordinates];
58
59 idx = [2*vidx(1)-1 2*vidx(1)...
60         2*vidx(2)-1 2*vidx(2) ...
61         2*vidx(3)-1 2*vidx(3)];
62
63 Vertices = Mesh.Coordinates(vidx,:);
64
65 % Compute element contributions
66
67 Aloc = EHandle(Vertices,Mesh.ElemFlag(i),varargin{:});
68
69 % Add contributions to stiffness matrix
70
71 I(loc) = set_Rows(idx,6);
72 J(loc) = set_Cols(idx,6);
73 A(loc) = Aloc(:);
74 loc = loc+36;
75
76 end
77
78 % Assign output arguments
79
80 if(nargout > 1)
81     varargout{1} = I;
82     varargout{2} = J;
83     varargout{3} = A;
84 else
85     varargout{1} = sparse(I,J,A);
86 end
87
88 return

```

3.1.1.16 MASS_LFE2.m

```

1 function Mloc = MASS_LFE2(Vertices,varargin)
2 % MASS_LFE2 Element mass matrix.
3 %
4 % MLOC = MASS_LFE2(VERTICES) computes the element mass matrix using
5 % LFE2 finite elements.
6 %
7 % VERTICES is 3-by-2 matrix specifying the vertices of the current element
8 % in a row wise orientation.
9 %
10 % Example:
11 %
12 % Mloc = MASS_LFE2(Vertices);
13
14 % Copyright 2005–2005 Patrick Meury & Mengyu Wang
15 % SAM – Seminar for Applied Mathematics
16 % ETH–Zentrum

```



```

17 %   CH-8092 Zurich, Switzerland
18
19 % Compute element mapping
20
21 BK = [Vertices(2,:)-Vertices(1,:); (Vertices(3,:)-Vertices(1,:))];
22 det_BK = abs(det(BK));
23
24 % Compute local mass matrix
25
26 Mloc = det_BK/24*[2 0 1 0 1 0;...
27                   0 2 0 1 0 1;...
28                   1 0 2 0 1 0;...
29                   0 1 0 2 0 1;...
30                   1 0 1 0 2 0;...
31                   0 1 0 1 0 2];
32
33 return

```

3.1.1.17 STIMA_Curl_LFE2.m

```

1 function Aloc = STIMA_Curl_LFE2(Vertices,varargin)
2 % STIMA_CURL_LFE2 element stiffness matrix.
3 %
4 %   ALOC = STIMA_CURL_LFE2(VERTICES) computes the element stiffness matrix
5 %   using nodal finite elements.
6 %
7 %   VERTICES is 3-by-2 matrix specifying the vertices of the current
8 %   element in a row wise orientation.
9 %
10 %   Example:
11 %
12 %   Aloc = STIMA_Curl_LFE2(Vertices);
13
14 %   Copyright 2005-2005 Patrick Meury & Mengyu Wang
15 %   SAM - Seminar for Applied Mathematics
16 %   ETH-Zentrum
17 %   CH-8092 Zurich, Switzerland
18
19 % Compute the area of the element
20
21 BK = [Vertices(2,:)-Vertices(1,:);Vertices(3,:)-Vertices(1,:)];
22 det_BK = abs(det(BK));
23
24 % Compute local mass matrix
25
26 K = [ Vertices(3,:) - Vertices(2,:) ...
27       Vertices(1,:) - Vertices(3,:) ...
28       Vertices(2,:) - Vertices(1,:) ];
29
30 Aloc = 1/(2*det_BK)*(K')*K;
31
32 return

```

3.1.1.18 STIMA_Reg_LFE2.m

```
1 function Aloc = STIMA_WReg_WlFa(Vertices,ElemInfo,varargin)
2 % STIMA_WREG.WlF Element stiffness matrix for the WlF finite element.
3 %
4 % ALOC = STIMA_WREG.WlF(VERTICES,ELEMINFO) computes the element stiffness
5 % matrix for the data given by function handle FHANDLE.
6 %
7 % VERTICES is a 3-by-2 matrix specifying the vertices of the current
8 % element in a row wise orientation.
9 %
10 % ELEMINFO is an integer parameter which is used to specify additional
11 % element information on each element.
12 %
13 % Example:
14 %
15 % Aloc = STIMA_WReg_WlF([0 0; 1 0; 0 1],0);
16 %
17 % See also grad_shap_LFE.
18
19 % Copyright 2005–2005 Patrick Meury & Mengyu Wang
20 % SAM – Seminar for Applied Mathematics
21 % ETH–Zentrum
22 % CH–8092 Zurich, Switzerland
23
24 % Preallocate memory
25
26 Aloc = zeros(3,3);
27
28 % Compute element mapping
29
30 P1 = Vertices(1,:);
31 P2 = Vertices(2,:);
32 P3 = Vertices(3,:);
33 bK = P1;
34 BK = [P2-bK;P3-bK];
35 inv_BK = inv(BK);
36 det_BK = abs(det(BK));
37 TK = transpose(inv_BK);
38
39
40 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);
41 Aloc = det_BK/2*L'*L;
42
43 return
```

3.1.1.19 assemLoad_LFE3.m

```
1 function L = assemLoad_LFE3(Mesh,QuadRule,FHandle,varargin)
2 % ASSEMBLOAD_LFE Assemble nodal FE contributions.
3 %
```

```

4 % L = ASSEMBLOAD_LFE2(MESH,QUADRULE,FHANDLE) assembles the global load
5 % vector for the load data given by the function handle FHANDLE.
6 %
7 % The struct MESH must at least contain the following fields:
8 % COORDINATES M-by-2 matrix specifying the vertices of the mesh.
9 % ELEMENTS N-by-3 matrix specifying the elements of the mesh.
10 % ELEMFLAG N-by-1 matrix specifying additional element information.
11 %
12 % QUADRULE is a struct, which specifies the Gauss quadrature that is used
13 % to do the integration:
14 % W Weights of the Gauss quadrature.
15 % X Abscissae of the Gauss quadrature.
16 %
17 % L = ASSEMBLOAD_LFE2(COORDINATES,QUADRULE,FHANDLE,FPARAM) also handles the
18 % additional variable length argument list FPARAM to the function handle
19 % FHANDLE.
20 %
21 % Example:
22 %
23 % FHandle = @(x,varargin)x(:,1).^2+x(:,2).^2;
24 % L = assemLoad_LFE2(Mesh,P706(),FHandle);
25 %
26 % See also shap_LFE2.
27
28 % Copyright 2005–2005 Patrick Meury & Mengyu Wang
29 % SAM – Seminar for Applied Mathematics
30 % ETH-Zentrum
31 % CH-8092 Zurich, Switzerland
32
33 % Initialize constants
34
35 nPts = size(QuadRule.w,1);
36 nCoordinates = size(Mesh.Coordinates,1);
37 nElements = size(Mesh.Elements,1);
38
39 % Preallocate memory
40
41 L = zeros(2*nCoordinates,1);
42
43 % Precompute shape functions
44
45 N = shap_LFE2(QuadRule.x);
46
47 % Assemble element contributions
48
49 for i = 1:nElements
50
51 % Extract vertices
52
53 vidx = Mesh.Elements(i,:);
54
55 % Compute element mapping
56
57 bK = Mesh.Coordinates(vidx(1),:);

```

```

58     BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
59     det_BK = abs(det(BK));
60
61     x = QuadRule.x*BK + ones(nPts,1)*bK;
62
63     % Compute load data
64
65     %FVal = FHandle(x,Mesh.ElemFlag(i),varargin{:});
66     FVal = FHandle(x,Mesh.ElemFlag(i),varargin{:});
67
68     % Add contributions to global load vector
69
70     L(2*vidx(1)-1) = L(2*vidx(1)-1) + sum(QuadRule.w.*FVal(:,1).*N(:,1))*det_BK;
71     L(2*vidx(2)-1) = L(2*vidx(2)-1) + sum(QuadRule.w.*FVal(:,1).*N(:,5))*det_BK;
72     L(2*vidx(3)-1) = L(2*vidx(3)-1) + sum(QuadRule.w.*FVal(:,1).*N(:,9))*det_BK;
73
74     L(2*vidx(1)) = L(2*vidx(1)) + sum(QuadRule.w.*FVal(:,2).*N(:,4))*det_BK;
75     L(2*vidx(2)) = L(2*vidx(2)) + sum(QuadRule.w.*FVal(:,2).*N(:,8))*det_BK;
76     L(2*vidx(3)) = L(2*vidx(3)) + sum(QuadRule.w.*FVal(:,2).*N(:,12))*det_BK;
77
78     end
79
80     return

```

3.2 Initial Value

3.2.0.20 initL.m

```

1  function y = initL(x,omega,phioffs)
2  % Initial electric field
3  % omega = 3/2*pi for L-shaped domain
4  % omega = pi/2 for square
5  omega = 3*pi/2;
6  phioffs=pi/2;%0.5*pi;
7  ep = pi/omega;
8  phi = atan2(x(:,2),x(:,1)) + phioffs;
9  rad = sqrt(x(:,1).*x(:,1)+x(:,2).*x(:,2));
10
11  sgty=find(rad < eps);
12  rad(sgty)=eps*ones(size(sgty));
13
14  p = rad.^(ep).*cos(ep.*phi);
15  cpx = ep*rad.^(ep-1).*(cos(ep.*phi).*(-x(:,2)) + ...
16      sin(ep*phi).*x(:,1))./rad;
17  cpy = ep*rad.^(ep-1).*(cos(ep.*phi).*x(:,1) + ...
18      sin(ep*phi).*x(:,2))./rad;
19  cp=[cpx cpy];
20
21
22  y=zeros(size(x));
23  cf=y;

```

```

24 f=ones(size(x(:,1)));
25 Loc1 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) < 0.5));
26
27 Loc2= find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) < 0.5));
28 f(Loc2) = sin(pi*x(Loc2,1)).^2;
29 cf(Loc2,2) = pi*sin(2*pi*x(Loc2,1));
30 Loc3 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) >= 0.5));
31 f(Loc3) = sin(pi*x(Loc3,2)).^2;
32 cf(Loc3,1) = pi*(-sin(2*pi*x(Loc3,2)));
33 Loc4 = find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) >= 0.5));
34 f(Loc4) = (sin(pi*x(Loc4,1)).*sin(pi*x(Loc4,2))).^2;
35 cf(Loc4,:) = pi*[-(sin(pi*x(Loc4,1)).^2).*sin(2*pi*x(Loc4,2)) ...
36 sin(2*pi*x(Loc4,1)).*(sin(pi*x(Loc4,2)).^2)];
37
38 y = [f.*cpx f.*cpy] + [p.*cf(:,1) p.*cf(:,2)];

```

3.2.0.21 initSq.m

