ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Auxiliary Space Method for Edge Elements

Gisela Widmer

Diploma Thesis in Computational Science and Engineering
WS 2003/04
Supervisor: Prof. R. Hiptmair
Seminar for Applied Mathematics, ETH Zürich

Contents

1 Introduction

When solving time-dependent Maxwell's equations in two dimensions, the differential equation

$$\begin{aligned}\mathbf{curl}\,\mathrm{curl}\,\boldsymbol{E}(\boldsymbol{x})+\tau\,\boldsymbol{E}(\boldsymbol{x}) &= \boldsymbol{f}(\boldsymbol{x}),\ \tau>0,\ \boldsymbol{x}\in\Omega\subset\mathbb{R}^2\\ \boldsymbol{E}(\boldsymbol{x})\times\boldsymbol{n}(\boldsymbol{x}) &= 0,\ \boldsymbol{x}\text{ on }\partial\Omega\end{aligned}$$

has to be solved in each time step [1]. Discretizing the equation with the finite edge element method described in section 2 results in a linear system $\boldsymbol{A}\,\boldsymbol{x}=\boldsymbol{b}$, with $\boldsymbol{A}$ symmetric, positive definite and sparse. This system can then be solved by a standard linear solver like e.g. the conjugate gradient method (CG). The main problem of this strategy is that the condition number of the matrix $\boldsymbol{A}$ scales like h^{-2} for shape-regular and quasi-unform meshes, where h is the length of an element edge. This deteriorates the convergence of the CG method for h $\rightarrow 0$, since

$$\frac{||\boldsymbol{e}^k||_A}{||\boldsymbol{e}^0||_A}\leq 2\Big(\frac{\sqrt{\chi(\boldsymbol{A})}-1}{\sqrt{\chi(\boldsymbol{A})}+1}\Big)^k,$$

where $\chi(\boldsymbol{A})$ is the condition number of the matrix $\boldsymbol{A}$ and $\boldsymbol{e}^k$ the error in the k^{th} step.

The goal of this diploma thesis is to test and to analyse a method for preconditioning the system to obtain a CG convergence rate independent of the system size. This is done by applying the 'Auxiliary Space Method' proposed by Xu ([2],[3]) as it is described in section 3. The main idea of this method is the transfer of the problem to a regular auxiliary mesh where it can be solved or preconditioned in a more efficient way by multigrid methods. This is known to work for standard finite elements and is tested for edge elements in this diploma thesis.

The main focus lies on the scaling properties of the method for shape-regular and quasi-uniform meshes: the effort to solve the linear system should scale linearly with the system size. The computational costs of the critical ingredients of the algorithm are analysed in section 4, where implementation aspects are taken into account.

The numerical results are described in section 5, where properties like condition number, multigrid convergence rate and CG convergence are presented for some test examples.

2 Discretization

2.1 Differential Equation and Variational Formulation

2.1.1 Differential Equation

On a domain $\Omega \in \mathbb{R}^2$, the following differential equation is considered for $\boldsymbol{u} = \boldsymbol{u}(\boldsymbol{x}) = (u_1, u_2)^t \in \mathbb{R}^2, \boldsymbol{x} = (x_1, x_2)^t \in \Omega$:

$$\begin{aligned} \mathbf{curl}\,\mathrm{curl}\,\boldsymbol{u} + \tau\,\boldsymbol{u} &= \boldsymbol{f}, \quad \tau > 0 \\ \boldsymbol{u} \times \boldsymbol{n} &= 0, \quad \boldsymbol{x} \text{ on } \partial\Omega \end{aligned} \tag{1}$$

($\mathrm{curl}\,\boldsymbol{u} = \frac{\partial u_2}{\partial x_1} - \frac{\partial u_1}{\partial x_2}$, $\mathbf{curl}\,\phi = (\frac{\partial \phi}{\partial x_2}, -\frac{\partial \phi}{\partial x_1})^t$, $\boldsymbol{n}$: outer unit normal to Ω)

2.1.2 Variational Formulation

Multiplying (1) with a smooth test function $\boldsymbol{v}(\boldsymbol{x})$ from the right and integrating over Ω leads to

$$\int_\Omega \mathbf{curl}\,\mathrm{curl}\,\boldsymbol{u} \cdot \boldsymbol{v}\; d\boldsymbol{x} + \tau \int_\Omega \boldsymbol{u} \cdot \boldsymbol{v}\; d\boldsymbol{x} = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v}\; d\boldsymbol{x}, \tag{2}$$

where $\boldsymbol{a} \cdot \boldsymbol{b}$ is the euclidean dot product of the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.
With

$$\begin{aligned} \int_\Omega \mathbf{curl}\,\omega \cdot \boldsymbol{v}\; d\boldsymbol{x} &= \int_\Omega \frac{\partial \omega}{\partial y} v_1 - \frac{\partial \omega}{\partial x} v_2\; d\boldsymbol{x} \\ &= \int_{\partial\Omega} \omega\; (-v_2, v_1)^t \cdot \boldsymbol{n}\; d\boldsymbol{x} + \int_\Omega \omega\; \mathrm{curl}\,\boldsymbol{v}\; d\boldsymbol{x} \\ &= \int_{\partial\Omega} \omega\; (\boldsymbol{v} \times \boldsymbol{n})\; d\boldsymbol{x} + \int_\Omega \omega\; \mathrm{curl}\,\boldsymbol{v}\; d\boldsymbol{x} \\ \omega &= \mathrm{curl}\,\boldsymbol{u} \qquad \text{and} \\ \boldsymbol{v} \times \boldsymbol{n} &= \mathbf{0} \end{aligned}$$

it follows that

$$\int_\Omega \mathbf{curl}\,\mathrm{curl}\,\boldsymbol{u} \cdot \boldsymbol{v}\; d\boldsymbol{x} = \int_\Omega \mathrm{curl}\,\boldsymbol{u}\; \mathrm{curl}\,\boldsymbol{v}\; d\boldsymbol{x}.$$

Defining a bilinear form $a(\boldsymbol{u}, \boldsymbol{v})$ as

$$a(\boldsymbol{u}, \boldsymbol{v}) := \int_\Omega \mathrm{curl}\,\boldsymbol{u}\; \mathrm{curl}\,\boldsymbol{v}\; d\boldsymbol{x} + \tau \int_\Omega \boldsymbol{u} \cdot \boldsymbol{v}\; d\boldsymbol{x} \tag{3}$$

and a linear functional $f(\boldsymbol{v})$ as

$$f(\boldsymbol{v}) := \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \; d\boldsymbol{x}, \tag{4}$$

for $\boldsymbol{u}, \boldsymbol{v} \in H_0(\mathrm{curl}, \Omega) := \{\boldsymbol{v} : \int_\Omega |\boldsymbol{v}|^2 \; d\boldsymbol{x} < \infty, \int_\Omega |\mathrm{curl}\, \boldsymbol{v}|^2 \; d\boldsymbol{x} < \infty, \; \boldsymbol{v} \times \boldsymbol{n} = \boldsymbol{0}$ on $\partial\Omega\}$ the variational form of (1) reads:
Seek $\boldsymbol{u} \in H_0(\mathrm{curl}, \Omega)$ with

$$a(\boldsymbol{u}, \boldsymbol{v}) = f(\boldsymbol{v}) \quad \forall \boldsymbol{v} \in H_0(\mathrm{curl}, \Omega). \tag{5}$$

2.2 Finite Element Spaces

In order to discretize equation (5), we need a finite elements space consisting of a partition of the domain Ω into elements as well as basis functions of a space that approximates $H_0(\mathrm{curl}, \Omega)$. In this work, we always use shape-regular and quasi-uniform triangulations $\mathcal{T}_h = \{T_i\}_i$, consisting of triangles, of the domain $\Omega \subset \mathbb{R}^2$. In contrast to standard finite elements, each edge is equiped with an intrinsic orientation. The space $H_0(\mathrm{curl}, \Omega)$ is approximated with Nédélec's lowest order edge elements [4], which is the following finite element space:

$$\begin{aligned}
\mathcal{ND}_1^0(\mathcal{T}_h) &:= \{\boldsymbol{u}_h \in H_0(\mathrm{curl}, \Omega); \boldsymbol{u}_{h|T} \in \mathcal{ND}_1(T) \; \forall T \in \mathcal{T}_h\}, \text{ where} \\
\mathcal{ND}_1(T) &:= (\mathcal{P}_0(T))^2 + \{\boldsymbol{p} \in (\mathcal{P}_1(T))^2; \boldsymbol{p}(\boldsymbol{x}) \cdot \boldsymbol{x} = 0 \; \forall \boldsymbol{x} \in T\} \\
&= \{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \boldsymbol{a} + b \begin{pmatrix} x_2 \\ -x_1 \end{pmatrix}; \boldsymbol{a} \in \mathbb{R}^2, b \in \mathbb{R}, \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in T\}
\end{aligned}$$

($\mathcal{P}_i(T)$ denotes the space of polynomials of degree < i on T.)
The construction of local basis functions on a triangle T (shape functions) and of global basis functions for this finite element space are discribed in the sections 2.2.1 and 2.2.5.
The discretized problem of (5) then reads:

$$a(\boldsymbol{u}_h, \boldsymbol{v}_h) = f(\boldsymbol{v}_h) \; \forall \boldsymbol{v}_h \in \mathcal{ND}_1^0(\mathcal{T}_h). \tag{6}$$

Another finite element space, used in section 3.2.5, is the space of continuous, piecewise linear function on $\mathcal{T}_h$ that vanish on the boundary:

$$S_{1,0}^h(\mathcal{T}_h) := \{\boldsymbol{v}_h = \sum_{i=1}^{Nv} \alpha_i \lambda_i; \alpha_i \in \mathbb{R}, \lambda_i : \text{hat function of internal vertex } i\} \tag{7}$$

2.2.1 Element Shape Functions

In order to construct a basis of $\mathcal{ND}_1(T)$ for a triangle T, we denote $\boldsymbol{a}_i$ (i=1 , 2, 3) the counterclockwise corners of triangle T, e_i the directed edge (fig. 1) between $\boldsymbol{a}_{i+1}$ and $\boldsymbol{a}_{i+2}$ (all indices have to be read as mod 3+1) and $\lambda_i^T(\boldsymbol{x})$ the barycentric coordinate functions of triangle T. Then the shape functions

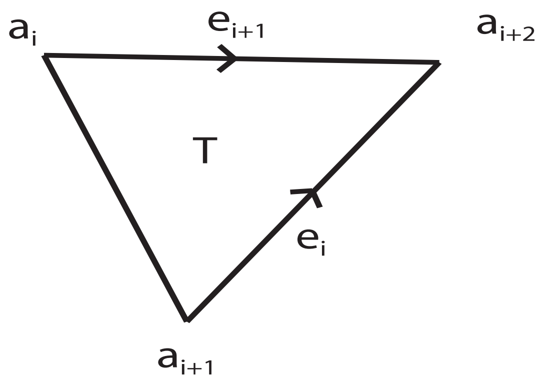

Figure 1: The edge with local index i is opposite the triangle vertex a_i.

$$\boldsymbol{b}_i^T = \lambda_{i+1}\boldsymbol{\nabla}\lambda_{i+2}^T - \lambda_{i+2}^T\boldsymbol{\nabla}\lambda_{i+1}^T, \quad i = 1, 2, 3$$

form a basis of $\mathcal{ND}_1(T)$. They satisfy:

$$\int_{e_i} \boldsymbol{b}_j^T \cdot d\boldsymbol{s} = \begin{cases} \delta_{ij} & \text{if } e_i = [a_{i+1}\ a_{i+2}] \\ -\delta_{ij} & \text{if } e_i = [a_{i+2}\ a_{i+1}]. \end{cases}$$

$\boldsymbol{u}_h|_T$ can then be written as

$$\boldsymbol{u}_h|_T \quad = \quad \sum_{j=1}^{3} \phi_j^T d_j^T \boldsymbol{b}_j^T,$$

where d_j provides infomation about the orientation of edge e_j with respect to triangle T:

$$d_j^T = \begin{cases} 1 & (e_j \text{ counterclockwise directed}) \\ -1 & (\text{otherwise}) \end{cases}$$

$$\Rightarrow \int_{e_i} \boldsymbol{u}_h|_T \cdot d\boldsymbol{s} \quad = \quad \sum_j \phi_j^T d_j^T \int_{e_i} \boldsymbol{b}_j^T \cdot d\boldsymbol{s} = \phi_i^T.$$

2.2.2 Local Mass Matrix

The second integral of the bilinear form (3), evaluated for all combinations of the shape functions, results in the local mass matrix for triangle T

$$M_{ij}^T = \int_T \boldsymbol{b}_i^T \cdot \boldsymbol{b}_j^T d\boldsymbol{x} = \frac{1}{3}|T| \sum_{k=1}^{3} \boldsymbol{b}_i^T(\boldsymbol{m}_k) \cdot \boldsymbol{b}_j^T(\boldsymbol{m}_k),$$

where $\boldsymbol{m}_k$ is the midpoint of edge k. Due to linear shape functions, the midpoint rule yields the exact integral.

2.2.3 Local curl curl Matrix

The first part of the bilinear form (3) can also be evaluated exactly for the shape functions. Since these functions are linear, curl $\boldsymbol{b}_i^T$ (i = 1, 2, 3) is constant ($= c_i$) and from Gauss' Theorem it follows that

$$c_i|T| = \sum_{j=1}^{3} d_j^T \int_{e_j} \boldsymbol{b}_i d\boldsymbol{x} = \sum_{j=1}^{3} d_j^T \delta_{ij} d_j^T = 1$$

which leads to

$$C_{ij}^T = \int_T \operatorname{curl} \boldsymbol{b}_i^T \operatorname{curl} \boldsymbol{b}_j^T d\boldsymbol{x} = c_i c_j |T| = \frac{1}{|T|}.$$

2.2.4 Local Right Hand Side

In order to approximate the local L^2 scalar product of the right hand side function and a shape function, the following one point quadrature rule is used, which yields the exact integral if $\boldsymbol{f}$ is constant on triangle T:

$$f_i^T = \int_T \boldsymbol{f} \cdot \boldsymbol{b}_i^T \, d\boldsymbol{x} \approx \frac{|T|}{3} \boldsymbol{f}(\boldsymbol{c}_T) \cdot (\boldsymbol{\nabla}\lambda_{i+2}^T - \boldsymbol{\nabla}\lambda_{i+1}^T).$$

($\boldsymbol{c}_T$ is the barycenter of triangle T.)

2.2.5 Global Basis Functions and Vector Representation of a Vector Field

If only one single value is assigned to each edge $e \in \mathcal{T}_h$, tangetial continuity of the vector field along edges $\boldsymbol{u}_h$ is forced.
Thus, the degrees of freedom ϕ_e representing a vector field $\boldsymbol{u}_h$ in the basis

mentioned above can be obtained by evaluating the line integrals along the edges:

$$\phi_e(\boldsymbol{u}_h) = \int_e \boldsymbol{u}_h \cdot d\boldsymbol{s}.$$

Let e be a directed edge with T_1 the triangle on its left and T_2 the triangle

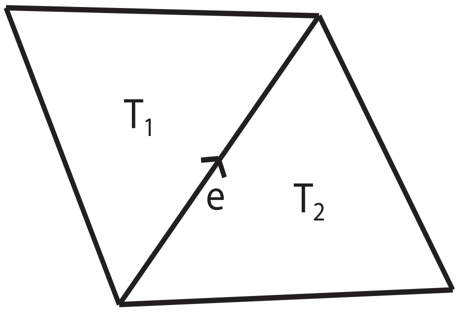

Figure 2: The support of a global basis function $\boldsymbol{b}_e$ are the two triangles on the left and on the right of the edge.

on its right side. If e is the i^{th} edge of T_1 and the j^{th} edge of T_2, the global basis function to this edge is (fig. 2):

$$\boldsymbol{b}_e(\boldsymbol{x}) = \begin{cases} \boldsymbol{b}_i^{T_1}(\boldsymbol{x}), & \boldsymbol{x} \in T_1 \\ -\boldsymbol{b}_j^{T_2}(\boldsymbol{x}), & \boldsymbol{x} \in T_2 \\ 0, & \text{otherwise} \end{cases}$$

$$\Rightarrow \boldsymbol{u}_h = \sum_{e \in \mathcal{E}_h} \phi_e \boldsymbol{b}_e, \tag{8}$$

where $\mathcal{E}_h$ designates the set of inner edges of $\mathcal{T}_h$.
A vector field $\boldsymbol{u}_h \in \mathcal{ND}_1^0(\mathcal{T}_h)$ can thus be represented by a vector $\boldsymbol{\phi}_h \in \mathbb{R}^{Ne}$ (Ne: number of inner edges of $\mathcal{T}_h$).
Let $\boldsymbol{u}_h(\boldsymbol{x})$ be represented by $\boldsymbol{\phi}_h$ and $\boldsymbol{v}_h(\boldsymbol{x})$ by $\boldsymbol{\psi}_h$. Then the following holds:

$$a(\boldsymbol{u}_h(\boldsymbol{x}), \boldsymbol{v}_h(\boldsymbol{x})) = \boldsymbol{\phi}_h^t \boldsymbol{A}\ \boldsymbol{\psi}_h$$

$$f(\boldsymbol{v}_h) = \boldsymbol{f}^t \boldsymbol{\psi}_h$$

(6) is then equivalent to the linear system

$$\boldsymbol{A}\boldsymbol{\phi}_h = \boldsymbol{f}. \tag{9}$$

2.2.6 Assembly of Global Stiffnes Matrix and Right Hand Side

Let $\boldsymbol{S}^T \in \mathbb{R}^{N\times 3}$ be the matrix that assigns each edge of triangle T its global edge index:

$$S_{ij}^T = \begin{cases} 1, & \text{global edge i is the counterclockwise oriented edge j of triangle T} \\ -1, & \text{global edge i is the clockwise oriented edge j of triangle T} \end{cases}$$

Then $\boldsymbol{A}$ and $\boldsymbol{f}$ are assembled as follows:

$$\boldsymbol{A} = \sum_{T\in T_h} \boldsymbol{S}^T(\boldsymbol{C}^T + \tau \boldsymbol{M}^T)(\boldsymbol{S}^T)^t \tag{10}$$

$$\boldsymbol{f} = \sum_{T\in T_h} \boldsymbol{S}^T \boldsymbol{f}^T. \tag{11}$$

Boundary conditions ($\boldsymbol{u} \times \boldsymbol{n} = 0$) are imposed by neglecting the rows and columns corresponding to the boundary edges.

3 The Auxiliary Space Method

3.1 Theory

The basic idea of the Auxiliary Space Method ([2],[3]) is to transfer a problem to an auxiliary space where it is simpler to solve. The solution in the auxiliary space is then transferred back to the original space. The mismatch between the auxiliary space and the full space is corrected by applying a smoothing scheme.
The auxiliary space method with application to unstructured grids is described in [3]. Given a space $\mathcal{V}_h$ with an inner product $(.,.)$ and symmetric positive definite (SPD) linear operator $\mathcal{A} : \mathcal{V}_h \to \mathcal{V}_h$, the components necessary for this method are following:

- a "simple" auxiliary space $\mathcal{V}_0$ with an inner product $[.,.]$,
- a transfer operator $\mathcal{I} : \mathcal{V}_0 \to \mathcal{V}$ linking the two spaces,
- a representation of the original problem in the auxiliary space, in particular a SPD operator $\mathcal{A}_0 : \mathcal{V}_0 \to \mathcal{V}_0$,
- an efficient solver in the auxiliary space or a SPD preconditioner operator $\mathcal{B}_0 : \mathcal{V}_0 \to \mathcal{V}_0$ for the operator $\mathcal{A}_0$ and
- a smoothing scheme in the original space described by a SPD operator $\mathcal{R} : \mathcal{V}_h \to \mathcal{V}_h$.

The proposed preconditioner $\mathcal{B}$ with an additive smoother $\mathcal{R}$ then reads:

$$\mathcal{B} = \mathcal{R} + \mathcal{I}\mathcal{B}_0\mathcal{I}^t \tag{12}$$

It is proven [3] that

$$\kappa(\mathcal{B}\mathcal{A}) := \frac{\lambda_{max}(\mathcal{B}\mathcal{A})}{\lambda_{min}(\mathcal{B}\mathcal{A})}$$

is bounded by a constant if the above-mentioned operators fulfil some conditions:

Theorem 1 *If there are some nonnegative constants $\alpha_0, \alpha_1, \lambda_0, \lambda_1$ and β_1 such that $\forall\, v \in \mathcal{V}_h$ and $w \in \mathcal{V}_0$:*

$$\frac{\alpha_0}{\lambda_{max}(\mathcal{A})}(v,v) \;\leq\; (\mathcal{R}v,v) \leq \frac{\alpha_1}{\lambda_{max}(\mathcal{A})}(v,v) \tag{13}$$

$$\lambda_0[w,w]_{A_0} \;\leq\; [\mathcal{B}_0\mathcal{A}_0 w, w]_{A_0} \leq \lambda_1[w,w]_{A_0} \tag{14}$$

$$||\mathcal{I}w||_A^2 \;\leq\; \beta_1||w||_{A_0}^2 \tag{15}$$

and if there exists a linear operator $\mathcal{P}$: $\mathcal{V}_h \to \mathcal{V}_0$ and positive constants β_0 and γ_0 such that

$$||\mathcal{P}v||_{A_0}^2 \leq \frac{||v||_A^2}{\beta_0} \quad \textit{and} \tag{16}$$

$$||v - \mathcal{I}\mathcal{P}v||^2 \leq \frac{||v||_A^2}{\gamma_0 \lambda_{max}(A)} \tag{17}$$

then the preconditioner $\mathcal{B}$ satisfies:

$$\kappa(\mathcal{B}\mathcal{A}) \leq (\alpha_1 + \beta_1\lambda_1)\Big(\frac{1}{\alpha_0\gamma_0} + \frac{1}{\beta_0\gamma_0}\Big) \tag{18}$$

This general technique can now be applied to precondition a problem on an unstructured grid. In [2], such an example is given for the laplace equation with Dirichlet boundary conditions. A structured grid of isosceles rectangular triangles (fig. 4) on $\Omega_0 \subset \Omega$ together with the finite element space of continuous piecewise linear functions that vanish on $\Omega \backslash \Omega_0$ ($S_{1,0}^h(\mathcal{T}_0)$) serves as auxiliary space and the spaces are linked by the standard nodal value interpolants $\mathcal{I}: \mathcal{V}_0 \to \mathcal{V}_h$ and $\mathcal{P}: \mathcal{V}_h \to \mathcal{V}_0$.
For this special case (in two dimensions), the inequalities (15) - (17) are rather simple to prove, because for standard nodal value interpolants and $||v||_A = ||\boldsymbol{\nabla} v||_0 \geq \tilde{C}||v||_1$ (Poincare inequality), there exists a close relation between the interpolation error $||v - \mathcal{I}v||_0$ and the energy norm $||v||_A$ (standard error estimate).
For $v \in \mathcal{V}_h, v_0 \in \mathcal{V}_0$ it can be shown that there are constants $C_1 - C_4$ such that

$$||v_0 - \mathcal{I}v_0||_0 \leq C_1 h_0 ||v_0||_1 \tag{19}$$

$$||\mathcal{I}v_0||_1 \leq C_2||v_0||_1, \tag{20}$$

$$||v - \mathcal{P}v||_0 \leq C_3 h||v||_1 \tag{21}$$

$$||\mathcal{P}v||_1 \leq C_4||v||_1, \tag{22}$$

where h and h_0 are the mesh sizes of $\mathcal{T}_h$ and $\mathcal{T}_0$. Constants for (15) - (17) can then easily be derived. If $h \approx h_0$ it follows that

$$\begin{aligned} ||v - \mathcal{I}\mathcal{P}v||_0^2 &\leq 2||v - \mathcal{P}v||_0^2 + 2||\mathcal{P}v - \mathcal{I}\mathcal{P}v||_0^2 \\ &\leq 2C_3^2h^2||v||_1^2 + 2C_1^2h^2||\mathcal{P}v||_1 \\ &\leq \tilde{C}h^2||v||_1^2. \end{aligned}$$

With $\lambda_{max}(A) \approx h^2$, it follows that there exists a constant for inequality 17. The proofs of (19) - (22) are based on the follwing inequalities:

$$\begin{aligned}
||v_0 - \mathcal{I}v_0||_{0,\infty,T} &\lesssim h|v_0|_{1,\infty,T} \text{ (standard error estimate)} && (23)\\
|v_0|_{1,\infty,\tilde{T}} &\lesssim h_0||v_0||_{1,\tilde{T}} \text{ (inverse inequality)} && (24)\\
\sum_{y\in\mathcal{N}_0\cap\partial\tilde{\Omega}} |v(y)|^2 &\lesssim \sum_{x\in\mathcal{N}_h\cap(\Omega\backslash\widehat{\Omega})} |v(x)|^2 \ (v(y),\, y \in \mathcal{N}_0 \cap \partial\tilde{\Omega}, && (25)\\
&\quad \text{are convex combinations of } v(x),\, x \in \mathcal{N}_h \cap (\Omega\backslash\widehat{\Omega}))\\
||w||_{0,\Omega\backslash\tilde{\Omega}} &\lesssim h|w|_{1,\Omega\backslash\tilde{\Omega}} \forall w \in H_0^1(\Omega) && (26)
\end{aligned}$$

Notation: $\tilde{T}$ is the union of elements in $\mathcal{T}_0$ that intersect with T, $\mathcal{N}_0$ and $\mathcal{N}_h$ are the triangle vertices of $\mathcal{T}_0$ and $\mathcal{T}_h$, respectively. $a \lesssim b$ means that there is a constant C with $a \leq Cb$. To prove (19) and (20), the domain Ω is divided into two parts: a domain $\tilde{\Omega}$ of elements in Ω_0 that do not intersect with $\partial\Omega_0$ and the complementary domain $\Omega\backslash\tilde{\Omega}$. Additionaly, the subset of $\tilde{\Omega}$ consisting of the interior elements of $\tilde{\Omega}$ is designated with $\widehat{\Omega}$ (fig. 3).
Inequality (26) is based on

$$||w||^2_{0,G^\nu} \lesssim \nu^2 \int_{G^\nu} |\nabla w|^2 dx \text{ (Poincaré inequality, scaling)}$$

(G^ν is a square of side length ν) that holds if w vanishes on one face of G^ν. $\Omega\backslash\tilde{\Omega}$ can then be covered with such squares with $\nu = O(h)$ which leads to (26).
Proof of (19):

$$\begin{aligned}
||v_0 - \mathcal{I}v_0||_{0,T} &\lesssim \sqrt{|T|}\,||v_0 - \mathcal{I}v_0||_{0,\infty,T} \lesssim h^2|v_0|_{1,\infty,T}\\
&\lesssim h^2h_0^{-1}||v_0||_{1,\tilde{T}} \lesssim h||v_0||_{1,\tilde{T}}
\end{aligned}$$

This holds if $h \approx h_0$. ($|T|$ is the area of triangle T.)