```

1 function y = initSq(x,omega,phioffs)
2 % Initial electric field
3 % omega = 3/2*pi for L-shaped domain
4 % omega = pi/2 for square
5 omega = pi/2;
6 phioffs=0;
7 ep = pi/omega;
8 phi = atan2(x(:,2),x(:,1)) + phioffs;
9 rad = sqrt(x(:,1).*x(:,1)+x(:,2).*x(:,2));
10
11 sgty=find(rad < eps);
12 rad(sgty)=eps*ones(size(sgty));
13
14 p = rad.^(ep).*cos(ep.*phi);
15 cpx = ep*rad.^(ep-1).*(cos(ep.*phi).*(-x(:,2)) + ...
16 sin(ep.*phi).*x(:,1))./rad;
17 cpy = ep*rad.^(ep-1).*(cos(ep.*phi).*x(:,1) + ...
18 sin(ep.*phi).*x(:,2))./rad;
19 cp=[cpx cpy];
20
21
22 y=zeros(size(x));
23 cf=y;
24 f=ones(size(x(:,1)));
25 Loc1 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) < 0.5));
26
27 Loc2= find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) < 0.5));
28 f(Loc2) = sin(pi*x(Loc2,1)).^2;
29 cf(Loc2,2) = pi*sin(2*pi*x(Loc2,1));
30 Loc3 = find((abs(x(:,1)) < 0.5) & (abs(x(:,2)) >= 0.5));
31 f(Loc3) = sin(pi*x(Loc3,2)).^2;
32 cf(Loc3,1) = pi*(-sin(2*pi*x(Loc3,2)));
33 Loc4 = find((abs(x(:,1)) >= 0.5) & (abs(x(:,2)) >= 0.5));
34 f(Loc4) = (sin(pi*x(Loc4,1)).*sin(pi*x(Loc4,2))).^2;

```

```

35     cf(Loc4,:) = pi*[-(sin(pi*x(Loc4,1)).^2).*sin(2*pi*x(Loc4,2)) ...
36         sin(2*pi*x(Loc4,1)).*(sin(pi*x(Loc4,2)).^2)];
37
38 y = [f.*cpx f.*cpy] + [p.*cf(:,1) p.*cf(:,2)];

```

3.2.1 Plotting

3.2.1.1 plotfield1.m

```

1  function plotfield1(Mesh,vals,BBox,titstr)
2
3  % Creates an arrow plot of a vectorfield whose values are stored
4  % in the vals column vector
5  %
6  % Mesh -> Data for 2D unstructured mesh
7  % vals -> column vector whose length must agree with mesh.Nv
8  %
9
10
11 nVertices=size(Mesh.Coordinates,1);
12 if (size(vals,1) ~= 2*nVertices) error('Size mismatch for argument vector'); end
13 if (size(vals,2) ~= 1), error('Vals must be a column vector'); end
14 if (nargin < 3), titstr = 'Arrowplot of vectorfield'; end
15
16 hold on;
17 title(titstr);
18 bb = [ 0 0 0 0 ];
19
20 % Generates plot
21
22 plot(Mesh.BdEdges_x,Mesh.BdEdges_y,'r-');
23
24 % Plot arrows
25 vx = vals(1:2:2*nVertices,1);
26 vy = vals(2:2:2*nVertices,1);
27 quiver(Mesh.Coordinates(:,1),Mesh.Coordinates(:,2),vx,vy,0.75,'b-');
28 axis(BBox*1.01);
29 hold off;

```

3.2.1.2 plotiteratel.m

```

1  function F = plotiteratel(Mesh,ev,nv,t,figno,NDofs,EDofs,mesh)
2  % Plots the current iterate during the leapfrog iteration
3  %
4  % mesh -> 2D triangulation
5  % ev -> vector of edge dofs of length #of active edges
6  % nv -> vector of nodal dofs of length #of active vertices
7  % t -> time (for title)
8  nVertices=size(Mesh.Coordinates,1);
9  evf = zeros(size(Mesh.Edges(:,1)));
10 nvf = zeros(2*size(Mesh.Coordinates(:,1)),1,1);

```

```

11
12 evf(EDofs) = ev;
13 nvf(NDofs) = nv;
14 nvfm=sqrt((nvf(1:2:2*nVertices-1,1).^2+nvf(2:2:2*nVertices,1).^2));
15 s = sprintf('Time = %f',t);
16
17 BBox=[-1 1 -1 1];
18 figure(figno);
19 clf;
20 h1=subplot(2,2,1);
21 h2=subplot(2,2,3);
22 plot_Norm_WLF(evf,Mesh,h1,h2,BBox,'Edge Elements');
23 subplot(h2);
24 axis([BBox 0 3]);
25 title(s);
26 view([-30 70]);
27 subplot(2,2,2);
28 plotfield1(Mesh,nvf,BBox,'Nodal elements');
29 h=subplot(2,2,4);
30 plot_LFE(nvfm,Mesh,h);
31 axis([BBox 0 3]);
32 title(s);
33 view([-30 70]);

```

3.2.1.3 plot_LFE.m

```

1 function varargout = plot_LFE(U,Mesh,fig)
2 % PLOT_LFE Plot finite element solution.
3 %
4 % PLOT_LFE(U,MESH) generates a plot of the finite element solution U on
5 % the mesh MESH.
6 %
7 % The struct MESH must at least contain the following fields:
8 % COORDINATES M-by-2 matrix specifying the vertices of the mesh.
9 % ELEMENTS N-by-3 matrix specifying the elements of the mesh.
10 %
11 % H = PLOT_LFE(U,MESH) also returns the handle to the figure.
12 %
13 % Example:
14 %
15 % plot_LFE(U,MESH);
16
17 % Copyright 2005–2005 Patrick Meury
18 % SAM – Seminar for Applied Mathematics
19 % ETH–Zentrum
20 % CH–8092 Zurich, Switzerland
21
22 % Initialize constants
23
24 OFFSET = 0.05;
25
26 % Compute axes limits

```

```

27
28 XMin = min(Mesh.Coordinates(:,1));
29 XMax = max(Mesh.Coordinates(:,1));
30 YMin = min(Mesh.Coordinates(:,2));
31 YMax = max(Mesh.Coordinates(:,2));
32 XLim = [XMin XMax] + OFFSET*(XMax-XMin)*[-1 1];
33 YLim = [YMin YMax] + OFFSET*(YMax-YMin)*[-1 1];
34
35 % Generate figure
36
37 if(isreal(U))
38
39     % Compute color axes limits
40
41     CMin = min(U);
42     CMax = max(U);
43     if(CMin < CMax) % or error will occur in set function
44         CLim = [CMin CMax] + OFFSET*(CMax-CMin)*[-1 1];
45     else
46         CLim = [1-OFFSET 1+OFFSET]*CMin;
47     end
48
49     % Plot real finite element solution
50     % Create new figure, if argument 'fig' is not specified
51     % Otherwise this argument is supposed to be a figure handle
52     if (nargin < 3), fig = figure('Name','Linear finite elements');
53     else %figure(fig);
54     subplot(fig);
55     end
56
57     patch('faces', Mesh.Elements, ...
58         'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) U], ...
59         'CData', U, ...
60         'facecolor', 'interp', ...
61         'edgecolor', 'none');
62     %set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 4]);
63
64     if(nargout > 0)
65         varargout{1} = fig;
66     end
67
68 else
69
70     % Compute color axes limits
71
72     CMin = min([real(U); imag(U)]);
73     CMax = max([real(U); imag(U)]);
74     CLim = [CMin CMax] + OFFSET*(CMax-CMin)*[-1 1];
75
76     % Plot imaginary finite element solution
77
78     fig_1 = figure('Name','Linear finite elements');
79     patch('faces', Mesh.Elements, ...
80         'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) real(U)], ...

```



```

81         'CData', real(U), ...
82         'facecolor', 'interp', ...
83         'edgecolor', 'none');
84     set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 4]);
85     fig_2 = figure('Name','Linear finite elements');
86     patch('faces', Mesh.Elements, ...
87         'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) imag(U)], ...
88         'CData', imag(U), ...
89         'facecolor', 'interp', ...
90         'edgecolor', 'none');
91     %set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 1]);
92     set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 4]);
93     if(nargout > 0)
94         varargout{1} = fig_1;
95         varargout{2} = fig_2;
96     end
97
98 end
99
100 return

```

3.2.1.4 plot_Mesh.m

```

1 function varargout = plot_Mesh(Mesh,varargin)
2 % PLOT_MESH Mesh plot.
3 %
4 %   PLOT_MESH(MESH) generate 2D plot of the mesh.
5 %
6 %   PLOT(MESH,OPT) adds labels to the plot, where OPT is a character string
7 %   made from one element from any or all of the following characters:
8 %       p Add vertex labels to the plot.
9 %       e Add edge labels/flags to the plot.
10 %      t Add element labels/flags to the plot.
11 %      a Display axes on the plot.
12 %      s Add title and axes labels to the plot.
13 %      f Do NOT create new window for the mesh plot
14 %      [c add patch color to elements according to their flags] TODO !
15 %
16 %   H = PLOT_MESH(MESH,OPT) also returns the handle to the figure.
17 %
18 %   The struct MESH should at least contain the following fields:
19 %       COORDINATES M-by-2 matrix specifying the vertices of the mesh.
20 %       ELEMENTS     N-by-3 or N-by-4 matrix specifying the elements of the
21 %                   mesh.
22 %
23 %   Example:
24 %
25 %   plot_Mesh(Mesh,'petas');
26 %
27 %   See also get_BdEdges, add_Edges.
28 %
29 %   Copyright 2005–2005 Patrick Meury

```

```

30 % SAM – Seminar for Applied Mathematics
31 % ETH-Zentrum
32 % CH-8092 Zurich, Switzerland
33
34 if(nargin > 1)
35     opt = varargin{1};
36 else
37     opt = ' ';
38 end
39 % Initialize constants
40
41 OFFSET = 0.05; % Offset parameter
42 EDGECOLOR = 'b'; % Interior edge color
43 BDEDGECOLOR = 'r'; % Boundary edge color
44
45 % Check mesh data structure and add necessary fields
46
47 if(~isfield(Mesh,'Edges'))
48     Mesh = add.Edges(Mesh);
49 end
50 nCoordinates = size(Mesh.Coordinates,1);
51 nElements = size(Mesh.Elements,1);
52 nEdges = size(Mesh.Edges,1);
53
54 % Compute axes limits
55
56 X = Mesh.Coordinates(:,1);
57 Y = Mesh.Coordinates(:,2);
58 XMin = min(X);
59 XMax = max(X);
60 YMin = min(Y);
61 YMax = max(Y);
62 XLim = [XMin XMax] + OFFSET*(XMax-XMin)*[-1 1];
63 YLim = [YMin YMax] + OFFSET*(YMax-YMin)*[-1 1];
64
65 % Compute boundary edges for piecewise linear boundaries
66
67 Loc = get_BdEdges(Mesh);
68 BdEdges_x = zeros(2,size(Loc,1));
69 BdEdges_y = zeros(2,size(Loc,1));
70 BdEdges_x(1,:) = Mesh.Coordinates(Mesh.Edges(Loc,1),1)';
71 BdEdges_x(2,:) = Mesh.Coordinates(Mesh.Edges(Loc,2),1)';
72 BdEdges_y(1,:) = Mesh.Coordinates(Mesh.Edges(Loc,1),2)';
73 BdEdges_y(2,:) = Mesh.Coordinates(Mesh.Edges(Loc,2),2)';
74
75 % Generate plot
76
77 if(isempty(findstr('f',opt)))
78     fig = figure('Name','Mesh plot');
79 end
80
81 if(~ishold)
82     hold on;
83 end

```

```

84 patch('Faces', Mesh.Elements, ...
85       'Vertices', Mesh.Coordinates, ...
86       'FaceColor', 'none', ...
87       'EdgeColor', EDGECOLOR);
88 plot(BdEdges.x,BdEdges.y,[BDEDECOLOR '-']);
89 hold off;
90 set(gca,'XLim',XLim, ...
91       'YLim',YLim, ...
92       'DataAspectRatio',[1 1 1], ...
93       'Box','on', ...
94       'Visible','off');
95
96 % Add labels/flags according to the string OPT
97
98
99 % Add vertex labels
100
101 if(~isempty(findstr('p',opt)))
102     add_VertLabels(Mesh.Coordinates);
103 end
104
105 % Add element labels/flags to the plot
106
107 if(~isempty(findstr('t',opt)))
108     if(isfield(Mesh,'ElemFlag'))
109         add_ElemLabels(Mesh.Coordinates,Mesh.Elements,Mesh.ElemFlag);
110     else
111         add_ElemLabels(Mesh.Coordinates,Mesh.Elements,1:nElements);
112     end
113 end
114
115 % Add edge labels/flags to the plot
116
117 if(~isempty(findstr('e',opt)))
118     if(isfield(Mesh,'BdFlags'))
119         add_EdgeLabels(Mesh.Coordinates,Mesh.Edges,Mesh.BdFlags);
120     else
121         add_EdgeLabels(Mesh.Coordinates,Mesh.Edges,1:nEdges);
122     end
123 end
124
125 % Turn on axes, titles and labels
126
127 if(~isempty(findstr('a',opt)))
128     set(gca,'Visible','on');
129     if(~isempty(findstr('s',opt)))
130         if(size(Mesh.Elements,2) == 3)
131             title(['{\bf 2D triangular mesh}']);
132         else
133             title(['{\bf 2D quadrilateral mesh}']);
134         end
135         xlabel(['{\bf # Vertices : ', int2str(nCoordinates), ...
136               ',          # Elements : ', int2str(nElements), ...
137               ',          # Edges : ', int2str(nEdges), '}']);

```

```

138         end
139     end
140
141     drawnow;
142
143     % Assign output arguments
144
145     if(nargout > 0)
146         varargout{1} = fig;
147     end
148
149     return
150
151
152     %% Add vertex labels %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
153
154     function [] = addVertLabels(Coordinates)
155     % ADD_VERTLABELS Add vertex labels to the plot.
156     %
157     % ADD_VERTLABELS(COORDINATES) adds vertex labels to the current
158     % figure.
159     %
160     % Example:
161     %
162     % addVertLabels(Mesh.Coordinates);
163
164     % Copyright 2005–2005 Patrick Meury
165     % SAM – Seminar for Applied Mathematics
166     % ETH–Zentrum
167     % CH–8092 Zurich, Switzerland
168
169     % Initialize constants
170
171     WEIGHT = 'bold';
172     SIZE = 8;
173     COLOR = 'k';
174
175     % Add vertex labels to the plot
176
177     nCoordinates = size(Coordinates,1);
178     for i = 1:nCoordinates
179         text(Coordinates(i,1),Coordinates(i,2),int2str(i), ...
180             'HorizontalAlignment','Center', ...
181             'VerticalAlignment','Middle', ...
182             'Color',COLOR, ...
183             'FontWeight',WEIGHT, ...
184             'FontSize',SIZE);
185     end
186
187     return
188
189     %% Add element labels %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
190
191     function [] = addElemLabels(Coordinates,Elements,Labels)

```

```

192 % ADD_ELEMLABELS Add element labels to the plot.
193 %
194 %   ADD_ELEMLABELS(COORDINATES,ELEMENTS,LABELS) adds the element labels
195 %   LABELS to the current figure.
196 %
197 %   Example:
198 %
199 %   add_ElemLabels(Mesh.Coordinates,Mesh.Elements,Labels);
200
201 %   Copyright 2005–2005 Patrick Meury
202 %   SAM – Seminar for Applied Mathematics
203 %   ETH–Zentrum
204 %   CH–8092 Zurich, Switzerland
205
206 % Initialize constants
207
208 WEIGHT = 'bold';
209 SIZE = 8;
210 COLOR = 'k';
211
212 % Add element labels to the plot
213
214 [nElements,nVert] = size(Elements);
215 for i = 1:nElements
216     CoordMid = sum(Coordinates(Elements(i,:),:),1)/nVert;
217     text(CoordMid(1),CoordMid(2),int2str(Labels(i)), ...
218          'HorizontalAlignment','Center', ...
219          'VerticalAlignment','Middle', ...
220          'Color',COLOR, ...
221          'FontWeight',WEIGHT, ...
222          'FontSize',SIZE);
223 end
224
225 return
226
227 %%% Add edge labels %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
228
229 function [] = add_EdgeLabels(Coordinates,Edges,Labels)
230 % ADD_EDGELABELS Add edge labels to the plot.
231 %
232 %   ADD_EDGELABELS(COORDINATES,EDGES,LABELS) adds the edge labels LABELS to
233 %   the current figure.
234 %
235 %   Example:
236 %
237 %   add_EdgeLabels(Coordinates,Edges,Labels);
238
239 %   Copyright 2005–2005 Patrick Meury
240 %   SAM – Seminar for Applied Mathematics
241 %   ETH–Zentrum
242 %   CH–8092 Zurich, Switzerland
243
244 % Initialize constants
245

```

```

246 WEIGHT = 'bold';
247 SIZE = 8;
248 COLOR = 'k';
249
250 % Add edge labels to the plot
251
252 nEdges = size(Edges,1);
253 for i = 1:nEdges
254     CoordMid = (Coordinates(Edges(i,1),:)+Coordinates(Edges(i,2),:))/2;
255     text(CoordMid(1),CoordMid(2),int2str(Labels(i)), ...
256          'HorizontalAlignment','Center', ...
257          'VerticalAlignment','Middle', ...
258          'Color',COLOR, ...
259          'FontWeight',WEIGHT, ...
260          'FontSize',SIZE);
261 end
262
263 return

```

3.2.1.5 plot_Norm_WlF.m

```

1 function varargout = plot_Norm_WlF(U,Mesh,fig1,fig2,BBox,titstr)
2 % PLOT_NORM_WlF Plot routine for the modulus of WlF results.
3 %
4 % FIG = PLOT_NORM_WlF(U,MESH) generates a plot of the modulus for the
5 % velocity field which is represented by the WlF solution U on the struct
6 % MESH and returns the handle FIG to the figure.
7 %
8 % The struct should at least contain the following fields:
9 % COORDINATES M-by-2 matrix specifying the vertices of the mesh, where
10 % M is equal to the number of vertices contained in the
11 % mesh.
12 % ELEMENTS M-by-3 matrix specifying the elements of the mesh, where M
13 % is equal to the number of elements contained in the mesh.
14 % EDGES P-by-2 matrix specifying the edges of the mesh.
15 % VERT2EDGE M-by-M sparse matrix which specifies whether the two
16 % vertices i and j are connected by an edge with number
17 % VERT2EDGE(i,j).
18 %
19 % Example:
20 %
21 % fig = plot_Norm_WlF(U,Mesh);
22 %
23 % Copyright 2005–2006 Patrick Meury & Mengyu Wang
24 % SAM – Seminar for Applied Mathematics
25 % ETH–Zentrum
26 % CH–8092 Zurich, Switzerland
27
28 % Initialize constant
29
30 nElements = size(Mesh.Elements,1);
31 nCoordinates = size(Mesh.Coordinates,1);

```

```

32
33 % Preallocate memory
34
35 ux = zeros(nElements,1);
36 uy = zeros(nElements,1);
37 PU = zeros(nCoordinates,1);
38 PUX = zeros(nCoordinates,1);
39 PUY = zeros(nCoordinates,1);
40
41 % Calculate modulus
42
43 for i = 1:nElements
44
45     vidx = Mesh.Elements(i,:);
46     P1 = Mesh.Coordinates(vidx(1),:);
47     P2 = Mesh.Coordinates(vidx(2),:);
48     P3 = Mesh.Coordinates(vidx(3),:);
49
50     bK = P1;
51     BK = [P2-P1;P3-P1];
52     TK = transpose(inv(BK));
53
54     % Locate barycenter
55
56     Bar_Node = 1/3*[P1 + P2 + P3];
57
58     % Compute velocity field at barycenters
59
60     eidx = [Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3)) ...
61             Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1)) ...
62             Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))];
63
64     % Determine edge orientation
65
66     if(Mesh.Edges(eidx(1),1) == vidx(2))
67         p1 = 1;
68     else
69         p1 = -1;
70     end
71
72     if(Mesh.Edges(eidx(2),1) == vidx(3))
73         p2 = 1;
74     else
75         p2 = -1;
76     end
77
78     if(Mesh.Edges(eidx(3),1) == vidx(1))
79         p3 = 1;
80     else
81         p3 = -1;
82     end
83
84     % Compute velocity field at barycenters
85

```