$$\begin{aligned}
\Rightarrow ||v_0 - \mathcal{I}v_0||^2_{0,\Omega} &= \sum_{T\in\mathcal{T}_h} ||v_0 - \mathcal{I}v_0||^2_{0,T} \lesssim \sum_{T\in\mathcal{T}_h} h^2||v_0||^2_{1,\tilde{T}}\\
&\lesssim \sum_{T\in\mathcal{T}_h} h^2||v_0||^2_{1,T} = h^2||v_0||^2_1
\end{aligned}$$

Proof of (20):

$$\begin{aligned}
|v_0 - \mathcal{I}v_0|_{1,T} &\lesssim \sqrt{|T|}\,|v_0 - \mathcal{I}v_0|_{1,\infty,T} \lesssim h\,|v_0|_{1,\infty,T} \lesssim hh_0^{-1}|v_0|_{1,\tilde{T}}\\
\Rightarrow ||v_0 - \mathcal{I}v_0||^2_1 &= |v_0 - \mathcal{I}v_0|^2_1 + ||v_0 - \mathcal{I}v_0||^2_0 \lesssim h^2h_0^{-2}|v_0|^2_1 + h^2||v_0||^2_0.
\end{aligned}$$

If $h \approx h_0$ and h $\leq$ C:

$$\begin{aligned} \Rightarrow ||v_0 - \mathcal{I}v_0||_1^2 &\lesssim |v_0|_1^2 + ||v_0||_0^2 = ||v_0||_1^2 \\ \Rightarrow ||\mathcal{I}v_0||_1 &\leq ||v_0||_1 + ||v_0 - \mathcal{I}v_0||_1 \lesssim ||v_0||_1. \end{aligned}$$

Proof of (21) and (22):
On the interior domain $\tilde{\Omega}$ the same arguments as above can be used to prove

$$\begin{aligned} ||v - \mathcal{P}v||_{\tilde{\Omega}} &\lesssim h_0||v||_{1,\Omega} \text{ and} \\ ||\mathcal{P}v||_{\tilde{\Omega}} &\lesssim ||v||_{1,\Omega}. \end{aligned}$$

For the boundary strip $\Omega\backslash\tilde{\Omega}$ the following holds:

$$|\mathcal{P}v|_{1,\Omega\backslash\tilde{\Omega}}^2 \lesssim \sum_{y\in\mathcal{N}_0\cap\partial\tilde{\Omega}} |v(y)|^2 \lesssim \sum_{x\in\mathcal{N}_h\cap(\Omega\backslash\widehat{\Omega})} |v(x)|^2 \lesssim |v|_{1,\Omega\backslash\widehat{\Omega}} \lesssim |v|_{1,\Omega}^2.$$

Applying (26) to $\mathcal{P}v$ and v yields

$$\begin{aligned} ||v||_{0,\Omega\backslash\tilde{\Omega}} &\lesssim h_0|v|_{1,\Omega\backslash\tilde{\Omega}} \text{ and} \\ ||\mathcal{P}v||_{0,\Omega\backslash\tilde{\Omega}} &\lesssim h_0|v|_{1,\Omega\backslash\tilde{\Omega}}. \\ \Rightarrow ||\mathcal{P}v||_1^2 &= ||\mathcal{P}v||_{1,\tilde{\Omega}}^2 + ||\mathcal{P}v||_{1,\Omega\backslash\tilde{\Omega}}^2 \lesssim ||v||_{1,\Omega}^2 + ||\mathcal{P}v||_{0,\Omega\backslash\tilde{\Omega}}^2 + |\mathcal{P}v|_{1,\Omega\backslash\tilde{\Omega}}^2 \\ &\lesssim ||v||_{1,\Omega}^2 + h_0^2|\mathcal{P}v|_{1,\Omega\backslash\tilde{\Omega}}^2 + |v|_{1,\Omega}^2 \lesssim (2 + h_0^2) \lesssim ||v||_{1,\Omega}^2 \text{ and} \\ ||v - \mathcal{P}v|| &\leq ||v - \mathcal{P}v||_{0,\tilde{\Omega}} + ||v||_{0,\Omega\backslash\tilde{\Omega}} + ||\mathcal{P}v||_{0,\Omega\backslash\tilde{\Omega}} \leq h_0||v||_{1,\Omega}. \end{aligned}$$

This proves inequalities (21) and (22).
These interpolation operators $\mathcal{I}$ and $\mathcal{P}$ allow a H^1-stable decomposition in the sense of

$$v = v_h + \mathcal{I}v_0, \ v_0 = \mathcal{P}v, \ v_h = v - \mathcal{I}\mathcal{P}v$$

with $h^{-2}||v_h||_0^2 + ||w||_1^2 \lesssim ||v||_1$. This is sufficient for $\mathcal{I}$ to safisfy the stability condition

$$\inf\{h^{-2}||\tilde{v}||_{L^2(\Omega)} + ||w||_{A_0}^2 : \tilde{v} + \mathcal{I}w = v, \tilde{v} \in \mathcal{V}, w \in \mathcal{V}_0\} \lesssim ||v||_A \forall v \in \mathcal{V}.$$

3.2 The Auxiliary Space Method for Maxwell's equations

In this work, the method is applied to construct a preconditioner for the problem (6) on a unstructured shape-regular mesh $\mathcal{T}_h$.
In this section, the required ingredients for the algorithm are defined. The preconditioner on the auxiliary mesh is described in section 3.4 and its application to the CG method in section 3.3. Theoretical aspects are discussed in section 3.5.

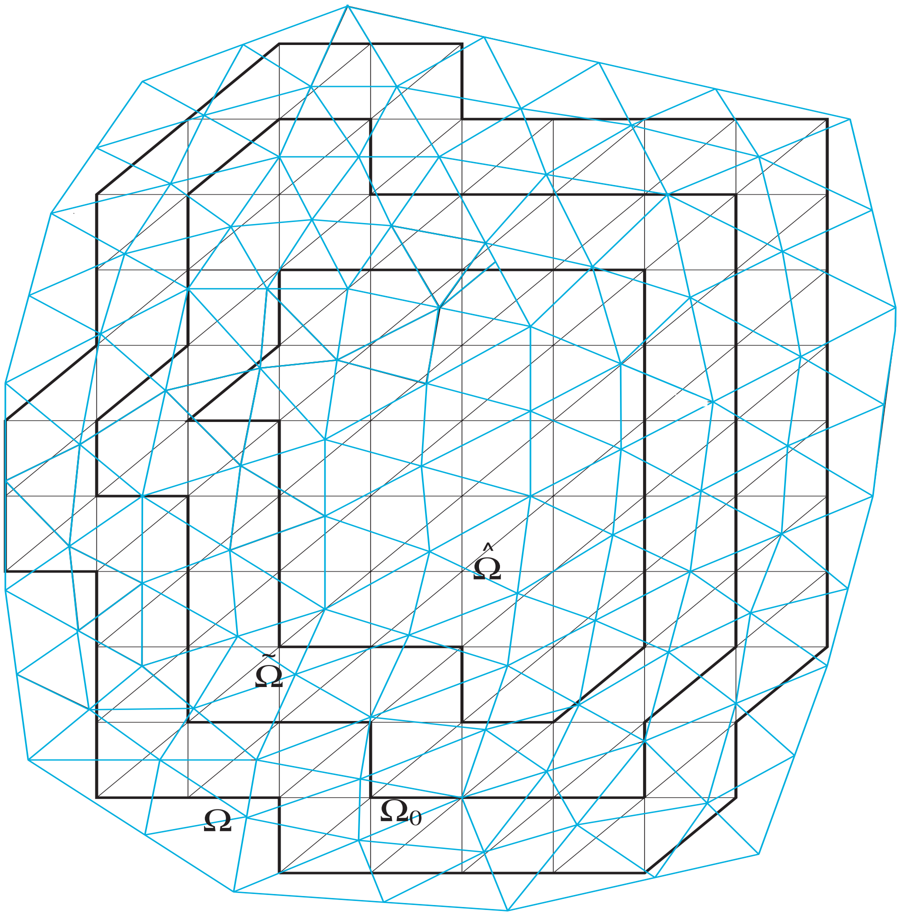

Figure 3: The domains Ω, Ω_0, $\tilde{\Omega}$ and $\widehat{\Omega}$.

3.2.1 The Auxiliary Space

The mesh $\mathcal{T}_0$ on the auxiliary domain $\Omega_0 \subset \Omega$ should have a very simple geometry. Therefore, it consists like in the above-mentioned example of congruent isosceles rectangular triangles of side length dx.
The finite element space $\mathcal{ND}_1^0(\mathcal{T}_0)$ used on this mesh is the same as on the unstructured mesh (see section 2.2).

3.2.2 The Transfer Operator $\mathcal{I}$ from the Auxiliary to the Unstructured Mesh

In order to solve the system $\boldsymbol{Ax} = \boldsymbol{f}$ on the auxiliary mesh, the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{f}$ have to be transferred to the auxiliary mesh.
Let $\mathcal{V}_h$ and $\mathcal{V}_0$ be finite element spaces on the unstructured and on the aux-

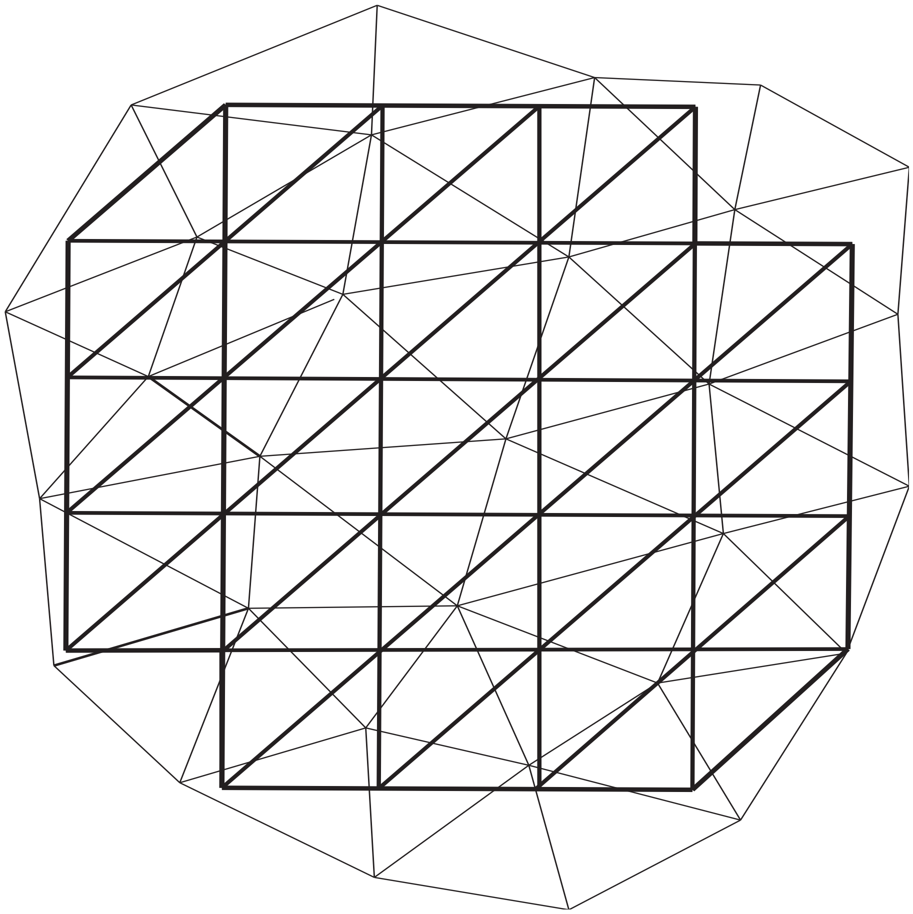

Figure 4: The structured auxiliary mesh.

iliary mesh, respectively, and $\mathcal{E}_h(\mathcal{T}_h)$ the set of internal edges of the unstructured mesh.
The two meshes are linked by the nodal interpolation operator $\mathcal{I}:\ \mathcal{V}_0 \to \mathcal{V}_h$ as follows:

$$\boldsymbol{u}_0 \quad \mapsto \quad \boldsymbol{u}_h = \mathcal{I}(\boldsymbol{u}_0) = \sum_{i \in \mathcal{E}_h} \phi_i^h \boldsymbol{b}_i^h, \tag{27}$$

$$\phi_i^h = \int_{e_i^h} \boldsymbol{u}_0 \cdot d\boldsymbol{s}. \tag{28}$$

This linear operator can be expressed as a matrix $\boldsymbol{I} \in \mathbb{R}^{Ne_h \times Ne_0}$ with

$$\phi_i^h = \sum_j \boldsymbol{I}_{ij}\, \phi_j^0.$$

3.2.3 The Transfer Matrix in the Opposite Direction

Let $\boldsymbol{\phi}^h \in \mathbb{R}^{Ne_h}$ represent a function on the unstructured mesh and $\boldsymbol{\psi}^0 \in \mathbb{R}^{Ne_0}$ a function on the auxiliary mesh. Then, following [2], an operator $\mathbf{\Pi}$: $\mathbb{R}^{Ne_h} \rightarrow \mathbb{R}^{Ne_0}$ must fulfil the condition

$$(\mathbf{\Pi}\ \boldsymbol{\phi}^h) \cdot \boldsymbol{\psi}^0 = \boldsymbol{\phi}^h \cdot (\boldsymbol{I}\ \boldsymbol{\psi}^0) \quad \forall \boldsymbol{\phi}^h \in \mathbb{R}^{Ne_h}, \boldsymbol{\psi}^0 \in \mathbb{R}^{Ne_0},$$

where $. \cdot .$ is the euclidean dot product in $\mathbb{R}^{Ne_0}$ or $\mathbb{R}^{Ne_h}$, respectively. This implies that

$$\mathbf{\Pi} = \boldsymbol{I}^t.$$

3.2.4 The Linear System in the Auxiliary Space

The linear operator $\mathcal{A}_0$ in the auxiliary space should correspond in a certain way to the operator $\mathcal{A}$ on the unstructured mesh. In particular, $\mathcal{A}_0$ should have a similar spectrum like the discretized partial differential operator (see [2]).

There are two fairly natural ways to obtain an operation $\mathcal{A}_0$:

a) Galerkin construction:

$$\boldsymbol{A}_0 = \boldsymbol{I}^t\ \boldsymbol{A}\ \boldsymbol{I}$$

This satisfies

$$(\boldsymbol{A}_0\ \boldsymbol{\psi}^0) \cdot \boldsymbol{\phi}^0 = (\boldsymbol{I}\ \boldsymbol{\psi}^0) \cdot (\boldsymbol{A}\ \boldsymbol{I}\ \boldsymbol{\phi}^0)\ \forall \boldsymbol{\phi}^0, \boldsymbol{\psi}^0 \in \mathbb{R}^{Ne_0}.$$

b) Direct computation:
$\boldsymbol{A}_0$ is obtained directly from the mesh geometry in the same way as $\boldsymbol{A}$ on the unstructured mesh (10).

The linear system on the auxiliary mesh then reads:

$$\boldsymbol{A}_0\ \boldsymbol{\phi}_0 = \boldsymbol{f}_0 \tag{29}$$

with $\boldsymbol{f}^0 = \boldsymbol{I}^t\ \boldsymbol{f}^h$.

Then $\boldsymbol{\phi}^h = \boldsymbol{I}\ \boldsymbol{\phi}^0$ yields a good correction for the solution of the original linear system, if $\boldsymbol{f}_h$ is the residual.

3.2.5 Smoother

Elementary iterative smoothing schemes for a system $\boldsymbol{Ax} = \boldsymbol{b}$ approximate the inverse of $\boldsymbol{A}$ with a simply invertable matrix $\boldsymbol{B}$ and iterate

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{B}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k),\ k = 1, 2, ...$$

In the popular Gauss-Seidel method is

$$\boldsymbol{B} = (\boldsymbol{D} - \boldsymbol{L})^{-1},$$

where $\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{U} - \boldsymbol{L}$ is the decompositioin of the matrix in its diagonal, lower triangular and upper triangular parts.
The Gauss-Seidel method is a good smoother for elliptic problems. But the bilinear form (3) is not elliptic on the kernel $\mathcal{N}(\text{curl})$ of the curl operator and, therefore, a simple Gauss-Seidel smoothing scheme won't work. That's why a more complicated hybrid smoother has to be used [5].
The relaxation scheme used in the precondition algorithm is based on the fact that

$$\text{grad}S^h_{1,0} \quad := \quad \{\boldsymbol{v}_h = \sum_{i=1}^{Nv} \alpha_i \boldsymbol{\nabla}\lambda_i; \alpha_i \in \mathbb{R}, \lambda_i : \text{hat function of internal vertex } i\}$$

(Nv: number of internal vertices of $\mathcal{T}_h$)
is a subspace of $\mathcal{ND}^0_1(\mathcal{T}_h)$. In fact, for simply connected Ω,

$$\text{grad}S^h_{1,0} = \{\boldsymbol{v}_h \in \mathcal{ND}^0_1(\mathcal{T}_h); \text{curl}(\boldsymbol{v}_h) = 0\}.$$

If $\mathcal{ND}^0_1(\mathcal{T}_h)$ is decomposed, that is

$$\mathcal{ND}^0_1(\mathcal{T}_h) = \mathcal{N}(\text{curl}) \oplus \mathcal{N}(\text{curl})^{\perp},$$

equation (5) can be solved in both subspaces. It turns out that in $\mathcal{N}(\text{curl})^{\perp}$ the differential equation is purely elliptic and a Gauss-Seidel smoothing scheme can be applied. Furthermore, $\forall \boldsymbol{u}_h \in \mathcal{N}(\text{curl})$, there is a potential Φ with $\nabla\Phi = \boldsymbol{u}_h$. In this potential space, equation (5) reads

$$a(\boldsymbol{\nabla}\Phi, \boldsymbol{\nabla}\Psi) = f(\boldsymbol{\nabla}\Psi) \quad \forall\Psi \in \text{grad}S^h_{1,0}(\mathcal{T}_h). \tag{30}$$

Since

$$\begin{aligned} a(\boldsymbol{\nabla}\Phi, \boldsymbol{\nabla}\Psi) &= \int_\Omega \text{curl}\boldsymbol{\nabla}\Phi\ \text{curl}\boldsymbol{\nabla}\Psi\ d\boldsymbol{x} + \tau \int_\Omega \boldsymbol{\nabla}\Phi \cdot \boldsymbol{\nabla}\Psi\ d\boldsymbol{x} \\ &= \tau \int_\Omega \boldsymbol{\nabla}\Phi \cdot \boldsymbol{\nabla}\Psi\ d\boldsymbol{x},\ \tau > 0 \end{aligned}$$

this bilinear form corresponds to the laplace equation and is therefore elliptic, so after lifting $\boldsymbol{u}_h \in \mathcal{N}(\text{curl})$ into potential space, it can be smoothed with a Gauss-Seidel scheme.
There is a simple discrete gradient operator $\mathcal{T}$: $S^h_{1,0} \rightarrow \text{grad} S^h_{1,0} \subset \mathcal{ND}^0_1(\mathcal{T}_h)$ that allows to tranfer a vector $\boldsymbol{\alpha} \in \mathbb{R}^{Nv}$ representing $\Phi \in S^h_{1,0}$ to a vector $\boldsymbol{\phi} \in \mathbb{R}^{Ne}$ representing $\nabla\Phi \in \mathcal{ND}^0_1(\mathcal{T}_h)$:

$$\begin{aligned} \Phi|_T(x) &= \sum_{i=1}^{Nv} \alpha_i \lambda_i(\boldsymbol{x}) \\ \Rightarrow \nabla\Phi|_T(x) &= \sum_{i=1}^{Nv} \alpha_i \boldsymbol{\nabla} \lambda_i(\boldsymbol{x}) \\ &\overset{!}{=} \sum_{j=1}^{Ne} \phi_j \boldsymbol{b}_j(\boldsymbol{x}) \ \forall \boldsymbol{x} \in \Omega_h. \end{aligned} \tag{31}$$

If e_{ij} is the directed edge $[\boldsymbol{a}_i\ \boldsymbol{a}_j]$, this leads to

$$\phi_{ij} = \alpha_j - \alpha_i \quad (\text{fig. } 5).$$

In global matrix representation the operator $\boldsymbol{T} \in \mathbb{R}^{Ne \times Nv}$ reads

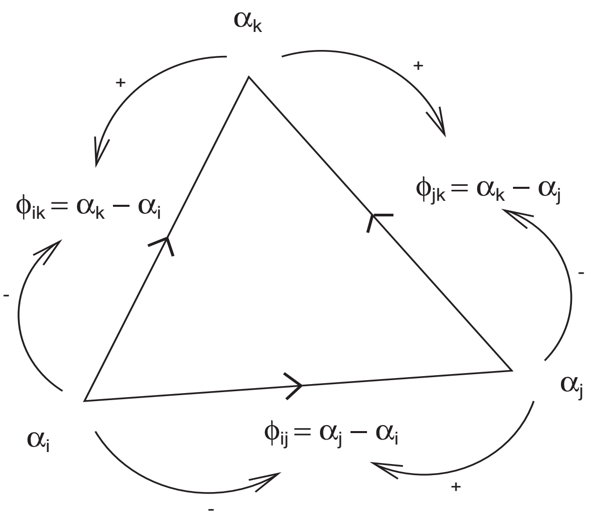

Figure 5: The discrete gradient operator.

$$\boldsymbol{T}_{i,j} = \begin{cases} 1, & \text{if } \boldsymbol{a}_j \text{ is end point of edge } i \\ -1, & \text{if } \boldsymbol{a}_j \text{ is starting point of edge } i \\ 0, & \text{otherwise.} \end{cases}$$

Equation (30) can then be transformed to the linear system as follows:

$$\boldsymbol{T}^t \boldsymbol{A} \boldsymbol{T} \boldsymbol{x} = \boldsymbol{T}^t \boldsymbol{f}. \tag{32}$$

Unfortunately, there exists no reasonable numerical FE space of $\mathcal{N}(\text{curl})^\perp$. Therefore, $\mathcal{ND}_1^0(\mathcal{T}_h)$ is used as an approximation of $\mathcal{N}(\text{curl})^\perp$.
In practice, the system is smoothed on $\mathcal{ND}_1^0(\mathcal{T}_h)$ with a subspace correction on $\text{grad} S_{1,0}^h$.
Given an initial guess $\boldsymbol{x}_0 \in \mathcal{ND}_1^0(\mathcal{T}_h)$, the hybrid smoothing algorithm is as follows:

$\boldsymbol{x} = \texttt{HYBRID\_SMOOTHER}(\boldsymbol{x}_0, \boldsymbol{f}, n_1, n_2, n_3)$:

1. Pre-smoothing:
   n_1 Gauss-Seidel iterations on $\mathcal{ND}_1^0(\mathcal{T}_h)$ with matrix $\boldsymbol{A}$ and right hand side $\boldsymbol{f}$

2. Lifting of the residual to the subspace $\text{grad} S_{1,0}$ with matrix $\boldsymbol{T}$:

$$\boldsymbol{r}_{pot} = \boldsymbol{T}^t \ (\boldsymbol{f} - \boldsymbol{A} \ \boldsymbol{x}).$$

3. Lifting of the matrix $\boldsymbol{A}$ to the potential space by means of a galerkin construction:

$$\boldsymbol{A}_{pot} = \boldsymbol{T}^t \ \boldsymbol{A} \ \boldsymbol{T}.$$

4. Smoothing in potential space:
   n_2 Gauss-Seidel iterations to solve the system

$$\boldsymbol{A}_{pot} \boldsymbol{e}_{pot} = \boldsymbol{r}_{pot}$$

   with initial guess $\boldsymbol{e}_{pot} = 0$.

5. Inverse lifting of $\boldsymbol{e}_{pot}$ to $\mathcal{ND}_1^0(\mathcal{T}_h)$:

$$\boldsymbol{e} = \boldsymbol{T} \ \boldsymbol{e}_{pot}.$$

6. Adding the correction to the solution

$$\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{e}.$$

7. Post-smoothing: n_3 Gauss-Seidel iterations on $\mathcal{ND}_1^0(\mathcal{T}_h)$ with matrix $\boldsymbol{A}$ and right hand side $\boldsymbol{f}$.