```

86         N = shap_WlF(Bar.Node);
87         NS(1:2) = N(1:2)*TK;
88         NS(3:4) = N(3:4)*TK;
89         NS(5:6) = N(5:6)*TK;
90
91         ux(i) = U(eidx(1))*p1*NS(1) + ...
92                 U(eidx(2))*p2*NS(3) + ...
93                 U(eidx(3))*p3*NS(5);
94
95         uy(i) = U(eidx(1))*p1*NS(2) + ...
96                 U(eidx(2))*p2*NS(4) + ...
97                 U(eidx(3))*p3*NS(6);
98
99     end
100
101     % Calculate value on each vertice
102
103     Mesh = add.Patches(Mesh);
104
105     for i = 1:nCoordinates
106         L_patch = Mesh.AdjElements(i,:);
107         loc = find(L_patch>0);
108         Eidx = L_patch(loc);
109         PUX(i) = sum(ux(Eidx))/Mesh.nAdjElements(i);
110         PUY(i) = sum(uy(Eidx))/Mesh.nAdjElements(i);
111     end
112
113     PU = sqrt(PUX.^2+PUY.^2);
114
115     % Plot solution
116
117     %%%%%%%%%Added by me%%%%%%%%%%%%%%
118     subplot(fig1);
119     hold on
120     title(titstr);
121     quiver(Mesh.Coordinates(:,1),Mesh.Coordinates(:,2),PUX,PUY,0.75,'b-');
122     plot(Mesh.BdEdges_x,Mesh.BdEdges_y,'r-');
123     axis(BBox*1.01);
124     hold off
125     %fig = plot_LFE(PU,Mesh,h);
126     plot_LFE(PU,Mesh,fig2);
127     % Assign output arguments
128
129     if(nargout > 0)
130         varargout{1} = fig;
131     end
132
133     return

```

3.2.2 Mesh

3.2.2.1 sqr_str_gen.m


```

1 function Mesh=sqr_str_gen(NREFS)
2 % Generates a square structured mesh of the unit square
3
4 % Copyright 2005–2005 Patrick Meury & Kah-Ling Sia
5 % SAM – Seminar for Applied Mathematics
6 % ETH–Zentrum
7 % CH–8092 Zurich, Switzerland
8
9 % Initialize constants
10
11 %NREFS = 4; % Number of uniform red refinements
12
13 % Load mesh from file
14
15 Mesh = load_Mesh('Coord_Sqr.dat','Elem_Sqr.dat');
16
17 % Add edge data structure
18
19 Mesh = add_Edges(Mesh);
20 Loc = get_BdEdges(Mesh);
21 Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
22 Mesh.BdFlags(Loc) = -1;
23
24 % Do NREFS uniform refinement steps
25
26 for i = 1:NREFS
27     Mesh = refine_REG(Mesh);
28 end

```

3.2.2.2 Lshap_str_gen.m

```

1 function Mesh=Lshap_str_gen(NREFS)
2 % Generates a triangular structured mesh of a L-shaped domain
3
4 % Copyright 2005–2005 Patrick Meury & Kah-Ling Sia
5 % SAM – Seminar for Applied Mathematics
6 % ETH–Zentrum
7 % CH–8092 Zurich, Switzerland
8
9 % Initialize constants
10
11 %NREFS = 4; % Number of uniform red refinements
12
13 % Load mesh from file
14
15 Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
16
17 % Add element flags
18
19 % Mesh.ElemFlag = [1 2 3 4]';
20
21 % Add edge data structure

```

```

22
23 Mesh = add_Edges(Mesh);
24 Loc = get_BdEdges(Mesh);
25 Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
26 Mesh.BdFlags(Loc) = -1;
27
28 % Do NREFS uniform refinement steps
29
30 for i = 1:NREFS
31     Mesh = refine_REG(Mesh);
32 end

```

3.2.2.3 load_Mesh.m

```

1 function Mesh = load_Mesh(CoordFile,ElemFile)
2 % LOAD_MESH Load mesh from file.
3 %
4 % MESH = LOAD_MESH(COORDFILE,ELEMFIL) loads a mesh from the files COORDFILE
5 % (list of vertices) and ELEMFIL (list of elements).
6 %
7 % The struct MESH contains the following fields:
8 % COORDINATES M-by-2 matrix specifying the vertices of the mesh.
9 % ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the mesh.
10 %
11 % Example:
12 %
13 % Mesh = load_Mesh('Coordinates.dat','Elements.dat');
14
15 % Copyright 2005–2005 Patrick Meury
16 % SAM – Seminar for Applied Mathematics
17 % ETH–Zentrum
18 % CH–8092 Zurich, Switzerland
19
20 % Load mesh from files
21
22 Mesh.Coordinates = load_Coordinates(CoordFile);
23 Mesh.Elements = load_Elements(ElemFile);
24
25 return
26
27 %%%
28
29 function Coordinates = load_Coordinates(File)
30 % LOAD_COORDINATES Load vertex coordinates from file.
31 %
32 % COORDINATES = LOAD_COORDINATES(FILE) loads the vertex coordinates from
33 % the .dat file FILE.
34 %
35 % Example:
36 %
37 % Coordinates = load_Coordinates('Coordinates.dat');
38 %

```

```

39
40 % Copyright 2005–2005 Patrick Meury
41 % SAM – Seminar for Applied Mathematics
42 % ETH-Zentrum
43 % CH-8092 Zurich, Switzerland
44
45 Coordinates = load(File);
46 Coordinates(:,1) = [];
47
48 return
49
50 %%%
51
52 function Elements = load.Elements(File)
53 % LOAD-ELEMENTS Load elements from a file.
54 %
55 % ELEMENTS = LOAD-ELEMENTS(FILE) load the elements of a mesh from the
56 % .dat file FILE.
57 %
58 % Example:
59 %
60 % Elements = load.Elements('Elements.dat');
61 %
62
63 % Copyright 2005–2005 Patrick Meury
64 % SAM – Seminar for Applied Mathematics
65 % ETH-Zentrum
66 % CH-8092 Zurich, Switzerland
67
68 Elements = load(File);
69 Elements(:,1) = [];
70
71 return

```

3.2.2.4 refine REG.m

```

1 function New_Mesh = refine_REG(Old_Mesh,varargin)
2 % REFINES REG Regular refinement.
3 %
4 % MESH = REFINES_REG(MESH) performs one regular red refinement step with the
5 % struct MESH.
6 %
7 % MESH = REFINES_REG(MESH,DHANDLE) performs one regular red refinement step
8 % with the struct MESH. During red refinement the signed distance function
9 % DHANDLE (function handle/inline object) is used to project the new vertices
10 % on the boundary edges onto the boundary of the domain.
11 %
12 % The struct MESH should at least contain the following fields:
13 % COORDINATES M-by-2 matrix specifying the vertices of the mesh.
14 % ELEMENTS N-by-3 or N-by-4 matrix specifying the elements of the mesh.
15 % EDGES P-by-2 matrix specifying the edges of the mesh.
16 % BDFLAGS P-by-1 matrix specifying the boundary type of each boundary

```

```

17 %           edge in the mesh.
18 %     VERT2EDGE     M-by-M sparse matrix which specifies whether the two vertices
19 %                   i and j are connected by an edge with number VERT2EDGE(i,j).
20 %
21 %     MESH = REFINE_REG(MESH,DHANDLE,DPARAM) also handles the variable argument
22 %     list DPARAM to the signed distance function DHANDLE.
23 %
24 %     Example:
25 %
26 %     Mesh = refine_REG(Mesh,@dist_circ,[0 0],1);
27 %
28 %     See also ADD_EDGES.
29
30 %     Copyright 2005–2005 Patrick Meury
31 %     SAM – Seminar for Applied Mathematics
32 %     ETH-Zentrum
33 %     CH-8092 Zurich, Switzerland
34
35 nCoordinates = size(Old.Mesh.Coordinates,1);
36 [nElements,nVert] = size(Old.Mesh.Elements);
37 nEdges = size(Old.Mesh.Edges,1);
38 nBdEdges = size(find(Old.Mesh.BdFlags < 0),1);
39
40 % Extract input arguments
41
42 if(nargin > 1)
43     DHandle = varargin{1};
44     DParam = varargin(2:nargin-1);
45 else
46     DHandle = [];
47 end
48
49 % Red refinement for triangular or triangular elements
50
51 if(nVert == 3)
52
53     % Preallocate memory
54
55     New_Mesh.Coordinates = zeros(nCoordinates+nEdges,2);
56     New_Mesh.Elements = zeros(4*nElements,3);
57     New_Mesh.BdFlags = zeros(2*nEdges+3*nElements,1);
58     if(isfield(Old.Mesh,'ElemFlag'))
59         New_Mesh.ElemFlag = zeros(4*nElements,1);
60     end
61
62     % Do regular red refinement
63
64     New_Mesh.Coordinates(1:nCoordinates,:) = Old.Mesh.Coordinates;
65     Bd_Idx = 0;
66     Aux = zeros(nBdEdges,4);
67     for i = 1:nElements
68
69         % Get vertex numbers of the current element
70

```

```

71     i1 = Old_Mesh.Elements(i,1);
72     i2 = Old_Mesh.Elements(i,2);
73     i3 = Old_Mesh.Elements(i,3);
74
75     % Compute vertex numbers of new vertices localized on edges
76
77     j1 = nCoordinates+Old_Mesh.Vert2Edge(i2,i3);
78     j2 = nCoordinates+Old_Mesh.Vert2Edge(i3,i1);
79     j3 = nCoordinates+Old_Mesh.Vert2Edge(i1,i2);
80
81     % Generate new elements
82
83     New_Mesh.Elements(4*(i-1)+1,:) = [i1 j3 j2];
84     New_Mesh.Elements(4*(i-1)+2,:) = [j3 i2 j1];
85     New_Mesh.Elements(4*(i-1)+3,:) = [j2 j1 i3];
86     New_Mesh.Elements(4*(i-1)+4,:) = [j1 j2 j3];
87
88     % Generate new vertex on edge 1, project to boundary if necessary
89
90     BdFlag_1 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i2,i3));
91     if(BdFlag_1 < 0)
92         if(~isempty(DHandle))
93             DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i2,:)-...
94                 Old_Mesh.Coordinates(i3,:));
95             x = 1/2*(Old_Mesh.Coordinates(i2,:)+Old_Mesh.Coordinates(i3,:));
96             dist = feval(DHandle,x,DParam{:});
97             grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
98                 feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
99             New_Mesh.Coordinates(j1,:) = x-dist*grad_dist;
100         else
101             New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i2,:)+...
102                 Old_Mesh.Coordinates(i3,:));
103         end
104         Bd.Idx = Bd.Idx+1;
105         Aux(Bd.Idx,:) = [BdFlag_1 i2 j1 i3];
106     else
107         New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i2,:)+...
108             Old_Mesh.Coordinates(i3,:));
109     end
110
111     % Generate new vertex on edge 2, project to boundary if necessary
112
113     BdFlag_2 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i3,i1));
114     if(BdFlag_2 < 0)
115         if(~isempty(DHandle))
116             DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i3,:)-...
117                 Old_Mesh.Coordinates(i1,:));
118             x = 1/2*(Old_Mesh.Coordinates(i3,:)+Old_Mesh.Coordinates(i1,:));
119             dist = feval(DHandle,x,DParam{:});
120             grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
121                 feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
122             New_Mesh.Coordinates(j2,:) = x-dist*grad_dist;
123         else
124             New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+...

```