3.3 Preconditioned CG based on the Auxiliary Space Method

Since the condition number of the matrix $\boldsymbol{A}$ scales like h^{-2}, the CG scheme converges very slowly when the size of the elements is scaled down. Therefore it is necessary to precondition the system $\boldsymbol{Ax} = \boldsymbol{f}$ with a SPD operator $\boldsymbol{B}$ by solving the equivalent system: $\boldsymbol{BAx} = \boldsymbol{Bf}$. This can speed up the convergence of the scheme drastically, since

$$\frac{||\boldsymbol{e}^k||_A}{||\boldsymbol{e}^0||_A} \leq 2\Big(\frac{\sqrt{\chi(\boldsymbol{BA})}-1}{\sqrt{\chi(\boldsymbol{BA})}+1}\Big)^k.$$

A good preconditioner $\boldsymbol{B} : \mathbb{R}^{Ne} \to \mathbb{R}^{Ne}$ for the linear system $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{f}$ approximates the inverse of the matrix $\boldsymbol{A}$ as well as possible but is computationally cheap. The preconditioner (12) proposed by Xu meets these requirements, if the transfer between the two spaces and the preconditioner in the auxiliary space as well as the smoothing scheme in the original space are computationally cheap. In this work, a multiplicative Gauss-Seidel scheme is used instead of an additive smoother. The general algorithm for the linear operator $\boldsymbol{B}$ then reads:

$\boldsymbol{x} = \boldsymbol{B}(\boldsymbol{v})$:

1. Initialize $\boldsymbol{x} = \boldsymbol{0}$.

2. Pre-smooting on the unstructured mesh with matrix $\boldsymbol{A}$ and right hand side $\boldsymbol{v}$:

$$\boldsymbol{x} = \texttt{PRE\_SMOOTHER}(\boldsymbol{x}).$$

3. Transfer of residual $\boldsymbol{r} = \boldsymbol{v} - \boldsymbol{Ax}$ to the auxiliary mesh:

$$\boldsymbol{r}_0 = \boldsymbol{I}^t\, \boldsymbol{r}.$$

4. The system $\boldsymbol{A}_0\boldsymbol{e}_0 = \boldsymbol{r}_0$ is solved using an efficient solver in the auxiliary space or a preconditioner $\boldsymbol{B}_0$ is applied:

$$\boldsymbol{e}_0 = \boldsymbol{B}_0\boldsymbol{r}_0.$$

5. Transfer of the correction to the unstructured mesh:

$$\boldsymbol{e} = \boldsymbol{I}\ \boldsymbol{e}_0.$$

6. Adding the correction to the solution

$$\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{e}.$$

7. Post-smoothing on the unstructured mesh with matrix $\boldsymbol{A}$ and right hand side $\boldsymbol{v}$:

$$\boldsymbol{x} = \texttt{POST\_SMOOTHER}(\boldsymbol{x}).$$

3.3.1 Pre- and Post-Smoothing

Pre-and Postsmoothing on the unstructured mesh is used to resolve what cannot be resolved on the auxiliary mesh.
The elementary smoothing schemes for the linear system $\boldsymbol{Av} = \boldsymbol{f}$ applied in the hybrid smoother (section 3.2.5) are the *Forward Gauss-Seidel* method

$$\boldsymbol{v}_{k+1} = \boldsymbol{v}_k + (\boldsymbol{D} - \boldsymbol{L})^{-1}(\boldsymbol{f} - \boldsymbol{Av}_k),\ k = 1, 2, ...$$

with error propagation

$$\boldsymbol{e}_{k+1} = (\mathbb{I} - (\boldsymbol{D} - \boldsymbol{L})^{-1}\boldsymbol{A})\boldsymbol{e}_k$$

and the *Backward Gauss-Seidel* method

$$\begin{aligned} \boldsymbol{v}^{k+1} &= \boldsymbol{v}^k + (\boldsymbol{D} - \boldsymbol{U})^{-1}(\boldsymbol{f} - \boldsymbol{Av}^k),\ k = 1, 2, ... \\ \boldsymbol{e}^{k+1} &= (\mathbb{I} - (\boldsymbol{D} - \boldsymbol{U})^{-1}\boldsymbol{A})\boldsymbol{e}^k, \end{aligned}$$

where $\mathbb{I}$ designates the identity matrix.
Due to the fact that the auxiliary mesh is smaller than the unstructured mesh and therefore, the Dirichlet boundary conditions on the auxiliary mesh induce errors near the boundary, smoothing is more important there and a restricted smoothing scheme in the subspace of the degrees of freedom in the boundary region is applied: Let $\boldsymbol{v}_1 \in \mathbb{R}^{n_1}$ be the restriction of $\boldsymbol{v} \in \mathbb{R}^{n_1+n_2}$ to the boundary region, $\boldsymbol{v}_2 \in \mathbb{R}^{n_2}$ the restriction to the inner domain and

$\tilde{\boldsymbol{A}}_1 \in \mathbb{R}^{n_1 \times n_1}$, $\tilde{\boldsymbol{A}}_2 \in \mathbb{R}^{n_1 \times n_2}$ the straightforward restrictions of $\boldsymbol{A}$. Then the *Forward Boundary Gauss-Seidel* iteration is

$$\begin{aligned}
\boldsymbol{v}_1^{k+1} &= \boldsymbol{v}_1^k + (\tilde{\boldsymbol{D}}_1 - \tilde{\boldsymbol{L}}_1)^{-1}[\boldsymbol{f}_1 - \tilde{\boldsymbol{A}}_1 \boldsymbol{v}_1^k - \tilde{\boldsymbol{A}}_2 \boldsymbol{v}_2^k] \\
\boldsymbol{v}_2^{k+1} &= \boldsymbol{v}_2^k \\
\boldsymbol{e}^{k+1} &= \begin{pmatrix} \tilde{\mathbb{I}}_1 - (\tilde{\boldsymbol{D}}_1 - \tilde{\boldsymbol{L}}_1)^{-1} \tilde{\boldsymbol{A}}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \tilde{\mathbb{I}}_2 \end{pmatrix} \boldsymbol{e}^k.
\end{aligned}$$

and the *Backward boundary Gauss-Seidel* scheme reads

$$\begin{aligned}
\boldsymbol{v}_1^{k+1} &= \boldsymbol{v}_1^k + (\tilde{\boldsymbol{D}}_1 - \tilde{\boldsymbol{U}}_1)^{-1}[\boldsymbol{f}_1 - \tilde{\boldsymbol{A}}_1 \boldsymbol{v}_1^k - \tilde{\boldsymbol{A}}_2 \boldsymbol{v}_2^k] \\
\boldsymbol{v}_2^{k+1} &= \boldsymbol{v}_2^k \\
\boldsymbol{e}^{k+1} &= \begin{pmatrix} \tilde{\mathbb{I}}_1 - (\tilde{\boldsymbol{D}}_1 - \tilde{\boldsymbol{U}}_1)^{-1} \tilde{\boldsymbol{A}}_1 & \boldsymbol{0} \\ \boldsymbol{0} & \tilde{\mathbb{I}}_2 \end{pmatrix} \boldsymbol{e}^k
\end{aligned}$$

When using a smoother with error propagation $\boldsymbol{M}_o$ in the original space and a smoother with propagation matrix $\boldsymbol{M}_p$ in potential space, the error propagation for the hybrid smoother is

$$\boldsymbol{e}^{k+1} = \boldsymbol{M}_o^{n_3}(\mathbb{I} - \boldsymbol{T}\boldsymbol{M}_p^{n_2}\boldsymbol{A}_0^{-1}\boldsymbol{T}^t\boldsymbol{A})\boldsymbol{M}_o^{n_1}\boldsymbol{e}^k. \tag{33}$$

To meet the symmetry requirement of the operator $\boldsymbol{B}$, the pre- and post-smoothing scheme must fulfil certain conditions. Let $\boldsymbol{M} := \mathbb{I} - \boldsymbol{B}\boldsymbol{A}$ denote the error propagation operator of the preconditioner. Then

$$\boldsymbol{B} \text{ symmetric } \Leftrightarrow \boldsymbol{A}\boldsymbol{M} = \boldsymbol{M}^{\boldsymbol{t}}\boldsymbol{A} \text{ for } \boldsymbol{A} \text{ symmetric .}$$

If $\boldsymbol{M}_1$ is the error propagation of the pre-smoother and $\boldsymbol{M}_2$ the error progagation of the post-smoother, the following holds:

$$\begin{aligned}
\boldsymbol{M} &= \boldsymbol{M}_2(\mathbb{I} - \boldsymbol{I}\boldsymbol{B}_0\boldsymbol{I}^t\boldsymbol{A})\boldsymbol{M}_1 \\
\Rightarrow \boldsymbol{A}\boldsymbol{M}_2 &= \boldsymbol{M}_1^t\boldsymbol{A}.
\end{aligned}$$

With

$$\boldsymbol{M}_1 := \boldsymbol{M}_o^{n_3}(\mathbb{I} - \boldsymbol{T}\boldsymbol{M}_p^{n_2}\boldsymbol{A}_0^{-1}\boldsymbol{T}^t\boldsymbol{A})\boldsymbol{M}_o^{n_1} \text{ (one hybrid smoother sweep)}$$

it follows that

$$\begin{aligned}
\Rightarrow \boldsymbol{M}_2 &\stackrel{!}{=} \boldsymbol{A}^{-1}(\boldsymbol{M}_o^{n_1})^t(\mathbb{I} - \boldsymbol{A}\boldsymbol{T}\boldsymbol{A}_0^{-t}(\boldsymbol{M}_p^{n_2})^t\boldsymbol{T}^t)(\boldsymbol{M}_o^{n_3})^t \\
&= \boldsymbol{A}^{-1}(\boldsymbol{M}_o^{n_1})^t\boldsymbol{A}(\mathbb{I} - \boldsymbol{T}\boldsymbol{A}_0^{-t}(\boldsymbol{M}_p^{n_2})^t\boldsymbol{T}^t\boldsymbol{A})\boldsymbol{A}^{-1}(\boldsymbol{M}_o^{n_3})^t\boldsymbol{A}.
\end{aligned}$$

This is fulfilled if in the post hybrid smoother the inverted number of smoothing steps (n_3, n_2, n_1) and inverted Gauss-Seidel iterations with respect to the pre-smoother are used. If several domain and boundary hybrid pre-smoothing sweeps are used, they have to be repeated in inverse order in the post-smoother.
In the numerical examples in section 5 the following pre- and post-smoothers are used:

$\boldsymbol{x}$=PRE_SMOOTHER($\boldsymbol{x}$):

$$\begin{aligned}
\boldsymbol{x} &= \texttt{FORWARD_BOUNDARY_HYBRID_SMOOTHER}(\boldsymbol{x}, n, n, n)\\
\boldsymbol{x} &= \texttt{FORWARD_HYBRID_SMOOTHER}(\boldsymbol{x}, m, m, m)\\
\boldsymbol{x} &= \texttt{FORWARD_BOUNDARY_HYBRID_SMOOTHER}(\boldsymbol{x}, n, n, n)
\end{aligned}$$

$\boldsymbol{x}$=POST_SMOOTHER($\boldsymbol{x}$):

$$\begin{aligned}
\boldsymbol{x} &= \texttt{BACKWARD_BOUNDARY_HYBRID_SMOOTHER}(\boldsymbol{x}, n, n, n)\\
\boldsymbol{x} &= \texttt{BACKWARD_HYBRID_SMOOTHER}(\boldsymbol{x}, m, m, m)\\
\boldsymbol{x} &= \texttt{BACKWARD_BOUNDARY_HYBRID_SMOOTHER}(\boldsymbol{x}, n, n, n)
\end{aligned}$$

3.4 Multigrid Solver/Preconditioner in the Auxiliary Space

In order to obtain an efficient algorithm, the computational costs to solve or precondition the linear system on the auxiliary mesh should be proportional to the system size. A natural approach to achieve this is to use a multigrid scheme. In this case, such a scheme can be understood as a particular kind of the auxiliary space method where a coarser mesh recursively serves as an auxiliary space of the current mesh. On the coarsest mesh the system is solved directly.

3.4.1 Nested Meshes

The meshes are nested in the most natural way: Four triangles of a mesh build one triangle of the next coarser mesh. Triangles of the coarser mesh that are only partly covered by triangles of the finer mesh are neglected. The number of edges so decreases at least by a factor 4 per mesh level.

3.4.2 Restriction and Prolongation

Let $\mathcal{T}_{2^i h}$ (i = 0, 1, ..., n) be the nested grids that contain at least one inner edge. Then the prolongation operator to the next finer grid reads, corresponding to the interpolation operator for the auxiliary space method (section 3.2.2):

$$\begin{aligned} P_{2h}^{h} : \mathcal{T}_{2h} &\rightarrow \mathcal{T}_h \\ \boldsymbol{v}_{2h}(\boldsymbol{x}) &\mapsto \boldsymbol{v}_h(\boldsymbol{x}) \\ \phi_i^h &= \int_{e_i^h} \boldsymbol{v}_{2h} \cdot d\boldsymbol{s}. \end{aligned}$$

As restriction matrix $\boldsymbol{R}_h^{2h}$ the transpose of the prolongation matrix is used:

$$\boldsymbol{R}_h^{2h} = (\boldsymbol{P}_{2h}^h)^t$$

.

3.4.3 Multigrid Algorithm

Let $\boldsymbol{A}_h\ \boldsymbol{x}_h = \boldsymbol{b}_h$ be the linear system to solve on $\mathcal{T}_h$ and $\boldsymbol{x}_{h0}$ an initial guess for the solution on $\mathcal{T}_h$. Then the full multigrid algorithm (`FMG`) reads:

$\boldsymbol{x}_h = \texttt{FMG}(\boldsymbol{x}_{h0}, \boldsymbol{A}_h, \boldsymbol{b}_h)$:

If $\mathcal{T}_h$ is the coarsest grid level:

return $\boldsymbol{x_h} = \boldsymbol{A}_h \backslash \boldsymbol{b}_h$

Else:

$\boldsymbol{x_h} = \boldsymbol{x}_{h0}$

for MG_cycle_no $= 1$: number of multigrid cycles

1. Pre-smoothing on $\mathcal{T}_h$
   $\boldsymbol{x}_h$=$\texttt{PRE\_SMOOTHER}(\boldsymbol{x}_h)$
2. Restriction of the residual to the coarser grid
   $\boldsymbol{r}_{2h} = \boldsymbol{R}_h^{2h}(\boldsymbol{b}_h - \boldsymbol{A}_h\ \boldsymbol{x}_h)$
3. Coarse grid correction on $\mathcal{T}_{2h}$
   $\boldsymbol{e}_{2h} = \texttt{FMG}(\boldsymbol{0}_{2h}, \boldsymbol{A}_{2h}, \boldsymbol{r}_{2h})$
4. Prolongation
   $\boldsymbol{e}_h = \boldsymbol{P}_{2h}^h\ \boldsymbol{e}_{2h}$
5. Adding the correction
   $\boldsymbol{x}_h = \boldsymbol{x}_h + \boldsymbol{e}_h$

6. Post-smoothing on $\mathcal{T}_h$
   $\boldsymbol{x}_h$=`POST_SMOOTHER`$(\boldsymbol{x}_h)$

end for
Applying this algorithm to the linear system $\boldsymbol{A}_0\,\boldsymbol{e}_0 = \boldsymbol{r}_0$ on the auxiliary mesh provides an approximation $\widetilde{\boldsymbol{e}}_0$ of $\boldsymbol{e}_0$.
Let $\mathcal{B}_0$ denote the linear operator

$$\begin{aligned}\mathcal{B}_0 \quad &: \quad \mathbb{R}^{Ne} \to \mathbb{R}^{Ne}\\ \mathcal{B}_0\,(\boldsymbol{r}_0) \quad &= \quad \widetilde{\boldsymbol{e}}_0\end{aligned}$$

This operator is symmetric and can be used as a preconditioner for $\mathcal{A}_0$.

3.5 Theoretical Upper Bound of the Condition Number

To get an upper bound for the condition number that is independent of the mesh size, constants have to be found that satisfy conditions (13) - (17). If $\boldsymbol{B}_0$ is a good preconditioner on the auxiliary space, and $\boldsymbol{R}$ a good smoother, constants $\alpha_0, \alpha_1, \lambda_0$ and λ_1 can be found.
Problems arise with the proof of a stable decomposition of the space. In contrast to the standard nodal interpolants in the space of $S^h_{1,0}(\Omega)$, $\mathcal{ND}^0_1$ is not a subspace of H^1 and there is no guarantee for the functions to be continuous. Additionly, although the interpolation operator is the idendity operator for linear functions on triangles of the target mesh, it is hard to say what happens to nonlinear and discontinuous functions.

3.5.1 The Operator $\mathcal{I}$

$\mathcal{I}$ must fulfil the condition

$$||\mathcal{I}w||^2_A \le \beta_1 ||w||^2_{A_0}$$

for β_1 independent of the mesh size h. If A_0 is obtained by the means of a galerkin construction, β_1 is equal 1, because

$$||\mathcal{I}w||^2_A = \boldsymbol{w}^t\boldsymbol{I}^t\boldsymbol{A}\boldsymbol{I}\boldsymbol{w} = \boldsymbol{w}^t\boldsymbol{A}_0\boldsymbol{w} = ||w||_{A_0}.$$

If

$$||w||^2_{A_0} := \int_{\Omega_0} \operatorname{curl} w \ \operatorname{curl} w \, dx + \tau \int_{\Omega_0} w \cdot w \, dx$$

it can be shown that

$$\int_{\Omega_h} \operatorname{curl}(\mathcal{I}w)\operatorname{curl}(\mathcal{I}w)\,dx \quad \le \quad \int_{\Omega_0} \operatorname{curl} w \operatorname{curl} w \, dx :$$

Theorem 1 *If $\mathcal{I}$ is the interpolation operator described in section 3.2.2 then the following holds:*

$$\int_{\Omega_h} curl\,(\mathcal{I}w)\; curl\,(\mathcal{I}w)\; dx \le \int_{\Omega_0} curl\, w\; curl\, w\; dx$$

Proof:
Let $\tilde{\mathcal{T}}_h$ be a triangulation that is a refinement of $\mathcal{T}_h$ and $\mathcal{T}_0$. Then for $T \in \mathcal{T}_h, T^i \subset T, T^i \in \tilde{\mathcal{T}}_h$. From Gauss' theorem it follows that

$$\operatorname{curl}(\mathcal{I}w)|_T = \sum_i \frac{|T^i|}{|T|} \operatorname{curl} w|_{T^i}.$$

With Jensens inequality for the square function and the fact that $\int_{\Omega_h \setminus \Omega_0} \operatorname{curl} w dx = 0$ it follows that

$$\begin{aligned}
|\operatorname{curl}(\mathcal{I}w)|_T|^2 &\le \sum_i \frac{|T^i|}{|T|} |\operatorname{curl} w|_{T^i}|^2 \\
\Rightarrow \int_{\Omega_h} |\operatorname{curl}(\mathcal{I}w)|^2 dx &= \sum_{T\in\mathcal{T}_h} \int_T |\operatorname{curl}(\mathcal{I}w)|_T|^2 dx = \sum_{T\in\mathcal{T}_h} |T||\operatorname{curl}(\mathcal{I}w)|_T|^2 \\
&\le \sum_{T\in\mathcal{T}_h} \sum_{T^i\subset T} |T^i||\operatorname{curl} w|_{T^i}|^2 = \sum_{T\in\tilde{\mathcal{T}}_h} |T||\operatorname{curl} w|^2 dx \\
&= \sum_{T\in\tilde{\mathcal{T}}_0} \int_T |\operatorname{curl} w|^2 dx = \int_{\Omega_0} \operatorname{curl} w \; \operatorname{curl} w\; dx
\end{aligned}$$

This completes the proof.
For the absolute term of the energy norm, no such proof has been found and the numerical results in section 5.6 indicate that possibly none does exist.

3.5.2 The Linear Operator $\mathcal{P}$

Another problem is the operator $\mathcal{P}$. A nodal interpolation operator like I causes problems at the boundaries, since the arificial boundary condition creates curl in the boundary stripe of Ω_0: Let T be a boundary triangle of $\mathcal{T}_0$ with boundary edge e_1. If v $\in \mathcal{V}_h$ has vanishing curl, the tangetial components on T, α_i, would sum up to zero after nodal interpolation. But if α_1 is set to zero, this creates $|\operatorname{curl}(Pv)|^2$ proportional to $\frac{\alpha_1^2}{|T|^2}$, where α scales like O(h). $\int_{\partial\Omega_0\times h} |\operatorname{curl} Pv|^2 dx$ then scales like $\frac{1}{h}$.
This problem can be fixed by using a more theoretical operator which is a

nodal interpolation operator inside $\tilde{\Omega}$ (see figure 3). The edge values in the boundary region are then adapted in order to satisfy $\int_{\Omega_0\backslash\tilde{\Omega}} |\mathrm{curl} Pv|^2 dx \lesssim \int_{\Omega_h\backslash\tilde{\Omega}} |\mathrm{curl} v|^2 dx$.
Let $\partial\tilde{\Omega}$ denote the boundary of this interiour domain. Inside this region, the required inequality is fulfilled according to section 3.5.1. The domain $\Omega_0\backslash\tilde{\Omega}$ is a stripe of width h_0 an its area is approximately $h_0|\partial\tilde{\Omega}|$. The domain $\Omega_h\backslash\tilde{\Omega}$ is thinner than a stripe of width $2h_0$ and has the approximate area of $\delta\, h_0|\partial\tilde{\Omega}|, \delta \in [1,2]$. With Dirichlet boundary conditions and $\int_{\partial\tilde{\Omega}} w \cdot ds = C_1$ it follows that (Green's Theorem)

$$\int_{\Omega_h\backslash\tilde{\Omega}} \mathrm{curl} w dx = C_1$$
$$\int_{\Omega_0\backslash\tilde{\Omega}} \mathrm{curl}(Pw) = C_1$$

If curl (Pw) is chosen to be constant on $\Omega_0\backslash\tilde{\Omega}$ this leads to

$$\mathrm{curl}(Pw) = \frac{C_1}{h_0|\partial\tilde{\Omega}|}.$$

This is approximatly possible if the number of triangles in the stripe $\Omega_0\backslash\tilde{\Omega}$ corresponds to the number of internal edges of this domain. This is fulfilled if the boundary has no fractal structure.
Lets define a constant $\tilde{c}$ as the weighted mean of the curl on the tringles T^i of a triangulation of $\tilde{\Omega}$:

$$\tilde{c} := \sum_i \frac{|T^i|}{|\Omega_h\backslash\tilde{\Omega}|} \mathrm{curl} w|_{T_i}.$$

Then the following holds:

$$\begin{aligned}
\delta h_0 \tilde{c}^2 |\partial\tilde{\Omega}| &\approx \int_{\Omega_h\backslash\tilde{\Omega}} \tilde{c}^2 dx \\
&= \sum_i \int_{T^i} \tilde{c}^2 dx \leq \sum_i \int_{T_i} \sum_j \frac{|T^j|}{|\Omega_h\backslash\tilde{\Omega}|} |\mathrm{curl} w|_{T_j}|^2 dx \\
&= \sum_i \sum_j \frac{|T^i||T^j|}{|\Omega_h\backslash\tilde{\Omega}|} |\mathrm{curl} w|_{T_j}|^2 = \sum_j |T^j||\mathrm{curl} w|_{T_j}|^2 \\
&= \int_{\Omega_h\backslash\tilde{\Omega}} |\mathrm{curl} w|^2 dx \\
\Rightarrow \tilde{c}^2 &\lesssim \frac{1}{\delta h_0 |\Omega_h\backslash\tilde{\Omega}|} \int_{\Omega_h\backslash\tilde{\Omega}} |\mathrm{curl} w|^2 dx
\end{aligned}$$

$$
\begin{aligned}
\delta|\partial\tilde{\Omega}|h_0\tilde{c} &\approx \int_{\Omega_h\setminus\tilde{\Omega}} \tilde{c}dx = \sum_j \int_{T_j} \tilde{c}dx \\
&= \sum_j \int_{T_j} \sum_i \frac{|T^i|}{|\Omega_h\setminus\tilde{\Omega}|}\mathrm{curl}w|_{T_i} = \sum_i \sum_j \frac{|T^i||T^j|}{|\Omega_h\setminus\tilde{\Omega}|}\mathrm{curl}w|_{T_j} \\
&= \sum_i |T_i|\mathrm{curl}w_{T_i} = \int_{\Omega_h\setminus\tilde{\Omega}} \mathrm{curl}w dx = C_1
\end{aligned}
$$

$$
\Rightarrow \int_{\Omega_0\setminus\tilde{\Omega}} |\mathrm{curl}(Pw)|^2 \approx \frac{C_1^2}{h_0|\partial\Omega|} \approx \delta^2\tilde{c}^2|\partial\tilde{\Omega}|h_0 \lesssim \delta^2 \int_{\Omega_h\setminus\tilde{\Omega}} |\mathrm{curl}w|^2 dx
$$

This justifies the assumption that the curl part of the energy norm scales properly, but no upper bound for the second part has been found.

4 Implementation and Computational Costs

In this section, the implementation aspects of the preconditioner are described. A special emphasis lies on the scaling of storage demands and computational costs with the system size (number of edges or vertices of the unstructured mesh) for all parts of the algorithm. The code is written in Matlab (for details see appendix).

4.1 Data Structures

The two main data structures are an unstructured 2D and a structured 2D mesh. All mesh information is stored in a Matlab struct. This implies information about the mesh geometry as well as other fields related with the mesh like stiffness or interpolation matrix. The 2D structured mesh is an extended unstructured mesh with additional fields describing the relevant part of the mesh where the differential equation is solved.

4.2 Setup of the System

Before applying the preconditoner, all system information that is independent of the right hand side function and algorithm parameters, such as geometries of auxiliary meshes, transfer and stiffness matrices, is computed in advance and stored in the above-mentioned mesh structures. Then the preconditioner can be used by any algorithm, for example the CG scheme.

4.2.1 The Unstructured Mesh

The mesh that covers the computational domain of the problem is generated by an external mesh generator such as femlab and converted to a 2D unstructured mesh where the matrices A and T are assembled. Since all these computations are local, the costs scale linearly with the system size.

4.2.2 Construction of the Auxiliary Meshes

Given an unstructured 2D mesh, the nested submeshes are built up in the following way:

1. Creation of a rectangular mesh that covers the whole domain of the unstructured mesh.
   Let $\Delta x \times \Delta y$ be the size of the smallest rectangle that covers the domain of the unstructured mesh and dx the length of a side of the

isosceles triangles on the auxiliary mesh. To simplify the construction of submeshes, there are

$$N_x := \min_{i \in \mathbb{N}} 2^i + 1 > \frac{\Delta x}{dx}$$

vertices in x and

$$N_y := \min_{j \in \mathbb{N}} 2^j + 1 > \frac{\Delta y}{dx}$$

vertices in y direction. This auxiliary mesh domain is not more than four times as large as the smallest rectangle that covers the domain of the unstructured mesh and its storage demands scale linearly with the system size, dependent on the relative size of the triangles of the two meshes and the shape of the unstructured mesh.

2. Determination of the vertices of the structured mesh that are located within the domain of the unstructured mesh. This is done by dividing the area of the auxiliary mesh into boxes and assigning each vertex of both meshes the index of its box. A vertex of the structured mesh can only lie inside a triangle that has at least one vertex inside the same box or inside one of the neighbouring boxes. This leads to a linear scaling with the system size of this classification algorithm.

3. Derivation of the rest of the partial mesh information. Due to pure local computations this again scales linearly with the system size.

4. Recursive submesh construction:

   - Creation of a structured 2D mesh of the same shape as the superordinate one with $N_{x/y} = \frac{N^{sup}_{x/y} - 1}{2} + 1$ vertices in x and y direction, respectively.
   - Evaluation of internal vertices by using the known relations between the two structured meshes.
   - Derivation of the rest of the partial mesh information.

   Let N^{sm}_i be the number of vertices of the i^{th} structured mesh, i=1..i_{max} (with i_{max} the finest mesh level).
   With this recursive mesh definition, the total number of vertices of all submeshes scales linearly with system size:

$$\begin{aligned} N^{sm}_{i_{max}} &= (2^j + 1)(2^k + 1) = 2^{j+k} + 2^j + 2^k + 1 \\ \Rightarrow N^{sm}_{i_{max}-l} &= (2^{(j-l)} + 1)(2^{(k-l)} + 1) \end{aligned}$$

$$
\begin{aligned}
\Rightarrow \sum_{l=1}^{i_{max}} N_l^{sm} &= \sum_{l=0}^{i_{max}-1} (2^{(j-l)}+1)(2^{(k-l)}+1) \\
&= \sum_{l=0}^{i_{max}-1} 2^{(j+k-2l)} + 2^{(j-l)} + 2^{(k-l)} + 1 \\
&= 2^{j+k} \sum_{l=0}^{i_{max}-1} 2^{-2l} + 2^j \sum_{l=0}^{i_{max}-1} 2^{-l} + 2^k \sum_{l=0}^{i_{max}-1} 2^{-l} + i_{max} \\
&\leq 2^{j+k} \cdot 2 + 2^j \cdot 2 + j^k \cdot 2 + i_{max} \\
&< 2 \cdot N_{i_{max}}^{sm} + \frac{\log_2(N_{i_{max}}^{sm})}{2} \\
&< \frac{5}{2} N_{i_{max}}^{sm}
\end{aligned}
$$

4.2.3 Transfer Matrix I

The transfer of a vector field from the auxiliary mesh to the unstructured mesh is done by evaluating the edge integrals of the vector field on the auxiliary mesh along the edges of the target mesh. In a first step, for each edge of the unstructured mesh all intersections with the auxiliary mesh are computed. Since the edge indices on the structured mesh can be obtained from the coordinates of the end points, the computational costs scale linearly with the number of edges on the unstructured mesh.
The integral over an edge is then computed as the sum over the integrals of the sections:

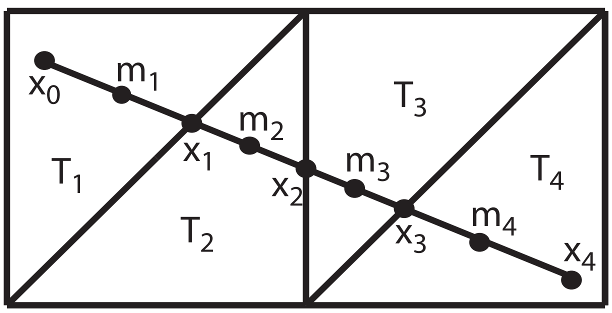

Figure 6: An edge $[x_0\ x_n]$ on the unstructured mesh is divided into n sections by the auxiliary mesh (n=4).

$$
\begin{aligned}
\int_{e^{us}} \boldsymbol{v}^0 \cdot d\boldsymbol{s} &= \sum_{i=1}^{n} \boldsymbol{v}^0(\boldsymbol{m}_i) \cdot (\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{Ne} \alpha_j \boldsymbol{b}_j^0(\boldsymbol{m}_i) \cdot (\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) \\
&= \sum_{j=1}^{Ne} \sum_{i=1}^{n} \boldsymbol{b}_j^0(\boldsymbol{m}_i) \cdot (\boldsymbol{x}_i - \boldsymbol{x}_{i-1}) \alpha_j \\
\Rightarrow \boldsymbol{I}_{e\ j} &= \sum_{i=1}^{n} \boldsymbol{b}_j^0(\boldsymbol{m}_i) \cdot (\boldsymbol{x}_i - \boldsymbol{x}_{i-1})
\end{aligned}
$$

with $\boldsymbol{b}_j^0(\boldsymbol{m}_i) = \boldsymbol{0}$ if e_j is not an edge of triangle T_i.
This interpolation matrix is obviously sparse if the edges of both meshes have approximately the same length.

4.2.4 Multigrid Transfer Matrices $\boldsymbol{P}_{2h}^h$

The known relations between two meshes $\mathcal{T}_h$ and $\mathcal{T}_{2h} \subset \mathcal{T}_h$ allows a very simple computation of the transfer matrix. For an auxiliary mesh with isosceles triangles, the local prolongation matrices for upper triangles ($\boldsymbol{P}_u$) and for lower triangles ($\boldsymbol{P}_l$) (see figures 7 and 8) read:

$$
\boldsymbol{P}_u = \begin{pmatrix}
0.25 & -0.25 & 0.25 \\
0 & 0.5 & 0 \\
0 & 0 & 0.5 \\
0.5 & 0 & 0 \\
0 & 0.5 & 0 \\
0.25 & 0.25 & 0.25 \\
0.5 & 0 & 0 \\
-0.25 & 0.25 & 0.25 \\
0 & 0 & 0.5
\end{pmatrix}
\quad
\boldsymbol{P}_l = \begin{pmatrix}
0.25 & 0.25 & -0.25 \\
0 & 0.5 & 0 \\
0 & 0 & 0.5 \\
0.5 & 0 & 0 \\
0.25 & 0.25 & 0.25 \\
0 & 0 & 0.5 \\
0.5 & 0 & 0 \\
0 & 0.5 & 0 \\
-0.25 & 0.25 & 0.25
\end{pmatrix}.
$$

The assembly of the global transfer matrix is done by identifying local and global indices on both meshes, a known relation between the closely related meshes.

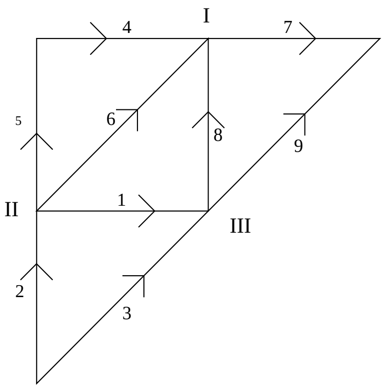

Figure 7: Local edge indices of an upper triangle

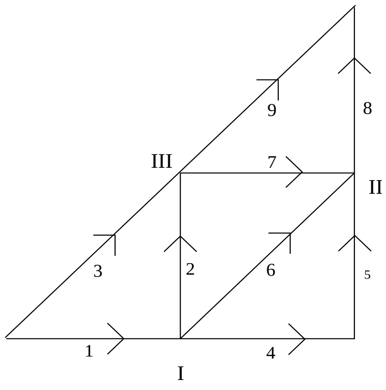

Figure 8: Local edge indices of an lower triangle

4.3 Preconditioner

The preconditioning algorithm consists of transfers of functions from one FE space to another and of smoothing in these spaces.

4.3.1 Transfers and Multigrid

Since all transfers are local, this guarantees linear scaling with the system size. The computational costs of one multigrid cycle scale linearly with the system size if on each level they are proportional to the size of this subsystem. If the computational costs for pre- and post-smoothing and transfers to and from the next coarser grid are proportional to the number number of degrees of freedom, they are approximately $C \cdot 4^i$ on the mesh of level i with system size $N \approx 4^{i_{max}}$. In the V-multigrid scheme, each level is only visited once, so the costs sum up to

$$
\begin{aligned}
C_{V-cycle} = \sum_{i=0}^{i_{max}} C \cdot 4^i \quad &< \quad \frac{3}{2}\, CN && (34)\\
&\gtrsim \quad \frac{5}{4}\, CN. && (35)
\end{aligned}
$$

In the W-multigrid scheme, each level is visited $2^{i_{max}-i+1}$ times which leads to costs of a W-cycle of

$$
C_{W-cycle} = \sum_{i=0}^{i_{max}} C \cdot 4^i \cdot 2^{i_{max}-i+1} = C\, 2^{i_{max}+1} \sum_{i=0}^{i_{max}} 2^i < 4\, C\, N \qquad (36)
$$

The total costs to achieve a given precision depends on the scaling of the convergence rate with the system size.

4.3.2 Smoothing

All smoothing schemes used in the algorithm are based on the Gauss-Seidel method. One *Forward Gauss-Seidel* iteration is equivalent to a loop over all indices:

`for i=1:N`

$$v_i = (b_i - \sum_{j \neq i} (A_{ij} v_j))/A_{ii}$$

`end i`

(The *Backward Gauss-Seidel* scheme passes through the indices in inverted order.) With $\boldsymbol{A}$ sparse, this scales linearly with N. If the scheme is only applied to the boundary region, i runs only over the degrees of freedom in the boundary layer and $\lfloor \frac{N}{\#\text{dof in boundary region}} \rfloor$ Gauss Seidel sweeps can be done instead of one sweep over all indices.

5 Numerical Experiments

The quality of the preconditioner depends on two main factors: on the one hand, it is influenced by the accuracy of the transfer of the problem to the auxiliary mesh and the smoothing properties of the hybrid smoother, i.e. the auxiliary space method, and, on the other hand, the quality of the preconditioner in the auxiliary space - in this case the multigrid scheme - is important. For the follwing experiments the test meshes and the right hand side function described in sections 5.1 and 5.2 are used. Section 5.3 contains the experiments that test the auxiliary space method and section 5.4 those testing the multigrid convergence. Results of the combination of the auxiliary space method with a multigrid preconditioner in the auxiliary space are presented in section 5.5.

5.1 The Unstructured Test Meshes

The numerical experiments are conducted on the meshes that approximate the circle $\Omega := \{\boldsymbol{x} : |\boldsymbol{x} - \boldsymbol{m}| \leq 0.45,\ \boldsymbol{m} = (0.5\ 0.5)^t\}$. The coarsest mesh is shown in figure 9. The meshes II - VI are recursive refinements of this mesh, by dividing each triangle into four subtriangles and adapting the mesh at the boundary.

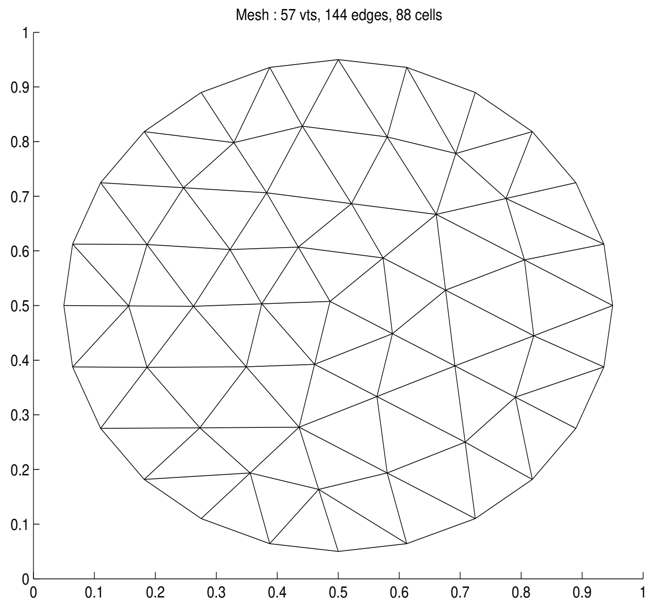

Figure 9: Mesh I

5.2 The Test Function

All tests are performed with the right hand side function

$$\boldsymbol{f}(\boldsymbol{x}) = \frac{3}{10|\boldsymbol{x}|}\begin{pmatrix} x_2 - 0.5 \\ -x_1 - 0.5 \end{pmatrix} + \boldsymbol{v}(\boldsymbol{x})$$

with the exact solution (for $\tau = 1$)

$$\boldsymbol{v}(\boldsymbol{x}) = \begin{pmatrix} (x_1 - 0.5) + 10\ (0.45 - |\boldsymbol{x}|) \cdot (x_2 - 0.5) \\ (x_2 - 0.5) - 10\ (0.45 - |\boldsymbol{x}|) \cdot (x_1 - 0.5) \end{pmatrix}$$

on the domain $\Omega := \{\boldsymbol{x} : |\boldsymbol{x} - \boldsymbol{m}| \leq 0.45,\ \boldsymbol{m} = (0.5\ 0.5)^t\}$.

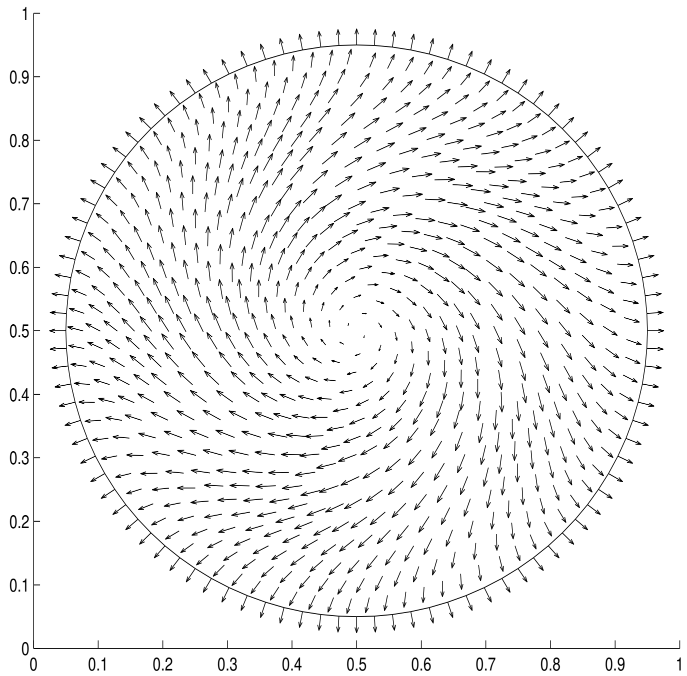

Figure 10: Exact Solution

5.3 Test of the Auxiliary Space Method

In order to test the mesh transfer, the problem on the auxiliary mesh is solved by a direct solver, which is actually not a problem on these rather small test meshes. The condition number of the preconditioned system and

the resulting convergence of the CG scheme are used as indicators of the quality of the auxiliary space method.
The results in section 5.3.4 show the sensivity of these properties with respect to the smoothing on the unstructured mesh, the relative element size of the unstructured and the auxiliary mesh and the weighting factor τ of the absolute term.

5.3.1 Condition Number

The condition number χ of a regular matrix $\boldsymbol{M}$ is defined as

$$\chi(\boldsymbol{M}) := \sqrt{\frac{\max\limits_{||\boldsymbol{x}||_2=1} ||\boldsymbol{M}^t\boldsymbol{M}\boldsymbol{x}||_2}{\min\limits_{||\boldsymbol{x}||_2=1} ||\boldsymbol{M}^t\boldsymbol{M}||_2}} = \sqrt{\frac{||\boldsymbol{M}^t\boldsymbol{M}||_2}{||(\boldsymbol{M}^t\boldsymbol{M})^{-1}||_2}},$$

where $||.||_2$ is the euclidean norm.
$\chi(\boldsymbol{BA})$ is used as a measure for the quality of the preconditioner $\boldsymbol{B}$. $||(\boldsymbol{BA})^t\boldsymbol{BA}||_2 = ||\boldsymbol{ABBA}||_2$ and $||((\boldsymbol{BA})^t\boldsymbol{BA})^{-1}||_2 = ||(\boldsymbol{ABBA})^{-1}||_2$ are evaluated by (inverse) vector iteration.

Forward Vector Iteration

for n = 1,2,...

$$\begin{aligned} \boldsymbol{x}_n &= \boldsymbol{ABBA}\boldsymbol{x}_{n-1} \\ \lambda_n &= \frac{||\boldsymbol{x}_n||_2}{||\boldsymbol{x}_{n-1}||_2} \to ||\boldsymbol{ABBA}||_2^2,\ (n \to \infty) \\ \boldsymbol{x}_n &= \frac{\boldsymbol{x}_n}{||\boldsymbol{x}_n||_2} \end{aligned}$$

end for

Inverse Vector Iteration

for n = 1,2,...

$$\begin{aligned} \boldsymbol{x}_n &= (\boldsymbol{ABBA})^{-1}\boldsymbol{x}_{n-1} \\ \lambda_n &= \frac{|\boldsymbol{x}_n|}{|\boldsymbol{x}_{n-1}|} \to ||(\boldsymbol{ABBA})^{-1}||_2^2,\ (n \to \infty) \\ \boldsymbol{x}_n &= \frac{\boldsymbol{x}_n}{||\boldsymbol{x}_n||_2} \end{aligned}$$

end for

The iterations are stopped if

$$\Big|\frac{\boldsymbol{x}_n}{|\boldsymbol{x}_n|}-\frac{\boldsymbol{x}_{n-1}}{|\boldsymbol{x}_{n-1}|}\Big| \leq 1e-5$$

or if n = 200 is reached. All vector iterations from which the following condition numbers are computed show reliable convergence with these stop criteria.

5.3.2 CG Convergence in the Energy Norm

There exists an analytical error bound in the energy norm which is directly connected with the condition number $\chi(\boldsymbol{BA})$ of the preconditioned system [3]:

$$\frac{||\boldsymbol{e}^k||_A}{||\boldsymbol{e}^0||_A} \leq 2\Big(\frac{\sqrt{\chi(\boldsymbol{BA})}-1}{\sqrt{\chi(\boldsymbol{BA})}+1}\Big)^k,$$

where $\boldsymbol{e}^k = \boldsymbol{x} - \boldsymbol{x}_k$ is the error in the k^{th} step.
Based on the condition number, the theoretical upper bound for the number of CG iterations n_{CG} to reduce the error in the energy norm by a factor c is

$$n_{CG} = \frac{\log\frac{c}{2}}{\log\frac{\sqrt{\chi(\boldsymbol{BA})}-1}{\sqrt{\chi(\boldsymbol{BA})}+1}}.$$

5.3.3 Computational Costs to Solve the Linear System in Praxis

When solving a linear system with a CG scheme, only the relative residual is available as a measure for the accuracy of the solution and it is used as a stop criterion for the algorithm. Therefore, the number of CG steps to reach a certain relative residual is a reasonable measure for the computational costs to solve a linear system in praxis. In the following section, there are experiments where the number of CG iterations to reach a relative residual of 1e-6 are counted.

5.3.4 Numerical Results

The condition number and the resulting CG convergence is analysed for different system parameters:

a) Number of domain smoothing sweeps m ($m_1 = m_2 = m_3 = m$)

b) Number of boundary smoothing sweeps n ($n_1 = n_2 = n_3 = n$) on the edges of which at least one end point belongs to a boundary triangle (one sweep corresponds to $\lfloor\frac{\#\text{dof in domain}}{\#\text{dof in boundary region}}\rfloor$ Gauss Seidel sweeps)

c) Relative mesh size ms, (ratio of dx (length of a side of a right triangle of the structured mesh) and the mean edge length of the unstructured mesh)

d) Constant τ in (1)

All these parameters are tested with both variants of the stiffness matrix on the auxiliary mesh, i.e. galerkin construction (G) and direct computation (D).
Tables 1 and 2 show the dependence of the condition number on the number of domain smoothing sweeps. With galerkin construction of the stiffness matrix, the linear system on the auxiliary mesh gets badly conditioned with decreasing element size, so the tests cannot be done on the meshes V and VI. Figure 11 presents the theoretical upper bounds of CG iterations n_{CG} corresponding to the condition numbers in table 1. The number or CG iterations grows logarithmically with the system size for all smoothing parameters. The number of iterations to reduce the relative residual in the euclidean norm by a factor of 1e-6 are presented in tables 3 and 4. Also this number grows slowly with the system size for these test cases.
The results in tables 5 and 6 indicate that it is reasonable to perform one boundary smoothing sweep on the unstructured mesh, whereas table 8 shows that the condition number does not diverge when τ is scaled down. In table 7 is shown how the relative element size of the unstructured mesh and the auxiliary mesh influences the condition number of the preconditioned system whereas figure 12 shows the corresponding number of CG iterations. Since the number of iterations cannot be reduced essentially by scaling down the relative mesh size, 1.0 is a reasonable choice for the relative mesh size.

m	1	2	3	5
Mesh I	1.2674	1.1912	1.1430	1.0810
Mesh II	1.5752	1.3963	1.3112	1.2231
Mesh III	2.0424	1.6800	1.5162	1.3596
Mesh IV	2.6595	2.0656	1.7969	1.5422
Mesh V	3.4233	2.5494	2.1474	1.7659
Mesh VI	4.4592	3.2242	2.6428	2.0843

Table 1: Condition numbers with varying number of domain smoothing sweeps in combination with direct computation of the stiffness matrix on the auxiliary mesh (n=1, ms=1, $\tau = 1$).

m	1	2	3	5
Mesh I	1.1243	1.0875	1.0658	1.0379
Mesh II	1.3293	1.2387	1.1901	1.1346
Mesh III	1.5421	1.3796	1.2989	1.2158
Mesh IV	1.8868	1.6066	1.4700	1.3331

Table 2: Condition numbers with varying number of domain smoothing iterations with in combination with galerkin construction of the stiffness matrix on the auxiliary mesh (n=1, ms=1, $\tau = 1$).

m	1	2	3	5
Mesh I	5	5	4	4
Mesh II	6	5	5	5
Mesh III	7	6	6	6
Mesh IV	8	7	6	6

Table 3: Number of CG Iterations to reduce the relative residual by a factor of 1e-6 (n=1, ms=1, τ=1, G).

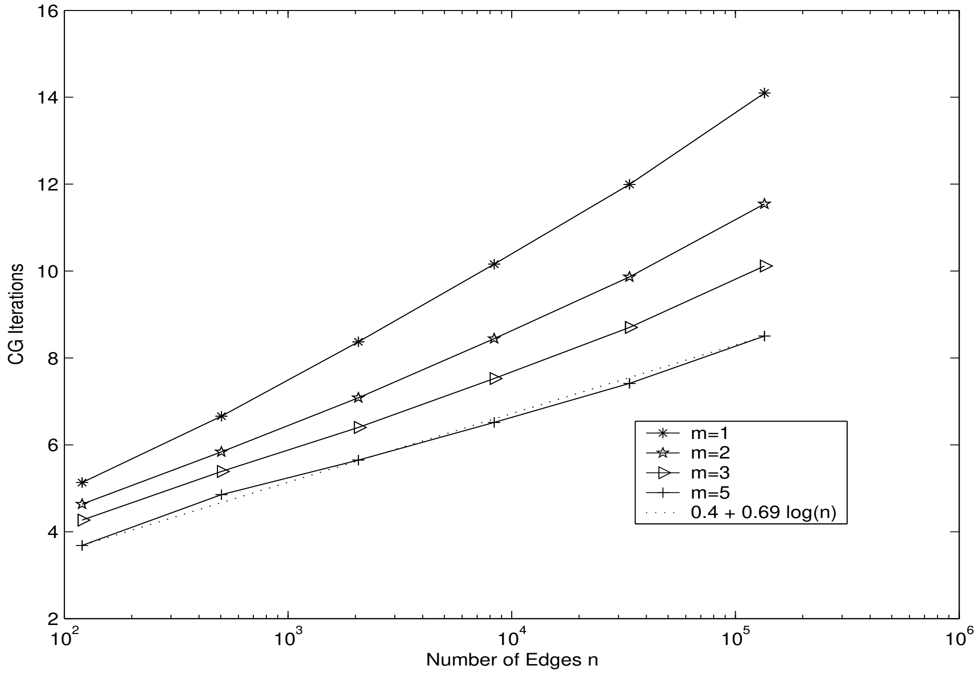

Figure 11: Theoretical number of CG iterations to reduce the error in the energy norm by a factor of 1e-6 (n=1, τ=1, ms=1, D).

m	1	2	3	5
Mesh I	4	4	4	3
Mesh II	5	5	5	4
Mesh III	6	5	5	5
Mesh IV	6	6	5	5
Mesh V	8	7	6	6
Mesh VI	8	7	7	6

Table 4: Number of CG Iterations to reduce the relative residual by a factor of 1e-6 (n=1, ms=1, τ=1, D).

n	0	1	2
Mesh I	1.4148	1.1912	1.1386
Mesh II	1.8671	1.3963	1.3268
Mesh III	2.1868	1.6800	1.6307
Mesh IV	2.8592	2.0656	2.0251
Mesh V	3.5317	2.5494	2.5169
Mesh VI	4.4917	3.2242	3.2060

Table 5: Boundary smoothing with galerkin construction with two domain smoothing sweeps (m=2, ms=1, τ=1).

n	0	1	2
Mesh I	1.2476	1.0875	1.0615
Mesh II	1.6387	1.2387	1.1840
Mesh III	1.7835	1.3796	1.3445
Mesh IV	2.3739	1.6066	1.5727

Table 6: Boundary smoothing with direct computation of the stiffness matrix with two domain smoothing sweeps (m=2, ms=1, τ=1).

ms	0.5	0.8	1.0	1.2	1.5	2.0
Mesh I	1.0927	1.1909	1.1912	1.3648	1.4038	1.5904
Mesh II	1.2042	1.3523	1.3963	1.5489	1.7577	1.7389
Mesh III	1.2872	1.4277	1.6800	1.7712	2.3878	2.7638
Mesh IV	1.3990	1.6307	2.0656	2.4519	2.8872	4.3013

Table 7: Condition numbers dependent on the relative size of the unstructured and the auxiliary mesh (m=2, n=1, τ=1, D).

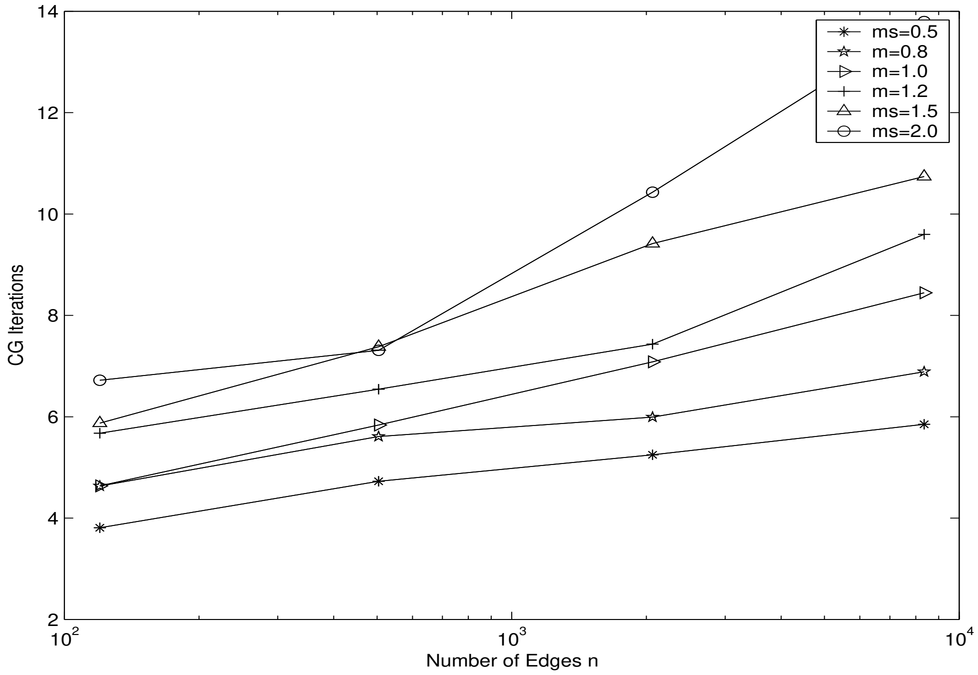

Figure 12: Theoretical number of CG iterations to reduce the error in the energy norm by a factor of 1e-6 with different relative mesh sizes (m=2, n=1, $\tau = 1$, D).

τ	1e-6	1e-3	1	1e3	1e6
Mesh I	1.2069	1.2068	1.1912	1.0000	1.0000
Mesh II	1.4188	1.4188	1.3963	1.0001	1.0000
Mesh III	1.7186	1.7186	1.6800	1.0257	1.0000
Mesh IV	2.1225	2.1224	2.0656	1.1041	1.0000
Mesh V	2.6299	2.6299	2.5494	1.2639	1.0000

Table 8: Dependence of the condition number on parameter τ (m=2, n=1, ms=1, D).

5.4 Test of the Multigrid Solver in the Auxiliary Space

5.4.1 Numerical Convergence Rate

In order to analyse the scaling properties of the multigrid solver with the system size, the numerical convergence rate q of the residual is used.

$$q_i := \frac{||\boldsymbol{f} - \boldsymbol{A}\boldsymbol{x}_i||_2}{||\boldsymbol{f} - \boldsymbol{A}\boldsymbol{x}_{i-1}||_2} \rightarrow q$$

with $\boldsymbol{x}_{i+1} = \texttt{FMG}(\boldsymbol{x}_i, \boldsymbol{A}, \boldsymbol{f})$ (see section 3.4.3) and $\boldsymbol{f} = \boldsymbol{I}^t \boldsymbol{f}^h$.

5.4.2 Numerical Results

The Dirichlet boundary conditions on the submeshes introduce inaccuracies in the boundary layers. Therefore, additional smoothing iterations are performed in these domains. Figures 13 and 14 show the influence of the width of these layers on the convergence rate for multigrid with V- and W-cycles, respectively. In the following numerical experiments the width of the boundary layer is chosen to be 3.
In table 9 the convergence rates for V-and W-cylce mutigrid are compared for both choices of the stiffness matrix. Like before, 'G' stands for stiffness matrix with galerkin construction and 'D' for direct computation of the stiffness matrix. It is obvious that the galerkin construction works neither with the V- nor with the W-cylce multigrid and therefore, in all the following experiments the stiffness matrix is always computed directly.
The sensitivity of the convergence rate with respect to the number of smoothing sweeps in the domain is shown in table 10. The convergence rate increases with the number of degrees of freedom (number of internal edges) for all choices of m. The dependence of the number of multigrid iterations to reduce the residual by a factor of 1e-16 on the number of degrees of freedom is shown in figure 15. Considering the fact that the costs of a W-multigrid cycle exceeds the costs of a V-cylce by a constant factor of approximately $\frac{16}{5}$, the W-cycle multigrid becomes relatively cheaper with increasing system size. If the results in figure 15 are extrapolated to a larger number of degrees of freedom, the number of iterations for W-cycle multigrid with 5 smoothing sweeps becomes (almost) independent of the system size.
Table 11 shows that the convergence rates do not increase when τ is decreased.