```

125         +Old_Mesh.Coordinates(i1,:));
126     end
127     Bd_Idx = Bd_Idx+1;
128     Aux(Bd_Idx,:) = [BdFlag_2 i3 j2 i1];
129 else
130     New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+Old_Mesh.Coordinates
131 end
132
133 % Generate new vertex on edge 3, project to boundary if necessary
134
135 BdFlag_3 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i1,i2));
136 if(BdFlag_3 < 0)
137     if(~isempty(DHandle))
138         DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i1,:)-Old_Mesh.Coordinates(i2,:));
139         x = 1/2*(Old_Mesh.Coordinates(i1,:)+Old_Mesh.Coordinates(i2,:));
140         dist = feval(DHandle,x,DParam{:});
141         grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
142             feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
143         New_Mesh.Coordinates(j3,:) = x-dist*grad_dist;
144     else
145         New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i1,...
146             +Old_Mesh.Coordinates(i2,:));
147     end
148     Bd_Idx = Bd_Idx+1;
149     Aux(Bd_Idx,:) = [BdFlag_3 i1 j3 i2];
150 else
151     New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i1,:)+Old_Mesh.Coordinates
152 end
153
154 % Update element flag if defined
155
156 if(isfield(Old_Mesh,'ElemFlag'))
157     New_Mesh.ElemFlag(4*(i-1)+1) = Old_Mesh.ElemFlag(i);
158     New_Mesh.ElemFlag(4*(i-1)+2) = Old_Mesh.ElemFlag(i);
159     New_Mesh.ElemFlag(4*(i-1)+3) = Old_Mesh.ElemFlag(i);
160     New_Mesh.ElemFlag(4*(i-1)+4) = Old_Mesh.ElemFlag(i);
161 end
162 end
163
164 % Add edges to new mesh
165
166 New_Mesh = add_Edges(New_Mesh);
167
168 % Update boundary flags
169
170 for i = 1:nBdEdges
171     New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,2),Aux(i,3))) = Aux(i,1);
172     New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,3),Aux(i,4))) = Aux(i,1);
173 end
174
175 else
176
177 % Preallocate memory
178

```

```

179 New_Mesh.Coordinates = zeros(nCoordinates+nEdges,2);
180 New_Mesh.Elements = zeros(4*nElements,4);
181 New_Mesh.BdFlags = zeros(2*nEdges+4*nElements,1);
182 if(isfield(Old_Mesh,'ElemFlag'))
183     New_Mesh.ElemFlag = zeros(4*nElements,1);
184 end
185
186 % Do regular red refinement
187
188 New_Mesh.Coordinates(1:nCoordinates,:) = Old_Mesh.Coordinates;
189 Bd_Idx = 0;
190 Aux = zeros(nBdEdges,4);
191 for i = 1:nElements
192
193     % Get vertex numbers of the current element
194
195     i1 = Old_Mesh.Elements(i,1);
196     i2 = Old_Mesh.Elements(i,2);
197     i3 = Old_Mesh.Elements(i,3);
198     i4 = Old_Mesh.Elements(i,4);
199
200     % Compute vertex numbers of new vertices localized on edges
201
202     j1 = nCoordinates+Old_Mesh.Vert2Edge(i1,i2);
203     j2 = nCoordinates+Old_Mesh.Vert2Edge(i2,i3);
204     j3 = nCoordinates+Old_Mesh.Vert2Edge(i3,i4);
205     j4 = nCoordinates+Old_Mesh.Vert2Edge(i4,i1);
206
207     % Compute vertex number of new vertex in the interior of each element
208
209     jc = nCoordinates+nEdges+i;
210
211     % Generate new elements
212
213     New_Mesh.Elements(4*(i-1)+1,:) = [i1 j1 jc j4];
214     New_Mesh.Elements(4*(i-1)+2,:) = [j1 i2 j2 jc];
215     New_Mesh.Elements(4*(i-1)+3,:) = [j4 jc j3 i4];
216     New_Mesh.Elements(4*(i-1)+4,:) = [jc j2 i3 j3];
217
218     % Generate new vertex on edge 1, project to boundary if necessary
219
220     BdFlag_1 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i1,i2));
221     if(BdFlag_1 < 0)
222         if(~isempty(DHandle))
223             DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i2,:)-...
224                 Old_Mesh.Coordinates(i1,:));
225             x = 1/2*(Old_Mesh.Coordinates(i1,:)+Old_Mesh.Coordinates(i2,:));
226             dist = feval(DHandle,x,DParam{:});
227             grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
228                 feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
229             New_Mesh.Coordinates(j1,:) = x-dist*grad_dist;
230         else
231             New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i1,:)+...
232                 Old_Mesh.Coordinates(i2,:));

```

```

233         end
234         Bd.Idx = Bd.Idx+1;
235         Aux(Bd.Idx,:) = [BdFlag_1 i1 j1 i2];
236     else
237         New_Mesh.Coordinates(j1,:) = 1/2*(Old_Mesh.Coordinates(i1,:)+...
238             Old_Mesh.Coordinates(i2,:));
239     end
240
241     % Generate new vertex on edge 2, project to boundary if necessary
242
243     BdFlag_2 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i2,i3));
244     if(BdFlag_2 < 0)
245         if(~isempty(DHandle))
246             DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i3,:)-...
247                 Old_Mesh.Coordinates(i2,:));
248             x = 1/2*(Old_Mesh.Coordinates(i2,:)+Old_Mesh.Coordinates(i3,:));
249             dist = feval(DHandle,x,DParam{:});
250             grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
251                 feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
252             New_Mesh.Coordinates(j2,:) = x-dist*grad_dist;
253         else
254             New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i2,:)+...
255                 Old_Mesh.Coordinates(i3,:));
256         end
257         Bd.Idx = Bd.Idx+1;
258         Aux(Bd.Idx,:) = [BdFlag_2 i2 j2 i3];
259     else
260         New_Mesh.Coordinates(j2,:) = 1/2*(Old_Mesh.Coordinates(i2,:)+...
261             Old_Mesh.Coordinates(i3,:));
262     end
263
264     % Generate new vertex on edge 3, project to boundary if necessary
265
266     BdFlag_3 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i3,i4));
267     if(BdFlag_3 < 0)
268         if(~isempty(DHandle))
269             DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i4,:)-...
270                 Old_Mesh.Coordinates(i3,:));
271             x = 1/2*(Old_Mesh.Coordinates(i3,:)+Old_Mesh.Coordinates(i4,:));
272             dist = feval(DHandle,x,DParam{:});
273             grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:}) ...
274                 feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
275             New_Mesh.Coordinates(j3,:) = x-dist*grad_dist;
276         else
277             New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+...
278                 Old_Mesh.Coordinates(i4,:));
279         end
280         Bd.Idx = Bd.Idx+1;
281         Aux(Bd.Idx,:) = [BdFlag_3 i3 j3 i4];
282     else
283         New_Mesh.Coordinates(j3,:) = 1/2*(Old_Mesh.Coordinates(i3,:)+...
284             Old_Mesh.Coordinates(i4,:));
285     end
286

```



```

287     % Generate new vertex on egde 4, prohject to boundary if necessary
288
289     BdFlag_4 = Old_Mesh.BdFlags(Old_Mesh.Vert2Edge(i4,i1));
290     if(BdFlag_4 < 0)
291         if(~isempty(DHandle))
292             DEPS = sqrt(eps)*norm(Old_Mesh.Coordinates(i1,:)-...
293                 Old_Mesh.Coordinates(i4,:));
294             x = 1/2*(Old_Mesh.Coordinates(i4,:)+Old_Mesh.Coordinates(i1,:));
295             dist = feval(DHandle,x,DParam{:});
296             grad_dist = ([feval(DHandle,x+[DEPS 0],DParam{:})...
297                 feval(DHandle,x+[0 DEPS],DParam{:})]-dist)/DEPS;
298             New_Mesh.Coordinates(j4,:) = x-dist*grad_dist;
299         else
300             New_Mesh.Coordinates(j4,:) = 1/2*(Old_Mesh.Coordinates(i4,:)+...
301                 Old_Mesh.Coordinates(i1,:));
302         end
303         Bd_Idx = Bd_Idx+1;
304         Aux(Bd_Idx,:) = [BdFlag_4 i4 j4 i1];
305     else
306         New_Mesh.Coordinates(j4,:) = 1/2*(Old_Mesh.Coordinates(i4,:)+...
307             Old_Mesh.Coordinates(i1,:));
308     end
309
310     % Generate new vertex in the interior
311
312     New_Mesh.Coordinates(jc,:) = 1/4*(New_Mesh.Coordinates(j1,:)+...
313         New_Mesh.Coordinates(j2,:)+New_Mesh.Coordinates(j3,:)+...
314         New_Mesh.Coordinates(j4,:));
315
316     % Update element flag if defined
317
318     if(isfield(Old_Mesh,'ElemFlag'))
319         New_Mesh.ElemFlag(4*(i-1)+1) = Old_Mesh.ElemFlag(i);
320         New_Mesh.ElemFlag(4*(i-1)+2) = Old_Mesh.ElemFlag(i);
321         New_Mesh.ElemFlag(4*(i-1)+3) = Old_Mesh.ElemFlag(i);
322         New_Mesh.ElemFlag(4*(i-1)+4) = Old_Mesh.ElemFlag(i);
323     end
324 end
325
326 % Add edges to new mesh
327
328 New_Mesh = add_Edges(New_Mesh);
329
330 % Update boundary flags
331
332 for i = 1:nBdEdges
333     New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,2),Aux(i,3))) = Aux(i,1);
334     New_Mesh.BdFlags(New_Mesh.Vert2Edge(Aux(i,3),Aux(i,4))) = Aux(i,1);
335 end
336 end
337
338 return

```

3.2.3 Viewing results

3.2.3.1 replay.m