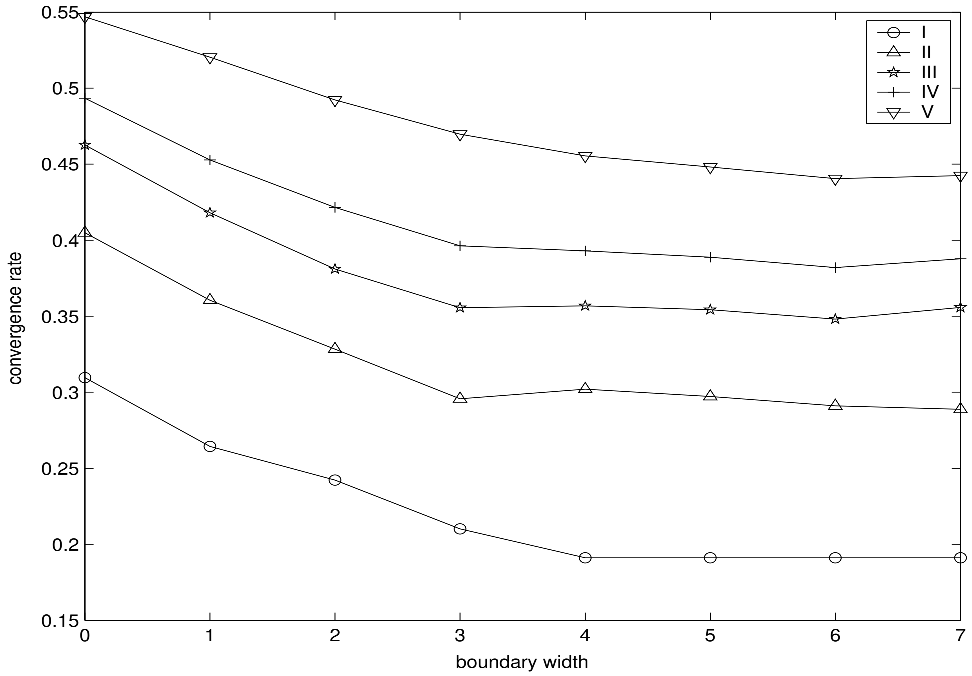

Figure 13: Dependence of the convergence rate of the multigrid V-cycle algorithm on the width of the boundary layer on the meshes I-V.

	D V	D W	G V	G W
Mesh I	0.1874	0.035118	0.03704	0.001372
Mesh II	0.27422	0.041805	0.86378	0.74612
Mesh III	0.33685	0.065237	0.93859	0.89922
Mesh IV	0.38956	0.13131	0.97608	0.97556
Mesh V	0.47737	0.15744	0.97277	0.96931
Mesh VI	0.54112	0.17006	0.97173	0.97248

Table 9: Comparison of galerkin construction and direct computation of the stiffness matrix (m=2, n=2, ms=1, τ=1).

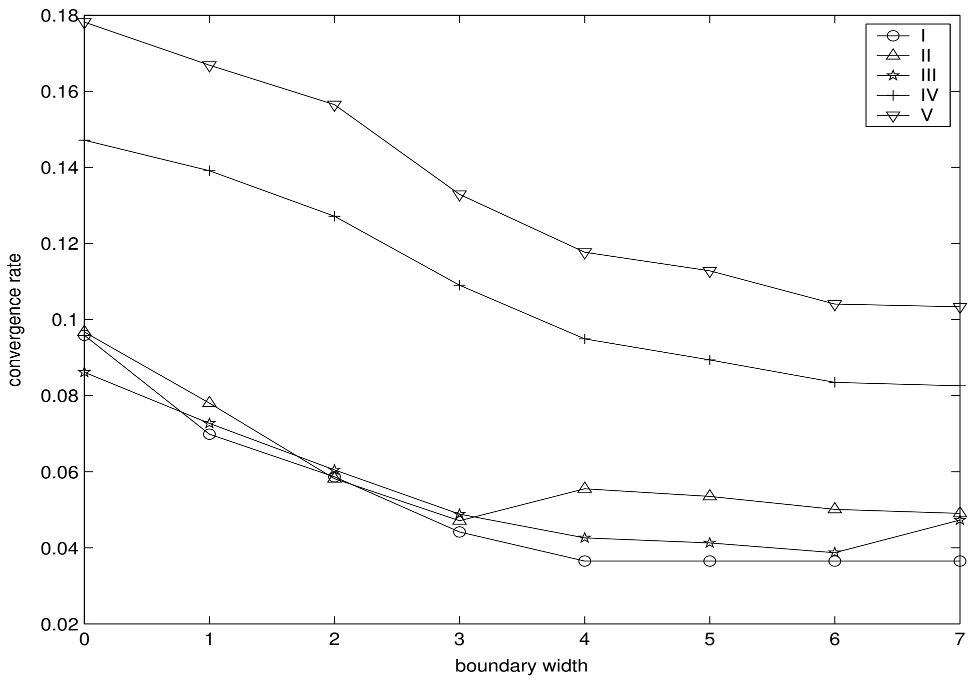

Figure 14: Dependence of the convergence rate of the multigrid W-cycle algorithm on the width of the boundary layer on the meshes I-V.

	V				W			
m	1	2	3	5	1	2	3	4
Mesh I	0.3414	0.2679	0.2101	0.1284	0.1165	0.0717	0.0442	0.0165
Mesh II	0.4242	0.3512	0.2957	0.2188	0.0956	0.0648	0.0471	0.0287
Mesh III	0.4769	0.4083	0.3556	0.2812	0.1377	0.0764	0.0488	0.0285
Mesh IV	0.5216	0.4484	0.3962	0.3226	0.2300	0.1505	0.1090	0.0655
Mesh V	0.6065	0.5279	0.4697	0.3912	0.2635	0.1796	0.1329	0.0828
Mesh VI	0.6558	0.5876	0.5364	0.4583	0.2795	0.1932	0.1441	0.0908

Table 10: Comparison of convergence rates with different number of domain smoothing sweeps (n=1, ms=1, τ=1).

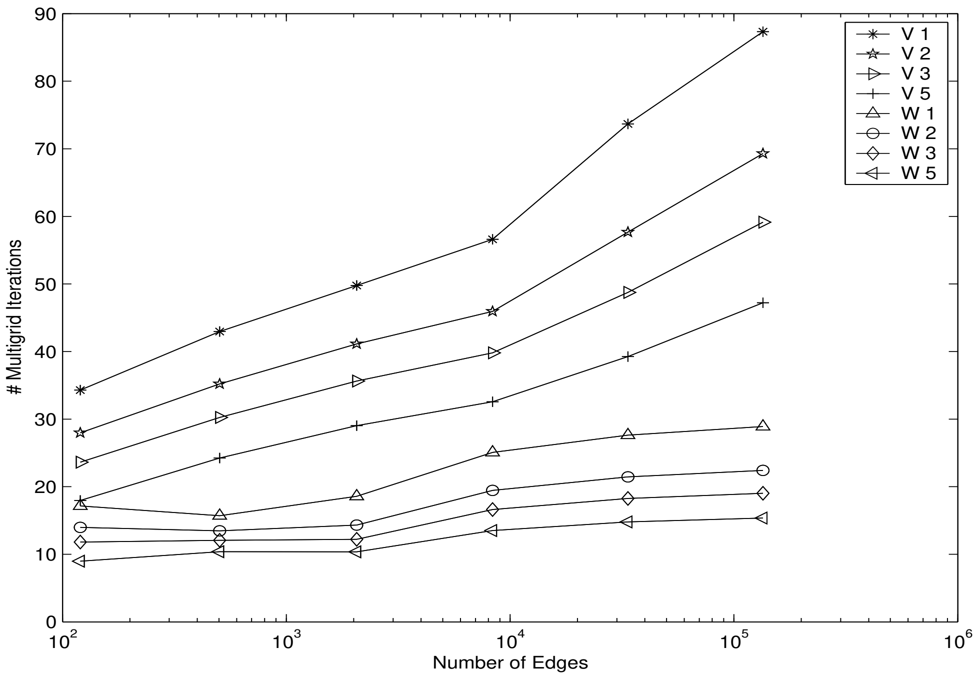

Figure 15: Dependence of the theoretical number of multigrid iterations on the number of degrees of freedom to reduce the residual by a factor of 1e-16.

τ	1e-6	1e-3	1.0	1e3	1e6
Mesh I	0.040013	0.040025	0.035118	1.4498e-13	7.5323e-11
Mesh II	0.04523	0.045229	0.041805	5.7363e-06	1.3204e-08
Mesh III	0.043182	0.065513	0.065237	0.0011432	1.4602e-08
Mesh IV	0.13194	0.13194	0.13131	0.0081309	5.1687e-09
Mesh V	0.15789	0.15789	0.15744	0.024132	1.3006e-10
Mesh VI	0.17037	0.17036	0.17006	0.043652	8.9123e-10

Table 11: τ-dependence of the convergence rate (W-cycle, m=2, n=2).

5.5 Preconditioned CG with Multigrid Solver in the Auxiliary Space

Figures 16 to 19 show the convergence of the preconditioned CG scheme on the test meshes I-VI, when one W-multigrid cycle is applied to precondition the system on the auxiliary mesh. The error in the energy norm ($||\boldsymbol{e}_k||_A$) converges independently of the mesh level for meshes above level 3 with both smoothing schemes, so the computational costs to reduce the energy error scale linearly with the system size.
Figure 19 shows that the convergence of the residual in the euklidean norm is almost independent of the system size. The number of smoothing steps influences only the general convergence rate, not the scaling with the system size.

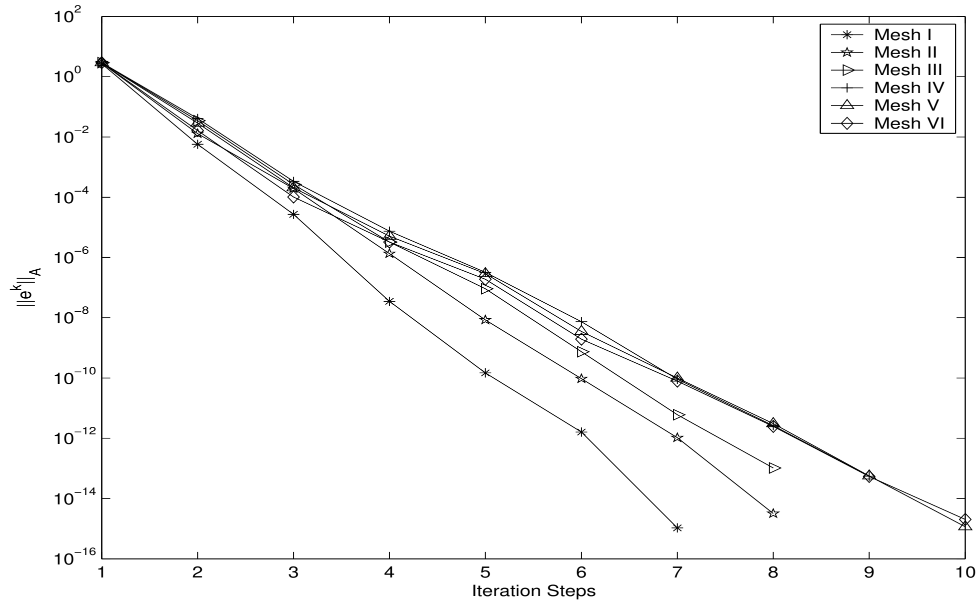

Figure 16: Convergence of the error in the energy norm (m=1, n=1, τ=1, ms=1).

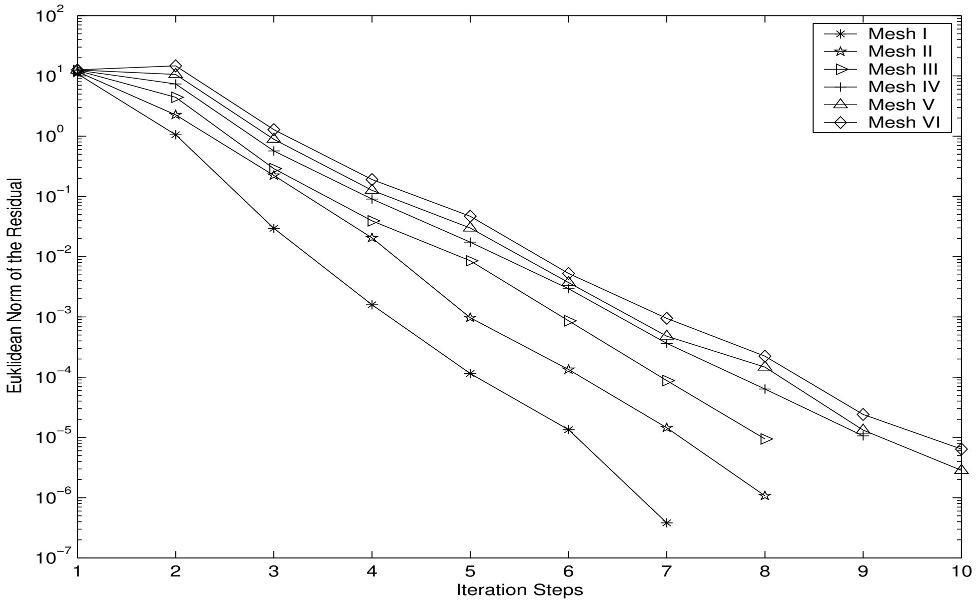

Figure 17: Convercence of the residual (m=1, n=1, τ=1, ms=1).

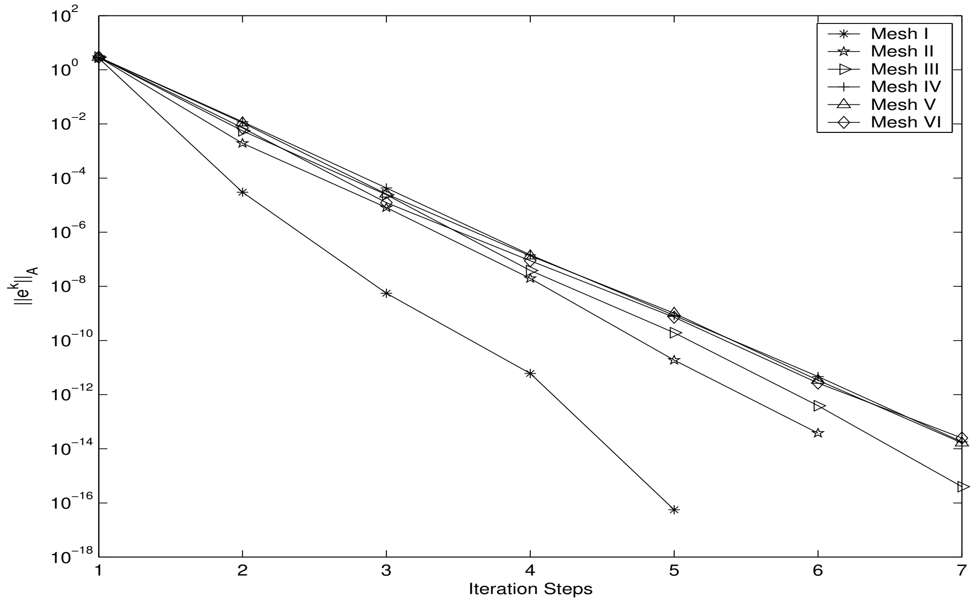

Figure 18: Convergence of the error in the energy norm (m=5, n=1, τ=1, ms=1).

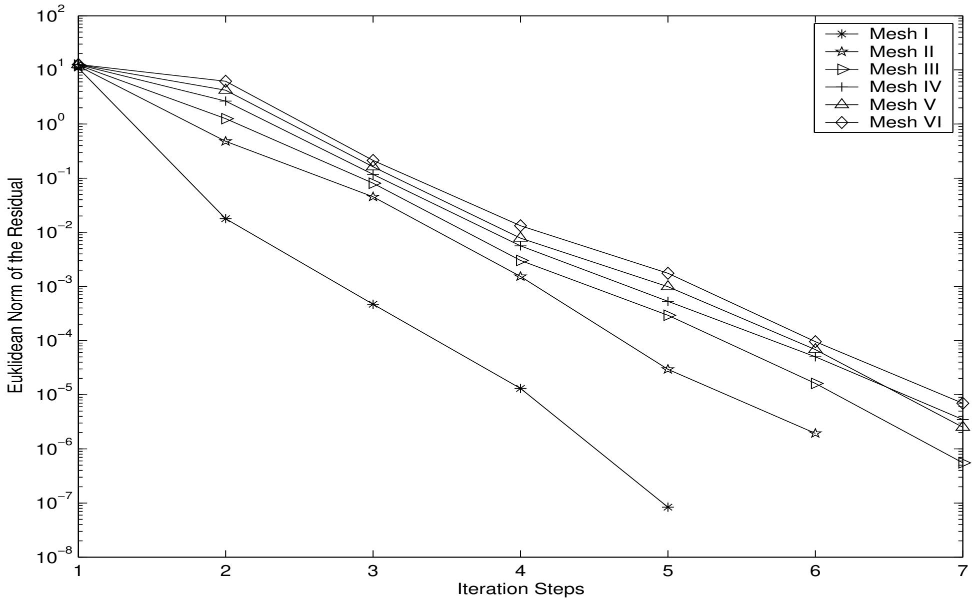

Figure 19: Convercence of the residual (m=5, n=1, τ=1, ms=1).

5.6 Stable Space Decomposition

To guarantee that the condition number of the preconditioned system is independent of the system size, the interpolation operator $\mathcal{I}$ must fulfil:

$$\inf\{h^{-2}||\tilde{v}||_{L^2} + ||w||^2_{A_0} : \tilde{v} + \mathcal{I}w = v, \tilde{v} \in \mathcal{V}, w \in \mathcal{V}_0\} \lesssim ||v||^2_A \forall v \in \mathcal{V}.$$

This could be proved if an operator $\mathcal{P}$ was found that satisfies:

$$h^{-2}||v_h||^2_{L^2} + ||w||^2_{A_0} \lesssim ||v||^2_A$$

with

$$v = v_h + \mathcal{I}v_0, v_0 = \mathcal{P}v, v_h = v - \mathcal{I}\mathcal{P}v.$$

Numerical experiments reveal that this is not the case, if nodal interpolation operators are used for $\mathcal{I}$ and $\mathcal{P}$.
Figures 20 and 21 show the dependence of μ and λ on the (relative) mesh size h for the family of test meshes.

$$||v - \mathcal{I}\mathcal{P}v||_{L^2} \leq \lambda(h)\, h||v||_A \tag{37}$$

$$||Pv||_{A_0} \leq \mu(h)\, ||v||_A \tag{38}$$

In these experiments, μ and λ are definitely not constant and therefore, inequalities (17) and (16) are not fulfilled.

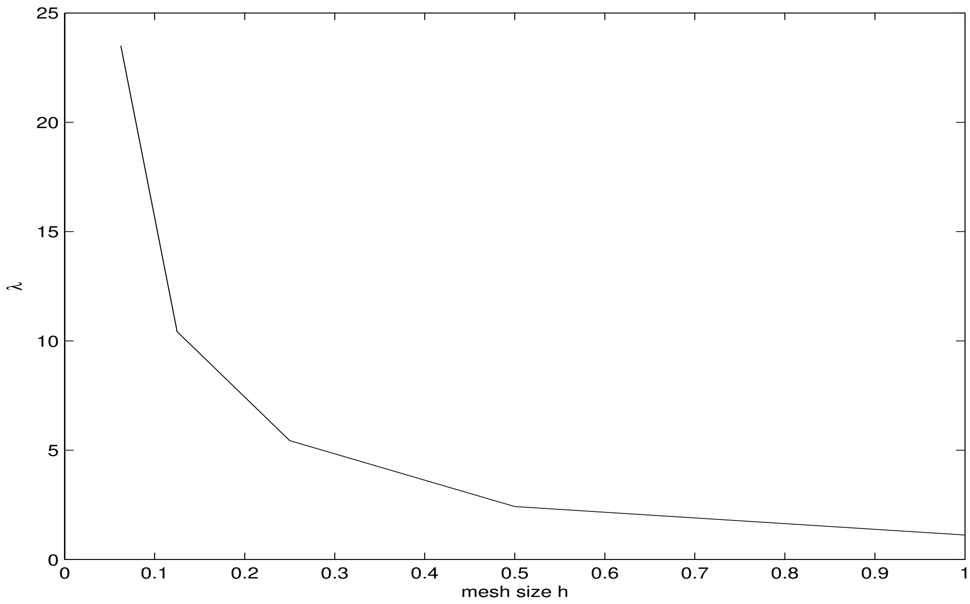

Figure 20: Dependence of λ on the mesh size h.

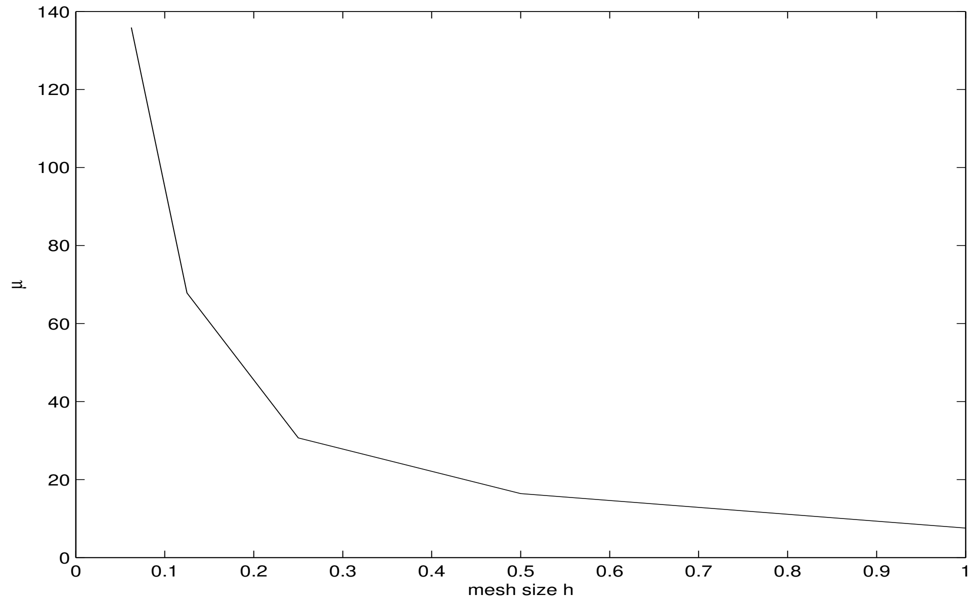

Figure 21: Dependence of μ on the mesh size h.

6 Conclusions and Outlook

In this diploma thesis, an auxiliary space method for edge elements has been implemented to precondition the linear system arising from Maxwell's equations. The computational costs of all parts of the algorithm scale linearly with the system size and for the right hand side function used in the numerical experiments, the number of CG iterations is (almost) independent on the system size. Thus, for this specific problem, with a right hand side function that changes slowly in space, the costs to solve the problem also scales linearly with the system size. However, the condition number shows a slow logarithmic growth, so there will probably be cases where the number of CG iterations will increase when the mesh is refined. The reason for this is assumed to be the insuffiicent interpolation operators $\mathcal{I}$ and $\mathcal{P}$ that allow no stable decomposition of the space. Since the other parts of the algorithm seem to fulfil the conditions for a linear scaling of the method, the challenge now will be to find better interpolation operators to link the original with the auxiliary space.

7 Acknowledgement

I would like to thank Prof. R. Hiptmair for the proposition of this diploma thesis and his support in numerical and analytical mathematics.
Furthermore, I want to acknowledge the staff from the IT support group for their assistance in computer matters.
Special thanks go also to my diploma colleagues E. Kunz and C. Winkelmann for the fruitful discussions and the good working atmosphere.

A Matlab Code

A.1 Mesh Construction

A.1.1 create_and_store_nested_meshes.m

```
function = create_and_store_nested_meshes()

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% creates meshes from femlab MESH structures (e.g. OVAL.mat)
% without assembly of stiffness matrix and boundary
% layer
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% mesh specifications
  s='oval';        %string specifying the mesh in MESH.mat
  MESH='OVAL';     %mat-file containing mesh structures
  mesh_nr=[2:6];   %mesh numbers
  mesh_size=[1 2]; %relative mesh size unstructured mesh<->
                     %auxiliary mesh
  prefix='name';   %prefix of name to store the created mesh

% load mat-file with unstructured mesh geometry
  eval(['load ',MESH]);


  for i=1:length(mesh_nr)
    for j=1:length(mesh_size)

% create 2D mesh structures: unstructured mesh (usm)
% and the finest square-shaped auxiliary mesh (sm),covering the
% at least [0 1][0 1]
      mesh=eval(strcat(s,num2str(mesh_nr(i))));
      p=mesh.p;
      e=mesh.e;
      t=mesh.t;
      [sm,usm]=create_log2_grids(p,e,t,mesh_size(j));

% Transfer matrix vertices -> edges
```

```
      usm.T = liftv2e(usm);

%add information of the partial mesh (part of the auxiliary mesh
%that approximates the unstructured mesh)
      sm = partialsm(sm,usm);

% create nested submesh structure
      sm = submesh(sm,10);

% Transfer matrix auxiliary mesh -> unstructured mesh
      sm.I=transfers2us(sm,usm);
      usm.sm = sm;
      eval(['save ',prefix,MESH,...
            'meshes',num2str(mesh_nr(i)),...
            '_size_',num2str(mesh_size(j)),' usm']);

    end
  end
return
```

A.1.2 create_log2_grids.m

```
function[sg,us] = create_log2_grids(p,e,t,scale)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% creates a structured and an unstructured
% mesh
% p,e,t : points, edges and triangles of the
%         unstructured mesh (output of pdetool or femlab)
% scale:  relative edge length of unstructured and
%         structured mesh
% sg: structured 2D mesh inside the domain [0 1]x[0 1] !!!
% us: unstructured 2D mesh
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% creation on the unstructured mesh
  us=unstruct_convert(p,e,t);
```

```
% dx: side length of the triangle on the aux. mesh
  v=us.vt(us.ep(:,1),:)-us.vt(us.ep(:,2),:);
  h=sqrt(v(:,1).^2+v(:,2).^2);
  dx=mean(h)*scale;

% number of triangle pairs in x and y direction to cover
% the square [0 1] x [0 1]
  N = ceil(log2(1/dx));
% !!! to get correct submesh interpolation,
% dx = dy for structured meshes!!!
  sg=structGrid(0,2^N*dx,0,2^N*dx,2^N+1,2^N+1);
return
```

A.1.3 unstruct_convert.m

```
function mesh = unstruct_convert(p,e,t)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conversion of simple mesh data structure from a mesh
% generated by pdetool or femlabto
% an extended data format describing an unstructured mesh
% of any shape
%
%
% p -> Meshpoints 2xN matrix
% e -> Boundary Edges 7x? matrix
% t -> Triangles 4x? matrix
%
% Data structure for describing a mesh
%
% mesh.Nv       -> # of vertices
% mesh.Ne       -> # of edges
% mesh.Nt       -> # of triangles
% mesh.vt(Nv,2) -> (x,y)-coordinates of vertices
% mesh.vbfl(Nv,1) -> boundary flag for vertex
%                  vbfl(k) = 1 -> vertex k is located on the
%                  boundary
%                          = 0 -> vertex k is not located on
%                  the boundary
% mesh.ep(Ne,2) -> index numbers of endpoints of edges
% mesh.ebfl(Ne,1) -> boundary flags for edges
```

```
%                    ebfl(k) = 1 -> edge is not located on
%                    the boundary
%                    ebfl(k) = 0 -> edge is not boundary
% mesh.etr(Ne,2) -> index numbers of triangles adjacent to an
%                   edge
% mesh.trv(Nt,3) -> index numbers of vertices of triangles
% mesh.tre(Nt,3) -> index numbers of edges of triangles
%                    tre(k,i) refers to the edge opposite the
%                    i-th vertex
%                    of the k-th triangle
% mesh.treo(Nt,3) -> relative orientations of edges of triangles
%                   (all vertices of triangles have to be
%                    arranged in counterclockwise order)
%
% Gisela Widmer 3/11/2003
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  x = p(1,:)';
  y = p(2,:)';
  tri = t([1 2 3],:)';
  N = length(x);

  mesh.Nv = N; % Total number of vertices
  mesh.Nt = size(tri,1); % Total number of triangles

% Copy vertex coordinates
  mesh.vt = zeros(N,1);
  for i=1:N
    mesh.vt(i,1) = x(i);
    mesh.vt(i,2) = y(i);
  end

% Copy vertex numbers of triangles
% make sure that triangles have uniform orientation
  mesh.trv = [];

  for i = tri'
    if (det(mesh.vt([i(2) i(3)],:) - mesh.vt([i(1) i(1)],:)) > 0)
      mesh.trv = [mesh.trv; i'];
    else
      mesh.trv = [mesh.trv; i([1 3 2])'];
```

```
    end
  end

% Retrieve edges from incidence information for triangles and vertices
  EL = [sort([mesh.trv(:,1) mesh.trv(:,2)]')' (1:mesh.Nt)' ...
        3*ones(mesh.Nt,1); ...
        sort([mesh.trv(:,1) mesh.trv(:,3)]')' (1:mesh.Nt)' ...
        2*ones(mesh.Nt,1); ...
        sort([mesh.trv(:,2) mesh.trv(:,3)]')' (1:mesh.Nt)' ...
        1*ones(mesh.Nt,1)];

% mesh.Nt, size(EL)
  [B,i1,j1] = unique(EL(:,1:2),'rows');
  [B,i2,j3] = unique(flipud(EL(:,1:2)),'rows');
  EM = [EL(i1,1:4) EL((3*mesh.Nt)-i2+1,3:4)];

% EM(k,1) -> first endpoint of edge k
% EM(k,2) -> second endpoint of edge k
% EM(k,3) -> index of first adjacent triangle of edge k
% EM(k,4) -> local index of edge k in triangle EM(k,3)
% EM(k,5) -> index of second adjacent triangle of edge k
%            (if edge is on the boundary EM(k,3) = EM(k,5) !)
% EM(k,6) -> local index of edge k in triangle EM(k,5)
  mesh.ep = EM(:,1:2); mesh.etr = EM(:,[3 5]);
  mesh.Ne = size(EM,1);

% Set boundary flags for edges
  mesh.ebfl = zeros(mesh.Ne,1);
  mesh.ebfl(find(EM(:,3) == EM(:,5))) = 1;

% Run through the boundary edges and set boundary flags for vertices

  mesh.vbfl = zeros(mesh.Nv,1);
  mesh.vbfl(e(1,:))=1;
  mesh.vbfl(e(2,:))=1;

% Set edges for triangles
  mesh.tre = zeros(mesh.Nt,3);
  for i=1:mesh.Ne
    mesh.tre(EM(i,3),EM(i,4)) = i;
    mesh.tre(EM(i,5),EM(i,6)) = i;
```

```
  end

% Determine orientation of edges w.r.t. to triangle
mesh.treo = zeros(mesh.Nt,3);
for i=1:mesh.Nt
  ed1 = [mesh.trv(i,2) mesh.trv(i,3)];
  ed2 = [mesh.trv(i,3) mesh.trv(i,1)];
  ed3 = [mesh.trv(i,1) mesh.trv(i,2)];
  if (ed1 == mesh.ep(mesh.tre(i,1),:)), mesh.treo(i,1) = 1;
  elseif (fliplr(ed1) == mesh.ep(mesh.tre(i,1),:)), ...
         mesh.treo(i,1) = -1;
  else fprintf('ERROR Edge 1 of triangle
      error('Inconsistent edge');
  end
  ed2 = [mesh.trv(i,3) mesh.trv(i,1)];
  if (ed2 == mesh.ep(mesh.tre(i,2),:)), mesh.treo(i,2) = 1;
  elseif (fliplr(ed2) == mesh.ep(mesh.tre(i,2),:)), ...
         mesh.treo(i,2) = -1;
  else fprintf('ERROR Edge 2 of triangle
    error('Inconsistent edge');
  end
  ed3 = [mesh.trv(i,1) mesh.trv(i,2)];
  if (ed3 == mesh.ep(mesh.tre(i,3),:)), mesh.treo(i,3) = 1;
  elseif (fliplr(ed3) == mesh.ep(mesh.tre(i,3),:)), ...
         mesh.treo(i,3) = -1;
  else fprintf('ERROR Edge 3 of triangle
    error('Inconsistent edge');
  end
end
```

A.1.4 structGrid.m

```
function mesh=structGrid(xmin, xmax, ymin, ymax, Nx, Ny)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Creates 2D structured rectangular mesh that covers the
% domain [xmin xmax] x [ymin ymax]
% and has Nx vertices in x direction and Ny vertices in y
% direction
%
```

```
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% x and y coordinates of the mesh
  v = [0:Nx-1]*(xmax-xmin)/(Nx-1)+xmin;
  w = [0:Ny-1]*(ymax-ymin)/(Ny-1)+ymin;
% coordinates of vertices
  p=[];
  for i=1:Ny
    p=[p;v',w(i)*ones(Nx,1)];
  end

% triangle list (vertex mapping)
  tr=[];
  for j=1:Ny-1
    for k=1:Nx-1
      i=(j-1)*Nx +k;
      tr=[tr; i, i+Nx+1, i+Nx; i, i+1, i+Nx+1];
    end
  end

% conversion to a 2D mesh
  mesh=convert(p(:,1),p(:,2),tr);

% add additional information
  mesh.Nx=Nx;
  mesh.Ny=Ny;
  mesh.dx=xmax-xmin;
  mesh.dy=ymax-ymin;
  mesh.orig=[xmin,ymin];
return
```

A.1.5 convert.m

```
function mesh = convert(x,y,tri)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% x: Mx1  array of x-coordinates
% y: Mx1  array of y coordinates
% tri:Nx3 mapping between triangles and coordinates
```

```
%
% Data structure for describing a mesh
%
% mesh.Nv       -> # of vertices
% mesh.Ne       -> # of edges
% mesh.Nt       -> # of triangles
% mesh.vt(Nv,2) -> (x,y)-coordinates of vertices
% mesh.vbfl(Nv,1) -> boundary flag for vertex
%                  vbfl(k) = 1 -> vertex k is located on the
%                  boundary
%                          = 0 -> vertex k is not located on
%                  the boundary
% mesh.ep(Ne,2) -> index numbers of endpoints of edges
% mesh.ebfl(Ne,1) -> boundary flags for edges
%                    ebfl(k) = 1 -> edge is not located on
%                    the boundary
%                    ebfl(k) = 0 -> edge is not boundary
% mesh.etr(Ne,2) -> index numbers of triangles adjacent to an
%                   edge
% mesh.trv(Nt,3) -> index numbers of vertices of triangles
% mesh.tre(Nt,3) -> index numbers of edges of triangles
%                   tre(k,i) refers to the edge opposite the
%                   i-th vertex
%                    of the k-th triangle
% mesh.treo(Nt,3) -> relative orientations of edges of
%                    triangles
%                   (all vertices of triangles have to be
%                    arranged in
%                    counterclockwise order)
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  mesh.Nv = length(x); % Total number of vertices
  mesh.Nt = size(tri,1); % Total number of triangles

% Copy vertex coordinates
  mesh.vt = zeros(N,1);
  for i=1:N
    mesh.vt(i,1) = x(i);
    mesh.vt(i,2) = y(i);
```

```
  end

% Copy vertex numbers of triangles
% make sure that triangles have uniform orientation
  mesh.trv = [];

  for i = tri'
    if (det(mesh.vt([i(2) i(3)],:) - mesh.vt([i(1) i(1)],:)) > 0)
      mesh.trv = [mesh.trv; i'];
    else
      mesh.trv = [mesh.trv; i([1 3 2])'];
    end
  end

% Retrieve edges from incidence information for triangles and vertices
  EL = [sort([mesh.trv(:,1) mesh.trv(:,2)]')' (1:mesh.Nt)'...
       3*ones(mesh.Nt,1); ...
       sort([mesh.trv(:,1) mesh.trv(:,3)]')' (1:mesh.Nt)' ...
       2*ones(mesh.Nt,1); ...
       sort([mesh.trv(:,2) mesh.trv(:,3)]')' (1:mesh.Nt)' ...
       1*ones(mesh.Nt,1)];

% mesh.Nt, size(EL)

  [B,i1,j1] = unique(EL(:,1:2),'rows');
  [B,i2,j3] = unique(flipud(EL(:,1:2)),'rows');
  EM = [EL(i1,1:4) EL((3*mesh.Nt)-i2+1,3:4)];

% EM(k,1) -> first endpoint of edge k
% EM(k,2) -> second endpoint of edge k
% EM(k,3) -> index of first adjacent triangle of edge k
% EM(k,4) -> local index of edge k in triangle EM(k,3)
% EM(k,5) -> index of second adjacent triangle of edge k
%            (if edge is on the boundary EM(k,3) = EM(k,5) !)
% EM(k,6) -> local index of edge k in triangle EM(k,5)
  mesh.ep = EM(:,1:2); mesh.etr = EM(:,[3 5]);
  mesh.Ne = size(EM,1);

% Set boundary flags for edges
  mesh.ebfl = zeros(mesh.Ne,1);
  mesh.ebfl(find(EM(:,3) == EM(:,5))) = 1;
```

```
% Run through the boundary edges and set boundary flags for vertices
  mesh.vl = zeros(mesh.Nv,2);
  mesh.vbfl = zeros(mesh.Nv,1);
  for i= find(mesh.ebfl(:,1) == 1)'

    edge_vertices = mesh.ep(i,:)';
    edge_direction = (mesh.vt(edge_vertices(2),:) - ...
              mesh.vt(edge_vertices(1),:))';
    if (abs(edge_direction(1)) < 10*eps)
      mesh.vl(edge_vertices,1) = mesh.vl(edge_vertices,1)+1;
      mesh.vbfl(edge_vertices) = 1;
    elseif (abs(edge_direction(2)) < 10*eps)
      mesh.vl(edge_vertices,2) = mesh.vl(edge_vertices,2)+1;
      mesh.vbfl(edge_vertices) = 1;
    else

      error('Code can only cope with either horizontal ...
             or vertical edges'); end
end

% Set edges for triangles
  mesh.tre = zeros(mesh.Nt,3);
  for i=1:mesh.Ne
    mesh.tre(EM(i,3),EM(i,4)) = i;
    mesh.tre(EM(i,5),EM(i,6)) = i;
  end

% Determine orientation of edges w.r.t. to triangle
  mesh.treo = zeros(mesh.Nt,3);

  for i=1:mesh.Nt
    ed1 = [mesh.trv(i,2) mesh.trv(i,3)];
    ed2 = [mesh.trv(i,3) mesh.trv(i,1)];
    ed3 = [mesh.trv(i,1) mesh.trv(i,2)];
    if (ed1 == mesh.ep(mesh.tre(i,1),:)), mesh.treo(i,1) = 1;
    elseif (fliplr(ed1) == mesh.ep(mesh.tre(i,1),:)), ...
        mesh.treo(i,1) = -1;
    else fprintf('ERROR Edge 1 of triangle
      error('Inconsistent edge');
    end
```

```
    ed2 = [mesh.trv(i,3) mesh.trv(i,1)];
    if (ed2 == mesh.ep(mesh.tre(i,2),:)), mesh.treo(i,2) = 1;
    elseif (fliplr(ed2) == mesh.ep(mesh.tre(i,2),:)), ...
        mesh.treo(i,2) = -1;
    else fprintf('ERROR Edge 2 of triangle
      error('Inconsistent edge');
    end
    ed3 = [mesh.trv(i,1) mesh.trv(i,2)];
    if (ed3 == mesh.ep(mesh.tre(i,3),:)), mesh.treo(i,3) = 1;
    elseif (fliplr(ed3) == mesh.ep(mesh.tre(i,3),:)), ...
        mesh.treo(i,3) = -1;
    else fprintf('ERROR Edge 3 of triangle
      error('Inconsistent edge');
    end
  end
return
```

A.1.6 liftv2e.m

```
function [T]=liftv2e(mesh)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transfer matrix vertex-dofs-> edge-dofs, used to lift a
% vectorfield to
% the potential space (and back)
%
% mesh: 2D mesh structure
% T: dof(edges) x dof(vertices)
%
% Gisela Widmer 21.1.04
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  T = sparse(mesh.Ne,mesh.Nv);
  for i=1:mesh.Ne
    T(i,mesh.ep(i,1))=-1;
    T(i,mesh.ep(i,2))=1;
  end
  T=T(Eactidx(mesh),Vactidx(mesh));
return
```

A.1.7 Eactidx.m

```
function act_idx = Eactidx(mesh)

% extracts indices of interior edges from mesh data structure
  act_idx = find(mesh.ebfl(:,1) == 0);
return
```

A.1.8 Vactidx.m

```
function act_idx = Vactidx(mesh)
% extractes indices for interior vertices of the mesh
  act_idx = find(mesh.vbfl == 0)';
return
```

A.1.9 partialsm.m

```
function[sm]=partialsm(sm,us)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% evaluates which part of the structured mesh
% lies within the area of the unstructured mesh
% and stores the information in additional fields
% of the structured sm
%
% sm: 2D structured square-shaped mesh that covers the
%     region of the unstructured mesh
% us: unstructured 2D mesh
%
%
% partial mesh fields:
% sm.part_vt : indices of vertices of the partial mesh
% sm.part_int_vt : indices of internal vertices of the partial
%                  mesh
% sm.partr  :            triangles
% sm.partine:            internal edges
% sm.partbe :            boundary edges
% sm.part_efl: edge flags for enclosing auxiliary mesh:
%               0: internal edge
%               1: boundary edge
```

```
% sm.T: matrix for transfer into potential space
%
% Gisela Widmer 19.2.04
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% indices of all nodes of the structured mesh
% within and on the boundary of the unstructured mesh
  [inner_vt] = innervertices(sm,us);

%triangles of the aux mesh entirely within the unstructured mesh
  [innertr] =innertriangles(inner_vt,sm);
%internal and boundary edges
  [innere,boundarye]=inneredges(innertr,sm);
% all vertices that belong to the partial mesh
  sm.part_vt  =  inner_vt;


% vertices inside the partial mesh (without vertices on the
% partial meshboundary)
  int_vertices = get_int_vertices(innertr,boundarye,sm);

% add remaining information of the partial mesh of the
% structured mesh within the area of the
% unstructured mesh

  sm.partr   =  innertr;
  sm.partine =  innere;
  sm.partbe  =  boundarye;
  sm.part_int_vt = int_vertices;

% transfer matrix vertex-dofs-> edge-dofs, used to lift a
% vectorfield to the potential space (and back)
  sm.T = part_liftv2e(sm);
return
```

A.1.10 innervertices.m

```
function[inner_vt]=innervertices(sm,usm)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% extraction of vertices of the structured mesh inside the
% domain of the unstructured mesh
%
% sm: 2D structured mesh
% usm: 2D unstructured mesh
%
% inner_vt: global indices of sm vertices inside the domain of
%           usm
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  inner_vt=[];

% Sort vertices into boxes
  [usmBoxNr,smBoxNr,N] = Box_nr(usm,sm);

% run through boxes
  for i=1:max([usmBoxNr,smBoxNr])

% index of neighboring boxes of box i
    if mod(i,N)~=1 & mod(i,N)~=N % not left or rigth boundary
      neighb_idx = [i-N-1,i-N,i-N+1,i-1,i,i+1,i+N-1,i+N,i+N+1];
    elseif mod(i,N)==1
      neighb_idx = [i-N,i-N+1,i,i+1,i+N,i+N+1];
    else
      neighb_idx = [i-N-1,i-N,i-1,i,i+N-1,i+N];
    end

% usm vertex indices in neighboring boxes
    usm_vt =find(ismember(usmBoxNr,neighb_idx));

% sm vertices in box i
    sm_vt = find(smBoxNr==i);

    tr=[];
    for j=1:length(usm_vt)
      tr = [tr ,find(usm.trv(:,1)==usm_vt(j))',...
       find(usm.trv(:,2)==usm_vt(j))',...
       find(usm.trv(:,3)==usm_vt(j))'];
```

```
    end
    tr = unique(tr);

    for k=1:length(tr)
      for l=1:length(sm_vt)
        if (is_inside(sm.vt(sm_vt(l),:),usm.vt(usm.trv(tr(k),1),:),...
            usm.vt(usm.trv(tr(k),2),:),usm.vt(usm.trv(tr(k),3),:)))
          inner_vt = [inner_vt,sm_vt(l)];
        end
      end
    end
  end
return
```

A.1.11 Box_nr.m

```
function [usmBoxNr,smBoxNr,N] = Box_nr(usm,sm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Devides the domain of the structured mesh sm into boxes
% of the size of two sm triangles and assigns each vertex
% of both meshes the index of the box it belongs to
%
% usm: unstructured 2D mesh
% sm: structured 2D mesh
%
% usmBoxNr: box indices of the vertices of the unstructred mesh
% smBoxNr:  "                                 structured  "
% number of boxes in x direction
%
% Gisela Widmer 19/2/04
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% dx: maximal edge length of the unstructured mesh-> length
%     of box sides
  v=usm.vt(usm.ep(:,1),:)-usm.vt(usm.ep(:,2),:);
  dx = max(sqrt(v(:,1).^2+v(:,2).^2));

% assign x and y box indices to all vertices
  for i=1:usm.Nv
```

```
    usm_idx_x(i) = ceil((usm.vt(i,1)-sm.orig(1))/dx);
    usm_idx_y(i) = ceil((usm.vt(i,2)-sm.orig(2))/dx)  ;
  end
  for i=1:sm.Nv
    sm_idx_x(i) = ceil((sm.vt(i,1)-sm.orig(1))/dx);
    sm_idx_y(i) = ceil((sm.vt(i,2)-sm.orig(2))/dx);
  end
% number of boxes used in x direction
  N=max([usm_idx_x,sm_idx_x]);

% global box indices
  usmBoxNr = usm_idx_y*N+usm_idx_x;
  smBoxNr = sm_idx_y*N+sm_idx_x;
return
```

A.1.12 is_inside.m

```
function inside = is_inside(x,a,b,c)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test whether the 2D point x is inside (or on the
% boundary of) the triangle
% with counterclockwise corners a,b and c
%
% x,a,b,c 2D vectors (2x1 or 1x2)
% inside -> true or false
%
% Gisela Widmer 5/3/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  v= b-a;
  w= x-a;
  if v(1)*w(2)-v(2)*w(1)< 0
     inside = false;
     return
  else
    v= c-b;
    w= x-b;
    if v(1)*w(2)-v(2)*w(1)< 0
      inside = false;
```

```
      return
    else
      v= a-c;
      w= x-c;
      if v(1)*w(2)-v(2)*w(1)< 0
         inside = false;
         return
      else
         inside=true;
      end
    end
  end
return
```

A.1.13 innertriangles.m

```
function[innert]=innertriangles(inner_nodes,sm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Given a structured mesh sm and a list of the indices
% of the inner nodes, the indices of the triangles
% where all corners belong to the list of inner nodes
% are evaluated
%
% inner_nodes: ?x1 vector with vertex indices
% sm         : 2D structured mesh
% innert     : ?x1 vector with triangle indices
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  counter=0;
  for i=1:sm.Nt
      if(find(sm.trv(i,1)==inner_nodes)&...
         find(sm.trv(i,2)==inner_nodes)&...
         find(sm.trv(i,3)==inner_nodes))
            counter = counter +1;
            innert(counter)=i;
      end
  end
```

```
return
```

A.1.14 inneredges.m

```
function[innere,boundarye]=inneredges(innertr,sm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Given a structured mesh sm and a list of indices of
% triangles of this mesh, the inner edges and the ones
% on the boundary of the area covered by the selected
% triangles are computed
%
% innertr:  ?x1 vector with triangle indices
% sm:       2D structured mesh
% innere:   inner edges of the area specified by
%           innertr
% boundarye:  boundary edges
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% indices of edges that belong to at least one
% of the triangles in innertr
  edges=unique(sm.tre(innertr,:));

  innere=[];
  boundarye=[];

% Loop over all edges of the selected region
  for i=1:length(edges)
%Triangles that are adjacent to the current edge
     find(sm.tre(innertr,:)==edges(i));

% if there are two adjacent triangles, it is an
% inner edge otherwise, the edge belongs to the
% boundary
     if length(find(sm.tre(innertr,:)==edges(i)))==2
        innere=[innere,edges(i)];
     else
        boundarye=[boundarye,edges(i)];
```

```
    end
  end
return
```

A.1.15 get_int_vertices.m

```
function[int_vertices]=get_int_vertices(innertr,boundarye,sm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% return array of global indices of the vertices inside the
% domain covered by the triangles (of the mesh sm) listed in
% innertr
%
% innertr : triangle indices
% boundarye: indices of boundaryedges of the domain covered by the
%          triangles
% sm :     2D structured mesh
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  all_vertices=[];
  for i=1:length(innertr)
    all_vertices = [all_vertices,sm.trv(innertr(i),:)];
  end
  all_vertices = unique(all_vertices);
  boundary_vt=[];
  for i=1:length(boundarye)
    boundary_vt=[boundary_vt,sm.ep(boundarye(i),:)];
  end
  boundary_vt = unique(boundary_vt);
  idx = find(~ismember(all_vertices,boundary_vt));
  int_vertices = all_vertices(idx);
return
```

A.1.16 part_liftv2e.m

```
function [T]=part_liftv2e(mesh)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transfer matrix node-dofs-> edge-dofs, used to lift a
% vectorfield
% on a partial mesh to
% the potential space (and back)
%
% mesh: 2D structured mesh with partial mesh information
% T: dof(edges) x dof(vertices)
%
% Gisela Widmer 21.1.04
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  T = sparse(mesh.Ne,mesh.Nv);

  for i=1:mesh.Ne
    T(i,mesh.ep(i,1))=-1;
    T(i,mesh.ep(i,2))=1;
  end

  T=T(mesh.partine,mesh.part_int_vt);

return
```

A.1.17 submesh.m

```
function[sm]=submesh(sm,level)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% recursive construction of submeshes of the structured mesh
% sm, maximal number of structured submeshes: level-1
%
% level: Integer # structured meshes
% sm:    structured 2D mesh from which submeshes are to be
%        constructed
%
% the constructed submesh is stored in sm.subm
%
% Gisela Widmer 21.1.04
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% number of nodes in x and y direction of the actual mesh
  Nx = sm.Nx;
  Ny = sm.Ny;

% structured mesh covering the same region am sm, but with doubled
% element edge lenth
  subm = structGrid(sm.orig(1),sm.orig(1)+sm.dx,sm.orig(2),...
                    sm.orig(2)+sm.dy,(Nx-1)/2 +1, (Ny-1)/2+1);

% triangle belonging to the partial submesh
  innertriangles=[];
  for i=1: (Nx-1)/2
    for j=1: (Ny-1)/2

% indices of the 8 triangles of sm in the square (i,j)
% of the submesh
      idx_1=(j-1)*(Nx-1)*4 + 4*(i-1)+1;
      idx_2=idx_1+1;
      idx_3=idx_1+2;
      idx_4=idx_1+3;
      idx_5=(j-1)*(Nx-1)*4+(Nx-1)*2 +4*(i-1)+1;
      idx_6=idx_5+1;
      idx_7=idx_5+2;
      idx_8=idx_5+3;

% triangles of the submesh must totally be covered by partsm-triangles
      if (all(ismember([idx_1,idx_5,idx_6,idx_7],sm.partr)))
         innertriangles=[innertriangles,(j-1)*(Nx-1)+2*(i-1)+1];
      end

      if (all(ismember([idx_2,idx_3,idx_4,idx_8],sm.partr)))
         innertriangles=[innertriangles,(j-1)*(Nx-1)+2*(i-1)+2];
      end
    end
  end

% triangles of the partial submesh
  subm.partr=sort(innertriangles);

% internal and boundary edges of the submesh
```

```
  [subm.partine,subm.partbe]=inneredges(innertriangles,subm);

% internal (not boundary) nodes
  subm.part_int_vt = get_int_vertices(innertriangles,subm.partbe,subm);

% all nodes
  subm.part_vt = unique(subm.trv(innertriangles,:));

% transfer matrix node-dofs-> edge-dofs, used to lift a vectorfield to
% the potential space (and back)
  subm.T = part_liftv2e(subm);

% create a submesh if requested and if the current submesh has
% internal edges, respectly more than 2 nodes in x and y direction
  if(level>1 && length(subm.partine)>0)
%recursive submesh construction
     subm=submesh(subm,level -1);
     subm.I=submesh_transfer(sm,subm)
     sm.subm=subm;
        sm.level = subm.level +1;
        return
  else
     sm.level=1
     return
  end
return
```