```
1
2 % Rendering of nodal/edge element solution for
3 % transient Maxwell problem in a cavity
4
5 % Initialize constants
6
7 InitREF = 2; % size of the Initial Mesh
8 NREFSs = 5; % Number of uniform mesh refinements
9 finaltime = 3;
10 timestep = 0.01;
11 step=1;
12 makemovie =0; % set to 1 to make avi files
13 rect=[100 100 1024 768]; % movie size (not length)
14
15 % Generate initial meshes, where the meshwidth depends on InitREF
16 %Square mesh
17 MeshS=sqr_str_gen(InitREF);
18 %Add to the mesh some useful information to handle edge elements
19 MeshS.ElemFlag = ones(size(MeshS.Elements,1),1);
20 MeshS = add_Edges(MeshS);
21 LocS = get_BdEdges(MeshS);
22 MeshS.BdFlags = zeros(size(MeshS.Edges,1),1);
23 MeshS.BdFlags(LocS) = -1;
24
25 %L-shaped mesh
26 MeshL=Lshap_str_gen(InitREF);
27 %Add to the mesh some useful information to handle edge elements
28 MeshL.ElemFlag = ones(size(MeshL.Elements,1),1);
29 MeshL = add_Edges(MeshL);
30 LocL = get_BdEdges(MeshL);
31 MeshL.BdFlags = zeros(size(MeshL.Edges,1),1);
32 MeshL.BdFlags(LocL) = -1;
33
34 % Do NREFS uniform refinement steps
35
36 for i = 1:NREFSs
37
38
39
40     % For the square mesh
41     % Refine Mesh
42     MeshS = refine_REG(MeshS);
43     % plot it
44     plot_Mesh(MeshS, 'as')
45
46     % Write some necessary information in the mesh
47     [MeshS.BdEdges_x MeshS.BdEdges_y]=dataBoundaryPlot(MeshS);
48     MeshS=setBdFlags(MeshS);
```

```

49     Loc = get_BdEdges(MeshS);
50     NDofs = [2*find(MeshS.VertBdFlags(:,1) == 0); 2*find(MeshS.VertBdFlags(:,2) == 0)-1];
51     DEdges = Loc(MeshS.BdFlags(Loc) == -1);
52     EDofs = setdiff(1:size(MeshS.Edges,1),DEdges);
53
54     %Loading Data
55     Sq_str1=['Square' int2str(i)];
56     Sq_str=['Square' int2str(i) '_res'];
57     load(Sq_str);
58     N=length(times);
59
60     if (size(sol_v,2) ~= N), error('Wrong number of samples in sol_v'); end
61     if (size(sol_e,2) ~= N), error('Wrong number of samples in sol_e'); end
62
63     if(makemovie) moviename=[Sq_str '.avi'];
64     aviobj = avifile(moviename,'fps',10); end;
65     figno=figure('position',rect);
66     set(gcf,'nextplot','replace');
67     axis off;
68     if(makemovie) Sqr = getframe(figno); aviobj = addframe(aviobj,Sqr); end;
69
70     for j=1:step:N/30
71
72         plotiteratel(MeshS,sol_e(:,j),sol_v(:,j),times(j),figno,NDofs,EDofs);
73
74         % add Energy information
75         [a,I]=min(abs(times(j)-en(:,1)));
76         subplot(2,2,3,'Parent',figno);
77         title(sprintf('Time = %f      Etot = %f ',times(j),en(I,4)+en(I,5) ));
78
79         subplot(2,2,4,'Parent',figno);
80         title(sprintf('Time = %f      Etot = %f ',times(j),en(I,2)+en(I,3)));
81         if(makemovie)Sqr = getframe(gcf); aviobj = addframe(aviobj,Sqr); end;
82
83     end
84     if(makemovie)aviobj = close(aviobj); end;
85
86
87     %plot energy evolution in time
88     figure; clf;
89     subplot(1,2,1);
90     plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
91     legend('Nodal scheme','Edge elements');
92     title([Sq_str1,': Electric energy']);
93     xlabel('time');
94     subplot(1,2,2);
95     plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
96     legend('Nodal scheme','Edge elements');
97     title([Sq_str1,': Magnetic energy']);
98     xlabel('time');
99     %     Sq_str1=['../Bericht/En_smoos_' Sq_str1 '.eps'];
100    %     saveas(gcf,Sq_str1,'psc2');
101    drawnow;
102    clear en sol_v sol_e times;

```

```

103     % end for the square mesh
104
105
106     %For the L-mesh
107     % Refine mesh
108     MeshL = refine_REG(MeshL);
109     plot_Mesh(MeshL, 'as')
110
111     [MeshL.BdEdges_x MeshL.BdEdges_y]=dataBoundaryPlot(MeshL);
112     MeshL=setBdFlags(MeshL);
113     Loc = get_BdEdges(MeshL);
114     NDofs = [2*find(MeshL.VertBdFlags(:,1) == 0); 2*find(MeshL.VertBdFlags(:,2) == 0)-1];
115     DEdges = Loc(MeshL.BdFlags(Loc) == -1);
116     EDofs = setdiff(1:size(MeshL.Edges,1),DEdges);
117     %Loading Data
118
119     L_str1=['Lshape' int2str(i)];
120     L_str=['Lshape' int2str(i) '.res'];
121     load(L_str);
122     N=length(times);
123
124     if(makemovie) moviename=[L_str '.avi']; aviobj = avifile(moviename,'fps',10); end;
125     figno=figure('position',rect);
126     set(gcf,'nextplot','replace');
127     axis off;
128     if(makemovie)L_shape = getframe(figno); aviobj = addframe(aviobj,L_shape);end
129
130     for j=1:step:N/10
131         plotiteratel(MeshL,sol_e(:,j),sol_v(:,j),times(j),figno,NDofs,EDofs);
132
133         % add Energy information
134         [a,I]=min(abs(times(j)-en(:,1)));
135         subplot(2,2,3,'Parent',figno);
136         title(sprintf('Time = %f      Etot = %f ',times(j),en(I,4)+en(I,5) ));
137
138         subplot(2,2,4,'Parent',figno);
139         title(sprintf('Time = %f      Etot = %f ',times(j),en(I,2)+en(I,3)));
140
141         if(makemovie)L_shape = getframe(figno); aviobj = addframe(aviobj,L_shape); end;
142     end
143     if(makemovie) aviobj = close(aviobj); end;
144
145     % Actualize energy plot
146     figure; clf;
147     subplot(1,2,1);
148     plot(en(:,1),en(:,2),'r-',en(:,1),en(:,4),'b-');
149     legend('Nodal scheme','Edge elements');
150     title([L_str1,': Electric energy']);
151     xlabel('time');
152     subplot(1,2,2);
153     plot(en(:,1),en(:,3),'r-',en(:,1),en(:,5),'b-');
154     legend('Nodal scheme','Edge elements');
155     title([L_str1,': Magnetic energy']);
156     xlabel('time');

```

```
157     drawnow;
158     clear en solv sol_e times;
159     %     L_str1=['../Bericht/En-smoo_' L_str1 '.eps'];
160     %     saveas(gcf,L_str1,'psc2');
161 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.

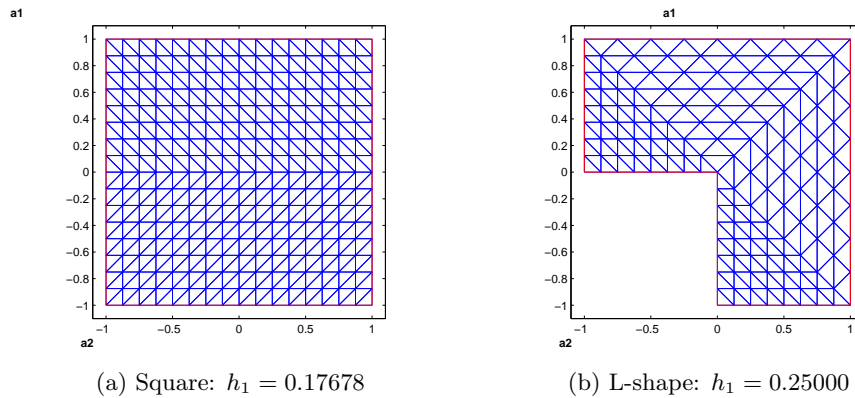


Figure 4.1: Initial meshes.

		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 $\text{div } E_0 = 0$, 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 Conditions¹

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

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

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

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

¹taked from [3]

Then, we introduce

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

and see

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

Now, let us introduce the cut-off function,

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

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

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

Note that

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

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

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

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 $\tilde{\mathbf{E}}^0$ 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 $i = 1, \dots, 5$. 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.