A.1.18 submesh_transfer.m

```
function[I]=submesh_transfer(sm,subm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transfer matrix from the auxiliary mesh 'sm' to the coarser
% submesh with doubled triangle edge length 'subm'
%
% The linear operator I: v_sm=I(v_subm) corresponds to evaluating
% edge integrals of the vector field described by v_sub on the
% coarser mesh along the edges of the finer mesh
% (v_subm: vector of edge degrees of freedom on the unstructured
%          mesh,
```

```
%  v_sm:   "                                       part of the
%        structured mesh that approximates the unstructured mesh
%
% sm: 2D mesh structure, contains information about the partial
%     mesh that approximates the unstructured mesh and a nested
%     submesh structure
% subm: submesh of sm
%
% I : dof(partialsm) x dof(partialsubm) transfer matrix
%
% Gisela Widmer  21.1.04
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  I=sparse(sm.Ne,subm.Ne);

%Local interpolation matrices, when the triangle on the submesh
%is an upper triangle or a lower triangle
  up_matrix=[0.25 -0.25 0.25;
             0 0.5 0;
             0 0 0.5;
             0.5 0 0;
             0 0.5,0;
             0.25,0.25,0.25;
             0.5 0 0;
             -0.25 0.25 0.25;
             0 0 0.5];

  low_matrix=[0.25 0.25 -0.25;
              0 0.5 0;
              0 0 0.5;
              0.5 0 0;
              0.25 0.25 0.25;
              0 0 0.5;
              0.5 0 0;
              0 0.5 0;
              -0.25 0.25 0.25];

  for i=1: length(subm.partr)

% index of the actual submesh triangle
    tr_idx=subm.partr(i);
```

```
%upper triangle
    if (mod(tr_idx,2)==1)

%(i,j) index of the square on the submesh, the triangle belongs
%to
       i= (mod(tr_idx,2*(subm.Nx-1))+1)/2;
       j= (tr_idx-2*i+1) /(2*(subm.Nx-1))+1;

% indices of the 3 triangle of the coarser mesh that are
% enclosed by the submesh triangle and have one corner in common
% with it
       tr_1_idx = (j-1)*(sm.Nx-1)*4 + 4*(i-1)+1;
       tr_2_idx =(j-1)*(sm.Nx-1)*4+(sm.Nx-1)*2 +4*(i-1)+1;
       tr_3_idx = tr_2_idx + 2;

%global edge indices on the coarser mesh
       sm_eidx =[sm.tre(tr_1_idx,:),sm.tre(tr_2_idx,:),...
                sm.tre(tr_3_idx,:)];

% assembly of the interpolation matrix
       I(sm_eidx,subm.tre(tr_idx,:))=up_matrix;

% lower triangle
     else

%(i,j) index of the square on the submesh, the triangle belongs
           %to
        i= mod((tr_idx/2)-1,subm.Nx-1)+1;
        j= (tr_idx-2*i) /(2*(subm.Nx-1))+1;

% indices of the 3 triangle of the coarser mesh that are
% enclosed by the submesh triangle and have one corner in common
           % with it
        tr_1_idx = (j-1)*(sm.Nx-1)*4 + 4*(i-1)+2;
        tr_2_idx = tr_1_idx+2;
        tr_3_idx =(j-1)*(sm.Nx-1)*4+(sm.Nx-1)*2 +4*(i-1)+4;

%global edge indices on the coarser mesh
        sm_eidx =[sm.tre(tr_1_idx,:),sm.tre(tr_2_idx,:),...
                  sm.tre(tr_3_idx,:)];
```

```
% assembly of the interpolation matrix
        I(sm_eidx,subm.tre(tr_idx,:))=low_matrix;
     end
  end

%neglect inactive edges
  I=I(sm.partine,subm.partine);
return
```

A.1.19 transfers2us.m

```
function[I]= transfers2us(sg,usg)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Transfer matrix from the auxiliary mesh 'sg' to the
%  unstructured mesh 'usg'.
% The linear operator I: v_usg=I(v_sg) corresponds to evaluating
% edge integrals of the vector field described by v_sg on the
% structured mesh along the edges of the unstructured mesh
% (v_usg: vector of edge degrees of freedom on the unstructured
%         mesh,
%  v_sg:   "                                       part of the
%         structured mesh that approximates the unstructured
%          mesh
%
% sg: 2D mesh structure, contains information about the partial
%     mesh that approximates the unstructured mesh and a nested
%     submesh structure
% usg: unstructured 2D mesh
%
% I : dof(usg) x dof(partialsm) transfer matrix
%
% Gisela Widmer  21.1.04
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% x and y values of the rectangular mesh
  x= [0:sg.Nx-1]*sg.dx/(sg.Nx-1)+sg.orig(1);
```

```
  y= [0:sg.Ny-1]*sg.dy/(sg.Ny-1)+sg.orig(2);

% initialization of the transfer matrix
  I = sparse(usg.Ne,sg.Ne);

% go through all edges of the unstructured mesh
  for e_nr=1:usg.Ne
     idx = [];
     a   = [];
     xm  = [];

% p1: end point of the oriented edge
% p2: tip of the oriented edge
     p1  = usg.vt(usg.ep(e_nr,1) ,:)';
     p2  = usg.vt(usg.ep(e_nr,2) ,:)';

% parameters of p1, p2 and all intersection point of the
% edge with the structured mesh (p1 + t(p2-p1))
     t  = intersection(sg,p1,p2);

% relative length of the sections
     dt = diff(t);
% parameter of the section midpoints
     tm= (t([1:length(t)-1])+t([2:length(t)]))/2;

% go through all sections
     for i=1:length(tm)

% midpoint
        xm(:,i)= p1+ tm(i).*(p2-p1) ;

% x- index of the lower left edge on the
% structured mesh with respect to xm
        idx(1,i) = max(find(x<=xm(1,i)&...
                     x>=xm(1,i) - sg.dx/(sg.Nx-1)));
% y index of the lower left edge
        idx(2,i) = max(find(y<=xm(2,i)&...
                  y>xm(2,i)-sg.dy/(sg.Ny-1)));

      end
% triangles where the section go through
```

```
      T=(idx(2,:)-1)*(sg.Nx-1)*2+2*idx(1,:)-1;

% vector (x,y)of the structured mesh edgt-> xm
      a=xm-[x(idx(1,:));y(idx(2,:))];
      v=[sg.dy,-sg.dx]/norm([sg.dx,sg.dy]);

% find indices where a x (dx,dy) > 0 and increase
% the indices of these triangles
      [dummy,id] = find(v*a > eps);
      T(id)=T(id)+1;

% compute the contribution of each edge value on the structured
% mesh to the path integrals on the unstructured mesh
      for i=1:length(T)
         xm(:,i);
         sg.trv(T(i),:);
         phi=localbf(xm(:,i),sg.vt(sg.trv(T(i),1) ,:)',...
         sg.vt(sg.trv(T(i),2) ,:)',...
         sg.vt(sg.trv(T(i),3) ,:)');
         sedofs=sg.tre(T(i),:);
         I(e_nr,sedofs)= I(e_nr,sedofs)+ (phi'*(p2-p1)*dt(i))'...
                         .*sg.treo(T(i),:);
      end

  end

% extract inner edges of the unstructured mesh and the edges of
% the submesh that belong to the partial mesh


  I=I(Eactidx(usg),sg.partine);
return
```

A.1.20 localbf.m

```
function[phi]=localbf(x,a,b,c)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% given 3 counterclockwise corners (2x1 vectors) a b c of a
% triangle and a point x within this triangle,
```

```
% the three basis functions of this triangle
% are computed a the position x
% (phi_i = lambda_i+1 * grad lambda_i+2 -lambda_i+2 *
%          grad lambda_i+1)
%
% a,b,c: 2x1 vectors (corners- counterclockwise)
% x    : 2x1 vector position where the basis functions
%        shall be evaluated
% phi     : 2x3 matrix [phi_a(x);phi_b(x);phi_c(x)]
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  l=lambda(x,a,b,c);
  gr=gradlambda(a,b,c);
  phi(:,1)=l(2)*gr(:,3)-l(3)*gr(:,2);
  phi(:,2)=l(3)*gr(:,1)-l(1)*gr(:,3);
  phi(:,3)=l(1)*gr(:,2)-l(2)*gr(:,1);
return
```

A.1.21 lambda.m

```
function[l]=lambda(x,a,b,c)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% given 3 corners (2x1 vectors) a b c of a triangle and
% a point x within this triangle,
% the value of the hut functions to each corner
% is computed a the position x
%
% a,b,c: 2x1 vectors (corners- counterclockwise)
% x    : 2x1 vector position where the hut functions
%        shall be evaluated
% l      : 3x1 vector [lambda_a(x);lambda_b(x);lambda_c(x)]
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  d = (b(1)-a(1))*(c(2)-a(2))-(c(1)-a(1))*(b(2)-a(2));
```

```
  area = abs(d)/2;
  l=1/(2*area)*[(x-b)'*[b(2)-c(2);c(1)-b(1)];
              (x-c)'*[c(2)-a(2);a(1)-c(1)];...
              (x-a)'*[a(2)-b(2);b(1)-a(1)]];
return
```

A.1.22 gradlambda.m

```
function[gl]=gradlambda(a,b,c)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% given 3 corners (2x1 vectors) a b c of a triangle,
% the gradient of the hut functions to each corner
% is computed
%
% a,b,c: 2x1 vectors
% gl: 2x3 matrix with gradients as columns
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  d = (b(1)-a(1))*(c(2)-a(2))-(c(1)-a(1))*(b(2)-a(2));
  area = abs(d)/2;
  gl=1/(2*area)*[[b(2)-c(2);c(1)-b(1)],...
               [c(2)-a(2);a(1)-c(1),],...
               [a(2)-b(2);b(1)-a(1)]];
return
```

A.1.23 intersection.m

```
function [ts, e]=intersection(structmesh,p1,p2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% given a structured 2D mesh and two points p1 and p2,
% the line between these two points is parameterized:
% p1 + t* (p2 -p1) and the parameters ts where this line
% intersects with edges of the structured mesh are
% computed and 0 and 1 are added to this parameter list.
%
% structmesh: structured 2D mesh
```

```
% p1,p2 : 2x1 vectors
%
% ts : 1 x N vector without duplicates (0 and 1 included)
% e : edgenumbers that belong to the intersection points in
%     ]0,1[
%
% Gisela Widmer 29/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% initialisation
  ts=[];
  t=[];
  idx=[];
  e=[];
% x and y values of the rectangular mesh
  x= [0:structmesh.Nx-1]*structmesh.dx/(structmesh.Nx-1)+...
     structmesh.orig(1);
  y= [0:structmesh.Ny-1]*structmesh.dy/(structmesh.Ny-1)+...
     structmesh.orig(2);

% intersections with vertical axis
  if p1(1)>= (p2(1))
     idx=find(x>p2(1)&x<p1(1));

  elseif p1(1)< (p2(1))
     idx=find(x<p2(1)&x>p1(1));

  end

  if(length(idx)>0)

% intersection parameters,
    t=(x(idx)-p1(1))/(p2(1)-p1(1));

% corresponding y values
    y_=p1(2)+t*(p2(2)-p1(2));

% add to intersection parameter list
    ts=[ts,t];

% determine numbers of intersectiong edges of the structured mesh
    for yl=1:length(y_)
```

```
      idy1= find(y<=y_(yl)&y>y_(yl)-structmesh.dy/(structmesh.Ny-1));
      idy2= find(y>=y_(yl)&y<y_(yl)+structmesh.dy/(structmesh.Ny-1));
      n1= (idy1-1)*structmesh.Nx + idx(yl);
      n2= (idy2-1)*structmesh.Nx + idx(yl);
      e1=[find(structmesh.ep(:,1)==n1);find(structmesh.ep(:,2)==n1)];
      e2=[find(structmesh.ep(:,1)==n2);find(structmesh.ep(:,2)==n2)];
      e = [e e1(ismember(e1,e2))];
    end
  end

% intersections with horizontal axis (like above)
  if p1(2)>= (p2(2))
     idy=find(y>p2(2)&y<p1(2));
  elseif p1(2)< (p2(2))
     idy=find(y<p2(2)&y>p1(2));
  end

  if(abs(p2(2)-p1(2))>eps)
    t=[(y(idy)-p1(2))/(p2(2)-p1(2))];
    x_=p1(1)+t*(p2(1)-p1(1));
    ts=[ts,t];
    for xl=1:length(x_);
      idx1= find(x<=x_(xl)&x>x_(xl)-structmesh.dx/(structmesh.Nx-1));
      idx2= find(y>=x_(xl)&x<x_(xl)+structmesh.dx/(structmesh.Nx-1));
      n1= (idy(xl)-1)*structmesh.Nx + idx1;
      n2= (idy(xl)-1)*structmesh.Nx + idx2;
      e1=[find(structmesh.ep(:,1)==n1);find(structmesh.ep(:,2)==n1)];
      e2=[find(structmesh.ep(:,1)==n2);find(structmesh.ep(:,2)==n2)];
      e = [e e1(ismember(e1,e2))];
    end
  end

% intersection with diagonals
  if( (abs(p2(1)-p1(1))<=eps)|...
      abs((p2(2)-p1(2))/(p2(1)-p1(1))-structmesh.dy/structmesh.dx)>eps)

% select diagonals with which intersection is possible (to achieve
% linear scaling)
    xedgeidx =find(x<=p1(1)&...
               x>p1(1)-structmesh.dx/(structmesh.Nx-1));
    yedgeidx =find(y<=p1(2)&...
```

```
              y>p1(2)-structmesh.dy/(structmesh.Ny-1));
    xedgeidx =sort([xedgeidx,find(x<=p2(1)&...
              x>p2(1)-structmesh.dx/(structmesh.Nx-1))]);
    yedgeidx =sort([yedgeidx,find(y<=p2(2)&...
              y>p2(2)-structmesh.dy/(structmesh.Ny-1))]);

    if(length(yedgeidx)>1&length(xedgeidx)>1)
% lower left vertices
      leftvert=[ones(1,yedgeidx(2)-yedgeidx(1)+1)*x(xedgeidx(1));...
                y([yedgeidx(1):yedgeidx(2)])]';
      leftvert=[leftvert;[x([(xedgeidx(1)+1):xedgeidx(2)]);...
                ones(1,xedgeidx(2)-xedgeidx(1))*y(yedgeidx(1))]'];
% upper right vertices
      rightvert=leftvert + [ones(size(leftvert,1),1)*structmesh.dx, ...
                ones(size(leftvert,1),1)*structmesh.dy] ;

      dx_mesh = rightvert(:,1)-leftvert(:,1);
      dy_mesh = rightvert(:,2)-leftvert(:,2);
      dx_edge = p2(1)-p1(1);
      dy_edge = p2(2)-p1(2);
      t=[];

      for j=1:size(dx_mesh)
        if(abs(dx_edge*dy_mesh(j)-dx_mesh(j)*dy_edge)>eps)
% intersection parameters with the diagonals that possibly intersect
% with [p1 p2]
          t=[t,(dy_mesh(j)*(leftvert(j,1)-p1(1))+dx_mesh(j)*...
            (p1(2)-leftvert(j,2)))/(dx_edge*dy_mesh(j)-...
             dx_mesh(j)*dy_edge)];
        end
      end
% select those intersection on the edge
      t=t(find(t>0&t<1));
      x_=p1(1)+t*(p2(1)-p1(1));
      y_=p1(2)+t*(p2(2)-p1(2));

%determine corresponding edge numbers
      for nr=1:length(x_)

        idy1= find(y<=y_(nr)&y>y_(nr)-structmesh.dy/(structmesh.Ny-1));
        idy2= find(y<=y_(nr)+structmesh.dy/(structmesh.Ny-1)&y>y_(nr));
```

```
        idx1= find(x<=x_(nr)&x>x_(nr)-structmesh.dx/(structmesh.Nx-1));
        idx2= find(x<=x_(nr)+structmesh.dx/(structmesh.Nx-1)&x>x_(nr));
        n1=(idy1-1)*structmesh.Nx + idx1;
        n2=(idy2-1)*structmesh.Nx + idx2;

        e1=[find(structmesh.ep(:,1)==n1);find(structmesh.ep(:,2)==n1)];
        e2=[find(structmesh.ep(:,1)==n2);find(structmesh.ep(:,2)==n2)];
        e=[e e1(ismember(e1,e2))];

      end
      ts=[ts,t];
    end
  end
  [ts tidx] = unique(ts);
  e=e(tidx);
% eliminate t values outside (0,1)

   tidx=find(ts>0 & ts<1);
   ts=ts(tidx);
   e=e(tidx);

% add 0 and 1
   ts=[0,ts,1];

return
```

A.2 Setup of Stiffness Matrix and Boundary Layer

A.2.1 mesh_setup.m

```
function[usm]=mesh_setup(meshfilename,stiffness_flag,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  reads nested mesh structure from 'meshfilename' (mat-file)
%  and creates a stiffness matrix  and the boundary layer
%  for each level
%
%  meshfilename      : string containing name of mat-file where
%                      the complete nested mesh structure is
```

```
%                      stored
%  stiffness_flag = 0: galerkin construction
%                 = 1: direct computation
%  params            : string specifying parameterfile
%
%
%
%  Gisela Widmer  19/2/04
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  global tau;
  eval(params),

% load nested mesh structure -> stored in usm
  try
      eval(['load ',meshfilename,' -mat']);
  catch
     try
        eval(['load ',meshfilename,'.mat -mat']);
     catch
        disp(['error: file',meshfilename,' not found']);
        return
     end
  end

% determine edges and vertices of the boundary layer (for
% boundary smoothing
   e_idx=Internal_bound_ed(usm,degree);
   act_e_idx=Eactidx(usm);
   [dummy,usm.e_idx]=ismember(e_idx,act_e_idx);

   v_idx=Internal_bound_vt(usm,degree);
   act_v_idx=Vactidx(usm);
   [dummy,usm.v_idx]=ismember(v_idx,act_v_idx);

% stiffness matrix on the unstructured mesh
   [C,M]=EdgeCurlMass(usm,1);
   usm.A=C + tau*M;

   usm.Pot_Mat=usm.T'*usm.A*usm.T;
```

```
% stiffness matrix on the auxiliary meshes by galerkin
% construction or direct computation
   switch stiffness_flag
     case 0

         usm.sm.A=usm.sm.I'*usm.A*usm.sm.I;
         usm.sm.Pot_Mat=usm.sm.T'*usm.sm.A*usm.sm.T;
         usm.sm=sm_Galerkin_Stiffness_matrix(usm.sm,params);

      case 1

         usm.sm=sm_direct_Stiffness_matrix(usm.sm,tau,params);

      otherwise
         disp('error: no stiffness matrix computed!');
   end
return
```

A.2.2 Internal_bound_ed.m

```
function[int_boundary_idx]=Internal_bound_ed(usm,degree)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Determines edge indices of internal edges in the boundary
% region for boundary smoothing
%
% usm: 2D unstructured mesh
% degree: width of boundary layer
%         0: no edges
%         1: edges with one end point on the boundary
%         2: edges of triangles with at least one edge
%            in (1)
%         3: edges with at least one end point in (2) and so on
%
%int_boundary_idx: global edge indices of edges in the boundary
%                 layer
```

```
%
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  tridx=[];
  innernodes = find(usm.vbfl == 0)';
  boundarye=find(usm.ebfl==1);
  innere=find(usm.ebfl==0);

  for i=1:usm.Nt
     if sum(ismember(usm.tre(i,:),boundarye))>0
        tridx=[tridx,i];
     end
  end
  eidx=unique(usm.tre(tridx,:));
  int_boundary_idx=intersect(eidx,innere);

  for deg_counter=1:degree-1
    for i=1:usm.Nt
      if sum(ismember(usm.tre(i,:),int_boundary_idx))>0
        tridx=[tridx,i];
      end
    end
    eidx=unique(usm.tre(tridx,:));
    int_boundary_idx=intersect(eidx,innere);
  end
return
```

A.2.3 Internal_bound_vt.m

```
function[idx]=Internal_bound_vt(usm,degree)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Internal vertices in the boundary layer of
% Internal_bound_ed(usm,degree)
%
% idx: global edge indices of vertices in this boundary layer
%
```

```
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  eidx=Internal_bound_ed(usm,degree);
  vidx=unique(usm.ep(eidx,:));
  idx=intersect(vidx,Vactidx(usm));
return
```

A.2.4 EdgeCurlMass.m

```
function [C,M] = EdgeCurlMass(mesh,withbd,partflag)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Computes "curlcurl matrix" C and "mass matrix" M
% for edge elements
%
% mesh -> data structure for 2D triangulation
% withbd -> If true drop d.o.f. on boundary, default = TRUE
% partflag: 0: the mesh is a 'normal' unstructured mesh and
%              the matrices are computed for all triangles
%           1: the mesh is a partial mesh and the matrices are
%              computed only for the internal triangles
%              specified in mesh.partr
%
%  Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  if (nargin < 2), withbd = 1; end;
  if(nargin<3),partflag=0; end;

  if partflag==0
    N = mesh.Ne;
  else
    edges=unique([mesh.partine,mesh.partbe]);
    N=length(edges);
  end
  Cp = sparse(N,N);
  Mp = sparse(N,N);

% Assembly of local matrices
```

```
  if partflag==0
    ntr=mesh.Nt;
  else
    ntr=length(mesh.partr);
  end
  for k=1:ntr
    if partflag==0
        i=k;
    else
        i=mesh.partr(k);
    end
    vidx = mesh.trv(i,:);
    x = mesh.vt(vidx,1);
    y = mesh.vt(vidx,2);
    if partflag==0
      dofs = mesh.tre(i,:);
    else
      [dummy,dofs]=ismember(mesh.tre(i,:),edges);
      %dofs = ismember(edges,mesh.tre(i,:));
    end

% Compute the determinant and the area.
    d = (x(2)-x(1))*(y(3)-y(1))-(x(3)-x(1))*(y(2)-y(1));
    area = abs(d)/2;

% Gradients of barycentric coordinate functions
    G = 1/(2*area)*[y(2)-y(3) , y(3)-y(1) , y(1)-y(2);...
                  x(3)-x(2) , x(1)-x(3) , x(2)-x(1)];

% Local mass matrix
    P = 0.5*[ G(:,3)-G(:,2) ,   G(:,1)       ,  -G(:,1)      ; ...
            -G(:,2)        , G(:,1)-G(:,3) ,  G(:,2)       ;
            G(:,3)         ,    -G(:,3)    , G(:,2)-G(:,1) ];
    MT = area/3*P'*P;

% Local curl-curl matrix
    Q = [1,1,1]/area;
    CT = area*Q'*Q;

% Get permutation matrix reflecting the mismatch
% between global and local edge orientations
```

```
    Peori = diag(mesh.treo(i,:));

    Cp(dofs,dofs) = Cp(dofs,dofs) + Peori*CT*Peori;
    Mp(dofs,dofs) = Mp(dofs,dofs) + Peori*MT*Peori;
  end

% Discard rows and columns corresponding to edges on the boundary
  N_vact = mesh.Nv;
  if (withbd == 1)
    if partflag==0
        e_act = Eactidx(mesh);
    else
        [dummy,e_act]=ismember(mesh.partine,edges);
    end
    Cp = Cp(e_act,e_act);
    Mp = Mp(e_act,e_act);
  end;

  C = Cp;
  M = Mp;
return
```

A.2.5 sm_Galerkin_Stiffness_matrix.m

```
function[sm]=sm_Galerkin_Stiffness_matrix(sm,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Recursive Construction of the stiffness matrix (Galerkin
% construction) and
% hybrid smoother matrix in the potential space (Pot_Mat)
% for all  submeshes of structured 2D mesh sm and setup of the
% boundary layer
%
% sm : 2D structured mesh
% params: name of parameter file (string)
%
% Gisela Widmer 29/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  sm.subm.A=sm.subm.I'*sm.A*sm.subm.I;
```

```
  sm.subm.Pot_Mat=sm.subm.T'*sm.subm.A*sm.subm.T;
  eval(params);
  sm.e_idx = part_Internal_bound_ed(sm,degree);
  sm.v_idx = part_Internal_bound_vt(sm,degree);
  if sm.level>2
    sm.subm=sm_Galerkin_Stiffness_matrix(sm.subm,params);
  end
return
```

A.2.6 sm_direct_Stiffness_matrix.m

```
function[sm]=sm_direct_Stiffness_matrix(sm,tau,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Recursive construction of the stiffness matrix with
% direct compution and
% hybrid smoother matrix in the potential space (Pot_Mat)
% for all  submeshes of structured 2D mesh sm and setup of the
% boundary layer
%
% sm : 2D structured mesh
% params: name of parameter file (string)
%
% Gisela Widmer 29/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  [C,M]=EdgeCurlMass(sm,1,1);
  sm.A=(C + tau*M);
  eval(params);
  sm.e_idx = part_Internal_bound_ed(sm,degree);
  sm.v_idx = part_Internal_bound_vt(sm,degree);
  sm.Pot_Mat=sm.T'*sm.A*sm.T;

  if (sm.level>1)
    sm.subm=sm_direct_Stiffness_matrix(sm.subm,tau,params);
  end
return
```

A.2.7 part_Internal_bound_ed.m

```
function[e_idx]=part_Internal_bound_ed(sm,degree)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Determines edge indices of internal edges in the boundary
% region for boundary smoothing
%
% usm: 2D unstructured mesh
% degree: width of boundary layer
%         0: no edges
%         1: edges with one end point on the boundary
%         2: edges of triangles with at least one edge
%            in (1)
%         3: edges with at least one end point in (2) and so on
%
%int_boundary_idx: edge indices relative to the internal edge
%                 indices of edges in the boundary layer
%
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  tridx=[];

  boundarye=sm.partbe;
  innere=unique([sm.partine,sm.partcrosse]);

  for i=1:sm.Nt
    if sum(ismember(sm.tre(i,:),boundarye))>0
        tridx=[tridx,i];
    end
  end

  eidx=unique(sm.tre(tridx,:));
  int_boundary_idx=intersect(eidx,innere);
  for deg_counter=1:degree-1
    for i=1:sm.Nt
      if sum(ismember(sm.tre(i,:),int_boundary_idx))>0
        tridx=[tridx,i];
      end
```

```
    end
    eidx=unique(sm.tre(tridx,:));
    int_boundary_idx=intersect(eidx,innere);
  end
  [dummy,e_idx]=ismember(int_boundary_idx ,...
               unique( [sm.partine,sm.partcrosse]));
return
```

A.2.8 part_Internal_bound_vt.m

```
function[v_idx]=part_Internal_bound_vt(sm,degree)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Internal vertices (relative to internal vertices) in the
%  boundary layerof part_Internal_bound_ed(usm,degree)
%
% idx: global edge indices of vertices in this boundary layer
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  eidx=part_Internal_bound_ed(sm,degree);
  all_edges=sm.partine;
  ine=all_edges(eidx) ;
  vidx=unique(sm.ep(ine,:));
  idx=intersect(vidx,sm.part_int_vt);

  [dummy,v_idx]=ismember(idx,sm.part_int_vt);
return
```

A.2.9 Erhs.m

```
function f = Erhs(mesh,f_field,Q,withbd)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute global load vector for 2D edge elements
%
% mesh -> data structure for 2D triangulation
% Q    -> M x 3 -matrix providing quadrature points and weights
% withbd -> If true drop components corresponding to d.o.f. on
```

```
%            boundary
% f_field -> M-file providing the vector field
%            Must permit the evaluation f_field(x),
%            where size(x) = [2 1]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  fnstr = [f_field,'(p)'];
  if (nargin < 4), withbd  = 1; end;
  N = mesh.Ne;
  fm = zeros(N,1);

% Preprocessing for quadrature
  M = size(Q,1);
  pimat = [Q(:,1) , Q(:,2) , 1-Q(:,1)-Q(:,2)];
  wvec = Q(:,3);

% Assembly of local element vectors

  for i=1:mesh.Nt
    vidx = mesh.trv(i,:);
    x = mesh.vt(vidx,1);
    y = mesh.vt(vidx,2);
    dofs = mesh.tre(i,:);

% Compute the determinant and the area.
    d = (x(2)-x(1))*(y(3)-y(1))-(x(3)-x(1))*(y(2)-y(1));
    area = abs(d)/2;

% Gradients of barycentric coordinate functions
    G = 1/(2*area)*[y(2)-y(3) , y(3)-y(1) , y(1)-y(2);...
                    x(3)-x(2) , x(1)-x(3) , x(2)-x(1)];

% Compute weighted sum of field values
    g1 = [0;0]; g2 = [0;0]; g3 = [0;0];
    for j=1:M
      p = [pimat(j,:)*x;pimat(j,:)*y];

      fval = wvec(j)*eval(fnstr);
      g1 = g1 + pimat(j,1)*fval;
      g2 = g2 + pimat(j,2)*fval;
      g3 = g3 + pimat(j,3)*fval;
```

```
    end
    g1 = 2*area*g1; g2 = 2*area*g2; g3 = 2*area*g3;

    tloc = [g2'*G(:,3) - g3'*G(:,2);...
            g3'*G(:,1) - g1'*G(:,3);...
            g1'*G(:,2) - g2'*G(:,1)];

% Permute, if local orientations do not reflect global
% orientations
    Peori = diag(mesh.treo(i,:));

    fm(dofs) = fm(dofs) + Peori*tloc;
  end

% Discard components corresponding to edges on the boundasry

  if (withbd == 1)
    act_idx = Eactidx(mesh);
    f = fm(act_idx);
  else
    f = fm;
  end;
return
```

A.3 Precondition Operator

A.3.1 Precond_Op_B.m

```
 function [Bv]= Precond_Op_B(v,usm,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Precondition Operator B that approximates A^(-1)
% <-> BAx ~ x
% if Ax ~ v -> Bv ~ x
%
% v: Nx1 vector to compute Bv
% usm: unstructured 2D mesh
% params: string, name of m-file containig
```

```
%               general parameters
%
% Gisela Widmer 13/1/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 Bv=zeros(length(v),1);

% read parameters
  eval(params);

%presmoothing
  Bv = boundary_hybrid_smoother(usm,v,params,Bv,1);
  Bv = hybrid_smoother(usm,v,params,Bv,2);
  Bv = boundary_hybrid_smoother(usm,v,params,Bv,2);


%Coarse grid correction of the residual
  r = v-usm.A*Bv;
  rs = usm.sm.I'*r;

  % solve the linear system on the structured mesh


  if PrecondSolver == 2
    xs = zeros(length(rs),1);
    es = MultiGrid_solve(rs,xs,params,usm.sm);
  elseif PrecondSolver ==1
    es = usm.sm.A\rs;
  else
    disp([' No valid Precondition Solver in ',params]);
    return
  end
  e_s = usm.sm.I*es;


%add correction
  Bv = Bv+e_s;
% smooth the solution on the unstructured
% mesh by a hybrid smoother
  Bv = boundary_hybrid_smoother(usm,v,params,Bv,1);
  Bv = hybrid_smoother(usm,v,params,Bv,1);
  Bv = boundary_hybrid_smoother(usm,v,params,Bv,1);
```

```
return
```

A.3.2 boundary_hybrid_smoother.m

```
function x = boundary_hybrid_smoother(mesh,b,params,x,dir)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% smoothes the approximate solution x of the linear
% system A*x=b  in 3 steps:
% 1) bound_smooth_it1 GaussSeidel interations with matrix A
%    and rhs b
% 2) after lifting the residual from the edged to the
%    vertices:
%    bound_smooth_it2 GaussSeidel interations with the matrix
%    T'*A*T (T ist the matrix that transfers values of
%    nodeelements of grad S1 to values of edgeelements),
%    rhs = residual and initial guess = 0
% 3) after backlifting the solution from the vertices to
%    the edges:
%    bound_smooth_it3 GaussSeidel interations like 1)
%
% mesh: (un)structured 2D mesh
% b : Nx1 vector
% params: string, contains name of parameter file
% x : Nx1 vector (optional)
% dir: direction of the Gauss-Seidel method:
%      1: Forward
%      ~=1 : Backward
% x (output): Nx1 vector (solution)
%
% Gisela Widmer 14/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  eval(params);

% bound_smooth_it1 GaussSeidel iterations on the (inner) edges
  e_bound_loops=floor(size(mesh.A,2)/length(mesh.e_idx));
  if dir == 1
    x = Mat_BoundaryGaussSeidel(mesh.A,b,x,e_bound_loops*...
        bound_smooth_it1,mesh.e_idx,dir);
```

```
  else
    x = Mat_BoundaryGaussSeidel(mesh.A,b,x,e_bound_loops*...
        bound_smooth_it3,mesh.e_idx,dir);
  end

% compute the residual on the edges
  e_res = b - mesh.A*x;


% lift the residual to the inner vertices
  n_res = mesh.T'*e_res;

% use 0 a initial guess for error
  e0= zeros(length(n_res),1);

% bound_smooth_it2 GaussSeidel interations on the (inner) vertices
  n_bound_loops=floor(size(mesh.Pot_Mat,2)/length(mesh.v_idx));
  n_r = Mat_BoundaryGaussSeidel(mesh.Pot_Mat,n_res,e0,...
        bound_smooth_it2,mesh.v_idx,dir);

% lift the result back to the (inner) edges
  e_r = mesh.T * n_r;

% add the correction on the edges from the solution x
  x = x + e_r;

% bound_smooth_it3 Gauss- Seidel iterations on the edges
  if dir==1
    x = Mat_BoundaryGaussSeidel(mesh.A,b,x,e_bound_loops*...
        bound_smooth_it3,mesh.e_idx,dir);
  else
    x = Mat_BoundaryGaussSeidel(mesh.A,b,x,e_bound_loops*...
        bound_smooth_it1,mesh.e_idx,dir);
  end
return
```

A.3.3 Mat_BoundaryGaussSeidel.m

```
function[v]=Mat_BoundaryGaussSeidel(A,b,v0,m,idx,dir)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Performs m Gauss- Seidel iterations for the components of v0
% with index 'idx to smooth the solution there
% the system Av=b with initial guess v0.
%
% A : nxn Matrix
% b : nx1 column vector (rhs)
% v0: nx1 coloum vector (initial guess of the solution)
% m : number of iteration cycles
% dir: direction of the Gauss-Seidel method:
%      1: Forward
%      ~=1: Backward
% v:  solution
%    Gisela Widmer 19/2/04
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  v=v0;

  if length(idx)==0
    return
  end

  if dir==1
    M=tril(A);
    U=A-M;
    M1=M(idx,idx);
    idx2=find(~ismember([1:length(b)],idx));
    M2=M(idx,idx2);

    for k=1:m
      v(idx)=M1\(-U(idx,:)*v +b(idx)-M2*v(idx2));
    end

  else
    M=triu(A);
    L=A-M;
    M1=M(idx,idx);
    idx2=find(~ismember([1:length(b)],idx));
    M2=M(idx,idx2);

    for k=1:m
      v(idx)=M1\(-L(idx,:)*v +b(idx)-M2*v(idx2));
```

```
    end
  end
return
```

A.3.4 hybrid_smoother.m

```
function x=hybrid_smoother(mesh,b,params,x,dir)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% smoothes the approximate solution x of the linear
% system A*x=b  in 3 steps:
% 1) smooth_it1 GaussSeidel interations with matrix A
%    and rhs b
% 2) after lifting the residual from the edged to the
%    vertices:
%    smooth_it2 GaussSeidel interations with the matrix
%    T'*A*T (T ist the matrix that transfers values of
%    nodeelements of grad S1 to values of edgeelements),
%    rhs = residual and initial guess = 0
% 3) after backlifting the solution from the vertices to
%    the edges:
%    smooth_it3 GaussSeidel interations like 1)
%
% mesh: (un)structured 2D mesh
% b : Nx1 vector
% params: string, contains name of parameter file
% x : Nx1 vector (optional)
% dir: direction of the Gauss-Seidel method:
%      1: Forward
%      ~=1: Backward
% x (output): Nx1 vector (solution)
%
% Gisela Widmer 13/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%read parameter
  eval(params);

% smooth_it1 GaussSeidel iterations on the (inner) edges
  if dir==1
```

```
    x = MatGaussSeidel(mesh.A,b,x,smooth_it1,dir);
  else
    x = MatGaussSeidel(mesh.A,b,x,smooth_it3,dir);
  end

% compute the residual on the edges
  e_res = b - mesh.A*x;


% lift the residual to the inner vertices
  n_res = mesh.T'*e_res;

% use 0 a initial guess for error
  e0= zeros(length(n_res),1);

% smooth_it2 GaussSeidel interations on the (inner) vertices
  n_r = MatGaussSeidel(mesh.Pot_Mat,n_res,e0,smooth_it2,dir);

% lift the result back to the (inner) edges
  e_r = mesh.T * n_r;

% add the correction on the edges from the solution x
  x = x + e_r;

% smooth_it3 Gauss- Seidel iterations on the edges
  if dir==1
    x = MatGaussSeidel(mesh.A,b,x,smooth_it3,dir);
  else
    x = MatGaussSeidel(mesh.A,b,x,smooth_it1,dir);
  end
return
```

A.3.5 MatGaussSeidel.m

```
function[v]=MatGaussSeidel(A,b,v0,m,dir)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Performs m Gauss- Seidel iterations to solve (or smooth)
% the system Av=b with initial guess v0.
%
```

```
% A : nxn Matrix
% b : nx1 column vector (rhs)
% v0: nx1 coloum vector (initial guess of the solution)
% m : number of iteration cycles
% dir: direction of the Gauss-Seidel method:
%      1: Forward
%      ~=1: Backward
% v:  solution
%
%    Gisela Widmer 19/2/04
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  v=v0;
  if dir==1
    M=tril(A);
    U=A-M;
    for k=1:m
      v=M\(-U*v+b);
    end
  else
    M=triu(A);
    L=A-M;
    for k=1:m
      v=M\(-L*v+b);
    end
  end
return
```

A.3.6 MultiGrid_solve.m

```
function[xs]= MultiGrid_solve(vs,xs,params,sm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Multigrid solver for the system As xs = vs
%
% As : nxn matrix
% vs : nx1 vector rhs
% xs : guess for the solution
% params: string, parameter file
% sm : 2D structured mesh with hierarchical submeshes
%
```

```
% Gisela Widmer 13/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% read parameters
  eval(params)  ;

% if sm is the coarstest grid: solve direct
  if sm.level == 1
       xs = sm.A\vs;
       return
  end
% otherwise smooth the error with multigrid

% MGcycle multigrid sweeps on sm
  for i=1:MGcycle

% one presmoothing sweep on sm
    xs = boundary_hybrid_smoother(sm,vs,params,xs,1);
    xs = hybrid_smoother(sm,vs,params,xs,1);
    xs = boundary_hybrid_smoother(sm,vs,params,xs,1);

%Coarse grid correction of the residual
    rs = vs-sm.A*xs;

    r_sub = sm.subm.I'*rs;
    e_sub = zeros(length(r_sub),1);
    e_sub = MultiGrid_solve(r_sub,e_sub,params,sm.subm);

    e_s=sm.subm.I*e_sub;

%add the correction
    xs = xs + e_s;

% postsmoothing
    xs = boundary_hybrid_smoother(sm,vs,params,xs,2);
    xs = hybrid_smoother(sm,vs,params,xs,2);
    xs = boundary_hybrid_smoother(sm,vs,params,xs,2);
  end
return
```

A.4 Solving the Edge Boundary Problem

A.4.1 solve_EBP.m

```
function [solve_out]=solve_EBP(usm, solver,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% solves the boundary value problem
% curl curl u + tau*u = rhsf in 2D on the unstructured mesh
% specified in "usm"
%
% usm: 2D unstructured triangle mesh
% solver: string with solver name
% params: string, name of m-file containing general parameters
%
% solve_out: struct of information about the solving procedure
% provided by the specific solver
%
% parameters are to be specified in 'read_EPB_param.m'
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% setup of parameters
    try
      [rhsf,quadraturefile]=read_EBP_param;
    catch
      disp(lasterr);
      disp('script read_EBP_param not found');
      return;
    end
    global tau;
    err = [];

% quadrature selection
   Q = dlmread(quadraturefile);

% computation of rhs- vector for the edge elements
    F = Erhs(usm,rhsf,Q,1);

% solve the linear system A*x = F and store the solution with
```

```
% 0 values on the boundary edges in sol_e
    sol_e=zeros(usm.Ne,1);
    switch(solver)
       case('direct')
           sol_e(Eactidx(usm))  = usm.A\F;
       case('CG')
           [sol_e(Eactidx(usm)),err]  = CG(usm.A,F);
       case('PrecCG')
           [solve_out]  = PrecCG(F,usm,params);
       otherwise
           disp('No valid solver chosen, select ''direct'', ...
           ''CG'' or ''PrecCG''');
           return;
    end
return
```

A.4.2 CG.m

```
function[x,err]=CG(A,b,x0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Conjugate gradient algorithm to solve the
% linear system Ax = b with initial guess x0
% for x. The algorithm stops if the number of
% steps exceeds kmax or if the norm of the residual
% is less than tol.
%
% A : NxN matrix
% b : Nx1 vector
% x0: Nx1 vector (optional)
%
% x: Nx1 vector
% err: error history 1x Nsteps vector that contains
%      the 2norm of the residual after each iteration
%      step
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% setup of parameters
  try
```

```
   load cg_param.dat;
  catch
    disp(' cg_param.dat not found ');
  end
  kmax = cg_param(1);
  tol  = cg_param(2);
  if (nargin<3), x0=zeros(length(b),1); end

% Initialisation
  x=x0;
  n = length(x);
  k = 1;

  r1 = A*x - b;
  err = norm(r1);
  if norm(r1) < 1e-12
        return;
  end
  h1 = -r1;
  t = ( r1'*r1 ) / ( h1'*A*h1 );
  x = x + t*h1;

  k = k+1;
  r2 = A*x - b;

  err = [err,norm(r2)];
  if norm(r2) < tol*err(1)
    return;
  end

% CG iterations
  while (norm(r2) > tol*err(1) &k <= kmax)

    a = ( r2'*r2 ) / ( r1'*r1 );
    h2 = -r2 + a*h1;
    t = ( r2'*r2 ) / ( h2'*A*h2 );
    x = x + t*h2;

    r1 = r2;
    r2 = A*x - b;
    h1 = h2;
```

```
    k = k+1;
    err=[err,norm(r2)];
  end
return
```

A.4.3 PrecCG.m

```
function[solve_out]=PrecCG(f,usm,params,u0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preconditioned conjugate gradient algorithm to solve the
% linear system Au=f with initial guess u0 for u.
% The preconditioning operator B is defined in the
% file Precond_Op_B.m
%
% f : Nx1 vector
% usm: unstructured 2D mesh
% params: string, name of the m-file containing
%              smoothing scheme parameters
% u0 : Nx1 vector -initial guess for the solution (optional)
%
% solve_out: struct containing various information about the
%            preconditioned CG procedure (see below)
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% parameter setup
  preccg_param;

  if (nargin<4), u0=zeros(size(usm.A,2),1); end


% preconditioned CG with information of the error in the energy norm
  [solve_out.sol_e,solve_out.flag,solve_out.relres,solve_out.iter,...
      solve_out.resvec,solve_out.Anormerr]= myPrecCG(f,usm,params,u0);

return
```

A.4.4 myPrecCG.m

```
function[x,flag,relres,iter,resvec,Anormerr]=myPrecCG(b,usm,params,x0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preconditioned conjugate gradient algorithm to solve the
% linear system Ap=f with initial guess p0 for p.
% The preconditioning operator B is defined in the
% file Precond_Op_B.m
%
% f : Nx1 vector
% usm: unstructured 2D mesh
% params: string, name of the m-file containing
%              smoothing scheme parameters
% x0 : Nx1 vector -initial guess for the solution (optional)
%
% x : Nx1 vector -solution
% flag,relres,iter,resvec: see pcg.m
% Anormerr: square of the error in the energy norm
%
%  adapted version of matlab-routine pcg.m
%  Gisela Widmer 19/2/2004
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% parameter setup
  preccg_param;
  if nargin<4
    x=zeros(size(usm.A,2),1);
  else
    x=x0;
  end

% Set up for the method
  flag = 0;        % a valid solution has been obtained
  relres = 0;      % the relative residual is actually 0/0
  iter = 0;        % no iterations need be performed
  resvec = 0;

  tol=1e-6;        % stop criteria
  maxit=50;
```

```
  A=usm.A;
  n2b=norm(b);

  flag = 1;
  xmin = x;         % Iterate which has minimal residual so far
  imin = 0;         % Iteration at which xmin was computed
  tolb = tol * n2b; % Relative tolerance

  ex_sol =  A\b;    % Exact solution
  r = b - A * x;    % Zero-th residual

  normr = norm(r);  % Norm of residual

  if (normr <= tolb) % Initial guess is a good enough solution
    flag = 0;
    relres = normr / n2b;
    iter = 0;
    resvec = normr;
    if (nargout < 2)
      disp('pcg nargout <2');
    end
    return
  end

  resvec = zeros(maxit+1,1); % Preallocate vector for norm of residuals
  Anormerr=zeros(maxit+1,1);
  resvec(1) = normr;         % resvec(1) = norm(b-A*x0)
  Anormerr(1)=(x-ex_sol)'*A*(x-ex_sol);
  normrmin = normr;          % Norm of minimum residual
  rho = 1;
  stag = 0;                  % stagnation of the method

% loop over maxit iterations (unless convergence or failure)

  for i = 1 : maxit
   disp('opB')
    y = Precond_Op_B(r,usm,params);

    if isinf(norm(y,inf))
      flag = 2;
      break
```

```
    end
    z=y;
    rho1 = rho;
    rho = r' * z;
    if ((rho == 0) | isinf(rho))
      flag = 4;
      break
    end
    if (i == 1)
      p = z;
    else
      beta = rho / rho1;
      if ((beta == 0) | isinf(beta))
         flag = 4;
         break
      end
      p = z + beta * p;
    end
    q = A * p;
    pq = p' * q;
    if ((pq <= 0) | isinf(pq))
      flag = 4;
      break
    else
      alpha = rho / pq;
    end
    if isinf(alpha)
      flag = 4;
      break
    end
    if (alpha == 0)             % stagnation of the method
      stag = 1;
    end

% Check for stagnation of the method
    if (stag == 0)
      stagtest = zeros(size(A,1),1);
      ind = (x ~= 0);
      stagtest(ind) = p(ind) ./ x(ind);
      stagtest(~ind & p ~= 0) = Inf;
      if (abs(alpha)*norm(stagtest,inf) < eps)
```

```
      stag = 1;
    end
  end

  x = x + alpha * p;               % form new iterate
  normr = norm(b - A * x);
  Anormerr(i+1)=(x-ex_sol)'*A*(x-ex_sol);
  resvec(i+1) = normr;

  if (normr <= tolb)               % check for convergence
    flag = 0;
    iter = i;
    break
  end

  if (stag == 1)
    flag = 3;
    break
  end

  if (normr < normrmin)   % update minimal norm quantities
    normrmin = normr;
    xmin = x;
    imin = i;
  end

  r = r - alpha * q;
end

% returned solution is first with minimal residual
if (flag == 0)
  relres = normr / n2b;
else
  x = xmin;
  iter = imin;
  relres = normrmin / n2b;
end

% truncate the zeros from resvec
if ((flag <= 1) | (flag == 3))
  resvec = resvec(1:i+1);
```

```
    Anormerr=Anormerr(1:i+1);
  else
    resvec = resvec(1:i);
    Anormerr=Anormerr(1:i+1);
  end

% only display a message if the output flag is not used
  if (nargout < 2)
    disp('pcg nargout < 2');
  end
return
```

A.5 Condition Number

A.5.1 Cond_AB.m

```
function [l_max,l_min,cond_nr] =Cond_AB(usm,tit,param_file)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% maximal and minimal eigenvalue and cond_2 of the edgeelement
% stiffness matrix A = C + tau*M on the unstructured mesh usm
% evaluated with vector iteration
%
% usm: unstructured 2D mesh
% tit: String that contains the title for the plot
% param_file: string, file with general parameters
%
% l_max: max eigenvalue of ABBA
% l_min: min eigenvalue of ABBA
% cond_nr: cond_2(AB)
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  global tau;

% normalized initial vectior with random entries
  x0=rand(size(usm.A,1),1);
  x0=x0/norm(x0);

% forward vector iteration
```

```
  [lambda1,n1]=Vec_it_eig(x0,usm,200,1e-5,tit,param_file);

% inverse vector iteration
  [lambda2,n2]=Inv_Vec_it_eig(x0,usm,200,50,1e-5,tit,param_file);

% maximal and minimal eigenvalue, condition number
  l_max=lambda1;
  l_min=1./lambda2;
  cond_nr=sqrt(l_max(length(l_max))/l_min(length(l_min)));
 return
```

A.5.2 Vec_it_eig.m

```
function [lambda,n]=Vec_it_eig(x0,usm,kmax,tol,tit,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% maximal eigenvalue of ABBA by vector interation
%
% A: matrix
% x0: Nx1 vector initial guess for the eigenvector
% usm: unstructured 2D mesh
% kmax: maximal CG steps to solve A x_n+1 = x_n
% tol : error tolerance (norm(x_n+1-x_n))
% tit: string that contains the title of the plot
% params: string of file with general parameters
%
% lambda: 1xM vector, approximation of lambda_max(A-1) after
%         each step
% n: 1xM vector, (norm(x_n+1-x_n)) after each step
%
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% parameter setup
  k = 0;
  x = x0;
  lambda=[];
  n=[];

%first iteration
```

```
  v = Precond_Op_B(usm.A*x,usm,params);
  v = Precond_Op_B(v,usm,params);
  v = usm.A*v;
  n=[n,norm(v-x)];
  nv = norm(v);
  lambda=[lambda,nv/norm(x)];

% vector iteration
  while(norm(v-x)>tol & k<kmax)

% normalization
    x= v./nv;
    v = Precond_Op_B(usm.A*x,usm,params);

    n=[n,norm(v-x)];
    nv = norm(v);
    lambda=[lambda,nv/norm(x)];
    k=k+1;
  end
return
```

A.5.3 Inv_Vec_it_eig.m

```
function [lambda,no]=Inv_Vec_it_eig(x0,usm,jmax,CGmax,tol,tit,params)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% maximal eigenvalue of (ABBA)^(-1) by inverse vector
% interation
%
% x0: Nx1 vector initial guess for the eigenvector
% usm: unstructured 2D mesh
% jmax: maximal vector iteration steps
% CGmax: maximal CG steps to solve A x_n+1 = x_n
% tol : error tolerance (norm(x_n+1-x_n))
% tot: string that contains the title of the plot
% params: string, file with general parameters
%
% lambda: 1xM vector, approximation of lambda_max(A-1) after
%                       each step
% no: 1xM vector, (norm(x_n+1-x_n)) after each step
%
```

```
% Gisela Widmer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% parameter setup
  j = 0;
  v = x0;
  x=zeros(size(v,1),1);
  no=[];
  lambda=[];


% vector iteration loop
  er=norm(v-x)
  while(j<jmax & er>tol)
    j=j+1

%   BA x_n+1 = x_n by BICGSTAB

    [x,flag,relres,iter,resvec]=pcg(@ABBA_Op,v,tol,CGmax,[],[],...
                                 zeros(size(v,1),1),usm,params);
    flag

    nx = norm(x);
    lambda=[lambda,nx/norm(v)];
    x= x/nx;
    er=norm(v-x);
    no=[no,norm(v-x)/length(v)];
    v=x;
    if(length(no)==0)
      no=0;
      lambda=1;
    end
  end
return
```

A.6 Multigrid Convergence Rate

A.6.1 MG_conv_rate.m

```
function [q,rs]= MG_conv_rate(MESH,mesh_nr,mesh_size,param_nr,tau_nr)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Computation of numerical convergence rate of the multigrid
% algorithm on the unstructure mesh to solve the problem
% A_0 x_0 = usm.sm.I'*F with right hand side function specified
% in read_EBP_param.m and random initial guess
%
% MESH: string of name of the 2D unstructured mesh
% mesh_nr: refinement level of the unstructured mesh
% mesh_size: relative element size of the unstructured and the
% auxiliary mesh
% param_nr : number of the parameter file to use
% tau_nr: weighting of the absolute term in the differential
%         equation
%
% q: vector of numerical convergence rates during iteration
% rs: vector of residuals during iteration
%
% Gisela Widmer 19/2/2004
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Iteration parameters

  max_it = 100; % max number of iterations
  tol = 1e-5;  % convergence tolerance

  global tau;

% mesh setup
  meshfile=[MESH,'meshes',num2str(mesh_nr),'_size_',num2str(mesh_size)];
  tau = tau_nr;
  param_file=['params_',num2str(param_nr)];
  eval(param_file);
  usm=mesh_setup(meshfile,stiffnessflag,param_file);


% read the parameters for quadrature
  [rhsf,quadraturefile]=read_EBP_param;
  Q = dlmread(quadraturefile);
```

```
% computation of rhs- vector for the edge elements
  F = Erhs(usm,rhsf,Q,1);

% random initial guess
  xs_old = 1e40.*rand(size(usm.sm.A,2),1);

% Right hand side on the auxiliary mesh
  Fs = usm.sm.I'*F;

% Initialization for interation
  dxs_old=0;
  rs=[];
  q=[];
  rs(1) = norm(Fs-usm.sm.A*xs_old);

  for i=0:max_it
    i
    xs_new= MultiGrid_solve(Fs,xs_old,param_file,usm.sm);
    dxs(i+1)=norm(xs_new-xs_old);
    if i>0
      if(dxs_old<eps)
             disp ('dxs_old<eps')
             break;
      else
             q(i)=dxs(i+1)/dxs_old;
             rs(i+1)=norm(Fs-usm.sm.A*xs_new);
      end
    end
    dxs_old = dxs(i+1);
    xs_old  = xs_new;

% check convergence criterion
    if (i>1 & norm(rs(i+1)-rs(i)) <tol)| norm(rs(i+1))<tol
        disp norm(q(i)-q(i-1));
        break;
    end
  end
  disp('convrate done');
  q

%store rs and q
```

```
  eval(['save ',MESH,'_conv_rate_OV_',num2str(mesh_nr),'_size_',...
        num2str(mesh_size),'_params_',num2str(param_nr),'tau_',...
        num2str(tau_nr),' rs q']);
return
```

A.7 Example of a Parameterfile

```
MGcycle=1; % number of multigrid cycles
stiffnessflag=1; % choice of stiffness matrix 0: Galerkin
                 %                            1: direct
degree=1;        % width of boundary layer for smoothing
PrecondSolver = 2; % choice of solver in the aux space
                   % 1: direct 2: Multigrid
smooth_it1=1;      % number of domain smoothing iterations
smooth_it2=1;
smooth_it3=1;
bound_smooth_it1=1;% number of boundary smoothing interatins
bound_smooth_it2=1;
bound_smooth_it3=1;
```

References

[1] R. Beck, P. Deuflhard, R. Hiptmair, R. Hoppe, and B. Wohlmuth. Adaptive multilevel methods for edge element discretizations of maxwell's equations. *Surv. Math. Ind.*, 8(3–4):271–312, 1999. Springer-Verlag.

[2] J. Xu. The auxiliary space method and optimal multigrid preconditioning techniques for unstructured grids. *Computing*, 56(3):215–235, 1996.

[3] Xu Jincho. An introduction to multilevel methods. *Math. Sci. Comput.*, 1997.

[4] J. Nédélec. A new family of mixed finite elements in $\mathbb{R}^3$. *Numerische Mathematik*, 50:57–81, 1986.

[5] R. Hiptmair. Multigrid methods for maxwell's equations. *SIAM Journal on Numerical Analysis*, 36(1):204–225, 1998.