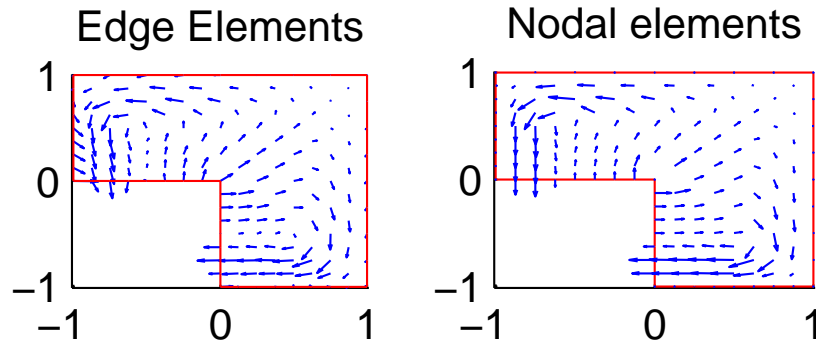


Figure 4.2: Discrete singular initial field on the L-shaped domain

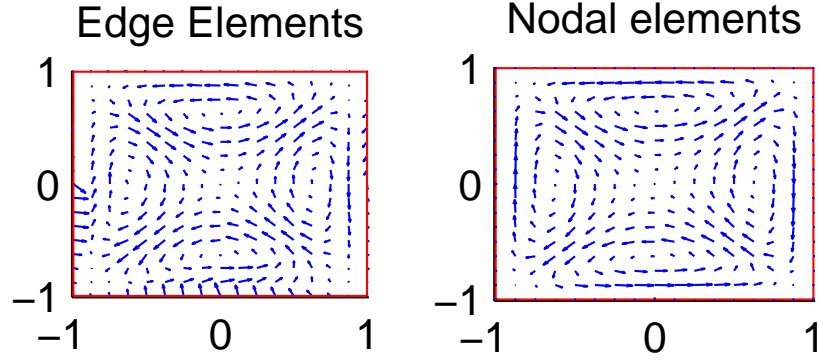


Figure 4.3: Discrete singular initial field on the square domain

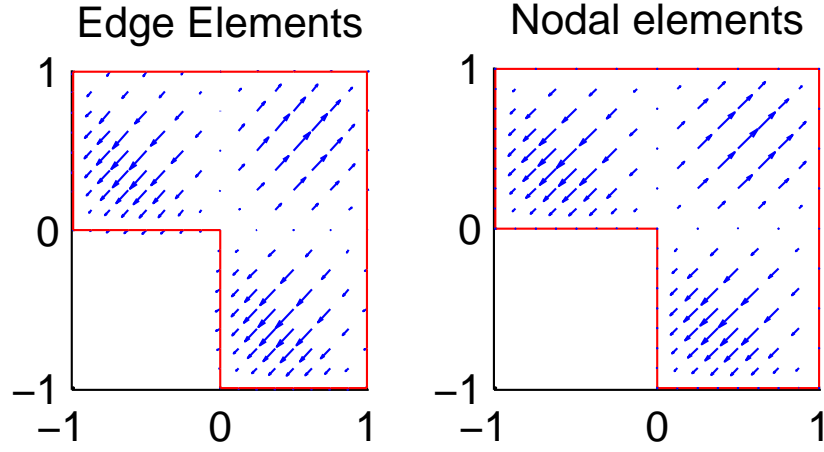


Figure 4.4: Discrete smooth initial field on the L-shaped domain

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

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

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

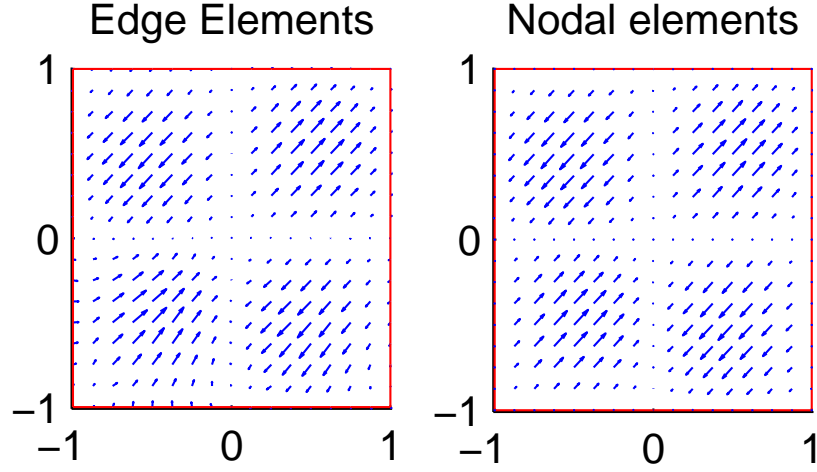
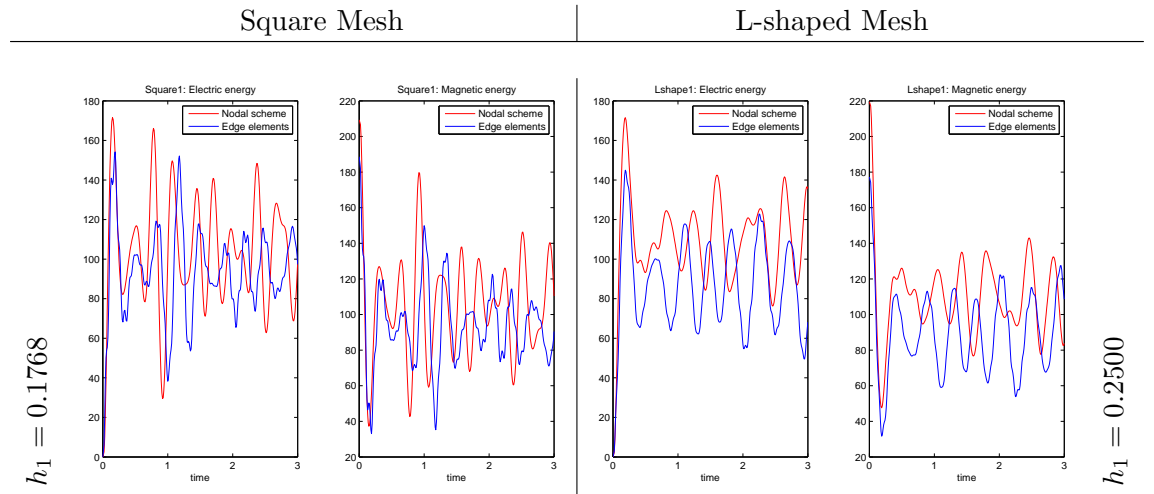
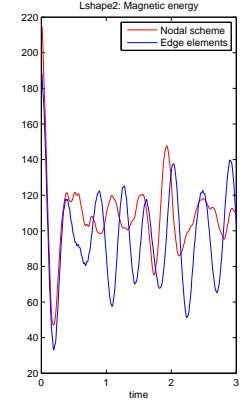
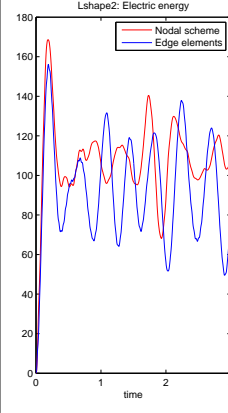
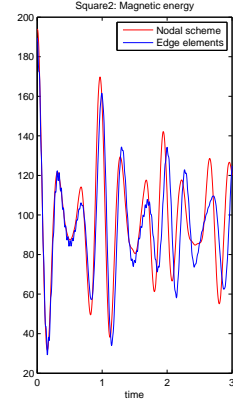
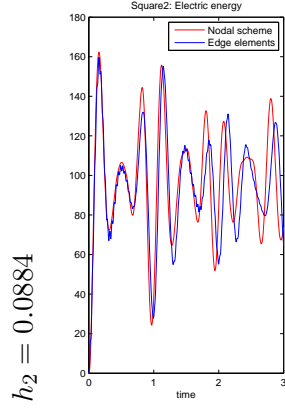


Figure 4.5: Discrete smooth initial field on the square domain

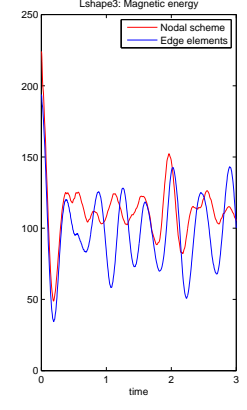
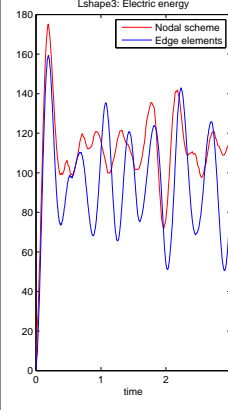
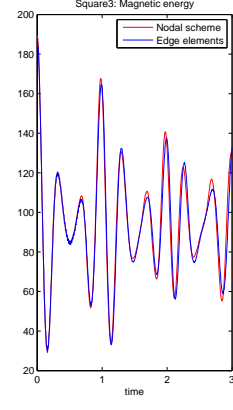
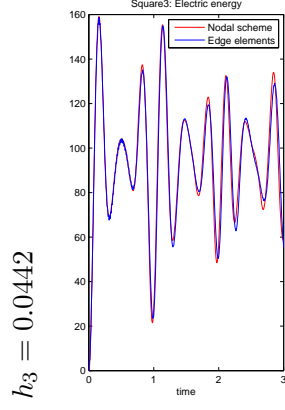
4.4.1 Energy Behaviour Using a Singular Initial Function

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

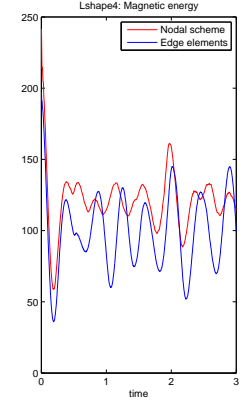
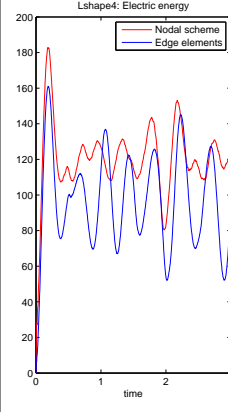
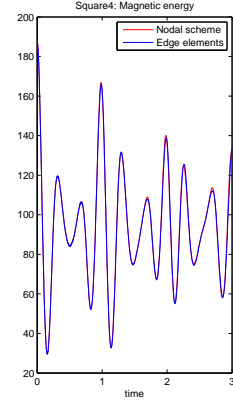
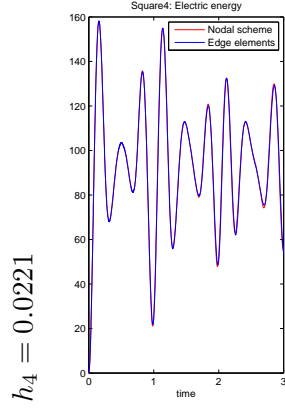




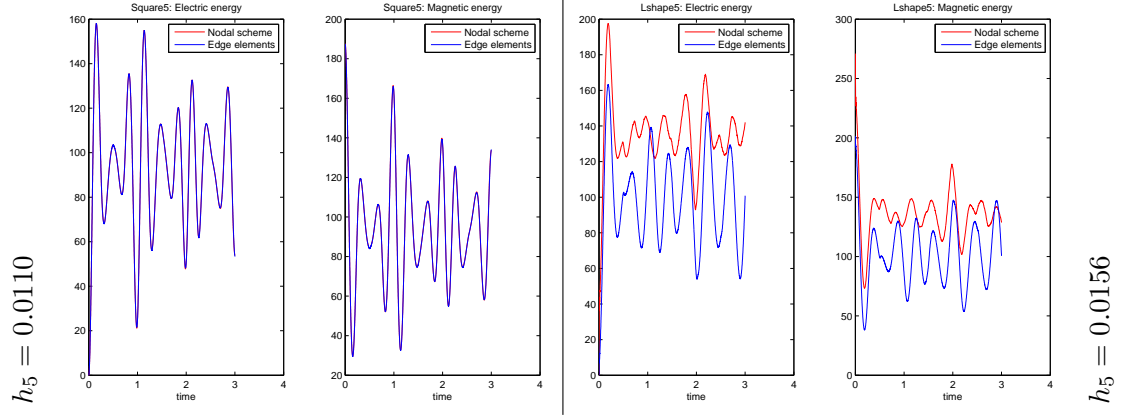
$h_2 = 0.1250$



$h_3 = 0.0625$

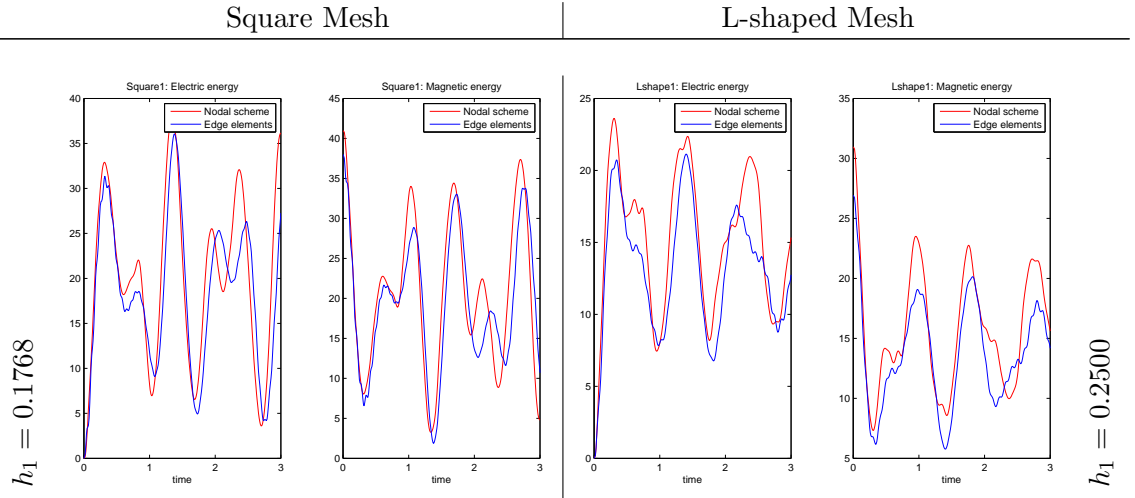


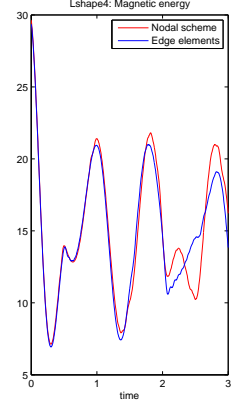
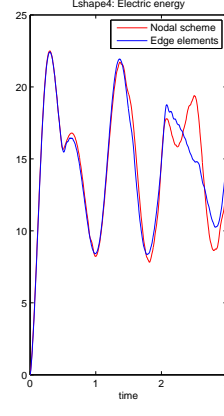
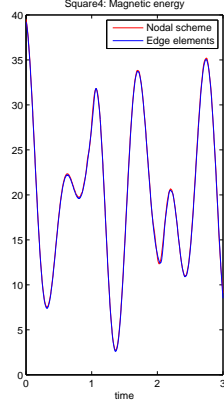
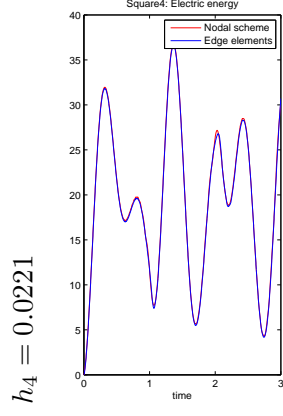
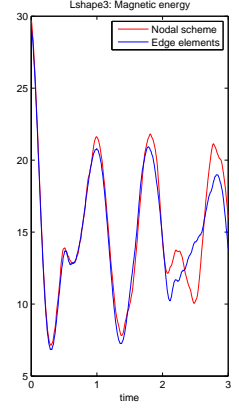
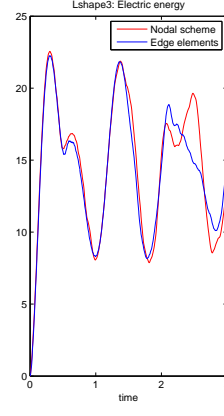
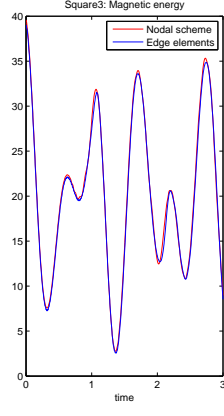
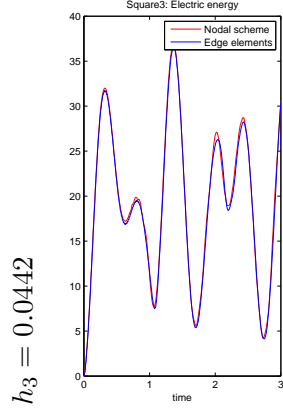
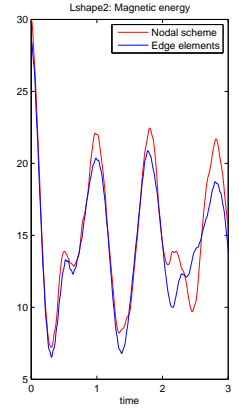
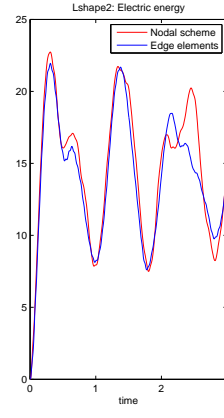
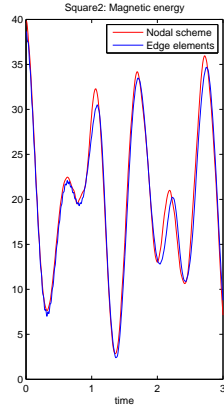
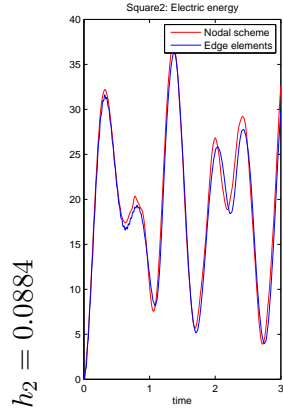
$h_4 = 0.0312$



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.

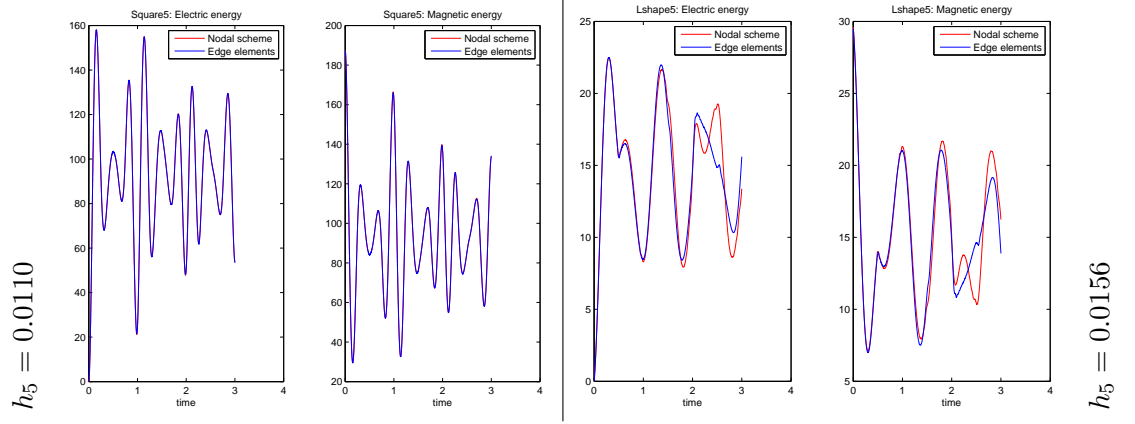




$h_2 = 0.1250$

$h_3 = 0.0625$

$h_4 = 0.0312$



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, \dots, h_4 with the approximation obtained for h_5 .

$$\text{error}_i := \|\mathbf{E}_{h_5}^T - \mathbf{E}_{h_i}^T\|_2 \quad \text{for } i = 1, \dots, 4.$$

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.

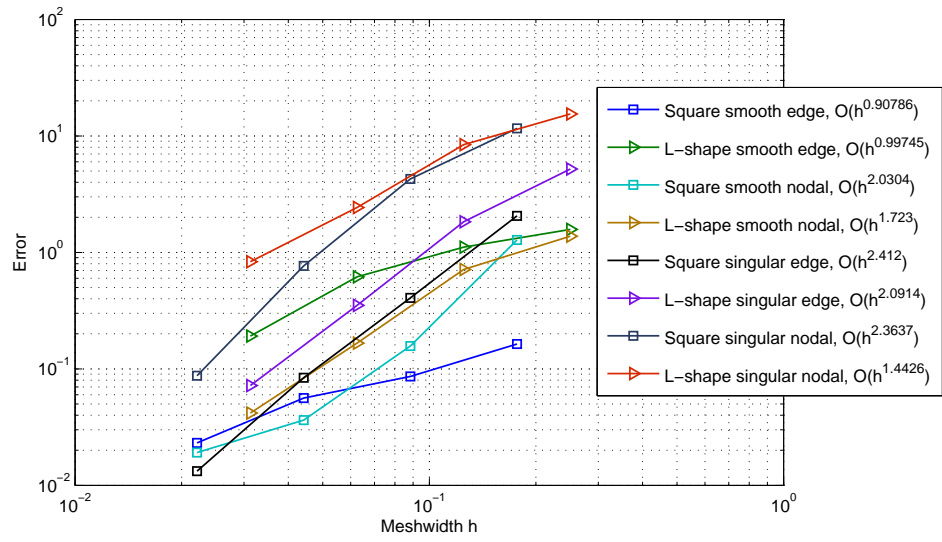


Figure 4.6: Convergence comparison between the approximations using different domains, elements and initial conditions

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*,